Eric Viara
IRCAM: Institut de Recherche et Coordination Acoustique/Musique
31, rue Saint-Merri
F-75004 Paris, France
eric.viara@ircam.fr

CPOS: A Real-Time Operating System for the IRCAM Musical Workstation

The IRCAM Musical Workstation, or IMW (Lindemann et al. 1991), combines one or more NeXT computers with several real-time coprocessors (CPs), and includes an extensive software package that takes advantage of this multiprocessing environment. The part of the software that runs on the CPs requires a special operating system. Not only must the processors work together in real time, but they must pass dozens or hundreds of channels of digital sound among themselves at low latencies, as well as sporadic messages defining real-time control. The processors may also require access to real-time sound and MIDI inputs and outputs via a Motorola DSP56001 digital signal processor.

The Role of the Coprocessor Operating System

The IMW design breaks new ground on three different levels: its high-level user interface; its functional capabilities and performance; and its hardware architecture. Each of these levels puts certain constraints on the operating system design. The top level raises such issues as the control and specification of digital signal processing; the ease of development on dedicated boards (one needs a good software production environment including debugging); and the desire for transparency with respect to the hardware architecture. At the functional level, certain DSP algorithms that are recurrent on computer music applications (e.g., frequency modulation synthesis or digital filtering) need to be executed very efficiently, whether in real time or not. Their execution may depend on external events, and they might require the dynamic loading of object files not known to the software in advance. Finally, at the hardware level, the choice of processor and memory architecture has to be taken into account.

These requirements lead to a stringent set of specifications for the operating system. It is necessary to support real-time tasks (involving signals sampled at 44.1 kHz or faster) controlled by external events (hundreds of transactions per second); to minimize the response time to external interrupts; to take advantage of the hardware architecture's capability for parallel and pipelined computation; to furnish a set of debugging primitives for real-time tasks; to be as deterministic and controllable as possible; to offer transparency between host and coprocessor boards; to take advantage of such features of the hardware as local memories; and to offer compatibility with a reasonable subset of the UNIX operating system. We have found no existing operating system which meets these specifications completely, so we were obliged to develop our own, which is called the "CoProcessor Operating System" (CPOS).

Table 1 shows a breakdown of CPOS's functionality into five major areas: process management; memory management; interprocess communication (IPC); event handling; and file I/O. These functional areas are based on the following abstractions: *CPU, task, thread* (process management), *region* (memory management), *port, message, PFIFO, packet* (interprocess communication), *event* (the event handler), and *file* and *filesystem* (file I/O). Each of these will be described below.

The CPOS design permits kernel configuration at compile time by choosing among the functionalities listed above. The minimum kernel configuration consists of the process and memory management modules for a monoprocessor environment to which one may add the IPC module for a multiprocessor environment. The System V UNIX layer is implemented by kernel code written on top of the CPOS-specific layer. It is possible to suppress the UNIX interfacing code (at compile time) or

Computer Music Journal, Vol. 15, No. 3, Fall 1991,

Table 1. CPOS abstractions

Section	*Abstraction*
Process management	CPU
	Task
	Thread
Memory management	Region
Interprocess communication	Port
	Msg
	PFIFO
	Pocket
Event unit	Event
Filesystem unit	File
	Filesystem

to write a different operating system interface, or even to support the coexistence of several operating systems.

Process Management

Like the Mach and Chorus operating systems, CPOS introduces two abstractions related to process management: tasks and threads. A task is defined as an environment in which a program is executed, including a protected virtual memory space and communication ports to and from other environments. A thread is an execution context running within a task. A thread's virtual memory is shared with all other threads in the same task. A process is defined as a task in which at least one thread is running.

This separation of a UNIX-like process into two different abstractions offers the advantage that a thread can be created much faster than a UNIX process. Moreover, threads of a single task in a multiprocessor environment may run on different processors, sharing a single address space. In CPOS, the mapping of threads to processors is under the control of the program (but the kernel can choose the mapping automatically if the parent thread directs it to do so). When a thread is created, it is associated with a context belonging to its parent. This context includes values for all general- and special-purpose registers. Upon creation, a thread is in the "sleeping" state; execution starts upon receipt of an event (see below).

Every thread has a mode and a priority, which are initially inherited from the parent thread. The mode specifies the range of priorities attainable by the thread, a set of privileges, and a preemption policy. Two fundamental modes are defined by CPOS: lambda and real-time. A real-time thread may reach priority levels higher than those available to a lambda thread, and may even control kernel parameters such as clock frequency and interrupt masking.

External Interrupts, Scheduling, and Preemption

When the processor is interrupted by an external event, the interrupts are automatically cached by the processor. A status word, made up of the process's status register and certain I/O registers, is saved on the interrupt stack. Interrupts are then immediately re-enabled; only then is the current thread context saved. The few lines of code that save this status word comprise the only critical code section in the CPOS kernel.

If the action coupled with an interrupt is simple (i.e., does not try to access any global resource), the interrupt may be serviced immediately. If the associated action does require access to global resources, the interrupt is deferred as long as those resources are locked. A resource is generally not kept in the locked state for longer than necessary to traverse a short linked list.

CPOS scheduling policy differs for real-time and lambda mode threads. The handling of an interrupt can result in the selection of a higher priority thread than the current one; if the current thread is in real-time mode, it may be preempted only if an interrupt selects a higher-priority real-time mode thread. For lambda mode threads, the scheduling policy is non-real-time; at regular intervals of a few milliseconds, the kernel selects one of the runnable lambda-mode threads.

Memory Management

Memory management is based on the notion of a virtual region. A virtual region is a contiguous area of virtual memory, endowed with a set of attributes

and attached to a process. A virtual region is determined by the virtual address at the beginning of the memory area; this address is used to identify the region during kernel-user communication. This address remains constant throughout the life of the virtual region. The virtual regions attached to a process are disjoint.

Region Attributes

The attributes of a region are initialized at the time of its creation. Some of these attributes are static, that is, they may not be modified after creation; others are dynamic and may be modified at any time. Table 2 summarizes the CPOS memory region attributes. The simplest region attribute is the size; it is dynamic and its value ranges (theoretically) between 0 and 4 GBytes. The size of a region may be increased or decreased under program control.

Another region attribute is the type, which may be one of the constants, **`TEXT, DATA, STACK, DYNAMIC HEAP, SYSTEM`** or **`IO`**, specifying the way the region is used. In contrast with UNIX in which only the heap—corresponding to a **`DYNAMIC HEAP`** region in CPOS—is explicitly resizable, CPOS offers the possibility to allocate or modify the attributes of a **`TEXT, DATA`** or **`STACK`** region. The type attribute is static. The **`IO`** type is used for memory that may be shared between a CP and an I/O device. The physical mapping of an I/O region is fully controllable. (A similar feature is available on some UNIX operating systems through the "mmap" system call.)

When the region is created, the *shareable* attribute is initialized to either **`PRIVATE`** or **`PUBLIC`**. A **`PRIVATE`** region may only be shared between threads of one task whereas a **`PUBLIC`** region may be shared between threads of different tasks. **`PUBLIC`** regions are useful for interprocess communication, whether inside or outside of the IPC support provided by the CPOS kernel. Shareability is a static attribute; however, a task may attach or detach a **`PUBLIC`** region at any time. At least for the time being, a **`PUBLIC`** region is not protected; any thread can access it.

One of CPOS's distinctive features is that one can specify the physical sector on which a virtual region will be mapped. This is essential because of the asymmetrical architecture of IMW's local memories and the varying access times to those memories. The physical area of a virtual region may be specified as any of the constants **`LOCAL MEMORY`** (anywhere on local memory), **`BOARD MEMORY`** (anywhere on the board), and **`ANYWHERE`**. The *wired down* attribute is defined to specify whether or not physical memory pages associated with a virtual region may be swapped to secondary storage.

The *cacheable* attribute of a region is dynamic. A region may be cacheable or not; but all pages of a region must have the same cacheability. Modification of the cacheable attribute causes a global flush of the data and instruction caches and of the "Translation Lookaside Buffer."

A region may be associated with a file-offset couple. This allows direct memory mapping of all or part of a file. For instance, the text and data regions of a program correspond to areas in the executable file. Finally, the *protection* attribute specifies whether a region is readable, writable, or both.

A region is created by calling the **`regionAlloc`** system call, giving its initial attributes as arguments. Any dynamic attribute change (i.e., size, wired, cacheable, or protection) is specified using

Table 2. CPOS Memory Region attributes

Attributes	*Possible Values*	*Dynamic*
Size	0–4 GBytes	Yes
Type	TEXT, DATA, STACK, DYNAMIC HEAP, or IO	No
Shareable	PRIVATE or PUBLIC	No
Physical sector	LOCAL MEMORY, BOARD MEMORY, or ANYWHERE	No
Wired down	TRUE or FALSE	Yes
Cacheable	TRUE or FALSE	Yes
File/Offset	File Descriptor and Offset	No
Protection	READ ONLY, WRITE ONLY, or READ/WRITE	Yes

the `regionControl` system call. A task may attach or detach a `PUBLIC` region to or from its virtual memory region using the `regionAttach` or `regionDetach` system calls.

A precise diagnostic is given by the kernel in case of memory management system call failure, such as the constant `K_REGION NO_PHYSICAL` (if not enough physical pages were available to carry out a system call), or `K_REGION_UNKNOWN_PROT` (if an unknown protection attribute was specified). The programming interface for memory allocation is via the functions `MemAlloc` and `MemFree`, which implement a multimap allocator interfaced to the CPOS region system calls.

File System

CPOS file system management is built around two abstractions, *file* and *filesystem*, which are similar to their UNIX equivalents. As in UNIX, a file may represent either a collection of data or an interface to a kernel driver. A file system is a hierarchical collection of files.

File systems in CPOS are typed, since in the future the IMW might use specialized file systems adapted to real-time applications. Such a file system might permit speedy sequential access to sound files by storing them contiguously. A set of control functions would be provided to fill the special needs of a given file system; for instance, to pack the set of files to recover contiguous space in memory. At the time of this writing, CPOS does not support any file system; all file system requests are filled by host servers, using IPC communication between the CPOS kernel and the host.

A file is opened and closed by way of the CPOS system calls `fileOpen` and `fileClose`; the `fileRead` and `fileWrite` system calls provide sequential access to files; the `open`, `close`, `read` and `write` UNIX system calls are written in terms of CPOS system calls. The `fileControl` CPOS system call provides services as in the `lseek` and `fcntl` UNIX system calls. The `fstat` UNIX system call is written using `fileInfo`. Finally, `fileSystemControl` and `fileSystemInfo` offer all the necessary functionality to implement such UNIX system calls as `mount` and `ustat`.

Interprocess Communication

Interprocess communication (IPC) in CPOS takes two forms: formatted message-based communication using ports; and packed data communication using packed first-in, first-out queues (PFIFOs).

Communication Ports

A port is an object attached to a task that supports the exchange of formatted messages to or from other tasks. A formatted message is a typed segment of structured data of fixed length. If the message to be sent is too long, one may either split the message into submessages (an expensive and sometimes inadequate solution), or else include in the message body a reference to a larger data structure in a shared region of memory.

The thread that sends a message is referred to as its *emitter* and the thread receiving it as the *receiver*. When an emitter sends a message on a port, one of the threads waiting on this port will catch it; normally, it will be caught by the one with the highest priority. The message will be deleted from the message queue unless the receiver tells the kernel to keep the message in the queue. To date, no real protection is implemented; that is, any thread can communicate (either as the emitter or receiver) via any port for which the identifier is known. A task could even "eavesdrop" on one or more ports to keep a record of transactions between other tasks.

Messages may be received synchronously or asynchronously; if they are received asynchronously, the receiver is blocked only for the local kernel's processing time. To receive a message synchronously, the receiver is blocked until a message of the specified type arrives. A UNIX remote procedure call is carried out synchronously; the initiator is blocked until a response to the request arrives. A receiver may optionally specify a time-out value for the blocking.

At its creation a task is given a global port to which any thread may send messages. This port is intended mainly for setup communication to obtain a thread or port identifier or a `PUBLIC` region address. Ports may be shared between several tasks

running on different processors; their implementation uses a lock mechanism (test-and-set) to ensure data coherency. As several emitters and receivers may share a single port, message ports sometimes incur a heavy overhead in coherency control.

A port is created using the system call `portCreate`. The kernel returns an identifier that will be used for subsequent communications on this port. The system calls `msgSend` and `msgRcv` permit sending and receiving of messages; the function `msgRpc` permits sending of a message and (synchronous) receipt of a reply. Finally, `portDelete` deletes a port from a task.

PFIFO Communications

A PFIFO supports unidirectional transmission of packed data between two tasks either synchronously or asynchronously. A PFIFO sends untyped packets between tasks; a packet may be any collection of unstructured data. When an emitter sends a packet on a PFIFO, it specifies the packet's size—the receiver will receive either a whole packet or nothing. An attempt to send a packet on a PFIFO may result in an error if the PFIFO does not have enough buffer space to hold the packet. PFIFOs have a very simple and efficient implementation that does not require a lock mechanism, even if the emitting and receiving tasks are not on the same CPU. This communication strategy is intended for use between real-time tasks.

The attributes of a PFIFO include the size of its communication area, a task to which the PFIFO is attached, and the status of the owning task (emitter or receiver). A PFIFO is created by calling the `pfifoCreate` system call with arguments including the communication buffer size. Packets are sent and received using `packSend` and `pack Rcv`. PFIFOs are deleted using `pfifoDelete`.

Events

An event in CPOS is anything that can alter the chain of execution of a thread. External interrupts, instruction or data access faults, floating-point exceptions, system calls, and software signals (whether originating from the kernel or from another thread) are all considered events. CPOS permits an event to be associated dynamically with a function to execute within a thread. Except for software signals, this function is executed in the context of a thread in the kernel (i.e., a thread running in the kernel's virtual address space). A function associated with a hardware interrupt event acts as an interrupt handler; since it may be changed dynamically, CPOS processes many affect CPOS's own interrupt servicing.

Except for hardware interrupt handlers, all events are treated on a per-process basis; a task initially inherits the event handlers of its parent task. Associating a user function with a software signal is similar in effect to UNIX's signal facility. Treating system calls as events allows different processes to handle system calls differently; for instance, one process could use UNIX system calls while another used VMS.

In the case of a real-time process all of whose virtual memory is wired down, the page fault handler (called on data or instruction access faults), appears to the process as a software signal, which it treats as desired. A thread may wait for the arrival of an event; in this case only hardware interrupts and software signals may arrive; a sleeping thread cannot generate a data access fault or floating-point exception.

In CPOS, events are described by data structures maintained in a task's virtual space, and the kernel is handed a pointer to any event structure relevant to a system call. The `eventHandle` system call associates an event with a function to be executed in the user's context; `eventWait` causes the calling task to wait for an event; `eventSend` sends an event to a given thread; and finally, `eventInfo` gets information concerning the function associated with an event.

Kernel Control

CPOS provides dynamic control over some kernel parameters and kernel policy choices (of course, this feature must be used with care). For instance,

in a very critical section of code for real-time digital signal processing, a task might mask all external interrupts. No other processing of any kind would then take place in that processor; an infinite loop in this program section would effectively freeze the kernel, forcing a reboot.

An intermediate solution would be to mask clock ticks so that a higher-priority thread could still preempt the running thread in order to kill or debug it. As described in the section on process management above, scheduler policy choices can be dynamically controlled by changing mode and priority levels. Finally, user-supplied event detection functions may be used to "step inside" the kernel since the functions are executed in kernel mode in the kernel's virtual space.

UNIX Compatibility

CPOS provides partial compatibility with a subset of AT&T UNIX System V, both in its system calls and in its executable file ("COFF") format. The compatibility is not perfect; some UNIX system calls are not implemented, such as shared memory, semaphoressemxxx), and the UNIX System V IPC package. CPOS does provide alternatives to these system V calls. The CPOS `ptrace` system call is not fully compatible with UNIX since the CPOS and UNIX "Uareas" are slightly different. At present, about 40 UNIX system calls are compatibly implemented in CPOS; in practice, they ensure binary compatibility for the great majority of UNIX programs.

To date, we have tested UNIX compatibility at the source and object level. We have ported such UNIX commands as "cat," "grep," and "awk," as well as such well-known computer music programs as csound (Vercoe and Ellis 1990). All these programs ported easily and successfully. We have also tested binary compatibility for programs for which we only have executable files: for instance, the C software development environment provided by Intel including the Metaware C compiler, the Intel assembler and linker; and various other tools such as an archiver and a name list dumper. They all work.

All UNIX system calls are rewritten in terms of CPOS system calls; this usually requires only a few lines of code. This UNIX implementation is not as modular as in the Mach or Chorus kernels, which use their IPC packages to exchange requests between the kernel and their UNIX servers, which may be dynamically loaded. On the other hand, the UNIX interface in CPOS is a very small layer and is therefore very efficient. As in Mach and Chorus, subsystems can coexist in the operating system (UNIX and VMS, for instance); however, this is not a driving concern in CPOS development, and we will see, as CPOS evolves, whether this feature needs further development.

CPOS Architecture in the Context of IMW

CPOS consists mainly of a kernel running on the IMW's Intel i860 coprocessors. For performance reasons, copies of the kernel reside in each local memory. Servers running on the NeXT host handle file-related system calls from CPOS tasks. The host and the processors running CPOS communicate using a NeXT/Mach driver. This driver is fairly complex; the whole CPOS IPC package is included in it. This permits the host machine to act as a processor running CPOS. Any request from the CPOS kernel to the host is sent through a CPOS IPC message.

The CPOS system calls `taskCreate`, `threadCreate`, and so forth, are connected with the IPC package in a run-time library running on the host. Thus, CP tasks may be created and controlled from the host using the same code as from tasks running on the CP boards. This feature permits a high level of uniformity in the host/CP architecture. Figure 1 shows the structure of CPOS in the IMW environment.

Measurement and Estimation of CPOS Performance

A set of operating system benchmarks has been run in the IMW environment using the i860 processors. From these measurements, we can estimate performance for some other benchmarks. However, these

Fig. 1. The structure of CPOS.

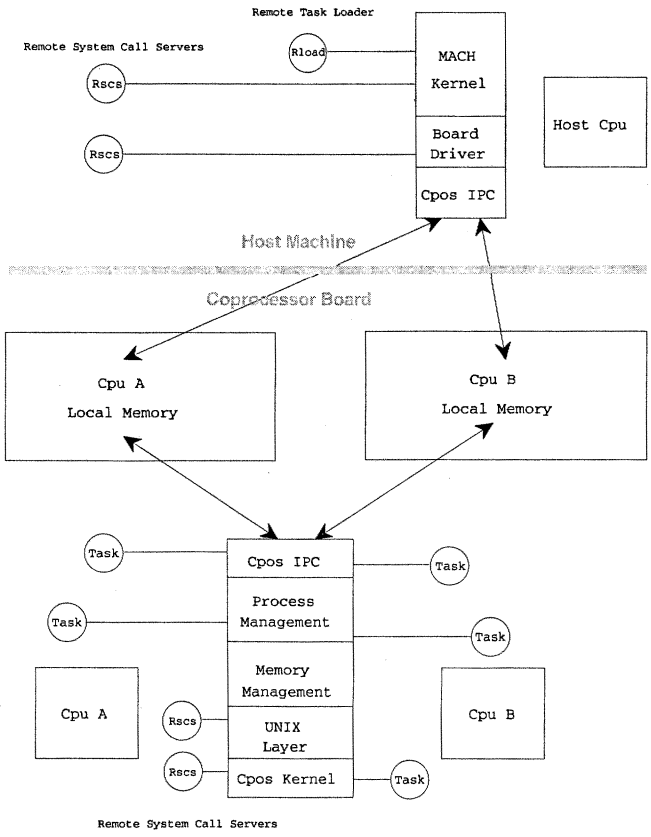

Table 3. Initial CPOS performance benchmarks

Operation	*Time Range* in Microseconds
External interrupt	(25, 40)
System call (getid())	(1.5,3)
Context switch	(40, 60)
Context switch in the same virtual space	(25, 40)
Duration of a critical section	(5, 15)

Table 4. CPOS kernel size for typical configurations

Kernel Configuration	*Kernel Size*
Process/memory management for a monoprocessor kernel	150 kBytes
Process/memory management and IPC for a multiprocessor kernel	180 kBytes
Process/memory management, IPC, filesystem, event unit, and UNIX interface for a multiprocessor kernel	240 kBytes

measurements give only a first approximation of the operating system's real performance; a tally of the measured and estimated minimal and maximal performance for each benchmark has yet to be made. Measurements were made for standard operations in the kernel such as context switching and external interrupt processing.

A context switch requires saving the context of the current thread, selecting a new thread, flushing the processor's caches, changing the memory mapping, and restoring the context of the new thread. Saving and restoring a thread's context on the i860 are time-consuming since the processor has 32 general-purpose and 32 floating-point registers; furthermore, one must save and restore the state of the floating-point and graphics pipelines, which requires several dozen cycles. Table 3 summarizes our CPOS performance measurements.

The kernel's size depends on the configuration. Table 4 shows the kernel's size for typical configurations such as the minimal configuration for a monoprocessor, including process and virtual memory management sections; minimal configuration for multiprocessors, including also the interprocess communication section; and finally, a standard multiprocessor configuration, including the event unit and partial UNIX System V compatibility.

Conclusion

CPOS was designed to support real-time processes for computer music as well as general-purpose programs such as compilers and debuggers. It provides

all the features offered by a standard operating system as well as real-time features such as its event unit. Protection policies are sometimes relaxed to give real-time processes more control over the kernel. Although CPOS has been designed to take particular advantage of the IRCAM Musical Workstation architecture, its design and implementation are modular enough that it could be ported to other environments.

References

Lindemann, E., et al. 1991. "The Architecture of the IRCAM Musical Workstation." *Computer Music Journal* (this issue).

Vercoe, B., and D. Ellis. 1990. "Real-Time CSOUND: Software Synthesis with Sensing and Control." In *Proceedings of the 1990 International Computer Music Conference.* San Francisco: Computer Music Association, pp. 209–211.