Introdução à Arquitetura de Computadores

Alexandre Meslin

Material baseado nos slides de:
Anderson Oliveira da Silva

Departamento de Informática
PUC-Rio

Parte III – Organização Paralela

- Processamento Paralelo (Threads) – Aula Prática
  - Conceito de Thread
    - » Threads Modo Usuário, Threads Modo Kernel e Threads Híbridas
  - Threads POSIX x Threads Nativas
  - Biblioteca PThread – Parte 1
    - » Exemplos Práticos 1
    - » Exercício Prático 1
  - Problemas de Concorrência e Sincronização
    - » Semáforos Mutex
    - » Semáforos Contadores
    - » Monitores (Signal e Wait)
  - Biblioteca PThread – Parte 2
    - » Exemplos Práticos 2
    - » Exercício Prático 2

Parte III – Organização Paralela

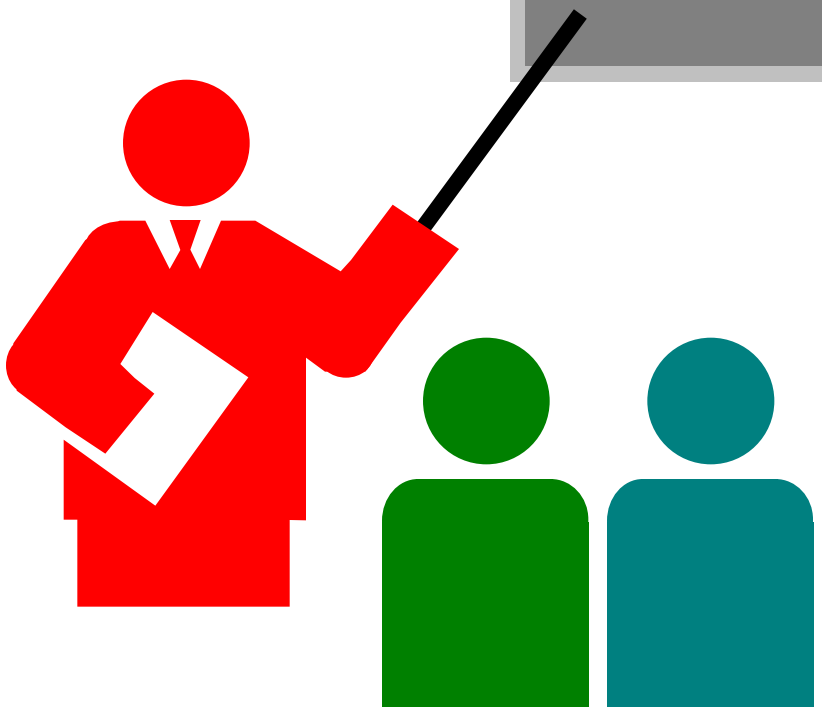
- Referências

  - The Single Unix Specification, Version 2.
    The Open Group. 1997.
    - » https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html

  - POSIX Threads Programming.
    Blaise Barney, Lawrence Livermore National Laboratory.
    - » https://computing.llnl.gov/tutorials/pthreads

  - Arquitetura de Sistemas Operacionais
    - » Machado, Francis B.; Maia, Luiz P.; Ed. LTC.

Organização Paralela



Processamento Paralelo – Conceito de Thread

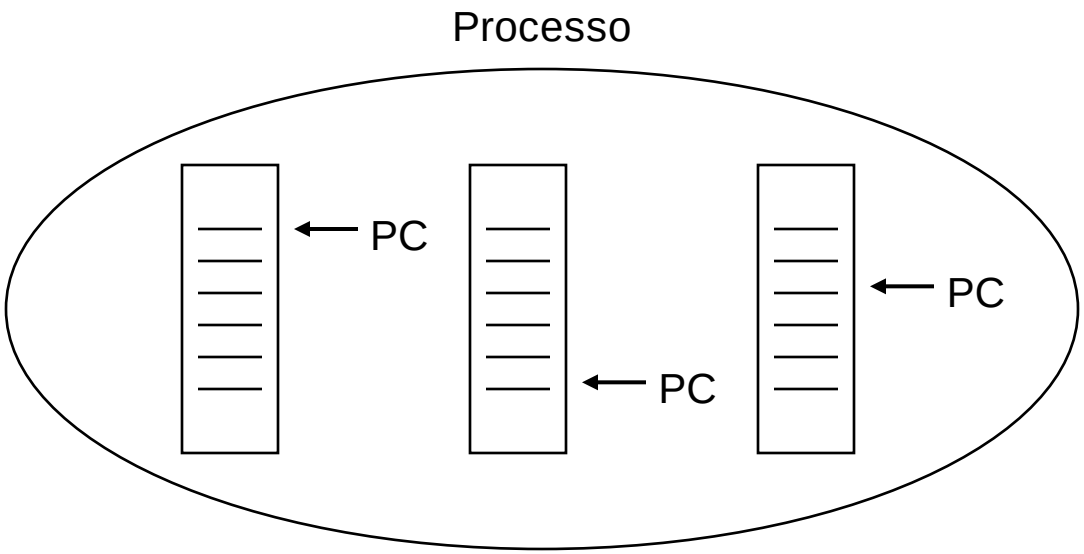
- Thread
  - Trecho de código (tipicamente uma sub-rotina) do programa que executa concorrentemente com outros trechos e que compartilha o contexto de software e a área de endereçamento de um mesmo processo.
    - » Podem alterar dados uns dos outros com menos overhead de processamento.
  - Desenvolvido para trabalhar de forma cooperativa desempenhando uma tarefa em conjunto.
    - » Ex: Thread principal recebe requisições e delega o tratamento delas para threads secundárias.
  - Menor overhead de criação quando comparado à criação de processos filhos.
  - Compartilham o processador da mesma maneira que um processo.

Processamento Paralelo – Conceito de Thread

- Processo x Thread
  - Processo lightweight (peso leve)
    - » Conceito introduzido no sistema operacional Toth, em 1979.
    - » A separação clara entre o conceito de processo e thread foi identificada durante o desenvolvimento do sistema operacional Mach (Carnegie Mellon, 1980).
  - Processo monothread
    - » Processo com um único thread associado.
  - Processo multithread
    - » Processo com vários threads associados.

Processamento Paralelo – Conceito de Thread

- Processo x Thread



Processamento Paralelo – Conceito de Thread

- Thread – Implementação
  - O conjunto de rotinas disponíveis para a utilização de threads é chamado *pacote de threads*.
    - » O tipo de implementação pode influenciar no desempenho, na concorrência e na modularidade.
  - Tipos de Threads:
    - » Modo Usuário
      - Biblioteca de rotinas fora do núcleo do sistema operacional.
    - » Modo Kernel
      - Biblioteca de rotinas do próprio núcleo do sistema.
    - » Modo Híbrido
      - Combinação dos anteriores ou pelo modelo scheduler activations.

Processamento Paralelo – Conceito de Thread

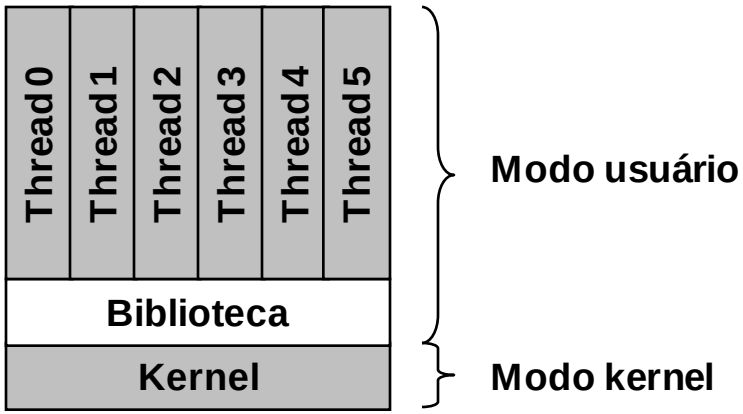
- Threads em Modo Usuário
  - A vantagem é poder implementar threads mesmo em sistemas operacionais que não oferecem esse recurso.
  - São rápidos e eficientes pois dispensam acessos ao kernel do sistema operacional, evitando assim a mudança de modo de acesso (usuário-kernel-usuário).
  - A grande limitação é que o sistema operacional gerencia o processo como se existisse apenas uma thread, o que faz com que todo o processo fique bloqueado se uma rotina bloqueante do sistema for chamada por qualquer uma das threads.
    - » Para contornar essa situação, todas as rotinas bloqueantes de um sistema devem ser implementadas pela biblioteca como rotinas não-bloqueantes.

Processamento Paralelo – Conceito de Thread

- Threads em Modo Usuário
  - O maior problema é o tratamento de sinais (informação assíncrona sobre eventos) enviados para os processos, no sentido de identificar para qual thread o sinal realmente foi enviado.
  - Outro problema é que em sistemas multiprocessados, não é possível que múltiplos threads de um processo sejam distribuídos entre os vários processadores pois o sistema trata apenas processos.

Processamento Paralelo – Conceito de Thread

- Threads em Modo Usuário

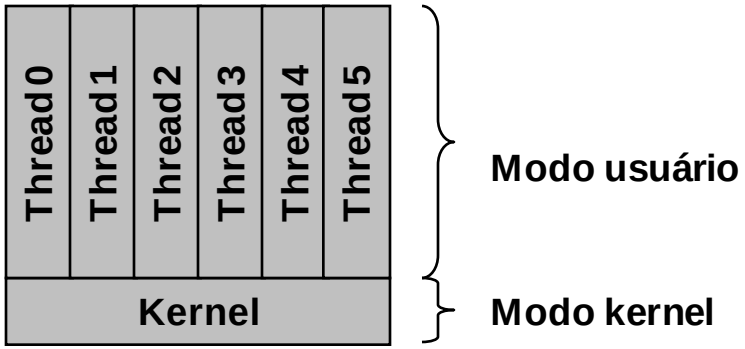


Thread

- Threads em Modo Kernel
  - São implementados diretamente pelo núcleo do sistema operacional, através de chamadas a rotinas do sistema.
    - » Oferecem todas as funções de gerenciamento e sincronização.
  - O sistema operacional tem conhecimento de cada thread do processo e pode selecionar qualquer uma delas, no estado pronto, para execução.
  - Podem ser executadas em paralelo por múltiplos processadores.
  - Overhead da mudança de acesso (usuário-kernel-usuário) se comparadas as threads em modo usuário.
  - Um thread que execute uma rotina bloqueante, não bloqueia outro thread.

Processamento Paralelo – Conceito de Thread

- Threads em Modo Kernel



Processamento Paralelo – Conceito de Thread

- Threads em Modo Kernel
  - Comparação entre tempos de latência:

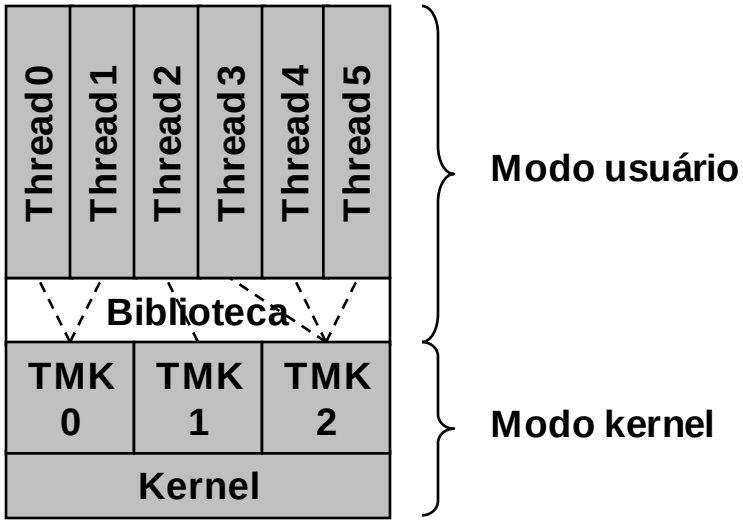
Implementação	Operação 1 (μs)	Operação 2 (μs)
Subprocessos	11.300	1.840
Threads em modo kernel	948	441
Threads em modo usuários	34	37

Processamento Paralelo – Conceito de Thread

- Threads em Modo Híbrido
  - Combina as vantagens dos threads em modo usuário e modo kernel.
  - Threads em modo kernel podem ter um ou vários threads em modo usuário.
  - Apesar da maior flexibilidade, herda os problemas dos outros modos:
    - » Um thread em modo kernel que é bloqueado, bloqueia todos os seus threads em modo usuário.
    - » Para que *n* threads em modo usuário possam executar em diferentes processadores, é necessário ter *n* threads respectivas em modo kernel.

Processamento Paralelo – Conceito de Thread

- Threads em Modo Híbrido

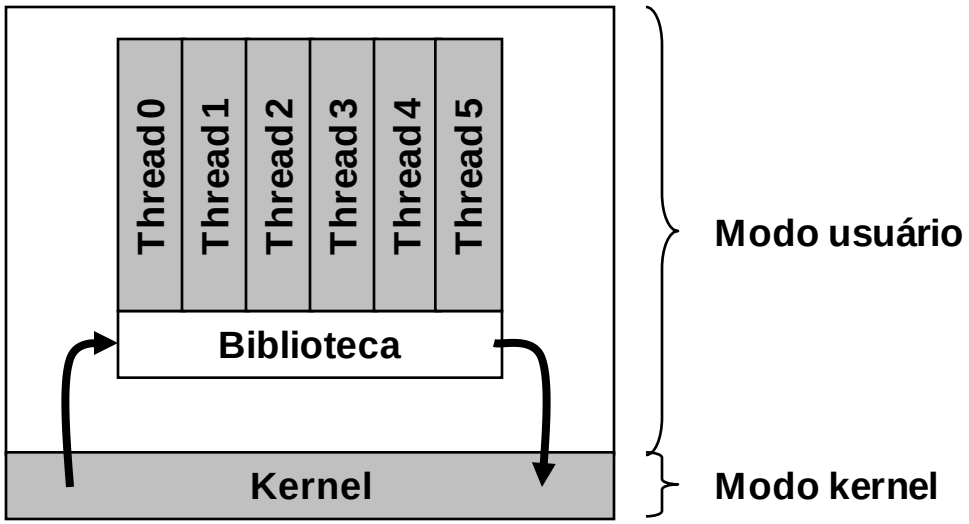


Processamento Paralelo – Conceito de Thread

- Scheduler Activations
  - O núcleo do sistema troca informações com a biblioteca de threads utilizando uma estrutura de dados chamada *scheduler activations*.
  - O objetivo é evitar as mudanças de modos de acesso desnecessárias (usuário-kernel-usuário).
  - Quando um thread utiliza uma rotina bloqueante, o kernel informa a biblioteca em modo usuário sobre o evento para que seja selecionado outro thread para execução.
    - » Trabalho colaborativo entre biblioteca e kernel.

Processamento Paralelo – Conceito de Thread

- Scheduler Activations

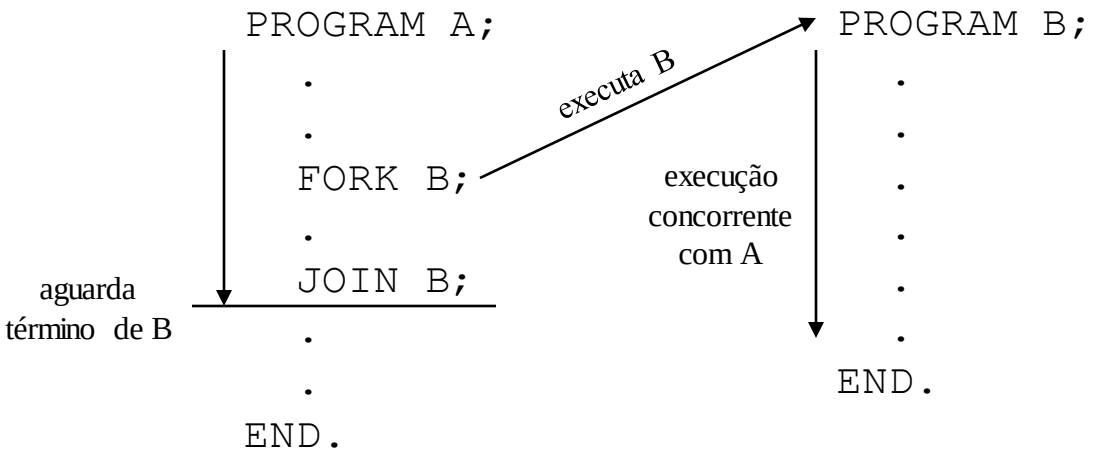


Processamento Paralelo – Conceito de Thread

- Formas de trabalho colaborativo
  - Manager/Worker
    - » Uma thread principal, o gerente, delega trabalho para outras threads, os trabalhadores.
    - » O gerente trata das entradas e divide o trabalho entre as outras threads, que podem ser grupos de threads estáticas ou grupo de threads dinâmicas.
  - Pipeline
    - » Uma tarefa é quebrada em uma série de suboperações que são tratadas em série, e não de forma concorrente, por diferentes threads.
  - Peer
    - » Semelhante à forma manager/worker, mas após a criação das outras threads, o gerente também participa do trabalho.

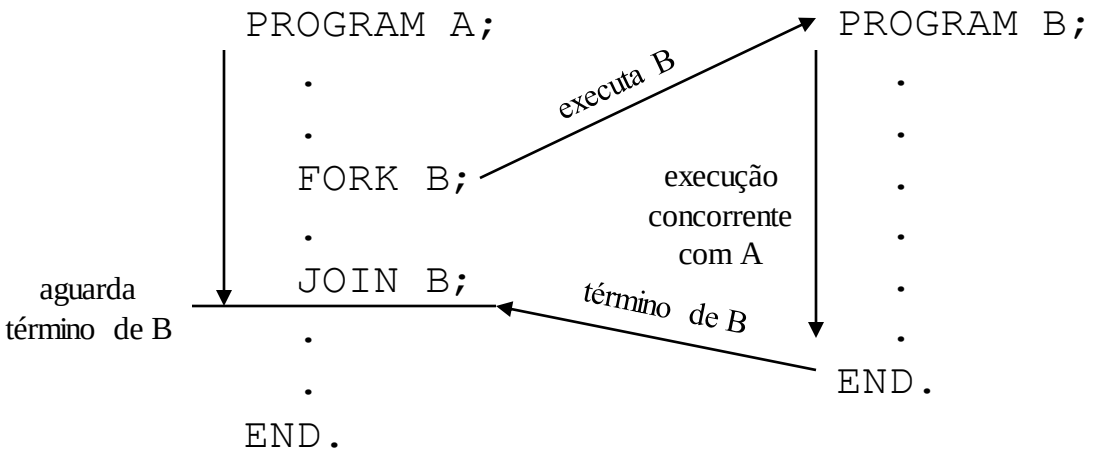
Processamento Paralelo – Concorrência e Sincronização

- Concorrência e Sincronização
  - FORK e JOIN



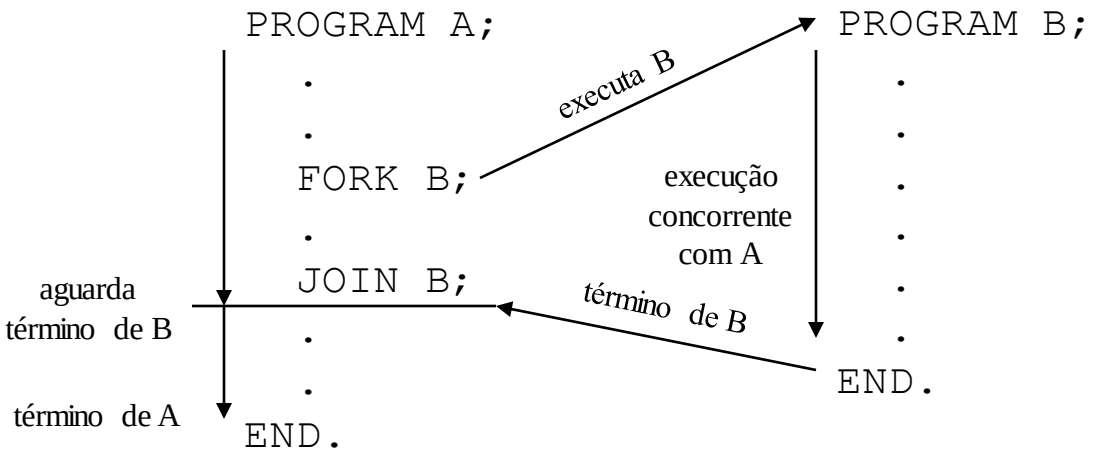
Processamento Paralelo – Concorrência e Sincronização

- Concorrência e Sincronização
  - FORK e JOIN



Processamento Paralelo – Concorrência e Sincronização

- Concorrência e Sincronização
  - FORK e JOIN



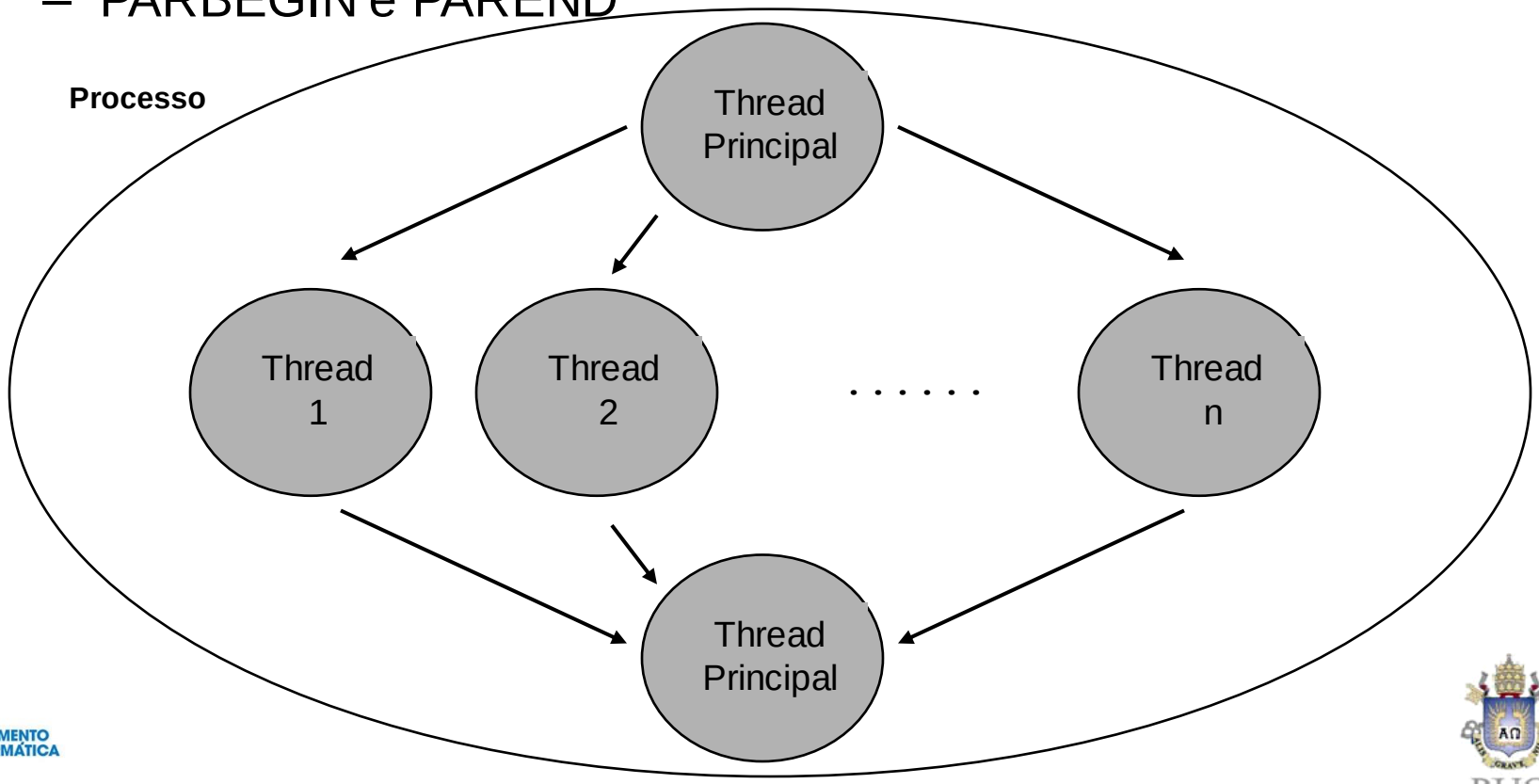
Processamento Paralelo – Concorrência e Sincronização

- Concorrência e Sincronização
  - PARBEGIN e PAREND

```
PROGRAM A;
BEGIN
  PARBEGIN
    Procedimento_1;
    Procedimento_2;
    .
    .
    Procedimento_n;
  PAREND;
END.
```

Processamento Paralelo – Concorrência e Sincronização

- Concorrência e Sincronização
  - PARBEGIN e PAREND



Processamento Paralelo – POSIX Thread

- Biblioteca POSIX – Portable Operating System Interface
  - Os sistemas operacionais disponibilizam *system calls* próprias para prover seus serviços, o que dificulta a portabilidade de programas.
  - Para garantir a portabilidade, o IEEE definiu uma interface de programação padronizada para interagir com sistemas operacionais.
    - » IEEE 1003.1 - IEEE Standard for Information Technology - Portable Operating System Interface (POSIX)
    - » https://standards.ieee.org/standard/1003_1-2008.html
  - O termo POSIX está associado a uma família de padrões relacionados. O IEEE 1003.1 é conhecido como POSIX.1.

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - A interface padrão do IEEE para o gerenciamento de threads está especificada no padrão IEEE 1003.1c (1995).
    - » IEEE 1003.1c-1995 - Standard for Information Technology--Portable Operating System Interface (POSIX(R)) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)
    - » https://standards.ieee.org/standard/1003_1c-1995.html
  - As implementações em conformidade com o padrão do IEEE são conhecidas como bibliotecas ***POSIX Threads*** ou ***Pthreads***.
  - A maior parte dos sistemas operacionais de hoje oferece a biblioteca Pthreads junto com a sua API proprietária.

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Os grupos de sub-rotinas da API podem ser organizadas em quatro grupos principais:
    - ***Thread Management:*** rotinas que trabalham diretamente com as threads (ex: criação, desacoplamento, junção, etc) e fazem o set/query de atributos das threads (joinable, scheduling, etc).
    - ***Mutexes:*** rotinas que lidam com a sincronização através de mutex (mutual exclusion) e permitem criar, destruir, bloquear e desbloquear mutexes, assim como fazer o set/modify de atributos associados aos mutexes.
    - ***Condition Variables:*** rotinas que fazem a comunicação entre threads que compartilham um mutex e incluem funções para criar, destruir, aguardar (wait) e sinalizar (signal) com base em valores de variáveis específicas, e fazer o set/query dos atributos de variáveis de condição.
    - ***Sincronização:*** rotinas que gerenciam bloqueios (locks) e barreiras (barriers) de operações de read/write.

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Convenção de nomes: prefixo ***pthread***_ em todas as funções.

Prefixo	Grupo Funcional
pthread_	Manipulação de threads e rotinas gerais.
pthread_attr_	Manipulação de atributos de threads.
pthread_mutex_	Manipulação de mutex.
pthread_mutex_attr_	Manipulação dos atributos do mutex.
pthread_cond_	Manipulação de variáveis de variáveis de condição
pthread_condattr_	Manipulação dos atributos de variáveis de condição.
pthread_key_	Manipulação de chaves de dados específicos de threads.
pthread_rwlock_	Manipulação de bloqueio de read/write.
pthread_barrier_	Manipulação de barreiras de sincronização.

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Cabeçalho com as primitivas da Pthread: ***pthread.h***
  - Diretiva de compilação do gcc: ***-pthread***
  - Exemplo:

```
gcc -Wall -pthread -o exemplo exemplo.c
```

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Verificando o limite máximo de processos do sistema:

Limite Máximo

```
$ ulimit -Hu
63144

$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 63144
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 4096
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

Limite Atual

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Configurando o limite máximo de processos do sistema:

```
$ ulimit -u 63144

$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 63144
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 63144
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Criando threads:

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

  - ***pthread_t *thread:*** ponteiro para o objeto thread que conterá o thread_id.
  - ***pthread_attr_t *attr:*** atributos que devem ser aplicados na thread.
  - ***void *(*start_routine)(void *):*** a função que a thread deve executar.
  - ***void *arg:*** argumentos que serão passados para a função que a thread irá executar.
  - ***retorno:*** em caso de sucesso, retorna 0 e, caso contrário, retorna um número de erro para indicar o erro.

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Terminando threads:

```
void pthread_exit(void *value_ptr);
```

  - ***value_ptr:*** ponteiro para o objeto (valor de retorno opcional) que será passado para a thread mestre que disparou essa thread, caso seja usado pthread_join na mestre.

  - Observações:
    - » Essa função não fecha os arquivos abertos por uma thread e os arquivos abertos dentro da thread permanecerão abertos quando a thread terminar.
    - » A função main() deve chamar essa função para garantir que as threads serão executadas até o final, caso contrário, o processo é encerrado e as threads serão destruídas sem terminarem o seu trabalho.

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: hello_pthread.c

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

#define NUM_THREADS 5

void *printHello(void *threadid) {
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: hello_pthread.c

```
int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;
    for(long t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, printHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    /* Last thing that main() should do */
    pthread_exit(NULL);
}
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: hello_pthread.c

```
$ gcc -Wall -pthread -o hello_pthread hello_pthread.c

$ ./hello_pthread
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
Hello World! It's me, thread #1!
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #0!
Hello World! It's me, thread #2!
Hello World! It's me, thread #4!
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Passando argumentos estruturados para threads:

```
struct thread_data {
   long thread_id;
   ……
};
```

  - Observação:
    - » Os argumentos estruturados devem ser preparados na thread mestre e devem ser passados por referência para as threads.
    - » Embora não obrigatório, é comum ter um campo na estrutura para fornecer o ID da thread.

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: args_pthread.c

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5
struct thread_data {
    long thread_id;
    int offset;
    char *message;
};
void *printHello(void *threadarg) {
    struct thread_data *my_data;
    my_data = (struct thread_data*) threadarg;
    printf("Thread #%ld: %s\n", my_data->thread_id, my_data->message+my_data->offset);
    pthread_exit(NULL);
}
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: args_pthread.c

```
int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;
    struct thread_data thread_data_array[NUM_THREADS];
    char message[]="ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    for(long t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\n", t);
        thread_data_array[t].thread_id = t;
        thread_data_array[t].offset = t * 4;
        thread_data_array[t].message = message;
----------------------------------------------------------
    }

    /* Last thing that main() should do */
    pthread_exit(NULL);
}
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: args_pthread.c

```
int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int rc;
    struct thread_data thread_data_array[NUM_THREADS];
    char message[]="ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    for(long t=0; t<NUM_THREADS; t++){
-------------------------------------------------------
        rc = pthread_create(&threads[t], NULL, printHello, (void *)&thread_data_array[t]
        if (rc) {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    /* Last thing that main() should do */
    pthread_exit(NULL);
}
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: args_pthread.c

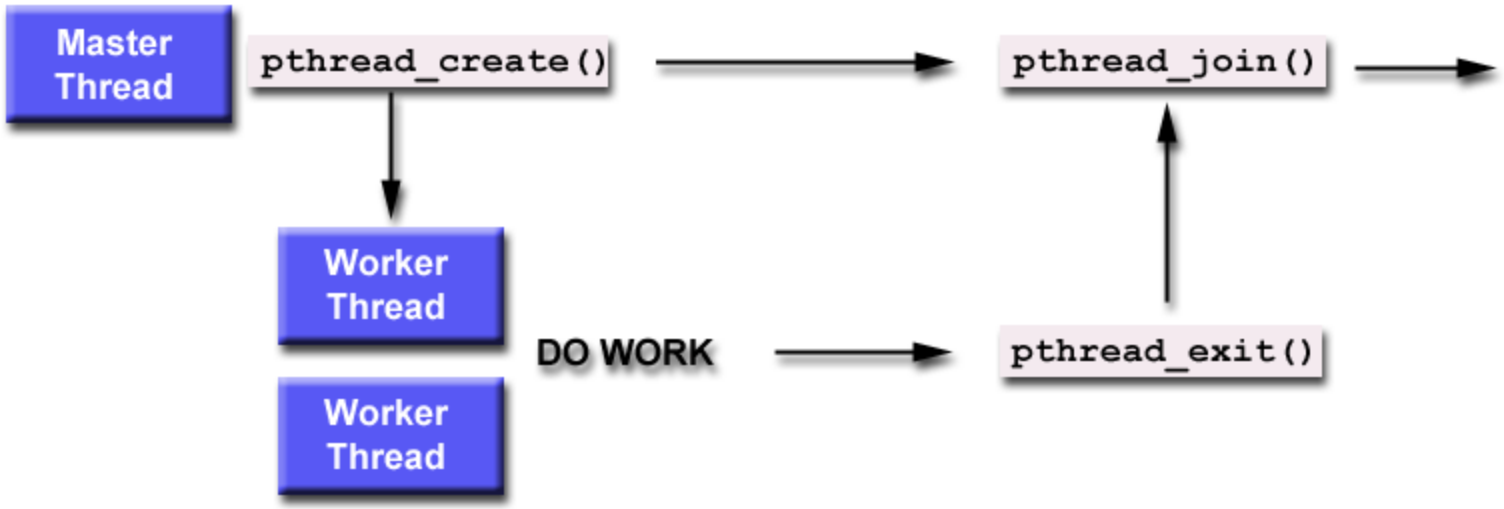
```
$ gcc -Wall -pthread -o args_pthread args_pthread.c

$ ./args_pthread
In main: creating thread 0
In main: creating thread 1
Thread #0: ABCDEFGHIJKLMNOPQRSTUVWXYZ
In main: creating thread 2
Thread #1: EFGHIJKLMNOPQRSTUVWXYZ
In main: creating thread 3
Thread #2: IJKLMNOPQRSTUVWXYZ
In main: creating thread 4
Thread #3: MNOPQRSTUVWXYZ
Thread #4: QRSTUVWXYZ
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Juntando Threads (Sincronização):



Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Juntando Threads (Sincronização):

```
int pthread_join(pthread_t thread, void **value_ptr);
```

  - ***pthread_t thread:*** objeto thread que contém o thread_id da thread alvo.
  - ****value_ptr:*** ponteiro para o objeto (valor de saída) que será passado pela thread alvo disparada pela mestre quando a função de saída for executada.
  - ***retorno:*** em caso de sucesso, retorna 0 e, caso contrário, retorna um número do erro para indicar o erro.

  - Observação:
    - » Essa função suspende a execução da thread que fez a chamada até que a thread alvo termine, à menos que a thread alvo já tenha terminado.

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Inicializando e destruindo atributos de threads:

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

  - ***pthread_attr_t *attr:*** inicializa o objeto attributo com os valores defaults que serão usados durante a criação da thread. Pode ser usado em múltiplas chamadas de *pthread_create()*;
  - ***retorno:*** em caso de sucesso, retorna o valor zero e, caso contrário, retorna um número de erro para indicar o erro.

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Habilitando/Consultando o estado de junção/disjunção de threads:

```
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                int *detachstate);
```

  - ***pthread_attr_t *attr:*** objeto attributo que deve ser modificado/consultado para um estado de junção/disjunção específico: PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE (default).
  - ***retorno:*** em caso de sucesso, retorna o valor zero e, caso contrário, retorna um número de erro para indicar o erro.

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: join_pthread.c

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS 4
void *busyWork(void *t) {
    long tid;
    double result=0.0;
    tid = (long)t;
    printf("Thread %ld starting...\n",tid);
    for (int i=0; i<100000000; i++) {
        result = result + sin(i) * tan(i);
    }
    printf("Thread %ld done. Result = %e\n",tid, result);
    pthread_exit((void*) t);
}
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://hpc-tutorials.llnl.gov/posix/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: join_pthread.c

```
int main (int argc, char *argv[]) {
    pthread_t thread[NUM_THREADS];
    pthread_attr_t attr;
    int rc;
    void *status;

    /* Initialize and set thread detached attribute */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    for(long t=0; t<NUM_THREADS; t++) {
        printf("Main: creating thread %ld\n", t);
        rc = pthread_create(&thread[t], &attr, busyWork, (void *)t);
        if (rc) {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: join_pthread.c

```
    /* Free attribute and wait for the other threads */
    pthread_attr_destroy(&attr);
    for(long t=0; t<NUM_THREADS; t++) {
        rc = pthread_join(thread[t], &status);
        if (rc) {
            printf("ERROR; return code from pthread_join() is %d\n", rc);
            exit(-1);
        }
        printf("Main: completed join with thread %ld having a status of %ld\n",
            t,(long)status);
    }
    printf("Main: program completed. Exiting.\n");
    pthread_exit(NULL);
}
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: join_pthread.c

```
$ gcc -Wall -pthread -o join_pthread join_pthread.c -lm

$ ./join_pthread
Main: creating thread 0
Main: creating thread 1
Thread 0 starting...
Main: creating thread 2
Thread 1 starting...
Main: creating thread 3
Thread 3 starting...
Thread 2 starting...
Thread 3 done. Result = 1.035132e+08
Thread 1 done. Result = 1.035132e+08
Thread 0 done. Result = 1.035132e+08
Main: completed join with thread 0 having a status of 0
Main: completed join with thread 1 having a status of 1
Thread 2 done. Result = 1.035132e+08
Main: completed join with thread 2 having a status of 2
Main: completed join with thread 3 having a status of 3
Main: program completed. Exiting.
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Gerenciando a stack das threads:

```
int pthread_attr_setstacksize(pthread_attr_t *attr,
                              size_t stacksize);
int pthread_attr_getstacksize(const pthread_attr_t *attr,
                              size_t *stacksize);
```

  - ***pthread_attr_t *attr:*** objeto attributo que deve ser modificado/consultado com um tamanho de stack específico para a thread que será criada.
  - ***retorno:*** em caso de sucesso, retorna o valor zero e, caso contrário, retorna um número de erro para indicar o erro.
  - Observação:
    - » O tamanho da stack está diretamente associado à quantidade de memória necessária para alocar as variáveis locais, os argumentos passados e o controle de execução da thread. **O valor default pode não ser suficiente.**

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: stack_pthread.c

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

#define NTHREADS 4

#define N 1024
#define MEGEXTRA 1024000

pthread_attr_t attr;
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: stack_pthread.c

```
void *dowork(void *threadid)
{
   double A[N][N];
   int i,j;
   long tid;
   size_t mystacksize;

   tid = (long)threadid;
   pthread_attr_getstacksize (&attr, &mystacksize);
   printf("Thread %ld: stack size = %li bytes \n", tid, mystacksize);

   for (i=0; i<N; i++)
     for (j=0; j<N; j++)
       A[i][j] = ((i*j)/3.452) + (N-i);

   pthread_exit(NULL);
}
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: stack_pthread.c

```
int main(int argc, char *argv[])
{
   pthread_t threads[NTHREADS];
   size_t stacksize;
   int rc;
   long t;

   pthread_attr_init(&attr);
   pthread_attr_getstacksize (&attr, &stacksize);
   printf("Default stack size = %li\n", stacksize);

   stacksize = sizeof(double)*N*N+MEGEXTRA;
   printf("Amount of stack needed per thread = %li\n",stacksize);

   pthread_attr_setstacksize (&attr, stacksize);
   printf("Creating threads with stack size = %li bytes\n",stacksize);
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: stack_pthread.c

```
    for(t=0; t<NTHREADS; t++){
      rc = pthread_create(&threads[t], &attr, dowork, (void *)t);
      if (rc){
        printf("ERROR; return code from pthread_create() is %d\n", rc);
        exit(-1);
      }
    }
    printf("Created %ld threads.\n", t);
    pthread_exit(NULL);
}
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: stack_pthread.c

```
$ gcc -pthread -o stack_pthread stack_pthread.c

$ ./stack_pthread
Default stack size = 8388608
Amount of stack needed per thread = 9412608
Creating threads with stack size = 9412608 bytes
Thread 0: stack size = 9412608 bytes
Thread 1: stack size = 9412608 bytes
Created 4 threads.
Thread 2: stack size = 9412608 bytes
Thread 3: stack size = 9412608 bytes
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Exercício 1
  - Implemente um programa com processamento paralelo de thread sem AVX e outro com AVX, para multiplicar os elementos de um array pelos elementos de outro array, posicionalmente. O resultado deve ser armazenado em um terceiro array.
  - O primeiro array (evens) deve ser inicializado com valores de ponto-flutuante igual a 2.0f em todos os elementos.
  - O segundo array (odds) deve ser inicializado com valores de ponto-flutuante igual a 5.0f em todos os elementos.
  - O número de threads e o tamanho dos arrays (múltiplo do número de threads vezes 8) devem ser fornecidos na linha de comando de execução do programa.
  - Após a multiplicação, um loop sequencial (sem AVX) deve validar o resultado de cada elemento do terceiro array.

Processamento Paralelo – POSIX Thread

- Exercício 1
  - Para fazer a comparação de desempenho entre as versões do programa sem AVX e com AVX, utilize a implementação do cronômetro (*timer.c*) acrescentando o arquivo de cabeçalho *timer.h* no seu programa fonte e compilando seu programa junto com o arquivo fonte *timer.c*.

gcc –std=c11 –pthread –o array_mult_pthread array_mult_pthread.c timer.c

gcc –std=c11 –pthread –mavx –o array_mult_pthread_avx array_mult_pthread_avx.c timer.c

Onde:

» *array_mult_pthread* e *array_mult_pthread_avx* são os nomes dos arquivos executáveis
» *array_mult_pthread.c* e *array_mult_pthread_avx.c* são os nomes dos programas-fontes.
» *timer.c* é o nome do arquivo do programa fonte do cronômetro.

Processamento Paralelo – POSIX Thread

- Exercício 1
  - Esqueleto dos programas *array_mult_pthread.c* e *array_mult_pthread_avx.c*.

```
#include <pthread.h>
……..……
#include "timer.h"
……
/* Thread to multiply arrays */
void *mult_arrays(void *threadarg) {
……
/* Thread to initialize arrays */
void *init_arrays(void *threadarg) {
……
int main(int argc, char *argv[]) {
……
/* Check and convert arguments */
……
/* Allocate three arrays */
……
/* Define auxiliary variables to work with threads */
……
```

Processamento Paralelo – POSIX Thread

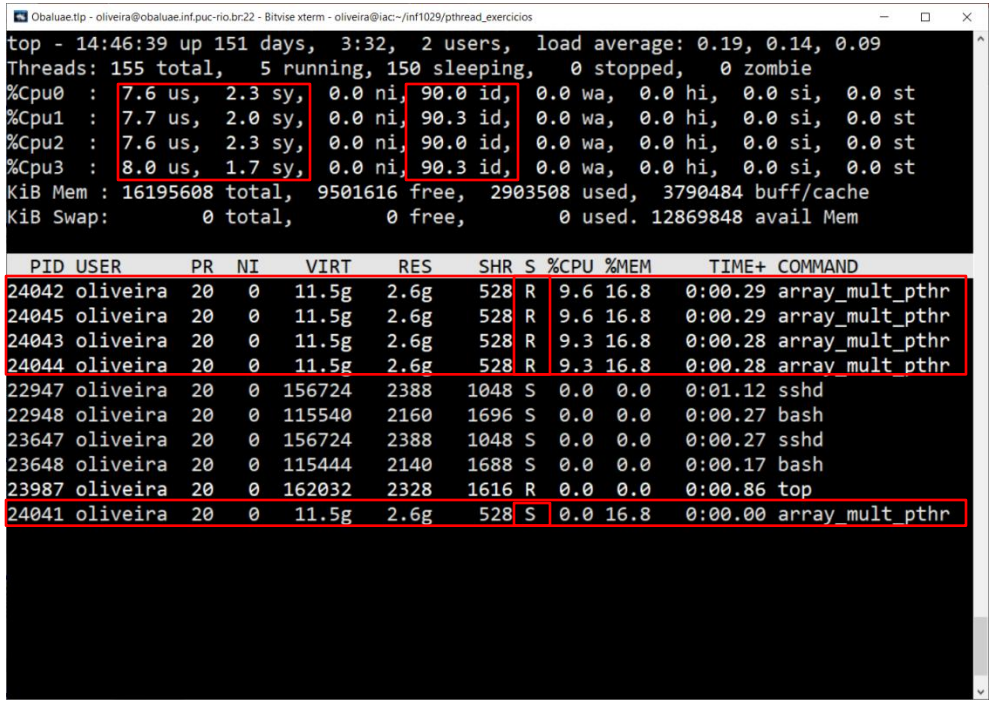
- Exercício 1
  - Esqueleto dos programas *array_mult_pthread.c* e *array_mult_pthread_avx.c*.

```
/* Initialize and set thread detached attribute */
.....
/* Create threads to initialize arrays */
.....
/* Free attribute and wait for the other threads */
.....
/* Initialize and set thread detached attribute */
.....
/* Create threads to calculate product of arrays */
.....
/* Free attribute and wait for the other threads */
.....
/* Check for errors (all values should be 10.0f) */
.....
/* Free memory */
.....
```

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Monitoramento das threads de um processo no Linux:

$ top –H –u user



Processamento Paralelo – POSIX Thread

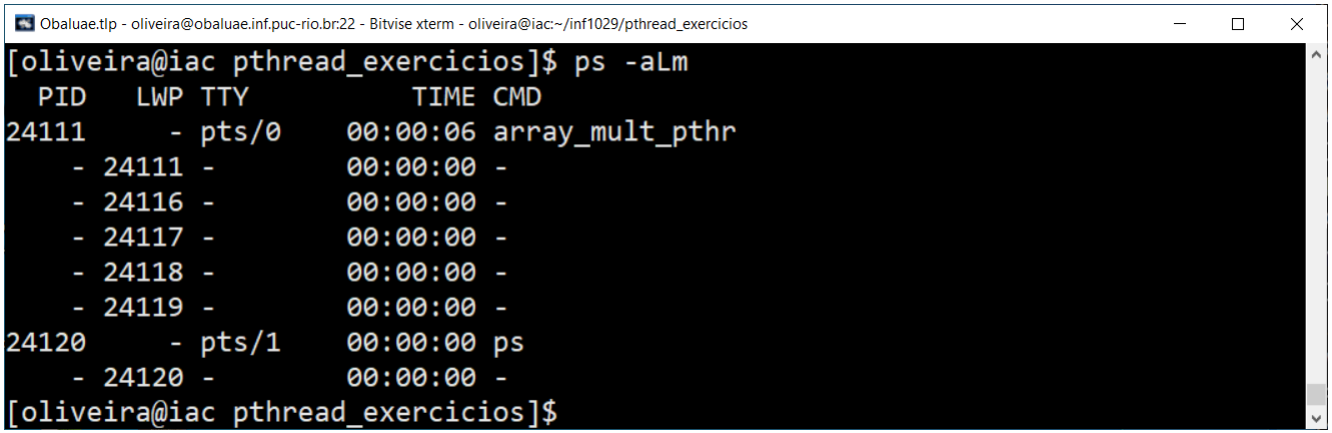
- Biblioteca POSIX Thread (Pthread)
  - Monitoramento das threads de um processo no Linux:

$ ps -aLf

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Monitoramento das threads de um processo no Linux:

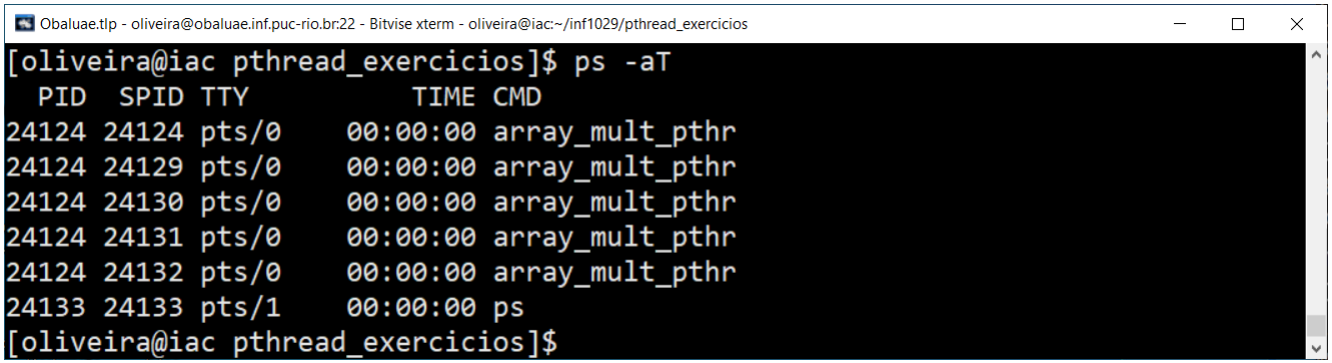
$ ps -aLm



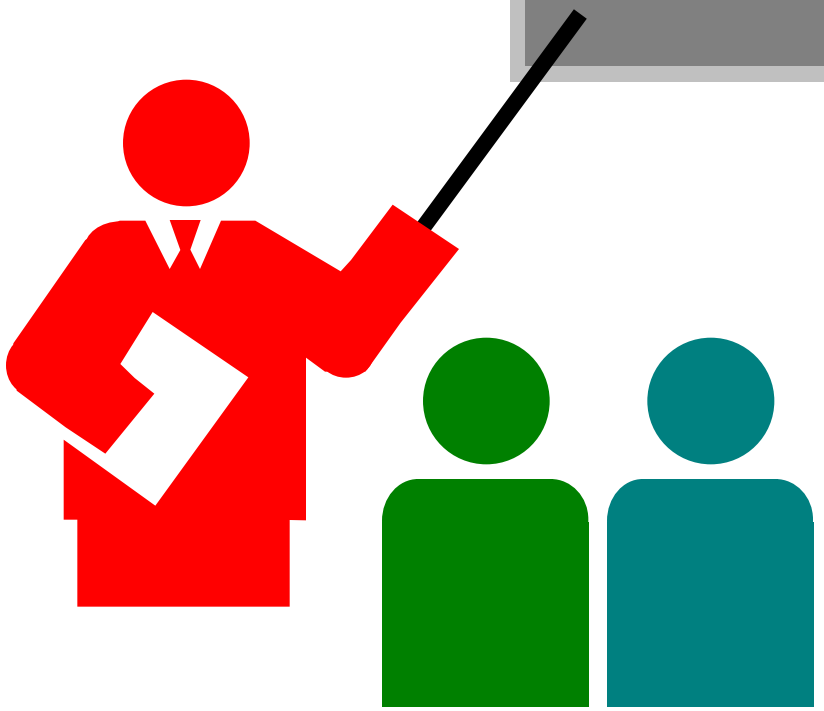
Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Monitoramento das threads de um processo no Linux:

$ ps -aT



Problemas de Concorrência



Processamento Paralelo – Concorrência e Sincronização

- Problemas de Compartilhamento de Recursos
  - Compartilhamento de uma variável em memória (Exemplo)
    - » Duas threads (A e B) compartilham a variável X.
    - » Variável X possui, inicialmente, o valor 2.
    - » A soma 1 a variável X.
    - » B subtrai 1 da variável X.

Thread A	Thread B
`X := X + 1;`	`X := X - 1;`
`LOAD x,Ra`	`LOAD x,Rb`
`ADD 1,Ra`	`SUB 1,Rb`
`STORE Ra,X`	`STORE Rb,X`

Processamento Paralelo – Concorrência e Sincronização

- Problemas de Compartilhamento de Recursos
  - Compartilhamento de uma variável em memória (Exemplo)
    - » A inicia a execução carregando o valor corrente de X, e soma 1.
    - » A é interrompido antes de armazenar o novo valor de X.
    - » B inicia a execução carregando o valor corrente de X, e subtrai 1.
    - » A volta a ser processado e armazena o novo valor de X.
    - » B volta a ser processado e armazena o novo valor de X.

Thread	Comando	X	Ra	Rb
A	LOAD X,Ra	2	2	*

Processamento Paralelo – Concorrência e Sincronização

- Problemas de Compartilhamento de Recursos
  - Compartilhamento de uma variável em memória (Exemplo)
    - » A inicia a execução carregando o valor corrente de X, e soma 1.
    - » A é interrompido antes de armazenar o novo valor de X.
    - » B inicia a execução carregando o valor corrente de X, e subtrai 1.
    - » A volta a ser processado e armazena o novo valor de X.
    - » B volta a ser processado e armazena o novo valor de X.

Thread	Comando	X	Ra	Rb
A	LOAD X,Ra	2	2	*
A	ADD 1,Ra	2	3	*

Processamento Paralelo – Concorrência e Sincronização

- Problemas de Compartilhamento de Recursos
  - Compartilhamento de uma variável em memória (Exemplo)
    - » A inicia a execução carregando o valor corrente de X, e soma 1.
    - » A é interrompido antes de armazenar o novo valor de X.
    - » B inicia a execução carregando o valor corrente de X, e subtrai 1.
    - » A volta a ser processado e armazena o novo valor de X.
    - » B volta a ser processado e armazena o novo valor de X.

Thread	Comando	X	Ra	Rb
A	LOAD X,Ra	2	2	*
A	ADD 1,Ra	2	3	*
B	LOAD X,Rb	2	*	2

Processamento Paralelo – Concorrência e Sincronização

- Problemas de Compartilhamento de Recursos
  - Compartilhamento de uma variável em memória (Exemplo)
    - » A inicia a execução carregando o valor corrente de X, e soma 1.
    - » A é interrompido antes de armazenar o novo valor de X.
    - » B inicia a execução carregando o valor corrente de X, e subtrai 1.
    - » A volta a ser processado e armazena o novo valor de X.
    - » B volta a ser processado e armazena o novo valor de X.

Thread	Comando	X	Ra	Rb
A	LOAD X,Ra	2	2	*
A	ADD 1,Ra	2	3	*
B	LOAD X,Rb	2	*	2
B	SUB 1,Rb	2	*	1

Processamento Paralelo – Concorrência e Sincronização

- Problemas de Compartilhamento de Recursos
  - Compartilhamento de uma variável em memória (Exemplo)
    - » A inicia a execução carregando o valor corrente de X, e soma 1.
    - » A é interrompido antes de armazenar o novo valor de X.
    - » B inicia a execução carregando o valor corrente de X, e subtrai 1.
    - » A volta a ser processado e armazena o novo valor de X.
    - » B volta a ser processado e armazena o novo valor de X.

Thread	Comando	X	Ra	Rb
A	LOAD X,Ra	2	2	*
A	ADD 1,Ra	2	3	*
B	LOAD X,Rb	2	*	2
B	SUB 1,Rb	2	*	1
A	STORE Ra,X	3	3	1

Processamento Paralelo – Concorrência e Sincronização

- Problemas de Compartilhamento de Recursos
  - Compartilhamento de uma variável em memória (Exemplo)
    - » A inicia a execução carregando o valor corrente de X, e soma 1.
    - » A é interrompido antes de armazenar o novo valor de X.
    - » B inicia a execução carregando o valor corrente de X, e subtrai 1.
    - » A volta a ser processado e armazena o novo valor de X.
    - » B volta a ser processado e armazena o novo valor de X.

Thread	Comando	X	Ra	Rb
A	LOAD X,Ra	2	2	*
A	ADD 1,Ra	2	3	*
B	LOAD X,Rb	2	*	2
B	SUB 1,Rb	2	*	1
A	STORE Ra,X	3	3	1
B	STORE Rb,X	1	*	1

Processamento Paralelo – Concorrência e Sincronização

- Exclusão Mútua
  - Impede que dois ou mais processos/threads acessem o mesmo recurso no mesmo instante.
  - Enquanto um processo/thread estiver acessando determinado recurso, todos os outros que queiram o mesmo recurso deverão esperar até que o primeiro termine o acesso.

Processamento Paralelo – Concorrência e Sincronização

- Exclusão Mútua
  - Região Crítica
    - » Parte do código do programa onde ocorre acesso ao recurso compartilhado.
  - Mecanismo
    - » Processo/Thread deve executar um protocolo de entrada antes de executar a região crítica.
    - » Processo/Thread deve executar um protocolo de saída após a execução da região crítica.

Processamento Paralelo – Concorrência e Sincronização

- Exclusão Mútua

```
BEGIN
  .
  Entra_Regiao_Critica; (* Protocolo de Entrada *)
  Regiao_Critica;
  Sai_Regiao_Critica; (* Protocolo de Saida *)
  .
END
```

Processamento Paralelo – Concorrência e Sincronização

- Starvation ou Indefinetely Postponed
  - Situação onde um processo/thread nunca consegue executar sua região crítica e, consequentemente, acessar o recurso compartilhado.
  - Exemplo
    - » Sempre que um recurso é liberado, o sistema sempre escolhe o processo/thread mais prioritário.
    - » Processo de baixa prioridade pode esperar indefinidamente pelo recurso.
  - Solução
    - » O esquema FIFO, aplicado na espera pela liberação do recurso, elimina o problema do starvation.

Processamento Paralelo – Concorrência e Sincronização

- Espera Ocupada (Busy Wait)
  - Ocorre quando o processo/thread não ganha acesso a sua região crítica e fica em looping, testando uma condição, até ganhar o acesso.
  - Processo/thread impedido de acessar o recurso permanece consumindo tempo do processador desnecessariamente.
  - Solução
    - » O sistema operacional deve colocar o processo/thread no estado de espera quando não se consegue acesso ao recurso solicitado.
    - » O processo/thread permanece em espera até que outro processo/thread notifique o sistema operacional sobre a liberação do recurso.

Processamento Paralelo – Concorrência e Sincronização

- Sincronização Condicional
  - Ocorre quando um recurso compartilhado não se encontra pronto para ser utilizado pelos processos/threads devido a uma condição específica.
    - » Ex: Operação de leitura em um buffer vazio.
  - Processo/thread que deseja utilizar o recurso deve ser colocado no estado de espera até o recurso ficar pronto.
  - Exemplo
    - » Qualquer operação onde existam processos/threads gerando informações (processos/threads produtores) utilizadas por outros processos/threads (processos/threads consumidores).

Processamento Paralelo – Concorrência e Sincronização

- Sincronização Condicional

```
PROCEDURE Produtor;
BEGIN
  REPEAT
    Produz_Dado(Dado);
    WHILE (Cont = TamBuf) DO ;
    Grava_Buffer(Dado, Cont);
  UNTIL False;
END;
```

```
PROCEDURE Consumidor;
BEGIN
  REPEAT
    WHILE (Cont = 0) DO ;
    Le_Buffer(Dado, Cont);
    Consome_Dado(Dado)
  UNTIL False;
END;
```

```
Dado = Dado a ser lido ou gravado no buffer.
Cont = Contador de posições ocupadas no buffer.
TamBuf = Limite máximo de posições do buffer.
```

Processamento Paralelo – Concorrência e Sincronização

- Fatores Fundamentais
  - Os processos/threads não podem executar suas próprias regiões críticas simultaneamente.
  - O número de processos/threads e o tempo de execução dos processos/threads concorrentes devem ser irrelevantes.
  - Um processo/thread, fora de sua região crítica, não pode impedir outros processos/threads de executarem suas próprias regiões críticas.
  - Um processo/thread não pode esperar indefinidamente para entrar em sua região crítica.

Processamento Paralelo – Concorrência e Sincronização

- Semáforos (Dijkstra, 1965)
  - Variável inteira, não negativa, que só pode ser manipulada por duas instruções:
    - » DOWN (também chamada instrução P)
      - Executado por um processo/thread antes de entrar na região crítica.
      - Se o semáforo não for 0, este é decrementado de 1, e o processo/thread que solicitou a operação pode executar a região crítica.
      - Se o semáforo é 0, o processo/thread que solicitou a operação fica no estado de espera, em uma fila associada ao semáforo.
    - » UP (também chamada instrução V)
      - Executado por um processo/thread ao sair da região crítica.
      - Se existem processos/threads esperando na fila associada ao semáforo, um deles é escolhido e muda seu estado para pronto.
      - Caso contrário, o semáforo é incrementado de 1.

Processamento Paralelo – Concorrência e Sincronização

- Semáforos
  - As operações DOWN e UP são indivisíveis
  - Normalmente são implementadas como *system calls*
  - Uso de instruções atômicas (indivisíveis) (ex.: LOCK XADD)

```
PROCEDURE Down (VAR S: Semáforo);
BEGIN
  IF (S = 0) THEN
    Coloca_Processo_Na_Fila_De_Espera
  ELSE
    S := S - 1;
END;
```

```
PROCEDURE Up (VAR S: Semáforo);
BEGIN
  IF (Tem_Processo_Esperando) THEN
    Retira_Da_Fila_De_Espera
  ELSE
    S := S + 1;
END;
```

Processamento Paralelo – Concorrência e Sincronização

- Semáforos *Mutexes* ou *Binários*
  - Aplicados a exclusão mútua (mutual exclusion).
  - Assumem apenas os valores 0 e 1.

```
PROCEDURE Thread_A;
BEGIN
  REPEAT
    DOWN(s);
    Regiao_Critica_A;
    UP(s);
  UNTIL False;
END;
```

```
PROCEDURE Thread_B;
BEGIN
  REPEAT
    DOWN(s);
    Regiao_Critica_B;
    UP(s);
  UNTIL False;
END;
```

Processamento Paralelo – Concorrência e Sincronização

- Semáforos *Mutexes* ou *Binários*
  - Exemplo (Semáforo s inicializado com 1)

Thread_A	**Thread_B**	**S**	**Pendente**
REPEAT		1	*
	REPEAT	1	*
DOWN(s)		0	*
	DOWN(s)	0	Thread_B
Regiao_Critica_A		0	Thread_B
UP(s)		0	*
	Regiao_Critica_B	0	*
UNTIL			
	UP(s)	1	*

Processamento Paralelo – Concorrência e Sincronização

- Semáforos para Sincronização de Processos/Threads Produtores e Consumidores
  - Utilizando três semáforos
    - » Semáforo Mutex permite a exclusão mútua na execução das regiões críticas Grava_Dado e Le_Dado.
    - » Semáforos Vazio indica se há posições vazias no buffer para serem gravadas:
      - Vazio = 0, buffer está cheio (produtor deve aguardar).
    - » Semáforo Cheio indica se há posições ocupadas a serem lidas:
      - Cheio = 0, buffer está vazio (consumidor deve aguardar).
  - Observação
    - » Vazio e Cheio são chamados de *Semáforos Contadores*.

Processamento Paralelo – Concorrência e Sincronização

- Semáforos para Sincronização de Processos/Threads Produtores e Consumidores

```
PROGRAM Produtor_Consumidor;
  CONST TamBuf = 2
  TYPE Tipo_Dado = (* qualquer tipo *)
  VAR
     Vazio  : Semaforo := TamBuf;
     Cheio  : Semaforo := 0;
     Mutex  : Semaforo := 1;
     Buffer : ARRAY [1..TamBuf] OF Tipo_Dado;
     Dado_1 : Tipo_Dado;
     Dado_2 : Tipo_Dado;
```

Processamento Paralelo – Concorrência e Sincronização

- Semáforos para Sincronização de Processos/Threads Produtores e Consumidores

```
PROCEDURE Produtor;
BEGIN
  REPEAT
    Produz_Dado(Dado_1);
    Down(Vazio);
    Down(Mutex);
    Grava_Dado(Dado_1, Buffer);
    UP(Mutex);
    UP(Cheio);
  UNTIL False;
END;
```

```
PROCEDURE Consumidor;
BEGIN
  REPEAT
    Down(Cheio);
    Down(Mutex);
    Le_Dado(Dado_2, Buffer);
    UP(Mutex);
    UP(Vazio);
    Consome_Dado(Dado_2);
  UNTIL False;
END;
```

Processamento Paralelo – Concorrência e Sincronização

- Semáforos para Sincronização de Processos/Threads Produtores e Consumidores
  - Exemplo (thread consumidor é o primeiro a ser executado).

Produtor	Consumidor	Vazio	Cheio	Mutex	Pendente
*	*	2	0	1	*
*	DOWN(Cheio)	2	0	1	Consumidor
DOWN(Vazio)	*	1	0	1	Consumidor
Down(Mutex)	*	1	0	0	Consumidor
Grava_Dado	*	1	0	0	Consumidor
UP(Mutex)	*	1	0	1	Consumidor
UP(Cheio)	*	1	0	1	*
DOWN(Vazio)	DOWN(Mutex)	0	0	0	*
DOWN(Mutex)	Le_Dado	0	0	0	Produtor
*	UP(Mutex)	0	0	0	*
Grava_Dado	UP(Vazio)	1	0	0	*

Processamento Paralelo – Concorrência e Sincronização

- Semáforos para Sincronização de Processos Produtores e Consumidores
  - Exemplo (continuação – Le_Dado agora demora mais)

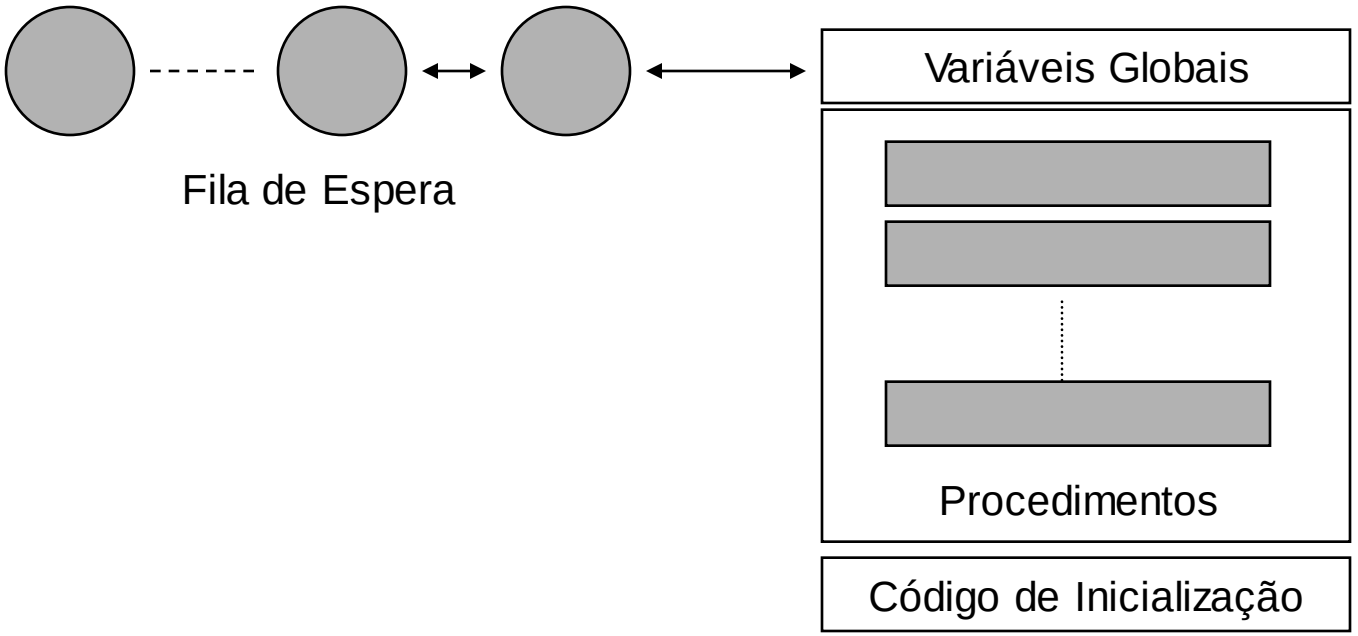
Produtor	Consumidor	Vazio	Cheio	Mutex	Pendente
UP(Mutex)	DOWN(Cheio)	1	0	1	Consumidor
UP(Cheio)	*	1	0	1	*
DOWN(Vazio)	DOWN(Mutex)	0	0	0	*
DOWN(Mutex)	Le_Dado	0	0	0	Produtor
*	Le_Dado	0	0	0	Produtor
*	UP(Mutex)	0	0	0	*
Grava_Dado	UP(Vazio)	1	0	0	*

Processamento Paralelo – Concorrência e Sincronização

- Monitores
  - Mecanismos de sincronização de alto-nível que facilitam o desenvolvimento e a correção de programas concorrentes.
  - Conjunto de procedimentos, variáveis e estruturas de dados definidos dentro de um módulo.
  - Implementação automática de exclusão mútua entre seus procedimentos.
  - Toda vez que um destes procedimentos é chamado por algum processo, é verificado se já existe algum processo executando algum procedimento do monitor.
  - Toda exclusão mútua é realizada pelo compilador, não mais pelo programador.

Processamento Paralelo – Concorrência e Sincronização

- Monitores
  - Estrutura



Processamento Paralelo – Concorrência e Sincronização

- Monitor para Exclusão Mútua
  - Exemplo

```
PROGRAM Exemplo;
  MONITOR Regiao_Critica;
    VAR X : INTEGER;
    PROCEDURE Soma;
    BEGIN
      X := X + 1;
    END;
    PROCEDURE Subtrai;
    BEGIN
      X := X - 1;
    END;
  BEGIN
    X := 0;
  END;
BEGIN
  PARBEGIN
    Regiao_Critica.Soma;
    Regiao_Critica.Subtrai;
  PAREND;
END.
```

Processamento Paralelo – Concorrência e Sincronização

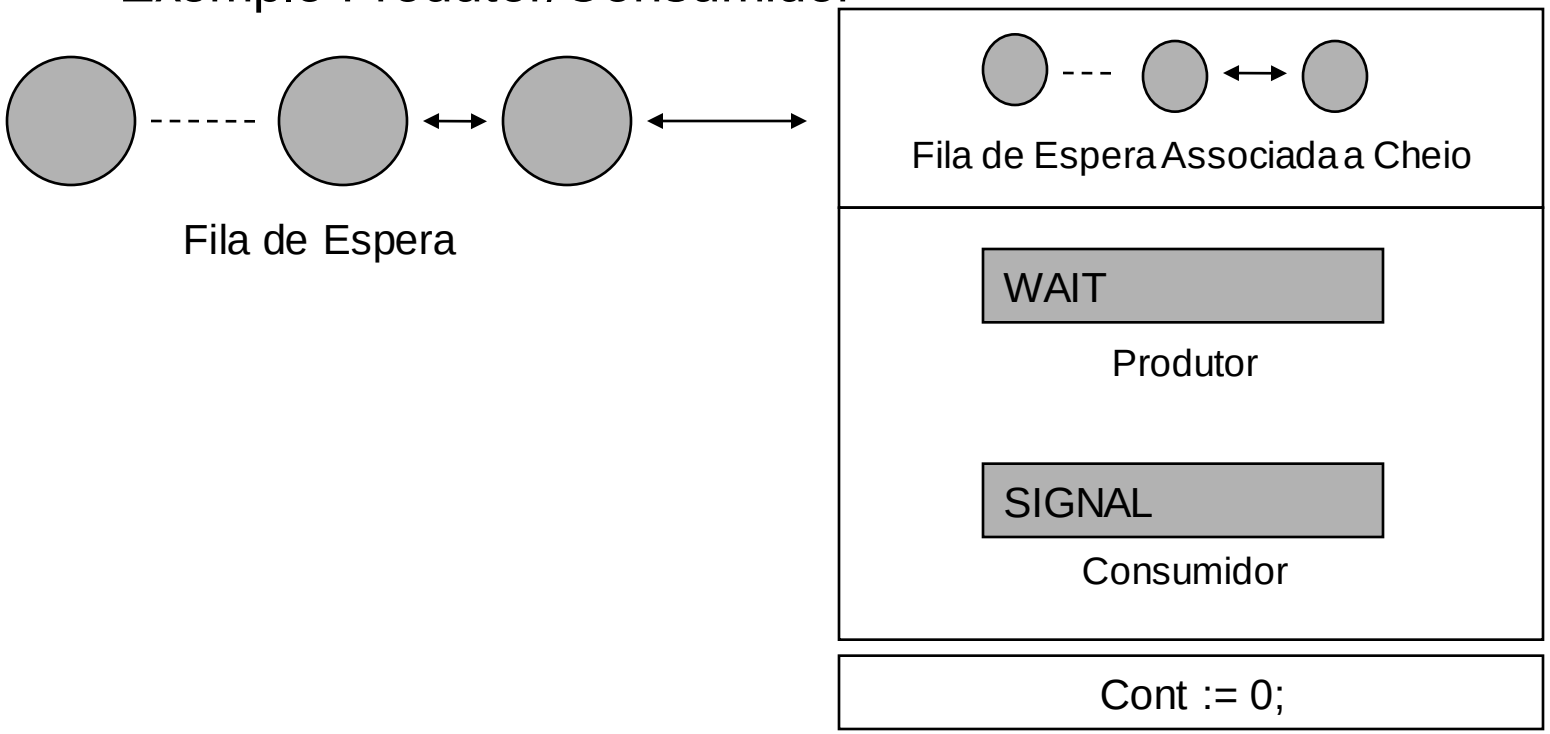
- Monitor para Exclusão Mútua
  - Exemplo (Observações)
    - » A inicialização da variável X só é feita uma vez, no momento da ativação do Monitor Regiao_Critica.
    - » É garantida a execução mutuamente exclusiva dos procedimentos Soma e Subtrai.

Processamento Paralelo – Concorrência e Sincronização

- Monitor e Sincronização Condicional
  - Variável de Condição
    - » Estrutura de dados do tipo fila onde os processos esperam por algum evento.
  - Instruções Auxiliares
    - » WAIT
      - Bloqueia um processo em uma fila de espera associada a uma variável de condição, se a condição impede a execução do processo.
    - » SIGNAL
      - Libera um processo de uma fila de espera associada a uma variável de condição.

Processamento Paralelo – Concorrência e Sincronização

- Monitor e Sincronização Condicional
  - Exemplo Produtor/Consumidor



Processamento Paralelo – Concorrência e Sincronização

- Monitor e Sincronização Condicional
  - Exemplo Produtor/Consumidor

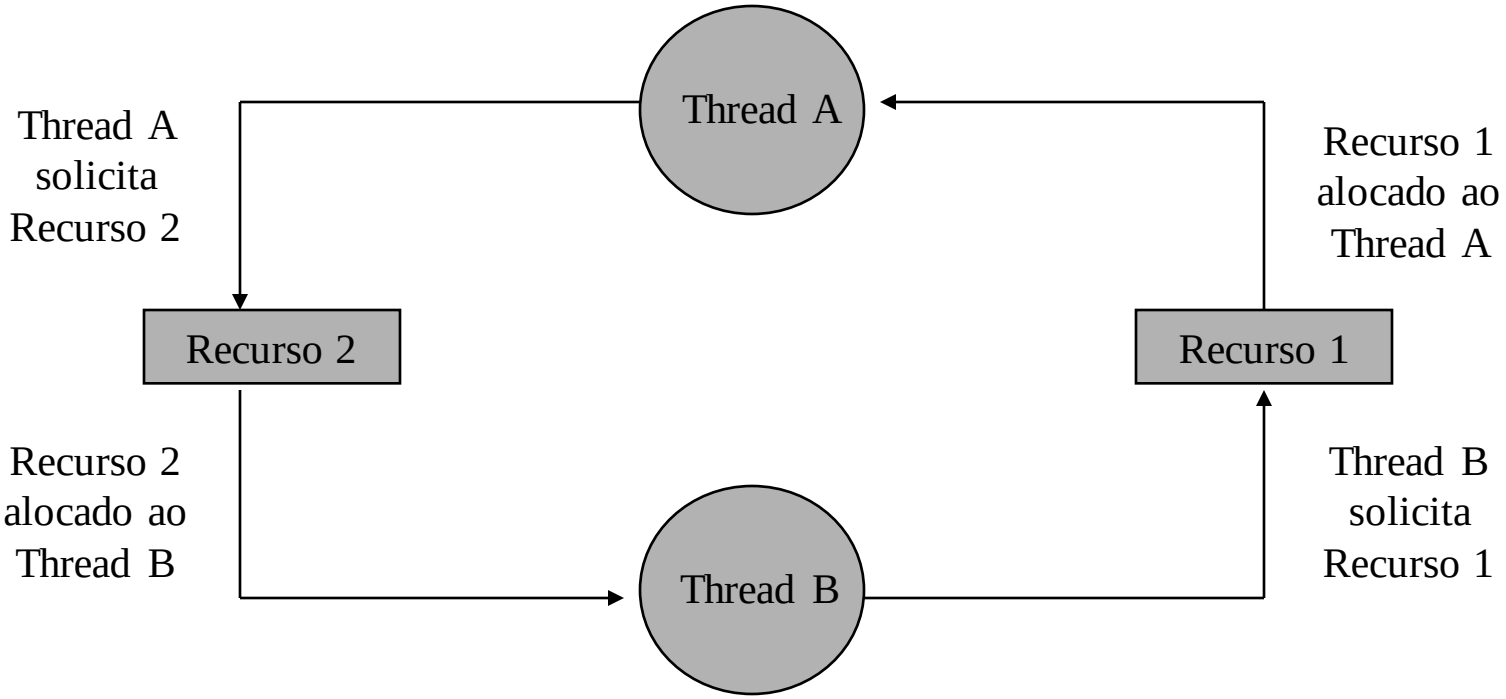
```
PROGRAM Exemplo;
  MONITOR Condicional;
    VAR Cheio : (* variável condicional *);
    PROCEDURE Produz;
    BEGIN
      IF (Cont = TamBuf) THEN WAIT(Cheio);
      .....
    END;
    PROCEDURE Consome;
    BEGIN
      .....
      IF (Cont = TamBuf - 1) THEN SIGNAL(Cheio);
    END;
  BEGIN
    Cont := 0;
  END;
```

Processamento Paralelo – Deadlock

- Definição
  - Situação em que um proc/thread aguarda por um recurso que nunca estará disponível ou por um evento que não ocorrerá.
- Condições
  - Cada recurso só pode estar alocado a um único proc/thread em um determinado instante (*exclusão mútua*).
  - Um proc/thread, além dos recursos já alocados, pode estar esperando por outros recursos (*posse e espera*).
  - Um recurso não pode ser liberado de um proc/thread só porque outro proc/thread deseja o mesmo recurso (*não-preempção*).
  - Um proc/thread pode ter de esperar por um recurso alocado a outro proc/thread e vice-versa (*espera circular*).

Processamento Paralelo – Deadlock

- Espera Circular
  - Modelo



Processamento Paralelo – Problemas Clássicos

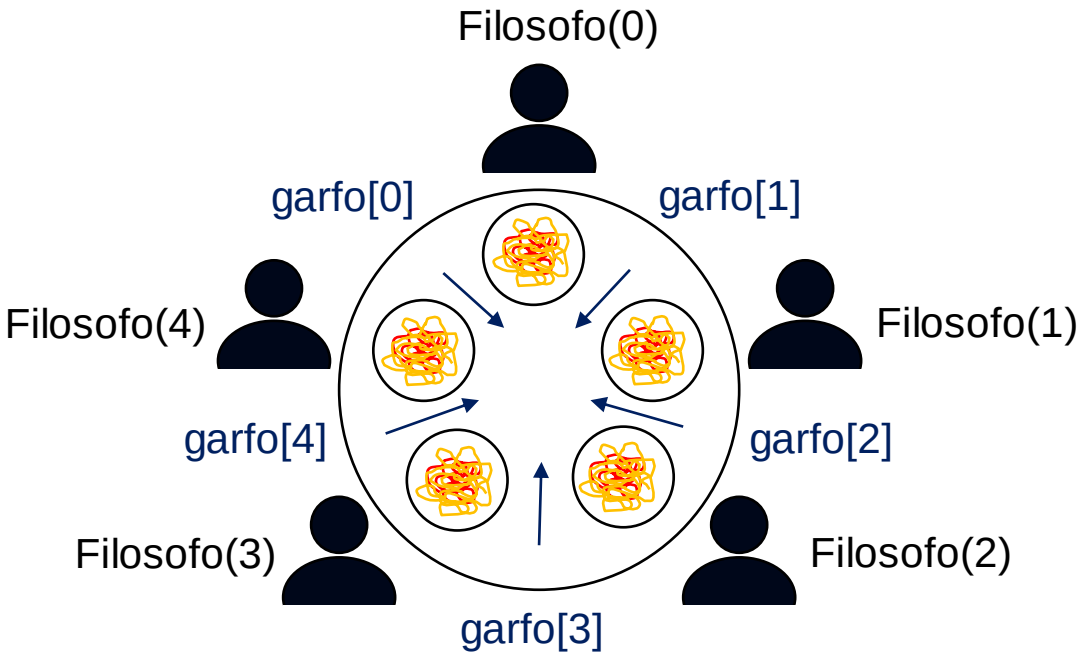
- Filósofos
  - Cinco filósofos sentados ao redor de uma mesa circular.
  - Cada filósofo tem um prato de macarronada a sua frente.
  - Um filósofo precisa de dois garfos para comer a macarronada.
  - Entre cada prato existe um garfo.
  - Um filósofo alterna entre comer e pensar.
  - Quando um filósofo fica com fome, ele tenta obter os garfos a sua direita e a sua esquerda.
  - Se conseguir pegar os dois garfos, o filósofo come um pouco, e, então, coloca os garfos sobre a mesa e volta a pensar.
  - Desafio:
    - » Escrever um programa onde os filósofos fazem o que têm que fazer e nunca entram em deadlock.

Processamento Paralelo – Problemas Clássicos

- Filósofos
  - Cenário:
    - » Observações:

garfo_direito_id = filosofo_id

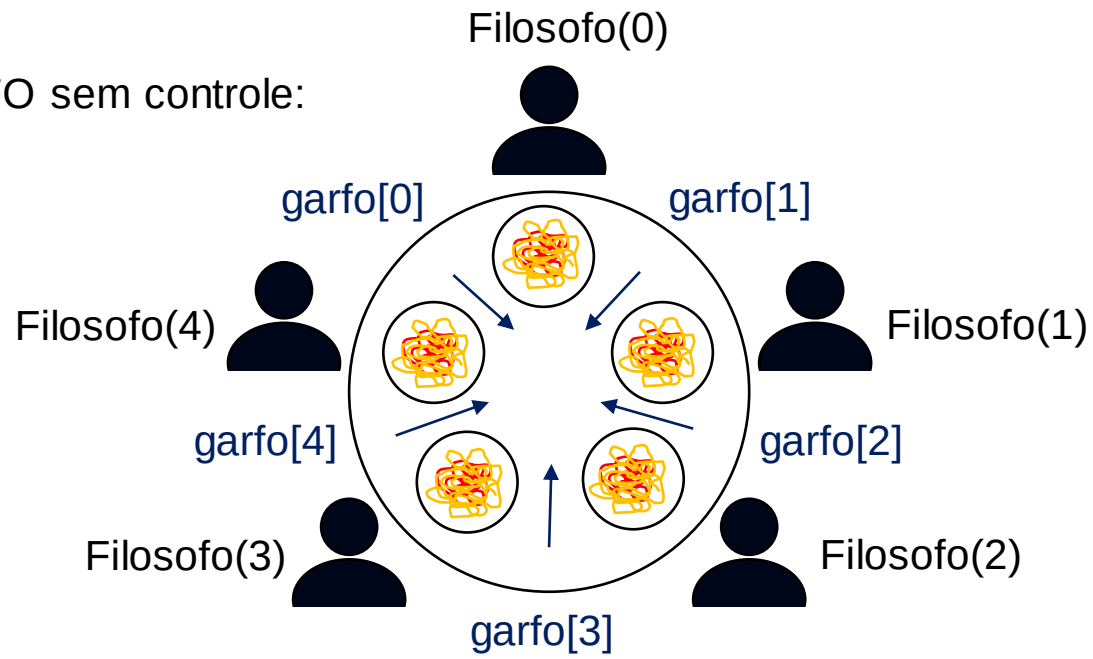
garfo_esquerdo_id =
(filosofo_id + 1) MOD total_filosofos



Processamento Paralelo – Problemas Clássicos

- Filósofos
  - Cenário:
    - » Thread FILOSOFO sem controle:

```
PROCEDURE FILOSOFO(I : INTEGER);
BEGIN
  REPEAT
    PENSAR;
    PEGAR_GARFO(I);
    PEGAR_GARFO((I+1) MOD N);
    COMER;
    LARGAR_GARFO(I);
    LARGAR_GARFO((I+1) MOD N);
  UNTIL FALSE;
END;
```

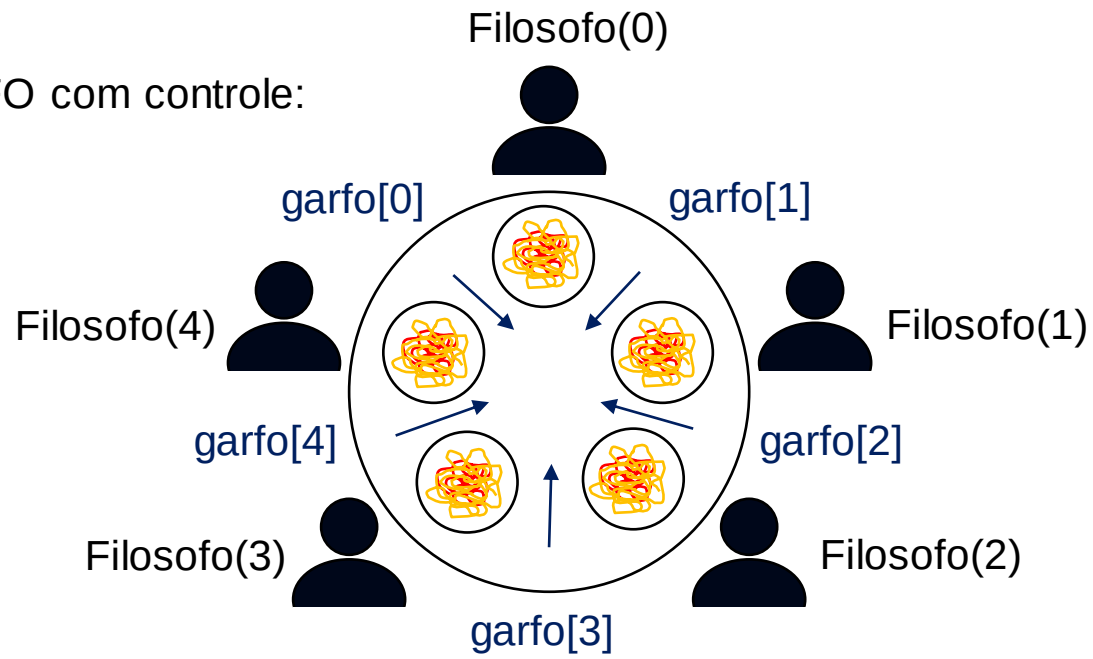


Esse procedimento implementa todas as atividades que um filósofo deve executar...
Mas, o que pode ocorrer se cada filosofo executar a rotina pegar_garfo(i) simultaneamente?

Processamento Paralelo – Problemas Clássicos

- Filósofos
  - Cenário:
    - » Thread FILOSOFO com controle:

```
PROCEDURE FILOSOFO(I : INTEGER);
BEGIN
  REPEAT
    PENSAR;
    DOWN (MUTEX);
    PEGAR_GARFO(I);
    PEGAR_GARFO((I+1) MOD N);
    COMER;
    LARGAR_GARFO(I);
    LARGAR_GARFO((I+1) MOD N);
    UP(MUTEX);
  UNTIL FALSE;
END;
```



Solução ineficiente!!! Apenas um filósofo por vez poderá comer!!!
Com 5 garfos na mesa, até 2 filósofos não adjacentes deveriam poder comer simultaneamente...

Processamento Paralelo – Problemas Clássicos

- Leitores e Escritores
  - Grande base de dados com muitas threads competindo para ler e escrever nela.
  - Múltiplas threads podem ler da base ao mesmo tempo.
  - Porém, se uma thread estiver escrevendo na base, nenhuma outra thread pode ter acesso de escrita ou de leitura na base.
  - Desafio
    - » Programar os leitores e os escritores.

Processamento Paralelo – Problemas Clássicos

- Leitores e Escritores
  - Qual o problema desta solução?

```
VAR
  MUTEX: SEMAFORO := 1;          (* BINARIO  *)
  BD: SEMAFORO := 1              (* BINARIO  *)
  CL: INTERGER;                  (* CONTADOR DE LEITORES  *)
  DADO_LIDO: TIPO_DADO;          (* DADO A SER LIDO *)
  DADO_PRODUZIDO: TIPO_DADO;     (* DADO A SER ESCRITO *)

PROCEDURE ESCRITOR;
BEGIN
  REPEAT
    DADO_PRODUZIDO := PRODUZ_DADO;
    DOWN(BD);
      ESCREVE_DADO(DADO_PRODUZIDO);
    UP(BD);
  UNTIL FALSE;
END;
```

```
PROCEDURE LEITOR;
BEGIN
  REPEAT
    DOWN(MUTEX);
      CL := CL + 1;
      IF CL = 1 THEN DOWN(BD); END;
    UP(MUTEX);
    DADO_LIDO := LE_BASE_DADOS;
    DOWN(MUTEX);
      CL := CL - 1;
      IF CL = 0 THEN UP(BD); END;
    UP(MUTEX);
    CONSOME_DADO(DADO_LIDO);
  UNTIL FALSE;
END;
```

Processamento Paralelo – Problemas Clássicos

- Leitores e Escritores
  - Qual o problema desta solução?

```
VAR
  MUTEX: SEMAFORO := 1;          (* BINARIO  *)
  BD: SEMAFORO := 1              (* BINARIO  *)
  CL: INTERGER;                  (* CONTADOR DE LEITORES  *)
  DADO_LIDO: TIPO_DADO;          (* DADO A SER LIDO *)
  DADO_PRODUZIDO: TIPO_DADO;     (* DADO A SER ESCRITO *)

PROCEDURE ESCRITOR;
BEGIN
  REPEAT
    DADO_PRODUZIDO := PRODUZ_DADO;
    DOWN(BD);
      ESCREVE_DADO(DADO_PRODUZIDO);
    UP(BD);
  UNTIL FALSE;
END;
```

```
PROCEDURE LEITOR;
BEGIN
  REPEAT
    DOWN(MUTEX);
      CL := CL + 1;
      IF CL = 1 THEN DOWN(BD); END;
    UP(MUTEX);
    DADO_LIDO := LE_BASE_DADOS;
    DOWN(MUTEX);
      CL := CL - 1;
      IF CL = 0 THEN UP(BD); END;
    UP(MUTEX);
    CONSOME_DADO(DADO_LIDO);
  UNTIL FALSE;
END;
```

Se um leitor é liberado para leitura, outros poderão fazê-lo em seguida, impossibilitando saber quando um escritor poderá escrever no BD (*starvation*)!!!

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Criando e destruindo os mutexes:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

  - ***pthread_mutex_t *mutex:*** objeto mutex que pode ser inicializado com a macro default `PTHREAD_MUTEX_INITIALIZER`.
  - ***pthread_mutexattr_t *attr:*** objeto atributo do mutex que pode ser NULL para assumir os valores default do mutex.
  - ***retorno:*** em caso de sucesso, retorna o valor zero e, caso contrário, retorna um número de erro para indicar o erro.

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Bloqueando e desbloqueando os mutexes:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  - ***pthread_mutex_t *mutex:*** objeto mutex que deve ser bloqueado ou desbloqueado.
  - ***retorno:*** em caso de sucesso, retorna o valor zero e, caso contrário, retorna um número de erro para indicar o erro.
  - Observação:
    - » A função `pthread_mutex_lock` solicita o bloqueio do mutex. Caso o mutex já esteja bloqueado, então a thread será bloqueada na fila de espera do mutex.
    - » A função `pthread_mutex_trylock` tentará bloquear o mutex mas, se o mutex já estiver bloqueado, então a thread não será bloqueada na fila de espera do mutex.

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: mutex_pthread.c

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

/* This global structure will be accessed by the threads.
 * The arrays (a and b) will be split in slices for each thread.
 * The sum field is the critical region and will be protected by
 * a mutex variable (semaphore).
 */

typedef struct
 {
   double     *a;
   double     *b;
   double     sum;
   int     slicelen;
 } DOTDATA;
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: mutex_pthread.c

```
/* Define globally accessible variables and a mutex */

#define NUMTHRDS 4
#define SLICELEN 100

DOTDATA dotstr;
pthread_t threads[NUMTHRDS];
pthread_mutex_t mutexsum;
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: mutex_pthread.c

```
/* This function will be executed by each thread. It calculates
 * the product of each element of a slice of the arrays a and b
 * and calculates the sum (mysum) of the result of each product.
 * Then, the critical region is accessed in order to update the
 * global sum.
 */
 void *dotprod(void *arg) {
    int i, start, end, len ;
    long offset;
    double mysum, *x, *y;

    /* Calculate the boundary of the slice of this thread */
    offset = (long)arg;

    len = dotstr.slicelen;
    start = offset*len;
    end   = start + len;
    x = dotstr.a;
    y = dotstr.b;
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: mutex_pthread.c

```
    mysum = 0;
    for (i=start; i<end ; i++)
      mysum += (x[i] * y[i]);

  /* Lock a mutex prior to updating the value in the shared
   * structure, and unlock it upon updating.
   */
    pthread_mutex_lock (&mutexsum);
    dotstr.sum += mysum;
    pthread_mutex_unlock (&mutexsum);

    pthread_exit((void*) 0);
  }
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: mutex_pthread.c

```
int main (int argc, char *argv[])
{
    int rc;
    long i;
    double *a, *b;
    void *status;
    pthread_attr_t attr;

    /* Assign storage and initialize values */
    a = (double*) malloc (NUMTHRDS*SLICELEN*sizeof(double));
    b = (double*) malloc (NUMTHRDS*SLICELEN*sizeof(double));

    for (i=0; i<SLICELEN*NUMTHRDS;  i++) {
      a[i]=1.0;
      b[i]=a[i];
    }

    dotstr.slicelen = SLICELEN;
    dotstr.a = a;
    dotstr.b = b;
    dotstr.sum=0;
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: mutex_pthread.c

```
pthread_mutex_init(&mutexsum, NULL);

/* Create threads to perform the dotproduct  */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

for(i=0; i<NUMTHRDS; i++) {
 /* Each thread works on a different set of data. The offset is specified
  * by 'i'. The size of the data for each thread is indicated by SLICELEN.
  */
 if (rc = pthread_create(&threads[i], &attr, dotprod, (void *)i)) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
 }
}
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: mutex_pthread.c

```
    pthread_attr_destroy(&attr);

    /* Wait on the other threads */
    for(i=0; i<NUMTHRDS; i++) {
      if (rc = pthread_join(threads[i], &status)) {
        printf("ERROR; return code from pthread_join() is %d\n", rc);
        exit(-1);
      }
    }

    /* After joining, print out the results and cleanup */
    printf ("Sum =  %f \n", dotstr.sum);
    free (a);
    free (b);
    pthread_mutex_destroy(&mutexsum);
    pthread_exit(NULL);
}
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: mutex_pthread.c

```
$ gcc -pthread -o mutex_pthread  mutex_pthread.c

$ ./mutex_pthread
Sum =  400.000000
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Criando e destruindo variáveis condicionais:

```
int pthread_cond_init(pthread_cond_t *cond,
    const pthread_condattr_t *attr);

int pthread_cond_destroy(pthread_cond_t *cond);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

  - ***pthread_cond_t *cond:*** objeto condicional que pode ser inicializado com a macro default `PTHREAD_COND_INITIALIZER`.
  - ***pthread_condattr_t *attr:*** objeto atributo do objeto condicional que pode ser NULL para assumir os valores default do objeto condicional.
  - ***retorno:*** em caso de sucesso, retorna o valor zero e, caso contrário, retorna um número de erro para indicar o erro.

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Sinalizando sobre uma variável condicional:

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

  - ***pthread_cond_t *cond:*** objeto condicional que deve ser sinalizado.
  - ***retorno:*** em caso de sucesso, retorna o valor zero e, caso contrário, retorna um número de erro para indicar o erro.
  - Observação:
    - » A função `pthread_cond_signal` sinaliza que o objeto condicional foi atendido, o que libera uma thread que esteja bloqueada na fila de espera do objeto condicional.
    - » A função `pthread_cond_broadcast` sinaliza que o objeto condicional foi atendido, o que libera todas as threads que estejam bloqueadas na fila de espera do objeto condicional.
    - » O teste da condição de sinalização deve ser feito dentro de uma região crítica com proteção de exclusão mútua com o objeto mutex indicado pelo wait.

Processamento Paralelo – POSIX Thread

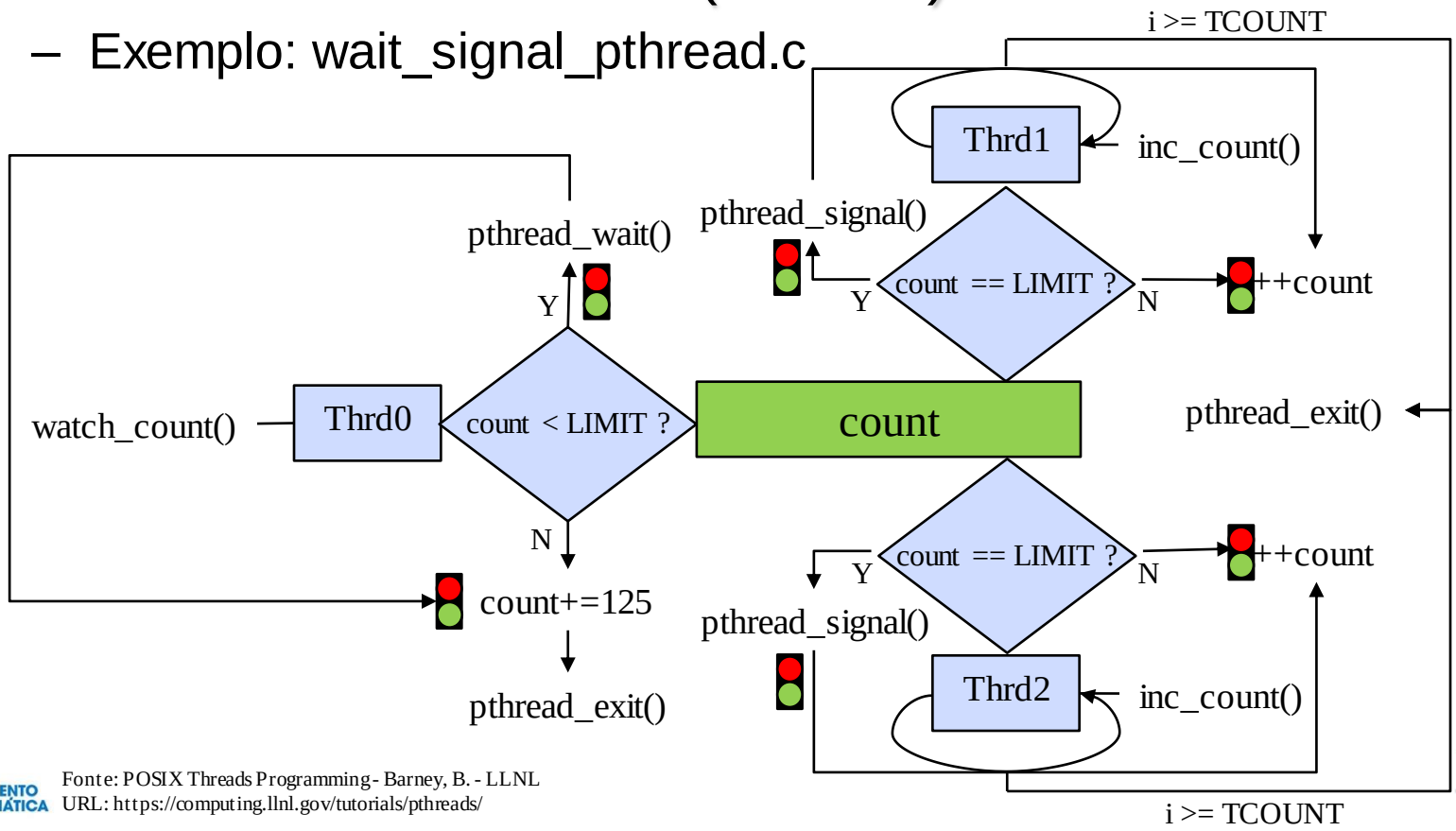
- Biblioteca POSIX Thread (Pthread)
  - Aguardando a sinalização de uma variável condicional:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t
                      *mutex);
```

  - ***pthread_cond_t *cond:*** objeto condicional no qual a thread que chama a função deve aguardar pela sinalização, ou seja, a thread fica bloqueada na fila de espera do objeto condicional (o mutex é automaticamente desbloqueado).
  - ***pthread_mutex_t * mutex:*** a thread deve fornecer o mutex de controle de exclusão mútua do objeto condicional.
  - ***retorno:*** em caso de sucesso, retorna o valor zero e, caso contrário, retorna um número de erro para indicar o erro.
  - Observação:
    - » O mutex de controle de exclusão mútua do objeto condicional deve ser bloqueado (pthread_mutex_lock) antes da chamada da função pthread_cond_wait, e quando retornar dessa função, deve desbloquear o mutex com ptthread_mutex_unlock.

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: wait_signal_pthread.c



Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: wait_signal_pthread.c

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

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT  12

int     count = 0;
long    thread_ids[3] = {0,1,2};

pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cond_var;
```


Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: wait_signal_pthread.c

```
void *inc_count(void *t) {
  int i;
  long my_id = (long)t;

  for (i=0; i<TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

     /* Check the value of count and signal waiting thread when condition is
        reached.  Note that this occurs while mutex is locked. */
    if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cond_var);
      printf("inc_count():  thread %ld, count = %d  Threshold reached.\n", my_id, count);
    }
    printf("inc_count():  thread %ld, count = %d, unlocking mutex\n", my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Hold on for 1 second so threads can alternate on mutex lock */
    sleep(1);
  }
  pthread_exit(NULL);
}
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: wait_signal_pthread.c

```
void *watch_count(void *t) {
  long my_id = (long)t;

  printf("Starting watch_count(): thread %ld\n", my_id);

  /* Lock mutex and wait for signal. Note that the pthread_cond_wait
   * routine will automatically and atomically unlock mutex while it waits.
   * Also, note that if COUNT_LIMIT is reached before this routine is run by
   * the waiting thread, the loop will be skipped to prevent pthread_cond_wait
   * from never returning. */
  pthread_mutex_lock(&count_mutex);
  while (count<COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cond_var, &count_mutex);
    printf("watch_count(): thread %ld Condition signal received.\n", my_id);
  }
  count += 125;
  printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: wait_signal_pthread.c

```
int main (int argc, char *argv[]) {
  int i, rc;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cond_var, NULL);

  /* For portability, explicitly create threads in a joinable state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
```


Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: wait_signal_pthread.c

```
if (rc = pthread_create(&threads[0], &attr, watch_count, (void *)thread_ids[0])) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
 }
 if (rc = pthread_create(&threads[1], &attr, inc_count, (void *)thread_ids[1])) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
 }
 if (rc = pthread_create(&threads[2], &attr, inc_count, (void *)thread_ids[2])) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
 }
```


Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: wait_signal_pthread.c

```
/* Wait for all threads to complete */
  for (i=0; i<NUM_THREADS; i++) {
    if (rc = pthread_join(threads[i], NULL)) {
      printf("ERROR; return code from pthread_join() is %d\n", rc);
      exit(-1);
    }
  }
  printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);

 /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cond_var);
  pthread_exit(NULL);
```


Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: wait_signal_pthread.c

```
$ gcc -pthread -o wait_signal_pthread wait_signal_pthread.c

$ ./wait_signal_pthread
Starting watch_count():  thread 0
inc_count():  thread 1, count = 1, unlocking  mutex
inc_count():  thread 2, count = 2, unlocking  mutex
inc_count():  thread 1, count = 3, unlocking  mutex
inc_count():  thread 2, count = 4, unlocking  mutex
inc_count():  thread 1, count = 5, unlocking  mutex
inc_count():  thread 2, count = 6, unlocking  mutex
inc_count():  thread 1, count = 7, unlocking  mutex
inc_count():  thread 2, count = 8, unlocking  mutex
inc_count():  thread 1, count = 9, unlocking  mutex
inc_count():  thread 2, count = 10, unlocking  mutex
inc_count():  thread 1, count = 11, unlocking  mutex
inc_count():  thread 2, count = 12  Threshold  reached.
inc_count():  thread 2, count = 12, unlocking  mutex
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Biblioteca POSIX Thread (Pthread)
  - Exemplo: wait_signal_pthread.c

```
$ gcc -pthread -o wait_signal_pthread wait_signal_pthread.c

$ ./wait_signal_pthread
……
watch_count(): thread 0 Condition signal received.
watch_count(): thread 0 count now = 137.
inc_count(): thread 1, count = 138, unlocking mutex
inc_count(): thread 2, count = 139, unlocking mutex
inc_count(): thread 1, count = 140, unlocking mutex
inc_count(): thread 2, count = 141, unlocking mutex
inc_count(): thread 1, count = 142, unlocking mutex
inc_count(): thread 2, count = 143, unlocking mutex
inc_count(): thread 1, count = 144, unlocking mutex
inc_count(): thread 2, count = 145, unlocking mutex
Main(): Waited on 3 threads. Done.
```

Fonte: POSIX Threads Programming - Barney, B. - LLNL
URL: https://computing.llnl.gov/tutorials/pthreads/

Processamento Paralelo – POSIX Thread

- Exercício 2
  - Implemente um programa com processamento paralelo de thread para resolver o problema de concorrência e sincronização dos Filósofos.
  - Utilize o código *filososfos_sem_controle_pthread.c*, que está com problema de concorrência e sincronização, como base para o exercício.
  - A solução deve garantir a exclusão mútua dos garfos disponíveis e impedir o deadlock das threads dos filósofos.
  - Lembre-se: com cinco garfos disponíveis, dois filósofos poderão comer simultaneamente, enquanto os demais deverão aguardar uma chance para pegar os garfos necessários para poderem comer.

Processamento Paralelo – POSIX Thread

- Exercício 2
  - Problema de concorrência e sincronismo no programa *filosofos_sem_controle_pthread.c*:

```
Filosofo 3 pensando...
Filosofo 4 pensando...
Filosofo 2 pensando...
Filosofo 1 pensando...
Filosofo 0 pensando...
Filosofo 4 pegou garfo 4...
Filosofo 2 pegou garfo 2...
Filosofo 1 pegou garfo 1...
Filosofo 0 pegou garfo 0...
Filosofo 3 pegou garfo 3...
Filosofo 4 pegou garfo 0...
Filosofo 3 pegou garfo 4...
Filosofo 1 pegou garfo 2...
Filosofo 2 pegou garfo 3...
Filosofo 0 pegou garfo 1...
Filosofo 3 comendo...
Filosofo 4 comendo...
Filosofo 1 comendo...
Filosofo 2 comendo...
Filosofo 0 comendo...
```

Todos os 5 filósofos pegaram, simultaneamente, os respectivos garfos à direita e à esquerda, quando deveriam pegar APENAS os garfos livres!!!

Todos os 5 filósofos estão comendo simultaneamente, quando APENAS 2 deles poderiam estar comendo simultaneamente!!!

Introdução à Arquitetura de Computadores

Alexandre Meslin

Material baseado nos slides de:
Anderson Oliveira da Silva

Departamento de Informática
PUC-Rio