



ESTGF

ESCOLA SUPERIOR DE TECNOLOGIA  
E GESTÃO DE FELGUEIRAS  
POLITÉCNICO DO PORTO

---

# Situações de Bloqueio (*deadlocks*)

## Sistemas Operativos

Licenciatura em Engenharia Informática  
2007/2008

---

António Pinto (apinto@estgf.ipp.pt)

Baseado na bibliografia da disciplina

# Sumário

---

- Modelo sistêmico
- Condições necessárias para ocorrência *deadlocks*
- Estratégias de tratamento de *deadlocks*

# Contexto

- Conjunto de processos concorrentes que
  - Detêm acesso exclusivo a recursos que outros necessitam
  - Necessitam de recursos que outros detêm exclusivamente
- Processos envolvidos tipicamente entram em bloqueio permanente
- Uns impedem os outros de executar, e vice-versa
- Recursos podem ser físicos (impressora, espaço de memória, ...) ou lógicos (secção crítica, ficheiro, ...)

# Contexto

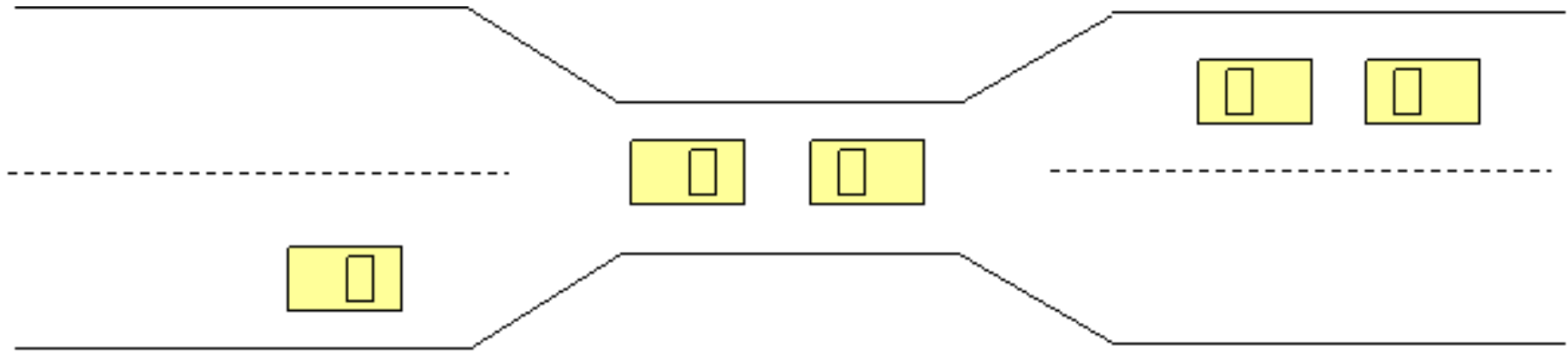
## - Exemplo: Semáforos

- Assumindo dois processos (P1 e P2) que partilham dois semáforos (A e B)
- Assumindo ainda que os semáforos foram inicializados a 1, e que os processos têm o seguinte código:

P1	P2
<code>down(A);</code> <code>down(B);</code>	<code>down(B);</code> <code>down(A);</code>

# Contexto

## - Exemplo: Travessia de uma ponte estreita



- Trânsito num só sentido de cada vez
- Situação de bloqueio (*deadlock*) pode ser resolvida se um carro recuar
- Poderá implicar que vários carros recuem em caso de bloqueio
- Existe possibilidade de mingua (*starvation*)

Preempção de recursos + *Rollback*

# Modelo sistémico

- Sistema consiste num número finito de recursos, distribuídos por um conjunto de processos concorrentes
- Recursos agrupados por tipos
  - $R1, R2, \dots, Rn$
  - Segmentos de memória, ciclos de CPU, ficheiros, dispositivos IO, ...
- Cada tipo de recurso poderá conter várias instâncias
  - $W1, W2, \dots, Wn$

# Modelo sistémico

- Sequência normal de utilização de recursos
  - Solicitação (*Request*)
    - `open()`, `fopen()`, `malloc()`, `alloc()`, ...
  - Utilização (*Use*)
    - `read()`, `write()`, `fseek()`, `fread()`, ...
  - Disponibilização (*Release*)
    - `close()`, `fclose()`, `free()`, ...
- Imposta pelo sistema operativo por intermédio de *systems calls*
  - `fread()` dá erro se for usado sem o prévio `fopen()`

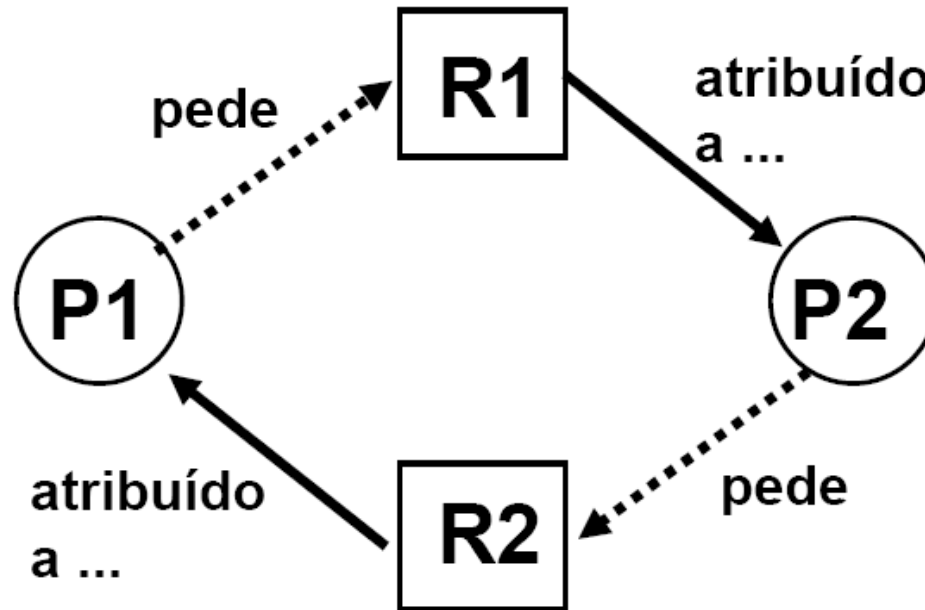
# Condições necessárias para ocorrência *deadlocks*

1. Exclusão mútua (só um processo pode deter um determinado recurso em cada instante de tempo)
2. Retenção e espera (processo detentor de um recurso, espera por mais recursos detidos por outros processos)
3. Não preempção (recursos só podem ser libertados voluntariamente pelo processo detentor, quando terminar de o utilizar)
4. Espera circular
  - Deve existir um conjunto de processos (P1, P2, ..., Pn) de tal forma que:
    - P1 aguarda pela libertação de um recurso detido por P2;
    - P2 aguarda pela libertação de um recurso detido por P3;
    - ...;
    - Pn aguarda pela libertação de um recurso detido por P1



# Condições necessárias para ocorrência *deadlocks*

- Espera circular



# Condições necessárias para ocorrência *deadlocks*

- *Deadlock* ocorre se, e só se, a condição de espera circular não tiver solução
- Condição de espera circular não têm solução quando se verificam as condições anteriores
- Três primeiras condições são necessárias, mas não suficientes para que se dê um situação de bloqueio

# Estratégias de tratamento de *deadlocks*

- Prevenir
  - Assegurar que pelo menos uma das 4 condições nunca se verifica
- Evitar
  - Não conceder recursos a um processo se tal se traduzir na possibilidade de bloqueio
- Detectar e recuperar
  - Conceder recursos sempre que solicitados e disponíveis
  - Periodicamente, verificar se existem situações de bloqueio
  - Caso existam, resolver a situação
- Ignorar o problema
  - Solução mais usual nos SO actuais 😊

# Prevenir *deadlocks*

- Garantir que pelo menos uma condição não se verifica
- Exclusão mútua
  - Muito difícil, se não impossível, obrigando à utilização de apenas recursos partilháveis
  - Certos recursos não podem ser partilhados
  - Existem técnicas que permitem minimizar, mas não eliminar, o problema
    - Impressoras e *spooling*



Espaço em disco não é ilimitado

# Prevenir *deadlocks*

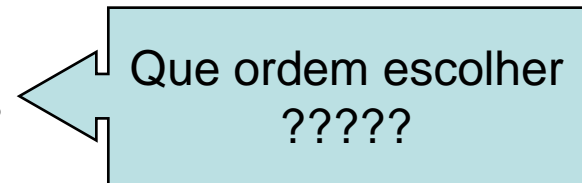
- Retenção e espera
  - Garantir que quando um processo requisita um recurso não detêm mais nenhum
  - Requisitar todos os recursos antes de começar a executar
  - Requisitar recursos incrementalmente, libertando-os quando não conseguir requisitar mais
  - Leva à fraca utilização dos recursos
  - Implica o conhecimento prévio de todos os recursos necessários (não se adapta a sistemas interactivos)
  - Mantém-se a possibilidade de mingua (*starvation*)

# Prevenir *deadlocks*

- Não preempção
  - Possibilitar a preempção de recursos
  - Libertação de todos os recursos no momento da recusa de uma solicitação de recurso
  - Forçar à libertação de recursos solicitados por outros processos
  - Impossível se não se poder guardar o estado do recurso temporariamente (ex.: impressora, gravador de DVD, ...)

# Prevenir *deadlocks*

- Espera circular
  - Tipos de recursos são ordenados, sendo também solicitados pela mesma ordem
  - Ex.: (1) Memórias, (2) Ficheiros, (3) Impressoras
  - Caso um processo solicite um recurso do tipo Ficheiros, já só poderá solicitar recursos do tipo Impressoras
  - Caso um processo necessita de várias instâncias do mesmo recursos, deve solicitá-las todas de uma só vez
  - Ineficiente por impor uma sequência de solicitação de recursos, levando à negação desnecessária de acesso a recursos



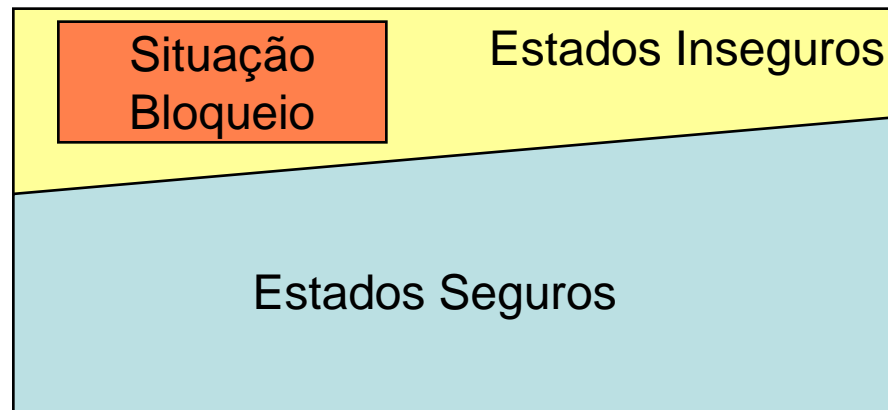
# Evitar *deadlocks*

- Permitir que as condições se verifiquem normalmente
- Analisar cada solicitação de recursos, caso a atribuição leve a uma situação de bloqueio, deve ser negada
- Examinar dinamicamente o estado global de alocação de recursos por forma a garantir que esperas circulares não sejam possíveis
- Duas estratégias para evitar *deadlocks*
  1. Não iniciar a execução de um processo se as suas necessidades (+ as já existentes) propiciar uma situação de bloqueio
  1. Não atribuir recursos adicionais, se isso propiciar uma situação de bloqueio



# Evitar *deadlocks*

- Conceito base para evitar situações de bloqueio é o conceito de estado seguro
- Se o sistema é capaz de atribuir recursos a cada processo de forma a evitar situações de bloqueio, então diz-se que está num estado seguro
- Estado inseguro, é o que pode conduzir a uma situação de bloqueio



# Evitar *deadlocks*

- 1ª estratégia
  - Início de execução de um novo processo é negado
  - Se as necessidades máximas de recursos do novo processo, mais as necessidade máximas dos processos já existentes
  - Não forem capazes de serem satisfeitas por qualquer um dos recursos
  - Estratégia simples de implementar mas muito restritiva
  - Leva a um fraco aproveitamento do sistema na sua globalidade

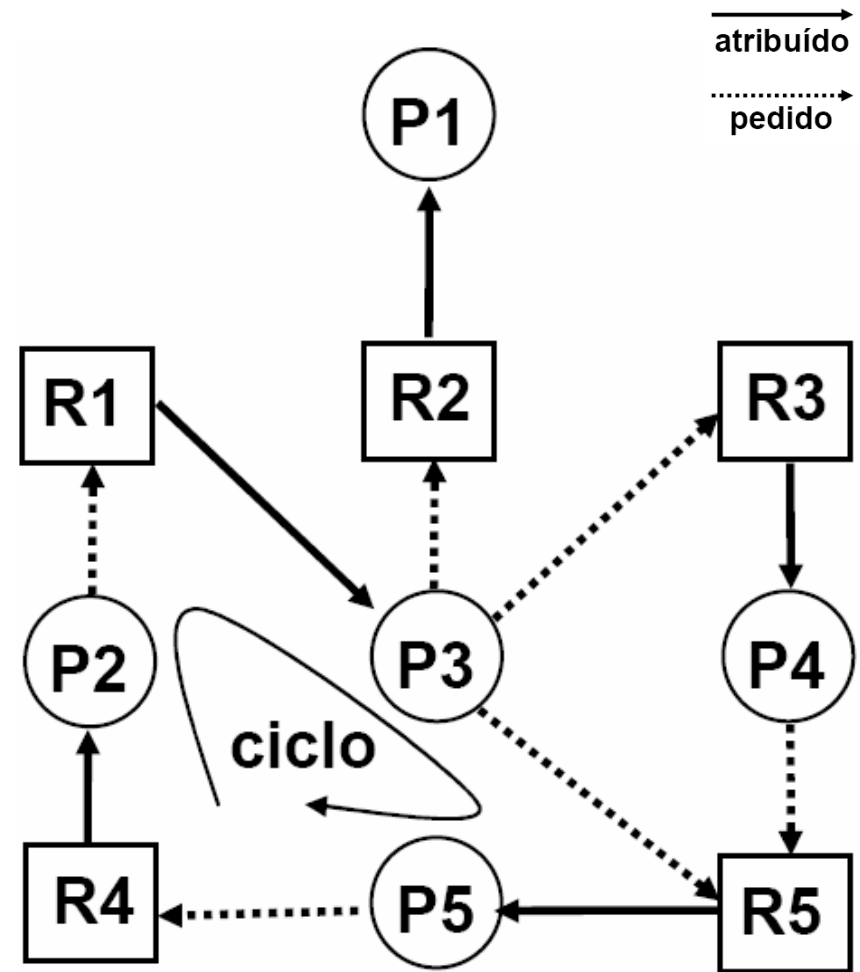
# Evitar *deadlocks*

- 2ª estratégia
  - Não aceitar novas solicitações de recursos se isso possibilitar uma situação de bloqueio
  - Estratégia mais otimizada mas também mais complexa
  - Solução usual passa pelo teste de estado seguro
    - Verificar se o estado actual é ou não um estado seguro
  - Cenários de tipos de recursos com uma única instância, recorre-se a um grafo de alocação de recursos
  - Cenários de múltiplas instâncias por tipo de recurso, recorre-se ao algoritmo do banqueiro

# Evitar *deadlocks*

## - Grafo de alocação de recursos

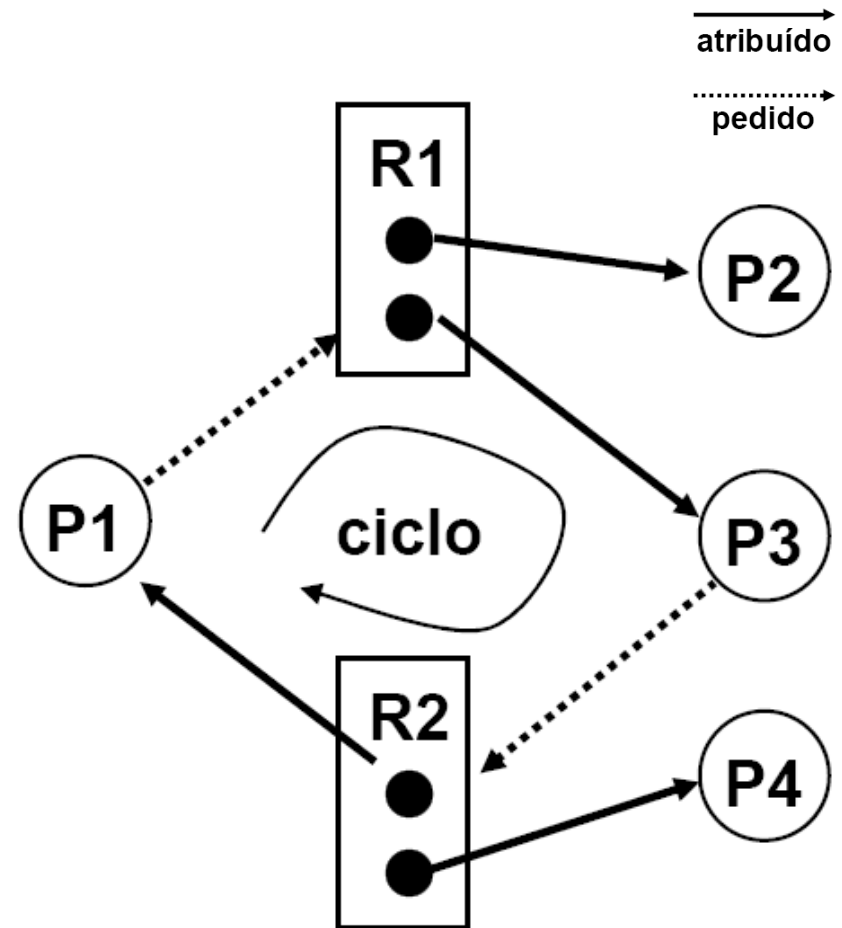
- Manter um grafo de alocação de recursos a processos
- Periodicamente procurar por ciclos (esperas circulares)
- Só funciona realmente se não existirem múltiplas instancias de cada tipo de recurso
- Caso existam ciclos, então existe a possibilidade de bloqueio



# Evitar *deadlocks*

## - Grafo de alocação de recursos

- Se existirem múltiplas instâncias de cada tipo de recurso
- Existência de um ciclo (espera circular) não implica forçosamente que se está perante uma situação de possibilidade de bloqueio



# Evitar *deadlocks*

## - Algoritmo do banqueiro

- Para  $n = n.^o$  de processos, e  $m = m.^o$  de tipos de recursos
  - Available[1..m]: indica a quantidade de recursos disponíveis de cada tipo, num determinado instante
  - Max[1..n, 1..m]: indica as necessidades máximas de cada processo, para cada tipo de recurso
  - Allocation[1..n, 1..m]: indica o número de recursos de cada tipo, atribuídos a cada processo
  - Need[1..n, 1..m]: indica as necessidades que faltam satisfazer para cada processo
    - $Need[i,j] = Max[i,j] - Allocation[i,j]$

# Evitar *deadlocks*

## - Algoritmo do banqueiro

### 1. Iniciar:

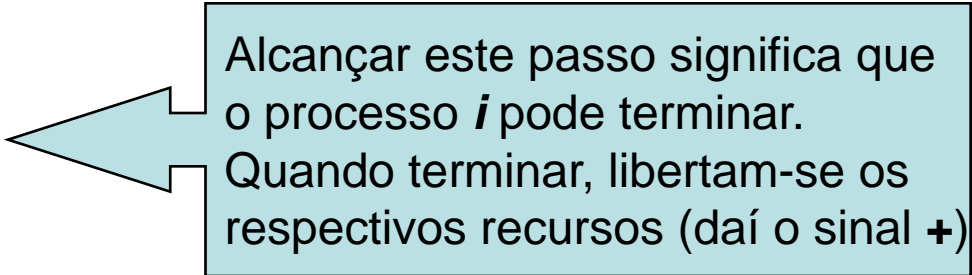
- $Work = Available$
- $Finish[i] = false$  for  $i = 0, 1, \dots, n-1$ .

### 2. Encontrar um $i$ tal que:

- (a)  $Finish[i] = false$
- (b)  $Need_i \leq Work$
- Caso não exista nenhum  $i$ , então passa-se para 4

### 3. Fazer:

- $Work = Work + Allocation_i$
- $Finish[i] = true$
- Voltar para 2



Alcançar este passo significa que o processo  $i$  pode terminar. Quando terminar, libertam-se os respectivos recursos (daí o sinal +)

### 4. Se $Finish[i] == true$ para todos, então o sistema encontra-se num estado seguro

# Evitar *deadlocks*

## - Algoritmo do banqueiro - Exemplo

- Assuma a existência

- 5 processos

- $P_0, P_1, \dots, P_4$

- 3 tipos de recursos

- A(10 instâncias)

- B(5 instâncias)

- C(7 instâncias)

	Allocation				Max				Need				Available		
	A	B	C		A	B	C		A	B	C		A	B	C
P0	0	1	0	---	7	5	3	---	7	4	3		3	3	2
P1	2	0	0	---	3	2	2	---	1	2	2				
P2	3	0	2	---	9	0	2	---	6	0	0				
P3	2	1	1	---	2	2	2	---	0	1	1				
P4	0	0	2	---	4	3	3	---	4	3	1				

Sistema está num estado seguro, já que a sequência de execução  $P_1, P_3, P_4, P_2, P_0$  satisfaz o algoritmo do banqueiro.

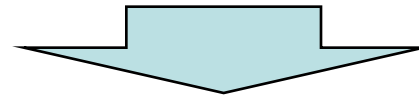


# Evitar *deadlocks*

## - Algoritmo do banqueiro - Exemplo

- Assumindo que P1 efectua um pedido de recursos
  - $\text{Request}_1 = (1, 0, 2)$
- Implica que se verifique que:
  - $\text{Request}_1 \leq \text{Available}$
  - $(1, 0, 2) \leq (3, 3, 2)$
  - Verdadeiro
- Após a execução do algoritmo do banqueiro, verifica-se que o sistema ainda se encontra num estado seguro

	Allocation				Max				Need				Available		
	A	B	C		A	B	C		A	B	C		A	B	C
P0	0	1	0	---	7	5	3	---	7	4	3		3	3	2
P1	2	0	0	---	3	2	2	---	1	2	2				
P2	3	0	2	---	9	0	2	---	6	0	0				
P3	2	1	1	---	2	2	2	---	0	1	1				
P4	0	0	2	---	4	3	3	---	4	3	1				



	Allocation				Max				Need				Available		
	A	B	C		A	B	C		A	B	C		A	B	C
P0	0	1	0	---	7	5	3	---	7	4	3		2	3	0
P1	3	0	2	---	3	2	2	---	0	2	0				
P2	3	0	2	---	9	0	2	---	6	0	0				
P3	2	1	1	---	2	2	2	---	0	1	1				
P4	0	0	2	---	4	3	3	---	4	3	1				

# Evitar *deadlocks*

- Vantagens
  - Menos restritivo que a prevenção de *deadlocks*
  - Não implica a requisição simultânea de todos os recursos necessários
  - Não obriga à preempção de recursos
- Desvantagens
  - Necessita de processamento adicional para validar estados seguros
  - Implica o conhecimento antecipado de todos os recursos necessário



**Muito limitativo**

# Detectar e Recuperar de *deadlocks*

- Recursos são atribuídos desde que estejam livres
- Periodicamente verifica-se a existência de bloqueios
- Caso existam, procede-se a recuperação
- Levantam-se alguns novos problemas
  - Quando efectuar a detecção de bloqueios?
  - Como efectuar a recuperação?

# Detectar e Recuperar de *deadlocks*

## - Quando efectuar a detecção de bloqueios?

- Sempre que se atribua um recurso...
  - Consumo de CPU elevado e , muitas vezes, desnecessário
- Periodicamente, com um intervalo de tempo fixo...
- Quando existir pouca utilização do CPU...

# Detectar e Recuperar de *deadlocks*

## - Como efectuar a recuperação?

### 1. Notificação do utilizador

- Resolução do problema compete ao utilizador

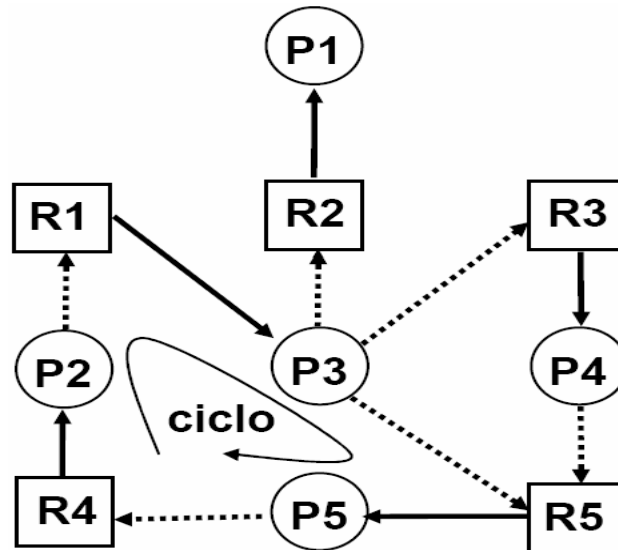
### 2. Sistema recupera por si

- “Matando” alguns processos para quebrar uma espera circular
- Implementando preempção de recursos

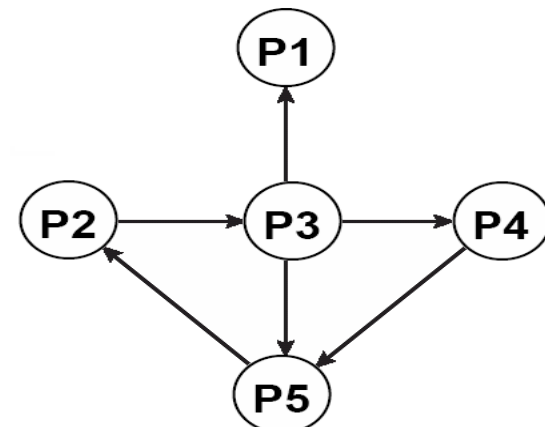
# Detectar e Recuperar de *deadlocks*

## - Detecção (uma instância por tipo de recurso)

- Manter um grafo de dependências, entre processos, derivadas das necessidades de recursos (também conhecido como *Wait-for Graph*)
- Periodicamente procurar por ciclos, caso existam significam *deadlocks*



Grafo de alocação de recursos



Grafo Wait-for

# Detectar e Recuperar de *deadlocks*

## - Detecção (várias instâncias por tipo de recurso)

- Algoritmo

1. Iniciar:

- $Work = Available$
- For  $i = 0, 1, \dots, n$ , se  $Allocation_i \neq 0$  então  $Finish[i] = false$ ; senão  $Finish[i] = true$

2. Encontrar um  $i$  tal que:

- (a)  $Finish[i] == false$
- (b)  $Request_i \leq Work$
- Caso não exista nenhum  $i$ , então passa-se para 4

3. Fazer:

- $Work = Work + Allocation_i$
- $Finish[i] = true$
- Voltar para 2

4. Se  $Finish[i] == false$  para algum  $i$ , então o sistema encontra-se bloqueado, nomeadamente o processo  $P_i$  está bloqueado

# Detectar e Recuperar de *deadlocks*

## - Recuperação

- Alternativas de recuperação possíveis
  - Terminação de processos
  - Preempção de recursos
- Terminação de processos
  - Termino abrupto de todos processos encravados
  - Termino sequencial de processos encravados até que termine o bloqueio
    - Que ordem utilizar na terminação de processos? Que factores considerar na definição dessa ordem?
      - Prioridade, tempo de CPU consumido, recursos usados, tipo de processo, etc.
  - Executar o algoritmo de detecção após cada término forçado de processo



# Detectar e Recuperar de *deadlocks*

## - Recuperação

- Preempção de recursos
  - Retirar sequencialmente recursos aos processos até que termine o bloqueio
  - Por quais recursos/processos começar?
    - Considerar factores de ponderação como o número de recursos detidos, etc.
  - Que acontece aos processos aos quais se retiram os recursos a meio da sua utilização?
    - Suporte para *rollback*? (muito difícil)
  - Como impedir a míngua?
    - Risco de ser sempre o mesmo a “sofrer”

# Estratégia combinada

- Assumindo que nenhum dos métodos é sempre bom
- Dependendo do cenário, os resultados variam de estratégia para estratégia
- Combinar as várias estratégias
- Aplicar a que produzir o melhor resultado para cada tipo de recursos

# Ignorar a existência de *deadlocks*

- Estratégia utilizada pelo sistema UNIX
- Considera-se que o custo de consumo de CPU para prevenir / evitar / detectar bloqueios não é compensador
- Negam-se pedidos se não existirem recursos disponíveis
- Funciona razoavelmente bem, se os *deadlocks* ocorrerem essencialmente nos processos de utilizador e não nos processos de sistema