

# Linguagem de Programação II

CARLOS EDUARDO BATISTA

CENTRO DE INFORMÁTICA - UFPB

BIDU@CI.UFPB.BR

# Introdução à Programação Concorrente / Processos

- ▶ Desafios da programação concorrente
- ▶ Processos
- ▶ Estados, ações, histórias e propriedades
- ▶ Paralelização: Encontrando padrões em um arquivo



# Desafios da Programação Concorrente

- ▶ **Sistemas *Multithreaded***

- ▶ Sistemas que apresentam mais *threads* do que processadores. Ex.: Gerenciador de janelas.

- ▶ **Sistemas Distribuídos**

- ▶ Sistemas que cooperam entre si, residindo em computadores distintos que se comunicam através de uma rede. Ex.: A Web.

- ▶ **Sistemas Paralelos**

- ▶ Sistemas que necessitam de grande poder de processamento. Ex.: Problemas de otimização.

# Desafios da Programação Concorrente

- ▶ Para desenvolver programas concorrente é necessário conhecer três elementos:
  - ▶ **Regras**
    - ▶ Modelos formais que ajudam a entender e desenvolver programas corretos.
  - ▶ **Ferramentas**
    - ▶ Mecanismos oferecidos pelas linguagens de programação para descrever computações concorrentes.
  - ▶ **Estratégias**
    - ▶ Paradigmas de programação adequados para as aplicações tratadas



# Desafios da Programação Concorrente

- ▶ As aplicações concorrentes devem ser vistas por dois ângulos:
  - ▶ **Computabilidade**
    - ▶ Princípios que abordam o que pode ser computado em um ambiente concorrente.
  - ▶ **Desempenho**
    - ▶ Avaliação do aumento no desempenho.

# Processos

- ▶ Os Sistemas Operacionais atuais executam vários programas de forma concorrente, cada um dos quais se apoderando da CPU por uma determinada fatia de tempo.
- ▶ O mecanismo de trocas rápidas entre os programa é chamado de **multiprogramação**.



# Processos

- ▶ **Conceito**

- ▶ Abstração de um programa em execução.

- ▶ As informações sobre o estado de execução de um programa constituem o **contexto de um processo**.

- ▶ São elas:

- ▶ Código do programa
  - ▶ Dados armazenados na memória
  - ▶ Pilha de execução
  - ▶ Conteúdo dos registradores
  - ▶ Descritores dos arquivos abertos

# Processos



Espaço de  
endereçamento  
único.



# Processos

Apontador da  
pilha

End. do quadro anterior  
Contador de programa  
Retorno da função  
Variáveis locais  
Parâmetros

**f2()**

Contador de programa (10324)  
Retorno da função  
Variáveis locais (a, b)  
Parâmetros (p1, p2)

**f1()**

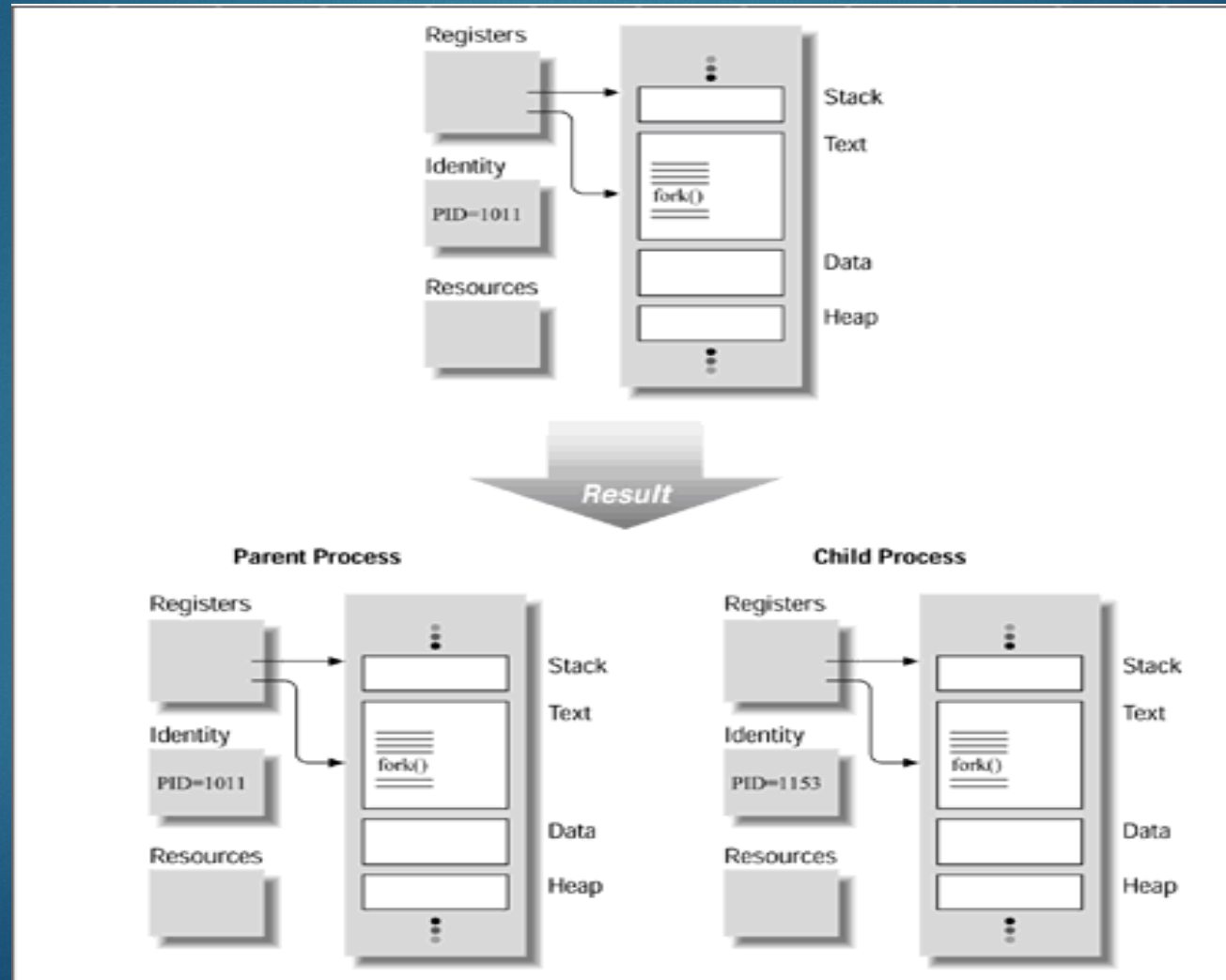
**Faz o  
mesmo  
para f3()**

Endereço  
10324

```
int main() {  
    int x, y, z;  
  
    x = f1(10, 20);  
    z = f3(50, 60);  
  
    return 0;  
}
```

```
int f1(int p1, int p2) {  
    int a = p1 + p2;  
    int b = p1 * p2;  
  
    return a + f2(a, b);  
}
```

# fork()





# Processos

- ▶ **Chamadas de sistema**

- ▶ Meio pelo qual os programas de usuário conversam com o sistema operacional.
- ▶ Forma de “proteger” a máquina dos programas de usuário.
- ▶ Tipos de chamadas:
  - ▶ Gerenciamento de processos
  - ▶ Gerenciamento de arquivos
  - ▶ Gerenciamento do sistema de diretório

# Processos

## ► Chamadas de sistema para o gerenciamento de processos

Chamada	Descrição
<code>pid = fork()</code>	Cria um processo filho idêntico ao processo pai.
<code>pid = waitpid(pid, &amp;statloc, options)</code>	Cria um processo filho e aguardo o seu termino.
<code>s = execve(name, argv, environp)</code>	Cria um processo filho a partir de um programa externo e um conjunto de parâmetros.
<code>exit(status)</code>	Termina a execução do processo corrente e retorna seu estado.
<code>kill()</code>	Finaliza um processo.



# fork() e exit()

```
#include <unistd.h> /* Symbolic Constants */
#include <sys/types.h> /* Primitive System Data Types */
#include <errno.h> /* Errors */
#include <stdio.h> /* Input/Output */
#include <sys/wait.h> /* Wait for Process Termination */
#include <stdlib.h> /* General Utilities */
int main() {
    pid_t childpid; /* variable to store the child's pid */
    int retval, status; /* child process: user-provided return code
                        and parent process: child's exit status */
    childpid = fork();
    if (childpid >= 0) { /* fork succeeded */
        if (childpid == 0) { /* fork() returns 0 to the child process */
            printf("CHILD: I am the child process!\n");
            printf("CHILD: My PID: %d\n", getpid());
            printf("CHILD: Parent's PID: %d\n", getppid());
            printf("CHILD: Value of copy of childpid is: %d\n", childpid);
            printf("CHILD: Sleeping for 1 second...\n");
            sleep(1); /* sleep for 1 second */
            printf("CHILD: Enter an exit value (0 to 255): ");
            scanf("%d", &retval);
            printf("CHILD: Goodbye!\n");
            exit(retval); /* child exits with user-provided return code */
        }
    }
```

```
else { /* fork() returns new pid to the parent process */
    printf("PARENT: I am the parent process!\n");
    printf("PARENT: My PID: %d\n", getpid());
    printf("PARENT: Value of copy of childpid: %d\n", childpid);
    printf("PARENT: Wait for child to exit.\n");
    wait(&status); /* wait for child to exit, and store its status */
    printf("PARENT: Child's exit code: %d\n", WEXITSTATUS(status));
    printf("PARENT: Goodbye!\n");
    exit(0); /* parent exits */
}
}
else { /* fork returns -1 on failure */
    perror("fork"); /* display error message */
    exit(0);
}
}
```

# Nested fork()

```
int pid = fork();
if (pid == 0){
    int pid2 = fork();
    if (pid2 == 0)
        execv("pathtoproducer", NULL);
    else
        execv("pathtoconsumer", NULL);
} else {
```



# fork() e execve()

```
#include <stdio.h>

int main( void ) {
    char *argv[3] = {"Command-line", ".", NULL};
    int pid = fork();
    if ( pid == 0 ) {
        execve( "find", argv ,NULL);
    }
    wait( 2 );
    printf( "Finished executing the parent process\n
           - the child won't get here--\n
           you will only see this once\n" );
    return 0;
}
```

```
$ gcc exec-a.c -o exec-a
$ ./exec-a
.
./exec-a
./exec-a.c
./src
./src/stdarg.c
./src/getpid.c
./src/fork.c
Finished executing the parent process
- the child won't get here
--you will only see this once
$
```

# fork()

```
int main(void) {
    pid_t    pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) { /* first child */
        if ( (pid = fork()) < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit(0); /* parent from second fork == first child */
        sleep(2);
        printf("second child, parent pid = %d\n", getppid());
        exit(0);
    }
    if (waitpid(pid, NULL, 0) != pid) /* wait for first child */
        err_sys("waitpid error");
    exit(0);
}
```



# execve()

```
#include<stdio.h>
main(int argc,char *argv[],char *envp[]){
    printf("File name: %s\n",argv[0]);
    printf("%s %s",argv[0],argv[1]);
}
```

```
$ gcc hello.c -o hello
$ gcc execve.c -o execve
$ ./execve
Filename: hello
hello world
```

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

main() {
    char *temp[] = {"hello","world",NULL};
    execve("hello",temp,NULL);
    printf("world");
}
```

# fork(), exit(), \_exit() e waitpid()

```
#include <sys/types.h>
/* pid_t */
#include <sys/wait.h>
/* waitpid */
#include <stdio.h>
#include <stdlib.h>
/* exit */
#include <unistd.h>
/* _exit, fork */
```

```
int main(void) {
    pid_t pid;
    pid = fork();
    if (pid == -1) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        printf("Hello from the child process!\n");
        _exit(EXIT_SUCCESS);
    } else {
        int status;
        (void)waitpid(pid, &status, 0);
    }
    return EXIT_SUCCESS;
}
```



# Processos

## ► O interior de um shell

```
#define TRUE 1

while (TRUE) {                                /* repita para sempre */
    type_prompt( );                          /* mostra prompt na tela */
    read_command(command, parameters);       /* lê entrada do terminal */

    if (fork( ) !=0) {                       /* cria processo filho */
        /* Parent code. */
        waitpid(-1, *status, 0);            /* aguarda o processo filho acabar */
    } else {
        /* Child code. */
        execve(command, parameters, 0);     /* executa o comando */
    }
}
```

# Leitura complementar

- ▶ <http://courses.engr.illinois.edu/cs241/sp2012/lectures/12-forks.c>



# Processos

## ▶ Criação de processos

- ▶ Quatro eventos principais motivam a criação de processos:
  - ▶ Início do sistema.
  - ▶ Execução de uma chamada de sistema para criação de processo ( `fork()` ) por outro processo.
  - ▶ Requisição do usuário.
  - ▶ Início de um job em lote.

# Processos

- ▶ **Término de processos**

- ▶ Condições de término

- ▶ Saída normal (voluntária – chamada `exit()`).
    - ▶ Saída por erro (voluntária).
    - ▶ Erro fatal (involuntário).
    - ▶ Cancelamento por um outro processo (involuntário – chamada `kill()` ).

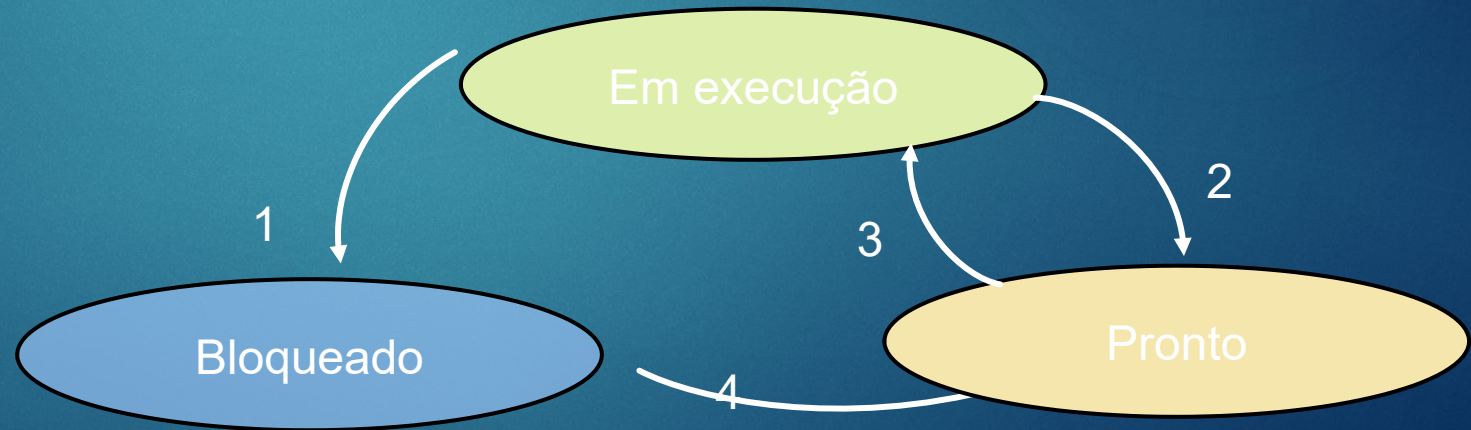


# Processos

- ▶ **Estados de um processo**

- ▶ **Ciclo de vida**

1. O processo bloqueia aguardando uma E/S.
2. O escalonador seleciona outro processo.
3. O escalonador seleciona este processo.
4. A E/S torna-se disponível.



# Processos

## ● Implementação de processos

### Gerenciamento de processos

Registradores  
Contador de programa  
Palavra de estado do programa  
Ponteiro de pilha  
Estado do processo  
Prioridade  
Parâmetros de escalonamento  
Identificador (ID) do processo  
Processo pai  
Grupos do processo  
Sinais  
Momento em que o processo iniciou  
Tempo usado da CPU  
Tempo de CPU do filho  
Momento do próximo alarme

### Gerenciamento de memória

Ponteiro para o segmento de código  
Ponteiro para o segmento de dados  
Ponteiro para o segmento de pilha

### Gerenciamento de arquivos

Diretório-raiz  
Diretório de trabalho  
Descritores de arquivo  
Identificador (ID) do usuário  
Identificador do grupo



# Estados, ações, histórias e propriedades

- ▶ **Estado**

- ▶ Consiste nos valores das variáveis do programa em um dado momento durante sua execução.

- ▶ **Ações atômicas**

- ▶ Um processo executa uma sequência de instruções, as quais são implementadas por uma ou mais ações indivisíveis, denominadas **ações atômicas**.

# Estados, ações, histórias e propriedades

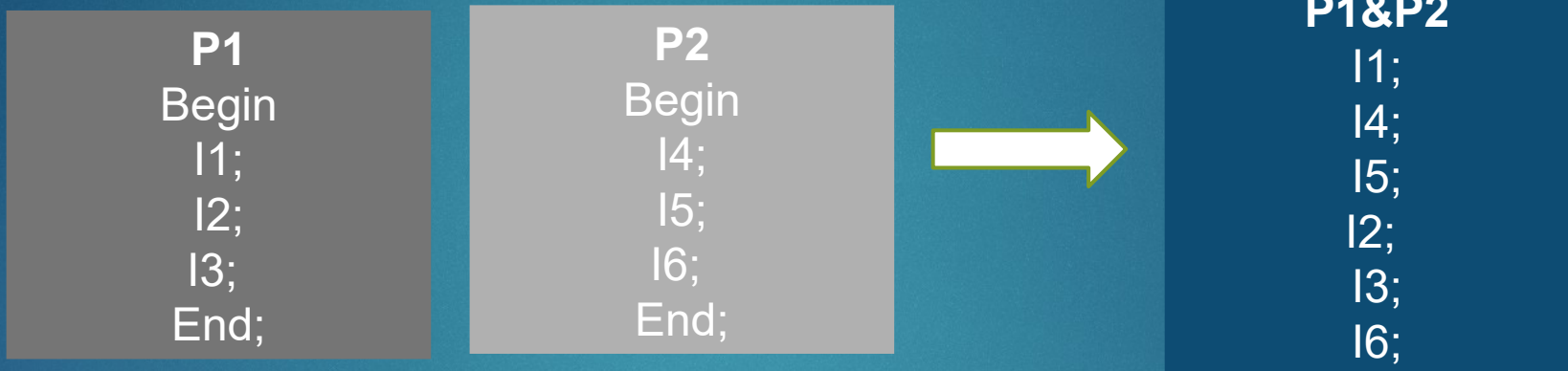
## ▶ Histórias

- ▶ A execução de um programa concorrente resulta em um entrelaçamento (*interleaving*) de sequências de ações atômicas executadas pelos processos.
- ▶ Uma história é uma sequência de estados  $s_0 \rightarrow s_1 \dots \rightarrow s_n$  pela qual passa um programa corrente em uma execução qualquer.



# Estados, ações, histórias e propriedades

## ► Histórias



- O número de histórias possíveis é de  $(n*m)!/(m!)^n$

Onde, ***n*** = número de processos e

***m*** = número de instruções atômicas em  
cada processo.

# Estados, ações, histórias e propriedades

## ▶ Propriedades

- ▶ Uma propriedade é um atributo válido para todas as possíveis histórias de um programa concorrente.
- ▶ Duas propriedades são desejadas em um software concorrente:
  - ▶ *Safety property*
  - ▶ *Liveness property*



# Estados, ações, histórias e propriedades

- ▶ ***Safety property***

- ▶ Garante que o programa nunca entra em um estado indesejado, ou seja, um estado em que as variáveis tem valores indesejados.
- ▶ Falhas levam a um comportamento indesejado.
- ▶ Exemplos
  - ▶ **Ausência de *deadlock*:** Processos não deixam de executar, esperando por eventos que nunca ocorrem.
  - ▶ **Exclusão mútua:** No máximo um processo executa uma região crítica.

# Estados, ações, histórias e propriedades

- ▶ ***Liveness property***

- ▶ Garante que o programa em algum momento entrará no estado desejado.
- ▶ Falhas levam à ausência de um comportamento esperado.
- ▶ Exemplo
  - ▶ **Terminação:** Um programa em algum momento terminará.
  - ▶ **Entrada eventual:** Um processo aguardando entrar na região crítica.



# Leitura complementar

- ▶ [http://www.gnu.org/software/libc/manual/html\\_node/POSIX-Safety-Concepts.html#POSIX-Safety-Concepts](http://www.gnu.org/software/libc/manual/html_node/POSIX-Safety-Concepts.html#POSIX-Safety-Concepts)

# Paralelização: Encontrando padrões em um arquivo

```
string line;  
read a line of input from stdin into line;  
while (!EOF) {      # EOF is end of file  
    look for pattern in line;  
    if (pattern is in line)  
        write line;  
    read next line of input;  
}
```



# Paralelização: Encontrando padrões em um arquivo

- ▶ Como o programa anterior pode ser paralelizado?
- ▶ O principal requisito para a paralelização de um programa é que ele contenha partes **independentes**.

# Paralelização: Encontrando padrões em um arquivo

- ▶ **Independência de processos paralelos**

- ▶ Considere que o conjunto de leitura de uma parte de um programa seja formado pelas variáveis que ele lê mas não altera, e o conjunto de escritas pelas variáveis que ele altera. Duas partes de um programa são independentes se o conjunto de escrita de cada parte é disjunto de ambos os conjuntos de leitura e escrita da outra parte.



# Paralelização: Encontrando padrões em um arquivo

```
string line;  
read a line of input from stdin into line;  
while (!EOF) {  
    co look for pattern in line;  
    if (pattern is in line)  
        write line;  
    // read next line of input into line;  
    oc;  
}
```

# Paralelização: Encontrando padrões em um arquivo

```
string line1, line2;  
read a line of input from stdin into line1;  
while (!EOF) {  
    co look for pattern in line1;  
    if (pattern is in line1)  
        write line1;  
    // read next line of input into line2;  
    oc;  
}
```



# Paralelização: Encontrando padrões em um arquivo

```
string line1, line2;  
read a line of input from stdin into line1;  
while (! EOF) {  
    co look for pattern in line1;  
    if (pattern is in line1)  
        write line1;  
    // read next line of input into line2;  
    oc;  
    line1 = line2;  
}
```

# Paralelização: Encontrando padrões em um arquivo

```
string buffer; # contains one line of input
bool done = false; # used to signal termination
co # process 1: find patterns
    string line1;
    while (true) {
        wait for buffer to be full or done to be true;
        if (done) break;
        line1 = buffer;
        signal that buffer is empty;
        look for pattern in line1;
        if (pattern is in line1)
            write line1;
    }
// # process 2: read new lines
string line2;
while (true) {
    read next line of input into line2;
    if (EOF) {done = true; break; }
    wait for buffer to be empty;
    buffer = line2;
    signal that buffer is full;
}
oc;
```



# Ações atômicas e o comando await

- ▶ Na execução de um programa concorrente, nem todos os entrelaçamentos de ações atômicas são aceitáveis.
- ▶ O papel da sincronização é evitar os entrelaçamentos indesejados.
- ▶ Isso pode ser feito através de duas maneiras:
  - ▶ Agrupar ações de granularidade fina em ações de granularidade grossa;
  - ▶ Congelar a execução de um processo até que uma condição seja satisfeita.

# Ações atômicas e o comando await

- ▶ **Ações atômicas de granularidade fina**

- ▶ Uma ação atômica é aquela que causa mudança de estado de forma indivisível, isto é, qualquer estado intermediário que possa existir na implementação da ação não é visível por outros processos.
- ▶ Uma ação atômica de granularidade fina é aquela que é implementada diretamente pelo hardware.



# Ações atômicas e o comando await

- ▶ Ações atômicas de granularidade fina

```
int y = 0, z = 0;  
co x = y + z; #op1  
// y = 1;      #op2  
   z = 2;      #op3  
oc;
```

Considerando que as operações são atômicas:

Ordem	y	z	x
op1-op2-op3	1	2	0
op2-op3-op1	1	2	3
op2-op1-op3	1	2	1

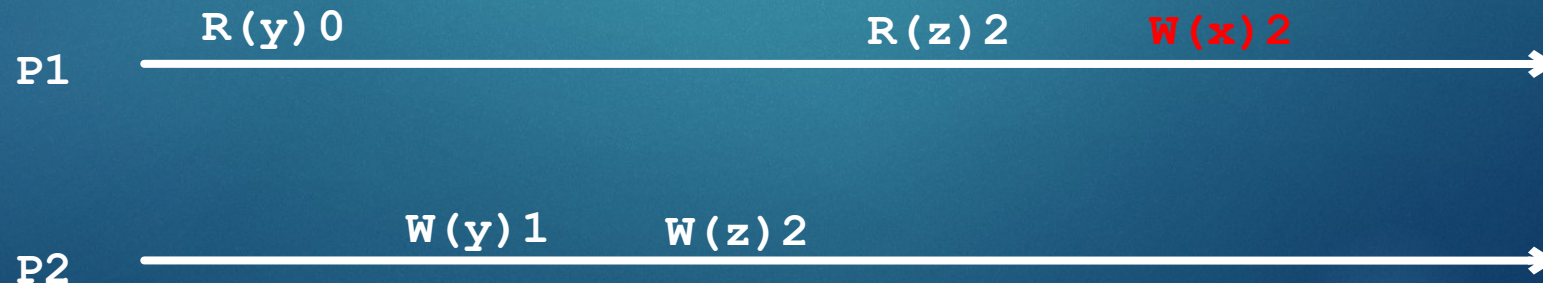
O resultado  $x = 2$  é possível porque op1 não é atômica!

# Ações atômicas e o comando await

- ▶ Ações atômicas de granularidade fina

```
int y = 0, z = 0;  
co x = y + z; #op1  
// y = 1;      #op2  
   z = 2;      #op3  
oc;
```

Ordem: **op1-op2-op3**





# Ações atômicas e o comando await

- ▶ Ações atômicas de granularidade fina
  - ▶ Existem duas situações nas quais uma atribuição vai se **comportar** como se fosse atômica:
    - ▶ A expressão do lado direito não faz referência a nenhuma variável alterada por outro processo;
    - ▶ A atribuição satisfaz a propriedade **At-Most-Once**.

# Ações atômicas e o comando await

- ▶ Propriedade At-Most-Once

- ▶ Uma **referência crítica** em uma expressão é uma referência a uma variável modificada por outro processo.

- Uma atribuição satisfaz a propriedade **At-Most-Once** se (1) contém no máximo uma referência crítica e não é lido por outro processo, ou (2) não possui referências críticas, caso no qual pode ser lido por outro processo.



# Ações atômicas e o comando await

## ► Propriedade At-Most-Once

```
int x = 0, y = 0;  
co x = y + 1; #op1  
// y = y + 1; #op2  
oc;
```

Considerando que todas as operação são atômicas:

Ordem	x	y
op1-op2	1	1
op2-op1	2	1

As operações **parecem** ser atômicas, pois elas satisfazem a propriedade **At-Most-Once**.

# Especificando sincronização: Comando await

- ▶ Em inúmeras situações, necessitamos executar uma sequência de instruções como se fosse uma única ação atômica.
  - ▶ Ex.: Operações bancárias.

< >

```
int y = 0, z = 1;  
co    x = y + z;    #op1  
    // y = 1;      #op2  
    z = x;          #op3  
oc;
```



# Especificando sincronização: Comando await

- ▶ Exclusão mútua

`<S ;>`

`<x = x + 1; y = y + 1>`

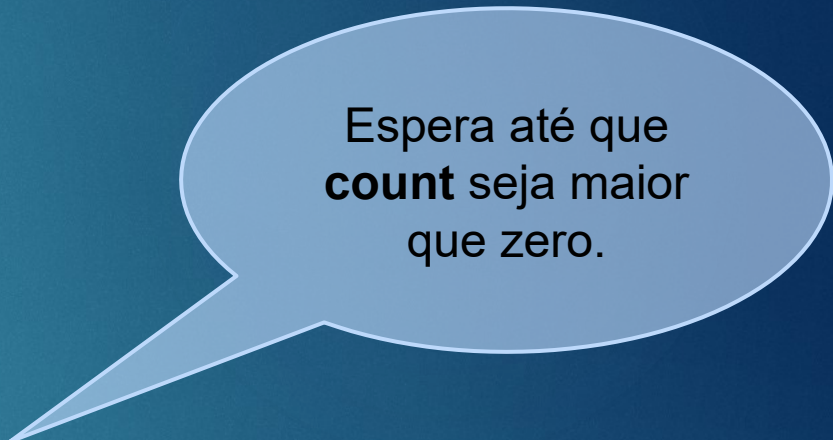
- Sincronização por condição

`<await (B)>`

`<await (count > 0) ;>`

- Ambos

`<await (B) S ;>`



Espera até que  
**count** seja maior  
que zero.

# Especificando sincronização: Comando await

- ▶ **Ações atômicas incondicionais**

- ▶ São aquelas que não contém uma condição de espera B.
- ▶ Ex.: Ações atômicas de granularidade fina ou grossa e instruções await em que a condição de guarda é sempre verdadeira.

- ▶ **Ações atômicas condicionais**

- ▶ Instruções await com uma condição de guarda B.



# Sincronização Produtor/Consumidor

```
int buf, p = 0, c = 0;
process Producer {
    int a[n];
    while (p < n) {
        <await (p == c);>
        buf = a[p];
        p = p+1;
    }
}

process Consumer {
    int b[n];
    while (c < n) {
        <await (p > c);>
        b[c] = buf;
        c = c+1;
    }
}
```

# Produtor/Consumidor

```
#include <iostream>
using namespace std;
#include <unistd.h>
#include "sem.h"

int CreateProcess(void (*)()); /* func. prototype */
int psem, csem; /* semaphores */
int *pn;

main() {
    void producer(), consumer();
    pn = (int *)shmcreate(sizeof(int));
    *pn = 0;
    csem = semcreate(0);
    psem = semcreate(1);
    CreateProcess(producer);
    consumer(); // let parent be the consumer
    semdelete(csem);
    semdelete(psem);
    shmdelete((char *)pn);
}
```



# Produtor/Consumidor

```
void producer() {
    int i;
    for (i=0; i<5; i++) {
        semwait(psem);
        (*pn)++; // increment n by 1
        semsignal(csem);
    }
}
void consumer() {
    int i;
    for (i=0; i<5; i++) {
        semwait(csem);
        cout << "n is " << *pn << '\n'; // print value of n
        semsignal(psem);
    }
}
```

# Produtor/Consumidor

```
int CreateProcess(void (*pFunc)()) {
    int pid;

    if ((pid = fork()) == 0) {
        (*pFunc)();
        exit(0);
    }
    return(pid);
}
```



# Leitura complementar

- ▶ <http://cs.unitbv.ro/~costel/solr/semafoare/shm.h>
- ▶ <http://cs.unitbv.ro/~costel/solr/semafoare/shm.c>

# Exercícios

- ▶ Usar **await** no exemplo de busca por padrão em arquivo de texto
- ▶ Usar **await** no problema produtor/consumidor (cópia de array)
- ▶ Implementar com processos (fork()) 😊



# Exercícios

- ▶ Quantas histórias possíveis?
- ▶ Quais valores finais são possíveis para **x** e **y**?

```
int x = 0, y = 0;  
co x = x + 1; x = x + 2;  
// x = x + 2; y = y - x;  
oc
```

# Referências

- ▶ Notas de Aula do Prof. Bruno Pessoa
- ▶ Andrews, G. *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley, 2000.
- ▶ Rosseto, S. , “Cap. I: Introdução e histórico da programação concorrente”. Disponível em: <<http://www.dcc.ufrj.br/~silvana/compconc/cap-1.pdf>>.