Interface Hardware/Software

Programação Paralela: PThreads

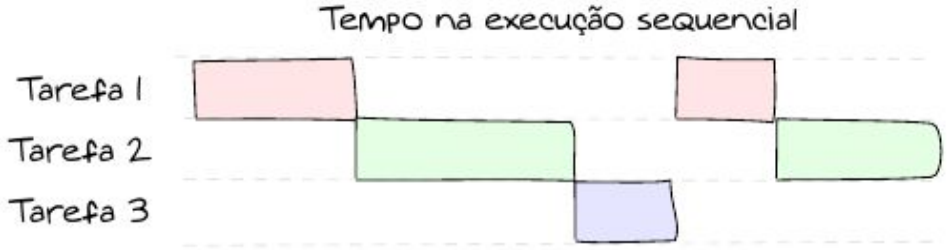
UNIVERSIDADE FEDERAL DE SERGIPE

Professor
Calebe Conceição

Uma breve revisão sobre Threads

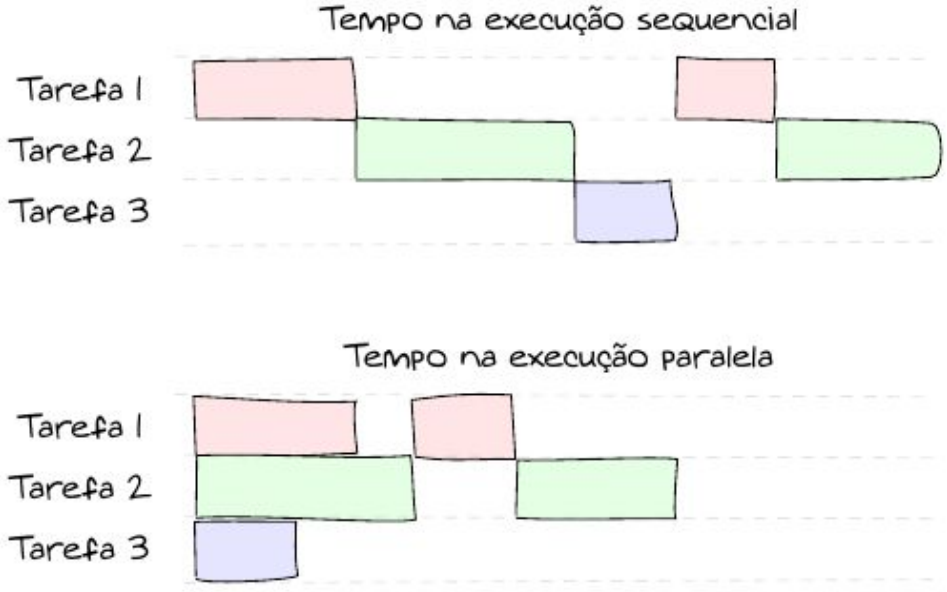
Introdução

- Modelo paralelo de programação (*thread*)
  - Os fluxos de execução são sobrepostos no tempo
  - As *threads* executam de forma independente em um processo, permitindo a execução paralela em um sistema com múltiplas unidades de processamento



Introdução

- Modelo paralelo de programação (*thread*)
  - Os fluxos de execução são sobrepostos no tempo
  - As *threads* executam de forma independente em um processo, permitindo a execução paralela em um sistema com múltiplas unidades de processamento

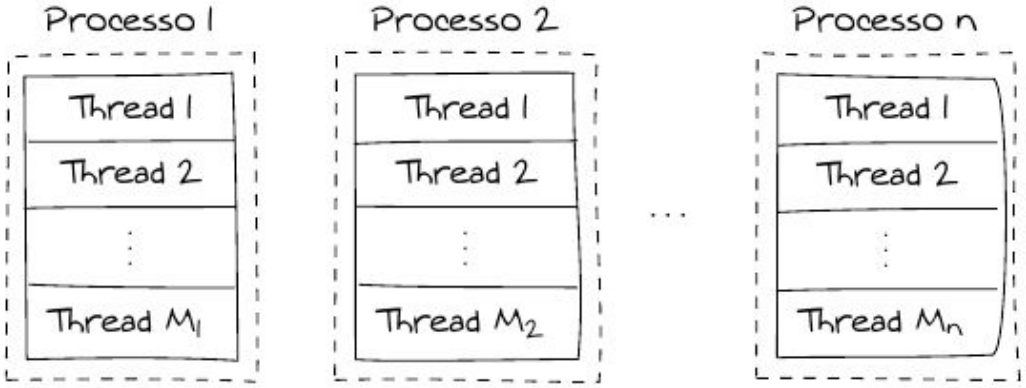


Introdução

- Qual a diferença entre um processo e uma *thread*?
  - Processo
    - É uma instância de uma aplicação executando
    - O escalonamento é feito pelo SO
    - Contexto + Memória Virtual + Recursos alocados

Introdução

- Qual a diferença entre um processo e uma *thread*?
  - Processo
    - É uma instância de uma aplicação executando
    - O escalonamento é feito pelo SO
    - Contexto + Memória Virtual + Recursos alocados
  - *Thread*
    - Só existe como parte de um processo
    - O escalonamento é feito pelo programador
    - Utiliza os mesmos recursos do processo

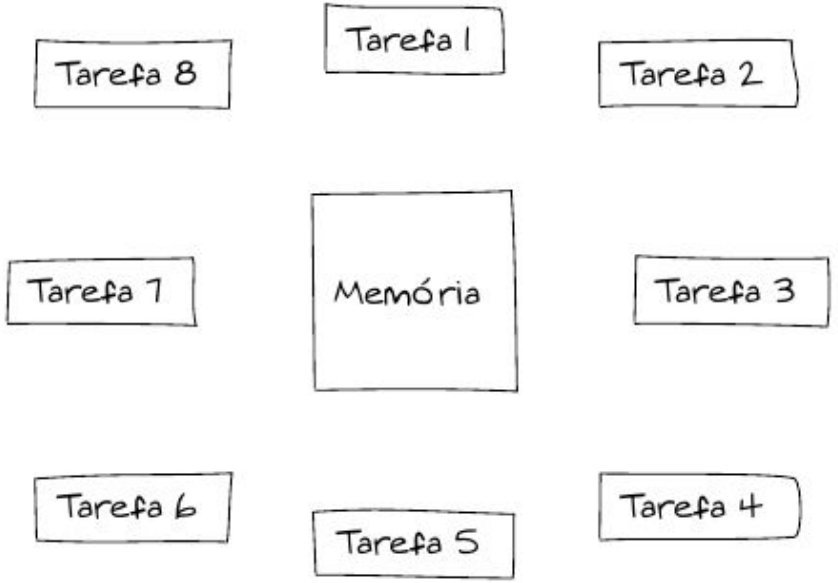


Concorrência x Paralelismo
Lei de Amdahl

O quanto eu ganho ao adotar paralelismo?

Introdução

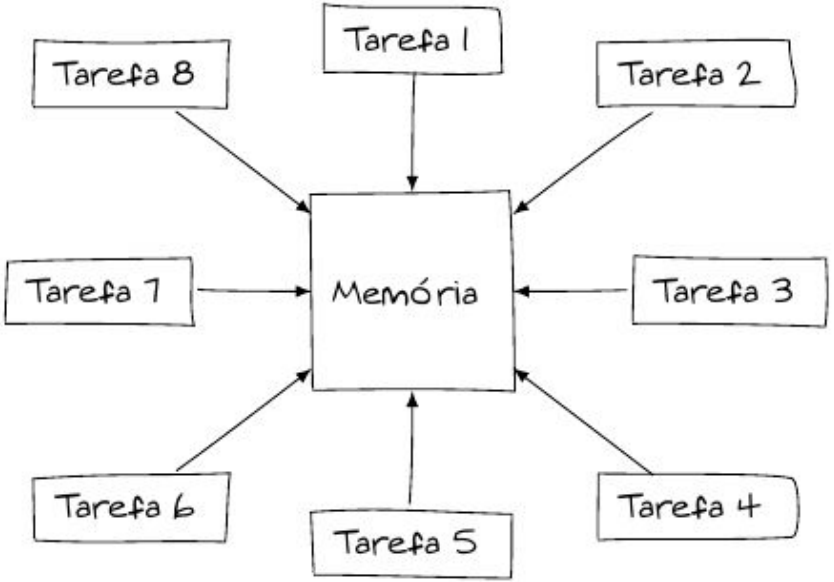
- Concorrência × Paralelismo
  - O compartilhamento de recursos do sistema implica em concorrência de acesso (retenção)
  - Esta espera impede a execução paralela, uma vez que os recursos disponíveis são limitados



Introdução

- Concorrência × Paralelismo
  - O compartilhamento de recursos do sistema implica em concorrência de acesso (retenção)
  - Esta espera impede a execução paralela, uma vez que os recursos disponíveis são limitados

```
T1.write(0xF0F0, 1);
T2.write(0xF0F0, 2);
T3.write(0xF0F0, 3);
T4.write(0xF0F0, 4);
T5.write(0xF0F0, 5);
T6.write(0xF0F0, 6);
T7.write(0xF0F0, 7);
T8.write(0xF0F0, 8);
```



Introdução

- Lei de Amdahl
  - A melhoria de desempenho está limitada a parte S do software que é inerentemente sequencial e não a parte $P = 1 - S$ que pode ser paralelizada

$$Melhoria = \frac{Execução\ sequencial}{Execução\ paralela}$$

$$Amdahl(S, N) = \frac{1}{S + \frac{(1-S)}{N}}$$

$$Amdahl(P, N) = \frac{1}{(1 - P) + \frac{P}{N}}$$

Tudo pode ser paralelizado?

Introdução

- O que pode ser paralelizado?
  - Função fatorial

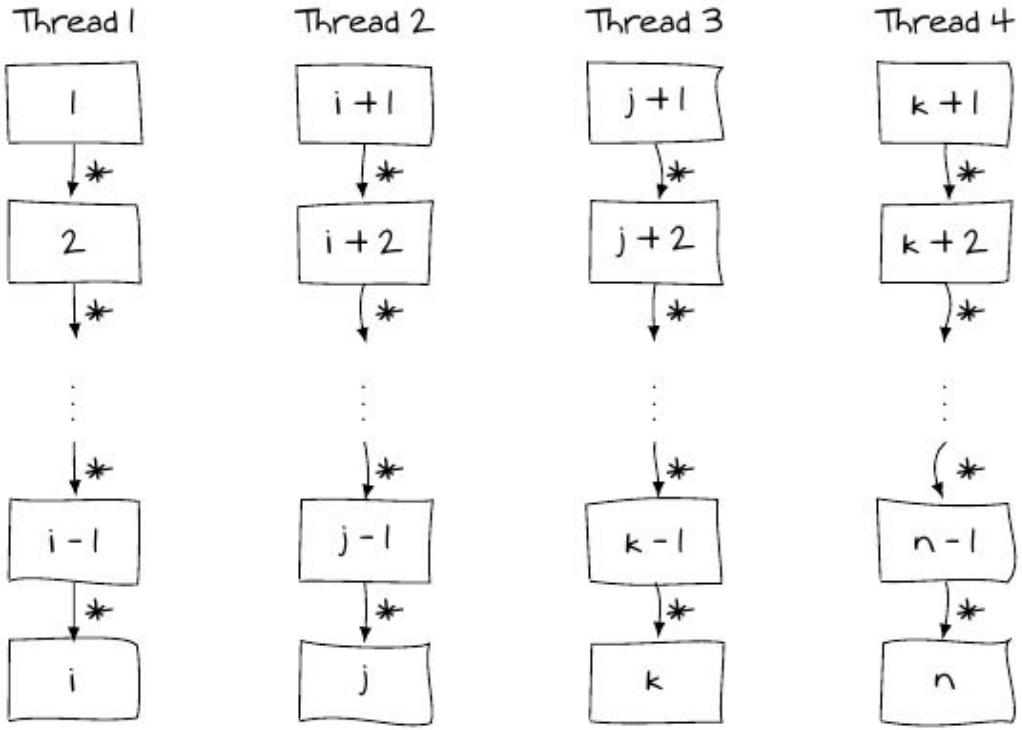
```
1  // Padronização de tipos inteiros
2  #include <stdint.h>
3  // Função fatorial
4  uint64_t fatorial(uint32_t n) {
5      // Resultado
6      uint64_t r = 1;
7      // Iterações de 2 até n
8      for(uint32_t i = 2; i <= n; i++)
9          // Multiplicação pelo índice
10         r = r * i;
11     // Retorno do resultado
12     return r;
13 }
```

Introdução

- O que pode ser paralelizado?
  - Função fatorial

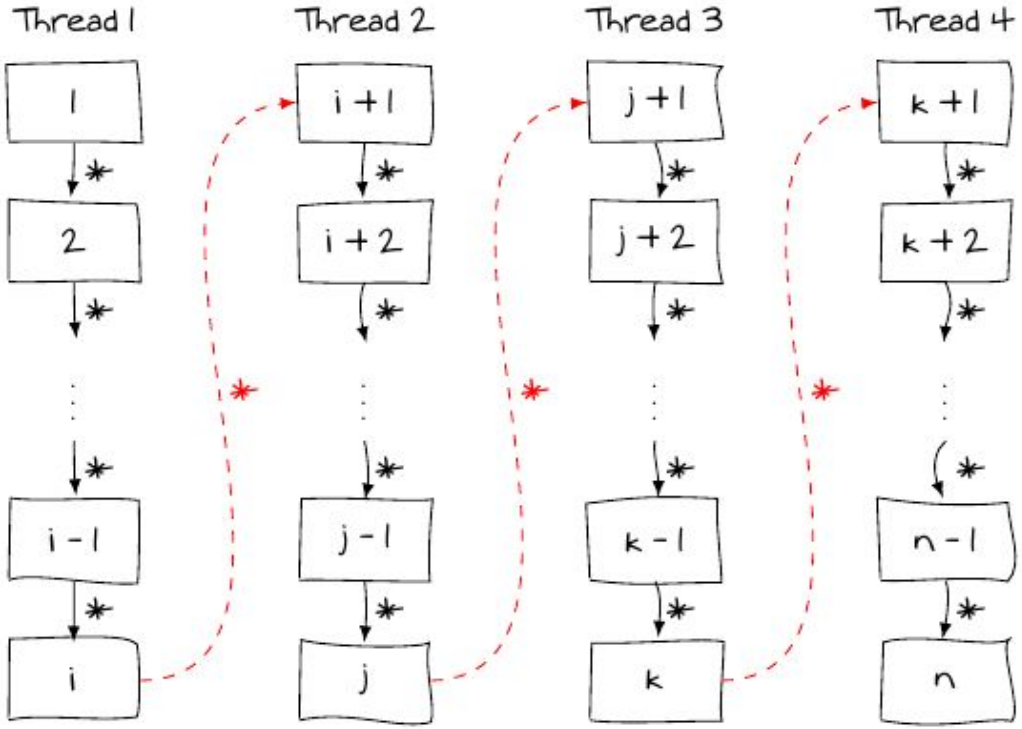
Introdução

- O que pode ser paralelizado?
  - Função fatorial



Introdução

- O que pode ser paralelizado?
  - Função fatorial



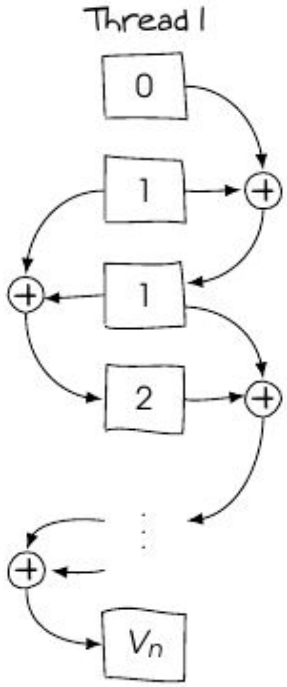
Introdução

- O que pode ser paralelizado?
  - Função fibonacci

```
1  // Padronização de tipos inteiros
2  #include <stdint.h>
3  // Função fibonacci
4  uint64_t fibonacci(uint32_t n) {
5      // T(n), T(n - 2) e T(n - 1)
6      uint64_t r = n, tn2 = 0, tn1 = 1;
7      // Iterações de 1 até n - 1
8      for(uint32_t i = 1; i < n; i++) {
9          // T(n) = T(n - 2) + T(n - 1)
10         r = tn2 + tn1;
11         // T(n - 2) = T(n - 1)
12         tn2 = tn1;
13         // T(n - 1) = T(n)
14         tn1 = r;
15     }
16     // Retornando T(n)
17     return r;
18 }
```

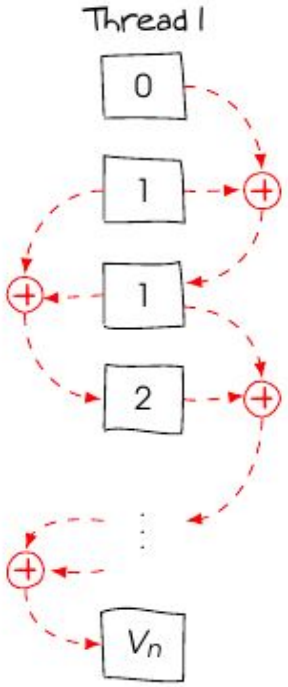
Introdução

- O que pode ser paralelizado?
  - Função fibonacci



Introdução

- O que pode ser paralelizado?
  - Função fibonacci



Introdução

- O que é uma função *thread-safe*?
  - Consiste em uma implementação que garante a execução simultânea sem interações indesejadas

Introdução

- O que é uma função *thread-safe*?
  - Consiste em uma implementação que garante a execução simultânea sem interações indesejadas
  - São utilizados mecanismos de sincronização para evitar condições de corrida

Introdução

- O que é uma função *thread-safe*?
  - Consiste em uma implementação que garante a execução simultânea sem interações indesejadas
  - São utilizados mecanismos de sincronização para evitar condições de corrida

```
1  // Número pseudo-aleatório
2  static uint32_t ri = 1;
3  // Função para ajuste de semente
4  void srand(uint32_t seed) {
5      ri = seed;
6  }
7  // Função de número pseudo-aleatório
8  uint16_t rand() {
9      // Calculando próximo número
10     ri = (1103515245 * ri) + 12345;
11     // Selecionando bits 30 até 16
12     return ((ri >> 16) & 0x7FFF);
13 }
```

Introdução

- O que é uma função *thread-safe*?
  - Consiste em uma implementação que garante a execução simultânea sem interações indesejadas
  - São utilizados mecanismos de sincronização para evitar condições de corrida

```
1  // Número pseudo-aleatório
2  static uint32_t ri = 1;
3  // Função para ajuste de semente
4  void srand(uint32_t seed) {
5      ri = seed;
6  }
7  // Função de número pseudo-aleatório
8  uint16_t rand() {
9      // Calculando próximo número
10     ri = (1103515245 * ri) + 12345;
11     // Selecionando bits 30 até 16
12     return ((ri >> 16) & 0x7FFF);
13 }
```

Este código-fonte é *thread-safe*?

Introdução

- Interfaces de programação (API)
  - POSIX *Threads* (pthreads)

Introdução

- Interfaces de programação (API)
  - POSIX *Threads* (pthreads)
  - OpenCL

Introdução

- Interfaces de programação (API)
  - POSIX *Threads* (pthreads)
  - OpenCL
  - *Thread Building Blocks* (TBB)
  - ...

PThreads

POSIX Threads

- ANSI/IEEE POSIX.1c: *thread extensions*
  - Descrita na linguagem de programação C
  - Baixo custo computacional
  - Maior portabilidade entre sistemas

POSIX Threads

- Funcionalidades principais

```
1  /**
2   *  Função para criação de threads
3   *  @param thread          Ponteiro da thread criada
4   *  @param attr            Atributos da thread
5   *  @param start_routine   Rotina de execução
6   *  @param arg             Argumento da rotina
7   *  @return                Retorna sucesso (0) ou erro
       (EAGAIN - falta de recursos ou limite de threads
       atingido, EINVAL - parâmetros incorretos em attr
       ou EPERM - sem permissão para política de
       escalonamento ou parâmetros especificados em attr
8   */
9  int pthread_create(pthread* thread, const
       pthread_attr_t* attr,
       void*(*start_routine)(void*), void* arg);
```

POSIX Threads

- Funcionalidades principais

```
1  /**
2   *  Função para tornar thread avulsa (detached)
3   *  @param thread  Identificador da thread avulsa
4   *  @return        Retorna sucesso (0) ou erro (EINVAL
        - a thread já é avulsa ou ESRCH - thread não
        encontrada)
5   */
6  int pthread_detach(pthread thread);
```

POSIX Threads

- Funcionalidades principais

```
/**
 *  Funções de acesso e ajuste dos atributos de thread
 */
int pthread_attr_destroy(pthread_attr_t* attr);
int pthread_attr_init(pthread_attr_t* attr);
int pthread_attr_getdetachstate(const pthread_attr_t*
    attr, int *detachstate);
int pthread_attr_getschedparam(const pthread_attr_t*
    attr, struct sched_param* param);
int pthread_attr_getschedpolicy(const pthread_attr_t*
    attr, int* policy);
int pthread_attr_setdetachstate(pthread_attr_t* attr,
    int detachstate);
int pthread_attr_setschedparam(pthread_attr_t* attr,
    const struct sched_param* param);
int pthread_attr_setschedpolicy(pthread_attr_t* attr,
    int policy);
```

POSIX Threads

- Funcionalidades principais

```
1  /**
2   *  Procedimento para finalização de thread
3   *  @param retval  Se a thread não for avulsa
       (detached), pode ser retornado um valor através
       deste parâmetro
4   */
5  void pthread_exit(void* retval);
```

POSIX Threads

- Funcionalidades principais

```
1  /**
2   *  Função para espera de finalização de thread
3   *  @param thread  Identificador da thread que terá a
        espera pela finalização de sua execução (join)
4   *  @param retval  Caso não seja nulo, será um
        ponteiro para o valor retornado pela função
        pthread_exit
5   *  @return        Retorna sucesso (0) ou erro
        (EDEADLK - deadlock, EINVAL - thread avulsa ou
        sendo esperada por outra thread ou ESRCH -
        thread não encontrada)
6   */
7  int pthread_join(pthread_t thread, void** retval);
```

POSIX Threads

- Funcionalidades principais

```
1  /**
2   *  Funções para obter acesso de mutex
3   *  @param mutex  Ponteiro para mutex
4   *  @return       Retorna sucesso (0) ou erro (EAGAIN
        - número máximo de acessos recursivos excedidos,
        EBUSY - mutex já obtido por outra thread ou
        somente para o trylock, EDEADLK - deadlock,
        somente para lock ou EINVAL - erro de prioridade
        de mutex)
5   */
6  int pthread_mutex_lock(pthread_mutex_t* mutex);
7  int pthread_mutex_trylock(pthread_mutex_t* mutex);
```

POSIX Threads

- Funcionalidades principais

```
1  /**
2   *  Função para liberar acesso de mutex
3   *  @param mutex  Ponteiro para mutex
4   *  @return       Retorna sucesso (0) ou erro (EAGAIN
        - número máximo de acessos recursivos excedidos,
        EINVAL - erro de prioridade de mutex ou EPERM -
        a thread atual não é dona de mutex)
5   */
6  int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

POSIX Threads

- Funcionalidades principais

```
1  /**
2   *  Funções para geração de sinal de condição
3   *  @param cond  Ponteiro para condição
4   *  @return      Retorna sucesso (0) ou erro (EINVAL -
        variável de condição não inicializada)
5   */
6  int pthread_cond_broadcast(pthread_cond_t* cond);
7  int pthread_cond_signal(pthread_cond_t* cond);
```

POSIX Threads

- Funcionalidades principais

```
1  /**
2   *  Função de espera por sinal de condição
3   *  @param cond     Ponteiro para condição
4   *  @param mutex    Ponteiro para mutex
5   *  @param abstime  Ponteiro para intervalo de tempo
6   *  @return         Retorna sucesso (0) ou erro
       (EINVAL - condição inválida ou mutex inválido ou
       intervalo de tempo inválido para função
       timedwait, ETIMEDOUT - intervalo de tempo
       atingido para função timedwait ou EPERM - a
       thread atual não é dona de mutex)
7   */
8  int pthread_cond_timedwait(pthread_cond_t* cond,
       pthread_mutex_t* mutex, const struct timespec*
       abstime);
9  int pthread_cond_wait(pthread_cond_t* cond,
       pthread_mutex_t* mutex);
```

Demonstrações

Main é uma Thread!

POSIX Threads

- A função *main* é a *thread* mãe

```c
// POSIX threads
#include <pthread.h>
// Entrada e saída padrão
#include <stdio.h>
// Padronização de tipos inteiros
#include <stdint.h>
// Função principal
int main() {
    // Retornando sucesso
    return 0;
}
```

```
$ gcc -Wall -g exemplo.c -o exemplo.elf -lpthread
$ gdb ./exemplo.elf
...

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./exemplo.elf...
(gdb)
```

POSIX Threads

- A função *main* é a *thread* mãe

```c
// POSIX threads
#include <pthread.h>
// Entrada e saída padrão
#include <stdio.h>
// Padronização de tipos inteiros
#include <stdint.h>
// Função principal
int main() {
    // Retornando sucesso
    return 0;
}
```

```
...

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./exemplo.elf...
(gdb) b main
Breakpoint 1 at 0x111d: file exemplo.c, line 10.
(gdb)
```

POSIX Threads

- A função *main* é a *thread* mãe

```c
// POSIX threads
#include <pthread.h>
// Entrada e saída padrão
#include <stdio.h>
// Padronização de tipos inteiros
#include <stdint.h>
// Função principal
int main() {
    // Retornando sucesso
    return 0;
}
```

```
(gdb) b main
Breakpoint 1 at 0x111d: file exemplo.c, line 10.
(gdb) r
Starting program: /home/bruno/Desktop/exemplo.elf

Breakpoint 1, main () at exemplo.c:10
10              return 0;
(gdb)
```

POSIX Threads

- A função *main* é a *thread* mãe

```
1  // POSIX threads
2  #include <pthread.h>
3  // Entrada e saída padrão
4  #include <stdio.h>
5  // Padronização de tipos inteiros
6  #include <stdint.h>
7  // Função principal
8  int main() {
9      // Retornando sucesso
10     return 0;
11 }
```

```
(gdb) r
Starting program: /home/bruno/Desktop/exemplo.elf

Breakpoint 1, main () at exemplo.c:10
10              return 0;
(gdb) thread
[Current thread is 1 (process 14824)]
(gdb)
```

Demonstrações

Criando e terminando Threads

POSIX Threads

- Criação e terminação de *threads*

```
1   // POSIX threads
2   #include <pthread.h>
... ...
7   // Número de threads
8   #define N   4
9   // Função tarefa da thread
10  void* task(void* p) {
11      // Imprimindo parâmetro
12      printf("I␣am␣#%u\n", *(uint32_t*)(p));
13      // Retornando nulo
14      return NULL;
15  }
... ...
```

POSIX Threads

- Criação e terminação de *threads*

```
1   // POSIX threads
2   #include <pthread.h>
... ...
17  // Função principal
18  int main() {
19      // Vetor de N identificadores de threads
20      pthread_t thread[N];
21      // Vetor de N parâmetros de threads
22      uint32_t parameters[N];
23      // Iterando de 0 até N - 1
24      for(uint32_t i = 0; i < N; i++) {
25          // Ajustando parâmetro i para i + 2
26          parameters[i] = i + 2;
27          // Criando thread i
28          pthread_create(&thread[i], NULL, task,
                (void*)(&parameters[i]));
29      }
... ...
```

POSIX Threads

- Criação e terminação de *threads*

```
1   // POSIX threads
2   #include <pthread.h>
... ...
17  // Função principal
18  int main() {
... ...
30      // Iterando de 0 até N - 1
31      for(uint32_t i = 0; i < N; i++) {
32          // Finalizando a thread i
33          pthread_join(thread[i], NULL);
34      }
35      // Retornando sucesso
36      return 0;
37  }
```

POSIX Threads

- Criação e terminação de *threads*

```
$ gcc -Wall -g exemplo.c -o exemplo.elf -lpthread
$ gdb ./exemplo.elf
...

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./exemplo.elf...
(gdb) b task
Breakpoint 1 at 0x1175: file exemplo.c, line 12.
(gdb) r
Starting program: /home/bruno/Desktop/exemplo.elf
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".
[New Thread 0x7ffff7d98640 (LWP 17533)]
[New Thread 0x7ffff7597640 (LWP 17534)]
[New Thread 0x7ffff6d96640 (LWP 17535)]
[Switching to Thread 0x7ffff7d98640 (LWP 17533)]

Thread 2 "exemplo.elf" hit Breakpoint 1, task (p=0x7fffffffdfb0) at ex
emplo.c:12
12              printf("I am #%u\n", *(uint32_t*)(p));
```

POSIX Threads

- Criação e terminação de *threads*

```
Reading symbols from ./exemplo.elf...
(gdb) b task
Breakpoint 1 at 0x1175: file exemplo.c, line 12.
(gdb) r
Starting program: /home/bruno/Desktop/exemplo.elf
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".
[New Thread 0x7ffff7d98640 (LWP 17533)]
[New Thread 0x7ffff7597640 (LWP 17534)]
[New Thread 0x7ffff6d96640 (LWP 17535)]
[Switching to Thread 0x7ffff7d98640 (LWP 17533)]

Thread 2 "exemplo.elf" hit Breakpoint 1, task (p=0x7fffffffdfb0) at ex
emplo.c:12
12              printf("I am #%u\n", *(uint32_t*)(p));
(gdb) c
Continuing.
[New Thread 0x7ffff6595640 (LWP 17610)]
[Switching to Thread 0x7ffff7597640 (LWP 17534)]

Thread 3 "exemplo.elf" hit Breakpoint 1, task (p=0x7fffffffdfb4) at ex
emplo.c:12
12              printf("I am #%u\n", *(uint32_t*)(p));
```

POSIX Threads

- Criação e terminação de *threads*

```
Thread 2 "exemplo.elf" hit Breakpoint 1, task (p=0x7fffffffdfb0) at ex
emplo.c:12
12              printf("I am #%u\n", *(uint32_t*)(p));
(gdb) c
Continuing.
[New Thread 0x7ffff6595640 (LWP 17610)]
[Switching to Thread 0x7ffff7597640 (LWP 17534)]

Thread 3 "exemplo.elf" hit Breakpoint 1, task (p=0x7fffffffdfb4) at ex
emplo.c:12
12              printf("I am #%u\n", *(uint32_t*)(p));
(gdb) c
Continuing.
I am #2
I am #3
[Thread 0x7ffff7597640 (LWP 17534) exited]
[Thread 0x7ffff7d98640 (LWP 17533) exited]
[Switching to Thread 0x7ffff6d96640 (LWP 17535)]

Thread 4 "exemplo.elf" hit Breakpoint 1, task (p=0x7fffffffdfb8) at ex
emplo.c:12
12              printf("I am #%u\n", *(uint32_t*)(p));
```

POSIX Threads

- Criação e terminação de *threads*

```
[Switching to Thread 0x7ffff7597640 (LWP 17534)]

Thread 3 "exemplo.elf" hit Breakpoint 1, task (p=0x7fffffffdfb4) at ex
emplo.c:12
12              printf("I am #%u\n", *(uint32_t*)(p));
(gdb) c
Continuing.
I am #2
I am #3
[Thread 0x7ffff7597640 (LWP 17534) exited]
[Thread 0x7ffff7d98640 (LWP 17533) exited]
[Switching to Thread 0x7ffff6d96640 (LWP 17535)]

Thread 4 "exemplo.elf" hit Breakpoint 1, task (p=0x7fffffffdfb8) at ex
emplo.c:12
12              printf("I am #%u\n", *(uint32_t*)(p));
(gdb) c
Continuing.
[Switching to Thread 0x7ffff6595640 (LWP 17610)]

Thread 5 "exemplo.elf" hit Breakpoint 1, task (p=0x7fffffffdfbc) at ex
emplo.c:12
12              printf("I am #%u\n", *(uint32_t*)(p));
```

POSIX Threads

- Criação e terminação de *threads*

```
[Thread 0x7ffff7597640 (LWP 17534) exited]
[Thread 0x7ffff7d98640 (LWP 17533) exited]
[Switching to Thread 0x7ffff6d96640 (LWP 17535)]

Thread 4 "exemplo.elf" hit Breakpoint 1, task (p=0x7fffffffdfb8) at ex
emplo.c:12
12              printf("I am #%u\n", *(uint32_t*)(p));
(gdb) c
Continuing.
[Switching to Thread 0x7ffff6595640 (LWP 17610)]

Thread 5 "exemplo.elf" hit Breakpoint 1, task (p=0x7fffffffdfbc) at ex
emplo.c:12
12              printf("I am #%u\n", *(uint32_t*)(p));
(gdb) c
Continuing.
I am #4
[Thread 0x7ffff6d96640 (LWP 17535) exited]
I am #5
[Thread 0x7ffff6595640 (LWP 17610) exited]
[Inferior 1 (process 17529) exited normally]
(gdb) q
$
```

Demonstrações

Demonstrando *Racing Condition*

POSIX Threads

- Condições de corrida

```
1   // POSIX threads
2   #include <pthread.h>
3   // Escalonamento de execução
4   #include <sched.h>
... ...
9   // Thread consumidor
10  void* consumer(void* p) {
11      // Conversão de tipo do parâmetro
12      int32_t* v = (int32_t*)(p);
13      // Iterando de 0 até N - 1
14      for(uint32_t i = 0; i < N; i++) {
15          // Liberando escalonamento
16          while(*v <= 0) sched_yield();
17          // Imprimindo valor do parâmetro
18          printf("Consumer = %i\n", (*v > 0) ? (--(*v))
                : (*v));
19      }
... ...
```

POSIX Threads

- Condições de corrida

```
1   // POSIX threads
2   #include <pthread.h>
3   // Escalonamento de execução
4   #include <sched.h>
... ...
23  // Thread produtor
24  void* producer(void* p) {
25      // Conversão de tipo do parâmetro
26      int32_t* v = (int32_t*)(p);
27      // Iterando de 0 até N - 1
28      for(uint32_t i = 0; i < N; i++) {
29          // Liberando escalonamento
30          while(*v >= N) sched_yield();
31          // Imprimindo valor do parâmetro
32          printf("Producer = %i\n", (*v < N) ? (++(*v))
                : (*v));
33      }
... ...
```

POSIX Threads

- Condições de corrida

```
1   // POSIX threads
2   #include <pthread.h>
3   // Escalonamento de execução
4   #include <sched.h>
... ...
37  // Função principal
38  int main() {
39      // Contador compartilhado
40      int32_t counter = 0;
41      // Identificadores consumidores e produtores
42      pthread_t consumers[N], producers[N];
43      // Iterando de 0 até N - 1
44      for(uint32_t i = 0; i < N; i++) {
45          // Criando threads
46          pthread_create(&consumers[i], NULL, consumer,
                (void*)(&counter));
47          pthread_create(&producers[i], NULL, producer,
                (void*)(&counter));
48      }
... ...
```

POSIX Threads

- Condições de corrida

```
1   // POSIX threads
2   #include <pthread.h>
3   // Escalonamento de execução
4   #include <sched.h>
... ...
37  // Função principal
38  int main() {
... ...
49      // Iterando de 0 até N - 1
50      for(uint32_t i = 0; i < N; i++) {
51          // Finalizando threads
52          pthread_join(consumers[i], NULL);
53          pthread_join(producers[i], NULL);
54      }
55      // Retornando sucesso
56      return 0;
57  }
```

POSIX Threads

- Condições de corrida

```
$ gcc -Wall -g exemplo.c -o exemplo.elf -lpthread
$ ./exemplo.elf
Producer = 1
Producer = 1
Producer = 2
Producer = 3
Producer = 4
Producer = 5
Producer = 6
Producer = 7
Consumer = 0
Consumer = 7
Consumer = 4
Consumer = 0
Consumer = 2
Consumer = 1
Consumer = 0
Producer = 8
Producer = 1
Producer = 2
Producer = 2
Producer = 3
...
```

Demonstrações

Sincronizando por MUTEX

POSIX Threads

- Proteção por exclusão mútua (*mutex*)

```
1   // POSIX threads
2   #include <pthread.h>
3   // Escalonamento de execução
4   #include <sched.h>
... ...
9   // Instanciando mutex
10  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
11  // Thread consumidor
12  void* consumer(void* p) {
... ...
19          // Bloqueando região crítica
20          pthread_mutex_lock(&mutex);
21          // Imprimindo valor do parâmetro
22          printf("Consumer = %i\n", (*v > 0) ? (--(*v))
                : (*v));
23          // Liberando região crítica
24          pthread_mutex_unlock(&mutex);
... ...
```

POSIX Threads

- Proteção por exclusão mútua (*mutex*)

```
1   // POSIX threads
2   #include <pthread.h>
3   // Escalonamento de execução
4   #include <sched.h>
... ...
9   // Instanciando mutex
10  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
11  // Thread consumidor
12  void* consumer(void* p) {
... ...
19          // Bloqueando região crítica
20          pthread_mutex_lock(&mutex);
21          // Imprimindo valor do parâmetro
22          printf("Consumer = %i\n", (*v > 0) ? (--(*v))
                : (*v));
23          // Liberando região crítica
24          pthread_mutex_unlock(&mutex);
... ...
```

POSIX Threads

- Proteção por exclusão mútua (*mutex*)

```
1   // POSIX threads
2   #include <pthread.h>
3   // Escalonamento de execução
4   #include <sched.h>
... ...
29  // Thread produtor
30  void* producer(void* p) {
... ...
37          // Bloqueando região crítica
38          pthread_mutex_lock(&mutex);
39          // Imprimindo valor do parâmetro
40          printf("Producer = %i\n", (*v < N) ? (++(*v))
                : (*v));
41          // Liberando região crítica
42          pthread_mutex_unlock(&mutex);
... ...
```

POSIX Threads

- Proteção por exclusão mútua (*mutex*)

```
1   // POSIX threads
2   #include <pthread.h>
3   // Escalonamento de execução
4   #include <sched.h>
... ...
29  // Thread produtor
30  void* producer(void* p) {
... ...
37          // Bloqueando região crítica
38          pthread_mutex_lock(&mutex);
39          // Imprimindo valor do parâmetro
40          printf("Producer = %i\n", (*v < N) ? (++(*v))
                : (*v));
41          // Liberando região crítica
42          pthread_mutex_unlock(&mutex);
... ...
```

POSIX Threads

- Proteção por exclusão mútua (*mutex*)

```
$ gcc -Wall -g exemplo.c -o exemplo.elf -lpthread
$ ./exemplo.elf
Producer = 1
Producer = 2
Producer = 3
Producer = 4
Producer = 5
Producer = 6
Producer = 7
Producer = 8
Consumer = 7
Producer = 8
Consumer = 7
Consumer = 6
Consumer = 5
Consumer = 4
Producer = 5
Producer = 6
Consumer = 5
Consumer = 4
Consumer = 3
Producer = 4
...
```

Demonstrações

Threads com execução condicional com e sem temporização

POSIX Threads

- Comunicação por condição sem temporização

```
1   // POSIX threads
2   #include <pthread.h>
3   // Instanciando condição
4   pthread_cond_t  condition = PTHREAD_COND_INITIALIZER;
5   // Instanciando mutex
6   pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
... ...
```

POSIX Threads

- Comunicação por condição sem temporização

```
1   // POSIX threads
2   #include <pthread.h>
…   ...
6   void* receiver(void* p) {
7       // Esperando por sinalização de condição
8       pthread_cond_wait(&condition, &mutex);
…   ...
11  }
12  void* sender(void* p) {
13      // Bloqueando região crítica
14      pthread_mutex_lock(&mutex);
15      // Enviando sinal de condição
16      pthread_cond_signal(&condition);
17      // Liberando região crítica
18      pthread_mutex_unlock(&mutex);
…   ...
21  }
```

POSIX Threads

- Comunicação por condição sem temporização

```
1   // POSIX threads
2   #include <pthread.h>
... ...
6   void* receiver(void* p) {
7       // Esperando por sinalização de condição
8       pthread_cond_wait(&condition, &mutex);
... ...
11  }
12  void* sender(void* p) {
13      // Bloqueando região crítica
14      pthread_mutex_lock(&mutex);
15      // Enviando sinal de condição
16      pthread_cond_signal(&condition);
17      // Liberando região crítica
18      pthread_mutex_unlock(&mutex);
... ...
21  }
```

Apenas uma *thread* é ativada pela condição

POSIX Threads

- Comunicação por condição sem temporização

```
1   // POSIX threads
2   #include <pthread.h>
... ...
6   void* receiver(void* p) {
7       // Esperando por sinalização de condição
8       pthread_cond_wait(&condition, &mutex);
... ...
11  }
12  void* sender(void* p) {
13      // Bloqueando região crítica
14      pthread_mutex_lock(&mutex);
15      // Enviando sinal de condição
16      pthread_cond_signal(&condition);
17      // Liberando região crítica
18      pthread_mutex_unlock(&mutex);
... ...
21  }
```

Apenas uma *thread* é ativada pela condição

POSIX Threads

- Comunicação por condição sem temporização

```
1  // POSIX threads
2  #include <pthread.h>
...
6  void* receiver(void* p) {
7      // Esperando por sinalização de condição
8      pthread_cond_wait(&condition, &mutex);
...
11 }
12 void* sender(void* p) {
13     // Bloqueando região crítica
14     pthread_mutex_lock(&mutex);
15     // Enviando sinal de condição
16     pthread_cond_broadcast(&condition);
17     // Liberando região crítica
18     pthread_mutex_unlock(&mutex);
...
21 }
```

POSIX Threads

- Comunicação por condição sem temporização

```
1   // POSIX threads
2   #include <pthread.h>
... ...
6   void* receiver(void* p) {
7       // Esperando por sinalização de condição
8       pthread_cond_wait(&condition, &mutex);
... ...
11  }
12  void* sender(void* p) {
13      // Bloqueando região crítica
14      pthread_mutex_lock(&mutex);
15      // Enviando sinal de condição
16      pthread_cond_broadcast(&condition);
17      // Liberando região crítica
18      pthread_mutex_unlock(&mutex);
... ...
21  }
```

Todas as *threads* são ativadas pela condição

POSIX Threads

- Comunicação por condição sem temporização

```
1  // POSIX threads
2  #include <pthread.h>
...
6  void* receiver(void* p) {
7      // Esperando por sinalização de condição
8      pthread_cond_wait(&condition, &mutex);
...
11 }
12 void* sender(void* p) {
13     // Bloqueando região crítica
14     pthread_mutex_lock(&mutex);
15     // Enviando sinal de condição
16     pthread_cond_broadcast(&condition);
17     // Liberando região crítica
18     pthread_mutex_unlock(&mutex);
...
21 }
```

Todas as *threads* são ativadas pela condição

POSIX Threads

- Comunicação por condição com temporização

```
1   // POSIX threads
2   #include <pthread.h>
... ...
6   void* receiver(void* p) {
7       // Estrutura de tempo de atraso
8       struct timespec delay;
9       // Obtendo tempo atual
10      clock_gettime(CLOCK_REALTIME, &delay);
11      // Adicionando 3 segundos
12      delay.tv_sec += 3;
13      // Espera por condição ou tempo de 3 segundos
14      pthread_cond_timedwait(&condition, &mutex,
            &delay);
15      // Retornando null
16      return NULL;
17  }
... ...
```

POSIX Threads

- Comunicação por condição com temporização

```
1   // POSIX threads
2   #include <pthread.h>
... ...
6   void* receiver(void* p) {
7       // Estrutura de tempo de atraso
8       struct timespec delay;
9       // Obtendo tempo atual
10      clock_gettime(CLOCK_REALTIME, &delay);
11      // Adicionando 3 segundos
12      delay.tv_sec += 3;
13      // Espera por condição ou tempo de 3 segundos
14      pthread_cond_timedwait(&condition, &mutex,
            &delay);
15      // Retornando null
16      return NULL;
17  }
... ...
```

A *thread* espera pela condição com *timeout*

Hora trabalho da semana.

Usando threads para dividir o trabalho de força bruta

Exercício

- Implemente um algoritmo para quebrar senhas armazenadas em *hash* de 64 bits, utilizando uma estratégia de *multithreading* para paralelização
  - Formato do arquivo de entrada
    $\#Quantidade\ de\ contas\ (n)$
    $Login_0 : Hash64_0$
    $\vdots$
    $Login_{n-1} : Hash64_{n-1}$

```
1 3
2 bruno:6249503ae72cf8d3
3 ihs:f88e8f310f6e1357
4 facil:9c348e2b9426caca
```

Exercício

- Implemente um algoritmo para quebrar senhas armazenadas em *hash* de 64 bits, utilizando uma estratégia de *multithreading* para paralelização
  - Formato do arquivo de entrada
    $\#Quantidade\ de\ contas\ (n)$
    $Login_0 : Hash64_0$
    $\vdots$
    $Login_{n-1} : Hash64_{n-1}$

```
1 3
2 bruno:6249503ae72cf8d3
3 ihs:f88e8f310f6e1357
4 facil:9c348e2b9426caca
```

Os nomes dos usuários são compostos por letras minúsculas, com tamanhos entre 3 e 8 caracteres

Exercício

- Implemente um algoritmo para quebrar senhas armazenadas em *hash* de 64 bits, utilizando uma estratégia de *multithreading* para paralelização
  - Procedimento de *hash* MAU-64

```
1  // Macro rand8
2  #define rand8 ((rand() >> 8) & 0b111)
3  // Função de hash MAU-64
4  void MAU_64(uint8_t* hash, const char* senha) {
5      // Declarando variáveis auxiliares
6      uint32_t i, n = strlen(senha), nr = 256, s = 0;
7      // Geração da semente a partir da senha (sdbm)
8      for(i = 0; i < n; i++)
9          s = senha[i] + (s << 6) + (s << 16) - s;
10     // Semente dos números pseudo-aleatórios
11     srand(s);
12     // Executando rodadas sobre os bytes do hash
13     for(i = 0; i < nr; i++)
14         hash[rand8] = hash[rand8] ^ rand();
15 }
```

Exercício

- Implemente um algoritmo para quebrar senhas armazenadas em *hash* de 64 bits, utilizando uma estratégia de *multithreading* para paralelização
  - Formato do arquivo de saída
    $Login_0 : Senha_0$
    $\vdots$
    $Login_{n-1} : Senha_{n-1}$

```
1 bruno:U42
2 ihs:1010
3 facil:ia
```

Exercício

- Implemente um algoritmo para quebrar senhas armazenadas em *hash* de 64 bits, utilizando uma estratégia de *multithreading* para paralelização
  - Formato do arquivo de saída

    $Login_0 : Senha_0$

    $\vdots$

    $Login_{n-1} : Senha_{n-1}$

```
1 bruno:U42
2 ihs:1010
3 facil:ia
```

As senhas possuem entre 2 e 4 caracteres que podem ser letras ou números

Hora-Trabalho da semana

Realize o exercício apresentado nos Slides 74 a 78, e monte um relatório com o desenvolvimento e seus comentários sobre a atividade.

Envie no AVA da disciplina.

Bons estudos

Dúvidas?

Na próxima aula...

Programação Paralela (OpenCL)

Não falte! 😉

Obrigado pela atenção

UNIVERSIDADE
FEDERAL DE
SERGIPE