AT&T Bell Laboratories Technical Journal
Vol. 63, No. 8, October 1984
Printed in U.S.A.

The UNIX *System:*

Multiprocessor *UNIX* Operating Systems

By M. J. BACH* and S. J. BUROFF*

(Manuscript received August 22, 1983)

This paper describes the problems posed by running the *UNIX*™ operating system on multiprocessors, as well as some solutions. The resulting systems function like their single-processor counterparts but yield 70 percent better throughput for two-processor configurations. Closely coupled multiprocessor *UNIX* systems currently run on IBM and AT&T Technologies hardware, but the implementation described in this paper ports to other architectures as well, and the design is not limited to two-processor configurations.

I. INTRODUCTION

The *UNIX* operating system has been ported to many processors, but only recently has it been ported to multiprocessor (MP) configurations. Porting to multiprocessor configurations further extends the range of machines on which *UNIX* systems are available and further supports the concept of a portable operating system. It also extends the range of *UNIX* system applications and provides an important extension to the upward migration for projects that begin using the *UNIX* system on a minicomputer and then outgrow that machine's capabilities. *UNIX* systems currently run in multiprocessing environments on IBM/370 architecture machines, and AT&T 3B20A and 3B5

* AT&T Bell Laboratories.

computers, but the ensuing discussion applies equally well to other machine architectures that support multiprocessor environments.

The *UNIX* systems that were devised for the various multiprocessors provide complete transparency to user programmers. That is, all system calls and commands operate the same way on the multiprocessor systems as they do on single-processor *UNIX* systems. Existing C programs can be moved from single-processor systems to their multiprocessor versions without recompilation, except for system-dependent code (e.g., the command to determine process status, `ps`). The terminal interface, file system format, process hierarchy, and all other user-visible aspects of the operating system appear identical to those on a single-processor *UNIX* system.

For the purposes of this paper, a multiprocessor hardware configuration is one that has two or more processors that share a common memory, corresponding to what is commonly called a tightly coupled system. It is distinguished from a loosely coupled system, where each processor has private memory, and where the processors communicate using a networking facility instead of shared memory.

Multiprocessor hardware configurations can be further classified by their symmetry with respect to input/output (I/O). In an Associated Processor (AP) configuration, only one processor is capable of doing I/O operations, while in a true multiprocessor configuration either processor can do I/O. Except as specified, the ensuing discussion applies to MP and AP configurations.

Another multiprocessor *UNIX* system[1] permits only one processor, the master, to execute the kernel of the operating system, avoiding the system data corruption problems described in Section 3.1. That system has modified the algorithm for scheduling processes to recognize the existence of more than one processor, and it schedules only user-level processes to the processor not allowed to execute kernel code, the slave. When a process executing on the slave processor does a system call, the operating system recognizes that the system call is originating on the slave processor, suspends the process, and reschedules it for the master processor. Since benchmark programs show that *UNIX* systems typically spend between 40 and 50 percent of their time executing operating system code, restricting one processor from executing kernel code prevents the system from achieving the full performance potential of the hardware except for specific workloads. The multiprocessor *UNIX* systems described in this paper permit all processors to execute kernel code simultaneously, yielding maximum efficiency from the hardware configuration.

This paper begins by describing the motivating factors for running the *UNIX* system on a multiprocessor, and continues by describing the special issues posed by multiprocessor configurations. The use of

semaphores to solve the multiprocessing issues is described in some detail, as is the special consideration given to device drivers. Concluding sections describe machine-specific issues and system performance. A basic knowledge of *UNIX* system internals is assumed.

II. MOTIVATION

UNIX systems are commonly used for software development, where programmers working on a project must communicate and share data with each other. But many software development projects, although they start out small, later outgrow their original computing capacity, so that a single computer no longer adequately supports all users.

When a project exceeds its machine capabilities, it can either acquire more machines and try to share the load between them or it can move up to a larger machine. But getting more machines to share the work load has several problems:

1. Communication of data across machines incurs high networking overhead.
2. The network is seldom transparent to the user; that is, users must understand the machine/project structure.
3. Data are frequently replicated across machines to reduce flow through the network, but replicated data may be inconsistent because of concurrent update problems across different machines.

On the other hand, moving a project to larger machines, sometimes of a different vendor, is frequently expensive in terms of hardware costs, data migration, and user productivity.

A multiprocessor capability allows a smooth growth path for projects that can start small with a single processor and, as their computing requirements expand, can add more processors to form a larger, more powerful system. Such growth is usually less expensive and less disruptive to end users than acquiring a new and larger machine.

Another advantage of a multiprocessor system is that it is potentially more robust. If a hardware failure makes one processor inoperable, the system can potentially recover from the problem. The users would not have to take any special action and would not notice any difference in system services except reduced performance. Diagnosing and fixing such problems on a multiprocessor *UNIX* system while the system is active is still an open problem, so the systems described here require a system reboot to restore operation. However, they execute in single processor mode so that failure of one processor does not prohibit booting and running the system on the other processors.

III. SYSTEM CHANGES

3.1 The problem of multiprocessors

The *UNIX* system was originally developed to run on a single

processor, and the code assumes that the kernel is never preempted except for processing of interrupts. Hence, kernel data structures do not need to be protected unless referenced by an interrupt routine, and if so, the data can be protected by locking out interrupts. This is normally done by raising the processor priority level high enough to prevent the type of interrupt from occurring.

For example, consider the code fragments taken from the functions `getc` and `putc` in Fig. 1, functions usually used for manipulating characters and queues for terminal drivers. Such characters are queued onto `cblocks`, and `cblocks` are chained together to form `clists`. The function `getc` removes a character from a `clist`, or, more properly, from the first `cblock` of the `clist`. If the `cblock` contains no more characters, the `cblock` is attached to the beginning of a free list of `cblocks`, and the `clist` is adjusted accordingly. The function `putc` places a character onto a `clist`, or, more properly, onto the last `cblock` of the `clist`. If that `cblock` contains no space for new characters, a new `cblock` is removed from the free list of `cblocks`, and the `clist` is adjusted accordingly.

The code fragments in Fig. 1 focus on placing and removing `cblocks` from the free list. Suppose a process executes statement 1 of `getc` but receives an interrupt before it executes statement 2. If the interrupt handler executes `putc`, it will remove the first `cblock` from the free list. When the process resumes control after the interrupt, it executes statement 2, making the returned `cblock` the free list header of `cblocks`. Unfortunately, the `cblock` in `getc` points to the `cblock`

```
getc(p)
struct clist *p;
{
        struct cblock *cp;
        .
        sp16();
        .
        cp->c_next = cfreelist.c_next;    /* 1 */
        cfreelist.c_next = cp;            /* 2 */
        .
        sp10();
        .
}


putc(c,p)
struct clist *p;
{
        struct cblock *cp;
        .
        sp16();
        .
        cp = cfreelist.c_next;
        cfreelist.c_next = cp->c_next;
        cp->c_next = NULL;
        .
        sp10();
        .
}
```

Fig. 1—Raising processor execution level for single processors.

just removed by `putc`, which severed its previous connection to the free list. The result is that the free list contains only one free `cblock` and one or more busy `cblocks`, and the remaining free `cblocks` are inaccessible.

UNIX systems traditionally avoid such problems by raising the processor execution level to prevent interrupts. In Fig. 1 the function `spl6` raises the processor execution level to six (presumably a level high enough to prevent interrupts whose handlers call `putc`), and the function `spl0` lowers it to zero, allowing all interrupts. Since no interrupts can occur between the calls to `spl6` and `spl0` in Fig. 1, the free list cannot be corrupted. Since processes in the kernel cannot be preempted unless they voluntarily relinquish use of the processor, raising the processor execution level to prevent interrupts protects all system data structures.

In the multiprocessor systems described in this paper, however, raising the processor execution level does not prevent corruption of system data structures, as all processors can simultaneously execute kernel code. In the example above, one processor could execute `getc`, but its `spl` does not necessarily prevent interrupts from occurring on the other processor, and hence the other processor could execute `putc` with catastrophic results. Similar corruption could occur without interrupts: processors could simultaneously write to terminals, execute `putc`, and remove the identical `cblock` from the free list with catastrophic results. Therefore, kernel code that references common data in multiprocessor systems must protect the data from access by other processors. The mechanism chosen to do this was based on Dijkstra's semaphores.[2-4] Although the use of semaphores is not new to multiprocessor *UNIX* systems, their use here is more extensive and system throughput is much higher than reported elsewhere.

3.2 Semaphores

3.2.1 Definition

A semaphore* is an integer-valued data structure on which the following restricted set of operations can be performed.

`init` Initialize the semaphore to an integer value.

`psema` Decrement the value of the semaphore. If the resulting value is less than zero, then suspend the executing process and place it on a linked list of processes sleeping on the semaphore. When awakened, the process priority is set to

* The semaphores being described here are a strictly internal mechanism and have nothing to do with the user interprocess communication facility of the same name that is described in Ref. 6.

the value supplied as one of the parameters to psema. If signals are pending against an awakened process, the value of the priority parameter determines whether they are deferred or caught.

vsema — Increment the value of the semaphore. If the resulting value is less than or equal to zero, then awaken a process that suspended itself doing a psema on the semaphore.

cpsema — If the value of the semaphore is greater than zero, then decrement it and return true. Otherwise, leave the semaphore unmodified and return false.

Semaphore operations are atomic. That is, if two or more processes try to do operations on the same semaphore, one completes the entire operation before the others begin.

3.2.2 Uses of semaphores

To protect a particular resource such as a table or linked list, a semaphore is associated with that resource and typically initialized to one when the system is booted. When a process wants to gain exclusive use of the resource, it does a psema on the semaphore, decrementing the semaphore value to zero (assuming it was one) but allowing the process to proceed. The process now has exclusive use of the resource. If other processes attempt to gain control of the resource, their psemas will decrement the semaphore value and suspend process execution. If the value of a semaphore is negative, then its absolute value is equal to the number of processes that are suspended waiting for that resource. When the process that has control of the resource is done with it, it does a vsema on the semaphore, releasing the semaphore and awakening a suspended process, if any. The awakened process is now eligible for scheduling when a processor becomes available and when no higher priority processes exist. When scheduled, the awakened process returns from the psema call without knowing that it was temporarily suspended, and when it finishes with the resource, it should do a vsema to release the semaphore and to awaken the next waiting process, if any.

A semaphore that is used to await an event is initialized to zero. Processes awaiting the event do a psema to suspend themselves until the event occurs, and processes recognizing the event do a vsema to awaken sleeping processes. A semaphore that is used to count the number of resources in the system is initialized to the appropriate number. When the resource is allocated, the psema decrements the semaphore value, and when the resource is freed, the vsema increments the semaphore value, so that it always conforms to the number of available resources. If the number of available resources drops to zero,

processes will sleep in the `psema` until another process releases a resource and does a `vsema`.

The `cpsema` operation is used to lock a resource only if it is immediately available, and other action besides sleeping is taken if the semaphore is unavailable. This is used in deadlock prevention and will be explained in Section 3.2.3.

Single processor *UNIX* systems use the `sleep` and `wakeup` mechanisms for process synchronization to voluntarily suspend and resume execution waiting for an event to occur. When a single processor system does a `wakeup` call on a resource, all processes sleeping for that resource are awakened. Often the resource must be used exclusively, so all but one of the awakened processes will test the resource, find it busy, and again go to sleep. In multiprocessor systems on the other hand, it is undesirable to awaken all sleeping processes because all such processes could not assume exclusive access to system structures. So a `vsema` only awakens a single process that will in turn awaken another sleeping process. A process that executes a `psema` knows that it has control of the resource and will not fall asleep again waiting for the resource to become ready.

The kernel of the multiprocessor systems has been modified to account for the change in semantics of sleeping. Calls to the `psema` and `vsema` functions replace calls to the old `sleep` and `wakeup` functions, as there is one set of process synchronization primitives (semaphores) instead of two.

3.2.3 Coding with semaphores

A serious problem in the use of semaphores is process deadlock. Figure 2 gives an example of deadlock where two processes, A and B, execute the shown code sequences.

At time T1, process A has locked semaphore `sema1` and process B has locked semaphore `sema2`. Process A now attempts to lock semaphore `sema2` and will be suspended because process B has control of the semaphore. Process B attempts to lock semaphore `sema1` but will be suspended because process A has control of it. Both processes will

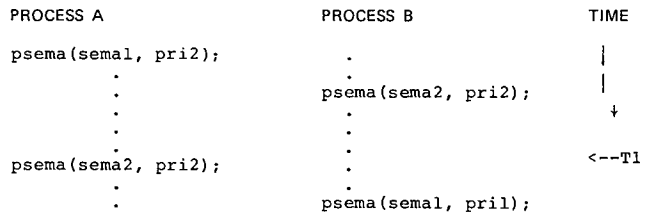

Fig. 2—Example of semaphore deadlock.

be suspended indefinitely because each is waiting for a resource that the other one has.

To avoid deadlocks, an ordering is imposed on the various resources in the system. All processes that simultaneously lock more than one resource do so in the prescribed order to guarantee that no deadlock can occur. More sophisticated schemes for deadlock detection and resolution would complicate the system code and slow down performance. Occasionally it is still necessary for a process to lock its semaphores in an order different from the prescribed order. For example, the system usually locks `inodes` before `text` slots since the `exec` system call first accesses the file before it determines whether or not to allocate a `text` slot. But the algorithm for cleaning swap space of unused program text first searches the `text` table and only sometimes needs to access and hence lock the `inode`. In such cases the process must use a `cpsema` to lock the second semaphore.

If the `cpsema` fails, then the process must take some other action to avoid the deadlock, usually releasing the semaphore it already holds and awaiting an event before attempting to execute the code again. Figure 3 contains code that corrects the potential deadlock of Fig. 2.

3.2.4 Semaphores in interrupt routines

Interrupt handlers usually share kernel data structures with higher-level kernel routines such as the `getc` and `putc` routines for terminal drivers of Section 3.1, so semaphore protection is required at the interrupt handler level as well as the rest of the kernel. It is preferable not to sleep in an interrupt routine for two reasons. First, it is desirable to service the interrupt as quickly as possible. Second, the process that would be suspended is often not related to the interrupt being processed. So, interrupt handlers use `cpsemas` instead of `psemas` and take other action if the semaphore is locked elsewhere. Section 3.5 gives more detail on driver interrupt handlers.

```
PROCESS A                PROCESS B                                  TIME

psema(sema1, pri1);        .                                         |
  .                      loop:
  .                        .
  .                      psema(sema2, pri2);                         |
  .                        .                                         ↓
  .                        .
  .                        .
psema(sema2, pri2);        .                                       <--T1
  .                        .
  .                      if (! cpsema(sema1)){
  .                              vsema(sema2);
  .                              /*other corrective action*/
  .                              .
  .                              goto loop;
  .                      }
```

Fig. 3—Example of deadlock avoidance.

3.2.5 Semaphores and performance

The use of semaphores must be carefully chosen to balance frequency of semaphore operations versus the "granularity" of semaphore protection, that is, how much data are protected by a single semaphore. If a semaphore locks a large set of resources such as the entire buffer pool, or if it is held for a long time, then many other processes may be suspended while waiting for the semaphore to unlock, delaying process flow through the system and resulting in excessive context switching. Contention for a semaphore can be measured by examining the mean number of processes sleeping on the semaphore and by examining the degree of contention for the semaphore, that is, the ratio of how frequently processes were denied access to the semaphore to how frequently they were attempted. If either of the above numbers is much higher than for other semaphores in the system, then semaphore usage in the system is unbalanced and new semaphores should be encoded to reduce semaphore contention.

Semaphore contention may be reduced by replacing a single semaphore with a set of semaphores. For example, suppose that there is a linked list of resources that must be searched, and items must be added to or deleted from the list. The list could be locked by a single semaphore, but if the list is large and frequently searched, processes may contend for the semaphore, and the semaphore could prove to be a system bottleneck. If so, performance can be improved by replacing the single linked list with a set of hash buckets, each heading a linked list containing those elements from the original list that hash to the same value. Instead of having one lock for the entire list, each hash bucket can have a separate lock spreading the original load over a set of semaphores and reducing the contention for each one. The buffer pool for example, contains one semaphore for each hashed (by device and block number) queue of buffers, one semaphore for each buffer, and one semaphore for the free list of buffers. Although the semaphore for the free list has one of the highest contention rates in the system, system throughput is much better than if there were only one semaphore for the entire buffer pool. Unfortunately there is no satisfactory way to divide the free list into separate lists with separate semaphores that does not adversely affect performance of the buffer algorithm.

Another issue in semaphore performance is whether a `psema` or a `cpsema` should be used to lock the semaphore; that is, if the semaphore is locked, whether the process should sleep until the semaphore becomes free or whether the process should execute a tight loop, attempting to lock the semaphore until it finally succeeds (see Fig. 4).

The issue is decided on a case by case analysis of the semaphores, comparing the average amount of time the semaphore is locked to the

```
psema(sema,pri);          while(!cpsema(sema))
                                  ;
```

Fig. 4—Sleep lock and spin lock.

time it takes to do a context switch. The results depend strongly on CPU performance characteristics.

3.2.6 Semaphore debugging

In spite of the best attempts at following ordering rules, deadlocks occur in multiprocessor systems, especially in early development stages. Deadlocks can be difficult to find because by the time the symptom appears (a stopped system), the cause of the problem has long since passed. To find these problems more easily, the system logs all semaphore operations. The log is a circular buffer where entries for each semaphore operation contain the type of operation performed, the text address where the operation was performed, the address of the semaphore, the process number, the semaphore value, and other useful information. The semaphore log gives a useful trace of processes as they execute kernel routines. Logging may be disabled when compiling the system or, to a lesser extent, while the system is executing to improve system performance.

In addition to the semaphore log, an extra field in each semaphore contains the process number of the last process that gained control of the semaphore. The semaphore log and the process number field in the semaphore structure are useful in diagnosing bugs in the multiprocessor system that never occur in a single processor system.

3.3 Example

Consider the code in Fig. 5 for the `xumount` function, called when unmounting device `dev`, that frees `text` slots belonging to the device. Although unmounting a device and calling `xumount` is a rare event in

```
xumount(dev)
        register dev_t dev;
{
        register struct inode *ip;
        register struct text *xp;
        register count = 0;

        for (xp = &text[0]; xp < (struct text *)v.ve_text; xp++) {
                if ((ip = xp->x_iptr) == NULL) /* not in use */
                        continue;
                if (dev != NODEV && dev != ip->i_dev) /* on device dev*/
                        continue;
                if (xuntext(xp))
                        count++;
        }
        return(count);
{
```

Fig. 5—Single processor code for `xumount`.

the lifetime of a system, the example illustrates the techniques for converting the code of a single processor *UNIX* system to a multiprocessor version. The function examines every `text` table entry to see if it is in use and if the file resides on the device `dev`. If so, it calls `xuntext` to free the swap space and free the text table slot.

Figure 6 shows the multiprocessor version of the `xumount` function. After the initial checks to ensure that the `text` table slot is in use and that its file is on the correct device, the semaphores for the `inode` and `text` slot are locked. The semaphores could be locked before the checks are done, but because `psema` and `vsema` are expensive operations, and because the probability that a `text` entry will be cleaned up here is low, the implementation is more efficient as shown. But until the `text` and `inode` slots are locked, it is possible for a process on another processor to change the `inode` pointer of the `text` slot or the device number of the `inode` if either is freed. Therefore, the code must check the conditions for calling `xuntext` again, and if either check fails, it must release the locked semaphores.

The `inode` semaphore is locked before the `text` semaphore, following the protocol established by the `exec` system call, where the `inode` is found first and locked before the `text` slot is allocated. If either `psema` call results in the process going to sleep, the process will later be rescheduled to run at priority `PSWP`.

Execution of the `xumount` function does not guarantee that the `text` table is free of program text from device `dev`, since a process executing on another processor could allocate a `text` slot that `xumount`

```
xumount(dev)
        register dev_t dev;
{
        register struct inode *ip;
        register struct text *xp;
        register count = 0;

        for (xp = &text[0]; xp < (struct text*)v.ve_text;xp++) {
                if ((ip = xp->x_iptr) == NULL)
                        continue;
                if (dev != NODEV && dev != ip->i_dev)
                        continue;
                psema(&ip->i_lock, PSWP);
                psema(&xp->x_lock, PSWP);
                if ((ip != xp->x_iptr)
                        |(dev != NODEV && dev != ip->i_dev))
                {
                        vsema(&xp->x_lock);
                        vsema(&ip->i_lock);
                        continue;
                }
                if (xuntext(xp))
                        count++;
                vsema(&xp->x_lock);
                vsema(&ip->i_lock);
        }
        return(count);
}
```

Fig. 6—Multiprocessor code for `xumount`.

already passed in its search for program text from the device. The calling code (`sumount`, not shown) prevents allocation of `text` slots to make such a guarantee.

3.4 Process execution

Processes executing in a multiprocessor environment are not aware of how many processors are running in the system. The only interaction between processes because of the multiprocessor environment is contention for semaphores, but subject to that restriction, each processor independently executes processes in both kernel and user mode, not in a master/slave fashion. Each processor schedules processes independently from a global set of runnable processes using conventional *UNIX* system scheduling algorithms. If a process is not scheduled by one processor, it is eligible for scheduling by the other processors. Multiple processes may be active in the kernel on separate processors, except for interaction of system semaphores. In particular, system calls give identical results in single or multiprocessor systems.

The major states of a process are

1. Running on a processor
2. Ready to run and loaded in main memory
3. Ready to run but not loaded in main memory
4. Sleeping and loaded in main memory
5. Sleeping and not loaded in main memory
6. Zombie (exited, waiting for its parent to acknowledge).

In the process table of single processor *UNIX* systems, no flag distinguishes the first state, currently running on a processor, from the second state, ready to run and loaded in main memory. But in multiprocessor *UNIX* systems, a new flag shows that a process is currently running on a processor. Without the explicit indication, it would be possible to schedule a process for simultaneous execution on multiple processors, or swap out a process currently executing on a processor, both clearly undesirable events.

3.5 Device drivers

In principle, there is no difference between device drivers and other parts of the operating system as far as conversion for running on a multiprocessor is concerned. Data structures must be locked, `sleep` and `wakeup` calls must be replaced by `psema` and `vsema` calls, and special consideration must be given to interrupt routines, as described previously.

But more than half of the the *UNIX* operating system currently consists of device drivers, and new drivers are being added at an accelerating rate to support new peripherals and to provide new or enhanced services. In practice, therefore, the number and volatility of

the drivers make it difficult to change them for multiprocessor systems and keep them up to date with changes made for other *UNIX* systems, so it is important to keep most driver code identical over all implementations. Three changes had to be made to the system to allow this.

First, drivers are locked before they are called. Driver calls are table driven via the `bdevsw` and `cdevsw` tables, and the drivers are locked and unlocked around the driver calls using driver semaphores added to the tables. Various methods of driver protection are encoded based on system configuration. The levels of protection vary from no protection (protection is then hard coded in the driver), to forcing the process to run on a particular processor (useful in AP configurations, where only one processor can do the I/O), to locking per major or per minor device type. Each call to a driver routine is now preceded by a call to a driver lock routine and followed by a call to a driver unlock routine.

The second change was to reimplement `sleep` and `wakeup` subroutines that could be called by device drivers, without changing the original driver code. Since the old *UNIX* operating system `sleep` routine uses arbitrary addresses in memory to sleep on, the new routines use hash lists of semaphores to actually suspend the process, and the address being slept on (a `sleep` parameter) is stored in the process table. Since a semaphore already heads a linked list of all processes suspended on the semaphore, the `wakeup` routine has only to search this list to find all processes to awaken. The `sleep` routine unlocks the driver semaphore so that other processes can access the driver while the original process sleeps, and it relocks the semaphore when it awakens from the sleep. The `sleep` and `wakeup` routines are intended to be used only from drivers. The main kernel code still uses `psema` and `vsema` directly.

In addition to the locking before calling driver routines, locking must also take place when handling interrupts, since the interrupt is no longer blocked by raising the processor execution level (see Section 3.1). Before the device interrupt handler is invoked, the semaphore for the device (if any) is locked via `cpsema`. If the lock succeeds, the interrupt gets handled; if the lock fails, the interrupt is queued but not handled immediately. When the process that currently has the semaphore locked is finished with the semaphore, it handles queued interrupt requests.

The above discussion does not hold for all multiprocessor *UNIX* systems, IBM/370 for example (see Section IV), but is the culmination of several years of evolution and represents the current state of development.

3.5.1 AP systems

As we discussed in Section I, AP systems do all I/O from one

processor, whereas MP systems can do I/O from all processors. Since it is desirable that the kernel and drivers have no knowledge of whether they are running on an AP or an MP system, the information is encoded in tables at the lowest software levels that send the direct memory access requests out to the hardware on AP systems. If the process is on the wrong processor, a context switch is done, and a special scheduling parameter forces the process onto the correct processor.

IV. IBM SPECIFIC ISSUES

The *UNIX* system for the IBM/370 does not run directly on IBM hardware, but is a two-level system where the upper level consists of *UNIX* system code, and the lower level consists of the resident supervisor of the Time-Sharing System (TSS). The resident supervisor handles all machine-dependent I/O operations, memory management (including paging), process scheduling, and hardware error handling. The *UNIX* system layer implements all *UNIX* system calls as well as the file system structure. The interface between the two layers consists of supervisor calls from the *UNIX* system to the resident supervisor, and pseudo-interrupts from the resident supervisor up to the *UNIX* system.

The major advantages of this approach are that the *UNIX* system on the IBM/370 does not have to concern itself with IBM hardware architecture that may change from processor to processor, and support for IBM peripherals comes for free, both via the resident supervisor. The disadvantages are that a performance penalty is paid in communication between the two layers, and that the system algorithms employed in the resident supervisor are not necessarily optimal for the *UNIX* operating system. For example, the semaphore operations are enhancements to enqueue/dequeue operations that previously existed in TSS and are much more general than required by the *UNIX* system.

V. 3B COMPUTER SPECIFIC ISSUES

The 3B family of machines is microcoded, so new semaphore instructions were encoded to boost performance of multiprocessor systems. The design of the instructions has been optimized for the most frequently occurring cases, namely, that `psema` usually finds the semaphore unlocked, and that `vsema` usually need not awaken sleeping processes. To this end, the instructions operate on registers containing the semaphore address and, if necessary, the address of a function that puts a process to sleep (for `psema`) or awakens a process sleeping on the semaphore (for `vsema`). Use of the new microcoded instructions

boosted overall system performance by 30 percent compared to a system that implemented semaphore operations in software.

A 3B hardware feature causes a problem in the implementation of a paging system for a multiprocessor configuration. Paging systems map the virtual address space of a process to physical pages in memory. The tables that define the mapping reside in memory, but for better performance they also reside in a special hardware cache called the Address Translation Buffer (ATB). Each processor has a private ATB and cannot flush the contents of the other processor's ATB. However, processes executing from shared text or using the shared memory interprocess communication facility (see Ref. 1) can share portions of their virtual address space. So the two processors' view of physical memory can diverge if one processor changes its address mapping, while the other processor continues to use the old mapping still contained in its ATB.

The paging problem is solved by observing the following protocol:

1. A processor flushes the user portion of its ATB during every context switch (this is done in systems without paging anyway, since the address mapping of the previously running process is invalid for the currently running process).
2. Kernel pages are never swapped from main memory.
3. Pages used by a process currently running on another processor cannot be swapped.

Since the paging process cycles through the `process` table swapping the oldest pages on a per-process basis, it is easy to satisfy the third rule above, provided the running process uses no shared text or shared data. If the running process does use shared text or shared data, the paging process verifies that the page to be swapped is not shared, or else it does not swap it.

VI. PERFORMANCE

Many *UNIX* operating system algorithms that use linear searches of system tables did not scale well from single processor to multiprocessor systems for two reasons. First, multiprocessor systems have greater capacity than their single processor counterparts, so systems tables such as the `inode` table and the `process` table have correspondingly more active entries, and consequently, searching for particular entries takes more time. Second, the system tables must be frequently locked so that processes accessing them find a consistent copy until they have finished using them. The two reasons combined imply that the system will spend more time searching the tables, locking them out from other processes and causing heavy contention for the table semaphores.

To avoid such problems, many algorithms were redesigned to avoid

linear searches of system tables. For instance, `inodes` are hashed by device number and `inode` number to a hash chain, and search algorithms that formerly searched the entire `inode` table for an `inode` now search for the `inode` on the hash chain, a much shorter search. Further, processes do not contend for a single semaphore for the `inode` table, but rather for a greater number of semaphores for the hash chains (see Section 3.2.5).

The `process` table is another example where linear searches were eliminated to gain performance. An exiting process, for example, finds all its "children" and reassigns their "parent" process identifier to be one, and it also sends a "death of child" signal to its parent. Instead of searching the entire `process` table for parent and child processes, the process structure now contains parent, child, and sibling pointers so that the search routines traverse a tree.

Benchmarking results show that two-processor *UNIX* systems run about 1.7 times as fast as a single-processor system. That is, 1.7 times as many processes are handled in the same amount of time as are handled on single-processor systems. The figures are based on benchmark programs that run job mixes typical of those found on *UNIX* systems, although CPU-bound job mixes run slightly faster, and I/O-bound job mixes run slightly slower. Performance enhancements are still being made and are expected to produce further improvements in these figures. Contention for semaphores is low, as less than 5 percent of the `psema` operations on lock semaphores result in the process going to sleep. By running the code for the multiprocessor system on a single processor and comparing its performance to that of a single-processor system running original *UNIX* system code, the overhead of semaphore operations was found to be less than 5 percent.

The multiprocessor system can be configured to run on a single processor by turning on a flag when compiling the system. The flag controls a macro that turns off selected semaphore operations. Performance of such a system is equal to that of regular single-processor systems. This has important ramifications for system support because one set of source code runs all system configurations.

VII. CONCLUSIONS

This paper has described the major problem of implementing multiprocessor *UNIX* systems, namely, concurrent destructive access of kernel data structures. It has discussed how to avoid concurrency problems in the kernel by using semaphores, and has outlined a scheme that allows drivers to stay common across single-processor and multiprocessor implementations. The resulting multiprocessor *UNIX* systems are functionally equivalent to single-processor *UNIX* systems

and provide 70 percent better throughput for two-processor configurations than their single-processor counterparts do.

The techniques outlined in this paper are applicable to all *UNIX* systems, independent of the machine on which they run. They are particularly applicable to microprocessors running the *UNIX* system, because they allow users to increase their computing power by adding more processors to their system.

VIII. ACKNOWLEDGMENTS

We would like to thank the following people, who have worked on the multiprocessing *UNIX* system projects: Bob Bison, Yuhlan Cho, Hugh Devore, Bob Earnst, Bill Felton, Ezra Goldman, Clyde Imagna, Robert Kennedy, Jeff Kinker, Steve Kiseli, Bart Prieve, Tom Richards, Doris Ryan, Tom Schlagel, Jeff Smits, Paul Swigert, Dan Tiernan, Tom Vaden, Mike Wilde, and Robert Zarrow. Special thanks to Ian Johnstone for his work on both projects.

REFERENCES

1. G. H. Goble and M. H. Marsh, "A Dual Processor VAX 11/780," Purdue University Technical Report, TR-EE 81-31, September 1981.
2. E. W. Dijkstra "Solution of a Problem in Concurrent Programming Control," CACM, *8*, No. 9 (September 1965), pp. 569–78.
3. E. W. Dijkstra "Cooperating Sequential Processes," *Programming Languages*, F. Genuys, ed., New York: Academic Press, 1968, pp. 43–112.
4. E. W. Dijkstra "The Structure of T.H.E. Multiprogramming System," CACM, *11*, No. 5 (May 1968), pp. 341–6.
5. J. A. Hawley and W. B. Meyer "MUNIX, A Multiprocessing Version of *UNIX*," M.S. Thesis, Naval Postgraduate School, Monterey, California, 1975.
6. *UNIX System Users Manual*, Release 5.0, June 1982, Bell Laboratories, Inc.

AUTHORS

Maurice J. Bach, B.A. (Physics), 1973, Yeshiva University; Ph.D. (Computer Science), 1979, Columbia University; AT&T Bell Laboratories, 1977—. At AT&T Bell Laboratories Mr. Bach first worked on database translation systems. Before joining the *UNIX* Systems Development department in 1982, he worked on experimental multi-microprocessor systems. Member, ACM.

Steven J. Buroff, B.S. and M.S. (Electrical Engineering), 1968 and 1969, respectively, Ph.D. (Computer Science), 1977, Illinois Institute of Technology; AT&T Bell Laboratories, 1977—. Mr. Buroff worked on the first porting of a *UNIX* system to a multiprocessor. He has also been involved with other multiprocessor *UNIX* system implementations and has recently helped design a new processor architecture. Mr. Buroff is currently working on converting the *UNIX* system from a swapping to a paging system.