Informatik-Fachberichte

Herausgegeben von W. Brauer
im Auftrag der Gesellschaft für Informatik (GI)

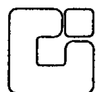
84

Fehlertolerierende Rechensysteme

2. GI/NTG/GMR-Fachtagung

Fault-Tolerant Computing Systems
2nd GI/NTG/GMR Conference

Bonn, 19.-21. September 1984



Herausgegeben von
K.-E. Großpietsch und M. Dal Cin

Springer-Verlag
Berlin Heidelberg New York Tokyo 1984

Entwurf und Struktur einer Betriebssystemschicht zur Implementierung von Fehlertoleranz

Design and Structure of an Operating System Layer Implementing Fault Tolerance

Th. Risse, R. Brause, M. Dal Cin, E. Dilger, J. Lutz
Institut für Informationsverarbeitung
Universität Tübingen
Köstlinstr. 6, 74 Tübingen

Zusammenfassung
In dieser Arbeit wird der Entwurf und im weiteren die Struktur einer Fehlertoleranz implementierenden Betriebssystemschicht für ein Mehrmikroprozessorsystem dargestellt. Diese setzt auf einem üblichen (kommerziellen) Betriebssystem auf. Sie ist für den Benutzer transparent und realisiert "softwareimplementierte Fehlertoleranz auf Betriebssystemebene" im Arbeitsplatzrechner ATTEMPTO.

Abstract
In this paper, design and structure of an operating system layer which is to implement fault tolerance in a multi-microprocessor system is described. This layer is installed on top of an off-the-shelf operating system. It is transparent to the user and realizes the "software implemented fault tolerance (on operating system level)" of the work station ATTEMPTO.

1. Einleitung

ATTEMPTO (A TesTable Experimental MultiProcessor system with fault-TOlerance) ist ein experimenteller Arbeitsplatzrechner, der in Tübingen entwickelt wird. Realisiert wird er mit kommerziell erhältlichen Hard- und Software-Komponenten (single board computer SBC, UNIX). Das System wird vorwiegend in Modula-2 /Wirth/ implementiert.

Mit ATTEMPTO soll gezeigt werden, daß es möglich ist, die Vorteile eines Rechners am persönlichen Arbeitsplatz mit denen eines fehlertoleranten Rechners zu verbinden, und daß es sinnvoll ist, den Benutzer das für jeden Anwendungsjob benötigte Maß an Fehlertoleranz selbst bestimmen zu lassen. Denn dieses Maß ist durch die jeweilige Anwendung bestimmt und daher nur dem Benutzer bekannt. Um dieses Ziel zu erreichen, müssen auf Betriebssystemebene effiziente Fehlertoleranzmechanismen implementiert werden, an die wir als wichtigste Forderungen stellen:

a) Transparenz für den Benutzer,
b) Rekonfigurierbarkeit der die Fehlertoleranz implementierenden Programm-Moduln
c) Dezentralisierung aller Mechanismen für Fehlertoleranz

Die Dezentralität dieser Mechanismen ist notwendig, da es aus Gründen der Zuverlässigkeit unangebracht ist, solche Aufgaben von einer zentralen Instanz wahrnehmen zu lassen.
Die Rekonfigurierbarkeit soll es ermöglichen, das Zuverlässigkeitsverhalten des Rechners durch Hinzunahme weiterer Software-Bausteine oder durch deren Austausch zu optimieren. ATTEMPTO dient somit auch als Testbett für die Erprobung von Verfahren der Fehlerdiagnose und Wiederherstellung.
Diese Forderungen führten zu einer in mehreren Stufen hierarchisch gegliederten, modularen Betriebssystem-Erweiterung, die auf einem konventionellen Betriebssystem-Kern aufsetzt. In dieser Arbeit wird die Struktur der Erweiterung und ihre Schnittstelle zum Betriebssystem-Kern beschrieben. Zuvor soll jedoch in Abschnitt 2 das Fehlertoleranzkonzept von ATTEMPTO kurz dargelegt werden. Eine ausführliche Darstellung dieses Konzepts findet man in /Ammann et al./. In Abschnitt 3 wird sodann die Grobstruktur der Betriebssystem-Erweiterung dargestellt. In Abschnitt 4 werden die Moduln, die für die Fehlertoleranz zuständig sind, näher charakterisiert. In Abschnitt 5 wird schließlich die Schnittstelle zum Betriebssystem-Kern beschrieben.

2. Das Fehlertoleranzkonzept von ATTEMPTO

Um Fehlertoleranz zu erreichen, ist sicher Redundanz nötig und Redundanz kann bei Multiprozessorsystemen auf verschiedene Art und Weise genützt werden, 'tightly synchron' z.B. im FTMP-System /Parker/, taktsynchron in iAPX 432-Systemen /Geyer/, 'frame synchron' im SIFT-System /Wensley/ oder asynchron. Für ATTEMPTO wurde der Weg der asynchronen Redundanz gewählt, das heißt, daß Kopien eines Benutzerjobs asynchron auf mehreren Prozessoren (Kollegen) abgearbeitet werden und, daß der Nachrichtenaustausch lose gekoppelt über Botschaften erfolgt. Diese Asynchronität bietet den Vorteil, daß sich gewisse kontextabhängige transiente Fehler nicht auf allen Prozessoren in derselben Weise auswirken.
Ein wichtiges Entwurfsziel ist, wie erwähnt, die Dezentralität. Jeder Prozessor bearbeitet seine Dispatching- und Schedulingaufgaben selbständig. Er kann auch keine Aufgaben an andere Prozessoren delegieren. Insbesondere übernimmt auch in Ausnahmesituationen kein Prozessor irgendwelche Überwachungsaufgaben bezüglich des Gesamtsystems. Weiterhin werden keine globalen Systemtafeln verwendet und es existiert kein gemeinsamer Speicher. Jeder Prozessor führt seine eigene Systemtafel und hält sie mit Hilfe von Botschaften anderer Prozessoren auf dem neuesten Stand. Dieses Konzept verhindert Systemzusammenbrüche auf-

grund von Fehlern in einem gemeinsamen Speicher. Über das damit verbundene Problem, die einzelnen Systemtafeln konsistent zu halten, und seine Lösung in ATTEMPTO wurde in /Brause et al./ berichtet.
Für den Benutzer stellt sich ATTEMPTO als Single-User-Multi-Tasking-System dar. Wie bereits erwähnt, soll der Benutzer selbst über das angemessene Verhältnis zwischen Fehlertoleranz (Redundanz) und Effizienz entscheiden. Der Benutzer teilt dem System zusätzlich zum Jobnamen einen sogenannten Fehlertoleranzindex t mit und legt damit fest, daß während der Durchführung des Programmes mindestens t Fehler toleriert werden sollen, genauer, daß während der Durchführung des Programms t der beteiligten SBCs (Kollegen) ausfallen dürfen und dennoch eine korrekte Ausgabe erzeugt wird. Solange also Fehlverhalten einzelner Prozessoren für den Benutzer nicht in Erscheinung tritt, wird dieses Fehlverhalten ignoriert. Erst wenn es sich nach außen auswirken könnte (z.B. falsche Ausgabe, gar keine Ausgabe), werden Fehler behandelt und zwar, für den Benutzer verborgen, mittels Maskierung und Diagnose basierend auf Vergleichstests /Ammann et al./. Dieser Ansatz ist weitgehend unabhängig von der Art der auftretenden Fehler. Eine Fehlerlokalisierung auf Bauteilebene wird dadurch nicht erreicht, doch ist diese auch nicht beabsichtigt (end-to-end strategy; vgl. /Saltzer/).

3. Die Fehlertoleranz-Schicht von ATOS

Auf jedem SBC gibt es ein eigenständiges Betriebssystem - wir nennen es ATOS (ATTEMPTO's local Operating System). ATOS gliedert sich in zwei funktionale Ebenen. Die untere Ebene wird durch einen üblichen Betriebssystem-Kern gebildet. Die obere Ebene ist für die Fehlertoleranz des Systems zuständig. Sie ist für den Benutzer transparent und hat aus der Sicht des Betriebssystem-Kerns den Status eines Anwenderprozesses. Ihr Entwurf entspricht dem Konzept virtueller Maschinen.

Die Software-Architektur der oberen ATOS-Ebene, der sogenannten Fehlertoleranz-Schicht FTL (fault-tolerance layer), basiert auf einer vierstufigen Hierarchie (s. Fig. 1), deren oberste Schicht 4 (FTI fault-tolerance instance) die eigentlichen Fehlertoleranz-Mechanismen enthält. Schicht 3 (communication support layer) ist im wesentlichen für die korrekte Kommunikation zwischen Benutzerjobs und FTI sowie - auf logischer Ebene - zwischen dem Host-SBC und den restlichen SBCs des Systems zuständig. Die beiden unteren Schichten stellen Dienstleistungen zur Verfügung, die für die Verwaltung der Modula-2 Prozesse (Coroutinen), der Datenstrukturen und des Botschaftenaustausches von den darüberliegenden Schichten benötigt werden.

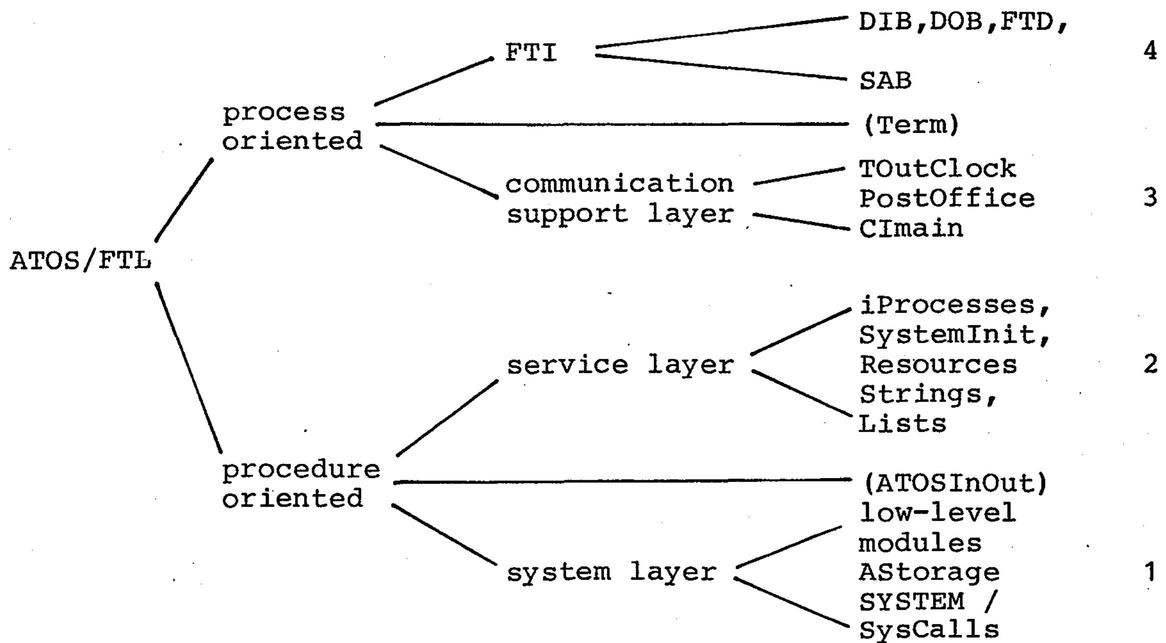


Fig. 1 Hierarchie der Fehlertoleranz-Schicht in ATOS

Dementsprechend sind die grundlegenden Software-Bausteine der FTL informationsverbergende Moduln von zweierlei Art:

1) Moduln mit einer strikt prozeduralen Schnittstelle, die meist abstrakte Datentypen implementieren (Schicht 2) und
2) Moduln, die Datenstrukturen zusammen mit einer aktiven Einheit, dem sogenannten (Modul-) Clerk (einem Prozeß im Sinne von Modula-2), enthalten (Schicht 4).

Die Datenstrukturen der letztgenannten Moduln sind von außerhalb der Moduln nicht zugänglich. Sie werden vielmehr allein von den entsprechenden Clerks verwaltet. Die Kommunikation zwischen den Clerks verschiedener Moduln geschieht durch Botschaftenaustausch. Dieses Prinzip erleichtert vor allem die Rekonfigurierbarkeit der für die Fehlertoleranz des Systems zuständigen obersten Schicht (vgl. dazu auch /Liskov/).

Eine Zwischenstellung nehmen die Moduln der Schicht 3 ein. Auch hier wurde nach Möglichkeit das Prinzip der Datenkapselung verwirklicht.

Das Modul iProcesses der Schicht 2 stellt den Clerks Prozeduren für den Botschaftenaustausch bereit. Daneben werden Dienstleistungen, z.B. die Erzeugung von Briefkästen erbracht.

In ATOS sind drei Ebenen der Kommunikation zu unterscheiden:

1. Kommunikation zwischen Clerks (Modula-2 Prozesse)
2. Kommunikation zwischen UNIX-Prozessen und
3. Inter-Prozessor-Kommunikation

Für den Botschaftenaustausch ist - wie erwähnt - jedem Clerk ein eigener Briefkasten zugeordnet. Die SBC-lokale Übergabe einer Botschaft erfolgt per Referenz, um unnötiges Kopieren zu ersparen.
Die Inter-Prozeß-Kommunikation folgt den UNIX-Konventionen. (Sie kann über pipes vermittels eines memory device als pipe device abgewickelt werden.)
Dagegen erfolgt der Nachrichtenaustausch zwischen SBC's über sogenannte dedizierte Ports auf der Grundlage eines speziellen Protokolls, das in /Brause et al./ näher beschrieben wurde. Zur Verwaltung dieser Ports wurde auf jedem SBC ein sogenannter Porthandler eingerichtet (s. Abschnitt 5).

Alle Briefkästen, Daten-Puffer, Warteschlangen usw. sind als Ausprägungen eines einzigen abstrakten Datentyps List implementiert. Um diese zentralen Datenstrukturen unempfindlich gegen Speicherfehler zu machen, wurde das Modul Lists konzipiert, das den abstrakten Datentyp List zur Verfügung stellt. Inhärenter Bestandteil der Listenstruktur ist die Möglichkeit, eine beschränkte Anzahl von Fehlern in der Verzeigerung zu erkennen und zu korrigieren /Risse et al./.

Die (geklammerten) Module Term und ATOSInOut in Fig. 1 haben nur für Simulation und Entwicklung Bedeutung. Für einen SBC simuliert das Modul Term die Umgebung jeweils gewünschter Moduln der FTI (Fig. 2).

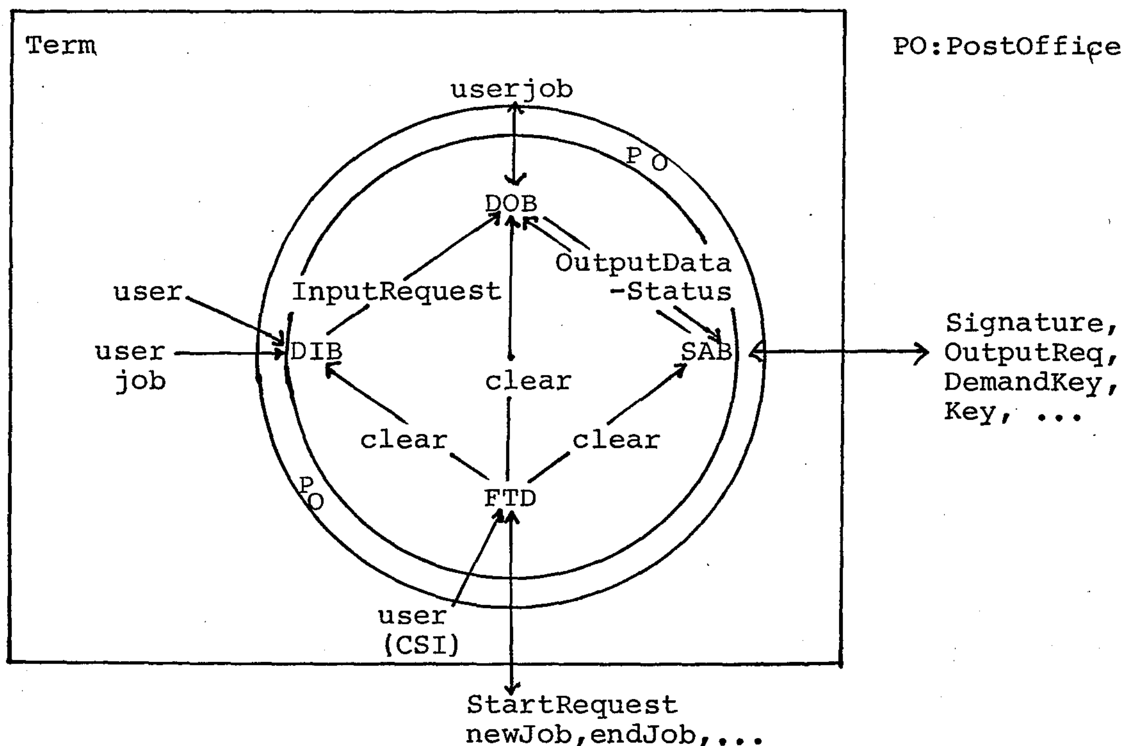


Fig. 2 Simulation des Austausches von Botschaften in ATTEMPTO

Zudem ermöglicht Term den Test der Fehlertoleranz-Eigenschaften von Lists durch gezielte Veränderung von Speicherinhalten per "exclusivem" Zugriff auf das ATOS-interne Modul AStorage zur Speicherverwaltung. In der späteren Realisierung soll das Modul Term in abgewandelter Form als Error Logger Verwendung finden.
Ebenso dient das Modul ATOSInout Simulationszwecken, indem es derzeit die Verbindung von in Modula-2 erstellten Benutzerprogrammen zu ATOS herzustellen gestattet. Deshalb stimmt seine Schnittstelle (DEFINITION MODULE) mit der des (Standard-Bibliotheks-) Moduls InOut überein.

4. Charaktisierung einiger Moduln der Fehlertoleranz-Schicht

Im folgenden sollen einige der Moduln, die für die Fehlertoleranz von ATTEMPTO zuständig sind, näher charakterisiert werden.

MODULE DIB
Datenstruktur: Data Input Buffer, Puffer für vom Benutzer eingegebene Daten (für READ des Benutzerjobs)
aktive Einheit: DIBclerk, verwaltet den Eingabe-Puffer
Botschaften:
1) UserInput: Eintragen der Input-Daten in den Puffer.
2) UserJobRead: Untersuchung, ob angeforderte Daten im Puffer vorhanden sind. Falls vorhanden, Versenden einer Botschaft mit Kennung 'InputData' über PostOffice an Benutzerjob, sonst Versenden einer Botschaft mit Kennung 'InputRequest' an DOBclerk.
3) clearDIB: Aushängen aller zum angegebenen Job gehörenden Input-Daten.
4) initDIB: Initialisierung des DIBuffer

MODULE DOB
Datenstruktur: Data Output Buffer, Puffer für die auszugebenden Daten
aktive Einheit: DOBclerk, verwaltet den Ausgabe-Puffer
Botschaften:
1) UserJobWrite: Falls Anzahl der Kollegen = 1, Ausgabe des entsprechenden Pufferinhalts; andernfalls Übermittelung des entsprechenden Pufferinhaltes an SABclerk in Botschaft mit Kennung 'OutputData'
2) OutStatus: Falls IO-Master, dann Beauftragen des Resource-Managers mit Ausgabe des entsprechenden Puffer-Eintrags, sonst Überwachen der Ausgabe des jeweiligen IO-Masters
3) InputRequest: Ausgabe der Aufforderung 'Input für JobID' an den Benutzer wie in 1) und 2) beschrieben
4) clear DOB: Aushängen aller zum angegebenen Job gehörenden Output-Daten
5) init DOB: Initialisierung des DOBuffers

MODULE FTD
Datenstruktur: Job Control Buffer (JCB), Liste von Job-Kontroll-Blöcken mit JobNamen, Fehlertoleranzindex, Kollegen-Liste, Anfangszeiten (für Timeout)
aktive Einheit: Fault-Tolerant Dispatcher, verwaltet den JCB-Puffer
Botschaften:
1) newJob: Erzeugung eines Job-Kontroll-Blockes, falls idle, Initiierung der Job-Bearbeitung, Mitteilung dieser Intention allen SBCs (Botschaft mit Kennung 'StartRequest')
2) StartRequest: falls noch nicht alle Kollegen vorhanden, Eintragen der Start-Zeit in zugehörigen Job-Kontroll-Block, und falls zudem eigene StartRequest-Botschaft, Botschaft an PostOffice für JobStart versenden
3) endJob: Terminierung einer Job-Bearbeitung, Botschaften 'clearDIB' an DIBclerk, 'clearDOB' an DOBclerk, 'clearSAB' an SABclerk

MODULE SAB
Datenstruktur: Signature Array Buffer, Liste von Verbunden mit Job-Namen, Nummer des Outputs, Signatur der Output-Daten usw.
aktive Einheit: SABclerk, verwaltet SAB, veranlaßt Vergleich bzw. Diagnose
Botschaften:

1) OutputData: Bildung der Signatur der empfangenen Output-Daten, Versenden dieser Signatur an alle Kollegen-SABclerks (aus Gründen der Synchronisation auch an sich selbst) mit Kennung 'Signature', zugleich Aufsetzen eines Timeout
2) Signature: Eintragen der empfangenen Signatur in den SA-Puffer. Sobald alle Signaturen vorhanden, Vergleich und Diagnose. Ausbleibende Signaturen gelten als fehlerhaft. Danach Versenden einer Botschaft mit Kennung 'DemandKey' an bestimmten Kollegen-SABclerk (KeyPartner), zugleich Aufsetzen eines weiteren Timeout
3) DemandKey: Versenden des vom KeyPartner angeforderten Schlüssels für die Ausgabe-Bewilligung in Botschaft mit Kennung 'Key'
4) Key: Wenn Ausgabe mit erhaltenem Schlüssel möglich, Versenden einer Botschaft an alle Kollegen-SABclerks (incl. sich selbst) mit Kennung 'OutputRequest'
5) OutputRequest: falls eigene 'OutputRequest'-Botschaft als erste empfangen, IO-Master sonst Überwacher. Versenden der entsprechenden Botschaft mit Kennung 'OutStatus' an DOBclerk
6) ClearSAB: Aushängen aller zum angegebenen Job gehörenden SAB-Einträge
7) initSAB: Initialisierung des SABuffer

MODULE TimeOut
Datenstruktur: ClockList, Liste von Verbunden mit Timeout-Zeitpunkt und Zeiger auf solche Botschaften, die mit Zusatzinformation 'TimeOut' über Postoffice wieder an den Sender zurückgegeben werden, sobald das Timeout-Intervall abgelaufen ist (vermittels der Routinen setTimeOut und clearTimeOut)
aktive Einheit: Clockclerk, überprüft regelmäßig ClockList nach Botschaften, deren TimeOut-Intervall abgelaufen ist, und gibt diese zurück
Botschaften: keine

Aktive Einheiten wie zum Beispiel der SABclerk sollen nicht beliebig lange auf Antworten anderer, möglicherweise ausgefallener Einheiten warten müssen. In gewissen Fällen wird dann die entsprechende Einheit als defekt angesehen. Die Problematik von Termin-Überschreitungen im Zusammenhang mit Fehlertoleranz wird schon in /Wensley et al./ aufgezeigt (vgl. auch /Lamport/).

ATOS enthält außer den hier beschriebenen Moduln das Modul FTLinit für die Initialisierung der FTL, das Modul Resources zur Verwaltung der systemweiten Betriebsmittel und Datei-Moduln, in denen allgemein verwendete Konstanten und Typen definiert sind. Dadurch wird vermieden, daß diese Objekte aus Moduln der oberen Schichten importiert werden müssen. Dies soll die Portierung von ATOS erleichtern.

5. Schnittstelle zum Betriebssystem-Kern

In unserer Implementierung bilden userjobs und die FTL eigene UNIX-Prozesse. Der UNIX-Prozeß FTL seinerseits ist als eine Anzahl von kooperierenden Modula-2 Prozessen realisiert. Der FTL sind außerdem

weitere UNIX-Prozesse wie PortHandler (PH) und TerminalHandler (TH) beigeordnet, die die Daten von den Ports bzw. dem Terminal (-Bus) an die FTL übertragen.
Die Kommunikation zwischen userjob und FTL kann nicht über gemeinsame Speicherbereiche erfolgen, da UNIX-Prozesse in der Regel zwischenzeitlich auf Massenspeicher ausgelagert werden. Sie wird wird stattdessen über sogenannte pipes abgewickelt, d.h. über gemeinsame Prozeß-unabhängige Speicherbereiche, die dem direkten Zugriff entzogen sind und nur über spezielle System-Calls angesprochen werden können.

Es soll nun möglich sein, auf dem ATTEMPTO-System jeden ablauffähigen Programm-Code auch fehlertolerant abarbeiten zu lassen. Dazu müssen insbesondere die Aufrufe von Dienstleistungen des Betriebssystemes geeignet behandelt werden. Bei ATTEMPTO haben wir uns dazu entschieden, solche System-Calls daraufhin zu überprüfen, ob die FTL zur Ausführung der Dienstleistung (z.B. read oder write) nötig ist.

Umleitung der System-Calls

Bei einem System-Call (trap), angefordert von einem UNIX-Prozeß oder einem Benutzerjob, wird statt der vorgesehenen Behandlungsroutine im Betriebssystem-Kern die Routine CIkernel aufgerufen. Diese entscheidet, ob zur weiteren Behandlung des Aufrufes eine Instanz der FTL aktiv werden muß oder ob der System-Call direkt an das Betriebssystem übergeben werden kann. Falls Dienstleistungen der FTL benötigt werden, werden das PostOffice und entsprechende Teile der FTI aktiviert. Nach der Bearbeitung innerhalb der FTL geht dann die Kontrolle an die Routine CIkernel und damit an den aufrufenden UNIX-Prozeß zurück.
Dieses Umleitungsverfahren bietet verschiedenen Vorteile:

- jedes ablauffähige Programm kann nach Maßgabe des Benutzers auch fehlertolerant abgearbeitet werden.
- Betriebssystem-Kerne sind in der Regel vor unzulässigen (beabsichtigt oder nicht) Eingriffen der Benutzer geschützt. Dieser Schutz erstreckt sich damit auch auf den Zugang zur FTL.
- da nur an eine einzige Stelle im Kern eingesprungen wird, bleibt die Änderung lokal und kontrollierbar.
- auch ohne die Kenntnis der Quellen des Betriebssystem-Kernes ist es möglich, eine Zwischenschicht einzufügen. (So können z.B. die Parameter der System-Calls anhand des jeweils zugehörigen, rückassemblierten Codes verifiziert werden.)
- ATOS ist bis auf die Routine CIkernel maschinenunabhängig. Da jedoch trap-Mechanismen auf weitgehend allen Mikroprozessoren

vorhanden sind, läßt sich das Umleitungskonzept leicht auf andere Prozessor-Typen übertragen. Die entsprechende Routine CIkernel ist an jeden Betriebssystem-Kern z.B. als pseudo-device anzulagern.

Funktion des PostOffice

Um eine Botschaft an Kollegen zu versenden, muß ein Clerk diese an das (eigene) PostOffice senden. Der PostOffice-Clerk ergänzt die Botschaft um die für die Inter-Prozessor-Kommunikation nötige Information und reicht die Botschaft in Form eines direkten System-Aufrufes (CI-Call) über den Betriebssystem-Kern an den entsprechenden Porthandler weiter.

Das Modul PostOffice bildet also die Schnittstelle zwischen FTL und dem Betriebssystem-Kern und stellt die Weiterleitung von Information zwischen FTL und Benutzerjob, Port- bzw. Terminalhandler und dem File-System sicher (vgl. auch Fig. 2).
Der PostOffice-Clerk wird genau dann aktiv, wenn er von einem Clerk der FTI durch dessen Botschaft beauftragt wurde oder wenn er eine Nachricht vom Porthandler-Prozeß, vom Terminalhandler-Prozeß oder von CIkernel erhält. Das PostOffice verwendet dazu verschiedene Prozeduren, die in ihrer betriebssytem-spezifischen Implementierung im Modul CImain zusammengefaßt sind.

Die Umleitung von System-Calls kann folgendermaßen skizziert werden:

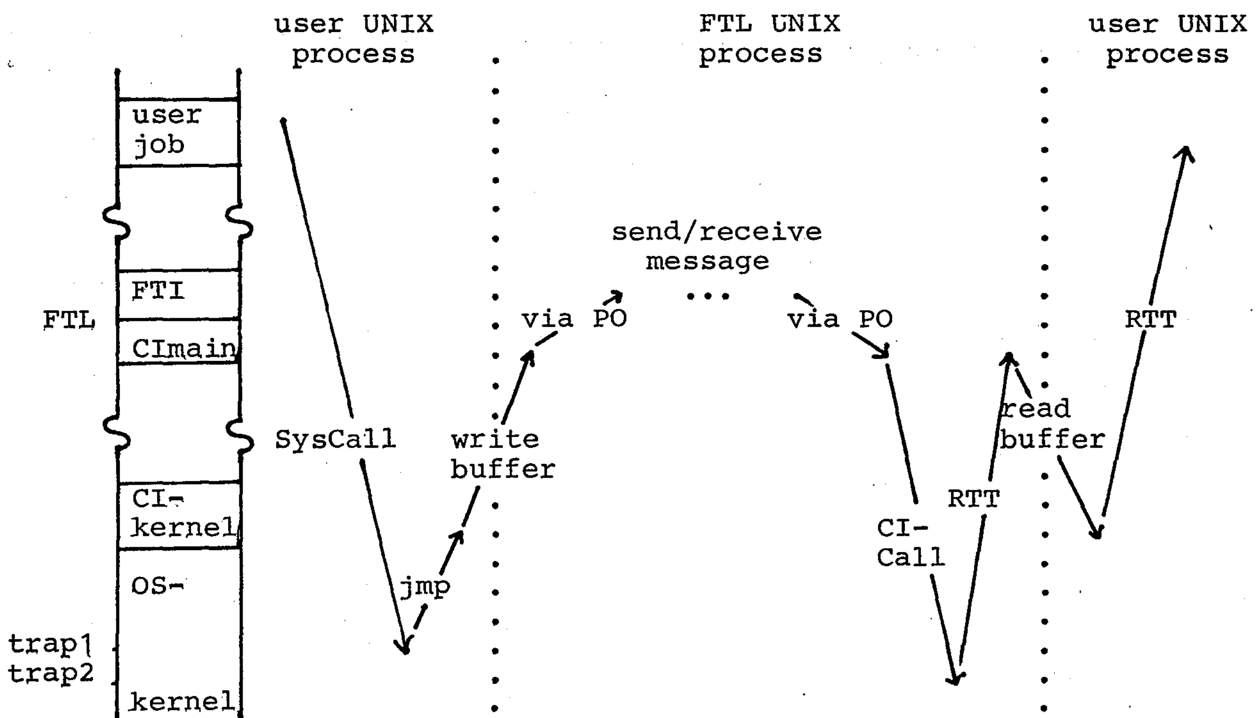


Fig. 3 Umleitung von SysCalls zur Bearbeitung in der FTL

Simulation der Interaktion in ATTEMPTO

Zur Verifikation unseres Konzeptes bildeten wir vermittels von UNIX-Prozessen, die über pipes kommunizieren, das zu realisierende Mehr-Mikroprozessorsystem auf unseren Host-Rechner ab. Man kann dabei eher von einer Software-implementierten Emulation mehrer SBC's durch einen (Host-) Rechner sprechen als von einer Simulation im herkömmlichen Sinn.

Die Eingabe vom Terminal, die von allen SBCs gleichermaßen entgegengenommen wird, wird durch den sogenannten Supervisor-Prozeß simuliert. Dieser kopiert die Daten und schickt sie dem Terminal-Prozeß einer jeden FTL zu. Auch die Kommunikation der Porthandler wird über pipes abgewickelt: Hardware-Interrupts werden dabei durch Software-Interrupts (Signale) ersetzt.

Soweit es auf einem Ein-Prozessor-System möglich ist, spiegelt die Simulation so die wesentlichen Elemente der Logik in der Kommunikation innerhalb und zwischen einzelnen Exemplaren von ATOS wieder. Das tatsächliche zeitliche Verhalten konkurrierender Prozesse und Prozessoren kann auf diese Weise natürlich nicht untersucht werden.

Das emulierende System kann für sich genommen als die Realisierung eines Konzeptes für Fehlertoleranz gegenüber transienten Fehlern angesehen werden: Zeit-Redundanz simuliert Hardware-Redundanz.

Als einen weiteren Vorteil dieses Konzeptes bietet uns diese Wirklichkeitsnähe eine komfortable Testumgebung unserer Software. Über dieses Simulationsverfahren soll an anderer Stelle noch berichtet werden.

6. Schlußbemerkung

Bei dem Entwurf von ATTEMPTO haben wir uns auf einige wenige, allgemeingültige Konzepte der Fehlertoleranz beschränkt. Dabei war uns die einfache Umsetzbarkeit dieser Konzepte in die gewählte Sprache zur Systemprogrammierung wichtig. Zudem durften nur geringfügige Änderungen der Hardware nötig werden.
Es ist nicht Ziel unserer derzeitigen Implementierung, hinsichtlich der Effektivität in Konkurrenz zu kommerziellen (womöglich nicht fehlertoleranten) Arbeitsplatz-Rechnern zu treten.
Wir sind allerdings der Überzeugung, daß die vorgestellte Kombination von asynchroner Fehler-Maskierung zusammen mit einer effizienten Fehler-Diagnose einem Vergleich mit anderen, aufwendigeren Verfahren wie Checkpointing oder roll back durchaus standhalten kann.

Dies zu zeigen, ist Ziel der Verwendung von ATTEMPTO als Testbett. In unserer Anwendung als Testbett ist z.B. die Möglichkeit vorgesehen, auch die Anzahl der Kollegen in gewissen Grenzen unabhängig vom Fehlertoleranzindex vorzugeben.

Insofern stellt ATTEMPTO den Prototyp eines Rechners dar, der etwa in der Entwicklung als Mehr-Prozessor-Multi-Tasking-System und bei zuverlässigkeitskritischen Anwendungen als fehlertolerantes System Verwendung finden kann.

Diese Arbeit wurde von der Deutschen Forschungsgemeinschaft (DFG) unterstützt.

Literatur

Ammann, E., Brause, R., Dal Cin, M., Dilger, E., Lutz, J., Risse, Th.: ATTEMPTO: A Fault-Tolerant Multi-processor Working Station; Design and Concepts; Proceedings of the 13-th International Symposion on Fault-Tolerant Computing, FTCS-13, IEEE New York 1983, p.10-13

Ammann, E., Brause, R., Dal Cin, M., Dilger, E., Lutz, J., Risse, Th.: Theoretical Aspects of Test and Diagnosis in ATTEMPTO, Proceedings of the FTSD'83, CSSR Brünn, 1983, p.84-87

Brause, R., Ammann, E., Dal Cin, M., Dilger, E., Lutz, J., Risse, Th.: Software-Konzepte des fehlertoleranten Arbeitsplatzrechners ATTEMPTO; in W.Remmele, H. Schecher (Hrsg.): Microcomputing II, Tagung III/83 des German Chapter of the ACM, Teubner Stuttgart, 1983, p.328-341

Geyer, Johann: 32-bit-Mikrocomputer besitzt neuartige Architektur, Elektronik, Mai 1981, p.59-66

Lamport, L.: Using Time Instead of Timeout for Fault-Tolerant Distributed Systems; ACM Trans. Program. Lang. Syst. 6.2, 1984, p.254-280

Liskov, B.: On Linguistic Support for Distributed Programs; IEEE Trans. on Software Engineering, VOL SE-8, No. 3, May 1982, p.203-210

Parker, Y.: Multi-Microprocessor Systems; Academic Press 1983

Risse, Th., Dal Cin, M., Dilger, E.: Zur Verwendung fehlertoleranter Daten-Strukturen im Arbeitsplatz-Rechner ATTEMPTO; erscheint in Informatik Fachberichte, Springer 1984

Wensley, J.H.: SIFT - Software Implemented Fault Tolerance; Fall Joint Computer Conference, 1972

Wensley, J.H., Lamport, L., Goldberg, J., Green, M.W., Levitt, K.N., Milliar-Smith, P.M., Shostak, R.E., Weinstock, C.B., Berson, D.: SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control; Proc. IEEE 66, 10, 1978, p.1240-1254

Wirth, Niklaus: Programming in Modula-2; Berlin, Heidelberg, New York, 1983

Saltzer, J.H. et al.: End-to-end arguments in system design, Proc. 2nd Int. Conf. Distributed Computing Systems, Paris 1981, p. 509-512