# オフィス手続き自動化システムＯＰＡにおける同時実行制御及び障害回復機能の設計

吉本真二、　岸本一男、　平松健司、　翁長健治
Shinji Yoshimoto, Kazuo Kishimoto, Takeshi Hiramatsu, Kenji Onaga
広島大学工学部

## １．はじめに

近年、ＯＡ（Office Automation）の研究・開発、そして、その実用化が広範囲にわたって進行してきている。この過程の中で、単一のオフィス業務の自動化から更に進んで、これらの処理を有機的に統合したオフィス手続きの自動化を実現することがＯＡ研究の一つの課題となっている。

著者らは、複数のワークステーション（以下、ＷＳと略す）をつなぐＬＡＮ（Local Area Network）上で、処理と通信を一体化したＯＡ用システムの記述を目的として、分散並列処理言語ＯＰＡＬ（Office Procedure Automation Languages）及び、その動作環境であるＯＰＡ（Office Procedure Automation）－システムを設計・作成してきた[8][9][11][16][17]。

ＯＰＡＬは、Ｐａｓｃａｌ風の手続き的言語であり、オフィス手続き記述のために、メッセージパッシングに基づくＬＡＮ上の複数ＷＳ上にまたがる並列処理機能及び、２次元表形式のファイルの操作能力をサポートしている。書類転送・スケジュール調整手続き・物品購入手続き等の事務手続きの自動化においては、書類転送等担当者間の事務通信の実現が不可欠であるが、ＯＰＡＬでは、これを複数ＷＳ間にまたがる遠隔手続き呼び出しを通じて記述する。

ＯＰＡＬのプロトタイプは、現在 NEC-PC-9801をつないだＬＡＮ上で稼働中である。しかし、ＯＰＡＬを実用に供するためには、その実行速度等の性能上の問題を別にしても、なお次の重大な未解決問題を残している。

１．同一のファイルに多数のトランザクションが同時にアクセスする場合、データの一貫性を保つための同時実行制御が必要なことは、データベースの理論等でよく知られた事実であるが、現在のところＯＰＡＬにおいてはこれは基本的に利用者の責任で解決しなければならない。（ここで、複数当事者間の不在等における通信の時間待ち等を考えると、プログラム処理時間が数日以上となり、通常の意味でのファイルロック等を直ちに用いることができないことが問題を更に困難なものにしている。）

２．複数のＷＳで処理を行っているとき、途中のＷＳの１つで０除算，誤動作による電源切断等の障害が発生した場合に、ＯＰＡＬ処理系上でデータの一貫性を保ちながら障害回復を行い、必要に応じて再実行を行わなくてはならないが、現在のところＯＰＡＬにはこの機能は設計実現されていない。

複数ＷＳ間にまたがる並列処理言語は既にいくつか報告されている[2][5][15]が、これら１，２の問題についてはいずれも全く考慮が払われておらず、今後の研究が待たれている。また、ＯＡの記述に関して本研究と同様なアプローチを目指すものとして[7]があるが、未だ実装を目指した十分な考察を払うところまではいたっていない。

本研究では、ＯＰＡＬに対してこれらの問題解決を考察し、同時実行制御，障害回復機能の設計を行ったので報告する。

## ２．ＯＰＡＬ／ＯＰＡ－システムの概略

本章では、著者らが提案してきたオフィス手続き自動化言語ＯＰＡＬとその実行環境であるＯＰＡ－システムの概略について、本研究で必要な範囲内で説明する。

### ２．１　オフィス手続き自動化言語ＯＰＡＬ

ＯＰＡＬは、ＬＡＮ上の複数ＷＳ上での人間の判断に基づく対話的入力処理を含むオフィス手続きの自動化システムの開発を目的として設計された並列処理用高級言語である。ＯＰＡＬプログラム（ＯＰ－プログラム）は、各ＷＳ上での処理，人間による対話的入出力処理，各ＷＳ間の通信の３つを単一のプログラム中で記述し、同期を取りながら並列実行する。ＯＰＡＬでは、このような機能を実現するために以下のような特徴を備えている。

1）ＬＡＮ上の各ＷＳ上で実行されるプロセス（プロック）を分散並列実行するために”プログラム分配実行方式”を採用しており、コンパイル時に各プロセスは関係ＷＳ上に転送される。

2）プロセス間の通信は遠隔手続き呼び出しで行うが、遠隔手続き呼び出し後も、呼び出し側のプロセスを返答待ちの地点まで続けて実行させることができる。この時、障害回復のための実行制御が行えるように、”メインプロセス”のみが遠隔手続きを実行できる。従って、ＷＳ間の通信は全てメインＷＳを通じて行われる。

3）各ＷＳ上には、他のＷＳからアクセス可能な２次元表形式のファイルシステムが用意されている。

4）対話的入出力画面生成ツール（ＯＰ－Ｓｃｒｅｅｎ）による自動入出力フォーマッティング機能を持つ。

## ２．２　ＯＰＡ－システム

ＯＰＡ－システムは、ＬＡＮによって結合された数十台程度のマイクロコンピュータ上で、特別なホストマシンを前提とせずに動作することを想定している。各ターミナル上には必ずディスクが装着されており、他の周辺装置（プリンタ，ハードディスク等）も必要に応じて準備されているものとする。また、マイクロコンピュータは必ずしもハード的に同一である必要はなく、ＯＳも異なっていてもよい。しかし、各マイクロコンピュータ上にはＯＰＡＬを動作させるためのソフトウェア及び、ディレクトリが準備されていなければならない。図１にＯＰＡ－システムのソフトウェア構成を示す。

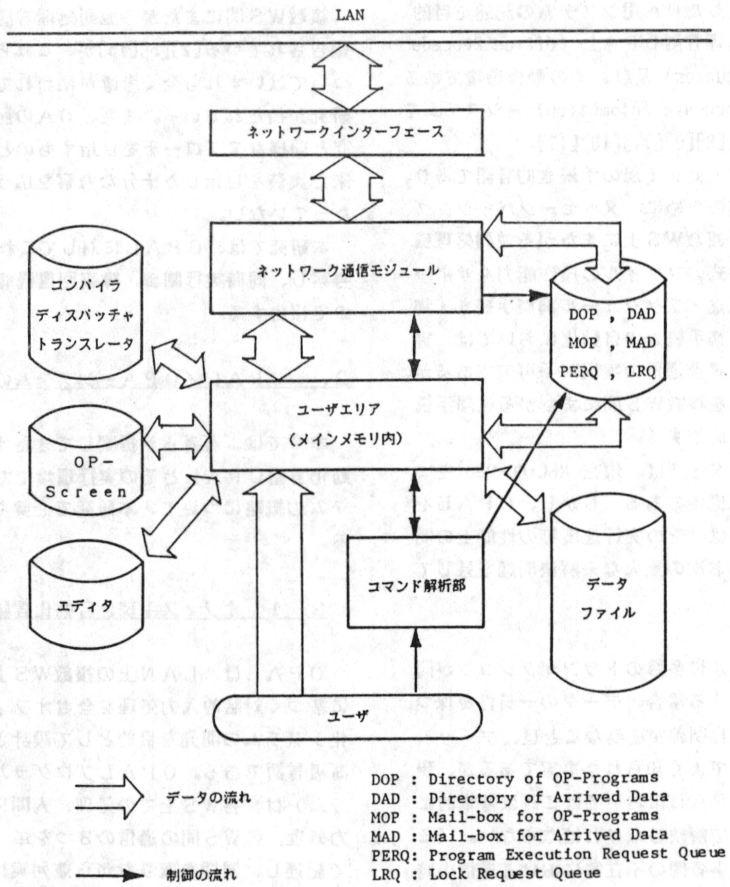


図1　１つのＷＳ上のＯＰＡシステムのソフトウェア構成

### 2．3　OP－プログラムの作成・実行方法

OP－プログラムは、複数のWS上で実行されるために各WSごとに分けて記述される。実行が複数のWSに及ぶため、OP－プログラムの作成・コンパイル・実行は次のような手順で行われる。

① エディタを用いてOP－プログラムを作成する。対話的入出力を含む画面のフォーマットはOPA－システムに準備された対話型入出力画面生成ツール（OP－Screen）を用いて定義する。
② ①で作成されたOP－プログラムをOPALコンパイラにかけ、中間コード及び分配に必要な情報を生成する。
③ ②で出力された中間コードを分配情報をもとに、ディスパッチャによりLAN上の電子メールシステムを用いて、各WSに自動的に分配する（OP－プログラムのLAN上へのロード）。
④ 分配完了後、各WSでは自動的にトランスレータによって、中間コードから実行すべき計算機に応じたオブジェクトコードを生成し、必要な情報をディレクトリに登録する。
⑤ 以上によってOP－プログラムは実行可能となり、メインプロセスの存在する任意のWS上からコマンドを入力することによって、プログラムの実行を開始することができる。メインプロセスを起動したWSがメインWSとなって実行の監視を行う。

OP－プログラムの作成から実行に至るまでの方法を”プログラム分配実行方式”と呼ぶ。

## 3．　同時実行制御

### 3．1　OPALにおける同時実行制御

OPAL／OPA－システムの目的は、会議調整手続き，承認手続き等に見られる”処理が複数部署にまたがるオフィス手続き”の自動化である。これらの手続きは、単一部署での処理とそれらの間での情報転送から成る。これらの手続きでは、各WS上で複数の共有ファイルにアクセスすることが許されており、この結果OPALにおいてデータの一貫性を保つための同時実行制御が必要になる。

OPA－システムにおいて同時実行制御を考える場合、オフィス手続きを次のように2つに分類してみると分かりやすい。

① 注文処理，決算処理，給与処理，検索処理等の手続きのように、人間による対話的入力が高々1回しかない手続き。すなわち、1回の対話的入力で検索ファイル名や初期値データを入力し、以後、対話的入力を含まずバッチ処理的に実行される手続き。
② スケジュール調整，書類の転送，物品購入等の手続きのように、処理が複数の部署にまたがり、人間による対話的入力が複数個あるような手続き。

①のような手続きの場合、その処理がたとえ複数の部署にまたがって実行されたとしても、実時間に近い処理が行えると考えられる。従って、通常のファイルのロック方式による同時実行制御でもよい。しかし、②のような手続きが自動化された場合、その処理は各部署のWS上での処理，複数担当者の判断・決定の入力，各部署間の通信の3つからなると考えられる。これら3者を全体として一体化したとき、担当者の不在のため各WS上の担当者の決定の入力に、場合によっては数日の遅れが予想され、通常のファイルロック方式による同時実行制御では処理の進行が著しく妨げられることになる。

この問題に対して、本研究では次のようなアプローチを取る。

（1）共有ファイルへの全てのアクセスをできるだけ全ての対話的入力が終了した後に行うようなプログラミングを奨励する。この様に実行されるトランザクションに対しては、通常のファイルロックに基づく一貫性制御を他のトランザクションよりも優先して割り振りつつ行う。
（2）対話的入力に先立つファイルへのアクセスがどうしても必要な場合（特に、ある種のスケジュール調整のような場合）、ユーザがプログラム中でできるだけ効率よく一貫性の自動判定を行うようなプログラムを書くことを奨励する。
（3）しかし、（2）の場合にユーザが誤って一貫性を損なうプログラムを作成した場合には、

タイムスタンプ方式に基づく一貫性制御を行い、違法なトランザクションをアボートして再実行させる。そのとき（１）を実現するために、各トランザクションのタイムスタンプが押される時間を全ての対話的入力が終了したときに設定する（のと等価な）アボート方式を採用し、この時点でファイルのロックを行う。

このようなアプローチにより、不適切に書かれたプログラムに対して実行効率が著しく低下する場合がありうるが、最悪の場合でもデータの一貫性だけは保証される。

### ３．２　同時実行制御の概略

同時実行制御の方法については、データベースの分野においてロックやタイムスタンプ等の技術[4]が開発されてきている。また、分散データベースの分野においては、それらの技術を基本としていろいろな方式が開発されている[1][3][10][14]。本研究では、３．１節で述べたように同時実行制御及びデータの一貫性確保の方法として、

（１）ファイルのロック方式
（２）データにタイムスタンプを付加したThomasの方式[10]

の２つを組み合わせた方法を考える。その概略は次のとおりである。

（Ａ１）ファイル上のデータレコードには、各レコード毎にタイムスタンプ欄が用意されている。タイムスタンプは各ＷＳ毎に局所的に定まる単調増加な自然数とし、データが更新されるときに値を１つインクリメントする。

（Ａ２）最後の対話的入力が完了する以前に、共有データへのアクセスがある場合そのデータをコピーし、最後の対話的入力が完了するまではそのコピーデータに対してアクセスする。他のトランザクションに対しては、まだこのデータを参照させない。（この”データ”は、概念的には個々のレコードであるが、実際のインプリメントにおいては１つのファイル全体のコピーを行うものとする。）

（Ａ３）コピーデータのアクセスについては、データを読み込んだか読み込まなかったか、またデータの書き込みを行ったか行わなかったかどうかの別を各レコード毎に記憶しておく。

（Ａ４）最後の対話的入力が完了した時点で必要なデータに対してロックをかける。但し、データにロックをかけることができるのは、コピーデータ上のタイムスタンプと原本データ上のタイムスタンプを比較して、”今までに読み込んだ全てのデータが他のトランザクションによって変更されてない”という条件を満足するときである。もし、読み込んだデータに１つでも変更があった場合には、データの一貫性を保つためにそのプログラムはアボートされる。

（Ａ５）必要な全てのファイルに対してロックが獲得できれば、コピーファイルを原本と見なして処理を継続する（実際のインプリメントについてはファイル単位でコピーを行っているため、読み込まなかったデータに対して原本上の値は変更されているがコピー上の値は変更されていないというケースが起こりうる。このような場合には、コピーファイル上に原本上の値とタイムスタンプを反映させる）。ロックはプログラム終了時に解放される。

以上のことをまとめると、次のようになる。

最後の対話的入力が完了して必要なファイルにロックをかけるまでの間、他のトランザクションは原本データに読み込み，書き込みを行うことができる。そこで各トランザクションは、最後の対話的入力が完了した時点でデータのタイムスタンプを比較することによりデータの一貫性をチェックする。もし、一貫性が保たれているならば、必要なファイルにロックをかけて処理を進める。しかし、この時点で一貫性が保たれていないことが分かれば、以後の実行を打ち切ってプログラムをアボートさせる。つまり、最後の対話的入力が完了するまではタイムスタンプによって、また、最後の対話的入力が完了した後はファイルのロックによって同時実行制御を行う。もし、対話的入力が全くない場合は、プログラムの実行開始時に必要な全てのファイルにロックをかけて処理を進める。この時は、タイムスタンプの比較は不要になる。

以上の処理は、ロック時にタイムスタンプを押して、その順序によってトランザクションを順序づけたのと等価になっており、データの一貫性が保証される（３．４節参照）。

### ３．３　ロックアルゴリズム

ファイルのロックは次のように実行される。

Ｓｔｅｐ１．

最後の対話的入力をメインＷＳが確認したら、直ちに必要なファイルが存在するＷＳにロック要求を出す。メインプロセスはロックが成功したかどうかの返答を待つ。

Ｓｔｅｐ２．

ロック要求を受けたＷＳでは、

割り込み処理により、他のプログラムが実行中でもロック管理テーブルを参照し、必要とするファイル全てにロックが掛けられているかどうかチェックする。

既に、１つでもロックされている場合

ロック失敗信号をメインＷＳに送信する

全てにロックされていない場合

全てのファイルにロックを掛け、ロック成功信号をメインＷＳに送信する。

このＷＳはメインＷＳからの返答を待つ。

Ｓｔｅｐ３．

返答待ちのメインＷＳでは、

全てのＷＳからロック成功信号を受信すれば、

関係ＷＳにロックＯＫ信号を送信し、返答を待つ。

もし、１つでもロック失敗信号を受信すれば、

ロック成功信号を出したＷＳに対し、ロック解除信号を送信する。

メインプロセスはディスクに待避され、タイマ割り込みにより定期的にロック要求を関係ＷＳに送信する（Ｓｔｅｐ１に戻る）。

全てのＷＳから、ロック成功信号を受信したとき、関係ＷＳにロックＯＫ信号を送信し、メインプロセスを再起動する。

Ｓｔｅｐ４．

ロックＯＫ信号を受信したＷＳでは、

今までに読み込んだ全てのレコードのタイムスタンプ欄を比較し、データを読み込んだ後に変更があったかどうかを調べる。

読み込んだデータに対して１つでも変更があった場合

データの変更があったためにプログラムをアボートすること（アボート発生信号）をメインＷＳに通知し、実行をアボートする。

読み込んだデータに対して全く変更がなかった場合

読み込まなかったデータについて、原本の値は変更されたがコピー上の値は変更されなかった場合に、コピーファイル上に原本の値を反映させる。

原本ファイルをバックアップコピーとして保存しておく。

メインＷＳに書き込み許可信号を送信する。

Ｓｔｅｐ５．

返答待ちのメインＷＳでは、

１つでもアボート発生信号を受信すれば、

関係ＷＳにアボート信号を送信する。

全てのＷＳから書き込み許可信号を受信すれば、

ロック獲得処理終了。その後の処理を継続する。

Ｓｔｅｐ６．

処理が最後まで終了すれば、ロックを解放する。

Ｓｔｅｐ１－Ｓｔｅｐ５までを図示すると図２のようになる。

### ３．４　データの一貫性

本節では、本研究で提案した同時実行制御を行った場合に、データの一貫性が正しく保たれていることを示す。一般に一貫性が保証されていることを示すには、プログラムの実行がSerializableであることを示せばよい[1][4][14]。

３．２節で述べたように、最後の対話的入力が完了する以前のファイルへのアクセスは、ファイルのコピーを用意してそのコピー上で行う。（ファイルのロックは最後の対話的入力が完了した後に獲得されるのだ

から、ロック以前に読み込んだ値はロックを獲得した時点の原本ファイル上のデータと必ずしも一致していないかもしれない。）

３．３節で提案したアルゴリズムでは、データの一貫性をたもつためにＳｔｅｐ４で読み込んだ全てのデータのタイムスタンプを比較し、データを読み込んだ後に変更があったかどうか調べている。もし、読み込んだデータに１つでも変更があれば、今までの処理はデータの一貫性が保証されなくなるので、以後の処理を進めてもその処理は無効である。本アルゴリズムではこの様な場合、プログラムをアボートするのでデータに矛盾が生ずることなく一貫性が保たれる。読み込んだ全てのデータに全く変更がないとき、読み込まなかったデータが原本上では変更されているが、コピーファイル上では変更されていない場合に限り、原本上の値をコピーファイル上に反映している。これによる結果は、ファイルにロックをかけた時点で各トランザクションにタイムスタンプを押したとして順序づけたことになっている。従って、このようなトランザクション集合はserializableである。

## ３．５　デッドロック

３．３節で述べたアルゴリズムでは、Ｓｔｅｐ１で最後の対話的入力が完了した時点で必要な全てのファイルのロックを行う要求を出す。この時、Ｓｔｅｐ２でロック要求を受けたＷＳでは、自分の管理下にある必要なファイル全てにロックできたときロック成功信号をメインＷＳに返す。Ｓｔｅｐ３で、メインＷＳは全ての関係ＷＳからロック成功信号を受けた時以後の処理を進めることができるが、もし、１つでもロックに失敗すれば、Ｓｔｅｐ２でロックされたファイルの解放を行い、以後定期的にロック要求を関係ＷＳに送信し、全てのファイルにロックが成功するまで待つ。

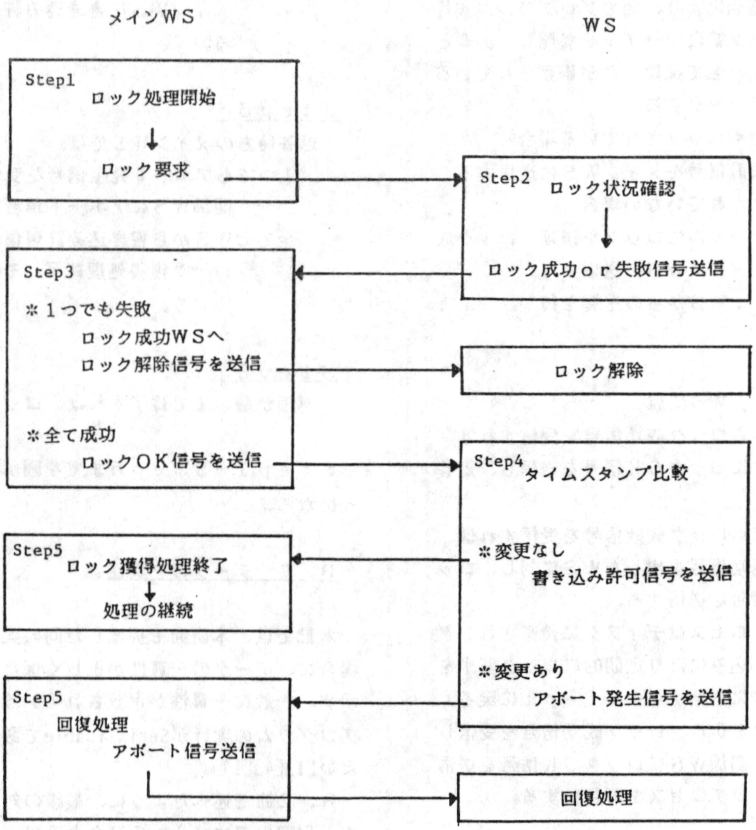


図２　ロックアルゴリズムの概略

従って、ロックが必要なファイル群の一部にロックをかけたままの状態で、残りのファイルにロックが獲得されるのを待つという状態は発生しないため、本研究で提案した同時実行制御方式では、デッドロックは発生しない。しかし、全てのファイルに対して同時にロックが獲得できなければならないために、いつまでたっても全てのファイルにロックが獲得できないまま待ちの状態が続く恐れがあるが、ファイルのロック中には対話的入力がないため、長期間にわたって待ち状態が続くことはないと考えられる。

### 3．6　コミット処理

メインプログラムの処理が全て終了した（アボート発生がなかった）のを確認したら、メインプロセスはコミット処理にはいる。本研究では2相コミット方式[10]によってコミット処理を行うものとする。以下に、本研究でのアルゴリズムを説明する。

① Logファイルを参照し、ファイルのアクセスがあったWSに対してデータのコミット信号を送信する。

② コミット信号を受信したWSでは、他のプログラムが実行中であってもロックをかけていたファイルのクローズを優先的に行う。
　ファイルのクローズに成功すれば、
　　クローズ成功信号をメインWSに送信する。
　もし、何等かの理由でファイルのクローズに失敗すれば、
　　メインWSにアボート信号を送信する。

③ 返答を待っていたメインWSでは、
　関係WSから全てクローズ成功信号を受信すれば、
　　ロック解放信号を関係WSに送信し、コミット完了信号を待つ。
　もし、1つでもアボート信号を受信すれば、
　　回復処理（4章参照）を行うために回復信号を全ての関係WSに送信し、回復完了信号を待つ。

④ 関係WSでは、
　ロック解放信号を受信すれば、
　　関係ファイルのロックを解放し、バックアップファイルを消去する。更に、コミット完了信号をメインWSに送信する。
　回復信号を受信すれば、
　　所定の回復処理（4．3．2節参照）を行う。

⑤ メインWSでは、
　コミット完了信号を全てのWSから受信すれば終了。
　もしくは、回復完了信号を全てのWSから受信すれば終了。

## 4．　障害回復

障害回復のメカニズムについては、データベースシステムにおいてトランザクションの概念が導入され、確認点やバックアップファイル，増分ダンプ，複写の写し等の冗長性による技術等が開発されている[3,4,6,10,12,13,14]。しかしながら、OPA－システムのように複数のマイクロコンピュータをLANで結合した比較的小規模なシステムでは、メモリの制約や管理保守、費用等の点からこのような技術はそのままの形では導入できない。また、障害が発生した場合、処理が複数の部署にまたがっているためにデータの一貫性を保ちながら回復しなくてはならないことにも注意が必要である。

一般に、障害の型を分類すると次のように3つに分類される[4]。

① プログラムの実行時エラーによる障害
　いわゆるトランザクション障害であり、不正な入力データによって発生する0除算，配列の添え字の範囲外等の実行時エラーや一貫性の違反・デッドロック検出によって正常な終了に至らない場合。

② システム障害
　オペレーティング・システムの誤りや停電等のために主記憶の内容が失われ、処理が無制御のまま終了してしまう場合。この時、ディスク上に記録された全ての情報は影響を受けない。

③ 媒体障害
　ヘッドクラッシュ等のために、ファイルやデータベースを保持している2次（不揮発性）記憶の一部もしくは全部が失われる場合。

また、分散システムでは上述の3つの障害に加え、

通信による障害も考慮しなければならない[3]。

④ 通信障害
メッセージ伝達中に発生するメッセージの損傷や損失、通信路の切断等によって発生するPartitionといった障害。

更にこれらの障害の他に、ユーザが入力ミス等の意味的な誤りに気付き、プログラムを意図的にアボートすること（通常の計算機でいう”ブレークキー”を押す操作に当たる）は、日常の計算機処理においては起こりがちなことである。プログラムが正常終了するのではなくアボートされるという意味で、ユーザによるアボートも１つの障害と考えることができる。

本研究では、これらの障害のうち①とユーザのアボートによる障害の２つを対象に考え、回復処理に対するアルゴリズムを提案するが、②，③，④の障害については考えないものとする（さらに、回復処理ではエラーは発生しないものとする）。また、ユーザのアボートによる障害が発生した場合、対話的入力によりアボートするプログラムのＩＤ，アボート理由等を入力することにより、プログラムの任意のブロックからの再実行を要求できるといった機能も含めて考える。

### ４．１ 障害の分類

本節では、本研究で取り扱う障害（プログラムの実行時エラーによる障害、ユーザのアボートによる障害）について説明する。また、それぞれの障害が発生した場合、プログラムの進行を監視しているメインＷＳに障害発生を通知する方法について説明する。

#### ４．１．１ プログラムの実行時エラーによる障害

この障害は、データベースの分野でいうトランザクション障害である。不正な入力データや計算処理の途中で発生する０除算、スタックのオーバーフロー、ファイルのオープン・クローズエラー、配列の添え字の計算の範囲外となるエラー等を取り扱う。また、ファイルのロック獲得時の一貫性チェックによるアボートもこのタイプのエラーと考える。表１に本研究で取り扱うプログラムの実行時エラーを示す。

エラーの検出については、表１中のエラーを検出するためのコードをコンパイラによって生成しておくため、検出は容易に行うことができる（ただし、一貫性のチェックは図１中のネットワーク通信モジュールが行う）。また、エラーが発生した場合にメインＷＳに通知するためのルーチン（エラー発生通知ルーチン）をライブラリー中に用意しておく。もし、エラーが発生すれば以後の処理は中止し、エラー発生通知ルーチンに制御を移してメインＷＳにエラーを通知する。エラーが発生したＷＳではメインＷＳからの指示を待つ。この場合、プログラムの処理は全てキャンセルされる（ＵＮＤＯ）。

メインＷＳに通知するデータを以下に示す。

1） プログラムーＩＤ
2） ブロックナンバー
3） ＷＳナンバー
4） アボートの原因
5） プログラムの実行時エラーであること

表１ プログラムの実行時エラー

| エラー番号 | エラータイプ |
|---|---|
| 1 | スタックオーバーフロー |
| 2 | 保護領域への不当なアクセス |
| 3 | ポインタ値が不当 |
| 4 | 変数の値が規定の範囲を越えている |
| 5 | 乗算によるオーバーフロー |
| 6 | ゼロによる割り算の実行 |
| 7 | ＣＡＳＥエラー |
| 8 | 入力ファイルに対する書き込みの実行 |
| 9 | ファイルポインタ値の誤り |
| １０ | 出力ファイルに対する読み込みの実行 |
| １１ | 入力オーバーフロー |
| １２ | ファイルの書き込みエラー |
| １３ | オープンされていないファイルの参照 |
| １４ | 全てのバッファが一杯である |
| １５ | ファイルの作成不能 |
| １６ | ディスク装置番号の指定誤り |
| １７ | 入力ファイルへの出力指定 |
| １８ | オープンされていないファイルのクローズ |
| １９ | 存在しないファイルからの入力指定 |
| ２０ | ファイルをクローズできない |
| ２１ | ファイルの読み込みエラー |
| ２２ | ＥＯＦを越えて読み込みを行った |
| ２３ | 出力ファイルへの入力指定 |
| ２４ | 最初のレコードの読み込み失敗 |
| ２５ | ファイルが既にオープン去れている |
| ２６ | 一貫性エラー |

### 4．1．2．　ユーザのアボートによる障害

ユーザのアボートとは、例えば届いたデータの中に誤りを見つけた場合や、ユーザが対話的入力によって自分のＷＳでの処理を終了した後に入力データ等の誤りに気が付いた場合、意図的にプログラムの実行をアボートする場合をいう。本システムではユーザのアボートのための手続き（アボート手続き）を用意し、プログラムをアボートするときに呼び出す。ユーザによるアボートでは、任意のプログラムのアボートを行うことができる。また、オフィスでは書類転送等で届いた書類の内容が間違っていた場合、前の担当者に書類を差し戻さねばならないということも起こりうる。この様な場合を考慮してユーザによるアボートでは、プログラムの実行を中止して任意のブロックから再実行できる機能もサポートしている。メインＷＳに引き渡すデータとして次のデータを入力する。

1）プログラムーＩＤ
2）再開始ブロックナンバー
3）ＷＳナンバー
4）アボートの理由
5）ユーザによるアボートであること

入力が完了しメインＷＳにデータを送信した後は、メインＷＳからの指示を待つ。

### 4．2　障害回復に必要な情報

本システムでは障害回復を行うために必要な情報として、Logファイル・ブロック起動情報ファイル・REDOバックアップファイルの３つの情報ファイルを用意している。以下、これらの３つのファイルについて説明を行う。

#### 4．2．1　Logファイル

本システムでは、１つのプログラムの実行をメインプロセスが存在するＷＳ（メインＷＳ）が監視し、障害が発生すると直ちに回復を行う。この時、１つのプログラムは複数のＷＳ上で実行されているので、回復に際してシステムはプログラムの処理がどこまで進んだかを知っていなければならない。この情報を格納しておくファイルがLogファイルである。

Logファイルは各ＷＳ上に１つずつ用意されたシステムファイルで、各ＷＳ上で実行されたプログラムの進行状況を記録している。このファイルへの書き込みアルゴリズムは次の通りである。

1）プログラムが起動されたとき、そのプログラムの識別子を持ったBEGIN-PROGRAMレコードを書き込む。

2）１つのプログラムを構成しているブロックが起動された時、そのプログラム及びブロックの識別子等を持ったBEGIN-BLOCKレコードを書き込み、ブロックが終了したときEND-BLOCKレコードを書き込む。但し、他のＷＳのブロックを起動する場合には、起動のための通信が行われたときにBEGIN-BLOCKレコードを、また、そのＷＳからメインプロセスのブロックを起動するための通信を受信したときにEND-BLOCKレコードを書き込む。

3）メインＷＳのデータコミット要求に対して、全てのＷＳからコミット完了信号を受信したとき、そのプログラムに対する全てのＬＯＧレコードを消去する。

#### 4．2．2　ブロック起動情報ファイル

このファイルは、次節で説明するREDOバックアップファイルと共に、ユーザのアボートによって任意のブロックから再起動する場合に必要とされるファイルである。

これは、各ＷＳ上で各ブロックが起動されたときに各ブロックごとに作られる。このファイルには、ブロックを起動したときに必要としたデータと、そのときのシステム変数領域・スタック領域の内容が保存されている（メインＷＳ以外のＷＳでは、ブロックを起動したときに必要としたデータのみが保存されている）。

#### 4．2．3　REDOバックアップファイル

このファイルは、ユーザのアボートによって任意のブロックから起動する場合に必要とされるファイルである。

これは、各ブロック中でファイルのアクセスがある場合に、ブロック起動時のファイルの状態を保持しておくためのバックアップファイルで、各ＷＳ上でブロックが起動されたときに各ブロックごとに作られる。このファイルは、前節のブロック起動情報ファイルと

共にプログラム終了時に全て消去される。

### ４．３　回復処理

本節では、障害回復のための処理を

１）アボート発生を受信したメインＷＳ上での処理

２）回復信号を受信したＷＳ上での処理

の２つに分けて説明する。

#### ４．３．１　アボート発生を受信したメインＷＳ上での処理

メインＷＳではアボート発生を受信すれば、どんな処理よりも最優先して回復処理にあたる。まず、この処理についてそのフローチャートを図３に示す。

処理１では、Logファイルを参照しアボートするプログラムが通信を行った全てのＷＳに対して回復信号を送信する。回復信号のデータは以下の３つである。

１）　プログラム－ＩＤ

２）　アボートの原因

３）　アボート発生ＷＳナンバー

この場合、回復が完了した後プログラムを再起動するかは、プログラムを起動した人間が決定する。

処理２では、Logファイルとユーザによって決定された再実行ブロックから、再実行ブロック以後のＷＳに回復信号を送信する。さらに、再実行ブロックが存在するＷＳには実行要求を送信する。回復信号データには、上述した３つのデータの他に

４）　再実行ブロックナンバー

も加えて送信する。

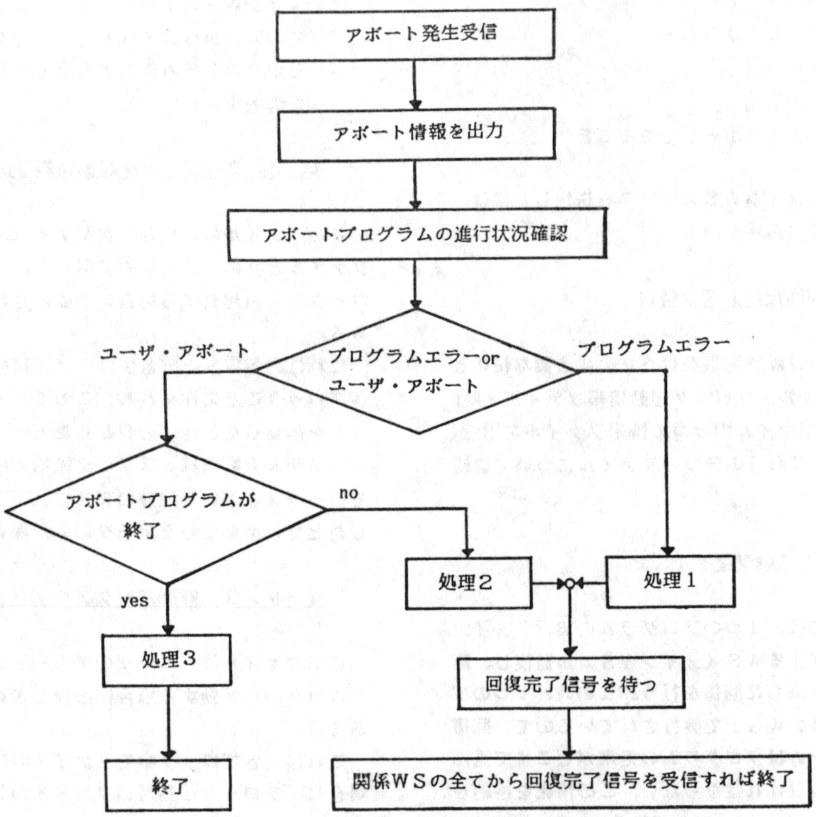


図３　メインＷＳ上での回復処理の流れ

処理３では、アボートを希望するプログラムが既に終了（コミット）しているので、実行結果は取り消すことができない。アボート要求を行ったＷＳに対してアボート不可を通知する。本研究ではこの場合の対処については全く考慮してなく、以後の対処についてはユーザの判断によるものとする。

メインＷＳは、回復完了信号を全ての関係ＷＳから受信すれば、全ての関係ＷＳに回復処理終了信号を送信し、回復処理を終了する。

### ４．３．２　回復信号を受信したＷＳでの処理

今までの説明から回復処理を行うことが可能な場合は、プログラムが終了（コミット）していない場合であることがわかる。各ＷＳでは、回復信号を受信すれば他の処理が実行中であっても最優先して以下の処理を行う。

① アボート情報を画面上に出力する。アボート情報は次のデータから成る。
　１）　アボートプログラム名
　２）　アボート理由
　３）　アボート要求をしたＷＳナンバー
　４）　再実行する場合には、再実行開始ブロックナンバー

② 実行要求キューの内容を参照し、アボートするプログラムの実行要求がキューの中にあるかどうかチェックする。
実行要求がキュー中にある（実行要求は受信しているが、実行はされてない）場合、
　実行要求をキューから削除する。

③ 実行要求がキュー中にない（実行要求が既に処理されている可能性がある）場合、
　各ＷＳ上のLogファイルを参照し、アボートするプログラムの進行状況をチェックする。
　アボートするプログラムが現在実行中であれば、実行中止。

④ ファイルへのアクセスがあり、
プログラムの実行を全て取り消す（プログラムの実行時エラー）場合、
　(1)　コピーファイルとブロック起動情報ファイル，REDOバックアップファイルを消去する。
　(2)　Logファイル上からアボートするプログラムに関する情報を消去する。
　(3)　ファイルに対してロックを獲得していれば、バックアップファイルを原本として復活させる。
プログラムを途中から再実行する（ユーザによるアボート）場合、
　(1')　(1)において、再実行ブロックのブロック起動情報ファイル，REDOバックアップファイルは残しておく。

⑤ 再実行要求がある場合は、実行要求をキューに登録する。

⑥ 回復完了信号をメインＷＳに送信し、メインＷＳから回復処理終了信号を待つ。回復処理終了信号を受信すれば中断していた処理を再開する。

## ５．　おわりに

著者らが提案しているオフィス手続き自動化システム開発のためのオフィス手続き自動化言語ＯＰＡＬ並びにＯＰＡ－システムのインプリメントにあたり、本研究では、

１）ロックとロック開始時でのタイムスタンプ比較という２つの方法を組み合わせた同時実行制御方式の提案

２）ＯＰＡＬ処理系上で障害が発生した場合、データの一貫性を保ちながらの障害回復方法の設計

を行った。

今後の課題としては、

① 早急にインプリメントを行い、本研究で設計した同時実行制御方法及び障害回復機能が正常に動作することを確認する

② 実行効率の評価を行う

③ 今回考慮しなかった障害のうち、システム障害について考察し回復機能の拡張を行う

こと等があげられる。

＜＜　謝　辞　＞＞

本研究は文部省科学研究費一般研究（Ａ）の援助を得て行っている。

## [[ 参 考 文 献 ]]


[1] Bernstein, P. A., Goodman, N.:"Multiversion concurrency control-theory and algorithms," ACM Trans. Database Syst., 8, 4, December 1983, pp.465-483.

[2] Campbell, R. H.:" Distributed path Pascal," in Distributed Computing Systems, Y.Paker, J.P.Verjus, Academic Press, London, pp.191-223, 1983.

[3] Ceri, S., Pelagatti, G. : " Distributed Databases, Principles and Systems," McGraw-Hill Book Company : New York, 1985.

[4] Date, C. J.:"An Introduction to Database Systems, Volume II," Addison-Wesley Publishing Company : Reading, 1983.

[5] Feldman, J. A.:" High level programming for distributed computing," Commun. ACM, 22, 6, June 1979, pp.353-368.

[6] Haerder, T., Reuter, A.:"Principles of transaction-oriented database recovery," Computing Surveys, Vol.15, No.4, December 1983.pp.287-317.

[7] 石井 裕:" オフィスモデルＯＭ１によるオフィスワーク記述と分析," 信学技報, No.86-143, pp.39-46, 1986.

[8] Kishimoto, K., Onaga, K., Utsunomiya, H.: OPAL : An office procedure automation language for local area network environments via active mailing and program dispaching, in "Language for Automation," Chang, S-K., Ed., Plenum Publishing Corporation : New York, 1985, pp.67-93.

[9] Kishimoto, K., Onaga, K., Orita, T., Yoshimoto, S. : "Lecture with dialogue through OPAL," Proceedings First Pacific Computer Communication Symposium, Oct. 1985, pp.355-357.

[10] Kohler, W. H.:"A survey of techniques for synchronization and recovery in decentralized computer systems," Computing Surveys, Vol.13,No.2, June 1981,pp.149-183.

[11] 織田, 岸本, 吉本, 翁長:" オフィス手続き記述言語ＯＰＡＬとその実行環境ＯＰＡーシステムの機能拡張," 信学技報, No.85-123, pp.39-45, 1985.

[12] Reuter, A.:"A fast transaction-oriented logging scheme for UNDO recovery," IEEE Trans. Softw. Eng., SE-6, 4, July 1980, pp.348-356.

[13] Shrivastava, S. K., Banatre, J.:"Reliable resource allocation between unreliable processes," IEEE Trans. Softw. Eng., SE-4, 3, May 1978, pp.230-241.

[14] Weihl, W. E.:"Data-dependent concurrency control and recovery," ACM Operating Systems Review, 19, 1, January 1985, pp.19-31.

[15] White, G. M.:"A Lisp network for the execution of Lisp programs," Proc. of IEEE Workshop on Languages for Automation, Kent Ridge, Singapore, Aug. 1986, pp.46-50.

[16] 吉本, 山本, 岸本, 織田, 翁長:" ＬＡＮを用いた研究室ＯＡ化システムの設計," 信学技報, No.85-123, pp.29-38, 1985.

[17] 吉本, 岸本, 平松, 翁長," オフィス手続き自動化システムＯＰＡの障害回復機能の設計," 情報処理学会第３３回全国大会講演論文集, pp.1085-1086, 1986.