Programação Concorrente e Paralela

Capítulo 3 – Simultaneidade e Paralelismo

Marcial Porto Fernández
marcial@larces.uece.br

Semestre 2018.2

Sumário

- Simultaneidade x Paralelismo
- Criação de um processo
- Comunicação e sincronização
- Técnicas de sincronização

Simultaneidade

- Um programa sequencial tem um único fluxo de controle.
  - Sua execução é chamado de processo.

- Um programa simultâneo tem múltiplos fluxos (threads) de controle.
  - Estes fluxos podem ser executados como processos paralelos.

Paralelismo

Um programa concorrente pode ser executado por:

Multiprogramação:	processos *compartilham um ou mais processadores e memória*
Multiprocessamento:	cada processo é executado em seu próprio processador, mas utilizando *memória compartilhada*
Processamento distribuído:	cada processo é executado em *seu próprio processador e própria memória* conectados por uma rede de comunicação

Dificuldades

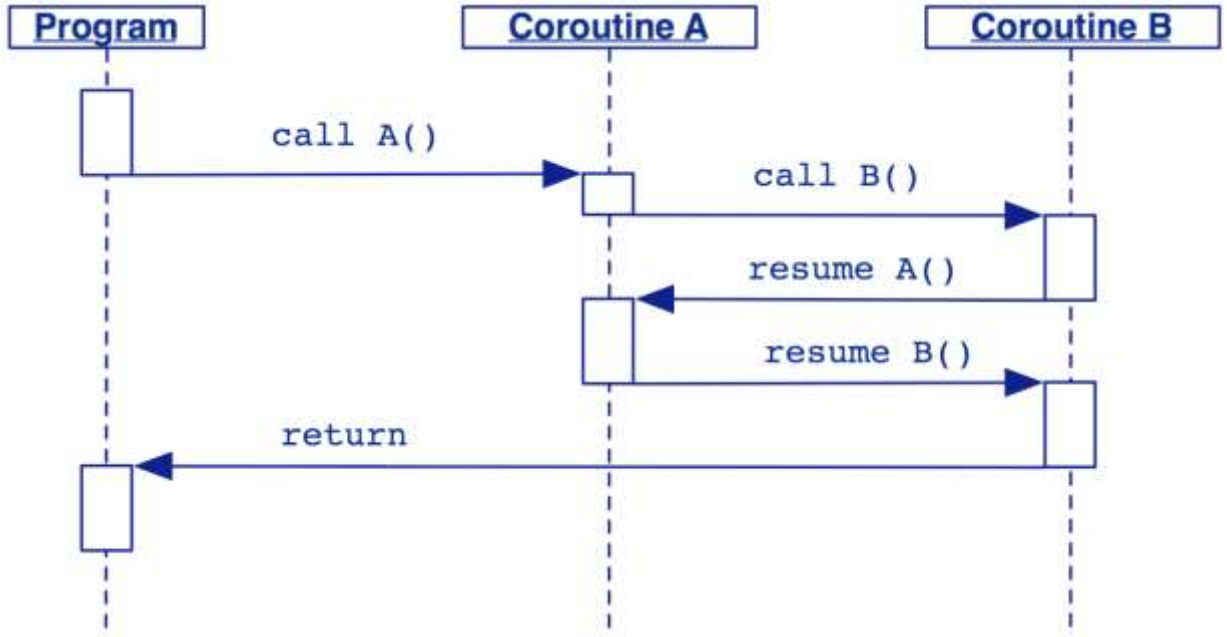
- Mas aplicativos simultâneos podem introduzir uma complexidade no sistema:
  - Segurança: processos simultâneos podem **corromper** os dados compartilhados
  - Vivacidade: processos podem se "**bloquear**" se não forem devidamente coordenados
  - Não-determinismo: a execução do mesmo programa duas vezes pode dar **resultados diferentes**
  - Overhead no processamento: troca de contexto e sincronização **consomem recursos**

Criação de um Processo

- A maioria das linguagens concorrentes oferecem algum dos seguintes procedimentos para criação de processos:
  - Sub-rotinas
  - Fork e Join
  - Cobegin / coend

Sub-rotinas

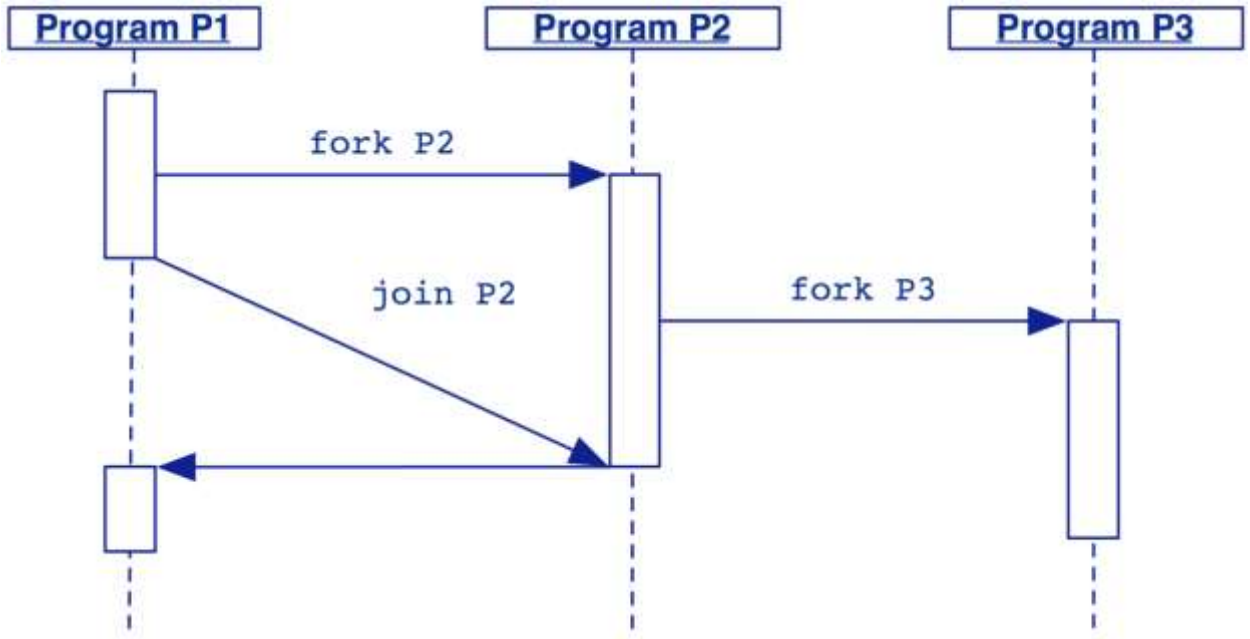
Sub-rotinas são apenas *pseudo-concorrente* e requerem *transferência explícita de controle*



Sub-rotinas podem ser usadas para implementar o nível mais alto (maior abstração) de mecanismos simultâneos.

Fork e Join

Fork pode ser usado para criar qualquer número de processos:
Join espera pelo término de um outro processo.



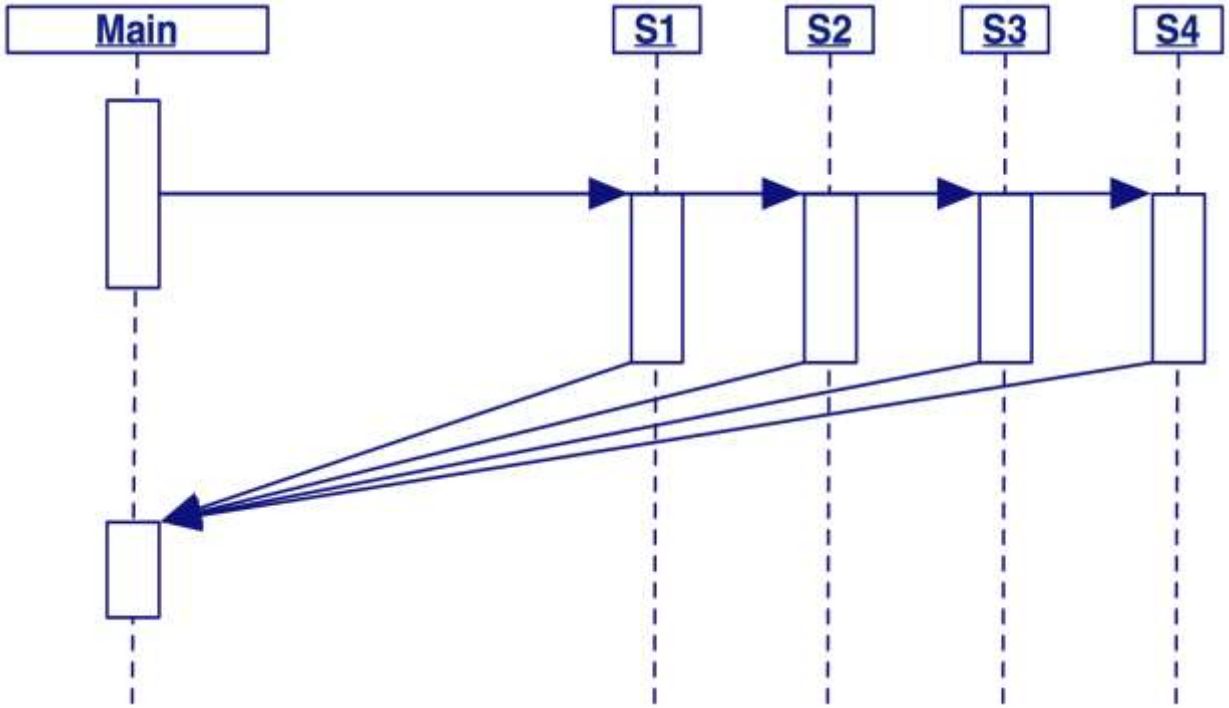
Fork e Join são desestruturadas, por isso necessitam de cuidados e disciplina!

Cobegin/Coend

Blocos cobegin / coend oferecem melhor estruturação:

```
cobegin S1 | | S2 | | ... | | Coend Sn
```

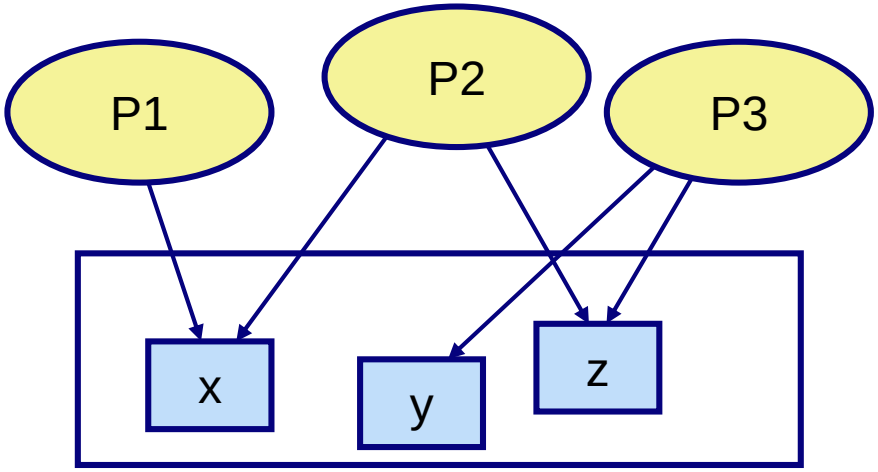
Mas só podem criar um *número fixo de processos*.
O programa principal continua apenas quando todos os coblocks terminarem.

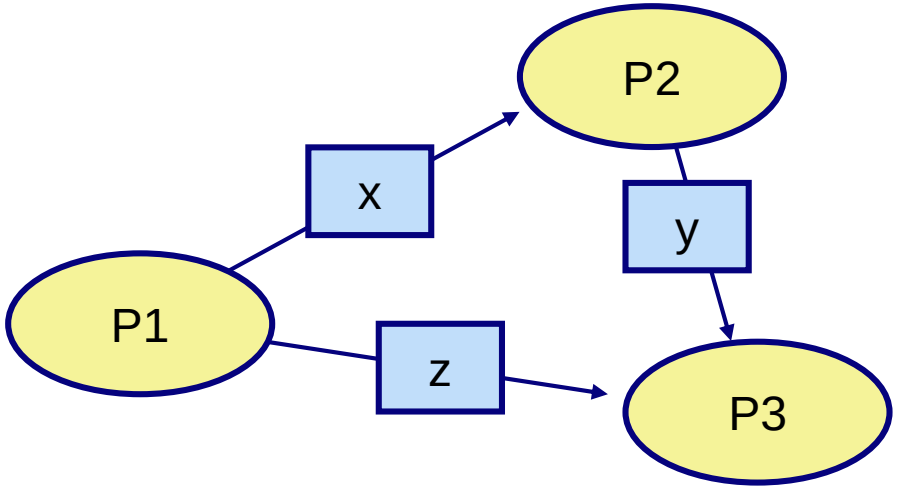


Comunicação e sincronização

Em abordagens baseadas em variáveis compartilhadas, os processos se *comunicam indiretamente*.

São necessários mecanismos de *sincronização explícita*.



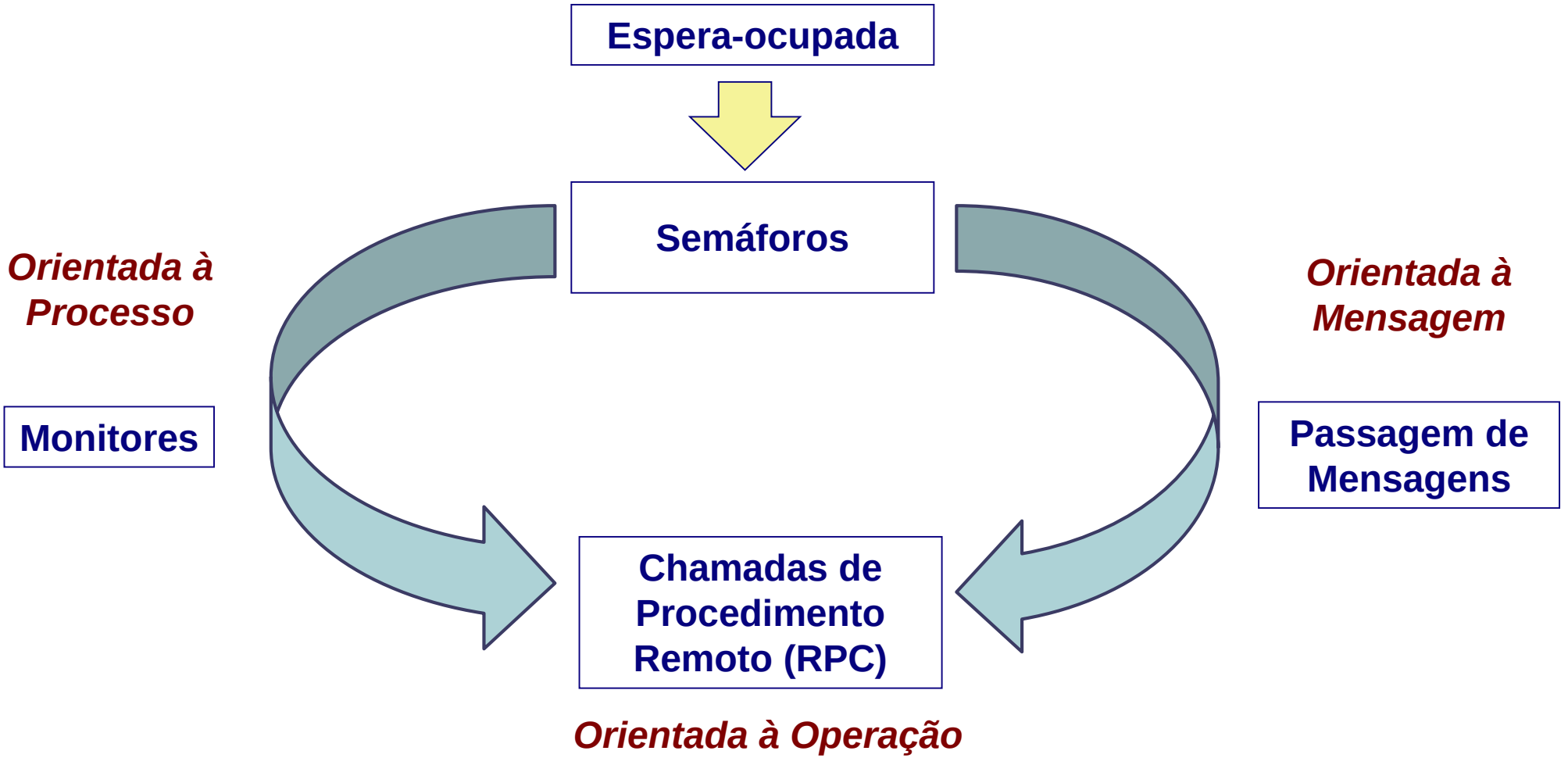


Em abordagens baseadas em passagem de mensagens, a *comunicação e sincronização* são combinados.

A comunicação pode ser *síncronas e assíncronas*.

Técnicas de Sincronização

Diferentes abordagens podem ser equivalentes em capacidade para resolver os problemas e podem ser usado concomitantemente.



Cada abordagem privilegia um estilo diferente de programação

Espera ocupada

- Espera-ocupada (Busy-waiting) é um mecanismo primitivo, mas eficaz.
  - Processos *atomicamente definem e testam* variáveis compartilhadas.
- **Condição de sincronização** é fácil de implementar:
  - Para sinalizar uma condição, um processo **atribui valor** a uma variável compartilhada
  - Para esperar por uma condição, um processo **testa repetidamente** a variável
  - **Exclusão mútua** é mais difícil de implementar de forma correta e eficiente ...

Semáforos

Semáforos foram introduzidos por Dijkstra (1968) como uma primitiva de alto nível para sincronização de processos.

O semáforo é uma variável não-negativa, de valor inteiro com duas operações:

P(s)/Down(s):

espera até que **s > 0**

em seguida, executa atomicamente **s = s – 1** (bloqueia)

V(s)/Up(s):

executa atomicamente **s = s + 1** (desbloqueia)

Programação com semáforos

Muitos problemas podem ser resolvidos utilizando *Semáforos binários*, que assumem valores 0 ou 1.

```
processo P1
     loop
          P(mutex)  {quer entrar}
          Seção Crítica
          V(mutex)  {saída}
          Seção não-crítica
     fim
fim
```

```
P2 processo
     loop
          P(mutex)
          Seção Crítica
          V(mutex)
          Seção não-crítica
     fim
fim
```

Os semáforos geralmente são implementados no núcleo do SO.

Monitores

- O monitor encapsula os recursos e operações que manipula:
  - Operações são invocadas como chamadas de procedimento ordinário
  - Invocações devem ser **mutuamente exclusivas**
  - A condição de sincronização é realizada utilizando primitivas de **wait (esperar)** e **signal (sinalizar)**
  - Existem muitas variações de implementar *wait* e *signal*...

Programação com monitores

```
type buffer(T) = monitor
var
   slots : array [0..N-1] of T;
   head,tail : 0..N-1;
   size : 0..N;
   notfull,notempty: condition;
   size := 0;
   head := 0;
   tail := 0;

procedure deposit(var p:T);
begin
   if size = N then
      notfull.wait
   slots[tail] := p;
   size := size + 1;
   tail := (tail+1) mod N;
   notempty.signal
end
```

```
procedure fetch(var it:T);
begin
  if size = 0 then
      notempty.wait
  it := slots[head];
  size := size - 1;
  head := (head+1) mod N;
  notfull.signal
end
```

Problemas com os monitores

- Os **monitores são mais estruturados do que os semáforos**, mas eles ainda são difíceis de programar:
  - Condições devem ser verificadas manualmente
  - Signal e retorno simultâneos não são suportados
- Um processo de **signal** é temporariamente suspenso para permitir processos **wait** entrar!
  - **Estado do monitor pode mudar** entre o signal e a retomada do sinal
  - Ao contrário dos semáforos, **múltiplos signals não são salvos**
  - Chamadas de **monitores aninhadas** devem ser tratadas com cuidado especial para prevenir deadlock

Passagem de Mensagem (Message Passing)

- Passagem de mensagem **combina comunicação e sincronização**:
  - O remetente especifica a **mensagem e o destino**:
    - um processo, uma porta, um conjunto de processos,...
  - O receptor especifica **variáveis de mensagem e a origem**:
    - a origem pode ou não ser explicitamente identificada

Passagem de Mensagem (Message Passing)

- Mensagem de transferência pode ser:
  - assíncrona: operações de envio **nunca bloqueia**
  - buffered: Remetente **bloqueia se o buffer está cheio**
  - síncrono: Emissor e receptor devem **ambos estar prontos**

Chamada Procedimento Remoto (RPC) - Cliente

```
#include <stdio.h>
#include <utmp.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

/* a program that calls the RUSERSPROG
* RPC program
*/

main(int argc, char **argv)

{
   unsigned long nusers;
   enum clnt_stat cs;
   if (argc != 2) {
     fprintf(stderr, "usage: rusers hostname\n");
     exit(1);
    }

   if( cs = rpc_call(argv[1], RUSERSPROG,
          RUSERSVERS, RUSERSPROC_NUM, xdr_void,
          (char *)0, xdr_u_long, (char *)&nusers,
          "visible") != RPC_SUCCESS ) {
              clnt_perrno(cs);
              exit(1);
            }

   fprintf(stderr, "%d users on %s\n", nusers, argv[1] );
   exit(0);
}
```

Chamada Procedimento Remoto (RPC) - Servidor

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

void *rusers();

main()
{
  if(rpc_reg(RUSERSPROG, RUSERSVERS,
        RUSERSPROC_NUM, rusers,
        xdr_void, xdr_u_long,
        "visible") == -1) {
           fprintf(stderr, "Couldn't Register\n");
            exit(1);
          }
   svc_run(); /* Never returns */
  fprintf(stderr, "Error: svc_run returned!\n");
  exit(1);
}
```

Fim do Capítulo 3

- Esse capítulo apresentou os conceitos de simultaneidade e sincronização.
- Revise os conceitos apresentados.
- Resolver os exercícios da Lista.