

# 容量を考慮した資源割当て問題の効率的な解法

和田 裕<sup>†</sup>, 程子学<sup>‡</sup>, 小山 明夫<sup>‡</sup>

<sup>†</sup>:会津大学大学院コンピュータ理工学研究科 <sup>‡</sup>:会津大学コンピュータ理工学部

## 概要

分散処理環境においては、メモリーの容量や通信帯域幅など、複数のプロセスが一つの資源を分割して同時に使用する機会が多くある。本稿では、各資源を個々の要求プロセスに量的に割当てる資源割当て問題への解法として、資源使用順序についての「非阻塞型割込み」手法を導入した手法を示す。これにより、要求頻度や資源容量が充分多い場合、資源の使用率が4割以上高まることが、実験結果から見込まれる。

## An Efficient Solution Method for Allocating Resources Using an Unobstructed Squeezing Technique

Yutaka Wada<sup>†</sup>, Zixue Cheng<sup>‡</sup>, Akio Koyama<sup>‡</sup>

<sup>†</sup>: Graduate School of Computer Science and Engineering, The University of Aizu

<sup>‡</sup>: School of Computer Science and Engineering, The University of Aizu

## Abstract

*In some distributed applications, some kinds of resources (such as a memory, a disk, or a communication channel) can be divided into many units and allocated to more than one process in the same time according to their requested amounts for the resource. In this paper, in order to use such kind of resources efficiently, we propose a new method for allocating the resources, by employing a "unobstructed squeezing" technique. A simulation shows that the performance of our method will be about 40% better than allocations without squeezing.*

## 1 はじめに

分散アプリケーションの増加に伴い、分散資源割当て問題の重要性は日に日に高まっている。そのため、資源割当て手法に関する研究は、数多く行なわれてきた。ただ一つの資源を複数のプロセスが互いに排他的に利用するモデルを扱った「相互排他問題 [1]」を始めとして、有名な「哲学者の食事問題 [2]」やいわゆる「資源割当て問題 [3]」など、複数のプロセスによって競合される資源が複数存在することを考慮した問題が多数研究されている。

しかし、現実にはプロセスに割当てられる資源は、必ずしも単一・排他的なものとは限らない。例えば、同等の性能を持つ資源を複数用意しておきプロセスからの要求に応じてそれらのうち任意の幾つかを割当てる手法をとるものや、一つの大きな資源を任意の大きさに切り分けて同時に複数のプロセスに割当てるものなどが考えられる（前者の例としては、同一電話番号で複数回線分受信するダイヤルアップルータなどが考えられ、後者の例としてはディスク容量やメモリ容量、通信帯域幅を量的に配分できる ATM ルータ等が挙げられる）。

前述の多くの先行研究においては、資源はそれぞれが単一の塊として扱われており、一つの資源を複数プロセスに対し分配する様な状況は考慮されていない。そういった問題に対処するために、複数プロセスに分配可能な一つの資源を複数のプロセスで競合する手法として「 $k$ -相互排他問題 [4]」が提起された。この問題では一つの資源を同時に最大  $k$  個のプロセスに対して割当てることのできるものとして、その円滑な割当てについて研究されている。

更に、一つの資源を  $k$  個の片に分割し、各プロセスからの要求数 ( $h$ ) に応じて適宜配分する方法について論じられている「 $h$ -out-of- $k$  相互排他問題 [5]」や、分配可能な複数の資源を複数のプロセスに対し割当てることを目的としたものなど、より拡張された研究も幾つか見受けられる [6][7]。しかし、それらの研究においては、資源は絶対的な優先順位に基づいた配分が行なわれることになっており、他のプロセスの資源獲得に影響を及ぼさないほど少量の資源しか要求しないプロセスであっても、その順位付けに従うことが義務づけられる。これは、資源の利用率（稼働率）や全体の作業効率の低下の要因になりうる。

そこで、我々は優先順位に基づく割当てに、高順位者を阻害しない「割込み」を導入した資源割当て手法を提案する。この手法においては、優先度の高い他のプロセスの資源獲得を阻害しないことが明らかな場合に限り、優先度が低いプロセスであっても必要なだけの資源を先に割当てられることができる。それによって、資源の稼働率を高め、全体の作業効率を向上させることが可能である。

以下では、我々の想定するモデルと問題の定義を第2節で行ない、第3節でその解法を示す。第4節では、解法の正当性を検証するとともに、性能の評価を行なっている。

## 2 諸定義

### 2.1 モデル定義

資源割当てモデルは、主に資源要求を行なうプロセス群と、それらによって要求される資源の群から構成される

(図:1). 現実のシステムでは、資源を管理するプロセスや資源要求を行わない多数のプロセス、メッセージ送受信に携わるプロセス等が存在するが、直接資源割当て作業に関与しないものはモデルにおいては省かれる。

各資源は、それぞれいくつかの部分に分割することができ、分割された一部分ごとにプロセスに割当てることができる。資源を分割するにあたって、それ以上細かく分割することができない最小の分割片をユニットと呼ぶ。ただし、ある一つの資源のすべてのユニットは、大きさや性能等の面で均一であるものとする。各プロセスは、要求可能な各資源ごとに必要とするユニット数を指定して要求を行ない、各資源はユニット単位でプロセスに割当てられる。

各資源の持つユニットの総数を、その資源の容量と呼ぶ。各資源について、そのプロセスにも割当てられていないユニットの数を、その資源の空き容量と呼ぶ。プロセスに割当てることのできるユニット数は、合計でその資源の容量以下に限られる。言い換えれば、空き容量よりも多くの資源を要求するプロセスには資源を割当てては出来ない。

各プロセスは、ユーザや上位アプリケーション等からの呼び出しに応じて、(あらかじめ定められていた)いくつかの資源を必要な量だけ要求する。各プロセスは、自分の要求した全ての資源に関して要求量分を割当てられた時点で、割当てられた容量分の資源ユニットをロックし、ユーザやアプリケーション等に獲得した資源を使用させる。資源の使用が終了した際に、ロックした資源を全て解放する。

本稿では、モデルやアルゴリズム等を表現するために、以下の記号を用いる。

- $p_i$ : 識別子  $i$  をもつプロセス。
- $r_j$ : 識別子  $j$  をもつ資源。
- $P$ : プロセスの集合。  $P = \{p_1, p_2, \dots, p_i, \dots, p_n\}$
- $R$ : 資源の集合。  $R = \{r_1, r_2, \dots, r_j, \dots, r_n\}$
- $R_i$ : プロセス  $p_i$  が要求する資源の集合。  $R_i \subseteq R$ 。
- $P_j$ : 資源  $r_j$  を要求するプロセスの集合。  $P_j \subseteq P$ 。
- $C_j$ : 資源  $r_j$  の容量 (総ユニット数)。  $C_j \geq 1$ 。
- $V_j$ : ある瞬間の  $r_j$  の空き容量。
- $D_i^j$ : プロセス  $p_i$  が要求する  $r_j$  の量 (ユニット数)。  
 $0 \leq D_i^j \leq C_j$ 。
- $ts_i$ : プロセス  $p_i$  が資源要求を行なった時刻を示すタイムスタンプ。

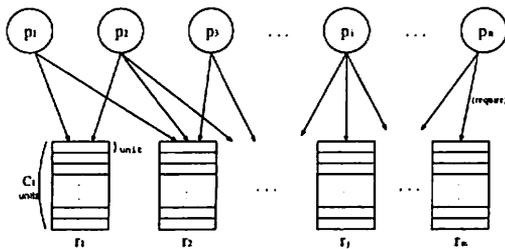


図 1: 資源割当てモデルの例

$P, R$  及び各  $P_j, R_i, C_j$  は静的なものとする。つまり、モデル中のプロセスや資源の数、およびそれらの要求・被要求の関係は変化しないものとし、また資源の容量は常に一定であるものとする。

各プロセス  $p_i$  は、それぞれ 1 つ以上の資源を要求する可能性をもつものとし ( $R_i \neq \emptyset$ )、資源要求を行なう際には  $R_i$  に含まれる資源をそれぞれ 0 個以上の任意の数のユニットを要求できるものとする。ただし、資源  $r_j$  を要求するプロセス  $p_i$  は、 $r_j$  の容量よりも多くの資源を要求する

ことはないものとし ( $0 \leq D_i^j \leq C_j$ )、一旦資源要求を行なった場合資源の獲得が完了するまで  $D_i^j$  は変化しないものとする。

また、各プロセス間には相互にメッセージを送受信できる通信経路が存在するものとする。各通信経路においてあるプロセス  $p_{i1}$  から他のあるプロセス  $p_{i2}$  に対してメッセージが送られる場合、その到達時間は不定だが、有限時間以内に確実に到達するものとする。また、同一プロセス間で送られるメッセージは、常に FIFO 的順序で受信されるものとする。

各資源には、それぞれ要求状況等を管理する資源マネージャプロセスが存在するものとする。各マネージャプロセスは、その資源を要求しているプロセスとその要求時のタイムスタンプおよび要求量を常に正確に把握しており、それらのデータに基づき資源をどのプロセスに割当ててくるかを決定する。

各プロセスと、そのプロセスが要求する各資源のマネージャとの間にも、双方向の FIFO 的な通信経路が存在するものとする。

## 2.2 問題の定義

旧来の資源割当て問題 [2][3] 等と同様に、以下の用語を定義する。

**デッドロック:** いずれのプロセスも、必要とする全ての資源を獲得することができない状態に陥ること。

**飢餓状態:** いずれかのプロセスが、決してその必要とする全ての資源を獲得することができない状態に陥ること。

本稿で解決を目指す資源割当て問題は、前述のモデルにおいて、デッドロックや飢餓状態を生じさせずに資源割当てを分散的に行なう方法を見出すものである。

資源配分の際には各資源を最大限効率的に割当ててことでシステムに含まれるプロセス群全体の作業効率を高めることを目指す。

## 3 割込み使用可能な資源割当て手法

### 3.1 資源競合解決の基本的な考え方

ある資源  $r_j$  に関し、複数のプロセス  $p_i$  が使用を望んでいるとする。このとき、それらのプロセスの要求数の総和  $\sum D_i^j$  が  $r_j$  の容量  $C_j$  以下であるならば、それら全ての  $p_i$  に対し、 $r_j$  の必要なだけの分量が割当てられる。要求される総量が容量  $C_j$  を上回った場合、つまり資源の容量が不足している場合、何らかの基準に基づいて、いくつかのプロセスの資源使用を制限しなくてはならない。具体的には、各資源  $r_j$  ごとにその資源を要求している全てのプロセス  $p_i \in P_j$  に優先度という値を与え、優先順位の高いプロセスから順に要求量分の資源を割当て、という処理を行なう。このとき、あるプロセス  $p_i$  に割当てするための十分な空き容量が存在しない場合、 $p_i$  および  $p_i$  よりも優先度の低いプロセスは、既に資源を割当てられているいずれかのプロセスが資源を解放することにより、十分な空き容量が確保できるまで待ち続けることになる。

また、単純な順位付けのみに従うのではなく、場合によって柔軟な資源割当てを行なうことにより、資源使用の効率がより高いものとなることが期待される。

例えば、「先着順に席に着かせる」という方針を持つ座席数 (資源容量) 10 の飲食店があるとすると、あるとき、6 人組のグループが順に 3 組 ( $g_1, g_2, g_3$  とする) やって来

て、その直後に2人組のグループが4組 ( $g_4, g_5, g_6, g_7$  とする) やって来、各グループがほぼ一律約30分滞在する場合について考える。もしも「先着順」という方針にのみ基づいた席の割当てを行なった場合、 $g_4$  と  $g_5$  が席に付くのは  $g_2$  が店を出た後となるので約60分待たされることになり、 $g_6$  と  $g_7$  は更に約30分待たされることになる(図2)。同じ状況で、 $g_1$  が食事中のうちに  $g_4$  および  $g_5$  を、 $g_2$  の食事中に  $g_6$  と  $g_7$  を席に案内したとしても、 $g_2$  や  $g_3$  が待たされる時間の長さは  $g_4$  以降のグループを席に着かせなかった場合と変わらず、また全員が食事を終えるまでに要する時間も30分短縮できる(図3)。

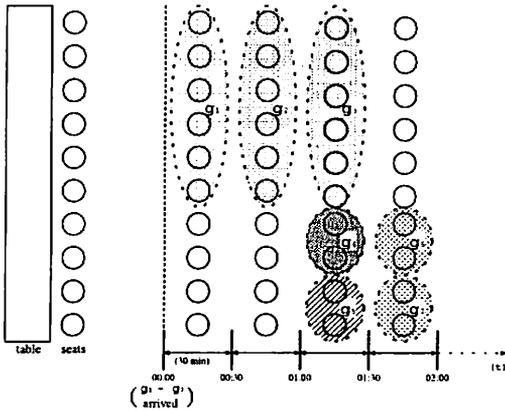


図2: 割り込みを認めない場合の例

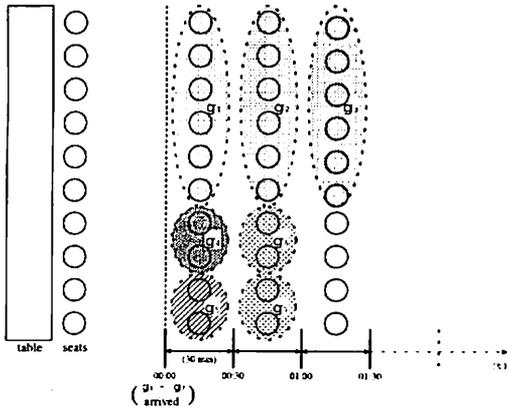


図3: 割り込みを認めた場合の例

### 3.1.1 優先度基準の重要性

資源割当ての優先度を定めるにあたって、不適切な基準を設けてしまうと、デッドロックや飢餓状態等の障害を招いてしまう危険性がある。

例えば、ある二つのプロセス  $p_{i1}, p_{i2}$  がそれぞれ資源  $r_{j1}, r_{j2}$  を要求している場合を考える。ある(不適切な)基準に基づいて各資源を割当てる優先度を定めた結果、 $r_{j1}$  の割当てを管理する機構の中では  $p_{i1}$  より  $p_{i2}$  の方が優先度順位が高く、 $r_{j2}$  に関しては  $p_{i2}$  よりも  $p_{i1}$  の方が高順位であったならば、 $p_{i1}$  が  $r_{j2}$  を獲得できるが  $r_{j1}$  を獲得できず、 $p_{i2}$  は  $r_{j1}$  を獲得できるが  $r_{j2}$  は獲得できないという事態を招く恐れがある。この状態は、典型的なデッドロックである。

また、ある優先度の低いプロセス  $p_i$  が、要求する資源  $r_j$  の空き容量が充分できるまで待っているという状況において、 $p_i$  よりも優先順位が高いプロセスが ( $p_i$  が資源を獲得できない間に) 次々と現れた場合、 $p_i$  は永久に資源  $r_j$  を獲得できないことになる。これは  $p_i$  が飢餓状態に陥っている状況を示す。

以上の様な理由から、資源を割当てる際には、(a) ある瞬間において、全てのプロセスに関し全順序的に順位が定まり、かつ (b) ある瞬間で順位の低かったプロセスも有限時間以内に(資源を獲得するために) 充分なだけ順位が高まる、という特性を持つ優先度基準値を設ける必要があると言える。

そういった優先度の定め方は多数考案されており [3][9]、それぞれ利点や欠点(実装の難しさ、煩雑さなど)を持っている。本稿では、(a) (b) の両特性を併せ持つ優先度基準としてタイムスタンプを利用する。ただし、これはタイムスタンプを用いる手法と他の優先度基準の優劣を比較して選択したものではない。3.2節で示すアルゴリズムは、より好ましい優先度基準が見出された場合はタイムスタンプの比較部分を他の優先度基準値(例えばトークンやカウンタ等を利用したもの)に置き換えても問題なく動作するのである。

タイムスタンプを用いた解法では、ある資源  $r_j$  に関し複数のプロセス  $p_1, p_2, p_3$  が競合を起こした場合、 $r_j$  はタイムスタンプの最も小さい(資源要求の開始時刻の最も早い)プロセスから順に割当てられる。例えば、 $ts_1 < ts_2 < ts_3$  であったとするならば、まず  $r_j$  のうち  $D_1^j$  個のユニットが  $p_1$  に割当てられる。

$V_j \geq D_2^j$  であったならば、さらに  $p_2$  にも  $D_2^j$  ユニットが割当てられる。 $V_j < D_2^j$  であったならば、 $r_j$  には  $p_2$  の要求に応じられるだけのユニットが残されていないので、 $p_2$  は充分な空きができるまで ( $p_1$  が資源の使用を終え、獲得していた  $D_1^j$  ユニット分の資源を解放するまで) 待たねばならない。

$p_2$  が  $p_1$  の資源解放を待っている間は、たとえ  $r_j$  の空き容量が  $p_3$  の要求を満たしていた ( $V_j \geq D_3^j$ ) としても、無条件に  $p_3$  に資源を割当てることは出来ない。もしもその様な状況で、 $p_3$  に資源を割当てることを許してしまうと、要求量の多いプロセス ( $p_2$ ) が飢餓状態を生じてしまう危険性があるからである。

$p_1$  が資源の使用を一旦終わると、獲得していた全ての資源を解放するのだが、その際にタイムスタンプ  $ts_1$  は破棄される。 $p_1$  が再び資源要求を開始する際に  $ts_1$  が改めて設定されるので、もしその時点で  $p_3$  が資源を獲得できていなかった場合  $ts_1 > ts_3$  となるため、 $p_3$  には  $p_1$  の二度目の資源要求よりは優先的に  $r_j$  を使用できる。

### 3.1.2 資源の割り込み使用

タイムスタンプによる資源使用の優先順位付けは、飢餓状態の回避に役立つと言える。しかし、単純にタイムスタンプによる優先順位にのみ固執して資源を割当てていては、作業効率の低下を招く危険性がある。

例えば、ある資源  $r_j$  とそれを要求するプロセス  $p_1, p_2, p_3$  について、 $C_j = 4, D_1^j = 2, D_2^j = 3, D_3^j = 1, ts_1 < ts_2 < ts_3$  という状況を考える。 $p_1$  に資源を  $2(=D_1^j)$  ユニット割当てた時点で、 $V_j = 2 < D_2^j$  より、 $p_2$  は  $p_1$  の資源解放を待たなくては資源を獲得することはできない。この間、 $p_3$  が空きスロット2つのうち1つを割当ててしまったとしても、 $p_1$  が資源を解放し次第  $p_2$  は充分な量(3ユニット)の資源を獲得できるため、 $p_2$  の待ち時間は変わらない。

この様な場合、優先度の高いプロセス  $p_2$  よりも先に優先度の低いプロセス  $p_3$  に資源  $r_j$  を割当ててことを「( $p_3$  による、 $p_2$  に対する  $r_j$  の) 割り込み使用」と呼ぶことにする。

資源競争が発生している際、安易に割り込み使用を許すべきではないことは前述の通りである。そこで、我々は、

- ある資源  $r_j$  を要求しているプロセスが存在する場合、できるだけ多くのユニットをプロセスに割当て、 $r_j$  の空き容量は極力少なくする。
- (割り込みを行なうことによる) 新たな飢餓状態やデッドロックの発生は回避する。

という2つの方針に基づく割り込み方法を提案する。

この割り込み方法を用いた資源割当てにおいては、 $p_i$  が、要求する全ての資源  $r_j \in R_i$  に関して  $D_i^j$  個のユニットを (a) 優先度に基づいて無難に割当てられるか、あるいは (b) 割り込みによって獲得できる見通しが立つかのいずれかの条件を満たした時点で、 $p_i$  は必要な資源全てを割当てられ、それら全ての資源を用いて  $p_i$  自身の作業処理を行なうことができるものとする。

## 3.2 資源割当てアルゴリズム

### 3.2.1 使用されるメッセージ

資源競争の解決のため、各プロセスは以下の数種類のメッセージを互いに送りあい、資源要求状況などを把握する。

以下は、資源  $r_j$  の容量  $D_i^j$  を要求しているあるプロセス  $p_i$  と、資源  $r_j$  を管理するマネージャプロセスとの間で送られるメッセージを示している。本来、資源自体にメッセージ送受信等の機能は備えられていないが、本稿では以後プロセス  $p_i$  から資源  $r_j$  のマネージャプロセスに対してメッセージを送ることを「 $p_i$  が  $r_j$  にメッセージを送る」の様に書く。資源マネージャから送られるメッセージについても同様である。

**REQUEST( $p_i, r_j, D_i^j, ts_i$ ):**  $p_i$  が、資源  $r_j$  に対し、資源要求の意思を伝えるために送るメッセージ。資源競争の解決のため、 $D_i^j, ts_i$  もあわせて送られる。

**RELEASE( $p_i, r_j, D_i^j$ ):** 資源  $r_j$  を使用していたプロセス  $p_i$  が、ロックしていた資源を解放することを伝えるために送られる。

**ALLOCATE( $r_j, p_i, D_i^j$ ):** 資源  $r_j$  が、 $p_i$  に対し要求された分量の資源を割当てて伝えるために送られるメッセージ。

### 3.2.2 アルゴリズム (概要)

[資源要求プロセスの動作] 資源要求を行なう各プロセス  $p_i$  は、3つの状態を遷移する。各状態におけるプロセスの動作は、以下のようになる。

- **Idle 状態:**  
ユーザやアプリケーションから資源要求を命じられる前の状態。メッセージの送信等は行なわない。ユーザ等から資源の要求を命じられると、 $p_i$  はその時刻をタイムスタンプ  $ts_i$  として記憶し、Wait 状態に遷移する。
- **Wait 状態:**  
ユーザ等の命令により、資源要求を行なっている状態。そのプロセスが要求する資源全て ( $\forall r_j \in R_i$ )

に **REQUEST** を送信し、資源使用の許可である **ALLOCATE** が送られてくるのを待ち続ける。要求する全ての資源からの **ALLOCATE** を受取った時点で、ユーザ等に資源の獲得を完了したことを報告し、Working 状態に遷移する。

- **Working 状態:**  
要求する全ての資源を獲得し、ユーザ等がそれらの資源を使用している状態。資源の使用を終了した旨をユーザ等から伝えられた時点で、獲得していた各資源 ( $\forall r_j \in R_i$ ) に **RELEASE** を送信し、資源を解放する。その後、 $p_i$  は Idle 状態に戻る。

[資源マネージャの動作] 資源マネージャは、各要求プロセスからのタイムスタンプや要求量に基づき、各プロセスに対する資源割当てを行なう。その際、3.1.2 節で述べた割り込み手法を用いることで資源使用の効率を高める。

以下は、 $r_j$  のマネージャプロセスの動作である。ただし、便宜的に、

- $P_{wait}$ : ある時点で、資源  $r_j$  を要求しており、未だ割当てられていないプロセスの集合。
- $P_{acq}$ : ある時点で、資源  $r_j$  を要求し必要なだけのユニットを割当てられているプロセスの集合。
- $P'_{wait}$ : 割り込み使用に関する計算過程において、資源を要求中であることが想定されるプロセスの集合。
- $P'_{acq}$ : 割り込み使用に関する計算過程において、資源を割当てられていることが想定されるプロセスの集合。
- $V_j'$ : 割り込み使用に関する計算過程において、資源  $r_j$  の想定される空き容量。
- $flag_i$ : 割り込み使用に関する計算において、プロセス  $p_i$  が割り込みを行なうことができるかどうか判断するためのフラグ。

という変数を用いる。

```
When  $r_j$  receives REQUEST( $p_i, r_j, D_i^j, ts_i$ ):
1 begin
2   Put  $p_i$  to  $P_{wait}$ ;
3   Record  $D_i^j$  and  $ts_i$ ;
4 end.
```

```
When  $r_j$  receives RELEASE( $p_i, r_j, D_i^j$ ):
5 begin
6   Remove  $p_i$  from  $P_{acquired}$ ;
7    $V_j := V_j + D_i^j$ ;
8    $D_i^j := 0$ ;
9    $ts_i := \infty$ ;
10 end.
```

```
When  $P_{wait} \neq \emptyset$ :
11 begin
12   ( $P_{wait}$  に含まれるプロセスを、
    その要求タイムスタンプ順に並べ換える)
13   if  $D_a^j \leq V_j$ , s.t.  $ts_a = \min_{p_h \in P_{wait}} ts_h$  then
14     begin
15       send ALLOCATE( $r_j, p_a, D_a^j$ );
16       Remove  $p_a$  from  $P_{wait}$ ;
17       Put  $p_a$  into  $P_{acquired}$ ;
18        $V_j := V_j - D_a^j$ ;
19     end
20   else
21     begin
22       forall  $p_i \in P_{wait}$ 
23         begin
```

```

24   P'wait := Pwait ;
25   P'acquired := P'acquired ;
26   Vj' := Vj ;
27   flagi := true ;
28   forall pi ∈ P'wait s.t. ts_i < ts_i
29     begin
30       if Vj' < Dj_i then
31         flagi := false;
32       if flagi = true then
33         begin
34           Find a Π ⊂ P'acquired'
           such that, δ_k is a
           minimum in all
           δ_k ≥ Dj_i - Vj', where
           δ_k = ∑_{p_k ∈ Π} Dj_k ;
35           Remove all p_k ∈ Π from
           P'acquired ;
36           Remove pi from P'wait ;
37           Put pi into P'acquired ;
38           Vj' := Vj' + δ_k - Dj_i
39         end
40       end ;
41   if flagi = true
42     begin
43       send ALLOCATE(r_j, pi, Dj_i) ;
44       Vj := Vj - Dj_i ;
45       Remove pi from Pwait ;
46       Put pi into P'acquired
47     end
48   end
49 end .
50 end .

```

## 4 実験

我々の、資源使用順序への割込みアルゴリズムの動作の確認と、割込み使用による作業効率の向上の度合いを調べるため、シミュレーションプログラムを作り、実験を行った。

### 4.1 実験モデル

割込み行為自体の効用を調べることが目的であるため、資源競合の構図を簡素化し、割込み使用を行なった場合と行なわなかった場合の性能の比較に重きを置くこととする。

容量  $C_j$  の仮想的な資源  $r_j$  を一つ設け、複数のプロセスがそれぞれ任意の要求量を以て  $r_j$  の獲得を目指すものとする。

プロセスの個数は規定せず、合計で(延べ)1000回の資源要求が発生し、それらの要求を行なった全てのプロセスが資源を使用・解放するまでに要した時間を計測し、またその間の、割込み発生回数、各プロセスの平均待ち時間、資源の使用率(資源総量に対する使用中のユニット量の比率)等を計測または計算した。

各プロセス  $p_i$  の要求量  $D_i^j$  は1以上  $h$  以下の乱数とする。資源を獲得したプロセスは、その後数ターン経過した時点で資源を解放する。解放までに掛ける時間(資源を使用して、そのプロセス自身の作業を行なっている時間を表す)を各プロセスごとに1~10の乱数で定める。この時間の単位を便宜上ターンと称する。1ターンは、各プロセス間での通信時間や内部計算の時間に比べ充分に長いものとする。

以上の条件の下で、資源の割当て制御機構をシミュレートするプログラムを作り、次のような結果を得た。

## 4.2 実験結果

資源容量  $C_j$  が10, 100, 1000の3通りの場合について、それぞれプロセスの要求量の上限  $h$  を数通りに変化させ、データを収集した。

同等の条件下で、割込みを許可した場合とそうでない場合のデータを求め、その比較を行なっている。表1において示している性能向上の比率は、

$$100.00(\%) - \frac{\text{割込みを許可した場合の値}}{\text{割込みを行なわなかった場合の値}}$$

の値である。

表中の  $\tau$  は最初のプロセスが資源を獲得した瞬間から1000個全てのプロセスが資源を解放するまでの所要時間(ターン数)、 $\bar{w}$  は各プロセスが資源を獲得するまでに要した待ち時間の平均値、 $\bar{V}_j$  はターンごとの空き容量の平均値、 $N_q$  は割込みの発生件数を示している。それぞれの条件下で5回ずつの計測を行なっており、表中の数値はその平均である。

### 4.3 実験結果に関する考察

図4のグラフを見ると、資源の容量  $C_j$  が小さい( $C_j = 10$ )場合、割込みはほとんど発生しないことがわかる。そのため、容量の小さい資源については所要ターン数( $\tau$ )や平均空き容量( $\bar{V}_j$ )等に関しても、割込みメカニズムを導入したことによる性能の向上はほとんど見られない。

しかし、ある程度以上の資源容量がある場合( $C_j = 100, C_j = 1000$ )は頻繁な割込みの発生が見られる。特に、資源の要求量の上限( $h$ )が大きい(我々の実験例では  $C_j$  の1/5以上)場合、要求を行なった1,000個のプロセスの半数以上、最大では全体の3/4以上ものプロセスが割込みを行なって資源を獲得している。

また、グラフからもわかる様に、 $C_j = 100$ の場合と  $C_j = 1000$ の場合とで、割込み発生数にあまり差は見られない。このことから、ある程度以上の資源容量と最大要求量が見込まれる場合、割込みメカニズムは十分作用することが期待できる。

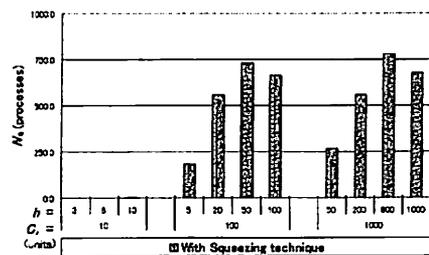


図4: 割込みの発生数

1000個のプロセスが資源要求を行ない、解放するまでの時間の長さ( $\tau$ )を見ると、最大資源要求量( $h$ )の大きいものほど必然的に多くのターン数を要している。我々の例では、割込みを利用することで総ターン数を最大で約2割減少させることができた。言い換えると、同じターン数で約1.2倍のプロセスが資源を獲得できると考えることができる(図5)。

また、各プロセスが資源を獲得するまでに待たされた時間( $\bar{w}$ )についても、やはり資源要求量の多いものほど多くのターン数を要している。いくつかの例では、割込みを利用した結果、待ち時間を平均約30%短縮することができている(図6)。

$r_j$		With squeezing technique				Without squeezing technique			Ratio (%)		
$C_j$	$h$	$\tau$	$\bar{w}$	$\bar{V}_j$	$N_q$	$\tau$	$\bar{w}$	$\bar{V}_j$	$\tau$	$\bar{w}$	$\bar{V}_j$
10	3	1408.0	700.52	0.73	0.0	1408.0	700.52	0.73	0.00	0.00	0.00
10	6	2761.0	1391.55	1.83	0.4	2761.0	1391.55	1.83	0.00	0.00	0.00
10	10	4645.6	2328.27	2.30	1.6	4647.8	2330.48	2.31	0.05	0.09	0.16
100	5	201.0	94.04	2.83	182.6	204.0	95.66	4.25	1.47	1.69	33.45
100	20	689.6	307.34	1.08	557.4	733.8	362.75	7.03	6.02	15.27	84.66
100	50	1735.8	721.35	2.97	732.6	2026.0	1020.45	16.87	14.32	29.31	82.38
100	100	3668.4	1425.57	9.31	661.6	4524.6	2259.81	26.47	18.92	36.92	64.81
1000	50	173.8	78.17	30.86	266.8	177.8	82.87	52.53	2.25	5.66	41.26
1000	200	669.4	291.02	15.75	558.6	711.2	350.45	73.53	5.88	16.96	78.58
1000	600	2034.4	840.01	38.18	776.6	2475.2	1229.08	209.34	17.81	31.66	81.76
1000	1000	3573.4	1376.07	96.18	676.6	4407.0	2212.05	267.14	18.92	37.79	64.00

表 1: 割込みの効用についての検証実験

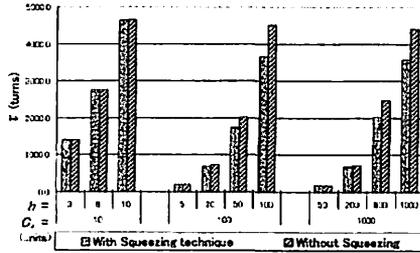


図 5: 所要ターン数比較

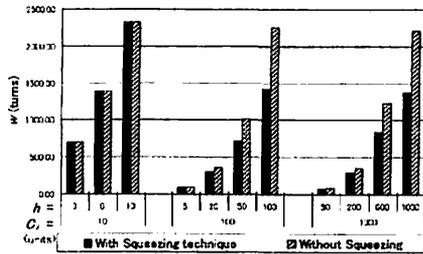


図 6: 平均待ちターン数比較

以上の実験結果によると、資源の容量がある程度以上あるならば、資源の割込み使用メカニズムを用いることによって、作業効率の向上を十分に見込むことができると言える (図 7)。

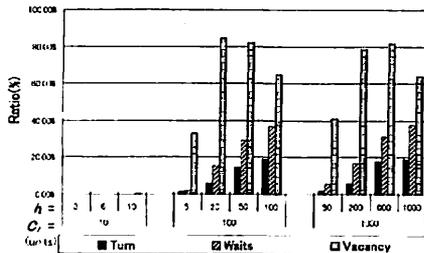


図 7: 割込みによる性能向上率

## 5 まとめ

本稿では、優先度による一意的な資源割当てに割込みの概念を導入することによって、資源利用の効率を高める手法を提案した。

デッドロックや飢餓状態が発生することはなく、割込み使用するプロセスの影響によって他のプロセスの資源獲得が妨げられることもない手法を示すことができた。また、

実験によって、我々の割込みメカニズムは十分に多くのプロセスに恩恵を与えることができるとの結論に至った。

今後は、割込みに関する計算処理の複雑さやメッセージ量の増加などについて詳しく検討し、それらを含めた上での程度の性能向上が見込めるかを検証する必要がある。また、その結果を考慮して、この割込みメカニズムの実装に取組む。

## 参考文献

- [1] Edsger W. Dijkstra: Solution of a problem in concurrent programming control. *Communications of the ACM*, Vol. 8, No. 9, pp. 569 (1967).
- [2] Edsger W. Dijkstra: Hierarchical Ordering of Sequential Processes. *Acta Informatica* Vol. 1, pp. 115-138 (1971).
- [3] Judit Bar-Ilan and David Peleg: Distributed Resource Allocation Algorithms, *LNCS 647*, pp. 277-291, (1992).
- [4] Shailaja Bulgannawar and Nitin H. Vaidya: A Distributed K-Mutual Exclusion Algorithm, *Proceedings of the 15th International Conference on Distributed Computing Systems*, pp. 153-160 (1995).
- [5] Yoshifumi Manabe and Naka Tajima: (h,k)-arbiters for h-out of-k Mutual Exclusion Problem, *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, (1999).
- [6] R. Baldoni: An  $O(N^{M/M+1})$  distributed algorithm for the k-out of-M resources allocation problem, *Proceedings of 14th ICDCS*, pp. 81-88, (1994).
- [7] Raynal, M.: A Distributed Solution for the k-out of-m Resources allocation problem, *Lecture Notes in Computer Sciences*, Springer Verlag, Vol. 497, pp. 599-609, (1991).
- [8] Injong Rhee: A Fast Distributed Modular Algorithm for Resource Allocation, *Proceedings of the 15th ICDCS*, pp. 161-168, (1995).
- [9] F.Belik, L.Kozma, and D. Krznarić: Avoiding Deadlock and Starvation in a Distributed Resource Allocation System, *Proceedings of the 13th International Conference on Parallel and Distributed Systems*, pp. 112-117 (1993).