U LISBOA | UNIVERSIDADE DE LISBOA

TP5. Apoio ao projeto 4: *Threads* e replicação passiva

Pedro Ferreira / Mário Calha

Departamento de Informática

Faculdade de Ciências da Universidade de Lisboa

Fonte da imagem: http://www.dsn.jhu.edu

Bibliografia

- [Stevens2004]
- [Kerrisk2010]

NOTA

Os acetatos que se seguem não substituem a bibliografia aqui referida, e deverão por isso ser vistos apenas como um complemento para o estudo da matéria.

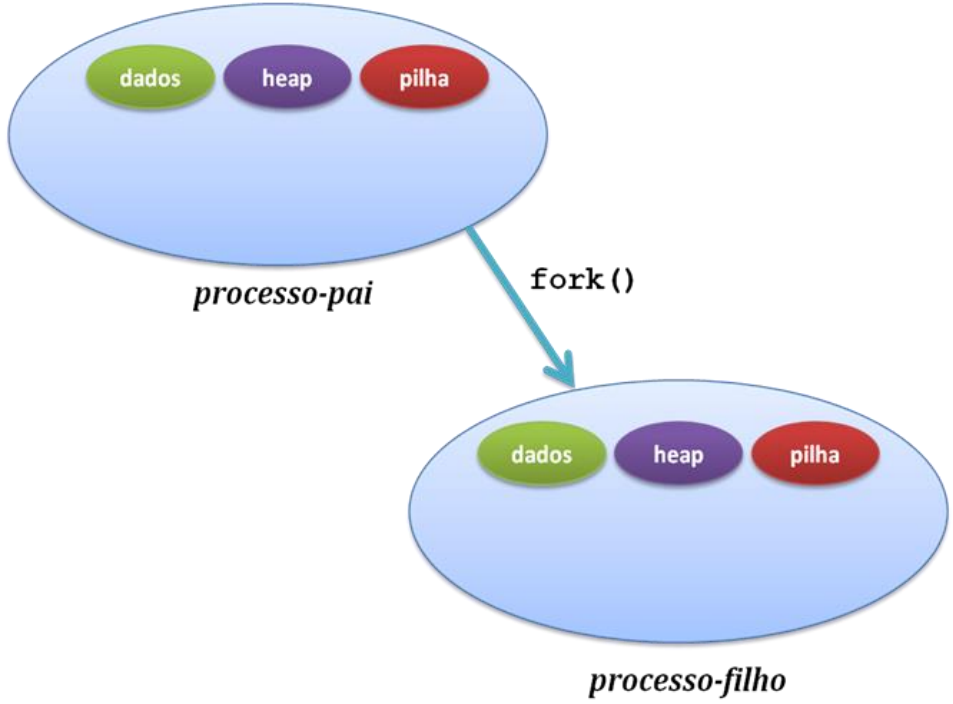
Revisitando a organização de um processo em memória

- Texto
  - instruções do programa
- Dados
  - variáveis **globais estáticas**;
- Heap
  - área onde os programa pode alocar memória (variáveis **globais**) de uma forma **dinâmica**
    - » e.g., usando malloc()
- Stack (pilha)
  - memória usada na chamada de funções, para passar parâmetros (de entrada e de saída) e para armazenar as variáveis temporárias usadas pela função.

Criação de novo processo: fork()

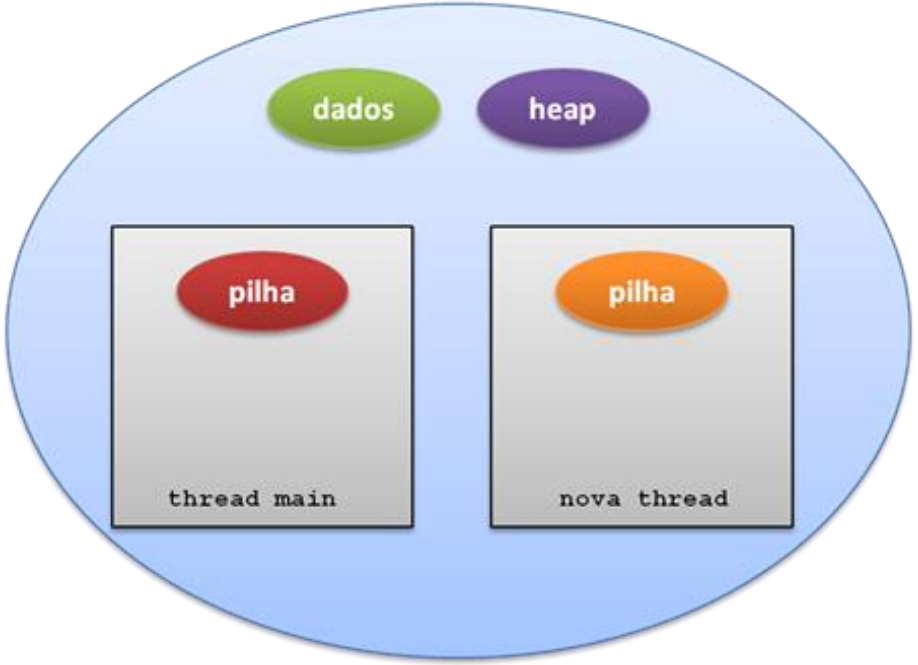
- Processo-filho é uma **cópia** do processo-pai.
  - todos os segmentos do processo (pilha, heap e dados) são **copiados** para o filho.

- Pai e o filho são **independentes**.
  - Qualquer dos dois pode alterar as variáveis definidas em qualquer destes segmentos sem afetar o outro processo.

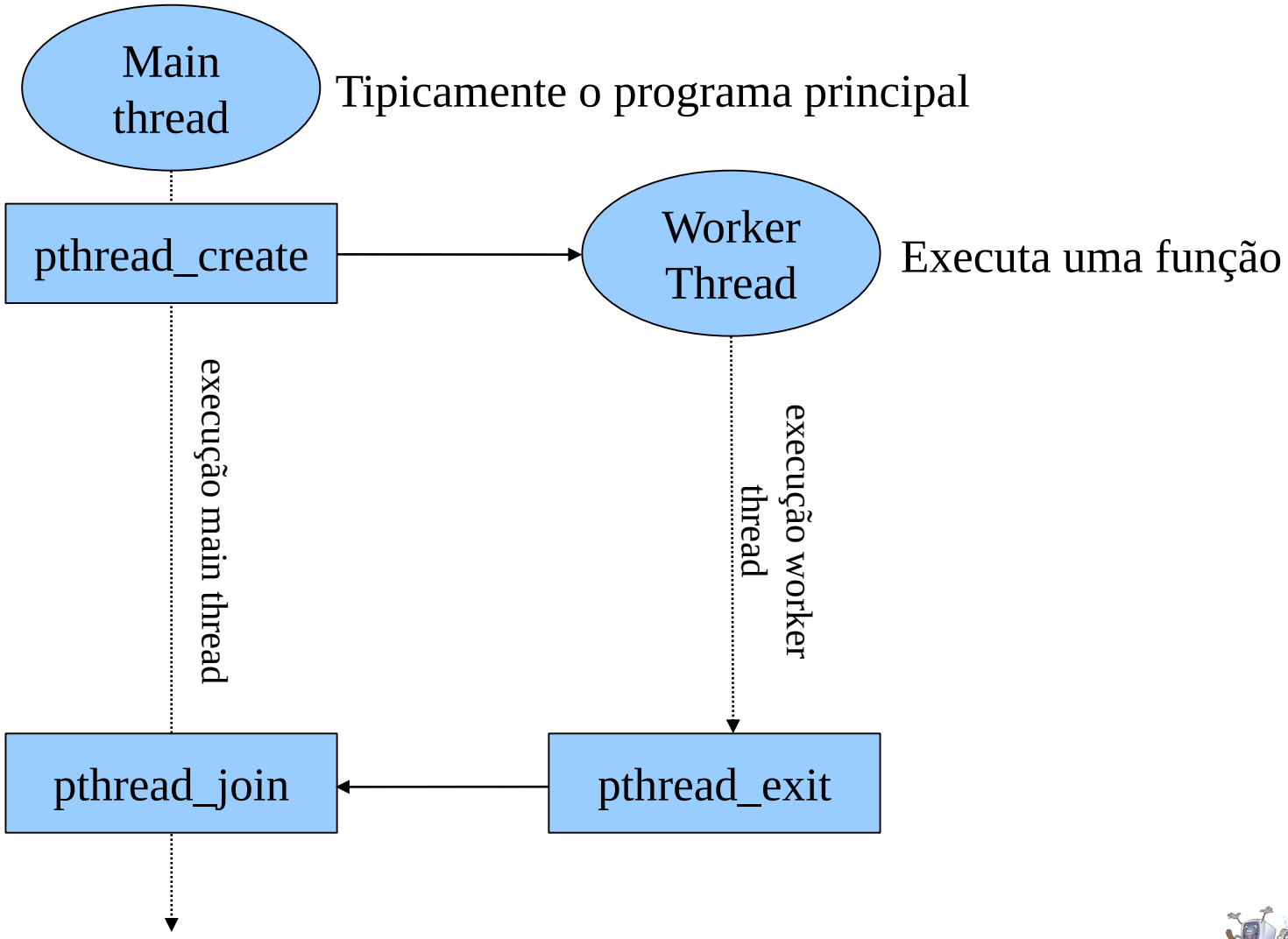


Threads

- As *threads* **partilham** o segmento de dados e o *heap*.
  - Cada *thread* mantém a sua própria pilha.
  - Vantagens:
    - » menos dispendioso e mais rápido do que criar processo
    - » facilidade de comunicação (através das variáveis globais partilhadas).
  - Desvantagem : necessário *sincronizar* o acesso às variáveis partilhadas.



Ciclo de vida duma thread (na API pthreads)



API *pthreads*: criação

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

Parâmetros:

thread: (*pthread_t* é um tipo numérico) contém o *id* da thread criada (é um número passado por referência que recebe um valor da função)

attr: lista de atributos da thread, se for a NULL usam-se os atributos padrão

func: apontador para a função a ser executada na thread. Esta função deve ter a seguinte assinatura:

```
void *<nome da func>(void* <nome do par>)
```

arg: argumento a ser passado para a função da *thread*

Retorna:

0 se não houver erro e o código do erro em caso de problemas

API *pthreads*: terminação e espera

Terminação da thread

- `return` da função que a thread estava a executar
- função `void` **`pthread_exit`**`(void *`**`status`**`);`
- thread pode retornar qualquer tipo de dados (endereço dos dados no `return` ou no `status`, usando tipo `void *`)

Esperar até a terminação de uma outra thread

(Nota: o valor de retorno fica retido em memória até que alguma outra *thread* execute pthread_join())

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

value_ptr: `void *` com status da terminação da thread, passado por referência que recebe o status.

Tornar thread "*detached*" (faz com que não seja possível retornar nada, isto é, não permite usar join a seguir)

```
int pthread_detach(pthread_t thread);
```

FACULDADE DE CIÊNCIAS UNIVERSIDADE DE LISBOA

Sincronização: mutexes

- Como as threads se executam no espaço de endereçamento do processo, normalmente utilizam-se *estruturas de dados partilhadas* para facilitar a comunicação e cooperação entre as threads
  - As threads têm por isso de ser sincronizadas para garantir o correcto acesso aos dados partilhados

- Como? Usando trincos (ou *MUTEX - MUTual EXclusion* ou *locks*)
  - utilizados para evitar a execução concorrente de um conjunto de instruções
  - tem dois estados: LOCK e UNLOCK

  Exemplo:

```
pthread_mutex_init(&mutex,NULL); /* criação do mutex */
pthread_mutex_lock(mutex);
/* aceder a uma estrutura de dados partilhada */
pthread_mutex_unlock(mutex);
```

FACULDADE DE CIÊNCIAS UNIVERSIDADE DE LISBOA

Sincronização: variáveis de condição

- Variáveis de Condição (*Condition Variables*)
  - utilizadas para suspender uma thread até que um *predicado sobre dados partilhados* se torne verdadeiro
  - assim permitem que as threads comuniquem a outras threads que o estado de uma variável partilhada foi alterada
  - normalmente são associadas a um trinco
  - operações : *wait(), wait_timeout(), signal(), broadcast()*

- Exemplo:

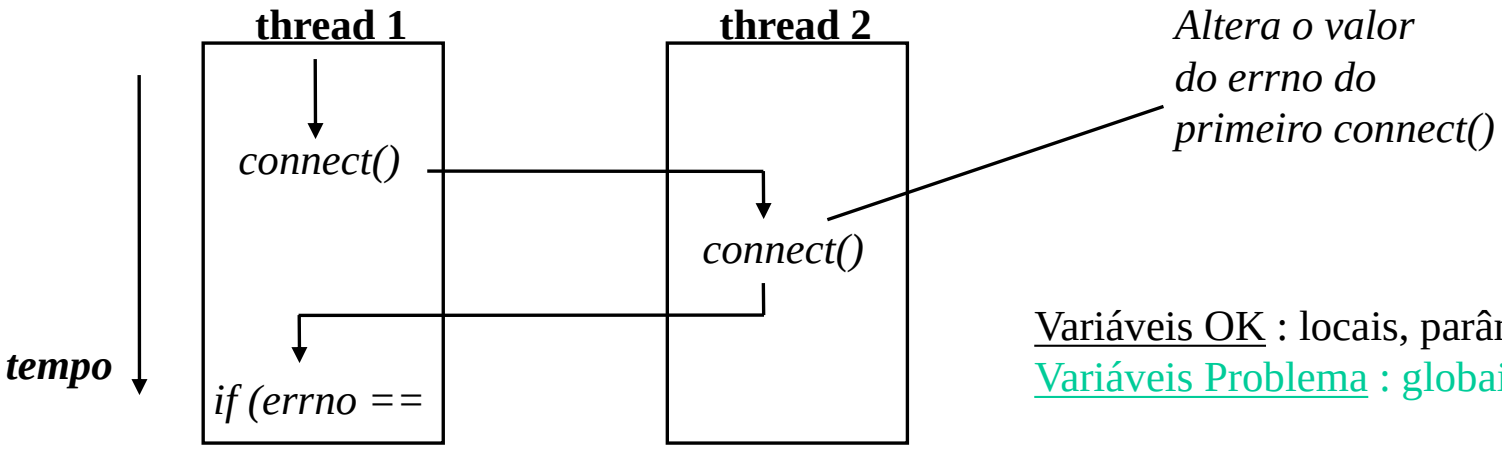
*(algures na thread **X**)*

```
pthread_mutex_lock(mut);
while (x <= y)
  pthread_cond_wait(cond, mut);
/* utilizar x e y */
pthread_mutex_unlock(mut);
```

*(algures na thread **Y**)*

```
pthread_mutex_lock(mut);
/* alterar x e y */
if (x > y)
      pthread_mutex_broadcast(cond);
pthread_mutex_unlock(mut);
```

Exemplo de Problemas: Variáveis Globais



Variáveis OK : locais, parâmetros
Variáveis Problema : globais, static

- Soluções
- 1 - proibir a utilização de variáveis que causam problemas (e.g., globais)
- 2 - cada thread tem uma cópia privada das variáveis usado pelas *pthreads* (ex., *errno* no Redhat Linux). É difícil de se conseguir automaticamente no caso geral
- 3 - utilizar um conjunto de funções de biblioteca que faz controlo de concorrência
- *create_global("bufptr")* – cria uma "variável global"
- *set_global("bufptr", &buf)* – altera o valor de uma "variável global"
- *bufptr=read_global("bufptr")* – lê o valor de uma "variável global"

API *pthreads*: manipulação de *locks (mutexes)*

Inicializar um *mutex*:

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

Parâmetros:

mutex: endereço do *mutex*

mutexattr: lista de atributos do *mutex* (a NULL usam-se os atributos padrão)

Retorna:

0 se não houver erro e o código do erro em caso de problema

Eliminar um *mutex*:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Parâmetro: endereço do *mutex*

Retorna: 0 se não houver erro e o código do erro em caso de problemas

FACULDADE DE CIÊNCIAS UNIVERSIDADE DE LISBOA

API *pthreads*: Manipulação de *locks (mutexes)* (II)

Trancar um *mutex*

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

Trancar se disponível (retorna EBUSY se o *mutex* se encontra trancado)

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

Destrancar um mutex

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

Parâmetros:

mutex: endereço do mutex

etornam:

0 se não houver erro e o código do erro em caso de problemas

API *pthreads*: manipulação de variáveis condicionais

Inicializar uma variável condicional

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

Parâmetros:

cond: endereço da variável condicional

mutexattr: lista de atributos da condição (a NULL usam-se os atributos padrão)

- Retorna:

0 se não houver erro e o código do erro em caso de problemas

Eliminar uma variável condicional

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

FACULDADE DE CIÊNCIAS UNIVERSIDADE DE LISBOA

API *pthreads*: Manipulação de variáveis condicionais (II)

Esperar numa variável condicional:

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

Função destranca *mutex* e espera até que a condição seja sinalizada, depois tranca novamente e o *mutex* e desbloqueia

Avisa **uma thread bloqueada** que esteja à espera numa variável condicional:

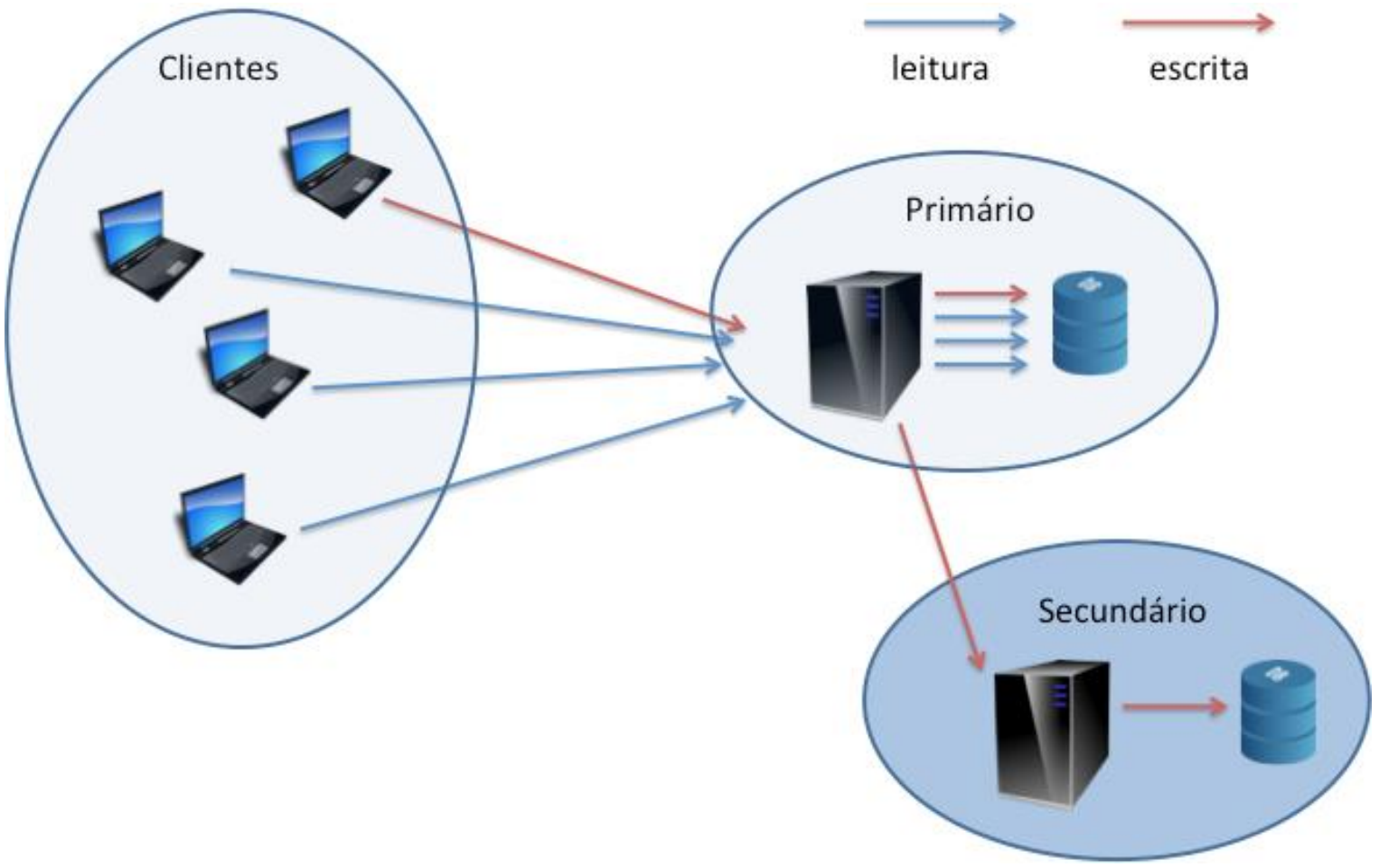
```
int pthread_cond_signal(pthread_cond_t *cond);
```

Avisa **todas as threads bloqueadas** que estejam à espera numa variável condicional:

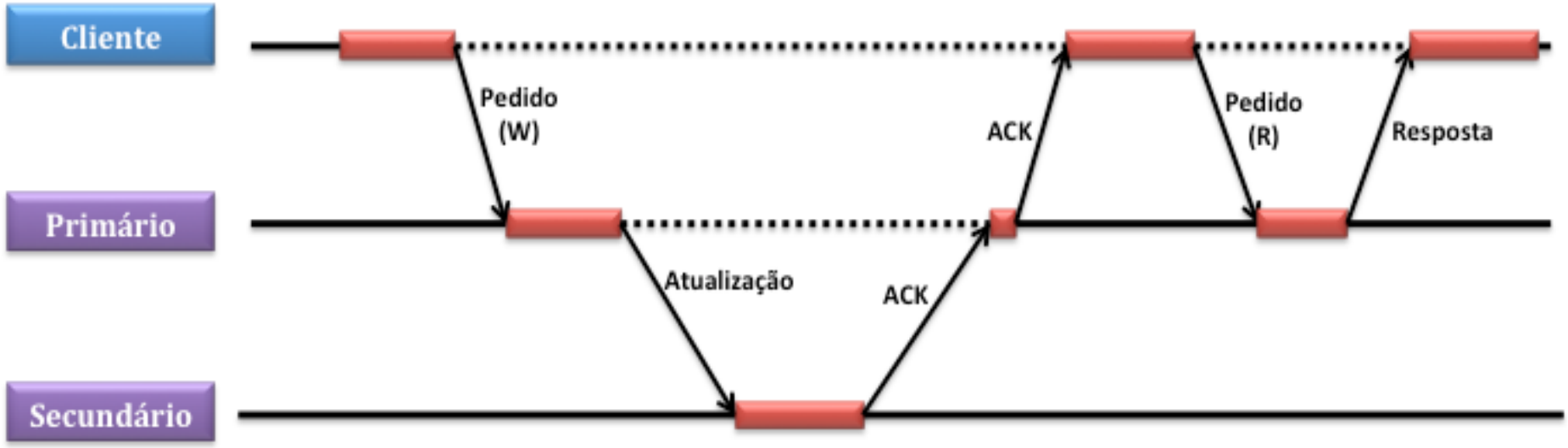
```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

FACULDADE DE CIÊNCIAS UNIVERSIDADE DE LISBOA

Objetivo: sistema de replicação passiva com primário



Passos de comunicação



Funcionamento base (13 valores)

1. Servidor replicado: primário e secundário.
   a. Nota: servidores à escuta do teclado (“print”)
2. Nas leituras o primário executa operação na sua tabela e envia resposta.
3. Nas escritas:
   a. o primário atualiza a sua tabela e
   b. faz o pedido de atualização ao secundário
   c. Nota: usar segunda thread no primário (cuidados com sincronização e eficiência)
4. Primário recebe OK do secundário e OK da sua escrita, manda OK ao cliente.
   a. Se der erro na escrita do primário este reporta o erro ao cliente.
   b. Não OK do secundário: marca-o “DOWN” e trabalha sozinho
      i. Quando secundário acorda pede atualização de estado (novos pedidos não processados) e passa a “UP”

FACULDADE DE CIÊNCIAS UNIVERSIDADE DE LISBOA

Funcionamento com falha do primário (7 valores)

1. Cliente deteta falha do primário
   a. Faz pedido para o secundário.
   b. Se está “morto”, serviço em baixo. O cliente tenta TIMEOUT segundos depois (primeiro no primário, depois no secundário), e caso persista o erro desiste.
   c. Se está ativo, secundário responde e passa a ser primário.

2. Antigo primário “acorda”, contacta novo primário, atualiza estado e passa a secundário (novos pedidos não processados, de novo).

Referências

- [Stevens2004]
  - W. R. Stevens, B. Fenner, A.M. Rudoff, *Unix Network Programming, The Sockets Networking API*, Volume 1, 3rd Edition, Addison Wesley, 2004
- [Kerrisk2010]
  - M. Kerrisk, “The Linux Programming Interface, A Linux and UNIX System Programming Handbook”, no starch press, 2010