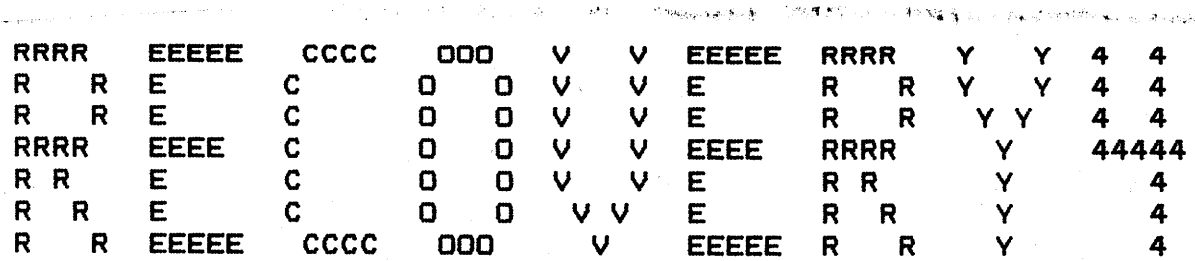

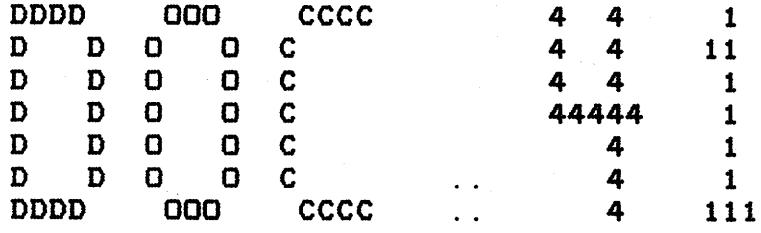

START Job RECOVE Req #112 for WAW Date 23-Jul-82 21:27:39 Monitor: Rational
File RM:<GPA.DOC>RECOVERY4.DOC.41, created: 18-Jul-82 12:04:19
printed: 23-Jul-82 21:27:40
Job parameters: Request created:23-Jul-82 21:27:39 Page limit:126 Forms:NORMA
File parameters: Copy: 1 of 1 Spacing:SINGLE File format:ASCII Print mode:A

Crash Recovery & Disk Recovery

Capabilities of the R1000

DRAFT 4

July 15, 1982

1. Overview

The purpose of this document is discuss the basic recovery capabilities of a rational machine. We are primarily concerned here with crashes, single processor failure, and IOP/disk failure. Hardware failures such as failing tape units, lineprinters, communications, etc are not considered here. System software component failure is also not considered here.

Table of Contents

2. Crash recovery semantics

This section defines snapshots and crash recovery from the point of view of an Ada programmer.

2.1 Definition of crash

A crash is an event which causes the system to stop operation. The event does not cause disk data loss other than writes in progress at the moment of the crash.

Crashes are typically caused by some sort of hardware failure. Software may also crash the system, but presumably only in VERY rare circumstances.

Crash recovery refers to the action taken by the system when it comes back up after a crash.

2.2 The lifetime tree

Let unit mean a package, task or collection. All concrete objects are either units, or integral components of a unit. If a unit is annihilated, its integral components also cease to exist. In this context, there is a parent-child relationship between units, as defined by Ada. Thus, there is a tree of units. The root is the R1000 package. A unit exists iff it is reachable from the root of this tree.

2.3 The notion of permanence

The permanent lifetime tree is defined as the lifetime tree less all of its temporary subtrees. A temporary subtree is always rooted by a task unit. By default, all tasks are temporary. There is "permanent" pragma which may be applied to a task to make it be part of the permanent subtree. We may also desire some other method of control over the permanent attribute so that not all instances of a particular task type are constrained to the same value of the permanent attribute. Of course, declaring a task to be permanent is legal iff there are no intervening temporary tasks in the path from the root (of the lifetime tree) to the task being declared permanent.

The above definition implies that all directory packages are permanent. And it implies that all non-task, non-collection objects placed in a directory are permanent.

Examples: Diana is in the permanent portion of the tree (expect for local copies serving as a sort of cache). Permanent files and other such objects are obviously in the permanent portion of the lifetime tree. Tasks created by the environment to run user commands are in temporary subtrees. Sessions run in the temporary portion of the lifetime tree.

2.4 Snapshot

Conceptually, a snapshot is defined as follows. The physical processors are stopped at instruction boundaries. The state of all pages of all address spaces which comprise the permanent portion of the lifetime tree are recorded (on disk) as an atomic action. The processors are then allowed to proceed. Note that the saved state is

retained until successful completion of the next snapshot, at which point it is discarded.

The snapshot is transparent to programs running on the system with the following exception: Permanent tasks will be unrunable for roughly a half a second while the snapshot is taking place. Temporary tasks continue execution.

Normally, a special system task causes snapshots to be taken at regular intervals (like every 3 minutes). However, it is possible for the customer to have explicit control over the timing of snapshots.

Crash recovery involves restoring the system to the previous snapshot. This implies that temporary objects vanish.

The execution state of a permanent task is included in the snapshot. Thus, a task which was waiting for a rendezvous (at the instant of the snapshot) will still be waiting for a rendezvous after recovery. If the task was running, it will be running after recovery (at wherever the saved PC indicates). If the task was delaying, its delay is restarted as part of recovery. The task state also includes all intact messages from calling tasks (including temporary ones). If placing a message into a task's queue is not atomic with respect to snapshots, then assume that partial messages are automatically discarded when encountered.

2.5 Rendezvous between permanent and temporary task

If we have a permanent client and temporary server involved in a rendezvous when the system crashes, recovery will cause the server to disappear. The client will receive TASKING_ERROR. This is consistent with Ada defined abort semantics.

If, on the other hand, the client is temporary and the server permanent then the server will carry the rendezvous (with a ghost) to completion without knowing that the client has vanished. This is also consistent with Ada abort semantics.

2.8 Atomic operations on permanent objects

Consider a temporary task making a change to an object in the permanent portion of the lifetime tree. The system can take a snapshot at any time; and it might later restore itself to that snapshot. This is equivalent to annihilation of the temporary task at any arbitrary point in its execution.

To properly program updates to a permanent object in this environment, one should always have a permanent task which protects the object. Assuming that an atomic action takes the object from one consistent state to another, then the permanent task should have entry calls for the various atomic actions which can be applied to the object. Since the state of the permanent task and the permanent object are snapshotted in synchrony, and once the task starts the action it gets to finish it, the action is atomic even in the face of arbitrary snapshot and recovery points.

It is theoretically possible to have atomic actions be as simple as assigning a single value to some variable in the permanent object. Consequently, one might argue that atomic updates to the permanent object need not be routed through tasks.

For example, suppose we have a tree represented in a permanent collection. To atomically change a particular subtree, one could make a new copy of the subtree which includes the desired change, and change the pointer to the subtree (in the parent node) with a single assignment statement. (Garbage, created by interruption of the algorithm, is automatically reclaimed by the garbage collector).

To program updates to permanent objects from temporary tasks, the Ada programmer needs a precise specification of the atomicity of all operations available in Ada. We certainly don't want the programmer to have to understand the architecture, and second guess the code generator, just to program updates to permanent objects. For example, suppose the name of the permanent object is A, and it has type T. Is the assignment "A := B" atomic with respect to crashes? The answer seems to depend on T. If T is a private type, then the answer is unknown in the context of the assignment statement. This implies that abstract type specifications should include information about the atomicity of operations on the type.

Programming in this fashion generally involves very subtle details totally unrelated to the conceptual object at hand; such details make program correctness arguments even more difficult. Thus, this programming practice is prone to bugs which exhibit themselves only after recovery.

Furthermore, a program which takes advantage of these subtleties will undoubtedly not be portable to other RMI architectures.

Consequently, there will be no specification of what Ada operations are atomic. One cannot update a permanent object from a temporary task and expect defined results.

2.7 Nested atomic actions

Suppose we have permanent tasks A and B which protect corresponding objects, and provide appropriate atomic actions. Suppose we wish to program a larger atomic action (like moving money from bank A to bank B). Simply construct a new permanent task C which supports the action of moving money between accounts.

2.8 Recovery notification

There is a special entry call known as the recovery call. If a permanent task contains an accept for the recovery call, then when recovery occurs the task will be called (at the recovery entry).

2.9 Snapshot info

There is a system function which will return the current system snapshot number. Note that snapshot numbers are not unique in the life of the system. From the user's point of view, the system is free to restart the sequence number generator whenever the system goes down, so don't store them in permanent data structures.

2.10 Limited support for TPS

There are lots of applications which embody the transaction concept used in transaction processing systems (TPS) but without the demand for high volume and guaranteed commit.

For example: The insurance agent fills out the application online. When it is completed, a hard copy is printed. The agent explains it to the customer. The customer signs it. This entire activity may span many minutes. The application program could give the agent confirmation when the next snapshot (following completion of the application) occurs. Confirmation will probably occur prior to the customer leaving the office. But in any case, the agent simply needs to keep track of unconfirmed applications for the few minutes between snapshots. In the event of a crash, the agent can reenter the application from the hard copy.

There are many significant applications which can be programmed in this fashion.

Even airline reservations could be handled this way. The terminal could record the transactions since last snapshot. In the event of a crash, the transactions could be automatically redone. (This works because we distributed the log function to the terminal; given disks as I/O devices, the log could be kept on the R1000 instead.)

2.11 Real time applications

The delay in execution of permanent tasks doesn't necessarily rule out real time applications. The RTS (real time system) could control the snapshot timing, allowing it to occur when the delay doesn't effect system operation. Or, the RTS could use buffering (temporary) tasks between permanent storage and the active temporary tasks to keep the active tasks from "hanging up" on permanent storage during the snapshot interval.

3. Disk recovery semantics

This section describes the backup and recovery capabilities that we know the R1000 will have (presumably in early deliveries). Appendix C discusses independent volume reconstruction - a proposed additional capability.

3.1 Virtual memory system

The "virtual memory" (or V-mem for short) is a set of one or more fixed media disk drives which comprise the permanent secondary storage of the system. There may be other drives, called "free drives", fixed or removable, but these are treated as garden variety I/O devices, like a tape drive. By default, all fixed media drives are part of V-mem.

The "backup and recovery" capabilities provide for repair of the V-mem following disk drive failures. They do not provide individual archival features. There is another tool for this purpose; it is described elsewhere.

3.2 Volume

The V-mem is partitioned into one or more volumes, each volume being comprised of one or more fixed media drives. The system will default to 1 drive per volume. Address spaces are wholly contained within volumes.

The programmer need know nothing about volumes. The system manager needs to know that the time to recover from disk drive failures is minimized with one drive per volume. There is a tradeoff between recovery time and allowing for gigantic address spaces (larger than free space per volume). This tradeoff is left to the system manager.

3.3 Changing disk configuration

The V-mem configuration is recorded in the V-mem itself. If component drives are missing at boot time, the system will not come up. Existence of drives other than those in the V-mem configuration are simply ignored at boot time. They have to be explicitly mounted, etc. Disks that are not components of V-mem may be added/removed at any time (not physically, of course).

New drives can be added to the V-mem. The physical connection is done with the system down. The initial formatting may be done online. Adding a physically connected formatted drive to the V-mem configuration can be done online. It may be added to an existing volume, or may become a new volume.

Drives may not be removed from V-mem without resorting to the drive failure recovery procedures (to be described).

3.4 Basic backup and recovery procedures

Let T denote the time when the full save is started. Let T′ denote the time of the snapshot taken just prior to T. A "full save" makes a complete copy of the state saved by the snapshot at time T′. That is, later restoration of the system using the full save is equivalent to a crash at time T.

Full saves can be created in one of two ways. There is a raw copy facility which dumps V-mem track-by-track to the backup media. Raw copy only works offline.

The compactifying copier copies the address spaces (in the snapshot) one at a time to the backup medium; it is called compactifying because the backup medium ends up with all blocks of a particular address space being contiguous. Compactifying copy only works online. The online full save manages to save the snapshot in a consistent fashion, even though the system is up and running.

An "incremental save" also saves a snapshot of V-mem. Again use T and T' to denote the save and prior snapshot times, respectively. The "base save" refers to some previous full/incremental save. Let S' denote the snapshot saved by this previous base save. The incremental (at time T) makes copies of only those (address space) pages that have changed between S' and T'. As for full saves, later restoration of the system using the incremental is equivalent to a crash at time T.

Incremental saves are always created with the compactifying copier. And incremental saves are consistent despite the fact that the system is up and running.

"Save set" refers to the set of target packs/tapes consumed by the full/incremental save.

A "recovery path" is a set of k, k > 0, save sets with the following properties: (a) The first save set is a full save. Let T(0) denote the snapshot time of the full save. Any additional save sets, in the recovery path, are incrementals. (b) Let S(i) and T(i) denote the base and snapshot times for the ith incremental in the recovery path. Then, S(i) = T(i-1), for i in 1..k.

The recovery procedure takes a recovery path and reconstructs the snapshot at time Tk. Thus, a recovery path is equivalent to a full save at time Tk. Consequently, we say that an incremental takes a snapshot of V-mem, on the assumption that the incremental is a member of some recovery path.

3.5 Failures

We are concerned here with the V-mem disk subsystem. Some failures can cause V-mem to get messed up (a disk head crash, for example).

Section A.3 provides illustrative figures for MTD (mean time to destruction). Given that such an event has occurred, one's only recourse is to use the recovery procedure to restore a snapshot taken prior to the failure. Appendix B illustrates a typical backup strategy and the corresponding recovery times and expected work lost.

Section A.2 provides illustrative figures for MTBF. Given a failure of one of the disks of V-mem, one has only two choices. (a) Wait until the problem is fixed, and hope that no data was lost. This avoids losing work (most of the time), but makes the system unavailable until the problem is fixed. (b) Immediately perform recovery to prior save point. This maximizes system up time, at the expense of losing work. (It also assumes spare capacity, etc; see section 3.7).

3.6 Mirrored V-mem

There is an option for mirroring V-mem. In this case, every drive (in V-mem) has a mated mirror image drive on another IOP. Loss of any single component of the disk subsystem will simply cause the loss of one or more mates. Loss of these mates does not cause the system to crash, it will simply keep running without the lost mates. The system may have to be momentarily stopped to allow the repair man to prune the damaged components, and to reconnect them once repaired, but otherwise the system will run without interruption. At some point following the repair of the mates, they can be reintegrated into V-mem. The reintegration period involves bringing a repaired mate back to a state where it mirrors the mate that was uneffected by the failures. This process occurs online.

Thus, mirrored V-mem can only fail if there is a second component failure before the first failure can be repaired and reintegration completed. Failure rates for mirrored V-mem are given in section A.4.

3.7 Choice of backup media

The backup and recovery system will operate with either tape or disk. The tape may be vanilla or streamer. The disk may be fixed or removable (but not part of V-mem).

As indicated by the figures in appendix B, there is no speed advantage in saving/restoring an incremental to/from disk instead of tape, due to the random access nature of of incrementals. However, saving/restoring a full save is significantly faster using a disk.

One might want to use disk for small incrementals so that they can be performed automatically without operator assistance in mounting media.

Although raw copy can make full save run 20 times faster than with compactifying copy (see Appendix B), it has one major disadvantage: The save set can only be restored to an identical set of drives. If you want to be able to restore prior to repair of a failed drive, and you want to use raw copy for speed, then you will need a spare drive which is not part of V-mem, and does not contain the save sets in the recovery path. (If you have different capacity drives in V-mem, you would need more than one spare).

When compactifying copy is used for full save, then recovery can reload V-mem onto a different configuration provided: (a) There is sufficient capacity. And (b) there aren't any problems with fitting gigantic address spaces into the new configuration.

3.8 Parallel backup/recovery

It is possible to do backup/recovery by volume. Given multiple backup devices, the real time required can be kept to a minimum by backing up (or recovering) several volumes simultaneously.

3.9 Fast recovery option

In this case, the incremental information is effectively kept both on the source volume, and the target backup medium. There is very little time overhead associated with keeping the info on the source volume; one is simply retaining the snapshotted generations until next

incremental.

Recovery is faster because: (a) For survivor volumes, simply roll back to the snapshot generations. (b) For the destroyed volume, reconstruct it from backup.

3.10 Compacting disk space

Doing a compactifying copy full save followed by restore will compact address spaces on disk. This may be advantageous in some cases.

3.11 Reliability

Sufficient error correction capabilities are provided in the tape subsystem that backing up and restoring V-mem to and from tape should be no less reliable than doing it to disk.

4. Crash recovery details

4.1 Address space structure

Each address space has a distinguished page, called the root page, from which all other pages of the address space may be located. For code/import spaces, this is page 0 of the code/import segment; for module spaces, this is page 0 of the control stack.

The root page contains mappings for some set of the pages of the address space. These are the so called "fast access pages". The (page --> disk block) mapping for the remaining pages is contained in an index.

Each index page is an array of R entries. Each entry is of the form (snapshot #, disk block #). The block number references a son of the index page. The son may be either another index page, or a data page in the logical address space. Let T denote the number of the last snapshot. The son has been changed since the last snapshot iff its snapshot number is T+1. The first time the son is changed after a snapshot, it is assigned a new block number; when this new block assignment is made, the number of the next snapshot is recorded in the index page entry. Blocks assigned to pages changed since the previous snapshot are known as shadows. Note that a change to a data page causes shadows to be created for the entire root to data page path.

The (page --> disk block) mapping for a particular page is obtained from the index as follows: Take the page number and represent it as a string of digits in radix R: d0,d1,...,dn. Digit di corresponds to the subscript of an entry in the index page at level i the tree. Thus, the digit string describes the path from the root to the desired page. (Obviously, if R is a power of 2, the digit string can be constructed with simple field extraction from the page number.)

The (page --> disk block) mapping for the fast access pages is a set of entries (as in the index) stored in the root page.

(In the case of module spaces, there is either 1 index per segment, or the indexes are combined, or some combination thereof; the exact details don't seem relevant to the remainder of the discussion.)

"Generation" refers to the version of an address space which is delimited by a snapshot.

This structure has the following important properties:

Property A: Recall that a block being written at the moment of a power failure induced crash has a high probability of being partially written: neither the before or after image survives the crash. The address space structure is not sensitive to these failures because updates to the address space always occur to shadows.

Property B: Recall that an incremental saves all pages changed since some prior snapshot. The index structure directly supports easy calculation of the changed page set. Note that the calculation need only traverse those portions of the index that have themselves changed.

Property C: The index structure provides a natural mechanism for

retaining old generations. This mechanism is useful in a variety of situations: full save snapshot retention, mutatory retention, etc (these will be discussed in later sections).

Property D: Recall that the garbage collector requires region lists for its implementation. Let P denote the time at which a region is made empty. Let Q denote the time at which it later becomes an evacuating region. All pages changed in the interval (P..Q) are in the region list. Let X denote the number of the earliest snapshot which occurred after time P. Similarly, Y denotes the snapshot occurring after Q. (Note that Y is probably in the future). The set of all pages whose snapshot number is in the range (X..Y) is a superset of the pages in the interval (P..Q). If the collector wants the region list to be static, it can wait until snapshot Y has occurred. As for incrementals, calculating the region list is quite simple and efficient.

Property E: Accessing pages and updating the index is relatively trivial. There are no reorganization issues, since the index is always balanced.

4.2 Task control block contents

The crash recovery mechanism assumes the task control block (TCB) contains the following information:

- Address space kind
- A bit indicating whether or not this address space is permanent
- For modules, a bit indicating that the module is a task
- For tasks (at least), the parent link (in the lifetime tree)
- For tasks, a state variable
- For tasks, a bit indicating that the task desires a recovery call

We assume the following values for task state:

- RUNABLE
- DELAYING
- WAITING_FOR_PAGE
- WAITING_FOR_IO
- WAITING_FOR_CALL
- WAITING_FOR_RETURN -- from rendezvous call

4.3 Disk catalog

The catalog is a data structure implemented in stable storage which retains the mapping (address space name --> block for root page). The mapping also includes the TCB info as described above. This assumes that all pages of the address space leave main memory prior to the TCB. The catalog contains a mapping for each retained generation of a permanent address space.

4.4 Catalog cache

The catalog cache is a set of main memory pages which contain copies of disk catalog entries, and entries which indicate additions, modifications or deletions to the disk catalog. Catalog entries are physically deleted only after the snapshot following the logical deletion.

The catalog entry for a temporary address space would not typically

make it to the disk catalog; but it might get there if the address space was paged out and the catalog cache doesn't have enough room.

4.5 Dump buffer

A "dump buffer" is a set of N contiguous disk blocks used by a particular snapshot. There are two dump buffers per processor. We assume that for a particular snapshot, no two processors have their dump buffers on the same disk drive.

The number of dirty permanent pages cannot exceed "N" for a particular processor. A counter of permanent writable pages, per processor, is sufficient to enforce this invariant.

Assuming 1 Gbyte of disk per processor, 8 Mbytes of main memory per processor, and N = 6000, the dump buffers would consume 1.2% of the disk space.

Note: it is tempting to reduce the dump buffer size by making the rule that if there are more dirty permanent pages than will fit in the dump buffer, then we do some cleaning prior to starting the snapshot. But: neither the hardware nor microcode currently keep track of the number of dirty pages. And it takes 30-50 milliseconds to scan the tag store to compute this value. We are trying to avoid any algorithm which requires a tag store scan that cannot be interleaved with task execution. Come back and read this argument later, it should make more sense then.

4.6 Tag store contents

In addition to various other goodies, the tag store contents include a permanent bit for each page, derived from the corresponding bit in the TCB.

4.7 Snapshot

Assume that the last snapshot was number X. The next snapshot is Y=X+1. The actions required to take snapshot Y are as follows:

4.7.1 Alternating dump buffers

Snapshots alternate between the two dump buffers. This provides protection in case there is a crash during a snapshot.

4.7.2 Coordinator and worker snapshot tasks

There is a special system task which controls the timing of snapshot, and is called the snapshot coordinator. Each processor has a snapshot worker task. These tasks are temporary, and (re)created when the system comes up.

4.7.3 Overview of phases

The snapshot occurs in two phases. In phase one, the workers tie up all permanent pages, writing them to the dump buffer. They also write the catalog cache to the buffer. During this phase, the workers proceed without synchronizing with the coordinator or other workers. When all the permanent info has been written, the worker waits for notification of phase two from the coordinator, who starts phase two only after all workers have completed phase one. In phase two, the

workers increment their snapshot numbers, and release the permanent pages for further activity.

4.7.4 Phase one

The first phase starts by the coordinator sending an appropriate message to each of the workers. During phase one, each worker independently takes the following actions:

4.7.4.1 Seek to dump buffer

The worker causes the cessation of further I/O on the drive containing the worker processor's dump buffer. When the last I/O finishes, the worker causes the drive to seek to the cylinder containing the first block of the dump buffer for this snapshot.

4.7.4.2 Don't run permanent tasks

If a permanent task is running when the seek completes, it stops running at an appropriate macro boundary. When the seek completes, the worker causes the processor to enter a state where permanent tasks will not be run and where servicing of faults for permanent pages is disabled; that is, a fault for a permanent pages causes the faulting task to enter the WAITING_FOR_PAGE state.

4.7.4.3 Don't modify permanent TCBs

We assume that permanent TCBs are kept in separate lists (etc) from temporary tasks. Thus, the only event that can now cause the TCB for a permanent task to be written is a response to previously issued I/O. We handle this as follows: (a) Requests by other processors to write this processor's pages cause the other processor to fault. (b) Data from the IOP (corresponds to reads issued prior to beginning the snapshot) is discarded. (c) All I/O completion status is recorded in a special temporary buffer for this purpose.

4.7.4.4 Write dirty permanent pages

The worker incrementaly scans the tag store. If it finds a writable permanent page, it changes the page state to some special form of read-only. If the page was dirty, it queues it up for writing to the dump buffer. As the dirty pages are written, their dirty bits are reset. For dirty permanent pages, it records the page id in a main memory information buffer. If it finds a page corresponding to a TCB for a temporary task, it gets the parent link, and records it in the info buffer.

The above process is interleaved with execution of temporary tasks.

4.7.4.5 Write cache and info buffer

After all of the permanent pages have been written, the catalog cache is copied (indivisibly) to a temporary memory buffer. This locks out temporary task execution for less than 0.5 milliseconds. From there, the cache is copied to the dump buffer. And the info buffer is written to the dump buffer.

4.7.4.6 Wait for phase two

Writing the processor's permanent state is now complete. At this

point, I/O on the drive containing the dump region is reenabled, and the worker sends a message to the coordinator indicating the completion of phase 1. It waits for the coordinator to send a message starting phase two. Temporary tasks continue to run while the worker is waiting for the start of phase two.

4.7.5 Phase two

To start phase two, the coordinator sends the appropriate message to each of the worker tasks. The message includes the number of the next snapshot.

The worker reenables fault handling for permanent pages. Tasks suspended in the WAITING_FOR_PAGE state (for permanent pages) can now be serviced. The permanent pages are released on demand; that is, if a task faults on a permanent page in the special read-only state, the page's state is changed to writable, and the task proceeds.

Any changes to (page --> disk block) mappings use the new snapshot number.

Previous I/O completion codes (recorded in the temp buffer) are reprocessed at this point. This may require reissuing I/O requests.

4.7.6 Assignment of snapshot numbers

We assume a 3 byte unsigned integer for snapshot numbers. The sequence number generator is stored on disk. It is incremented by the snapshot coordinator. We assume that the coordinator takes several hundred snapshot numbers from the disk at a time, in such a fashion that crashes never reuse snapshot numbers.

Assuming snapshots are taken no more frequently than once every 100 seconds, snapshot numbers are not reused for 53 years.

4.7.7 Snapshot time

Assume: 8000 pages per physical processor. One IOP (with disk) per processor. 50% of pages are dirty. 10% of dirty pages belong to permanent address spaces. This equals 0.4 Mbyte of stuff to write to the dump buffer. We assume that with contiguous sector allocation we can make the IOP write 1 Mbyte to disk per second. Thus, the ballpark figure for snapshot time is 0.4 seconds.

4.7.8 Activity concurrent with snapshot

During this 0.4 second interval, the system can run, but with certain limitations: (a) Permanent pages cannot be changed; but a temporary page can be changed. (b) A temporary task which faults on a page not in main memory will probably have to wait until the snapshot is over before the fault can be serviced. Similarly, page cleaning operations probably have to wait. (c) Other high bandwidth I/O activities, such as tape I/O, will probably have to wait until the end of the 0.4 seconds. Presumably, terminals, lineprinters and other relatively slow devices will still operate without noticeable degradation.

4.7.9 Hidden snapshot overhead

Also note that each snapshot causes the minor computational overhead

of allocating shadows and updating indexes for permanent pages that are truly in the working set of dirty permanent pages. Since we do not keep separate generations for temporary address spaces, there is no shadowing overhead for temporary address spaces.

4.7.10 Generation retention

The normal scenario is: Assume that the last snapshot was number X, the next snapshot will be X+1. After snapshot X+1 occurs, the generation created by snapshot X may be discarded. However, there are special cases where old generations are retained (during online full save, for example).

Logically deleted address spaces (both permanent and temporary) are physically retained until the snapshot following their logical deletion.

4.8 Crash recovery

4.8.1 Alternating dump buffers

Recovery needs to choose between two sets of dump buffers. Dump buffers are written in such a manner that if a crash occurs during a snapshot that recovery will discard that buffer. Otherwise, recovery chooses the dump buffer corresponding to the latest snapshot.

4.8.2 Simple reload doesn't work

Intuitively, one would think that recovery simply involves reading the dump buffer back in, incrementing the current snapshot number, and taking off. Unfortunately, the dump buffer may not fit! Suppose we have 4 processors; three have 8 Mbytes, and 1 has 4 Mbytes. Suppose we have to recover the system with just the single 4 Mbyte processor. To guarantee that all permanent pages would fit, we would have to restrict the world to 2 permanent pages per line. Rather than make this assumption, recovery assumes that the dump buffer will not all fit.

4.8.3 Idempotent recovery

It is vital that a crash during the recovery procedure not destroy the recovered snapshot. Thus, recovery must take a new snapshot number to cause updates to go to shadows. This helps make the recovery procedure idempotent.

4.8.4 VPid reassignment

Due to the VPid reassignment following the crash, processors may have to read stuff that belongs to their VPids from several dump buffers.

4.8.5 Dump buffer reload

Set the current snapshot number to a number that is larger than the number of the snapshot to which we are recovering. This is vital to the idempotency of the recovery. Think of it this way: The snapshot info (being recovered from) must last until the next snapshot is taken.

Reload the catalog cache. This is done by conceptually reading each

individual catalog entry (belonging to this processor) from the dump buffer, and copying it into a fresh cache (created by recovery). This avoids collisions between pages of the cache, and confusion over VPids.

Remove any catalog entry for a permanent address space generation created after the snapshot was taken. This undoes work following the snapshot, including any partially completed recovery activity.

Reload the entries from the information buffer(s) that belong to the VPid set for this processor.

Now begin loading permanent pages (for this processor) from the dump buffer(s). Pages are loaded for a single address space at a time. The root page (contains the TCB) is loaded first. Then any index blocks, top down. Then any data pages. Of course, these pages are marked as dirty as they are read in.

During the loading process, we may encounter a situation where a page cannot be loaded, because the target line in memory is full. In this case, clean a page on that line.

Note that the page cleaning operation invokes the normal procedure for integrating changed pages into a shadow index structure for the new generation. It is vital that the pages for a particular address space be loaded top down; this ensures that index information in the dump buffer is used in constructing the new generation.

4.8.6 Single processor failure

This recovery scheme requires that a single processor failure be treated by crashing the system.

4.8.7 Update catalog

We make sure that all permanent tasks in main memory are cataloged (in the cache is OK). This simplifies the logic that follows.

4.8.8 Fix child lists

Permanent address spaces with temporary task children must get their child lists fixed to reflect the vanishing temporary tasks. Recall that a child list contains the address space name for each of the children, and a count of the number of children which are not X, where X is one of (terminable, terminated). Because we do not keep the states of temporary tasks synchronized with the states of their permanent parents, one would think we have to recompute the dependency count.

Rather than do that, let there be two dependency counts in a child list, one for temporary task children, and one for other children. Then we need only set the temporary dependency count to zero. The other count is correct since the counted dependents are all permanent.

For each parent link recorded in the info buffer (see 4.7.4), and for each cataloged temporary task address space, consider the parent link in its TCB. If the link references a permanent address space, then go look at the child list in the parent. It is possible for the list to no longer be there; forget it in this case. Note that the

tagged architecture of the type/control stacks allows us to verify that the supposed parent link really references a child list (perhaps not the same one).

Assuming the parent link references a child list, reset the temporary dependent count to zero. If the permanent count is also zero, and the parent was waiting to terminate, then change the parent's state to RUNABLE (or whatever it takes to later get it to realize it can terminate). If the temporary task is in the child list, remove it.

Note that we do NOT ignore catalog entries corresponding to deleted temporary tasks.

4.8.9 Get rid of faults for temporary pages

Find permanent tasks in the WAITING_FOR_PAGE state (look in the catalog). If the desired page corresponds to a temporary page, mangle its state such that the appropriate exception will occur, and make the task RUNABLE (or equivalent).

4.8.10 Get rid of I/O waits

Find permanent tasks in the WAITING_FOR_IO state (look in the catalog). Mangle its state such the appropriate exception will occur, and make the task RUNABLE (or equivalent).

Reissuing old (non disk) I/O does not make sense for lots of reasons.

4.8.11 Fix client death

Find permanent tasks in the WAITING_FOR_RETURN state. If the task is not waiting for a permanent task to return, mangle its state such that TASKING_ERROR will result, and make the task RUNABLE (or equivalent).

4.8.12 Take snapshot

4.8.13 Zap temporary address spaces

If there are any catalog entries corresponding to temporary address spaces, delete them. Must take snapshot prior to this action, since the snapshot from which we are recovering requires the temporary task states for fixing up the child lists.

4.8.14 The system is up

All processors synchronize prior to this event.

4.8.15 Restart tasks

Any task whose state is (or was made) RUNABLE should be reentered into the appropriate run queues. Similarly, a DELAYING task needs to get its delay restarted.

4.18.16 Restart fault I/O

Find permanent tasks in the WAITING_FOR_PAGE state (look in the catalog). The desired page must be permanent. Reenter the task into

the appropriate fault queue. Reissued the I/O.

4.8.17 Miscellaneous pruning

A permanent task may have been elaborating a temporary task. If it wasn't caught by the above actions, it will eventually fault on the non existent task, and get the exception at that point. Similarly, a permanent server which uses a reference parameter from a ghost temporary task will fault and get an exception.

4.8.18 Transparent return to ghost

In a rendezvous where the client is temporary and the server permanent, IN parameters are passed by value, IN OUT parameters are passed by value result, and OUT parameters are returned by value. When the server returns its results and finds the client has vanished, the results are placed in the bit bucket, transparent to the server.

An argument has been advanced that the programmer be required to explicitly program the parameter copying, and handling of the exceptions that result from referencing a non existent address space (because of reference OUT parameters).

There seems to be a strong counter argument that the requirement for copying parameters and handling exceptions in this fashion is going to be difficult to explain. Because lots of people either don't understand it, or they are simply not methodical enough, this copying technique won't be used all the time. Which means that you won't find out about the places where you forgot to program the copy and exception handling until a crash occurs at just the right moment.

It is not at all clear that we gain any efficiency by making the programmer deal with this issue. In fact, it may be less efficient. An abstraction which claims to have any general purpose utility would have to do the copy and exception handling. Consequently, we might find that a very large percentage of servers are doing the copy, even though the call does not cross the permanent/temporary boundary.

4.8.19 Recovery time

We assume that all processors can recover in parallel. The following time estimates are per processor. Assume 20K catalog entries at 32 bytes each. Reading the entire catalog at random takes 30 seconds. But it should be primarily contiguous. Consequently, we assume 10 seconds here. To random read the TCB for 200 tasks takes 10 seconds. We assume most temporary tasks are rooted under just a few permanent parents. Therefore, fixing the child lists should be short compared to the above. In the worst case, assume we have to clean half of the permanent pages. That takes 15 seconds. The total is around 35 seconds.

4.9 Block reclamation

Given two generations, X and Y=X+1, we wish to discard generation X. Let P denote some page of the address space. If P is in both X and Y, but has different snapshot numbers in the two generations, the block corresponding to P in generation X can be freed. This applies to index pages has well as data pages.

Rather than explicitly free these pages as generations are deleted,

we propose to simply toss the shadow structure for the discarded generation, and rely on garbage collecting the lost blocks.

Assuming that shadows are created at the rate of 5 per second per drive, shadows would consume 18 Mbytes per hour. Assuming that the garbage collector has to look at 1% of the disk in order to garbage collect free blocks, the garbage collector consumes 1.5 minutes of random access disk time. If this is spread over the hour, the collector would consume 2.5% of the disk bandwidth, and garbage would consume an average of 3.6% of the disk.

4.10 Lifetime tree maintenance

As long as the operations which maintain the permanent portions of the lifetime tree run in permanent tasks, there should be no problems related to losing address spaces or bogus pointers.

4.11 Sleeping the system

To gracefully take the system down, abort all temporary tasks, clean all pages, force all cached catalog entries to disk, and take a snapshot.

5. Disk recovery details

5.1 Error detection/correction capabilities

Given the way the system is structured, it seems that hard read errors are just as disastrous as head crashes. So, we have 2 choices: (a) Do read after write verification (at the cost of 10% reduction in expected disk bandwidth). (b) Use a disk controller which has state of the art error correction capabilities such that the probability of a hard read error is significantly less than once every 100,000 hours. This would mean that hard read errors are masked by the media destruction errors, and can be ignored.

We assume that the UDA50 controller has these properties and that EMULEX will have a knock-off for the UDA50 prior to first deliveries. That is, we have chosen strategy #b.

Appendix A

MTBF and MTD for Disk System

A.1 Subsystem MTBFs

Subsystem	MTBF	Comments
R1000 processor	8,000 hrs	Includes the processor, its memory, the sysbus, and single IO adapter, in single processor config.
IO Processor	9,000 hrs	Includes the PDP-11, its memory, fans, power supply, Unibus, etc.
Disk Ctlr	30,000 - 70,000 hrs	
Comm Ctlr	30,000 - 50,000 hrs	
Disk Dr (fxd)	10,000 hrs	Somewhere between 10 and 100% of these failures cause loss or more than last block written.

A.2 Disk subsystem MTBF for various configurations

Configuration	MTBF
1 each of (IOP, ctlr, drive)	5.7 months
2 IOP, 2 ctlr, 4 drive	2.0 months
3 IOP, 4 ctlr, 8 drive	1.1 months
4 IOP, 8 ctlr, 16 drive	0.6 months

A.3 Disk subsystem MTD

We assume that the probability that the drive will cause massive disk data loss far exceeds the probability that an IOP or a disk controller will cause massive data loss. Let MTD stand for Mean Time to Destruction of disk data.

	System MTD	
Single drive MTD	2 drives	16 drives
100,000 hrs (10% of total failures)	5.8 yrs	8.7 mos
50,000 hrs (50% of total failures)	2.9 yrs	3.3 mos
10,000 hrs (100% of total failures)	6.9 mos	0.8 mos

A.4 Disk subsystem MTBF for mirrored configuration

We redefine MTBF to mean failure in one component plus a failure in a second component prior to re-integration of repaired first component. We assume that repair and integration can be performed within 24 hours.

Configuration	MTBF
2 IOP, 2 ctlr, 4 drive	16.7 years
4 IOP, 8 ctlr, 16 drive	14.5 years

Appendix B

Backup and Recovery Times

B.1 Full save times

The following table indicates the time it takes to perform a full save (complete backup) under various assumptions about available media.

We assume a system configuration of 4 250 Mbyte winchester drives. For tape backup, we assume one tape unit in the configuration. For disk backup, we assume one removable media in the configuration.

"Raw copy" refers to the scenario of copying track-by-track. That is, media copy, without regard for its structure.

"Compactifying copy" refers to the scenario of copying address spaces, with the system up and running. In this case, we assume the disk capacity is 75% utilized, and that backup is allowed to consume 2/3 of the source drives random access bandwidth while it is backing up address spaces on that drive. In this case, the backup is limited by the random access capability of the source drives, and the target backup media has little or no effect on the speed of the full save.

Technology	Time/Gbyte	Units/Gbyte
1600 bpi 75 ips vanilla tape raw copy	6.2 hrs	25 tapes
3200 bpi streamer tape raw copy	1.8 hrs	11 tapes
removable disk raw copy	.55 hrs	4 packs
any tape or disk compactifying copy	10.5 hrs	as above

B.2 Incremental times

The time to take an incremental save is calculated from M * T; where "M" is the fraction of used disk capacity which has changed from previous save; and where "T" is the "compactifying copy" time from B.1.

B.3 Restoration times.

Assuming the system is down during restoration, restoring a full save consumes the time indicated by one of the "raw copy" rows of the table in section B.1.

Restoring an incremental save is 33% faster than the time consumed

in creating it.

B.4 Typical backup strategy.

Assume most activity occurs between 8AM and 6PM, Monday through Friday. Assume slack time in the following slots: 12PM .. 1PM and 6PM. . 7PM.

Once each week, take a full save. This would usually be done on a weekend, probably at night. Once each day, take an incremental save, back to the previous full save. This might be done at 6 AM. At 12PM and 6PM, take an incremental save, back to the morning incremental.

As indicated in B.1, the full save takes from 1.8 hrs to 10.5 hrs, depending upon method. As indicated in B.2, the 6AM incremental would take up to 2.6 hrs at the end of the week, assuming 25% of the used capacity has been changed by the end of the week. Assuming that no more than 5% of used capacity has changed since the 6AM incremental, the 12PM and 6PM incrementals should take at most 0.52 hrs.

B.5 Parallel backup.

It is relatively straight forward for backup to be able to proceed in parallel to independent backup units. Thus, quadrupling the system size needn't increase backup time. Assuming multiples of the above configuration, the real time should stay constant. (Of course, one may need multiple operators to mount the 100 tapes consumed by full save to vanilla tape.)

B.6 Relative cost of backup media

A 250 Mbyte removable media drive is roughly 3-4 times more expensive than a vanilla tape drive. The cost per byte of the disk pack is roughly 6-7 times more expensive than the tape. However, the tape may only have 1/3 the lifetime. So the disk media may only be 2-3 times more expensive.

B.7 Recovery times

The following table indicates the time to recover to a "consistent" state following a drive failure.

Technology	Recovery time
vanilla tape	6.2 + 1.7 + .34 = 8.25 hrs
streamer tape	1.8 + 1.7 + .34 = 3.84 hrs
removable disk	.55 + 1.7 + .34 = 2.59 hrs

B.8 Parallel recovery

In a fashion analogous to that specified in section B.5, recovery can be made to run in parallel, to keep the recovery times from exceeding the above numbers.

B.9 Fast recovery option

This option will cost you 30% more disk capacity. But recovery time is only 25% of the time indicated in section B.7.

B.10 Average amount of work lost

Despite the faster recovery, the average amount of work lost is still quite high. For daytime workers, its 3 hours. For graveyard types, its 6 hours.

Appendix C

Independent Volume Reconstruction

C. The problem

In section 3.5, we discuss the customer's options for recoverying from a disk failure. They are: (a) Wait until the drive is fixed. In 90% of the cases, the data was not lost, and we come back up without losing any data. But the system was down for a lengthy period of time. (b) Or, we can immediately recovery the entire system to a previous save point. This gets the system back up (in a minimum of roughly 40 minutes), but at the expense of losing all work that took place after the save point.

We would like to add one additional option: When the drive fails, let the system run without the "down" drive. The system is up during the repair period. If the data was not lost, bring the drive back online. In this case we get the best of both worlds: the system is not down and we do not lose work already done. If the data was lost, we reconstruct the data for just that drive from the save sets. The reconstruction can happen while the system is up. Then we somehow reintegrate the drive back into V-mem.

The remainder of this appendix discusses problems associated with implementing such a capability, and presents solutions to many of the problems. But there are many unresolved issues. We are certainly not presenting a fully-debugged design strategy. (That's why this stuff is stuck in an appendix.)

C.1 General problems

C.1.1 Partitioning V-mem into useful divisions

Somehow, we have to partition V-mem into sections such that stuff used by a particular programmer is mostly in one section. That way, if we lose 1 of 5 sections, 4/5ths of the programmers can still do work (mostly).

C.1.2 Faults for pages on a down drive

When a drive goes down, tasks may be waiting for page fault service from that drive. Additional tasks may later fault for pages on that drive. How do we deal with that? (Do we suspend or create exceptions, or both?) Furthermore, how do we clean pages that belong to the down drive?

C.1.3 Reintegration

Assuming a down drive is repaired, but we find the data is bad, we can certainly reconstruct its contents as of the prior save point. But how do we reintegrate this "old version" back into a running system?

C.2 The volume tree as a possible partitioning

All of the so called "directory packages" live on the root volume. All of the system catalogs live on the root volume. There are other

volumes, called leaves. (We have just a 2-level tree of volumes). Each directory has a single leaf volume associated with it. Certain objects "in the directory" really live on the associated leaf volume (as opposed to the root volume where the directory package itself lives). In fact, the only objects that can be placed on leaf volumes are elaborated tasks, and then only in certain ways.

Assume the following declarations:

```
task type T is ...
type T_PTR is access T;
```

Given a directory package "GLENN", the only way a task of type T can be placed on the leaf volume (for Glenn) is by inserting one of the following definitions into GLENN's directory package:

```
X: T;
P: constant T_PTR := new T;
```

Actually, the above restrictions (on task creation) need only apply to permanent tasks. Presumably, temporary tasks go away during reintegration, and are not a problem. Note that the above definitions imply that task X cannot create a permanent child of type T, without calling the system to insert "Q: constant T_PTR := new T;" into some directory.

C.3 Reintegration: putting an object back into the directory

Suppose T is a container for some data, like a phonelist. I have a phonelist, X. Then the system takes a snapshot. I create a new phonelist, Y, and copy the contents of X into Y. I make some changes to Y. The changes look good. So I delete X. Now, the leaf volume (which contains X and Y) fails, and is restored to the save point taken before Y was created. Do I have X, or Y, neither, or both? On a conventional machine, I would have just X. An answer of neither X nor Y seems terribly unfriendly. Note however, that if X and Y are on different volumes, the conventional machine would leave you with neither X nor Y.

By the above argument, it seems that objects saved by an incremental should reappear when the volume is recovered, even if they were deleted after the incremental was taken.

Under that definition, deleting X from GLENN's directory package must be implemented as follows: The deletion request deletes the address space(s) consumed by the task (named by X), and the entry is removed from the catalog. However, the DIANA node which describes X must remain in a sort of "deleted but not yet expunged" shadow state. Likewise, the control stack slot for X and the tasks participation in the child list (and dependency count) of the GLENN package must remain. When the reconstructed volume (which contains the address spaces for the task that used to be named by X) is reintegrated into the system, the catalog has to get updated (to reference these revived address spaces), and DIANA must be informed, to "undelete" X's declaration. The directory information for X cannot really go away until the save point following its deletion (assuming the rule that the reconstructed volume goes back exactly one save point; otherwise, restore the entire system).

Observe that this deleted but not yet expunged paradigm must apply

to directories themselves. If, in addition to deleting X, I delete the directory package GLENN, reviving X also requires reviving the directory. (Also handled by conventional machine.)

Notes: (A) If we were deleting P instead of X, we would similarly have to delay updating the collection's child list until after the next save point. (B) The task control block identifies the creator, in addition to the parent in the lifetime tree, of the task. Thus, the TCB has a back link to the directory package, in both the X and P cases. This information can be used to simplify finding the DIANA node during reintegration.

C.4 Partitioning DIANA

Unfortunately, the above definition does not go far enough. It seems clear that if we allow all of DIANA to reside on the root node we will have a lousy partition. The DIANA partition needs to correspond to something tangible (from the user's point of view).

So, it seems we need the following rule: The DIANA task for a compilation unit living in directory GLENN must be placed on GLENN's leaf volume (along with the tasks for X and P). And any code segments need to go along with their corresponding DIANA task. The DIANA task would be named by a variable in the directory package, just like X and P; the directory entry might want to be elided.

C.5 DIANA reintegration

Assume that there is one DIANA task which stores the parse tree for a particular compilation unit. When you add semantic information to the tree, you end up with references from one DIANA task to nodes contained in a different DIANA task. How do we reintegrate an "old DIANA task" back into the whole structure?

Of course, this raises the general issue of configuration management. (Which we will try to sidestep.)

Let's assume that "A is different from B" (where A and B are distinct versions of some compilation unit) is defined elsewhere. (If the visible part changed between A and B, they are clearly different. If just the body changed, it is not clear whether they should be considered different; a minor bug fix might want to be considered not different, whereas a major change in function might want to be considered different, even though only the body changed. So, we assume some prior definition.)

Given DIANA tasks R and S, and a semantic reference from R to S, we assume that the representation of the reference is such that if we make a new version of S, S', and S' is different from S, that existing semantic references from R to S are invalidated. R must be resemanticised.

Deleting and reviving compilation units should be handled the same as for the tasks X and P (as described above). If a reconstructed volume contains some DIANA task S, we can stick it back in the directory the same way we did for X and P. If some other DIANA task R (on a different volume) had a semantic reference to S prior to the save point (to which we recovered), but then got changed to reference S' (which is different from S), then all the references from R to S become invalidated (by the previous paragraph).

When the system runs across one of these invalidated references, it could automatically resemanticised R, to create new references that are valid. But wouldn't the user want to know? The user might consider it a favor to be informed that the volume reconstruction caused some work to get lost (namely the change from S to S'), particularly since the user doesn't really know when the save point happened!

There is an analogous problem with code segments. Prior to the volume failure, I have a running program. I like what it does. Because of the reconstruction, one of the code segments used by the program is restored to an old value, which is not "different", but nevertheless now has bugs which were previously fixed. After reintegration, is it really reasonable to have the program still run?

C.6 The effect of partitioning on collections

The volume tree definitions have certain implications for collections. A "leaf collection" is one whose declaration occurred in a task stored on the leaf.

By definition of tasks and leaf volumes, all access variables for a leaf collection live on the same volume as the collection. Furthermore, there can be no references to collection members (created by parameter passing mechanisms, for example) stored on other than the leaf volume.

The type T_PTR is a good example of a "global root collection". In this case, access variables and references may be stored on any volume.

Exiting the scope of a global root collection is not exacerbated by the delayed child list update, since the global root collection must be statically nested.

Any partitioning scheme based on divisions at task boundaries will have the above properties.

C.7 The effect of reintegration on garbage collection

Restoring a volume to some prior state means that a leaf collection may have a different flip state than the rest of the system. This implies that any partitioning scheme must have the property that a collection living on a leaf have all references to its members stored on the same leaf, otherwise you have the incredibly messy problem of one member reference being is one flip state while another reference is in a different flip state.

It also implies that the hardware gadgets for detecting mutator references to collection members must be able to handle different flip states on a per volume basis.

Given the current hardware, one can do this as follows: Apply meaning to 5 more bits of collection names (only). These 5 bits identify the volume on which the collection lives. (All regions of a collection live on the same volume). Then the hardware gadget is programmed using the 5 volume bits, the 3 region bits, and the read/write bit. This achieves the objective of independent flip states, per volume.

Leaf collections may garbage collect and flip at will.

However, root collections are somewhat constrained. One has to keep the save point state around on all the leaf volumes (using the same generation retention mechanism that is used to take a consistent save in the first place) until the next save point. The garbage collector also scans these old generations. This guarantees that volume reconstruction does not revive pointers to members whose space has since been reassigned. Furthermore, one has to synchronize flips with save points, since all references (which may be on different volumes) must be in the same flip state at the same time).

C.8 The effect of reintegration on rendezvous

C.8.1 The problems

A cross volume rendezvous causes difficulty because during the rendezvous the server or client may disappear or be reset to some state prior to entering the rendezvous due to volume reconstruction and subsequent reintegration. This problem seems to divide as follows:

First, given that a client is waiting for the completion of a rendezvous that can no longer complete, it must be given a TASKING_ERROR exception.

Second, given that a server is processing in a rendezvous that can no longer complete, it must be allowed to rendezvous with the ghost in a transparent fashion. An architectural concern arises from the fact that the server may have reference parameters that are out of date; this is a "security problem".

C.8.2 The solution

Use a small integer (about 3 bytes will do) to uniquely identify save points.

Keep track of clients. Given sections 4.2 and 4.3, the catalog gives us a handle on all clients, since their task state is something like WAITING_FOR_RETURN.

When a client initiates a rendezvous, it's state records the following information: (a) A reference to the server task. (b) The number of the most recent save point.

A busy server is one which is not blocked waiting for a rendezvous at call level 0. Call level is defined as follows: A task initially has call level 0. Every time an accept body is entered, the call level is incremented. When an accept body is exited, the call level is decremented.

When the call level is incremented, the identity of the client is recorded in the state of the server. In addition, the client passes its recorded value of save point number. The server records this value along with the reference to the client.

In a cross volume call, the server is required to take all IN parameters by value. IN OUT parameters are passed by value result. OUT parameters are returned by value. When the server returns its results, it is obligated to first verify that the client is still

waiting for rendezvous completion with this server, and that its wait state has the same save point number. In the event that these two conditions are not met, the results are placed in the bit bucket, transparent to the server.

The reintegration procedure is as follows:

Examine all clients (tasks blocked waiting for rendezvous completion). Examine the server on which the client is waiting. The server must exist. It must be at call level 1 or higher. Look through its state. Exactly one of its call levels must correspond to the client and have the identical save point number. If any of the above conditions is not met, give the client a TASKING_ERROR exception.

We don't need to examine busy servers. They will complete their rendezvous with ghosts in a transparent fashion, and there is no "security problem" since the server verifies that the client is still waiting for results (from the same rendezvous) prior to returning them.

We do not take any explicit action for messages in the queue space of a server. After reintegration, it may service stale requests.

C.8.3 Notes

An argument has been advanced that the programmer be required to explicitly program the parameter copying in cross volume calls. The reintegration procedure would scan the entire control stack of all servers, "nilling" any references to the address spaces of a client that is no longer blocked waiting for the particular rendezvous. The programmer is required to explicitly handle the exceptions that result from referencing a non existent address space or a parameter whose reference has been "nilled". See section 4.8.18 for arguments to the