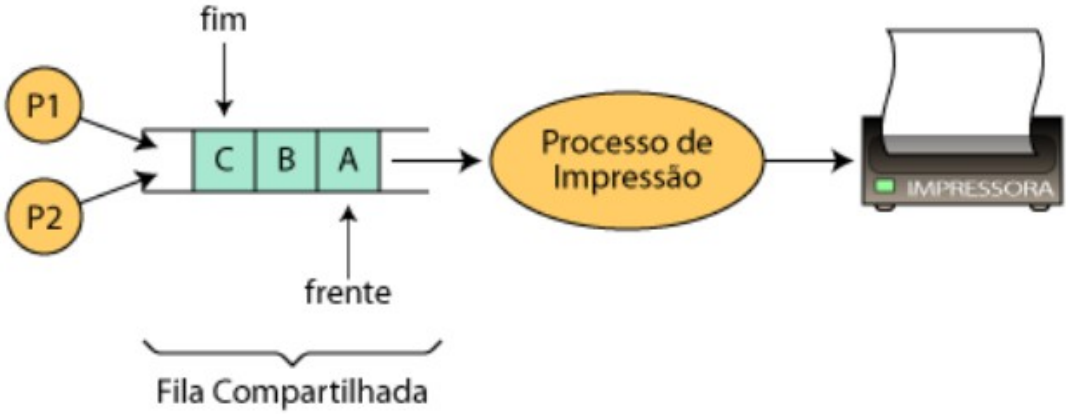
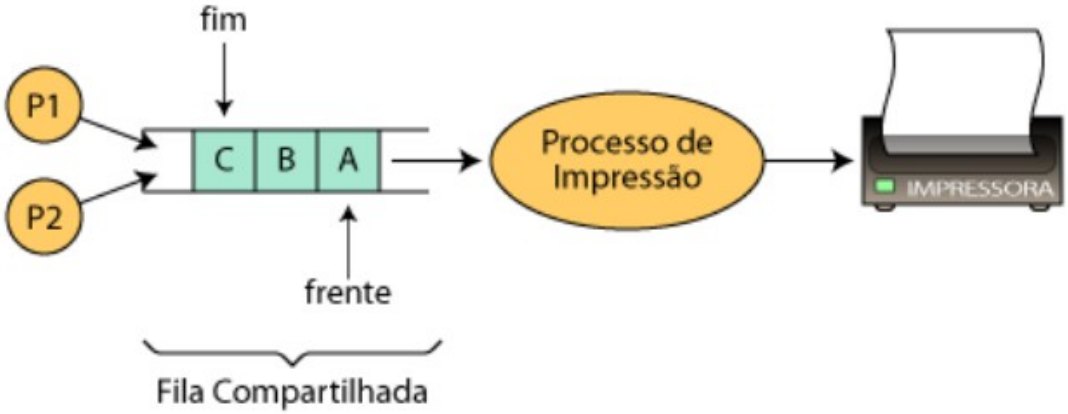
Sincronização de Threads / Processos

Condições de Corrida



- Exemplo: Fila de impressão.
  - Qualquer processo / thread que queira imprimir precisa colocar o seu documento na fila de impressão (compartilhada).
  - O processo de impressão retira os documentos na ordem em que chegaram na fila
  - Se a fila é compartilhada, isto significa que seus dados, assim como os indicadores de **frente** e **fim** da fila também o são

Condições de Corrida



1. fim++ (incrementa o indicador do fim da fila)
2. coloca documento na posição do novo fim da fila

- dois processos resolvem simultaneamente imprimir um documento
- o primeiro processo foi interrompido (por ter acabado o seu quantum) entre os comandos 1 e 2
- o segundo processo insere seu documento na fila antes que o primeiro processo tenha acabado : **qual é o erro ????**
- Há uma **condição de corrida** quando dois ou mais processos estão acessando dados compartilhados e o resultado depende da ordem de execução

Condições de Corrida

- **Condições de corrida** são situações onde dois ou mais processos ou threads acessam dados compartilhados e o resultado final depende da ordem em que são executados
  - Ordem de execução é ditada pelo mecanismo de escalonamento do S.O.
  - Torna a depuração difícil.
- Condições de corrida são evitadas através da introdução de mecanismos de **exclusão mútua**:
  - A exclusão mútua garante que somente um processo estará usando os dados compartilhados num dado momento.
- **Região Crítica**: parte do programa (trecho de código) em que os dados compartilhados são acessados
- **Objetivo da Exclusão Mútua**:
  - Proibir que mais de um processo entre em sua Região Crítica

Tipos de Soluções

- Soluções de Hardware
  - Inibição de interrupções
  - Instrução TSL (*test set and lock*)
- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Algoritmo de Decker
  - Algoritmo de Peterson
- Soluções de software com bloqueio
  - Semáforos, Monitores

Inibição de Interrupções

- Usa um par de instruções do tipo DI / EI.
  - DI = *disable interrupt* EI = *enable interrupt*
- O processo desativa todas as interrupções imediatamente antes de entrar na sua R.C., reativando-as imediatamente depois de sair dela.
- Com as interrupções desativadas, nenhum processo que está na sua R.C. pode ser interrompido, o que garante o acesso exclusivo aos dados compartilhados.

Problemas da Solução DI/EI

- É desaconselhável dar aos processos de usuário o poder de desabilitar interrupções.
- Não funciona com vários processadores.
- Inibir interrupções por um longo período de tempo pode ter conseqüências danosas. Por exemplo, perde-se a sincronização com os dispositivos periféricos.
  - OBS: inibir interrupções pelo tempo de algumas poucas instruções pode ser conveniente para o *kernel* (p.ex., para atualizar uma estrutura de controle).

Soluções com *Busy Wait*

- *Busy wait* = espera ativa ou espera ocupada.
- Basicamente o que essas soluções fazem é:
  - Quando um processo quer entrar na sua R.C. ele verifica se a entrada é permitida. Se não for, ele espera em um laço (improdutivo) até que o acesso seja liberado.
    - Ex: While (vez == OUTRO) do {nothing};
  - Conseqüência: desperdício de tempo de CPU.

Variável de Bloqueio (não funciona!)

- Variável de bloqueio, compartilhada, indica se a R.C. está ou não em uso.
  - *turn = 0* ⇒ R.C. livre *turn = 1* ⇒ R.C. em uso
- Tentativa para *n* processos:

```
var turn: 0..1
turn := 0

Process Pi:
...
while turn = 1 do {nothing};
turn := 1;
< critical section >
turn := 0;
...
```

Problemas

- A proposta não é correta pois os processos podem concluir “simultaneamente” que a R.C. está livre, isto é, os dois processos podem testar o valor de *turn* antes que essa variável seja feita igual a *true* por uma delas.

Uma versão que funciona (quase)

```
Process P0:
...
flag[0] := true;
while flag[1] do
  begin
    flag[0] := false;
    <delay for a short time>
    flag[0] := true
  end;
< critical section >
flag[0] := false;
...
```

```
Process P1:
...
flag[1] := true;
while flag[0] do
  begin
    flag[1] := false;
    <delay for a short time>
    flag[1] := true
  end;
< critical section >
flag[1] := false;
...
```

Uma versão que funciona (quase)

- Esta solução é quase correta. Entretanto, existe um pequeno problema: a possibilidade dos processos / threads ficarem cedendo a vez um para o outro "indefinidamente" (problema da "mútua cortesia")
  - Livelock
- Na verdade, essa é uma situação muito difícil de se sustentar durante um longo tempo na prática, devido às velocidades relativas dos processos. Entretanto, ela é uma possibilidade teórica, o que invalida a proposta como solução geral do problema.

Solução de Dekker

- Trata-se da primeira solução correta para o problema da exclusão mútua de dois processos (proposta na década de 60).
- O algoritmo combina as idéias de variável de bloqueio e *array* de intenção.
- É similar ao algoritmo anterior mas usa uma variável adicional (*vez/turn*) para realizar o desempate, no caso dos dois processos entrarem no *loop* de mútua cortesia.

Algoritmo de Dekker

```
var flag: array[0..1] of boolean;
    turn: 0..1; //who has the priority

flag[0] := false
flag[1] := false
turn := 0   // or 1
```

```
Process p0:
     flag[0] := true
     while flag[1] {
         if turn ≠ 0 {
            flag[0] := false
            while turn ≠ 0 {}
            flag[0] := true
          }
     }

    // critical section
    ...
    // end of critical section
    turn := 1
    flag[0] := false
```

```
Process  p1:
   flag[1] := true
   while flag[0] {
      if turn ≠ 1 {
        flag[1] := false
        while turn ≠ 1 {}
        flag[1] := true
      }
   }

   // critical section
   ...
   // end of section
   turn := 0
   flag[1] := false
```

Algoritmo de Dekker (cont.)

- Quando *P0* quer entrar na sua R.C. ele coloca seu *flag* em *true*. Ele então vai checar o *flag* de *P1*.
- Se o *flag* de *P1* for *false*, então *P0* pode entrar imediatamente na sua R.C.; do contrário, ele consulta a variável *turn*.
- Se *turn = 0* então *P0* sabe que é a sua vez de insistir e, deste modo, fica em *busy wait* testando o estado de *P1*.
- Em certo ponto, *P1* notará que é a sua vez de declinar. Isso permite ao processo *P0* prosseguir.
- Após *P0* usar a sua R.C. ele coloca o seu *flag* em *false* para liberá-la, e faz *turn = 1* para transferir o direito para *P1*.

Algoritmo de Dekker (cont.)

- Algoritmo de Dekker resolve o problema da exclusão mútua
- Uma solução deste tipo só é aceitável se houver um número de CPUs igual (ou superior) ao número de processos que se devam executar no sistema. Porquê?
  - Poderíamos nos dar 'ao luxo' de consumir ciclos de CPU,
  - Situação rara na prática (em geral, há mais processos do que CPUs)
  - Isto significa que a solução de Dekker é pouco usada.
- Contudo, a solução de Dekker mostrou que é possível resolver o problema inteiramente por software, isto é, sem exigir instruções máquina especiais.
- Devemos fazer uma modificação significativa do programa se quisermos estender a solução de 2 para N processos:
  - flag[] com N posições; variável turn passa a assumir valores de 1..N; alteração das condições de teste em todos os processos

Solução de Lamport (algoritmo da padaria)

- Proposto em 1974 por Leslie Lamport é uma solução para N processos para o algoritmo de Dekker
- Utilizado para o compartilhamento de um recurso em uma seção crítica

Solução de Lamport (algoritmo da padaria)

```
volatile bool entering[0..N] := 0
volatile int number[0..N] := 0;

lock(i) {
   entering[i] = true;
   number[i] = 1 + max(number[0], ..., number[N-1]);
   entering[i] = false;
   for (j = 0; j < N; j++) {
      // Wait until thread j receives its number:
      while (entering[j]);
      // Wait until all threads with smaller numbers or with the same
      // number, but with higher priority, finish their work
      while ((number[j] != 0) && ((number[j], j) < (number[i], i)));
   }
}

unlock(i) {
   number[i] = 0;
}
```

Solução de Peterson

- Proposto em 1981, é uma solução simples e elegante para o problema da exclusão mútua, sendo facilmente generalizado para o caso de *n* processos.
- O truque do algoritmo consiste no seguinte:
  - Ao marcar a sua intenção de entrar, o processo já indica (para o caso de empate) que a vez é do outro.
- Mais simples de ser verificado

Algoritmo de Peterson

```
flag[0]   := false
flag[1]   := false
turn      := 0

Process P0:
    flag[0] := true
    turn := 1
    while ( flag[1] && turn == 1 ){
         // do nothing
    }
    // critical section
    ...
    // end of critical section
    flag[0] := false
```

```
Process P1:
    flag[1] := true
    turn := 0
    while ( flag[0] && turn == 0 ){
          // do nothing
    }
    // critical section
    ...
    // end of critical section
    flag[1] := false
```

Algoritmo de Peterson (para N fios)

```
volatile int level[0..N] = 0
volatile int waiting[0..N-1] = 0

lock(task_id) {
    for (l = 0; l < N-1; l++) {
        level[task_id] = l
        waiting[l] = task_id
        for (k = 0; k < N; k++) {
             while (k != task_id && level[k] >= l && waiting[l] == task_id);
        }
    }
}

unlock(task_id) {
    level[task_id] = 0;
}
```

Solução de Peterson (cont.)

- Exclusão mútua é atingida.
  - Uma vez que *P0* tenha feito *flag[0] = true*, *P1* não pode entrar na sua R.C.
  - Se *P1* já estiver na sua R.C., então *flag[1] = true* e *P0* está impedido de entrar.
- Bloqueio mútuo (deadlock) é evitado.
  - Supondo *P0* bloqueado no seu *while*, isso significa que *flag[1] = true* e que *turn = 1*
  - se flag[1] = true e que turn = 1, então P1 por sua vez entrará na sua seção crítica
  - Assim, *P0* só pode entrar quando **ou** *flag[1]* tornar-se *false* **ou** *turn* passar a ser 0.

Semáforos (1)

- Mecanismo criado pelo matemático holandês E.W. Dijkstra, em 1965.
- O semáforo é uma **variável inteira** que pode ser mudada por apenas duas operações primitivas (atômicas): ***P*** e ***V***.
  - *P* = *proberen* (testar)
  - V = *verhogen* (incrementar).
- Quando um processo executa uma operação ***P***, o valor do semáforo é **decrementado**. O processo pode ser eventualmente bloqueado (semáforo for < 0 após decrementar) e inserido na **fila de espera** do semáforo.
- Numa operação *V*, o semáforo é **incrementado** e, eventualmente, um processo que aguarda na **fila de espera** deste semáforo é acordado (semáforo <= 0 após incrementar).

Semáforos (2)

- A operação *P* também é comumente referenciada como:
  - *down* ou *wait*
- *V* também é comumente referenciada
  - *up* ou *signal*

- Semáforos que assumem somente os valores *0* e *1* são denominados *semáforos binários* ou *mutex*. Neste caso, P e V são também chamadas de *LOCK* e *UNLOCK*, respectivamente.

Semáforos (3)

```
P(S):
   S := S – 1
   If S < 0 Then
      bloqueia processo (coloca-o na fila de S)


V(S):
   S := S + 1
   If S <= 0 Then
      acorda algum processo (remove-o da fila de S)
```

Uso de Semáforos (1)

- Exclusão mútua (semáforos binários):

```
...
Semaphore mutex = 1;      /*var.semáforo,
                iniciado com 1*/


Processo P1   Processo P2    ... Processo Pn
...           ...            ...
P(mutex)      P(mutex)             P(mutex)
// R.C.       // R.C.        // R.C.
V(mutex)      V(mutex)             V(mutex)
...           ...            ...
```

Uso de Semáforos (2)

- Alocação de Recursos (semáforos contadores):

```
...
Semaphore S := 3;


Processo P1     Processo P2      Processo P3
...             ...              ...
P(S)            P(S)             P(S)
//usa recurso   //usa recurso    //usa recurso
V(S)            V(S)             V(S)
...             ...              ...
```

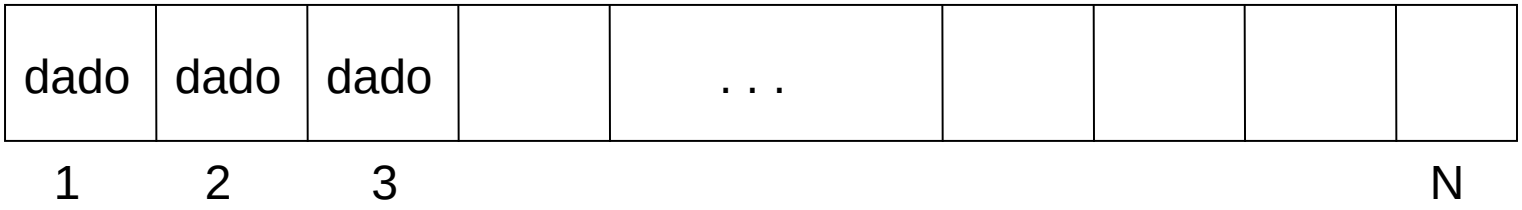
Uso de Semáforos (3)

- Relação de precedência entre processos:
  (Ex: executar *p1_rot2* somente depois de *p0_rot1*)

```
semaphore S = 0 ;
parbegin
      begin                          /* processo P0*/
            p0_rot1()
            V(S)
            p0_rot2()
      end
      begin                          /* processo P1*/
            p1_rot1()
            P(S)
            p1_rot2()
      end
parend
```

O Problema do Produtor e Consumidor c/ *Buffer* Limitado

- Thread produtora gera dados e os coloca em um *buffer* de tamanho N.
- Thread consumidora retira os dados do *buffer*, na mesma ordem em que foram colocados, um de cada vez.
- Se o *buffer* está **cheio**, o produtor deve ser bloqueado
- Se o *buffer* está **vazio**, o consumidor é quem deve ser bloqueado.
- Apenas um única thread, produtor ou consumidor, pode acessar o *buffer* num certo instante.



Exemplo: Produtor - Consumidor c/ *Buffer* Limitado

```
#define N 100                   /* number of slots in the buffer */

semaphore mutex = 1;            /* controls access to critical region */
semaphore empty = N;            /* counts empty buffer slots */
semaphore full = 0;             /* counts full buffer slots */

void producer(void){
  int item;
  produce_item(&item); /* generate something to put in buffer */
  P(&empty);           /* decrement empty count */
  P(&mutex);           /* enter critical region */
  enter_item(item);    /* put new item in buffer */
  V(&mutex);           /* leave critical region */
  V(&full);            /* increment count of full slots */
}

void consumer(void){
  int item;
  P(&full);            /* decrement full count */
  P(&mutex);           /* enter critical region */
  remove_item(&item);  /* take item from buffer */
  V(&mutex);           /* leave critical region */
  V(&empty);           /* increment count of empty slots */
  consume_item(item);  /* do something with the item */
}
```

Uso de Semáforos (4)

- Sincronização do tipo barreira:

  (*n-1* processos aguardam o *n-ésimo* processo para todos prosseguirem)

```
semaphore X = 0;
semaphore Y = 0;
semaphore Z = 0;
...
```

```
P1:
...
V(X);
V(X);
P(Y);
P(Z);
do_A;
...
```

```
P2:
...
V(Y);
V(Y);
P(X);
P(Z);
do_B;
...
```

```
P3:
...
V(Z);
V(Z);
P(X);
P(Y);
do_C;
...
```

Problemas e uso dos Semáforos (1)

- Exemplo: suponha que os dois *down* do código do produtor estivessem invertidos. Neste caso, *mutex* seria diminuído antes de *empty*. Se o *buffer* estivesse completamente cheio, o produtor bloquearia com *mutex = 0*. Portanto, da próxima vez que o consumidor tentasse acessar o *buffer* ele faria um *down* em *mutex*, agora zero, e também bloquearia. Os dois processos ficariam bloqueados eternamente.
- Conclusão: erros de programação com semáforos podem levar a resultados imprevisíveis.

Problemas e uso dos Semáforos (2)

- Embora semáforos forneçam uma abstração flexível o bastante para tratar diferentes tipos de problemas de sincronização, ele pode ser inadequado em algumas situações.
- Semáforos são uma abstração de alto nível baseada em primitivas de baixo nível, que provêem atomicidade e mecanismo de bloqueio, com manipulação de filas de espera e de escalonamento. Tudo isso contribui para que a operação seja lenta.