

オペレーティングシステム

資料 第6分冊(H27)



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

演習問題No.5の解説

/* P1(生産者) */

i = 1; /* 送信データ */ (1)

wait(s1); /* writeセマフォ */ (2)

buffer = i; /* 書き込み */ (3)

signal(s2); /* readセマフォ */ (4)

i = i+1; /* 送信データ */ (5)

wait(s1); /* writeセマフォ */ (6)

buffer = i; /* 書き込み */ (7)

signal(s2); /* readセマフォ */ (8)

/* P2(消費者) */

wait(s2); /* readセマフォ */ (9)

j = buffer; /* 読み出し */ (10)

signal(s1); /* writeセマフォ */ (11)

/* jを出力 */

wait(s2); /* readセマフォ */ (12)

j = buffer; /* 読み出し */ (13)

signal(s1); /* writeセマフォ */ (14)

/* jを出力 */

s1, s2はセマフォ(初期値はs1=1, s2=0)であり
bufferはプロセス1とプロセス2で共有されている変数
レディキューの先頭にP1, P2の順につながれている

演習問題No.5 (1)の解説

/* P1(生産者) */

i = 1; (1)

wait(s1); (2)

buffer = i; (3)

signal(s2); (4)

i = i+1; (5)

wait(s1); (6)

buffer = i; (7)

signal(s2); (8)

/* P2(消費者) */

wait(s2); (9)

j = buffer; (10)

signal(s1); (11)

/* jを出力 */

wait(s2); (12)

j = buffer; (13)

signal(s1); (14)

/* jを出力 */

実行 (1) (2) (3) (4) (5) (6) (9) (10) (11) (12) (7) (8) (13) (14)

s1 = 1 0 block(P1) restart(P1) 1

s2 = 0 1 0 block(P2) restart(P2)

演習問題No.5 (2)の解説

/* P1(生産者) */

i = 1; (1)

wait(**s1**); (2)

buffer = i; (3)

signal(**s1**); (4)

i = i+1; (5)

wait(**s1**); (6)

buffer = i; (7)

signal(**s1**); (8)

/* P2(消費者) */

wait(**s1**); (9)

j = buffer; (10)

signal(**s1**); (11)

/* jを出力 */

wait(**s1**); (12)

j = buffer; (13)

signal(**s1**); (14)

/* jを出力 */

実行	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)
s1 =	1	0	1	1	0		1	0		1	0			1

演習問題No.5 (3)の解説

/* P1(生産者) */

i = 1; (1)

wait(**s1**); (2)

buffer = i; (3)

signal(**s2**); (4)

i = i+1; (5)

wait(**s1**); (6)

buffer = i; (7)

signal(**s2**); (8)

/* P2(消費者) */

wait(**s1**); (9)

j = buffer; (10)

signal(**s2**); (11)

/* jを出力 */

wait(**s1**); (12)

j = buffer; (13)

signal(**s2**); (14)

/* jを出力 */

実行 (1) (2) (3) (4) (5) (6) (9)

s1 = 1 0 block(P1) block(P2)

s2 = 0 1

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
 - 受信側の都合に関係なく送信は完了する(非同期通信)

通信の形

- センダーレシーブ(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
 - 異なる計算機間では困難

間接指定方式

- 論理的な通信媒体を指定
 - 共用バッファ
 - パイプ
- 通信媒体
 - メッセージを送り付ける先
 - メッセージを取り出す対象
- 送受信の際には通信媒体を指定する

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

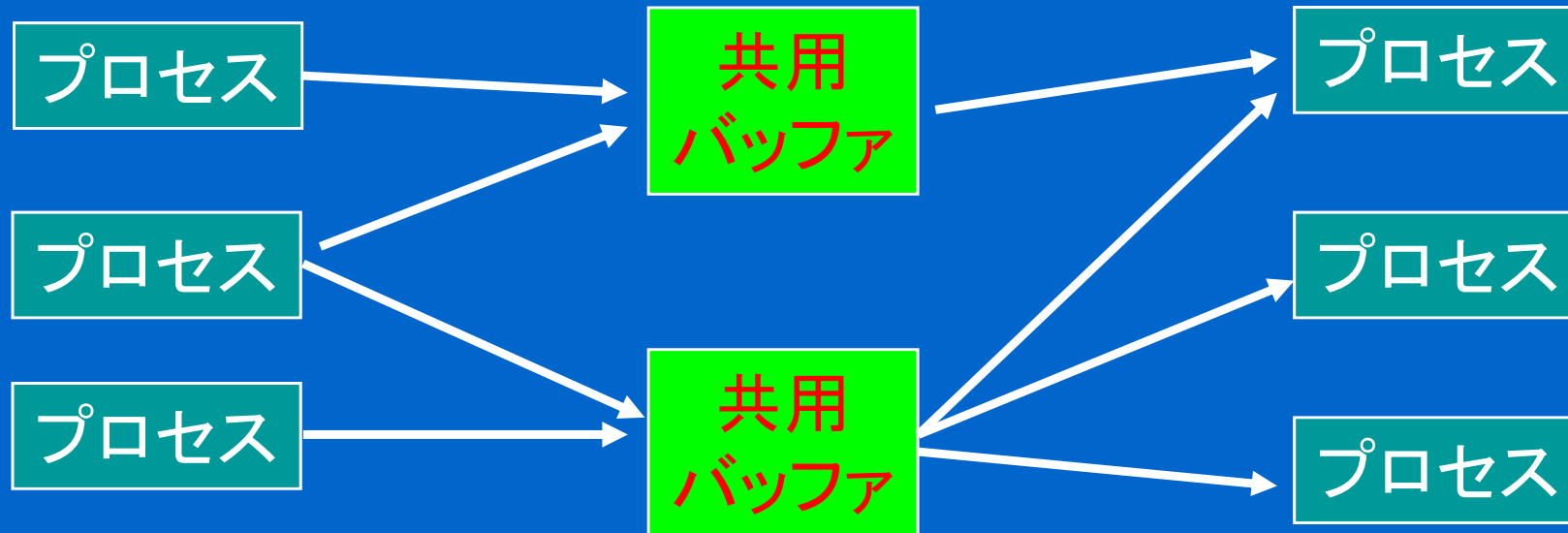
サーバ・クライアントモデル

- 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特定し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

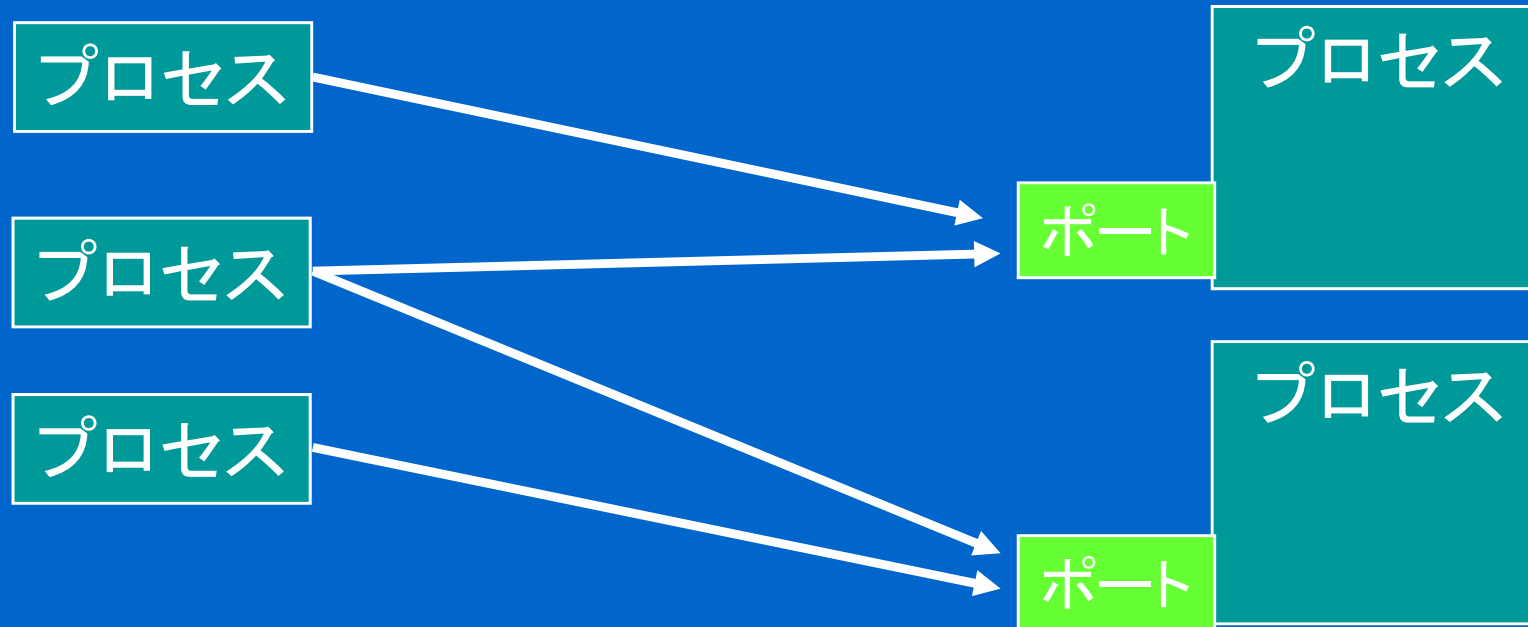
共用バッファを使った プロセス間通信(図2.32)



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

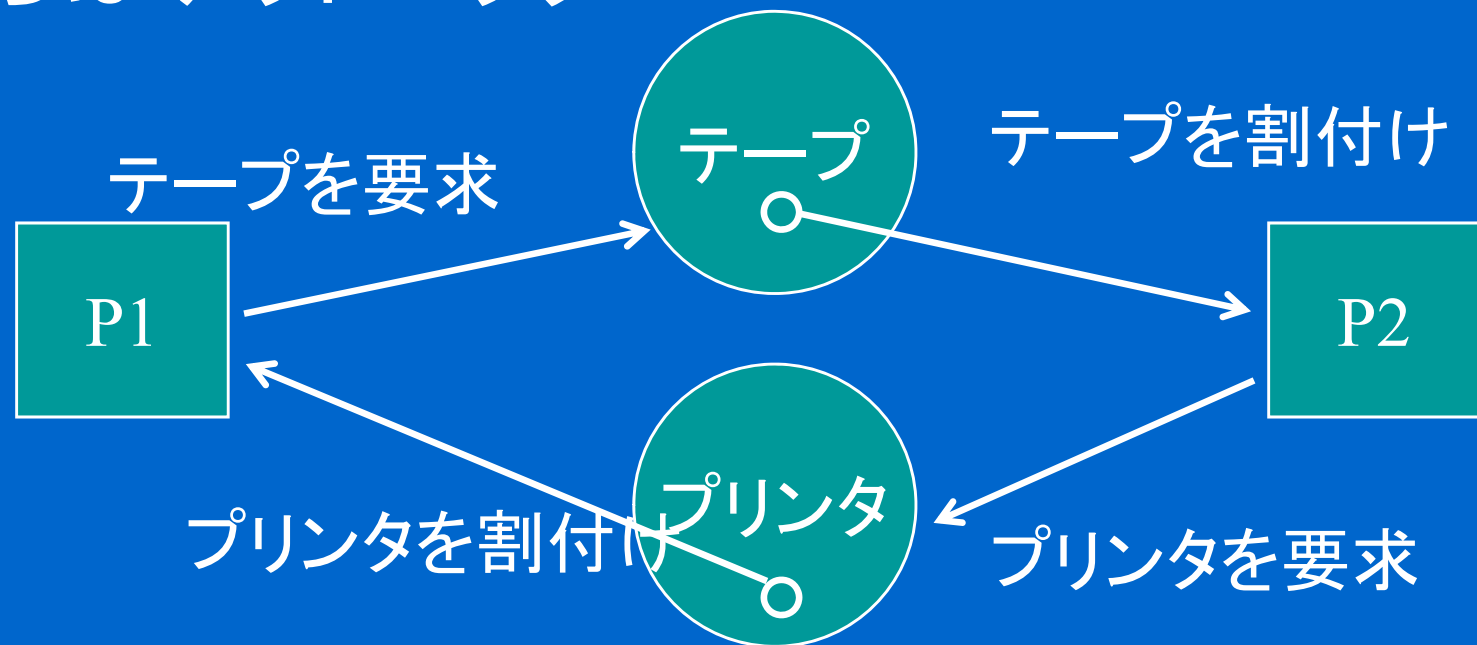
	資源1	資源2
プロセスP1	使用中	割り付け待ち
プロセスP2	割り付け待ち	使用中

デッドロックとOS

- OSの仕事:
 - デッドロックを起こしてい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

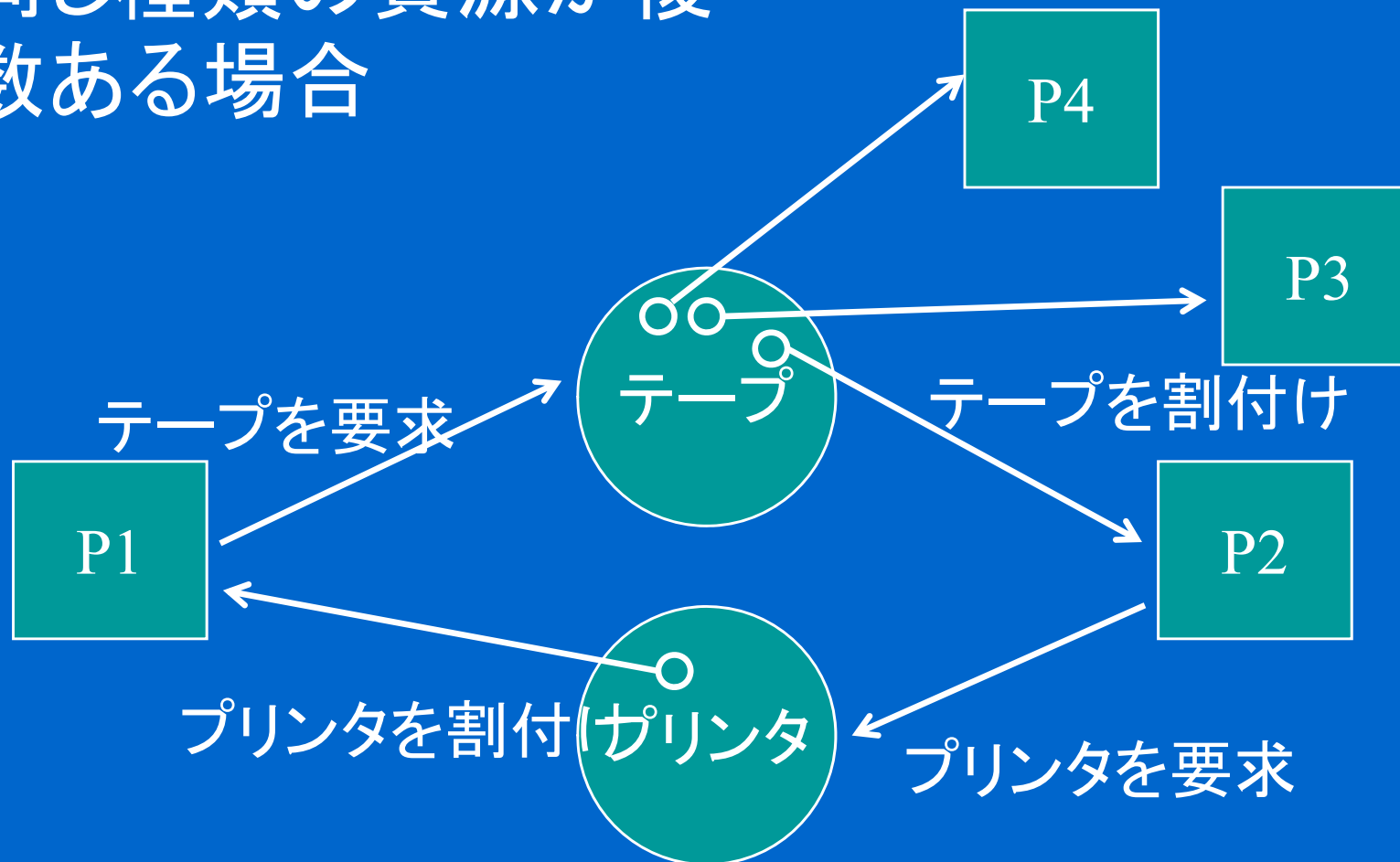
資源割り付けグラフ(1)

- 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)
 - $\bigcirc \rightarrow \square$: 割り付け、 $\square \rightarrow \bigcirc$: 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



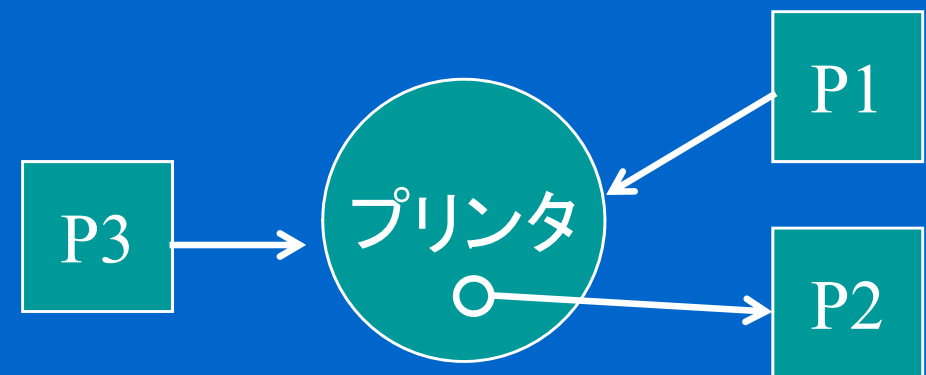
飢餓状態

- デッドロックに似た状況
- 待っている対象の資源は開放されるが、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

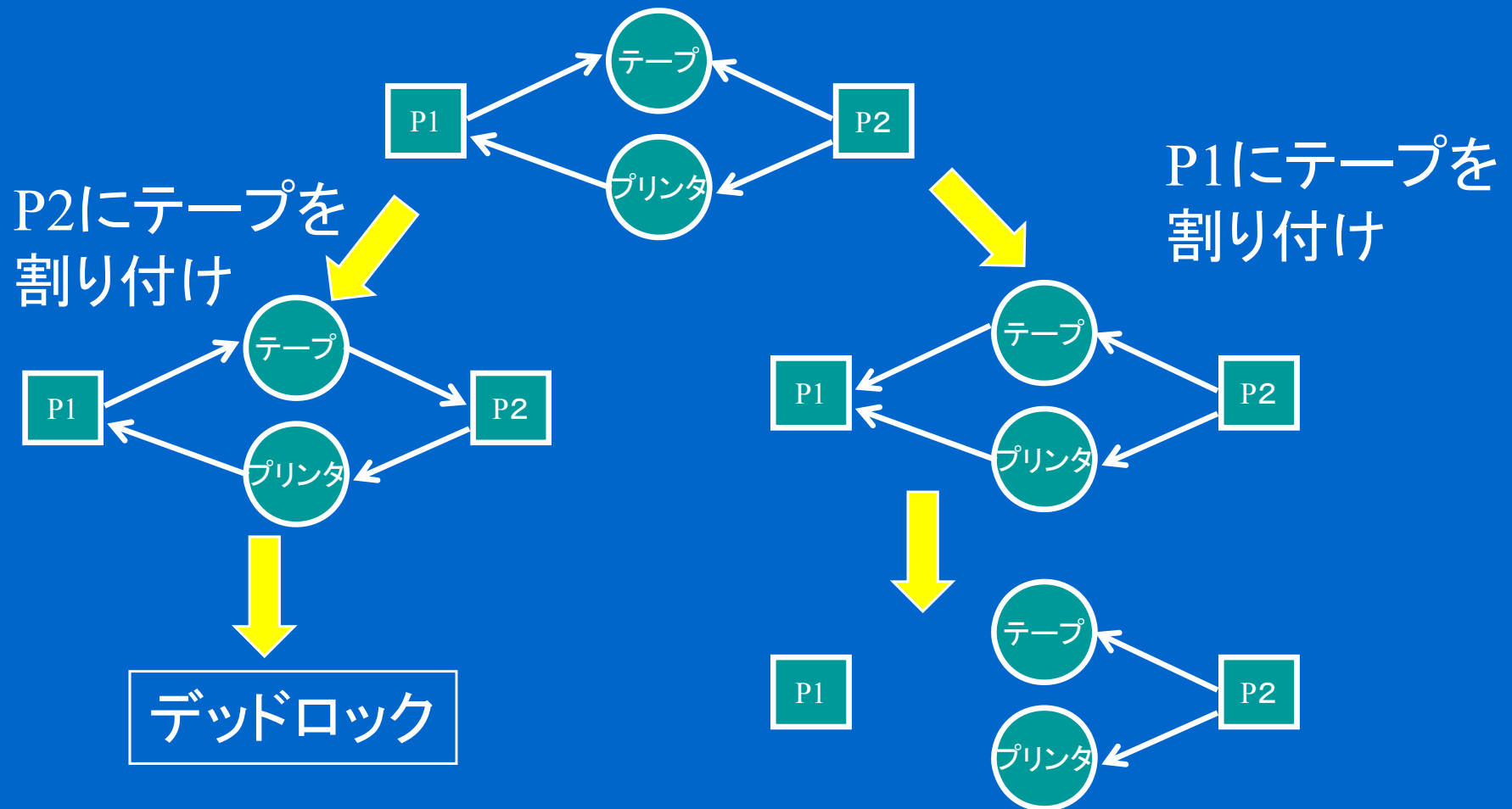
(2) 待ちの防止：必要な資源は全部をまとめて要求する

(3) 横取りなしの禁止：資源を要求して割り付けられなかったら、待ち状態に入るのではなく不許可とし、すで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

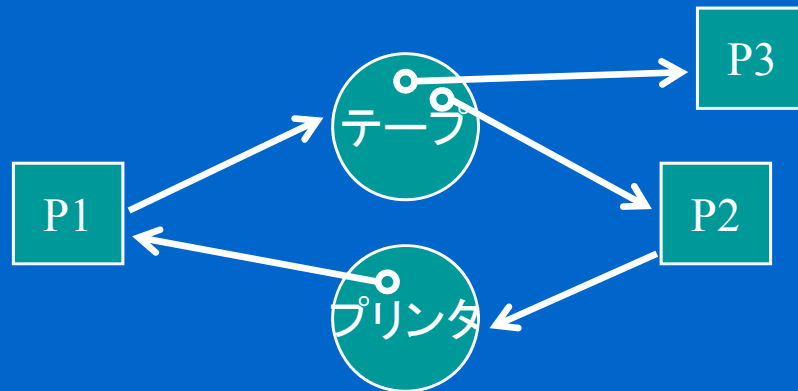
- 資源割り付け要求に対するスケジューリング
 - 資源を要求するプロセスが複数同時に存在する場合に、どのプロセスから先に資源を割り付けるか
- スケジューリング次第で、デッドロックに陥ったり、デッドロックを回避できたりする
- どのようにスケジューリングしてもデッドロックが不可避免な場合がある

デッドロックの検出

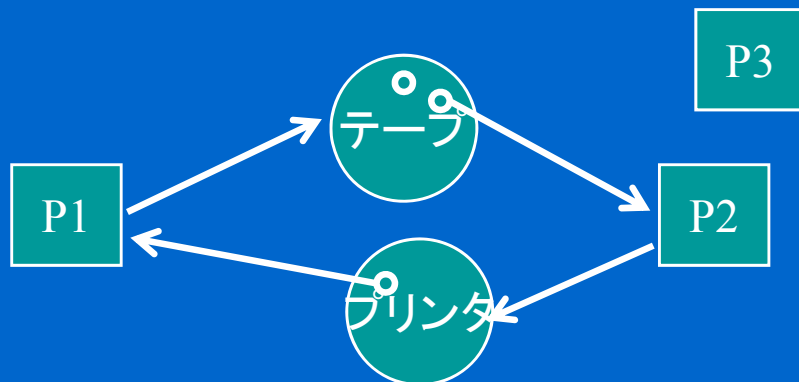
- デッドロックの検出 (deadlock detection)
 - デッドロックの存在を検知する
 - デッドロックに関与しているプロセスと資源を特定する
- 資源割り付けグラフに閉路
 - デッドロックの必要条件(十分条件ではない)
- 資源割り付けグラフの簡約を行って判定

資源割り付けグラフの閉路

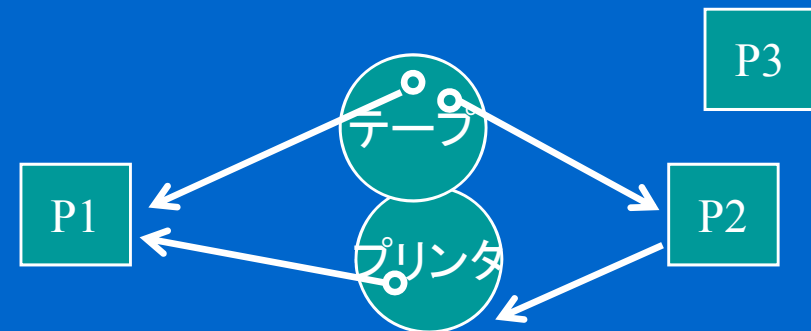
- 閉路のある資源割り付けグラフ



- P3が終了



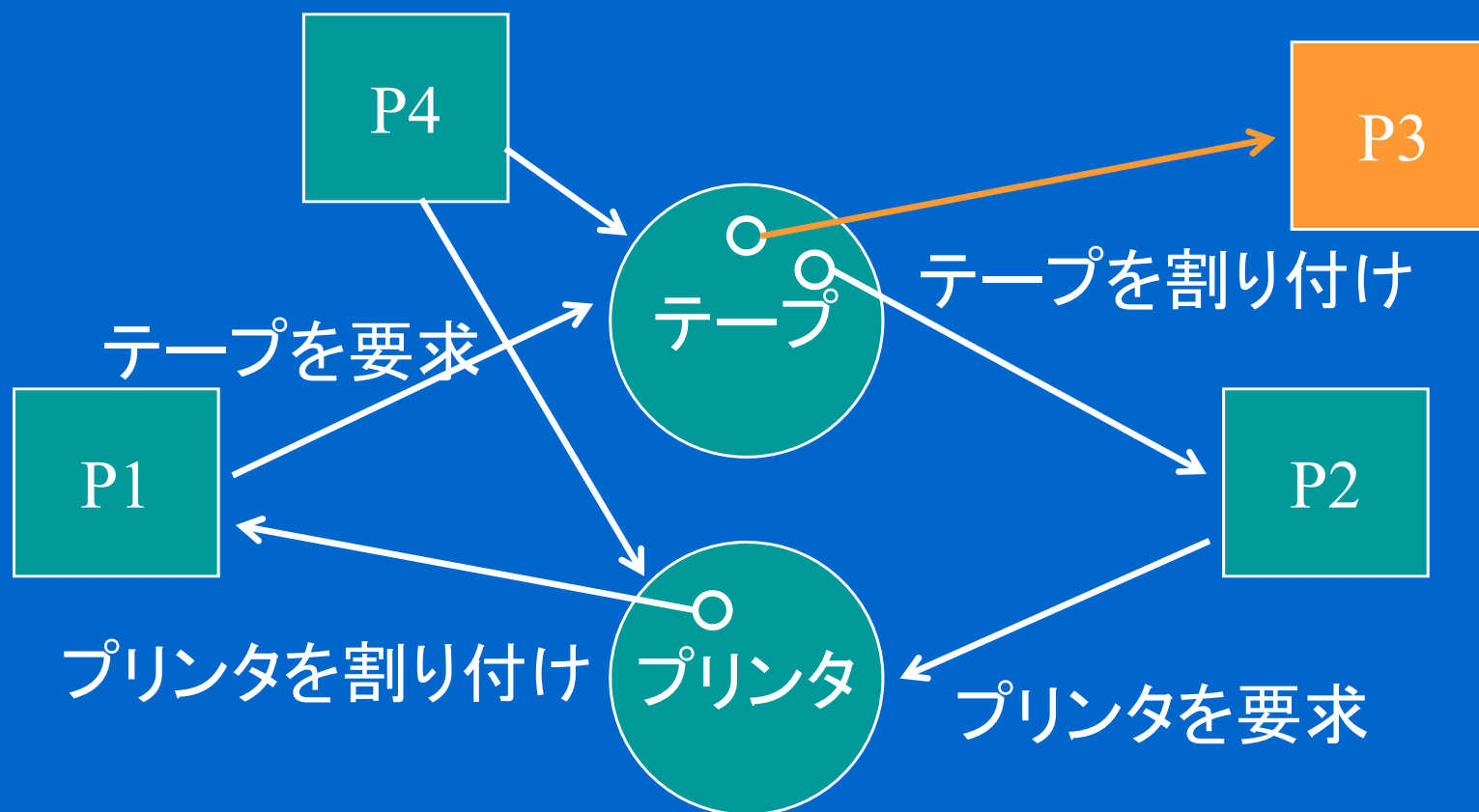
P1にテープ割り付け可



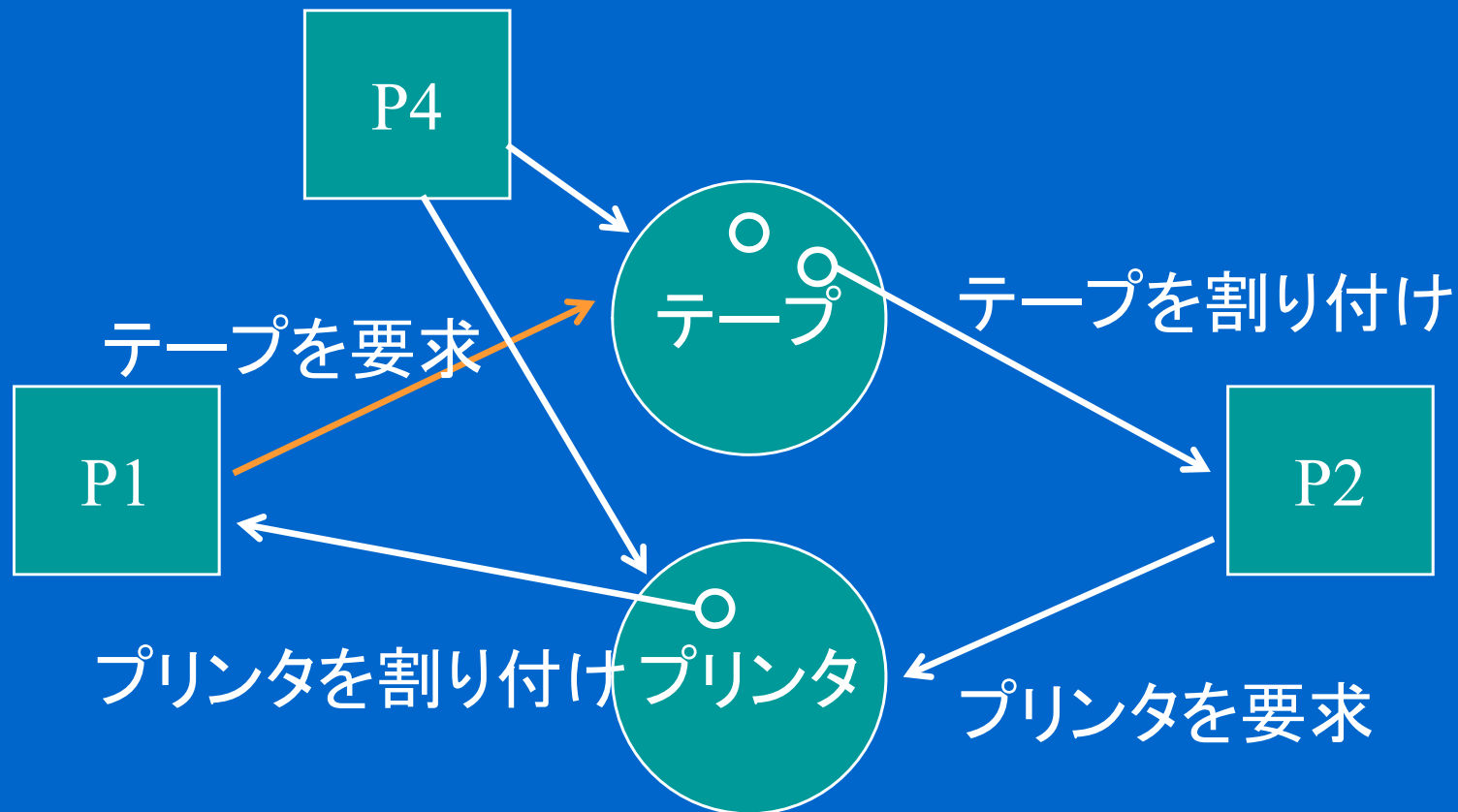
デッドロックの判定

- 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

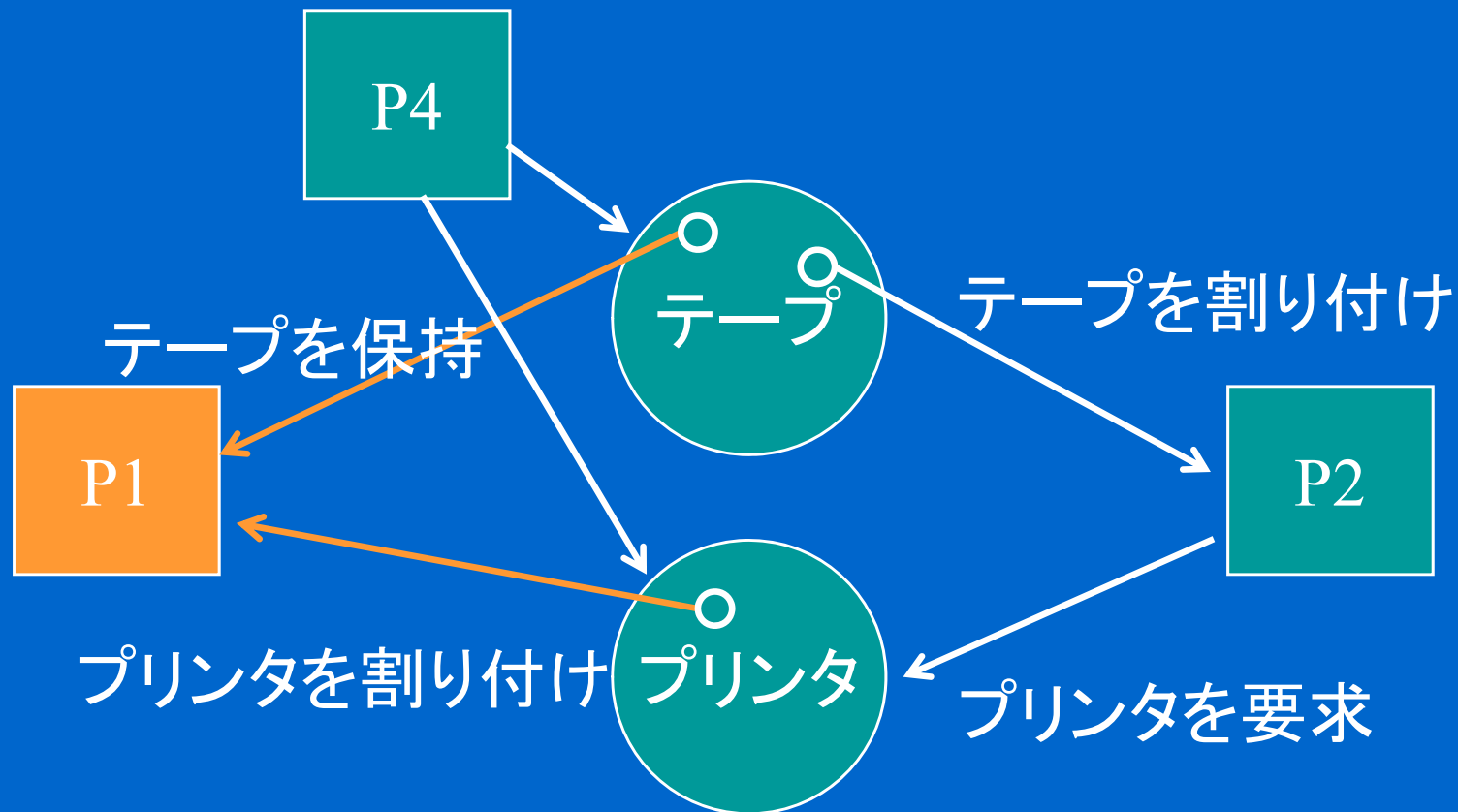
グラフの簡約(1)



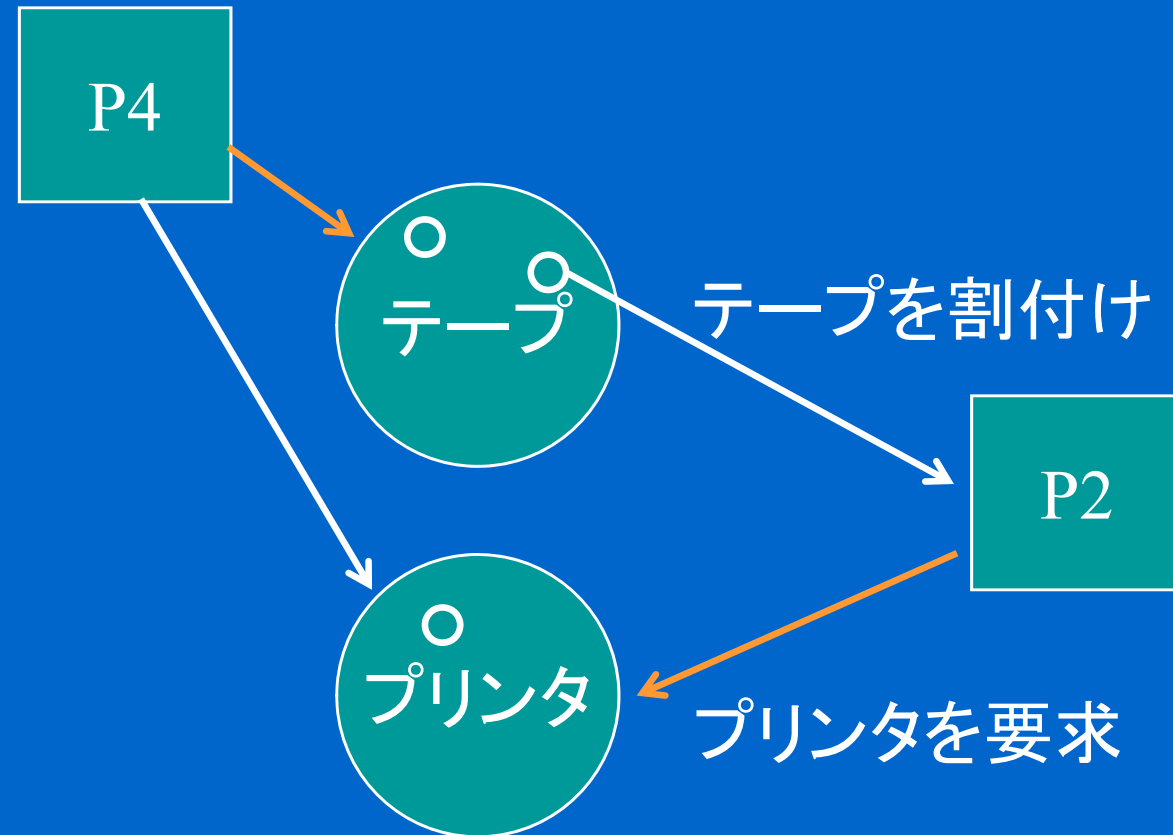
グラフの簡約(2)



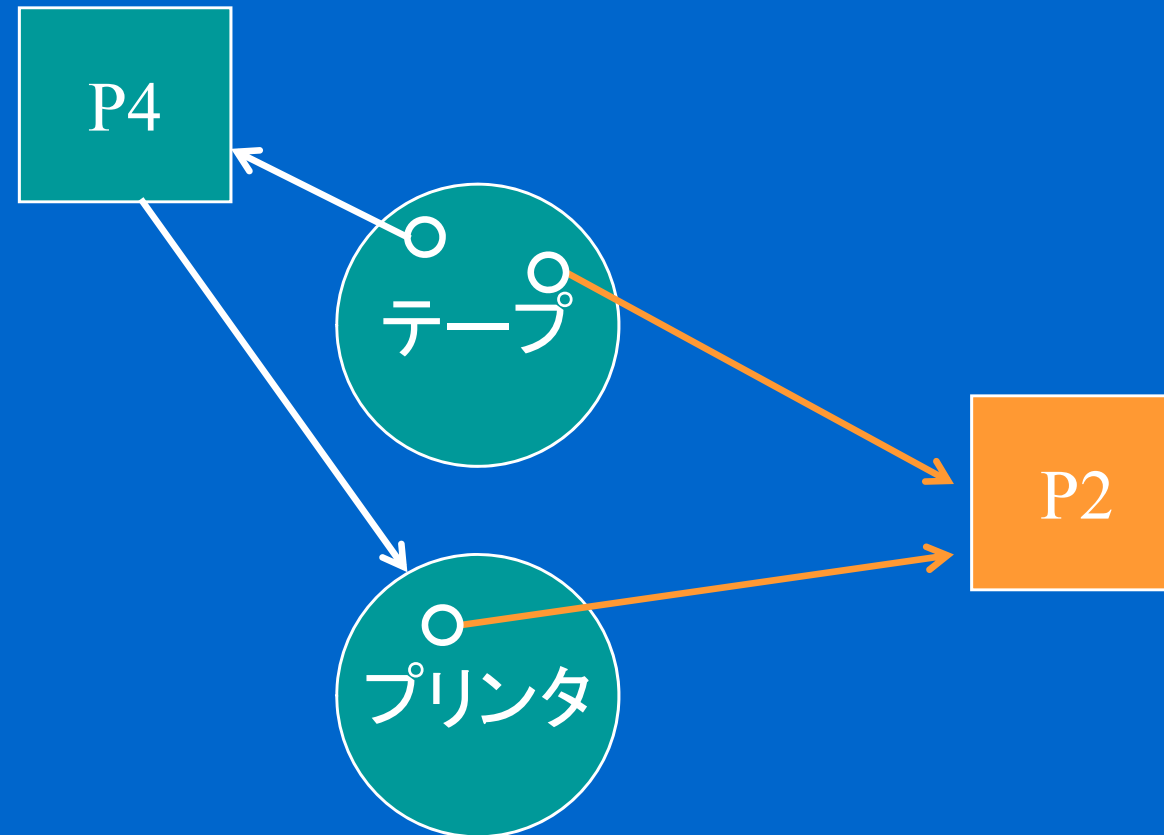
グラフの簡約(3)



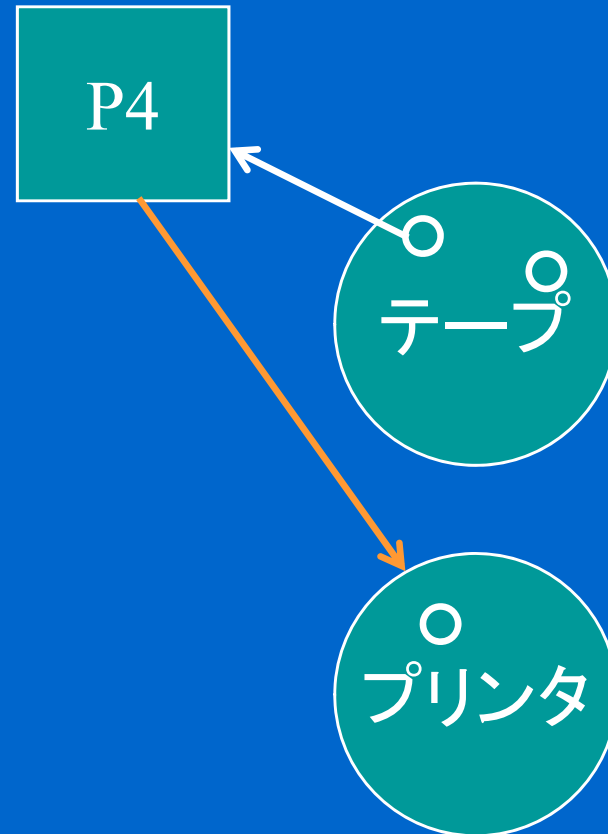
グラフのリダクション



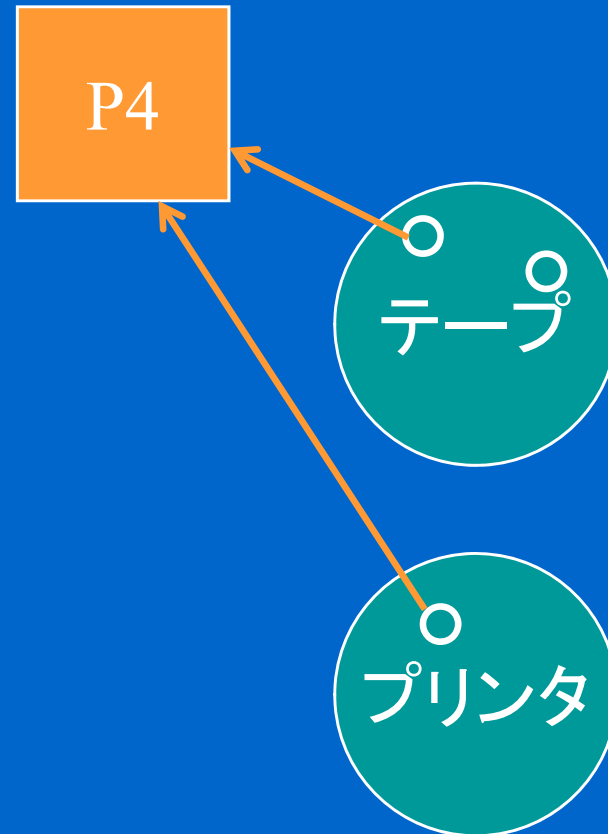
グラフのリダクション



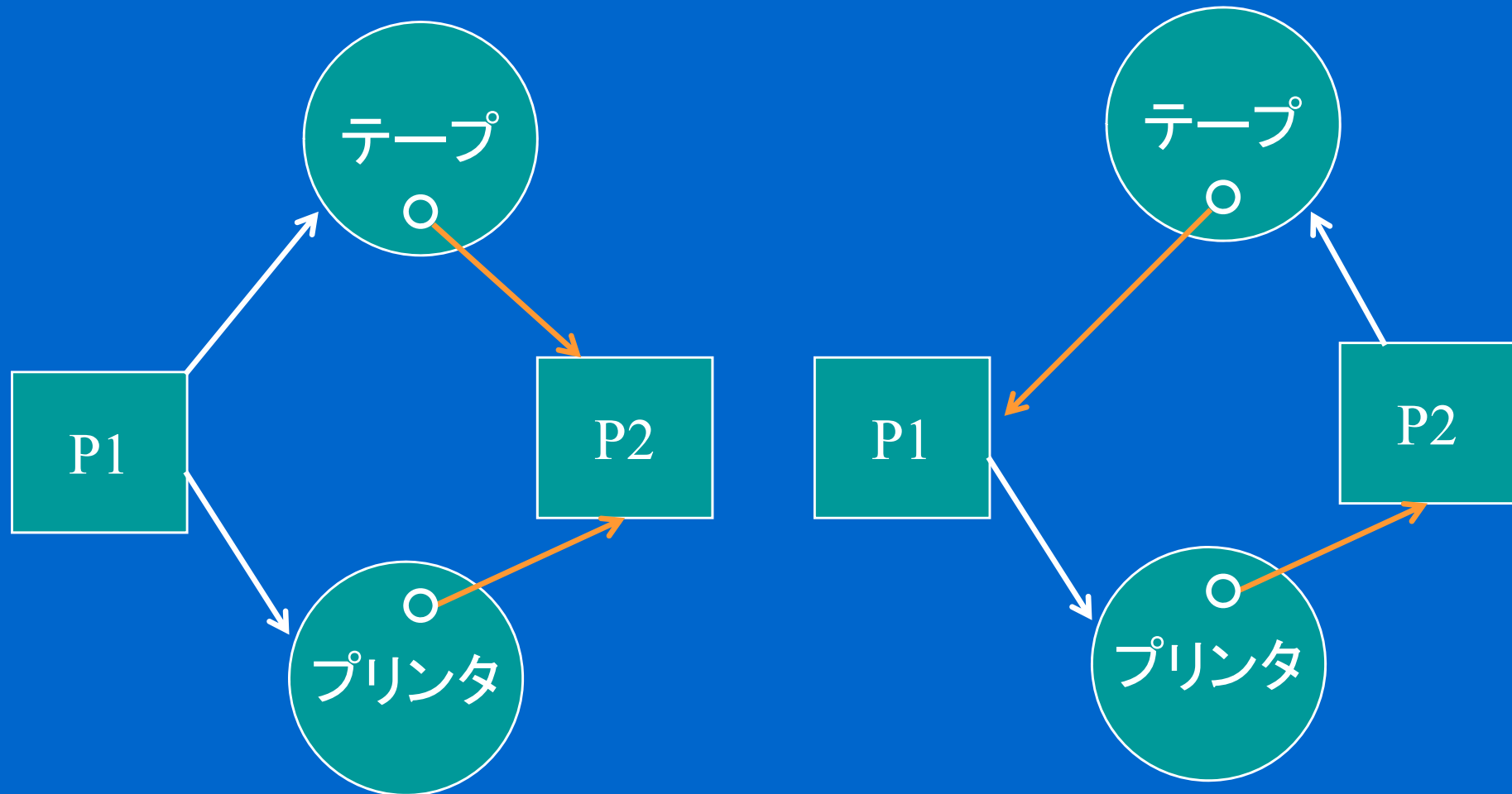
グラフのリダクション



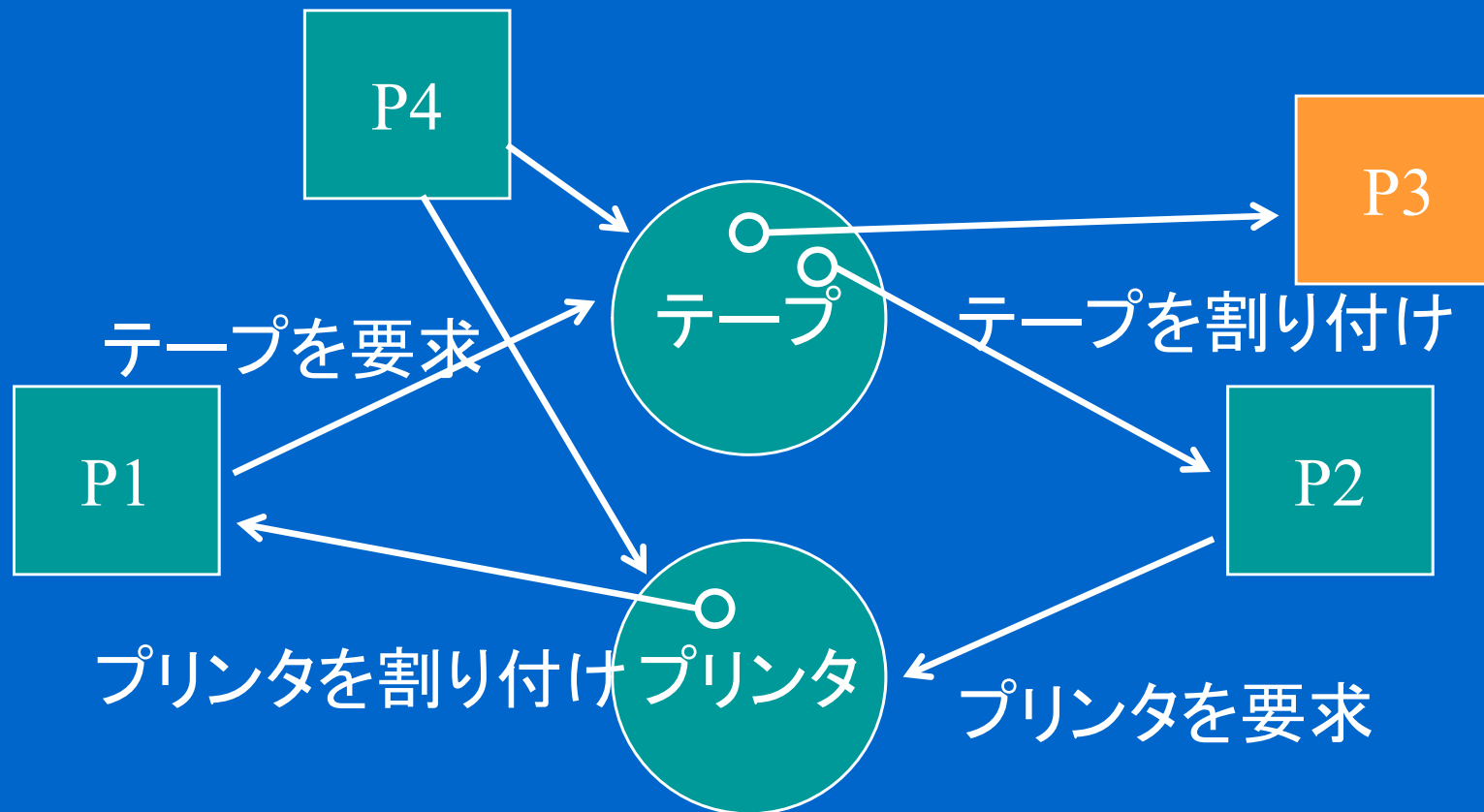
グラフのリダクション



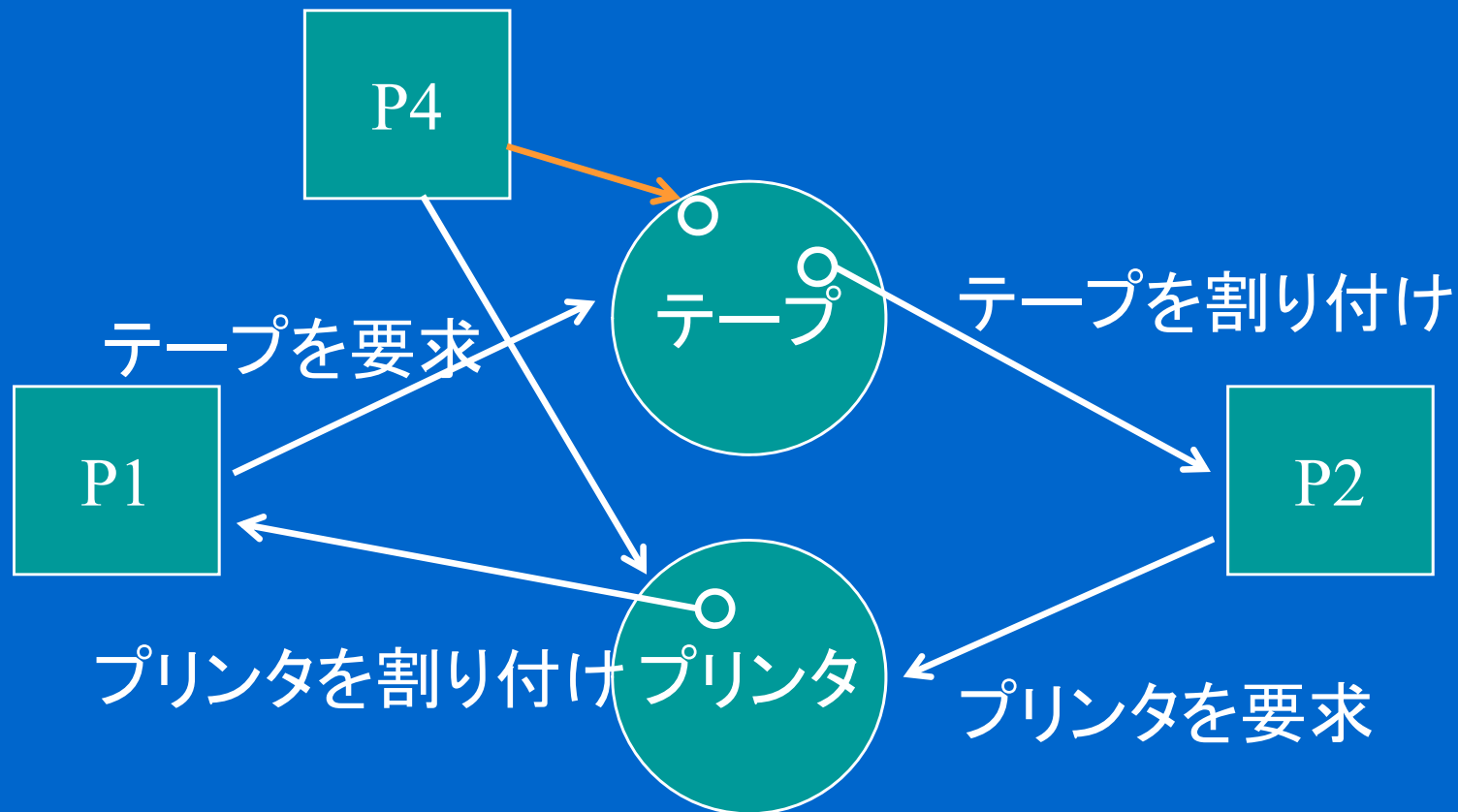
簡約の失敗1



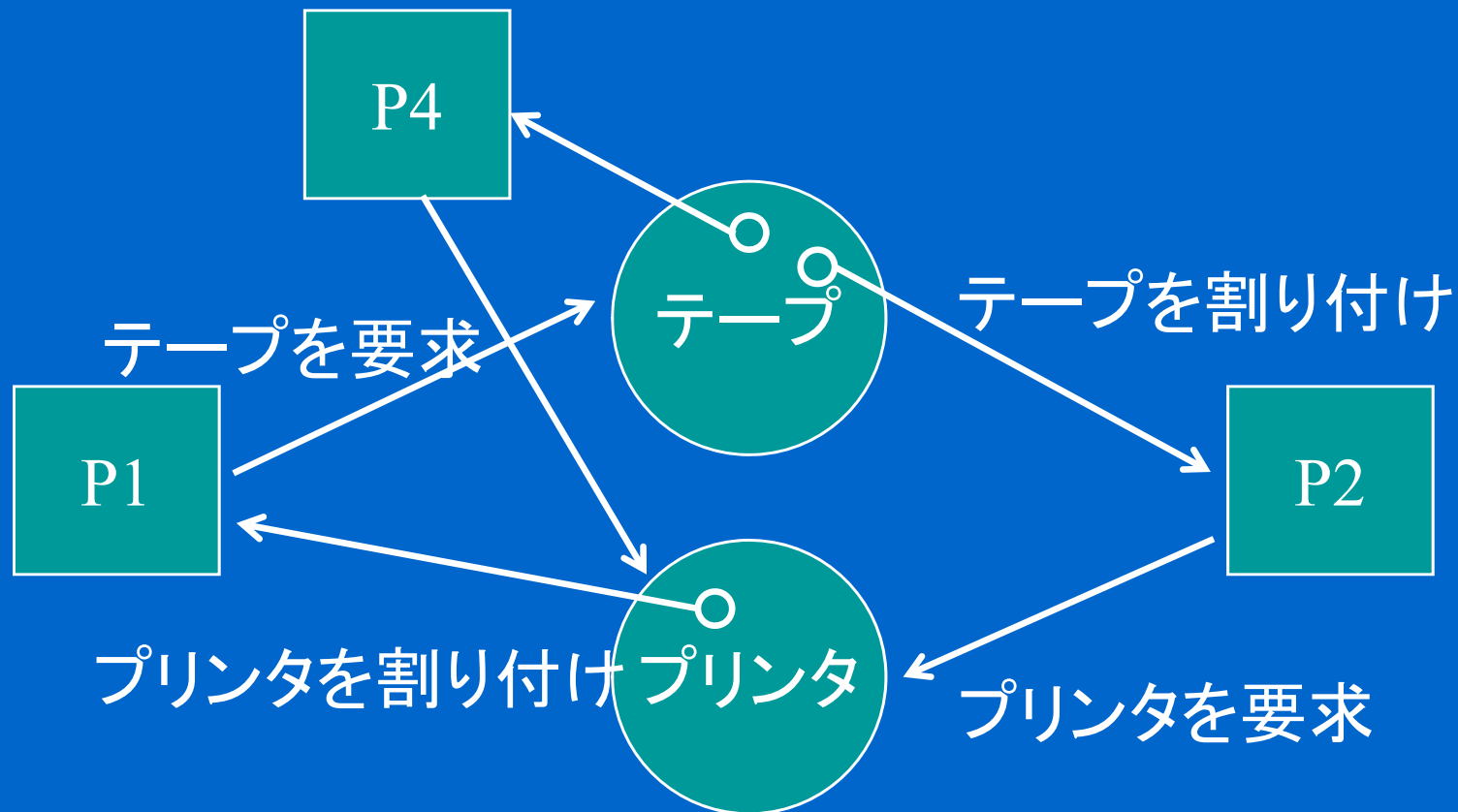
簡約の失敗2 (1)



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. $Work(j) \leftarrow Available(j)$ ($1 \leq j \leq m$);
 $Finish(i) \leftarrow false$ ($1 \leq i \leq n$);
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 $Need(i,j) \leq Work(j)$
この条件を満たす i が見つければ次へ、なければステップ4へ
3. $Work(j) \leftarrow Work(j) + Allocation(i,j)$ ($1 \leq j \leq m$);
 $Finish(i) \leftarrow true$;
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
 $\text{Request}(i,j) \leq \text{Need}(i,j)$ ($1 \leq j \leq m$) 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. $\text{Request}(i,j) \leq \text{Available}(i,j)$ ($1 \leq j \leq m$) 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
 $\text{Available}(j) \leftarrow \text{Available}(j) - \text{Request}(i,j)$ ($1 \leq j \leq m$);
 $\text{Allocation}(i,j) \leftarrow \text{Allocation}(i,j) + \text{Request}(i,j)$ ($1 \leq j \leq m$);
 $\text{Need}(i,j) \leftarrow \text{Need}(i,j) - \text{Request}(i,j)$ ($1 \leq j \leq m$);
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 <i>j</i>	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 <i>j</i>	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 <i>j</i>	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 <i>j</i>	1	2	3	1	2	3	1	2	3	1	2	3
プロセス										<u>2</u>	<u>2</u>	<u>1</u>
P1	<u>3</u>	<u>1</u>	<u>1</u>	3	2	2	<u>0</u>	<u>1</u>	<u>1</u>			
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 <i>j</i>	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 <i>j</i>	1	2	3	1	2	3	1	2	3	1	2	3
プロセス										<u>0</u>	<u>0</u>	<u>1</u>
P1	2	0	0	3	2	2	1	2	2			
P2	2	1	1	2	2	2	0	1	1			
P3	<u>3</u>	<u>3</u>	<u>3</u>	4	3	3	<u>1</u>	<u>0</u>	<u>0</u>			

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 <i>j</i>	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
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