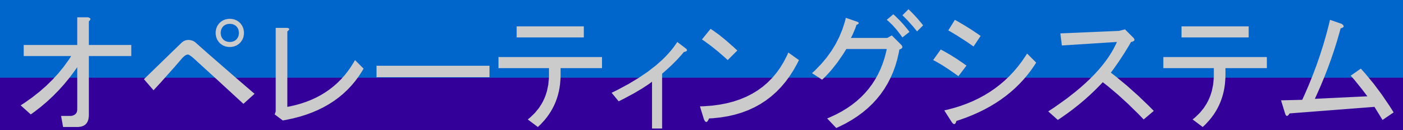


資料　第6分冊(H30)

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

2.2 並行プロセス
－プロセスの管理の理論－

- 同時に実行可能な複数のプロセスを**並行プロセス（concurrent process）**という
  - 「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う
- 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**という

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）

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）

相互排除が必要な例

- 配列でスタックを実現したとする(<u>プロセスのスタック領域とは異なることに注意</u>)
- 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

プロセス切り替えの例

プロセス1

top = top − 1

stack(top)=item

時間の流れ↓

プロセス2

item=stack(top);

top = top ＋ 1;

横取りにより、プロセス1のプッシュ操作が、プロセス2の操作により分断される可能性がある

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

同期問題

- 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
- 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る

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

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'
```

テストアンドセットの問題点

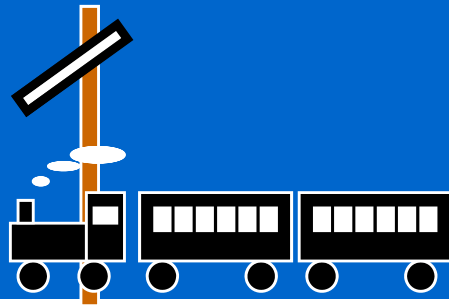
- busy waiting（プロセッサを割付けている状態で待つこと、spin lockともいう）になっている
- プロセッサ時間の無駄使い
- 長くは待たないことが明らかな場合に利用

セマフォ

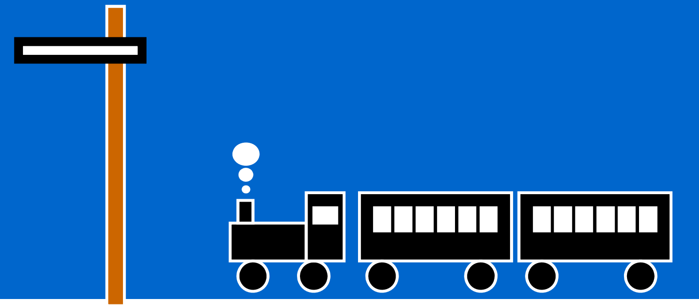
- 同期問題を解決する道具の一つ
  - 1968年E.W.Dijkstraが考案
- 整数型変数 とそれに対する手続き
- signal(semaphore)とwait(semaphore)操作
  - P(semaphore)とV(semaphore)操作と言う表記もある（教科書はこれで表記されている）
  - P: Paseren、V: Verhoog(オランダ語)
- セマフォ:　鉄道の腕木信号機

腕木信号機(図2.28)

進入可



進入不可



セマフォの種類

- 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

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

セマフォを用いたスタック操作

- 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);
```

生産者消費者問題

- 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
- バッファにはN個のデータが入る

生産者消費者問題のプログラミング例

```
/* プロセスP1（生産者）*/
for(;;){
product =送信データ;
wait(write);
put(buffer, product);
signal(read);
}
```

```
/* プロセスP2（消費者）*/
for(;;) {
wait(read);
get(buffer, product);
signal(write);
productを読み出す;
}
```

- writeの初期値：バッファのサイズ
- readの初期値：0

例題1

```
/* プロセスP1(生産者) */
product =送信データ;
wait(write);
buffer=product;
signal(read);
```

```
/* プロセスP2(消費者) */
wait(read);
product = buffer;
signal(write);
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)

例題1の解答

```
/* プロセスP1(生産者) */
②product =送信データ;
③wait(write);
④buffer=product;
⑤signal(read);
```

```
/* プロセスP2(消費者) */
①wait(read);
⑥product = buffer;
⑦signal(write);
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2

```
/* プロセスP1(生産者) */
product =送信データ;
wait(read);
buffer=product;
signal(write);
```

```
/* プロセスP2(消費者) */
wait(read);
product = buffer;
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？

例題2の解答例

```
/* プロセスP1(生産者) */
②product =送信データ;
③wait(read);
buffer=product;
signal(write);
```

```
/* プロセスP2(消費者) */
①wait(read);
product = buffer;
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**)

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）

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

直接指定方式

- プロセスを指定する
- 1対1通信に有効
  - send(PID, data)
  - receive(PID,　data)
- 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らない
  - 異なる計算機システム間では困難

間接指定方式

- 論理的な通信媒体を指定
  - 通信媒体の例
    - 共用バッファ
    - パイプ
- 通信媒体の機能
  - メッセージを送り付ける先
  - メッセージを取り出す対象
- 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**)

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

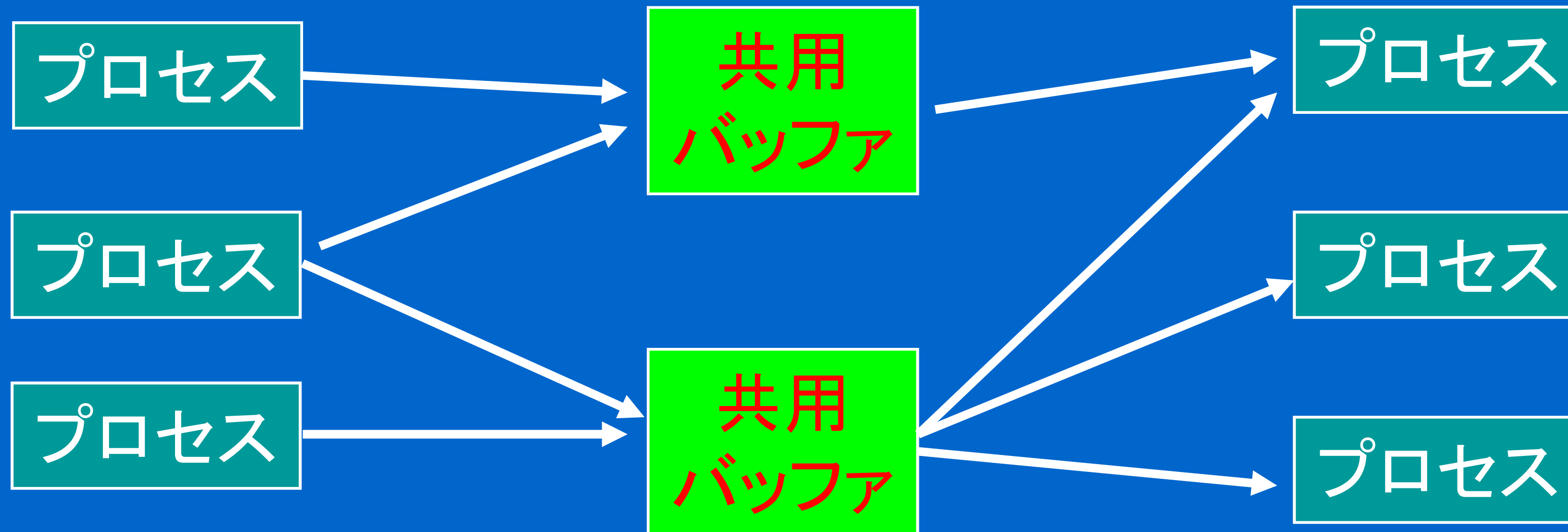
サーバ･クライアントモデル

- サーバ　プロセス:
  - クライアントから要求を受け付け、処理する
  - 返事を返すこともある
- モデルのバリエーション
  - 1対1、多対1
  - サーバ、クライアントとも複数存在して良い
- 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）

共用バッファ

- メッセージを入れる(図2.31)
  - メッセージキュー、メールボックスとも呼ばれる
  - 大域的な存在
- 送信先／受信元として共用バッファを指定
- サーバ／クライアントモデルに適する

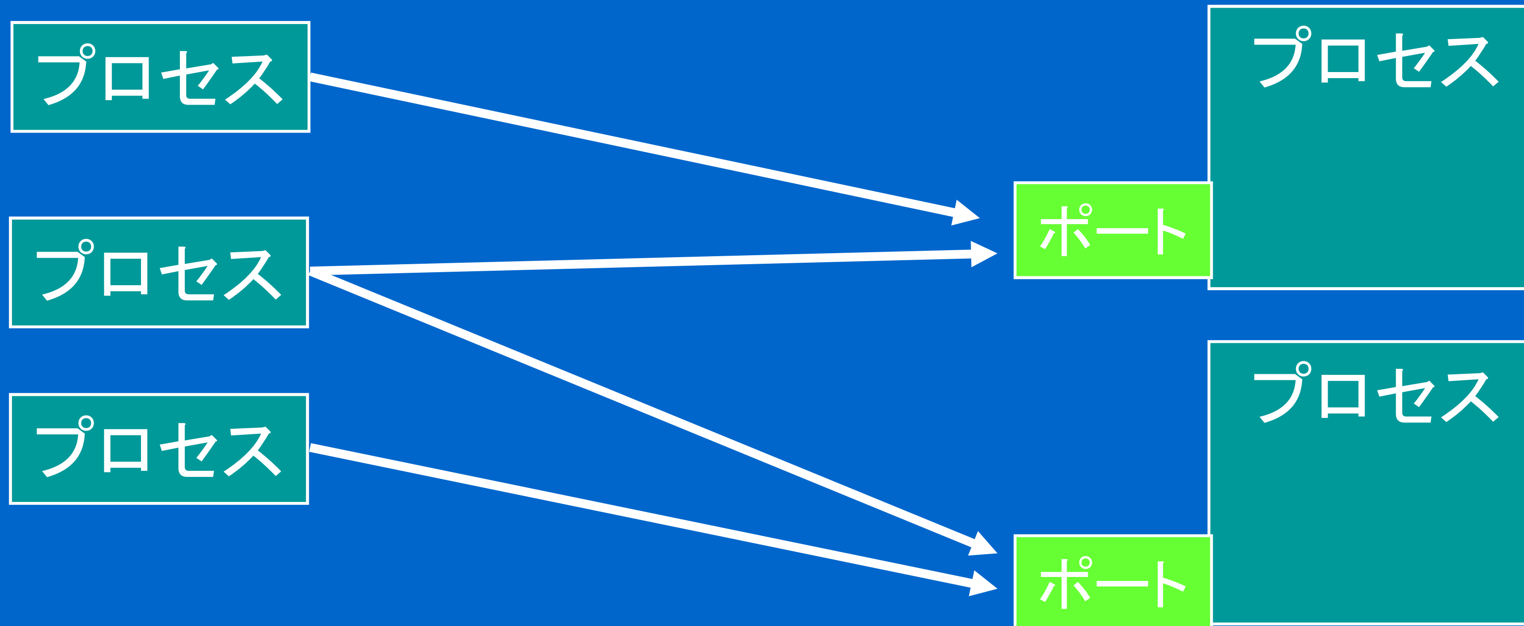
共用バッファを使ったプロセス間通信(図2.32)



ポート

- 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）
  - →ポートを指定したら受信プロセスを指定したことになる
- 多数クライアント-単一サーバモデルに適する
  - ➢単一ホスト上でのマルチサーバは可

ポート



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

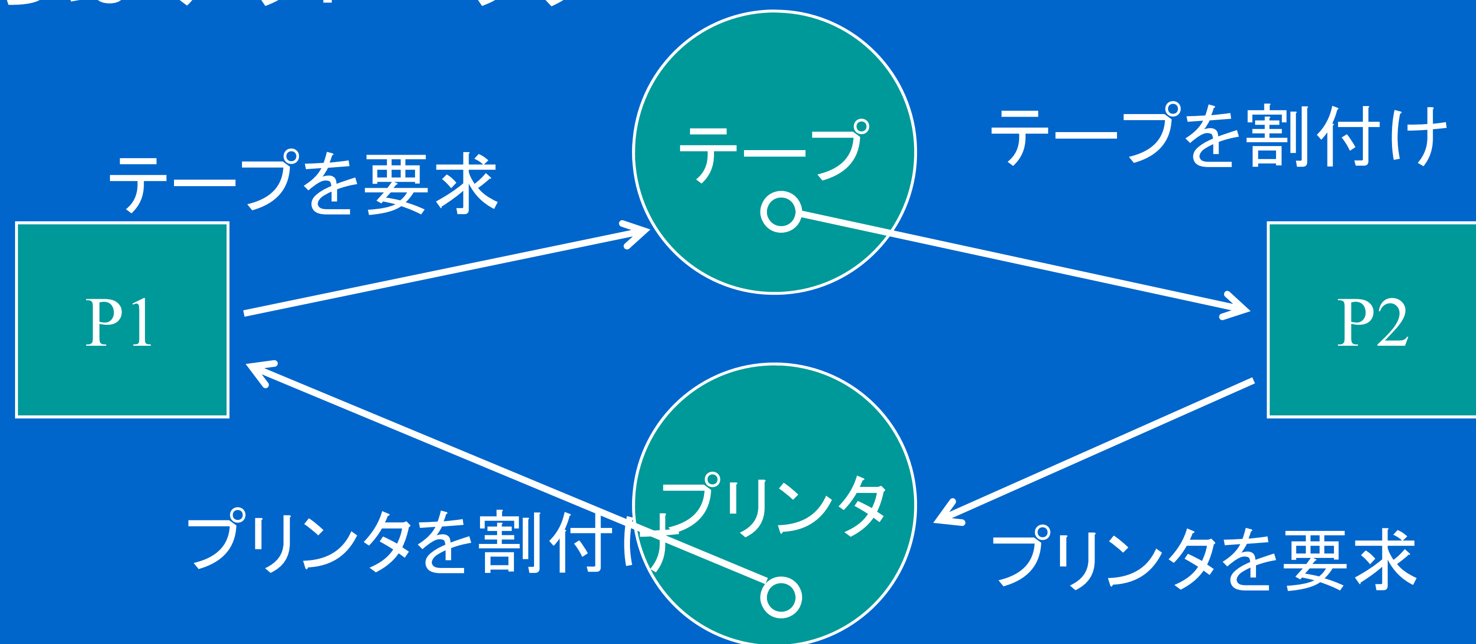
	資源1	資源2
プロセスP1	使用中	割り付け待ち
プロセスP2	割り付け待ち	使用中

デッドロックとOS

- OSの仕事:
  - 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
  - デッドロックが起きそうだったら回避
- 多くのデッドロックは資源の競合で起こる
- プロセスと資源の割り付け・要求状況
  - **資源割り付けグラフ**で表現

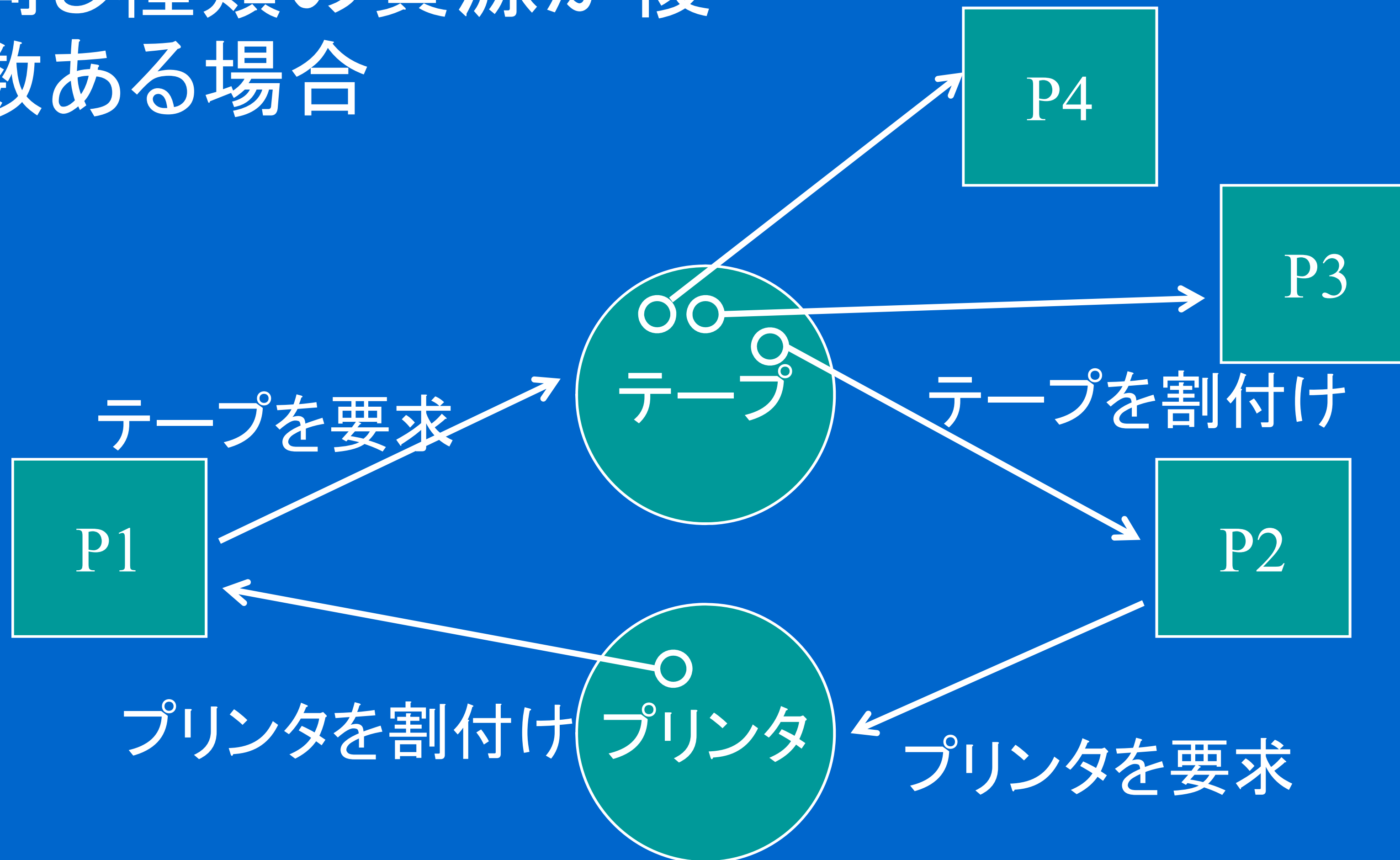
資源割り付けグラフ(1)

- 資源の割り付けと要求状況をグラフで表現
  - 丸: 資源、四角: プロセス（図2.33とは表記が逆）
  - ○→□: 割り付け、□→○: 要求
- 循環待ちはデッドロック



資源割り付けグラフ(2)

- 同じ種類の資源が複数ある場合



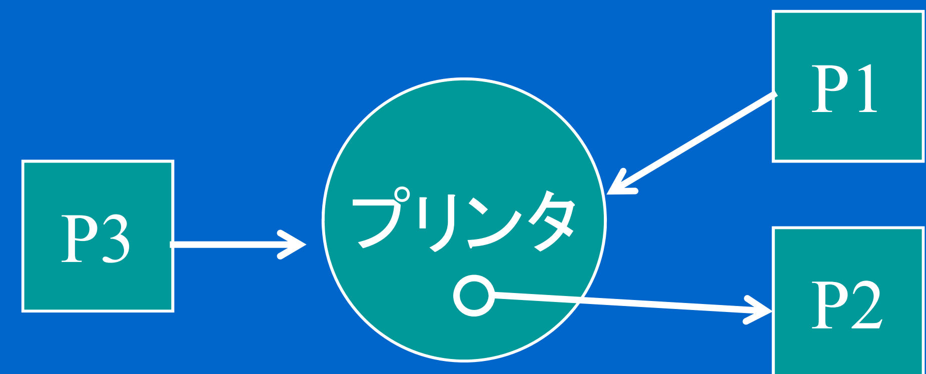
飢餓状態

- 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)）



デッドロックの必要条件

(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

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

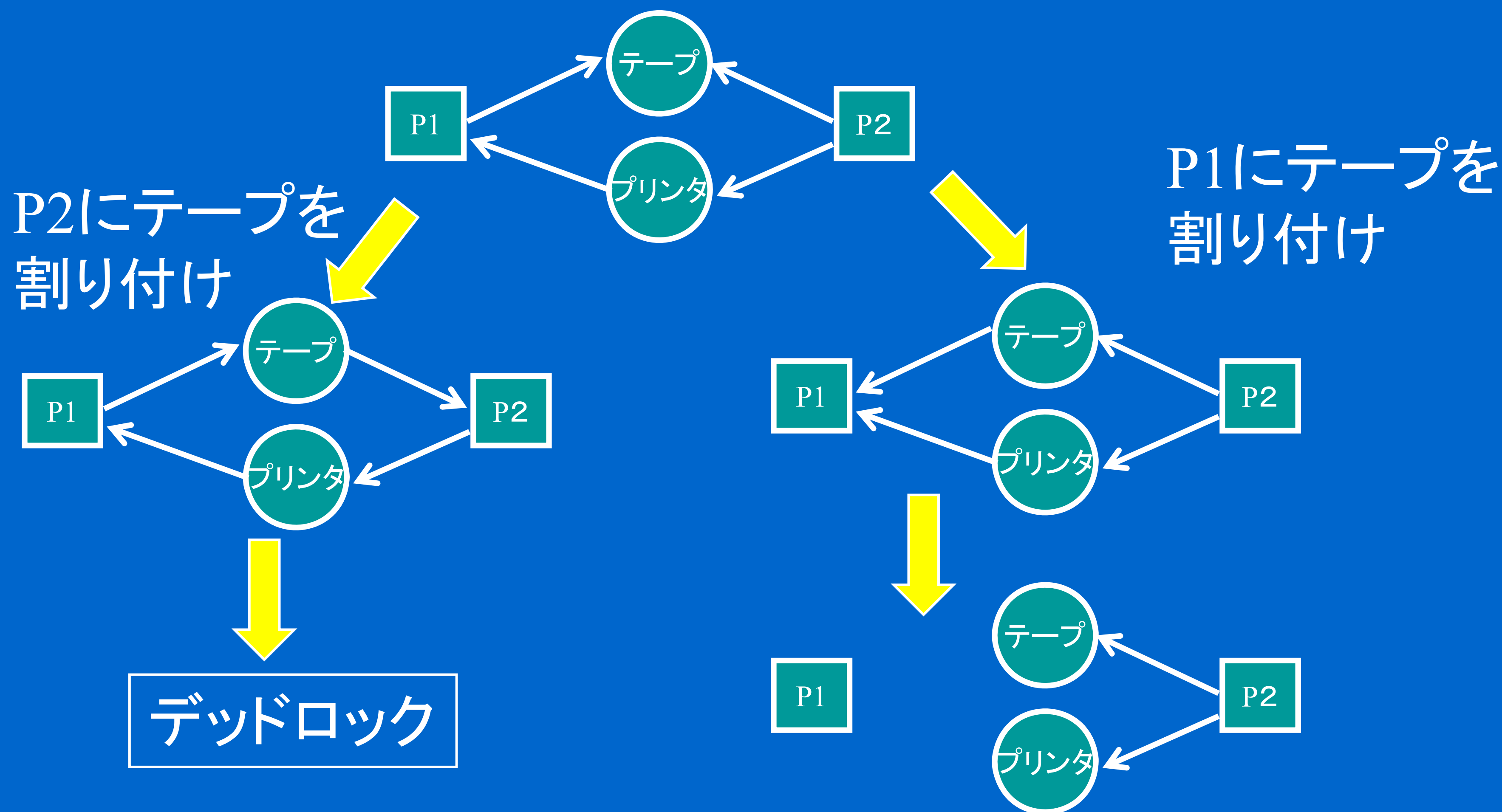
(2) **待ちの防止**：必要な資源は全部をまとめて要求する

(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

資源割り付けスケジューリング

- デッドロックになる例と回避される例



スケジューリングとデッドロック

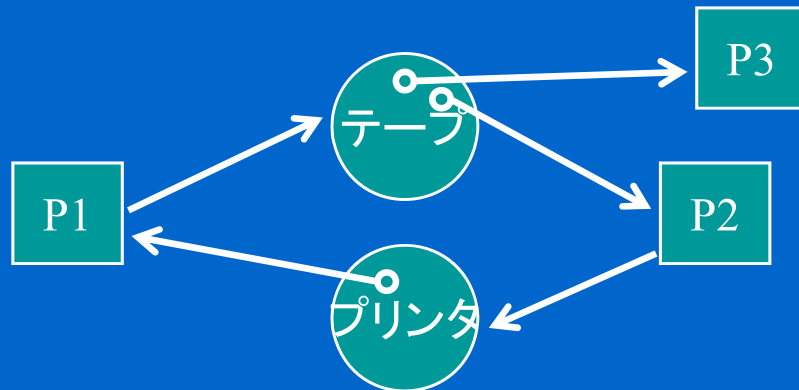
- 資源割り付け要求に対するスケジューリング
  - 資源を要求するプロセスが複数同時に存在する場合に、どのプロセスから先に資源を割り付けるか
- スケジューリング次第で、デッドロックに陥ったり、デッドロックを回避できたりする
- どのようにスケジューリングしてもデッドロックが不可避な場合がある

デッドロックの検出

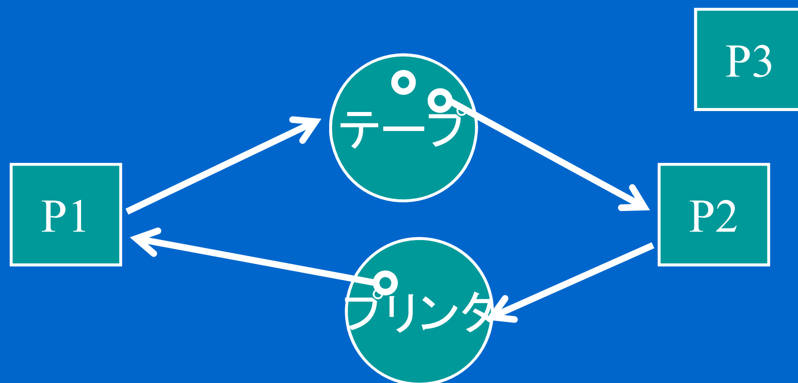
- デッドロックの検出（deadlock detection）
  - デッドロックの存在を検知する
  - デッドロックに関与しているプロセスと資源を特定する
- 資源割り付けグラフに閉路
  - デッドロックの必要条件（十分条件ではない）
- 資源割り付けグラフの簡約を行って判定

資源割り付けグラフの閉路

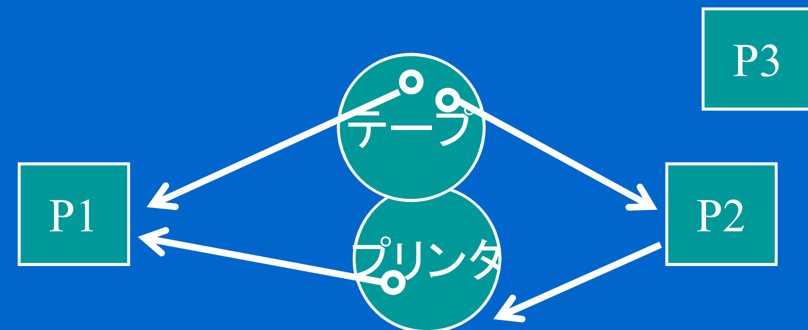
- 閉路のある資源割り付けグラフ



- P3が終了



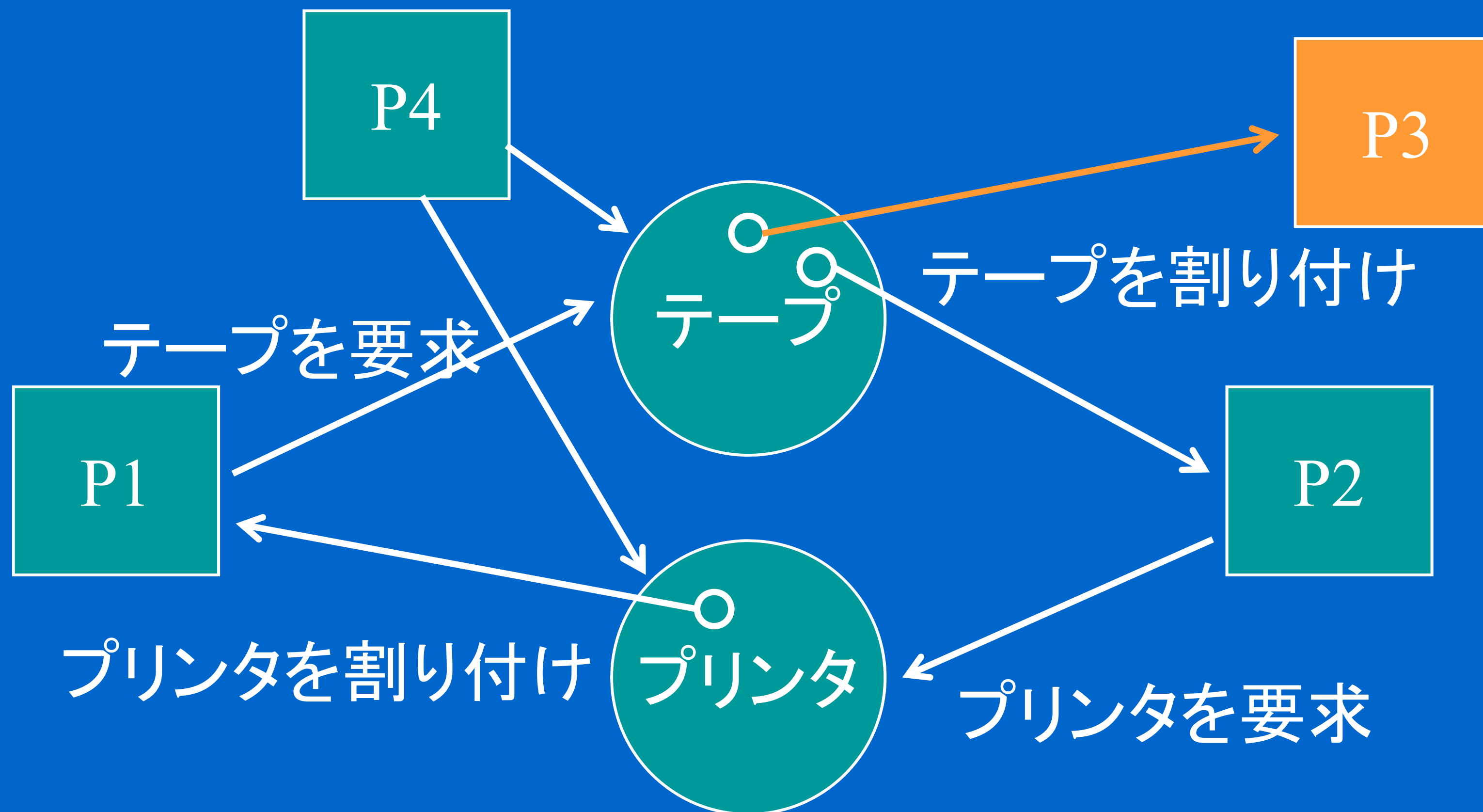
P1にテープ割り付け可



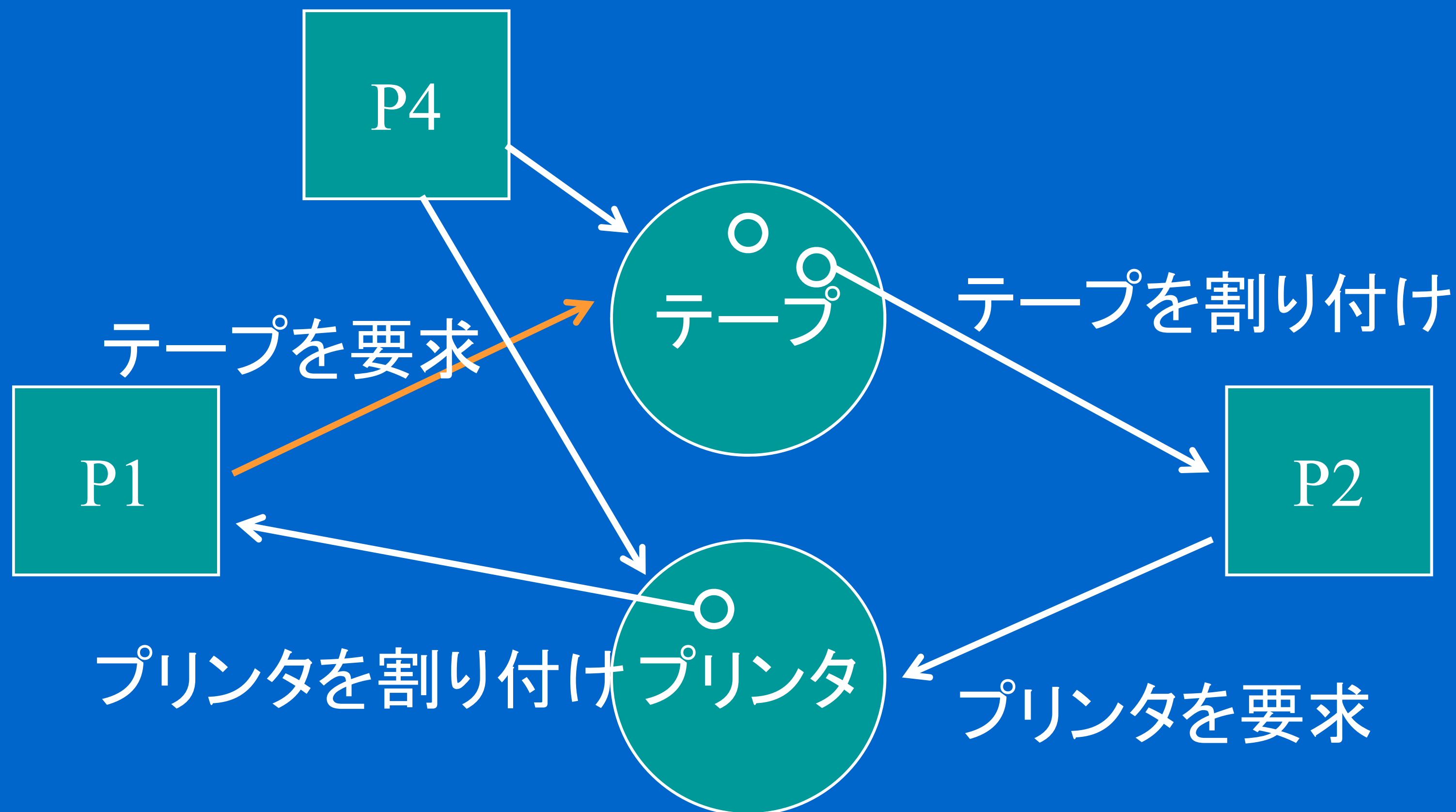
デッドロックの判定

- 閉路があっても必ずしもデッドロックではない
- 必要資源を全部得たプロセスは、いずれは資源を解放する
  - 出辺のないプロセス→辺と共に抹消
  - 割り付け可能な資源が増加→要求しているプロセスに割り付け
  - 最後に残ったグラフに閉路はあるか?
- 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

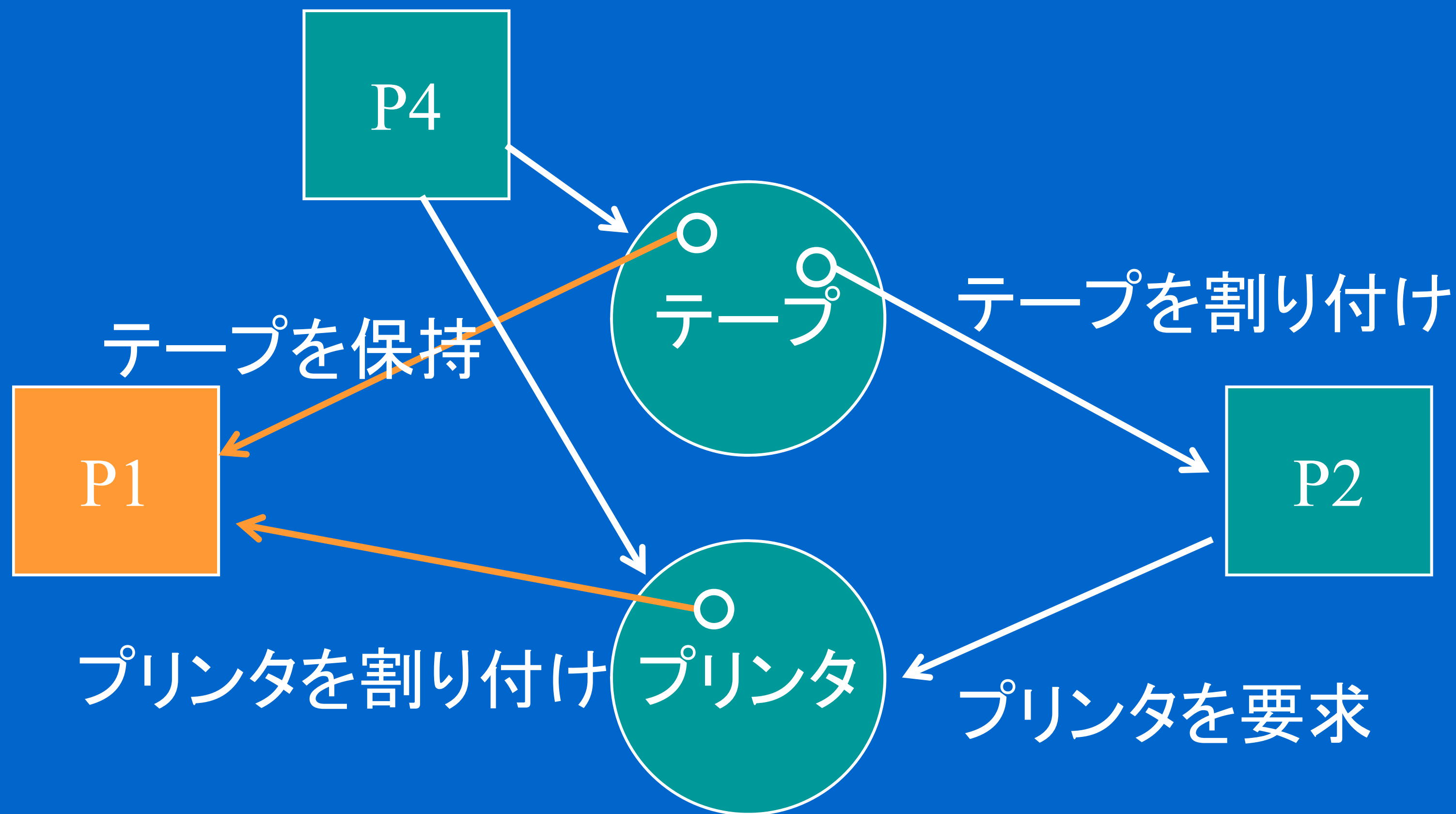
グラフの簡約(1)



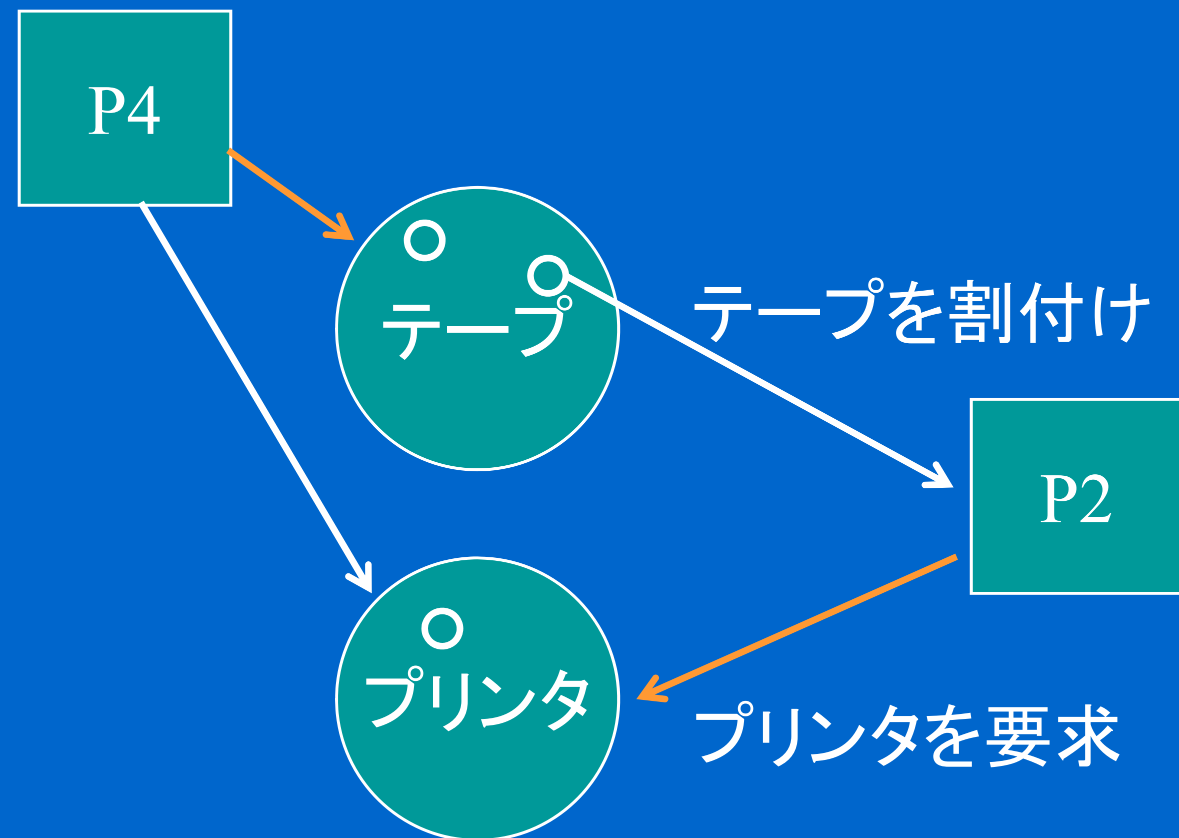
グラフの簡約(2)



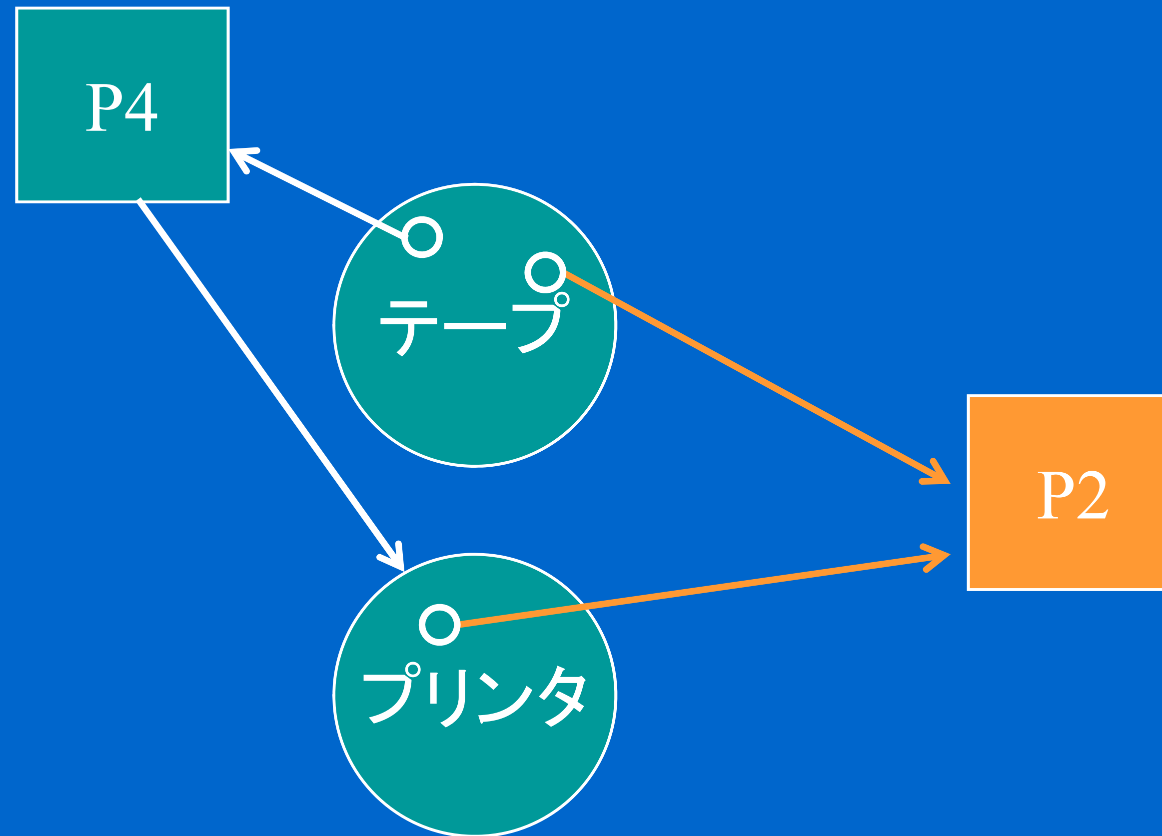
グラフの簡約(3)



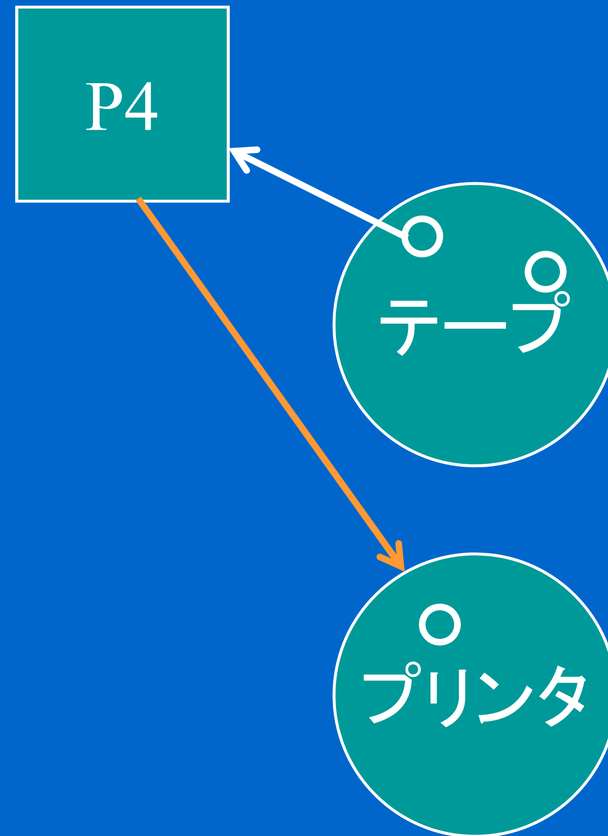
グラフのリダクション



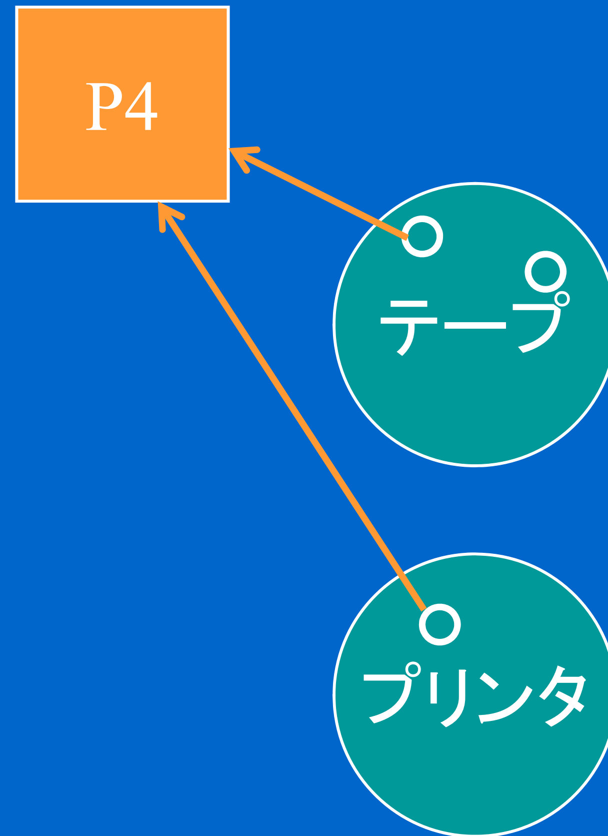
グラフのリダクション



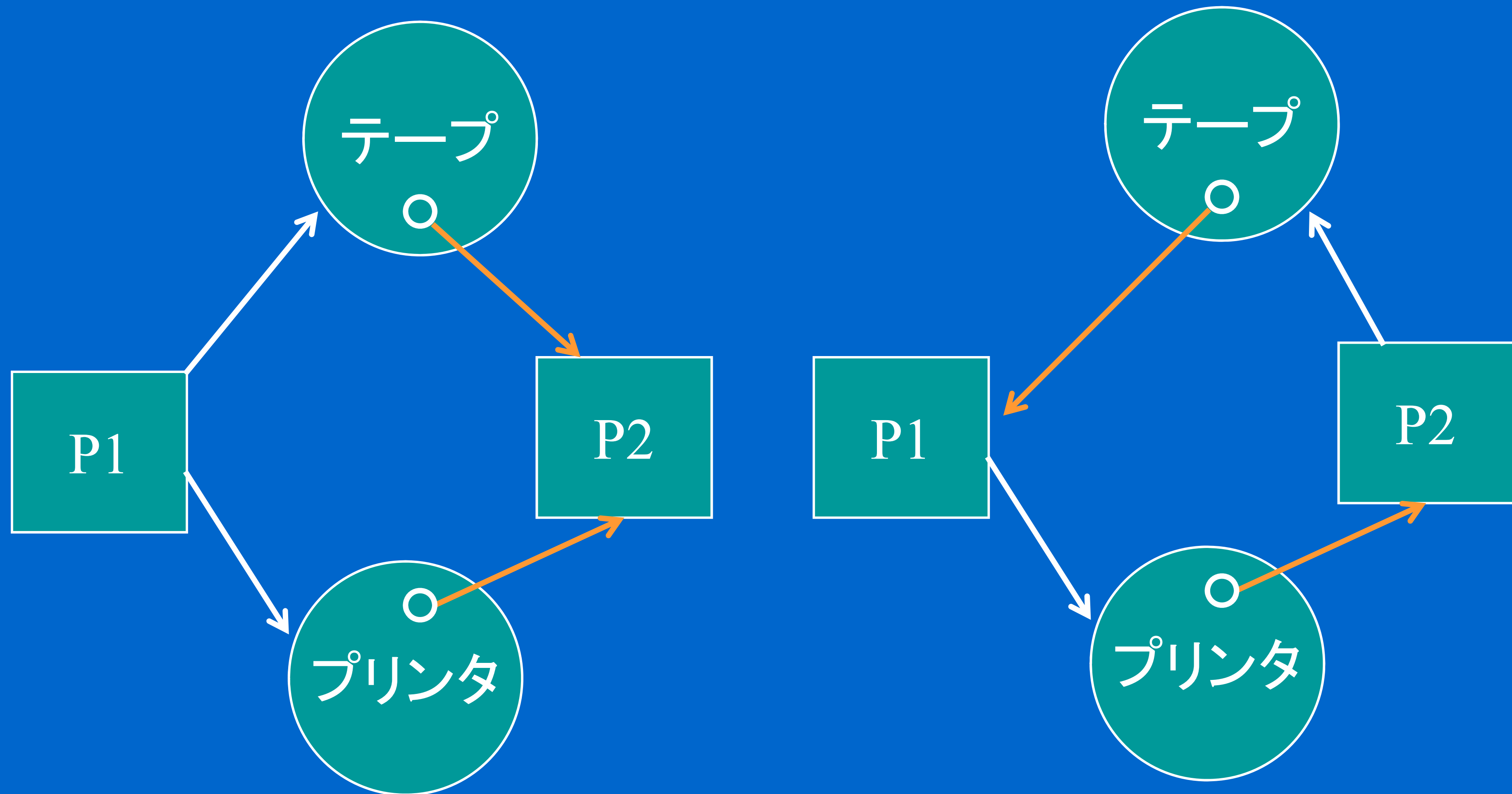
グラフのリダクション



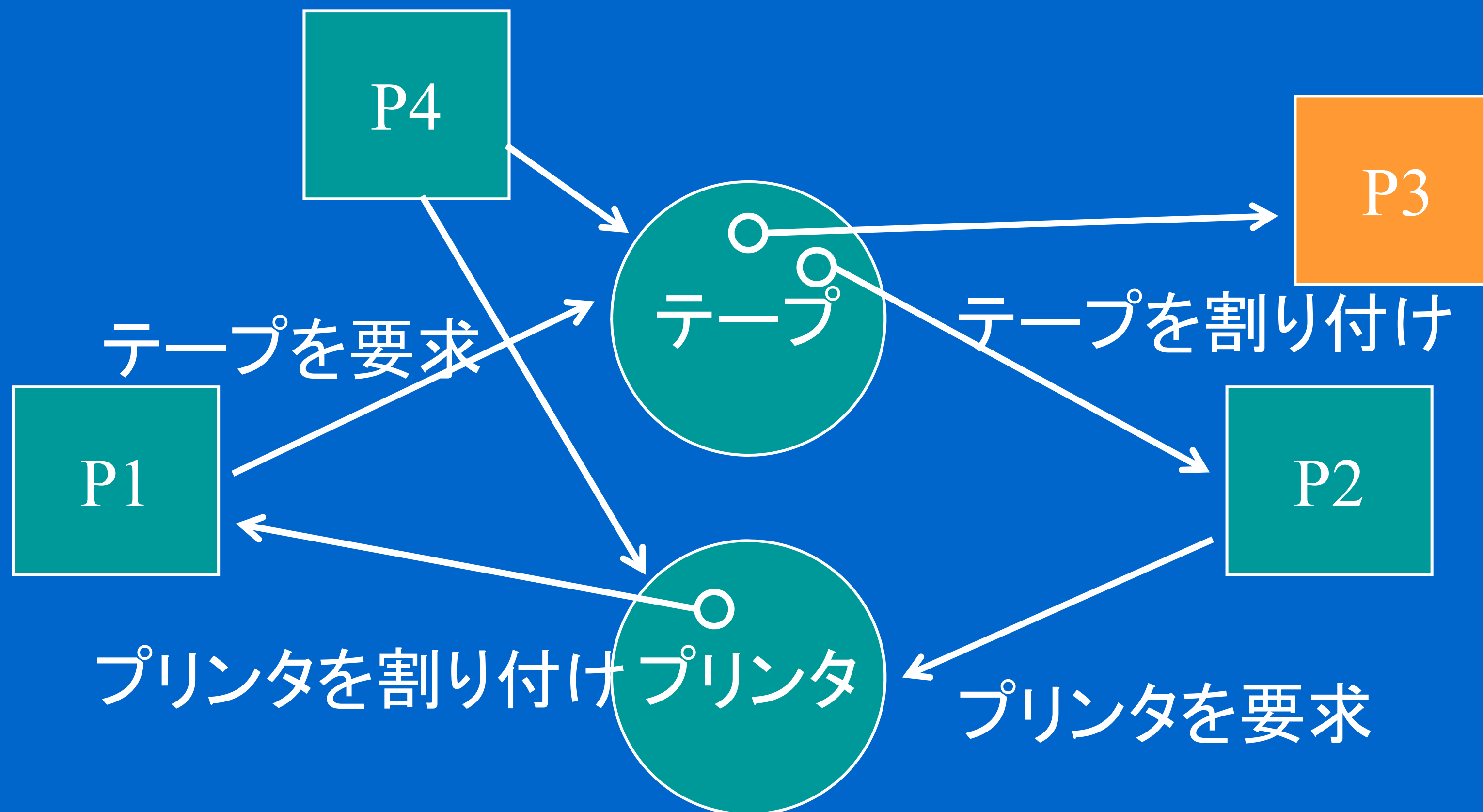
グラフのリダクション



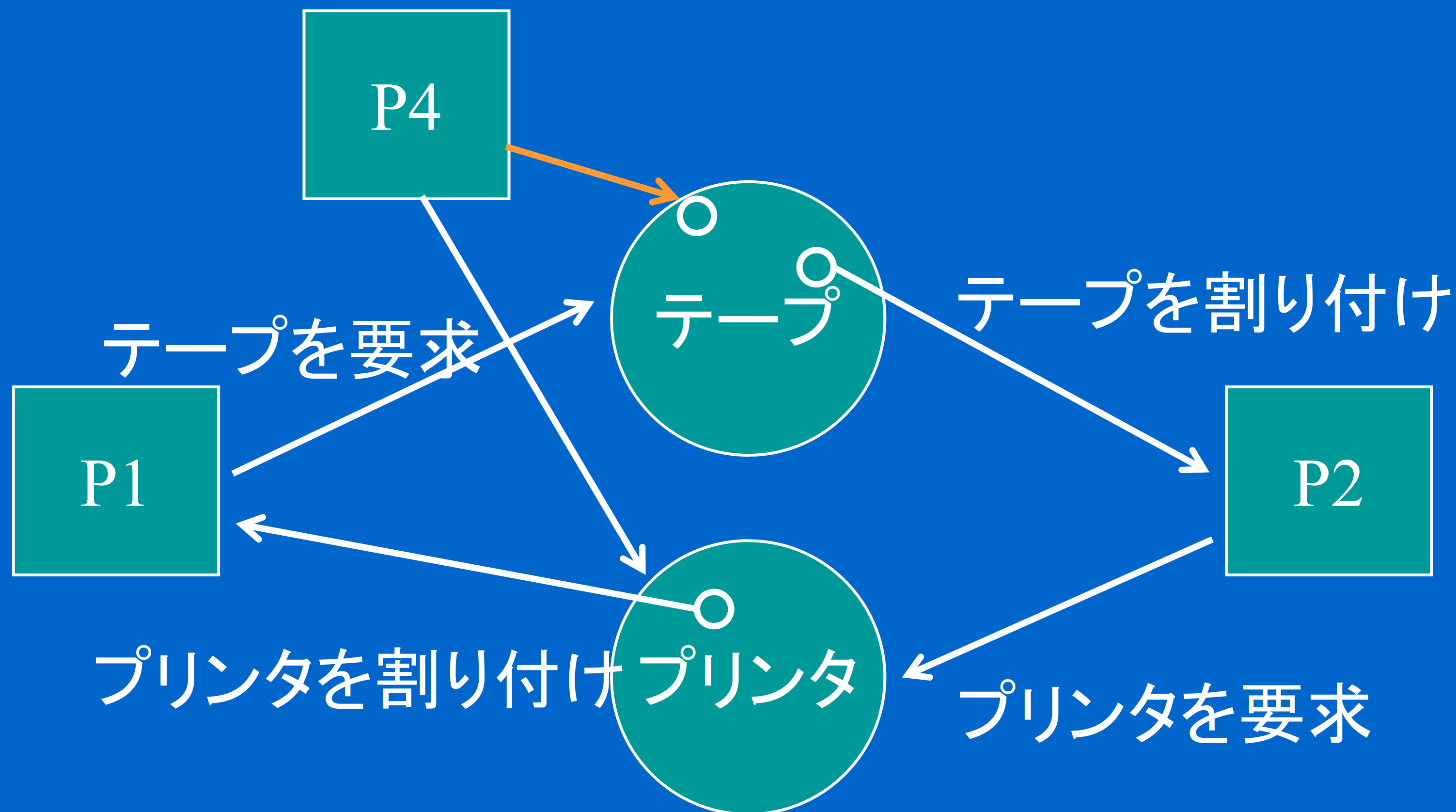
簡約の失敗1



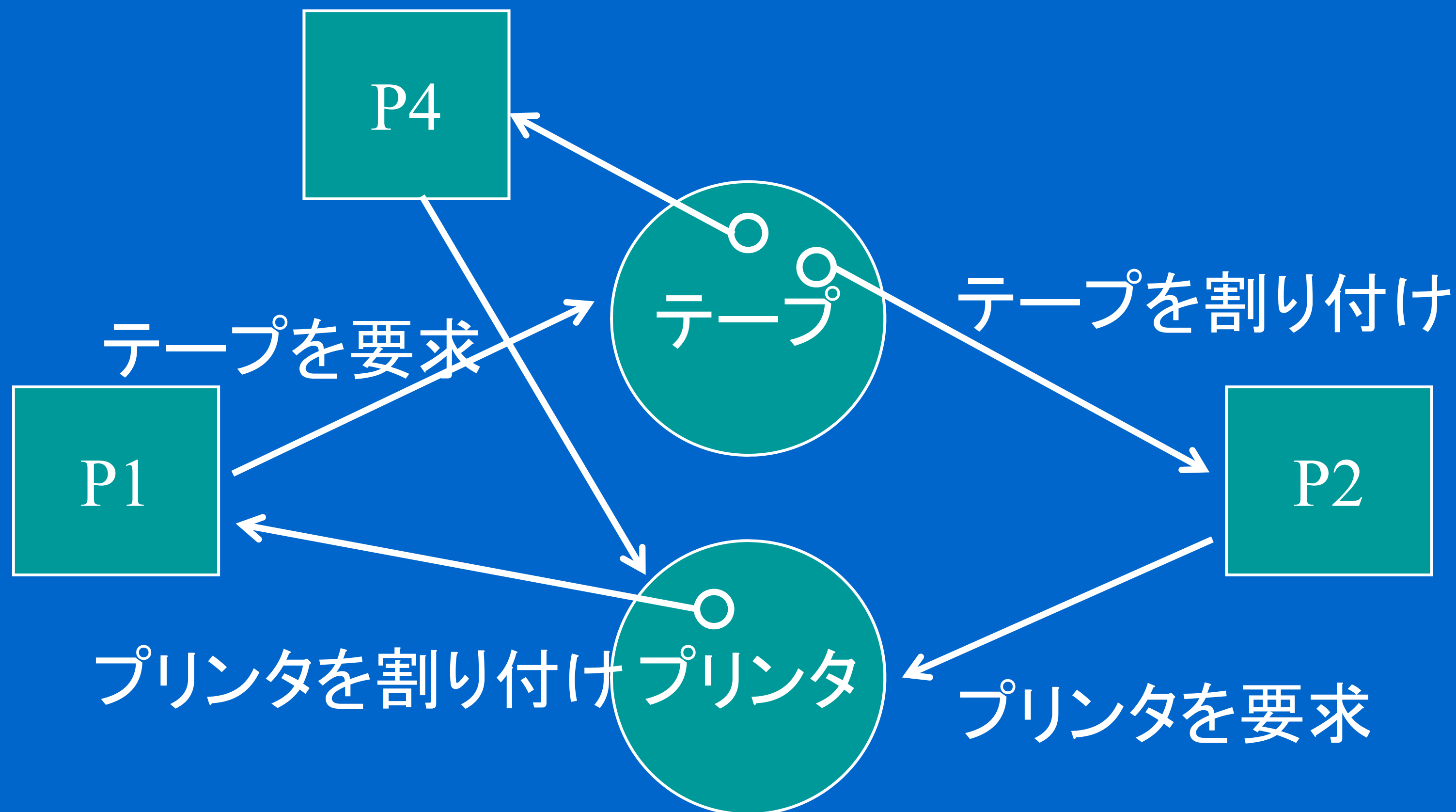
簡約の失敗2 (1)



簡約の失敗2 (2)



簡約の失敗2 (3)



デッドロックからの回復

- 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る
  - どのプロセスを中断させればよいかの判定は難しい
- 現在のOSでの実装
  - 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

デッドロックの回避

- デッドロックに陥らないことを保証する方法
  - 資源の割り付け／待ち状態を動的に調査
- 安全性
  - ある順序で資源を割り付け可能
  - デッドロックを回避可能

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

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)

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

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

例題　安全状態の検査(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、この状態は安全か？

例題　安全状態の検査(2)

	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

例題　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？

例題　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

例題　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**

例題　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？

例題　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

例題　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**