Tempo-real em sistemas embutidos Linux

Sistemas de Tempo-Real
DETI/UA

Paulo Pedreiras
DETI/UA
Set/2012

Adaptado de: Real-time in embedded Linux systems, por M. Opdenacker, T. Petazzoni e G. Chanteperdrix

Índice

- Linux e tempo-real
- Melhorando o desempenho do Linux
- Fontes de latência
- O projeto PREEMPT_RT
- Desenvolvimento de aplicações tempo-real
- Leitura adicional

Embedded Linux e tempo-real

- O SO Linux, bem como outro software *open-source,* é cada vez mais usado no desenvolvimento de sistemas embutidos
- Todavia, muitas destas aplicações apresentam **requisitos tempo-real**
- Idealmente gostaríamos de:
  - Ter as vantagens do Linux: suporte de hardware, baixo custo, reutilização de componentes, etc.
  - E cumprir *deadlines*!

?

Embedded Linux e tempo-real

- O SO Linux foi **desenhado** como um sistema ***time-sharing***
  - Objetivo principal é otimizar o *throughput,* maximizando o uso dos recursos de hardware (CPU, memória, I/O)
  - O determinismo temporal não é fator primordial
- Os sistemas operativos de **tempo-real** são desenhados em função do **determinismo temporal**, frequentemente com prejuízo do *throughput*
- Acresce que *throughput* e determinismo temporal são **requisitos contraditórios**
  - A otimização em função de um degrada o outro!

Melhorando o comportamento temporal do Linux

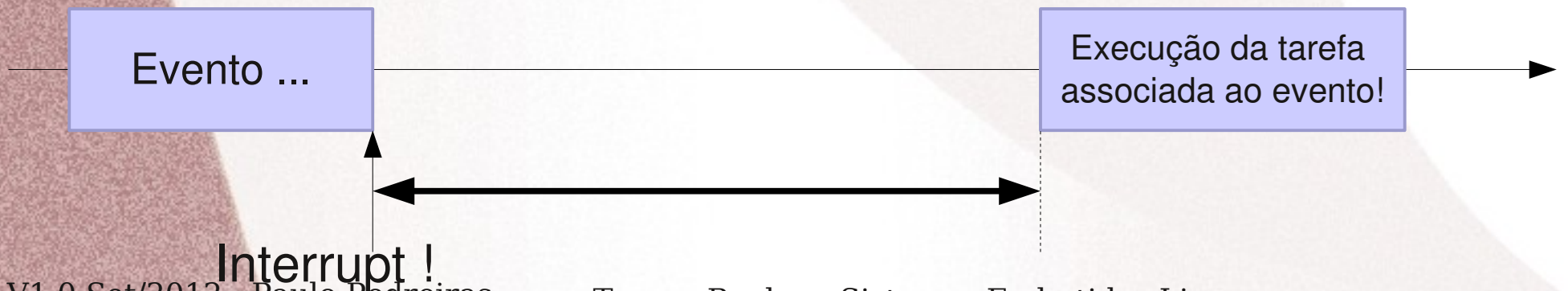
- Este conflito intrínseco foi abordado segundo duas estratégias:
- **Abordagem 1**
  - Introdução de melhoramentos no kernel Linux, por forma a melhorar o comportamento temporal deste.
    - Modificações no kernel (e.g. limitar a latência), disponibilização de serviços específicos tempo-real, etc.
  - Esta abordagem tem sido seguida pela “linha principal” do kernel Linux, bem como por projetos como o PREEMPT_RT
- **Abordagem 2**
  - Adição de uma camada, sob o kernel Linux, que efetua a gestão dos requisitos temporais. O Linux passa a ser uma tarefa de *background*
    - Abordagem seguida pelo RTLinux, RTAI e Xenomai

Abordagem 1

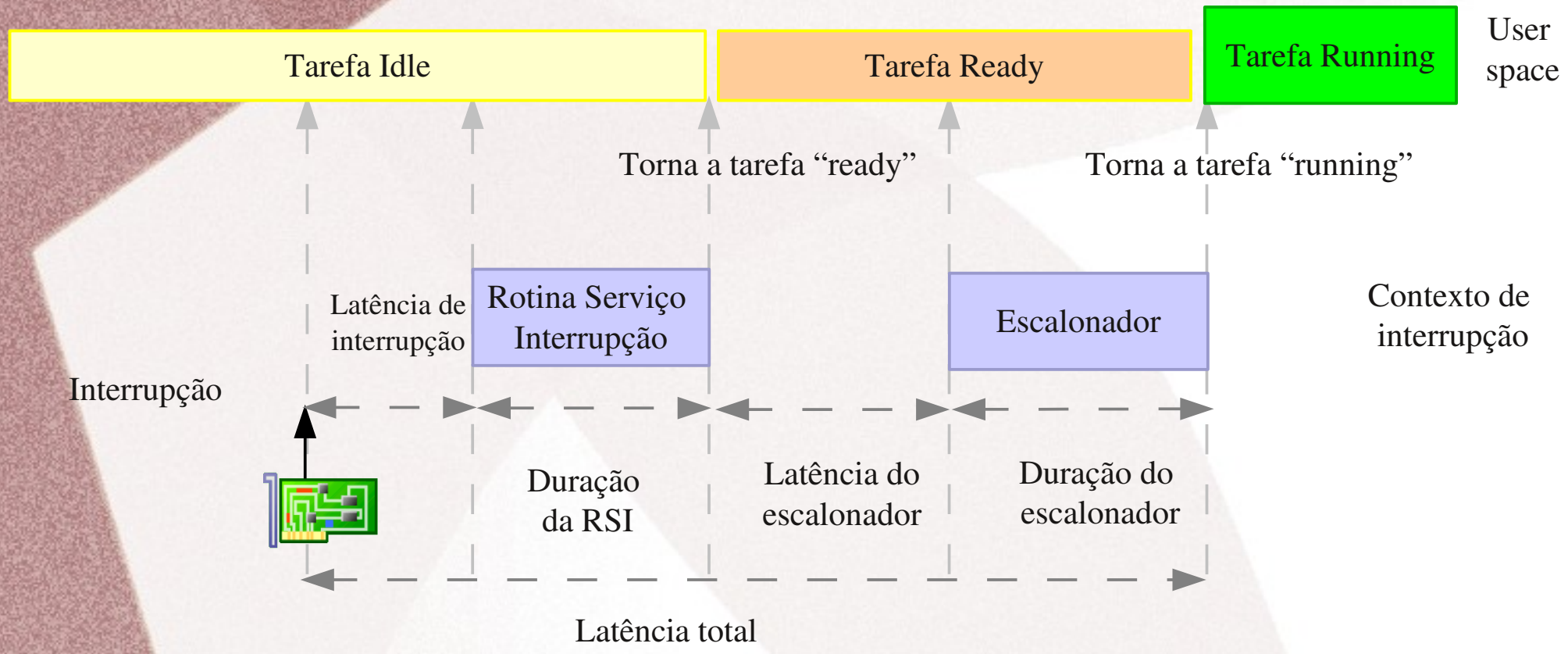
Melhorando o o desempenho temporal do kernel Linux com o PREEMPT_RT

Fontes de latência

- A **sequência típica de eventos** numa aplicação tempo-real é a seguinte:
  - Um evento, gerado no “ambiente”, ocorre e causa uma interrupção no CPU
  - A correspondente rotina se serviço à interrupção é executada, colocando no estado *ready* a tarefa localizada em *user space* associada ao evento
  - Algum tempo depois esta tarefa é iniciada, completando-se assim a reação ao evento - latência
- **O objetivo é limitar o valor máximo da latência**

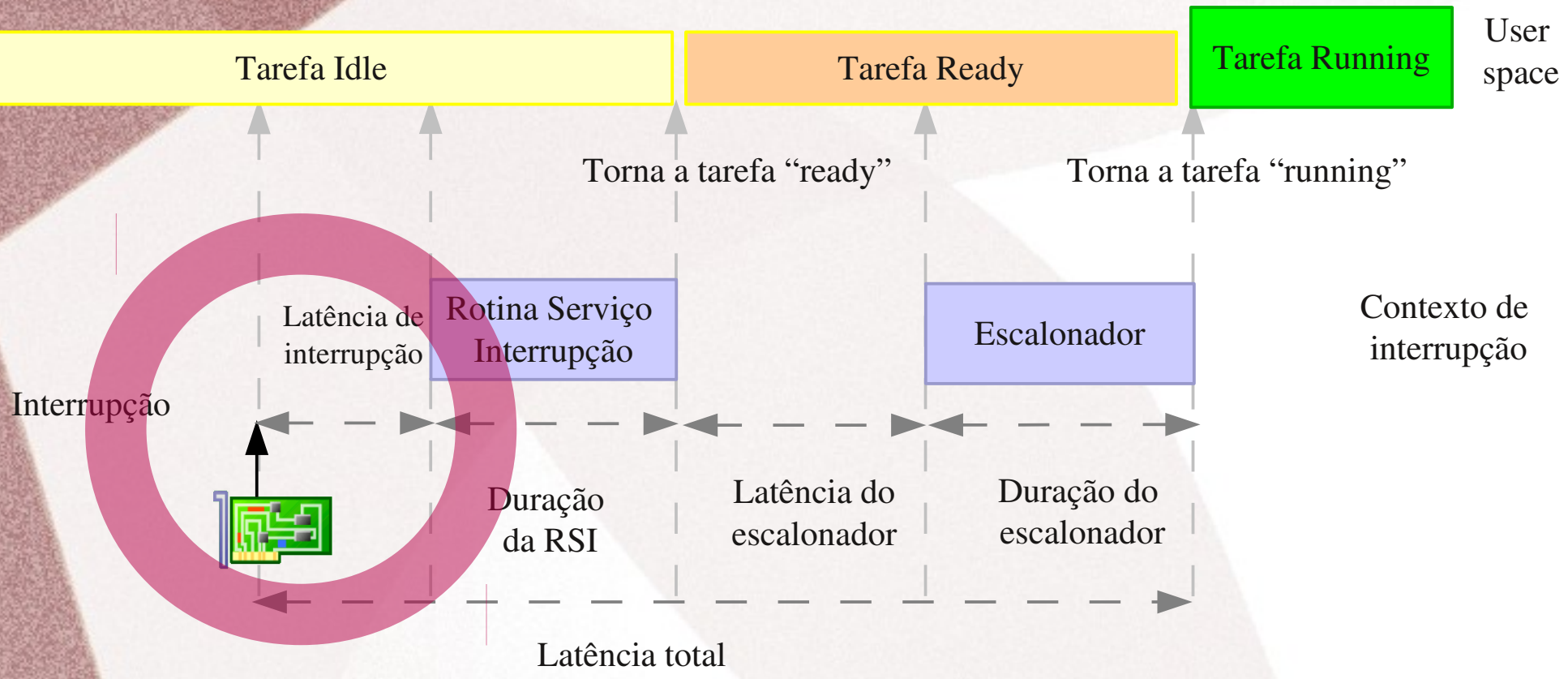


Kernel Linux – fontes de latência



Latência do kernel = latência de interrupção + tempo execução RSI
+ latência do escalonador + tempo exec. escalonador

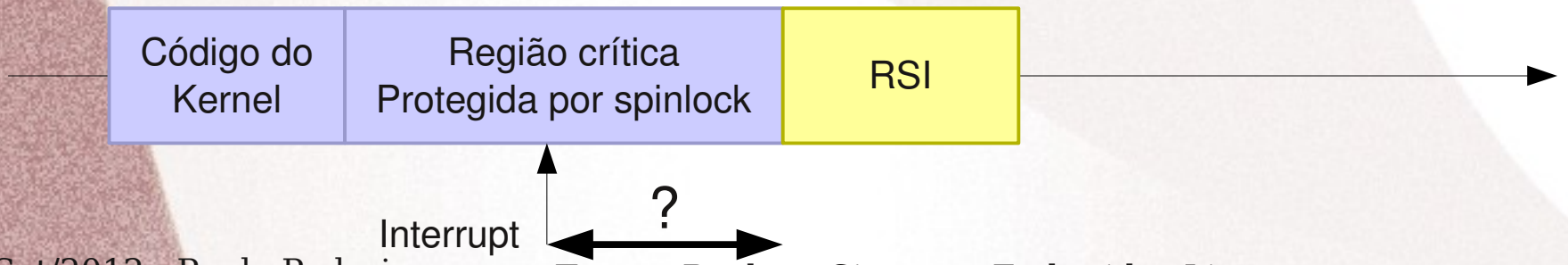
Kernel Linux – latência de interrupção



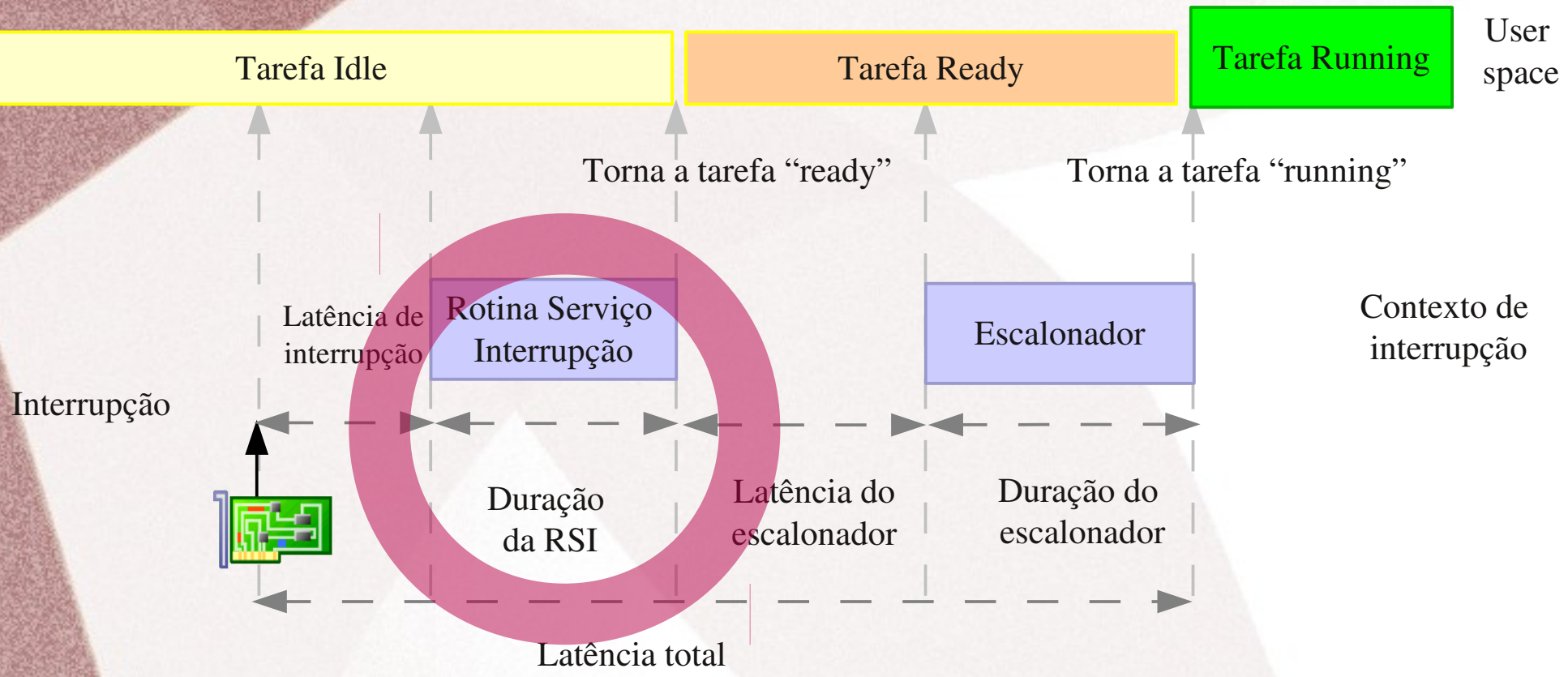
Latência de interrupção – tempo que decorre até início de execução da RSI

Kernel Linux – latência de interrupção

- Um dos mecanismos de controlo de concorrência usado no kernel é o **spinlock**
- Há vária variantes. Uma das mais comuns evita a mudança entre contexto de processo e de interrupção inibindo as interrupções
- Regiões criticas protegidas por spinlocks, bem como outras em que as interrupções sejam explicitamente desativadas, irão atrasar a execução da RSI
  - A duração destas regiões criticas não é limitada superiormente!
- Adicionalmente há que considerar outras fontes de perturbação, e.g. interrupções partilhadas



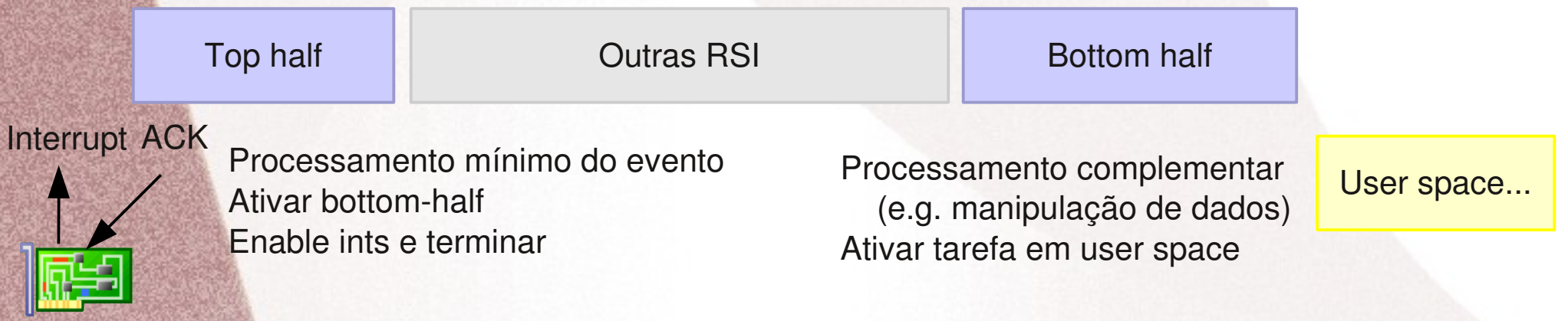
Kernel Linux – tempo de execução da RSI



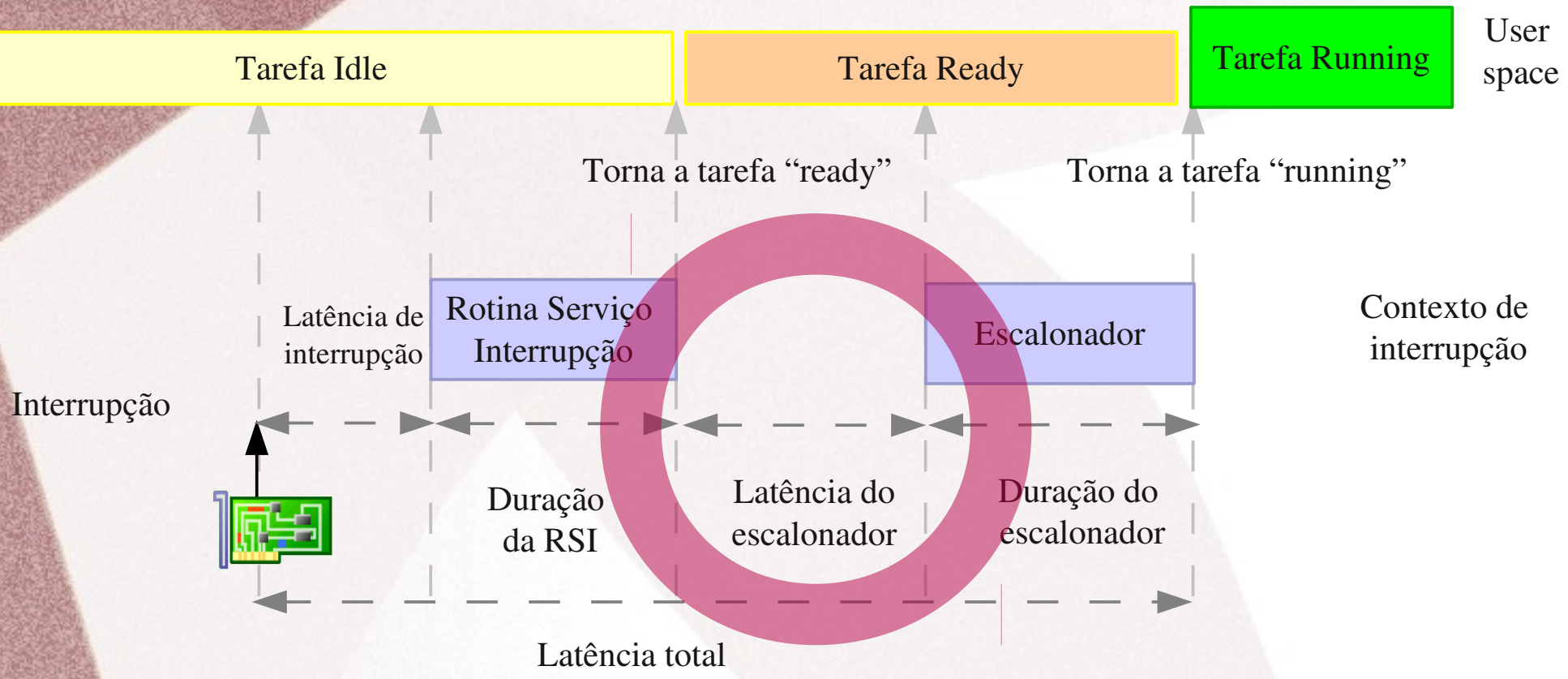
Duração da RSI - tempo necessário até à completa execução da RSI

Kernel Linux – tempo de execução da RSI

- Em Linux as RSI são divididas em duas partes:
  - A *top-half/fast-handler*, executado pelo CPU em resposta direta à interrupção. Executa com as interrupções desligadas e deve ser tão curta quanto possível
  - A *bottom-half/slow handler*, ativada pela *top-half* e começando após esta. É executada com as interrupções ativas e colocada numa fila. Há concorrência com outras tarefas bem como com outras RSIs.
  - Assim, para melhor comportamento temporal, as *bottom-half* não devem ser usadas



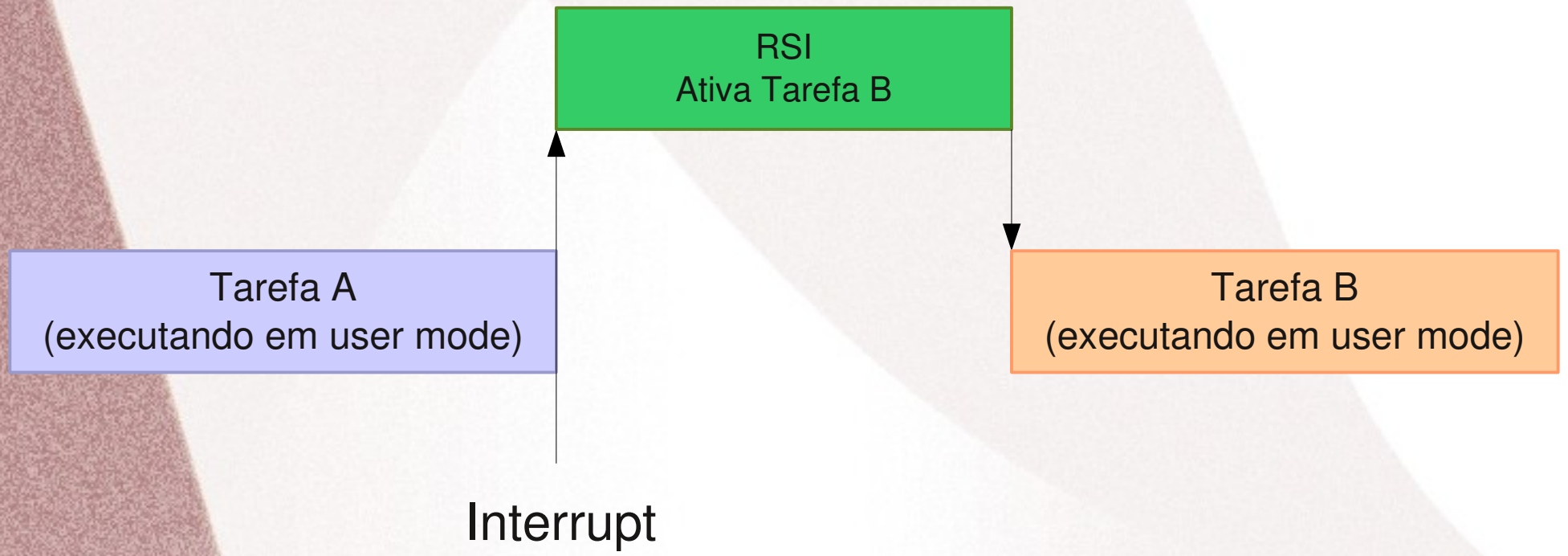
Kernel Linux – Latência do escalonador



Latência do escalonador – tempo até início de execução do escalonador

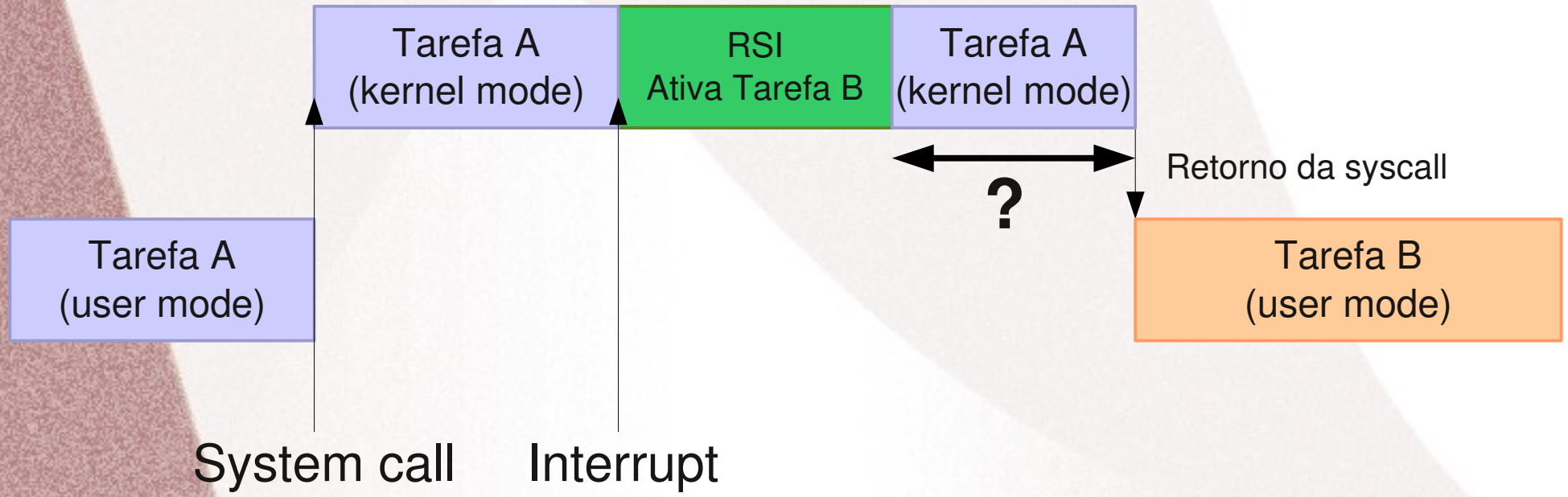
Kernel Linux – Latência do escalonador

- O kernel Linux kernel é preemptivo
  - Quando uma tarefa (em *user space*) é interrompida por uma interrupção, se a RSI ativa uma tarefa de de maior prioridade esta pode ser escalonada logo após o terminus da RSI

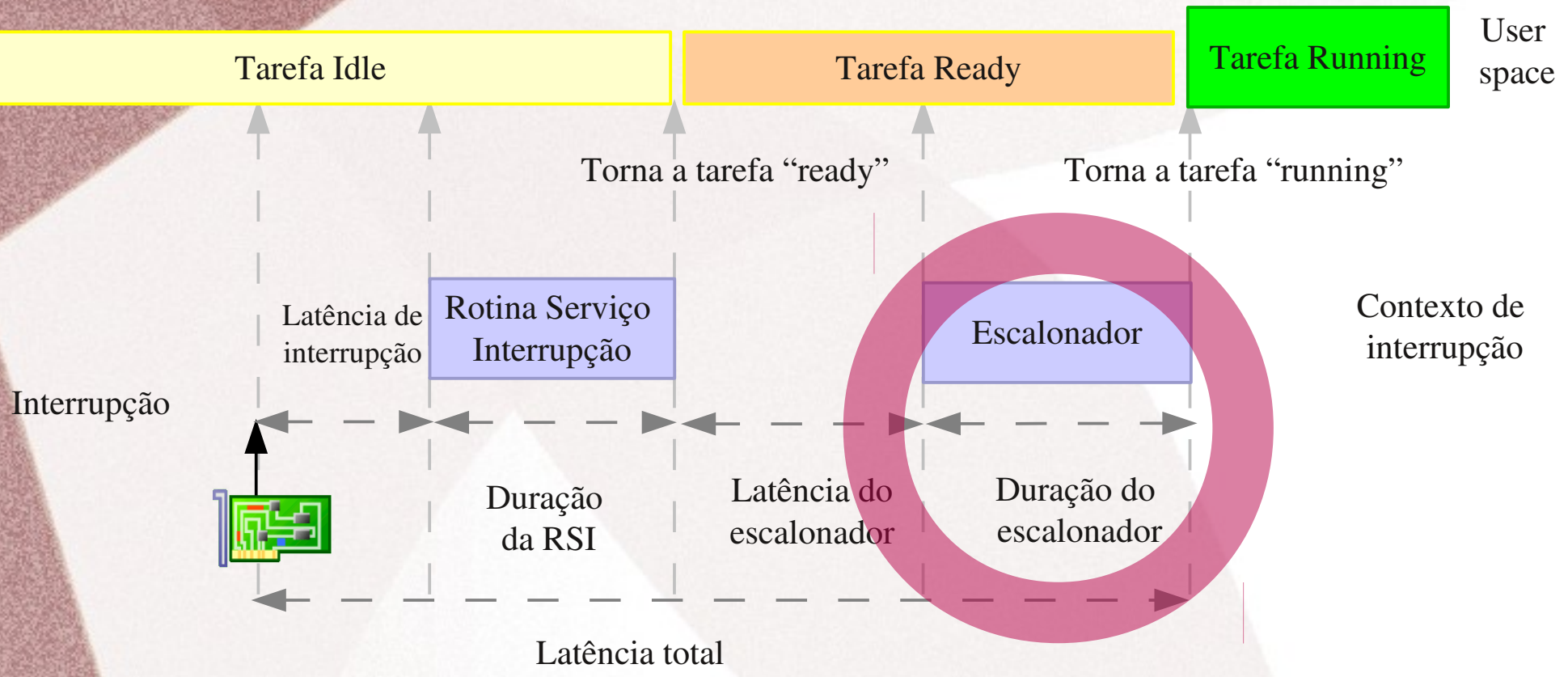


Kernel Linux – Latência do escalonador

- Todavia, se a interrupção acontecer quando a tarefa está a executar uma system call, só após o terminus da system call é que a nova tarefa pode ser executada
- Por defeito o kernel Linux não suporta preempção do kernel
- Isto implica que o tempo necessário para se inciar a execução de uma tarefa não é limitado (é *unbounded*)



Kernel Linux – Execução do escalonador

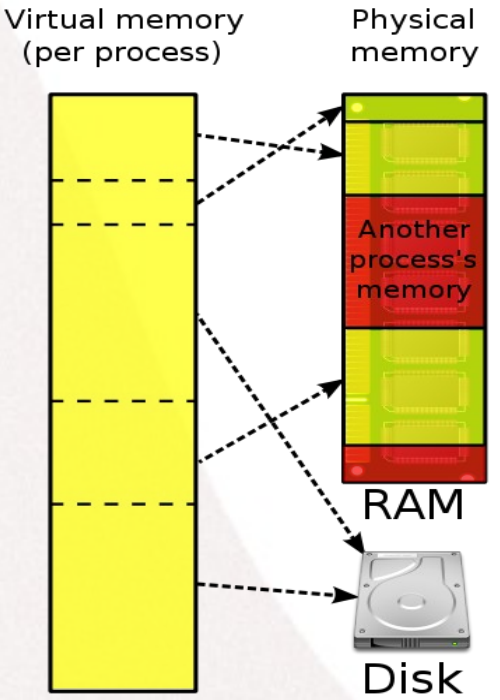


Tempo de execução do escalonador – tempo necessário para completar a execução do escalonador e iniciar a execução da tarefa

Dependo do tipo de escalonador. Pode ser bounded!

Kernel Linux – Outras fontes de latência

- Para além dos aspetos mencionados anteriormente, há ainda a considerar **outros mecanismos não determinísticos** que podem afetar a execução de tarefas tempo-real:
  - O Linux faz uso intensivo de memória virtual. Sempre que uma tarefa acede a dados que estão em disco há atrasos significativos
  - Idem para mecanismo de cache
  - Muitas bibliotecas de funções “C” e serviços de kernel não foram desenhados segundo critérios de determinismo
  - Inversão de prioridades
    (devido a partilha de recursos)
  - Prioritização de interrupções
  - ...



O projeto PREEMPT_RT

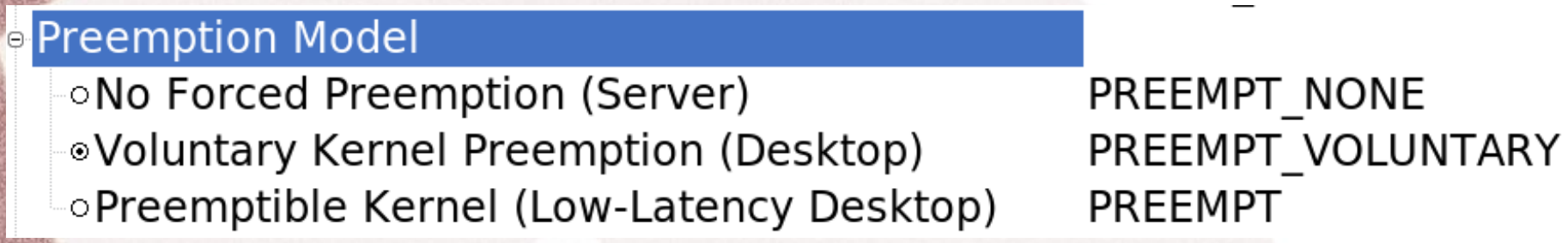
- Desenvolvido por Ingo Molnar, Thomas Gleixner and Steven Rostedt (https://rt.wiki.kernel.org)
- Objetivo: **melhorar gradualmente o comportamento temporal do kernel Linux e introduzir esses melhoramentos no "mainline kernel"**
  - Muitos dos melhoramentos fazem já parte do "mainline Linux kernel"
  - Poderá vir a ser completamente integrado no kernel, deixando de existir de per si

PREEMPT_RT – melhoramentos integrados no mainline kernel

- Desde o lançamento da versão 2.6
  - Escalonador O(1)
  - Preempção do Kernel
  - Melhoramentos aos suporte ao POSIX real-time API
- Desde a versão 2.6.18
  - Mutexes com suporte a "Priority inheritance"
- Desde a versão 2.6.21
  - Timers de alta resolução
- Desde a versão 2.6.30
  - Threaded interrupts
- Desde a versão 2.6.33
  - "Spinlock annotations" controlo de spinlocks que não podem ser convertidos em "sleeping spinlocks"

Controlo de preempção em Linux 2.6

O Linux 2.6 diversos modos de preempção:



Preempção em Linux 2.6: no forced preemption

`CONFIG_PREEMPT_NONE`

Código do kernel (interrupts, exceções, system calls) nunca sofrem preempção

- Configuração por defeito em kerneis standard
- Melhor opção para sistemas que efetuam processamento intensivo em que o objetivo é maximizar o *throughput*
  - Reduz as mudanças de contexto e *overhead* associado

Preempção em Linux 2.6: voluntary kernel preemption

`CONFIG_PREEMPT_VOLUNTARY`

O código do kernel pode interromper-se a si próprio

- Útil em ambiente *desktop*, para aumentar a reatividade a utilizadores (humanos)
- São adicionados ao kernel pontos explícitos de reescalonamento
- Impacto reduzido no *throughput*.

Preempção em Linux 2.6: preemptible kernel

`CONFIG_PREEMPT`

A maior parte do *kernel* pode ser interrompida em qualquer altura.

- Quando um processo é ativado pode entrar em execução sem ter de aguardar a terminação de eventuais *system calls* pendentes
- Todavia algumas regiões críticas do kernel (protegidas por holding *spinlocks*) não são preemptíveis.
  - Melhor opção para sistemas embutidos com latências na casa dos milisegundos
  - Reduzido impacto no *throughput*.

Herança de prioridades

- Mecanismo para o controlo de inversão de prioridades, que consiste em **elevar temporariamente** a prioridade de uma tarefa que esta a bloquear outra de maior prioridade
  - Desde a versão 2.6.18 do Linux que os mutexes suportam herança de prioridades
  - Em user-space, o mecanismo tem de ser explicitamente ativado para cada mutex

Os mecanismos de controlo de acesso a recursos partilhados serão estudados em detalhe na Aula 7 do curso

Timers de alta resolução

A resolução ds timers estava condicionada à resolução do tick do sistema

- O tick do sistema era habitualmente 100 Hz ou 250 Hz (1KHz em algumas configurações)
- Resolução de 10 ms, 4 ms ou, no máximo, 1ms
  - A **frequência do *tick* tem impacto elevado no *overhead*** do sistema. Não pode ser aumentada arbitrariamente!!!

- A infraestrutura de timers de alta resolução foi integrada na versão 2.6.21. Suporta o uso de **timers de hardware com elevada resolução**
  - Os timers de hardware são multiplexados
    - suporte a um largo número de timers
  - Utilizável diretamente do user-space, usando a API standard

Interrupções Threaded

A fim de resolver o problema de inversão de prioridades, o PREEMPT_RT introduziu os *threaded interrupts*

- As RSI são executadas como threads normais do kernel. Assim, as prioridades das diferentes RSI pode ser configurada de uma forma adequada
- A RSI "real" apenas mascara a interrupção e ativa a thread associada
- Integrada no *mainline kernel* desde a versão 2.6.30
  - Não é automático! Requer modificação dos drivers.
  - Processo moroso ...
  - Situação será alterada quando os principais drivers tiverem sido portados

Interrupções Threaded

Versão integrada no *mainline kernel* é **diferente** da do PREEMPT_RT. Ao aplicar o patch PREEMPT_RT:

- É adicionado um novo nível de preempção, denominado CONFIG_PREEMPT_RT
- Neste:
  - Todos os interrupt handlers são transformados em threaded interrupts
  - Todos os spinlocks do kernel são substituídos por mutexes (tb designados sleeping spinlocks)
  - Exclusão mutua deixa de ser efetuada desligando as interrupções, passando a ser por meio do bloqueio de processos. Quando há contenção o processo é bloqueado e um novo processo é selecionado pelo scheduler
    - Funciona com threaded interrupts, pois as threads podem bloquear, ao passo que os interrupt handlers normais não podem.

Desenvolvimento de aplicações tempo-real

Desenvolvimento e compilação

- Não são necessárias bibliotecas especiais ... caso se use a linguagem C
  - A API POSIX realtime faz parte da biblioteca C standard
- Efetuar o *link* com -lrt.
  - E.g. `gcc -o myprog myprog.c -lrt`
- Documentação da API
  - `man functioname`

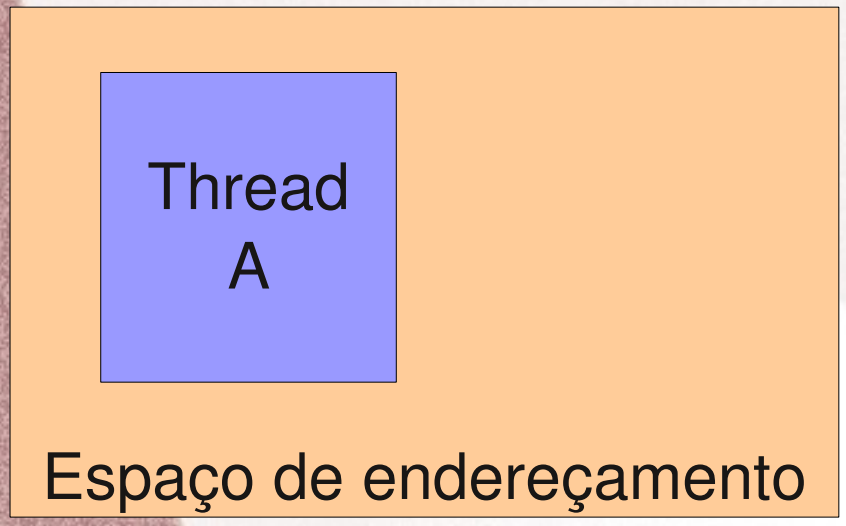
Process, thread ?

Alguma confusão entre os termos “processo”, “thread” e “task”/tarefa

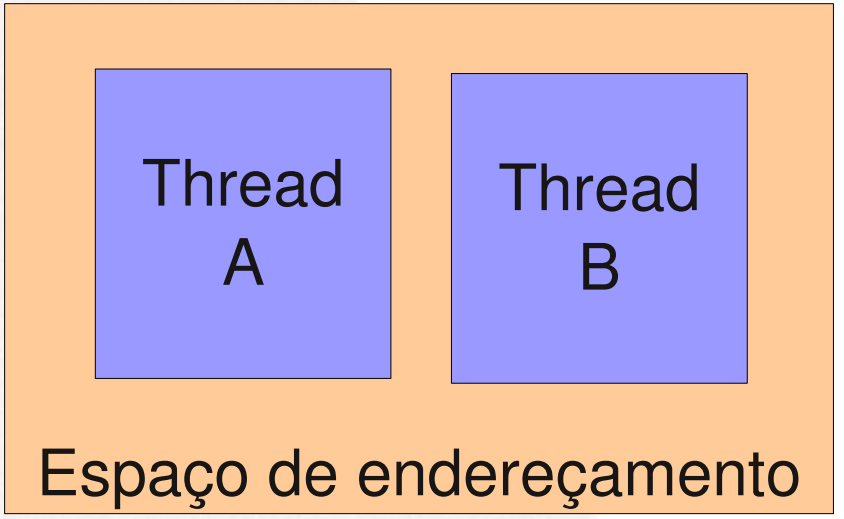
- Em Unix os processos são criados com a primitiva `fork()` e são compostos por:
  - Um espaço de endereçamento (para código, dados, stack, ...)
  - Uma thread, que inicia a execução da função main()
    - Após a criação, um processo contém uma thread
    - Threads adicionais podem ser criadas dentro do processo por meio da primitiva `pthread_create()`
      - Estas partilham o mesmo espaço de endereçamento que a thread inicial ...
      - e executam a função passada como argumento a `pthread_create()`
  - Frequentemente é também usado o termo “task”, como sinónimo de processo (e.g. Buttazzo)

Processos e threads do ponto de vista do kernel

- O kernel representa as threads por uma estrutura do tipo “task_struct”
- Do ponto de vista do escalonamento, não há qualquer diferença entre a thread inicial e as threads criadas dinamicamente com pthread_create()



Processo após
`fork()`



Mesmo processo após\
`pthread_create()`

Criando threads

- O Linux suporta a API POSIX
- Para criar uma nova thread
  - ```
    pthread_create(pthread_t *thread,
                   pthread_attr_t *attr,
                   void *(*routine)(*void*),
                   void *arg);
    ```
  - A nova thread é executada no mesmo espaço de endereçamento, mas escalonada como uma entidade independente
- Terminando uma thread
  - ```
    pthread_exit(void *value_ptr);
    ```
- Aguardando pelo terminus de uma thread
  - ```
    pthread_join(pthread_t *thread, void **value_ptr);
    ```
    ```

Classes de escalonamento

O kernel Linux suporta diversas classes de escalonamento

- A classe por defeito, é do tipo time-sharing
  - Todos os processos recebem algum tempo de CPU, independentemente da sua prioridade
  - A proporção de CPU que cada processo obtém é dinâmica
    - Afetada pelo valor “nice”, que varia de -20 (maior) a 19 (menor)
    - Depende do tipo de operações efetuadas
      - CPU-bound, I/O-bound
    - Pode ser alterada com os comandos nice e renice
  - Não determinística
  - **Mau desempenho do ponto de vista tempo-real**

Classes de escalonamento

- Há **duas classes tempo-real**:
  SCHED_FIFO e **SCHED_RR**
  - A tarefa ready com maior prioridade obtém o CPU
  - A prioridade é definida estaticamente
    - Varia de 0 (menor) a 99 (maior)
  - SCHED_RR vs SCHED_FIFO
    - SCHED_RR: é aplicada uma política round-robin aos processo que partilham a mesma prioridade
    - SCHED_FIFO: é aplicada uma política FIFO aos processo que partilham a mesma prioridade
  - Escalonamento determinístico
  - Bom desempenho para tempo-real

Classes de escalonamento

- Um processo pode ser executado numa classe específica de escalonamento por meio da syscall chrt
  - Exemplo: `chrt -f 99 ./myprog`

    `-f FIFO ; 99 prioridade`
- A syscall `sched_setscheduler()` permite modificar a classe de escalonamento e prioridade de um processo

```
int sched_setscheduler(pid_t pid, int policy,
const struct sched_param *param);
```

  - `policy`: `SCHED_OTHER, SCHED_FIFO, SCHED_RR`, etc.
  - `param`: estrutura que contém a prioridade

Classes de escalonamento

- A prioridade **individual** de cada thread pode ser definida aquando da sua criação:

```
struct sched_param parm;
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setinheritsched(&attr,
                             PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
parm.sched_priority = 42;
pthread_attr_setschedparam(&attr, &parm);
```

- A thread é criada usando `pthread_create()`, passando como argumento a estrutura `attr`
- Outras opções podem ser configuradas desta forma (e.g. tamanho da stack)

Bloqueio de memória

- Para **evitar o indeterminismo** resultante do uso de memória virtual, é possível efetuar o bloqueio de memória (*memory lock*)
  - Memória usada pelo espaço de endereçamento do processo mantém-se sempre em RAM

```
mlockall(MCL_CURRENT | MCL_FUTURE);
```

  - Bloqueia todas as páginas de memória atualmente usadas pelo espaço de endereçamento, bem como páginas futuramente alocadas (se `MCL_FUTURE`)
    - Heap, stack, shared memory, ...
- Outras syscalls relacionadas:
  - `munlockall, mlock, munlock.`

Mutexes

- **Mutext**: pemitem implementar exclusão mútua entre threads do mesmo processo
  - Inicialização/destruição

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

  - Lock/unlock

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

- Caso se deseje o uso de herança de prioridades:

```
pthread_mutexattr_t attr;
pthread_mutexattr_init (&attr);
pthread_mutexattr_getprotocol
      (&attr, PTHREAD_PRIO_INHERIT);
```

Timers

- Criar um timer.

```
timer_create(clockid_t clockid,struct sigevent *evp,
                        timer_t *timerid)
```

  - **clockid** é habitualmente CLOCK_MONOTONIC
  - **sigevent** define a acção a executar quando o timer expira (enviar um sinal ou iniciar uma função numa nova thread)
  - **timerid** devolve o identificador do timer
- Configurar um timer para expirar num certo instante

```
timer_settime(timer_t timerid, int flags,
                        struct itimerspec *newvalue,
                        struct itimerspec *oldvalue)
```

- Outras syscalls
  - timer_delete(), clock_getres(); timer_getoverrun(), timer_gettime().

- **Sinais**: mecanismos de notificação assíncrona
  - A notificação pode ser efetuada:
    - Pela chama de um “signal handler” (algumas limitações)
    - Desbloquendo de uma primitiva **sigwait**(), **sigtimedwait**() ou **sigwaitinfo**().
      - Método preferível!!
  - Comportamento do sinal pode ser definido usando a syscall **sigaction**()
  - Mascara de sinais pode ser efetuada com **pthread_sigmask**()
  - Envio de um sinal pode ser efetuado por **pthread_kill**() ou **tgkill**()
  - Podem ser usados os sinais entre **SIGRTMIN** e **SIGRTMAX** (32 sinais em Linux)

Comunicação entre processos

- Semáforos
  - Podem ser usados entre processos diferentes (*named semaphores*)

    `sem_open()`, `sem_close()`, `sem_unlink()`, `sem_init()`, `sem_destroy()`, `sem_wait()`, `sem_post()`, etc.
- Filas de mensagens (*message queues*)
  - Permitem troca de dados entre processos na forma de mensagens.

    `mq_open()`, `mq_close()`, `mq_unlink()`, `mq_send()`, `mq_receive()`, etc.
- Memória partilhada
  - Permitem troca de dados entre processos por meio da partilha de um segmento de memória

    `shm_open()`, `ftruncate()`, `mmap()`, `munmap()`, `close()`, `shm_unlink()`

Debug de latências de kernel

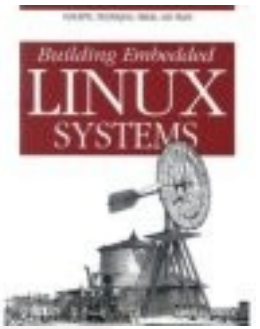
Ftrace – infraestrutura que pode ser usada para debug e análise de latência, bem como para detetar problemas de desempenho no kernel

- Desenvolvido por por Steven Rostedt e incluído no kernel 2.6.27.
- Para kerneis anteriores, pode ser encontrado no patch rt-preempt
- Muito bem documentado (`Documentation/ftrace.txt`)
- *Overhead* muito pequeno quando o trace não está ativado
- Pode ser usado para efetuar o *trace* de qualquer função do kernel

Para aprofundar estes assuntos ...

- Internals of the RT Patch, Steven Rostedt, Red Hat, June 2007
  http://www.kernel.org/doc/ols/2007/ols2007v2-pages-161-172.pdf
- The Real-Time Linux Wiki: http://rt.wiki.kernel.org
  "The Wiki Web for the `CONFIG_PREEMPT_RT` community, and real-time Linux in general."
- Building Embedded Linux Systems, O'Reilly
  By Karim Yaghmour, Jon Masters,
  Gilad Ben-Yossef, Philippe Gerum and others
  (including Michael Opdenacker), August 2008

  Inclui cobertura do Xenomai e do RT patch
  http://oreilly.com/catalog/9780596529680/



Organizações

- http://www.realtimelinuxfoundation.org/
  Portal da comunidade real-time Linux.
  Organiza uma workshop anual
- http://www.osadl.org
  Open Source Automation Development Lab (OSADL)
  Entre outras atividades, dedica-se ao desenvovimento e integração de RT preempt patches no mainline Linux kernel (HOWTOs, live CD, patches).