システムソフトウェア講義の概要

1. 計算機システムの復習：中央演算処理装置(CPU)，プログラムの実行，主記憶装置，補助記憶装置
2. オペレーティングシステムとは：CPU，主記憶装置，補助記憶装置などの抽象化
3. CPUの抽象化１：スレッドとプロセス，割り込み
4. CPUの抽象化２：CPUの割り当てアルゴリズム
5. 主記憶の抽象化：アドレス空間と仮想記憶
6. 補助記憶装置の抽象化：ファイルシステム
7. 演習問題
8. プログラミングシステムの概要，文法とそのクラス，字句解析と正規文法
9. 正規表現からの非決定性オートマトンの生成、決定性オートマトンへの変換
10. 字句解析用オートマトン生成ソフトウエアの実際
11. 構文解析と導出，文脈自由文法の構文解析法：LL構文解析
12. 文脈自由文法の構文解析法：LR構文解析
13. コンパイラ-コンパイラと構文解析の実際
14. 演習問題
15. 講義の総括と試験

CPUの高度な動作（復習）

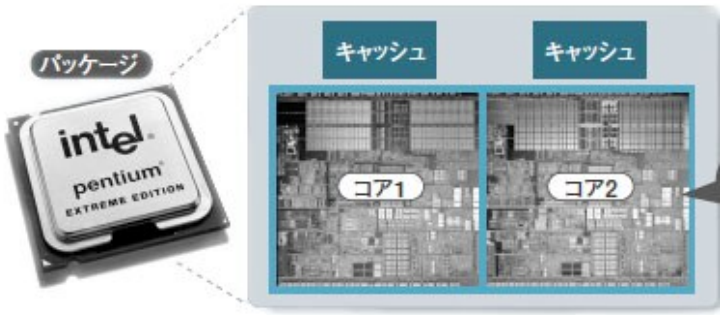
並行処理: Concurrent Processing

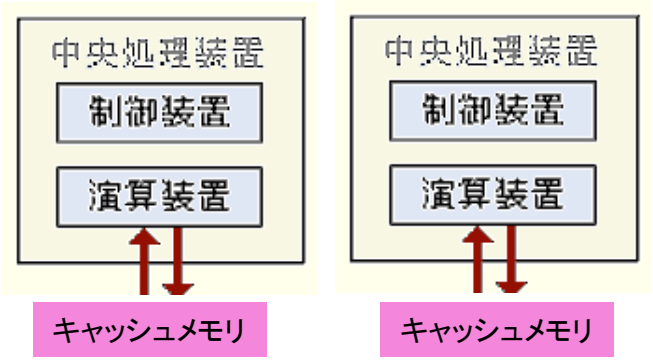
並列処理:Parallel Processing

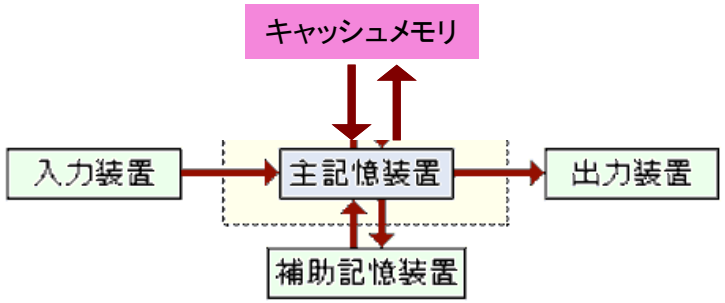
複数の仕事を「同時」にこなす

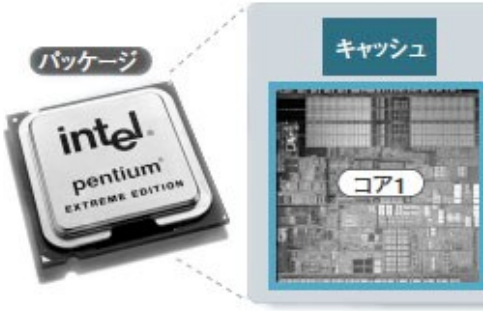
- 並列処理
  - CPUが複数あり，それらが同時に動作する (SMP:対称型並列処理＝メモリ共有型並列計算機による)
    - CPUのコア（制御部と演算部のペア）が複数存在する場合(マルチコア)　後述するキャッシュメモリは共通である
    - 制御部のみが複数存在する場合（Hyper Threading）この場合も，OSは複数のCPUを検出する.
- 並行処理
  - 時分割処理システム(TSS:Time Sharing System)によって一台の計算機で複数の処理が行えるようになっている.

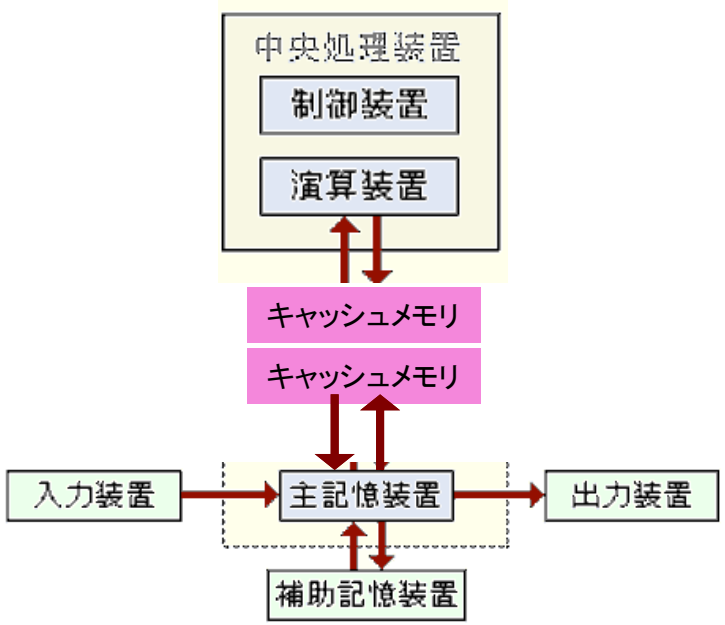
マルチコア


パッケージ
intel
pentium
EXTREME EDITION
キャッシュ
キャッシュ
コア1
コア2


中央処理装置
制御装置
演算装置
中央処理装置
制御装置
演算装置
キャッシュメモリ
キャッシュメモリ

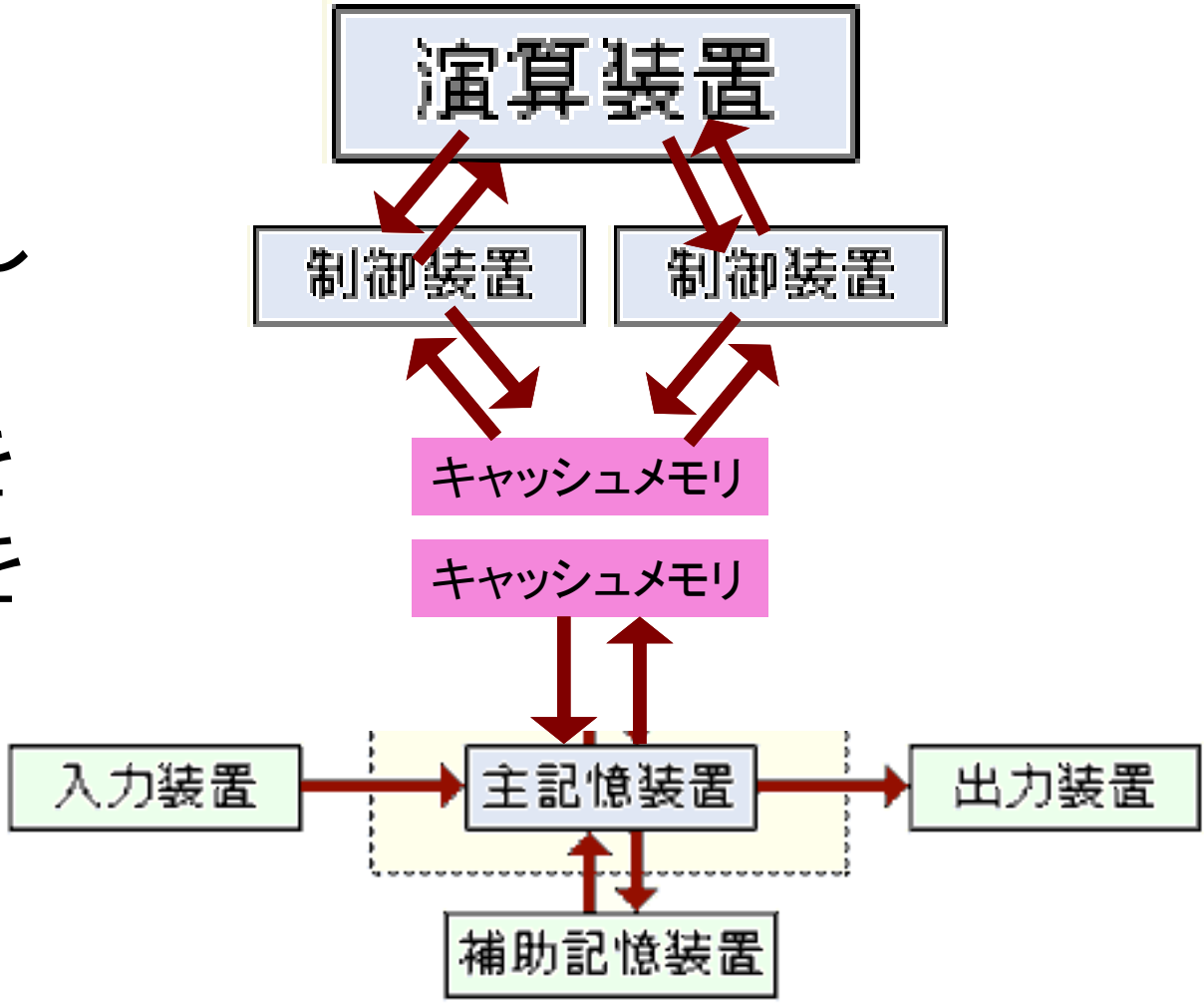

キャッシュメモリ
入力装置
主記憶装置
出力装置
補助記憶装置


パッケージ
intel
pentium
EXTREME EDITION
キャッシュ
コア1

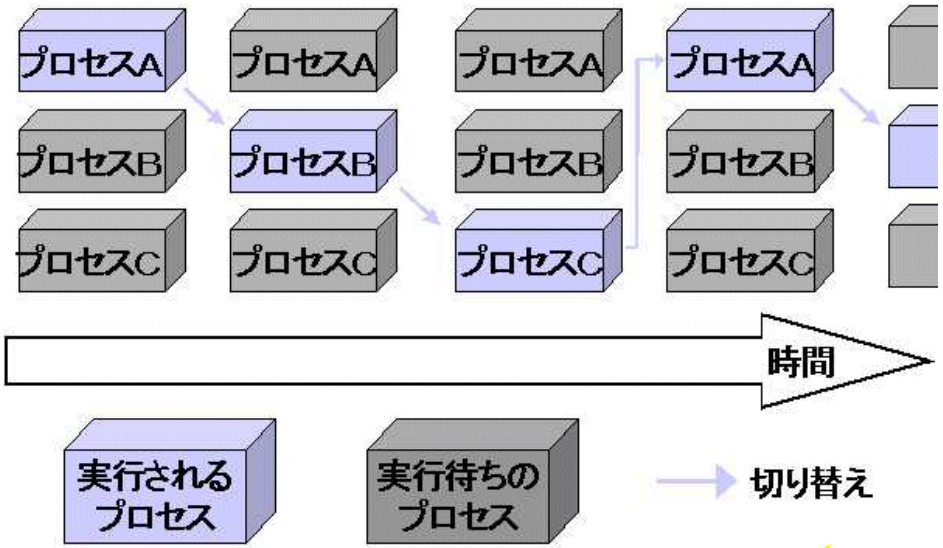

中央処理装置
制御装置
演算装置
キャッシュメモリ
キャッシュメモリ
入力装置
主記憶装置
出力装置
補助記憶装置

Hyper Threading

- 命令読み取りの部分だけ二重化し,
- 空いているALUを使って並列処理を行う.
- OSから見るとCPUは二つに見える



1つのCPUで並行処理を行う.



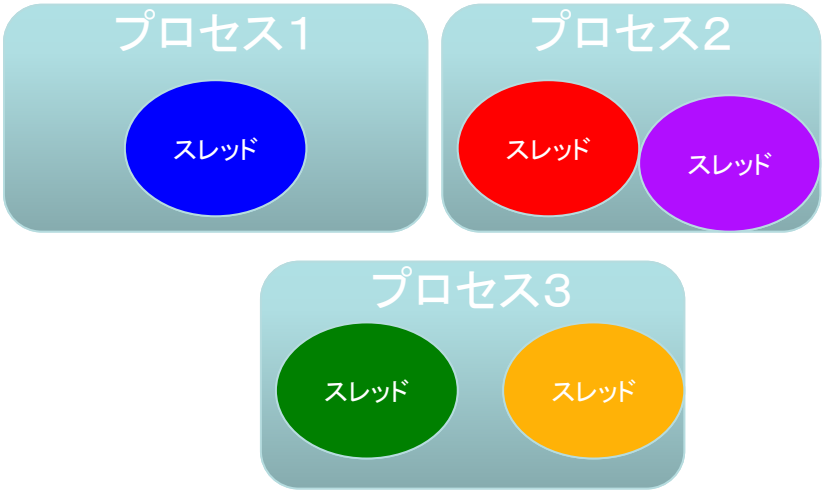
単一のCPUでも，時間を分割して複数のプログラムを実行させることができる.

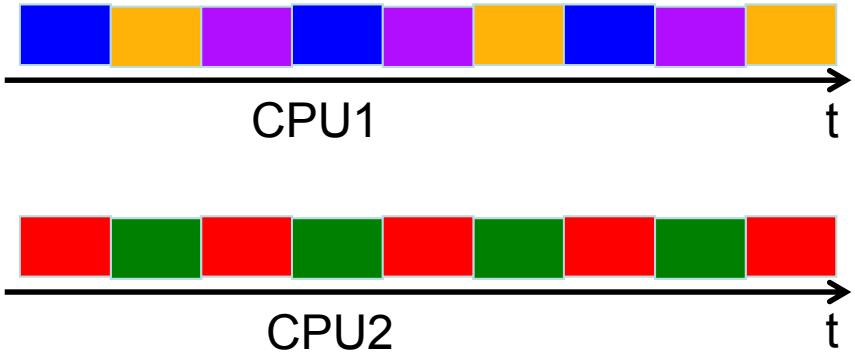
- プロセスとは，実行中のプログラムのこと．タスクとも呼ぶ.
- 同じプログラムでも，異なる実行であれば，プロセスとしては異なる.
- スレッドとは異なり，OSから記憶領域やディスク資源を割り当てられて実行される.
- プロセスが停止状態でも入出力は行える.

これが時分割処理システム
（Time Sharing System）

実際の時分割処理はプロセスではなくスレッド単位で行われる.

- プロセスとスレッドの対応関係は，プログラムによって決められる.
- スレッドとCPUの対応関係はスレッド起動時にOSが決める.
- 各スレッドが時分割で動作する.





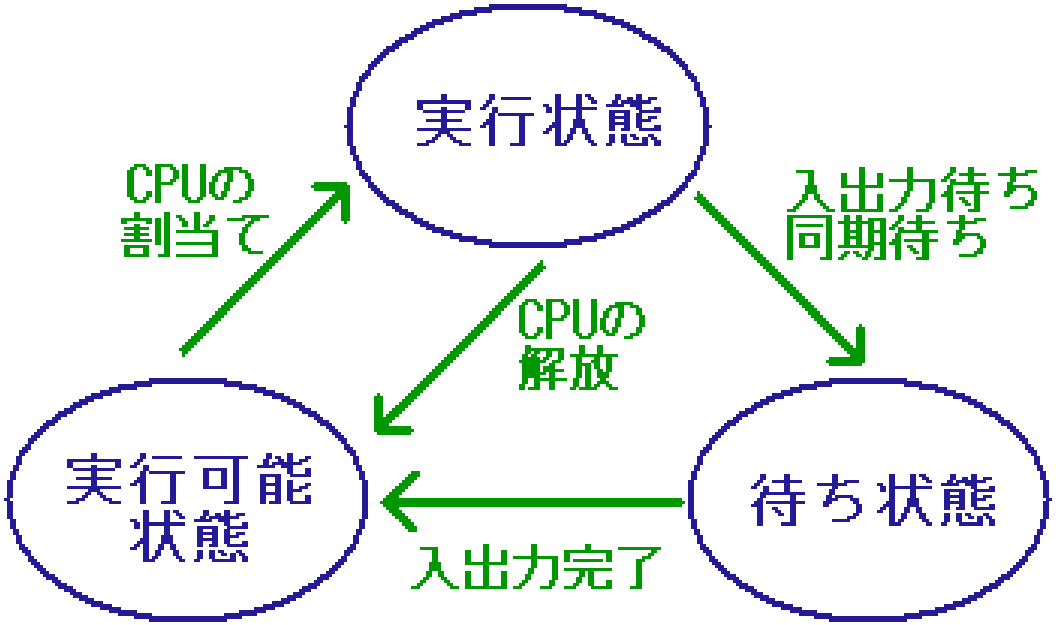
OS内での計算の実行主体は仮想化されたCPU

- 仮想CPU上で実行されるプログラムのことをプロセスまたはタスクと呼ぶ.
  - プロセスには1つ以上のスレッドが対応付けられている.
  - 同じプログラムでも，異なる実行であれば，プロセスとしては異なる.
  - スレッドとは異なり，OSから記憶領域やディスク資源を割り当てられて実行される.
  - スレッドが停止状態でも入出力は行える.

スレッドの状態

1. 実行状態のスレッドは，一定時間CPUを割り当てられるとCPUを解放しなければならない.
2. 実行状態のスレッドが入出力待ちになったときはCPUを解放して待ち状態になる. この場合OSは，実行可能スレッドの中からCPUを割当てる.
3. 入出力待ちのスレッドの入出力が済めば，実行可能状態に遷移し，CPUの割当てを待つ.

CPUの割り当てと解放は，後述するスケジューリングアルゴリズムに従う.



スケジューリング

- スケジューリングとは，各スレッドに優先順位を与え，優先順位の高い順にCPUへの割り当てを行う処理である．実際には「優先度付きキュー」が用いられる．
- 必ずしもTSS（時分割処理）を前提としない．
- 複数のCPUが存在する場合は，優先順位の高い順にCPUへの割り当てを行う．
  - 先着順
  - 時間順
  - ラウンドロビン
  - 締め切り順
  - レート・モノトニック

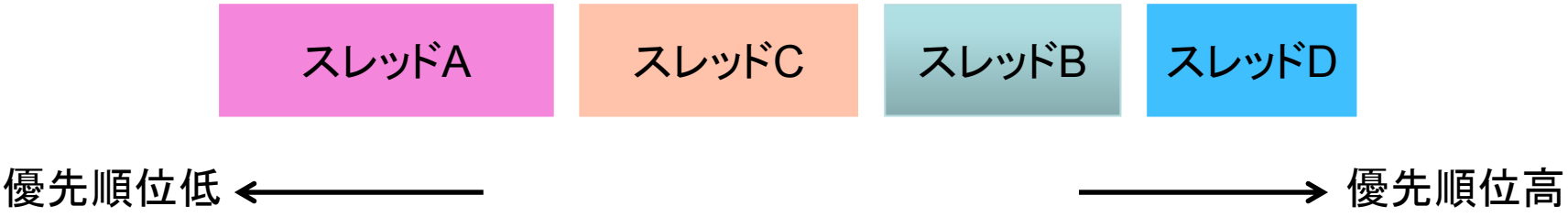
先着順:FCFS
First Come First Serve

- スレッドA,B,C,Dがこの順に実行可能になった場合

こちらから
CPUに割当
られる

時間順：SPTF
Shortest Processing Time First

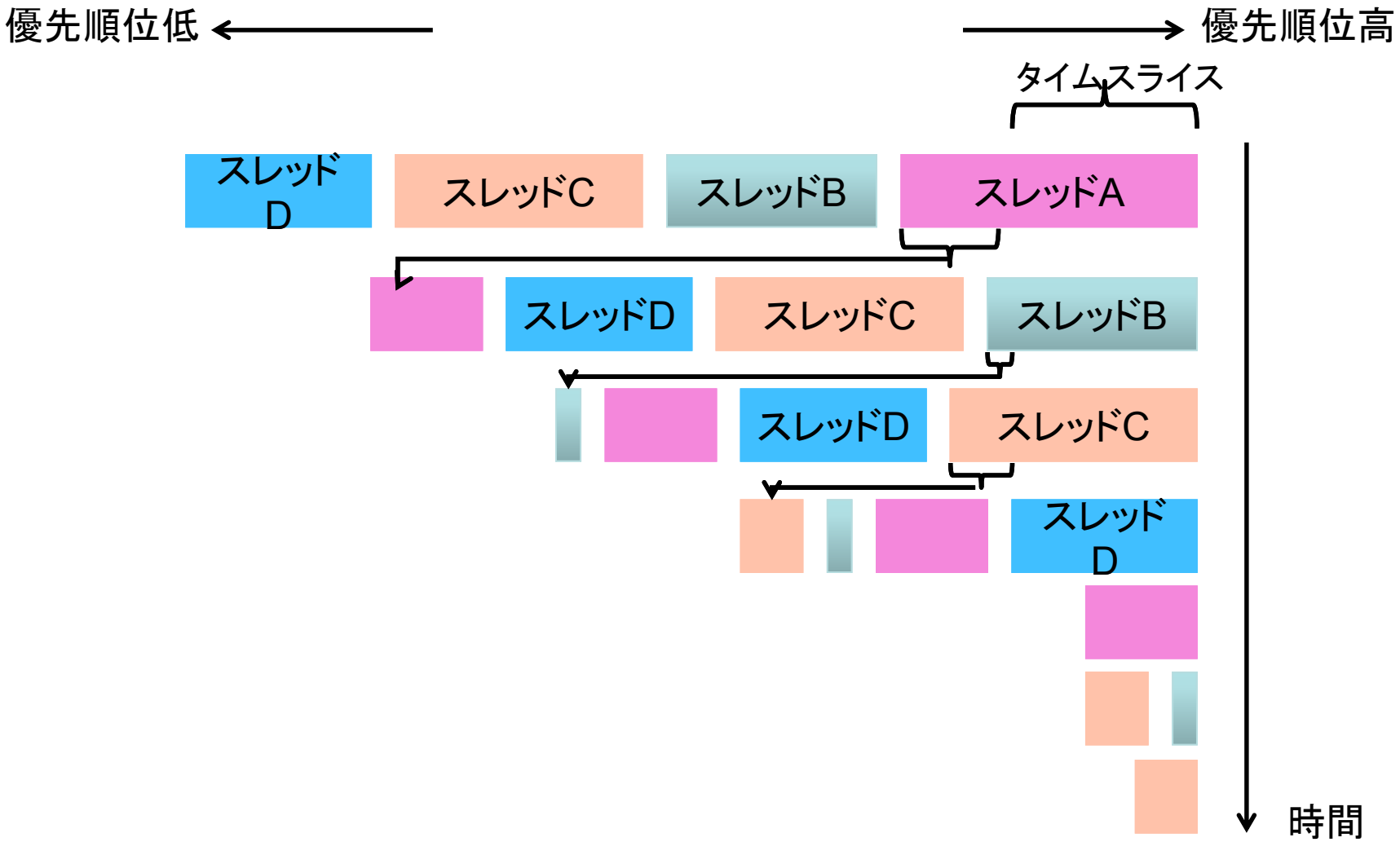
- スレッドD,B,C,Aがこの順に実行時間が短い場合



- スレッドの平均応答時間を最小にする
- 各スレッドの実行時間があらかじめ分かっていなければならない.

ラウンドロビン：RR
Round Robin

- CPUが1つの場合



実時間スケジューリング Real-time Scheduling

- 締め切り順(EDF: Earliest Deadline First)

各スレッドの終了までの時間（締め切り）が短い順に優先順位を上げる.

- レート・モノトニック(RMS: Rate Monotonic Scheduling)

処理が起動される頻度が高いスレッドに高い優先順位を与える.

スレッド・スケジューリングの例

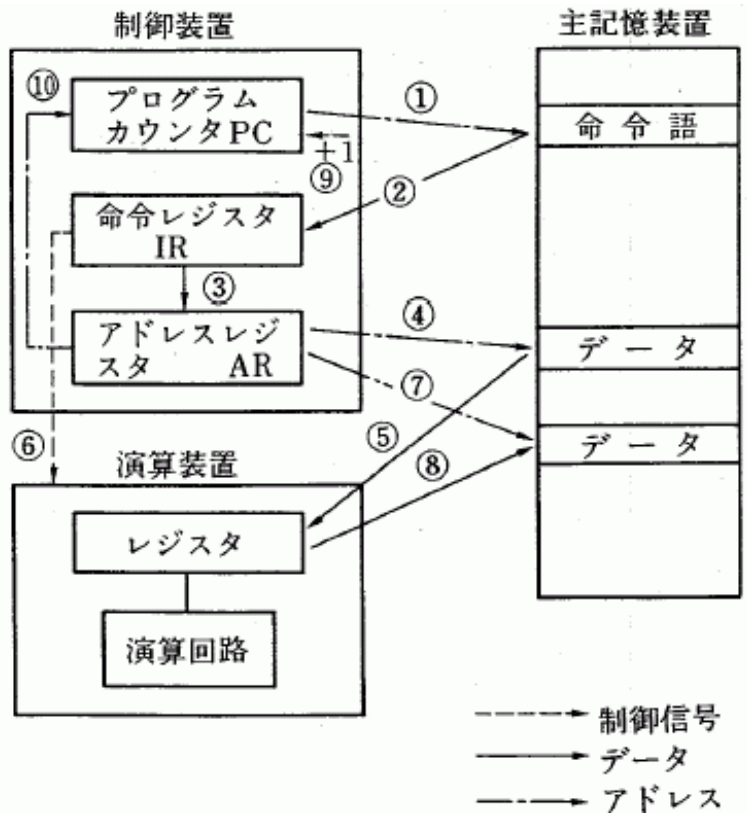
	スレッドの属性			スケジューリング・アルゴリズム			
	実行可能になった時刻	残余実行時間(R)	処理完了締め切り時刻(D)	FCFS	SPTF	RR	EDF（D-R順）
スレッドA	1	6	18	1	4	1	2
スレッドB	3	8	30	2	5	2	4
スレッドC	4	1	35	3	1	3	5
スレッドD	7	2	13	4	2	4	1
スレッドE	9	4	25	5	3	5	3

時刻10でスケジューリングが行われた場合の結果

優先順位：小さいほど優先順位は高い

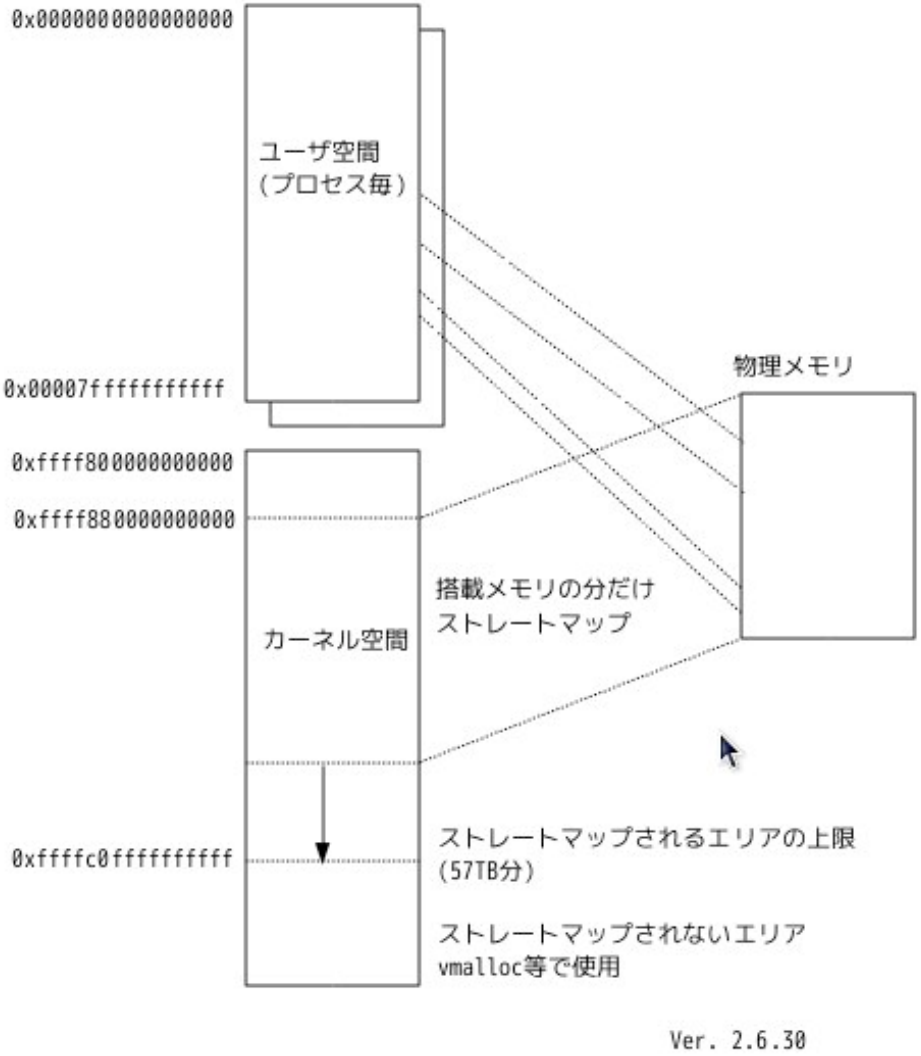
スレッドの切り替え:ディスパッチャ

- スレッドの文脈(コンテクスト)
  - PC: Program Counter
  - PSW: Processor/(Program) Status Word
  - レジスタ類
- 割り込みによってこれらを退避/復帰させる
- これによって1つのCPUが複数のスレッドを時分割実行できる



プロセスに割り当てられたリソースと属性

- リソース
  - ユーザ・アドレス空間
  - オープンしたファイル
  - ネットワークソケット
  - 他
- 属性
  - 所有者
  - 優先順位
  - 実行時間
  - 他



前回の問題の答え

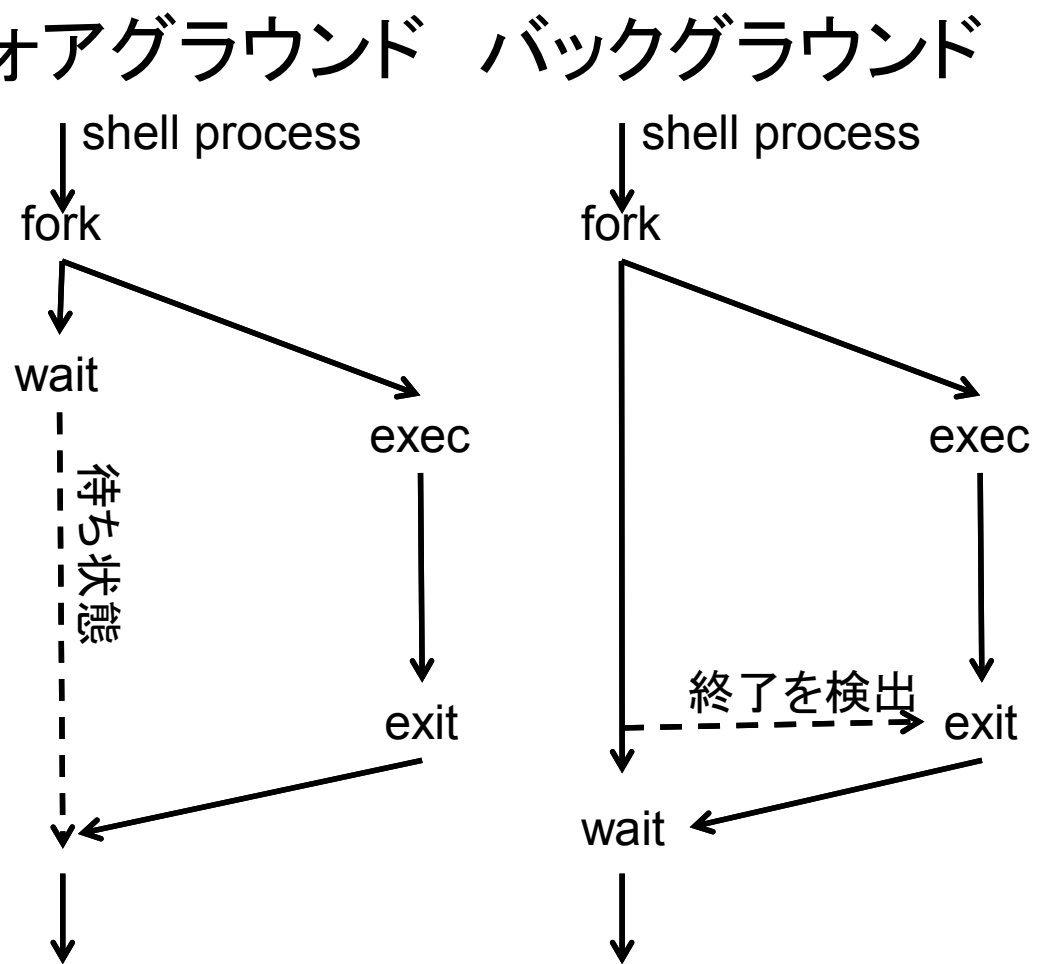
問題1

	スレッドの属性			スケジューリング・アルゴリズム			
	実行可能になった時刻	残余実行時間(R)	処理完了締め切り時刻(D)	FCFS	SPTF	RR	EDF
スレッドA	1	9	20	1	5	1	1
スレッドB	4	7	30	2	4	2	4
スレッドC	6	1	25	3	1	3	5
スレッドD	7	4	23	4	3	4	2
スレッドE	8	3	23	5	2	5	3

時刻10でスケジューリングが
行われた場合の結果

問題2

- UNIXのシェルのバックグラウンドジョブでは, fork, exec, wait, exitがどのように動作しているか.



問題3

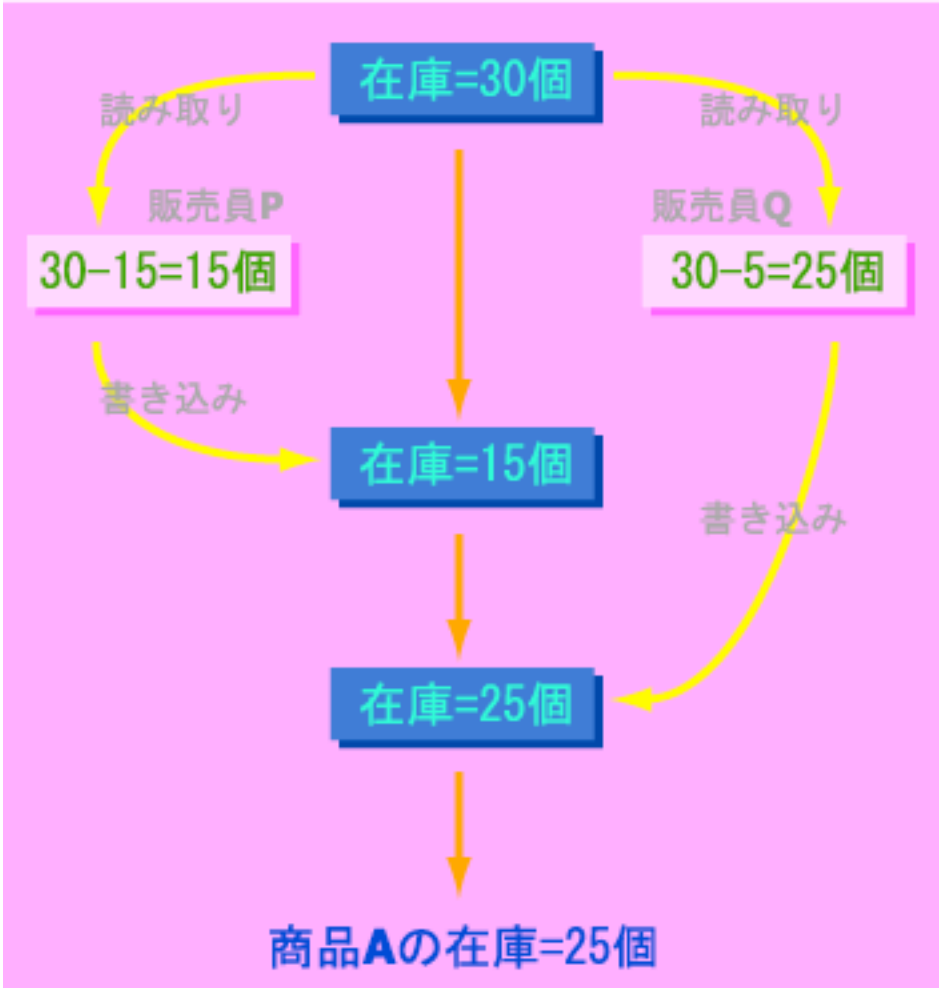
- なぜプロセスごとに異なるアドレス空間が割り振られるのか考えを述べよ.

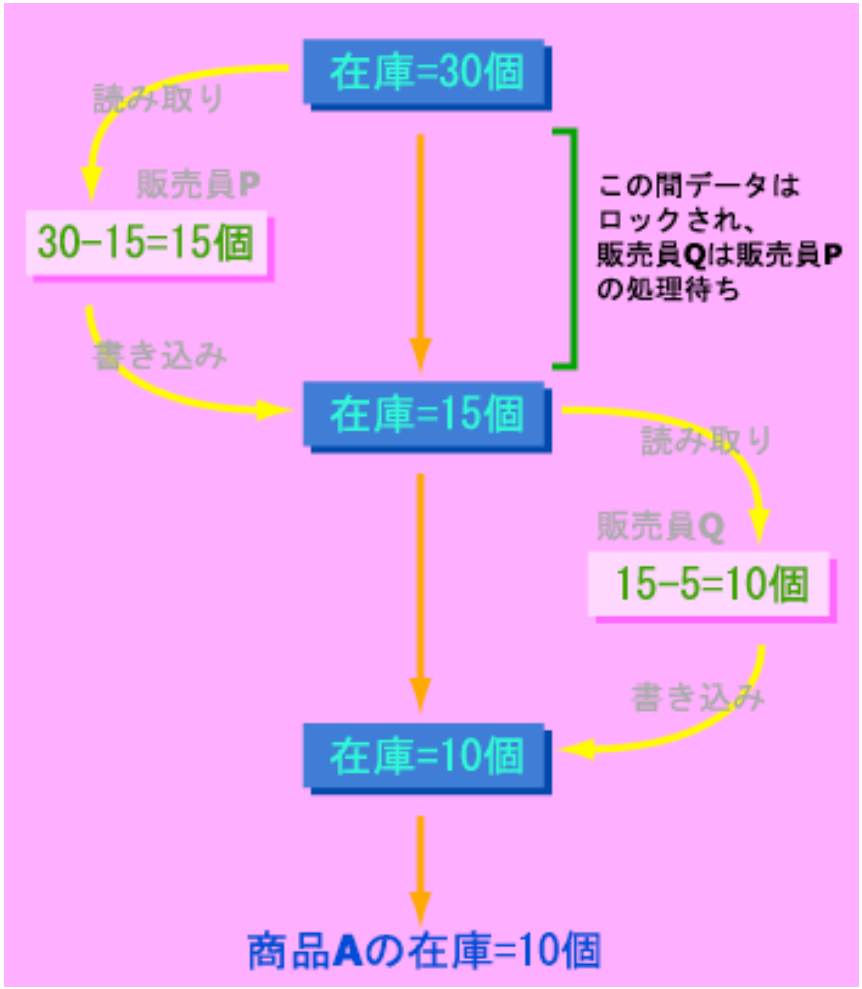
機械語のプログラム等では，絶対番地を指定してジャンプするものがある．このようなプログラムが配置される番地は固定されており，共通のアドレス空間に複数のプログラムを配置すると，衝突を起こす可能性がある．この問題を回避するために，プロセス毎に異なるユーザアドレス空間が用意されている.

並行処理における同期・通信

相互排除（排他制御）

- なぜ排他制御が必要か？





複数のスレッド(プロセス)間で同じ資源を取り合う

- これが，銀行預金の引き落としで起きたらどうだろうか？9万9千円＋千円＝10万円を引き下ろしても，残額が9万9千円残っていたら，銀行は倒産する.
- 前頁の例では，販売員がスレッド，在庫の量が資源.
- 上の例では，引き落としをする人がスレッド，口座残高が資源
- 資源にアクセスするときに，排他制御が必要.

相互排除の方法

- 割り込み禁止：スレッドの切り替わりは割り込みによって起こるので，割り込みを禁止する．
- フラグ：現在アクセス中というフラグを立てる．
- セマフォ：整数値を用いた一般化されたフラグ．排他制御以外に，有限個のリソースを複数スレッドに分配する際にも使う．
- モニタ：排他制御専用．セマフォと論理的に等価．

割り込み禁止

- スレッドの切り替わりは，ディスパッチャが行なっており，これは割り込みによって起動するので，割り込みを禁止すれば，問題も発生しない.
- しかし，この方法では排他制御を必要としないスレッドも停止させてしまうため，スレッドの待ち状態が発生する. このため，この方法は一般的には用いられない.

フラグ

- フラグ：リソースに対して現在アクセス中である，というフラグを立てる．
  - 不可分命令：これを行うには，1命令でフラグが立たなければならない．（フラグ変更途中に他のスレッドがフラグを操作しないようにするため．）
    - TAS命令： Test and Set命令
    - CAS命令： Compare and Swap命令

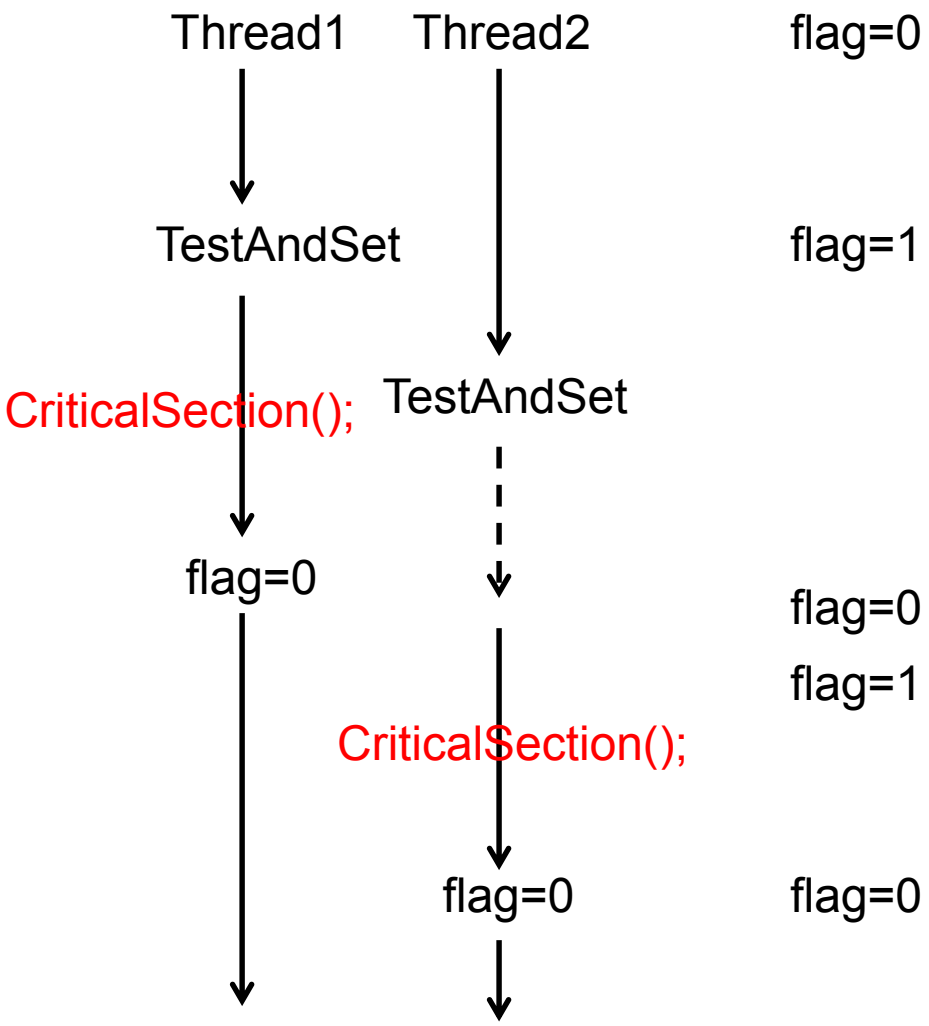
TAS命令：Test and Set命令

```
int TestAndSet(int* Vaddr){
    if (*Vaddr) return 0;
    else { *Vaddr=1;
            return 1;}
}

int flag=0;

void ThreadExecution(){
    while(!TestAndSet(&flag));
    CriticalSection();
    flag=0;
}
```

- TestAndSetは，不可分命令として実装されていなければならない．
- flagの値が先に1にセットされていれば，TestAndSet(&flag)は0になり，whileで空回りする．flagが0であれば，戻り値が1になり，whileループから抜ける．

TAS命令：Test and Set命令

```
int TestAndSet(int* Vaddr){
    if (*Vaddr) return 0;
    else { *Vaddr=1;
            return 1;}
}

int flag=0;

void ThreadExecution(){
    while(!TestAndSet(&flag));
    CriticalSection();
    flag=0;
}
```



CAS命令：Compare and Swap命令

```
int CompareAndSwap(int* Vaddr, int Expect, int New){
    if (*Vaddr != Expect) return 0;
    else { *Vaddr=New;
            return 1;}
}

int flag=0;

void ThreadExecution(){
    while(!CompareAndSwap(&flag,0,1));
    CriticalSection();
    flag=0;
}
```

- CompareAndSwapも，不可分命令として実装されていなければならない.

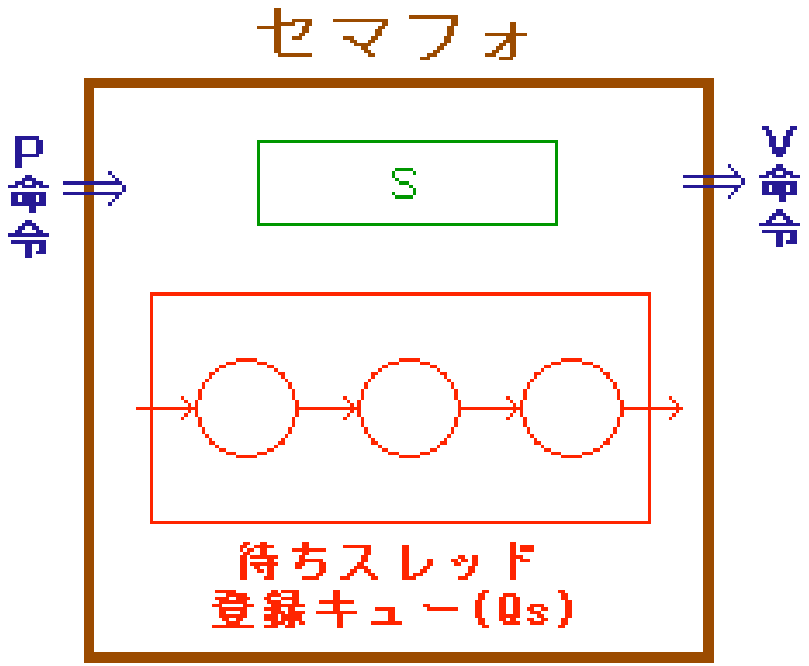
ビジーウエイト

- 何度もTASあるいは, CAS命令を実行することで, flagの値が変化するのを監視すること. スピンロック, あるいは, ビジーウエイトとも言う.

- ビジーウエイトは, CPUを無駄に消費するため, 資源の無駄遣いになる.

セマフォ

- セマフォ(Semaphore)は整数値sと事象発生待ちのスレッドを貯えるキューQsからなる.



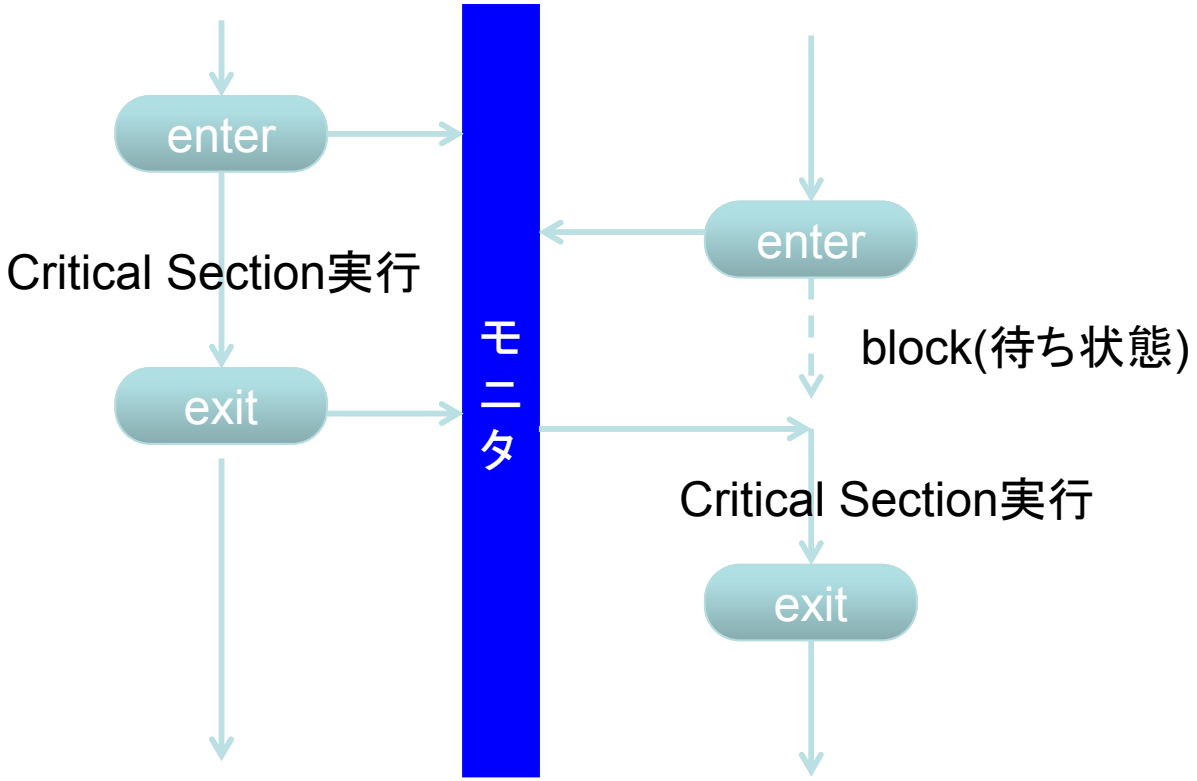

```
int s=1;    //Binary Semaphore
P(){
  if(s>0) s--;
  else{自スレッドをQsに入れる }
 }

V(){
  if(Qsが空でない){
  Qsからスレッドを取り出し実
  行可能にする
  }else s++;
}
```

セマフォ

- スピンロックではなく，サスペンドロック
- 空回りをしないので，CPU資源を無駄遣いしない.
- 起動が多少遅れる可能性がある.
- sの初期値を1以外にすれば，複数のリソースを複数のスレッドで分け合う場合にも使える.

モニタ

モニタに対して，enterし，enterできた場合にCriticalSectionを実行．その後に，exitする．

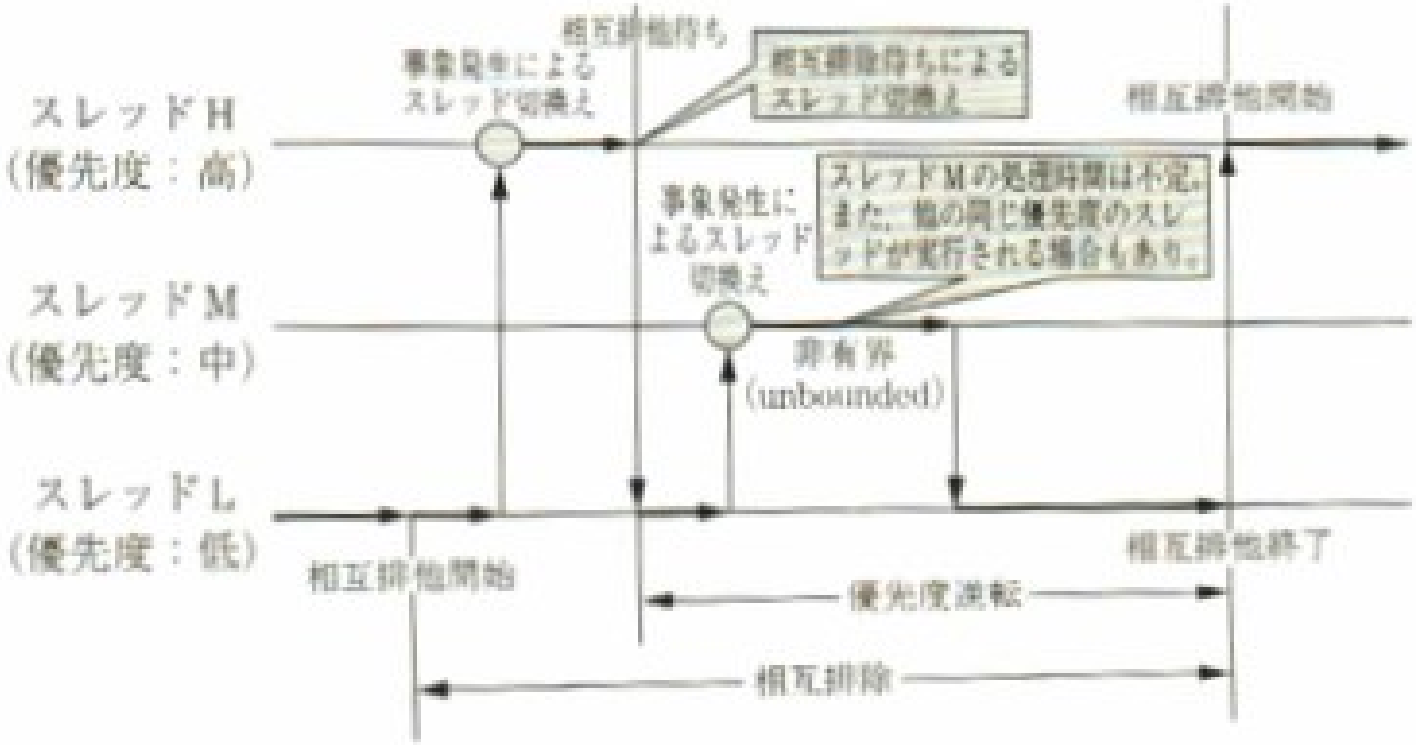


優先度逆転

- 実行の優先度が高いスレッドでも，優先度が低いスレッドがセマフォをロックするなどして，資源を占有している場合は，待たなければならない．
- このように優先度の高いスレッドが優先度の低いスレッドによって実行をブロックされる現象を「優先度逆転」と呼ぶ．
- 通常は問題にならないが，リアルタイムシステムでは問題になる．

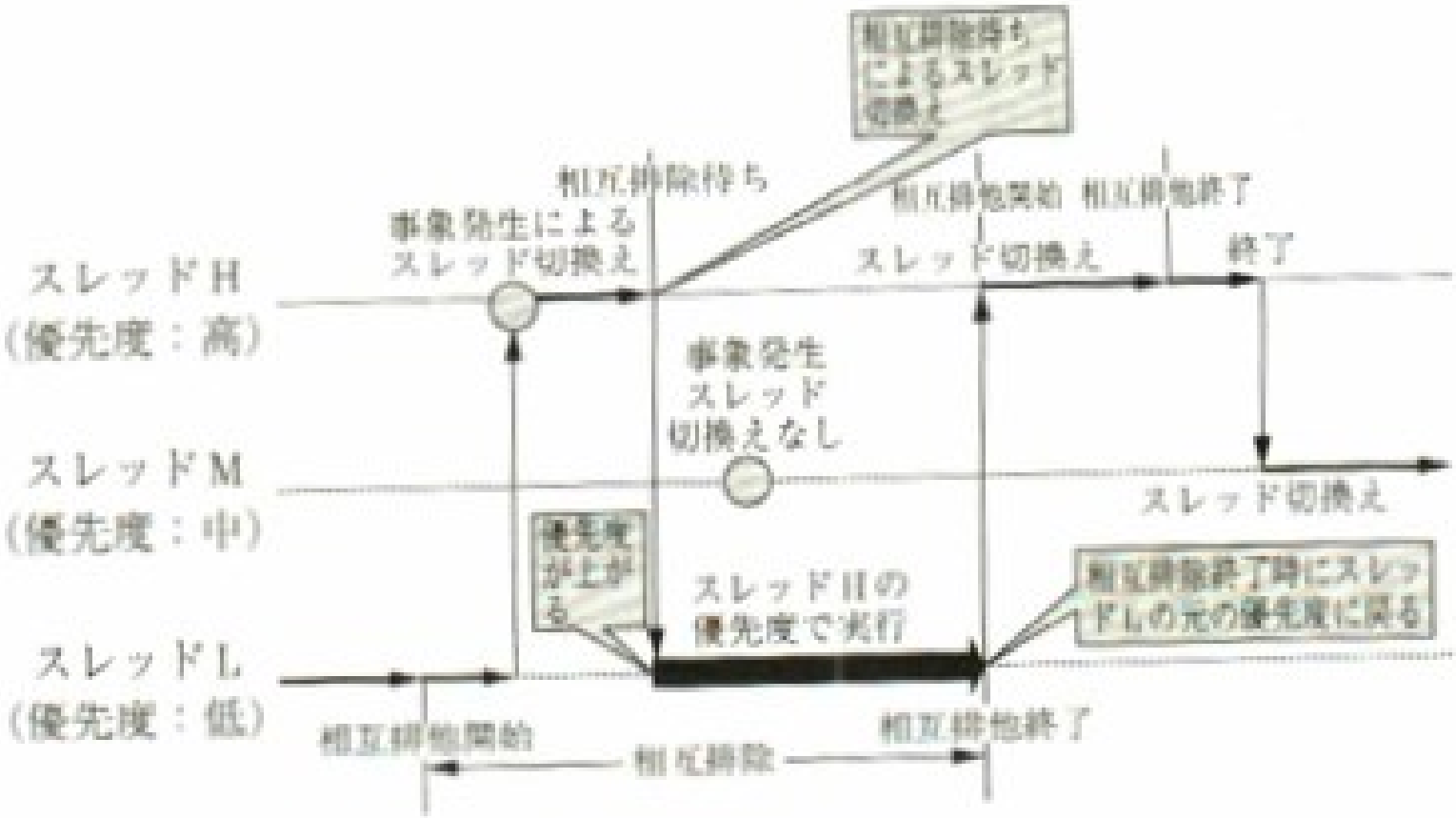
優先度逆転によって生じる問題

- 優先度逆転により，相互排除期間が伸びるケース



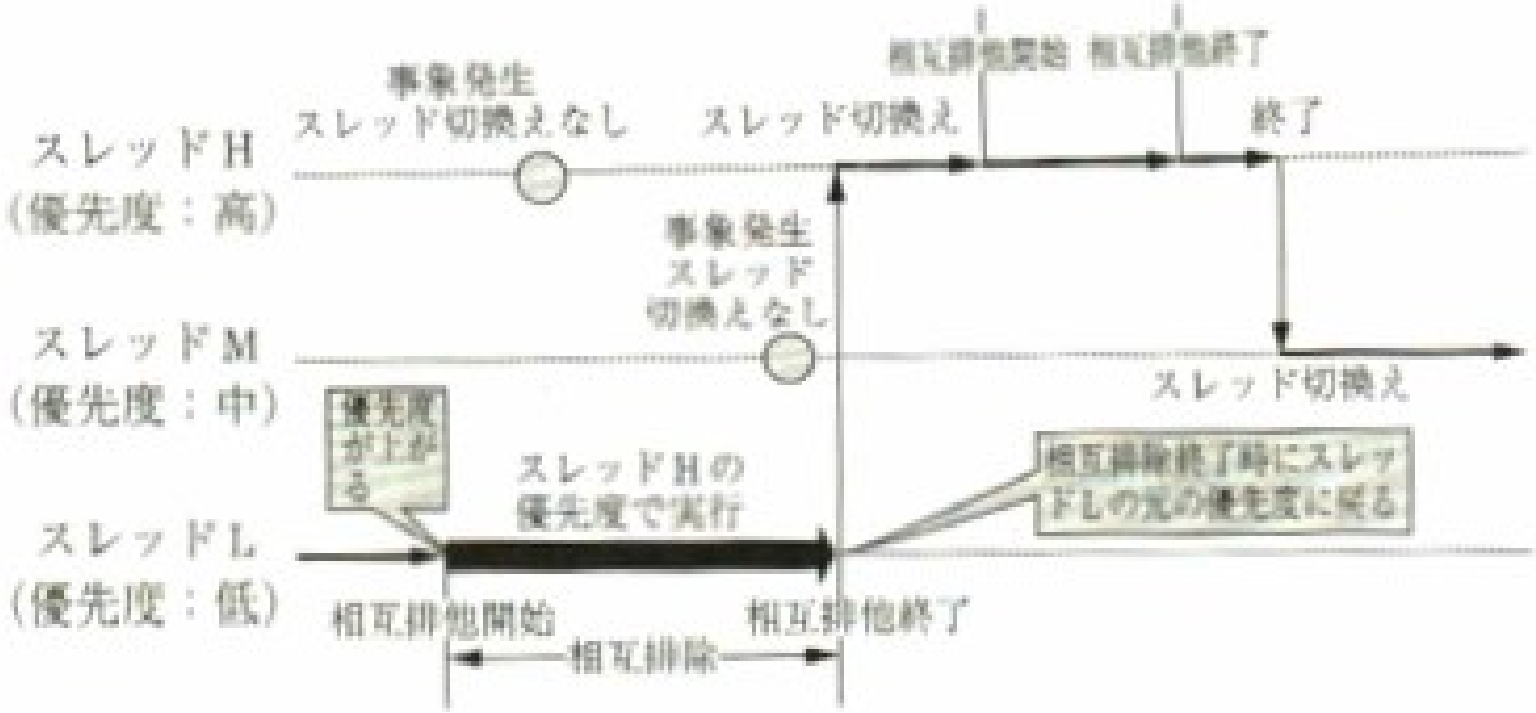
（a） 優先度逆転

優先度継承アルゴリズム



(b) 優先度継承アルゴリズム

優先度上限アルゴリズム



（c） 優先度上限アルゴリズム

デッドロック

プロセスPがその実行過程でOSに資源Rの割り当てを要求するとき，他のプロセスとの関係で，その割当が絶対に行われないという状況をデッドロックと呼ぶ.

デッドロックは，資源割付グラフ上のループとして検出することが出来る.

資源割付グラフ

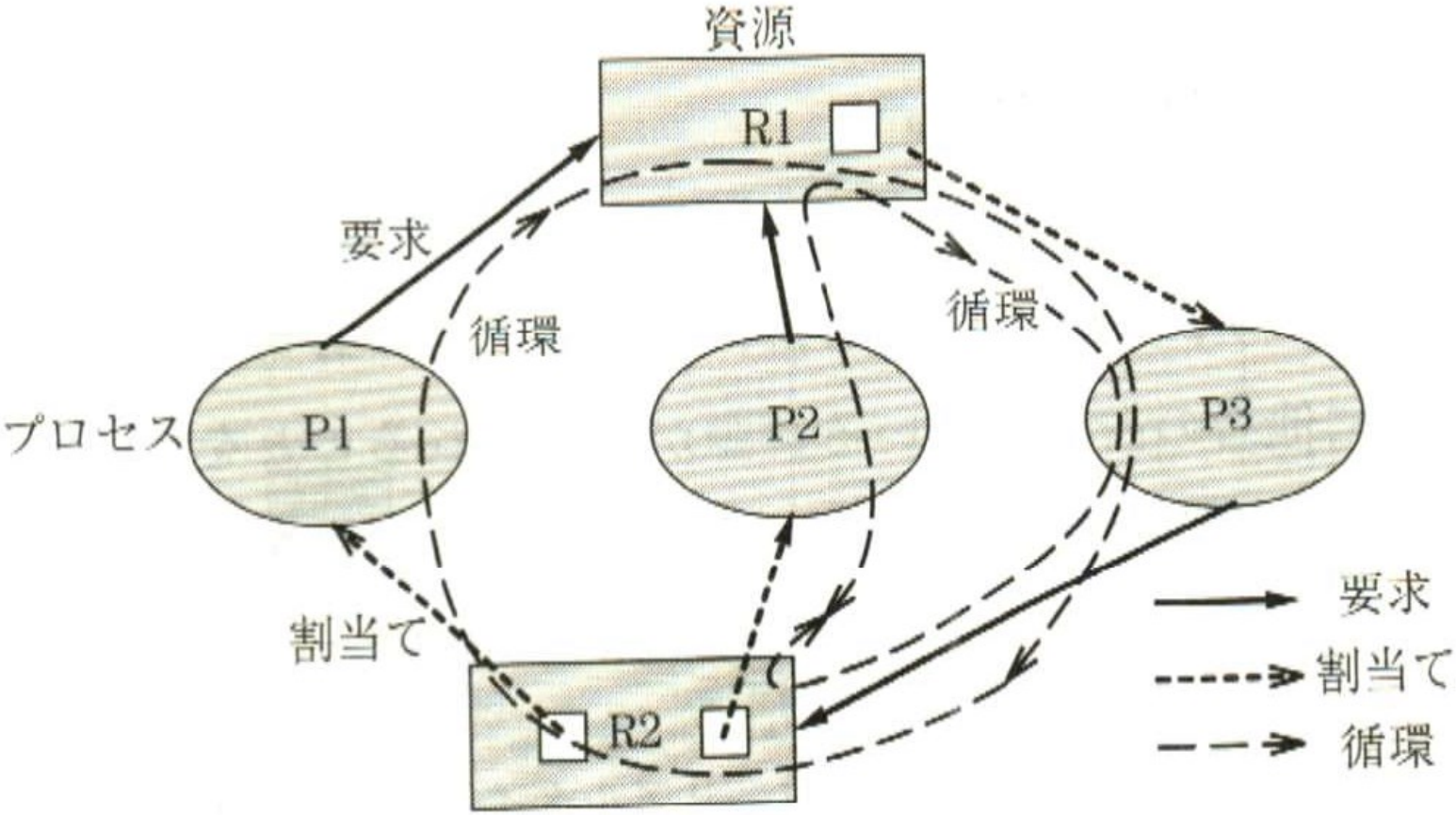


図 3.11　資源割付グラフ

問題

```
int TestAndSet(int* Vaddr){
    int OldValue;
    OldValue=*Vaddr;
    *Vaddr =1;
    return OldValue;
}

int flag=0;

void ThreadExecution(){
    while(!TestAndSet(&flag));
    CriticalSection();
    flag=0;
}
```

TestAndSetを左のように書きなおしたが，間違いがある．それはどこか？