Selected components of tightly-coupled multiprocessing programming support are presented. Included are design rationale and reference to prior multiprocessing systems to give additional perspective.

Design of tightly-coupled multiprocessing programming

by J. S. Arnold, D. P. Casey, and R. H. McKinstry

One of the components of OS/VS2 Release 2 is the support of a tightly-coupled multiprocessing environment in which two Central Processing Units (CPUs) share a single main storage containing one copy of the operating system. This paper describes five aspects of the programming support of the multiprocessing (MP) hardware:

- Locking.
- Service management.
- CPU affinity.
- Dispatching.
- Alternate CPU recovery.

The locking section describes the serialization technique used to provide disablement across CPUs. It also describes how the system was subdivided to allow what were disabled, mutually exclusive functions to run in parallel on an MP system.

The service management section discusses a new set of system services that provide a new unit of dispatchability in the system. This unit of dispatchability has less overhead and better performance than tasks and is provided to encourage increased parallelism in system functions.

The CPU affinity section describes a way to force dispatching of work to a specific CPU. It provides a way of forcing an emulator job step which requires a particular hardware feature to the CPU having that feature.

The dispatching section describes the changes in the Dispatcher and related functions to support MP, multiple-address spaces, and the above three changes to the system.

The Alternate CPU Recovery (ACR) section discusses the process invoked when a CPU in a tightly-coupled MP environment can no longer function. ACR occurs as a result of a signal (either hardware or software generated) that is sent by the malfunctioning CPU before it enters a disabled wait or check-stop state. The hardware-generated signal, Malfunction Alert, occurs when continued instruction execution is impossible.[1,2] The software-generated signal, Emergency Signal, is issued by the Machine Check Handler if it determines that the CPU cannot function properly.

Locking

In a multiprogramming control program, it is necessary to serialize functions (in the control program) that reference the same data or control blocks. One OS/VS method of doing this is to disable the CPU so that no interrupts can occur to cause a switch from the current process. This provides the necessary serialization on a uniprocessor, but is not effective on a multiprocessor because disablement affects only one CPU. If a process executing on one CPU disables, it serializes only the one CPU; nothing prevents another process on the other CPU from also disabling and running in parallel with the first process.

The simplest solution to this problem is to provide a mechanism by which one CPU, when it enters the disabled state, locks out the other CPU from entering the disabled state. This can be done by defining a *lockword* that all processes check when they disable. If the lockword is zero, the process stores a CPU identifier into the lockword and continues. If the lockword is not zero, the processor spins on the lockword until it becomes zero. The process that holds the lock sets it to zero just before enabling. This provides serialization of two CPUs in a way equivalent to disabling. A technique similar to this was used in the System/360 Model 65 MP. Hardware support is needed for this function since it is necessary to perform the comparison and store into the lockword as a single operation. This prevents the other CPU from changing the lockword while the operation is being executed. If this is not done, both CPUs could compare against the lockword at the same time, and because both would find a zero value, both would store into the lockword and both would continue assuming they owned the lock. In the Model 65 MP, the Test and Set instruction was used for this purpose. In System/370 VS2 Release 2, the Compare and Swap instruction is used.

While this technique does provide the necessary serialization, it also introduces a system bottleneck. Because the serialization covers a large part of the control program, a significant percent-

age of the CPU power can be wasted if one CPU is executing the control program with the lock held and the other CPU is spinning on the lock waiting to enter the control program.

This bottleneck could be effectively eliminated by defining the system so that each item requiring serialization had its own lock. This would distribute the locking requests such that the probability of both CPUs contending for the same lock would be very small. However, since a conflict is possible, the locks must be maintained and this can result in significant overhead if there is a large number of locks.

In OS/VS2 Release 2, a compromise solution was used. The control program was analyzed and subdivided into components having a minimum of interaction. Locks were assigned to the components. This resulted generally in locks being assigned to collections of related control blocks instead of to each control block (some locks were assigned to specific control blocks). The locks defined are such that they should provide a significant reduction in wasted CPU time as compared to the single lock approach without an unacceptable level of overhead.

In analyzing the control program, one apparent split was into global and local components:

- The *local supervisor* which contained those functions in the local program logically associated with a particular user (such as Getmain for a user subpool, or Attach a subtask).
- The *global supervisor* which contained those functions in the control program logically associated with more than one user (such as Getmain for the System Queue Area, or system wide ENQ).

This split was heavily influenced by a decision to swap all users' address spaces, and to swap as much of the system control information about a user with the user. To do this, it was necessary to separate the control blocks and queues for each user into separate areas. Some queues, such as the Job Pack Area queue and the Getmain queues for user subpools, already were unique to a job and were easy to move. Others, such as the TCB queue, had to be split into separate queues for each user. Some, such as the ENQ queues, could not be split or swapped because their use was for system-wide serialization including swapped-out users. The queues that could be isolated by user were put into the Local System Queue Area (LSQA) and Scheduler Work Area (SWA) in the users' private area. The system-wide queues were put in the common area. The private area is addressable only to functions running in the user's address space with the user's segment table. The common area is addressable in every address space and with any segment table.

Because the private area is addressable to only the one user and because all the data and control blocks are unique to that user, any function that references only private-area information could, if coded reentrantly, run in parallel with itself in different address spaces without any serialization. These functions have been called the local supervisor. The remaining control program functions which provide system-wide services or use control information in the common area and must serialize across address spaces have been called the global supervisor.

local supervisor

The local supervisor functions do not have to serialize across address spaces, but must serialize with other executions in the same address space. This can occur due to multitasking in the address space. To provide the serialization, a *local lock* was defined. One local lock exists for each address space and is used by local supervisor functions to serialize with other local supervisor functions in the same address space. Local supervisor functions can be active and own local locks in other address spaces without serializing with the current address space.

The local supervisor functions request the local lock, instead of disabling, in order to serialize. When the local lock is held in an address space, the dispatcher does not dispatch any other TCB in the address space. If another CPU is already executing work in the address space when the lock is obtained, it continues to run until it is interrupted and switched away from, or until it asks for the local lock. At that point it is suspended and the CPU is dispatched to another address space.

The local supervisor is logically disabled by the local lock; that is, no local processing can take place until the local lock is released. Therefore, local supervisor functions can be dispatched in a hardware-enabled state and still be logically disabled. To support this, a save area was provided into which status could be saved if an interrupt occurred. The availability of this save area also allowed the system to switch from a local supervisor function to another higher-priority user. Because of these changes, a number of control program functions that formerly ran disabled, now run enabled. This increase in enabled code allows greater responsiveness to interrupts and to high-priority functions.

The local lock is defined as an *enabled suspend lock*. That is, the holder of the lock executes enabled and, if another process requests the lock while it is held, the requester is suspended and the CPU dispatched to another process. The suspended process is not redispatched until the lock is released.

While additional local locks could have been defined to further subdivide the local supervisor, it did not seem to be necessary

because the (1) local lock-holder ran enabled, (2) any other request for the lock was suspended and did not tie up the CPU, and (3) the local supervisor functions could continue to execute in other address spaces. We felt, however, that the split of the system into a multiple local lock supervisor and a single lock global supervisor would not sufficiently reduce wasted CPU spin time, particularly since the most frequently executed disabled functions in past systems were lumped into the global supervisor.

global supervisor

In a first pass, the global supervisor was split into four functional areas—chosen because the control blocks and queues used by the areas were, for the most part, independent of each other. A lock was defined for each area:

- *Dispatcher lock*—used by all functions associated with dispatching and/or changing the queues, and control blocks used in dispatching.
- *Storage Management lock*—used by the real and virtual storage allocation functions, the paging supervisor, the auxiliary storage manager, and any functions that must serialize with these functions to reference their queues.
- *I/O Supervisor* (IOS) *lock*—used to serialize references to the IOS control blocks and queues.
- *Miscellaneous lock*—used by all other parts of the global supervisor.

These locks were defined as *disabled spin locks*; that is, the CPU that holds the lock executes disabled and, if the other CPU requests the lock, it spins in a disabled state until the lock is available. However, when one lock is held by a CPU, any other lock can be held by the other CPU. It was felt that the four locks would provide sufficient parallelism to reduce the wasted lock-spin time to an acceptable level.

Since the first pass, functions have been redesigned or new functions added to the system in such a way that it was convenient to define additional locks, thereby reducing the probability of contention for a lock without significantly increasing the number of lock requests. The additional locks result from the splitting of the storage management lock and IOS lock into multiple locks, and the addition of a set of locks for VTAM and a lock for the System Resource Manager.

In addition, the miscellaneous lock, renamed the Cross Memory Services (CMS) lock, has been redefined as an enabled suspend lock. As with the local lock, the owner of the CMS lock runs enabled and can be interrupted and/or switched away from to run higher-priority work. If there is another request for the lock

while it is held, the requestor is suspended and other work is dispatched. This enabled global lock has been provided for two reasons:

- Disabled page faults are not allowed in the system. We felt that some global functions could use a lock that did not require them to fix all their code and control blocks.
- Some functions required significant amounts of time under the lock and could impact the responsiveness of the system. By running these functions logically disabled under the lock, responsiveness was retained at the expense of some increased contention for the lock.

The other locks were left as disabled spin locks because normally the functions that run under the locks are of short duration. In addition, the cost in system overhead to perform the necessary status saving to accept interrupts and allow switching would offset the gain in responsiveness. Also, the more frequently used functions (that is, the IOS interrupt handler, dispatcher, and storage manager) are needed to perform interrupt stacking and task switching, and therefore would have to remain disabled.

current locks

The result of these changes is the following set of locks:

- The *Dispatcher (DISP) Lock*, a disabled spin lock, is used to serialize all funcitons associated with the dispatcher queues. It is also used for a number of miscellaneous functions that did not fit under the other locks and could not use the CMS lock.
- The *Auxiliary Storage Management (ASM) Lock*, a disabled spin lock, is used by ASM functions for global serialization.
- The *Space Allocation (SALLOC) Lock*, a disabled spin lock, is used to serialize Real Storage Management and the global portions of Virtual Storage Management.
- The *IOS Synchronization (IOSYNCH) Lock*, a disabled spin lock, is used to serialize the IOS purge function and other parts of IOS.
- The *IOS Channel Availability Table (IOSCAT) Lock* is used to serialize the selection of a channel by IOS.
- A set of locks for the *IOS Unit Control Blocks (IOSUCB)* is used by IOS to serialize the changing of status in the UCBs. There is one disabled spin lock per UCB.
- A set of locks for the *IOS Logical Channel Queues (IOSLCH)* is used by IOS to serialize access and updates to the logical channel queues. There is one disabled spin lock per logical channel queue.
- A set of locks for the *VTAM Node Control Blocks (TPNCB)* is

used by VTAM when scheduling work via node control blocks. There is one disabled spin lock per node control block.

- A set of locks for *VTAM Destination Node Control Blocks (TPDNCB)* is used to schedule work in VTAM. There is one disabled spin lock for each destination node control block.
- A set of locks for *VTAM Access Method Control Block DEBs (TPACBDEB)* is used by VTAM to serialize feedback processing. There is one disabled spin lock per ACBDEB.
- The *System Resource Manager (SRM) Lock*, a disabled spin lock, is used to serialize the various SRM functions when they are updating their control blocks.
- The *Cross Memory Services (CMS) Lock*, a global-enabled suspend lock, is used by any global system functions that can or must run enabled but need serialization. A local lock must be held while the CMS lock is held.
- A set of locks for the *local supervisor (LOCAL)*, each a local-enabled suspend lock is used to serialize functions in a single address space. There is one lock per address space.

locking rules

Even with the defined set of locks, many control program services must obtain multiple locks to perform their function. Therefore, it was necessary to define a locking hierarchy to prevent interlocks. The locks are arranged from high to low in the order just presented. IOSUCB, IOSLCH, TPNCB, TPDNCB, TPACBDEB, and LOCAL comprise a set of locks hereafter called *class locks*. The following is a set of rules for operations on the hierarchy (a CPU is considered the owner of a lock):

- A CPU can hold only one lock of a given class lock set. Two CPUs can hold two different locks in the same class lock set.
- A CPU may only request locks higher in the hierarchy than locks already held.
- It is not necessary to obtain all locks in the hierarchy up to the highest lock needed. Only the needed locks have to be obtained, but they must be in hierarchy sequence.

For the LOCAL and CMS locks, it is possible for the CPU to switch away from the process requesting the lock. In this case, the lock is placed in an interrupted or suspended state, and ownership of the lock is removed from the CPU. The CPU can then obtain other local locks for other processes. Requests for a lock that has been interrupted or suspended are handled the same as if the lock were held. When the process owning the lock is resumed, it can be run on either CPU, not just the one on which it was suspended. When resuming the process, the dispatcher gives ownership of the lock to the CPU that will be running the process.

When a local lock is held or suspended, the dispatcher will not dispatch any tasks in an address space in which the local lock is held except the one holding the local lock. Because of this, when the CMS lock is held, a local lock must also be held to prevent an interlock from occurring.

The interlock would occur as follows. Tasks A and B are in the same address space. Task A gets the CMS lock and is interrupted. Task B is then dispatched, gets the local lock, and then asks for the CMS lock. The tasks can be running sequentially on the same or different CPUs, or in parallel on two CPUs. In the latter case, Task A must have obtained the CMS lock before Task B asked for it, and must be interrupted after Task B has received the local lock and before Task A has released the CMS lock. Task B, because the CMS lock is held when it asks for it, is suspended until the lock is available. Task A, because the local lock is held by Task B, is not redispatched by the dispatcher following the interrupt. Therefore, Tasks A and B are interlocked, each waiting for the other to release the lock it owns. By requiring that the local lock be held when the CMS lock is held, the interlock is avoided because the owner of the CMS lock cannot be made nondispatchable by another task getting the local lock.

The lock hierarchy is based on the expected nesting of system functions and, therefore, the expected sequence in which the locks will be requested. There are some system functions that will be nested in such a way that obtaining the locks in the nesting sequence would result in a hierarchy violation. In these cases, the routine needing the higher-level lock must first obtain all lower locks needed by subsequent routines.

The interface used to obtain and release locks is provided through a macro (SETLOCK) that can be used only by supervisor mode, key 0 functions because it generates a branch to the Lock Routine which uses privileged instructions and protected storage.[3]

locking and ACR

The Lock Routine performs two functions in support of Alternate CPU Recovery (ACR). First, it maintains a bit mask on a per-CPU basis indicating which locks are held by the CPU. When a lock is obtained by a CPU, a specific bit identifying that lock is turned on in the bit mask. If a CPU failure occurs, ACR running on the functioning CPU will use the bit mask for the failing CPU to determine how to switch between the work for the functioning CPU and the work for the failing CPU until all the locks for this CPU are freed. This allows it to avoid interlocks on the locks while it is invoking recovery routines for the failing CPU's work. This bit mask is also used as a validity check for hierarchy violations. The bits for each type of lock are ordered in the bit mask according to the hierarchy. The Lock Routine using a second mask containing a bit in the position for the requested lock type

compares the two masks. If the CPU-bit mask has a higher value, a hierarchy violation is indicated. This check was provided as a debugging tool so that system errors in the use of locks could be easily found. It also allows the system to recognize deadlocks before they occur and, by invoking recovery routines, allows the system to continue operation.

The second function performed to support ACR occurs in the disabled spin for a global spin lock. In this case, one CPU is spinning waiting for the other CPU to release the lock. If the other CPU should fail such that the lock cannot be released, an interlock would occur. To prevent this, the Lock Routine will enable in its disabled spin path for Malfunction Alert (MFA) and Emergency Signal (EMS) interrupts.

These interrupts are generated by the failing CPU either by the hardware (MFA interrupt) or by the software (EMS interrupt) depending on which one decides the CPU cannot continue. By accepting the interrupt, the running CPU can break out of the disabled spin routine and give control to ACR. ACR can then recover the failing CPU's work and, using the lock-held bit masks, avoid the interlock situation.

Service management

Service Management is a new set of primitive functions provided in OS/VS2 Release 2. This basic set of services allows internal system components to structure themselves to run enabled, non-serialized, and in parallel on a multiprocessing system with less overhead than would be required by the utilization of existing task management services. The main facilities of this support, transparent to all problem-program tasks and available only to key 0 system services, are:

- A control block, called a *Service Request Block* (SRB), defined to represent a service request. This block, like a TCB, identifies a unit of work to the dispatcher. It is, however, significantly smaller and requires less information to be initialized for each request.
- A simple macro service, called SCHEDULE, which enters service requests into the queue of dispatchable work with a minimum of overhead.[3]
- Changes to the Dispatcher to operate from a new service request control structure in addition to the task structure. The changes are optimized to provide maximum performance when dispatching service requests while providing the ability to schedule the SRBs to different address spaces and at a priority either independent of and higher than the priority of the address space or at the priority of the address space.

The Service Management feature resulted from an effort to move OS, which is a general-purpose, highly functional system, into the high-performance terminal-oriented environment. Service Management is an attempt to provide a dispatching facility in the system that could be used by system services and application programs to better utilize the two CPUs in an MP environment. The application programs of concern are those which, even though having independently dispatchable units of work, found it necessary to run as a single OS task and to provide their own dispatching structure because of the overhead of the OS task structure and OS services. These dispatchers were tailored to the specific application and could, therefore, be fast and have small storage requirements compared to OS services. Because of the tailoring, however, each one was a unique implementation. The Service Management features are an attempt to provide a service in the system that is small and fast enough to be used by these types of subsystems. This service is only used to a limited extent by the subsystems available with OS/VS2 Release 2.

The Service Management features provide a solution to two problems that exist because of special dispatchers. Both of these problems exist in the case where a subsystem is implemented such that there is a single OS task under which the subsystem dispatches its own units of work. Because the operating system is aware of only the one task and not the subsystem's units of work, it cannot dispatch the subsystem to more than one CPU at a time in an MP system. With the Service Management features, the system is made aware of the subsystem's units of work and can dispatch them in parallel on multiple CPUs.

The second problem occurs when two or more subsystems run on the same system. Again because of the single task, both the high- and low-priority work of one subsystem will run before the second subsystem. Unless the subsystems are designed to cooperate by voluntarily releasing time at specific intervals, the second subsystem's ability to meet its high-priority response requirements could be impacted. With the Service Management features, it is possible to run the high-priority work of both subsystems at a priority independent of and higher than the priority of the subsystem jobs. The low-priority work of the subsystem would continue to compete on a job priority basis. This capability should improve the ability of two or more subsystems to exist in the same system with acceptable performance in all subsystems.

In addition to the subsystem reasons for developing the Service Management features, it turned out to be very useful in the system. It provided a mechanism used for almost all communica-

tions between address spaces, and it was used to run some parts of interrupt handlers as service requests, allowing more enablement and parallelism for these services.

An example of the usage is the IOS interrupt handler. When an interrupt occurs, the interrupt handler collects the necessary information about the interrupt and schedules an SRB. The interrupt handler can then start any I/O request waiting for the I/O path and accept any additional pending interrupts. By delaying complete processing of the interrupt, this approach allows faster reuse of channels and lower disabled interrupt time. The scheduling of the SRB provides the ability (1) to complete the interrupt process on any CPU, and not just the one that took the interruption, (2) to process the interrupt enabled except where specific serialization through locks is used, and (3) to switch from the random address space where the interrupt was taken to the address space of the user originally requesting the I/O. This latter capability provides the interrupt handler routine with the addressability to the user's control blocks that is necessary to complete the interrupt processing.

service request control structure

The basic control structure utilized by the Service Management features incorporates two levels of system priority: global and local. Service requests queued at the global level are given a priority above that of any address space, regardless of the actual address space in which they will be dispatched. Service requests queued at the local level are given a priority equal to that of the address space in which they will be dispatched but higher than that of any task within that address space.

At each level there exists a *Service Priority List* (*SPL*). This list is a static, contiguous list of queue anchors and simply serves as a mechanism for allowing prioritization among the various types of service requests that may exist. Each element of the SPL serves as an anchor for a queue of service requests, and the dispatching algorithm is such that it starts at the top of an SPL and takes any request queued at the first element prior to looking for a request queued at a lower element. Thus, the SPL is effectively a list of priority levels, with a single global SPL for the system and one local SPL per address space.

There are two levels in each SPL. One level is for general system usage, and the other has a nonquiescable attribute and is restricted to functions requiring this attribute. Currently, this second level is restricted to SRBs that are suspended and rescheduled, SRBs scheduled to resolve page faults, and SRBs scheduled to initialize a new address space. All service requests from a single SPL level are defined to have equal priority. No assumptions can be made with respect to the actual order in which they are dispatched. When the Dispatcher selects a service request to dis-

patch, it removes the request from the queue. Thus this structure has only ready, dispatchable service requests queued. The SRBs representing service requests are fixed in real storage and are addressable from any address space. These control blocks are not owned by the Dispatcher, but are supplied by the function requesting a service and may be freed or reused as soon as they are dispatched. The SRBs may be in any system key and are not modified by the Dispatcher except for queuing.

One level in each SPL is defined as nonquiescable, but this level has very restricted usage. It is needed because at times it is necessary to stop the dispatching of SRBs in an address space (for example, when an address space is to be swapped out or when it is terminating). The definition of stopping SRBs is to prevent dispatching of new SRBs and to allow completion of SRBs already dispatched. Because SRBs can be suspended due to lock requests or page faults and because page fault processing and rescheduling of suspended SRBs make use of SRBs, it is necessary to have the nonquiescable level at which these SRBs can be scheduled and dispatched while the other SRBs are stopped.

For implementation reasons, two additional queues, called *Local* and *Global Service Manager Queues* (*LSMQ* and *GSMQ*), are used. These queues were introduced so that the locking requirements associated with dispatching SRBs can be limited to the Dispatcher and so that the SCHEDULE service that introduces service requests to the system can run unlocked. The SCHEDULE service requires that the invoker supply a previously obtained and initialized SRB to represent the request until it is actually dispatched, and to supply the priority of the request (either global or local). The schedule routine queues the SRB to the appropriate service manager queue (LSMQ or GSMQ) and returns to the invoker. On the next entry to the Dispatcher, the presence of the SRBs is detected and the Dispatcher moves any SRBs that may have accumulated on the SMQs since the last entry to the dispatcher to the appropriate SPLs. If there were any global SRBs, the Dispatcher then dispatches one. Any local SRBs are placed on the local SPL and are dispatched when that address space becomes the highest-priority address space with ready work. The information needed by the Dispatcher to dispatch a service request is contained in the SRB, provided by the invoker of SCHEDULE. It includes the address space that the service request is to be dispatched in, the entry point of the routine to receive control, the protect key it is to run in, the SPL level the SRB is to be queued to, and a parameter to be passed in register one on entry to the Service Request Routine.

characteristics of service routines

Service Request Routines have some restrictions and operating characteristics that are different from task-oriented routines. These are:

- They are entered in supervisor state, enabled, unlocked and in relocate mode; they must return to the address provided on entry in register 14 in the same condition because the SRB exit is a direct entry to the Dispatcher and no cleanup or status restoring is performed at this entry.
- The service routine is responsible for freeing the SRB or making it available for reuse since once the SRB is dispatched, the Dispatcher no longer keeps track of its existence.
- Service routines cannot issue SVCs, but instead may use branch entries to system functions provided that the functions do not have an implicit TCB requirement; that is, that the function does not assume the caller is executing under a TCB and, therefore, uses the current TCB pointer to get addressability. When a service routine is running, the current TCB pointer is invalid.
- Service routines may lose control because of a page fault or because of an unconditional request for a suspend type lock (CMS or LOCAL) that is currently held. In both of these cases, the full status of the process is saved and other work is dispatched. When the page fault is resolved or the lock is available, the service routine is made eligible for redispatching by scheduling a special SRB containing the saved status at the nonquiescable level of the appropriate local SPL. The Page Fault Handler and the Lock Routine obtain the special SRBs and save the status. When it finds the special SRB, the Dispatcher restores the status, frees the SRB, and resumes the service routine at the point of suspension.
- Except for the noted suspension cases, service routines are non-preemptable. Thus, even though they run enabled and may be interrupted by asynchronous interrupts, they will not be switched away from until they voluntarily give up control. Interrupts that occur are processed but any dispatchable units of work made ready by the interrupt processing are ignored and control returned directly to the service routine.

This last characteristic of service processing is provided because these routines are expected to be relatively short in duration and, therefore, it is preferable to run them to completion instead of going through the overhead of satus saving and restoring, and task switching just to run a higher priority unit of work. This choice also eliminates the need to retain a status save-area control block in anticipation of a task switch during a service routine's execution.

STOP SRB

A STOP SRB function is provided in the system as part of the STATUS service. This function is intended primarily to support the quiescing and swapping of address spaces although it is also used for other reasons. As mentioned when discussing the nonquiescable SPL level, service requests are stopped by preventing the dispatching of new SRBs and allowing all service routines

already dispatched, including suspended SRBs that will be redispatched, to complete. While SRBs are stopped, all scheduled SRBs are moved to the local SPL. If the SRB had been scheduled on the global SPL, it is moved to the corresponding priority level on the appropriate local SPL and, when SRBs are restarted, it is dispatched as if it had originally been scheduled as a local request. If the address space is swapped out, the Dispatcher, when it puts the SRBs on the local SPL, notifies the System Resource Manager of the existence of work for the address space. The System Resource Manager, based on the workload in the system, subsequently causes the address space to be swapped in and reactivated.

CPU affinity

In a multiple-processor environment, certain hardware features may not be available on both CPUs. In these environments, the system must be directed to run those programs requiring a certain feature only on the CPU with the feature installed. This capability in OS/VS2 Release 2, called *CPU affinity*, has two defined forms: *SRB affinity* and *task affinity*. Although both have the same effect of causing a unit of work indicated by a TCB or SRB to be dispatched to a specific CPU, they differ in the way the affinity requirements are determined.

The dispatching part of the process works the same for both types of affinity. When the Dispatcher is entered, it selects a TCB or SRB to dispatch. It then checks a bit mask in the TCB/SRB to see if the TCB/SRB can run on the CPU currently running the Dispatcher. If it can, the dispatcher dispatches the unit of work; if not, the Dispatcher leaves the work for the other CPU to pick up when it enters the Dispatcher and searches for another TCB or SRB to dispatch.

SRB affinity is strictly an internal function since the system function that SCHEDULEs the SRB determines the affinity requirements, if any, and indicates them by setting the appropriate bits in the SRB affinity field. Task affinity is a misnomer because it is really job-step affinity; that is, the affinity requirement is determined by the job scheduler when the job step is started and the affinity is propagated to any subtask attached by the job step. Task affinity is provided to support emulator job steps in an MP environment when the emulator hardware feature is installed on only one of the CPUs.

The control information is found in the Program Properties Table (PPT) which has been extended to contain entries that relate program names to a bit mask indicating which CPUs have the feature installed.[3]

The PPT is a job-scheduler module and now contains the names and symbols for OS/VS2 Release 2-supported emulators. These are obtained at SYSGEN time from information supplied in the AFFINITY macro.[4]

This table is used by the Initiator to determine if affinity is required by a job step and if at least one of the required CPUs is online. The Initiator searches the PPT for the program name specified on the EXEC statement. If found, the associated bit mask is ANDed with the CCA field of the online CPU. The result is the affinity requirement—that is, a bit is on for each online CPU with the feature installed. If non-zero, the mask is saved and the ATTACH routine propagates it into the job-step TCB and any subsequent subtask TCBs. The affinity field is reset when the step terminates. If the result is zero, there is no online CPU with the needed feature. If this occurs for the first step of a job, the job is put on the hold queue and can be released by the operator when the needed CPU becomes available. If the step is not the first in the job, the job is aborted.

Dispatching

The main function of the Dispatcher in OS/VS2 Release 2 (as it was in previous releases) is to select and give control to the highest-priority dispatchable unit of work.[5] However, the implementation has changed significantly. The major changes are reflected in the new control structure the Dispatcher must use. Whereas in previous releases the Dispatcher worked from a TCB queue containing all the TCBs currently defined in the system and only had to find the highest-ready TCB on the queue, in Release 2 the Dispatcher has to work with six different queues:

- The Global Service Manager Queue (GSMQ).
- The Global Service Priority List (GSPL).
- The Local Service Manager Queue (LSMQ).
- The Address Space Control Block (ASCB) queue.
- A Local Service Priority List (LSPL) per address space.
- A TCB queue per address space.

In addition, the Dispatcher must recognize a request for CPU affinity, a redispatch of a suspended SRB, and a redispatch of an interrupted or suspended local supervisor. Also it must keep track of what is dispatched on each CPU to prevent the dispatching of the same process on two CPUs or the loss of a process that could run.

The GSMQ, LSMQ, GSPL, LSPL and SRB queues are a result of the Service Manager services that defined a new non-task-dispatchable unit represented by an SRB. The ASCB queue and the TCB

queue per address space are due to the local/global supervisor split. The ASCB queue is a global queue identifying the address spaces currently in storage. The TCB queue is a local queue in each address space and contains all the currently defined TCBs in the address space. Another queue affected by the local supervisor is the asynchronous exit queue, which is split into a queue per address space.

Dispatcher entry points

Six new entry points in support of the Service Manager, locking, MP, and paging extensions to the control program have also been added to the Dispatcher. The normal entry point continues to be used when a task has been interrupted or has caused a task switch. On this entry, the Dispatcher checks if any higher-priority dispatchable unit (either a TCB or SRB) has been made ready. If not, and the current task is still ready, the Dispatcher redispatches it. If the current task is not ready or higher-priority work is available, the Dispatcher saves the status of the current task, searches the queues for other work, and dispatches it. The status saving can be handled in one of two ways, depending on whether or not the local lock was held by the task when it was interrupted. If the local lock is not held, normal status saving is done; that is, job-step timing is performed, task timing (if any) is stopped, and floating point registers are saved. (The general registers and the PSW were saved before the Dispatcher was entered.) In addition, the Dispatcher:

- Clears the TCB NEW/OLD fields used to indicate which task is active on a CPU. A set of these fields exists for each CPU in the PSA of the CPU. System services running under a task determine the current TCB address from these fields.
- Clears the CPUID field and TCB active bit in the TCB. These fields are used to prevent the dispatching of a task on two CPUs at the same time.
- Decrements the count of CPUs running tasks in the address space. This is a field in the ASCB used in conjunction with another ASCB field containing a count of ready TCBs to determine if the Dispatcher should search the TCB queue of an address space for a ready task not dispatched on another CPU. The ready TCB counter saves a TCB queue search if there are no ready tasks. The pair of counters saves a search if there are ready TCBs but they are active on the other CPU.

If the local lock is held by the task, the same status saving has to be done. However, it is saved in a special save area, called the Interrupt Handler Save Area (IHSA). There is one of these per address space. (Only one is needed because there can only be one holder of the local lock at a time in an address space.) In addition to the normal status, the Dispatcher also has to save the NEW/OLD fields and the Functional Recovery Routine (FRR) stack in the IHSA and must change the ownership of the lock. The

FRR stack contains information about error recovery routines to be given control if an error occurs while the lock is held.[2] This information is specified and deleted dynamically by the routines running with the lock held. The ownership of the lock is changed by storing an interrupt ID into the lockword and zeroing the local lock bit in the CPUs lock-held bit mask. These changes leave the local lock held, but make the current CPU no longer the owner. This allows the CPU to be dispatched to another address space and to get another local lock.

The other entries to the Dispatcher all indicate that the process which was running has either completed or cannot continue, and that status saving different from the normal entry is needed. Because the current process cannot be redispatched, the status saving is done immediately and before the search for other ready work is performed. Two of these entries are from the Lock Routine and two are from the Page Fault Handler indicating that the current process is suspended. The other two entries indicate that the current process is complete. One of each type is for TCBs; the other for SRBs.

The lock entry for TCBs is used when the current task that owns the local lock is suspended while requesting the CMS lock. (The local lock must be held when the CMS lock is requested.) The status saving at this entry is the same as the normal entry except that the Lock Routine has placed a suspend identification in the local lock word and the Dispatcher does not have to put an interrupt identification in it.

The lock entry for SRBs is used when an SRB routine is suspended while requesting either the local or the CMS lock. In this case, the Lock Routine obtains an SRB with a save area and saves all the necessary status including updating the local lock ownership if necessary. The Dispatcher only does the job-step timing for the SRB's address space and resets the SRB mode indicator. There is an SRB mode indicator for each CPU to show that an SRB routine is active on the CPU. The primary users of the indicator are the interrupt handlers. They normally pass control to the Dispatcher; but if the SRB mode switch is on, the interrupt handlers return control directly to the interrupted SRB routine.

The Lock Routine uses the normal entry to the Dispatcher when suspending a task requesting the local lock. The task is not placed in wait state, but the Dispatcher will not redispatch it as long as the local lock is held.

The entries from the page-fault suspend routine are used when a task or SRB routine takes a page fault that cannot be satisfied by a page in main storage. The status saving at these entries is the

same as at the corresponding Lock Routine entries. The only difference is that the Dispatcher lock must be obtained at these entires while it is provided as input by the Lock Routine.

The task-termination entry point is used by the End of Task (EOT) routine when it has completed deleting a task. In this case, the task has been removed from the TCB ready queue and there is no status to save. The Dispatcher does, however, perform job-step timing calculations and resets control information by zeroing the NEW and OLD pointers in the Prefix Save Area and decrementing the count of the number of CPUs in the address space.

The SRB termination entry point is the return point from SRB routines. Again there is no status saving needed, but the Dispatcher performs job-step timing calculations and turns off the SRB mode indicator.

Memory Switch Routine

The Memory Switch Routine provides the Dispatcher with an indicator that work is ready. (This routine is called by other system functions that make work ready.) For SRBs, the Dispatcher calls the Memory Switch Routine whenever it moves an SRB to a local SPL. For TCBs, a number of different functions (such as POST and STATUS) call the Memory Switch Routine when they make a task ready.

The Memory Switch Routine makes use of two fields per CPU. One of these fields (PSAAOLD) indicates the current address space active on a CPU. The other (PSAANEW) indicates the highest-priority address space with ready work except when a global SRB routine is dispatched. In this case, it contains the highest ready address space excluding the address space in which the SRB routine is running (the SRB routine can be either higher or lower priority).

When the Dispatcher dispatches a work unit in an address space, it stores the address of the ASCB into PSAAOLD and, if it is not a global SRB being dispatched, into PSAANEW. If, while the unit of work is executing, the Memory Switch Routine is called because other work is made ready, the Memory Switch Routine checks the priority of the address space in which the new ready work will run. If it is higher than or equal to the priority of the address space pointed to by PSAANEW, the new ASCB address is stored into PSAANEW. If it is lower, the value is unchanged. The Dispatcher uses the fields to determine if a switch from the current address spaces is needed. When searching for an address space to dispatch, the Dispatcher starts at the one pointed to by PSAANEW.

In an MP system, the Memory Switch Routine checks the priority of the address space in which new work was made ready against the PSAANEW field for each CPU and updates the field for the CPU with the lowest priority address space in PSAANEW. In addition, it causes an interrupt to occur on the CPU that had the field updated using the Signal Processor (SIGP) instruction. The interrupt is taken as soon as the other CPU enables, and causes an entry to the Dispatcher. The reason the Dispatcher entry is forced, rather than depending on normal task switching activity, is to avoid the condition where the CPU is in wait state and there are no outstanding interrupts for the CPU. If the SIGP were not issued, the CPU would never come out of wait state. Another reason for the forced entry to the Dispatcher is to provide better responsiveness to higher-priority work.

work selection

The Dispatcher makes use of the queues and control information to select work to run. Work is selected based on position in the queues. Position on the SPL queues is random. Position on the ASCB and TCB queues is determined by priority when the control blocks are put on the queues.

The Dispatcher selects work in the following sequence: global SRBs, highest-priority address space, local SRBs, interrupted local supervisor, and TCBs. A global SRB is selected from the GSPL queue and is dispatched if it is not prevented from running on the current CPU by CPU affinity specifications and if the address space in which it is to run has not had SRBs stopped.

If there are no dispatchable global SRBs, the Dispatcher searches the ASCB queue starting with the ASCB pointed to by PSAANEW. For each ASCB, the Dispatcher checks first for SRBs on the LSPL, then for ready TCBs. The TCB check is made by comparing the count of ready TCBs with the count of CPUs active in the address space. If there is ready work, the Dispatcher switches addressability to the selected address space. To this point, the dispatcher has been running in the last dispatched address space. To switch addressability, the Dispatcher loads the address of the segment table for the new address space into the hardware control register.

After selecting the address space and switching addressability to it, the Dispatcher first searches for SRBs on the LSPL. If one is found that can run on the current CPU and SRBs are not stopped in the address space, then the SRB is dispatched. If there are no dispatchable SRBs for the address space, the Dispatcher requests the local lock for the address space. If it is available, the Dispatcher searches from the top of the TCB queue for a ready TCB. The first TCB that is ready and not active on another CPU is selected, and if it is not prevented by CPU affinity requirements from running on the current CPU, it is dispatched.

The local lock can be unavailable for three reasons. Either the lock is held by another CPU or it was held by a task or SRB that was suspended or interrupted. The Dispatcher checks this by looking for the interrupt ID in the lock word. If it is not there, the Dispatcher looks for another address space to run. If the lock holder was interrupted, the Dispatcher adjusts the lock word and the CPU lock-held bit mask to make the current CPU the owner of the lock, and then redispatches the interrupted local supervisor routine.

If there are no dispatchable global SRBs and no address spaces with dispatchable SRBs or TCBs, the Dispatcher places the CPU into wait state. The CPU remains in wait state until an interrupt occurs from the current CPU because of previously started I/O or timer requests, or from the other CPU when it has made work ready.

Alternate CPU recovery

Alternate CPU Recovery (ACR) is a process that is invoked when a CPU in a tightly-coupled multiprocessing environment can no longer function. The invocation of ACR occurs as a result of a signal sent by the failing CPU before it enters a disabled wait or check-stop state. This signal may be either hardware generated (Malfunction Alert) or software generated (Emergency Signal).

objectives of ACR

The objective of ACR is to enable the system to continue without the use of the failing CPU. While the system may be able to continue, it does so in a degraded fashion. Obviously, the reduction of available CPU power contributes to this degradation. Also, jobs that require the failing CPU in order to execute (for example, an emulator feature or a device available only to the failing CPU) will cease (if in progress) or will not be permitted to run. If there are a significant number of such jobs, the meaningfulness of continued system operation is questionable. However, ACR does not attempt to pass judgment on the meaningfulness of system operation, but rather enables the system to continue. The decision to terminate the system is left to the system operator.

In designing the ACR process, three major design objectives were adopted. The normal Recovery Termination Manager (RTM) facilities are used to interface with recovery and retry routines; that is, no special facilities for the ACR environment are provided. A further objective of the ACR design in OS/VS2 Release 2 was to make the recovery capability from a CPU failure equal to its recovery capability from a machine check or program error. This is accomplished by using the normal system recovery routines. No attempt has been made to design new

recovery routines for each of the possible states in which a CPU might be when it fails. A third objective of the ACR design was to provide an environment in which the majority of recovery routines could be insensitive to the ACR environment. The problems associated with achieving this objective and the way in which it was accomplished are explained in the following sections.

disabled spin loops

In order to make a full ACR capability possible, the system must be able to receive a Malfunction Alert (MFA) or an Emergency Signal (EMS) whenever one CPU is waiting for another CPU to do something. Clearly, if one CPU is in a totally disabled spin loop waiting, for example, for a failed CPU to release a lock, recovery from the CPU failure would be impossible. The running CPU, unable to receive the MFA/EMS, would never know that the CPU for which it is waiting had failed. Thus, the first consideration of the ACR design is to ensure that the system is able to receive notification of a CPU failure. This is accomplished by having all of the OS/VS2 system components open an MFA/EMS "window" whenever they enter disabled loops waiting for another CPU to perform some function. The opening of the window involves periodically enabling the CPU for MFA and EMS interrupts.

The components of the VS2 system that enter spin loops are:

- Lock Manager—spins waiting for a global spin lock to be released.
- Real Storage Manager—spins during PTLB processing.
- Timer Supervisor—spins during time-of-day clock synchronization.
- Inter-processor communications—spins waiting for another CPU to acknowledge that it has received or completed the processing associated with a SIGP instruction.

ACR initialization

When the MFA or EMS is received by the External Interrupt Handler, control is routed to the Recovery Termination Manager, at a special entry point, to begin the ACR process.

The first phase of ACR is concerned with handling the two processes that were in progress at the time of the CPU failure. RTM views these two processes differently. The process that was in control on the failed CPU has been abnormally interrupted; that is, an abend has occurred. RTM must, therefore, ensure that control is passed to any recovery routines (FRRs, STAE or ESTAE exits) that were established on the failed CPU prior to the malfunction.[2,6]

The process in control on the functioning CPU (the CPU performing the ACR process) has also been interrupted. However, with respect to this process, the interrupt is not abnormal; it has

not caused an error. RTM views this process as simply interrupted work, much as IOS views work stopped by an I/O interrupt. Thus RTM ensures the normal resumption of this process.

Thus, during the first phase of ACR, RTM takes responsibility for two processes. Upon entry, it indicates that the failed CPU is no longer available (by turning off its "alive" bit in the Common System Data area), marks the failed CPU's timer as permanently damaged, and sets the system in ACR mode (by turning on the LCCAACR bit in both CPUs' LCCAs). While in ACR mode, RTM alternately passes control to the interrupted work of the good CPU and to the recovery routines (and any requested retry routines) of the interrupted work of the failed CPU. The system remains in ACR mode until a state is reached in which normal system operation can be resumed. RTM begins this "switching" process by returning control to the External Interrupt Handler to resume the interrupted work of the functioning CPU. At this point, the ACR initialization phase is over.

locks and the disabled state

When the CPU malfunction occurs, the process in progress may hold one or more global spin locks and/or be executing in the disabled state. The same is true of the process in progress on the good CPU. This is the reason for the special ACR processing that is described in the following sections.

If both CPUs owned locks, the problem is that no matter which process RTM chooses to execute first (the FRR for the work on the failing CPU or the resumption of work on the good CPU), either process may request a lock held by the other. Under normal circumstances, this would result in a lock spin. In an ACR environment, this obviously must not occur since the functioning CPU now essentially owns both sets of locks.

The problem with the disabled state is that a disabled process is theoretically non-suspendable. The disabled state serializes activity on a CPU. Thus, CPU-oriented serially-reusable resources can normally be used by a process in disabled state with no possibility that the resources will be preempted. Fields in the PSA, such as the FRR stack and register save areas, and the Configuration Control Array are therefore available to a process in the disabled state. In the ACR environment, RTM owns the work of two CPUs. If either or both of the processes were disabled at the time of the failure, RTM must provide an environment in which the CPU-oriented serially-reusable resources are preserved.

ACR dispatching algorithm

As stated previously, following a CPU failure the system remains in ACR mode until a point is reached at which normal system operation can be resumed when both processes have entered the

Dispatcher; that is, when both processes take some action that causes the Dispatcher to receive control (such as exit from an SRB, or a WAIT SVC).

Entrance into the Dispatcher implies that the process in control has reached a suspendable state such that it can be suspended and resumed by normal system functions. When a process can be suspended another process can be dispatched on the CPU. Thus a process that can be suspended no longer has any claim to CPU-oriented serially-reusable resources. In addition, since a process owning a global spin lock cannot be suspended, entrance into the Dispatcher also indicates that the process in control owns no global spin locks.

When both processes have entered the Dispatcher all global spin locks have been freed and no dependency exists on CPU-oriented serially-reusable resources. At this point, the final cleanup routines of ACR may be invoked to complete the ACR process.

Following ACR initialization, RTM returns control to the interrupted work of the functioning CPU. RTM is again entered for ACR processing under either of two conditions:

- The process enters the Dispatcher, which detects that the system is in ACR mode (by testing the LCCAACR bit set by ACR initialization) and passes control to RTM. RTM then restores the status of the failed CPU and passes control to the recovery routine for the process that was in progress.
- The process requests a global spin lock that is not available. At this point, the Lock Manager must determine if a spin on the lock is possible. If the currently suspended process (from the failed CPU) owns a lock higher in the hierarchy, a spin cannot be allowed. The process in progress must be suspended, its status saved, and the suspended process resumed. The Lock Manager calls RTM to perform the suspension and to resume the other process.

The suspension rule for lock conflicts is the crucial algorithm of the ACR process. Simply put, the algorithm is "On locking conflicts, dispatch the process with the highest lock." This prevents deadlocks in the ACR environment. The algorithm is executed each time a lock conflict is encountered while the system is in ACR mode. Thus, it is possible to suspend and resume both the recovery work of the failed CPU and the normal work of the good CPU multiple times before both processes finally enter the Dispatcher.

In addition to preventing deadlocks, the suspension algorithm also makes it possible for recovery routines to be independent of the ACR environment. The recovery routine need not be sensi-

tive to the fact that it is running on a different CPU than the original process (although this information is provided in the RTM interface). The recovery routine can get and release locks (either explicitly, or implicitly through branch-entered supervisor routines) and request retry routines. The implementation of the suspension algorithm requires the existence of the lock-held bit masks that are maintained by the Lock Manager. It uses these masks when a locking conflict is encountered to determine which process owns the highest lock.

saving status

As stated previously, processes that hold global spin locks are disabled, are normally non-suspendable, and "own" CPU-oriented control information and data areas. However, ACR must be able to suspend such processes if the failed CPU owned a global spin lock when the failure occurred and when locking conflicts occur while in ACR mode. Before we explain how this is done, the nature of the CPU-oriented control information and data areas will be discussed.

The CPU-oriented control information and data areas consist of two types of fields—physical and logical. In general, *physical fields* are those that describe the hardware associated with a CPU. Examples of physical fields would be the channel availability table and the Time-of-day clock, clock comparator, and interval timer status indicators. *Logical fields* are software-oriented fields essentially independent of the hardware. Examples are the lock-held bit masks and the FRR stack.

A fundamental part of the ACR design is its approach to the two different types of fields. With respect to the FRRs for the work that was in progress on the failed CPU, ACR makes it appear that they are still running on the same logical CPU. This implies that when the recovery routines of the failed CPU are "dispatched" by RTM, the logical fields that are normally addressable to the recovery routine are those of the failed CPU. This eliminates the need for special ACR code in the recovery routines that are hardware independent.

The physical fields are not modified by ACR since they always reflect the hardware status of the CPU that is performing the ACR process, even when the failed CPU's FRRs are dispatched. Thus, those FRRs that must reference the physical PSA/CCA fields that were in use by the abended process require special code for the ACR environment. The physical CPU identification of the failed CPU is provided in the FRR interface so that this information can be found by these FRRs. (The design rationale was that any attempt to "fool" a process into thinking that it is operating on a different physical CPU is a dangerous procedure. It is obviously doomed to failure if the process attempts to act on the basis of the simulated hardware status by, for example,

starting I/O with respect to the good CPU on a non-existent channel. For this reason the design was chosen that requires these hardware-dependent FRRs to be aware of the actual hardware on which they are executing.

In order to enable ACR to provide "normal" addressability to the logical fields without requiring it to be aware of each field, the physical and logical *Configuration Control Arrays* (*PCCA* and *LCCA*) were created. In general, the LCCA contains pointers to all the logical fields; the PCCA contains pointers to the physical fields. The only exceptions to this are the fields of the PSA. The System/370 architecture requires the PSA to contain certain physical fields such as the new Program Status Word and the Channel Address Word. The PSA also contains a small number of logical fields for compatibility (TCB new) or severe performance reasons.

The PSA contains the addresses of the LCCA and PCCA. Thus, RTM suspends a process by moving the logical fields of the PSA to an ACR work area (pointed to by the LCCA). In order to resume a process, RTM moves the logical fields from the ACR work area to the PSA and stores the address of the appropriated LCCA in the PSA.

I/O restart

When both processes have entered the Dispatcher, RTM invokes the I/O Restart function which handles incomplete I/O operations that were initiated by the CPU prior to its failure. For each outstanding I/O request, I/O Restart simulates a channel error and calls the I/O Second-Level Interrupt Handler to pass control to the appropriate Error Recovery Procedure (ERP). If the functioning CPU has a path to the device on which the I/O request was initiated, a "retry possible" indicator is set in the ERPIB. If the functioning CPU has no path to the device, the "no retry" indicator is set in the ERPIB.

The ERP routine restarts the I/O on an alternate path. If none exists, the ERP invokes Dynamic Device Reconfiguration to attempt moving tape or direct access volumes to different devices.[7] If neither of these can be done, the ERP will post to the user a permanent error indicator.

I/O Restart also marks offline all device paths that were available to the failed CPU. In addition, if a device is found that no longer has available paths to the functioning CPU, the associated UCB is marked offline (to prevent subsequent allocation of the device) and an indicator is set that causes subsequent I/O requests to the device to be posted with permanent error. I/O Restart schedules a message to the operator which indicates that a device has been lost due to the CPU failure.

If the failed CPU had an outstanding reserve on a device when it failed, an integrity problem could arise if I/O Restart released the reserve (via SIGP RESET). The integrity problem exists if a loosely-coupled CPU was waiting to reserve the device to update data that were under the control of the failed CPU. On the other hand, if the reserve is not released, the loosely-coupled CPU would wait forever. In order to solve this problem, I/O Restart first puts the system into a wait state. At this point, the operator has two options; he can re-IPL the system, or stop the loosely-coupled CPUs and press the restart button on the functioning CPU. If he chooses the latter, I/O Restart again receives control, frees the reserves on the failed CPU via a SIGP RESET, and attempts to acquire the reserves for the functioning CPU. (The stopping of the loosely-coupled CPU prevents it from reserving the device after the SIGP RESET.) If I/O Restart can and does successfully acquire the reserves, the I/O Restart process is complete and control is returned to RTM. If it cannot, it puts the system into a hard wait state and a re-IPL is necessary.

The wait state loaded by I/O Restart when it first detects the outstanding reserves is either a 041 or a 042 wait state which indicates to the operator that the failed CPU had outstanding reserves. The 042 code is loaded by I/O Restart if the reserved device is logically offline to the functioning CPU.[8] If the operator decides to restart the system, I/O Restart brings a path to the device logically online to reserve the device for the functioning CPU.

ACR termination

When the I/O Restart function completes, RTM posts the console switch ECB in the Unit Control Matrix. This causes the Communications Task to reroute any messages destined for consoles attached to the failed CPU. The console switch routine also automatically switches to any secondary console that is associated with a lost console and selects a new master console if necessary. RTM then issues an SRM event indicating that a CPU has been lost (SYSEVENT code ALTCPREC), cleans up its internal work areas, and branches to the Dispatcher. At this point the ACR process is complete.

In addition to the software components mentioned previously in this section, the following also have special support for the ACR environment.

other system resources affected by ACR

The Dispatcher. In addition to passing control to RTM when the system is in ACR mode, the Dispatcher also handles tasks and SRBs that have CPU affinity to the failed CPU. When the Dispatcher finds an SRB or task that it cannot dispatch on the current CPU, it checks to determine if there is a CPU available on

which the SRB or task can be dispatched. If there is not, the Dispatcher abends the SRB or task with a completion code that indicates a CPU failure has occurred.

Real Storage Management (RSM). RSM enters two lock spins during page-invalidation processing. In the first spin, RSM waits for the other CPU to stop in preparation for a PTLB. In the second, RSM waits for the other CPU to issue a PTLB instruction so that it can invalidate a page. RSM detects when a CPU failure occurs in either of these spins (by checking a bit that indicates which CPUS are still functioning) and simply stops waiting for the failed CPU. If PTLB processing was in control on the CPU when the failure occurred, its recovery routine retries the operation on the functioning CPU. (PTLB is an instruction that clears an internal hardware buffer which allows optimization of address translation by the hardware. This buffer has to be cleared each time a page is invalidated by RSM. In MP it is necessary to clear the buffer on both CPUs at the same time. The above sequence is used to synchronize the CPUs and clear the buffers.)

Inter-Processor Communications (IPC) Manager. A return code of 20 is returned if a CPU failure occurs while the IPC Manager is processing a RISGNL request.

Concluding remarks

Programming support of tightly-coupled multiprocessing hardware has been discussed, particularly the OS/VS2 Release 2 components: locking, service management, CPU affinity, dispatching, and alternate CPU recovery. These facilities improve the control program utilization of the two-CPU environment in that it can now run parallel disable functions. Also, spin time on locks is reduced, and the new dispatchable unit allows more parallelism in new system functions. Furthermore, the system is able to recover from the loss of one of the two CPUs.

CITED REFERENCES AND FOOTNOTE

1. R. A. MacKinnon, "Advanced function extended with tightly-coupled multiprocessing", in this issue.
2. *IBM System/370 Principles of Operation*, Form GA22-7000, IBM Corporation, Data Processing Division, White Plains, New York.
3. *OS/VS2 System Programming Library: Job Management, Supervisor, and TSO*, Form GC28-0682, IBM Corporation, Data Processing Division, White Plains, New York.
4. *OS/VS2 System Programming Library: System Generation Reference*, Form GC26-3792, IBM Corporation, Data Processing Division, White Plains, New York.
5. The Dispatcher when entered will select the highest-priority work ready to run. The rest of the supervisor services, however, will under some conditions bypass the Dispatcher and, for limited periods of time, lower-priority work

will be running while higher-priority work is ready. This is done to avoid some system overhead and is expected to have little visible effect on priority support.
6. *OS/VS2 Supervisor Services and Macro Instructions*, Form GC28-0683 IBM Corporation, Data Processing Division, White Plains, New York.
7. *Operator's Library: OS/VS2 Reference (JES2)*, Form GC38-0210, IBM Corporation, Data Processing Division, White Plains, New York.
8. *OS/VS Message Library: VS2 System Codes*, Form GC38-1008 IBM Corporation, Data Processing Division, White Plains, New York.