Optimistic Semaphores with Non-deterministic Choice Operation for Heterogeneous Manycore Systems *

Enrique Soriano-Salvador
Gorka Guardiola Muzquiz
Francisco J. Ballesteros

LSUB, GSyC, ETSIT
Universidad Rey Juan Carlos
{esoriano,paurea,nemo}@lsub.org

Abstract

The Nix operating system permit different roles to be assigned to the cores. One of the roles is able to run userspace code with no interrupts from the operating system, which is particularly useful for HPC. System calls are especially expensive to a core playing this role. This paper presents a new implementation of optimistic semaphores that avoids performing expensive system calls in an uncontended scenario. The implementation is straightforward and somewhat unorthodox: a semaphore is based on a data structure and a lock that are shared between userspace and kernelspace. This study aims at evaluating if such an approach is viable. In addition, the implementation includes a non-deterministic choice operation over a collection of semaphores, *altsems*. This novel operation facilitates the creation of higher level communication mechanisms, such as sockets and channels. To support this claim, we implemented a new kind of buffered communication channels named *tubes*, tailored for communicating processes running on different (heterogeneous) cores. The paper describes the implementation of the semaphores and the tubes, a comparative analysis of optimistic and non-optimistic semaphores on Nix, and a comparative analysis of tubes and other kinds of communication channels that are available on the Nix operating system.

1 Introduction

This paper presents an implementation of optimistic semaphores for user and kernel processes suited for the Nix [2] operating system, but which are themselves of intrinsic interest. Nix is a new operating system that follows a heterogeneous multi-core model,

*This work has been funded by Comunidad de Madrid projects CLOUDS S2009/TIC-1692 and Cloud4BigData (CAM) S2013/ICE-2894 (cofunded by FSE & FEDER) and Spanish Ministry of Education projects TIN2010-17344 and TIN2013-47030-P.

which started as a fork of Plan 9 [18] (the Unix successor implemented at Bell Labs and released in the early nineties). Although semaphores have been around for a long time, in some cases, such as the implementation of optimistic semaphores, it is not trivial to implement them correctly and efficiently. Moreover, the performance of semaphores heavily depends on the concrete implementation and the system of interest (operating system and hardware). The purpose of this paper is to present an implementation tailored for Nix and analyze its performance. This implementation does not depend on specific capabilities of the Nix operating system. Thus, it can be ported to other operating systems with little effort. In addition, the paper presents a new operation for semaphores called `altsems` and the implementation of communication channels for IPC (Inter Process Communication) based on it.

Nix is an operating system designed for current manycore shared-memory (ccNUMA) machines. While Nix is a general purpose operating system, it includes special capabilities for HPC and cloud computing [2]. One of these mechanisms enables the user to assign different roles to heterogeneous cores. By *heterogeneous cores* we mean cores with different capabilities. For example, some of the cores may not be able to run in Ring 0 or may have limited interrupt facilities. Other operating systems, such as Linux, can benefit from this approach. For example, FusedOS [17] added core roles to a Linux kernel, inspired by Nix. Some of the core roles available in Nix are: (i) Time- sharing Core (TC), which is a common core running kernel and user code in a time sharing fashion; (ii) Application Core (AC), which is a core running user code without any interrupt (even without clock interrupts); and (iii) Kernel Core (KC), which is a core that only runs kernel code on demand. The cores communicate by sending active messages that include a function to be executed, together with its arguments (e.g. by using shared memory).

For applications running on ACs, the cost of a system call is very high, because the application has to send an active message to a KC to perform the system call. In other words, any operation requiring a system call uses the KC as a proxy (the AC cannot execute kernel code). Therefore, it is of paramount importance to be able to perform IPC without entering the kernel if that is feasible.

Note that performing IPC without entering the kernel is profitable in general, no matter what operating system or application is executing. Thus, the implementation described in this paper is not only of interest for the Nix operating system. The AC scenario described above is just an extreme example of the general problem.

A semaphore is a well known synchronizing primitive, invented by Dijkstra in 1965. A general semaphore is basically a counter ($\mathbb{N}$) for saved *wakeups* [29]. A *wakeup* represents the permission to unblock (i.e. *wake up*) a process. A semaphore makes this permission persistent. If there are no blocked processes to be awakened, the permission is saved for future use (i.e. a counter is incremented). Although there are several implementations of semaphores providing different operations and semantics [21], Dijkstra defined two basic atomic operations [4]:

- *Down* (also known as *P* or *wait*) checks if the counter is greater than zero. In this case, the caller consumes one stored wakeup. If there are no saved wakeups, the process will block until there is an available wakeup.

- *Up* (also known as *V* or *signal*) stores a wakeup in the semaphore. If there are blocked processes waiting for a wakeup, only one of them consumes the new wakeup

generated by this operation. If there are no processes blocked, the wakeup is saved (i.e. the counter is incremented).

Semaphores are traditionally implemented within the operating system kernel, that is, user level processes must use a set of system calls to access them. As stated before, the cost of a system call in our manycore system is not negligible. In particular, for some core roles and heterogeneous manycore machines it can be extremely expensive (e.g. the AC role of Nix). On the other hand, semaphores can be implemented by a userspace library. Even in this case, in order to avoid busy waiting, processes must enter the kernel to be blocked (using any synchronizing mechanism, for example *rendezvous*). A problem with this approach is that the kernel does not know about semaphores, so it is not able to use them to synchronize a kernel task with a userspace process (e.g. synchronizing a driver and a user process using a semaphore). We follow a mixture of the two approaches. Our semaphores are defined by two data structures, one for userspace and another for kernelspace, and can be used from both modes.

The semaphores described here are *unnamed*, they depend on shared memory. In Nix, like in Plan 9, processes are created by the `rfork` system call and they can share memory (data and bss/heap segments). FreeBSD's `rfork` and Linux's `clone` system calls are modeled on Plan 9's `rfork` [20].

Our new semaphores provide strong semantics: they are *blocked-queue general semaphores* [26]. They are also *optimistic* in the sense they are optimized for the uncontended case. In the contended case, they do not perform as well as standard kernel semaphores. In other words, we rely on the observation that the common scenario is uncontended and processes can perform operations upon semaphores without entering the operating system kernel. The kernel only mediates when the contention is high (or the semaphore is used to block a process).

Other works also follow an optimistic approach [8, 24, 9, 15]. They usually rely on atomic operations to increment and decrement the integer value that represents the state of the primitive. Nevertheless, we provide a straightforward and unorthodox approach to implement optimistic semaphores. Kernelspace and userspace share a data structure and a spin-lock to protect it. Commonly, sharing data structures and locks between userspace and kernelspace is considered harmful. One of the aims of this work is to determine if the approach is viable for a system like Nix. Section 2.4 discusses the trade-offs and safety issues. In this paper we also present a comparative analysis of the performance of our optimistic semaphores and the standard non-optimistic semaphores of Nix. The analysis was made for scenarios with two core roles (Application Cores and Time-sharing Cores) and different levels of contention.

The semaphores described in this paper include a new operation, `altsems`. This operation is a non-deterministic *down* over a collection of semaphores. It is based on Hoare's non-deterministic choice operator (also known as *alternative command*) of Communicating Sequential Processes (CSP) [11]. To the best of our knowledge, adding this operation to the interface for semaphores is novel. A process calling `altsems` tries to perform a `down` operation on a list of semaphores. The operation acquires exactly one saved wakeup from any of the semaphores in the list. This new operation over semaphores provides a generic mechanism to implement non-deterministic choice operations for higher level abstractions. It is highly useful to implement, for example, communication channels

(e.g. Google Go channels [12]) or the `select` operation that permits I/O multiplexing on POSIX-compatible systems. To illustrate the potential of the operation, we present a new implementation of channels for IPC, named `tubes`. A tube is an optimistic buffered channel tailored to communicate userspace processes running on different Nix Application Cores avoiding expensive system calls. The operations available for tubes are similar to the ones of Go and Plan 9 channels, and can be used to implement CSP-like programs. This paper also presents a comparative analysis of tubes and other kinds of channels available in Nix.

A preliminary version of the Nix semaphores was briefly described in [2]. Since then, we have implemented and tested different versions of the semaphores that resulted in incorrect behaviour. After trying different algorithms and data structures, we found the design and implementation described here. We believe the algorithm and the implementation to be correct after thoroughly testing them. Note that we have tried different approaches, including formal verification, to try to guarantee the correctness of the algorithms. The authors tried to verify one of the versions of the algorithm formally using Spin. While the model guaranteed the algorithm to be correct, it turned out not to be, because of inconsistencies between the model and the implementation. We detected the problem while performing stress tests. This is not to say that Spin models are useless. The point is that writing the model can be as complex as writing the code itself, and the specific model may have its own errors. New verification techniques are appearing, which may make this problems easier to solve.

2 Optimistic Semaphores

Our approach to implement optimistic semaphores is different than others found in the literature. As stated before, the semaphores are optimistic in the sense that they rely on the hope that the common scenario is uncontended and there will be saved wakeups in the semaphore. The point is to improve the performance in an optimistic scenario, that is, when it is likely to obtain a saved wakeup when calling `down` without blocking (and it is unlikely to find blocked processes to be waken up when calling `up`). The gain of avoiding a system call is substantial, specially for applications running on Nix Application Cores (AC).

On the other hand, if contention is high, the semaphores will perform worse that standard kernel semaphores because they need extra time to check the state of the semaphore from userspace.

The common approach to implement optimistic semaphores is performing atomic increment/decrement operations on a shared integer variable. We use a different approach: userspace and kernelspace share a per semaphore structure and a lock to protect it. Section 2.4 discusses the trade-offs and the possible drawbacks of this approach regarding safety and performance.

2.1 Interface

The interface (except the new operation, `altsems`) is quite similar to the interface of the standard non-optimistic semaphores [32] [1]. The C interface of the library is:

```
void  initsem(Sem *s, int val);
int   downsem(Sem *s, int block);
void  upsem(Sem *s);
int   altsems(Sem *ss[], int nsems);
```

`Initsem` initializes a semaphore with `val` saved wakeups. Once a semaphore is initialized, the user must use only `downsem`, `upsem`, or `altsems`. If the second parameter of `downsem` is zero, it is a non-blocking down. In this case, if there are no available wakeups in the semaphore, the process is not blocked (it just fails). A non-blocking down never enters the kernel. `Altsems` requires two parameters: an array of semaphores and its length.

The data structure that defines the semaphore is split into two different data structures: `Sem` and `Ksem`. The former is used by applications (i.e. userspace). The later is used by the OS kernel.

2.2 Userspace

The userspace part of the semaphore is the `Sem` data structure[2], which is composed of the following fields:

- `wakeups`: ($\mathbb{N}$) counter of wakeups stored in the semaphore.
- `waiting`: ($\mathbb{N}$) estimated number of processes that *may* be blocked in the semaphore.
- `going`: ($\mathbb{N}$) number of processes that are in transit from userspace to the kernel part of the semaphore (to block).
- `lock`: lock to protect the data structure.

If N is the actual number of blocked processes in the semaphore, the main invariant is:

$$waiting \geq N$$

The Sem data structure is allocated in a shared memory segment. All the processes that need to be synchronized must share the segment. The structure pursues three objectives: (i) permit the user processes to share the state of the semaphore; (ii) let the user processes know when they must enter the kernel; and (iii) be a kernel handle for the semaphore.

```
downsem(s: Sem, block: Boolean): Boolean
       begin
                busywait(s, NAttempts)
                lock(s)
                if not block and s.wakeups = 0 then
                        unlock(s)
                        return False    //fail
                elsif s.wakeups > 0 then
                        decrement s.wakeups
                        unlock(s)
                        return True
                else
                        increment s.going
                        unlock(s)
                        semsleep-syscall(s)
                        return True
                end if
       end
```

Figure 1: Pseudo-code for userspace function `downsem`.

2.2.1 Downsem

The userspace part of the semaphores is quite simple. Figures 1 and 3 show the pseudo-code. The key point of the userspace part of the semaphore is to increment or decrement the counter of saved wakeups when the kernel does not have to mediate (that is, in the optimistic scenario).

The procedure `busywait`, which is called in the beginning of `downsem`, performs a limited busy wait until `wakeups` is greater than zero (without acquiring the lock). The number of iterations for the busy wait is configurable (`NAttempts`), so it can be fixed after a careful consideration of the trade-off between the cost of a system call and the cost of the busy waiting. Note that this depends on the machine architecture and the role of the core. The aim of this busy wait is to maximize the probability that there are available wakeups and minimize system calls. Note that the presence of an available wakeup is checked always while holding the lock.

The field `lock` guarantees mutual exclusion while dealing with the structure. Note that this lock is only used to protect a tiny critical region required to check the value of the field `wakeups` and select the path to go (enter the kernel by performing a system call or continue in userspace). This field is a spin-lock implemented with the test-and-set primitive. The implementation of the `lock` procedure is tailored for the role of the current core. For example, for a Time-sharing Core it is not a pure spin-lock [14] . In this case, it applies an exponential back-off after a number of unsuccessful attempts to acquire the lock. The back-off is composed of several steps (yielding the processor, sleeping 100 ms, sleeping 1 s, etc.). This back-off is not suitable for other types of role, such as Application Cores. For Application Cores, it is strictly a spin-lock.

In a contended scenario (that is, there are no saved wakeups in the semaphore) the

[1]Note that the interface described in [2] is deprecated. The interface of the optimistic semaphores has been unified with the standard Plan 9 semaphores.

[2]Defined in the Nix C standard library.

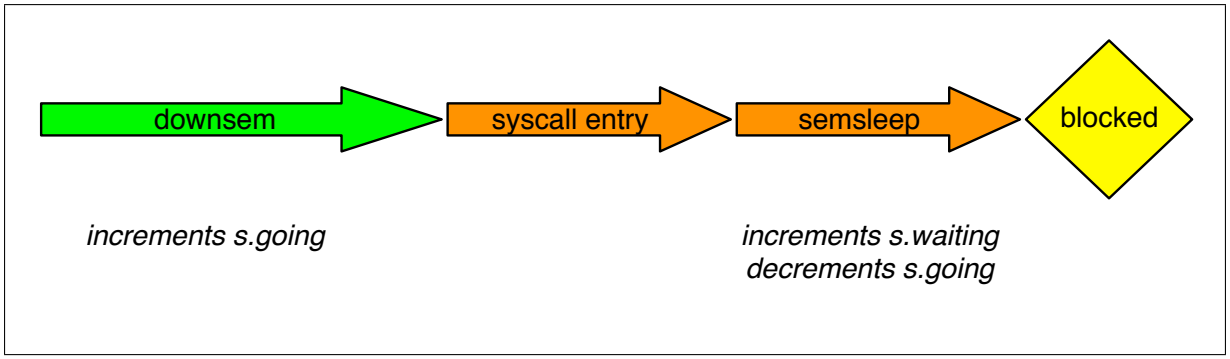


Figure 2: This process is blocked in the semaphore. Green arrows represent userspace execution. Orange arrows represent kernelspace execution.

```
upsem(s: Sem)
       begin
                lock(s)
                if s.wakeups = 0 and (s.waiting > 0 or s.going > 0) then
                        unlock(s)
                        semwakeup-syscall(s)
                else
                        increment s.wakeups
                        unlock(s)
                end if
       end
```

Figure 3: Pseudo-code for userspace function `upsem`.

process calling `downsem` will enter the kernel to be blocked. In this case, the process performs the `semsleep` system call. Before entering the kernel, it increments the `going` field. As stated before, this field counts the number of processes that are in transit from `downsem` to the `semsleep` system call. Figure 2 shows the progression to a process that blocks in the semaphore.

2.2.2 Upsem

A process calling `upsem` may have to enter the kernel to wake up an already blocked process. This happens when the counter `waiting` is positive. The field `waiting` is updated by the kernel (in the `semsleep` syscall) when a process is queued and blocked.

The counter `going` is also used to decide if the process calling `upsem` has to enter the kernel: another process calling `downsem` may be in transit from userspace to kernelspace to be blocked. Note that `going` is incremented when a process is going to enter the kernel to block, as shown in Figure 2. Later, the kernel part of *down* (the `semsleep` system call) will convert the `going` unit to a `waiting` unit, before the process gets blocked.

2.2.3 Altsems

Our implementation of semaphores includes a new operation called `altsems`. This operation requires a list of semaphores. It performs a blocking `downsem` on strictly one semaphore of the list. It selects the first available semaphore of the list in a fair manner.

```
altsems(ss: Array of Sem, nsems: Integer): Integer
       begin
                set start to random
                busysearch(ss, start, NAttempts)
                for s in ss from start loop
                        if downsem(s, False) then
                                return index of s
                        end if
                end loop
                return altsem-syscall(ss, nsems)
       end
```

Figure 4: Pseudo-code for userspace function `altsems`.

Note that the semaphores aggregated in an `altsems` operation can be used as regular semaphores simultaneously by another processes (e.g. a process can perform a down or an up in one semaphore that is being used within an `altsems` by other process).

The userspace part of this function is straightforward, as shown in Figure 4. The process performs a bounded busy wait (`busysearch`) looking for an available wakeup in any semaphore of the list (`ss`). The role of this procedure is analogous to the busy wait in `downsem`. This procedure receives the variable `start` by reference and updates it while searching. The search is started at a random point to guarantee fairness for all the semaphores in the list. The number of iterations for the busy wait is configurable (`NAttempts`). When it ends (it finds an available semaphore or reaches the maximum number of iterations), it tries to acquire the wakeup from the semaphores in the list by performing a sequence of non-blocking `downsem` operations. It starts with the semaphore in which the busy waiting stopped searching. Note that none of these non-blocking downs enter the kernel. If this attempt is unsuccessful, then it performs a system call, `altsem`. Note that `altsem` is a system call used by the `altsems` userspace function. In the optimistic scenario, the process will find an available semaphore in the first iteration of the busy waiting, and the non-blocking `downsem` will acquire it.

2.3 Kernel

The system calls `semsleep`, `semwakeup` and `altsem` use the `Ksem` data structure. The synchronized processes share a memory segment, where the `Sem` structure (the userspace part of the data structure of the semaphore) is allocated. In addition, a hash table of `Ksem` structures is attached to the kernel data structure that describes the shared segment. The `Ksem` structure has the following fields:

- `sem`: a pointer to the corresponding `Sem` structure. This field is used to store the semaphore from which the process has obtained the saved wakeup in an `altsem` system call.
- `q`: a queue to store the processes blocked in the semaphore.
- `nowait`: ($\mathbb{N}$) a counter used to avoid processes to block when calling `downsem`. It is an alternative counter storing wakeups which are already spoken for.

- `lock`: lock used to protect the data structure.
- `rwlock`: a reference to a per-segment readers-writer lock, shared by all the semaphores that could be combined in an *altsem* operation. Note that this lock is shared between all semaphores belonging to the same segment.
- `state`: a value describing the current state of the semaphore (`SEMDEAD` or `SEMALIVE`). It is used to protect the system against deadlocks caused by errors in the programs executed by user processes.

Therefore, the semaphore system calls deal with three different locks: (i) the lock of the `Sem`, the userspace part of the semaphore, used to protect the data shared by the user processes and the kernel; (ii) the lock of the `Ksem` used to protect the kernel data of the semaphore; and (iii) the readers-writer lock, needed to implement `altsem`. From the point of view of the readers-writer lock, the `semsleep` and `semwakeup` system calls are the *readers* and the `altsem` system call is the *writer*.

The data structure that defines a process in the kernel includes a boolean field named `semawaken` to mark the process as not blocked in a semaphore. It also includes a pointer named `waitsem` that points to the semaphore in which the process has been awoken. The field `waitsem` is null if `semawaken` is false.

2.3.1 Semsleep

The `semsleep` system call, shown in Figure 5, implements the kernel part of `downsem`. A process calls it if there are no available wakeups when it executes the userspace part. In this case, the process increments `s.going` and enters the kernel.

To acquire the userspace lock of the corresponding `Sem` structure, the kernel code uses the procedure `userlock`. This procedure is described later, in section 2.5.

The counter `ks.nowait` determines if the process has to block (if greater than zero, the process does not have to wait). It represents the number of wakeups that are already spoken for by processes in transit. It is incremented by `semwakeup` (as explained next) and it is used to handle *overtakings*. Figure 7 shows an example. If `ks.nowait` is zero, the process is queued and blocked in the semaphore. In this case, *its reservation is made*.

2.3.2 Semwakeup

The `semwakeup` syscall is described in Figure 6. A process calls it when there are no stored wakeups and (i) there are blocked processes in the queue (i.e. `s.waiting` is positive), or (ii) there are processes in transit from userspace to the kernel (i.e. `s.going` is positive).

If there is a blocked process queued in the semaphore, it is awoken and the system call finishes. Note that, due to the *altsems* operation, the same process can be sleeping in different semaphores. As a consequence, there may be processes in the queue that have been already awoken (by another semaphore). In addition, two concurrent processes could try to wake up the same process in two different semaphores. For this reason, the process is marked as awoken by using a *test-and-set* instruction to ensure that the process is awoken only once and it acquires exactly one wakeup.

If it does not find a blocked process to wake up, it continues. If there is any process in transit from userspace to the kernel (i.e. `s.going` is positive), `ks.nowait` is incremented.

```
semsleep(s: Sem)
       begin
                set ks to the Ksem of s
                rlock(ks.rwlock)
                lock(ks.lock)
                if ks.nowait > 0 then
                        userlock(ks)
                        decrement s.going
                        userunlock(ks)
                        decrement ks.nowait
                        unlock(ks.lock)
                        runlock(ks.rwlock)
                else
                        userlock(ks)
                        increment s.waiting
                        decrement s.going
                        userunlock(ks)
                        set process.waitsem to null
                        set process.semawaken to False
                        queue current_process in ks.q
                        unlock(ks.lock)
                        runlock(ks.rwlock)
                        block current_process
                enf if
       end
```

Figure 5: Pseudo-code for the system call `semsleep`.

This value is used to pass the wakeup directly to the process in transit, without saving it in the semaphore (`s.wakeups`).

If there are no processes blocked in the semaphore and there are no processes in transit from userspace, the process just increments the number of saved wakeups of the semaphore (`s.wakeups`) and returns. In this case, the value seen for `s.waiting` in userspace was greater than the actual number of processes blocked in the semaphore. This could happen if a process blocked in the semaphore *cancels its reservation*. In other words, a process that was blocked in this semaphore does not want the wakeup anymore (e.g. a process that was blocked in several semaphores due to an `altsem` acquired a wakeup from another semaphore and canceled the reservation for the rest).

2.3.3 Altsem

The `altsem` system call is described in Figures 8 and 9. `Altsem` acquires the readers-writer lock (rwlock) as a *writer*. That means that while a process is looking for an available wakeup in `altsem`, no other process can acquire the rwlock (as a reader or writer). Thus, no other process (sharing this segment) is able to execute the body of `semsleep`, `semwakeup` or `altsem` while this process is searching. The process iterates through the list of `Ksem` structures, looking for a saved wakeup. While inspecting a semaphore, it also holds the lock that protects the corresponding userspace structure (`Sem`).

If there are no available wakeups in a semaphore, the process is queued and the counter

```
semwakeup(s: Sem)
       begin
                set ks to the Ksem of s
                rlock(ks.rwlock)
                lock(ks.lock)
                while ks.q not empty loop
                        dequeue process from ks.q
                        userlock(ks)
                        decrement s.waiting
                        userunlock(ks)
                        if test_and_set process.semawaken to True then
                                set process.waitsem to ks
                                unlock(ks.lock)
                                unlock(ks.rwlock)
                                wake up process
                                return
                        end if
                end loop
                userlock(ks)
                if s.going > 0 then
                        increment ks.nowait
                else
                        increment s.wakeups
                end if
                userunlock(ks)
                unlock(ks.lock)
                runlock(ks.rwlock)
       end
```

Figure 6: Pseudo-code for the system call `semwakeup`.

of waiting processes for this semaphore is incremented, but it does not block yet: it has to keep searching in the rest of the semaphores. Note that `s.waiting` is an estimation of the actual number of blocked processes already queued in the semaphore, and is always greater than or equal to the actual value, as stated in Section 2.2. Note also that the process could not be awoken in a preceding semaphore while searching, because it holds the readers-writer lock as a writer.

If the process finds a saved wakeup, it does not have to block. In this case, the process is marked as awoken by setting `process.waitsem`. Then, it must *cancel the reservations* made for previous semaphores and return. The cancellation is made by the procedure `cancel_reservations`.

While iterating to find the wakeup, another process could call `upsem` (userspace) to generate a wakeup for a semaphore with the reservation already made (a preceding semaphore in the array). In this case, `upsem` finds that `s.waiting` is greater than zero and enters the kernel (`semwakeup`). Once in `semwakeup`, it cannot proceed until the readers-writer lock is released.

If the process does not find one saved wakeup in any of the semaphores, it releases the readers-writer lock and blocks. When the process is awoken (by another process that executes `semwakeup`) it *cancels the rest of reservations* and returns.

An important detail in `altsem` is that, for each iteration, the process must hold the

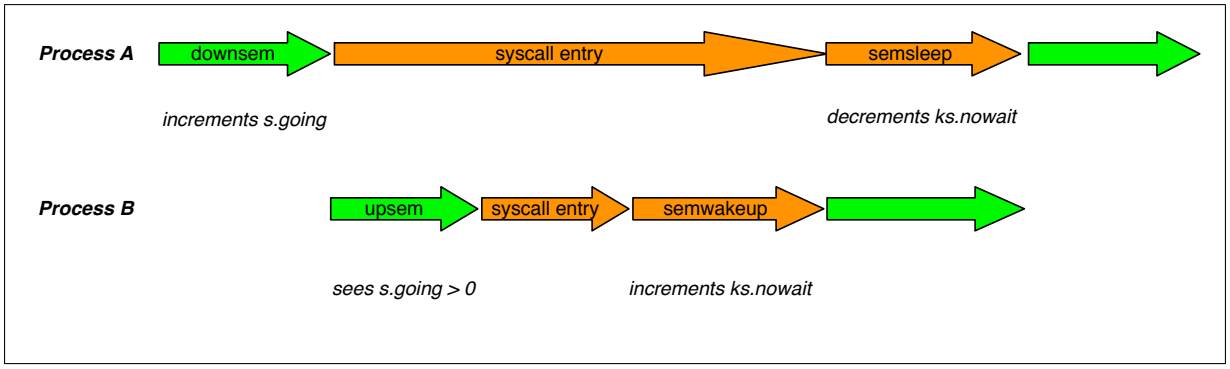


Figure 7: Process A does `down`, enters the kernel and finds that it does not have to block because process B generated a wakeup in the interim (there is an *overtaking*). Green arrows represent userspace execution. Orange arrows represent kernelspace execution.

lock of the current `Ksem` structure when inserting the process in the queue. There may be concurrent processes calling `cancel_reservations` (due to the completion of different `altsems` calls) while the process is being queued, and `cancel_reservations` does not acquire the global rwlock. If the lock of `Ksem` is not held, there is a race condition while dealing with the queue. This was actually a real bug in previous versions of the implementation, detected by the stress tests.

The procedure `cancel_reservations`, shown in Figure 9, extracts the process from the queue of all the semaphores. That is, the process dequeues itself from all the semaphores. This procedure is called from `semwakeup` in two cases: (i) the process finds an available semaphore in the array and it does not block; or (ii) the blocked process is awoken.

`Cancel_reservations` iterates through the array, dequeuing the process and decrementing `s.waiting` (in this order, because `s.waiting` must never be less than the actual number of processes blocked). Note that the process could be already dequeued from any of the queues by another process executing `semwakeup` in the interim. In this case, it does not decrement `s.waiting`.

It is important to notice that the process cannot be awoken in several semaphores; a process is awoken in strictly one semaphore. This is guaranteed by the field of the process data structure `process.semawaken`, atomically tested and set in `semwakeup`, and set in the critical section of `altsem` (protected with the rwlock held in *writer* mode).

2.4 Trade-offs and Safety

In Nix, the virtual memory space of a process is split in two: userspace addresses (from 0 to the middle of the virtual address space[3]) and kernel addresses (from the middle to the end of the virtual space). The kernel address space is shared for all processes. Of course, each process has its own mapping for userspace addresses. The memory of a process is structured into *segments*[4]. A process knows it has a set of segments attached at concrete virtual addresses with concrete sizes. All processes have at least four segments:

[3]Note that the first virtual memory page is invalid

[4]This segment is not a hardware segment, it is an abstraction used by the operating system. Hardware segments are only used for protection.

```
altsem(ss: Array of Sem, n: Integer)
       begin
                allocate kss as Array of n Ksems
                for s in ss loop
                        add the Ksem of s to kss
                end loop
                wlock(kss[0].rwlock)
                set process.waitsem to null
                set process.semawaken to False
                for ks in kss from random index loop
                        set s to ks.sem
                        userlock(ks)
                        if s.wakeups > 0 then
                                decrement s.wakeups
                                userunlock(ks)
                                set process.waitsem to s
                                set process.semawaken to True
                                wunlock(kss[0].rwlock)
                                cancel_reservations(ks)
                                return index of ks in kss
                        end if
                        increment s.waiting
                        userunlock(ks)
                        lock(ks.lock)
                        queue current_process in ks.q
                        unlock(ks.lock)
                end loop
                wunlock(kss[0].rwlock)
                block current_process
                cancel_reservations(ks)
                return index of process.waitsem in kss
       end
```

Figure 8: Pseudo-code for the system call `altsem`.

Text (code), *Data* (initialized data), *BSS* (uninitialized data and the heap) and *Stack* segments. Segments are made of virtual memory pages. These pages can be either at physical memory or at secondary storage (on-demand paging). Nix does not support swapping. In addition, Nix permits a process to page-in all its pages, that is, disable on-demand paging [2] (i.e. prepaging). Processes can share segments (e.g. processes that share memory, share the Data and the BSS segments).

The semaphore's data structure (`Sem`) is allocated in a shared segment. A userspace address is always checked before being accessed by the kernel. The kernel checks if the address is in a valid segment and the kind of access is allowed. If not, the process is signaled (and it usually dies).

Sharing the semaphore data structure and its lock between kernelspace and userspace is very delicate. In general, when the userspace address is accessed from the kernel (a process executing kernel code in the context of a system call), there may be a page fault. If this is done while holding a lock, the lock will be held for a long time. All the processes that deal with this lock will be stalled for a long time. This practice is not common and it is considered harmful by kernel developers. Nevertheless, our implementation does it

```
cancel_reservations(kss: Array of Ksem)
       begin
                for ks in kss loop
                        set s to ks.sem
                        lock(ks.lock)
                        dequeue process from ks.q
                        if process was in ks.q then
                                userlock(ks)
                                decrement s.waiting
                                userunlock(ks)
                        end if
                        unlock(ks.lock)
                end loop
       end
```

Figure 9: Pseudo-code for the procedure `cancel_reservations`.

carefully. There are no risks:

1. The locks that are held while accessing the `Sem` structure are only related to the semaphore. The readers-writer lock is part of the data structure that defines the memory segment shared by the processes (where the userspace data structure is allocated). Therefore, there is a readers-writer lock for each application (i.e. a group of processes sharing their memory). The rest of the system does not depend on these locks. Should a page fault occur while holding a lock, only the processes that are sharing this segment and accessing a semaphore can be affected.

2. The initialization of the semaphore (the function `initsem`) writes the whole structure. If it is the first time that this page is accessed, there is a page fault. Even if the semaphore has not been initialized (which is an error), the process always acquires the lock of the `Sem` and checks the counters before calling the system. Therefore, the first page fault for the process always happens in userspace. The page frame containing the `Sem` structure is always allocated and associated with the corresponding segment before performing the first semaphore system call. Thus, should a page fault happen in the system call, it is a soft page fault. In other words, the system will not have to allocate a new page frame and fill it or bring the page date from secondary storage. Note that Nix does not support swapping, so the page cannot be evicted to assign the page frame to a different page. Systems that support swapping should mark the page as not swappable to reduce the impact.

3. In our system, it is not possible to generate a page fault while dealing with the semaphores in the system call (if the user program is correct and the whole `Sem` structure is allocated in the same page). The translation for the page is always installed in the page table of the process before the system call is started, because `s.lock` is acquired and s.wakeups is read by the userspace functions. This page cannot become invalid while the system call is in progress.

An incorrect or malicious userspace program would be able to corrupt the lock which is used to protect the userspace data structure. If the kernel code does not take this into

```
userlock(ks: Ksem)
       begin
              set try to ERRORLIMIT
              set s to ks.sem
              set n to 0
              while test_and_set s.lock to LOCKED fails loop
                     if s.state = SEMDEAD then
                            raise_error
                     end if
                     increment n
                     if n mod WARNINGLIMIT = 0 then
                            write warning message to the log
                            decrement try
                            if try = 0 then
                                   write error message to the log
                                   set s.state to SEMDEAD
                                   raise_error
                            end if
                     end if
                     waitwhile(s.lock, LOCKED)
              end loop
       end
```

Figure 10: Pseudo-code for the procedure used to acquire the userspace lock in the semaphore system calls.

account, user programs can provoke deadlocks in processes executing system calls (i.e. privileged mode). For example, if an incorrect program overwrites the value of the lock, all the other processes will be blocked when they try to acquire it. Moreover, if the lock is a spin-lock, the unbounded busy waiting of the deadlocked processes will overload the system. To solve this problem, the kernel's implementation of `lock` used to acquire the spin-lock of the `Sem` structure retries a limited number of times. The number of retries is big enough to discard possible extreme contention scenarios. If all attempts are made and it is not able to acquire the lock, the semaphore is marked as *dead* and the process (that is executing in the context of the system call) is signaled. The procedure `userlock`, shown in Figure 10, is in charge of raising the error. From this moment on, any process trying to acquire the lock of the dead semaphore will be signaled. The signal will be handled by the userspace program (non-privileged mode). In the common case, the signal will cause the user program to exit. Note that, if the lock has been corrupted, the user program is incorrect. The program can handle this signal and try to recover, like with any other signal. From the point of view of the userspace program this situation is similar to, for example, dereferencing a null pointer.

2.5 Atomic Operations

In order to avoid overloading the system while trying to acquire the userspace lock, the implementation of `userlock` uses the `MONITOR` and `MWAIT` instructions of the AMD64 architecture. An extra benefit of using these instructions is power saving in spin-locks. On the other hand, the cost of this approach is latency. This trade-off can be tipped to one side or the other by using different parameters available for these instructions.

The pseudocode is shown in Figure 10. These instructions permit the monitoring of a linear address to detect stores to it. When `MWAIT` is executed, the processor enters a different power state and waits until a store occurs in the address [16]. Therefore, the procedure is not polling the main memory in a closed loop. Any unmasked interrupt or control transfer causes the instruction to exit. The procedure `waitwhile` blocks while the lock is acquired by using the `MWAIT` instruction. The constants `ERRORLIMIT` and `WARNINGLIMIT` control the thresholds to abort the operation.

An apparent disadvantage of our approach is that it uses a spin-lock instead of using atomic instructions to implement the optimistic approach. Atomic operations over an integer value (e.g. using an atomic hardware instruction) intuitively seem faster than using a spin-lock. Surprisingly, this is not true for all cases, at least for our system and hardware (a 32-core K10 machine).

We conducted an experiment [25] to compare incrementing a counter for atomic operations and incrementing the counter while holding a spin-lock (implemented with test-and-set). The experiment was executed for different levels of contention. Two different assembler implementations for atomic increments were measured and compared with locks: (i) a *lock-free-like* version based on the compare-and-exchange primitive that retries the operation if two or more increments collide; and (ii) a version based on the `LOCK` prefix of the AMD64 architecture combined with the `ADD` instruction. The particular implementation of these operations is described in [25].

Figure 11 shows two Tukey diagrams with the time (in ns) needed to increment the counter. The test has been executed with 1, 2, 5, 10, 20, 50 and 100 processes. Each process increments the counter a thousand times. All processes start at the same time (this is ensured by a barrier) to cause the maximum level of contention in each case. Note that with one process, there is no contention. With a hundred processes, the contention is extreme. The program was run with 32 TCs (Time-sharing Cores). Our experiment shows that for our setting, the spin-lock performs better when there is low and high contention. On the other hand, atomic increments are a little faster when there is no contention at all ($\approx \frac{1}{3}$ faster for operations of the order of 50 ns). For a more detailed description of the experiment and further results, see [25].

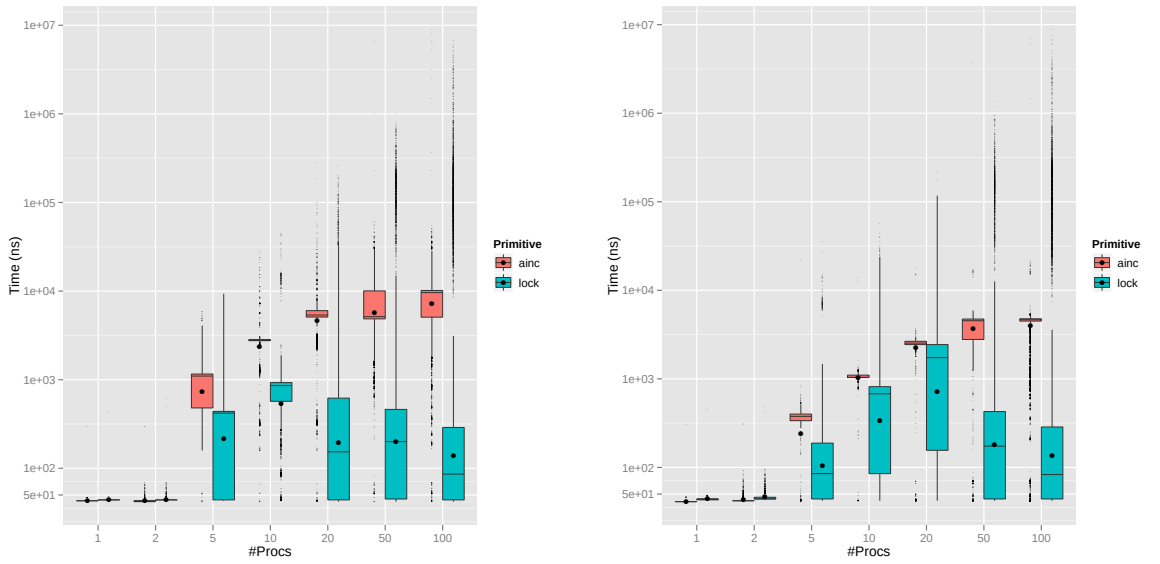
3 Optimistic Tubes

The semaphores are used in Nix to implement *tubes*, that are unidirectional buffered channels [2]. The messages are fixed-size.

Tubes are optimistic channels. Although there are many different implementations of communication channels for CSP-like programming (see for example [22, 6, 33, 12, 19]), it is interesting to describe the implementation of tubes as a use case for the new semaphores. Using the semaphores described above, the implementation of tubes is extremely easy and straightforward. The library has only 335 lines of C code (including headers). Even so, this simple and unoptimized implementation of tubes can compete with the other alternatives available in Nix (results are shown in section 4.3).

A tube is represented by a record with the following fields:

- `msz`: ($\mathbb{N}$) size of the messages.



(a) Ainc implementation (i)

(b) Ainc implementation (ii)

Figure 11: Comparison between `ainc` and `lock` to increment a counter.

```
xsend(t: Tube, m: Message, block: BlockValue): Integer
        begin
            if block != ALREADY then
                if downsem(t.nhole, block = BLOCK) = 0 then
                    return -1
                end if
            end if
            insert_in_buffer(m)
            upsem(t.nmsg)
        end
```

Figure 12: Pseudo-code for the function `xsend`.

- `tsz`: ($\mathbb{N}$) size of the buffer.
- `nmsg`: a semaphore to count the number of messages in the buffer.
- `nhole`: a semaphore to count the number of free slots in the buffer.
- `buffer`: the buffer of the tube.

The userspace interface includes the following operations:

```
Tube* newtube(ulong msz, ulong n);
void freetube(Tube *t);
void trecv(Tube *t, void *p);
void tsend(Tube *t, void *p);
int nbtrecv(Tube *t, void *p);
int nbtsend(Tube *t, void *p);
int talt(Talt a[], int na);
```

```
xrecv(t: Tube, m: Message, block: BlockValue): Integer
        begin
            if block != ALREADY then
                if downsem(t.nmsg, block = BLOCK) = 0 then
                    return -1
                end if
            end if
            extract_from_buffer(m)
            upsem(t.nhole)
        end
```

Figure 13: Pseudo-code for the function `xrecv`.

`Newtube` creates a new tube for messages of size `msz` with a buffer for `n` messages. `Freetube` destroys a tube. `Tsend` sends a message. If the buffer is full, the operation blocks until the message can be placed in the buffer. `Nbsend` is a non-blocking send. If there is no room in the buffer, the operation fails. `Trecv` receives a message from the tube. If there are no messages in the buffer, it blocks. `Nbtrecv` is the non-blocking version.

Finally, `talt` is the non-deterministic choice operation to send or receive from any of the tubes included in the array of `Talt` structures passed as an argument. The `Talt` record has three fields:

- `t`: the tube.
- `m`: the message to send or receive.
- `op`: the operation (blocking/non-blocking send or receive).

The implementation of the tubes is very simple. Figures 12 and 13 show the implementation of `xsend` and `xrecv`. These functions are the core of the library. They are used by both the blocking and non-blocking operations of send/receive.

`Xsend` tries to get a free slot in the buffer (i.e. get a saved wakeup from the semaphore `nhole`). If it is a non-blocking send, the down is non-blocking. The value `ALREADY` is only used for `talt` and can be ignored. In this case, if `downsem` fails, send fails. In the other case, the process blocks until a free slot is available. Then, the message is inserted in the buffer. Finally, it generates a wakeup for the semaphore used to count the number of messages in the buffer. `Xsend` and `Xrecv` are symmetric. `Xrecv` tries to get a message from the buffer (i.e. acquire a wakeup from the semaphore `nmsg`). Then the message is extracted from the buffer.

Functions `insert_in_buffer` and `extract_from_buffer` are atomic.
`Extract_from_buffer` does a busy wait in case the message has not been put in the buffer yet. Note that, if this happens, the message is being copied and the busy wait will be very short. Note that messages are normally small, because channels are aimed at transferring ownership. If we are transferring big chunks of data, we should send a reference (i.e. a pointer to the data). It is not worth blocking the process, even when executing on a Time-sharing Core.

The function `talt` receives an array of `Talt` records. The first loop builds the array of semaphores to call `altsems` and tries to perform the non-blocking operations. If any

```
talt(ts: Array of Talt, ntubes: Integer): Integer
        begin
              ss = new Array of Sem
              for talt in ts from start  loop
                     case talt.operation
                     SEND:
                            add talt.nhole to ss
                     RECEIVE:
                            add talt.nmsg to ss
                     NBSEND:
                            if xsend(talt.t, talt.m, NONBLOCK) != -1 then
                                   return index of talt
                     NBRECEIVE:
                            if xrecv(talt.t, talt.m, NONBLOCK) != -1 then
                                   return index of talt
                     end case
              end loop
              i = altsems(ss, length of ss)
              case ts[i].op
              SEND:
                     xsend(ts[i].t, ts[i].m, ALREADY)
              RECEIVE:
                     xrecv(ts[i].t, ts[i].m, ALREADY)
              end case
              return i
        end
```

Figure 14: Pseudo-code the function `talt`.

non-blocking operation succeeds, `talt` is done. If not, it calls `altsems` with the array of semaphores. Later, when a wakeup has been acquired from any of the semaphores, it calls the corresponding tube operation (`xsend` or `xrecv`). It passes the value `ALREADY` as the third argument. This way, `xsend` or `xrecv` will not call `downsem`, because the wakeup has been already acquired. The rest of the operation proceeds as explained before.

4 Evaluation

This section shows some of the results of three different experiments we made to evaluate the implementation of the optimistic semaphores, the non-deterministic choice operation and the tubes. The complete results are too large to be presented in this paper. This section provides the most representative and interesting results of these experiments[5].

In all the experiments, the time is gathered using the core timestamp. The timestamp is obtained by executing a userspace instruction. The clocks of all the cores are synchronized while booting and the experiments were always executed right after. The relative drift of the clocks is below the margin of error of the measurements presented here.

[5]See also http://lsub.org/export/semeval.pdf

```
experiment1()
        begin
                for i in 0..1000 loop
                        set t1 to current time
                        dowmsem(mutex)
                        for j in 0..CSS-1 loop
                                waste some time
                                increment the shared counter
                        end loop
                        upsem(mutex)
                        set t2 to current time
                        set time to t2 - t1
                        save time in experiment_data
                        for j in 0..NSS-1 loop
                                 waste some time
                        end loop
                end loop
        end
```

Figure 15: Pseudo-code for the routine executed by each process in the Experiment 1. The variable `time` does not need to perform a system call to save the time (it is stored in an array).

4.1 Experiment 1: Simple Mutex

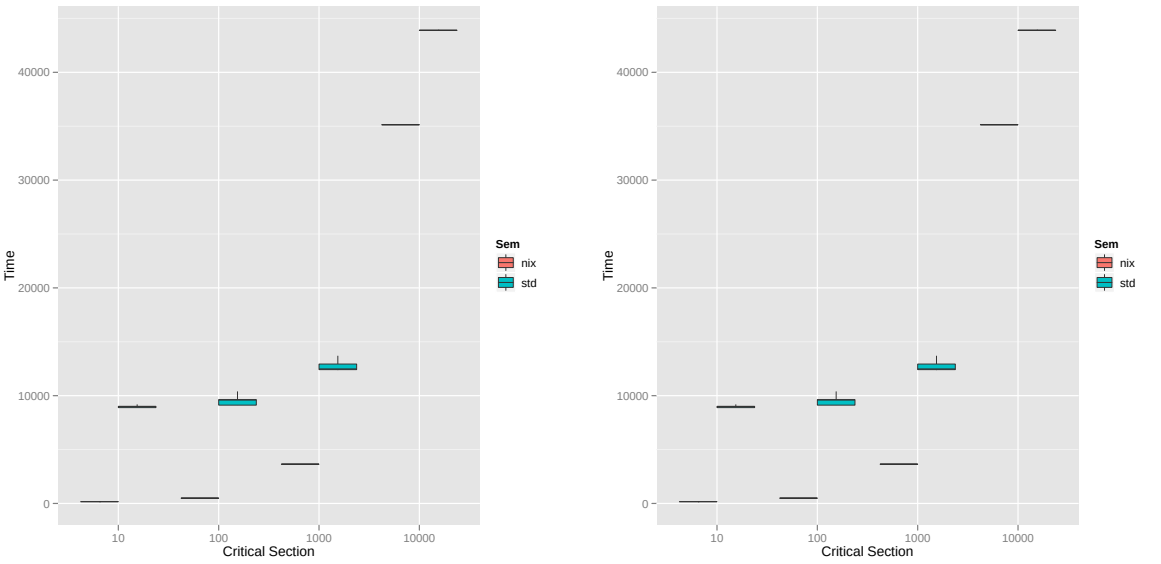
This experiment measures the time for entering a critical section, using a semaphore as a mutex. In the critical section, each process gets the value of a shared counter, wastes some time looping, increments the local copy, and updates the shared counter. The number of iterations of the loop is *the critical section size*, from now on, CSS. The process also wastes some time in a loop outside the critical section. The number of iterations of this loop is the *non-critical section size*, from now on, NSS. Each process repeats the operation 1000 times. The pseudocode is shown in Figure 15.

For this experiment, two types of graphs have been generated:

- α *graphs* are Tukey diagrams showing the time within the critical section, including the time required to acquire and release the mutex, in ns. It does not include the time wasted outside the non-critical section. The size of the non-critical section (NSS), the number of cores, and the number of concurrent processes are fixed for each graph. The X-axis represents the size of the critical section (CSS). The times for the standard semaphores (`semacquire` system call) are labeled as `Std`. The times for the optimistic semaphores are labeled as `Nix`.

  Figure 16 shows the α graphs for 5 processes, for blocking operations. The experiment was executed in Time-sharing Cores. The α graphs for $NSS = 100$ and $NSS = 1000$ are omitted because they show similar results. The α graphs for 10 processes are also similar. 5 processes does not cause high contention. As expected, the optimistic semaphores are faster when performing blocking operations.

  With more processes, the contention is higher and results are quite different. Figures 17 and 18 show the α graphs for 10, 20, 50 and 100 processes, with $NSS = 10000$. The results for lower values of NSS are quite similar. As shown, with 100 concur-



(a) 5 procs, NSS=10

(b) 5 procs, NSS=10000

Figure 16: α graphs for 5 procs running on 32 Time-sharing Cores (TC), for blocking operations. Time is in ns. The X axis is measured in iterations.

rent processes, the contention is too high. In this case, the optimistic semaphores perform worse than the standard ones. Note that the contention in this case is very high and unrealistic.

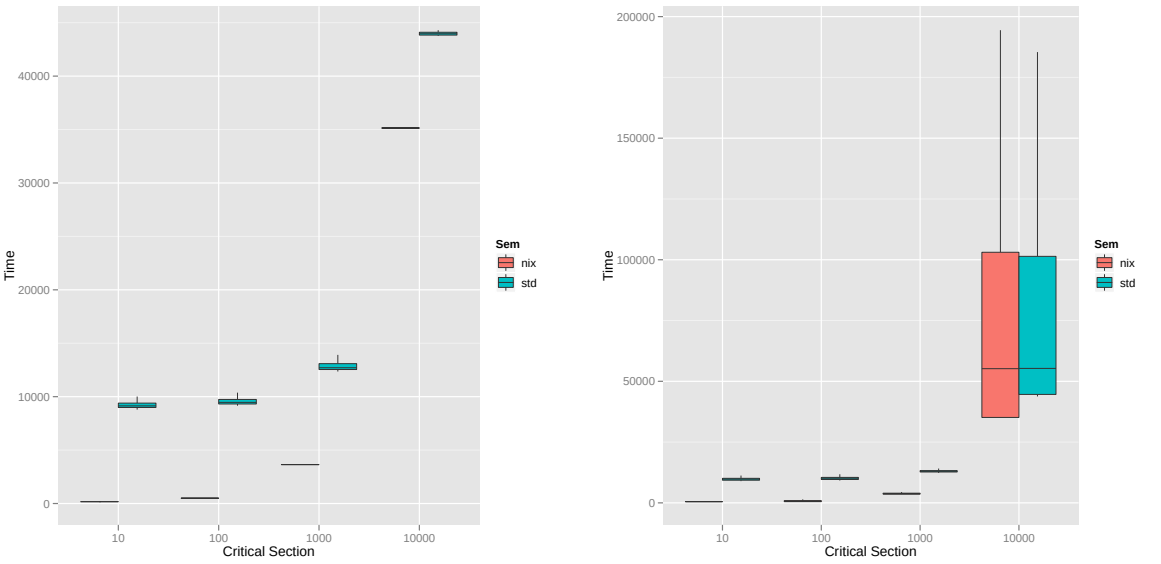
We will omit results for non-blocking downs. The results for non-blocking downs are similar to the ones shown in figure 16. Note that in our implementation, a non-blocking down never enters the kernel. The only notable effect we have detected is that, when the contention is extreme, the time to perform the non-blocking down suffers a large variance. For example, this happens for 100 processes and $CSS = 10000$.

Figure 19 shows the results of the same experiment executed on Application Cores. Again, we only provide the results for $NSS = 10000$ (the results for lower values are similar). In this case, the gain of using an optimistic approach is evident. The optimistic semaphores are several orders of magnitude faster than the standard semaphores.

- β *graphs* are Tukey diagrams showing the time required to increment the shared counter for different ratios between the size of the critical section and the non-critical section. The X-axis represents the ratio. The number of cores and processes are fixed for each graph. Figure 20 shows the β graphs for Time-sharing Cores for 5, 20, 50 and 100 processes. Figure 21 shows the β graphs for Application Cores.

4.2 Experiment 2: Semalt

This experiment measures the non-deterministic choice operation for semaphores, `altsems`. In this experiment, there is an array of counters. Each counter is protected by a semaphore



(a) 10 procs, NSS=10000

(b) 20 procs, NSS=10000

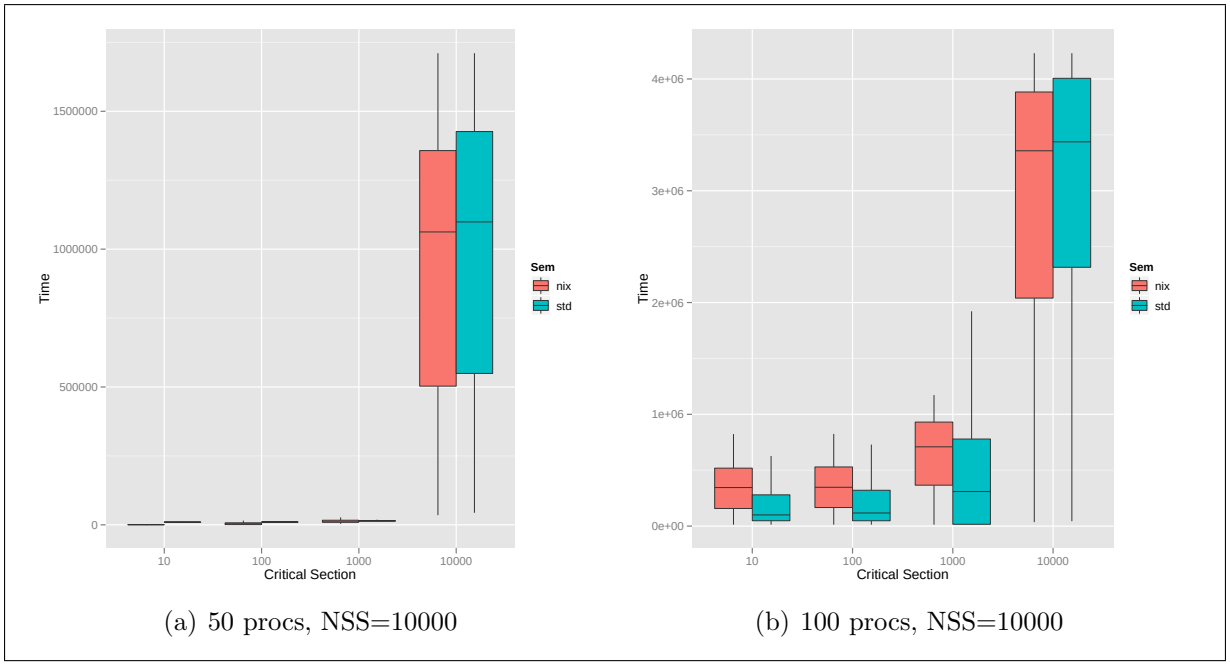
Figure 17: α graphs for different numbers of concurrent processes running on 32 Time-sharing Cores (TC). Time is in ns. The X axis is measured in iterations.

(used as a mutex), so the number of semaphores is equal to the number of counters to increment. The processes call `altsems` and increment the corresponding counter (the counter associated with the acquired semaphore).

As in the previous experiment, the critical section includes acquiring the current state of the counter, wasting some time within a loop (CSS), and updating the counter. The time measured includes the time to acquire the wakeups, increment the counter, and release the wakups. As in the previous experiment, the process also wastes some time outside the critical section (NSS). Each process repeats the operation 1000 times.

We compare the new operation with a straightforward, ad-hoc implementation of this program using the standard semaphores [32]. Note that this implementation does not replace the `altsems` operation (it does not provide the semantics of `altsems`). This implementation uses a central semaphore to control the access to the semaphores in the array. The pseudo-code is shown in Figure 22.

For this experiment, we have generated Tukey diagrams (γ *graphs*) showing the time required to increment the shared counter (time in the critical section, including the time required to acquire and release the mutex) in ns. Note that the time does not include the time wasted in the non-critical section. The size of the non-critical section (NSS), the number of cores, the number of concurrent processes, and the number of semaphores in the array (and shared counters) are fixed for each graph. The X-axis represents the size of the critical section (CSS). The times for the implementation with standard semaphores are labeled as `Std`. The times for the optimistic semaphores are labeled as `Nix`. Figure 23 shows the results for 10 and 50 processes executing the operation over 2 and 10 semaphores, on Time-sharing Cores.



(a) 50 procs, NSS=10000

(b) 100 procs, NSS=10000

Figure 18: α graphs for different numbers of concurrent processes running on 32 Time-sharing Cores (TC). Time is in ns. The X axis is measured in iterations.

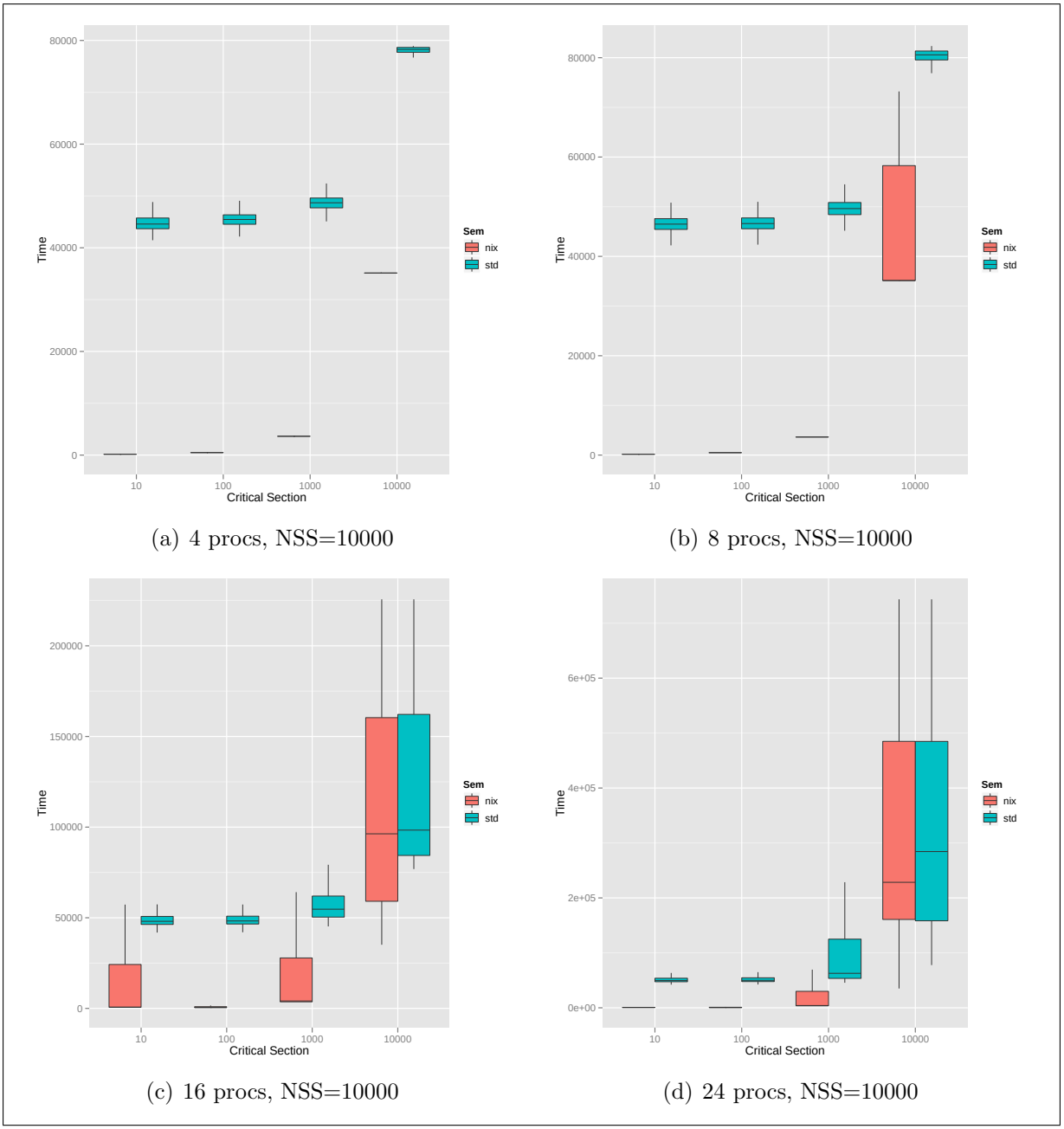
4.3 Experiment 3: Tubes

This experiment creates one process (*pinger*) that sends a message (*ping*) through a tube and receives a response (*pong*). Each execution of the experiment completes 1000 *ping-pong* rounds.

In this experiment we compare the tubes with two alternative communication channels available in Nix: the channels of the Plan 9 thread library [31] and pipes. In the case of tubes and channels, the response is received though another tube/channel. In the case of pipes, the response is read from the same pipe, because in Nix (as in Plan 9) pipes are full-duplex. Note that the *pong* message received does not necessarily match the *ping* message in this round (because of the buffering). Thus, this experiment does not measure the time needed for a round-trip. This experiment measures the time to complete a round of the *pinger*. Figure 24 illustrates the three programs. The number of *pongers* is a variable of the experiment. We have executed the experiment with 1, 8, 16 and 24 *pongers*.

The tubes and the channels used in this experiment have a buffer of size 32 (messages). Note that a pipe has a 8 Kb buffer. The size of the messages is another variable for the experiment. We have used messages of 8, 32, 64, 512 and 4096 bytes.

The results are presented in ϵ graphs, that are Tukey diagrams showing the time required to complete a round. These graphs also depict the mean (with a point). Each graph presents the times for the three IPC mechanisms, for different message sizes. The number of *pongers* is fixed for each figure. Figure 25 shows the results for Time-sharing Cores. Figure 26 shows the results for Application Cores.



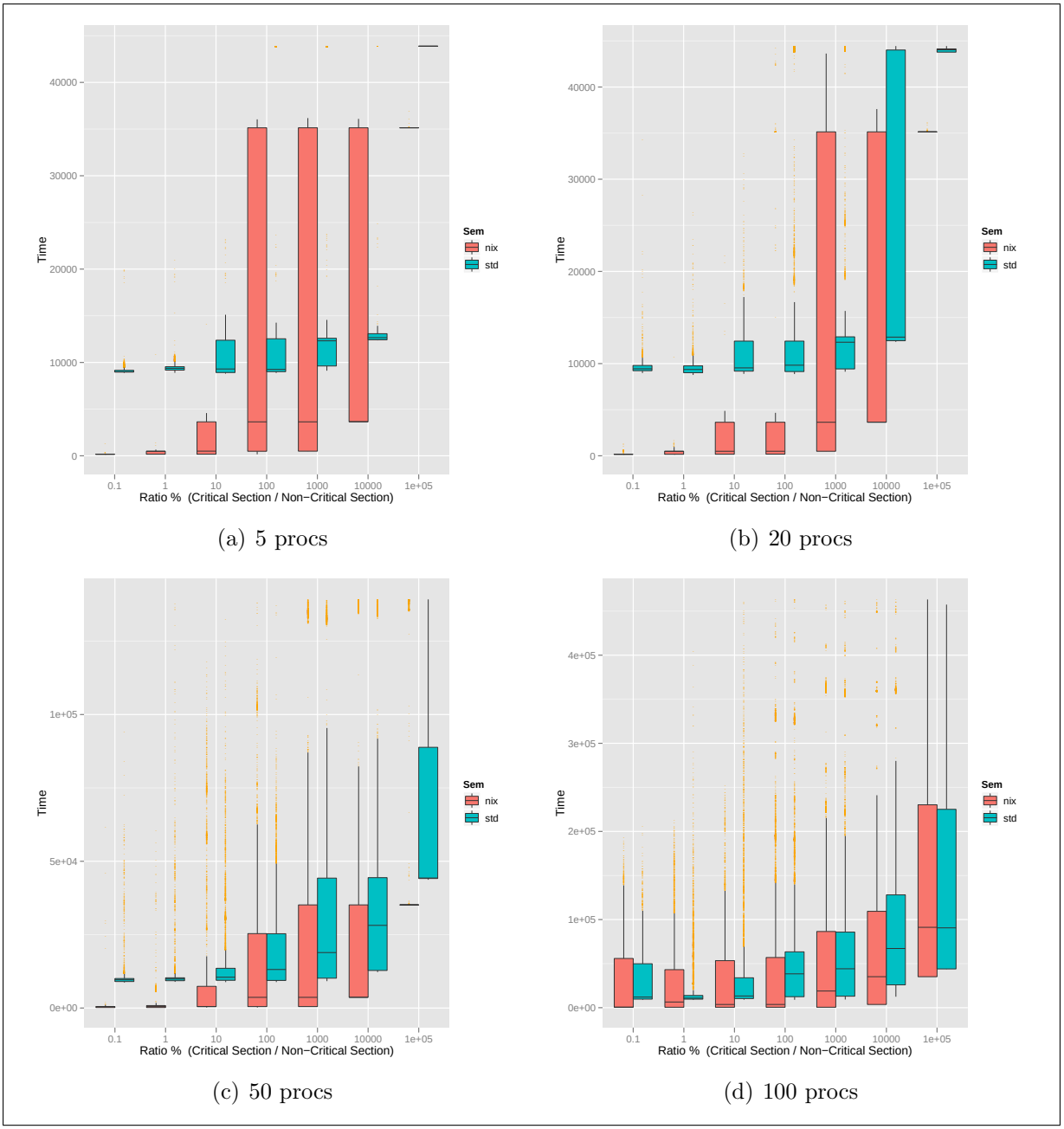
(a) 4 procs, NSS=10000

(b) 8 procs, NSS=10000

(c) 16 procs, NSS=10000

(d) 24 procs, NSS=10000

Figure 19: α graphs for different numbers of concurrent processes running on Application Cores (AC). Time is in ns. The X axis is measured in iterations.



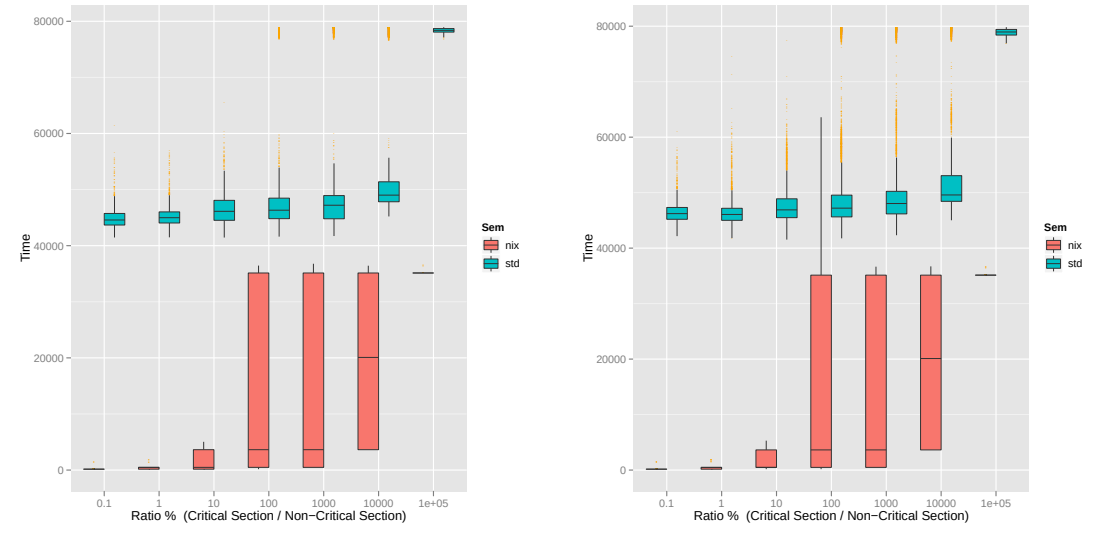
(a) 5 procs

(b) 20 procs

(c) 50 procs

(d) 100 procs

Figure 20: β graphs for different numbers of concurrent processes running on Time-sharing Cores (TC). Time is in ns.



(a) 4 procs

(b) 8 procs

(c) 16 procs

(d) 24 procs

Figure 21: β graphs for different numbers of concurrent processes running on Application Cores (AC). Time is in ns.

```
simulated_altsems()
        begin
                for i in 1..1000 loop
                        down(mainsem, BLOCK)
                        set acquired to false
                        set j to 1
                        while j <= NUMSEMS and not acquired loop
                                set acquired to down(sems[j], DONTBLOCK)
                                if not acquired then
                                        increment j
                                end if;
                        end loop
                        critical_section(j)
                        up(sems[j])
                        up(mainsem)
                        non_critical_section()
                end loop
        end
```

Figure 22: Pseudo-code for the ad-hoc program.

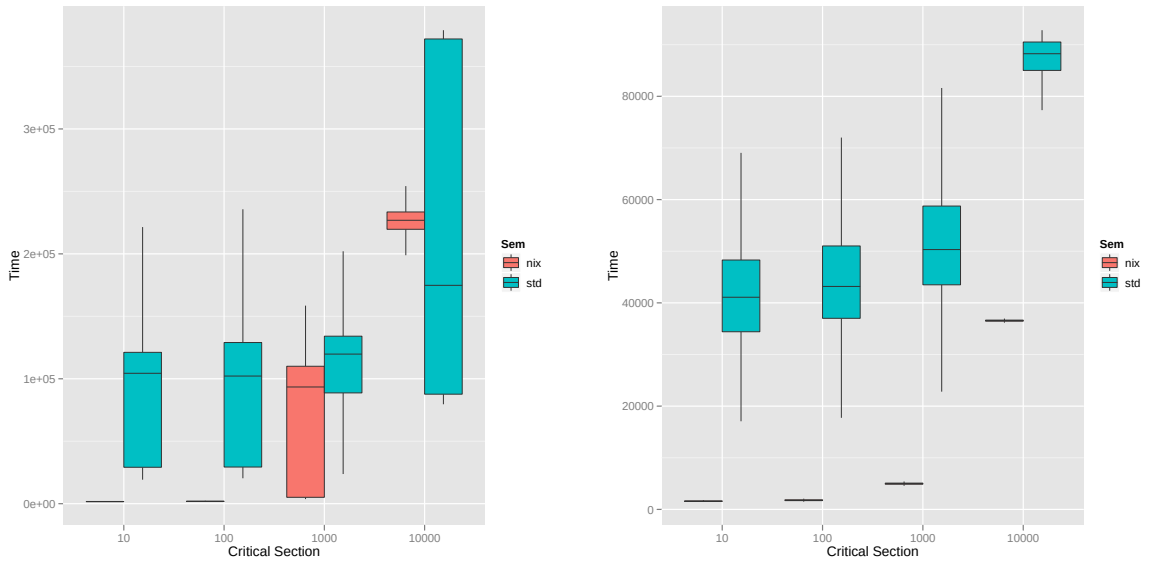
5 Related work

Semaphores are a well known synchronizing primitive. Most operating systems provide them. The related work is too broad to be covered here exhaustively. In this section we discuss some popular implementations and some related to our work.

As far as we know, no other implementation of semaphores provides a non-deterministic choice operation like *altsems*. Other authors have proposed new operations for semaphores. For example, Tai and Carver [28] proposed an operation called *VP* that operates on two semaphores. It permits *up* to be done on the first semaphore and *down* on the second, guaranteeing that the caller gets the next saved wakeup from the second semaphore if the operation is started. Other systems also include operations over a set of semaphores. For example, in System V, the *semop* system call permits the application of a different operation to each of the semaphores given in an array. The system call succeeds when all the operations have been performed [10]. Note that this is completely different from *altsems*. In *altsems*, only one operation is committed (any of the operations). The semantics are totally different.

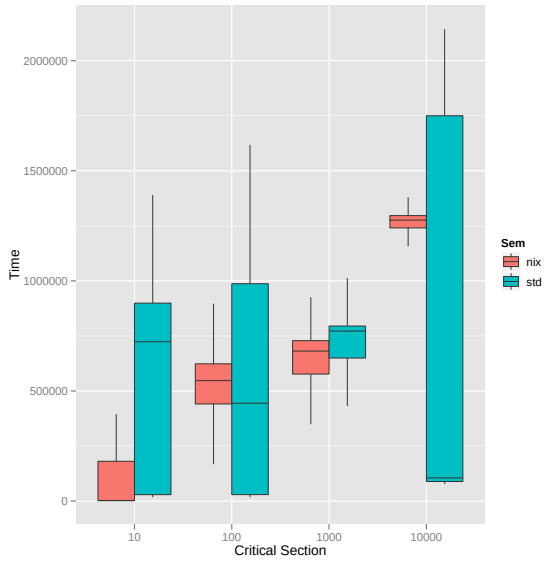
An important difference between our semaphores and others is that ours provide strong semantics (*block-queue strong semantics* [26, 21]). This means that they ensure that an unblocked process will be able to continue even if a new process issues a down operation before the unblocked process begins to run again. Weak semaphores cannot make this guarantee. This point is important when implementing other abstractions upon semaphores, for example monitors [21]. Most POSIX-compliant semaphores provide weak semantics [21]. Semaphores based on busy waiting also provide weak semantics, see for example the semaphores of the Multicomputer and Cedar multiprocessors [5].

Semaphores are typically heavy kernel objects. This means that a system call is always needed to operate on a semaphore, regardless of contention. See for example the implementation of Unix System V [10]. The interface of the System V [10] and XSI [27, 23] semaphores is extremely complex. For example, the system call `semctl` can

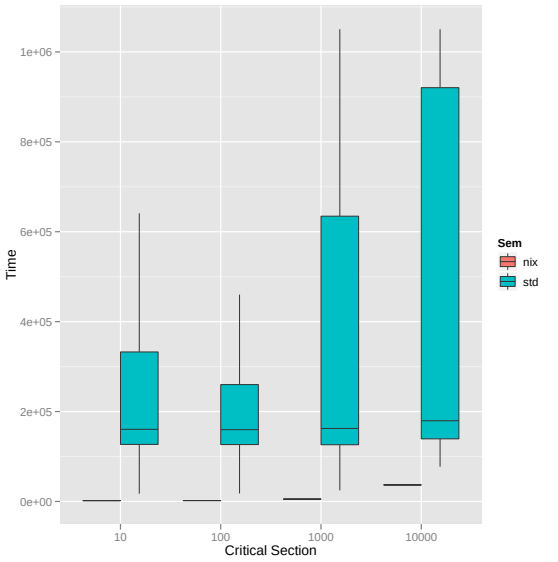


(a) 10 procs, 2 semaphores, NSS=10000

(b) 10 procs, 10 semaphores, NSS=10000



(c) 50 procs, 2 semaphores, NSS=10000



(d) 50 procs, 10 semaphores, NSS=10000

Figure 23: γ graphs for different numbers of semaphores and processes, running on Time-sharing Cores (TC). Time is in ns.

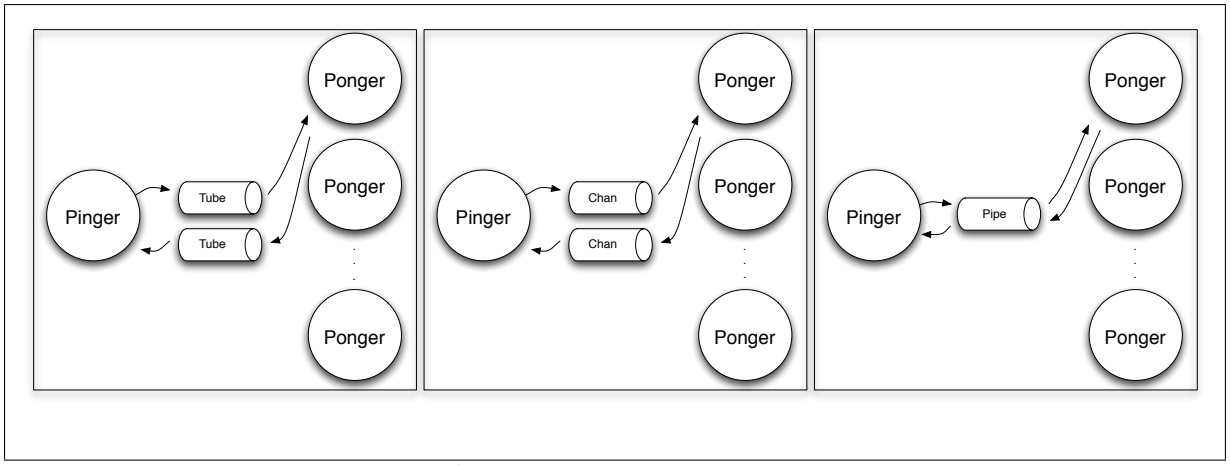


Figure 24: Experiment 3.

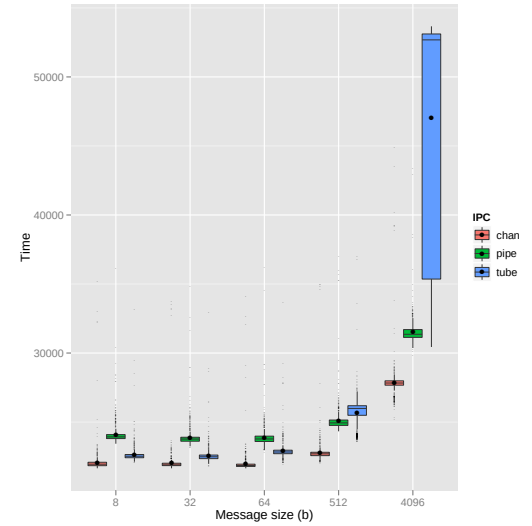
accept up to ten different commands [27], operations can be undone, the semaphores can be managed programmatically and manually with administration tools, etc. They also have serious flaws [27]. First, they are kernel objects that share a unique id space and must be explicitly allocated and deallocated. Moreover, there may be leaks if they are not deallocated before the process dies. Another issue is that they cannot be created and initialized atomically (it requires two operations).

POSIX semaphores are simpler than System V and XSI semaphores. POSIX defines two kinds of semaphores: named and unnamed. Named semaphores are similar to named pipes. They are used to synchronize processes that do not share memory [23], like System V semaphores. Unnamed semaphores in POSIX are just shared memory variables. There are different implementations of POSIX semaphores. The Linux implementation of POSIX semaphores is described later in this section.

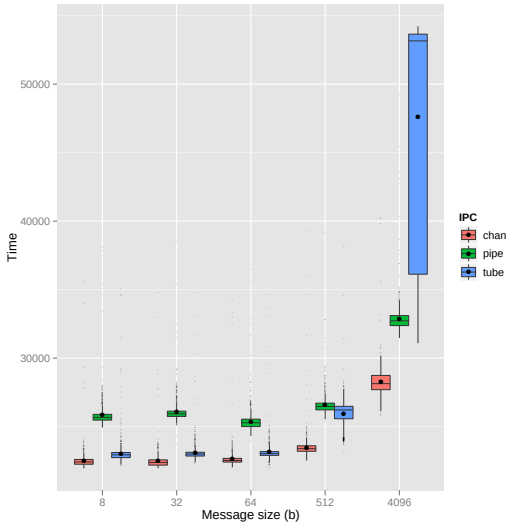
The Plan 9 implementation [32] addresses the flaws of System V semaphores. The interface is very simple (two system calls). A semaphore is represented by an integer value located in a shared memory segment, so processes do not have to allocate and deallocate them from a kernel data structure and semaphore leaks are not possible. Note that our implementation also prevents leaks, because the semaphore's structures are allocated together with the shared memory segment or in the segment itself. The shared segment and the corresponding structures are deallocated when the processes exit.

Some platforms provided hardware support for mutual exclusion, like Sequent Balance Systems [30]. In these systems, each processor has a controller named SLIC. The SLICs provide a set of binary semaphores named *gates*, whose values are coherent for all processors. The SLICs are directly connected to the processors and communicate through their own bus, so polling the gates is not a problem. Dynix (the Unix system for Sequent machines) implemented general semaphores upon this low-level mechanism [30]. The semaphore data structure (basically a counter and a queue) is protected by a gate. This kind of hardware support would be very useful for our implementation, mainly to implement the lock that protects the userspace semaphore data structure.

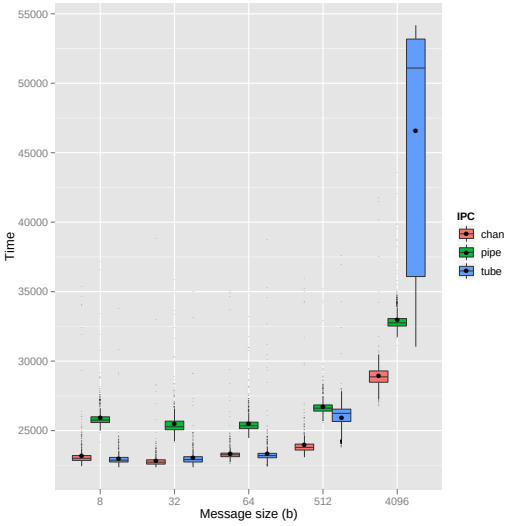
There are several implementations of optimistic semaphores [8, 24, 15]. For example, BeOS *benaphores* [24] were based in atomic increments and decrements of an integer value located in shared memory. When the *benaphore* has no saved wakeups (i.e. the counter is 0), the process blocks in a real semaphore. Another example is the Linux



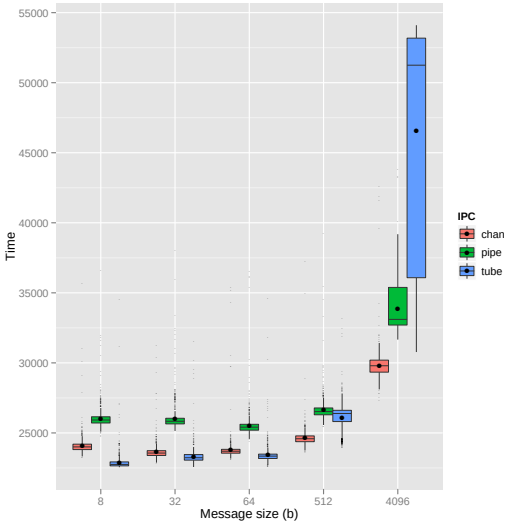
(a) 1 ponger



(b) 8 pongers

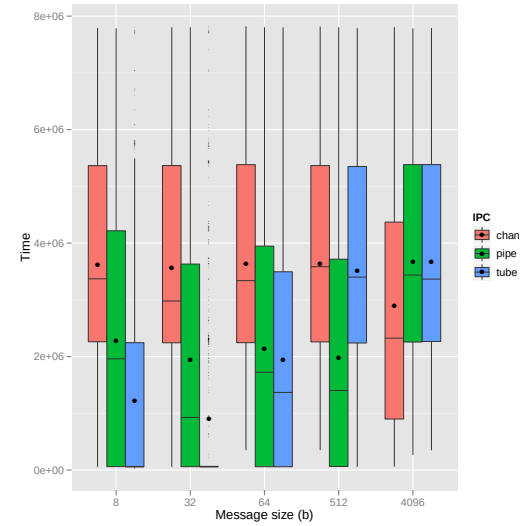


(c) 16 pongers

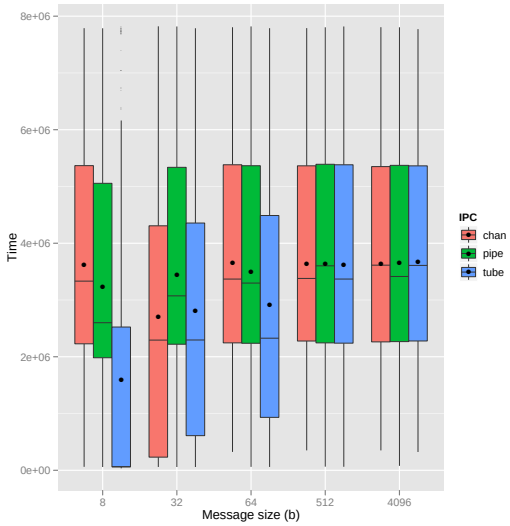


(d) 24 pongers

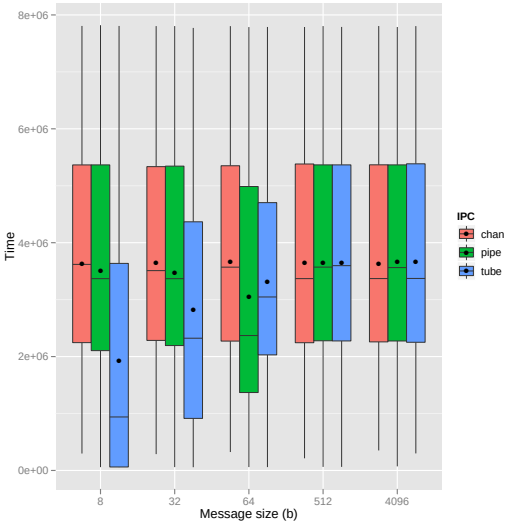
Figure 25: ϵ graphs, running in TCs. Time is in ns.



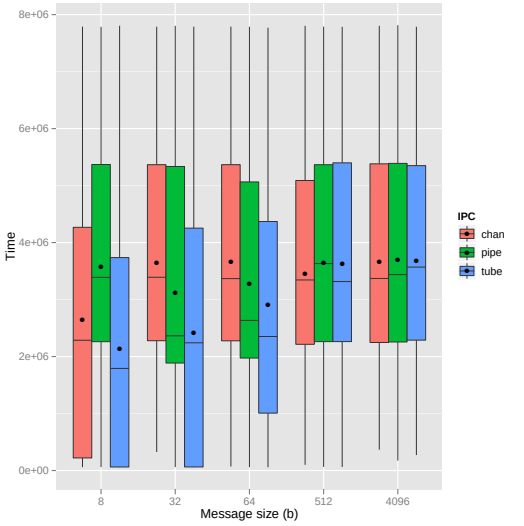
(a) 1 ponger



(b) 8 pongers



(c) 16 pongers



(d) 24 pongers

Figure 26: ϵ graphs, running in ACs.Time is in ns.

implementation of POSIX semaphores (e.g. GNU libc 2.9). They are also based on atomic increments/decrements and a synchronization mechanism called *futex* (which is discussed below). Mullender and Cox also implemented optimistic semaphores for Plan 9 [15]. This implementation uses two counters, one for userspace and the other for the kernel. The counters are modified with atomic increments and decrements. This implementation is very elegant, but it provides neither an optimistic non-blocking down operation nor a non-deterministic choice operation for semaphores. Our implementation of optimistic semaphores is different to these three examples. We use a lock to protect the data structure that is accessed from userspace and kernelspace.

Linux provides a different primitive called futex (fast userspace mutex) [9]. A futex is represented by an integer value in shared memory and a queue within the kernel. In short, a futex provides two operations: (i) *wait* blocks the process if the integer value is equal to a value provided as an argument (in this case, the process enters the kernel and is queued); (ii) *wake* wakes up a number (specified by an argument) of processes that are sleeping in the queue. In an uncontended scenario, processes can modify the integer atomically and call *wait* without performing system calls. Futexes are difficult to understand and use [7]. On the other hand, semaphores are well understood and the literature describes multiple algorithms based on them. As stated before, the Linux 2.6 implementation of POSIX semaphores is based on futexes [13]. It is clear that semaphores are a more convenient primitive.

In this paper, we also provide a simple implementation of unbuffered channels as a use case for the semaphores. There are many different implementations of communication channels described in the literature. For example, FastFlow [1] is a framework for POSIX threads that provides streams for multicore systems. The framework supplies pre-defined algorithmic skeletons for most popular parallel patterns. It follows a lock-less approach and achieves good latency and throughput. Other systems provide unbuffered channels to create concurrent programs following the approach of Hoare's CSP [11, 3], like Google Go [12], Plan 9 (its thread library), [31], Limbo [22, 6], Alef [33], and Newsqueak [19]. For instance, the Plan 9 thread library permits the creation of buffered and unbuffered channels to communicate coroutines and/or processes. The tubes provide a similar interface. The main difference is that tubes are aimed at communicating between Nix ACs efficiently.

6 Conclusions

We have described a new implementation of semaphores for Nix, a manycore operating system that supports different kinds of roles for the cores. Some kinds of cores are specially susceptible to the cost of system calls. The semaphores are optimized for this kind of role. The semaphores provide strong semantics and their implementation is unusual.

The semaphores also include a novel operation, a non-deterministic choice operation over a set of semaphores. This operation makes it easier to implement communication channels on top of the semaphores.

We provide the implementation of tubes, a new kind of communication channel, as a use case for our semaphores. Although it is a naïve and straightforward implementation, it can compete with other kinds of channels when executing on Timesharing Cores and

they are much better for the case they are optimized for: small messages and Application Cores. They can be used for direct communication from user/kernel space to user/kernel space.

The paper also includes different comparative analyses of semaphores and tubes with other alternative mechanisms. The experimental results show that the new semaphores perform well in the target scenario. The simple implementation of tubes shows that the non-deterministic choice operation for semaphores can be particularly useful to implement other high level mechanisms, like channels for CSP programs like those of Go and Plan 9, sockets, and so on. Lastly, the study also shows that our unorthodox approach is viable.

References

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core. In *Programming Multi-core and Many-core Computing Systems, ser. Parallel and Distributed Computing, S. Pllana*, page 13, 2012.

[2] Francisco J. Ballesteros, Noah Evans, Charles Forsyth, Gorka Guardiola, Jim McKie, Ron Minnich, and Enrique Soriano. Nix: a case for a manycore system for cloud computing. *Bell Labs Technical Journal*, 17(2):41–54, 2012.

[3] Russ Cox. Bell labs and CSP threads. http://swtch.com/ rsc/thread.

[4] Edsger W. Dijkstra. The structure of the THE-multiprogramming system. *Commun. ACM*, 11(5):341–346, May 1968.

[5] A. Dinning. A survey of synchronization methods for parallel computers. *Computer*, 22(7):66–77, 1989.

[6] Sean M. Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard W. Trickey, and Philip Winterbottom. The Inferno^{TM} operating system. *Bell Labs Technical Journal*, 2(1):5–18, 1997.

[7] U. Drepper. Futexes are tricky. *Futexes are Tricky, Red Hat Inc, Japan*, 2005.

[8] Neil Dunstan. Semaphores for fair scheduling monitor conditions. *SIGOPS Oper. Syst. Rev.*, 25(3):27–31, May 1991.

[9] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *AUUG Conference Proceedings*, page 85. AUUG, Inc., 2002.

[10] Berny Goodheart and James Cox. *The magic garden explained: the internals of UNIX System V Release 4: an open systems design.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[11] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[12] *The Go Programming Language*, 2013.

[13] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, San Francisco, CA, USA, 1st edition, 2010.

[14] *Lock. NIX Programmer's Manual, Section 2: Library Functions*, 2012.

[15] S. Mullender and R. Cox. Semaphores in Plan 9. In *3rd International Workshop on Plan 9*, pages 53–61, 2008.

[16] *AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions*, 2011.

[17] Yoonho Park, Eric Van Hensbergen, Marius Hillenbrand, Todd Inglett, Bryan Rosenburg, Kyung Dong Ryu, and Robert W Wisniewski. Fusedos: Fusing lwk performance with fwk functionality in a heterogeneous environment. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 211–218. IEEE, 2012.

[18] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In *Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, London, UK, 1990.

[19] Rob Pike. The implementation of Newsqueak. *Software: Practice and Experience*, 20(7):649–659, 1990.

[20] Eric S. Raymond. *The Art of UNIX Programming*. Pearson Education, 2003.

[21] Kenneth A. Reek. The well-tempered semaphore: theme with variations. *SIGCSE Bull.*, 34(1):356–359, February 2002.

[22] Dennis M. Ritchie. *The Limbo Programming Language, Inferno Programmer's Manual, Vol. 2*, 2000.

[23] K. A. Robbins and S. Robbins. *Unix Systems Programming: Communication, Concurrency, and Threads*. Prentice Hall PTR, 2003.

[24] Benoit Schillings. Be engineering insights: Benaphores. http://www.haiku-os.org/legacy-docs/benewsletter/Issue1-26.html#Engineering1-26.

[25] E. Soriano-Salvador and G. Guardiola. Atomic Increments. In *7th International Workshop on Plan 9*, pages 56–63, 2012.

[26] Eugene W. Stark. Semaphore primitives and starvation-free mutual exclusion. *J. ACM*, 29(4):1049–1072, October 1982.

[27] Richard W. Stevens and Stephen A. Rago. *Advanced Programming in the UNIX(R) Environment (2nd Edition)*. Addison-Wesley Professional, 2005.

[28] K. C. Tai and Richard H. Carver. Vp: a new operation for semaphores. *SIGOPS Oper. Syst. Rev.*, 30(3):5–11, July 1996.

[29] A.S. Tanenbaum and A.S. Woodhull. *Operating systems: design and implementation*, volume 68. Prentice Hall, 1997.

[30] S. Thakkar, P. Gifford, and G. Fielland. The balance multiprocessor system. *Micro, IEEE*, 8(1):57–69, 1988.

[31] *Thread. Plan 9 Programmer's Manual, Section 2: Library Functions*, 2012.

[32] *Semacquire. Plan 9 Programmer's Manual, Section 2: Library Functions*, 2012.

[33] Phil Winterbottom. *Alef Language Reference Manual. Plan 9 Programmer's Manual, Vol. 2*, 1995.