September 10, 2009

RT0878
Computer Science pages

Research Report

Migrating contexts in asymmetric multiprocessors

Takeshi Ogasawara

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Migrating contexts in asymmetric multiprocessors

Takeshi Ogasawara

1. Background

近年，省エネルギーの観点からコンピュータの消費電力が大きな課題となっている．そこで消費電力と性能のバランスをとるために，非対称なマルチプロセッサ（AMP）の構成をとることができる．AMPとは，メモリを共有したマルチプロセッサでありながら，プロセッサ１つ１つの性能が異なるシステムである．性能が異なる理由は，演算パイプラインなどハードウェアアーキテクチャが異なる場合や，同じアーキテクチャで電圧・周波数が異なる場合などがある．こうしたAMP上では，並列性が高いプログラムは比較的電力を消費しないプロセッサで実行し，並列性が低いプログラムは比較的電力を消費するプロセッサで実行することで，消費電力を抑えながら高スループットを得られるAMPの特性を活かすことができる．そうしたハードウェアを活かすため，ソフトウェアは並列に実行されるコードで記述し，それぞれのコードはスレッドで実行される．スレッド間でデータ共有する場合が多い．そうした共有データにスレッドがアクセスする際には排他性制御が必要である．スレッドが共有資源にアクセスする際，排他制御のためにスレッドはロックを取得する．同時に複数のスレッドが同時に共有資源にアクセスしようとすると，一方のスレッドがロックを取得し，残りのスレッドが待たされる．こうした複数スレッド間でロックの競合が起きると，スレッド実行が直列化され，並列性があるプログラムの性能がプロセッサ数だけスケールしない．そのため高性能のためには，スレッド間でロックが競合することをできるだけ減らすことが重要である．排他制御が必要なコード断片はクリティカルセクションと呼ばれている．クリティカルセクションの実行時間が短いほどロックは競合しにくい．AMPは高速なプロセッサを持つので，もしクリティカルセクションを全て高速なプロセッサで実行することに制約が何もなければロックは競合しにくくなる．しかし遅いプロセッサで実行していたスレッドがロック取得毎に高速なプロセッサに遷移すると，大きな遷移コストによるオーバーヘッドが問題となる．さらに，同時に複数の遷移が高速なプロセッサに集中すると，遷移しても直ぐに実行されずに待ち時間が問題になる．クリティカルセクションの実行時間が確実に減らせるときに，小さなオーバーヘッドで遷移することが重要な課題である．

2. クリティカルセクション実行プロセッサの選択による最適化

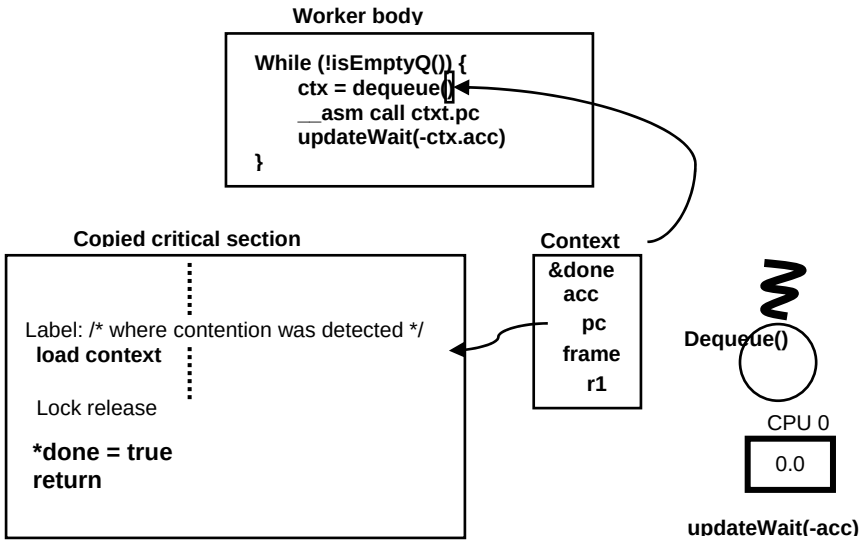

図 1：手法の概要

AMP環境で排他制御を行うスレッド実行系において高速プロセッサでのクリティカルセクションの実行を実行時間が高速化される場合に限定するために，AMPの利用状況を管理し高速プロセッサの仕事量を制御することにより，高速プロセッサを飽和させずにクリティカルセクションを高速化しロック競合によるマルチプロセッサシステムの性能低下を最小限化する．

概要は以下の通り．スレッドはクリティカルセクションの実行時間の統計を取る．高性能プロセッサそれぞれに高優先度のワーカースレッドを待機させる．ロックが競合したら，後からロックを取ろうとしているスレッドがロックのオーナースレッドに通知して，クリティカルセクションの残る実行をワーカースレッドに委譲する．委譲する際に，対応するプロセッサ速度とワーカースレッドの空き状況を見て，クリティカルセクションが最も早く終わるスレッドを選ぶ．

クリティカルセクションは平均実行時間の統計を持つ．そのために，各スレッドはクリティカルセクションの実行時間の統計をサンプリング手法で取る．スレッドはロックを取得した後，サンプリングが有効かをチェックし，そうであれば時刻を取得し，スレッドの一時変数に保存する．これがクリティカルセクション開始時刻に相当する．そしてロックを解放する前に，サンプリングが有効かをチェックし，そうであれば再び時刻を取得し，保存したクリティカルセクション開始時刻と差を取って，現在のクリティカルセクションの実行時間とする．例えばJava Just-in-Timeコンパイラは，Java言語がサポートするロック機構

をコンパイルする際にクリティカルセクションを識別できるので，クリティカルセクションの最初と最後に上記コードを挿入することができる．

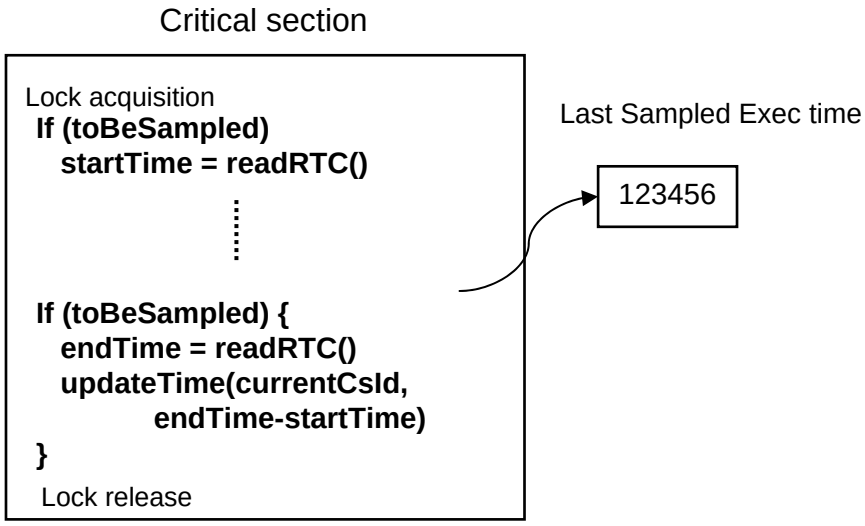

図 2：クリティカルセクション実行時間統計の管理

図 2は上記フローに相当する擬似コードを表す．toBeSampledはサンプリングが有効であることを表す変数である．一定周期でtrueになる．startTimeとendTimeはスレッドの一時変数である．readRTC()は現在時刻を取得する関数である．updateTime()はクリティカルセクションに対応づけられた変数を更新する関数である．

システムの初期化時にクリティカルセクションの実行を担当する高優先度のワーカースレッドを高速コアに待機させる．優先度が高いため，高速コアではワーカースレッドの実行が常に優先される．ワーカースレッドは仕事待ち行列を持ち，ワーカースレッドが実行する仕事は待ち行列に追加され，ワーカースレッドは先頭から順に取得し仕事を実行する．またワーカースレッドの空き状況を管理する．待ち行列に追加される仕事は予測実行時間を持つ．仕事を追加するたびに待ち行列の総待ち時間に予測実行時間が追加され，仕事が完了するたびに逆に総待ち時間から予測実行時間分が削除される．こうした待ち時間の管理により，仕事が溜まっているワーカースレッドへ仕事を依頼して仕事が待たされてしまうことを防ぐ．

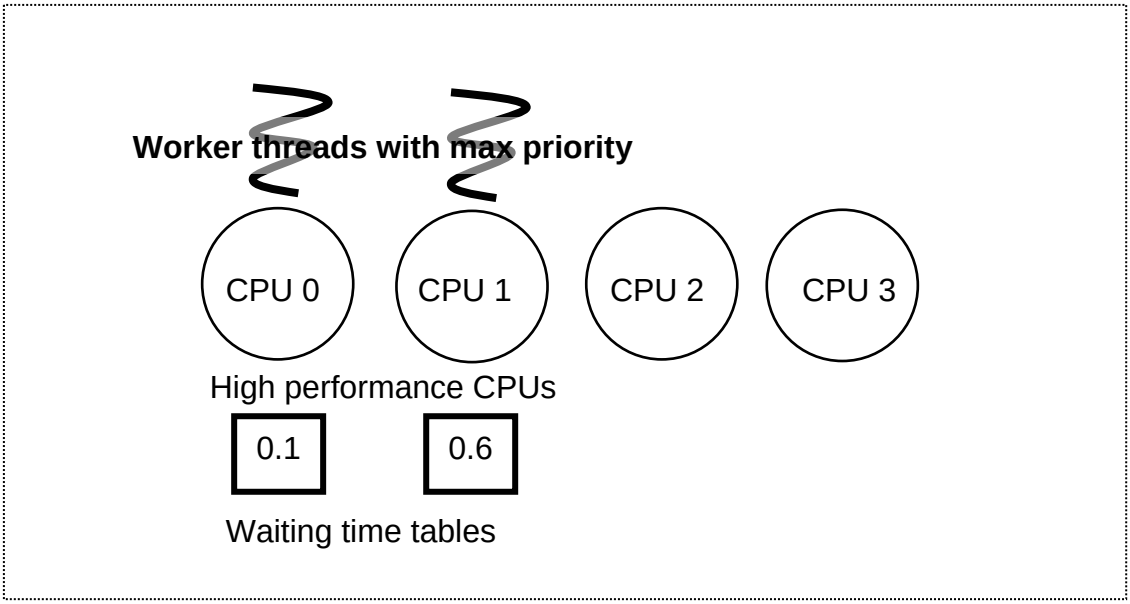

図 3：ワーカースレッドと空き状況

図 3は高速プロセッサが２個（CPU 0とCPU 1），低速プロセッサが２個（CPU 2とCPU 3）を持つシステムの例を示す．CPU0とCPU1にワーカースレッドが配置され，空き状況が管理されている．CPU0のワーカースレッドに対する仕事の待ち時間は0.1，CPU1のワーカースレッドに対する仕事の待ち時間は0.6であり，CPU0がより待ち時間が少ない．

次にロックが競合した場合のスレッドの動作を説明する．スレッドがロックを取得できなかった場合，ロックの所有者スレッドにロックを待つスレッドがいることを伝える．そしてロックを待つスレッドはロックが解除されるまで待つ．ロックの所有者スレッドはロック競合が通知されるとクリティカルセクションの残り実行時間を調べる．

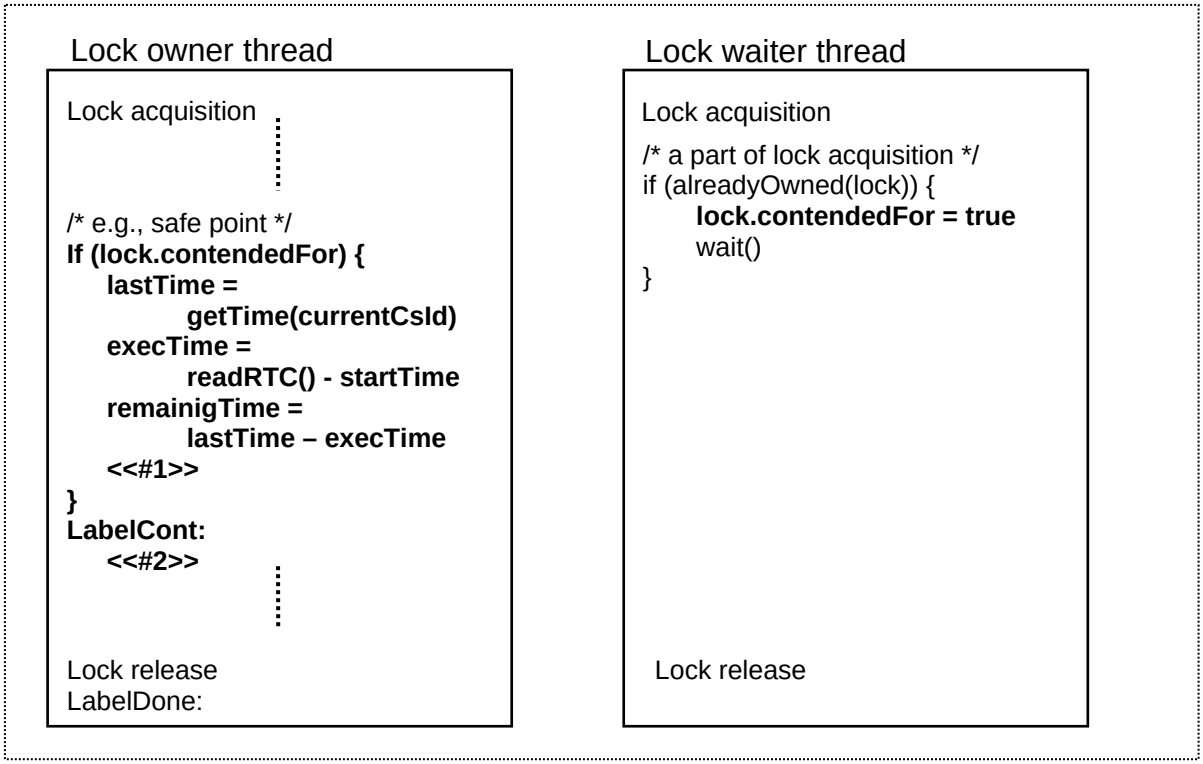

図 4：ロック競合の通知

図 4はロック競合のスレッド間通知の擬似コードを示す．図の右側がクリティカルセクションでロックを待つスレッド，左側がロックの所有者スレッドを表し，同じロックで互いに競合するクリティカルセクション（lock acquisitionからlock releaseまで）を実行する．待つスレッドはロック取得作業の途中でロック（lock）が既に他のスレッドに取得されていることを検出したら（alreadyOwned(lock)がtrue），ロック競合フラグ（contendedFor）をセットし，ロック所有者スレッドにロック解除（lock release）により起こされるまで待機する（wait()）．ロック所有者スレッドはクリティカルセクション実行中に時々（例：関数呼び出しやループのバックエッジ）ロック競合フラグをチェックし，もしセットされていたらクリティカルセクションの残り実行時間を計算する．そのために図 2で示したクリティカルセクションの実行時間統計を取得する（getTime(currentCsId)）．クリティカルセクションの現時点での実行時間（execTime）を現在時刻（readRTC()）と開始時刻（startTime）の差分により計算する．最後に実行時間統計（lastTime）から現時点での実行時間（execTime）の差分により残り時間（remainingTime）を計算する．なお図 4の<<#1>>はさらに図 5に対応し<<#2>>は図 6に対応する．

ロック所有者スレッドは現在のプロセッサで実行を続けた場合のクリティカルセクションの残り時間を計算した後，次にクリティカルセクションを最も早く終えられるワーカースレッドを探す．そのためにワーカースレッド毎に，クリティカルセクションをそのプロセッサで実行した場合の完了時刻を計算する．完了時刻はそのプロセッサのワーカースレッドが仕事を開始するまでの待ち時間と仕事を開始してから完了するまでの実行時間の和である．まずワーカースレッドに対する待ち時間を取得する．待ち時間はワーカースレッド

毎に管理されている．次にそのプロセッサでクリティカルセクションを実行した場合のクリティカルセクションの残り時間を計算する．高速プロセッサは，実行が始まれば，CPU クロックが高いなどでコードを速く実行できるのでクリティカルセクションをより短い時間で完了できる．そのプロセッサに仕事を任せた場合のクリティカルセクションの完了時間，すなわち計算した待ち時間と実行時間を合わせた時間が，現在のプロセッサでクリティカルセクションを実行し続ける場合よりも短くなる場合，クリティカルセクションの残りの実行をそのプロセッサに委譲する．

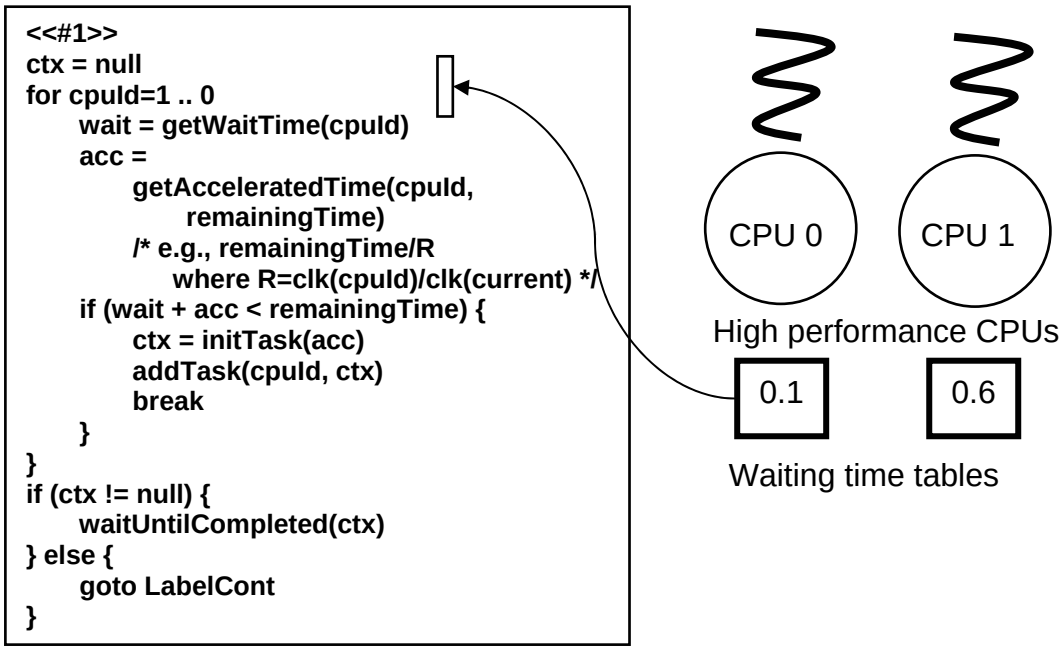

図 5：ワーカースレッドの選択

図 5はロック所有者スレッドがクリティカルセクションの残りを委譲する先のワーカースレッドを選択する上記フローに対応する擬似コードである．ロック所有者スレッドが，図 4に示した残り時間（remainingTime）の計算に引き続いて図 5が実行される．この例では高速プロセッサはCPU0とCPU1の２個であり，各プロセッサ（cpuId）についてループし，以下のことを行う．まずワーカースレッドの待ち時間を取得する（getWaitTime()）．表で管理された待ち時間の値を取得する．次に，そのプロセッサで実行した場合のクリティカルセクションの残り時間を計算する（getAcceleratedTime()）．この値は高速プロセッサと現在のプロセッサの間の速度比を用いて計算できる（速度比は事前に実験で求め，例えば速度２倍であれば残り時間は半分）．次に，待ち時間（wait）と残り時間（acc）の和すなわちそのプロセッサでの残り時間と，現在のプロセッサで残り時間（remainingTime）を比較する．もし前者が小さければそのプロセッサにクリティカルセクションの残りの実行を委譲する．

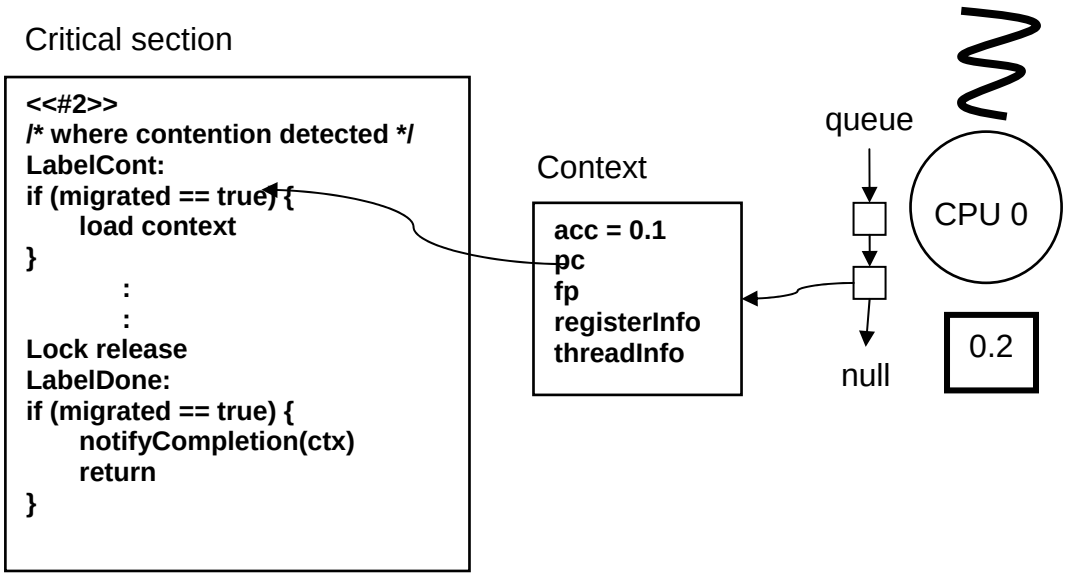

図 6：クリティカルセクションの残りの登録

前者が小さい場合，クリティカルセクションの残りを実行するために必要なコンテキスト（ctx）を作り（initTask()）ワーカースレッドの待ち行列につなげ（addTask()）ループを終了する（break）．図 6は左から順にクリティカルセクション，コンテキスト，待ち行列を表す．クリティカルセクションについては，ワーカースレッドが仕事を実行する手続きのところで後で説明する．コンテキストはクリティカルセクションの残り開始位置（LabelCont）を指すプログラムカウンタ（pc），現在のスタックフレームポインタ（fp），レジスタ情報（クリティカルセクションでuseから始まるレジスタの識別子，フレームに最新の値を保存していないレジスタの識別子とその値；registerInfo），クリティカルセクションを実行していた元のスレッド情報（threadInfo）を保持する，プログラムカウンタはコンパイラのコード生成が持つ情報より生成できる．フレームポインタは動的に現在の値を取得する．レジスタ情報はコンパイラのレジスタ割付手続きが持つ情報（各レジスタがフレーム内の値より新しいかどうか）より生成できる．スレッド情報はスレッドが待機するために使った同期機構の識別子である．コンテキストを保持する領域は動的にメモリに生成する．コンテキストを待ち行列の最後尾につなげる．つなげる際，そのワーカースレッドの待ち時間を更新するため，つなげる仕事の残り時間（acc）を追加する（図 5の0.1にacc=0.1を追加され図 6の0.2になった）．図 5のループ中でワーカースレッドに仕事を渡した場合（ctx != null）ワーカースレッドが仕事を完了するまで待つ（waitUntilCompleted()）．ループ中で仕事を渡さなかった場合，現在のスレッドが引き続きクリティカルセクションの残りを実行する（goto LabelCont）．

次にワーカースレッドが待ち行列から仕事を取得して実行し，仕事を依頼したスレッドに完了を通知する手続きを示す．ワーカースレッドは待ち行列から登録されたコンテキストを列の先頭から順に取得し，コンテキストを復帰してクリティカルセクションの残りを実行する．実行を完了したら，待ち時間を更新し，クリティカルセクションを実行していた

元のスレッドに完了を通知する．

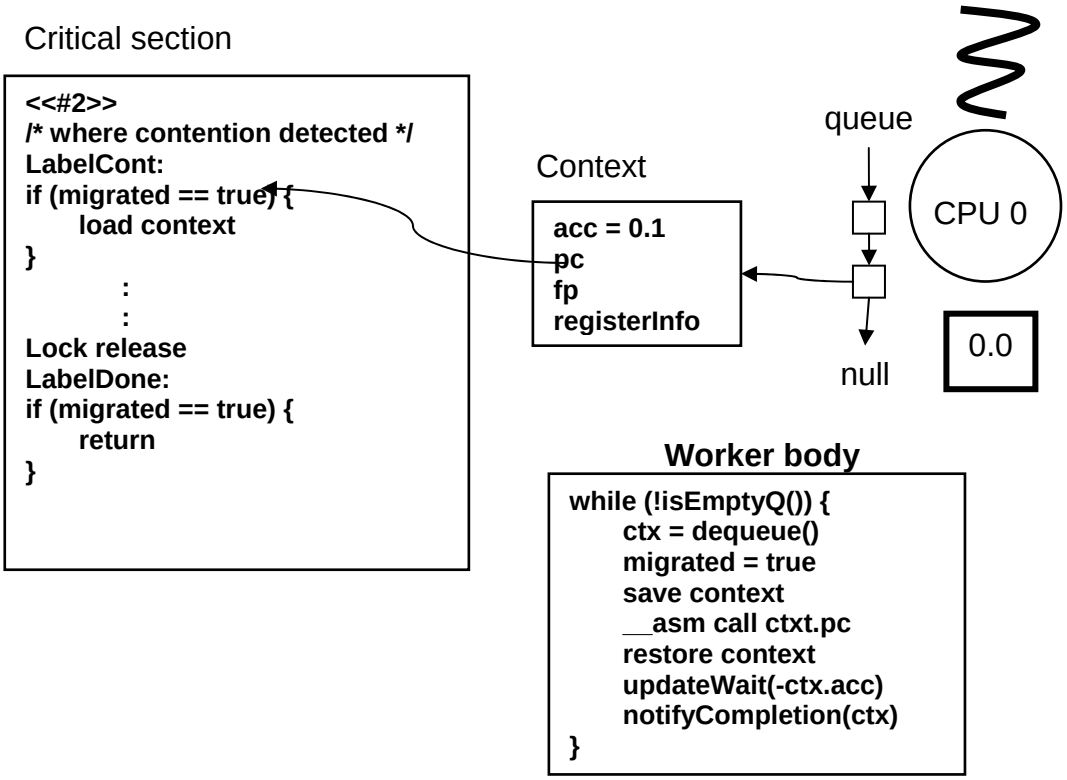

図 7：クリティカルセクションの残りの実行

図 7は図 6に新たにワーカースレッドのコードを示した．ワーカースレッドはループを実行し，待ち行列が空（isEmptyQ()）であれば待機し，空でなければ以下のことを行う．待ち行列の先頭からコンテキスト（ctx）を削除しつつ取得する（dequeue()）．スレッド固有の領域（例：スレッドローカル変数，スタック変数など）に置かれたフラグ（migrated）をセットする．このフラグは元のスレッドがクリティカルセクションを続行する場合はセットされていない．そしてワーカースレッドはコンテキストからプログラムカウンタ（pc）を得て，コードを呼び出す（call ctxt.pc）．この時点からワーカースレッドは図 6の左側で示したクリティカルセクションの残りを実行する．クリティカルセクションのコードはフラグ（migrated）を参照し，セットされているのでコンテキストからフレームポインタを復帰する．またクリティカルセクションでuseから始まるレジスタについて，コンテキストおよびフレームから値を取得する．クリティカルセクションを完了すると（LabelDone:）再びフラグを参照し，セットされているのでコードから戻る（return）．この時点からワーカースレッドはワーカースレッドのループに戻る．クリティカルセクションを完了したので，待ち時間をクリティカルセクションの残り時間（ctx.acc）分だけ減らす（updateWait()は待ち時間0.1からacc=0.1を引いて0.0にする）．そして元のスレッドに完了を通知する（notifyCompletion()）．

クリティカルセクション中でフラグのチェックはオーバーヘッドなので，コンパイラで元

のクリティカルセクションのコピーを作り，コンテキストがコピーを指すことによって，そのコピーではフラグのチェックを行わないようにすることができる．

別のプロセッサでクリティカルセクションを実行すると元のプロセッサに入っているキャッシュの内容が利用できずにキャッシュミスを起こして遅くなる場合がある．そこでワーカースレッドがコンテキストを順に取得する際，キャッシュのレイテンシを隠すために次のコンテキストの情報を元にデータをキャッシュにプリフェッチすることができる．例えばレジスタの復帰の際に使われるフレームの内容やフレーム内のポインタが指すデータをプリフェッチする．