PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 5 :		(11) International Publication Number:
G06F 11/20, 11/14	A1	WO 93/15461
		(43) International Publication Date: 5 August 1993 (05.08.93)

(21) International Application Number: PCT/US93/00618

(22) International Filing Date: 22 January 1993 (22.01.93)

(30) Priority data:
07/824,134 22 January 1992 (22.01.92) US

(71) Applicant: UNISYS CORPORATION [US/US]; Township Line and Union Meeting Roads, P.O. Box 500, Blue Bell, PA 19424 (US).

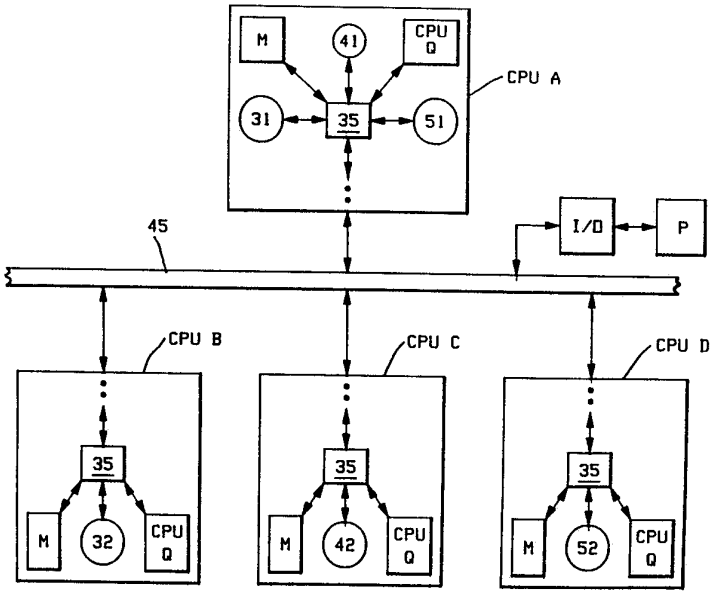
(72) Inventor: GLEESON, Barry, John ; 3380 Waverley Street, Palo Alto, CA (US).

(74) Agent: STARR, Mark, T.; Unisys Corporation, Township Line and Union Meeting Roads, P.O. Box 500, Blue Bell, PA 19424 (US).

(81) Designated States: JP, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: FAULT TOLERANT COMPUTER SYSTEM WITH PROVISION FOR HANDLING EXTERNAL EVENTS



(57) Abstract

A fault tolerant computer system employing primary tasks (31, 41, 51) and corresponding backup tasks (32, 42, 52). The system operates to provide fault tolerant operation even where uncontrolled external events (Table C and D) may occur whose time of occurrence may affect task performance. For this purpose, external event data is stored (in CPUQ) for each external event occurring during performance of a primary task which indicates the event type and the relationship between the occurence of the external event and the occurence of a predetermined primary task event, such as a memory (M) access operation. This external event data is sent to eac respective backup task (32, 42, 52) along with messages transmitted to the respective primary task (31, 41, 51). In the event a primary task fails, (Table E) the backup task (32, 42, 52) will replay (Table F) the failed primary task by processing these transmitted messages while using the transmitted external event data to redeliver each external signal to the backup task at an appropriate time which will assure that the backup task properly recovers the primary task.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	FR	France	MR	Mauritania
AU	Australia	GA	Gabon	MW	Malawi
BB	Barbados	GB	United Kingdom	NL	Netherlands
BE	Belgium	GN	Guinea	NO	Norway
BF	Burkina Faso	GR	Greece	NZ	New Zealand
BG	Bulgaria	HU	Hungary	PL	Poland
BJ	Benin	IE	Ireland	PT	Portugal
BR	Brazil	IT	Italy	RO	Romania
CA	Canada	JP	Japan	RU	Russian Federation
CF	Central African Republic	KP	Democratic People's Republic of Korea	SD	Sudan
CG	Congo			SE	Sweden
CH	Switzerland	KR	Republic of Korea	SK	Slovak Republic
CI	Côte d'Ivoire	KZ	Kazakhstan	SN	Senegal
CM	Cameroon	LI	Liechtenstein	SU	Soviet Union
CS	Czechoslovakia	LK	Sri Lanka	TD	Chad
CZ	Czech Republic	LU	Luxembourg	TG	Togo
DE	Germany	MC	Monaco	UA	Ukraine
DK	Denmark	MG	Madagascar	US	United States of America
ES	Spain	ML	Mali	VN	Viet Nam
FI	Finland	MN	Mongolia		

FAULT TOLERANT COMPUTER SYSTEM WITH PROVISION FOR HANDLING EXTERNAL EVENTS

BACKGROUND OF THE INVENTION

This invention relates to improved means and methods for providing fault tolerance in a data processing system.

As computer systems increase in speed, power and complexity, it has become of increasing importance to provide fault tolerance in such systems to prevent the system from "going-down" in the event of hardware and/or software failure. However, providing fault tolerant capabilities in a computer system has proven to be expensive as well as introducing significant performance penalties.

A basic way of achieving fault tolerance in a data processing system is to provide each task (also called a process) with a backup task such that, if the primary task fails, the backup task is automatically able to recover and continue execution. For example, a primary task and its backup task could be provided using a pair of simultaneously executing CPUs (central processing units) intercoupled such that, if one fails, execution continues on the other. It will be appreciated that the need to provide such duplicate hardware is a very expensive way of achieving fault tolerance, particularly since the simultaneously operating duplicate hardware cannot be used to provide additional data processing power.

One known approach for avoiding hardware duplication is to provide a first CPU for the primary task, and a second CPU for the backup task, the backup becoming active to recover and continue execution only if the primary fails. Until then, the backup CPU can do other processing. In order to assure that the backup process can take over in the event the primary process fails, this known approach provides for a checkpointing operation to occur whenever the primary data space changes. This checkpointing operation copies the primary's state and data space to that of the backup so that the backup task will be able to continue execution if the primary task fails. However, the frequent checkpointing required by this approach detrimentally affects performance and also uses up a significant portion of the added computing power.

Another known approach is disclosed in U.S. Patent No. 4,590,554. Although this approach also uses checkpointing, it provides the advantage of employing a fault tolerant architecture which significantly reduces the frequency of checkpointing. However, the approach has the disadvantage of requiring a message transmission protocol

which is essentially synchronous in that it requires messages to be transmitted to primary and backup processors substantially simultaneously. Also, the disclosed approach in the aforementioned patent has the additional disadvantage of requiring atomic transmission, wherein transmittal of a message by a task is not allowed unless the receiving tasks and all backups indicate they are able to receive the message. Furthermore, no receiving task is allowed to proceed until all receiving tasks and backups have acknowledged receipt of the message. These message transmission protocol requirements introduce constraints that add complexity to the system, as well as having a significant detrimental effect on performance.

Similar approaches to that disclosed in the aforementioned patent No. 4,590,554 are described in an article by A. Borg, et al., "A Message System Supporting Fault Tolerance," Ninth Symposium on Operating Systems Principles (Breton Woods, N.H., Oct. 1983), Pages 90-99, ACM, New York, 1983, and in an article by A. Borg, et al., "Fault Tolerance Under UNIX," ACM Transactions on Computer Systems, Vol. 7, No. 1, February 1989, pages 1-24.

A significantly improved approach over that disclosed in the aforementioned Patent No. 4,590,554 is described in my aforementioned patent application Serial No. 07/521,283, which provides a fault tolerant data processing system having the advantages of Patent No. 4,590,554, while reducing message transmission restraints. More particularly, the system of Serial No. 07/521,283 requires neither simultaneity nor atomicity of transmission in order to provide fault tolerant operation, whereby enhanced performance is achieved. This system of Serial No. 07/521,283 will henceforth be referred to as the Flexible Fault Tolerant System.

SUMMARY OF THE PRESENT INVENTIONS

A broad object of the present invention is to provide improved means and methods for achieving fault tolerance in a data processing system.

A more specific object of the invention is to provide a fault tolerant system similar to that disclosed in the aforementioned Patent No. 4,590,550 or similar to the Flexible Fault Tolerant System, wherein a process (task) is able to accommodate an uncontrolled external event in situations where the task has little or no control over when this external event occurs, and where the behavior of the task may differ depending on the time of occurrence of this external event.

A further object of the invention is to provide a fault tolerant data system, in accordance with the foregoing objects, which is implemented in a relatively simple and economical manner.

In a particular preferred embodiment of the invention, a fault tolerant system similar to the Flexible Fault Tolerant System is provided, wherein additional hardware is provided for counting "write" references to memory. This "write" memory reference data count is treated as part of the task's context. The times of occurrence of external events are associated with this memory reference "write" data count such that, during recovery of a task by its backup, each uncontrolled external event is redelivered to the backup at the same logical point as it was delivered to the primary task, thereby assuring that a failed task will properly recover, despite the occurrence of such external events.

The specific nature of the invention as well as other objects, features, advantages and uses thereof will become evident from the following detailed description of a preferred embodiment along with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a block diagram of a prior art fault tolerant system taken from U.S. Patent No. 4,590,554.

Fig. 2 is a generalized block diagram of one of the task processors in Fig. 1, also taken from U.S. Patent No. 4,590,554.

Fig. 3 is a block diagram of a preferred embodiment of a fault tolerant system.

Fig. 3A is a generalized representation of a task in Fig. 3.

Figs. 4-9 are flow charts illustrating various operating examples for the preferred embodiment of Fig. 3.

INTENTIONALLY

LEFT

BLANK

DETAILED DESCRIPTION

Like numeral and characters designate like elements throughout the figures of the drawings.

Summary of U.S. Patent No. 4,590,554 (Figs. 1 and 2)

It will be helpful in understanding the contribution of the present invention and the detailed description to be provided herein to initially summarize the construction and operation of the embodiment disclosed in the

aforementioned U.S. Patent No. 4,590,554, the contents of which are incorporated herein. For this purpose, reference is directed to Figs. 1 and 2 herein which respectively correspond to Figs. 1 and 2 of Patent No. 4,590,554.

Fig. 1 illustrates a parallel computer system PCS comprised of primary processors 11 and 21, their respective associated backup processors 12 and 22, a common memory CM, and an interconnecting message bus MB.

Fig. 2 illustrates one of the processors in Fig. 1, and comprises a read counter RC, a write counter WC, a message queue MQ and the remainder of the processor RP. A primary processor (11 or 21) uses only the read counter RC, and a backup processor (12 or 22) uses only the write counter WC. Both RC and WC are initialized to zero at the start of operations. During operation, the read counter RC in each primary processor accumulates a count of the number of messages which its respective primary processor (11 or 21) reads from its message queue MQ to the remainder of the processor RP. The write counter WC in each backup processor (12 or 22) accumulates a count of the number of messages transmitted by its respective primary processor (11 or 21).

The operation described for the embodiment disclosed in U.S. Patent No. 4,590,554 assumes that a first process (task) is being executed on primary processor 11 and a second process (task) is being executed on primary processor 21. Each message transmitted by a primary processor (e.g., 11) is sent substantially simultaneously to three processors, the destination primary processor (e.g., 21), the backup processor 22 of the destination processor 21, and the backup processor 12 of the transmitting processor 11. Only when all three processors have received the message and so acknowledged is the message transmission considered complete (atomicity). Both the destination processor 21 and its backup processor 22 load the message

into their respective message queue MQ. However, the transmitting processor's backup processor 12 uses the received message merely to increment its write counter WC, the message thereafter being discarded. Each time a primary processor processes a received message, it increments its read counter by one.

Checkpointing is automatically initiated between primary and backup processors in the embodiment of U.S. Patent No. 4,590,554 when the number of messages in the message MQ of a backup processor becomes too large. Checkpointing causes the backup process to be brought to the same state as its primary process, including making their data spaces identical. In addition, checkpointing causes the primary process to zero its read counter RC after sending the accumulated read count to its backup process. It will be remembered that this read count RC indicates the number of messages read by the primary process from its message queue MQ since the start or the last checkpointing. The backup process uses this read count during checkpointing to discard the same number of messages from its message queue MQ. Thus, if the primary process should fail, the backup process will not process messages already processed by the primary process.

As an example of the operation of the embodiment of aforementioned Patent No. 4,590,554, it will be assumed that primary processor 21 fails. In such a case, its backup processor 22 will start from the point of the last checkpointing (or from the start), and begin processing the messages in its message queue MQ (Fig. 2). These are the same messages which were sent to the primary processor 21. In order to provide proper recovery, the backup processor 22 is prevented from retransmitting any messages that its failed primary processor 21 transmitted before failure. This is accomplished by using the accumulated count in the write

counter WC of the backup processor 22, which it will be remembered corresponds to the number of messages sent by its respective primary processor 21. Each item an output message is produced by the backup processor 22 during recovery, this rite counter WC is decremented by one. The backup processor 22 is allowed to transmit a message only after the write counter WC has reached zero. The backup processor 22 is thus brought up to the state of its failed primary processor 21 and can now take over processing of the process (task) which the failed primary processor 21 had been executing.

Description of Flexible Fault Tolerant System (Figs. 3-9)

Initially, it will be helpful to consider some insights relevant to fault tolerant operation.

If a task receives a message, and then fails immediately, one may proceed as if the task failed before receiving the message.

In fact, one may choose to proceed as if the task failed before receiving the message until the task performs some action that will persist after the failure, for example, the task writes to a disk or terminal, or sends a message to another task that survives the failure. Since a CPU failure causes the failure of all tasks resident in that CPU, a message sent to another task running in the same CPU is not an action that will persist after a CPU failure, unless the receiver in turn performs a "persistent action".

More generally, if a CPU fails, it is important that all devices and tasks external to that CPU (other CPUs, disks, terminals, etc.) agree on the state of the CPU at the time of failure. It is not important whether the agreed upon state is the actual state of the CPU at the time of the failure.

The failed CPU may in fact have performed many additional processing steps, but no persistent actions, in

which case the backups need not take them into account in order to recover properly. Recovery can thus commence at the agreed on state, and recompute the processing steps up to and beyond the actual state at the time of failure. In fact, recovery may perform different processing steps than the original CPU, but this is transparent to the user, as will be understood from the above insights, since no consequences of the original processing steps are visible.

As shown in Fig. 3, three primary tasks 31, 41 and 51 are implemented on CPU A, and communicate with each other and with an outgoing CPU Queue via an internal message network 35, which may be a conventional bus arrangement. Although only three primary tasks 31, 41 and 51 are illustrated on CPU A, it will be understood that additional tasks could be provided.

As also shown in Fig. 3, task 31 on CPU A is provided with a backup task 32 implemented on CPU B, task 41 on CPU A is provided with a backup task 42 implemented on CPU C, and task 51 on CPU A is provided with a backup task 52 implemented on CPU D. More than one backup task could be implemented on the same CPU. Each CPU includes a memory M coupled to the internal communication network 35 which provides a respective data space for each task implemented on the CPU. CPU's B, C and D may each have a CPU Queue (as does CPU A), but it is not used if the CPU contains only backup tasks. Communication between CPUs A, B, C and D14 is provided by an external communication network 45 which may take various forms known in the art, such as indicated by the message BUS MB in the aforementioned patent No. 4,590,554. As shown in Fig. 3, peripherals P are also coupled to the external communication network 45 via an I/O. The peripheral P may, for example, include one or more disk drives.

Each of the primary tasks 31, 41, 51 and their respective backup tasks 32, 42, 52 will now be considered in more detail. One skilled in the art will understand from the description herein that different arrangements can be used with additional tasks and CPUs. For the purposes of the embodiment being considered herein, it will be assumed that primary tasks 31, 41, 51 which are all on CPU A, receive messages only from each other, via internal communication network 35, and not from outside of their respective CPU A. It will also be assumed that message deliveries from tasks 31, 41, 51 outside of CPU A are only to their respective backup tasks 32, 42, 52 on CPUs B, C, D respectively, via external communication network 45. As will be evident to those skilled in the art, the structure and operations described herein for a task are implemented by its respective CPU.

As shown in Fig. 3A, each task (31, 32, 41, 42, 51, 52 in Fig. 3) includes a message queue MQ for receiving and storing messages. Each task also includes a read counter RC and a write counter WC. If the task is a primary task (such as 31, 41, 51 in Fig. 3), then only the read counter RC is used, this use being to accumulate a count of the number of messages read by the primary task from its message queue MQ. If, on the other hand, the task is a backup task (such as 32, 42 and 52 in Fig. 3), then only the write counter WC is used, this use being to accumulate a count of the messages sent by its respective primary task (31, 41, 51 in Fig. 3).

The operation of the message queue MQ, read counter RC and write counter WC may typically be as previously described herein with respect to the aforementioned Patent No. 4,590,554. Also, checkpointing and recovery by a backup task may likewise typically be provided as described in aforementioned Patent No. 4,590,554, except for the differences pointed out herein resulting from taking

advantage of the previously considered "insights."The operation of the embodiment illustrated in Figs. 3 and 3A will now be considered in more detail. As mentioned previously, it is assumed that primary tasks 31, 41, 51 on CPU A communicate only with each other, via internal communication network 35, and that respective backup tasks 32 are provided on CPUs B,C and D, respectively. Each message sent by a primary task (31, 41 or 51) typically includes an associated task address which is used by the internal communication network 35 to direct the message to the indicated task. Messages required to be sent to backup tasks (32, 41, 52 on CPUs B,C,D, respectively) are sent by the internal communication network 35 to the outgoing CPU Queue, which operates in a first-in, first-out (FIFO) manner.

An important feature is that, by taking advantage of the insights considered earlier herein, a primary task which transmits a message to another task on the same CPU is allowed to continue its processing immediately, so long as delivery of the message to this other task and the respective CPU Queue are assured, even though corresponding backup messages in the CPU Queue have not been sent to the backup tasks, thereby providing high speed processing. Unlike in the aforementioned Patent No. 4,590,544, these backup messages can be sent to the appropriate backup tasks via the external communication net work 45 when convenient to do so. This applies so long as a primary task does not perform a persistent action, which it will be remembered is an action taken by a primary task which will persist after failure, such as when the task writes to a disk or terminal, or sends a message to another task that survives the failure.

When a primary task (21 or 31) is required to perform a persistent action, the primary task first checks the outgoing CPU Queue to determine whether all backup messages corresponding to messages already processed by the

task have been delivered to the backups. If the delivery of all such required messages has been assured, the task performs the persistent action and continues processing. If not, the primary task initiates the required delivery to the backups, after which the primary task then performs the persistent action and continues processing. The task may again continue processing without being concerned about delivery of processed messages to their backups until the next persistent action is required to be performed. It will be understood that various techniques well known in the art may be employed for assuring the delivery of a transmitted message, such as for example, by using acknowledgment signals, handshaking, echoes, error checking, or other appropriate means.

Various examples illustrative of operations of Fig. 3 will next be presented. These examples are presented in summarized form in the flow charts provided in Figs. 4-9. These flow charts also indicate the point in the flow corresponding to the state tables included for each example. Additionally, it will be helpful to compare these examples and tables to those presented in the aforementioned Patent No. 4,590,554.

The examples presented below involve only primary tasks 31 and 41 on CPU A, and respective backup tasks 32 and 42 on CPU B and CPU C, respectively. Accordingly, only these tasks and CPUs are referred to in these examples. In addition, since only CPU A need have a CPU Queue for these examples, references to a CPU Queue refer to the CPU Queue of CPU A. In addition, it is assumed that appropriate provision is made for assuring the delivery of transmitted messages, as indicated above.

Example 1 (Fig. 4)

TABLE I below shows the start state of the write counter WC, the read counter RC, the message queue MQ and the CPU Queue for primary tasks 31, 41 on CPU A, and their respective backup tasks 32, 42 on CPUs B and C, respectively.

TABLE I (Example 1, Fig. 4):

Task/CPU	Write Counter WC	Read Counter RC	Message Queue MQ	CPU-Queue
31/A	unused	0	empty	empty
32/B	0	unused	empty	
41/A	unused	0	empty	
42/C	0	unused	empty	

Assume that primary task 31 transmits three messages M1, M2, M3 to primary task 41, which are stored in task 41's message queue MQ. These messages are also stored in the CPU 1 for later delivery to backup CPUs B and C. Task 31 may continue its processing even though messages M1, M2, M3 are not transmitted to backup CPUs B and C which contain backup tasks 32 and 42, respectively. CPU A may transmit messages M1, M2, M3 at its leisure so long as no persistent action is required by primary tasks 31 or 41. For this example, it is assumed that CPU A does not transmit these messages M1, M2, M3 at this time. The result is shown in TABLE II below:

TABLE II (Example 1, Fig. 4):

Task/CPU	Write Counter WC	Read Counter RC	Message Queue MQ	CPU-Queue
31/A	unused	0	empty	M1,M2,M3
32/B	0	unused	empty	
41/A	unused	0	M1,M2,M3	
42/C	0	unused	empty	

Next, task 41 reads M1 and M2 stored in its message MQ, processes them, and advances its read counter RC to two to indicate that two messages have been processed. The result is shown in TABLE III below:

TABLE III (Example 1, Fig. 4):

Task/CPU	Write Counter WC	Read Counter RC	Message Queue MQ	CPU-Queue
31/A	unused	0	empty	M1,M2,M3
32/B	0	unused	empty	
41/A	unused	2	M3	
42/C	0	unused	empty	

In response to messages M1 and M2, task 41 generates two messages M4 and M5, and sends them to task 31. Messages M4 and M5 are stored in task 31's message queue MQ and are also stored in the CPU Queue for later delivery to CPUs B and C. The result is shown in TABLE IV below:

TABLE IV (Example 1, Fig. 4):

Task/CPU	Write Counter WC	Read Counter RC	Message Queue MQ	CPU-Queue
31/A	unused	0	M4,M5	M1,M2,M3,M4,M5
32/B	0	unused	empty	
41/A	unused	2	M3	
42/C	0	unused	empty	

Assume that CPU A fails at this point, taking down primary tasks 31 and 41. Backup tasks 32 and 42 agree CPU A was in a state such that no messages were sent or processed by primary tasks 31 and 41 (since none were sent by the CPU Queue of CPU A). Backup tasks 32 and 42 thus replay based on this agreed on state, starting from the last known state, which is the initial state. Thus, the entire processing up to this point is correctly repeated from the initial state by backup tasks 32 and 42 which communicate with each other via external communication network 45. Note that successful recovery is achieved even though the state of CPU A prior to its failure (TABLE IV) was in fact very different from that agreed to by backup tasks 32 and 42.

Example 2 (Fig. 5):

The beginning state of this example is represented by TABLE IV from example 1 above, which shows the state prior to CPU A's failure. This Example 2 assumes that CPU A transmits message M1 in its CPU Queue to backup tasks 32 and 42 on CPUs B and C respectively, before CPU A fails. Message M1 is thus stored in backup task 42's message Queue MQ and backup task 31's write counter WC is advanced to one to indicate one message sent by its respective primary task 41. The result of this transmission by CPUA V is shown in TABLE V below:

TABLE V (Example 2, Fig. 5):

Task/CPU	Write Counter WC	Read Counter RC	Message Queue MQ	CPU-Queue
31/A	unused	0	M4,M5	M2,M3,M4,M5
32/B	1	unused	empty	
41/A	unused	2	M3	
42/C	0	unused	M1	

If CPU A now fails, backup tasks 32 and 42 both agree that CPU A was in a state where only M1 had been sent by primary task 31 to primary task 41. Recovery by backup tasks 32 and 42 is thus performed based on this agreement with tasks 32 and 42 restarting from the last known state (the initial state). This recovery may typically be provided as described in connection with the aforementioned patent 4,590,554. It will thus be understood that, when task 32 regenerates M1 during recovery, its write counter WC (which is at 1 as shown in TABLE V above), is decremented by one to zero, and M1 is discarded. When M2 and M3 are regenerated by backup task 32, they are transmitted normally to task 42 via the external communication network 45, since task 32's write counter WC is now zero. When task 42 restarts and attempts to process its first message, it is given the original message M1, stored in its message queue MQ (TABLE V above). Since message queue MQ is now empty, further message reads by backup task 42 use the regenerated M2 and M3 transmitted from recovering backup task 32.

Example 3 (Fig. 6):

The beginning state of this example is shown by TABLE III from Example 1 above. This Example 3 assumes that task 41 needs to perform a persistent action at this time, such as a write-to-disk (this disk may typically be located

in peripheral P in Fig. 3). Before the disk is written, all messages processed in CPU A must be transmitted to their respective backup tasks. Thus, messages M1 and M2 (which have been processed) must be transmitted to CPUs B and C containing backup tasks 32 and 42 before the write-to-disk, since M1 and M2 have been processed (by task 41). To insure that messages M1 and M2 are sent before the write-to-disk is performed) a marker D is stored in the CPU Queue at a position at least after M1 and M2 so that D is not reached for performance until after M1 and M2 have been sent. The result of storing D in the CPU Queue is shown in TABLE VI below:

TABLE VI (Example 3, Fig. 6):

Task/CPU	Write Counter WC	Read Counter RC	Message Queue MQ	CPU-Queue
31/A	unused	0	empty	M1,M2,D,M3
32/B	0	unused	empty	
41/A	unused	2	M3	
42/C	0	unused	empty	

Note with respect to TABLE VI above that D could be placed in the CPU Queue at any point after M1 and M2 (for example, after M3) since sending M3 along with M1 and M2 will not interfere with recovery.

In order to permit primary task 41 to perform the write-to-disk, CPU A now transmits M1 and M2 from its CPU Queue to CPUs B and C. Messages M1 and M2 are thus stored in the message queue MQ of backup task 42 on CPU C, and the write counter WC of backup task 32 is advanced to 2 to indicate that two messages (M1 and M2) have been sent by its respective primary task 31 on CPU A. The result is shown in TABLE VII below:

TABLE VII (Example 3, Fig. 6):

Task/CPU	Write Counter WC	Read Counter RC	Message Queue MQ	CPU-Queue
31/A	unused	0	empty	D,M3
32/B	2	unused	empty	
41/A	unused	2	M3	
42/C	0	unused	M1,M2	

Task 41 now deletes the D entry from CPU A's queue, and performs the write-to-disk.

In order to prevent task 41's backup task 42 on CPU from repeating the write-to-disk if CPU A should fail, the performance of the write-to-disk by primary task 41 also results in a message being sent to CPU C which causes backup task 42's write counter WC to be advanced to 1. The result is shown in TABLE VIII below:

TABLE VIII (Example 3, Figs. 6 and 7):

Task/CPU	Write Counter WC	Read Counter RC	Message Queue MQ	CPU-Queue
31/A	unused	0	empty	M3
32/B	2	unused	empty	
41/A	unused	2	M3	
42/C	1	unused	M1,M2	

Assume that task 41 next reads M3 from its message queue MQ, processes M3, and then replies by sending messages M4 and M5 to task 31, which are stored in task 31's message queue MQ and also in the CPU Queue. The result is shown in TABLE IX below:

TABLE IX (Example 3, Fig. 7):

Task/CPU	Write Counter WC	Read Counter RC	Message Queue MQ	CPU-Queue
31/A	unused	0	M4,M5	M3,M4,M5
32/B	2	unused	empty	
41/A	unused	3	empty	
42/C	1	unused	M1,M2	

If CPU A fails at this point (TABLE IX above), both CPUs B and C agree with respect to CPU A that Messages M1 and M2 have been sent, and that the write-to-disk is done. The fact that task 41 processed M3 and sent M4 and M5 to task 31 before the failure is irrelevant to satisfactory recovery since no further persistent action occurred prior to CPU's failure. Recovery thus proceeds normally in the manner previously described. Since no checkpointing has yet occurred, recovery starts from the initial state (TABLE I). More specifically, with respect to backup task 32, messages M1 and M2 generated by task 41 during recovery are not sent but discarded, since write counter WC will not have decremented to "0" until after M2 is regenerated. With respect to backup task 42, messages M1 and M2 in its message queue MQ will be processed as occurred for the primary task 41 in now failed CPU A. When recovering backup task 41 reaches the point at which the write-to-disk is to be performed (which it will be remembered was performed by primary task 41), this write-to-disk operation is prevented from being performed again as result of task 42's write counter WC being "1" at this time. It is only after task 42's write counter is decremented to "0" (following this disk-to-write prevention) that messages are sent by task 42. Accordingly since WC will thus be "0" when messages M4 and M5 are generated by task 42, they will be sent to task 32,

thereby achieving recovery to the point reached prior to CPU A's failure. Processing then continues beyond the recovery point using backup tasks 32 and 42 communicating via external communications network 45.

Example 4 (Fig. 8):

The purpose of this Example 4 (and Example 5) is to demonstrate checkpointing in the embodiment of Fig. 3, and assumes a beginning state corresponding to TABLE IV from Example 1 above.

Assume that, after reaching the state shown in TABLE IV above, task 41 initiates a checkpointing operation. This is a persistent action, since checkpointing requires that state information about task 41 be transmitted outside of CPU A. Accordingly, task 41 places the checkpointing data (or an appropriate checkpointing marker CK) in the CPU Queue at a position at least after M1 and M2, since they have been processed. The result is shown in TABLE X below:

TABLE X (Example 4, Fig. 8):

Task/CPU	Write Counter WC	Read Counter RC	Message Queue MQ	CPU-Queue
31/A	unused	0	M4,M5	M1,M2,M3,M4,M5,CK
32/B	0	unused	empty	
41/A	unused	0	M3	
42/C	0	unused	empty	

Note in TABLE X above that task 41's read counter RC has been zeroed since, as far as task 41 is concerned, the required checkpointing has already occurred. Also note that both tasks 31 and 41 can proceed with processing without concern as to when the checkpointing data is actually sent to its backup task 42 in CPU C, so long as any subsequently

occurring persistent actions are delayed until after the checkpoint data is transmitted to its respective backup. Also note in TABLE X that CK was placed in the CPU Queue after M5, rather than directly after M1 and M2, which means that M3, M4 and M5 as well as M1 and M2 will be transmitted before the checkpointing data CK. This will not cause any problem, since CPU A is able to transmit messages from its CPU Queue at its convenience (as mentioned earlier), unless a persistent action is encountered, in which case processed messages have to be transmitted to their respective backups before the persistent action can be performed.

Assume for the purposes of Example 4 that CPU A now begins to transmit M1 through M5 to backup tasks 32 and 42 on CPUs B and C, respectively, but that CPU A fails after successfully transmitting M1, M2, M3, M4 so that neither M5 nor the checkpointing data CK are transmitted. The resulting state just prior to failure is shown in TABLE XI below:

TABLE XI (Example 4, Fig. 8):

Task/CPU	Write Counter WC	Read Counter RC	Message Queue MQ	CPU-Queue
31/A	unused	0	M4,M5	M5,CK
32/B	3	unused	M4	
41/A	unused	0	M3	
42/C	1	unused	M1,M2,M3	

Backup tasks 32 and 42 on CPUs B and C, respectively, initiate recovery from the initial state (TABLE I) based on their agreed on perceptions that only messages M1, M2, M3, M4 were transmitted, and that checkpointing has not yet occurred.

Example 5 (Fig. 9):

This Example assumes an initial state corresponding to TABLE X of Example 4 above. However, instead of failing after transmitting M1-M4, as described in Example 4, this Example 5 assumes that CPU A's transmission of M1-M5 and CK is successful, resulting in TABLE XII below:

TABLE XII (Example 5, Fig. 8):

Task/CPU	Write Counter WC	Read Counter RC	Message Queue MQ	CPU-Queue
31/A	unused	0	M4,M5	empty
32/B	3	unused	M4,M5	
41/A	unused	0	M3	
42/C	0	unused	M3	

As will be remembered from the previous discussion, checkpointing brings the backup task 42 to the same state as its primary task 41, as well as making their data spaces identical in their respective memories M.

It will be understood that while messages M1 through M5 and CK are being transmitted, CPU A is free to continue processing further work for primary tasks 31 and 41, provided that further persistent actions are delayed until after the checkpointing data has been successfully transmitted.

If a failure of CPU A should subsequently occur, backup task 32 will recover from START and backup task 42 will recover from the above checkpoint.

Handling External Events (TABLES A - F)

In the previously described systems, inputs to a task are provided by messages. Calls to the operating system may be treated as messages and, thus, are readily handled by

these systems. However, task behavior may be affected in other ways which are not easily handled by these systems. For example, the state of a task may be changed by an uncontrolled external event where the task has little or no control over when the external event occurs, and where the behavior of the task may differ depending on the time of occurrence of this external event.

For example, such an external event may occur because of the action of an interrupt handler, or as a result of another process writing to a common memory. The following examples, although simplistic, illustrate how such an uncontrolled external event may affect the behavior of a task.

For this purpose, assume that a task executes the program illustrated in TABLE A. If no uncontrolled external event occurs during performance of this program, the program output (occurring at program step PC=2 in TABLE A) will be as shown in TABLE B.

It will be understood from the previous descriptions of Patent No. 4,590,554 and the Flexible Fault Tolerant System that, if the task should fail, the Program Output of TABLE B can be accurately reproduced using a backup task running the same program and initialized in the same manner.

Now assume that the above program is run again with the difference that an uncontrolled external event (such as might be produced by another process writing to a common memory) causes a memory action of MEM(STATE <- GREEN) to occur while the task program above is being run. If this externally caused memory action occurs after the second RED output in TABLE B (i.e., between Program Output lines 3 and 4 in TABLE B), and between program steps PC=2 and PC=3 in TABLE A , then the program will proceed to PC=5, which will

change the memory state back to MEM(STATE) = RED). The resulting Program Output will then be as shown in TABLE C.

On the other hand, if the external event MEM(STATE <- GREEN) were to have occurred after the first GREEN Program Output (i.e., between Program Output Lines 2 and 3 in TABLE B) the original Program Output shown in TABLE B would not be changed, since the external event would not affect the state of the memory at PC=2 in TABLE A. The original Program Output in TABLE B. would likewise not be changed if the external event were to occur between program steps PC=3 and PC=4 in TABLE A , since this also would not affect the state of the memory at PC=2 in TABLE A.

Clearly then, the time at which an uncontrolled external event occurs may affect how a task will perform. Thus, if a backup task is to properly replay a failed primary task which can be affected by such uncontrolled external events, provision has to be made to appropriately account for these external events during task performance. The present invention provides a particularly advantageous way of solving this problem.

The approach employed by the preferred embodiment of the present invention for handling uncontrolled external events in a fault tolerant system, such as previously disclosed herein, will next be described. The basic approach is to relate the occurrence of these external events to particular events occurring in a primary task. These particular events are treated as part of the task context so that, during recovery of a failed task, each external event can be redelivered during playback of the backup task at the same logical point as it occurred during performance of the primary task, thereby assuring that the backup will properly recover. In the preferred embodiment described herein, this is accomplished by providing for counting "write" data references to memory, the resulting "write" counts being part

of the task context. Such memory "write" counting can readily be provided by these skilled in the art. For example, if the CPU employs Motorola 88000 chips, this counting can be performed by counting completed memory store (write) instructions.

Reference is now directed to a preferred embodiment of the present invention illustrated in Fig. 10, which is basically similar to Fig. 3, except that CPU A, CPU B and CPU C in TABLE B each have added thereto a memory reference counter MRC and a memory reference counter compare register MRCCR which are used in performing the memory "write" counting function and in providing recovery. In each CPU, MRC and MRCCR may communicate with tasks 31, 41, 51 and memory M via internal communication network 35.

During operation of a primary task, each "write" data reference to memory M causes MRC to be incremented by one, except when a task performs a call to the operating system and starts executing operating system code, the task remaining asleep until the operating system call is completed. It is advantageous to zero MRC after each operating system call, as well as after each checkpoint, since this results in smaller counts, and thus reduces the possibility of a counter overflow. Zeroing MRC after each operating system call does not create any problem with respect to memory "write" counting, since system calls are treated as messages.

When an uncontrolled external event occurs during performance of a primary task, such as an external signal which changes the task's memory (as exemplified previously), the existing memory reference count in MRC along with an indication of the type of external signal and the task's register context have to be sent to the respective backup task for storage in the memory M of its respective CPU. Using the fault tolerant system of Patent No. 4,590,554

(Figs. 1 and 2), this data has to be sent immediately to the backup, as explained in the summary of this patent. In the Flexible Fault Tolerant System (Figs. 3-9), however, this data is treated like other messages and is placed in the CPU Queue of CPU A. For example, if it is assumed that the state of the fault tolerant system is as illustrated in TABLE II, the occurrence of an uncontrolled external signal with respect to task 31 will cause a marker S indicative of the task and type of signal to be placed in the CPU Queue along with the task 31 register context R, which includes the current count of MRC. This is illustrated in TABLE XIII below:

TABLE XIII (Example 5, Fig. 8):

Task/CPU	Write Counter WC	Read Counter RC	Message Queue MQ	CPU-Queue
31/A	unused	0	empty	M1,M2, M3,S,R
32/B	0	unused	empty	
41/A	unused	0	M1,M2,M3	
42/C	0	unused	empty	

Alternatively, the register context R stored in CPU Queue could merely be a marker, the full register context being stored in task 31's memory space in memory M. Then, when the marker S is reached in the CPU Queue, the stored register context R would then be called up from memory M for transmission to the backup for storage in the backup memory M.

Next, an example is presented of how the recovery of a failed task by a backup is accomplished when an uncontrolled external event occurs during performance of a primary task prior to its failure. For this purpose, it will be assumed that a primary task performs the program

illustrated in TABLE D , which is the same as shown in TABLE A . This example assumes that the CPU on which the primary task is to be run at least contains a memory reference counter MRC (Fig. 10), and that its respective backup CPU at least contains a memory reference counter MRC and a memory reference compare counter register MRCCR.

TABLE E illustrates performance of the program of TABLE D by a primary task on CPU A in Fig. 10. Note that TABLE E indicates, during running of the program, the occurrence of "Events", the "Hardware Context" and the "Program Output". The first "Event" is a CHECKPOINT which sets MRC=0. At this time PC=3 and CF=FALSE, and the state of the memory is MEM(STATE=RED). As shown in TABLE E , the running of the program begins with this CHECKPOINT and then proceeds in an expected manner with the "Program Output" alternating GREEN, RED, GREEN, RED, GREEN and the value of MRC being incremented by one in response to each memory change (write).

As shown in TABLE E , the EXTERNAL SIGNAL MEM(STATE <- GREEN) occurs after MRC=5, when MEM(STATE=RED). At this time, the "Hardware Context" registers have values of PC=8, CF=FALSE, and MRC=5. In the system of 4,590,554, this register context is sent to the respective task backup along with an indication of signal type. In the Flexible Fault Tolerant System, sending of this data to the backup could be delayed by storing this data in the CPU Queue, as illustrated in TABLE XIII.

Continuing with the example in TABLE E , the occurrence of the EXTERNAL SIGNAL MEM(STATE<-GREEN) changes the state of the memory from MEM(STATE=RED) to MEM(STATE=GREEN). Thus, the next program output is GREEN (as was the previous output instead of RED, as it would have been if the EXTERNAL SIGNAL MEM(STATE<-GREEN) had not occurred.

The program then continues. A FAILURE occurs when MRC=8, as shown.

Attention is now directed to TABLE F , which is an example of how a backup would play back the failed task in TABLE E . This example assumes that the register context and the signal type were sent to the backup prior to failure. Since there were no system calls following the CHECKPOINT in TABLE E , the backup starts at this CHECKPOINT, which sets PC=3, CF=FALSE and MRC=0 to correspond to the values which they had at the CHECKPOINT in TABLE E . In addition, MRCCR is set to MRCCR=5, which was the value of the primary task's MRC when the EXTERNAL SIGNAL occurred, and which value was sent to the backup as part of the register context, as explained in connection with TABLE E . It will be understood that MRCCR=5 tells the backup that the external signal (whose type was also sent to the backup) should be delivered after five "write" references to memory.

Thus, as illustrated in TABLE F , backup proceeds normally with MRC being incremented by one for each memory "write". At each incrementing of MRC, a comparison is made with MRCCR=5. When MRC=5, the registers are set in accordance with the register context which existed just prior to the time that the EXTERNAL SIGNAL was delivered during performance of the primary task, this register context having been sent to the backup and stored therein, as explained previously. Accordingly, PC and CF are set to PC=8 and CF=FALSE, respectively; in addition, MRCCR is zeroed so that MRCCR=0. The EXTERNAL SIGNAL MEM(STATE <- GREEN) is then derived from backup storage and delivered, following which playback continues correctly replaying the primary task, as will be evident from a comparison of the Program Outputs of TABLES E and F.

It will be understood that, if a second external signal had been sent to the backup before the primary task

failed, MRCCR would not have been zeroed when MCR=MRCCR in TABLE F , but would have been set to the value MCR had in the primary task at the time that this second signal occurred. Operation with this second external signal would then have been the same as described for the first external signal. In this regard, note in TABLE F that MRC continues to increment beyond MRC=5 so as to provide a count for controlling the time of delivery of other external signals which may have occurred during the performance of the primary task and sent to the backup as previously described. It is further to be noted that, because the count of MRC is part of the task context, the replay illustrated in TABLE F does not need to be continuous. The replay could be preempted at any time, and other tasks performed without affecting the correct redelivery of an external signal.

While the present invention has been described herein with respect to particular preferred embodiments and operational examples, it is to be understood that a wide variety of modifications, additions and extensions in construction, arrangement, use and operation are possible without departing from the scope of the invention. For example, the invention is not limited to employing memory "write" references for providing a count to which uncontrolled external events can be related for accurate playback. For example, both memory "reads" and "writes" could be counted if it were more convenient to do so. Also, program steps could be counted, but this is unduly burdensome in most cases.

It is also to be understood that the reference count set into MRCCR at the start of playback in TABLE F could be used in various other ways for determining when to deliver an external signal. For example, instead of comparing MRCCR with MCR, as in TABLE F , MRCCR could be

counted down to zero in order to indicate when delivery of the external signal should be provided.

It is additionally to be understood that the page fault capability of a CPU's operating system could be used to implement delivery of the external signal in the backup. For example, after MRC=MRCCR in TABLE F, the backup's operating system could be used to produce a page fault on the next memory reference in order to initiate delivery of the external signal.

It is further to be understood that, although the example of TABLE E and F does not include a call to the operating system, such calls would not interfere with proper playback. As mentioned previously, an operating system call can be used to zero MRC during performance of the primary task, since such zeroing can be accurately reproduced during playback. This has the advantage of preventing overflow of MRC.

The above examples of possible modifications and extensions are merely representative and not exhaustive. Accordingly, the present invention is to be considered as including all possible modifications, variations and extensions encompassed by the appended claims.

APPENDIX I

```
Program:
PC    Instruction
1      STATE <- RED              //Initialize memory to "RED"
2      OUTPUT (STATE)            //Print Value of Memory
3      CF <- (STATE=RED)         //Compare memory to RED,
                                   CF=true/false
4     IF CF-FALSE PC <- 7        //If CF=FALSE go to pc = 7
5     STATE <- GREEN             //Set memory to "GREEN"
6     PC <- 2                    //Set pc = 2
7     STATE <- RED               //Set memory to "RED"
8     PC <- 2                    //Set pc = 2
```

Where:

PC - Program Counter Register
CF - Condition Flag register (value TRUE or FALSE)
STATE - Program Memory
RED/GREEN - value Program Memory may contain
"<-" symbolic "is assigned the value"

TABLE A

Program Output

1 RED
2 GREEN
3 RED
4 GREEN
5 RED
•
•
ETC.

TABLE B

Program Output

1 RED
2 GREEN
3 RED
4 RED
5 GREEN
•
•
ETC.

TABLE C

```
Program:
PC    Instruction
1      STATE <- RED              //Initialize memory to "RED"
2      OUTPUT (STATE)            //Print Value of Memory
3      CF <- (STATE=RED)         //Compare memory to RED,
                                   CF=true/false
4      IF CF=FALSE PC <- 7       //If CF=FALSE go to pc = 7
5      STATE <- GREEN            //Set memory to "GREEN"
6      PC <- 2                   //Set pc = 2
7      STATE <- RED              //Set memory to "RED"
8      PC <- 2                   //Set pc = 2
```

Where:

PC - Program Counter Register
CF - Condition Flag register (value TRUE or FALSE)
STATE - Program Memory
RED/GREEN - values Program Memory may contain
"<-" symbolic "is assigned the value"
MRC - Memory Reference Counter
MRCCR - Memory Reference Counter Control Register

TABLE D

Events	Hardware Context	Program Output
--- CHECKPOINT ---	REGS (PC=3, CF= FALSE MRC=0, MRCCR=0)	
	MEM (STATE=RED)	
	MRC=1	GREEN
	MRC=2	RED
	MRC=3	GREEN
	MRC=4	RED
	MRC=5	GREEN
External Signal MEM (STATE<-GREEN)-	MEM (STATE=RED)	
	REGS (PC=8, CF=FALSE, MRC=5, MRCCR=0)	
	MEM (STATE=GREEN)	
	MRC=6	GREEN
	MRC=7	RED
	MRC=8	GREEN
--- FAILURE ---		

TABLE E

--- CKPNT RESTART ---	REGS (PC=3, CF=FALSE, MRC=0, MRCCR=5)	[MRCCR=MRC, from signal]
	MEM (STATE=RED)	
	MRC=1	GREEN
	MRC=2	RED
	MRC=3	GREEN
	MRC=4	RED
	MRC=5	GREEN
	MEM (STATE=RED)	
(MRC=MRCCR) --set regs to --	REGS (PC=8, CF=FALSE, MRC=5, MRCCR=0)	
deliver signal---	MEM (STATE=GREEN)	
	MRC=6	GREEN
	MRC=7	RED
	MRC=8	GREEN
	MRC=9	RED

TABLE F

What is claimed is:

1. In a fault tolerant computer system, a method comprising:

providing a primary task for performing data processing operations, said primary task being subject to receiving one or more external events whose time of occurrence affects the performance of said primary task;

storing external event data for each external signal indicative of the external event type and the relationship between the occurrence of said external event and the occurrence of a predetermined primary task event;

providing a backup task for replaying said primary task in the event of failure thereof;

transmitting messages to said primary task;

also transmitting said messages to said backup task;

additionally transmitting said external event data to said backup task; and

in the event of failure of said primary task, causing said backup task to replay said primary task by processing said messages transmitted thereto while using said external event data transmitted thereto to redeliver each external signal to said backup task such that said backup task will properly recover.

2. In a fault tolerant computer system including a data processor providing at least one task which performs a persistent action and wherein said task is subject to receiving one or more external signals whose time of occurrence may affect the performance of said task, the method comprising:

providing a backup task external to said data processor for backing up said one task,

transmitting messages to said one task;

storing messages transmitted to said one task;

also storing external event data indicative of each external event type and its occurrence relationship to a predetermined primary task event;

transmitting at least certain ones of the stored messages to said backup task subsequently to said storing in a manner such that at least those particular messages which have been processed by said one task are transmitted to said backup task prior to performance of said persistent action;

also transmitting external event data to said backup task prior to performance of said persistent action;

said one task continuing to process messages transmitted thereto so long as the aforementioned transmitting of messages and external event data to said backup task is met; and

in the event of failure of said one task, causing said backup task to process said messages transmitted thereto while using said external event data transmitted thereto to redeliver each external signal to said backup task such that said backup task will properly recover.

3. The method of claim 2, wherein the steps of storing include storing external event data and messages in a queue of said data processor, and wherein the steps of transmitting include transmitting external data and processed messages in said queue to said backup task prior to performing said persistent action.

4. The method of claim 3, including counting the number of messages processed by said one task.

5. The method of claim 4,including providing for the performance of a second task, and transmitting and storing a message from said one task to said second task.

6. The method of claim 5, including transmitting to said backup task the stored message transmitted from said one task to said second task prior to performance of said persistent operation if it has been processed by said second task.

7. The method of claim 6, including said backup task providing a count of the number of messages transmitted by said one task.

8. The method of claim 7, wherein during processing by said backup task said count is used to determine when a message is to be transmitted by said backup task.

9. The method of claim 2, including performing a checkpointing operation between said one task and said backup task after at least a plurality of messages have been transmitted to said one task, said checkpointing operation including storing checkpointing data corresponding to the state of said one task, and transmitting said checkpointing data to said backup task at a time which is no later than required by the transmitting of processed messages to said backup task, said checkpointing data causing said backup task to be brought to the same state as said one task at the time of checkpointing.

10. The method of claim 9, wherein said checkpointing data includes a count of the number of messages processed by said one task since its start or last checkpointing, and wherein said backup task uses said count for discarding a corresponding number of said messages sent thereto.

11. In a fault tolerant computer system including a data processor providing a plurality of primary tasks, at least one of said primary tasks performing a persistent action, and wherein each primary task is subject to receiving one or more external signals whose time of occurrence may affect the performance of its respective task;

providing a corresponding plurality of interconnected backup tasks external to said data processor for backing up said plurality of primary tasks, the method comprising:

transmitting messages between said primary tasks for processing thereby;

storing the transmitted messages;

also storing external event data indicative of each external event type and its occurrence relationship to a predetermined primary task event;

each primary task processing the messages transmitted thereto;

transmitting at least particular ones of the stored messages to said backup tasks subsequently to said storing in a manner such that there is transmitted to said backup tasks, prior to the performance of a persistent action, at least those particular messages required for said backup tasks to recover from a failure of said data processor;

also transmitting external event data to respective backup tasks prior to performance of said persistent action;

said primary tasks continuing to process messages transmitted thereto so long as the aforementioned transmitting of messages and external event data to said backup tasks is met; and

recovering from said failure by causing said backup tasks to process the messages and external event data transmitted thereto.

12. The method of claim 11, wherein said particular messages are chosen such that said backup tasks and other parts of said system will agree on a particular state of each primary task prior to said failure which need not be the actual state thereof at the time of failure.

13. The method of claim 12, wherein said primary tasks sequentially process messages transmitted thereto, and wherein said particular messages comprise at least the most recently processed message and all earlier messages transmitted by said primary tasks.

14. The method of claim 13, wherein said primary tasks sequentially process messages transmitted thereto, wherein said storing of transmitted messages is in a queue which stores transmitted messages in the order of their transmission, wherein external event data is stored in said queue in occurrence order, wherein the performance of a persistent action is indicated in said queue by storing a persistent action indication at a position based no earlier than the position of the most recently processed message, and wherein said particular messages and external event data are transmitted to said backup tasks prior to the performance of said persistent action based on the position of said persistent action indication in said queue.

15. The method of claim 12, wherein said recovering includes causing said backup tasks to process the messages and external event data transmitted thereto in a manner which will result in each backup task arriving at its respective particular state.

16. The method of claim 11, wherein said particular messages and said external event data are transmitted to said backup tasks such that each backup task receives each message of said particular messages which was transmitted to its respective primary task and also receives external event data corresponding to each external event occurring for its respective primary task.

17. The method of claim 16, wherein said particular messages are transmitted to said backup tasks such that each backup task also receives each message of said particular messages which was transmitted by its respective primary task.

18. The method of claim 17, including each backup task providing a count of the number of messages received which were transmitted by its respective primary task.

19. The method of claim 18, wherein each backup processor uses said count during said recovering to determine when a message is to be transmitted.

20. The method of claim 12, including performing a checkpointing operation between a primary task and its respective backup task after at least a plurality of messages have been transmitted to said primary task, said checkpointing operation including storing checkpointing data corresponding to the state of the primary task, and transmitting said checkpointing data to the respective backup task subsequently to said storing and at a time no later than required for said backup tasks to recover from failure of said data processor using the messages transmitted thereto from their respective primary tasks, said checkpointing data causing the backup task to which it is transmitted to be brought to the same state as its respective primary task at the time of checkpointing.

21. The method of claim 20, wherein said checkpointing data includes a count of the number of messages processed by the primary task since its start or last checkpointing, and wherein the respective backup task uses said count for discarding a corresponding number of messages.

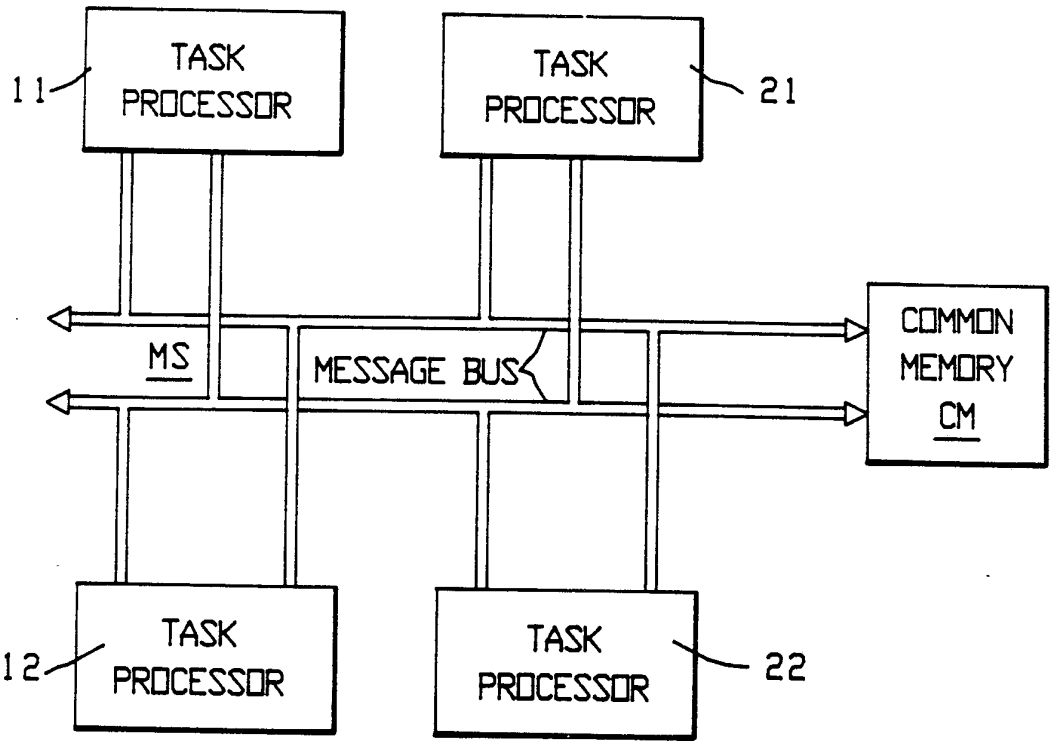
22. The method of claim 20, wherein said primary tasks sequentially process messages transmitted thereto, wherein said storing of transmitted messages is in a queue which stores transmitted messages in the order of their transmission, wherein external event data is stored in said queue in occurrence order, wherein said queue also stores a checkpointing indication at a position indicative of the time of performance of said checkpointing operation, wherein the performance of a persistent action is indicated in said queue by storing a persistent action indication positioned in said queue based on the position of the most recently processed message, and wherein said particular messages and external event data are transmitted to said backup tasks prior to the performance of said persistent action if positioned earlier in said queue than said persistent action indication, and wherein said checkpointing data is also transmitted to the respective backup if it is positioned earlier than the position of said persistent action indication.

23. The method of claim 2, 2, 3, 11 or 12, wherein said predetermined primary task event is a memory access operation.

24. The method of claim 1, 2, 3, 11 or 12, wherein said predetermined primary task event occurs a plurality of times during performance of said primary task, and wherein said external event data includes for each external event a count related to the number of times said predetermined primary event occurred from a reference point prior to occurrence of the external event.

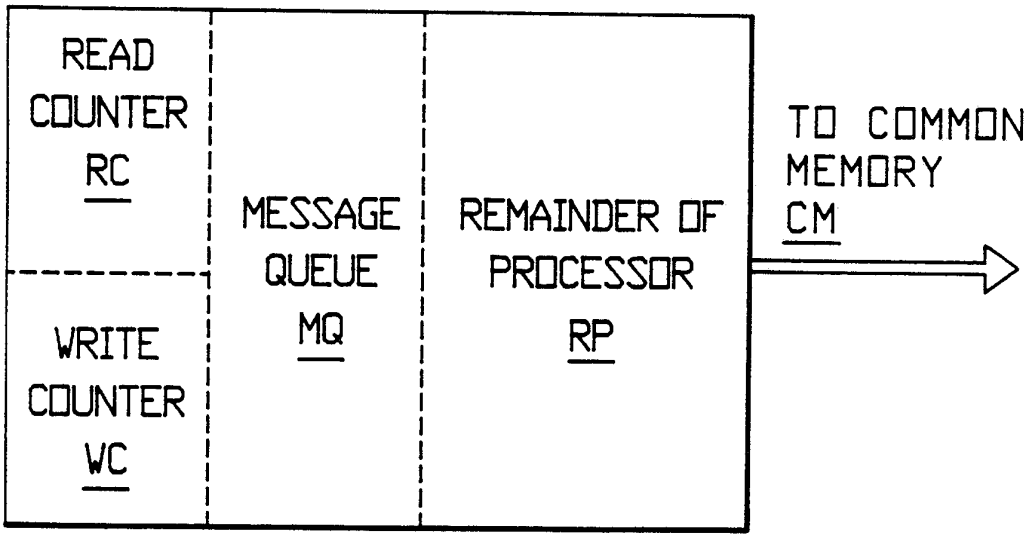
25. The method of claim 6, wherein said count is used by said backup task during replay to determine when the corresponding external event is to be delivered.

FIG. 1 (PRIOR ART)



PARRALLEL COMPUTER SYSTEM PCS

FIG. 2 (PRIOR ART)



TASK PROCESSOR XY

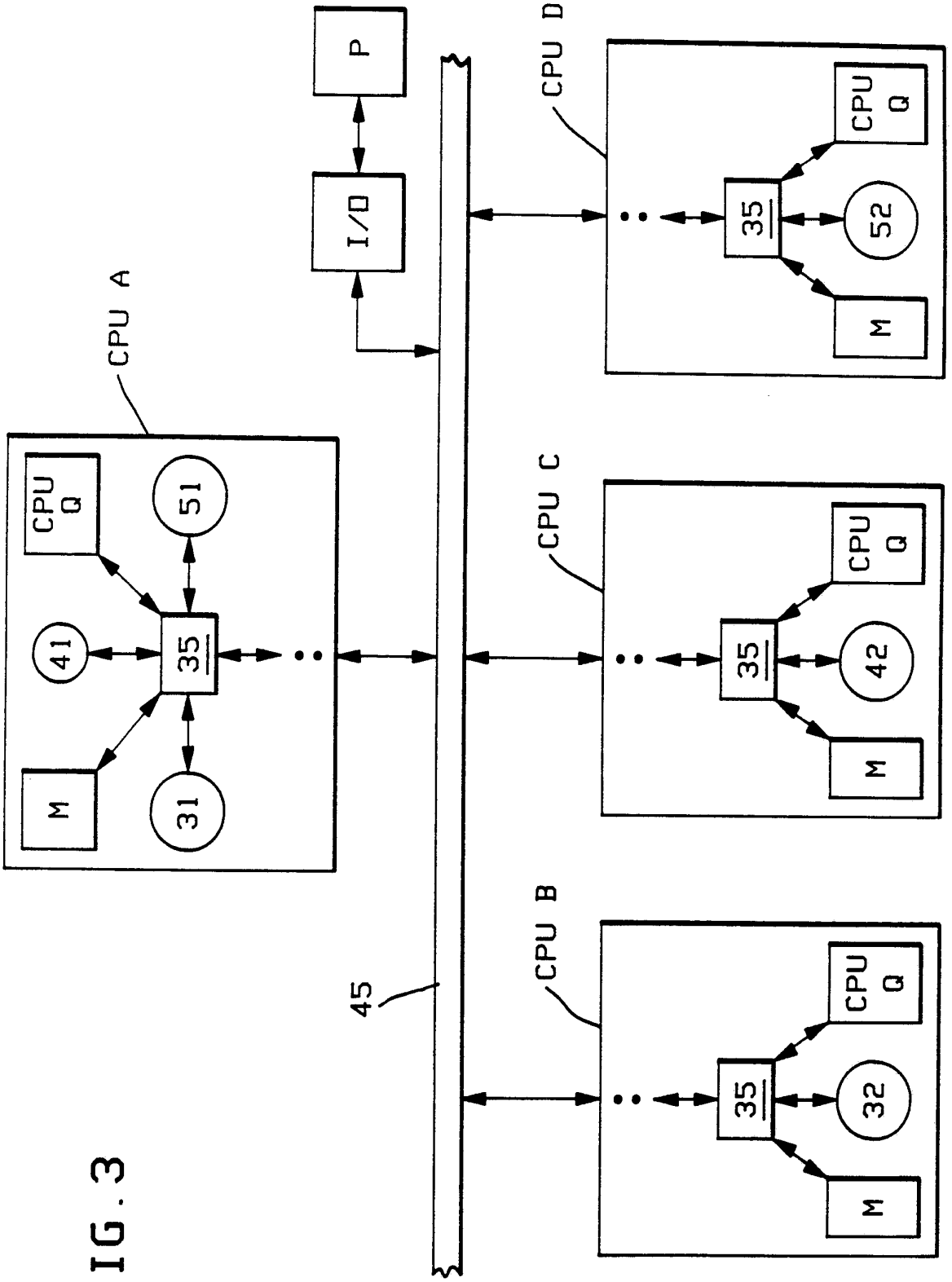

CPU A
CPU Q
51
41
35
31
M
I/O
P
45
CPU D
CPU Q
35
52
M
CPU C
CPU Q
35
42
M
CPU B
CPU Q
35
32
M

FIG. 3

FIG. 3A

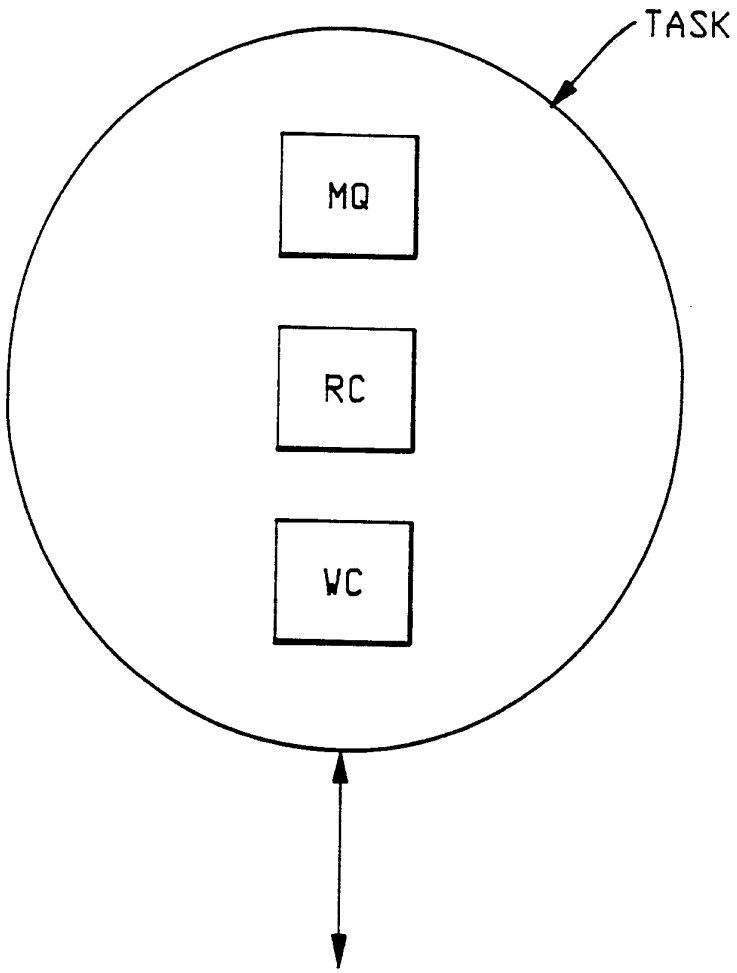


FIG. 4

EXAMPLE 1

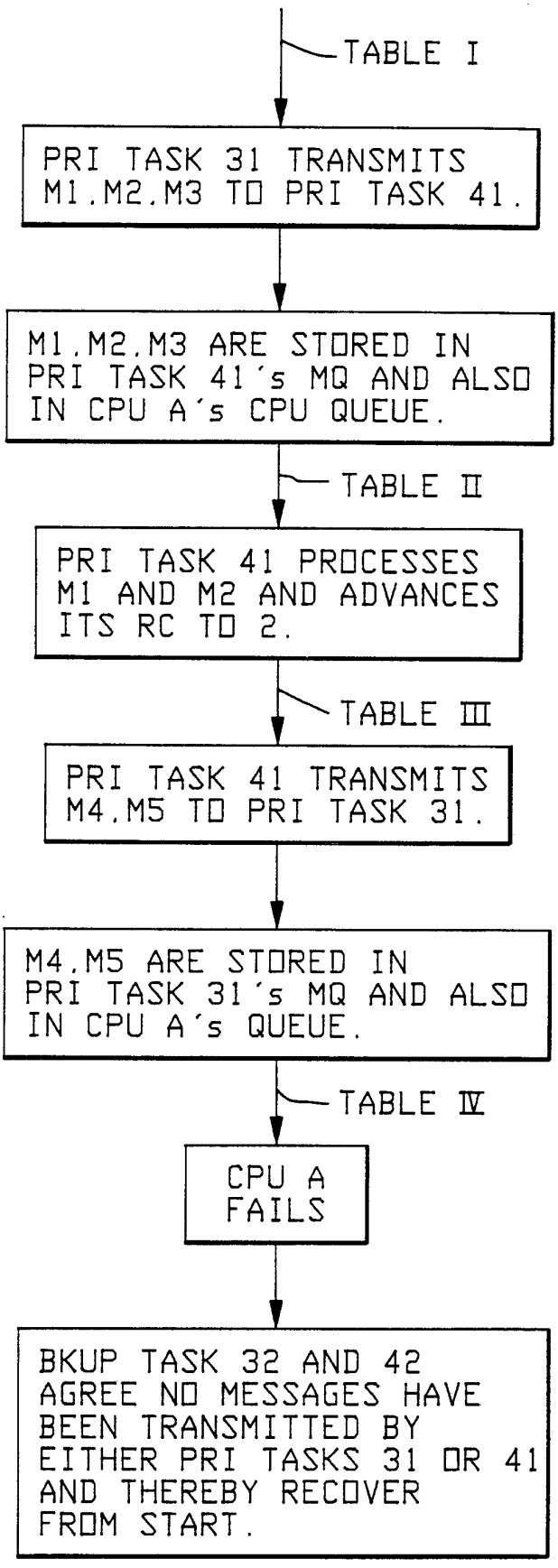


FIG. 5

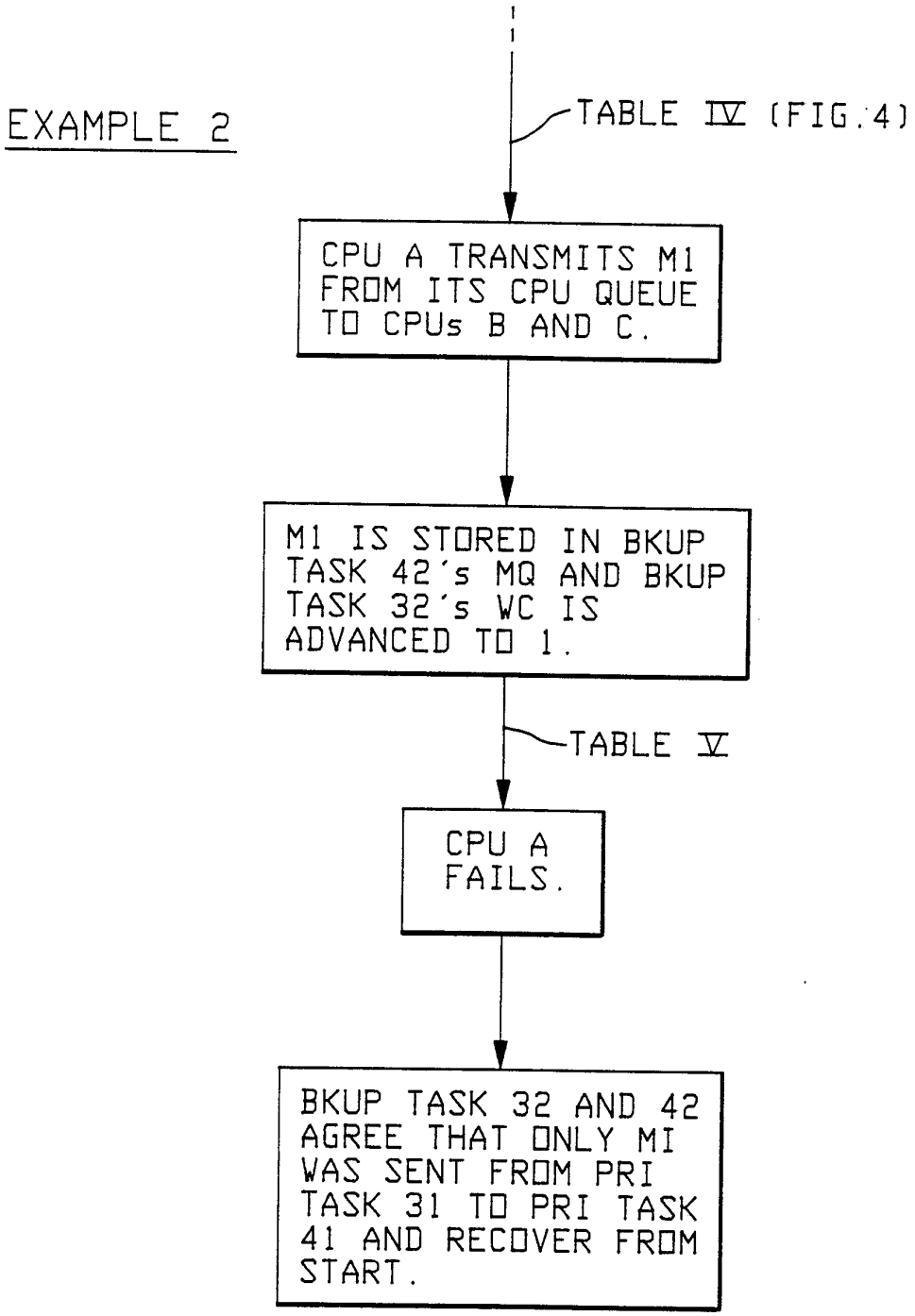

EXAMPLE 2
TABLE IV (FIG. 4)
CPU A TRANSMITS M1 FROM ITS CPU QUEUE TO CPUs B AND C.
M1 IS STORED IN BKUP TASK 42's MQ AND BKUP TASK 32's WC IS ADVANCED TO 1.
TABLE V
CPU A FAILS.
BKUP TASK 32 AND 42 AGREE THAT ONLY MI WAS SENT FROM PRI TASK 31 TO PRI TASK 41 AND RECOVER FROM START.

FIG. 6

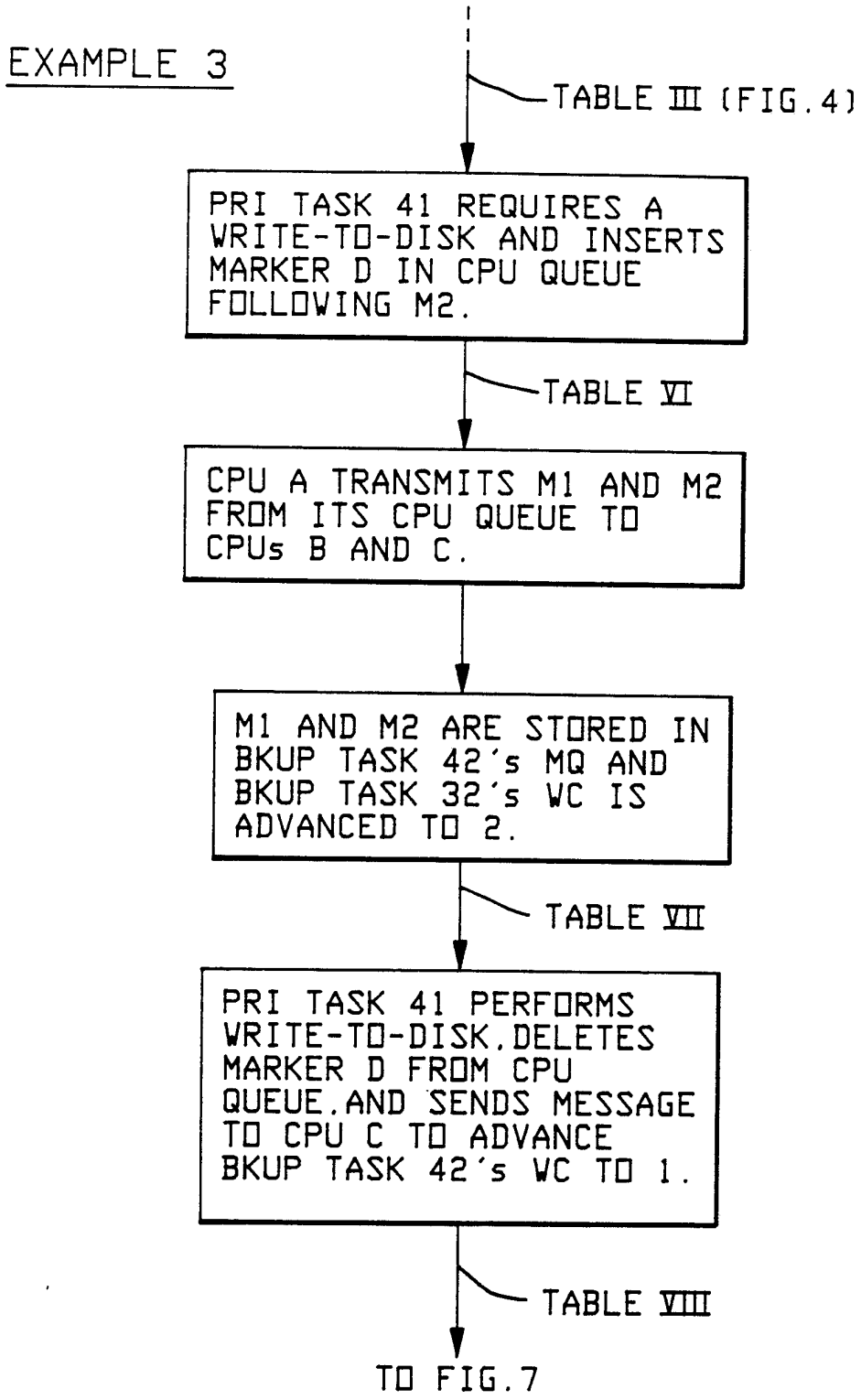

EXAMPLE 3
TABLE III (FIG. 4)
PRI TASK 41 REQUIRES A WRITE-TO-DISK AND INSERTS MARKER D IN CPU QUEUE FOLLOWING M2.
TABLE VI
CPU A TRANSMITS M1 AND M2 FROM ITS CPU QUEUE TO CPUs B AND C.
M1 AND M2 ARE STORED IN BKUP TASK 42's MQ AND BKUP TASK 32's WC IS ADVANCED TO 2.
TABLE VII
PRI TASK 41 PERFORMS WRITE-TO-DISK. DELETES MARKER D FROM CPU QUEUE. AND SENDS MESSAGE TO CPU C TO ADVANCE BKUP TASK 42's WC TO 1.
TABLE VIII
TO FIG. 7

FIG. 7

EXAMPLE 3
(CONTINUED FROM FIG.6)

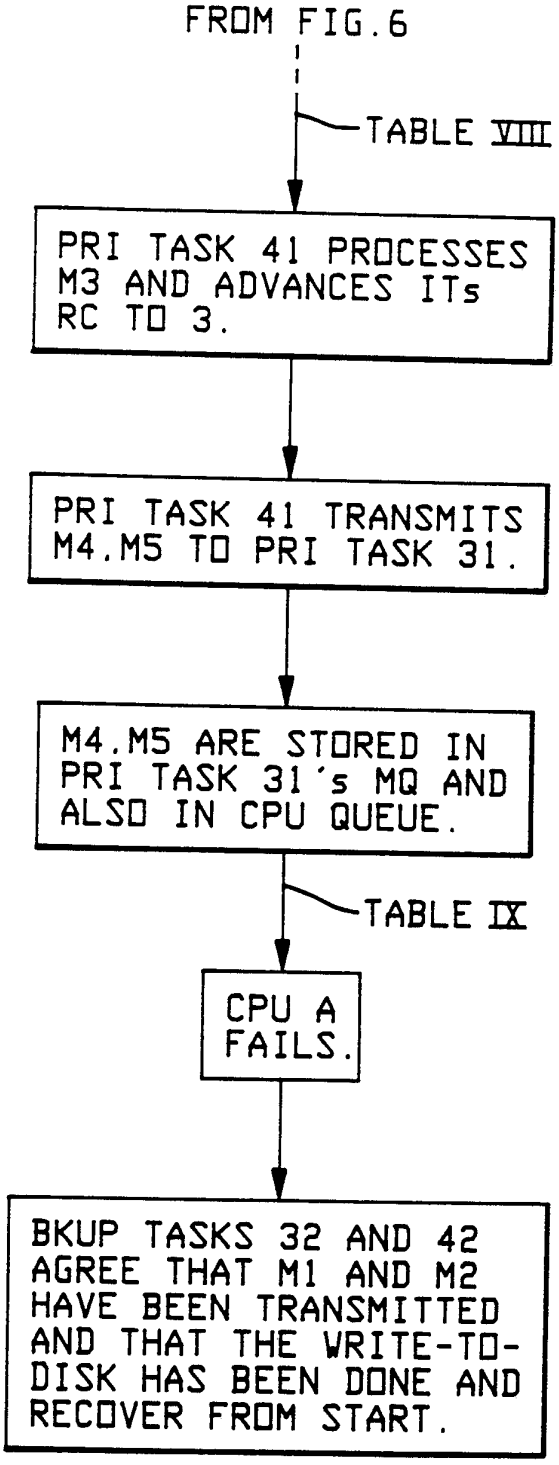
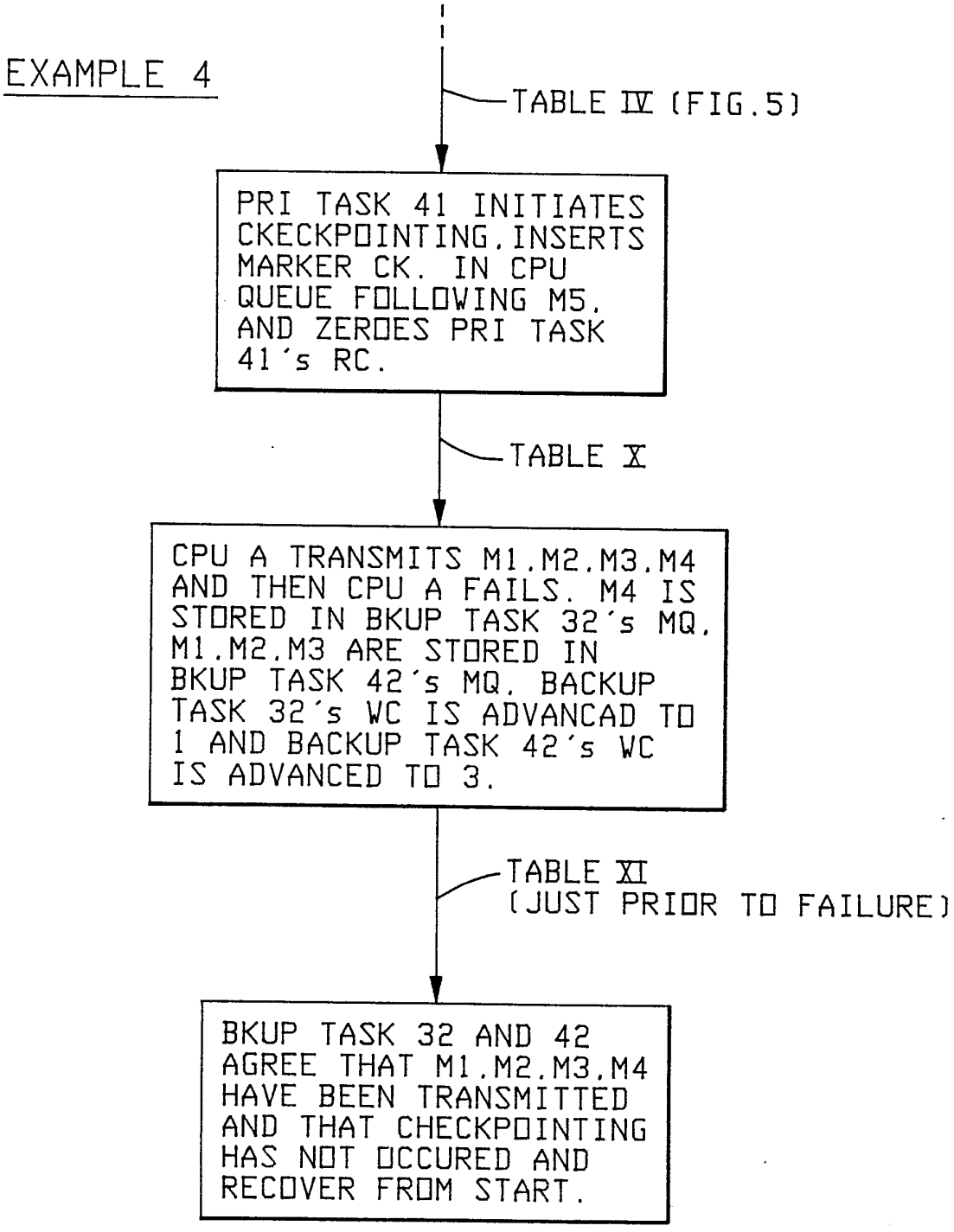


FIG.8


EXAMPLE 4
TABLE IV (FIG.5)
PRI TASK 41 INITIATES CKECKPOINTING, INSERTS MARKER CK. IN CPU QUEUE FOLLOWING M5, AND ZEROES PRI TASK 41's RC.
TABLE X
CPU A TRANSMITS M1,M2,M3,M4 AND THEN CPU A FAILS. M4 IS STORED IN BKUP TASK 32's MQ, M1,M2,M3 ARE STORED IN BKUP TASK 42's MQ. BACKUP TASK 32's WC IS ADVANCAD TO 1 AND BACKUP TASK 42's WC IS ADVANCED TO 3.
TABLE XI
(JUST PRIOR TO FAILURE)
BKUP TASK 32 AND 42 AGREE THAT M1,M2,M3,M4 HAVE BEEN TRANSMITTED AND THAT CHECKPOINTING HAS NOT OCCURED AND RECOVER FROM START.

EXAMPLE 5

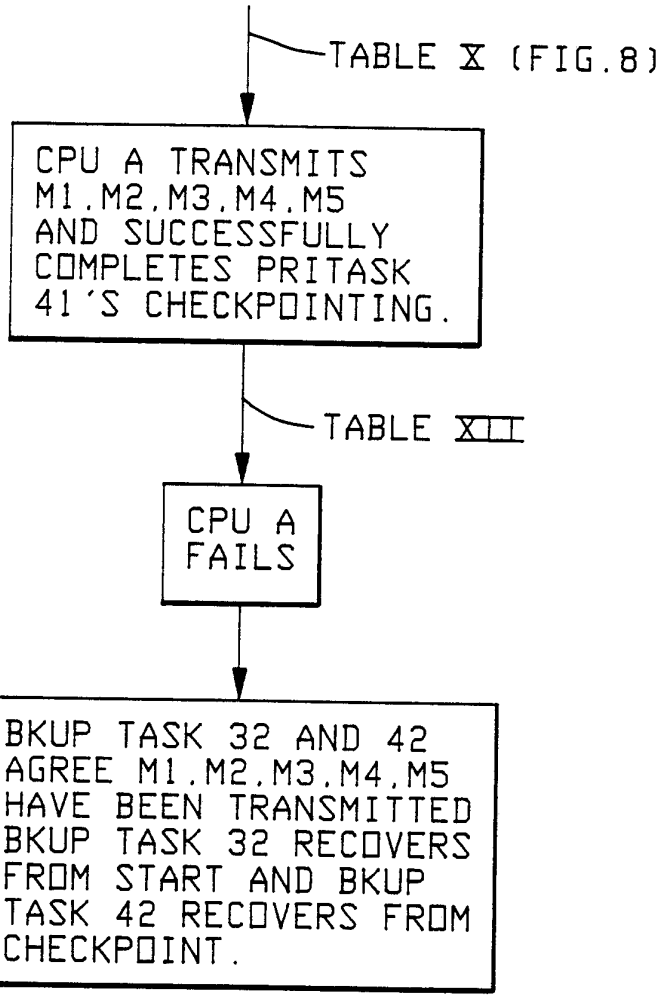


FIG.9

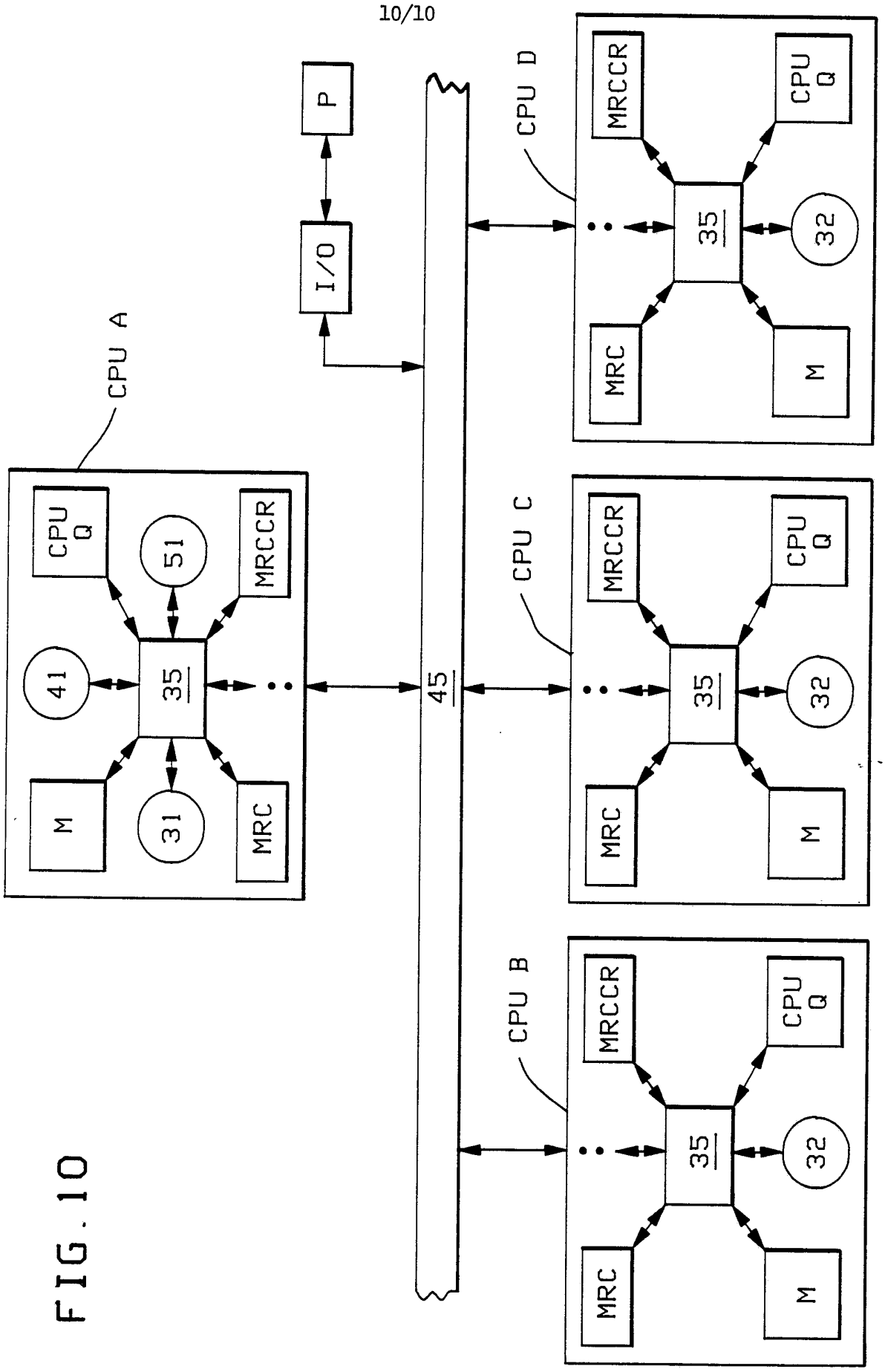


FIG. 10

INTERNATIONAL SEARCH REPORT

International Application No PCT/US 93/00618

I. CLASSIFICATION OF SUBJECT MATTER (if several classification symbols apply, indicate all)[6]

According to International Patent Classification (IPC) or to both National Classification and IPC

Int.Cl. 5 G06F11/20; G06F11/14

II. FIELDS SEARCHED

Minimum Documentation Searched[7]

Classification System	Classification Symbols
Int.Cl. 5	G06F

Documentation Searched other than Minimum Documentation to the Extent that such Documents are Included in the Fields Searched[8]

III. DOCUMENTS CONSIDERED TO BE RELEVANT[9]

Category °	Citation of Document, [11] with indication, where appropriate, of the relevant passages [12]	Relevant to Claim No.[13]
A	ACM TRANSACTIONS ON COMPUTER SYSTEMS vol. 7, no. 1, February 1989, NEW YORK, NY, USA pages 1 - 24 , XP000037157 A. BORG ET AL 'Fault Tolerance Under UNIX' cited in the application see page 11, line 12 - page 12, line 3 ---	1
A	EP,A,0 441 087 (INTERNATIONAL BUSINESS MACHINES CORPORATION) 14 August 1991 see abstract see page 10, line 41 - page 11, line 26 ---	1

-/--

° Special categories of cited documents : [10]

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

IV. CERTIFICATION

Date of the Actual Completion of the International Search	Date of Mailing of this International Search Report
07 MAY 1993	[illegible] 93
International Searching Authority	**Signature of Authorized Officer**
EUROPEAN PATENT OFFICE	ABSALOM R.

III. DOCUMENTS CONSIDERED TO BE RELEVANT (CONTINUED FROM THE SECOND SHEET)

Category °	Citation of Document, with indication, where appropriate, of the relevant passages	Relevant to Claim No.
A	THE 14TH INTERNATIONAL CONFERENCE ON FAULT-TOLERANT COMPUTING 20 June 1984, KISSIMMEE, FLORIDA, USA pages 374 - 379 R.E. STROM ET AL. 'OPTIMISTIC RECOVERY: AN ASYNCHRONOUS APPROACH TO FAULT-TOLERANCE IN DISTRIBUTED SYSTEMS' see the whole document ---	1
A	EP,A,0 260 625 (ASEA AB) 23 March 1988 see the whole document ---	1
A	WO,A,9 117 504 (UNISYS CORPORATION) 14 November 1991 cited in the application -----	

ANNEX TO THE INTERNATIONAL SEARCH REPORT ON INTERNATIONAL PATENT APPLICATION NO.

US 9300618
SA 69486

This annex lists the patent family members relating to the patent documents cited in the above-mentioned international search report.
The members are as contained in the European Patent Office EDP file on
The European Patent Office is in no way liable for these particulars which are merely given for the purpose of information. 07/05/93

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
EP-A-0441087	14-08-91	JP-A- 4213736	04-08-92
EP-A-0260625	23-03-88	SE-B- 454730	24-05-88
		DE-A- 3781486	08-10-92
		SE-A- 8603945	20-03-88
		US-A- 4941087	10-07-90
WO-A-9117504	14-11-91	EP-A- 0482175	29-04-92
		JP-T- 5500430	28-01-93

EPO FORM P0479