SISTEMAS OPERACIONAIS

Deadlock

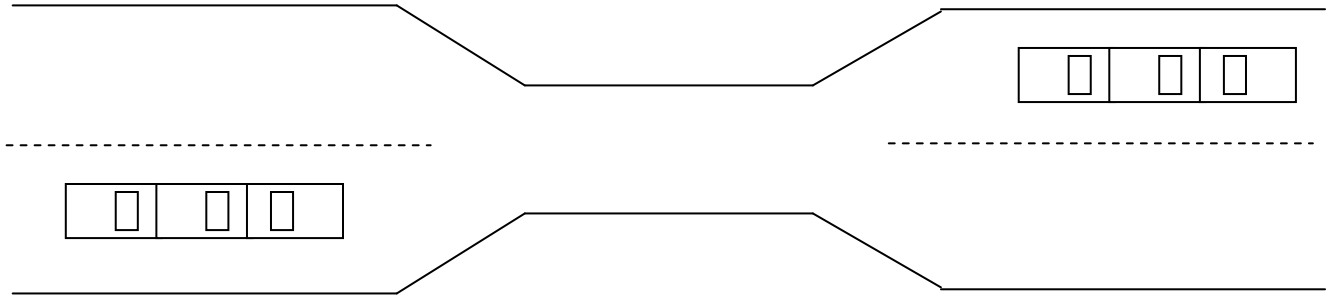
Andreza Leite
andreza.leite@univasf.edu.br

Plano da Aula

- Introdução
- Modelo de um Sistema Computacional
- Característica de um Deadlock
- Modelagem de um Deadlock (Grafos de Alocação)
- Tratamento de Deadlock(Identificação, Prevenção e Recuperação)

Definição de Deadlock

Exemplo



Deadlock - Situação onde dois ou mais processos estão esperando por um evento que só pode ser gerado por algum dos mesmos processos em espera.

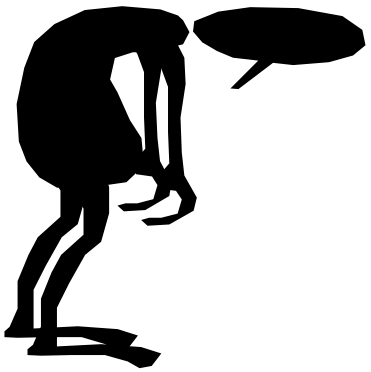
Definição de Deadlock

- *DeadLock* (Interbloqueio): caracteriza uma situação em que ocorre um impasse e dois ou mais processos ficam impedidos de continuar suas execuções, ou seja, ficam bloqueados.

Definição de Deadlock

Processo A

Detém recurso **Y**
E espera pelo recurso **X**



Processo B

Detém recurso **X**
E espera pelo recurso **Y**

Caracterização do Deadlock

CONDIÇÕES SUFICIENTES E NECESSÁRIAS

TODAS necessitam acontecer simultaneamente para ocorrer o deadlock:

- **Exclusão Mútua (Mutual Exclusion)**
- **Manter e Esperar (Hold and Wait)**
- **Não preempção (No Preemption)**
- **Espera Circular (Circular Wait)**

Caracterização do Deadlock

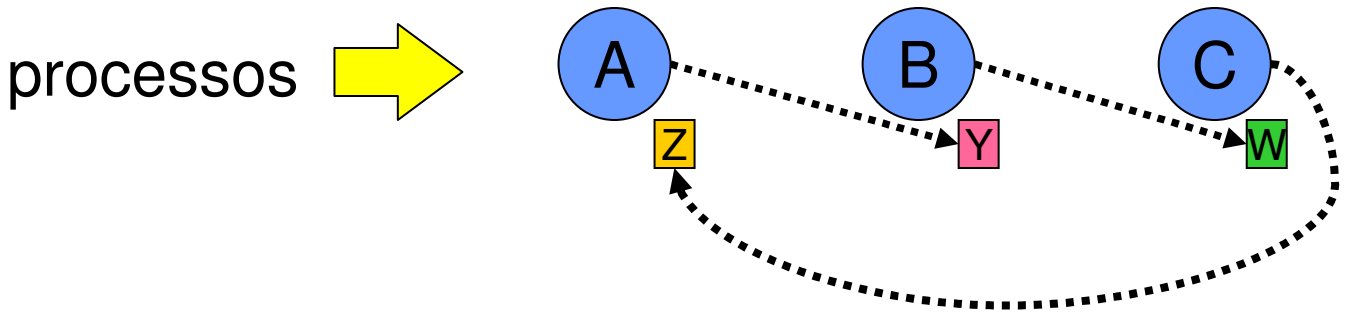
- Exclusão Mútua (*Mutual Exclusion*)
  - Existência de recursos que precisam ser acessados de forma exclusiva
  - Cada recurso só pode estar alocado a um único processo em um determinado instante
- Manter e Esperar (*Hold and Wait*)
  - Possibilidade de processos manterem os recursos alocados enquanto esperam por recursos adicionais
  - Um processo, além dos recursos já alocados, pode ficar na espera por outros;

Caracterização do Deadlock

- Não preempção (*No Preemption*)
  - Necessidade de recursos serem liberados pelos próprios processos que os estão utilizando
  - Um recurso **não** pode ser retirado de um processo porque outros processos o estão desejando
- Espera Circular (*Circular Wait*)
  - Possibilidade da formação de uma espera circular
  - Um processo pode aguardar por um recurso que esteja alocado a outro processo e vice versa.

Espera Circular

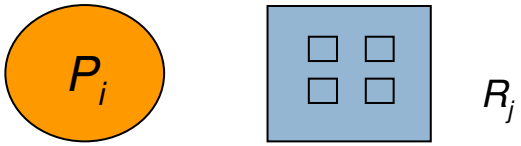
- Exemplo de uma **Espera circular** por recursos.
  - O processo "A" espera pelo processo "B", que espera pelo processo "C", que espera pelo processo "A".



Modelagem de Deadlocks utilizando Grafos

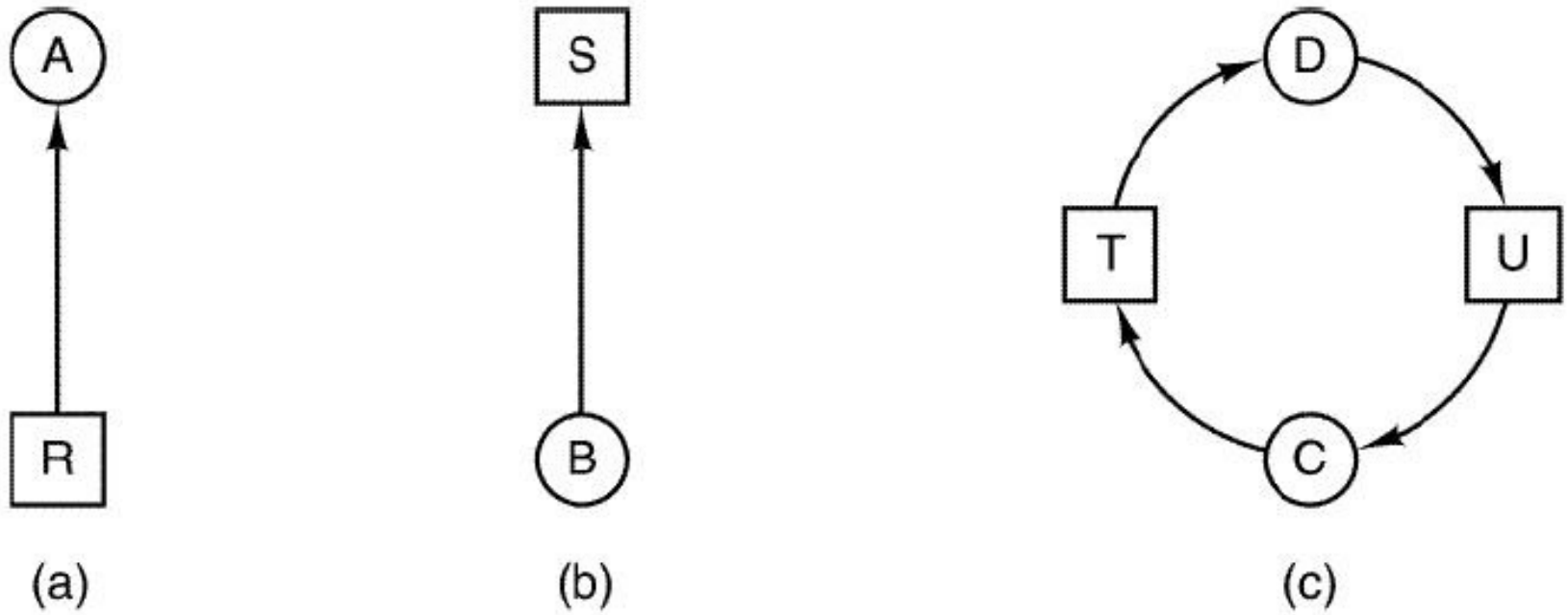
Introdução aos Deadlocks

- **Modelagem de deadlock.**
  - A modelagem pode ser feita a partir de **grafos dirigidos.**
  - Um **processo** é representado por um **círculo** e um **recurso** por um **quadrado**.



  - Um arco entre um processo e um recurso indica a requisição deste recurso pelo processo, estando o processo no estado bloqueado.
  - Um arco entre um recurso e um processo indica que o recurso está alocado ao processo.

Introdução aos Deadlocks



(a) Processo A de posse de um recurso.

(b) Processo B bloqueado esperando um recurso.

(c) *Deadlock*.

Definição Matemática Deadlock

A visão matemática de um Deadlock pode ser expressa por um Grafo de Alocação de Deadlock:

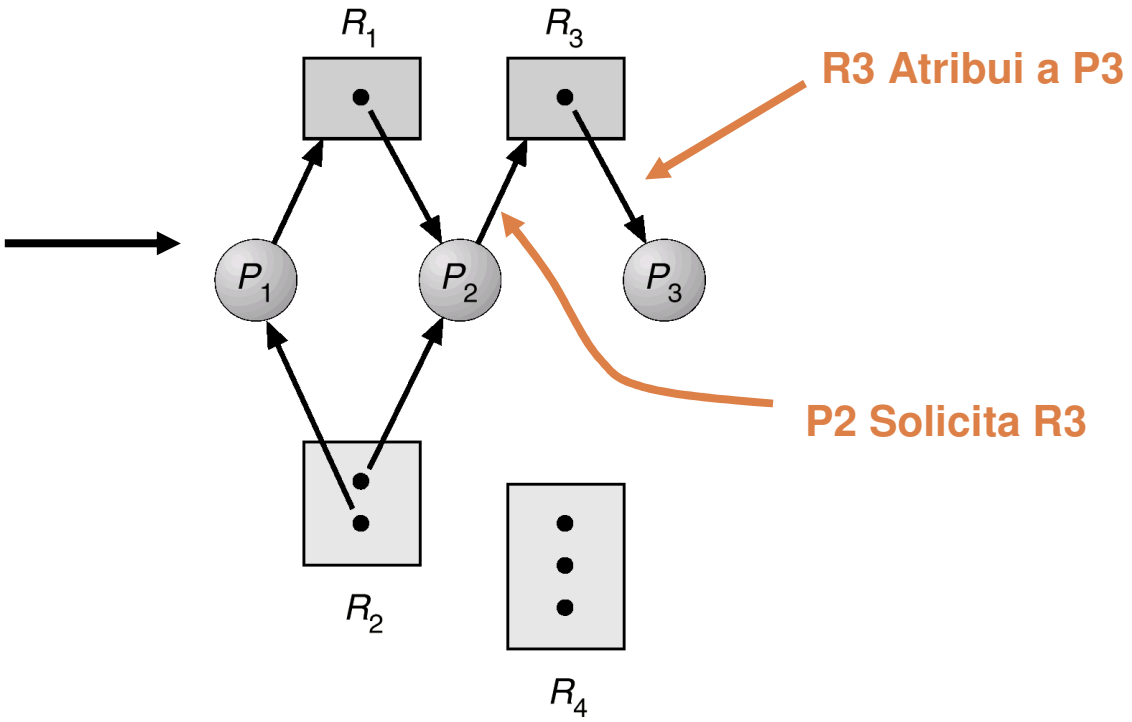
G = (V, A) O Grafo contém **nós** e **arestas**.

V → Os nós são os processos = { P_1, P_2, P_3, ...} e recursos alocados ={ R_1, R_2, ...}

A → As arestas são as relações (P_i, R_j) or (R_i, P_j)

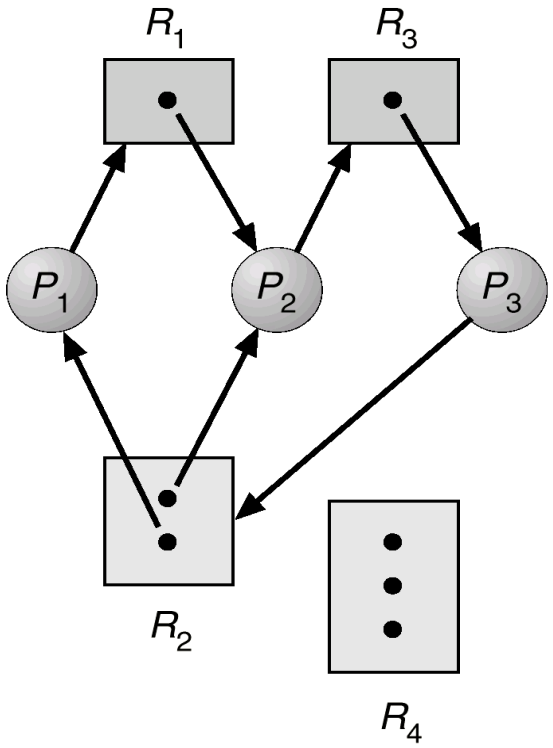
Definição Matemática Deadlock

Alocação de recursos
(Grafos)

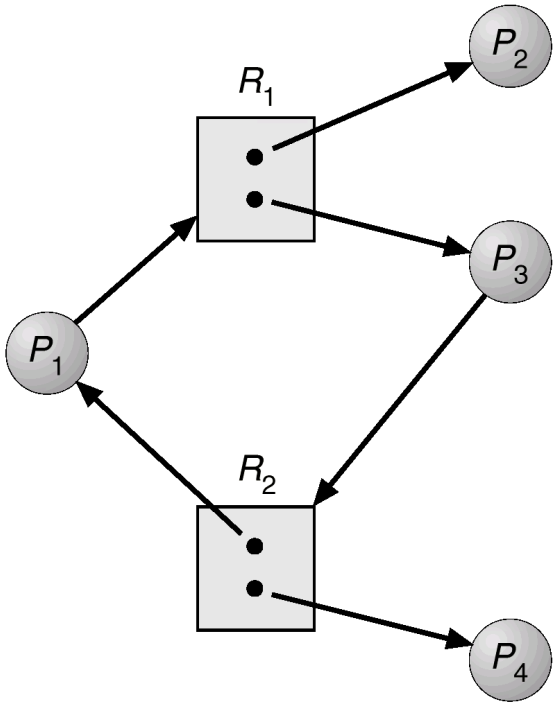


Deadlock

Grafo com Deadlock



Alocação Circular sem Deadlock



Tratamento de Deadlocks

Tratamento de Deadlock

- Identificar
- Prevenir
- Recuperar

Identificação de Deadlocks

- **Algoritmo de Busca por Espera Circular.**
  - Permite identificar:
    - Situação de deadlock.
    - Processos e Recursos envolvidos.

Prevenção de Deadlocks

- Estabelecer o critério de que todos os recursos sejam previamente alocados, antes do processo ganhar acesso à CPU;
- Admitir a prática da preempção, isto é, o sistema ter a possibilidade de retirar um recurso alocado para um processo e dar para outro processo;
- Forçar que um processo não aloque mais do que um recurso de cada vez.

Recuperação de Deadlocks

- **Recuperação através da Eliminação de processos.**
  - Mais simples.
  - Pode ser por prioridade.

- **Recuperação através de *Rollback*.**
  - Volta ao Passado (*Checkpoint*).
  - Dependente da Aplicação.

Prevenção de Deadlocks

- **Algoritmo do Banqueiro** (Dijkstra – 1965)
  - Exige que todos os processos informem o número máximo de cada tipo de recurso necessário a sua execução.
  - Assim é possível definir:
    - O estado de alocação de um recurso.
    - Número de recursos alocados e disponíveis.
    - O número máximo de processos que necessitem desses recursos.
  - Problemas:
    - Necessidade de definir o número fixo de processos ativos e recursos disponíveis.

Prevenção de Deadlocks

- Algoritmo do Banqueiro (Dijkstra – 1965)
  - Evita condições inseguras.
  - Exemplo (Linhas Gerais):

    Total de Créditos: 22 Total em caixa: 10

Situação Inicial:

	Possui	Máximo
A	0	6
B	0	5
C	0	4
D	0	7

Situação Segura:

	Possui	Máximo
A	1	6
B	1	5
C	2	4
D	4	7

Situação Insegura:

	Possui	Máximo
A	1	6
B	2	5
C	2	4
D	4	7

Analise os algoritmos


```
typedef int semaphore;
        semaphore resource_1;
        semaphore resource_2;

        void process_A(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }

        void process_B(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }
```

(a)

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
}

void process_B(void) {
        down(&resource_2);
        down(&resource_1);
        use_both_resources( );
        up(&resource_1);
        up(&resource_2);
}
```

(b)

Figura 6.2 (a) Código sem impasse. (b) Código com possibilidade de impasse.

- Por que pode ocorrer o impasse no código (b)?

Analise os algoritmos


```
typedef int semaphore;
        semaphore resource_1;
        semaphore resource_2;

        void process_A(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }

        void process_B(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }
```

Os processos solicitam os recursos na mesma ordem

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
}

void process_B(void) {
        down(&resource_2);
        down(&resource_1);
        use_both_resources( );
        up(&resource_1);
        up(&resource_2);
}
```

Os processos solicitam os recursos em uma ordem diferente

Analise os algoritmos

- Em (a):
  - Um dos processos vai adquirir o primeiro recurso antes do outro e este processo também será bem sucedido na aquisição do segundo recurso e poderá executar seu trabalho.
  - Se o outro processo tentar acessar o recurso 1 antes de este ser liberado, esse processo ficará bloqueado até o recurso em questão estar disponível.

Analise os algoritmos

- Em (b) é possível que:
  - Um dos processos adquira os dois recursos e bloqueie o outro até seu trabalho estar pronto.
  - Mas, ainda é possível que o processo A adquira o recurso 1 e o B adquira o recurso 2.
  - Cada um ficará bloqueado quando tentar adquirir o outro recurso.
  - Nenhum dos processos poderá continuar a execução.

Analise os algoritmos


```
typedef int semaphore;
        semaphore resource_1;
        semaphore resource_2;

        void process_A(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }

        void process_B(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }
```

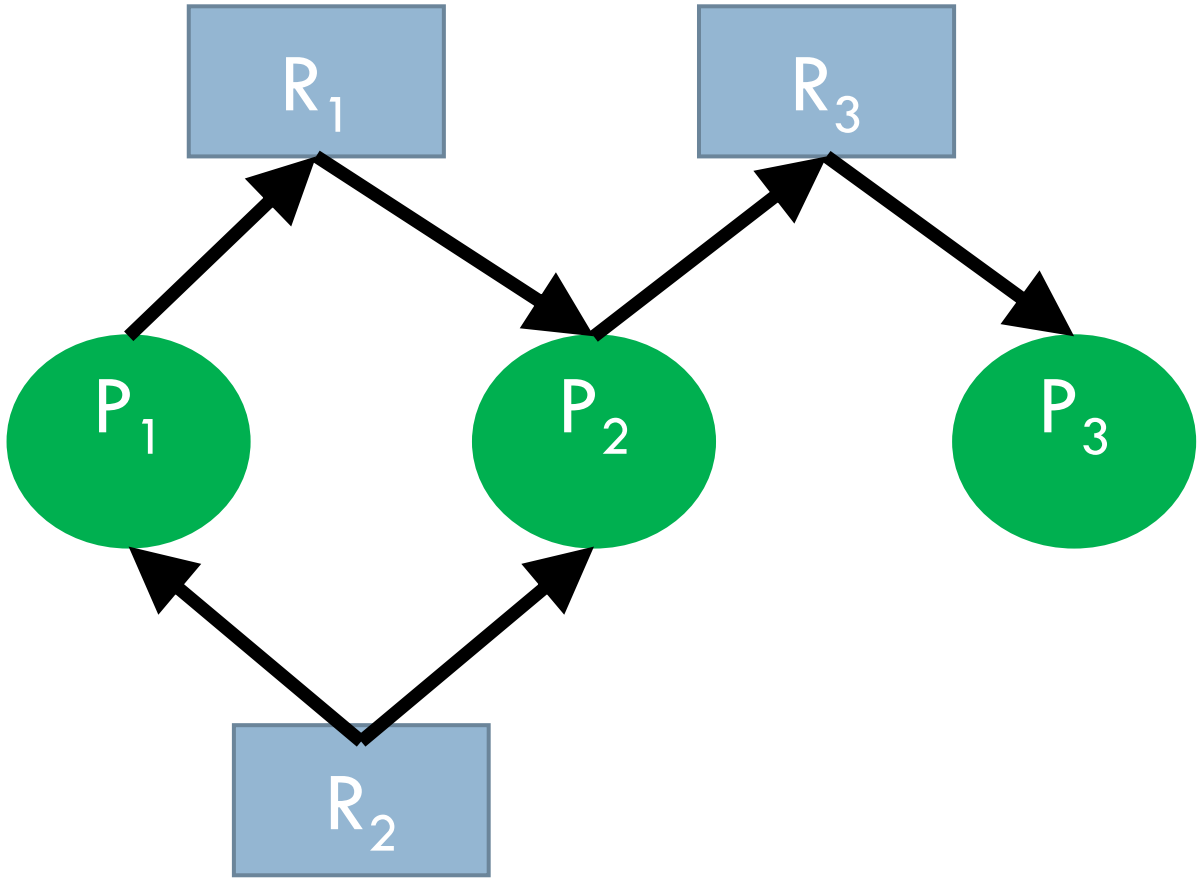
(a)

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
}

void process_B(void) {
        down(&resource_2);
        down(&resource_1);
        use_both_resources( );
        up(&resource_1);
        up(&resource_2);
}
```

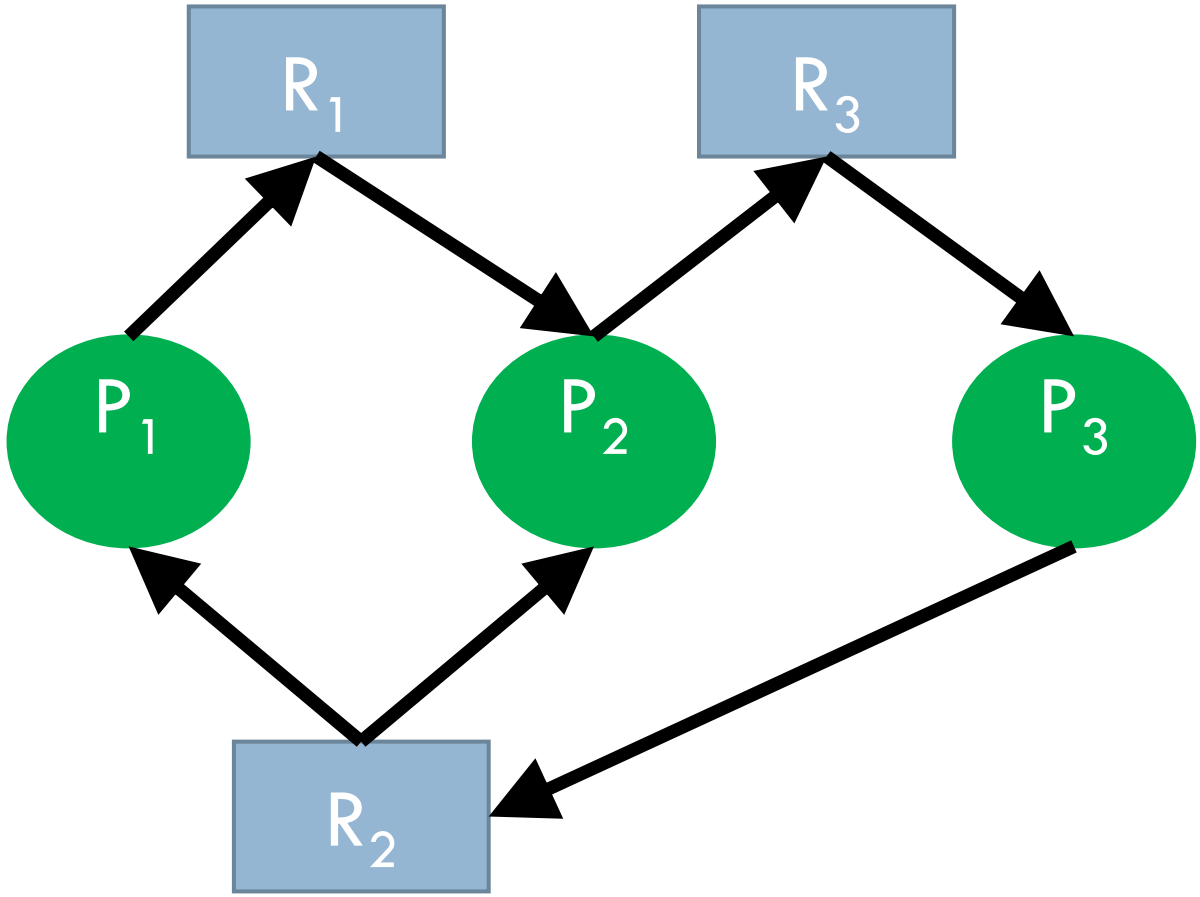
(b)

Figura 6.2 (a) Código sem impasse. (b) Código com possibilidade de impasse.

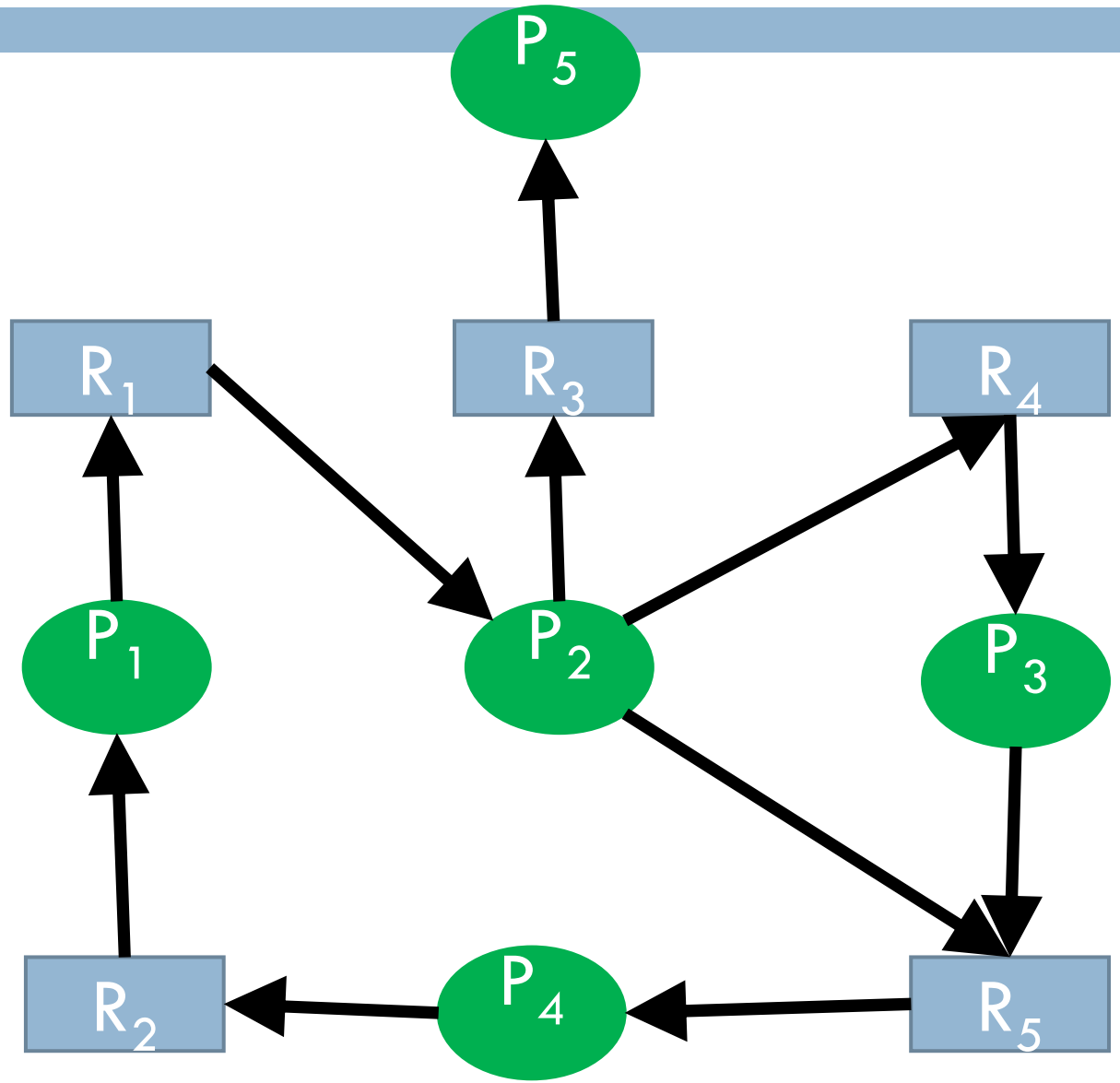
Exemplos



Exemplos



Exemplos



Exemplos