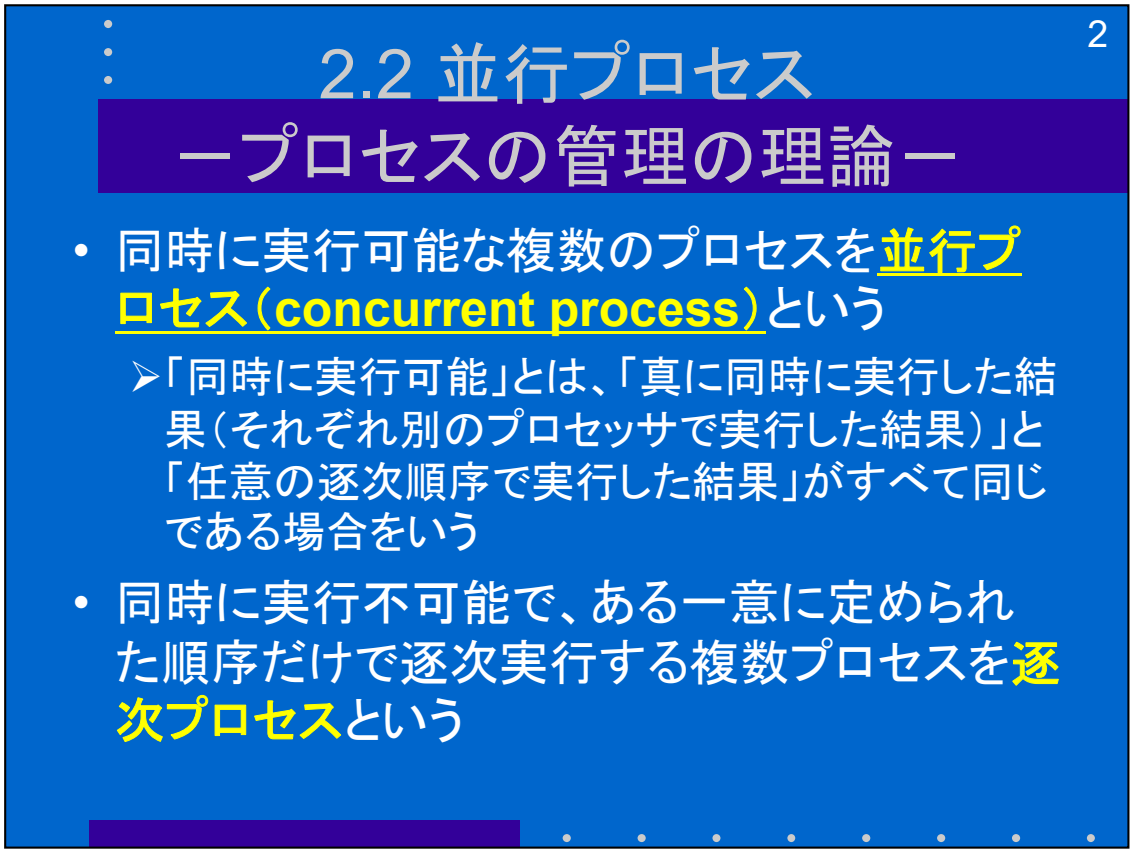
オペレーティングシステム

資料　第6分冊(2021)

村田正幸（murata@ist.osaka-u.ac.jp)
○松田秀雄(matsuda@ist.osaka-u.ac.jp)



同時に実行可能な複数のプロセスを並行プロセス(concurrent process)といいます。

今回は、まず、この並行プロセスとそれに対する制御について説明します。

ここで、「同時に実行可能」とは、「真に同時に実行した結果(それぞれ別のプロセッサで実行した結果)」と「任意の逐次順序で実行した結果」がすべて同じである場合をいいます。

同時に実行不可能で、ある一意に定められた順序だけで逐次実行する複数プロセスを逐次プロセスといいます。

3

並行プロセスの生成

- 並行プロセスの生成を指定する実例には、次のようなものがある（図2.21）
- コルーチン
  - ユーザプログラムレベルでの並行プロセスの一つの実行概念であり、OSで実装されている例は少ない
- フォークアンドジョイン
  - **UNIXで標準的に実装**されている並行プロセスの生成方法
- 並行文
  - フォークアンドジョインの多重版
  - 並行プロセスの教科書の一つで、並行プロセスの生成方法として紹介されている（現在のOSでの実装例は少ない）

並行プロセスの生成を指定する実例には、次のようなものがあります

コルーチン

フォークアンドジョイン

並行文

なかでも、フォークアンドジョインはUNIXで標準的に実装されている並行プロセスの生成法となっています

4

2.2.2 プロセスの同期と相互排除

- 並行プロセスでは、次のような場合について並行プロセスの制御、すなわち**同期**が必要になる（図2.23）
  - ➢共有する資源（プログラム中での特定の領域やハードウェア装置など）の使用要求が競合する
  - ➢プロセス間で通信する（プロセス間通信）
- ある1個の共有資源を複数プロセスが同時に使用しないように制御する同期機能を、**排他制御**または**相互排除**制御という（図2.24）

並行プロセスでは、次のような場合について並行プロセスの制御、すなわち同期が必要になります（図2.23）

共有する資源（プログラム中での特定の領域やハードウェア装置など）の使用要求が競合する

プロセス間で通信する（プロセス間通信）

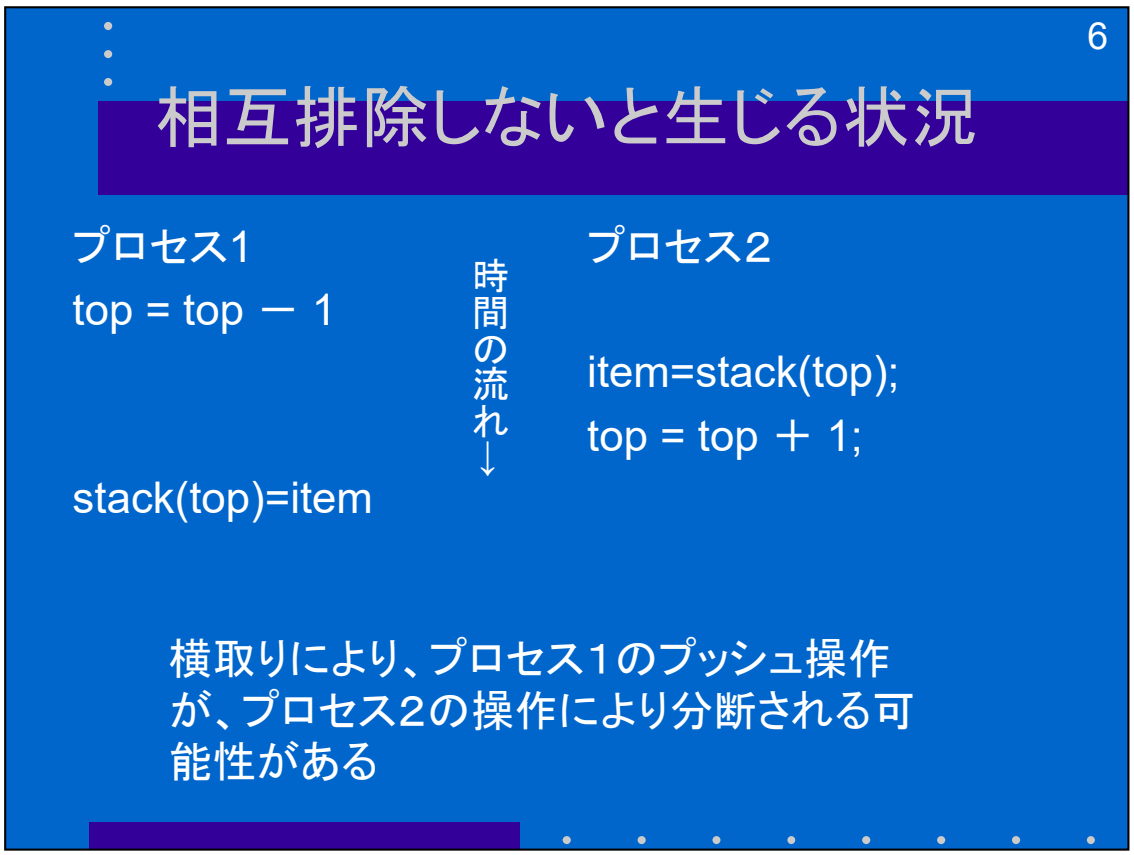
ある1個の共有資源を複数プロセスが同時に使用しないように制御する同期機能を、排他制御または相互排除制御といいます（図2.24）

5

相互排除が必要な例

- 配列でスタックを実現したとする（プロセスのスタック領域とは異なることに注意）
- 共有メモリ上に置いて複数のプロセスで利用
- プロセス1:データをプッシュ
  - top = top－1;
  - stack(top) = item;
- プロセス2：データをポップ
  - item=stack(top)
  - top = top＋1

相互排除が必要な例を示します。
スタックのプログラムの例です。
データのプッシュとポップの操作があります。



プッシュ操作をするプロセスと、ポップ操作をするプロセスの間で相互排除されていないと、プロセスの実行途中で横取りが生じて、意図しない結果になることがあります。

7

クリティカルセクション

- スタックのプッシュ, ポップ操作
  1. 分割して実行されてはならない
  2. 実行するプロセスは一度には1個だけでないといけない
  - ➢ プログラム中で2.の条件が必要な部分を、クリティカルセクションという（臨界領域、または際どい部分ということもある）
- **不可分操作**（atomic action）
  - ➢ 分割が許されない一連の実行操作
- **相互排除**（mutual exclusion）
  - ➢ 一つのプロセスしかクリティカルセクションには入れないようにする

先にあげた、スタックのプログラムの例では、プログラムの実行にあたって、

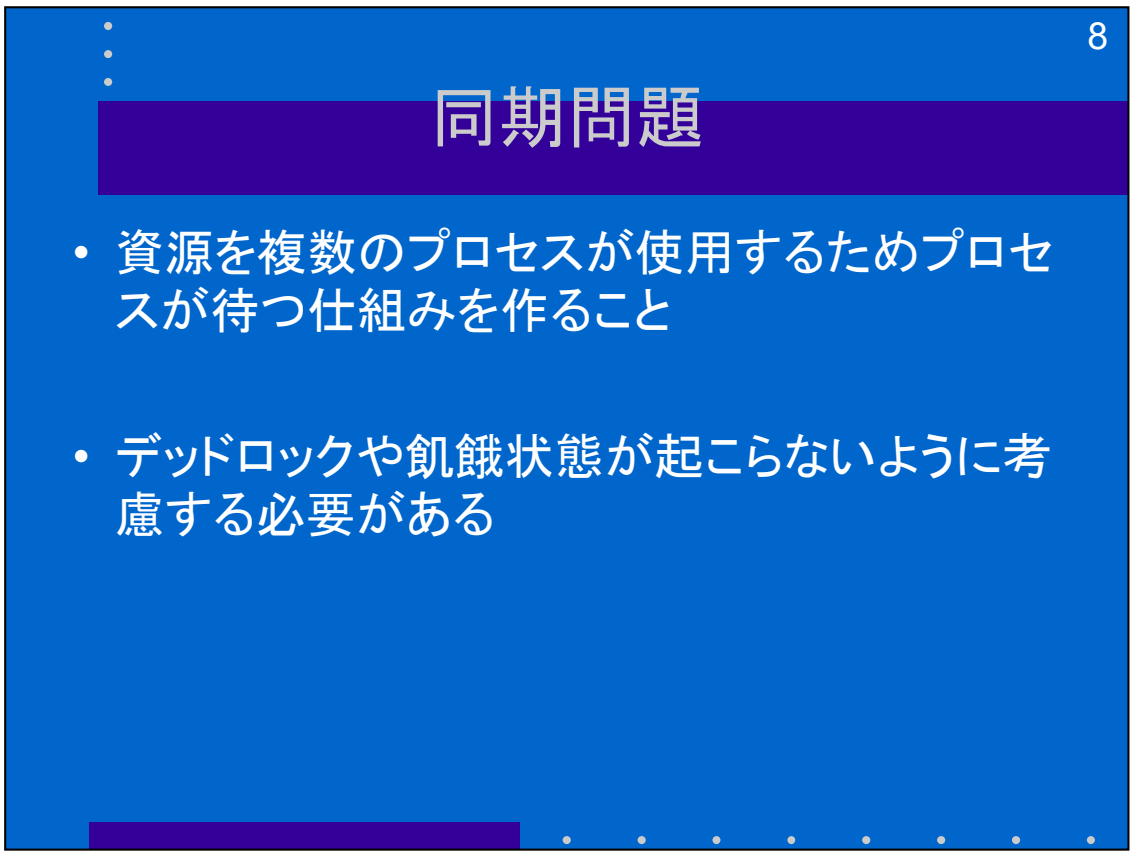
1. 分割して実行されてはならない
2. 実行するプロセスは一度には1個だけでないといけない

の2つの条件が必要なことがわかります。

この2条件のうちの、2. の条件が必要な部分を、クリティカルセクションといいます（臨界領域、または際どい部分ということもあります）

なお、1. は、不可分操作（atomic action）（分割が許されない一連の実行操作）といいます。

2. の条件は、相互排除（mutual exclusion）、つまり、一つのプロセスしかクリティカルセクションには入れないようにする、と表現することができます。



ここで、同期問題とは、資源を複数のプロセスが使用するためプロセスが待つ仕組みを作ることを指します。

なお、同期問題では、デッドロックや飢餓状態が起こらないように考慮する必要があります。

9

デッドロックと飢餓状態

- **デッドロック (deadlock)**
  - 資源を持っているプロセスが他の資源を待ってブロックしたままになる
- **飢餓状態　(starvation)**
  - あるプロセスが待っている資源が、解放されるたびに、他のプロセスに取られてしまう
  - そのプロセスにはずっと割り当てられない
    →待ちっぱなし

デッドロックとは、資源を割り付けられている複数のプロセスが、相互に他のプロセスに割り付けられている資源を要求して、ブロックしたままになることを指します。

一方、飢餓状態は以前も出てきましたが、あるプロセスが割り付けを要求している資源が、常に他のプロセスに割り付けられ続けることを指し、そのプロセスにはずっと割り付けられるのを待ちっぱなしとなります。

デッドロックとの違いは、待ち続けるプロセスは資源の割り付け待ちをするのみですが、デッドロックだとお互いに相手のプロセスに割り付けられた資源を要求しながら、自分に割り付けられた資源は解放せずに持ち続けることにあります。

10

テストアンドセットによる相互排除

- テストアンドセット：1ビットのフラグによってクリティカルセクションに対する相互排除を実現する
- 不可分に行なうマシン命令（TS命令）を利用（プロセッサのクロックの不可分性を利用）
  - テスト：フラグの値が0（空き）か1（使用中）かチェック
  - セット：フラグを1（使用中）に設定

使用例:

```
LOCK：TS ENTER
      BNZ   LOCK
     <critical section>
UNLOCK: MVI   ENTER, '00'
```

相互排除を実現するのに、古典的な方法としては、テストアンドセットがあります。

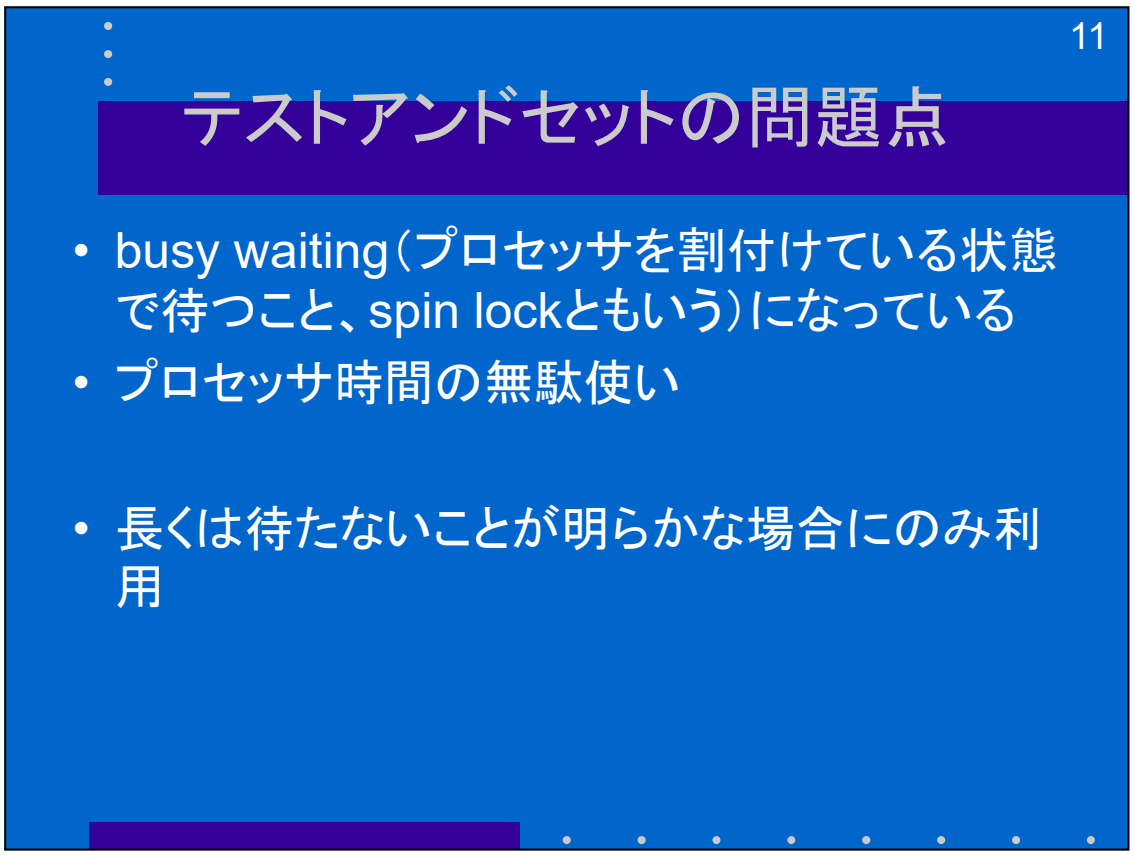
これは、1ビットのフラグによってクリティカルセクションに対する相互排除を実現することを指します。

ここでは、以下の不可分に行なうマシン命令（TS命令）を利用します。

テスト：フラグの値が0（空き）か1（使用中）かチェック

セット：フラグを1（使用中）に設定

これはプロセッサのクロックの不可分性を利用しています。



テストアンドセットの問題点は、busy waiting（プロセッサが割り付けられている状態でループして待つこと、spin lockともいう）になっていることです。
これだと、プロセッサ時間の無駄使いとなります。

このため、長くは待たないことが明らかな場合にのみ利用されます。

12

セマフォ

- 同期問題を解決する道具の一つ
  - 1968年E.W.Dijkstraが考案
- 整数型変数 とそれに対する手続き
- signal(semaphore)とwait(semaphore)操作
  - P(semaphore)とV(semaphore)操作と言う表記もある（教科書はこれで表記されている）
  - P: Paseren、V: Verhoog(オランダ語)
- セマフォ:　鉄道の腕木信号機

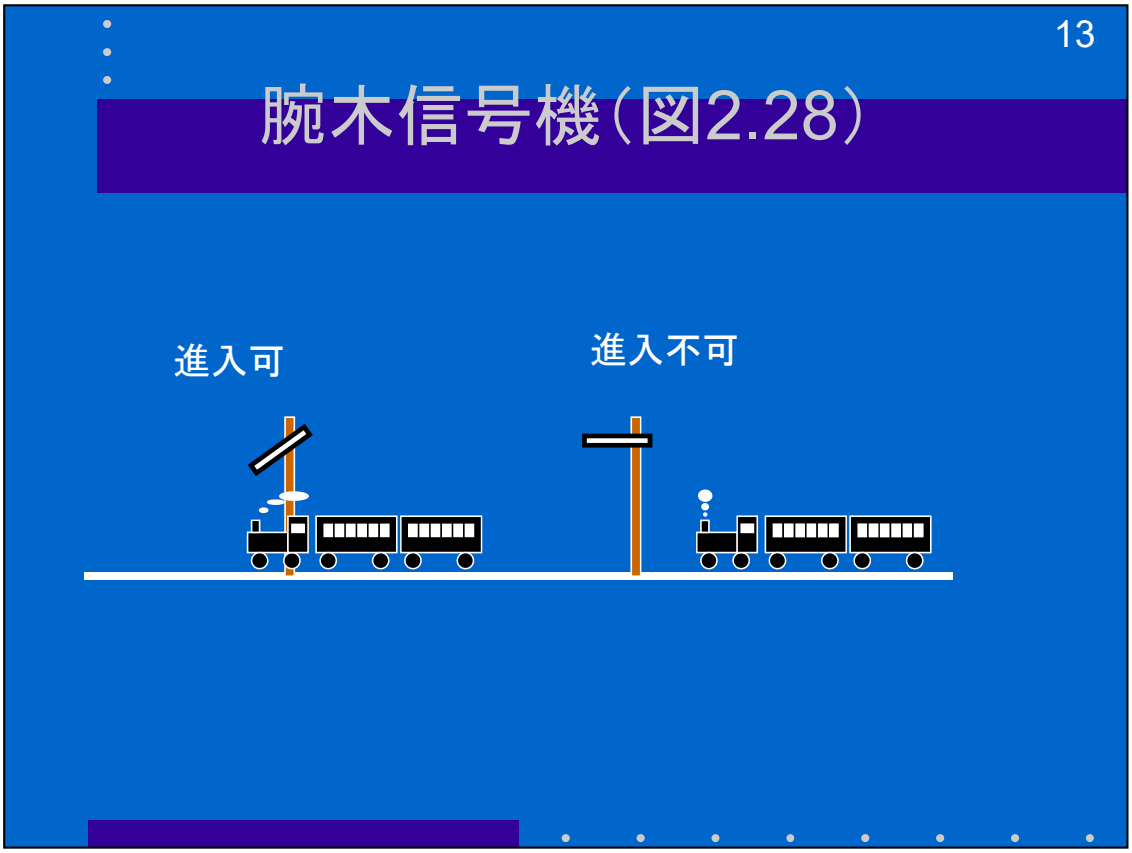
他の方法として、「セマフォ」があります。

セマフォは、同期問題を解決する道具の一つです。
1968年E.W.Dijkstraが考案しています。

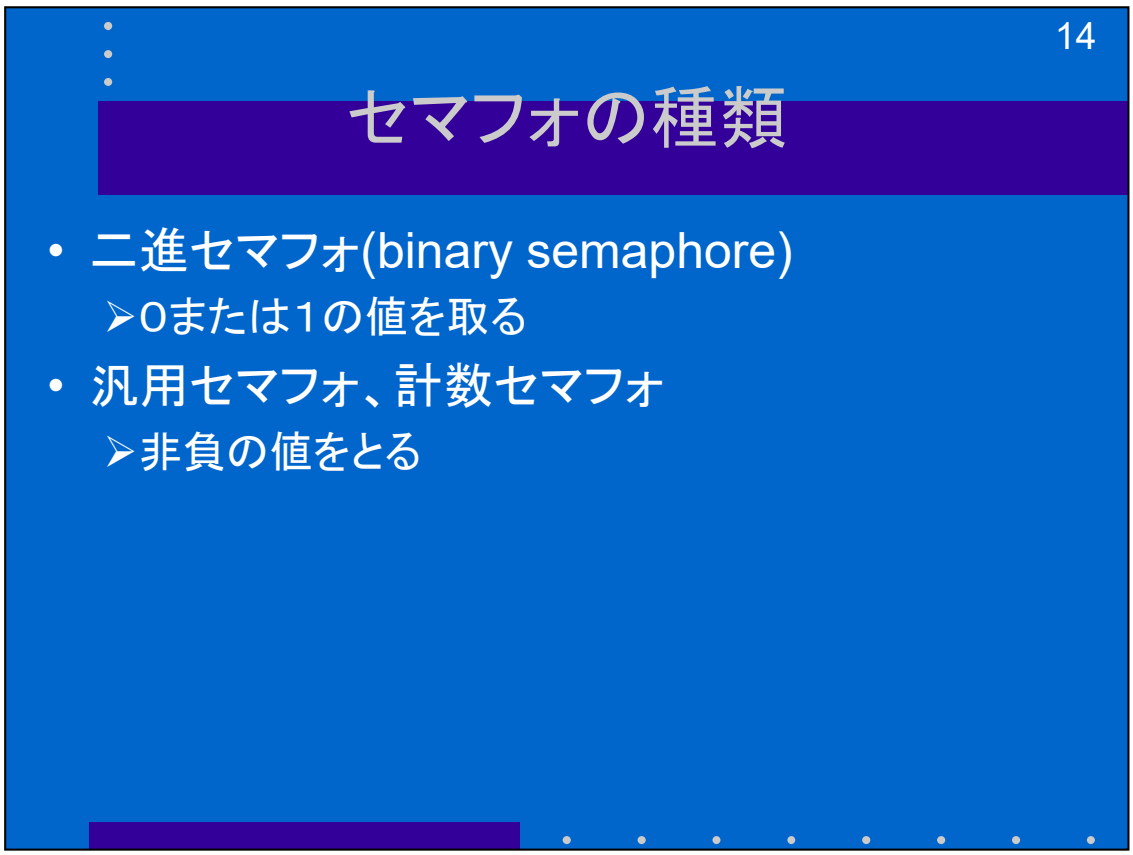
整数型変数 とそれに対する手続きとなっていて、
signal(semaphore)とwait(semaphore)操作が使われます。

ここで、P(semaphore)とV(semaphore)操作と言う表記もあります（教科書はこれで表記されています）
P: Paseren、V: Verhoog(オランダ語)を指します。

セマフォとは鉄道の腕木信号機のことです。



腕木信号機の例です。



セマフォには次のような種類があります。

二進セマフォ(binary semaphore)、0または1の値を取る
汎用セマフォ、計数セマフォ、非負の値をとる

15

セマフォの操作（図2.29, 図2.30）

- **wait(semaphore)**
  - ➢semaphoreの値が正なら値を1減らす
    - 次の文へ
  - ➢0なら、ブロック（待ち状態のキューにつながれる）
- **signal(semaphore)**
  - ➢waitでブロックしているプロセスがあるとき
    - ブロックしているプロセスの実行を再開する（実際には、実行可能キューにつなぐ）
  - ➢ないとき
    - semaphoreの値を1増やす

セマフォには、次の2種類の操作ができます。

wait(semaphore)

semaphoreの値が正なら値を1減らす

次の文へ

0なら、ブロック（待ち状態のキューにつながれる）

signal(semaphore)

waitでブロックしているプロセスがあるとき

ブロックしているプロセスの実行を再開する（実際には、実行可能キューにつなぐ）

ないとき

semaphoreの値を1増やす

16

セマフォを用いた スタックのプログラム例

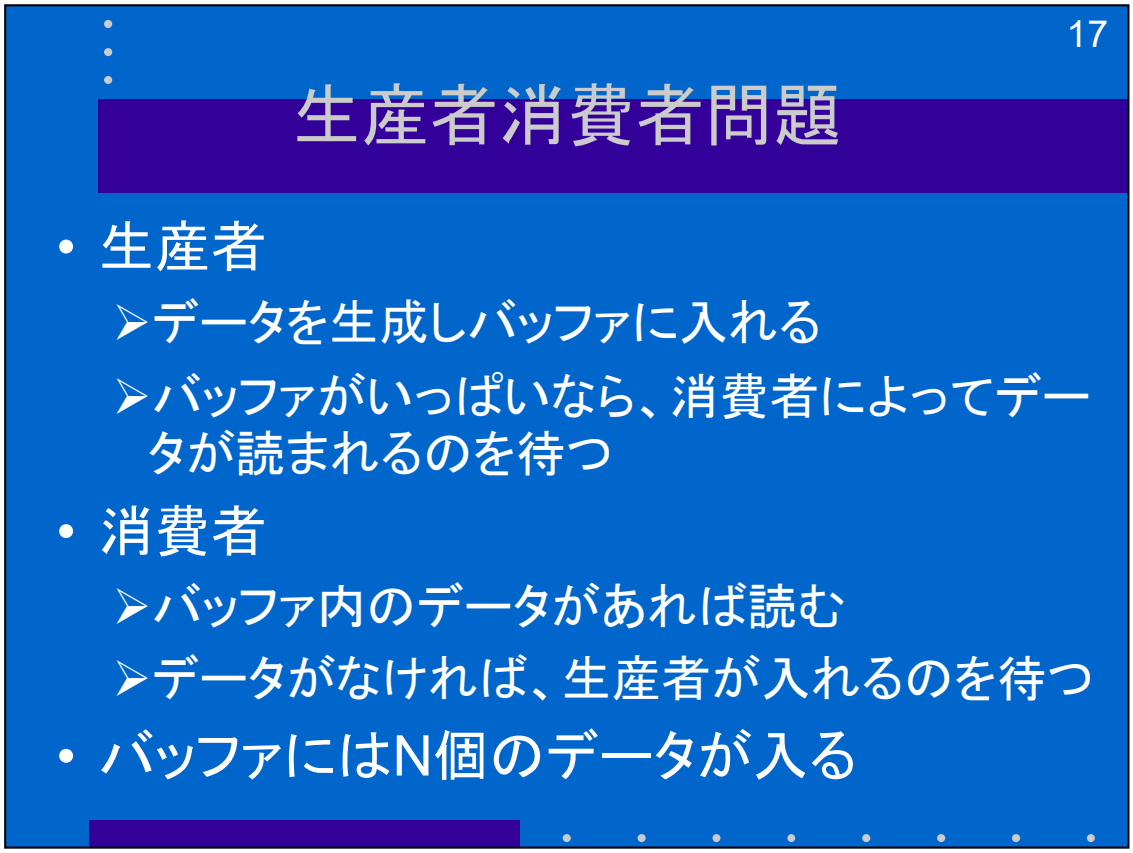
- semaphoreの初期値は1
- push操作

```
wait(semaphore);
top=top-1;
stack(top)=item;
signal(semaphore);
```

pop操作

```
wait(semaphore);
item=stack(top);
top=top+1;
signal(semaphore);
```

セマフォを用いた、スタックプログラムの実装の例は上のようになります。

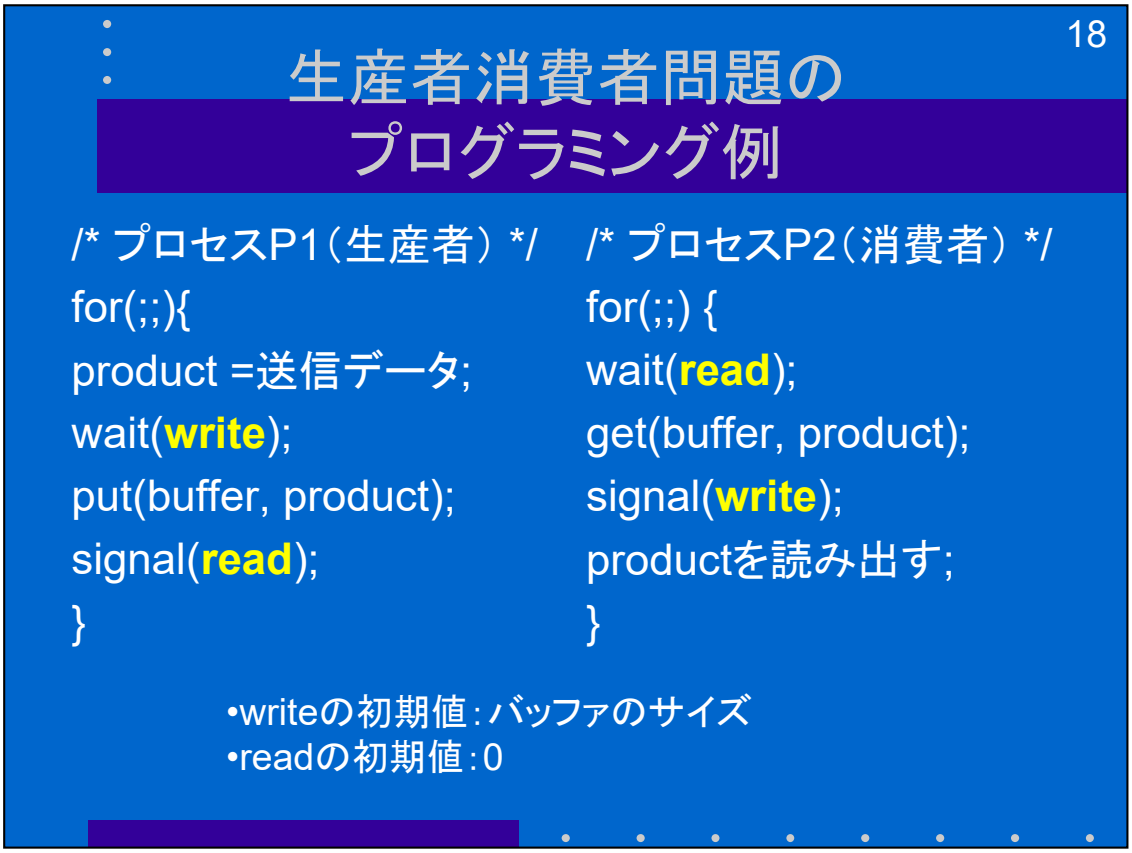


単純な相互排除の実現以外に、セマフォでは、生産者消費者問題を解くことができます。

これは、以下の2つの操作を実現することです。

生産者:
　データを生成しバッファに入れる
　バッファがいっぱいなら、消費者によってデータが読まれるのを待つ
消費者
　バッファ内のデータがあれば読む
　データがなければ、生産者が入れるのを待つ

なお、バッファにはN個のデータが入るとします。



生産者消費者問題のプログラミング例をあげておきます。

この例のように、このプログラムでは、2個のセマフォを使用して実現しています。

write は書き込み用のセマフォであり、正の値であればバッファに書き込んで、値を1減らし、0なら書き込まずにブロックします。

read は読み出し用のセマフォであり、正の値であればバッファから読み出して（読んだデータはバッファから取り除かれる）、値を1減らし、0なら読み出さずにブロックします。

セマフォの初期値は、writeはバッファのサイズ（その数だけバッファに書き込める）、readは0（バッファにデータがないので読み出せない）となっています。

19

例題１

/* プロセスP1（生産者）*/	/* プロセスP2（消費者）*/
product =送信データ;	wait(read);
wait(write);	product = buffer;
buffer=product;	signal(write);
signal(read);	productを読み出す;

writeとreadはセマフォであり、bufferはP1とP2で共有されている変数とする（初期値は、write=1, read=0）
プログラムの実行の順番はどのようになるか？（実行可能キューの先頭に**P2**, その次に**P1**がつながれていたとする）

生産者、消費者問題の例題を示します。

writeとreadはセマフォであり、bufferはP1とP2で共有されている変数とします（初期値は、write=1, read=0）

この例だと、プログラムの実行の順番はどのようになるでしょうか？（実行可能キューの先頭にP2, その次にP1がつながれていたとします）

例題1の解答

/* プロセスP1（生産者） */	/* プロセスP2（消費者） */
②product =送信データ;	①wait(read);
③wait(write);	⑥product = buffer;
④buffer=product;	⑦signal(write);
⑤signal(read);	⑧productを読み出す;

- P2の①を実行（readの値が0のため、wait操作によりP2はブロック）し、P1に切り替わる
- P1の②、③を実行（writeの値が1のため、writeを0にして実行継続）し、④、⑤を実行（signal操作により、P2を実行可能キューにつなぐ）した後、P1は終了
- P2の⑥から実行を再開し、⑦を実行（writeの値を1に）した後、⑧を実行して終了

例題1の解答は、このスライドの通りです。

21

例題2

/* プロセスP1（生産者）*/	/* プロセスP2（消費者）*/
product =送信データ;	wait(read);
wait(**read**);	product = buffer;
buffer=product;	signal(write);
signal(**write**);	productを読み出す;

例題1のプログラムで、P1のreadとwriteを逆に書いたらどうなるか？

次に、先ほどの例題を一部アレンジした、例題2を考えます。

つまり、例題2では、例題1のプログラムで、P1のreadとwriteを逆に書いたらどうなるか？
ということです。

例題2の解答例

/* プロセスP1（生産者）*/	/* プロセスP2（消費者）*/
②product =送信データ;	①wait(read);
③wait(**read**);	product = buffer;
buffer=product;	signal(write);
signal(**write**);	productを読み出す;

- P2の①を実行（readの値が0のため、P2はブロック）し、P1に切り替わる
- P1の②、③を実行（readの値が0のため、P1はブロック）
- P1,P2が共にブロックしてしまい、プロセスの実行は停止（**デッドロック**）

例題2の解答例はこのようになります。

この通りで実行すると、P1,P2が共にブロックしてしまい、プロセスの実行は停止（デッドロック）となります。

2.2.3 プロセス間通信

- 同期(synchronization)
  - 複数のプロセスが、それぞれのプログラムの特定の箇所で実行を揃えること（その箇所に早く到着したプロセスは待ち状態になる）
- バッファの有無で送信側プロセスと受信側プロセスが同期を取るかどうかが変わる（図2.31）
- バッファなし
  →受信側プロセスの準備ができるまで送信は**待たされる**（**同期**通信）
- バッファあり
  →受信側プロセスの実行の状況に関係なく送信は**完了する**（**非同期**通信）

プロセス間通信とは、セマフォを使うと次のようになります。

同期(synchronization)

複数のプロセスが、それぞれのプログラムの特定の箇所で実行を揃えること（その箇所に早く到着したプロセスは待ち状態になる）

バッファの有無で送信側プロセスと受信側プロセスが同期を取るかどうかが変わります

バッファなし

・受信側プロセスの準備ができるまで送信は待たされます（同期通信）

バッファあり

・受信側プロセスの実行の状況に関係なく送信は完了します（非同期通信）

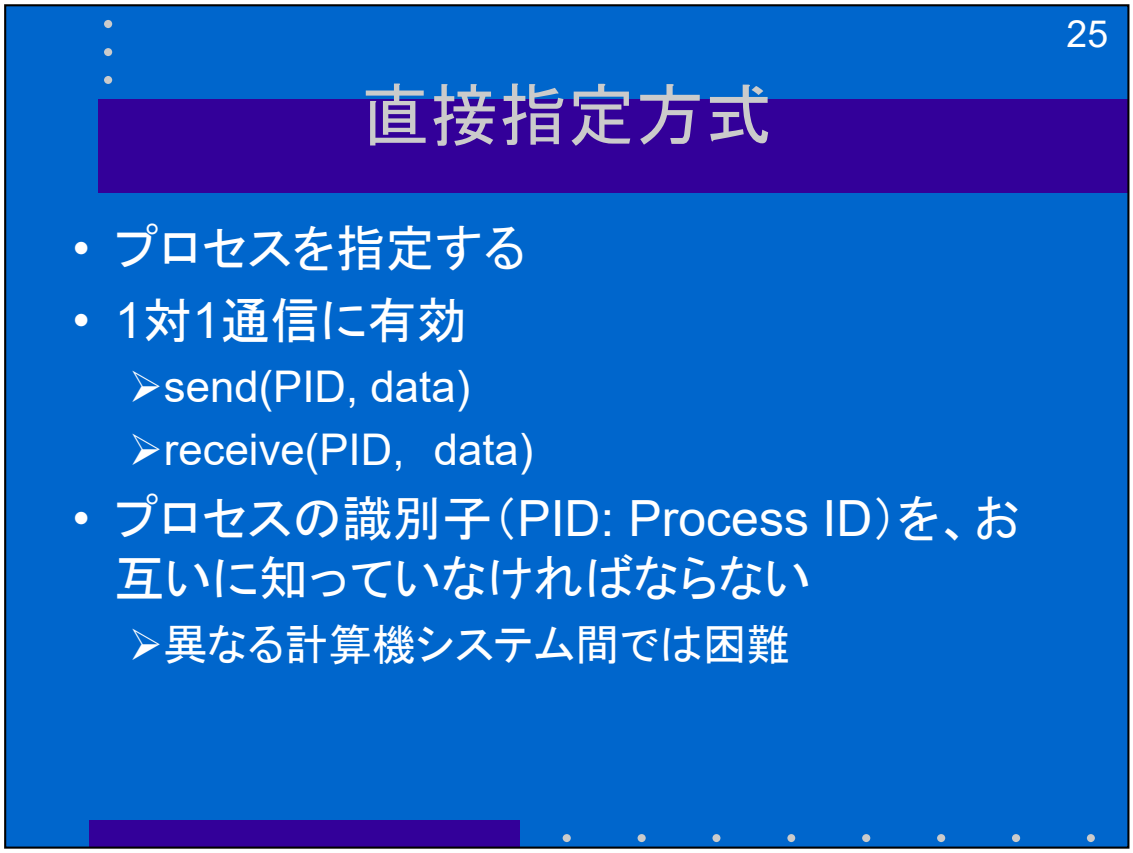
24

通信の形

- センドーレシーブ（図2.32）
  - 送受信されるデータのこと（一般的にはバイト列）
  - send メッセージリスト to 宛先
  - receive 変数リスト from 送り元
- メッセージ通信の基本
  - 送受信相手の指定の形態
  - 同期をどう行なうか
- 通信チャネル: 双方の相手指定をまとめる

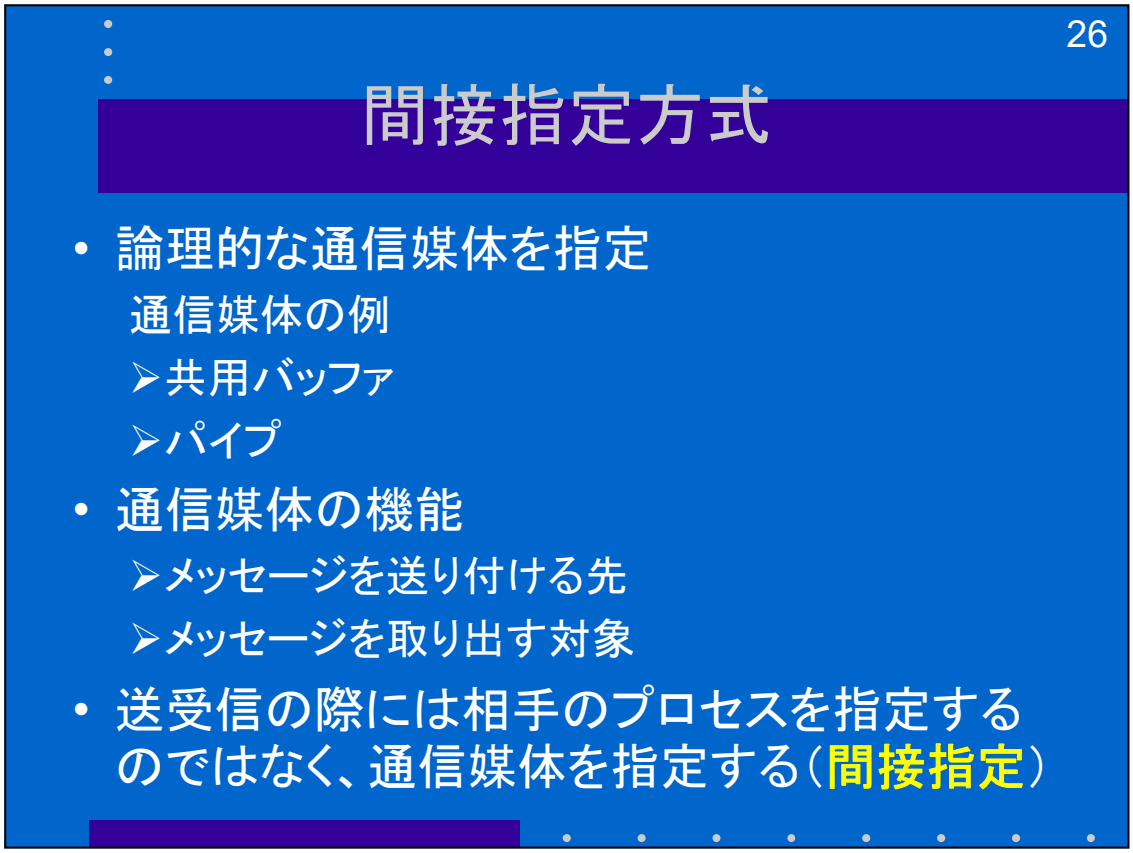
このような相互排除の実現の他に、プロセス間通信があります。

これの基本は、センドーレシーブと呼ばれます。



直接指定方式とは、文字通り、直接、プロセスを指定する方式です。
1対1通信に有効です。

プロセスの識別子（PID: Process ID）を、お互いに知っていなければなりません。
これは、異なる計算機システム間では困難です。



間接指定方式では、論理的な通信媒体を指定します。
通信媒体の例としては、共用バッファ、パイプがあります。

通信媒体の機能としては、
メッセージを送り付ける先、
メッセージを取り出す対象、
があります。

送受信の際には相手のプロセスを指定するのではなく、通信媒体を指定する（間接指定）のが一般的です。

27

パイプ

- 2個のプロセスをつないだ通信チャネル
- write操作と read操作が行える
  - 対象は パイプ(の端点)
  - パイプの端点はプロセスにとって「ローカル」
  - 他端がどのプロセスにつながっているかを意識しなくてよい
- あらかじめパイプを宣言しておかなければ使えない
- UNIXのパイプの語源

通信媒体としてのパイプとは、2個のプロセスをつないだ通信チャネルを指します。

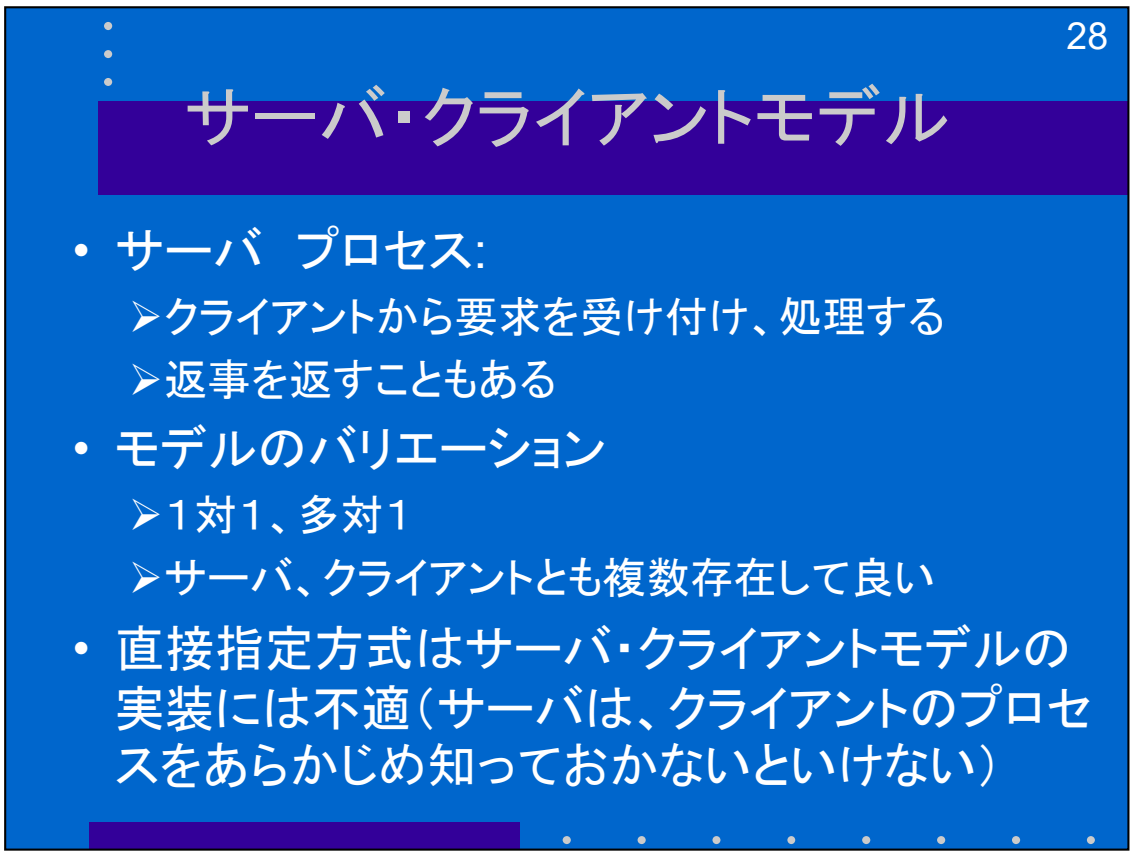
write操作と read操作が行えます。

対象は パイプ(の端点)であり、パイプの端点はプロセスにとって「ローカル」となります。

つまり、他端がどのプロセスにつながっているかを意識しなくてよいです。

しかし、あらかじめパイプを宣言しておかなければ使えません。

これは、UNIXのパイプの語源です。

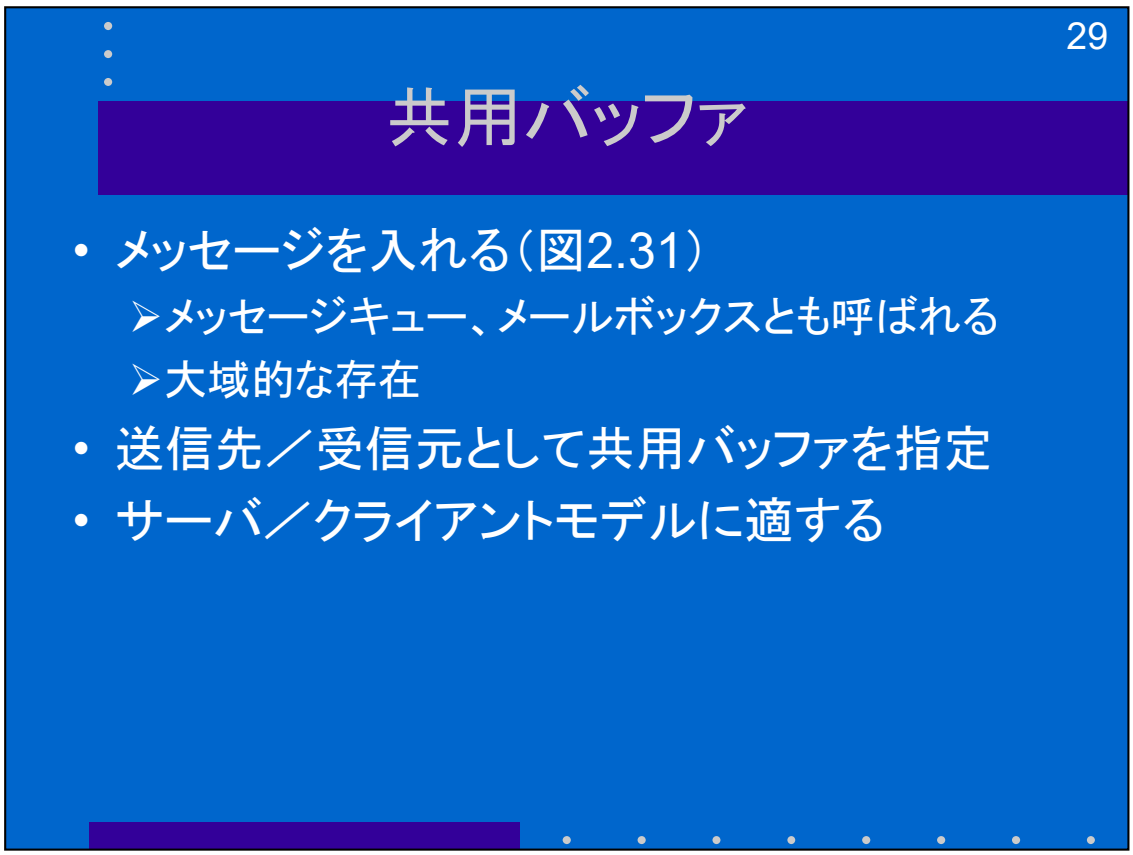


別の方式として、サーバ・クライアントモデルがあります。

サーバ　プロセスは、クライアントから要求を受け付け、処理し、返事を返すこともあります。

モデルのバリエーションは、1対1、多対1があります。
サーバ、クライアントとも複数存在して良いなどがあります。

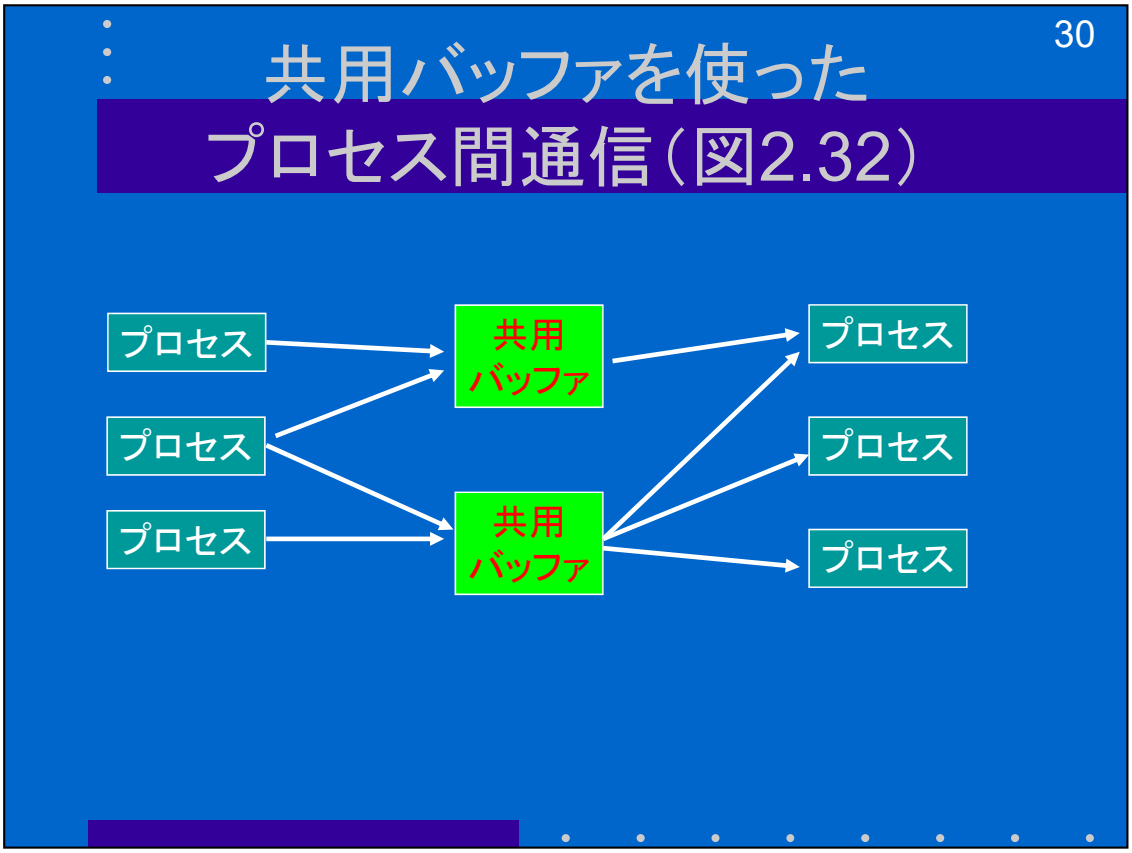
直接指定方式はサーバ・クライアントモデルの実装には不適（サーバは、クライアントのプロセスをあらかじめ知っておかないといけない）となります。



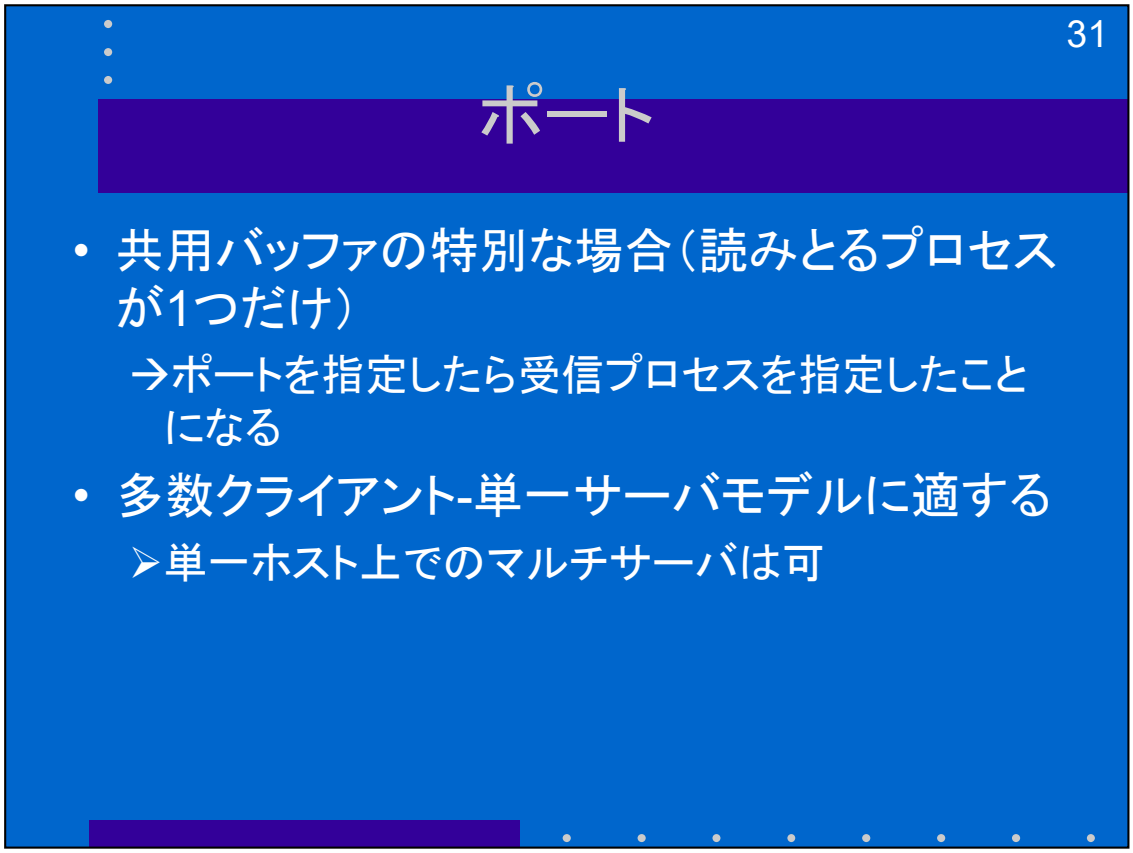
他の通信媒体に、共用バッファがあります。

これは、メッセージを入れるためのものです。
メッセージキュー、メールボックスとも呼ばれます。
大域的な存在となります。

送信先／受信元として共用バッファを指定します。
サーバ／クライアントモデルに適しています。



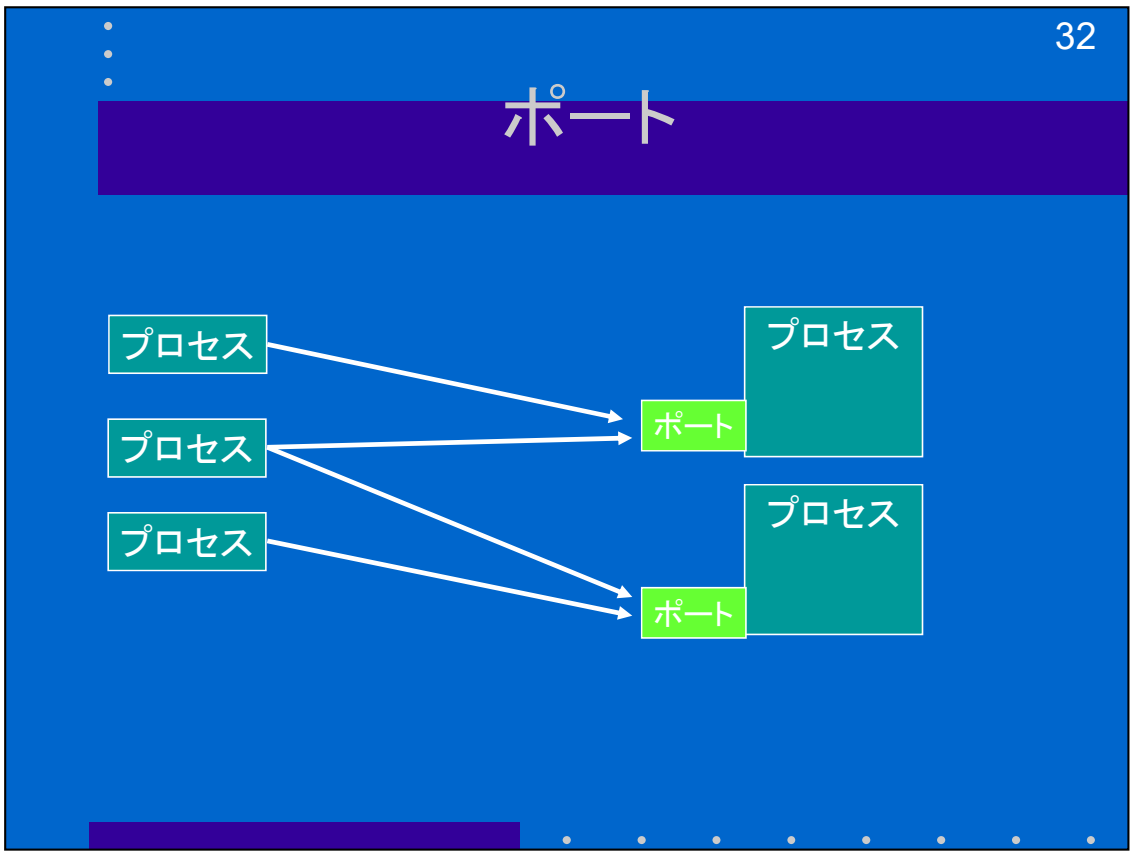
共用バッファを使ったプロセス間通信の例をあげておきます。



最もよく使われている通信媒体に、ポートがあります。

これは、共用バッファの特別な場合（読みとるプロセスが1つだけ）とみなせます。
ポートを指定したら受信プロセスを指定したことになります。

多数クライアント−単一サーバモデルに適しています。
単一ホスト上でのマルチサーバは可となります。



ポートによる通信の例をあげておきます。

33

2.2.4 デッドロック

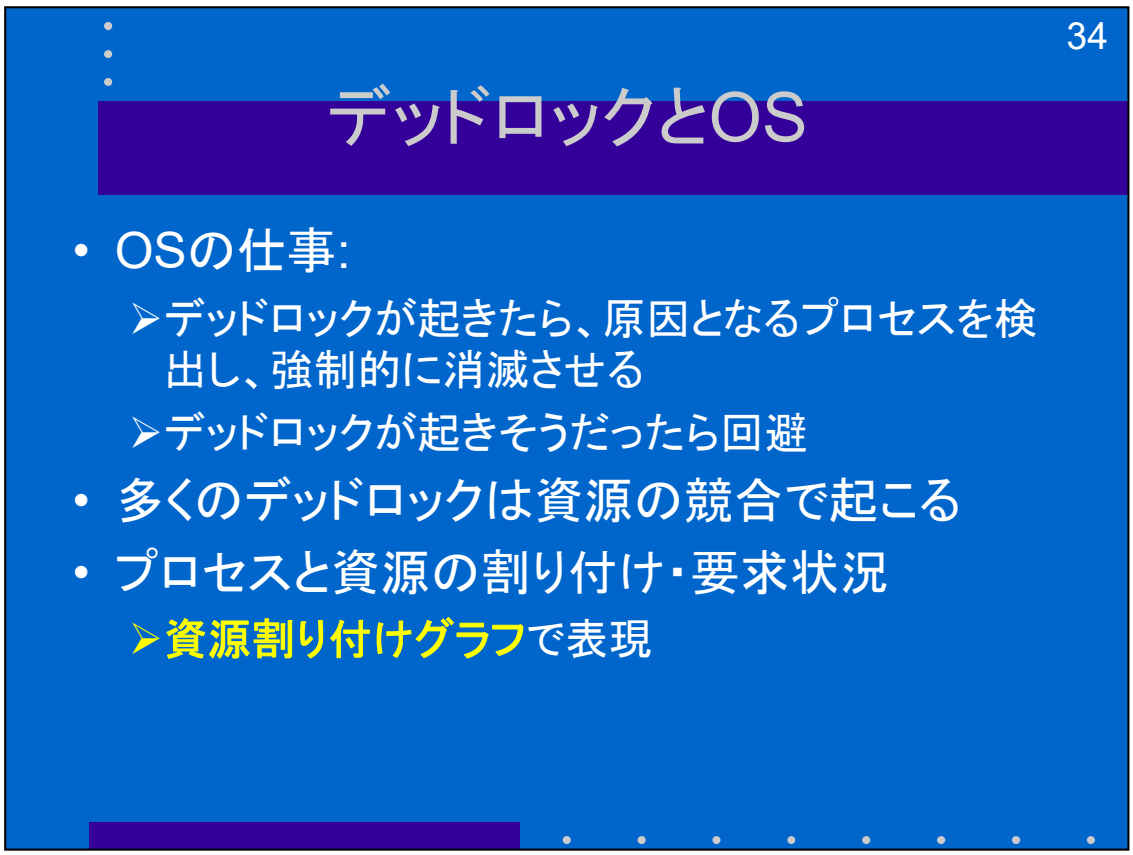
- すべてのプロセスが他のプロセスによる事象の生起を待つ待ち状態になり、どのプロセスも実行できなくなったプロセッサ状態のこと
- プロセスと割り付け待ち資源の関係が循環していると、どのプロセスも実行されない
- デッドロックの例

	資源1	資源2
プロセスP1	使用中	割り付け待ち
プロセスP2	割り付け待ち	使用中

デッドロックでは、すべてのプロセスが他のプロセスによる事象の生起を待つ待ち状態になり、どのプロセスも実行できなくなったプロセッサ状態のことを指します。

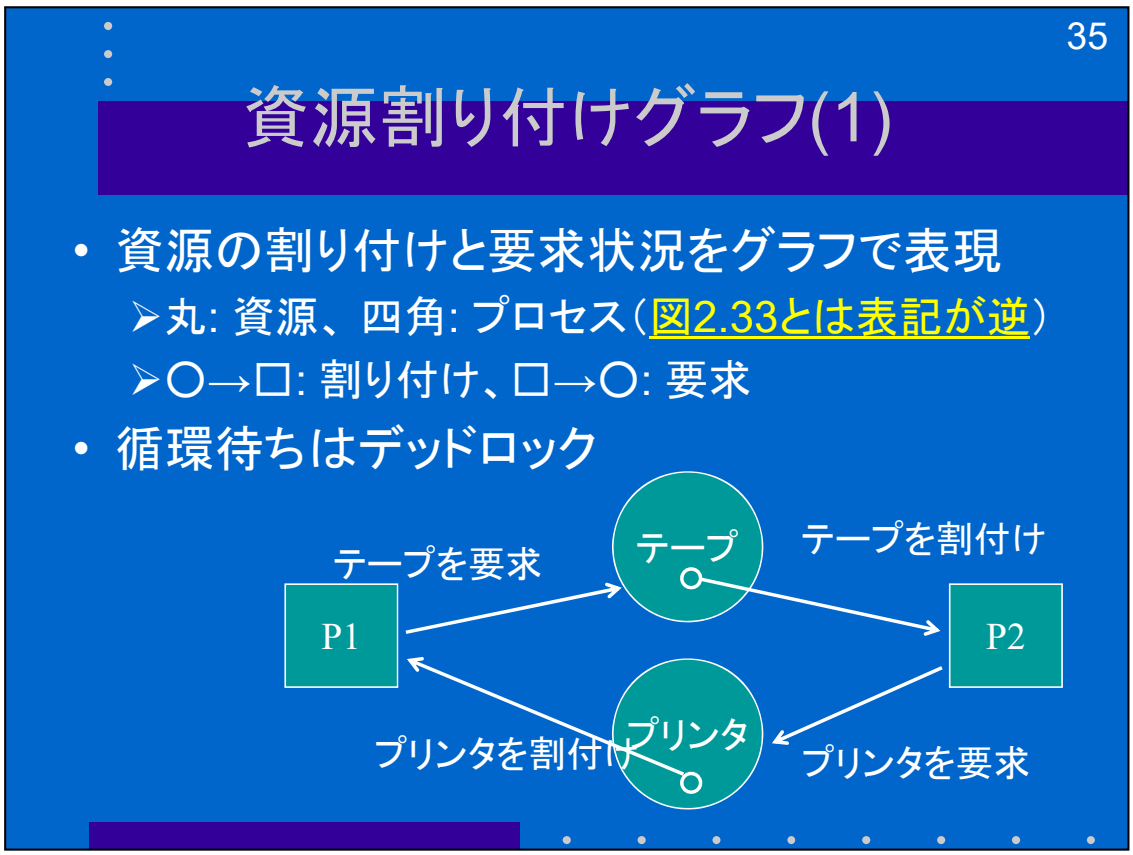
プロセスと割り付け待ち資源の関係が循環していると、どのプロセスも実行されなくなります。

これが、デッドロックであり、上のような例があります。



OSの仕事に、デッドロックが起きたら、原因となるプロセスを検出し、強制的に消滅させることがあります。
しかし、それよりも、デッドロックが起きそうだったら回避する方が有効です。

多くのデッドロックは資源の競合で起こるので、
プロセスと資源の割り付け・要求状況を把握することが重要となります。
これは、資源割り付けグラフで表現できます。

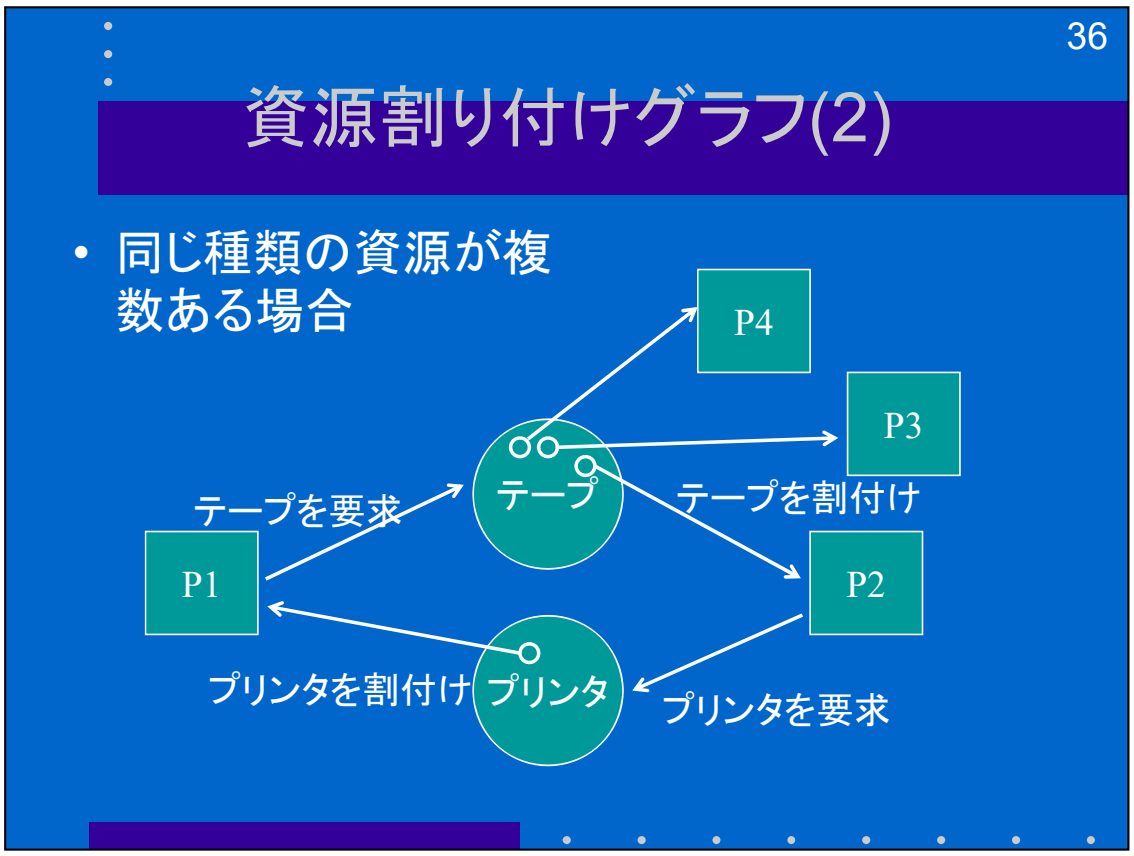


資源割り付けグラフとは、資源の割り付けと要求状況をグラフで表現したものです。

丸: 資源、四角: プロセス(教科書の図2.33とは表記が逆になっています)

〇→□: 割り付け、□→〇: 要求

循環待ちはデッドロックとなります。

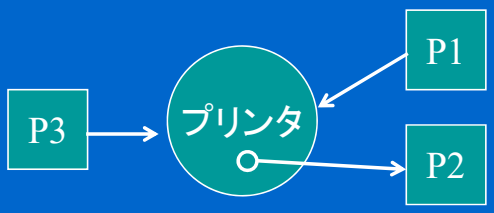


この例は、同じ種類の資源が複数ある場合を指します。

37

飢餓状態

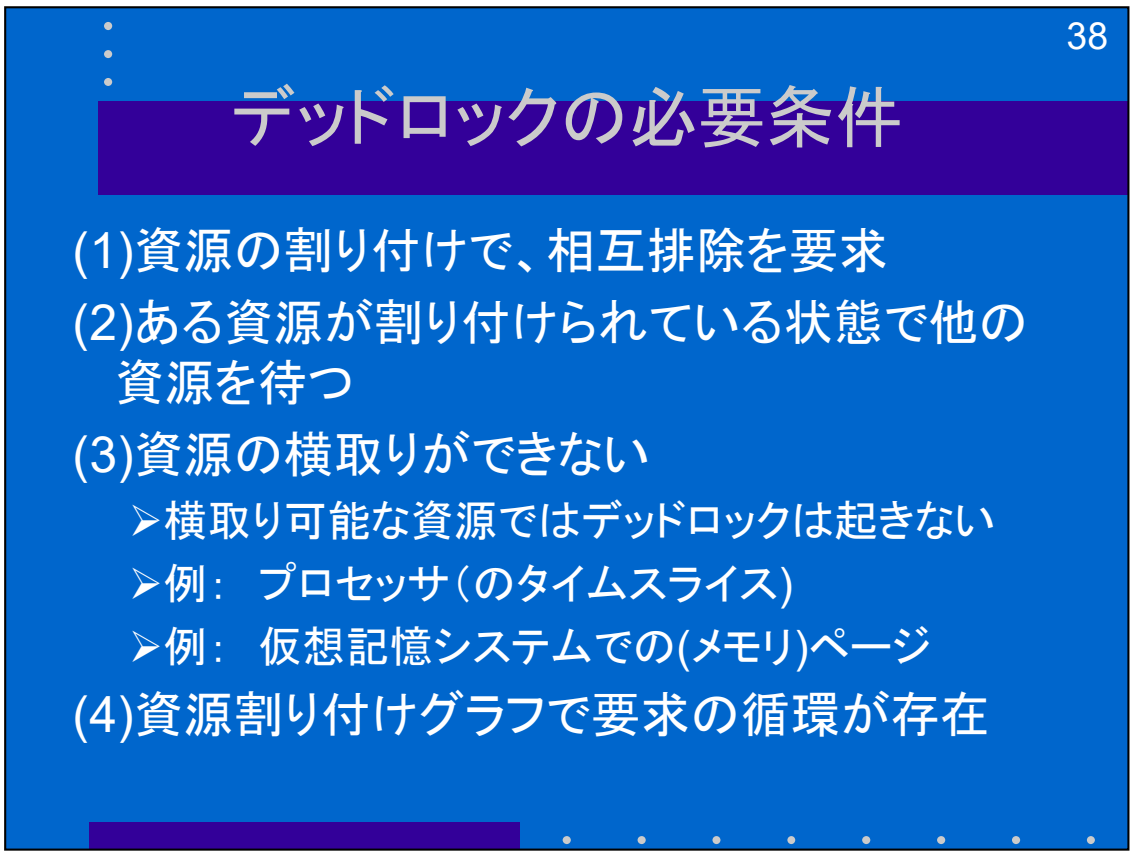
- 待っている対象の資源は解放されるが、常に別のプロセスが取ってしまう
  （例：プロセスP1とP2に交互にプリンタが割付けられ、P3が要求し続けてもプリンタが割付けられない）
- 対策例
  待ち時間に応じて、
  プロセスへの資源割り付け
  の優先度を上げる
  （**エージング**(aging)）



飢餓状態とは、待っている対象の資源は解放されるが、常に別のプロセスが取ってしまうことを指します

（例：プロセスP1とP2に交互にプリンタが割付けられ、P3が要求し続けてもプリンタが割付けられない）

対策例は、待ち時間に応じて、プロセスへの資源割り付けの優先度を上げる（エージング(aging)）です。



デッドロックが起こるのに必要な条件は以下です。

(1)資源の割り付けで、相互排除を要求
(2)ある資源が割り付けられている状態で他の資源を待つ
(3)資源の横取りができない
　　横取り可能な資源ではデッドロックは起きない
　　例： プロセッサ（のタイムスライス）
　　例： 仮想記憶システムでの(メモリ)ページ
(4)資源割り付けグラフで要求の循環が存在

39

デッドロックの防止手法

(1) 相互排除を要求しない

× 欠点： 相互排除が必要なときがある

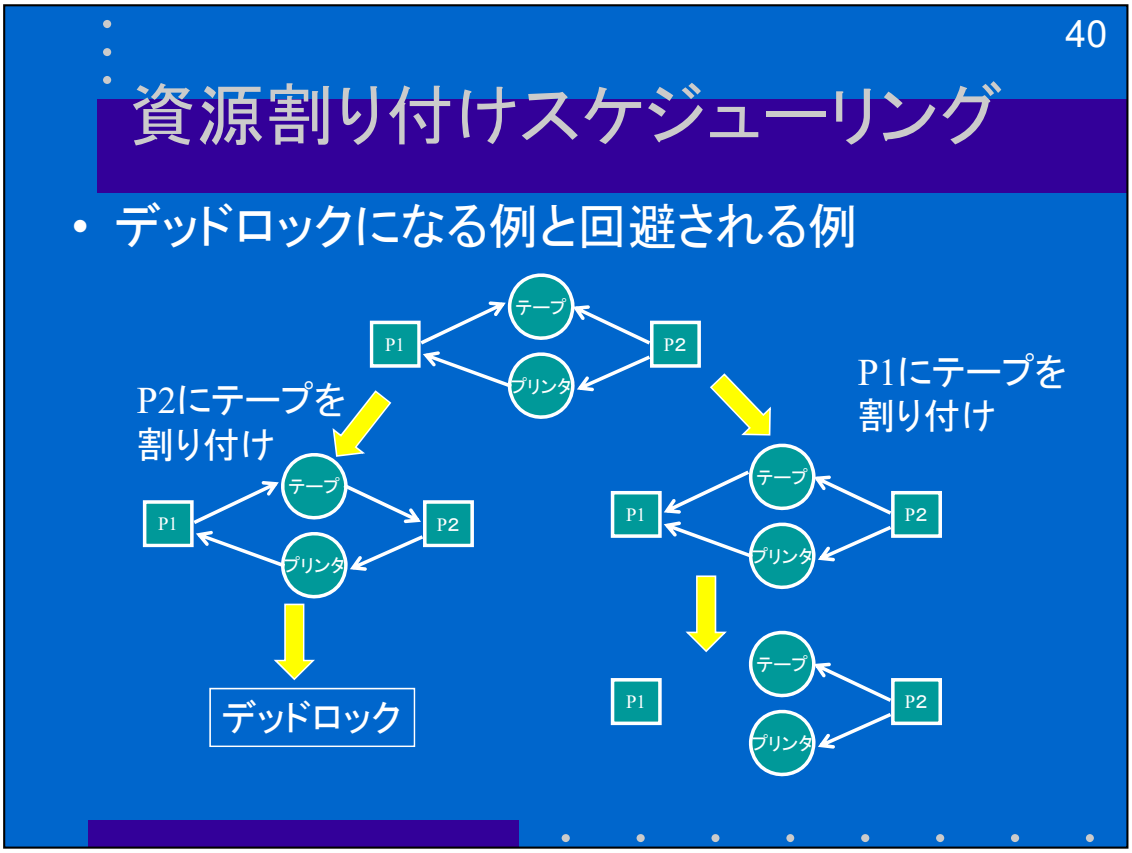
(2) **待ちの防止**：必要な資源は全部をまとめて要求する

(3) **横取りなしの禁止**：資源を要求して一部しか割り付けられなかったら、待ち状態に入るのではなく、既に割り付けられた資源をいったん解放する

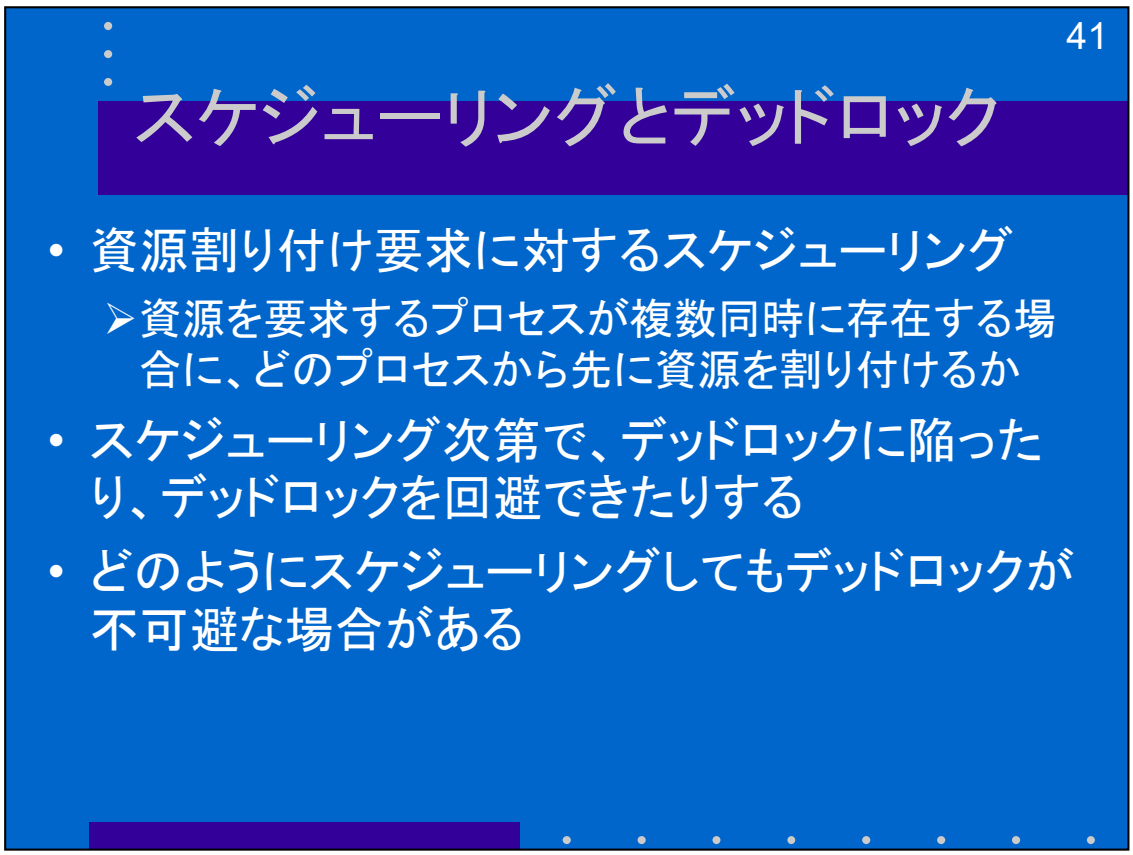
(4) **循環待ちの防止**：資源の型に線形順序をつけ、その順にしか要求しないようにする

デッドロックを防止するには、前のスライドの条件が発生しないようにすることになります。

具体例をここにあげておきます。



同じ、資源割り付けグラフから出発しても、デッドロックになる例と回避される例があることに注意が必要です。



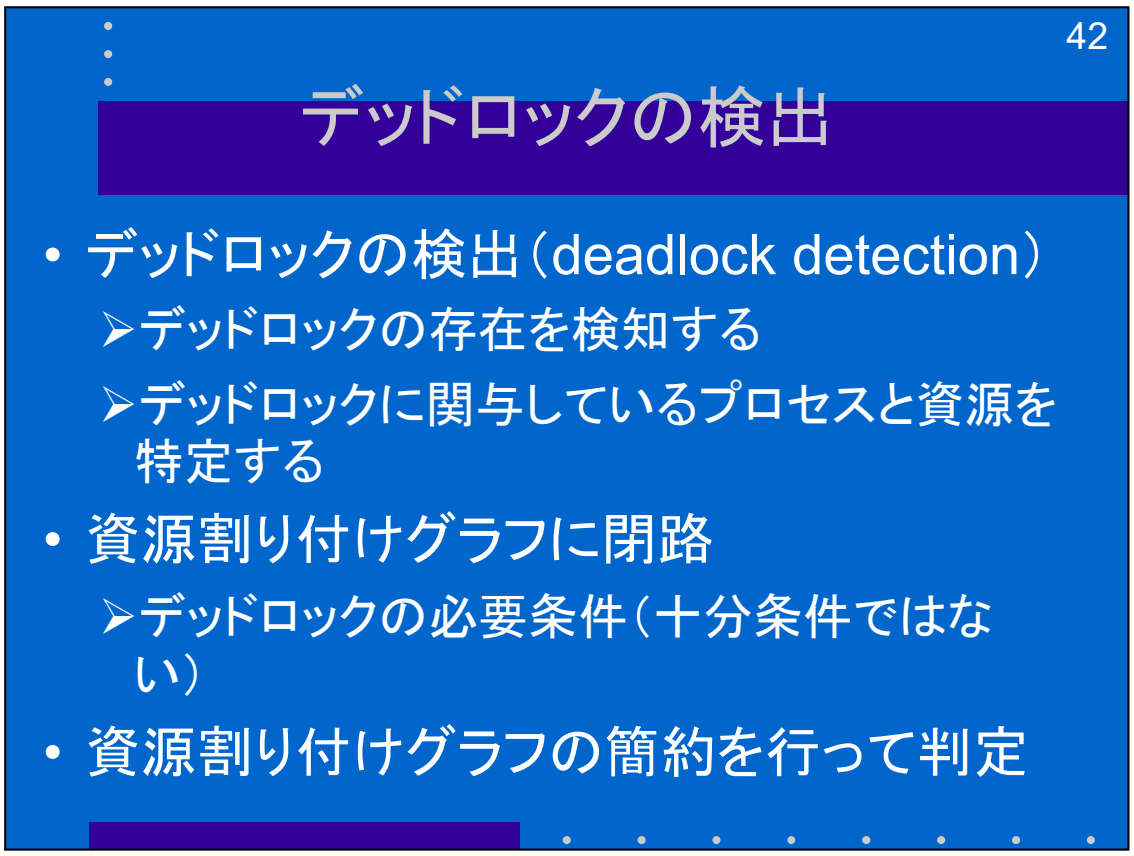
つまり、デッドロックが起こるかどうかは、スケジューリングが関係します。

資源割り付け要求に対するスケジューリング

資源を要求するプロセスが複数同時に存在する場合に、どのプロセスから先に資源を割り付けるかとなります。

スケジューリング次第で、デッドロックに陥ったり、デッドロックを回避できたりします。

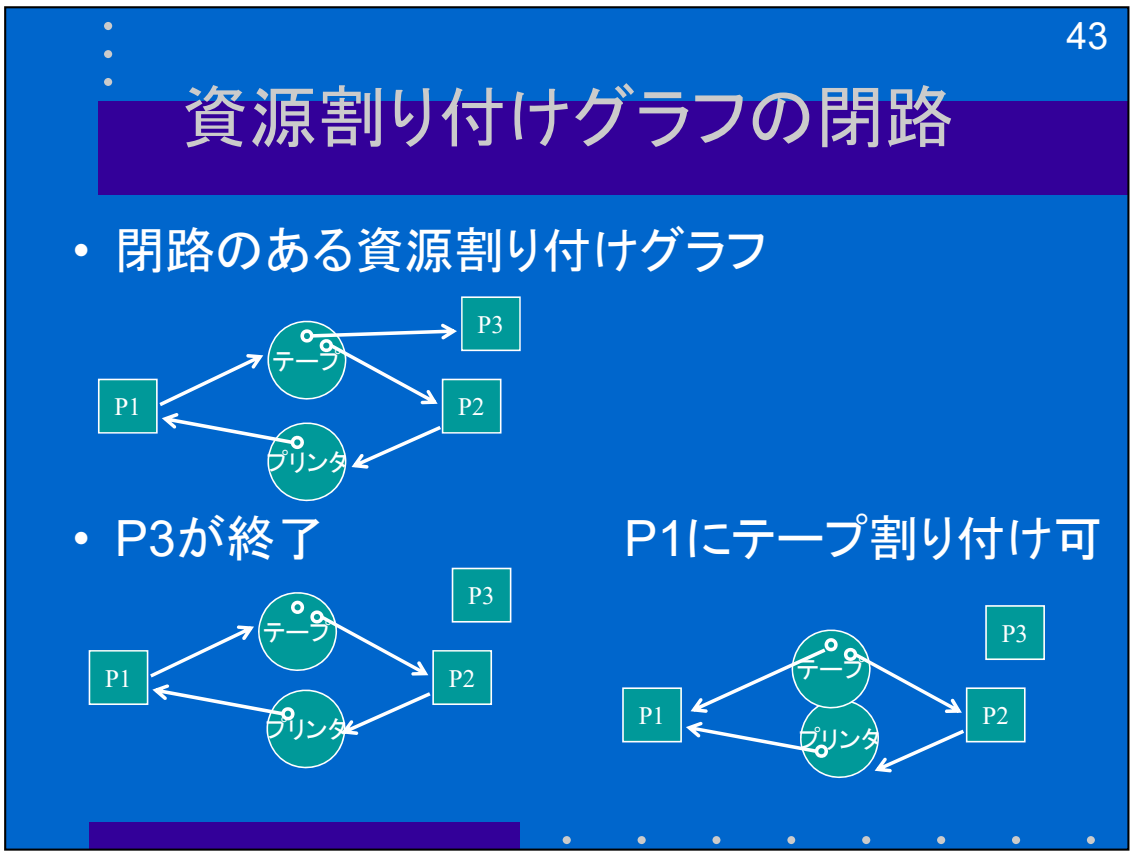
一方で、どのようにスケジューリングしてもデッドロックが不可避な場合があります。



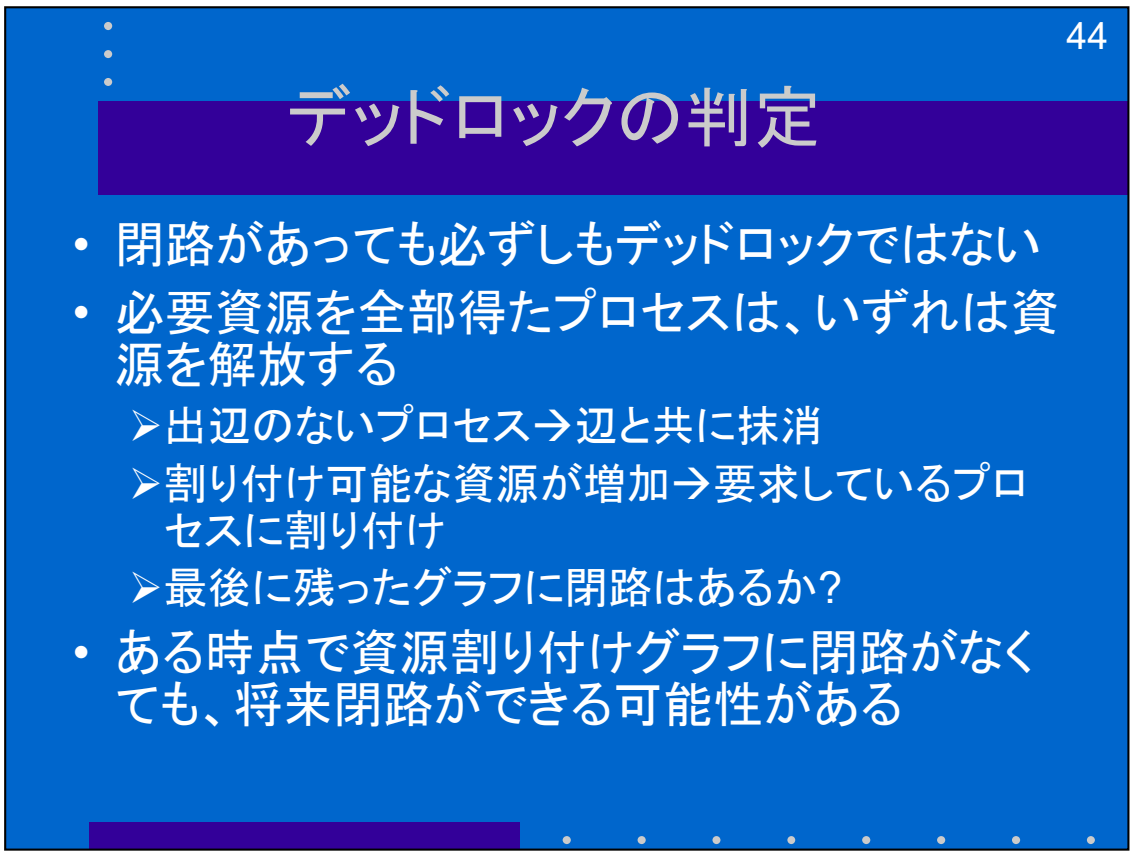
デッドロックの検出（deadlock detection）とは、デッドロックの存在を検知し、デッドロックに関与しているプロセスと資源を特定することにあります。

資源割り付けグラフに閉路があることは、デッドロックの必要条件（十分条件ではない）となります。

これは、資源割り付けグラフの簡約を行って判定します。



閉路のある資源割り付けグラフの例を示します。



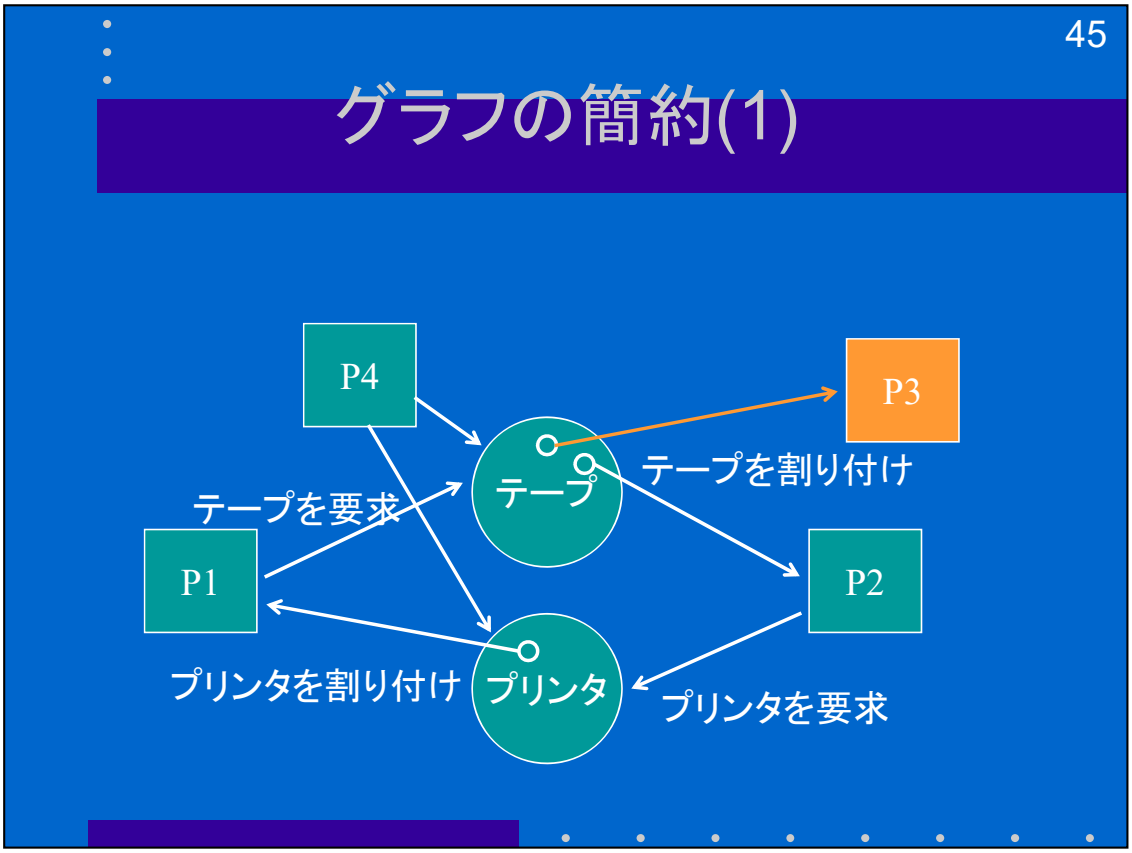
デッドロックかどうかを判定するには、閉路があっても必ずしもデッドロックではないことに注意が必要です。

必要資源を全部得たプロセスは、いずれは資源を解放します。

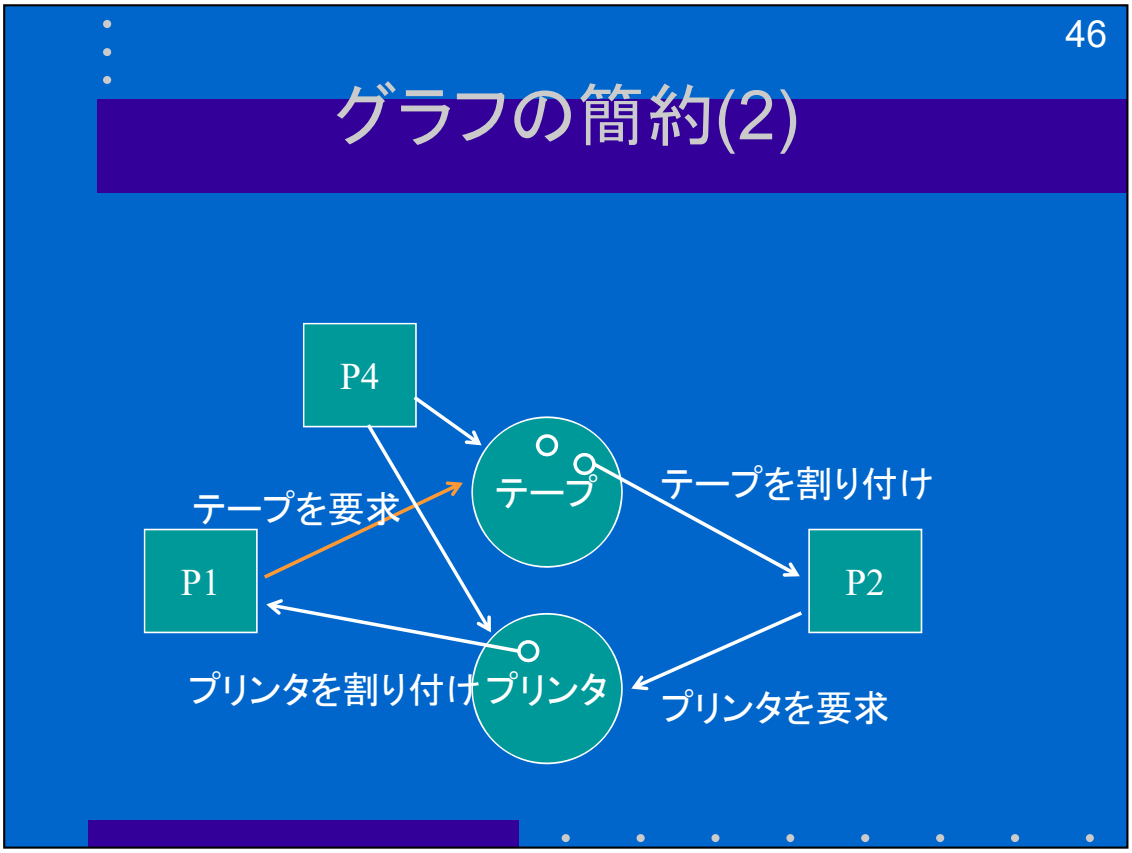
グラフで、出辺のないプロセス・辺と共に抹消し、割り付け可能な資源が増加していれば、要求しているプロセスに割り付けます。

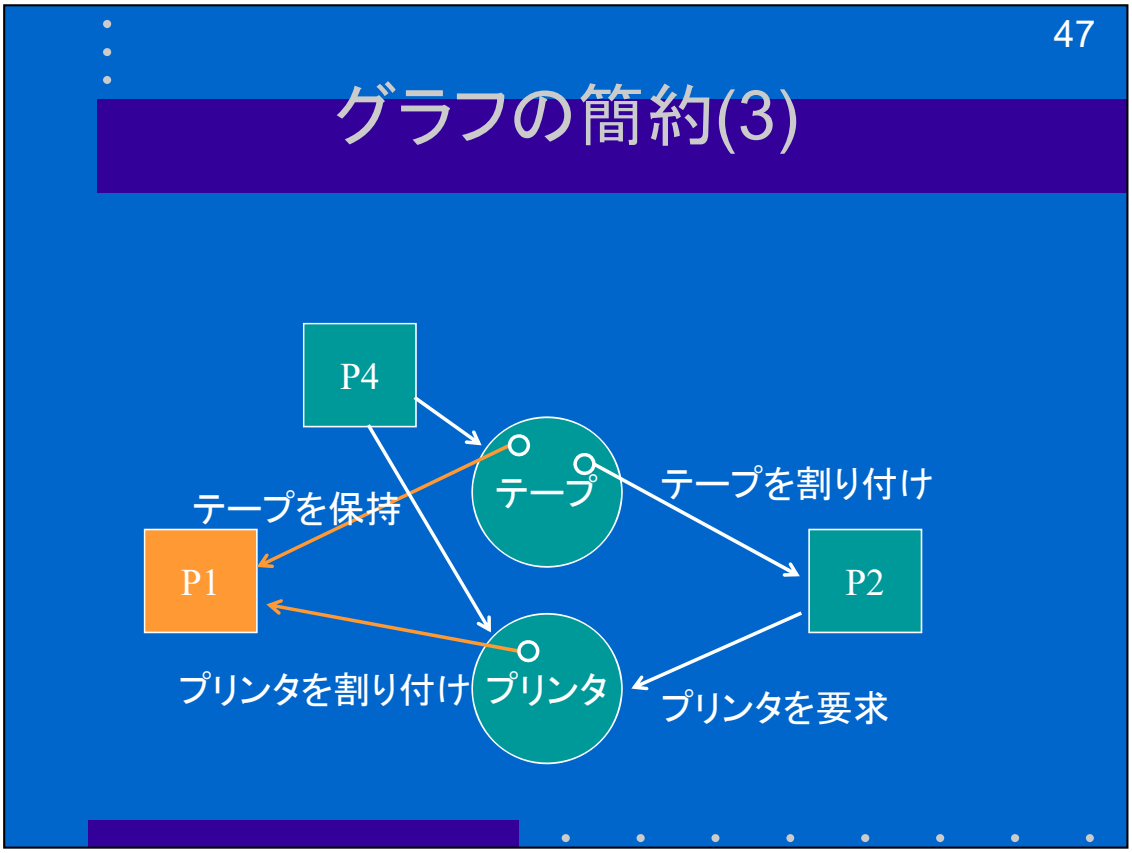
最後に残ったグラフに閉路はあるかどうかで、判定します。

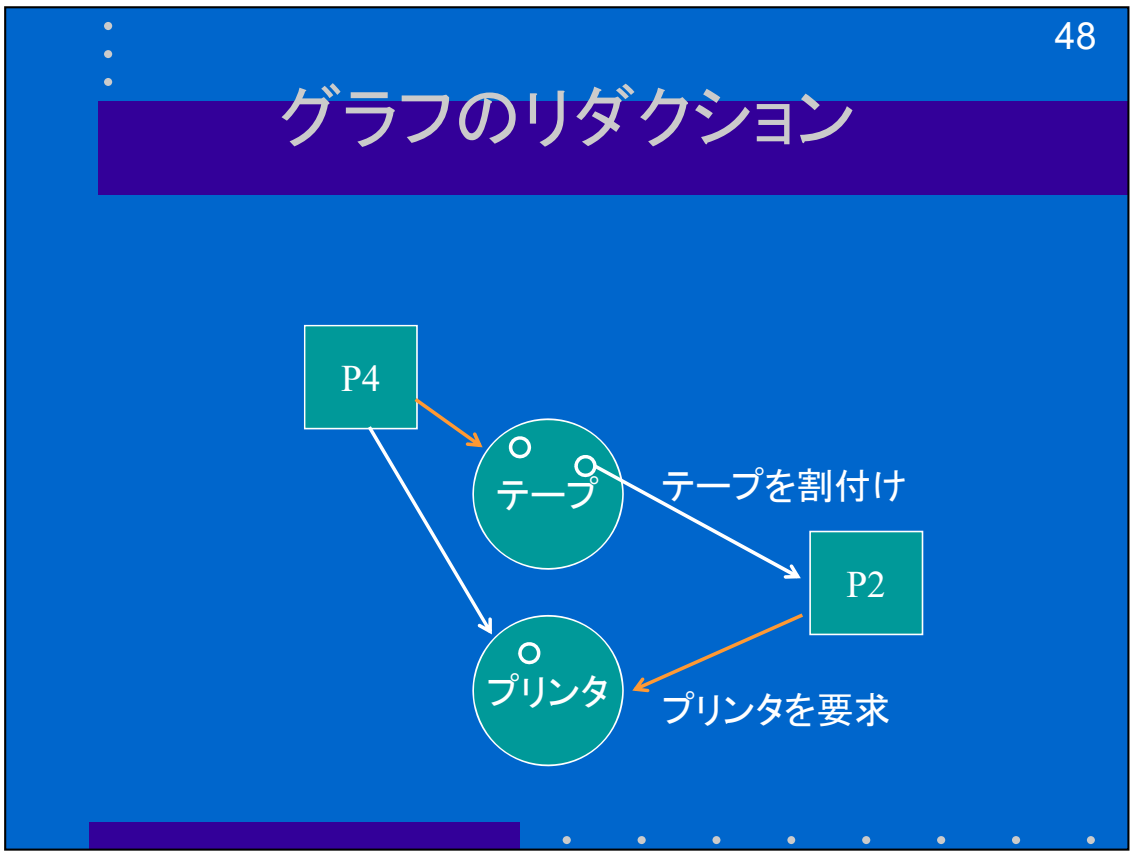
ある時点で資源割り付けグラフに閉路がなくても、将来閉路ができる可能性があるので注意が必要です。



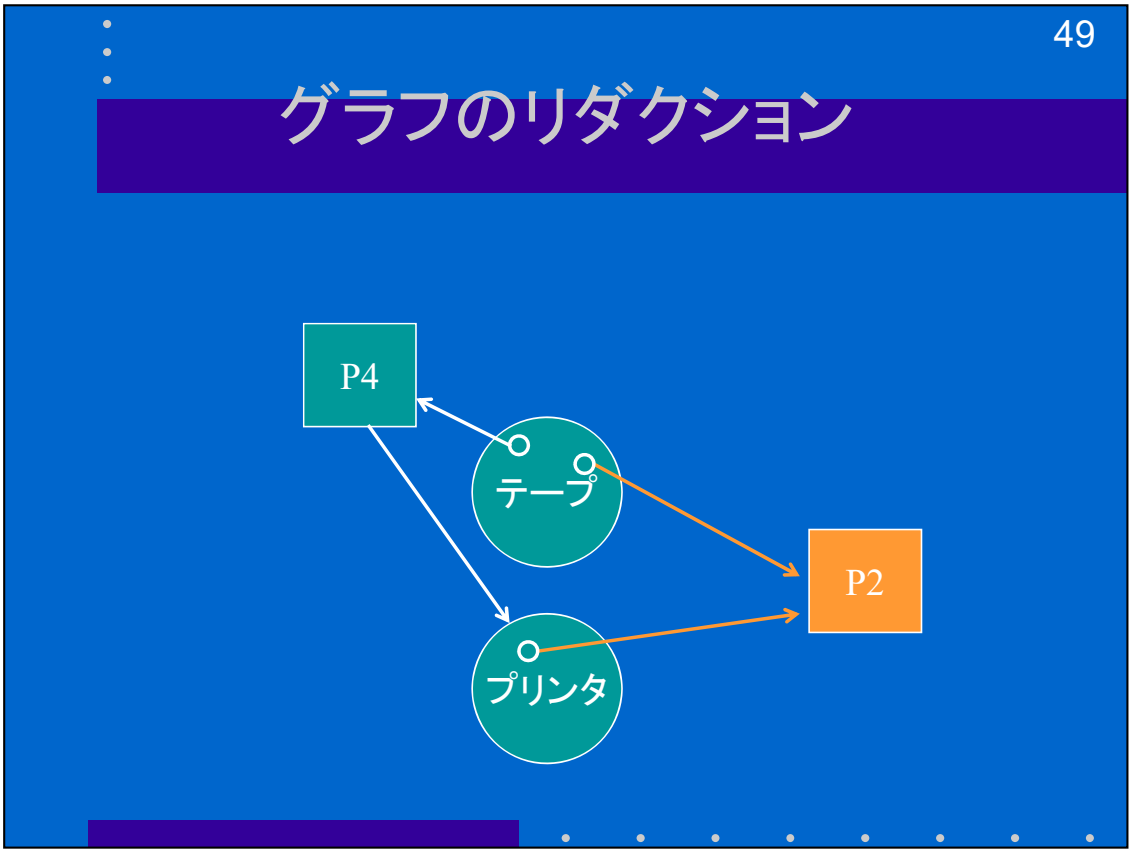
グラフの簡約の例を示します（スライド45から47）

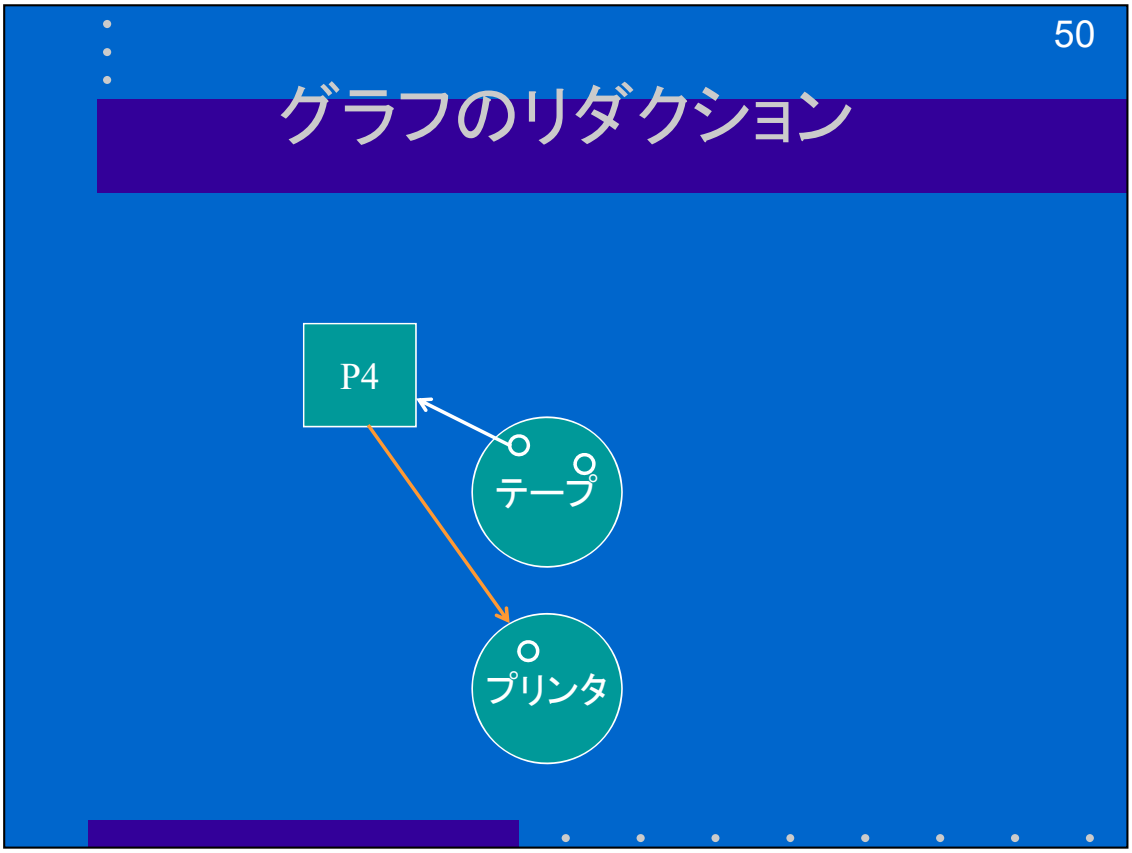


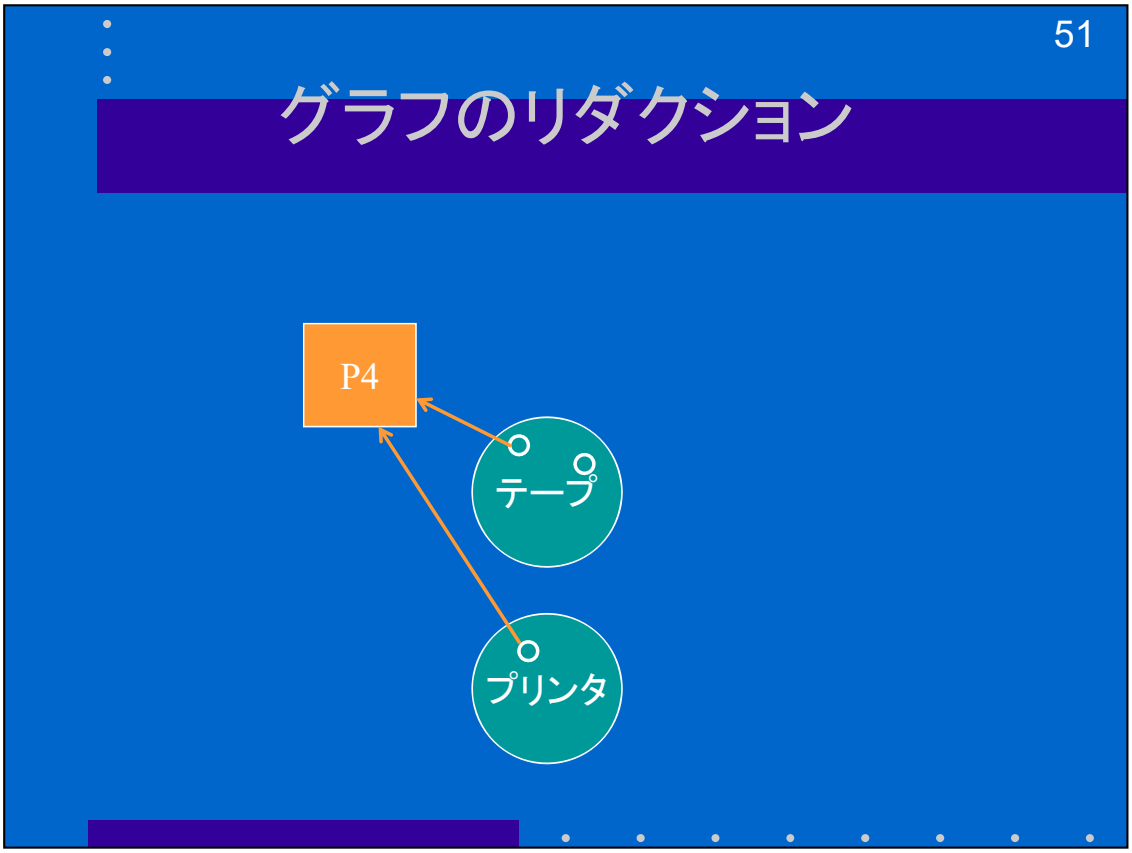


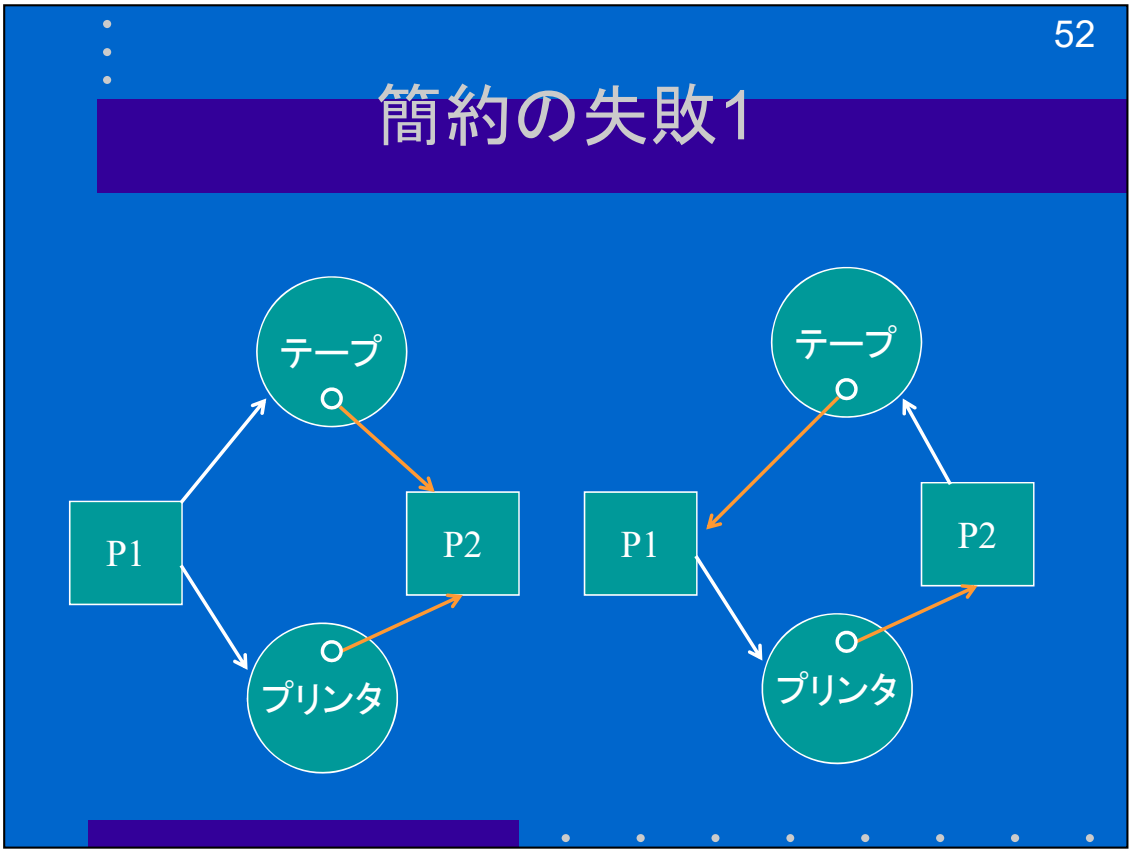


グラフのリダクションの例です（スライド48から51）

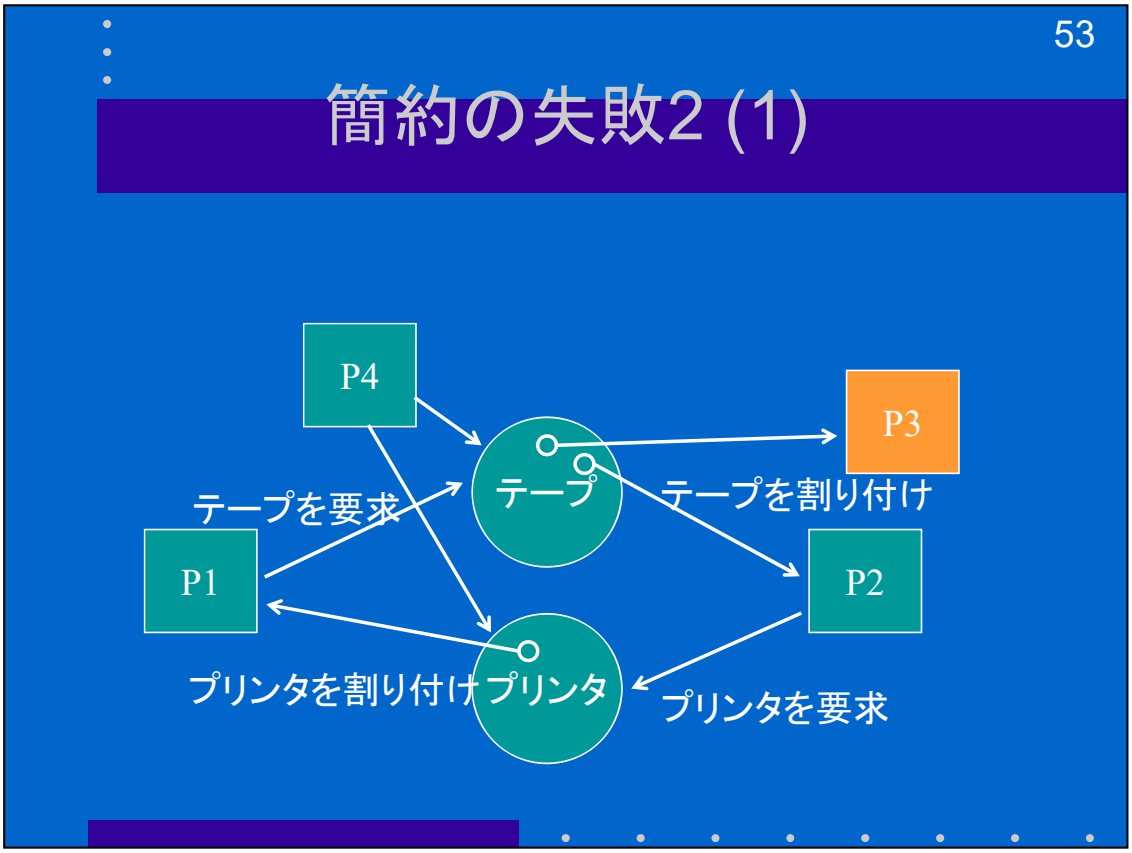




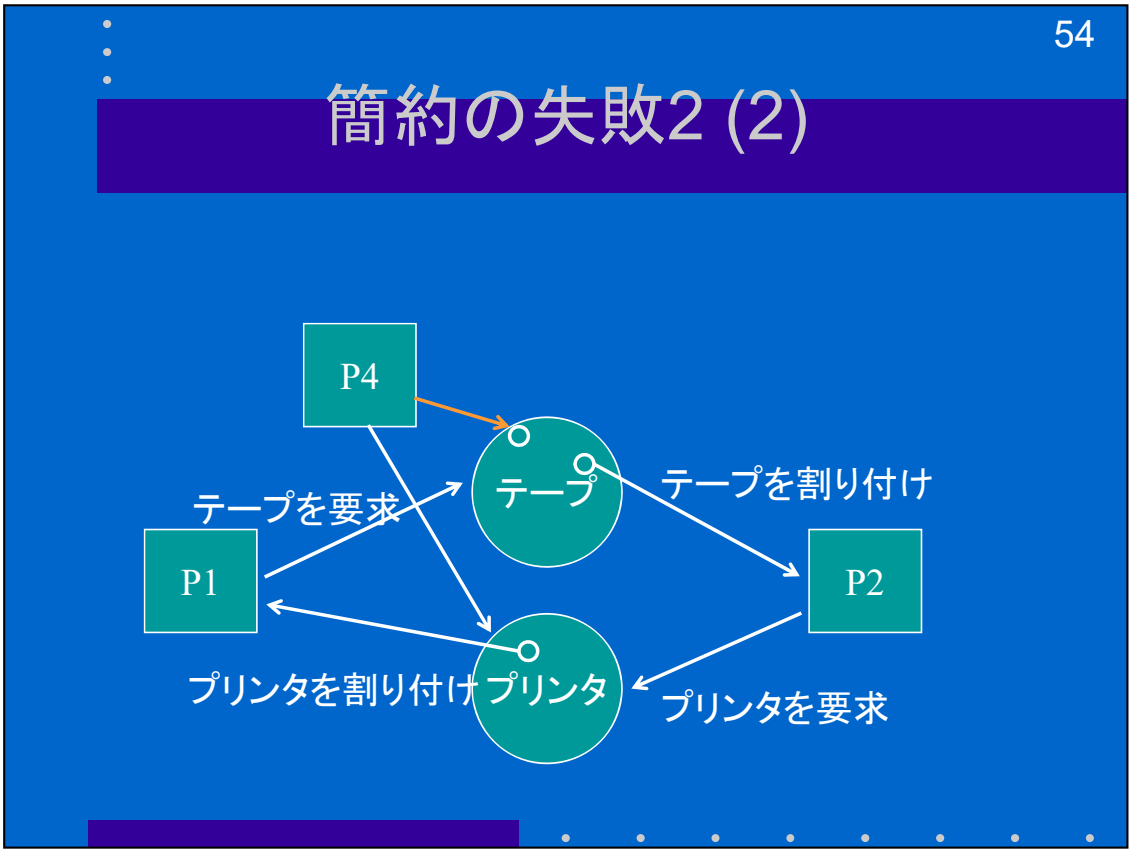


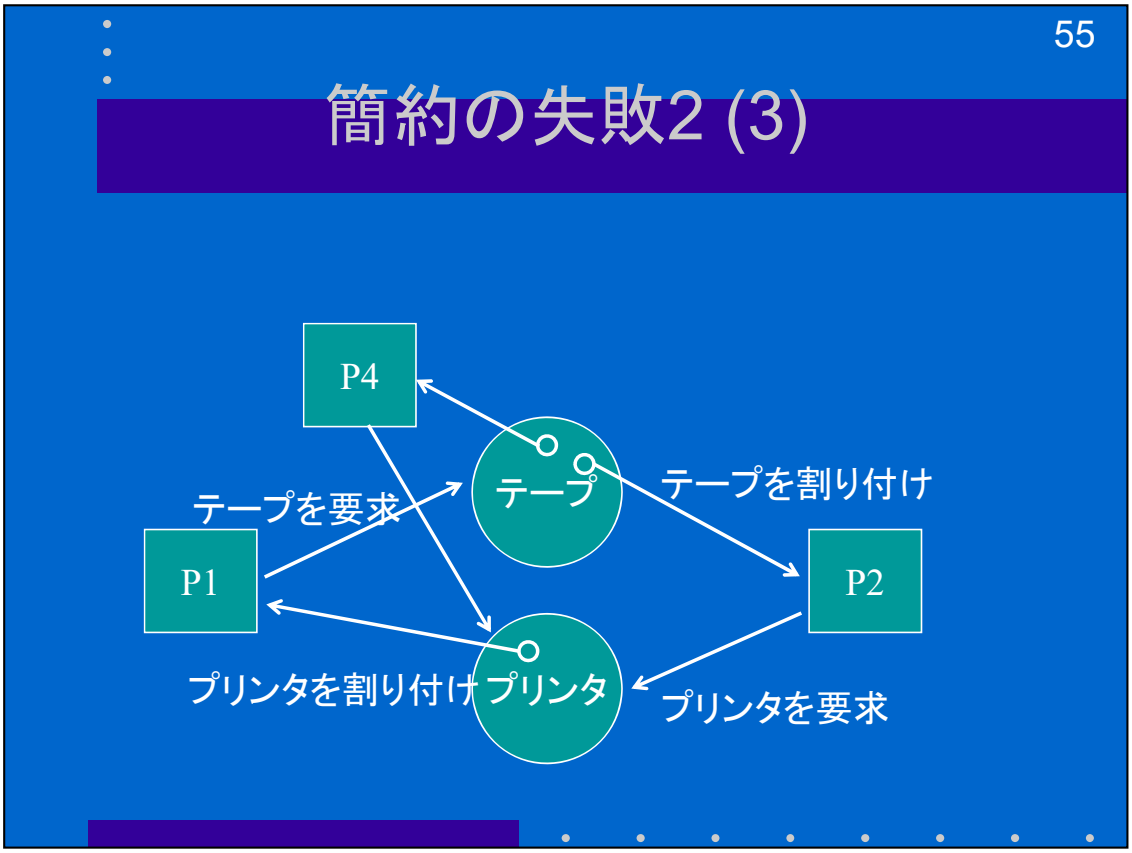


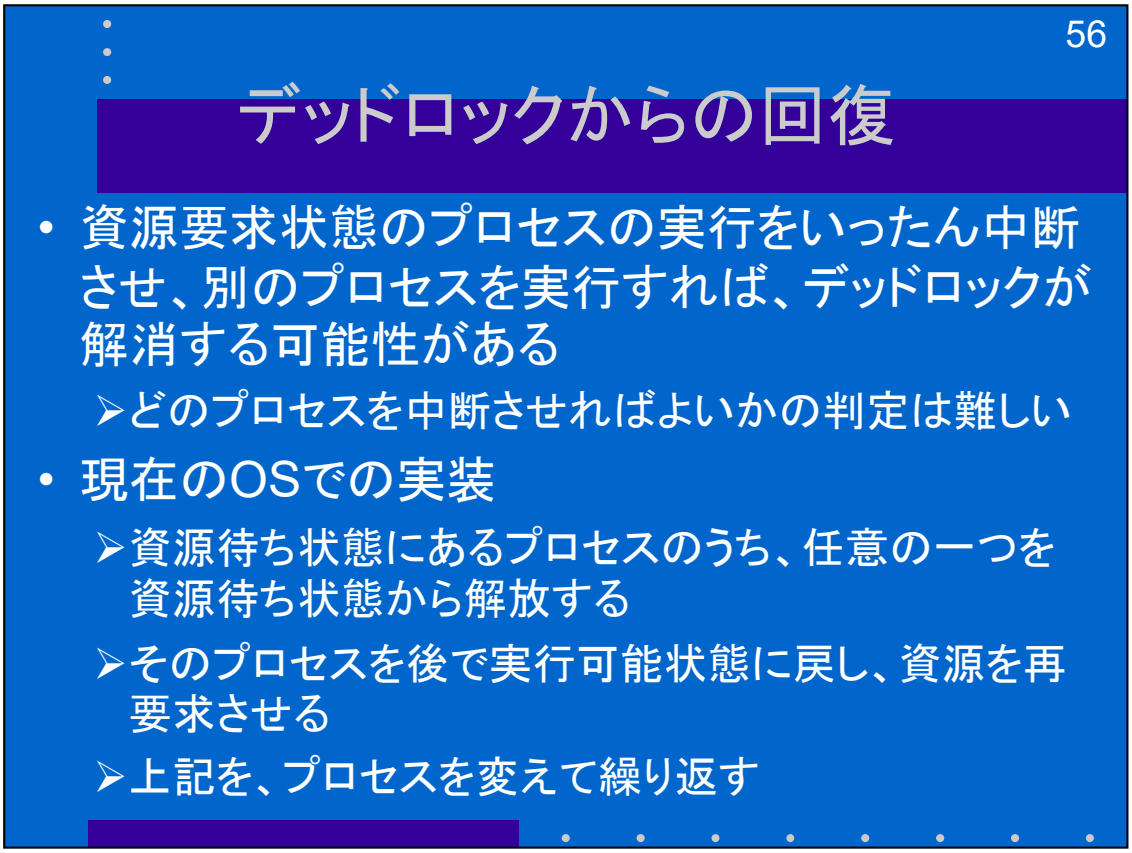
簡約の失敗例1です。



簡約の別の失敗例です（スライド53から55）



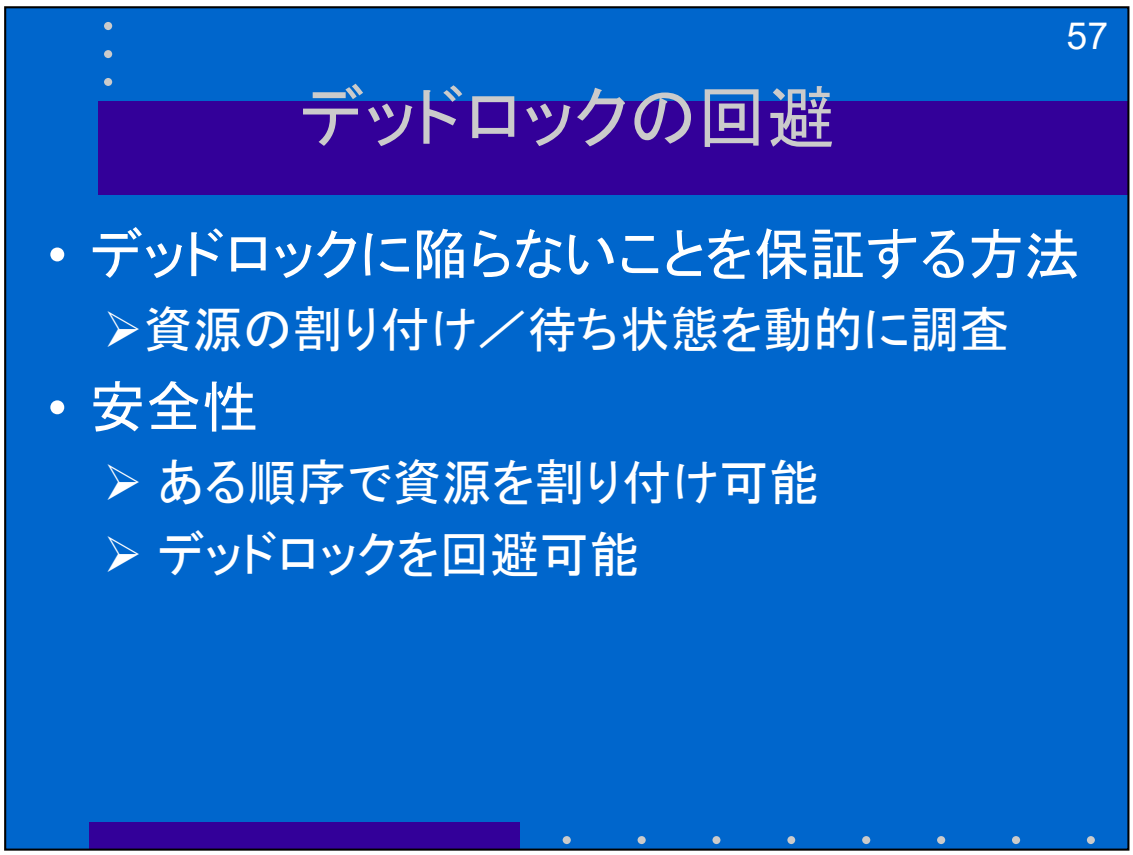




デッドロックからの回復について説明します。

資源要求状態のプロセスの実行をいったん中断させ、別のプロセスを実行すれば、デッドロックが解消する可能性があります。
どのプロセスを中断させればよいかの判定は難しいということです。

現在のOSでの実装では、
資源待ち状態にあるプロセスのうち、任意の一つを資源待ち状態から解放する
そのプロセスを後で実行可能状態に戻し、資源を再要求させる
上記を、プロセスを変えて繰り返すというものです。



デッドロックの回避とは、デッドロックに陥らないことを保証する方法を指します。
資源の割り付け／待ち状態を動的に調査します。

ここで、安全性とは、ある順序で資源を割り付け可能かどうか、デッドロックを回避可能かどうかを判定することです。

銀行家のアルゴリズム(1)（Banker's Algorithm）

- DijkstraとHabermannが考案
- 資源の割り付け状態が変化するたびに、安全性を検査する
- プロセスが要求した通りに資源割り付けを行ったと仮定したときに、安全性が満たされない状態になれば、その要求を許可しない
- 各資源を資源の型ごとに分類して抽象化

  例：テープが2台、プリンタが3台

  →資源型Aの資源が2個、資源型Bの資源が3個

デッドロックを回避する方法の一つに、銀行家のアルゴリズムがあります。

DijkstraとHabermannが考案したもので、資源の割り付け状態が変化するたびに、安全性を検査します。

プロセスが要求した通りに資源割り付けを行ったと仮定したときに、安全性が満たされない状態になれば、その要求を許可しないことで、デッドロックを回避します。

各資源を資源の型ごとに分類して抽象化します。

例：テープが2台、プリンタが3台

・資源型Aの資源が2個、資源型Bの資源が3個

59

銀行家のアルゴリズム(2)

- アルゴリズムで使う変数の説明
  - *n*: プロセスの個数
  - *m*: 資源型の個数
  - Allocation(*i*,*j*): プロセスP*i*に割り付け中の資源型*j*の資源の個数
  - Max(*i*,*j*): プロセスP*i*が資源型*j*の資源を割り付けるときの最大値
  - Available(*j*): 現在、割り付け可能な資源型*j*の資源の個数
  - Request(*i*,*j*): プロセスP*i*が資源型*j*の資源を要求する個数
  - Need(*i*,*j*): これからプロセスP*i*が資源型*j*の資源を要求する可能性のある最大値(Max(*i*,*j*)-Allocation(*i*,*j*))
  - Work(*j*): 利用可能な資源型*j*の資源の個数(一時変数)
  - Finish(*i*): プロセスP*i*の安全性が確認されたかどうか(論理変数)

アルゴリズムで使う変数の説明です。

多数の変数がありますので、次のスライド以降で実際の使われ方を見てから、理解してもらえばよいと思います。

60

銀行家のアルゴリズム(3)
安全状態の検査

1. Work(*j*) ← Available(*j*) (1≦*j*≦*m*);
   Finish(*i*) ← false (1≦*i*≦n);
2. 次の条件を満たす *i* を見つける
   条件：Finish(*i*) = falseで、かつすべての *j* でNeed(*i*,*j*)≦Work(*j*)
   この条件を満たす *i* が見つかれば次へ、なければステップ4へ
3. Work(*j*) ← Work(*j*)+Allocation(*i*,*j*) (1≦*j*≦*m*);
   Finish(*i*) ← true;
   ステップ2へ
4. すべての *i* に対してFinish(*i*)=trueならシステムは安全状態

アルゴリズムで行う安全性の検査の手順です。

変数が多数あって煩雑なので、次のスライドの例題をまず見てから、理解してもらえばよいです。

例題1　安全状態の検査

	Allocation	Max	Need	Available
資源型j	1 2 3	1 2 3	1 2 3	1 2 3
プロセス				3 3 2
P1	2 0 0	3 2 2	1 2 2	
P2	2 1 1	2 2 2	0 1 1	
P3	0 0 2	4 3 3	4 3 1	

このとき、この状態は安全か？

安全状態の検査の例題を説明します。

Allocation, Max, Needは、引数にプロセスと資源型を取ります。

例えば、プロセスP1の右の(2 0 0)とは、それぞれ資源型1, 2, 3のAllocation（割り付け中の資源の個数）が2個、0個、0個であることを表します。

Availableの下の数は、(1 2 3)がそれぞれ資源型1, 2, 3を指し、その下の(3 3 2)はそれぞれ、資源型1, 2, 3が3個、3個、2個利用可能であることを示します。

例題1　安全状態の検査（つづき）

	Allocation	Max	Need	Available	Work
資源型j	1 2 3	1 2 3	1 2 3	1 2 3	1 2 3
プロセス				3 3 2	
P1	2 0 0	3 2 2	1 2 2		① 3 3 2
P2	2 1 1	2 2 2	0 1 1		② 5 3 2
P3	0 0 2	4 3 3	4 3 1		③ 7 4 3

安全なプロセスの系列P1, P2, P3が存在するため、安全状態

安全状態の検査では、プロセスに資源を割り付けることのできる実行順が存在することを検査します。

この例では、プロセスP1, P2, P3に割り付け可能であることを示します。

具体的には、まず、P1のNeed（割り付けを要求している個数）を見ると、(1 2 2)（資源1, 2, 3をそれぞれ1個、2個、2個要求）していますが、これはAvailableが(3 3 2)であるため割り付け可能です。

P1の割り付けができると、P1はディスパッチされ、実行が終了すると、自分に割り付け中の資源（Allocationの個数）である(2 0 0)（つまり資源1を2個）解放します。

すると、割り付け可能な資源の数（ここではWorkの列）は（5 3 2）となります。これが、次に割り付けるプロセスP2のWorkに書かれています。

次にP2を見ると、Needが(0 1 1)なので割り付け可能です。P1もディスパッチされて終了すると、Allocationにある（2 1 1）が解放されて、次のWorkの値は（7 4 3)となります。

これが、その次に割り付けるプロセスP3のWorkに書かれています。

P3もNeedが(4 3 1）なので、(7 4 3）と比べると割り付け可能です。

結果的に、P1, P2, P3の系列が存在するので、これは安全状態となります。

銀行家のアルゴリズム(4)
資源の割り付け

1. プロセスP*i*からの要求を処理する場合
   Request(*i*,*j*)≦Need(*i*,*j*) (1≦*j*≦m)ならステップ2へ
   そうでなければエラーで終了(本来、要求する以上の資源を要求している)
2. Request(*i*,*j*)≦Available(*i*,*j*) (1≦*j*≦*m*)ならステップ3へ
   そうでなければP*i*の要求は受けられないとして終了
3. P*i*の要求に対して、次のように値を更新
   Available(*j*) ← Available(*j*)-Request(*i*,*j*) (1≦*j*≦*m*);
   Allocation(*i*,*j*) ← Allocation(*i*,*j*)+Request(*i*,*j*) (1≦*j*≦*m*);
   Need(*i*,*j*) ← Need(*i*,*j*)-Request(*i*,*j*) (1≦*j*≦*m*);
4. 値の更新後が安全状態かどうか検査
5. 安全状態であれば資源割り付けを行う
   そうでなければ行わない

アルゴリズムで、資源の割り付けを行うかどうかの手順はこのようになります。

こちらも、次のスライドにある例題を先に参照して、動作を理解してもらえればよいと思います。

例題2　資源割り付け要求の許可(1)

	Allocation	Max	Need	Available
資源型*j*	1 2 3	1 2 3	1 2 3	1 2 3
プロセス				3 3 2
P1	2 0 0	3 2 2	1 2 2	
P2	2 1 1	2 2 2	0 1 1	
P3	0 0 2	4 3 3	4 3 1	

このとき、プロセスP1の資源割り付け要求(1 1 1)は許可されるか？

次に、資源割り付け要求の許可するかどうかの判定です。

この状態では、プロセスP1の資源割り付け要求(1 1 1)は許可されるでしょうか？

例題2　資源割り付け要求の許可(2)

	Allocation	Max	Need	Available
資源型j	1 2 3	1 2 3	1 2 3	1 2 3
プロセス				**2 2 1**
P1	**3 1 1**	3 2 2	**0 1 1**	
P2	2 1 1	2 2 2	0 1 1	
P3	0 0 2	4 3 3	4 3 1	

プロセスP1の資源割り付け要求(1 1 1)を処理

判定するため、プロセスP1の資源割り付け要求(1 1 1)を処理してみます。

P1が割り付け要求（1 1 1）を実行すると、その数だけ割り付け中のAllocationの値が増え、これから要求する値であるNeedは減ることになります。

また、Availableも割り付け可能な資源の数が減るので、更新されます。

66

例題2　資源割り付け要求の許可(3)

	Allocation	Max	Need	Available	Work
資源型j	1 2 3	1 2 3	1 2 3	1 2 3	1 2 3
プロセス				2 2 1	
P1	3 1 1	3 2 2	0 1 1		① 2 2 1
P2	2 1 1	2 2 2	0 1 1		② 5 3 2
P3	0 0 2	4 3 3	4 3 1		③ 7 4 3

安全なプロセスの系列P1, P2, P3が存在するため、安全状態→**資源割り付け要求を許可**

ここで、例題1と同様の手順で、安全状態の検査を行います。

安全なプロセスの系列P1, P2, P3が存在するため、安全状態となります。

資源割り付けを行っても、安全状態ですので、P1の資源割り付け要求は許可されます。

67

例題3　資源割り付け要求の不許可(1)

	Allocation	Max	Need	Available
資源型*j*	1 2 3	1 2 3	1 2 3	1 2 3
プロセス				3 3 2
P1	2 0 0	3 2 2	1 2 2	
P2	2 1 1	2 2 2	0 1 1	
P3	0 0 2	4 3 3	4 3 1	

このとき、プロセスP3の資源割り付け要求(3 3 1)は許可されるか？

それでは、P3が(3 3 1)を要求した場合はどうでしょうか？

68

例題3　資源割り付け要求の不許可(2)

	Allocation	Max	Need	Available
資源型j	1 2 3	1 2 3	1 2 3	1 2 3
プロセス				**0 0 1**
P1	2 0 0	3 2 2	1 2 2	
P2	2 1 1	2 2 2	0 1 1	
P3	**3 3 3**	4 3 3	**1 0 0**	

プロセスP3の資源割り付け要求(3 3 1)を処理

実際に、P3に(3 3 1)を割り付けてみます。

69

例題3　資源割り付け要求の不許可(3)

	Allocation	Max	Need	Available	Work
資源型j	1 2 3	1 2 3	1 2 3	1 2 3	1 2 3
プロセス				0 0 1	
P1	2 0 0	3 2 2	1 2 2		① 0 0 1
P2	2 1 1	2 2 2	0 1 1		
P3	3 3 3	4 3 3	1 0 0		

安全なプロセスの系列なし（安全状態でない）

→資源割り付け要求は許可されない

割り付け後に、安全なプロセスの系列がない、つまり安全状態ではありません。
この場合には、資源割り付け要求は許可されません。