A Lock-Free Multiprocessor OS Kernel

Henry Massalin and *Calton Pu*
Department of Computer Science
Columbia University
New York, NY 10027

Technical Report No. CUCS-005-91

calton@cs.columbia.edu

Revised June 19, 1991

Abstract

Typical shared-memory multiprocessor OS kernels use interlocking, implemented as spin-locks or waiting semaphores. We have implemented a complete multiprocessor OS kernel (including threads, virtual memory, and I/O including a window system and a file system) using only lock-free synchronization methods based on Compare-and-Swap. Lock-free synchronization avoids many serious problems caused by locks: considerable overhead, concurrency bottlenecks, deadlocks, and priority inversion in real-time scheduling. Measured numbers show the low overhead of our implementation, competitive with user-level thread management systems.

Contents

1 Introduction

Mutual exclusion is commonly accepted as the standard synchronization technique in OS kernels. In multiprocessors, mutual exclusion may be achieve through waiting locks and spin-locks. Anderson [1] analyzed the performance and overhead trade-offs of spin-locks for multiprocessor systems, finding considerable potential for system performance degradation due to mutual exclusion.

Herlihy [4] has shown that in wait-free and non-blocking synchronization, atomic read-modify-write instructions such as compare-and-swap are more powerful than test-and-set, the instruction commonly used in the implementation of spin-locks. However, there has been no empirical evidence that OS kernels using this kind of lock-free synchronization methods achieve better performance than state-of-art multiprocessor OS kernel implementations using lock-based synchronization (e.g., spin-locks and semaphores), such as Mach [3] and Psyche [11].

The version 1 (abbreviated V.1) of Synthesis kernel implemented on a dual-68030 Sony NEWS workstation is based entirely on lock-free synchronization, developed from the optimistic synchronization methods developed for the single-processor version 0 of Synthesis [8]. The implementation of Synthesis V.1 shows three things. First, lock-free synchronization is sufficient for all synchronization needs of a multiprocessor OS kernel supporting threads, virtual memory and all the usual I/O. Second, a lock-free multiprocessor OS kernel is practical. Third, such an OS kernel achieves very high performance.

The remaining of this paper is organized as follows. Section 2 discusses the general problem of synchronization in an OS kernel. Section 3 describes the lock-free objects the Synthesis kernel uses. Sections 4, 5, and 6 describe the use of lock-free synchronization in the implementation of threads, virtual memory and input/output. Section 7 summarizes related work and Section 8 concludes the paper.

2 Synchronization in OS Kernels

2.1 Disabling Interrupts

There are three basic kinds of synchronization problems in an OS kernel: hardware interrupts, software interrupts (also called signals), and critical sections. Inside interrupt handlers, temporary disabling of hardware interrupts is necessary for saving the hardware event information. Otherwise, hardware interrupts can remain enabled even during the other kinds of synchronization.

However, in a single processor kernel (including many flavors of UNIX), all three kinds of problems can be solved cheaply by disabling hardware interrupts. While interrupts are disabled the executing procedure is guaranteed to continue uninterrupted, so no interleaving may occur. Since disabling and enabling interrupts cost only two instructions, which is orders of magnitude cheaper than other synchronization mechanisms such as semaphores, its use is widespread. For example, 112 of the 653 procedures that make up the Sony NeWS kernel (a BSD 4.3 derivative) disable interrupts, most of which protect kernel data structures under updates. In processing scheduling and dispatching, all interrupts are disabled to prevent context switches while the run queue is examined or modified, for example changing priorities.

Despite the cost advantage of disabling interrupts, it has several limitations. First, interrupts cannot remain disabled for too long a period of time, otherwise frequent hardware interrupts such as clock may be lost. This places a limit on the length of execution path within critical regions protected by disabled interrupts. Second, in a shared-memory multiprocessor, data structures may be modified by different CPUs. Therefore, some explicit synchronization mechanism is needed.

Disabling signals is considerably more complex than disabling interrupts. Before a critical section, a kernel call disables signals. The signal delivery routine must test whether that particular signal is currently allowed. If not, take the appropriate action to stack up the signal processing at the end of the critical section. (One example is procedure chaining in Synthesis [8].) After each critical section, the system must test and execute any pending signals. More serious than kernel overhead, disabling signals may cause processes to become stuck when expected events do not happen. For example, the NFS kernel calls in SUNOS cannot be interrupted with Control-C, so a server crash makes its clients to loop until server is restored. Thus disabling signals is not an ideal solution to protect critical sections.

2.2 Locking Synchronization Methods

Mutual exclusion can protect a critical section by allowing only one process at a time to execute in it. The many styles of algorithms and solutions for mutual exclusion may be divided into two kinds: busy-waiting (usually implemented as spin-locks) and blocking (usually implemented as semaphores). Spin-locks sit in tight loops while waiting for the critical region to clear. Blocking semaphores (or monitors) explicitly send a waiting process to a queue. When the currently executing process exits the critical section, the next process is dequeued and allowed into the critical section.

The main problem with spin-locks is the waste of CPU time while spinning. The justification is that every process is dedicated to some processor. This assumption is false when multiple threads are mapped to a physical processor. The main difficulty with waiting locks is the considerable overhead to maintain a waiting queue and to set/reset the semaphore.

Besides the overhead in acquiring and releasing locks, a major disadvantage of using locks is the potential for poor performance due to lock contention. For example, Mach uses a single lock for the global run-queue. This can cause significant contention if several processors try to access the queue at the same time, as would occur when the scheduler clocks are synchronized [3]. One way to reduce the lock contention in Mach relies on scheduling "hints" from the programmer. For example, hand-off hints may give control directly to the destination thread, bypassing the run queue. Although hints may decrease lock contention for specific cases, their use is difficult and their benefits uncertain.

Besides the overhead and lock contention problems, there are two additional problems with locks. First, locks may cause deadlocks. Typically OS kernel avoid deadlocks by imposing a request order for the resources. Second, in real-time systems locks may cause priority inversion, when a low priority process in a critical section is preempted for some reason and causes other high priority processes waiting for that very critical section. There are sophisticated solutions for priority inversion problem, but they contribute to make locks less appealing. For all these reasons, we want to build a lock-free multiprocessor OS kernel.

2.3 Lock-Free Synchronization Methods

Compare-and-Swap is the foundation of lock-free synchronization. It is designed to atomically update one or two words (both are supported by the Motorola 68030 processor on our machine). Two-word Compare-and-Swap lets us efficiently implement many basic data structures such as stacks, queues, and linked lists because we can atomically update both a pointer and the location being pointed to.

Herlihy [4] defines an object to be *wait-free* if it guarantees that *each* process will complete an operation in a finite number of steps. An object is *non-blocking* if it guarantees that *some* process will complete an operation in a finite number of steps. The main difference is that wait-free prevents starvation. In this paper, we use the term *lock-free* as a synonymous with non-blocking. We have chosen to use lock-free synchronization instead of wait-free because the cost of wait-free is higher and the probability of starvation in an OS kernel is low.

Herlihy introduced a general methodology [4] to transform a sequential implementation of any data structure into a wait-free, concurrent one using Compare-and-Swap. His concurrent data structures, however, carry relatively high CPU and memory overhead even when there is no interference. For example, updating a limited-depth stack is implemented by copying the entire stack to a newly allocated block of memory, making the changes on the new version, and switching the pointer to the stack with a Compare-and-Swap. This cost is too high for a low-overhead OS kernel such as Synthesis, so we designed special-purpose objects for Synthesis V.1.

The first step in our approach is to try to squeeze the shared data into one or two words. If that succeeds, then we can use Compare-and-Swap on one or two words directly. If the shared data is larger than two words, then we try to encapsulate it in one of the lock-free objects we have designed (explained in Section 3): LIFO stacks, FIFO queues, general linked lists.

If we need to maintain consistency over data that does not fit into one of those lock-free objects, we use the general optimistic strategy to write a critical section. At the beginning we encode the system state in one local word. The encoding is such that each thread will produce a different word. Then we enter the critical section. Note that in optimistic synchronization the critical section should save enough information for retries. At the end of the critical section we check whether the system state encoding has changed. If negative then no other thread has entered the critical section and we exit safely. Otherwise interference happened and we must retry.

In addition to using only lock-free objects and optimistic critical sections, we also work hard to minimize the length of each critical section to decrease the probability of retries. Sometimes we can divide a critical section into two shorter ones by finding a consistent state in it. One way to produce a consistent state is to carefully shift some code between readers and writers. Another way to reach consistency is to encapsulate data in a temporary lock-free object known only to this critical section. Since the temporary object is free of external interference, it can help avoid problems. A third way to minimize critical sections is code isolation, where only specialized code handles the manipulation of data. An example of code isolation is the run-queue. Typically a run-queue is protected by semaphores or spin-locks in UNIX and Mach implementations. In Synthesis, only code residing in each element can change it, so we separate the run-queue traversal (done lock-free, safely and concurrently) from the queue element update (done locally).

When all else fails, it is possible to create a separate thread to act as a server that serializes the operations. Using lock-free queues to communicate with the server will assure consistency. Even though we occasionally use this escape mechanism as a first implementation, we have been able to find better solutions for every instance. There are no such servers in the current implementation of Synthesis V.1.

3 Lock-Free Quajects

The V.1 kernel is composed of *quajects*, chunks of code with data structures. Some quajects are instances of abstract data types, such as stacks, queues, and linked lists described in this section. Other quajects represent OS abstractions, such as threads, memory segments, and I/O devices described in subsequent sections.

```
Push(elem)
{
retry:
    old_SP = SP;
    new_SP = old_SP - 1;
    old_val = *new_SP;
    if(CAS2( old_SP,old_val, new_SP,elem, &SP,new_SP) == FAIL)
        goto retry;
}

Pop()
{
retry:
    old_SP = SP;
    new_SP = old_SP + 1;
    elem = *old_SP;
    if(CAS(old_SP,new_SP,&SP) == FAIL)
        goto retry;
    return elem;
}
```

Figure 1: Stack Push and Pop

3.1 LIFO Stacks

Stack Pop is implemented in almost the same way as a counter increment. We read the current value of the stack pointer into a private variable, de-reference it to get the top item on the stack, and increment the stack pointer using Compare-and-Swap to detect changes. Stack push is more complicated because we must make sure when we are writing the element on the stack, that we do not overwrite data that was pushed by another concurrent push operation. This requires a two-word Compare-and-Swap. We read the current stack pointer into a private variable, decrement it placing the result into another private variable, test that the stack pointer hasn't changed and store the new value of the stack pointer and also store the element on the top of stack. But to perform two stores with Compare-and-Swap we also have to perform two tests, so we have to read the old value above the top-of-stack into a third private variable to have something to test against.

3.2 FIFO Queues

In a previous paper [8] we described an array implementation of FIFO queues using lock-free synchronization (called optimistic synchronization in that paper). We summarize here the properties of these queues to make this paper self-contained. FIFO queues support two main operations, `Q_put` and `Q_get`. These queues, synthesized by the Synthesis kernel, callback the user of a queue on four conditions, `Q_full` and `Q_empty` to handle reaching the boundary conditions plus `Q_full-1` and `Q_empty+1` to handle leaving the boundary conditions. The queue elements may be one of three types: byte-at-a-time, longword-at-a-time, and byte-stream. They all support `Q_put` and `Q_get` operations on fixed-size data (byte or longword). In addition the byte-stream type also supports elements of arbitrary-sized blocks of data.

For efficiency, a programmer can choose from four kinds of queues: (1) single producer and single consumer, (2) single producer and multiple consumers, (3) multiple producers and sin-

```
Insert(elem)
{
retry:
    old_first = list_head;
    *elem = old_first
    if(CAS(old_head,elem,&list_head) == FAIL)
        goto retry;
}

Delete()
{
retry:
    old_first = list_head;
    if(old_first == NULL)
        return NULL;
    second = *old_first;
    if(CAS2(old_head,old_first, second,0, &list_head,old_first) == FAIL)
        goto retry;
    return old_first;
}
```

Figure 2: Linked-List Insert and Delete at Head

gle consumer, (4) multiple producers and multiple consumers. Each implements the minimum synchronization necessary for its intended use. For example, blocking queues are implemented by connecting the `Q_full` and `Q_full-1` callbacks to the corresponding thread's suspend and resume procedures. To give an idea of relative costs, the current implementation of multiple-producer, single-consumer has a normal execution path length of 11 MC68030 instructions through `Q_put`. In the case where two threads are trying to write an item to a sufficiently empty queue, they will either both succeed (if they attempt to increment `Q_head` at different times), or one of them will succeed as the other fails. The thread that succeeds consumes 11 instructions. The failing thread goes once around the retry loop for a total of 20 instructions.

3.3 General Linked Lists

Figure 2 shows how we use Compare-and-Swap to perform linked-list insert and delete from the head of the list. Insert reads the address of the list's first element into a private variable (`old_first`), copies it into the link field of the new element to be inserted, and then uses Compare-and-Swap to atomically update the list's head pointer if it had not been changed since the initial read.[1] Insert and delete to the end of the list can be carried out in a similar manner, but maintaining a list-tail pointer.

Allowing delete operations on interior nodes of the list is much harder, because a node may be deleted and deallocated while another thread is traversing it. If a deleted node is the allocated and reused for some other purpose, its new pointer values may cause invalid memory references by the other thread still traversing it. Herlihy's solution [4] uses reference counts (Figure 3). Visiting a node uses a two-word Compare-and-Swap to load the pointer and increment the reference count. Leaving a node similarly decrements the reference count. A node is not actually freed until the reference count reaches zero.

To avoid the overhead of incrementing and decrementing reference counts during a visit, we

[1] The one-word Compare-and-Swap operation (CAS) and two-word CAS2 are defined in appendix Section A.1.

Operation	No Sync	Opt., 0 retry	Opt., 1 retry	Opt-0/No-sync
null procedure call in C	1.0	—	—	—
Increment counter	0.3	1.4	2.6	4.4
Linked-list Insert	0.4	1.4	2.7	3.2
Linked-list Delete	0.6	2.2	4.3	3.6
Circular-Queue Insert	1.5	2.9	5.6	1.9
Circular-Queue Delete	1.7	2.5	4.8	1.5
Stack Push	0.6	2.1	3.9	3.3
Stack Pop	0.7	1.5	2.9	2.2

Times in microseconds
68030 CPU, 25MHz, 1-wait-state main memory, cold cache

Table 1: Cost of Lock-Free Operations

restrict the delete operation to "safe" situations. A delete operation is "safe" if the deleted node's link pointers continue to be valid, i.e., pointing to nodes that eventually take it back to the main list where the Compare-and-Swap will detect the change and retry the operation. This happens if a node is deleted but not placed on the free list. Herlihy's reference count is a good example. Also, if we insert and delete at the head of the list, a two-word Compare-and-Swap can guarantee safety by simultaneously checking the previous node's pointer, which is always valid at the head. In the middle of the list, we can achieve the same effect by deleting a node only when the permanence of the previous node is guaranteed. We do this in two steps: (1) mark the nodes to be deleted and leave them in the list; (2) if the previous node is not marked for deletion, sit on it and delete the original node marked for deletion. Since step 2 may require going back the list an arbitrary number of nodes, usually we do the step 2 the next time we traverse the list to avoid the overhead of traversing the list just for deletion.

The main difficulty with linked list traversal is that nodes can disappear while visiting them. Herlihy's reference count is a general solution. In the Synthesis run-queues, there is only one thread visiting a TTE at any time. So we can simplify the implementation to use a binary marker instead of counters. We set the mark at the same time we enter the node using a two-word Compare-and-Swap. This is easier than incrementing a counter because we don't have to read the mark beforehand – it must be zero to allow entrance. Non-zero means that node is being visited, so we skip to the next one repeating the test. We omit the traversal code since the only difference from unsynchronized access is the Compare-and-Swap.

3.4 Lock-Free Synchronization Overhead

Table 1 shows the overhead measured for the lock-free objects we have described in this

```
VisitNextNode(current)
{
    nextp = & current->next;        // Point to current node's next node field
retry:
    next_node = *nextp;             // Point to next node
    if(next_node != NULL) {         // If not null...
        refp = & next_node->refcnt; // Point to next node's ref. count field
        old_ref = *refp;            // Get value of next node's ref. count
        new_ref = old_ref + 1;      // ... Increment
        if(CAS2( next_node,old_ref,  next_node,new_ref,  nextp,refp) == FAIL)
            goto retry;
    }

    return next_node;
}

ReleaseNode(current)
{
    refp = & current->refcnt;       // Point to current node's ref. count field
retry:
    old_ref = *refp;                // Get value of current node's ref. count
    new_ref = old_ref - 1;          // ... Decrement
    if(CAS(old_ref,new_ref,refp) == FAIL)
        goto retry;
    if(new_ref == 0) {
        Deallocate(current);
        return NULL;
    } else {
        return current;
    }
}
```

Figure 3: Linked List Traversal

section. The "No Sync" column shows the time taken to execute an implementation of the operation without synchronization. The "Opt., 0 retry" column shows the time taken by the lock-free implementation when there is no interference. The "Opt., 1 retry" column shows the time taken when interference causes the first attempt to retry, with success on the second attempt.[2] The numbers shown are for in-line assembly-code implementation and assume a pointer to the relevant data structure already in a machine register. The lock-free code measured is the same as that produced by the Synthesis kernel code generator.

4 Threads

We describe here how thread operations can be implemented so they are lock-free. Synthesis has a general-purpose kernel-threads system with performance an order-of-magnitude greater than other comparable kernel-threads systems such as Mach, and similar in performance to user-level threads [2] from University of Washington.

4.1 Scheduling and Dispatching

Each thread is described by a *thread table entry* (TTE). V.0 had a single run-queue with round-robin scheduling. Each thread has its own dispatcher and scheduler, synthesized and stored in its TTE, including the thread context save area and other thread-specific data. The dispatcher is divided into two halves, the switch-out routine, which saves a thread's context, and the switch-in routine, that load in the thread's context and installs its switch-out routine as the current clock interrupt handler. When a CPU quantum expires, the clock interrupt handler (the exiting thread's switch-out routine) stores the thread's context and branches into the next ready thread's switch-in routine.

V.0 had only a single run-queue that contained all the TTEs. A fine-grain scheduling mechanism based on software feedback [9] changes the CPU quantum of each thread according to its need. For example, if a thread's input queue is full, the software feedback increases its quantum. If the total run-queue length is reasonably short, and we can traverse the run-queue within a short time, a relatively large CPU quantum simulates a higher priority (in terms of useful CPU time accumulation rate). However, if the queue grows long, the response of the system could become slower.

In V.1, TTEs are organized into multiple run-queues for scheduling and dispatching. Priority is explicitly represented in these queues, each level a separate queue. The number of levels is a parameter that can be changed at system generation time. Currently we have 8 levels and the total CPU time given to the jobs at each level decreases exponentially. Each level allocates CPU to the threads at that level according to the previously described single-queue policy. The highest level, level 0, receives half of the total CPU time, with the other half allocated for all the remaining levels. In general, level $N+1$ receives half the CPU left over from level N. If there is insufficient demand at any level, the extra CPU is made available to the lower levels.

With multiple run queues, V.1 uses a global counter and a priority table that tells the dispatcher which level's queue is next. The priority table contains the scheduling policy described above, i.e., 0 every other entry, 1 every fourth entry, 2 every eighth entry, etc $(0, 1, 0, 2, 0, 1, 0, 3, \cdots)$. Using the counter to follow the priority table, the kernel dispatches a thread from level 0 at every second context-switch, from level 1 at every fourth context-switch, level 2 at every eighth, and so on. The dispatching is illustrated by Figure 4. The search is limited by the number of levels.

[2] This case is produced with a special version of the PGA program described in Appendix A.2, which generated an interfering memory reference between the initial read and the Compare-and-Swap. Otherwise the interference would be very difficult to produce and measure.

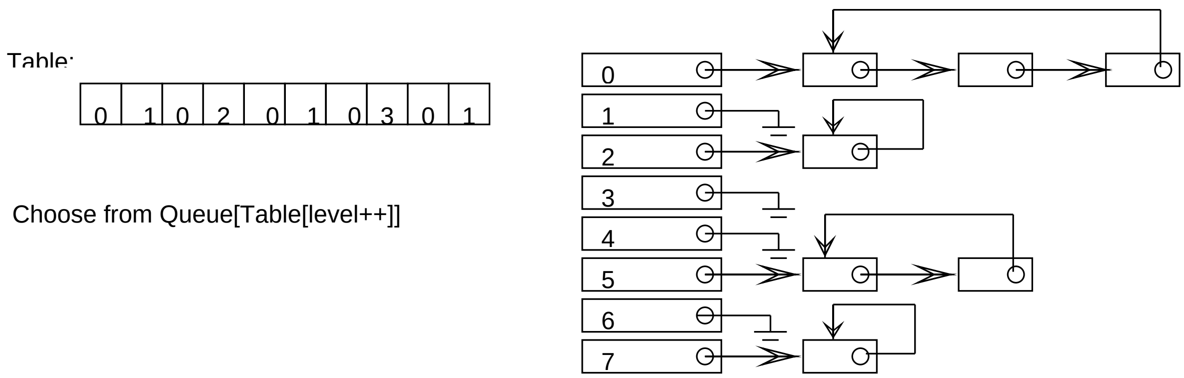

Figure 4: The Organization of Run-Queue

Currently, the CPU division between levels is static, i.e., the table content is fixed. It is easy to see that changing the entries in the priority table will change the proportion of CPU given to each level. We use second-order software feedback mechanisms to adapt the CPU distribution during high-load situations according to specified policies. This topic is beyond the scope of this paper.

When multiple CPUs attempt to dispatch threads from the run-queues, each active dispatcher (switch-out routine) acquires a new TTE by marking it with Compare-and-Swap. If successful, the dispatcher branches to the switch-in routine in the marked TTE. Otherwise, some other dispatcher has just acquired the attempted TTE, so this dispatcher moves on to try to mark the next TTE. The mark does not prevent other active dispatchers from accessing the rest of run-queues, just that they should avoid visiting this particular TTE.

Virtual memory support introduces some extra overhead for context switch. If we are switching between different address spaces, we need to (1) load the root of new page table and (2) flush the memory management unit's translation lookaside buffer (TLB, 22 entries on the 68030). Table 2 shows this cost.

4.2 Thread Operations

The overhead of create-thread operation has been reduced considerably, from 150 microseconds (in V.0 on 68020) down to 20 microseconds (in V.1 on 68030). The main cost of Synthesis V.0 thread-create was the large amount of state information that had to be initialized. TTEs contained pointers to routines that handled the various system calls and hardware interrupts. These included 16 system-call trap vectors, 21 program exception vectors, from 8 to 192 interrupt vectors depending on the hardware configuration, and 19 vectors for hardware failure detection. Almost all of the speedup in create thread was obtained through a copy-on-write optimization. Now, newly-created TTEs point to the same vector table of their creator and defer the creation of their own until they need to change the vector table.

Currently, there are only two operations that change a thread's vector table: opening and closing quajects. If a quaject is not to be shared, `open` and `close` test if the TTE is being shared, and if so they first make a copy of the TTE and then modify the new copy. Alternatively, threads may share the changes in the common vector table. For example, threads can now perform system calls such as `open file` and naturally share the resulting file descriptor with the other threads using the same vector table.

Figure 5 shows the thread state-transition diagram. We now explain how the other thread

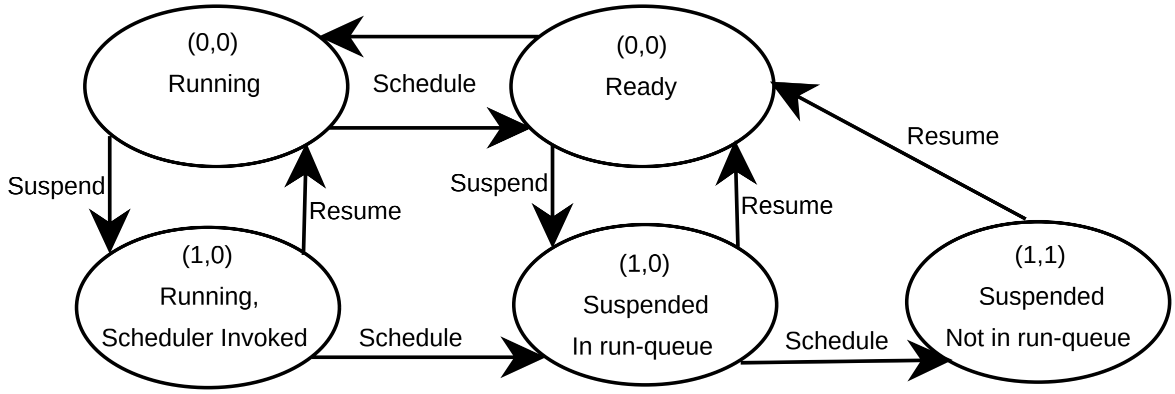

Figure 5: Thread State Transition Diagram

operations are made lock-free. The general strategy is the same. First, mark the intended operation on the TTE. Second, perform the operation. Third, check whether the situation has changed. If negative, the operation is done. If positive, we retry the operation. An important observation is that all state transitions and markings are done through Compare-and-Swap to avoid race conditions.

Suspend: The thread-suspend procedure sets the STOPME flag in the target thread's TTE indicating that it is to be stopped. If the target thread is currently running, a hardware interrupt is sent to its CPU forcing a context-switch. (We optimize the case when a thread is suspending itself by directly calling the scheduler instead.) Thread-suspend does not actually remove the thread from the run-queue.

When a scheduler encounters a thread with the STOPME flag set, it removes its TTE from the run-queue and sets the STOPPED flag to indicate that the thread has been stopped. This is done using the two-word compare-and-swap instruction to synchronize with other CPU's schedulers that may be operating on the adjacent queue elements. The mark on the TTE guarantees that only one CPU is visiting each TTE at any given time. This also makes the delete operation safe.

Resume: First, the STOPME and STOPPED flags are read and the STOPME flag is cleared to indicate that the thread is ready to run. If the previously-read STOPPED flag indicates that the thread had not yet been removed from the run-queue, we are done. Otherwise, we remove the TTE and insert the thread directly into the run queue. The main problem we have to avoid is the case of a neighboring TTE being deleted due to the thread being killed. To solve that problem, when a thread is killed, we mark its TTE as "killed", but do not remove it from the run-queue immediately. When a dispatcher realizes the next TTE is marked "killed" during a context switch, it can safely remove it.

Signal: Thread-signal is synchronized in a way that is similar to thread-resume. Each thread's TTE has a stack for pending signals which contains addresses of signal-handler procedures. Thread-signal uses a two-word Compare-and-Swap to push a new procedure address onto this stack. It then sets a signal-pending flag, which the scheduler tests. The scheduler removes procedures from the pending-signal stack, one at a time, and constructs procedure call frames on the thread's runtime stack to simulate the thread having called that procedure.

Step: Thread-step is intended for instruction-at-a-time debugging; calling it concurrently from more than one thread defeats its purpose. Therefore we do not give any particular meaning to concurrent calls of this function except to preserve the consistency of the kernel. In the current implementation, all calls after the first fail. We implement this using an advisory lock.

Type of context switch	Synthesis V.0	Synthesis V.1
Integer registers only	7	11
Floating-point	15	21
Integer, change address space	—	$13 + 1.6 * TLB\_fill$
Floating-point, change address space	—	$23 + 1.6 * TLB\_fill$

Table 2: Overhead of Scheduling and Context Switch

4.3 Thread Operation Overhead

Table 2 shows the overhead of scheduling and dispatching. The "Synthesis V.1" column shows the numbers for Synthesis V.1 on the Sony NeWS 1850 machine, a dual 68030 each at 25 Mhz. These numbers were measured on the Quamachine (see Section A.2) and calculated for the NeWS. For comparison, we also list the numbers from V.0 [8] under the column "Synthesis V.0". Those numbers are scaled by 16/25 to make up for the different clock rate of the machine (a SUN-3/260) on which they were obtained.

Context-switch has become a little slower because now we schedule from multiple run queues, and because there are some extra synchronization that was not necessary in the previous, single-CPU version. in addition, the floating-point number is slower because the Sony machine has the new 68882 floating-point coprocessor with more state information to save and restore.

Version V.1 of Synthesis supports virtual memory, while V.0 did not. The numbers in Table 2 is the scheduling and context-switch time including loading the memory management unit's translation table pointer and flushing the translation cache. Extra time is then used up to fill the translation cache. This is the "$+1.5 * TLB\_fill$" time. Depending on the thread's locality of reference, this can be as low as 4.8 microseconds for 3 pages (code, global data, and stack) to as high as 35 microseconds to fill the entire TLB cache.

Table 3 shows how the cost of other thread operations have been affected with the addition of multiprocessor support and virtual memory. Thread create is significantly faster because of the copy-on-write optimization. The other thread operations are somewhat slower because of the multiprocessor synchonization. Thread suspend, destroy, and signal have been split into two parts: the part done by the requestor and the part done by the dispatcher. The time for these are given in the form "XX + XX", the first number is the time taken by the requestor, the second number is the time taken by the dispatcer. Thread resume has two cases, the case where the thread had been stopped but the scheduler had not removed it from the run queue yet, shown by the first number, and the case where it was removed from the run queue and must be re-inserted, shown by the second number.

5 Virtual Memory

5.1 Memory Model and Interface

We use the term *address space* to refer to the total amount of memory a CPU can directly address. So an address space on the 68030 means 4GBytes that 32-bit registers can refer to. The 68030 page table has two protection bits per page, SUPERVISOR and READ-ONLY.

Thread Operation	Synthesis V.0	Synthesis V.1
Create	95	19.2
Destroy	7	2.2 + 6.1 in dispatcher
Suspend	5	2.2 + 3.7 in dispatcher
Resume	5	3.0 if in Q; 6.6 not in Q
Signal	5	4.6 + 4.4 in dispatcher
Step (no FP, no VM switch)	25	20

Table 3: Thread operations

The unit of protection in Synthesis is *quaspace*. An address space may contain many quaspaces, but each quaspace is wholly contained in a single address space. By unit of protection we mean that each quaspace has a uniform view of its address space, mapped by its own copy of the page table. Each quaject resides in one quaspace.

The unit of sharing in Synthesis is a *segment*, which is a chunk of contiguous pages. By unit of sharing we mean that whenever two quajects share memory, they must share entire segments, not parts of a segment. I.e., each quaspace may contain many segments, but each segment is part of a single quaspace. A segment may be mapped into several quaspaces or even address spaces, each one with its own protection setting.

We reserve part of each address space for the kernel (called kernel quaspace for reasons that will become clear later). The size of kernel quaspace is reconfigurable at system generation time. In the current configuration of V.1 we reserve the upper half of the address space (2GBytes). Kernel quaspace size can be reduced to 16MBytes without negative impact on system performance. We chose to make the kernel quaspace present in every address space for three reasons:

1. This avoids the switching between kernel and user spaces during kernel calls (Psyche [11] had this problem).

2. Kernel quaspace is so small (16 MB expected) and grows so slowly (as a function of number of processors and threads) that there is no need to allocate an entire address space.

3. The kernel quaspace is similar to a user quaspace in structure and implementation.

Kernel quaspace is the only quaspace with SUPERVISOR bit turned on. In other aspects it is just another quaspace. When a new address space is created, it starts out either empty or sharing everything with the creator's current address space. Segments may be created or simply added.

5.2 Real Memory Management

The real memory management component of Synthesis kernel keeps track of the real memory pages.

Allocate and Deallocate physical page: Free pages are linked together in a freelist. A page is allocated by deleting the first one from the freelist using the previously mentioned list operations. A page is freed by inserting it back into the head of the freelist, a linked list with insert and delete at the list head, therefore lock-free.

Allocate and Deallocate page table entries. These are done the same way as allocate and free physical pages, except we use two freelists because page map tables come in two sizes: 64 entries (256 bytes) for the leaf tables and 128 entries (512 bytes) for the upper level tables. If either freelist becomes empty, additional table space is obtained by allocating a physical page, splitting it into several chunks of the right size, and inserting them into the appropriate freelist.

5.3 Virtual Memory Management

Synthesis V.1 adopts a 3-level tree-structured page table, with 7-bit addresses at the first level, 7 bits at the second level, and 6 bits at the third level for a page size of 4KBytes. Each thread's TTE contains a pointer to its address space. By hardware constraint, each thread can only access one address space at any given time. As in other light-weight process models, many threads may share an address space. Each address space has an address space descriptor (ASD), which contains two things: (1) the address space's page table and (2) the linked list pointing to all the pager quajects contained in the address space.

Each segment is described by a segment descriptor, which contains three things: (1) the address range of the segment, (2) the pager that knows about this segment, and (3) a custom-generated active pager buffer. The pager buffer contains the procedures that convert page I/O requests from the kernel to the pager's (synchronous) read and write calls. It buffers data to match the system page size to the pager's optimal I/O granularity and it maintains the list the of real pages allocated to the pager.

The page table is built "on demand" as references are made. At a page fault, if the address does not correspond to any quaspace, no quaject is there and the thread is terminated with a memory access exception. If there is a quaject at that address, the quaject's pager is expected to supply the physical content of the page. The pager handles three cases:

1. The access to the address is prohibited. This is shown by the page table entry. In this case we send a memory access exception signal to the thread.
2. The page continuing the faulted address is in real memory buffer but missing from the page table. In this case it fills the page table and returns from the interrupt.
3. The page is not in real memory. The active buffer allocates a new page frame, calculates the page address on disk, and calls the pager to read in the page from disk. Then do the item 2.

Add mapping to address space. This function installs new virtual-to-physical mappings into an address space's page table. It begins by traversing the page tables to find the leaf table where the new mapping is to be placed. A null table pointer at any point in the traversal means that that part of the mapping tree is not yet allocated. When that happens, we allocate a new table, mark all its entries invalid, and store the its address into the previous level's table pointer. Compare-and-Swap ensures that no other thread has already allocated a new table for that entry. If one had, we simply free our table. After the leaf table has been located, the virtual to physical mapping is stored using Compare-and-Swap, testing against null. If this fails, it means that a mapping already exists, so we test that it is identical to the one we're adding. Different mappings imply a programming error, since a logical address can map to at most one physical address, so we destroy the address space, which then terminates all threads within it.

Mark page invalid. Similar to the above, except that an invalid descriptor is stored in the leaf table, replacing the previous mapping.

Operation	Time (in microseconds)
Allocate page (pre-zeroed)	2.4
Allocate page (needs zeroing)	152
Allocate page (none free; replace)	152 + time to replace
Zero-fill a page (4 Kbytes)	148
Free page	1.6
Memory Access Exception	13.6

Table 4: Low-level Memory Operation Overhead

Finding custom pager. This procedure is called from the memory-fault handler to discover which pager will supply the missing page. It begins by traversing the page table to obtain the page descriptor for the faulted address. The page descriptor usually contains a hint pointing to the pager. We check the hint, and if it matches, we are done. Otherwise, we traverse the address space's directory – a linked list representing a set of pager pointers and the virtual-address range that they map to.

Page fault. Page faults are synchronous, which simplifies synchronization for handling them, since each thread handles its own page faults. First determine the fault address and the type of fault (e.g., invalid translation, write to read-only page) by examining the memory management unit status register and the CPU exception stack frame. Then find which quaject corresponds with the faulted address using the previous procedure. Next, call the quaject's custom-generated VM interface procedure, passing it the memory offset and type of fault. This code invokes procedures within the quaject to perform page I/O operations, and when finished, calls the previously-described "Add mapping" function and returns, causing the faulted access to be retried.

No other synchronization is necessary because each thread executes its own fault handler and the quaject's pager performs its own paging I/O. I/O waits invoke thread-suspend as usual, preventing the thread from executing until its page arrives. Deadlock is not possible since a blocked thread cannot page-fault.

5.4 Memory Management Overhead

Table 4 contains the numbers of basic memory management operations calculated for the Sony machine. The page zeroing is implemented with the MC68030 multiple register save instruction (13 words at a time), which is faster than bcopy.

6 Input/Output

We describe two examples of I/O subsystems in V.1 to illustrate the lock-free synchronization used in I/O. Although the primary reason for efficiency in I/O processing is kernel code synthesis [10].

Quaject	Create (microseconds)	Write (microseconds)
TTY-Cooker	27	2.3 + 2.1 per char
VT-100 terminal emulator	532	14.2 + 1.3 per char
Text window	71	23.9 + 27.7 per char

Table 5: Selected Window System operations

6.1 Terminal and Display

A terminal window is a pipeline composed of primarily three quajects: a TTY-Cooker, a VT-100 Terminal Emulator, and a Text-Window. Each quaject has a fixed cost of invocation and a per-character cost that varies as a function of the character being processed. These costs are summarized in Table 5. The invocation cost occurs each time the quaject is called, independent of the number of chars, and is given by the column of the same name. The per-character costs are the average cost summed over the characters in `/etc/termcap`.

The sum of the total time for the three quajects exceeds the wall-clock time actually observed. This unexpected result happens because Synthesis kernel can optimize the data flow, resulting in fewer calls and less actual work than from a straight concatenation of the three quajects. For example, in a fast window system like Synthesis, many characters may be scrolled off the screen between the consecutive vertical scans of the monitor. Therefore the window manager bypasses the drawing of those characters by sampling the content of the virtual VT100 screen 60 times a second and drawing the parts of the screen that have changed. The VT-100 emulator still has to parse each character and maintain the virtual screen. But this is much faster than updating the framebuffer because we store the characters as ascii codes rather than as bitmaps and use pointers to speed scrolling. The TTY-Cooker is a finite-state machine that converts editing commands such as escape characters and newline.

The terminal driver receives the interrupts and passes the characters to the TTY-Cooker, which calls the VT-100 emulator through synchronous co-routine calls. The VT-100 emulator maintains an internal buffer (the virtual screen) that is shared with the Text-Window. In a window system, there is no problem with Text-Window sharing the virtual screen with the VT-100 emulator, since it is OK to show partial updates of a window. Synthesis guarantees that a screen update is reflected in a window within two vertical scans after the application's `putchar` is finished.

6.2 File System

The file system is also a pipeline of several quajects. At the bottom is the disk driver, which sends data to File-Mapper. File-Mapper translates a logical array into disk cylinders and sectors to supply the disk driver with appropriate hardware commands. File-Mapper stores data in a File-Buffer, which implements UNIX-style file I/O at the same abstract level of UNIX read, write and seek system calls. At the top is the File-System quaject, which synthesized the actual kernel call code in an `open file`. Synchronous communications (between driver and File-Mapper) are through co-routines and asynchronous communications (between File-Buffer and File-Mapper or File-System) through lock-free queues.

Although the hard disk driver has not been finished at the time of this writing, we have a ramdisk driver. All the virtual memory management development and debugging, including the measurements, are done by connecting the pagers to a file system using the ramdisk driver. The ramdisk is independent of architecture (works on both the Sony machine and Quamachine) and fully debugged.

7 Related Work

Synthesis V.1 kernel differs from production multiprocessor OS kernels such as Mach [3], Topaz [13], and Psyche [11] in being lock-free. All the synchronization problems in a shared-memory multiprocessor are solved using lock-free synchronization methods based on Compare-and-Swap. A small number of lock-free objects such as packed flags, stacks, FIFO queues and linked lists connect all of Synthesis kernel components. To the best of our knowledge, this has not been attempted even in experimental multiprocessor OS kernels such as Elmwood [7].

From the lock-free data structure point of view, our work is most close related to that of NYU Ultracomputer by Hummel [6, 14] and IBM RP3 [5]. Even though the algorithms are not exactly the same, in particular due to our use of double-word Compare-and-Swap, the objectives are the same – applying lock-free synchronization to increase concurrency and decrease overhead. They have described practical and useful solutions for particular concurrent data structures in the context of scheduling. Another paper describing concurrent access to queues is by Stone [12], who used Compare-and-Double-Swap involving two double words. In contrast, Synthesis use of lock-free synchronization is much more general since we have applied these techniques to implement an entire OS kernel.

Anderson et al. [2] have argued that kernel implementation of threads are necessarily more expensive than user-level thread management systems. Our implementation shows that with kernel code synthesis, kernel-level threads may be as efficient as user-level threads even in a multiprocessor kernel with full funcionality.

8 Conclusion

We have used only lock-free synchronization techniques used in the implementation of Synthesis V.1 mutiprocessor kernel on a dual-68030 Sony NeWS workstation. This is in contrast to other implementations of multiprocessor kernels that use interlocking. Locking synchronization methods such as disabling interrupts, spin-locking, and waiting semaphores have many problems. Semaphores carry high management overhead and spin-locks may waste significant amount of CPU. (A typical argument for spin-locks is that the processor would be idle otherwise. This may not apply for synchronization inside the kernel.) A completely lock-free implementation of a multiprocessor kernel demonstrate that we can reduce synchronization overhead, increase concurrency, avoid deadlocks, and eliminate priority inversion.

We achieved this completely lock-free implementation with a careful kernel design. First we reduced the kind of data structures used in the kernel to a few simple abstract data types such as LIFO stacks, FIFO queues, and linked lists. Then, we restricted the uses of these abstract data types to a small number of safe interactions. Finally we implemented efficient special-purpose instances of these abstract data types using single word and double word Compare-and-Swap. The V.1 kernel is a fully functional kernel supporting threads, virtual memory, and I/O devices such as window systems and file systems. The measured numbers show the very high efficiency of the implementation, competitive with user-level thread management systems.

We learned two lessons from this experience. First, a lock-free implementation is a viable and desirable alternative to the development of shared-memory multiprocessor kernels. Up to now, the usual strategy is to evolve a single-processor OS kernel to a multiprocessor kernel by surrounding critical sections with locks carries some performance penalty and potentially limits the system concurrency. One way to alleviate the lock overhead is to provide hardware lock support,[3] but this does not alleviate the concurrency bottleneck. Second, single and double word Compare-and-Swap are important for lock-free shared-memory multiprocessor OS kernels. RISC architectures that do not support these instructions will force the OS implementors to use locks, since efficient emulations of Compare-and-Swap by weaker operations in Herlihy's hierarchy (e.g., Swap in SPARC) are an open research problem.

A Architectural Support

A.1 Definition of Compare-and-Swap

Single word Compare-and-Swap (CAS) and double word Compare-and-Swap (CAS2) are defined here. These are particular cases of the Read-Modify-Write family of operations. Typical hardware implementations of these operations (in machine instructions) lock the memory bus or some other unique hardware resource to guarantee memory access atomicity for the duration of the instruction.

```
CAS(compare,update,mem_addr)
{
    if(*mem_addr == compare) {
        *mem_addr = update;
        return SUCCEED;
    } else
        return FAIL;
}

CAS2(compare1,compare2,update2,update2,mem_addr1,mem_addr2)
{
    if(*mem_addr1 == compare1 && *mem_addr2 == compare2) {
        *mem_addr1 = update1;
        *mem_addr2 = update2;
        return SUCCEED;
    } else
        return FAIL;
}
```

A.2 Hardware Measurement Tools

Although Synthesis V.1 has been implemented and debugged on the Sony NeWS workstation, the measurements described in this paper have been made on the Quamachine with special hardware support. The numbers reported in Tables 1, 3, and 2, for example, are calculated from a direct measurement of the number of machine cycles it took to run the respective programs. The main advantage of counting machine cycles directly is the avoidance of spurious code when doing the measurements in software. Another reason is the relative independence of these numbers with respect to architectural differences.

The Quamachine has a MC68030 CPU (converted from 68020 since the last SOSP [8]), 2.5 MB no-wait state main memory, 390 MB hard disk, $3\frac{1}{2}$ inch floppy drive. The Quamachine is designed and instrumented to aid systems research. The basis of such support is a Programmable

[3] Stratus and SGI are reported to provide inexpensive hardware support for locks.

Gate Array (PGA) on board. The 128-pin PGA is a finite-state machine with several functions. First, the PGA is responsible for generating all the I/O interrupts for the CPU. Each I/O device connects its interrupt line to a PGA pin, and the PGA translates it to the appropriate 68030 interrupt format for that device. This substitute for an "interrupt controller" chip. Second, the PGA is capable of performing DMA on request from devices without DMA capability. Third, the PGA arbitrates memory bus accesses between the CPU, the disk controller DMA, and itself. Fourth, the PGA monitors the memory bus to map certain memory locations to I/O requests, including the system clock. Since the PGA is programmable, old functions may be enhanced or new functions may be introduced as necessary or convenient.

To measure the number of cycles of a program fragment, we use a stopwatch in the PGA. Since the PGA generates all I/O interrupts, it also implements the system clock. A 50 MHz crystal generates a pulse every 20 nanoseconds, which is fed into the PGA to increment a 64-bit counter. The time-of-day clock is a register holding a copy of this counter. The time-of-day clock is mapped to a privileged memory location, monitored by the PGA. When the "clock address" is referenced, time-of-day register is updated. Similarly, we have two additional 64-bit memory-mapped registers capable of storing the current time (content of the crystal impulse counter), called start-time clock and stop-time clock for convenience.

When we start a measurement, we store the starting timestamp in the start-time clock and we store the finishing timestamp in the stop-time clock. Subtracting one from the other we have an interval timer of 20 nanosecond resolution, with each tick corresponding to one machine cycle.

All the measurements were taken with the caches turned off. We turned off the on-chip cache because kernel calls typically are not expected to remain in cache, assuming that the user threads actually do some useful work. The lack of an on-board cache in the Quamachine also means that we are not considering the effect of on-board cache in the Sony machine. This makes our numbers an overestimate of overhead, which is on the safe side. In any case, due to the preference of using registers instead of memory accesses in the V.1 kernel, turning on the Sony on-board cache seems to improve the kernel performance by only a few percent. The main memory cycle runs at 100 nanosecond with 1 wait state on both the Sony machine and the Quamachine.

The measurements are done by loading the V.1 kernel code into the Quamachine, running it, and taking the timestamps to find out what is the number of cycles it takes to execute. If there are several executions paths we run the program a number of times to check the variance.

References

[1] T.E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[2] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levey. Scheduler activations: Effective kernel support for the user-level management of parallelism. Technical Report 90-04-02, Department of Computer Science, University of Washington, April 1990.

[3] D.L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, 23(5):35–43, May 1990.

[4] P.M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1), January 1991. To appear.

[5] S. F. Hummel and E Schonberg. Low-overhead scheduling of nested parallelism. Technical Report RC 16424, T.J. Watson Research Center, IBM Research Division, 1990.

[6] Susan Flynn Hummel. *SMARTS – Shared-memory Multiprocessor Ada Run-Time Supervisor.* PhD thesis, Department of Computer Science, New York University, 1988.

[7] T.J. LeBlanc, J.M. Mellor-Crummey, N.M. Gafter, L.A. Crowl, and P.C. Dibble. The elmwood multiprocessor operating system. *Software – Practice and Experience*, 19(11):1039–1055, November 1989.

[8] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 191–201, Arizona, December 1989.

[9] H. Massalin and C. Pu. Fine-grain adaptive scheduling using feedback. *Computing Systems*, 3(1):139–173, Winter 1990. Special Issue on selected papers from the Workshop on Experiences in Building Distributed Systems, Florida, October 1989.

[10] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[11] M.L. Scott, T.J. LeBlanc, B.D. Marsh, T.G. Becker, C. Dubnicki, E.P. Markatos, and N.G. Smithline. Implementation issues for the Psyche multiprocessor operating system. *Computing Systems*, 3(1):101–138, Winter 1990.

[12] J.M. Stone. Managing a shared FIFO queue with Compare-and-Swap. In *Proceedings of the 1990 Supercomputing Conference.* ACM, 1990.

[13] C.P. Thacker, L.C. Stewart, and E.H. Satterthwaite, Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, C-37(8):909–920, August 1988.

[14] James M. Wilson. *Operating System Data Structures for Shared-Memory MIMD Machines with Fetch-and-Add.* PhD thesis, Department of Computer Science, New York University, 1988.