主記憶に不揮発メモリを用いたシステムの実行状態復元手法

大村 廉[†] 山崎信行[††] 安西祐一郎[††]

本稿では，不揮発メモリを主記憶とするシステムにおいて，突然の電源切断が生じた場合にもその実行状態を復元可能とする手法を提案する．システムの実行状態を回復するためには，システムに接続される周辺デバイスの状態を，CPU，主記憶の状態とともに一貫性のとれた形で復元する必要がある．本手法は，デバイスドライバとデバイスの間で復元状態の一貫性を保証し，これを各デバイスに適用することによってシステム全体の一貫性を保証する．デバイスドライバにはこのためのコードを追加する．本稿ではまず，CPU，主記憶とデバイスの状態を一貫性を維持しつつ復元するための要求事項をメッセージパッシングシステムのモデルを用いて明らかにする．そして，実際のデバイスドライバへの適用方法をコード例を用いて説明する．実験によって提案手法が適切に動作し，またシステムの性能にほとんど影響を与えず実装が可能であることが確認された．

Running State Recovery in Non-volatile Main Memory Systems

Ren Ohmura,[†] Nobuyuki Yamasaki[††] and Yuichiro Anzai[††]

This paper proposes a scheme to recover the running system state when a system is confronted with an unpredictable power failure. For this to be viable, peripheral devices' state must be retrieved as maintaining the consistency with the CPU and main memory state. Our approach is to ensure the consistency of the state between a device driver and device in the first place, adding some codes preserving the information required for the recovery to a device driver, and to apply it to each device driver for retrieving the entire, consistent system state. We disclose the requirement for maintaining the consistency using message-passing system model. We also illustrate the actual way of adapting our scheme to a device driver with sample codes. The experiments showed our scheme performs correctly and could be implemented as barely increasing the system execution time.

1. はじめに

突然の電源切断に面したときにも，その実行状態が復元可能なシステムを構築することは，広く一般に普及する計算機システムの信頼性や利便性を大きく向上させる．さらに現在，「ユビキタス電源」と呼ばれ，様々な形で計算機システムに対して電力を供給する試みがさかんに行われている[15]．これらの電力は基本的に不安定であり，システムへの電力供給は断続的にならざるをえない．このような環境下で動作する計算機システムには，電源切断時のシステムの実行状態を即座に回復するような能力が求められる．電源が失われるごとにOSやアプリケーションの再起動を必要とするようでは，大きくその利便性を損なう．

システムが突然の電源切断に面したときに，その実行状態を復元するためには，電源切断前のシステム全体の状態が保存されており，リカバリ時にそれらが復元可能である必要がある．また，復元される状態はシステム全体で一貫性が保証される必要がある．システムの実行状態には，CPUや主記憶に加えて，周辺デバイスが保持する状態が含まれる．

耐故障性の向上を目的として，fail-stopモデルでの故障に対しシステム状態の復元をするための研究が多く行われている[5),8),9)]．しかし，主にCPUと主記憶の状態を対象としており，システムに接続される周辺デバイスの状態についてはあまり考慮がなされてこなかった．たとえば，libckpt[10)]ではデバイスについてはオープンしているファイルディスクリプタおよびファイル上の注視点（lseek）の情報を復元するのみであり，デバイス自体が持つ状態については考慮されていない．

APM[7)]やACPI[3)]などのシステムの電源管理仕様

† 慶應義塾大学大学院理工学研究科解放環境科学専攻
Science for Open and Environmental Systems, Graduate School of Science and Technology, Keio University
†† 慶應義塾大学理工学部情報工学科
Department of Information and Computer Science, Faculty of Science and Technology, Keio University

では，周辺デバイスの状態も含めたシステムの実行状態復元（サスペンド/レジュームやハイバネーション）を実現する．これらの実装では，システムの電源状態の遷移要求が通知された際に，周辺デバイスを含めたシステムの状態を保存/復元する．しかし，状態の保存処理は電力が十分に残されている間に行われることを前提とするため，突然電源が消失するような場合や保存処理に必要な電力が十分に残されていないような場合には対応できない．

さらに，現在までの技術は，そのほとんどが主記憶の内容は電源切断とともに失われるものとし，明示的な主記憶状態の保存作業を必要とした．また，保存先にはディスクなどの低速なデバイスが利用されていた．一方，現在主記憶として用いることが可能な不揮発メモリの実用化がなされてきている．電源切断時に不揮発である主記憶に残る状態とともに，適切な CPU や周辺デバイスの状態が復帰されれば，ほぼ電源切断時そのままのシステム実行状態を高速に復元可能となる．

そこで本稿では，主記憶が不揮発であるシステムを前提とし，システムが突然の電源切断に面した場合にも，主記憶，CPU，周辺デバイスの状態を一貫性を維持しつつ復元可能とする手法を提案する．本手法は，ソフトウェアベースのアプローチであり，補助的な電源を仮定しない．このため，小型化や軽量化などの要求から，補助的な電源の利用が困難なシステムにも適用可能である．また，不安定な電源環境下で，補助電源に十分な容量の電力が残されていなかった場合など，前述の電源管理仕様を補うことができる．

後に述べるように，主記憶が不揮発となった場合には，CPU の状態についても電源切断時の状態を復元可能と考えることができる．このため，本稿では主に周辺デバイスの状態について注目する．まず，デバイスとデバイスドライバの関係に着目し，これらの間で復元状態の一貫性を保証する．そして，これを各デバイスドライバに適用することでシステム全体の実行状態の回復を行う．具体的には，デバイスドライバにコードを追加し，デバイスや CPU，主記憶の一貫性維持に必要となる情報を保存する．また，コールバック関数を設け，リカバリ時にはこの情報をもとにこれらの状態を復元する．

以下，2 章で，対象とするシステムの前提条件と本稿で用いるモデルについて述べる．3 章では，これらをもとに，基本的なデバイスハンドリングのシーケンスから，必要となる処理について解析する．4 章では例を用いてデバイスドライバへの適用方法について述べ，5 章では提案手法の動作を確認する実験とその結果について述べる．6 章で関連研究と本手法の適用範囲について議論を行い，7 章で本稿をまとめる．

2. 前提条件とシステムモデル

2.1 対象とするシステム

本研究では，主記憶が不揮発メモリで構成されたシステムを対象とする．現在，MRAM や FeRAM など，主記憶に用いられるメモリと同様の方法で read/write アクセスが可能な不揮発メモリが実用化されつつある．これらは速度や容量の面においても，現状の主記憶の代替として十分な能力を持つことが期待されている．

また，電源切断時の CPU の状態は復帰可能とする．電源を監視する IC を用い，システムに供給される電源が一定値以下になった場合に CPU に対して割込みをかける．この割込み処理において，特定の領域に CPU の状態を保存すれば，これは実現可能である．必要であれば，キャパシタなどによって電源の減衰を鈍らせればよい．

周辺デバイスには以下のような前提を設ける．デバイスが動作していないときの状態（idle 状態）は，再初期化および必要な情報の設定を行えば復元可能とする．また，電源切断前のデバイスの状態を復元し同じデバイスへのアクセス（コマンド）を再実行すれば，デバイスは停止前と同一の動作をするものとする．そして，システム上の各デバイスはそれぞれ独立に動作するものとする．デバイスが電源切断によって損傷する場合は考えない．

2.2 システムモデル

複数の要素で構成されるシステムにおいて，各要素間に依存関係が発生しなければ，各要素が独立に状態を復元しシステム全体の一貫性を維持することが可能である．よって，まずデバイスとの依存関係が発生する要素であるデバイスドライバに注目する．そして，個々のデバイスとデバイスドライバの間で一貫性を維持するリカバリ手法を確立する．その手法をシステムに接続される各デバイスドライバにそれぞれ適用することによって，システム全体の状態を回復することができる．

本手法ではデバイスドライバにコードを追加し，デバイスドライバやデバイスの情報を保存/復元する．追加されるコードでは，デバイスドライバとデバイスの間で一貫性を維持するように，必要となる情報を保存する．次章で，いつ，何を保存すれば一貫性を維持できるか，ということについて解析する．これには，メッセージパッシングシステムで用いられるモデル[5] を利用する．

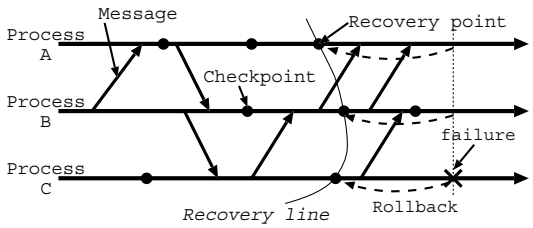

図 1 メッセージパッシングシステムでのモデルとリカバリライン
Fig. 1 An example of the model for message passing systems and the recovery line.

メッセージパッシングシステムのモデルでは，**図 1** に示すようにシステムを構成するプロセスの実行を横矢印で表し，それらの間で取り交わされるメッセージを斜めの矢印で表す．また，あるプロセスが故障により停止した後，リカバリ作業によって各プロセスが復元されるポイントを「リカバリポイント」と呼ぶ．リカバリポイントは各プロセスが独立に取得するチェックポイントから選ばれる．また，各リカバリポイントを結んだ線を「リカバリライン」と呼ぶ．このリカバリラインがメッセージを横切る場合に，特別な場合を除き，回復後のシステム状態の一貫性が維持できないことが知られている．

本研究では，デバイスをシステムを構成する一プロセスと見なす．なぜなら，メッセージパッシングシステムにおける各プロセスと同じく，通常各デバイスは CPU と並列に実行するためである．同様に，デバイスドライバを一プロセスと見なし，また，デバイスドライバ内の割込み処理についても非同期に起動されるため，単独のプロセスと見なす．そして，割込みを含むデバイスとのやりとり（アクセス）をメッセージと見なす．これらのアクセスによって，デバイスドライバとデバイスの実行状態に依存関係が発生し，これはメッセージパッシングシステムでのメッセージの取扱いに類似するためである．

このようにモデル化したとき，システムの一貫性を保つためには，各アクセスを横切らないリカバリラインを発見すればよいこととなる．前節で述べた前提から，まずデバイス状態の復元可能なポイント，つまりデバイスの idle 状態を基準として，適切なリカバリラインを発見する．そして，デバイスドライバでの処理について電源切断時点の状態を利用することができない場合には，コードを追加して適切なリカバリラインを構成するポイントのデバイスドライバの状態を保存するようにする．このようにして，デバイスとデバイスドライバ間での整合性の維持を図る．

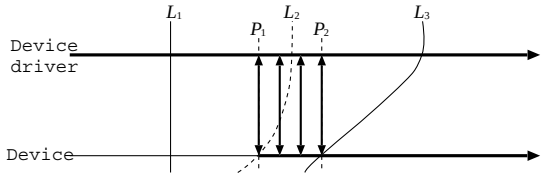

図 2 基本的なデバイスアクセス
Fig. 2 Basic device access.

3. 解　　析

3.1 デバイスへの要求発行

図 2 にモデル化したデバイスハンドリング時の様子を示す．横の矢印は時間の経過と各要素の実行状態を表し，細い部分はその要素が実行されていないことを表す．これは，デバイスについては idle 状態を意味する．縦の矢印はデバイスへのアクセスを表す．read および write アクセスが混在する場合が多いため，上下両方を指す矢印として示してある．

P_1 において，デバイスドライバはデバイスへの要求を発行し始める．デバイスが持つレジスタの設定や読み込みを複数回行った後，P_2 においてその要求の発行が終了する．そして，デバイスは要求に対応する実行を開始する．

P_1 以前にシステムの電源切断が生じた場合，デバイスは idle 状態である．よって，リカバリラインは L_1 のようになり，デバイスドライバについて電源切断時の状態が利用可能である．デバイスについて適切な初期化を行えば，電源切断時の主記憶と CPU 状態を用いて実行を再開すればよい．

P_1 と P_2 の間で電源切断が生じた場合，復元可能なデバイスの状態は P_1 での状態である．そのままデバイスドライバの電源切断時の状態を復元したとすれば，このときのリカバリラインは L_2 で示される点線となる．このとき，P_1 から電源切断までのアクセスに依存するデバイスの状態は失われ，デバイスは要求を正確に実行できなくなる．これは，リカバリラインがデバイスアクセスを横切るとき，リカバリ後のシステム状態の一貫性が維持できない，ということに対応する．

この解決には 2 通りの方法が考えられる．1 つは，P_1 での CPU，主記憶，デバイスの状態を復元し，実行を再び P_1 から開始する方法である．これは，リカバリラインを P_1 での垂直線にすることである．もう 1 つは，実行中に各アクセスのログを保存し，P_1 でのデバイス状態を復元した後に，ログを用いてアクセスを再実行する方法である．それぞれチェックポイン

ティング手法，ロギング手法に相当し，デバイスドライバの製作者が P_1 から P_2 に行われる処理内容に応じて選択すればよい．ただし，チェックポインティング手法を用いる場合には，P_1 から P_2 においてデバイスドライバ外へ影響を及ぼす処理が行われないことが条件となる．この理由は後に述べる．

P_2 以降，デバイスが動作を開始した後に電源切断が生じた場合には，リカバリラインは L_3 に示す形でよい．P_1 から P_2 に行われるデバイスへのアクセスを保存しておき，P_1 におけるデバイスの状態を復元した後それらを再度実行すれば，P_2 でのデバイスの状態を復元できる．P_2 以降，デバイスドライバはデバイスとは無関係に動作するので，CPU や主記憶の状態は電源切断時の状態をそのまま使用してかまわない．

この方法はロギング手法に類似する．チェックポインティング手法を用いて P_1 でのデバイスドライバの状態を復元し，リカバリラインを P_1 での垂直線とすることも考えられる．しかし，以下の理由のため，この場合にはあまり適さない．デバイスが動作している間，デバイスドライバはデバイスドライバ以外の要素と依存関係を生じる処理が行われる可能性が高い．このため，デバイスドライバの状態を P_1 にロールバックした場合，システム全体の一貫性の維持のためにはデバイスドライバと依存関係を生じた各要素の状態も P_1 時点に復元しなければならない（ロールバックプロパゲーション）．最終的に，システム全体の一貫性を保証するためには，P_1 におけるシステム全体の状態を保存し，これを復元して実行を再開させなければならなくなる．よって，P_2 でのデバイスの状態を復元するには，P_1 から P_2 間のアクセスを保存し，リカバリ時に再実行する方法が適切である．以降，P_1 と P_2 間のアクセスを一塊として，「デバイスへの要求」と呼ぶ．

同様の議論は P_1 から P_2 の間で電源切断が生じた場合にも適用される．前述したように，チェックポインティング手法を用いる場合には，P_1 と P_2 間で外部との依存関係が発生しないことが必要である．

3.2 割込み処理

非同期に動作するデバイスの動作の完了は通常割込みを用いて行われる．また，マウスやキーボードなど，デバイス側から発生するイベントも同様に割込みを用いて CPU に通知される．それぞれの場合をモデル化した図を**図 3** および**図 4** に示す．

図 3 は P_1 から P_2 にかけて発行された要求の完了通知（割込み）が P_3 において発生した場合である．割込みに応じて，対応するデバイスドライバ内の割込みサービス処理（ISR: Interrupt Service Routine）がカーネルから呼び出される．ISR の処理では，デバイスの制御やデバイスからのデータの取得などのためのアクセスが発生する（以降，このようなデバイスアクセスを「要求の後処理」と呼ぶ）．

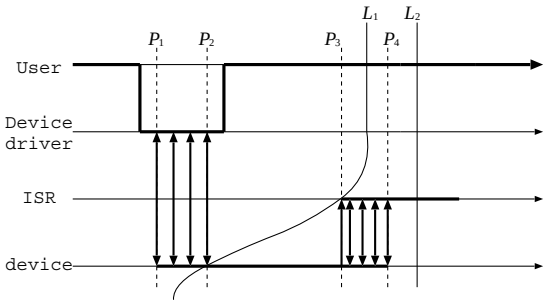

図 3 要求に対応する割込み処理
Fig. 3 An interrupt handling for a request.

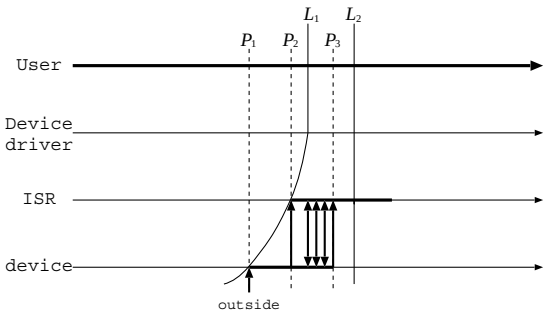

図 4 外部イベントに対応する割込み処理
Fig. 4 An interrupt handling for an external event.

P_3 から P_4 において，デバイスからデータを取得するための処理が行われるとすると，P_4 でこの処理が完了する以前に電源切断が生じた場合，デバイスから取得されるデータは不完全なものとなる．このため，システムの再起動時 P_1 から P_2 で発行される要求を再度実行し，デバイスが同じデータを保持するようにする必要がある．これは，前節で述べたように，保存されたデバイスへの要求の再実行により行うことができる．ここから L_1 に示すリカバリラインが導き出される．そして，ISR では電源切断までに行われた処理を無効化し，状態を割込み発生以前（P_3 以前）のものにする．つまり，ISR では P_3 での状態を保存しておき，リカバリ時に復元するようにする必要がある．

P_4 以降 ISR 内においてデバイスアクセスが終了した後には，各処理は CPU と主記憶の間で完結するため，電源切断時点から再実行させることが可能となる．このためリカバリラインは L_2 のようになる．このときもはやデバイスの状態を P_2 に戻す必要はなくなる

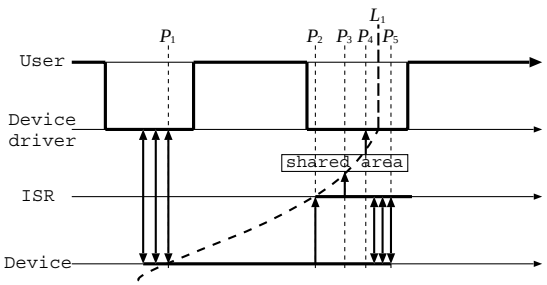

図 5 ロールバックによる問題
Fig. 5 A case of roll-back.

ので，保存された「デバイスへの要求」は P_4 の時点で破棄可能となる．

図 4 は P_1 においてキーボードの入力やパケットの到着などのイベントが発生し，P_2 において CPU に対して割込みが発生した場合である．P_1 と P_2 は等しくてもかまわない．

P_2 と P_3 の間で電源切断が生じた場合，デバイスが保持するデータはまだ取得されていない．また，デバイスの状態は失われ P_1 時点のものしか復元できない．このときのリカバリラインは L_1 となり，前述の議論と同様に ISR の状態は P_2 以前に戻す必要がある．

P_1 以降のデバイスの状態を回復するための情報は，CPU 側からは取得および保存する術がないため，このとき生じた割込みはシステムのリカバリ後完全に失われることとなる．しかし，このようなイベントは，もともと生じるかどうか不確定であるため，無視できる場合が多い．キーボードやマウスの入力がこれにあたる．ネットワークパケットの Ack のように，あらかじめ期待されるものも存在するが，これらについてはより上位層の問題となるため本稿では扱わない．

なお，この場合にも，デバイスアクセスが終了する P_3 以降では，リカバリラインは L_2 のようになり，電源切断時点の主記憶，CPU 状態をそのまま用いて再開することが可能である．

3.3 共有領域のアクセス

前節で，ISR の状態はロールバックされる必要がある場合があることについて述べた．3.1 節でも述べたように，ロールバックされる要素が存在する場合，ロールバックプロパゲーションの可能性について考慮しなければならない．

図 5 は ISR がロールバックされるときにおいて，この状況が発生する場合である．ISR 内でデバイスへのアクセスが終了する P_5 以前に，P_3 において ISR がデバイスドライバとの共有データに対して書き込みを行い，かつ P_4 においてデバイスドライバがそのデータを読み込み作業を行ったとする．このとき，ISR とデバイスドライバの状態に依存関係が発生する．そして，P_4 と P_5 の間で電源切断が生じた場合，ISR の状態は P_2 にロールバックされるため，デバイスドライバの状態も P_4 以前に戻さなければならない．

問題の解決には，ロールバックされる範囲内で ISR とデバイスドライバの間に依存関係が発生しないようにすればよい．このことから，1 つの解決策は，ISR から共有領域への write アクセスは，デバイスへのアクセスが終了した後に行うように ISR をプログラムすることである．それが困難な場合，ISR が共有領域に関連する同期プリミティブのロックをデバイスのアクセスが終了するまで保持し続けるようにする．これによって，デバイスドライバからの共有領域への read アクセスを P_5 以降まで行われないようにすればよい．

3.4 解析のまとめ

本章で述べた解析をまとめると以下のようになる．

(1) デバイスへのアクセスは各アクセスごとにログをとるか，デバイスアクセス前にチェックポインティングを行う．
(2) デバイスへ発行される要求を保存する．
(3) 保存されたデバイスへの要求は，対応する割込みが生じ，その要求の後処理が終了した時点で破棄可能となる．
(4) ISR 内でデバイスへのアクセスが完了する前に電源切断が生じた場合，その状態は割込み発生以前にロールバックさせる．
(5) ISR 内で他の要素との共有領域へ write アクセスを行う場合，デバイスアクセス終了後に行うようにするか，ISR がデバイスアクセス終了までロックを保持し続けるようにする．

4. デバイスドライバへの適用

本章では，**図 6** に示す例を用い，前述の事項のデバイスドライバへの適用方法について述べる．なお，図 6 中，`write`，`read`，`ioctl` の各メソッドの役割は，POSIX で規定される同名の API の機能と同一である．また，`interrupt` メソッドは割込み発生時に実行される ISR である．

4.1 idle 状態の復帰

まず，idle 時のデバイスの状態を復元できるようにしなければならない．これは基本的にはデバイスの初期化作業を行えばよい．しかし，`ioctl` システムコールのように，明示的なデバイスの動作要求とは別にデバイスの状態を変化させる場合がある．たとえば UART デバイスのボーレート設定などがこれに該当

```
 1 void write(void* request){
 2   if(idle == get_device_state()){
 3     DEV_ACCESS_START();
 4     tell_device_request(request);
 5     DEV_REQUEST_SAVE(request_log,request);
 6     DEV_ACCESS_END();
 7   }else{
 8     lock(request_queue);
 9     enqueue(request_queue,request);
10     request_count++;
11     unlock(request_queue);
12   }
13 }
14 void read(void* return_buf){
15   lock(event_buf);
16   while(!event_buf_count){
17     unlock(event_buf);
18     sleep(read_wait);
19     lock(event_buf);
20   }
21   memcpy(return_buf,event_buf);
22   event_buf_count--;
23   unlock(event_buf);
24 }
25 void ioctl(void* new_state){
26   dev_state = *new_state;
27   device_state_change(new_state);
28 }
29 void interrupt(){
30   do{
31     if(request_done & interrupt_reason()){
32       ISR_STATE_SAVE();
33       post_processing_for_request();
34       ISR_STATE_DISCARD();
35       DEV_REQUEST_DISCARD(request_log);
36       lock(request_queue);
37       if(request_count){
38         request = dequeue(request_queue);
39         DEV_ACCESS_START();
40         tell_device_request(request);
41         request_count--;
42         DEV_REQUEST_SAVE(request_log,request);
43         DEV_ACCESS_END();
44       }
45       unlock(request_queue);
46     }
47     if(event_arose & interrupt_reason()){
48       ISR_STATE_SAVE();
49       lock(event_buf);
50       data = get(event_buf);
51       post_processing_for_event(&data);
52       event_buf_count++;
53       ISR_SATE_DISCARD();
54       unlock(event_buf);
55       if(someone_is(read_wait))
56         wakeup(read_wait);
57     }
58   }while(interrupt_reason());
59 }
```

図 6 コード例
Fig. 6 An example code.

する．

これらの設定を復元するため，デバイスの状態を変更する際に，変更後のデバイスの状態をリカバリ時に参照可能な領域へ保存しておく必要がある（図 6 中 26 行）．リカバリ時にはこの情報を参照し，デバイスの状態の初期化と再設定を行うようにする．なお，要求の実行などがデバイスの idle 状態に影響する場合も，同様にリカバリ時に参照可能な領域へその情報を保存し対応する．

4.2 デバイスアクセス

デバイスへの要求を発行するためのアクセスは，チェックポインティング手法もしくはロギング手法によって対応することを述べた．

チェックポインティング手法の場合には，デバイスへのアクセス開始時にデバイスドライバの主記憶状態，CPU 状態の保存を行う．そして，アクセスが終了した時点でそれらの情報を破棄する．リカバリ時には，これらの情報が存在するかどうかを調べる．存在する場合には，保存された主記憶状態の復元を行った後，このときの CPU 状態を用いてシステムの実行を再開すればよい．

このとき，主記憶状態の保存については，デバイスドライバ内すべての領域を保存する必要はない．デバイスアクセス開始時の状態が復元できればよいため，デバイスアクセスが終了するまでに変更され，かつ復元が必要となる主記憶の状態を保存すればよい．

ロギング手法を用いる場合には，各デバイスアクセスごとに，どのようなアクセスが行われるかを示すログをとる．そしてデバイスアクセスが終了した時点でこれらのログを破棄する．リカバリ時には，ログの存在を確認し，存在する場合には，それらのログを実行する．そして，電源切断時の CPU 状態を用いてシステムの実行を再開すればよい．

図 6 中 `DEV_ACCESS_START` および `DEV_ACCESS_END`（3, 6, 39, 43 行）は，それぞれ，この議論においてデバイスへのアクセスの開始とデバイスへのアクセスの終了に対応する．ロギング手法では `DEV_ACCESS_START` において何も行う必要はなく，`tell_device_request` 関数内でログをとるための変更を加える．

4.3 デバイスへの要求の保存

デバイスへの要求は，デバイスへのアクセスが終了した後，かつこれらアクセスを復元するための情報の破棄（`DEV_ACCESS_END`）が行われる前に保存する必要がある．アクセスの情報を破棄した後，要求を保存したとする．もしこれらの処理の間で電源切断が生じたとすると，アクセスを再実行するための情報は失われ，かつデバイスへの要求もまだ保存されていないこととなる．このため，デバイスの実行状態を復元できなくなってしまう．

また，3.2 節で述べたように，この情報が破棄されるのは一般に ISR 内となる．ISR のロールバックと関連するため，破棄作業の注意点については次節に述べる．

3.1 節では，デバイスドライバのコンテキスト内で

要求が発行される場合を述べたが，デバイスへの要求は ISR の中でも発行される場合がある．デバイスが busy 状態のときに新たなデバイスへの要求が発行された場合，デバイスドライバのコンテキストではその要求をキューイングすることがある．キューイングされた要求は，実行中の要求が終了したとき，ISR の中でデバイスに発行される．しかし，ISR 内で実行されたとしても，ロールバックされる範囲でなければ，3.1 節で述べた議論と同一となる．つまり，通常実行時と同じ対応を ISR 内で行えばよい．このことは図 6 中，7〜12 行および 37〜44 行に対応する．

図 6 中，要求のログをとる処理に対応するのが DEV_REQUEST_SAVE（5，42 行）である．また，ログの破棄に対応するのは DEV_REQUEST_DISCARD（35 行）である．

4.4 ISR のロールバック

ISR の中でデバイスアクセスが終了する以前に電源切断が生じた場合，ISR の状態をロールバックしなければならない．このため，ISR が実行を開始するときに，ISR の主記憶状態を保存する必要がある．これは，デバイスへのアクセスが終了した時点で，破棄可能となる．

図 6 中，ISR の状態保存を行う処理に対応するのが ISR_SATE_SAVE（32 行，48 行），保存された情報を破棄する処理が ISR_STATE_DISCARD（34 行，53 行）である．

これらの処理は，デバイスアクセスをチェックポインティング手法で復元させる場合の対応と類似するが，復帰後 ISR を再実行する必要はないため，CPU の状態を保存する必要はない．リカバリ時には，カーネルが割込み受け付けたときに保存した CPU 状態（割込み時に実行されていたコンテキスト）を用いてシステムを再開させればよい．

再実行が必要な「デバイスへの要求」について，ISR がロールバックされるときに失われないようにする必要がある．このため，この情報の破棄は，ISR の情報が破棄された後に行うようにする（35 行）．もしくは，ISR の状態がロールバックされたときに，破棄された要求ログも復元するようにする必要がある．

ISR は複数の要因で起動されることが多い．たとえば，ネットワークに対する送信終了割込みと受信割込みは一般に同一の ISR で処理される．このため，ISR では，割込み要因の特定を行った後，対応する割込みを処理する．また，ISR 実行中に割込みが発生した場合に備えて，ISR の処理は割込み要因がなくなるまでループする場合がある．

```
1  int recovery_call_back(CPU_t **cpu_state){
2    initialize_device(dev_state);
3    recover_memory_area();
4    replay_request();
5    return flag;
6  }
```

図 7 コールバック関数の例
Fig. 7 An example of callback function.

ロールバックされる範囲はそれぞれの割込み要因の特定がなされてからのものとする必要がある．たとえば，要求の終了とイベントによる割込みが同時に発生したとする．このとき ISR が起動され，要求の後処理終了に続いて，イベントの処理が行われることになる．要求の後処理が終わり，50 行目の処理中に電源切断が生じたとすれば，この場合ロールバックされなければならない内容はイベント処理に関するもののみである．要求の後処理までロールバックする必要はない．よって，ISR_STATE_SAVE と ISR_STATE_DISCARD は図 6 のように挿入される．

また，ISR 内で割込み要因を特定するときに，「現在の状態」を確認する必要がある．同上の例において，41 行目で電源切断が生じたとする．リカバリ後 41 行目以降の処理が継続されるが，デバイスの状態は失われているため電源切断前のイベントの処理を続けて行うことはできない．このため，復帰後には 31，47 行目や 58 行目でその時点の割込み状態の観察がなされるようにする必要がある．

49 行目と 54 行目のロックの操作は 3.3 節で述べた議論に対応するものである．50 行目で read メソッドとの共有領域である event_buf をアクセスする．このため 49 行目で取得したロックは 54 行目まで保持される．ただし，ISR とデバイスドライバのコンテキストが同時に実行しないことが分かっている場合には，これらの操作は必要ない．

4.5 リカバリ

コールバック関数の例を**図 7** に示す．この関数はリカバリ時にカーネルから呼び出される．

前節までの議論から，この関数では主に，デバイスの再初期化，主記憶状態の復元，デバイスへの要求の再実行を行うことになる．まず，4.1 節で述べたように，デバイスの再初期化は通常実行中保存された情報をもとに，idle 時のデバイス状態を復元する（2 行）．次に，DEV_ACCESS_START（チェックポインティング手法の場合）や，ISR_STATE_SAVE で保存された主記憶状態が存在する場合にはそれぞれを復元する（3 行）．そして，デバイスアクセスのログ（ロギング手法の場合）や再実行すべきデバイスへの要求が存在する場合にはこれらを実行する（4 行）．

また，ISR_STATE_SAVE で保存された情報の破棄を行う必要がある場合には，ISR の状態を復元した後に行えばよい．DEV_ACCESS_START での主記憶状態やログについては，リカバリ後の通常動作によって破棄されるため，この関数内では何も行う必要はない．

4.2 節や 4.4 節で述べたように，ロールバックが必要となる範囲で電源切断が生じた場合，システムを再開させるための CPU 状態をカーネルが把握する必要がある．このため，このメソッドは図 7 に示すように，引数として CPU 状態の構造体の参照受け渡しや，フラグを返戻値として返すようにする．

たとえば，フラグは ISR 内で電源切断が生じた場合にセットする．この場合，カーネルは割込み発生時に保存された CPU 状態を用いて復帰する．また，CPU 状態には，DEV_ACCESS_START で保存された CPU 状態が存在する場合にはこの情報を返すようにする．カーネルはこれを参照し，システムを復帰する．

なお，これまでの議論は基本的な場合について述べたものである．たとえば，要求の再実行が必要ないようなデバイスでは，リカバリ時に要求を再実行しなくてもよい．また，デバイスへの要求自体を保存しなくてもかまわない．これまでの議論をもとに，対象となるデバイスの特徴に応じてこれまで述べた方法を応用し，実際のデバイスドライバに適用すればよい．

5. 実　　験

提案する手法を linux 2.4.4 カーネルに対し適用した．CPU はモトローラ社の MPC860（40 MHz）を用いた．また FeRAM を用いた不揮発 RAM ボードを作成し，これを主記憶として用いた（**図 8**）．デバイスとしては，オンチップ上に存在する UART（9,600 bps）および Ethernet（10 Mbps）コントローラを用いた．

まず，提案手法に基づいてシステムの実行状態の復元が可能であるかどうかを調べる実験を行った．システム上でデバイスを利用するプロセスを実行している最中に，わざとシステムの電源の切断/復帰を行いシステムの挙動を観察した．

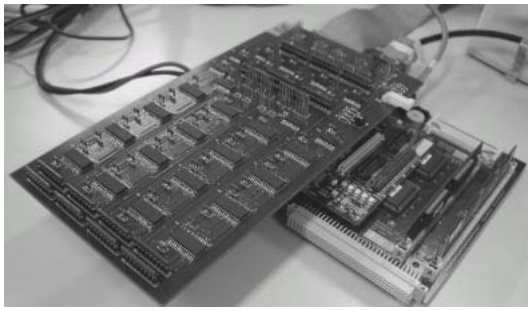

図 8 FeRAM ボードと MPC860FADS
Fig. 8 MPC860FADS with FeRAM board.

次に，提案手法がシステムのパフォーマンスに与える影響について調べた．デバイスを高頻度で扱うプロセスを走らせ，提案手法を適用しない場合，および適用した場合について，プロセスの実行時間を計測した．

以下，それぞれの結果について述べる．

5.1 実行状態復元の確認

提案手法を実装したシステムにおいて，システム上で UART デバイスに対して出力を続けるプロセスを実行し，わざと電源を切断した．そして電源を再度入れシステムを再起動した．

このとき，走行していたプロセスの実行が復元され，再び UART デバイスに出力を開始することを確認した．これは，システムが適切にその実行状態を回復したことによるものである．また，この実験中 UART デバイスから電源切断直前の文字が出力される場合があることを確認した．これは，UART デバイスに対して，要求の再実行が適切に行われた結果といえる．

また，実験ではルートファイルシステムに NFS（バージョン 2）を用いたが，システム再起動後，電源切断以前と同様にファイルに対してアクセス可能であることを確認した．さらに，NFS 上のファイルのコピー作業中に電源を切断した場合にも，復帰後ファイルのコピーが継続することを確認した．

さらにより詳細にデバイスアクセスの復元や ISR の状態復元が適切に動作していることを調べるための実験を行った．システム上の最も高いレベルの割込みにプッシュボタンを取りつけ，この割込みを仮想的な電源切断として，このときの CPU 状態を保存するようにした．そして，システムの実行中にこの割込みを発生させた．割込み発生時のアドレスを確認し，一度システムの電源を切断した後，再度システムの電源を投入してシステムを再起動させた．

UART デバイス，Ethernet デバイス両方について，デバイスアクセス中および ISR 処理中に割込みが発生した場合でも，システムが適切に処理を継続することを確認した．このことから，デバイスへのアクセスの復帰や，ISR の状態復元処理が適切に行われているといえる．

5.2 オーバヘッドの測定

実装したシステム上での提案手法によるオーバヘッドの測定を行った．提案手法を用いない場合（通常実行），提案手法においてデバイスのアクセスに対しチェックポインティング手法を用いた場合，同じくロギング手法を用いた場合それぞれについて 2 つのタス

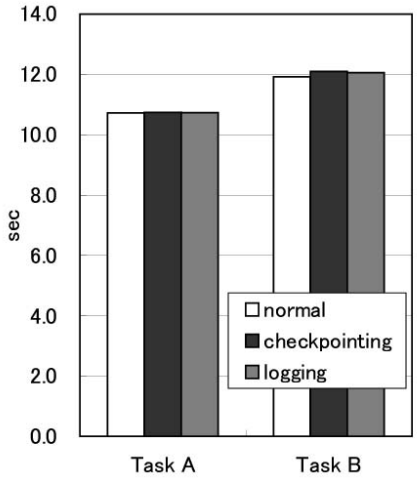

図 9 実行時間
Fig. 9 Execution time.

クを実行した．タスク A は，UART デバイスに対し 1 文字を出力する動作を 10,000 回繰り返すプロセスを実行した．タスク B は NFS 上の 2 Mbyte のファイルを cp コマンドによりコピーした．コピー先も同様に NFS 上とした．それぞれ time コマンドによって実行時間を測定した結果を**図 9** に示す．

通常実行に対し，チェックポインティング手法，ロギング手法いずれを用いた場合にも実行時間の増加がほとんどなく，提案手法がほとんどオーバヘッドをともなわずに実装可能であることを示している．

6. 議　　論

6.1 関 連 研 究

1 章でも述べたように，現在まで多くのシステム耐故障性の向上に関する研究がなされている．しかし，周辺デバイスが保持する状態についてはあまり扱われてこなかった．

Plurix システムでは，デバイス操作に対する redo や undo 操作が困難なことに着目し，“smart buffer” を提案している[2]．smart buffer は，commit までデバイスに対する実際の要求発行を遅延させることで，デバイス操作の原子性を保証するが，電源切断の対応を可能にするものではなく，デバイスが保持する状態の回復については考慮されていない．

Orthogonal Persistence と呼ばれ，実行状態を含めたシステム上で扱われるすべてのデータを永続化し，アプリケーションの実行状態を復元可能としようとする試みもなさている．そして，そのサポート用オペレーティングシステムの開発も行われている[4),6),12)]．しかし，いずれも周辺デバイスが持つ状態には触れられてこなかった．

APM[7] や ACPI[3] などのシステム電源管理手法は，サスペンド/レジュームやハイバネーション機能を提供する．しかし，これらは “smart battery” などによりシステムの電源切断があらかじめ通知され，かつシステムに供給される電力量が十分に存在するときに動作することを前提としている．さらに，システムの電源切断が要求されたとき，各周辺デバイスの状態を調べ，これらの状態が適切に保存/復元可能でない（デバイスが稼動中など）場合には，システムの電源切断自体を遅延させる．このため，これらの仕様に従ったとしても，突然電源が消失する場合の完全なシステムの状態の復帰は保証されない．なお，ACPI の仕様では “Emergency Shutdown” として突然電源が完全に消失する場合の対応についても触れらている．しかし，システムの状態損失や破損を最小限に抑えるために各デバイスの状態を「なるべく」保存すると述べられているのみであり，具体的な対処については述べられていない．

予備電池などの補助的な電源を用いて実際のシステム電源切断を遅延させ，これらシステム電源管理手法を適用することも可能である．しかし，システムに接続されるデバイスの種類や数，その処理にかかる時間は不定であり，結果として補助電源の大型化を招かざるをえない．これはできる限りの小型化や軽量化が必要となるシステムでは問題となる．本稿で提案した手法は，基本的にソフトウェアベースのアプローチであり，主記憶が不揮発であることと，電源切断時に CPU 状態が保存可能であることを除けば，特別なハードウェアを前提としない．このため，システムハードウェアの簡略化や小型化，軽量化が可能となる．

また，本手法は，補助電源および既存のシステム電源管理手法と組み合わせることで，システムの更なる高信頼化に寄与する．電力供給が不安定であることを前提とするシステムにおいて，補助電源に十分な容量の電力が残されていない場合など，たとえば電源切断時に物理的な破損が生じる可能性があるようなデバイスのみに補助電源の電力を供給し，その他のデバイスについては本手法を適用するなどの応用を考えることができる．

6.2 本手法の適用範囲

本稿では，各周辺デバイス自身が持つ状態と，CPU および主記憶の状態について一貫性を保証し回復する手法を実現した．しかし，システムの実行状態復元には以下のような未解決事項も残されている．

6.2.1 より上位層への枠組みの提供

デバイスドライバ外で保持される状態

まず，デバイスに関連してデバイスドライバ外で扱われる実行状態についても考慮しなければならない．たとえば，ネットワークやモデムの接続状態などは，デバイスドライバ外で扱われる．

TCP のコネクションを例にとれば，電源切断中に相手ホスト側でタイムアウトする可能性がある．この場合，アプリケーションに透過に実行状態を回復するには，システム再起動後プロトコルスタック内で接続状態の確認や再接続を試みる必要がある．また，モデムの接続状態についても同様に，下位のレイヤで接続先の電話番号や通信プロトコルを保持しておき，再びリモートのモデムに対して再接続を行うなどの対応が必要となる．

一方で，これらの状態はアプリケーション自身による対応も可能である．TCP の例では，send や recv などのシステムコールの失敗によってアプリケーションプログラム自身で再接続を試みるようにすることが可能である．また，モデムについては，再起動後，アプリケーションに対して電源切断が生じたことを通知する枠組みを用意し，アプリケーション自身が再接続を行うなどの対応が可能である．

時間に依存する状態

また，時間に依存する処理についても問題を生じる．たとえば，音声や画像の入力が該当する．画像の録画最中に電源切断が生じた場合，ビデオフレームに欠落が生じる．このとき，アプリケーションに対して，電源切断が生じたことや，その時間を通知する必要が生じると考えられる．

システム外部への影響

さらに，デバイスが外部の世界に対して影響を及ぼす場合についても考慮する必要がある．このようなデバイスでは，デバイスへの要求が冪等に実行されることが望まれる．たとえば，開ループで制御されるモータなどでは，要求が再実行された場合，モータに対する指令が 2 度発行されることになり，アプリケーションが期待する物理状態と異なる可能性が生じる．

本手法では，デバイスドライバの実装によって要求の再実行を行わない，などの柔軟性を提供している．しかしながら，厳密に行うためには，デバイスのコントロールを閉ループでの制御で行うなどの対応を必要とする．モータの例では，ポテンショメータなどと組み合わせることによって，アプリケーションレベルで閉ループを構築し対応することが考えられる．

なお，デバイス自身が対応する場合も存在する．ディスク装置はデバイスで対応している好例である．ディスクドライブは要求の実行によってヘッドの位置がずれるものの，ブロックアドレスの指定となるためその要求は冪等に繰り返される．

まとめると，本手法外で対応する必要があるシステムの実行状態として，以下のようなものがあげられる．

- デバイスドライバ外で保持され，電源切断とともに失われる可能性のある状態
- 物理的な時間の経過に依存する状態
- 要求の実行がシステム外部に影響を及ぼし，連続するデバイスの実行がその状態に依存する場合

これらについては，別途その対応を検討する必要がある．

これらは，最終的にデバイス動作に対してどのレイヤ（デバイス，デバイスドライバ，プロトコルスタックなどの他カーネルでの処理，アプリケーション）で対応するか，という問題となる．このとき，様々なアプリケーションへの柔軟性の提供という側面から，アプリケーションに対してこれらの問題を扱うための枠組みを提供することが妥当と考えられる．今後より詳細な検討を行い，どこまでアプリケーションに対して透過であるべきか，また，アプリケーションで対応するとすれば，どのような枠組みを提供すべきであるかということの検討を行う必要があると考える．

しかし，一方でこれらの枠組みはデバイス自身が持つ状態の回復のうえに成り立つものである．したがって，これらの枠組みは本手法の拡張として提供されるものであり，本手法の有効性を損なうものではないと考える．また，換言すれば，これらの特別な対応が必要ないシステムでは，本手法のみによってアプリケーションはそのまま実行を継続可能である．

6.2.2 デバイスのアーキテクチャ

本研究では，デバイスに対してデバイスどうしに依存関係が存在しない場合を想定した．そして，USB や PCI など，バスコントローラが持つ状態については未考慮であった．5 章の実験では，オンチップ上に存在し，物理アドレスにマップされたデバイスを用いた．

また，デバイスの状態は，idle 状態とデバイスからの要求を復帰することによって復元可能であることを想定し，デバイスがデバイス自身やデバイスどうしのやりとりなどによって，CPU を介さずその状態を変化させる場合は考慮されていない．

さらに，デバイス自身が大量のバッファを持ち，デバイスドライバの制御なしに次々と要求を処理する場合，その対応は難しくなる．しかし，パフォーマンスは低下させるが，バッファ容量をデバイスドライバか

ら発行される要求 1 つ分に制限することで対応可能である．5 章の実験ではこの方法を用いた．

また，DMA やバスマスタ転送についても未考慮ではあるものの，これらの処理を 1 つの要求であると考えることによって，本稿で述べた方法論を適用することができる．5 章の実験で用いたデバイスは，バスマスタとして動作する．

以上，デバイスのモデルに対する未考慮事項を述べたが，本稿で述べた手法は，デバイスに対するデータの書き込み（write アクセス）とデータの読み込み（read アクセス）という，最も一般化されかつ基本的なデバイスの操作をもとに解析を行ったものである．このため，まだ多くの研究を必要とするものの，本稿で提案する手法は，デバイスに対してその基本的な状態復帰の方法論を提供するものであると考える．

7. ま と め

本稿では，不揮発メモリを主記憶としたシステムにおいて，突発的な電源切断が生じた場合にも，システム全体の状態を一貫性を維持しつつ復元可能とする手法について述べた．本手法は，デバイスドライバとデバイス間の一貫性を保証し，これをシステム上の各デバイスドライバに適用することによってシステム全体の復元状態の一貫性を保証した．デバイスドライバとデバイスの関係において，一貫性維持のための要求事項を，メッセージパッシングシステムのモデルを用いた解析によって明らかにした．また，例を用いてこれらの要求事項の実際のデバイスドライバへの適用方法を示した．そして，linux2.4.4 カーネルおよび UART デバイス，Ethernet デバイスに提案手法を適用したシステムを用いた実験によって，提案手法によるシステムの実行状態の回復が適切に動作することを確認した．また，提案手法はほとんどシステムの性能に影響を与えず実現可能であることを示した．

周辺デバイスの種類やそのアーキテクチャは多岐にわたり，6.2 節で述べた事項が未解決事項として存在する．今後これらについても研究を進めていく予定である，

参 考 文 献

1) Basile, C., Whisnant, K., Kalbarczyk, Z. and Iyer, R.: Loose Synchronization of Multithreaded Replicas, *Proc. 21st IEEE Symposium on Reliable Distributed Systems* (*SRDS'02*), pp.250–255 (2002).
2) Bindhammer, T., Göckelmann, R., Marquardt, O., Schöttner, M., M. Wende and Schulthess, P.: Device Driver Programming in a Transactional DSM Operating System, *7th Asia-Pacific Computer Systems Architecture Conference* (*ACSAC'2002*), pp.65–71 (2002).
3) Compaq Computer Corp., Intel Corp., Microsoft Corp., Pheonix Technologies Ltd. and Toshiba Corp.: Advanced Configuration and Power Interface Specification Rev. 2.0a (2002). http://www.acpi.info
4) Dearle, A. and Hulse, D.: Operating system support for persistent systems: past, present and future, *Software—Practice-and-Experience*, Vol.30, No.4, pp.295–324 (2000).
5) Elnozahy, M., Alvisi, L., Wang, Y.-M. and Johnson, D.B.: A Survery of Rollback-Recovery Protocols in Message-Passing Systems, Technical Report CMU-CS-99-148, Carnegie Mellon University (1999).
6) Elphinstone, K., Russell, S., Heiser, G. and Liedtke, J.: Supporting Persistent Object Systems in a Single Address Space, *Proc. 7th POS* (1996). http://www.cse.unsw.edu.au/~disy/papers/index.html
7) Intel and Microsoft: Advanced Power Management BIOS Interface Specification Rev. 1.2 (1996). http://www.microsoft.com/hwdev/archive/BUSBIOS/apm_12.asp
8) Joe, D., Glasco, D. and Flynn, M.: Fault Tolerance: Methods of Rollback Recovery, Technical Report CSL-TR-97-718, Stanford University (1997).
9) Morin, C. and Puaut, I.: A Survey of Recoverable Distributed Shared Memory Systems, *IEEE Trans. Parallel and Distributed Systems*, Vol.8, No.9, pp.959–969 (1997).
10) Plank, J.S., Beck, M. and Kingsley, G.: Libckpt: Transparent Checkpointing under Unix, *USENIX Winter 1995 Technical Conference*, pp.213–223 (1995).
11) Rubini, A. and Corbet, J.: *Linux Device Drivers*, 2nd edition, O'Reilly & Associates, Inc. (2001).
12) Shapiro, J.S., Smith, J.M. and Farber, D.J.: EROS: A fast capability system, *17th ACM Symposium on Operating System Principles* (*SOSP '99*), pp.170–185 (1999).
13) Slye, J.H. and Elnozahy, E.N.: Support for Software Interrupts in Log-Based Rollback-Rnecovery, *IEEE Trans. Comput.*, Vol.47, No.10, pp.1113–1123 (1998).
14) Wang, Y.-M. and Fuchs, W.K.: Lazy Checkpointing Coordination for Bounding Rollback Propagation, *Symposium on Reliable Distributed Systems*, pp.78–85 (1993).

15) 河合基伸，蓮田宏樹：電源だってユビキタス，日経エレクトロニクス，No.849, pp.103–133, 日経BP 社 (2003).

(平成 15 年 5 月 12 日受付)
(平成 15 年 10 月 18 日採録)

大村　廉（学生会員）

1999 年慶應義塾大学理工学部電気工学科卒業．2001 年同大学大学院理工学研究科前期博士課程修了．現在，同大学院理工学研究科後期博士課程在学中．

山崎　信行（正会員）

1991 年慶應義塾大学理工学部物理学科卒業．1996 年同大学大学院理工学研究科計算機科学専攻博士課程修了．工学博士．同年電子技術総合研究所入所．1998 年 10 月慶應義塾大学理工学部情報工学科助手．2000 年同専任講師．現在，産業技術総合研究所特別研究員を兼務．1997 年〜2000 年科学技術振興事業団さきがけ研究 21 研究員．並列分散処理，リアルタイムシステム，システム LSI の研究に従事．ロボット学会，IEEE 各会員．

安西祐一郎（正会員）

1974 年慶應義塾大学大学院博士課程修了．工学博士．慶應義塾大学工学部助手，北海道大学文学部助教授，慶應義塾大学理工学部学部長を経て，現在慶應義塾塾長．1981 年〜1982 年カーネギーメロン大学客員助教授．計算機科学，認識の情報処理過程の研究に従事．計測自動制御学会技術論文賞，情報処理学会論文賞等受賞．著書に『認識と学習』（岩波書店），『知識と表象』（産業図書），『問題解決の心理学』（中央公論社），『脳科学の現在』（共著，中央公論社），『認知科学のハンドブック』（共著，共立出版）等．訳書に『心の社会』（ミンスキー，産業図書）．