To appear in the proceedings of
"Second USENIX Symposium on Microkernels and Other Kernel Architectures", September 1993.
Reprinted by permission of the USENIX Association.

Kernel Support for the Wisconsin Wind Tunnel*

Steven K. Reinhardt, Babak Falsafi, and David A. Wood

Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA
wwt@cs.wisc.edu

Abstract

This paper describes a kernel interface that provides an untrusted user-level process (an *executive*) with protected access to memory management functions, including the ability to create, manipulate, and execute within subservient contexts (address spaces). *Page motion callbacks* not only give the executive limited control over physical memory management, but also shift certain responsibilities out of the kernel, greatly reducing kernel state and complexity.

The *executive interface* was motivated by the requirements of the Wisconsin Wind Tunnel (WWT), a system for evaluating cache-coherent shared-memory parallel architectures. WWT uses the executive interface to implement a fine-grain user-level extension of Li's shared virtual memory on a Thinking Machines CM-5, a message-passing multicomputer. However, the interface is sufficiently general that an executive could act as a multiprogrammed operating system, exporting an alternative interface to the threads running in its subservient contexts.

The executive interface is currently implemented as an extension to CMOST, the standard operating system for the CM-5. In CMOST, policy decisions are made on a central, distinct control processor (CP) and broadcast to the processing nodes (PNs). The PNs execute a minimal kernel sufficient only to implement the CP's policy. While this structure efficiently supports some parallel application models, the lack of autonomy on the PNs restricts its generality. Adding the executive interface provides limited autonomy to the PNs, creating a structure that supports multiple models of application parallelism. This structure, with autonomy on top of centralization, is in stark contrast to most microkernel-based parallel operating systems in which the nodes are fundamentally autonomous.

*This work is supported in part by NSF PYI Award CCR-9157366, NSF Grant MIP-9225097, a Wisconsin Alumni Research Foundation Fellowship, an A.T.&T. Bell Laboratories Ph.D. Fellowship, and donations from Xerox Corporation, Thinking Machines Corporation, and Digital Equipment Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the Univ. of Wisconsin Graduate School.

1 Introduction

This paper describes the kernel interface designed to support the Wisconsin Wind Tunnel (WWT) [13], a system for parallel simulation of parallel computers. WWT currently runs on the Thinking Machines CM-5 (a message-passing machine) and simulates cache-coherent shared-memory multiprocessors. Shared memory applications execute directly on the CM-5 node processors, with WWT simulating references to remote data. Shared memory functionality is provided using a fine-grain user-level extension of Li's shared virtual memory [10], as described in Section 2.2. WWT uses a separate address space for each simulated (target) node and services all of its exceptions (e.g., MMU faults) and system call requests (e.g., file I/O). In order to study machines larger than the host, several target nodes timeshare a single CM-5 node. In many ways, WWT behaves like an operating system for shared-memory applications. Alternatively, WWT can be thought as providing a virtual machine abstraction—with a shared-memory MIMD machine atop a message-passing pseudo-SIMD machine—for these applications [8].

WWT requires several unusual features from the underlying operating system. Specifically, the kernel must allow WWT to:

- Create subservient contexts (address spaces)
- Manipulate page mappings within sub-contexts
- Initiate execution in sub-contexts
- Handle traps generated during execution in sub-contexts
- Manage physical memory tags in sub-contexts[1]

All of these features must coexist with traditional memory management functions, including paging and/or swapping.

We have defined and implemented an interface which provides these features and call any application that makes use of them an *executive*. While the interface is motivated by our specific application, it provides any untrusted user-level process with protected access to memory management functions, including the ability to create, manipulate, and execute within subservient contexts (address spaces). *Page motion callbacks* not only give the executive limited control over physical memory management, but also shift certain responsibilities out of the kernel, greatly reducing kernel state and complexity.

Because an executive creates contexts and controls them completely, it can act as the operating system for other applications, providing an execution model not available under the native operating system. For example, an executive can export various thread and memory abstractions without adding complexity to the kernel itself.

While this flexibility is useful in a uniprocessor context, it is particularly important on the CM-5, since standard CMOST is a centralized, synchronous operating system, allowing little autonomy for the node kernels. CMOST's structure efficiently supports an important class of parallel applications, i.e. fine-grain data-parallel codes, but cannot take advantage of more autonomous execution models. By extending CMOST with the executive interface, we provide a kernel structure that can efficiently support both synchronous and asynchronous applications.

[1]We have effectively extended the CM-5 architecture to support two bits of tag information for each 32-byte block of physical memory; see Section 2.3.

The combination of CMOST and executive interface results in a unique kernel structure that provides autonomy on top of synchrony, rather than the more traditional approach of coordinating fundamentally autonomous nodes. This new kernel structure may prove superior because centralized control appears to have advantages for supporting fine-grain synchronous codes and managing global hardware resources, while the executive interface provides flexible support for other execution models.

The next section provides background on the Thinking Machines CM-5 system, the Wisconsin Wind Tunnel, and our method of synthesizing memory tags on the CM-5. While the interface is not tied to any of these, this section provides context and motivation for the rest of the paper. Section 3 defines the interface, consisting of context manipulation calls, page motion callbacks, and execution management calls. Section 4 describes our implementation of the interface in CMOST and its performance. Section 5 discusses the implications of this work for the structure of multiprocessor operating systems. Finally, we discuss related work and our conclusions.

2 Background

2.1 Thinking Machines CM-5 and CMOST

The Thinking Machines CM-5 [16] is a distributed-memory message-passing multiprocessor. Each processing node consists of a 33 MHz SPARC microprocessor with a cache and memory management unit, up to 128 MB of memory, a custom network interface chip, and optional custom vector units. The processing nodes are grouped into partitions of 32 or more processors. Each partition is managed by a control processor (CP), distinct from the processing nodes (PNs).

The standard operating system for the CM-5 is CMOST. Under CMOST, all policy decisions, including scheduling, swapping, and memory allocation, are made on the control processor. The processing nodes execute a minimal microkernel (the *PN kernel*) which provides the bare mechanisms required to implement the CP's policy. Because all processors in the partition are managed as a synchronous unit, CMOST gives the CM-5 some SIMD-like qualities. For example, when the CP decides to context switch, all nodes simultaneously switch to the new context. Similarly, when a new process is created, the CP selects the physical pages which the process will occupy on the PNs and broadcasts that process's memory map.

CMOST and the CM-5 are optimized to run data-parallel applications, where all nodes synchronously apply similar operations to a local subset of a global data structure. In particular, the CM-5 contains a "control network", distinct from the message-passing network, which provides hardware support for global operations such as barriers, reductions, and broadcasts [9]. To efficiently utilize this control network, all nodes in a partition must concurrently execute the same user process. The centralized CMOST structure automatically satisfies this condition.

2.2 The Wisconsin Wind Tunnel

The Wisconsin Wind Tunnel (WWT) provides a platform for evaluating parallel computer systems—specifically cache-coherent shared-memory computers—by accurately modeling the performance of real workloads on proposed hardware [13]. WWT helps computer engineers evaluate computer architectures much like a wind tunnel helps aeronautical engineers design aircraft. WWT uses the execution of a parallel shared-memory application to drive a distributed discrete-event simulation, accurately calculating the execution time of that application on a modeled hardware system (the *target*). Events generated by the simulation, such as lock acquisitions and memory reference completions, are used in scheduling the application, guaranteeing that the application's execution proceeds exactly as it would on the target system.

We call WWT a *virtual prototype* because it uses direct execution to leverage similarities between the target system and the system on which it executes (the *host*) [5]. This means that the target application executes directly on the host hardware as much as possible—for example, a target floating-point multiply runs as a host floating-point multiply. Software simulation is required only for those features of the target system not provided by the host.

Because WWT executes on a message-passing machine, the primary feature it must simulate is the shared memory abstraction. We do this using a fine-grain extension of Li's shared virtual memory [10]. Shared virtual memory constructs a distributed shared memory using standard address translation hardware to control memory access on each node. If a node has a copy of a shared data page, it is mapped into the address space on that node; if a node has no copy, the virtual page is not mapped. Multiple read-only copies are easily supported using the page protection facilities. Program accesses that require a data transfer to acquire a valid or exclusive copy are signaled as page faults. Unfortunately, relying on address translation hardware alone restricts the granularity of coherence to at least the virtual memory page size.

The shared-memory machines we wish to model maintain coherence at a finer granularity, typically tens of bytes rather than thousands. We have synthesized the ability to tag each 32-byte block in physical memory as *invalid*, *read-only*, or *writable* (see Section 2.3). Using these tags in combination with the address translation hardware, we implement a distributed shared memory that maintains coherence at a 32-byte granularity. The first reference to a shared page causes a page fault, as with shared virtual memory. We allocate and map a physical page, but initially mark each cache block invalid. Cache blocks are marked valid (read-only or writable) only as they are referenced. Accesses to invalid blocks (and writes to read-only blocks) cause faults, and initiate software that fetches the data and marks the block valid. The distinction between read-only and writable tags allows read replication at the cache block granularity.

WWT uses this fine-grain shared virtual memory to directly execute shared-memory applications as they would execute on a cache-coherent target machine. A context is allocated for each target node; the shared data accessible from this context reflects the contents of the simulated cache on that target node. Page and tag faults correspond to target cache misses, which invoke WWT and are handled according to the target's coherence protocol. Large target systems are studied by allocating several contexts per host node and multiplexing their execution.

2.3 Memory tags

To implement fine-grain shared virtual memory, blocks in memory must have three states: invalid, read-only, and writable. Any access to an invalid block and write accesses to read-only blocks must provide restartable exceptions. To achieve this functionality, we have logically extended the CM-5 architecture to support two additional bits of information—*writable* and *invalid*—per 32-byte physical memory block.

Although memory tags with access semantics have appeared in numerous machines, e.g., the Denelcor HEP [15], most contemporary commercial machines—including the CM-5—do not provide this capability. However, we are able to synthesize an invalid tag on the CM-5 by forcing uncorrectable errors in the memory's error correcting code (ECC) via a diagnostic mode. Using the SPARC cache in write-back mode causes all SPARC cache misses to appear to the memory as cache block fills. A fill that encounters an uncorrectable ECC error generates a precise exception.

Synthesizing a read-only state is more convoluted, since it requires using the page tables to make entire pages read-only. On a write fault, we must distinguish between a write to a read-only block and a write to a writable block that resides on the same page as one or more read-only blocks. We make this distinction by maintaining a bit vector—one bit per block—to indicate whether the block is writable. The write fault handler checks this bit; if set it performs the write and resumes the application, rather than signaling a fault.

Memory tags introduce extra state—two additional bits per 32-byte block—making paging and swapping more complex to implement. The tag bits make the "extended" physical page no longer a power of two, causing a mismatch with typical disk block sizes, and requiring more bookkeeping and I/O operations. In addition, since memory tags are unused for many pages, e.g., text and non-shared data, any overhead maintaining them is wasted. The executive interface reduces the kernel's complexity by shifting responsibility for maintaining memory tags to the executive.

3 The executive interface

The executive interface provides an *executive*—an untrusted user-level process—with protected access to memory management functions. An executive can use the interface's memory management calls to create subservient contexts (address spaces), and exert complete control over them, including adding, modifying, and deleting page-level mappings. An executive can invoke execution within a subcontext, and regain control on all faults and exceptions. Page motion callbacks not only give the executive limited control over physical memory management, but also shift certain responsibilities out of the kernel, greatly reducing kernel state and complexity.

A primary goal of the executive interface is to minimize kernel state and complexity. Beside the aesthetic appeal, keeping most of the code and complexity at user level makes bugs less catastrophic and easier to eliminate. In addition, because we are modifying a continuously developing system, minimizing and isolating the kernel source changes makes it easier for us to keep up with vendor revisions.

A particular challenge is maintaining the address mappings and memory tag values installed by an executive in the face of paging/swapping activity by the kernel. The brute-force solution is to have the kernel remember all of the mapping requests made by the executive and transparently maintain them when a page is swapped out and back in at a different physical address, and to transparently swap tag information as well as page data. We have solved this problem with greatly reduced kernel state and complexity using *page motion callbacks*. These callbacks allow the kernel to notify an executive immediately before a page is to be swapped out and immediately after it is swapped back in so that the executive itself can maintain address mappings and tag values.

Another requirement is to avoid trusting the executive. A fully protected interface makes the system robust through even the earliest phases of executive development, and means that normal users have the ability to write or modify executives. A protected interface also makes other sites more willing to adopt our kernel so that we can distribute the Wisconsin Wind Tunnel. Two features of the interface contribute to this protection:

- The executive cannot even refer to resources not explicitly allocated to it by the kernel. The executive never sees physical addresses or hardware context numbers.

- The kernel guarantees that the executive never has an alias to a physical page it does not own by maintaining a count of aliases for each physical page. If the executive does not decrement this count to zero by deleting mappings before a page is removed from its control, the kernel will terminate it.

```
int create_ctx();
void *executive_brk(void *new_brk);
void *executive_sbrk(int incr);
void *executive_vbrk(void *new_brk);
void *executive_vsbrk(int incr);
int add_mapping(int cd, void *va, void *pp, int attr);
int change_pg_attr(int cd, void *va, int attr);
int delete_mapping(int cd, void *va);
void jump_to_ctx(int cd, struct regs *p, void *stackp);
```

(a) General kernel calls

```
int set_page_motion_cbs(void *(*page_going)(),
                        void (*page_coming)(),
                        void *stackp);
int set_ctx_fault_cb(void (*ctx_fault_cb)());
```

(b) Callback registration calls

```
void *page_going(void *pp);
void page_coming(void *pp);
void ctx_fault(int fault_code, struct regs *p, ...);
```

(c) Callbacks

Table 1: The executive interface. Parts (a) and (b) list the functions exported by the kernel. Part (c) describes the callbacks exported by the executive.

Table 1 lists the calls and callbacks that comprise the executive interface. The kernel exports the general calls and callback registration functions, while the executive exports the callbacks, which it registers during initialization. The bulk of the interface is directly related to managing virtual and physical memory. The remaining functions, jump_to_ctx() the ctx_fault() callback, provide the ability to execute within subcontexts.

3.1 Memory management

The executive manages virtual and physical memory resources via the context management calls and page motion callbacks. Pages managed by the executive are distinct from the executive's own text, data, and stack pages. This allows the kernel to easily distinguish the former for special handling while manipulating the latter as it would for any other user process. For example, the kernel can swap the executive's text segment, share it among multiple instances of the same executive, or allow a debugger to attach to an executive without interfering with the executive's memory management functions.

3.1.1 Context management calls

The context management calls allow an executive to create new contexts, allocate pages to map into them, and add, modify, and delete page-level mappings. Using these calls, an executive has complete control over these subservient address spaces.

The create_ctx() call allocates a new context and returns an integer context descriptor (similar to a Unix file descriptor). We refer to these contexts as *subcontexts* when it is necessary to distinguish them from the executive's context. A new subcontext is completely empty, i.e. it contains no valid address mappings (except for the kernel mappings required by the SPARC architecture). Context descriptor 0 is never returned by create_ctx() and is used to indicate the executive's own address space. The notation cd:va refers to virtual address va in context cd.

To keep executive-managed pages distinct from kernel-managed pages, the executive allocates pages from a special segment in the executive's own context, the *executive-managed heap*. The executive_brk() and executive_sbrk() calls allow the executive to change the size of this segment in the same way that CMOST's Unix-like brk() and sbrk() work with the standard heap. Allocation on the executive-managed heap is always rounded up to the next multiple of the page size, so that an integral number of pages are allocated. The virtual addresses of these pages in the executive's context are the *primary mappings* (i.e., handles) which the executive uses to refer to these pages across the kernel interface. Only pages in the executive-managed heap—referred to as *executive-managed pages*—may be aliased via add_mapping() or have their memory tag values changed.

The *executive virtual heap* allows the executive to manage a region of its own address space the same way that it manages subcontexts. The executive_vbrk() and executive_vsbrk() calls simply reserve a contiguous collection of virtual pages, but do not allocate physical pages behind them. The executive can then alias these virtual pages to pages in the executive-managed heap (possibly with different attributes, e.g. read-only or non-cacheable) without fear that the kernel will grow the standard heap or stack to conflict. This call is not necessary for subcontexts because the executive automatically has complete control of those.

The add_mapping() call creates a *secondary mapping* from virtual address va in context cd to the page at virtual address pp, i.e. it aliases cd:va and 0:pp, where pp must point to an executive-managed page. The mapping attributes (protection and cacheability) are set according to attr. To prevent interference between the kernel and executive, if cd is zero then va must be in the executive virtual heap. The alias count for the corresponding physical page is incremented.

The change_pg_attr() and delete_mapping() calls allow the executive to change the attributes of and delete mappings, respectively. Only mappings created via add_mapping() can be modified or deleted. delete_mapping() also decrements the physical page's alias count.

3.1.2 Page motion callbacks

The two page-motion callbacks—page_going(), invoked when the kernel must reclaim an executive-managed page, and page_coming(), invoked when a page returns—serve a dual role. First, they allow the kernel and executive to cooperate in physical memory management, similar to the way scheduler activations allow management of physical processors in a shared-memory multiprocessor [2]. By explicitly saving and restoring data in response to page_going() and page_coming() calls, the executive can control exactly which data are resident in physical memory. Second, the callbacks significantly reduce the kernel's bookkeeping requirements, by giving the executive responsibility for maintaining secondary mappings and memory tags across physical page movements.

The two page motion callbacks only affect executive-managed pages; the executive must register the callbacks before allocating pages on the executive-managed heap. When the kernel decides to reclaim an executive-managed page (e.g., to allocate it to a different process), it notifies the appropriate executive using the page_going() callback. In general, the kernel will call page_going() with the argument pp set to NULL, indicating that the executive can select any executive-managed page for reclamation. The executive can apply its own replacement policy and return the selected page-aligned pp as the return code. By default, the kernel discards the page contents; however, the executive may request that they be saved (i.e. moved to backing store) by overloading the return code (setting the least-significant bit to one). Occasionally, the kernel may need contiguous physical pages—e.g., for the CM-5 vector units—requiring it to reclaim a specific executive-managed page. In this case, the argument pp points to the selected page, and only the least-significant bit of the return value is meaningful.

The executive must use delete_mapping() to delete all secondary mappings for the selected page before returning, at which point the kernel removes the primary mapping from pp to the physical page. Note that virtual address pp is still "in use" as it uniquely identifies this page and will be provided as the argument to a future call to page_coming().

The kernel returns an executive-managed page via the page_coming() callback. The primary mapping is recreated, i.e. the virtual address pp again maps to a physical page, though not necessarily the same physical page as before the page_going() call. If the executive returned a zero in the LSB on the previous call to page_going(), the page has been zeroed; otherwise, the contents have been restored. In either case, the physical memory tags have been cleared. This call allows the executive to restore any secondary mappings and/or memory tags for the page. Secondary mappings could also be restored on demand.

These callbacks allow user-level management of physical memory: even when the kernel reclaims a specific physical page, the executive can choose the data that get replaced at the expense of additional copying and page-table manipulation. Alternatively, the executive can let the kernel save and restore data in evicted pages. The executive can always regain access to a "gone" page by dereferencing pp and causing a fault. The kernel will handle the fault by obtaining a free page (possibly by calling page_going() on this or another executive), restoring the old data (if necessary), and calling page_coming() to signal the return of the needed page.

Because the executive is untrusted, the kernel cannot rely on it to delete all secondary mappings on a page_going() callback. Conversely, it must guarantee that these mappings are removed—otherwise, the executive may retain an alias to a physical page re-allocated to a different process. A brute-force solution is for the kernel to automatically delete all secondary mappings. However, this approach requires that the kernel maintain all of the reverse translations, duplicating state already maintained by the executive. Instead, we require that the executive delete all secondary mappings to the selected page before returning from the page_going() call or face process termination. Process termination guarantees that all mappings are deleted, because the process and all its sub-contexts are destroyed. Thus the kernel only need know *how many* secondary mappings exist, but not their individual identities. A simple counter per physical page, incremented on each add_mapping() call and decremented on each delete_mapping call, is sufficient to maintain this state.

In addition, to protect against deadlock or infinite loops in the executive, the kernel requires that all callbacks be completed within a fixed time. The kernel sets a virtual timer before invoking a callback; if the timer expires before the callback returns the executive's process is terminated.

3.2 Execution management

Creating and manipulating address spaces is uninteresting without the ability to execute within them. Two calls suffice to provide this functionality. The kernel call jump_to_ctx() causes the current thread of execution to switch into the specified context. When the thread executing in the subordinate context encounters a fault (either an instruction fault or an explicit software trap), control resumes in the executive's context via the ctx_fault() callback.

Because the jump_to_ctx() call continues the current thread in a different context rather than creating a new thread, the contents of the register file are largely unchanged across the switch. The struct regs structure passed as an argument to jump_to_ctx(), though implementation-dependent, conceptually consists of only the program counter and stack pointer in the new context and the original contents of the registers required to pass the three arguments of jump_to_ctx() into the kernel.[2] The third argument, stackp, specifies a stack in the executive context to use when ctx_fault() is invoked. From the executive's perspective, jump_to_ctx() does not return. After a thread passes through a jump_to_ctx() call, it is in a subordinate execution context where all faults are handled by the executive.

When the thread executes a trapping instruction (either due to a fault or an explicit software trap), control resumes in the executive's context at the ctx_fault() entry point. The first argument indicates the type of fault and the second passes back a pointer to the same structure originally passed to jump_to_ctx(). The state saved is exactly the state required by jump_to_ctx(), so supplying this buffer unmodified as the second argument to jump_to_ctx() restarts execution in the other context at the faulting instruction.

Additional arguments are passed from the kernel to the executive depending on the type of the fault—for example, an MMU fault will also provide the virtual address of the access and the nature of the fault (invalid address, protection violation, etc.).

We have, to the greatest extent possible, separated thread management issues from this interface. However, with the addition of thread management code, this simple interface is sufficient for the executive to behave as a multiprogrammed operating system. Implementing a threads package on top of this interface simply requires code to allocate and manage multiple stacks, and to save and restore the registers not contained in the struct regs structure. For example, a context switch is as simple as having the ctx_fault() handler save and restore the CPU registers and call jump_to_ctx() with a different context descriptor and struct regs pointer. Separating the thread and context abstractions gives the executive flexibility, e.g. to support a kernel thread abstraction within its sub-contexts. If the underlying kernel provides a programmable timer interrupt, the interrupt can be made to appear through the ctx_fault() entry as well, making the multithreading preemptive.

The executive does not normally handle its own faults, except for those to the executive virtual heap. Having the kernel handle the executive's faults facilitates demand paging of the executive's text and data, and makes growing the executive's stack the same as for any other user process. If the executive wants to handle its own faults, it can call jump_to_ctx() with the first argument set to zero (its own context). This creates a singular situation where the thread is in a subordinate execution context—so faults still invoke the ctx_fault() callback—but not a subordinate addressing context. If the executive decides that the kernel should handle a specific fault, it need only retry the faulting instruction from within its fault handler. There is no danger of a recursive call to ctx_fault() because any fault encountered by the executive's handler will be handled directly by the kernel.

Subordinate execution contexts cannot be nested because jump_to_ctx() is a system call implemented as a software trap; "recursive" calls will show up in the executive via ctx_fault(). As with any other traps, system calls made in a subordinate execution context can be forwarded to the kernel simply by re-executing the call in the executive. The only complication occurs with pointer arguments, since the kernel will interpret these in the executive, rather than subordinate, context.

[2] On the SPARC architecture, this structure also contains the NPC (to resume after faults in delayed branch slots) and the condition codes (since these cannot be saved and restored from user mode).

Function	Lines of C	
	PN	CP
executive_{v}{s}brk		
Total, four calls	10	35
create_ctx	134	0
Page table initialization	17	11
Process termination	90	0
Page callback support	222	426
Total	473	472

(a) C language additions

Function	Machine Instructions
add_mapping	187
change_pg_attr	108
delete_mapping	123
jump_to_ctx	50
ctx_fault	40
set_page_motion_cbs	7
set_ctx_fault_cb	5
Total	520

(b) PN assembly additions

Table 2: Code added to CMOST to implement executive interface.

4 Implementation

We have implemented the executive interface in CMOST version 7.2 Beta 1. As shown in Table 2, the entire interface required less than one thousand lines of C and just over 500 machine instructions. Only jump_to_ctx() and ctx_fault() required assembly-level coding; the other functions were implemented in assembly to improve performance. Although the majority of the additional code is in the PN kernel, most of the complexity lies in the CP portion. This follows from the centralized structure of CMOST: the PN kernel implements only mechanisms, while all policy decisions occur on the CP.

The executive interface requires very little additional kernel state. The PN kernel requires a few additions to the process control block (PCB) and an array with an entry per physical page. The PCB maintains the entry points and stack addresses for the callbacks and the struct regs and stackp pointers from the last jump_to_ctx() call (for use in the subsequent ctx_fault() callback). The array, indexed by physical page number, contains the alias count for each page and a pointer field. The pointer field is used to maintain a linked list, whose head is in the PCB, of all physical pages in the executive-managed heap to facilitate resetting the alias counts when the executive process terminates.

We also added two new memory segments to every process: the executive-managed heap and the executive virtual heap. Both segments have special semantics:

- Any time the CP decides to move a page in the executive-managed heap, it must first invoke the executive's page motion callbacks.
- Allocations in the executive virtual heap segment are not backed by physical memory. This segment simply provides a region in the executive's virtual address space that is guaranteed not to conflict with regions used by the control processor. Also, faults to this segment are always handled by the executive.

On the CP, two fields were added to the per-physical-page structure to record the process ID and virtual address of each physical page that is allocated to an executive-managed heap segment. This information is required to perform the page_going() callback when the physical page needs to be moved.

Table 3 summarizes the performance of the executive interface calls, as measured using

Function	Time[3] cycles (μs)
executive_sbrk	
with CP communication	
alloc 1 page	1.6M (48 ms)
alloc 100 pages	4.9M (148 ms)
w/o CP communication	20K (606)
executive_vsbrk	
with CP communication	1.4M (42 ms)
w/o CP communication	20K (606)
set_page_motion_cbs	117 (4)
set_ctx_fault_cb	108 (3)

Function	Time[3] cycles (μs)
create_ctx	19K (575)
add_mapping	
no table allocation	359 (11)
alloc level 3 table	1166 (35)
alloc level 2 & 3 tables	2641 (80)
change_pg_attr	340 (10)
delete_mapping	855 (26)
jump_to_ctx	180 (5)
ctx_fault	154 (5)

Table 3: Performance of executive interface calls on the CM-5.

the CM-5's cycle counter and averaging tens of iterations.[3] The executive_{v}{s}brk() and create_ctx() calls are implemented as full traps, where the user's register windows are flushed, execution switches to the kernel stack, and interrupts are re-enabled. All other calls are implemented as "fast" traps, and execute without flushing any windows or re-enabling interrupts. The overheads for the two types of traps are approximately 300 μs and 3 μs, respectively.

The callback registration functions simply store their arguments in fields in the process's PCB. No validation is required since illegal values can at worst cause an immediate fault on the invocation of a callback, which will terminate the executive process.

4.1 Memory management

4.1.1 Context management calls

The executive_{v}{s}brk() calls are implemented in the same fashion as CMOST's brk() and sbrk(). The first node requesting a page generates a system-wide interrupt, which causes the control processor to grow the appropriate segment on all nodes. Subsequent requests on other nodes can be satisfied locally by returning pages from the now-larger segment.

The create_ctx() call allocates a SPARC hardware context number and an empty level-one table (the SPARC has a three-level page table structure). A call to add_mapping() validates its arguments, performs the page table insertion (allocating level two and three tables as necessary), and increments the alias counter for the physical page. Both change_pg_attr() and delete_mapping() validate arguments and do a page table walk, with the latter also decrementing the appropriate alias counter. None of these calls require communication from the node to the control processor.

Because the CM-5 node has a virtually-tagged writeback cache, delete_mapping() must also flush data brought in using the deleted mapping. This requires iteratively flushing 128 cache lines, causing it to take significantly longer than change_pg_attr().

[3]This information was derived using a pre-release version of CMOST (7.2 Beta 1 of Feb. 1993). Performance on released versions may be significantly different.

4.1.2 Page motion callbacks

The page motion callbacks account for most of the design complexity. Our current implementation focused on minimizing changes in the CP code. As a result, our implementation is influenced by several existing CMOST features:

- Swapping is supported, but not demand paging. All of a process's pages must be in memory before it is allowed to run.
- Memory management is performed entirely on the control processor, which assumes that the memory maps of all nodes are identical. When the CP reclaims a page, it must select a specific physical page and force all nodes to release it, even if there is no need for physical contiguity. In other words, in this implementation, `page_going()` is never called with a `NULL` argument because the CP cannot deal with different nodes freeing different pages.
- Both the PN kernel and the CP portion of CMOST are single-threaded, i.e. there is only one stack in each. In addition, the PN kernel does not maintain any kernel stack state across communications with the CP.

While this implementation satisfies our needs for WWT, and serves as a proof-of-concept for the executive interface, the CP code requires more radical changes for a clean and efficient implementation.

In CMOST, pages are freed for two reasons: i) to satisfy an allocation request for a currently executing process or, ii) to swap in an idle process. In either case, the CP performs the necessary memory management operations while the PNs are idle, waiting for instructions to resume. Page moves not requiring callbacks are performed immediately, but those requiring callbacks are recorded and deferred. Before resuming the user process, the CP performs the deferred callbacks, scheduling the affected processes (executives) as needed.

The callbacks are executed in the executive context by invoking the registered callback function using the registered stack pointer. The callback cannot be run on the current process stack because it could be in a different address space (i.e. if the process was suspended while running in a subcontext). A fault in the callback will result in the executive's termination. When the callback returns, the PN kernel either executes another callback pending for this executive, or waits for the other nodes to complete. Once all callbacks are done for this executive (across all nodes), the scheduler is re-invoked to run the next executive with pending callbacks. When all deferred callbacks have executed, the originally scheduled process can run.

Because the executive is scheduled for callbacks the same way it is scheduled for normal execution, there are only a few constraints on callback execution. First, the callback cannot allocate memory of any kind since this could create a circular dependency. Second, each callback is provided at most one scheduling quantum; the executive is terminated if the quantum expires during a callback. Third, the callback also cannot call any blocking kernel function, since this would interfere with the callback timeout mechanism. Finally, the current implementation does not allow the callback to use the CM-5 network interface. This last restriction is not inherent to the architecture, but would add significant complexity and overhead for a feature our application would not use.

Our implementation is designed to support swapping, but we have not yet tested this portion of the code. However, the CM-5 vector unit architecture severely constrains physical memory allocation, causing CMOST to frequently reallocate specific pages, moving data from one physical page to another. By treating these page moves as a swap out followed immediately by a swap in, we have completely exercised the callback mechanisms.

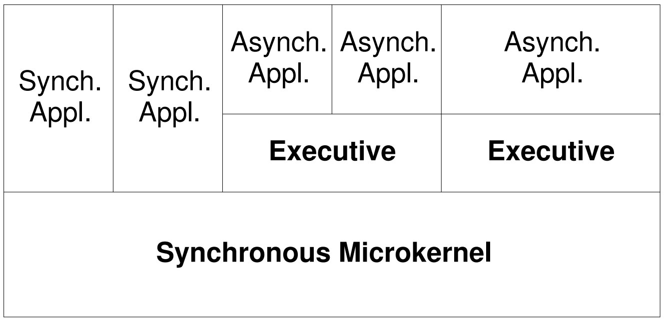

Figure 1: Mixed-model parallelism using the executive interface.

4.2 Execution management

The subordinate execution context is simply the same CMOST process executing with a different trap vector and (perhaps) a different hardware MMU context. All trap vector entries, except hardware interrupts, jump to the ctx_fault() kernel stub.

The jump_to_ctx() and ctx_fault() functions are implemented as "fast" traps, i.e. they execute without re-enabling interrupts in a partial SPARC register window. In addition to loading state from the struct regs structure, (or storing it in the case of ctx_fault()), both calls must change the hardware context, manipulate the register window mask, and change the trap vector base address. jump_to_ctx() requires ten extra instructions because it must read the processor status register, mask in the desired condition codes, and write it back.

4.3 Other implementation issues

The executive needs to specify a stack for all interrupt or signal handlers, as it does for the page motion callbacks, since the current process stack may not exist in the executive's own address space. We have not added this extension yet, but it would be simple to do so.

The cache controller used on the CM-5 node (Cypress 604) is a 64KB direct-mapped virtually-tagged cache. The hardware will handle aliases that map to the same cache block, but cannot guarantee consistency otherwise. To avoid cache flushing, aliases must be congruent modulo 64K. Rather than complicating the interface with this issue, we force the executive to deal with it on its own. We have added an additional call to the PN kernel, int cache_flush_page(int cd, void *pp), which flushes the specified page from the SPARC cache. This allows the executive to make the tradeoff between keeping aliases congruent and performing cache flushes according to its own needs.

5 Discussion

While the executive interface is interesting in isolation, it is more striking when considered in the context of the CM-5 and CMOST. The resulting kernel structure is—we believe—unique. In most microkernels designed for parallel systems, nodes are fundamentally autonomous. Cooperation, e.g. for gang-scheduling, occurs as a policy at a higher level of abstraction. Our extended version of CMOST turns this structure on its head: the control processor maintains central, synchronous control of physical memory and scheduling. The control processor still forces all processors to context switch simultaneously; however, some of the processes may now be executives. Executives, because of the autonomy provided by the executive interface, can schedule execution in their subcontexts however they choose. This flexibility can be used to support applications or groups of applications which may benefit from more dynamic allocation and scheduling policies (see Figure 1). Thus our extended CMOST provides autonomy on top of synchrony, rather than the more traditional alternative of synchrony on top of autonomy.

The CMOST/executive structure was motivated by our implementation of the Wisconsin Wind Tunnel on the CM-5. However, the resulting structure is arguably the right way to structure an operating system for large-scale parallel machines. Efficiently supporting fine-grain parallel applications requires a global perspective for resource allocation, because a page fault or scheduling delay on one node can seriously impact the performance of the entire application. Centralizing control, as in CMOST, makes global resource allocation significantly easier. For example, CMOST's guarantee that one user process runs simultaneously on all nodes allows direct user access to the CM-5's network interface hardware, avoiding costly system calls for message operations.

Operating systems that fail to efficiently manage global resources will have a particularly difficult time exploiting hardware features such as the CM-5's control network [9], which performs a global barrier or reduction in a few microseconds. Because hardware barriers are both cheap and fast—they are essentially AND-gates—we expect them to appear in most future parallel machines.[4] The operating systems for these machines must be able to exploit this hardware to efficiently execute fine-grain data-parallel codes. We believe this may prove easier with a synchronous microkernel structure, rather than a more traditional asynchronous kernel structure.

While the CMOST/executive structure supports timesharing among different execution models, a hierarchical control structure can integrate space-sharing as well. For example, a central scheduler on a 128-node machine can reserve some time-slices for 128-node synchronous applications and some for 128-node applications or sets of applications with more dynamic executive-managed scheduling. The remaining time-slices can be delegated to two other schedulers, each of which can recursively do identical centralized scheduling within disjoint 64-node processor groups.

In order for a single executive to manage multiple users' applications, the executive must be run with some additional privilege, e.g. as a Unix "setuid root" process, to access system resources with the effective permissions of the user on whose behalf the current application is being executed. Such an executive would also need a way to adjust its scheduling priority within the kernel so that processing resources can be fairly allocated across all user jobs, whether they are executing directly under the kernel or are one of several running under an executive.

6 Related Work

The interface described in this paper was motivated by the needs of the Wisconsin Wind Tunnel. The centralized structure of CMOST, with all policy enacted on the control processor, was insufficient to support the user-level fine-grain distributed shared memory needed by WWT. The executive interface extends CMOST to provide nodes with limited autonomy in the way they manage their virtual address spaces and physical memory. This interface is interesting from two different perspectives: on its own, as a means of exporting memory-management functions to the user of a uniprocessor; and in conjunction with CMOST, as a means of supporting multiple models of application parallelism on a single machine.

[4] The Cray T3D also has hardware support for global barriers; barriers are actually faster than remote memory operations on this machine [14].

6.1 Uniprocessor aspects

The executive interface provides a complete set of low-level virtual memory functions. The interface is simpler and lower-level than the virtual memory interfaces of either Mach [12] or Chorus [1], which both impose significant semantics on the use of memory. To the first order, the executive interface merely exposes the underlying hardware mechanisms to the user in a protected manner.

The executive interface is similar to the "inferior spheres of protection", described by Dennis and Van Horn [6]. Their execution model allowed processes to create subcontexts, initiate execution within them, and handle any resulting faults. The primary difference is our page orientation rather than their more general segments and capabilities.

More recently, Probert, et al, proposed SPACE, an object-oriented operating system [11]. SPACE allows applications to create, manipulate, and execute within *spaces*, i.e., address spaces, thereby facilitating protected objects. However, SPACE is much more general than our interface, allowing different "executives" to manage different parts of a single address space.

Appel and Li surveyed the most common uses of user-level virtual memory, and identified the set of primitives needed by these applications [3]. The set includes primitives to modify protection on pages and create aliases within an address space; however, they did not include the ability to create new address spaces, nor get callbacks when pages are reclaimed by the kernel.

Using page motion callbacks to manage physical memory allocation is analogous to using scheduler activations to manage physical processor allocation [2]. Both provide the user with notification of kernel allocation decisions so that the application can adapt knowledgeably to its new circumstance. A key difference is that the page_going() callback notifies the user *before* the page is taken away, while the scheduler activation model notifies the user *after* a processor has been taken away. This adds some complexity to the page motion callbacks (i.e. the necessity for the kernel to enforce a finite completion time), but reflects a fundamental difference between memory and processors: it is reasonable to have the user allocate space to save a processor's state in case the kernel takes it away, but nonsensical to apply the same principle to memory pages.

The Mach external memory manager interface is similar to our page motion callbacks. However, if an external memory manager does not remove a page in a timely fashion, the Mach kernel can always write the page to backing store using the default memory manager. Our interface does not permit this, because the kernel cannot clean up secondary mappings itself nor can it permit them to point into another process.[5]

6.2 Multiprocessor aspects

User control over multiprocessor scheduling with the intent of supporting multiple models of parallelism (including gang-scheduling) is provided in Mach by a processor allocation server [4]. In this model, an application requests a certain number of processors to create a "processor set" to which threads can be bound. The binding of actual processors to processor sets is performed by a privileged user-level server. The server can be modified to support site- or usage-specific policies, but there can only be one per platform. In our scheme, a process requiring a fixed number of processors can be handled simply by scheduling it at the appropriate level in the hierarchy. A process with more dynamic needs could be served by an appropriate executive that

[5]Our kernel could write the page to backing store, so long as it also saved and restored the memory tags and guaranteed to return it to the *original* physical page before resuming the process.

balances its requirements by scheduling it in conjunction with other applications having similar dynamic parallelism. The fundamental differences are that our model provides gang-scheduling more as the rule than the exception, and allows for multiple executives running simultaneously to support multiple abstractions.

The hierarchical integration of time- and space-sharing discussed in Section 5 is similar to Feitelson and Rudolph's distributed hierarchical control [7], except that they assume a hardware hierarchy of control processors, while we believe a software hierarchy of control processes may be just as effective. Also, their model does not have the executive interface to provide different scheduling models underneath the global hierarchical structure.

7 Conclusion

The executive interface exports a complete, abstract model of virtual memory management to a user process, including the ability to create, manipulate, and execute in multiple address spaces. The interface allows the user process to participate in physical memory management using page motion callbacks. The callbacks also serve to minimize the kernel complexity of implementing the interface.

Giving a user-level executive the ability to define a complete virtual memory environment in a protected fashion allows multiple executives providing multiple process abstractions to coexist on a single system. Though interesting from a uniprocessor perspective, it is more significant in the context of large-scale multiprocessors. Instead of having the operating system view the machine as a set of autonomous nodes upon which coordination mechanisms must be imposed, it can start with a global perspective and selectively delegate nodes in both space and time to executives which allow increasing amounts of autonomy. The resulting structure combines the advantages of centralization and decentralization: the underlying global perspective simplifies efficient support of fine-grained synchronous (e.g. data-parallel) applications and management of global resources such as barrier hardware, while executives provide the flexible support for other application models that a completely centralized system lacks.

Acknowledgements

Mark Hill, Frans Kaashoek, Jim Larus, Bart Miller, Yannis Schoinas, and Marv Solomon provided helpful comments that greatly improved this paper.

References

[1] Vadim Abrossimov and Marc Rossier. Generic Virtual Memory Management for Operating System Kernels. In *Proceedings of the Twelveth ACM Symposium on Operating System Principles (SOSP)*, pages 123–136, December 1989.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[3] Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 96–107, April 1991.

[4] David L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23(5):35–43, May 1990.

[5] M. D. Canon, D. H. Fritz, J. H. Howard, T. D. Howell, M. F. Mitoma, and J. Rodriguez-Rosell. A Virtual Machine Emulator for Performance Evaluation. *Communications of the ACM*, 23(2):71–80, February 1980.

[6] Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. In *ACM Programming Languages and Pragmatics Conference*, August 1965.

[7] Dror G. Feitelson and Larry Rudolph. Distributed Hierarchical Control for Parallel Processing. *IEEE Computer*, 23(5):65–77, May 1990.

[8] Robert P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, June 1974.

[9] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fifth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, July 1992.

[10] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[11] D. Probert, J. Bruno, and M. Karaorman. SPACE: A New Approach to Operating System Abstraction. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, October 1991.

[12] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William J. Bolosky, and Jonathan Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[13] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[14] S. L. Scott. Personal communication, June 1993.

[15] Burton J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *Proc. of the Int. Soc. for Opt. Engr.*, pages 241–248, 1982.

[16] Thinking Machines Corporation. *Connection Machine CM-5 Technical Summary*, October 1991.