Sincronização de Sistemas Concorrentes
Parte II

Patrício Domingues

SINCRONIZAÇÃO "AUTOMÁTICA"

Monitores – caso de estudo em Java

Ainda sobre semáforos...

- ✓ Os semáforos são convenientes, mas é...
  - Muito fácil cometer um erro
- ✓ Exemplos
  - É fácil esquecer um “wait()” ou um “post()” …
  - É fácil (erradamente) inverter a ordem de chamada de semáforos em processos cooperativos…
  - É difícil certificar-se da correcção do código/solução quando estão vários semáforos envolvidos
- ✓ Idealmente, são soluções automáticas
  - Na programação, o ser humano é quase sempre o “elo mais fraco”...

Monitor (programação concorrente) # 1

- ✓ Objeto apto a ser empregue por mais do que uma *thread*
  - *Monitor = mutex + variável de condição*

- ✓ Os métodos de um objeto monitor são executados com exclusão mútua
  - Num dado instante, apenas uma thread pode estar a executar um (e um só) dos métodos do objeto
    - A *thread* está na posse do monitor
  - As restantes *threads* estão bloqueadas

Continuação >>

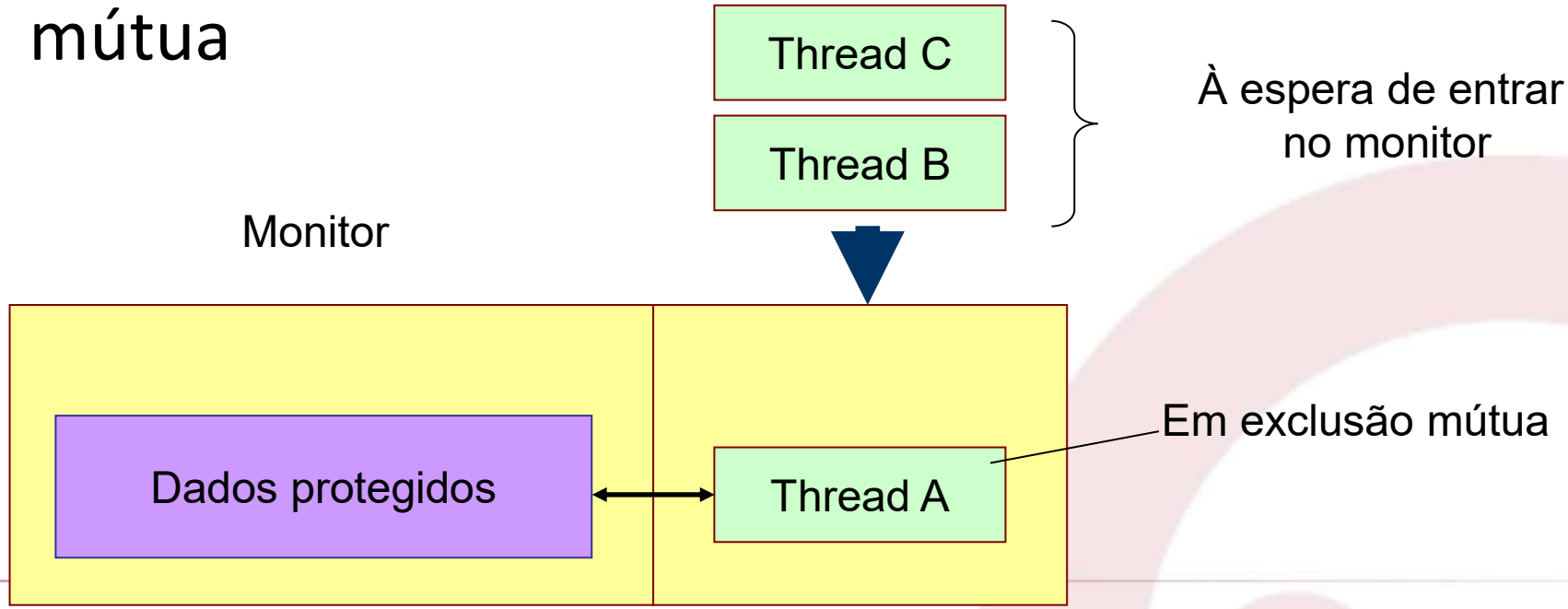
Monitor (programação concorrente) # 2

- ✓ Um monitor faculta ainda a possibilidade de uma *thread* abdicar temporariamente do acesso exclusivo ao objeto
  - para aguardar que uma determinada condição se verifique
- ✓ Um monitor permite ainda que uma thread assinale a outras threads que uma determinada condição passou a ser verdadeira

- ✓ Conclusão
  - O monitor simplifica a programação concorrente nos ambientes orientados para os objetos

Monitor (programação concorrente) # 3

- ✓ Um "monitor" é uma abstração associada a um objeto que permite que num dado instante, apenas uma *thread* possa estar em execução
  - Dentro do monitor, a thread executa em exclusão mútua



Monitor em Java (1)

- ✓ No Java, qualquer objeto pode ser empregue sob a supervisão de um monitor
  - Os métodos que necessitem de exclusão mútua devem ser declarados **synchronized**

```
public synchronized void add(int value)
{ this.count += value; }
```

  - Blocos de código também podem ser marcados como **synchronized**

```
public void add(int value){
    synchronized(this){
        this.count += value; // exclusão mútua
    }
}
```

Monitor em Java (2)


```
import java.util.*;
public class Buffer{
        //----------------------
        // Dados protegidos via monitor
        private final static int MAX_SIZE=10;
        private LinkedList<Integer> elements;
        private int totalElements;
        //----------------------
        public Buffer(){
                elements = new LinkedList<Integer>();
                totalElements = 0;
        }
        // Apenas uma thread pode estar neste método
        public synchronized void putValue(int e){
                        //...
        }
        public synchronized int getValue(){
                        //...
        }
}
```

Monitor em Java (3)

- ✓ Três primitivas associadas aos monitores
  - wait()
    - Suspende a execução da thread corrente, libertando de imediato o monitor
    - A thread é colocada na "lista de threads bloqueadas", aguardando por notificação que algo mudou
  - notify()
    - Informa uma das "threads bloqueadas" da ocorrência da "condição" que aguardava
      - Essa thread é colocada na "lista de threads prontas a executar"
      - Essa thread só poderá ser executada após que a thread que chamou "notify()" tenha saído do monitor
  - notifyAll() (variante do notify)
    - Notifica todas as threads para verificarem a condição que aguardam

Monitor em Java (4)

- ✓ Nota importante
  - As primitivas wait() e notify() **não** são como as primitivas wait() e post() dos semáforos
    - Não existe memória, i.e. se notify é chamado quando não existem threads à espera, o efeito do notify é perdido
    - Num semáforo, o post() leva a que o contador associado ao semáforo fique com mais uma unidade, independentemente de existirem ou não processos à espera do semáforo
  - Num semáforo, o post() faz sempre qualquer coisa
  - Num ambiente thread, o notify() nem sempre produz uma ação

JAVA -- putValue() e getValue()

```
public synchronized void putValue(int e)
        throws InterruptedException{
                while(totalElements == MAX_SIZE)
                        wait();
                elements.add(e);
                ++totalElements;
                notifyAll();
}
public synchronized int getValue()
        throws InterruptedException{
                while(totalElements == 0)
                        wait();
                int e = elements.remove();
                --totalElements;
                notifyAll();
}
```

Continua >>

Programa principal


```
public class ProducerConsumer{
      public static void main(String[] args){
            Buffer boundedBuffer = new Buffer();
            Consumer cons = new Consumer(boundedBuffer);
            Producer prod = new Producer(boundedBuffer);
      }
}
```

Continua (classe Producer) >>

Classe Producer


```
// Extends the Thread class
class Producer extends Thread{
        private Buffer buf;
        public Producer(Buffer buf){
                this.buf = buf;
                start();
        }
        public void run(){
                int total=0;
                while(true){
                        try{
                                System.out.println(
                                        "[Producer] putting "+total);
                                buf.putValue(total);
                                ++total;
                        }catch(InterruptedException e){}
                }
        }
}
```

Continua (classe Consumer) >>

Classe Consumer


```
// Extends the Thread class
class Consumer extends Thread{
        private Buffer buf;
        public Consumer(Buffer buf){
                this.buf = buf;
                start();
        }
        public void run(){
                while(true){
                        try{

        System.out.println("[Consumer]:"+buf.getValue());
                                Thread.sleep(1000);
                        }catch(InterruptedException e){}
                }
        }
```

Porquê notifyAll() e não apenas notify()? >>

Ainda sobre monitores…

```
public synchronized void putValue(int e)
        throws InterruptedException{
                while(totalElements == MAX_SIZE)
                        wait();
                elements.add(e);
                ++totalElements;
                notifyAll();
}
```

- ✓ Porquê “notifyAll()” e não “notify()”?
  - **Resposta:** podem estar vários Consumers à espera
- ✓ Porquê um ciclo *while* e não um *if*?
  - **Resposta:** Uma thread pode acordar por ação do **notifyAll()** mas quando pretende executar, já outra thread (que acordou também) pode estar no “monitor”!

VARIÁVEIS DE CONDIÇÃO

E nas *pthreads*, há *monitores*?

- ✓ As pthreads não incluem monitores, mas suportam "**variáveis de condição**"
  - Designação anglo-saxónica: *condition variables*
- ✓ As variáveis de condição permitem:
  - Suspender uma thread até que uma determinada condição esteja satisfeita

    OU
  - Notificar uma thread que determinada condição se verificou
- ✓ A condição pode ser o que se quiser, por exemplo:
  - Variável inteira a atingir um pré-determinado valor
  - Existência de um ficheiro
  - Etc.

Primitivas – *pthreads + condition variables*

```
// Creates a new initialized condition variable
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

// Explicitly initializes a condition variable
int   pthread_cond_init(pthread_cond_t *, const pthread_condattr_t *);

// Signals a condition variable -- only one thread (if any) is notified
int   pthread_cond_signal(pthread_cond_t *);

// Signals a condition variable -- all waiting threads are notified
int   pthread_cond_broadcast(pthread_cond_t *);

// Waits on a condition variable. 'mutex' is released while waiting
// and automatically reacquired when a thread is unblocked
int   pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

// Same as pthread_cond_wait() but allows for a timeout
int   pthread_cond_timedwait(pthread_cond_t *cond,
        pthread_mutex_t *mutex, const struct timespec *abstime);

// Release a condition variable
int   pthread_cond_destroy(pthread_cond_t *);
```

Variáveis de condição (2)

- ✓ Regra importante
  - Uma variável de condição requer sempre um *mutex*
    - A variável de condição tem que ser verificada em estado de exclusão mútua, com o mutex *locked*
- ✓ Exemplo

```
// Thread A
pthread_mutex_lock(&mutex);
while(condition() != true){
    pthread_cond_wait(&cond_var,
                      &mutex);
}
// After this step, we know that
// the condition is true and we
// are in mutual exclusion...
pthread_mutex_unlock(&mutex);
```

```
// Thread B
pthread_mutex_lock(&mutex);

// Do something that may
// make condition() change:
// notify the other thread
// ("Thread A")to recheck it.
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

Explicação >>

Variáveis de condição (3)

- Regra importante
  - Uma variável de condição requer sempre um *mutex*
    - A variável de condição tem que ser verificada em estado de exclusão mútua, com o mutex *locked*

```
// Thread A
// Point number 1                                   1
pthread_mutex_lock(&mutex);
while(condition() != true){                         2
    // Point number 2
    pthread_cond_wait(&cond_var,&mutex);
}
// After this step, we know that
// the condition is true and we
// are in mutual exclusion...                       3
pthread_mutex_unlock(&mutex);

// Thread B
pthread_mutex_lock(&mutex);
// Do something that may
// make condition() change:
// notify the other thread
// ("Thread A")to recheck it.
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

- Ponto 1: A thread testa a condição em exclusão mútua (mutex)
- Ponto 2: Se a condição for falsa, pthread_cond_wait() liberta o mutex e aguarda até que uma outra thread assinale que a condição pode ser novamente testada
- Ponto 3: Quando a condição é assinalada e o mutex está disponível, pthread_cond_wait() readquire o mutex e liberta a thread

Variáveis de condição (4)

- Regra importante
  - Uma variável de condição requer sempre um *mutex*
    - A variável de condição tem que ser verificada em estado de exclusão mútua, com o mutex *locked*

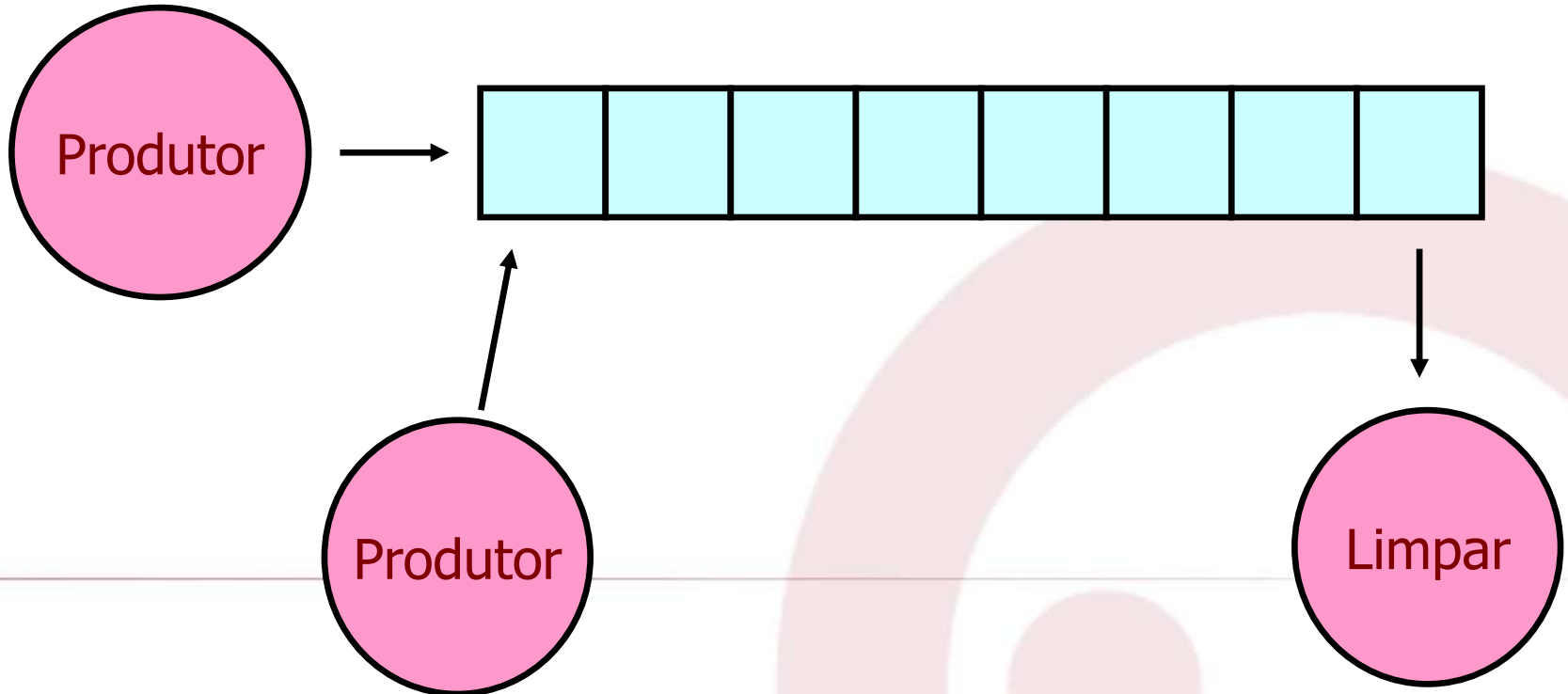
```
// Thread A
// Point number 1
pthread_mutex_lock(&mutex);
while(condition() != true){
    // Point number 2
    pthread_cond_wait(&cond_var,&mutex)
;
}
// After this step, we know that
// the condition is true and we
// are in mutual exclusion...
pthread_mutex_unlock(&mutex);

// Thread B
pthread_mutex_lock(&mutex);
// Do something that may
// make condition() change:
// notify the other thread
// ("Thread A")to recheck it.
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

- pthread_cond_signal() indica que apenas uma thread bloqueada deve testar a condição
  - Se não existir thread bloqueada o sinal é perdido…
- Se se pretender que todas as threads testem a condição, deve ser empregue pthread_cond_broadcast().
  - Dado a variável de condição estar associada a um mutex, cada thread irá testar a condição em condições de exclusão mútua

Buffer limitado – com variáveis de condição

- ✓ Considerando um *buffer* que pode conter até um máximo de N elementos.
  - Quando se encontra cheio, deve ser imediatamente esvaziado
  - Enquanto decorre o esvaziamento do *buffer,* o *buffer* não pode receber nenhum elemento



"buffer_limitado.c" – código do produtor (1)

```
void *producer(void *id){
      int my_id = *((int*)id);
      int i = my_id;

      while(1){
            pthread_mutex_lock(&mutex);
            // if it's full, notify someone to take care of it
            while(n_elements==N){
                  pthread_cond_signal(&is_full);
                  pthread_cond_wait(&go_on,&mutex);
            }

            // we have space and we're in mutual exclusion
            printf("[PRODUCER %3d] Writing %d into the buffer\n",
                  my_id, i);
            buffer[n_elements++] = i++;
            pthread_mutex_unlock(&mutex);
            sleep(1);
      }
       return NULL;
}
```

produtor

Variáveis de condição
pthread_cond_t **go_on**, **is_full**;

"buffer_limitado.c" – código do limpador (2)

"limpador"

```
void *cleaner(void* arg){
        while(1){
                pthread_mutex_lock(&mutex);
                // If it's not full, just wait
                while(n_elements != N){
                        pthread_cond_wait(&is_full,&mutex);
                }
                //It's full and we're in mutual exclusion
                printf("[CLEANER] Buffer ->");
                for(int i=0;i<N;i++){
                        printf("%d ", buffer[i]);
                }
                printf("\n");
                n_elements = 0;
                // Allows everyone waiting to check if they can
                pthread_cond_broadcast(&go_on);
                pthread_mutex_unlock(&mutex);
        }
        return NULL;
}
```

Variáveis de condição – regras

- ✓ Uma condição deve ser sempre verificada e assinalada dentro de uma secção protegida por um mutex
- ✓ A condição deve ser sempre testada num ciclo while, nunca num if! Ser desbloqueado de uma variável de condição, apenas significa que a condição deve ser novamente testada (e não que a condição se tornou verdadeira)
- ✓ Mesmo que condition() devolva true, durante a execução de Thread B, algo pode ocorrer no tempo que medeia entre a condição ser assinalada (pthread_cond_signal) e a thread A ser desbloqueada (e.g. uma outra thread altera a condição)

```
//== Thread A ==
pthread_mutex_lock(&mutex);
// Errado: usar while e não if!
if(condition() != true){
    pthread_cond_wait(&cond,&mutex);
}

// Critical zone
// ...
pthread_mutex_unlock(&mutex);

//== Thread B ==
pthread_mutex_lock(&mutex);

//...something that may make
// condition() to change
pthread_cond_signal(&cond);

pthread_mutex_unlock(&mutex);
```

LEMBRETE: usar "while" em vez de "if"!

PTHREAD_BARRIER

Sincronização “barreira” nas *POSIX Threads*

O que é uma Barreira (Barrier)?

- Um primitivo de sincronização que permite que várias *threads* aguardem que todas atinjam um determinado ponto na sua execução
  - Ponto barreira
- Permite que threads esperem umas pelas outras num ponto específico
  - Útil para coordenar processamento paralelo
- Todas as threads participantes ficam bloqueadas na barreira até que o número necessário (*count*) de *threads* chame a função.
  - Assim que a contagem é atingida, todas as *threads* em espera são libertadas simultaneamente.

Principais funções “barrier”

- `int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrierattr_t *attr, unsigned count);`
  - Inicializa a barreira *barrier*
  - 'count' especifica o número de threads necessárias para satisfazer a barreira.

- `int pthread_barrier_wait(pthread_barrier_t *barrier);`
  - Bloqueia a *thread* chamante até que o *count* especificado de *threads* tenha chamado esta função

- `int pthread_barrier_destroy(pthread_barrier_t *barrier);`
  - Destrói um objeto barreira, libertando quaisquer recursos que este detenha.
  - Não deve estar nenhuma threads bloqueadas na barreira

Exemplo Prático (Linguagem C)

```c
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 5
pthread_barrier_t barreira;
void *funcao_thread(void *arg) {
    long id = (long)arg;
    printf("Thread %ld: A fazer a primeira parte do trabalho...\n", id);
    sleep(1); // Simula trabalho
    // Todas as threads esperam neste ponto
    printf("Thread %ld: A aguardar na barreira.\n", id);
    pthread_barrier_wait(&barreira);
    // O código abaixo só é executado quando todas chegam
    printf("Thread %ld: Barreira alcançada! A continuar para a fase 2.\n", id);
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    // Inicializar a barreira para NUM_THREADS (5)
    pthread_barrier_init(&barreira, NULL, NUM_THREADS);
    for (long i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, funcao_thread, (void *)i);
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    // Destruir a barreira no final
    pthread_barrier_destroy(&barreira);
    return 0;
}
```

Exemplo pthread_barrier

```
Thread #2: A fazer a primeira parte do trabalho...
Thread #3: A fazer a primeira parte do trabalho...
Thread #3: A fazer a primeira parte do trabalho...
Thread #2: A aguardar na barreira.
Thread #3: A aguardar na barreira.
Thread #3: A aguardar na barreira.
Thread #3: Barreira alcançada! A continuar para a fase 2.
Thread #2: Barreira alcançada! A continuar para a fase 2.
Thread #3: Barreira alcançada! A continuar para a fase 2.
```

```c
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 3
pthread_barrier_t barreira;
void *funcao_thread(void *arg) {
        long id = *(long*)arg;
        printf("Thread #%ld: A fazer a primeira parte do trabalho...\n", id);
        sleep(1); // Simula trabalho
        // Todas as threads esperam neste ponto
        printf("Thread #%ld: A aguardar na barreira.\n", id);
        pthread_barrier_wait(&barreira);
        // O código abaixo só é executado quando todas chegam
        printf("Thread #%ld: Barreira alcançada! A continuar para a fase 2.\n", id);
        return NULL;
}
int main(void) {
        pthread_t threads[NUM_THREADS];
        // Inicializar a barreira para NUM_THREADS
        pthread_barrier_init(&barreira, NULL, NUM_THREADS);
        for (long i = 0; i < NUM_THREADS; i++) {
                pthread_create(&threads[i], NULL, funcao_thread, &i);
        }
        for (int i = 0; i < NUM_THREADS; i++) {
                pthread_join(threads[i], NULL);
        }
        // Destruir a barreira no final
        pthread_barrier_destroy(&barreira);
        return 0;
}
```

Implementação pthread_barrier

- Implementação assenta num mutex e numa variável de condição
- Estrutura "pthread_barrier_t"

```
typedef struct {
  pthread_mutex_t           mutex;
  pthread_cond_t            cond;
  int                       count;
  int                       tripCount;
} pthread_barrier_t;
```

```
int
pthread_barrier_wait (
  pthread_barrier_t           *barrier
) {
  pthread_mutex_lock(&barrier->mutex);
  ++(barrier->count);
  if (barrier->count >= barrier->tripCount) {
    barrier->count = 0;
    pthread_cond_broadcast(&barrier->cond);
    pthread_mutex_unlock(&barrier->mutex);
    return 1;
  } else {
    pthread_cond_wait(&barrier->cond, &(barrier->mutex));
    pthread_mutex_unlock(&barrier->mutex);
    return 0;
  }
} /* pthread_barrier_wait() */
```

Fonte: https://github.com/jonahharris/pthread-barrier/tree/master

DEADLOCK/LIVELOCK

Caso de estudo

DEADLOCK – SEGUNDO INTERCALAR ("LEAP")

Deadlock: "leap second" (#1)

- ✓ Leap second (segundo intercalar)
  - UTC - Universal Time Coordinated
  - Manter o *tempo universal* (TU/UTC) em sintonia com o globo terrestre
    - Uma rede de 450 relógios atómicos determina o Tempo Atómico Internacional (TAI)
      - » Para um relógio atómico, 1 dia é rigorosamente 86400 segundos
    - O planeta Terra regista (pequenas) variações na rotação
      - » i) atração gravítica da Lua e ii) alterações sazonais da atmosfera
      - » Tal pode criar diferenças entre o TAI e o UTC
    - O *International Earth Rotation and Reference Systems Service* (IERS) calcula a diferença entre o UTC e o ângulo de rotação do planeta Terra
    - Se diferença >= 0.9 segundos é aplicado uma correção, chamada de "leap second"
    - **UTC = TAI +/- "leap second"** – aplicado em 30 junho ou 31 dezembro

Deadlock: "leap second" (1)

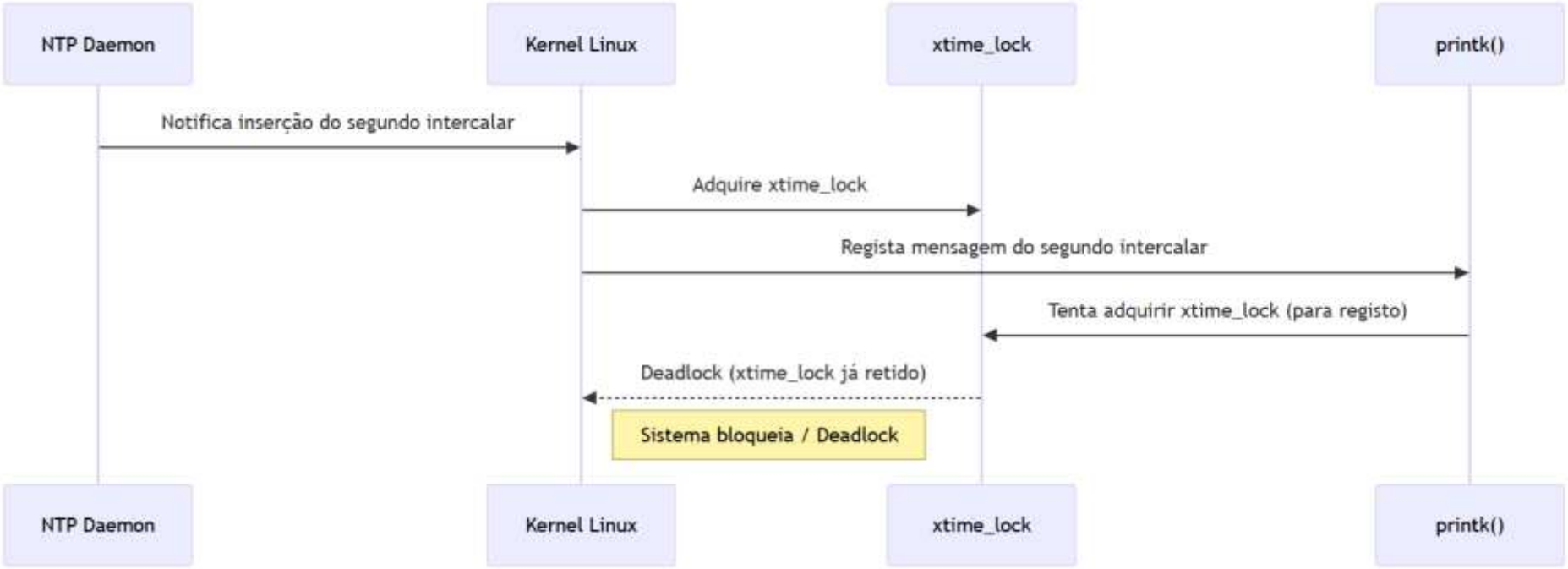
- ✓ Transição 31 dezembro 2008 para 1 janeiro 2009
  - Foi determinada a aplicação de um segundo de *leap*
  - Contudo, ocorreram sérios problemas com o segundo de "leap"
    - Sistemas Linux com kernel anteriores ao 2.6.9 entraram em *deadlock…*
    - Porquê?

Deadlock: "leap second" (2)

- Eventos que levaram ao *deadlock*
  - O código para aplicação do *segundo intercalar* é chamado pelo código de tratamento da interrupção de *timer* que tem o lock ***xtime_lock***
  - O código chama o `printk` (*printf do kernel*) para notificação sobre o "leap second"
  - O código do `printk` code tenta acordar o *klogd* e o escalonador tenta obter a hora corrente que tenta obter o ***xtime_lock****…*
    - *klogd: serviço de log do kernel cujas mensagens são precedidas da data/hora*
- Resultado: ***deadlock!***

Eventos “leap second” 2008

- ✓ Nota
  - NTP: *Network Time Protocol*
  - NTP Daemon: serviço para manter data/hora no sistema



LIVELOCK – SEGUNDO INTERCALAR ("LEAP")

Livelock: "leap second" (#3)

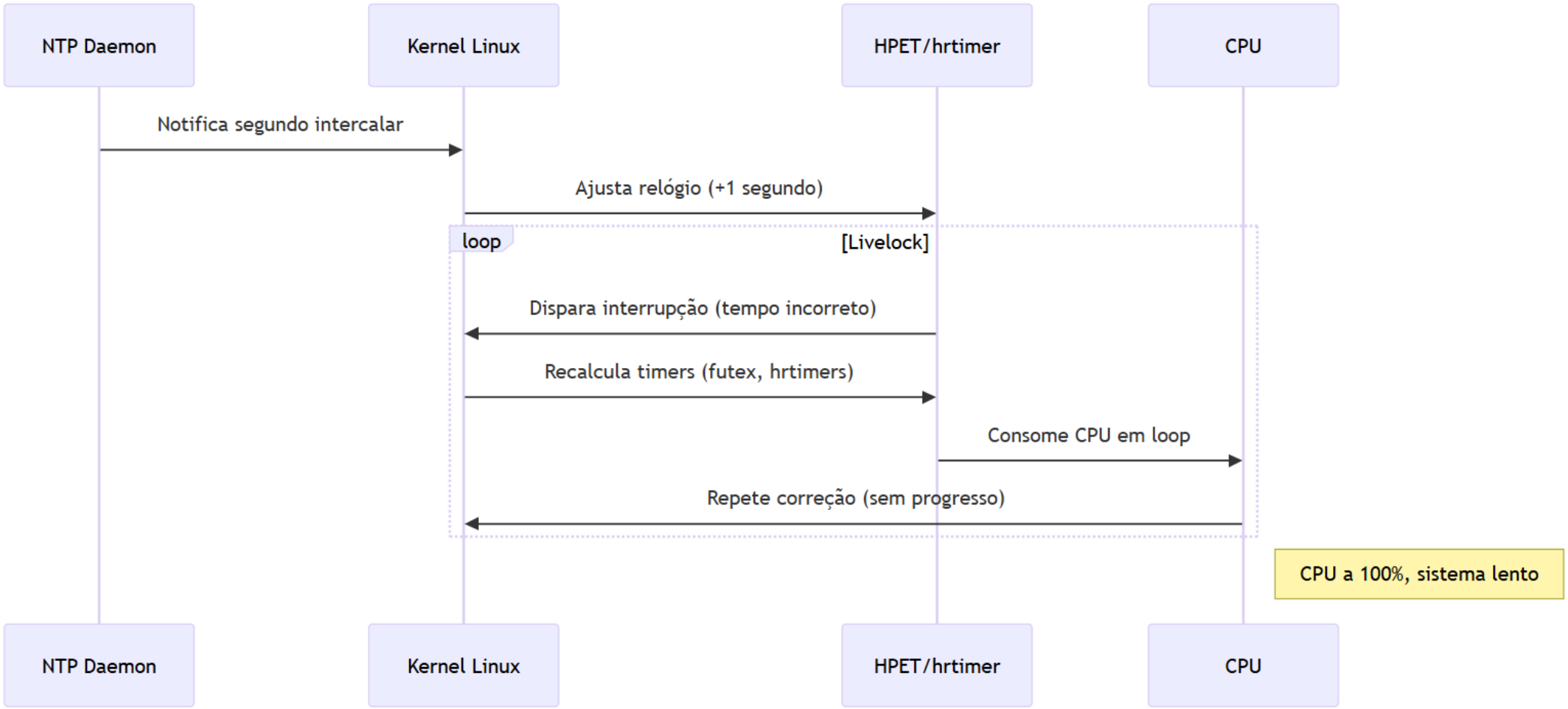
- ✓ Resolução: atualizar kernel para versão >= 2.6.9
  - Mas… passou-se para um livelock (CPU a 100%...)
    - Nos kernels 2.6.22 a 2.6.39
    - Sistemas com *timers* de elevada resolução *(hrtimers)* que suportam *High Precision Event Timers* (HPET)
    - Acréscimo de um segundo intercalar
      - Data/hora no kernel: 23h59:59 ➔ Segundo intercalar ➔ 23h59:**60**
    - Os temporizadores de alta precisão
      - Contam o tempo para a frente (lógico!)
      - Não estão preparados para lidar com um segundo intercalar

continua >>

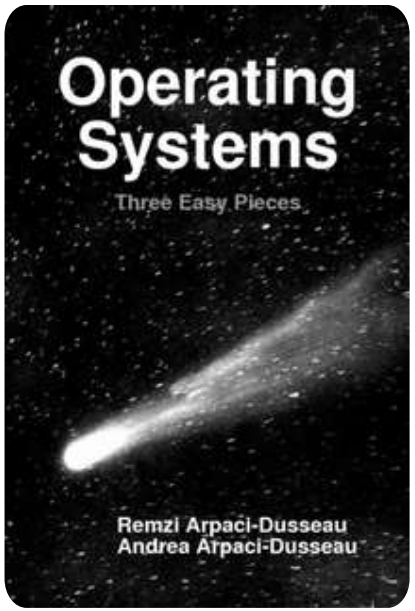
Livelock: "leap second" (#4)

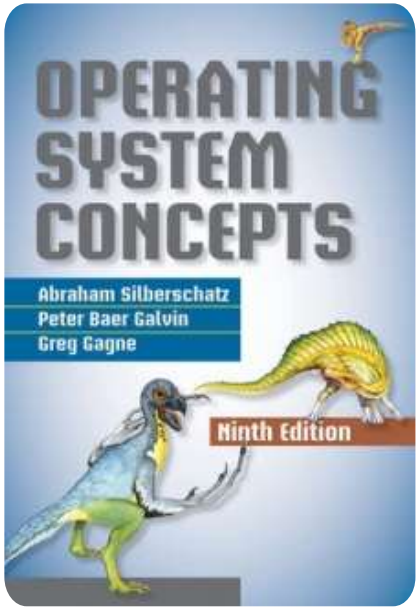
- ✓ Sequência de eventos que despoletam o livelock
  1. Um processo (qualquer) agenda uma tarefa para um futuro próximo (< 1 segundo) configurando o HRTimer
  2. O segundo intercalar é inserido. O relógio do kernel *salta para trás* em 1 segundo (23h59:**60**)
  3. O processo verifica o tempo e determina que o evento já devia ter ocorrido
  4. O processo agenda novamente a tarefa, mas o kernel continua a calcular uma hora no "passado" em relação ao evento original
  5. O processo fica preso num ciclo infinito de i) agendar e ii) "falhar", usando 100% do CPU.

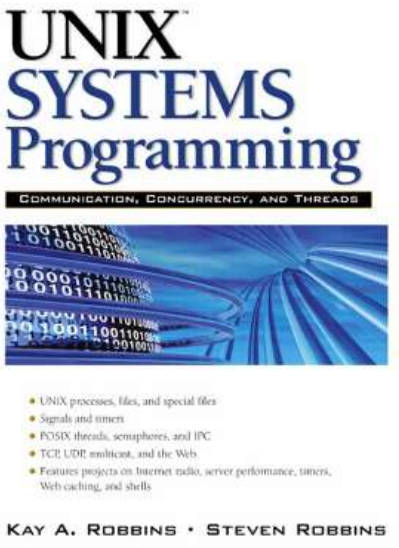
Livelock: “leap second” (#5)



Explain *deadlock…*







Referências

- [Silberschatz2012]
  - Capítulo 5: Sincronização Processos
- [Robbins2003]
  - Capítulo 12: POSIX Threads
  - Capítulo 13: Thread Synchronization
  - Capítulo 14: Critical Sections and Semaphores
- [Arpaci-Dusseau2018]
  - Part 2 – Concurrency - “Operating Systems: three easy pieces”, Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, 2018. www.ostep.org

Sincronização
Envolvimento do *hardware*

Buffer finito – possível solução (errada!)

✓Implementação com espera ativa que…falha!

```
void putItem(int e){
        // Busy waiting until there's a place
        while(nElments == BUFFER_SIZE){
                ;
        }
        // Put element
        buffer[writePos] = e;
        writePos = (writePos+1) % BUFFER_SIZE;
        ++nElments;
}
int getItem(void){
        // Busy waiting until there's something new,
        // that is nElments is changed from the outside
        while(nElments == 0){
                ;
        }
        // Get element
        int e = buffer[readPos];
        readPos = (readPos-1) % BUFFER_SIZE:
        --nElments;
        return e;
}
```

Explicação >>

Buffer finito - solução errada…

- ✓ Solução coloca CPU a 100% e não funciona
- ✓ ++nElements e --nElements podem não ser atómicos

```
void putItem(int e){
        // Busy waiting until there's a place
        while(nElments == BUFFER_SIZE){
                ;
        }
        // Put element
        buffer[writePos] = e;
        writePos = (writePos+1) % BUFFER_SIZE;
        ++nElments;
}
int getItem(void){
        // Busy waiting until there's something new,
        // that is nElments is changed from the outside
        while(nElments == 0){
                ;
        }
        // Get element
        int e = buffer[readPos];
        readPos = (readPos-1) % BUFFER_SIZE:
        --nElments;
        return e;
      }
```

Pode não ser atómico!

Pode não ser atómico!

Explicação >>

Problemas com **++nElements;**

- ✓ O compilador pode gerar o seguinte código assembler

```
LD    R1, @nElements
ADD   R1, R1, 1
SW    @nElements, R1
```

- ✓ A execução pode ser interrompida a meio (processo é retirado do CPU)
  - Código não atómico
  - Dependendo do encadeamento de "`putItem()`" e de "`getItem()`" (determinado pelo escalonamento do SO), o valor final pode ser -1, correcto ou +1…

Possível solução (#1)

- Possível solução: **desativar (momentaneamente)** as interrupções

```
CLI
LD    R1, @nElements
ADD   R1, R1, 1
SW    @nElements, R1
STI
```

- **CLI**: *clear interrupt* – inibe interrupções
- **STI**: *set interrupt* – re-ativa interrupções
- Pergunta...
  - porque é que a desactivação de interrupções resolve o problema?

Resposta >>

Possível solução (#2)

- ✓ **Pergunta**
  - Porque é que a desativação de interrupções resolve o problema?
- ✓ **Resposta**
  - O escalonamento de processos é feito pelo escalonador do sistema operativo
    - Para que haja troca de processo é necessário que o escalonador do sistema operativo seja executado
  - O sistema operativo só pode retomar o controlo do CPU quando ocorre uma interrupção
    - Se se desligarem as interrupções, garante-se que o sistema operativo não irá retomar, no intervalo de tempo que durar a inibição das interrupções, o controlo do sistema
    - Ou seja o escalonador não irá ser executado e não haverá troca de processos!

Limitações da inibição das interrupções (1)

- Limitação #1
  - A inibição das interrupções só funciona em processadores simples com núcleos **não-preemptivos**
  - Núcleo não preemptivo
    - Um processo em execução no modo kernel **não** pode ser suspenso.
    - Exemplos: Windows XP, Traditional UNIX
  - Núcleo preemptivo
    - Um processo em execução no modo kernel pode ser suspenso
    - Exemplos: Linux >= 2.6.XX e Solaris 10
- Pergunta
  - O que é ao certo um núcleo/*kernel* preemptivo?

Resposta >>

O que é um núcleo preemptivo?

- ✓ Núcleo preemptivo
  - Núcleo (*kernel*) em que um processo ou thread pode ser interrompida (preempção) quando se encontra a executar em modo kernel
  - Num núcleo **não** preemptivo, um processo a executar uma chamada ao sistema não pode ser retirado do CPU enquanto durar a chamada ao sistema
  - Num núcleo preemptivo isso já pode ocorrer, podendo o SO executar outras threads
- ✓ Um núcleo preemptivo potencia
  - Diminuição da latência
    - Importante para sistemas de tempo real e aplicações multimédia (audio, video, etc.)

Limitações da inibição das interrupções (2)

- ✓ O que pode suceder em sistemas multi-processadores (e.g. SMP or *multicores*)?
  - Um processador pode ter as interrupções desativadas, mas um processo a correr num outro processador/core pode alterar o valor de uma variável partilhada...
  - Solução
    - Mecanismo de sincronização ao nível do sistema operativo
  - Exemplo
    - Na versão 2.0 do Linux (+/- 1999), o suporte para SMP era feito à custa do **Big Kernel Lock**
      - Sempre que o código do sistema operativo pretendia aceder a uma estrutura do sistema operativo, recorria ao **Big Kernel Lock**
      - O **Big Kernel Lock** foi substituído por locks localizado (um *lock* por estrutura, etc.)

Outra limitação da inibição das interrupções >>

Limitações da inibição das interrupções (3)

- ✓ Limitação 2
  - – As interrupções só podem ser desligadas por curtíssimos períodos de tempo
    - Uma centena de instruções do CPU, o que representa intervalos de tempo da ordem do **microssegundo**
    - Se as interrupções forem desligadas por períodos maiores, o sistema entra em colapso...

Solução: Instruções CAS >>

Alternativas – instruções CAS

- Designação genérica CAS (Compare-and-Swap)
- Instrução do processador que, de forma atómica:
  - Compara o conteúdo de endereço de memória com um dado valor
    - se forem idênticos, modifica o conteúdo do endereço de memória, colocando lá um (outro) novo valor
- O resultado da instrução deve indicar se houve lugar à escrita do novo valor
  - A escrita falha se entre a comparação e a escrita um processo/thread alterou o conteúdo do endereço de memória
- CAS é empregue para a implementação de primitivas de sincronização
  - Semáforos, mutexes, futexes

CAS na arquitetura x86 >>

CAS no x86 – CMPXCHG (#1)

- ✓ Exemplo de instrução CAS
  - CMPXCHG na arquitetura X86
  - Compare and exchange

- ✓ **CMPXCHG** destination, source
- ✓ Funcionamento (Acc: registo EAX (modo 32 bits), ZF: Zero flag)
  - if (Acc == dest) THEN

```
{
  ZF <- 1;
  dest <- source;
}
ELSE
{
  ZF <- 0;
  Acc<- dest;
}
END
```

Description
Compares the value in the AL, AX, or EAX register (depending on the size of the operand) with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, or EAX register. This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

LOCK CMPXCHG
(sistemas multicores e sistemas multiprocessadores)

http://x86.renejeschke.de/html/file_module_x86_id_41.html

CAS no x86 – CMPXCHG (#2)

- ✓ Em sistemas multicores/multiprocessadores, é ainda necessário garantir que outros CPU/cores não acedem à memória
- ✓ O mesmo para sistemas single-core em que existam dispositivos a aceder à memória (e.g., via DMA)
  - Prefixo LOCK para garantir que o BUS de memória fica bloqueado durante a execução a instrução CAS
    - LOCK CMPXCHG
    - Bloqueia a "linha de cache"
- ✓ A norma C11 foi a primeira a apresentar funções atómicas
  - #include <stdatomic.h>

Exemplo

```
bool atomic_compare_exchange_strong(volatile A *object,C *expected, C desired);
```

  - (…)
- ✓ Nem todos os compiladores C11 implementam o stdatomic.h
  - Se a macro **__STDC_NO_ATOMICS__** estiver definida significa que o compilador NÃO suporta o `<stdatomic.h>`

Falha no kernel do Linux

Exemplo de “competição por recursos”

Falha "Dirty COW" / linux

DIRTY COW

- CVE-2016-5195
  - Acesso root por contas regulares ("*privilege escalation*")
- Ativo desde 2.6.22 (2007) até 2016…
- Competição por recursos envolvendo a metodologia COW: Copy-On-Write e *mmap*
  - *Kernel race condition*
- Permite escrita em ficheiro protegido do root a partir de conta regular!
- "Proof of concept": https://github.com/dirtycow/dirtycow.github.io/wiki/PoCs

- Receita para ativar problema
  - Partilha de páginas de memória através de mmap ao ficheiro

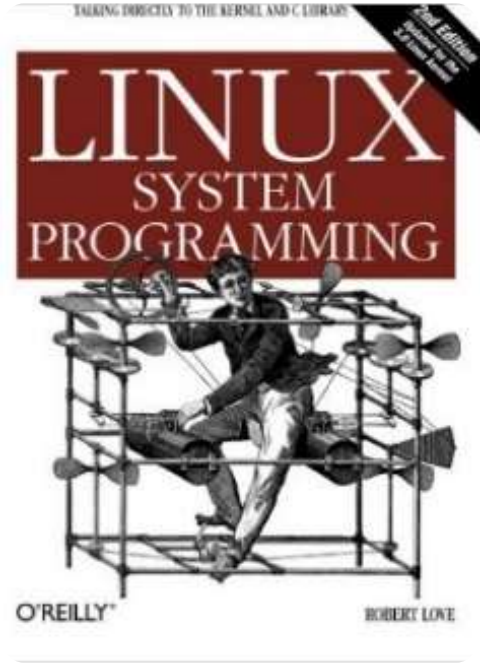
➔ Uso de **duas** threads concorrentes **madvise** e **procselfmem**

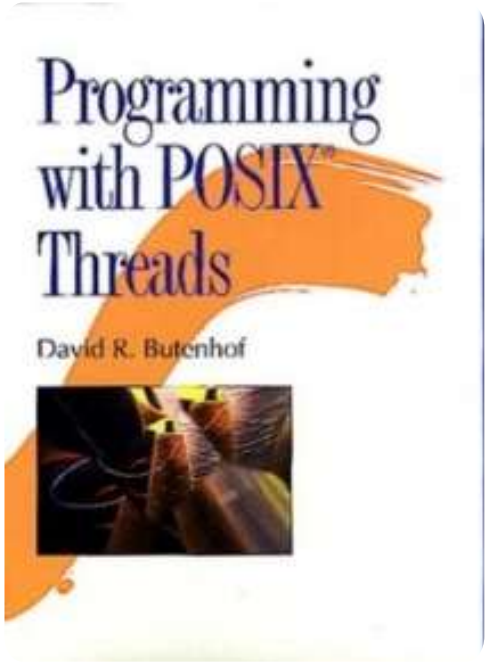
1) Thread **madvise**: chama "madvise". Avisar kernel de que não se vai usar a memória:
   `madvise(map, 100, MADV_DONTNEED)`
2) Thread **procselfmem**: abre pseudo-ficheiro /proc/self/mem, escrevendo para o mesmo
3) Sempre que escreve no ficheiro, o kernel deve criar cópia (**copy-on-write**). Contudo, devido a *race condition*, de vez em quando (raramente) falha e permite escrita por terceiro…

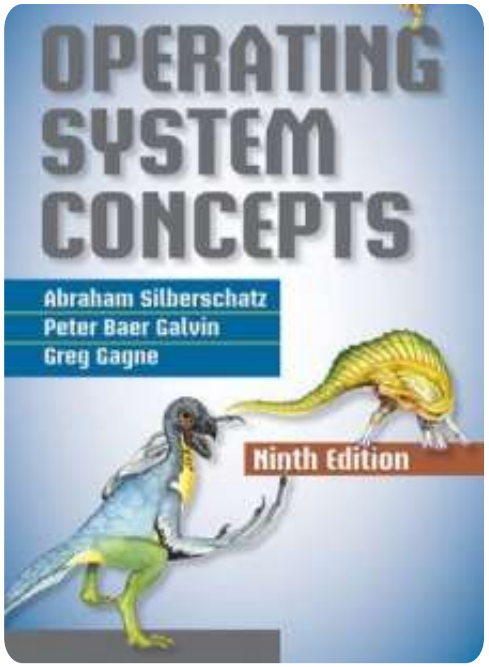
- Video: http://bit.ly/2dKMvm1
- 
"Most serious" Linux privilege-escalation bug ever is under active exploit (updated)

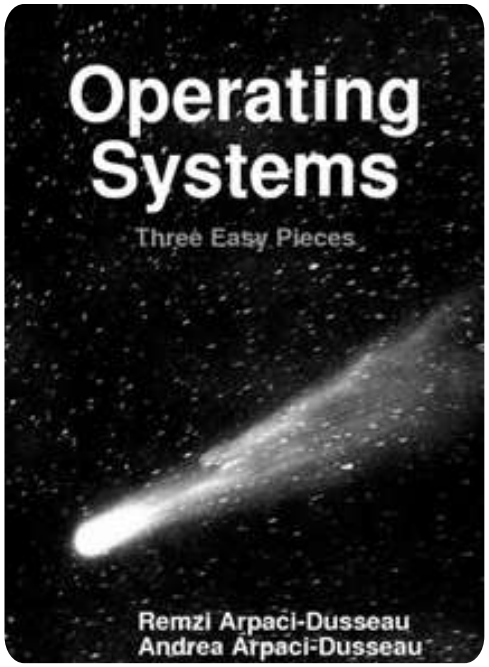
Lurking in the kernel for nine years, flaw gives untrusted users unfettered root access.

DAN GOODIN - 10/20/2016, 9:20 PM









Bibliografia

- “Linux System Programming”, Robert Love, Cap. 7 - Threading, O’Reilly, 2ª edição, 2013.
- “Operating Systems: three easy pieces” – part 2 – concurrency, Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, 2018. www.ostep.org
- "Programming with POSIX Threads", David R. Butenhof, Addison-Wesley, 1997, ISBN-13: 978-0201633924
- Operating System Concepts, 9th Edition, Abraham Silberschatz, Peter B. Galvin, Greg Gagne, 2012 – Cap. 5 (Process Synchronization) & Cap. 7 (Deadlocks)