



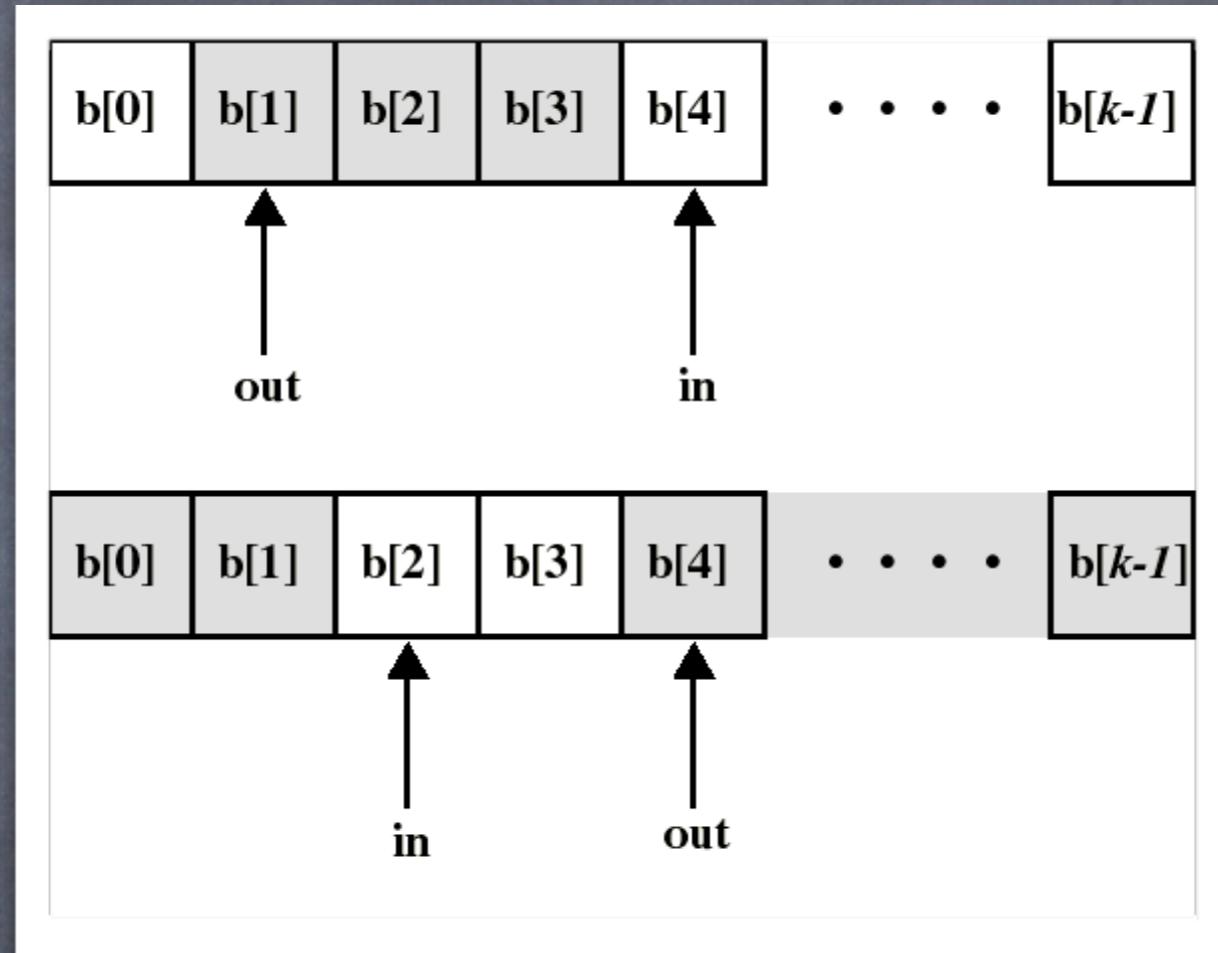
MAC422 Sistemas Operacionais

Prof. Marcel P. Jackowski

Aula #11

Barreiras, monitores e deadlocks

Buffer circular de tamanho k



- Podemos consumir somente quando o número de itens totaliza pelo menos 1
- Pode produzir desde que tenha pelo menos 1 espaço livre

Produtor-consumidor: buffer finito

- Novamente:
 - Usamos um semáforo para exclusão mútua para acesso ao buffer e apontadores
 - Um semáforo contador para sincronizar o produtor e consumidor em relação ao número de itens consumíveis (“espaços cheios”)
- Adicionamos:
 - Um outro semáforo para sincronizar produtor e consumidor em relação ao número de espaços “vazios”

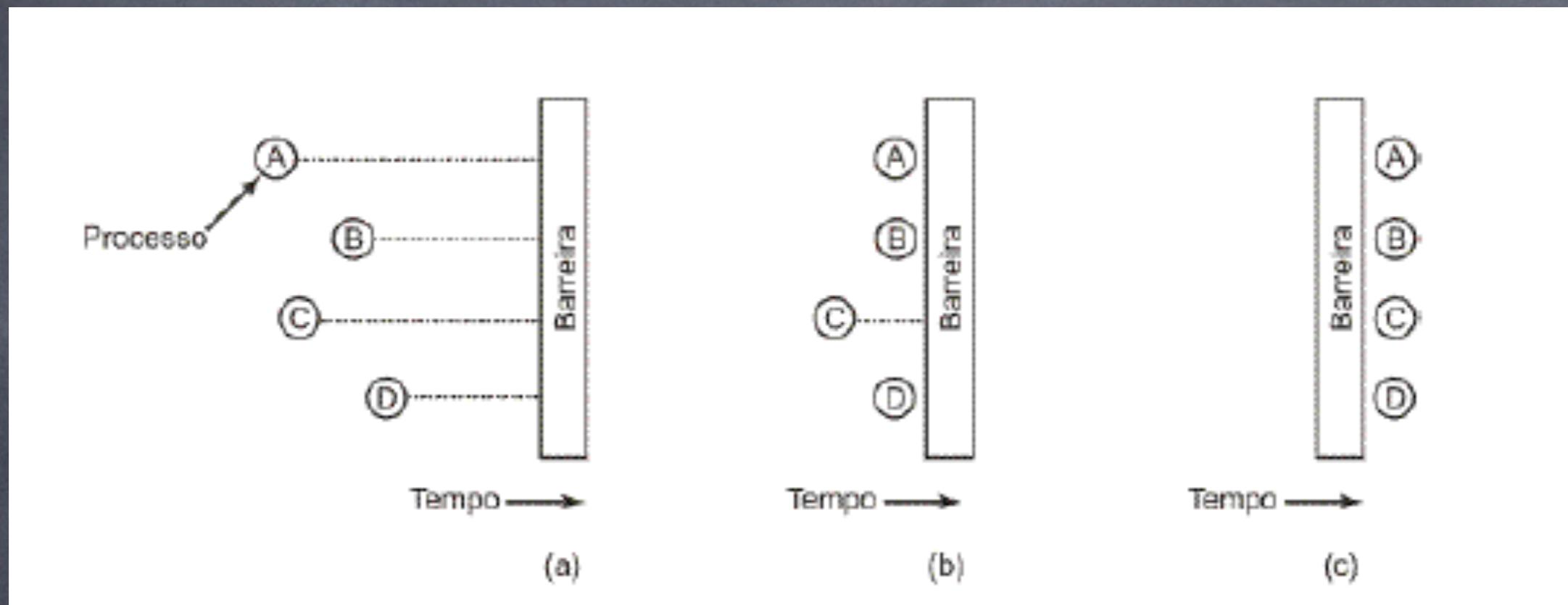
Solução com semáforos

```
/* variáveis compartilhadas */
int in = 0, out = 0;
int b[MAX]; /* MAX == k */
semaphore mutex = 1;
semaphore empty = MAX, full = 0;

/* PRODUTOR */                                /* CONSUMIDOR */
void main()                                     void main()
{
    for(;;) {
        wait(empty);
        wait(mutex);
        b[in] = produz_item();
        in = (in + 1) % MAX;
        signal(mutex);
        signal(full);
    }
}

for(;;) {
    wait(full);
    wait(mutex);
    consome_item(b[out]);
    out = (out + 1) % MAX;
    signal(mutex);
    signal(empty);
}
```

Barreiras



- a) processos se aproximando de uma barreira
- b) todos os processos, exceto um, bloqueados pela barreira
- c) último processo chega, todos passam

Forma geral

```
process worker[ i = 1 to N ] {  
    while (true) {  
        /* código para implementar tarefa i */  
        /* espera por todas N tarefas completarem */  
    }  
}
```

Exemplo de barreira

Tentativa com 3 processos

```
semaphore p1=0, p2=0, p3=0;

/* Processo 1 */          /* Processo 2 */          /* Processo 3 */
void main()               void main()               void main()
{
    for(;;) {
        /* tarefa #1 */
        signal(p1);
        wait(p2);
        wait(p3);
    }
}

for(;;) {
    /* tarefa #2 */
    signal(p2);
    wait(p1);
    wait(p3);
}

for(;;) {
    /* tarefa #3 */
    signal(p3);
    wait(p1);
    wait(p2);
}
```

Funciona ?

Exemplo com coordenador

```
semaphore terminei=0, p[i]={0,0,...,0};

/* Processo 1 */          /* Processo 2 */          /* Processo coordenador */
void main()               void main()               {
{
    for(;;) {           for(;;) {           for(;;) {
        /* tarefa #1 */     /* tarefa #2 */     for(int i=0;i<2;i++)
        signal(terminei);   signal(terminei);   wait(terminei);
        wait(p[0]);         wait(p[1]);       /* colhe resultados */
    }                   }                   for(int i=0;i<2;i++)
}                           }                           signal(p[i]);
}
```

Semáforos

- Mais eficientes e estruturados que os spin locks
 - Processos são bloqueados em filas FIFO enquanto aguardam por um semáforo
- Normalmente utilizados em processos do usuário
- Uso correto é de completa responsabilidade do usuário
 - Falha em interpretar a semântica das operações `signal()` ou `wait()`, ou operações não pareadas podem levar à deadlock.

Semáforos em Linux

- Linux: duas implementações POSIX e System V
 - POSIX: utilizado em threads
 - `sem_init()`, `sem_wait()`, `sem_post()`, etc
 - System V: processos
 - `semget()`, `semctl()`, `semop()`, etc.
 - Um pouco mais complicados, pois requerem a existência de memória compartilhada.

Semáforos usando System V IPC

- Assim como segmentos de memória compartilhada e filas de mensagens (“message queues”), podemos criar conjuntos de semáforos
 - \$ ipcs
- Instrumentos para gerir a coordenação entre processos
- Normalmente associados com operações em recursos compartilhados.

Alocação

- `int semget(key_t key, int nsems, int semflg)`
 - key: identificador do semáforo.
 - IPC_PRIVATE cria um identificador único
 - nsems: número de semáforos no conjunto
 - semflg:
 - IPC_CREAT: cria um novo conjunto quando um valor de key é passado
 - IPC_EXCL: exclusive, usado em conjunto com IPC_CREAT, caso o segmento já exista, retorna erro
 - Mode Flags: flags de acesso 9 bits

Inicialização

- `int semctl(int semid, int semnum, int cmd, union semun arg)`
 - semid: identificador retornado pelo semget
 - semnum: número do semáforo
 - cmd:
 - GETALL
 - GETNCNT
 - GETPID
 - GETVAL
 - GETZCNT
 - SETALL
 - SETVAL

Operações

- **int semop (int semid, struct sembuf *sops, unsigned nsops)**
- semid: identificador retornado pela operação semget

```
struct sembuf {  
    ushort sem_num; /* semaphore index */  
    short sem_op;   /* semaphore operation */  
    short sem_flg; /* operation flags */  
};
```

- sem_op: -1 (wait), 1 (signal)
- nsops: número de operações

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/sem.h>
#include <sys/types.h>
#include <sys/ipc.h>

#define SHM_SIZE 4096

union semun {
    int val; struct semid_ds *buf;
    unsigned short *array;
    struct seminfo *__buf;
};

int main()
{
    int shmid, semid;
    char* saddr;
    union semun arg;
    struct sembuf wait={0, -1, 0};
    struct sembuf signal={0, 1, 0};

    /* aloca memória compartilhada e semáforo */
    shmid= shmget(IPC_PRIVATE, SHM_SIZE, IPC_CREAT | S_IRUSR | S_IWUSR);
    semid= semget(IPC_PRIVATE, 1, IPC_CREAT | S_IRUSR | S_IWUSR);
    saddr= (char *)shmat(shmid, 0, 0);

    /* inicialização */
    arg.val=1;
    semctl(semid, 0, SETVAL, arg);
```

```
if (fork() == 0) { //children
    saddr[0] = 65;
    printf("Filho esperando semaforo\n");
    if (semop(semid, &wait, 1) == -1) {
        perror("semop");
        return 1;
    }
    printf("Filho na secao critica %s\n", saddr);
    sleep(20);
    if (semop(semid, &signal, 1) == -1) {
        perror("semop");
        return 1;
    }
    printf("Filho liberou semaforo\n");
    return 0;
}
saddr[1] = 66; sleep(2);
printf("Pai esperando semaforo\n");
if (semop(semid, &wait, 1) == -1) {
    perror("semop");
    return 1;
}
printf("Pai na secao critica %s\n", saddr);
if (semop(semid, &signal, 1) == -1) {
    perror("semop");
    return 1;
}
printf("Pai liberou semaforo\n");
/* Falta liberar semáforos! */
shmctl(shmid, IPC_RMID, 0);
return 0;
```

Regiões Críticas

- Mecanismo de sincronização de alto nível
- Uma variável compartilhada v é somente acessível dentro de:
 - $\text{region } v \text{ when } B \text{ do } S$
- Onde B é uma expressão booleana
- Quando S é executado nenhum outro processo pode acessar a variável v .

Regiões Críticas

- Regiões que se referem à mesma variável compartilhada cuja execução é feita de forma mutualmente exclusiva
- Quando um processo tenta executar a região, a expressão booleana B é avaliada. Se B é verdade, S é executado
- Se B é falso, o processo é bloqueado até B se tornar verdade e nenhum outro processo está dentro da região associada com v .

Exemplo

```
struct buffer {
    int pool[MAX];
    int count, in, out;
}

/* Produtor */

region buffer when (count < MAX)
{
    pool[in] = nextp;
    in = (in+1) % MAX;
    count++;                                /* Consumidor */
}

region buffer when (count > 0)
{
    nextc = pool[out];
    out = (out+1) % MAX;
    count--;
}
```

Monitores

- Unidade básica de sincronização de alto nível
 - Módulos de programas
 - Mais estruturados que semáforos
- Podem ser vistos como abordagem O-O
 - Mecanismo de abstração de dados
 - Os processos podem chamar os procedimentos mas não podem acessar as estruturas internas diretamente
- Ou seja, agrupam
 - Representação/Implementação de um recurso compartilhado

Formato básico

```
monitor nome {  
    declarações de variáveis  
    funções e procedimentos  
    inicializações  
}
```

Monitores

- Um monitor é compartilhado por processos executantes concorrentes
- Exclusão mútua é implícita
 - Isso é desejável pois evita interferência automaticamente
- Já a sincronização condicional é programada explicitamente porque
 - Varia com diferentes programas e
 - Pode ser implementada eficientemente via variáveis condicionais

Exemplo de monitor

```
monitor Nome {  
  
    int i;  
    cond c;  
  
    function f1() {  
        ...  
    }  
  
    ...  
  
    procedure p1() {  
        ...  
    }  
  
    inicialização;  
}
```

Variáveis condicionais

- Usadas em sincronização condicionais
 - Retardar processo até que estado do monitor satisfaça uma condição booleana
- Só podem ser declaradas dentro do monitor
 - **cond cv**
- Representam uma fila de processos atrasados
 - Inicialmente vazio

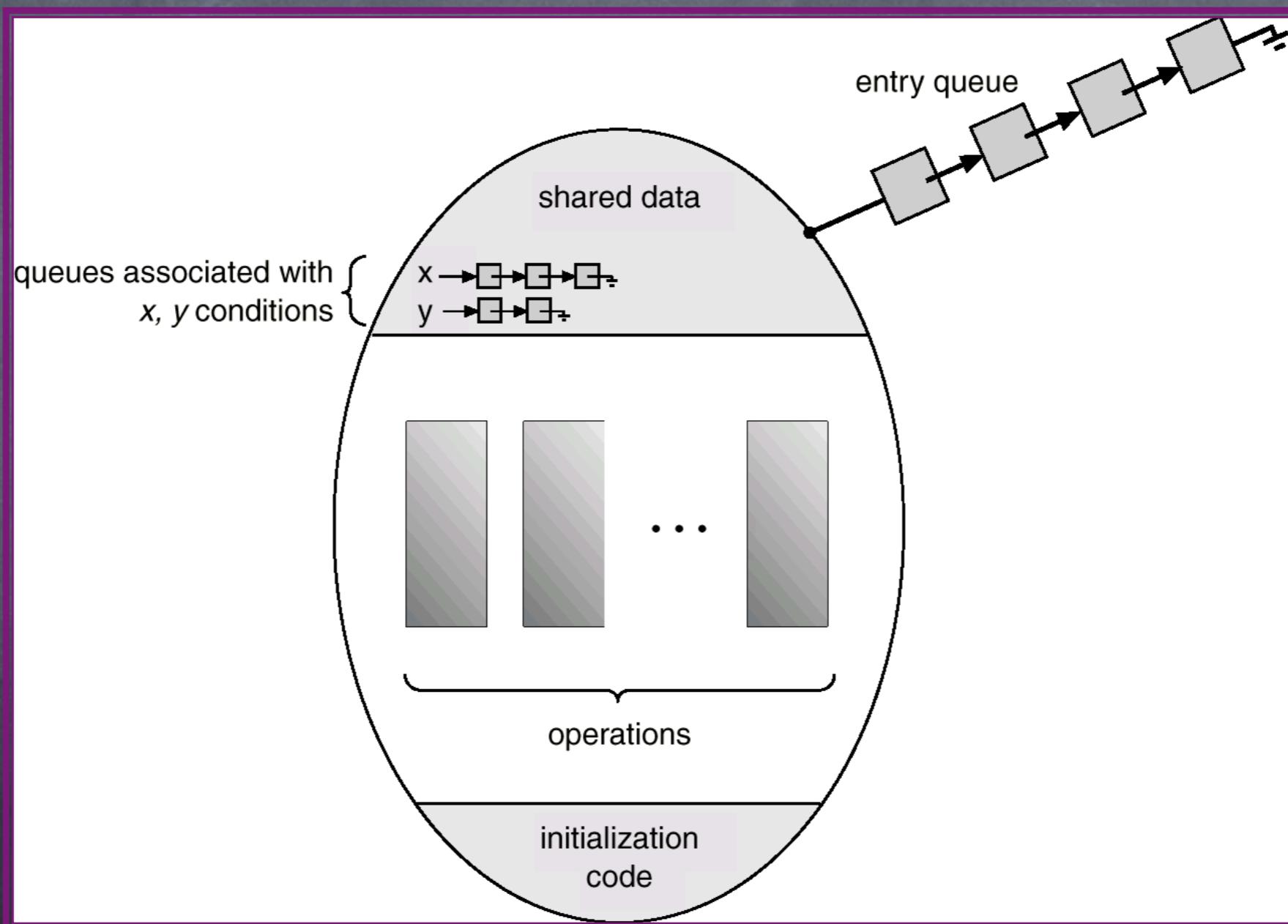
Operações

- E o valor não é lido diretamente, mas através de operações
- Verifica estado da fila através de
 - `empty(cv)`
- Bloqueia na variável condicional cv com
 - `wait(cv)`
 - Processo executante entra no final da fila de atraso de cv
 - E também libera acesso exclusivo ao monitor

Operações

- Acorda processo bloqueado em cv
 - `signal(cv)`
 - Examina fila de atraso de cv
 - Se fila estiver vazia então sem efeito
 - Caso contrário, acorda o primeiro processo da fila
 - Assim, as operações wait e signal criam disciplina FIFO

Monitores



```
monitor ProdCons {                                /* produtor */
    cond full, empty;
    int b[MAX], total=0;

    void insere(int item)
    {
        if (total==MAX)
            wait(full);
        insere(b, item); total++;
        if (total==1)                      /* consumidor */
            signal(empty);
    }

    int retira(void)
    {
        int item;
        if (total==0)
            wait(empty);
        item = remove(b); total--;
        if (total==MAX-1)
            signal(full);
        return item;
    }
}

void main()
{
    int item;
    for(;;) {
        item = produz_item();
        ProdCons.insere(item);
    }
}

void main()
{
    int item;
    for(;;) {
        item = ProdCons.retira();
        consome_item(item);
    }
}
```

Disciplinas

- Quando um processo executa signal(), sabe-se que ele possui o lock (implícito) associado ao monitor
- Surge então o seguinte dilema:
 - Como a execução de signal() acordará um outro processo, além do atual, então
 - Existem 2 processos ativos (quem executou signal é acordado agora)
 - Mas só 1 deve executar

Disciplinas

- Signal e Continue (SC)
 - Não-preemptivo: thread sinalizador continua e o thread sinalizado executa depois
 - Signal é tipo de aviso que o thread acordado deve continuar; ele volta à fila e espera pelo lock do monitor
- Signal e Wait (SW)
 - Preemptivo: sinalizador passa controle ao sinalizado e volta à fila
 - Sinalizado passa a executar imediatamente

Suporte à monitores: Java

- Algumas linguagens suportam monitores. Ex: java
- Java suporta threads de usuário e também permite que métodos sejam agrupados em classes
- Palavra-chave *synchronized* à declaração de um método
 - Uma vez iniciado qualquer thread executando aquele método, a nenhum outro thread será permitido executar qualquer outro método *synchronized* naquela classe

```
public class ProducerConsumer {  
    static final int N = 100;                      // constante com o tamanho do buffer  
    static producer p = new producer( ); // instância de um novo thread produtor  
    static consumer c = new consumer( );// instância de um novo thread consumidor  
    static our_monitor mon = new our_monitor( ); // instância de um novo monitor  
  
    public static void main(String args[ ]) {  
        p.start( );                                // inicia o thread produtor  
        c.start( );                                // inicia o thread consumidor  
    }  
  
    static class producer extends Thread {  
        public void run() {                         // o método run contém o código do thread  
            int item;  
            while (true) {                          // laço do produtor  
                item = produce_item( );  
                mon.insert(item);  
            }  
        }  
        private int produce_item( ) { ... } // realmente produz  
    }  
  
    static class consumer extends Thread {  
        public void run() {                         // método run contém o código do thread  
            int item;  
            while (true) {                          // laço do consumidor  
                item = mon.remove( );  
                consume_item (item);  
            }  
        }  
        private void consume_item(int item) { ... } // realmente consome  
    }  
}
```

```
static class our_monitor {          // este é o monitor
    private int buffer[ ] = new int[N];
    private int count = 0, lo = 0, hi = 0; // contadores e índices
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep(); // se o buffer estiver cheio, vá dormir
        buffer [hi] = val;           // insere um item no buffer
        hi = (hi + 1) % N;          // lugar para colocar o próximo item
        count = count + 1;          // mais um item no buffer agora
        if (count == 1) notify();    // se o consumidor estava dormindo, acorde-o
    }
    public synchronized int remove( ) {
        int val;
        if (count == 0) go_to_sleep(); // se o buffer estiver vazio, vá dormir
        val = buffer [lo];           // busca um item no buffer
        lo = (lo + 1) % N;          // lugar de onde buscar o próximo item
        count = count - 1;          // um item a menos no buffer
        if (count == N - 1) notify(); // se o produtor estava dormindo, acorde-o
        return val;
    }
    private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
}
```

Recursos

- Exemplos de recursos:
 - Impressoras, dispositivos de armazenamento, etc
 - Tabelas do SO
 - Memória, etc
- Processos precisam utilizar recursos em uma dada ordem
- Supondo que um processo possui um recurso A e pede o recurso B
 - Ao mesmo tempo outro processo possui B e pede A
 - Ambos estão bloqueados e assim permanecerão

Recursos e Deadlocks

- Recursos “Preemptíveis”
 - Podem ser retirados ao processo sem efeitos prejudiciais (e.g. memória)
- Recursos “Não-preemptíveis”
 - Causarão a falha do processo se lhe for retirado
 - Deadlocks ocorrem quando...
 - Processos têm acesso exclusivo a certos recursos
 - Deadlocks surgem de recursos “Não-preemptíveis”

Deadlock

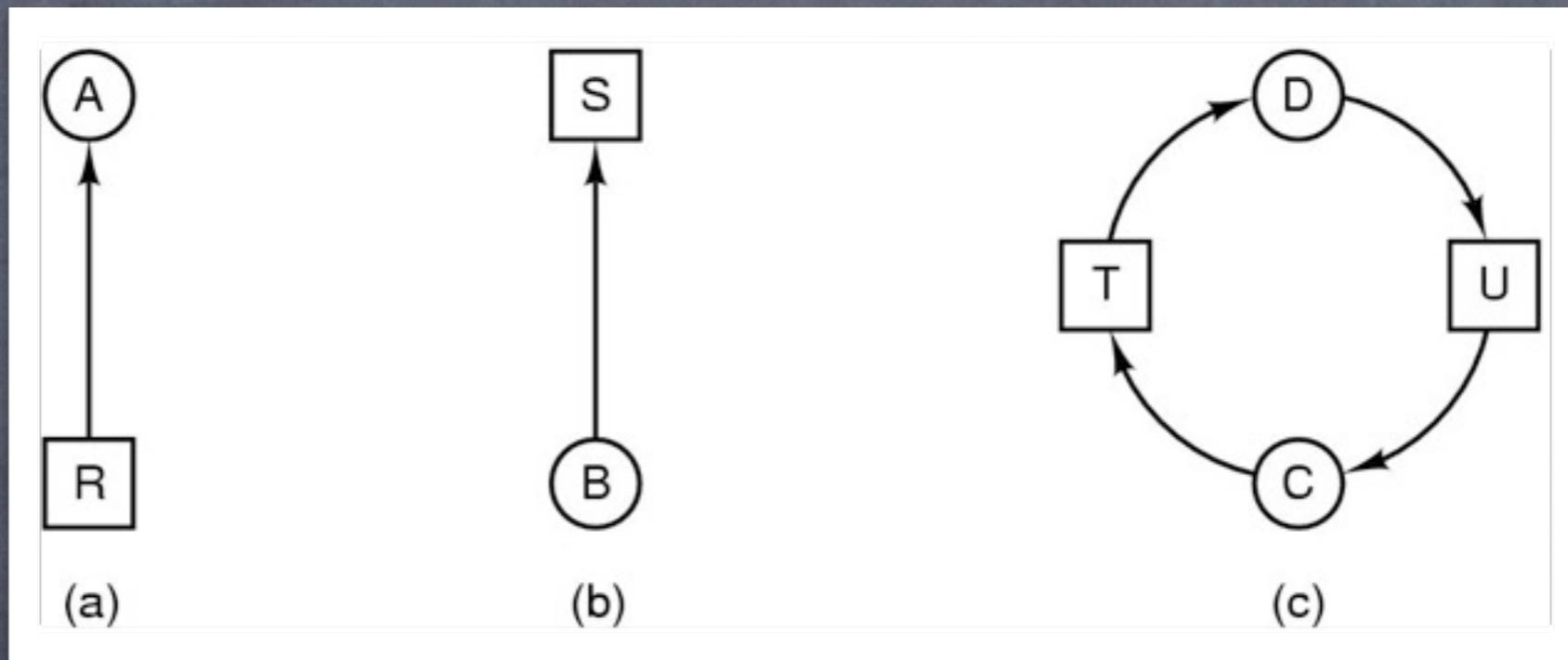
- Definição formal:
 - Um conjunto de processos está estado de “deadlock” se cada processo pertencente ao conjunto estiver esperando por um evento que somente um outro processo no conjunto pode provocar.
- Normalmente o “evento” corresponde à liberação de um recurso.
- Nenhum dos processos pode:
 - Fazer progresso
 - Liberar recursos
 - Ser acordado

Condições para deadlock

- Condição de Exclusão Mútua
 - Cada recurso está atribuído a 1 processo ou está disponível
- Condição de Posse e Espera (“Hold and wait”)
 - Processos possuindo recursos podem pedir mais recursos
- Condição de Não-preempção
 - Recursos já atribuídos não podem ser retirados
- Condição de Espera Circular
 - Tem de ser uma cadeia circular de 2 ou mais processos
 - Cada processo está à espera de recursos atribuídos pelo próximo membro da cadeia.

Modelagem gráfica

- Grafos direcionais



- Recurso R atribuído ao processo A
- Processo B está à espera/pedi o recurso S
- Processo C e D estão num deadlock sobre os recursos T e U

Modelando ocorrência

Request R
Request S
Release R
Release S

(a)

Request S
Request T
Release S
Release T

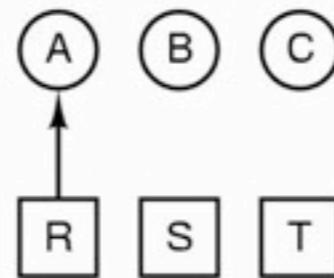
(b)

Request T
Request R
Release T
Release R

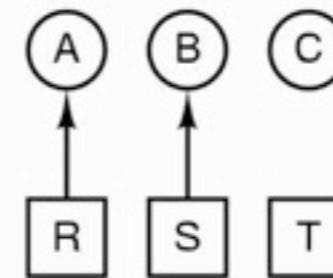
(c)

1. A requests R
 2. B requests S
 3. C requests T
 4. A requests S
 5. B requests T
 6. C requests R
- deadlock

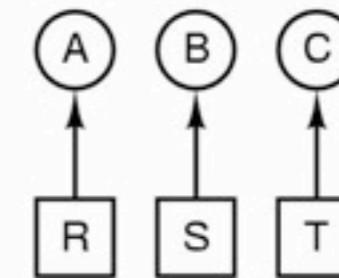
(d)



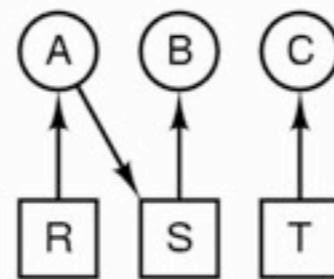
(e)



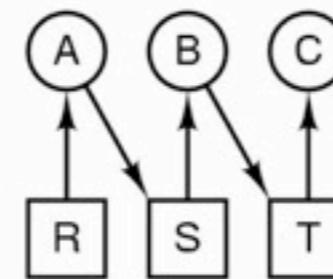
(f)



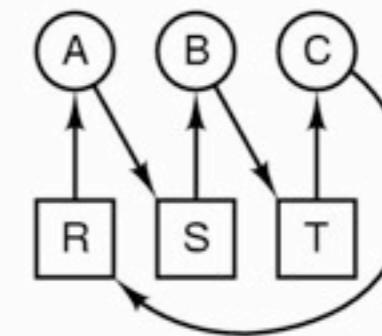
(g)



(h)



(i)

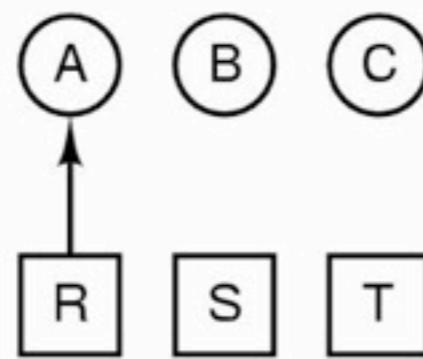


(j)

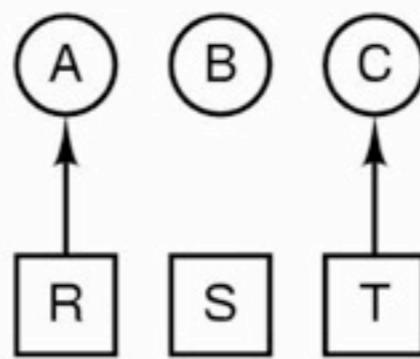
Modelando ocorrência

1. A requests R
 2. C requests T
 3. A requests S
 4. C requests R
 5. A releases R
 6. A releases S
- no deadlock

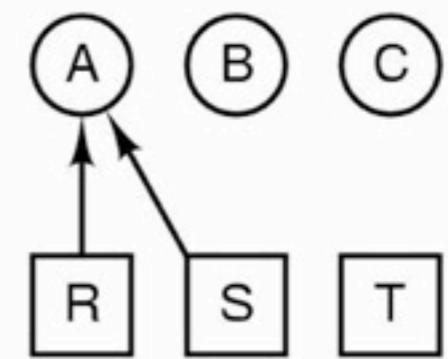
(k)



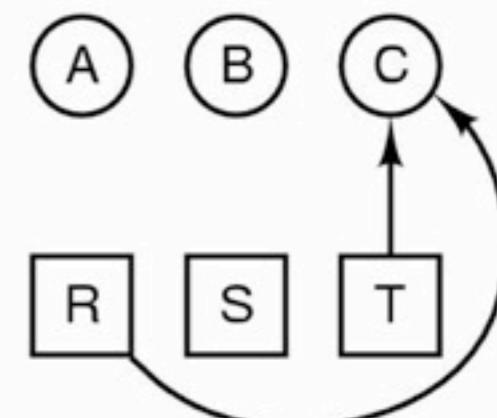
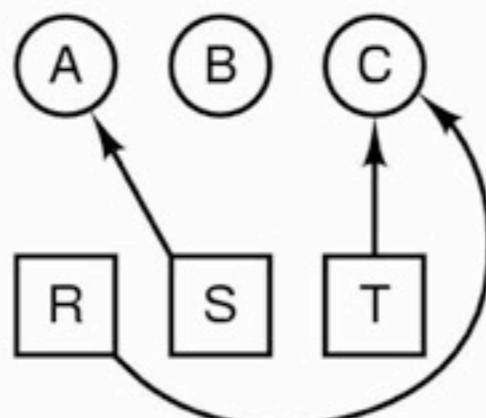
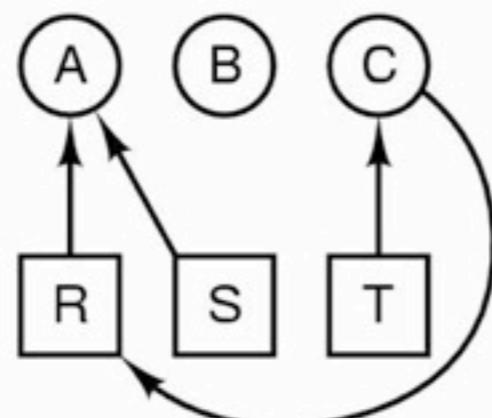
(l)



(m)



(n)



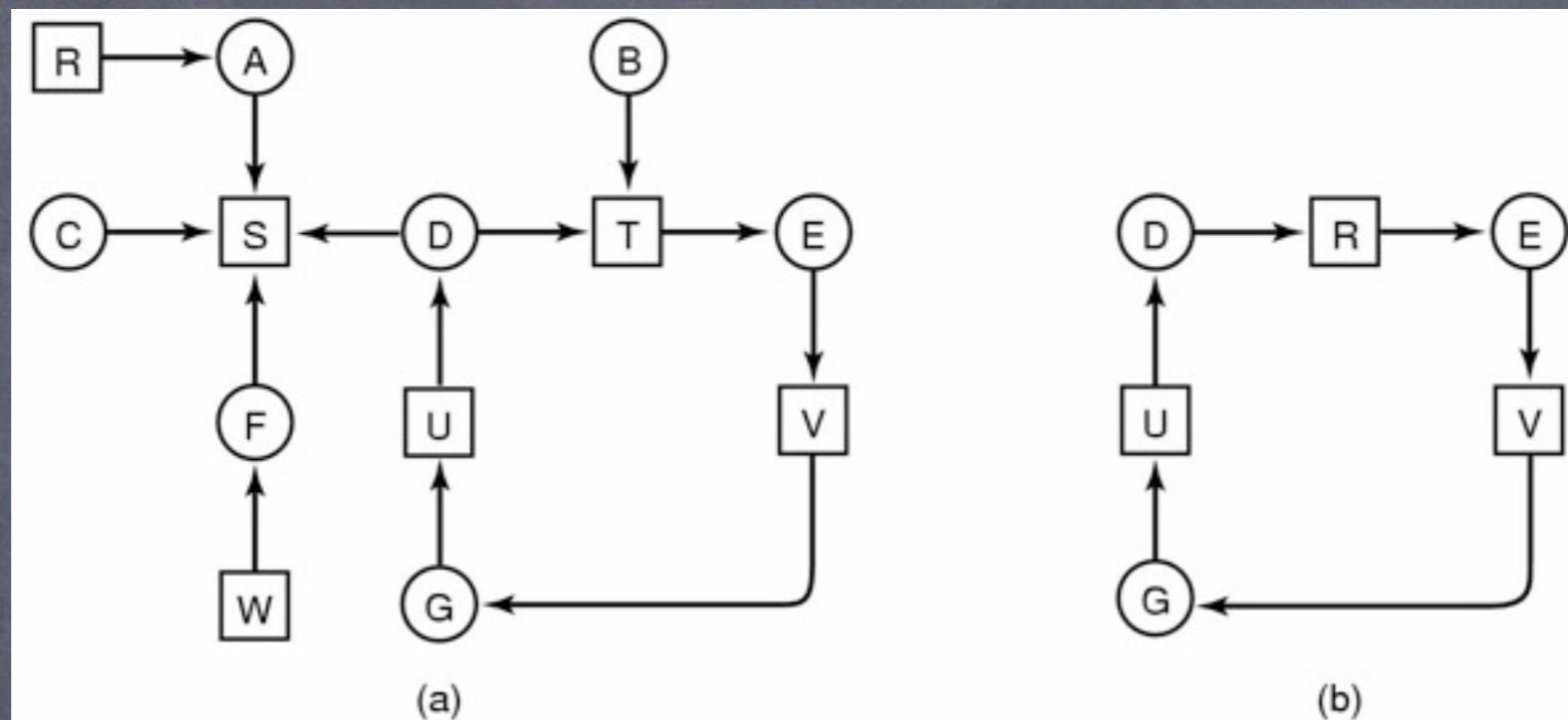
Como lidar com deadlocks

- Algoritmo do “Avestruz”
- Detecção e recuperação
 - Detectar a ocorrência de um deadlock e agir de acordo.
- Evitar deadlocks
 - Alocação supervisionada de recursos
- Prevenir deadlocks
 - Negar uma das quatro condições necessárias

Algoritmo do Avestruz

- Fingir que não há problema
 - É aceitável se
 - Deadlocks ocorrerem com pouca frequência
 - Custo da prevenção for elevado
 - Linux e Windows seguem esta aproximação
 - É uma escolha entre:
 - Conveniência
 - Correção

Detecção e recuperação



- Notar a posse de recursos e os pedidos
- Um ciclo pode ser encontrado no grafo, mostrando um deadlock
- É necessário um algoritmo para detecção de deadlocks

Algoritmos de recuperação

- Através de Preempção
 - Retirar o recurso de um outro processo sem este notar e devolver mais tarde
- Através de “Rollback”
 - É necessário gravar o estado do processo periodicamente
 - Recomeçar o processo se se encontrar um deadlock
- “Matando” processos
 - Grosseiro, mas uma forma simples de quebrar deadlocks
 - Matando um dos processo no ciclo, outros processos poderão eventualmente obter os recursos
 - Escolher processos que podem voltar a executar desde o início

Evitar deadlocks

- Pode-se evitar deadlocks?
 - Sim, desde que se saiba um dado conjunto de informações logo de início
 - Número de instâncias de recursos

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

	Has	Max
A	3	9
B	0	-
C	2	7

Free: 5

	Has	Max
A	3	9
B	0	-
C	7	7

Free: 0

	Has	Max
A	3	9
B	0	-
C	0	-

Free: 7

Estados seguros e inseguros

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

Prevenção

- Atacar a condição de *exclusão mútua*
 - Alguns dispositivos (ex: impressora) podem utilizar uma lista de serviço de tarefas
 - Só o *daemon* da impressora usa o recurso
 - Deadlock na impressora eliminado
 - Princípio:
 - Evitar atribuir recursos quando não é absolutamente necessário
 - Reduzir ao mínimo o número de processos que pedem o recurso

Prevenção

- Atacar a condição de *Posse-e-Espera* (“*hold and wait*”)
 - Precisa que os processos peçam os recursos antes de começarem
- O processo nunca tem de esperar pelo recurso que precisa
- Problemas:
 - Ele pode não saber os recursos necessários quando começa a executar
 - Ele detêm recursos que outros podiam usar

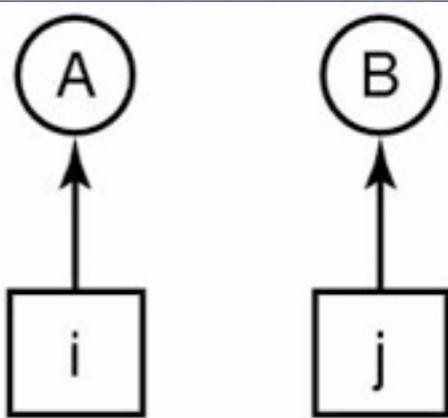
Prevenção

- Atacar a condição de *não-preempção*
 - Não é uma opção viável
 - Considere o exemplo de um processo que tem atribuída uma impressora
 - ...e no meio do seu trabalho o recurso é lhe retirado.

Prevenção

- Atacar a condição de *espera circular*

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive



- Ordenação de recursos
- Pedido de recursos feitos por ordem numérica
- Grafo de recursos nunca terá ciclos
- Todos os processos terminam !

Resumo

- Exclusão mútua
 - Lista de espera de acessos
 - Usar “*spooling*” em tudo
- Posse-e-espera
 - Requisitar inicialmente todos os recursos
- Não preempção
 - Retomar os recursos alocados
- Espera circular
 - Ordenar numericamente os recursos.