Programação de Sistemas

Corridas
Races

Condições de corrida

- Concorrência nos sistemas operativos possibilita o acesso de várias tarefas (processos ou fios de execução) a informação comum com alteração, pelo menos por uma das unidades.
- A ordem em que são executados os acessos e as alterações pode levar a resultados distintos.
  - Vários fios de execução partilham acesso à mesma variável global.
  - Vários processos partilham acesso ao mesmo dispositivo.
  - Vários processos partilham acesso ao mesmo ficheiro.

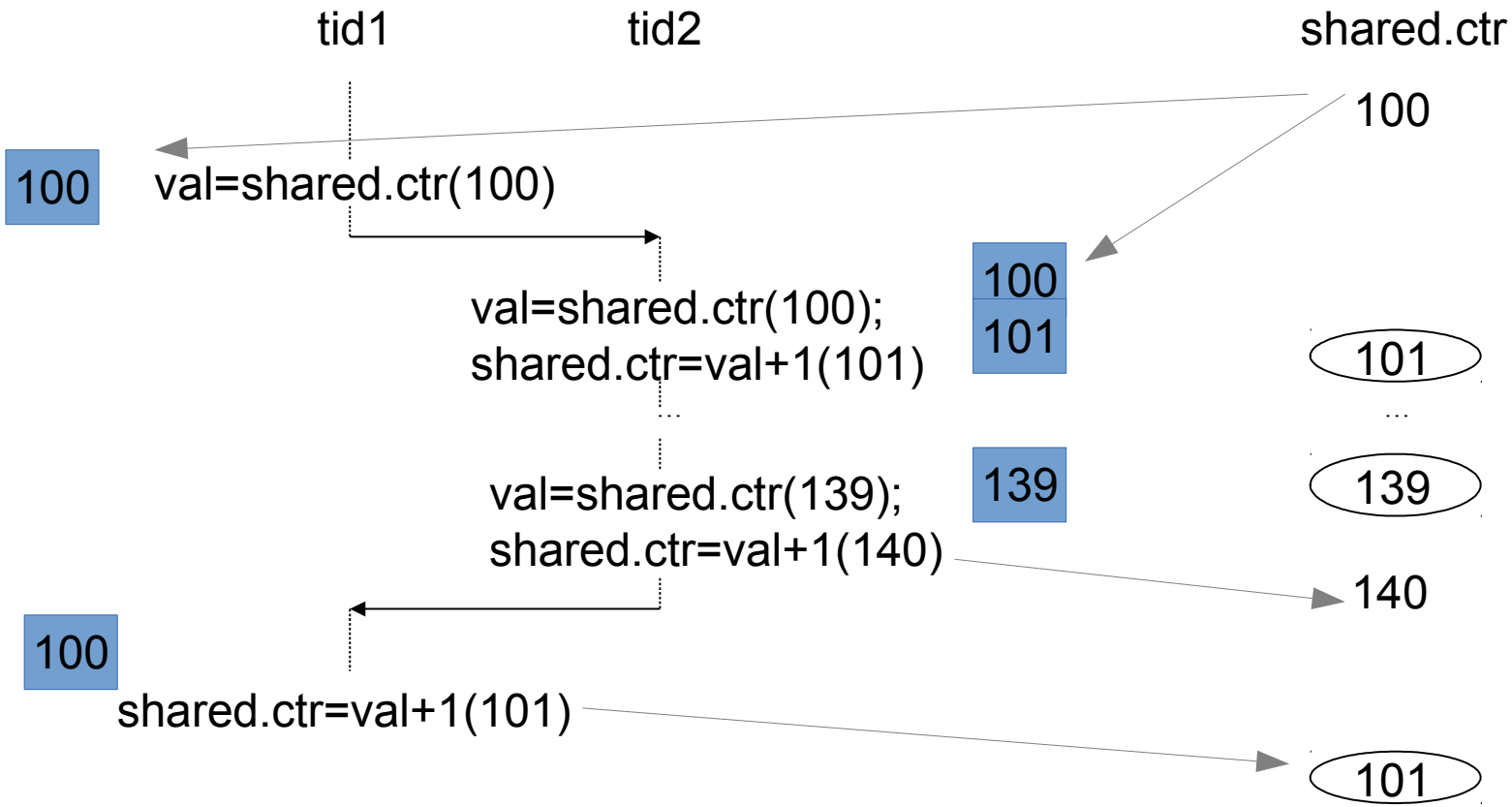
acesso à mesma variável global

- Fios de execução a partilhar acesso à mesma variável global.
  - Programa com duas threads,que somam NITERS à variável global.

```
void *count(void *arg);
struct {
   int ctr;
} shared;
void *count(void *arg) {
  int i,val;
  long this=
      (long)pthread_self();
  for (i=0; i<NITERS; i++) {
     val=shared.ctr;
     printf("%lu: %d\n“,
              this,val);
     shared.ctr=val+1; }
  return NULL;
}
```

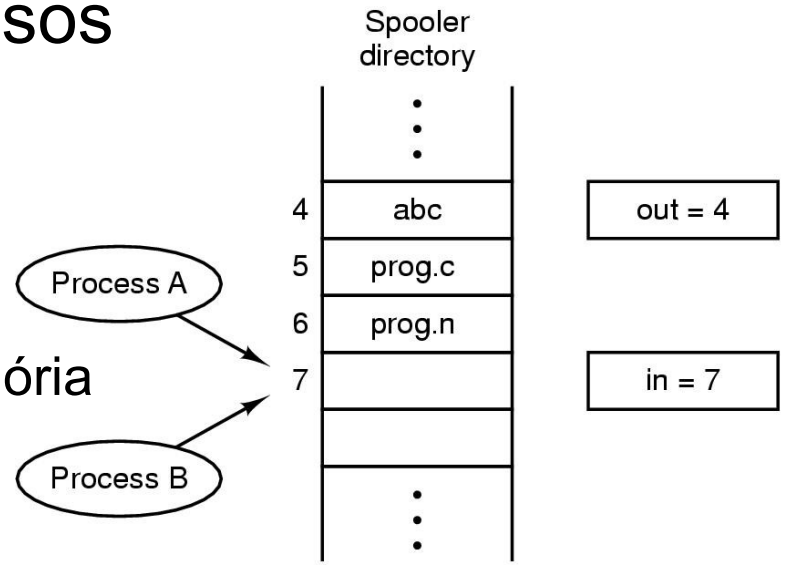
```
int main() {
   pthread_t tid1, tid2;
   pthread_create(&tid1,NULL,
                  count,NULL);
   pthread_create(&tid2,NULL,
                  count, NULL);

   pthread_join(tid1,NULL);
   pthread_join(tid2,NULL);
   if (shared.ctr != NITERS*2)
      printf("BOOM! ctr=%d\n",
              shared.ctr);
   else
      printf("OK ctr=%d\n",
              shared.ctr);
}
```

- A execução de uma LWP pode ser interrompida
  - entre val=shared.ctr; e shared.ctr=val+1;



Acesso ao mesmo dispositivo

- Processos a partilhar acesso ao mesmo dispositivo
- Considere um print spooler, com 2 variáveis acessíveis a todos os processos
  - in: primeira ranhura (“slot”) livre
  - out: primeiro documento a imprimir
- Fragmento de programa dos processos A e B
  - Se out≠in, então
    - Inscrever documento.
    - Incrementar in módulo dimensão da memória tampão (“buffer”)



Acesso ao mesmo dispositivo

```
•
• if (out!=in) {
•
• spoolDir[in] = fd;
•
•
•     in=(in+1) % SIZE;
• }
```

```
•       MOV  %EAX,in
•       CMP  %EAX,out
•       JNE  ok
•       JMP  equal
• ok:   LEA  %EBX,spoolDir
•       MOV  %EDX,fd
•       MOV  [%EBX+%EAX],%EDX
•       INC  %EAX
•       CDQ  %EAX ;EAX->EDX:EAX
•       MOV  %EBX,SIZE
•       IDIV %EBX
•       MOV  in,%EDX
•       JMP  goOn
• equal:
```

Acesso ao mesmo dispositivo

- Um processo (A) pode ser interrompido pelo “scheduler”
  - entre as instruções MOV %EAX,in e MOV in,%EDX.
- O registo EAX mantém o valor inicial de in
  - (na figura, o índice 7).
- O outro processo (B) pode executa todo código sem interrupção:
  - coloca o descritor de ficheiro no índice 7
  - altera valor de in para 8
- Quando o processo A retoma o controlo do CPU
  - insere o descritor do seu ficheiro no índice 7.
- Logo, o processo B não vai ter impresso o seu documento!

Acesso ao mesmos “ficheiros”

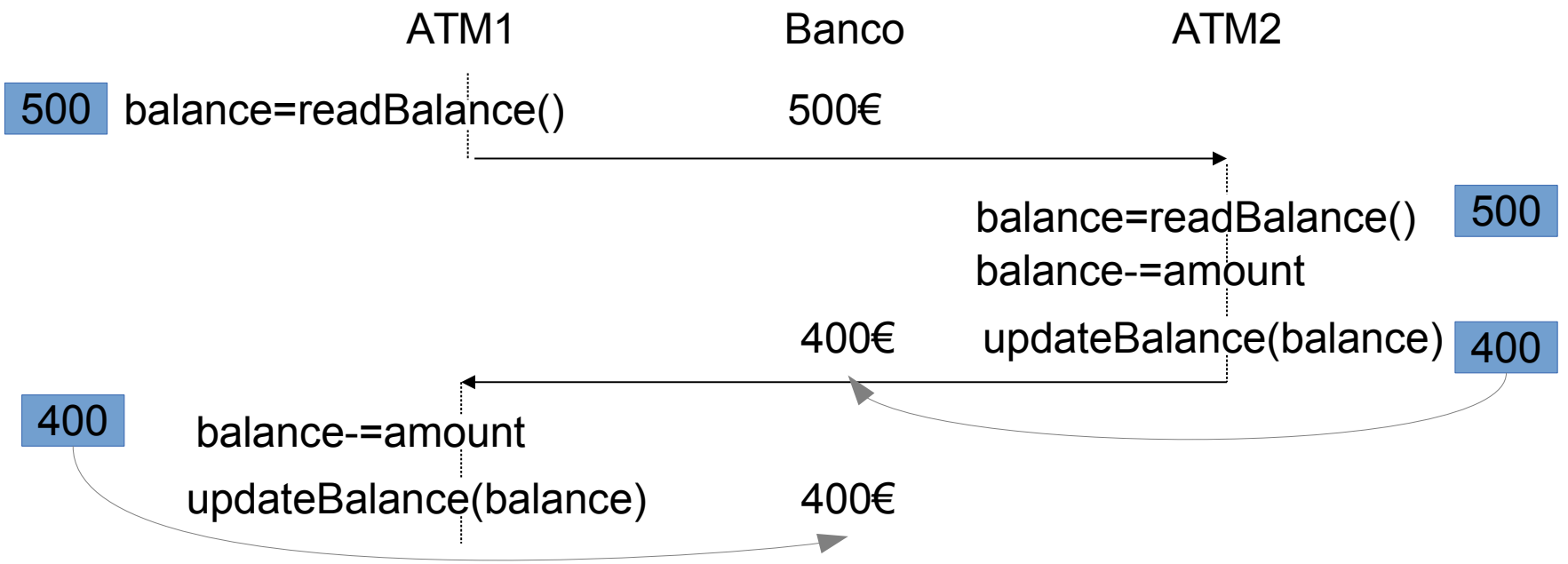
- Processos a partilhar acesso ao mesmos ficheiros
  - sejam duas funções de movimentação de contas bancárias, acedidas por tarefas distintas.

```
int deposit(int account,
            int amount) {
   balance =
         readBalance(account);
   balance += amount;
   updateBalance(account,
                 balance);
   return balance;
}
```

```
int withdraw(int account,
             int amount) {
   balance =
         readBalance(account);
   balance -= amount;
   updateBalance(account,
                 balance);
   return balance;
}
```

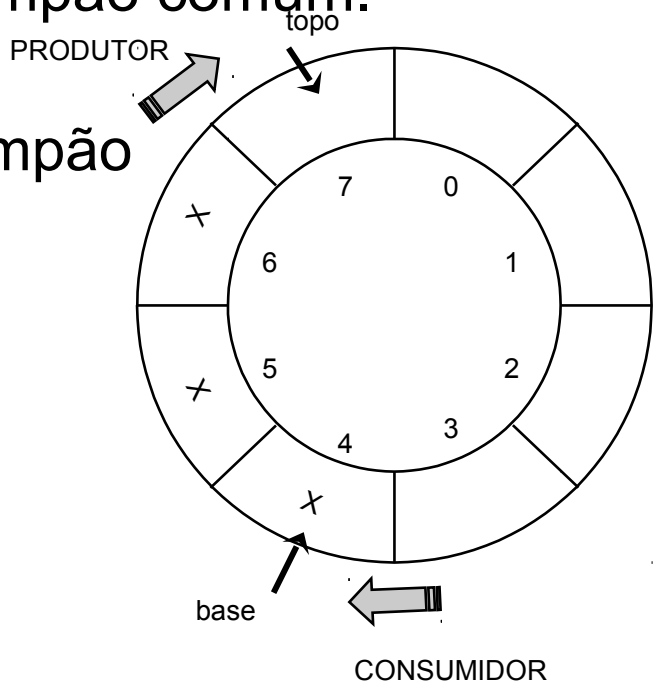
Acesso ao mesmos “ficheiros”

- Duas pessoas com acesso à mesma conta,
  - com saldo inicial de 500€,
  - levantam 100€ em dois multibancos distintos



Produtor-consumidor

- O problema produtor-consumidor,
  - igualmente designado por tampão limitado (“bounded-buffer”),
  - manifesta condições de corrida.
    - Duas tarefas partilham uma memória tampão comum.
      - produtor e consumidor,
    - O produtor insere dados na memória tampão
      - enquanto não estiver cheia
      - bloqueia quando memória enche
    - O consumidor recolhe dados da tampão
      - enquanto não estiver vazia
      - bloqueia quando memória vaga



Produtor-consumidor

- Produtor
  - while(1) {
  - /* gera dado X */
  - while(counter==SIZE)
  - /*buffer cheio*/ ;
  - buffer[topo]=X;
  - counter++;
  - topo=(topo+1)%SIZE;
  - }

MOV %EAX,counter

INC %EAX

MOV counter,%EAX

- Consumidor
  - While(1) {
  - /* consome dados */
  - while(counter==0)
  - /*buffer vazio*/ ;
  - Y=buffer[base];
  - counter--;
  - base=(base+1)%SIZE;
  - }

MOV %EAX,counter

DEC %EAX

MOV counter,%EAX\

Produtor-consumidor

- Consideremos a seguinte ordem de execução de instruções, com counter==4

Entidade	INstrução	Registo	Counter
produtor	MOV %EAX,counter	EAX ← 4	4
produtor	INC %EAX	EAX ← 5	4
consumidor	MOV %EAX,counter	EAX ← 4	4
consumidor	DEC %EAX	EAX ← 3	4
produtor	MOV counter, %EAX	EAX ← 5	5
consumidor	MOV counter, %EAX	EAX ← 3	3

Produtor-consumidor

- Para evitar que o produtor fique a consumir CPU quando a memória tampão estiver cheia,
  - ele deve bloquear.
- O mesmo deve ser feito para o consumidor quando a memória tampão estiver vazia.

- O consumidor deve ser acordado quando
  - produtor inserir um elemento na memória vazia.
- Igualmente, o produtor deve ser acordado quando
  - o consumidor retirar um elemento da memória cheia.

Produtor-consumidor

- Produtor
  - while(1) {
  - /* gera dado X */
  - if (counter==SIZE)
  - pause() ;
  - buffer[topo]=X;
  - counter++;
  - if (counter==1)
  - signal(consumer,SIGALRM);
  - topo=(topo+1)%SIZE;
  - }
- Consumidor
  - while(1) {
  - if (counter==0)
  - pause() ;
  - Y=buffer[base];
  - counter--;
  - if (counter==N-1)
  - signal(producer,SIGALRM);
  - base=(base+1)%SIZE;
  - }

Produtor-consumidor

- As acções de dormir/acordar também manifestam uma condição de corrida!
- Vejamos o seguinte traço (sequência de execução das instruções)
  - O consumidor recolhe o único elemento da memória.
  - Após ter verificado que conter==0,
    - o consumidor é suspenso pelo sistema operativo,
    - SO transfere CPU para o produtor.
  - O produtor insere um elemento.
  - Ao verificar que conter==1,
    - executa a instrução **signal(consumer,SIGALRM)**
    - que não tem efeito porque o consumidor foi suspenso imediatamente antes de executar **pause().**
  - O controlo do CPU volta para o consumidor,
    - que executa **pause().**
  - O produtor enche a memória tampão e acaba por auto suspender-se!

Casos reais

- 1985-1987:
  - Therac25 (aparelho de emissão de feixes de electrões e raiosX)
    - Morte de 3 de 6 pacientes oncológicos
    - Devido a condições de corrida nos controladores dos emissores.
- 14 Agosto 2003:
  - um apagão ocorre no noroeste dos EUA e Canadá, afectando cerca de 50 milhões de pessoas.
  - Uma das causas
    - condição de corrida num monitor de alarmes da rede electrica,
  - programado pela GE em C++.
    - “We had in excess of three million online operational hours in which nothing had ever exercised that bug. I'm not sure that more testing would have revealed it.”
- -- GE Energy's Mike Unum

Condição de corrida

- [Def] Condição de corrida
- [Def] “race condition”
  - Quando resultados dos recursos partilhados dependem da ordem de acesso das tarefas.
- Provém do facto que o resultado depender da tarefa que termina em primeiro lugar.
- Para evitar condições de corrida,
  - as tarefas têm de ser sincronizadas.
- A sincronização força a ordem de eventos (i.e., acesso a recursos partilhados).

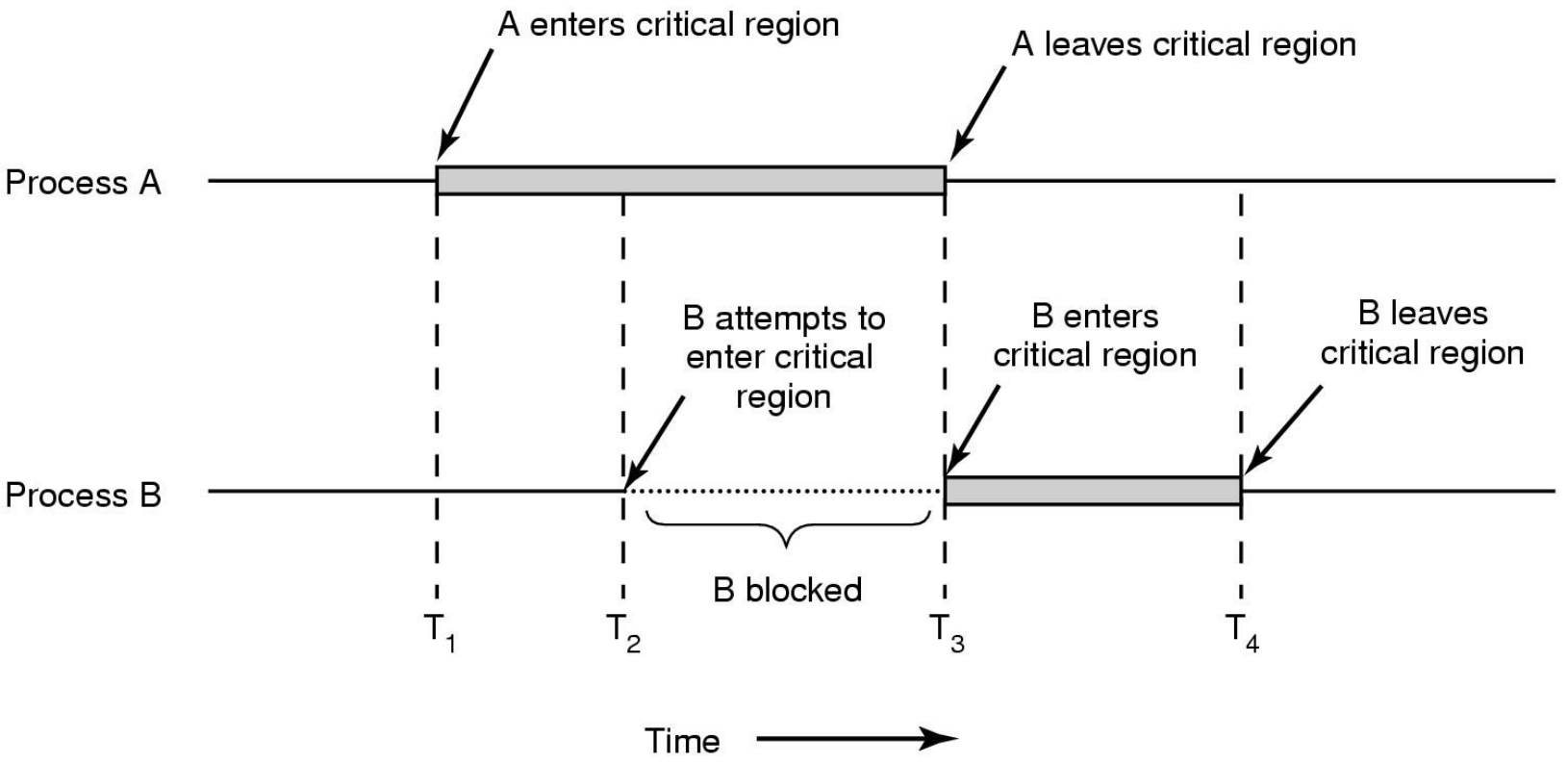
- A race occurs when two threads can access (read or write) a data variable simultaneously and at least one of the two accesses is a write.
  - (Henzinger 04)

Região Crítica

- [Def] Região crítica-RC:
  - fragmento de programa onde são acedidos recursos partilhados, e que só pode ser executado por uma única tarefa de cada vez.
- Região crítica é delimitada por instruções de leitura e escrita do recurso partilhado
- se apenas houver instruções de leitura, execução concorrencial de acessos não gera incoerência nos resultados

- Se uma tarefa se encontrar dentro da região crítica,
  - outra que tente entrar deve ficar bloqueada até que a tarefa saia da região crítica.
-
- Este capítulo é dedicado ao estudo de métodos para as tarefas poderem usar RCs sem depender da ordenaçao de instruçoes (mesmo com vários CPUs).

Região Crítica



Região Crítica

- [Def] Exclusão mútua:
  - mecanismo que assegura que região crítica se encontra a executar, no máximo, numa tarefa.re
  - Nao existem duas(ou mais) tarefas a executar a RC

- Problemas a resolver na sincronização
  - Carência (“starvation”):
    - uma tarefa é sistematicamente ultrapassada por outras e fica indefinidamente à espera de entrar numa região critica.
  - Impasse (“deadlock”):
    - várias tarefas estão à espera das outras para libertar recursos cativados.
  - Garantir que a tarefa não bloqueie dentro da região crítica.

Região Crítica

- Região de entrada (RE)
  - fragmento de programa na qual uma tarefa pede autorização para entrar na RC.
- Região de saída (RS)
  - fragmento de programa na qual uma tarefa se retira da RC.
- Região restante (RR)
  - fragmento de programa que não acede a recursos partilhados

- Formato dos programas
- do {
- Região de Entrada
- Região Crítica
- /* acesso dados partilhados */
- Região de Saída
- Região Restante
- /* zona segura */
- } while(X);

Exclusão Múltipla

- Existem diversas soluções para assegurar a exclusão mútua:
  - Algoritmos de espera activa (Peterson, Lamport)
  - Por hardware,
    - com instruções especiais do processador
  - Por variáveis de tipo especial
    - semáforos, mutexes
  - Por serviços do sistema operativo
    - monitores, passagem de mensagens

Exclusão Múltipla

- Requisitos a satisfazer pelas soluções das RCs
  - Exclusão mútua
    - só uma tarefa pode entrar na RC.
  - Progresso
    - as tarefas em RR não podem impedir outra tarefa de entrar mais tarde na RC.
  - Espera limitada
    - uma tarefa que pretenda entrar na RC, deve poder fazê-lo em tempo limitado.
- Consideram-se assumidas as seguintes condições:
  - A tarefa permanece dentro da RC num tempo finito.
    - Nota: evitar a todo o custo sleep dentro da RC!
  - A velocidade do processador e número de processadores podem ser quaisquer.

Região Crítica

- Se o programador pretender que a região crítica não seja interrompida por sinais, deve mascarar os sinais na forma
  - `sigset_t newmask,oldmask;`
  - `…`
  - `sigprocmask( SIG_BLOCK, &newmask, &oldmask );`
  - `/* região crítica */`
  - `sigprocmask( SIG_SETMASK, &oldmask,NULL );`

- Opção não é atractiva para garantir exclusão entre processos em geral
  - poderiam nunca repor as interrupções
- mas o kernel usa este mecanismo frequentemente

IPC

- [DEF] Comunicação entre processos
- [DEF] IPC-InterProcess Communication
  - conjunto de técnicas usadas na troca de informação entre várias tarefas,
- Divididas em
  - Sincronização: coordenação de várias tarefas.
  - Memória partilhada: acesso da mesma zona de memória por várias tarefas.
  - Passagem de mensagens: envio de uma mensagem do emissor ao recipiente.
  - Invocação remota de procedimentos (RPC-Remote Procedure Call): processo executa programa noutro espaço de endereços.

- As tarefas intervenientes podem residir no mesmo computador, ou em computadores distintos ligados em rede.
- Neste capítulo focamos na sincronização e passagem de mensagens.