Laboratório de Pesquisa em Redes e Multimídia

# Escalonamento no Unix

Universidade Federal do Espírito Santo
Departamento de Informática

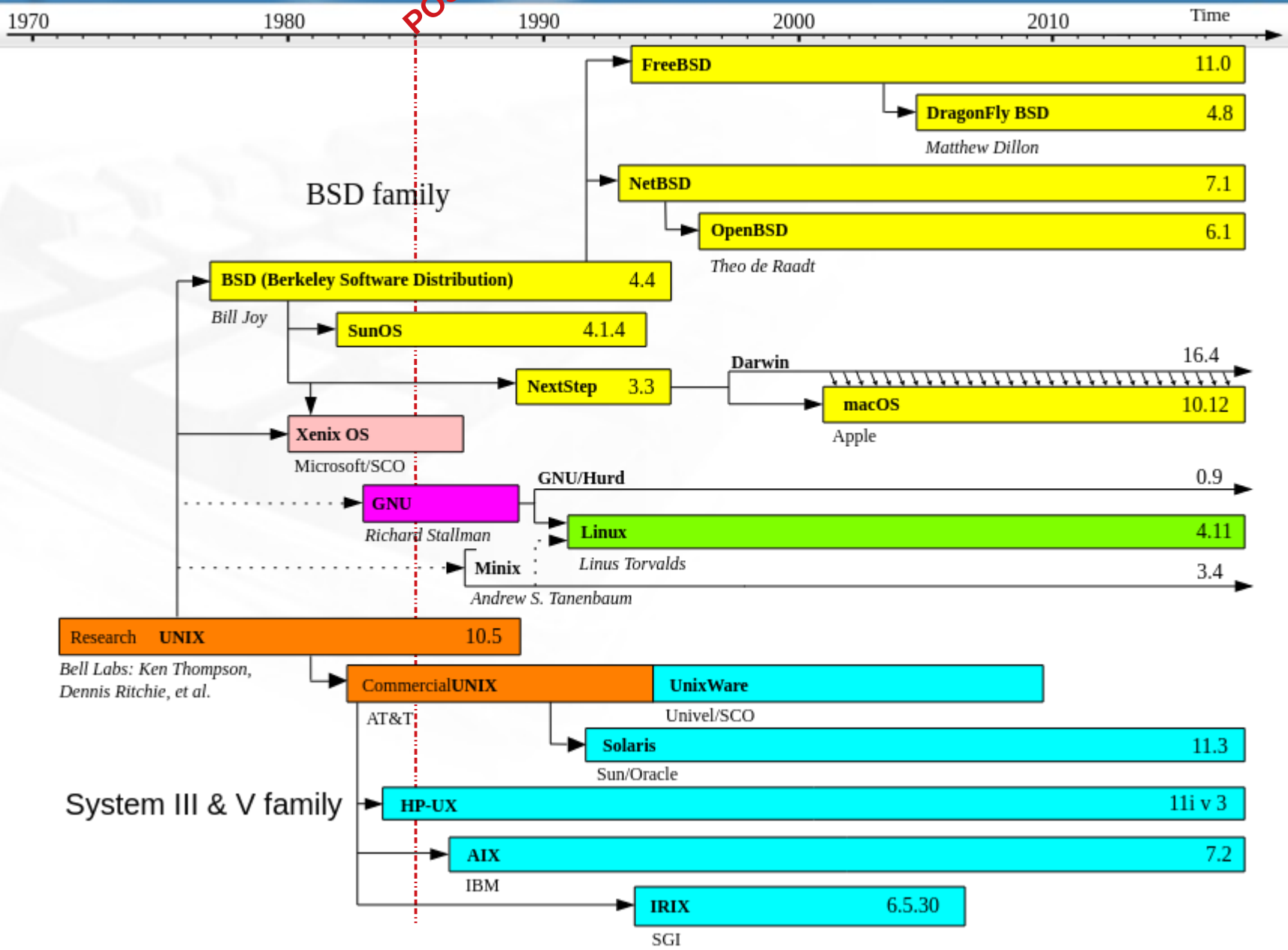
**Sistemas Operacionais**

# Projeto do Escalonador

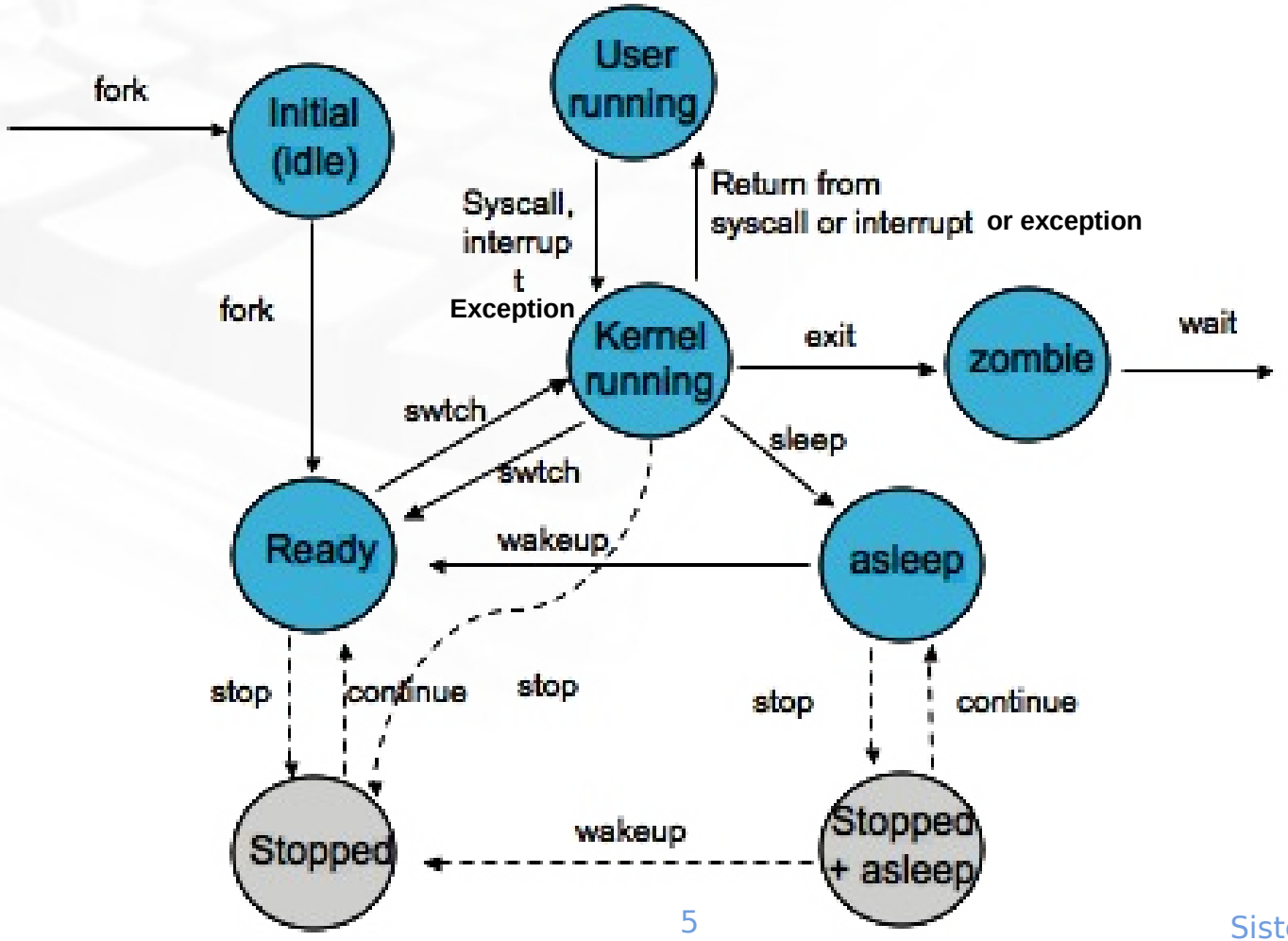
- O projeto de um escalonador deve focar em dois aspectos:
  - ***Política de escalonamento*** – estabelece as regras usadas para decidir para qual processo ceder a CPU e quando chaveá-la para um outro processo.
  - ***Implementação*** – definição dos algoritmos e estruturas de dados que irão executar essas políticas.
- A política de escalonamento deve buscar:
  - **Tempos de resposta** rápidos para aplicações interativas.
  - Alto ***throughput*** (vazão) para aplicações em *background.*
  - Evitar ***starvation***
- Os objetivos acima podem ser **conflitantes**!

# Objetivos do Escalonador (cont.)

- Processos **interativos** (*shells*, editores, interfaces gráficas, etc.)
  - Interações devem ser processadas rapidamente
  - Requisito: reduzir os tempos de resposta médios (e a variância), de forma que o usuário não detecte/perceba este atraso (50-150 ms)
- Processos ***batch*** (que rodam em *background*, sem interação com o usuário)
  - Requisito: o tempo para completar uma tarefa na presença de outras atividades, comparado ao tempo para completá-la em um sistema "inativo", sem outras atividades paralelas (*turnarround*).
- Processos de **tempo real (soft)**
  - Requisito: como as aplicações são "*time-critical*", o escalonador deve ter um comportamento previsível, com limites garantidos nos tempos de resposta.
  - Ex: aplicações de vídeo.
- As **funções do kernel**
  - Devem ser executadas prontamente (gerência de memória, tratamento de interrupções e gerência de processos)


1970
1980
POSIX
1990
2000
2010
Time
FreeBSD 11.0
DragonFly BSD 4.8
Matthew Dillon
BSD family
NetBSD 7.1
OpenBSD 6.1
Theo de Raadt
BSD (Berkeley Software Distribution) 4.4
Bill Joy
SunOS 4.1.4
Darwin 16.4
NextStep 3.3
macOS 10.12
Apple
Xenix OS
Microsoft/SCO
GNU/Hurd 0.9
GNU
Richard Stallman
Linux 4.11
Linus Torvalds
Minix 3.4
Andrew S. Tanenbaum
Research UNIX 10.5
Bell Labs: Ken Thompson, Dennis Ritchie, et al.
CommercialUNIX
UnixWare
Univel/SCO
AT&T
Solaris 11.3
Sun/Oracle
System III & V family
HP-UX 11i v 3
AIX 7.2
IBM
IRIX 6.5.30
SGI

# Máquina de Estados do Unix



# Modos de Operação da CPU

- O Unix requer do hardware a implementação de apenas 2 modos:
  - *user mode* - menos privilegiado. Execução de certas instruções e acesso a certos endereços é proibido.
  - *kernel mode* - mais privilegiado. Todas as instruções podem ser executadas, acesso à memória é irrestrito.

- "**Código de usuário**" é executado em ***user mode***; logo, não podem – acidental ou maliciosamente –, corromper outro processo ou mesmo o kernel.

- "**Código de kernel**" roda em ***kernel mode***

# Escalonamento Tradicional

- Usado originalmente nos sistemas Unix **SVR3** e **4.3BSD**:
  - Tempo compartilhado
  - Ambientes interativos
  - Processos *background* (batch) e *foreground* rodando simultaneamente
    - Melhorar os tempos de resposta para os usuários interativos, e
    - Garantir, ao mesmo tempo, que processos *background* não sofram *starvation*
- Prioridades dinâmicas
  - A cada processo é atribuída uma prioridade de escalonamento, que é alterada com o passar do tempo.
    - Se o processo não está no estado *running*, o kernel periodicamente aumenta a sua prioridade.
    - Quanto mais o processo recebe a posse da CPU mais o kernel reduz a sua prioridade.
  - **Por que essa estratégia previne *starvation* e favorece processos *I/O bound*?**

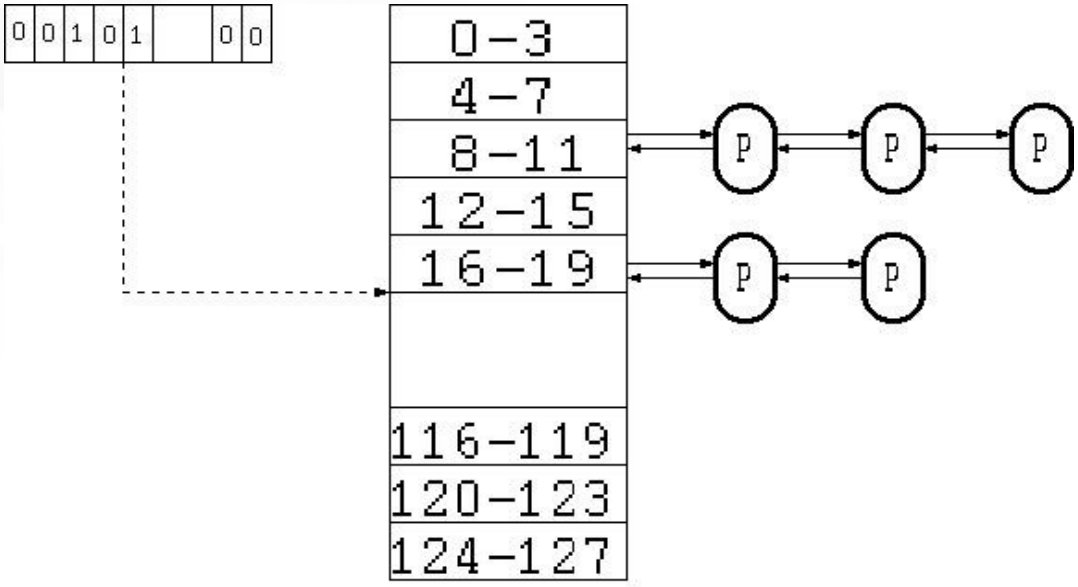
# Escalonamento Tradicional (cont.)

- O escalonador sempre seleciona o processo com a **prioridade mais alta** dentre aqueles no estado pronto-para-execução (*ready/runnable process*).
- Processos **com a mesma prioridade**:
  - "***Preemptive round robin***": escalonamento circular com preempção
  - O *quantum* possui um valor fixo, tipicamente de 100 ms.
- O kernel do Unix tradicional é **não-preemptivo**
  - A chegada de um processo de mais alta prioridade na fila de prontos força a preempção do processo em execução SE ele está executando em *user mode*
  - Um processo executando em *kernel mode* nunca é preemptado (nunca perde a posse da CPU para um outro processo).
  - Quando o sistema for retornar para *user mode*, é verificado se há um processo mais prioritário na fila de prontos. Se sim, ocorre a preempção (troca de contexto).

# Prioridades dos Processos

- Prioridades recebem valores entre 0 e 127 (quanto menor o valor numérico maior a prioridade)
  - 0 – 49 : processos do kernel (*kernel priorities*)
  - 50 – 127 : processos de usuário (*user priorities*)

**Ex: 4.3BSD**



# Prioridades dos Processos (cont.)

- Campos de prioridade na *proc struct*
  - p_pri prioridade de escalonamento atual
  - p_usrpri prioridade em *user mode*
  - p_cpu medida de uso recende da CPU
  - p_nice fator *nice* (gentileza) controlável pelo usuário (SVC *nice*)

$$p\_usrpri = PUSER + \frac{p\_cpu}{4} + 2 \times p\_nice$$

$$p\_usrpri = PUSER + \frac{p\_cpu}{4} + 2 \times p\_nice$$

## 4.3 BSD

- ***PUSER = 50***
  - prioridade base dos processos de usuário

- ***p_nice***=[0-39] (controlado pelo usuário via SVC *nice()* )
  - Default: *p_nice* = 20
  - ***nice(x)*** : -20 ≤ x ≤ +19 **=>** p_nice = p_nice + x
    - Se **x>0**, a chamada **diminui a prioridade** final do processo
    - Se **x<0**, a chamada **aumenta a prioridade** final do processo (apenas superusuário)
  - Processos *background* recebem automaticamente grandes valores de *p_nice* , por isso são menos prioritários.

$$p\_usrpri = PUSER + \frac{p\_cpu}{4} + 2 \times p\_nice$$

# Prioridades dos Processos (cont.)

## 4.3 BSD

- A cada ciclo (CPU tick ... 1 tick = 10ms):
  - Incrementa o p_cpu do processo que está rodando
- A cada 4 ciclos:
  - **Recalcula** p_usrpri do processo que está **rodando**
  - Se houver um processo mais prioritário, *context switch*
- A cada 10 ciclos
  - Round-robin entre processos de mesma prioridade (*context switch*)
- A cada 100 ciclos
  - Aplica um *decay factor* no p_cpu de TODOS os processos
  - **Recalcula** p_usrpri de **todos** os processos (possível *context switch*)

# *Sleep Priority*

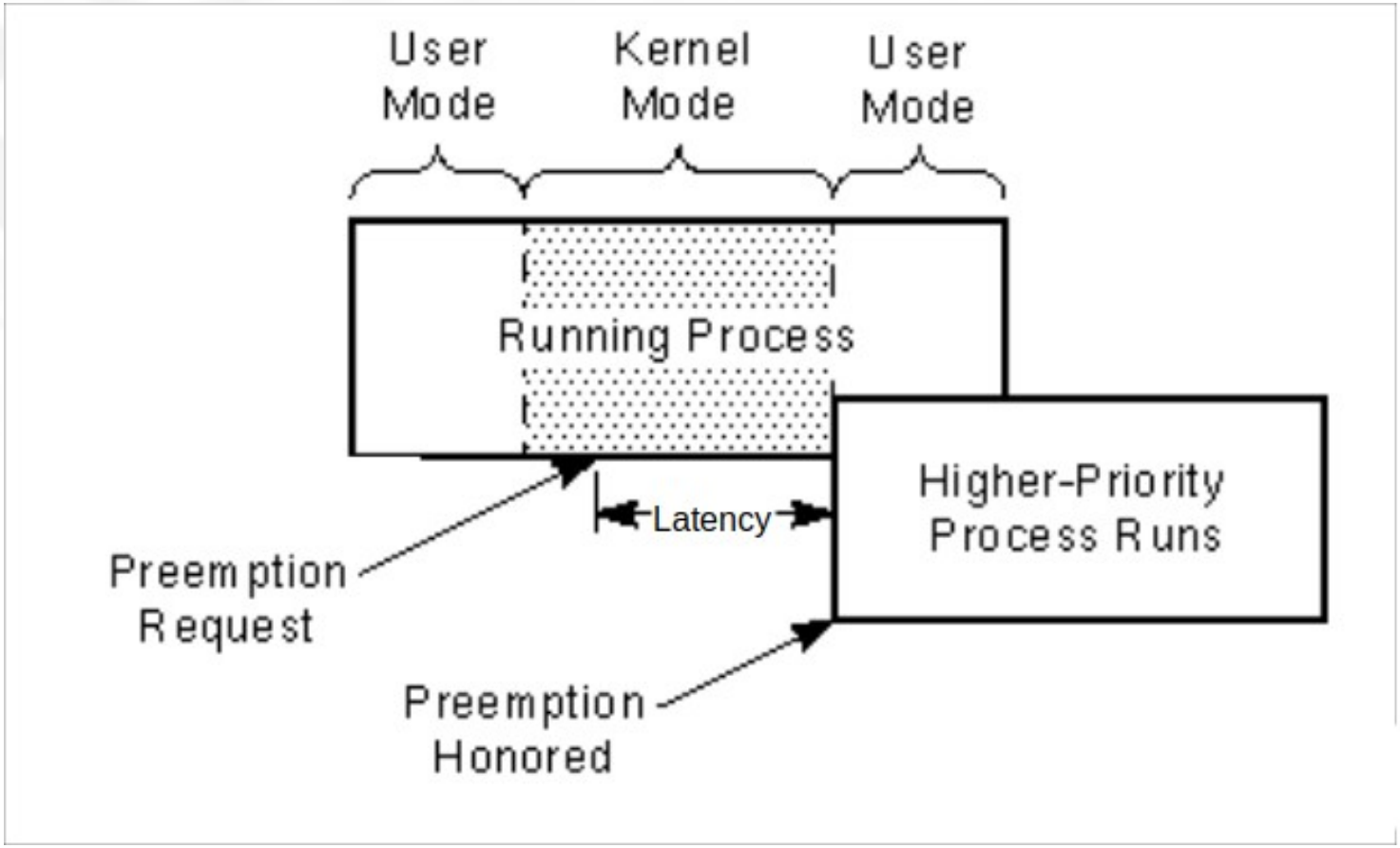
- Quando o processo está bloqueado, ele é associado a uma *sleep priority*
  - Relacionada com o dispositivo pelo qual ele está esperando
  - É uma prioridade de kernel (<50)
- Quando ele é acordado, o kernel seta o valor de *p_pri* para o mesmo valor da *sleep priority*
  - Oprocesso será escalonado na frente de outros processos de usuário e continuará a sua execução em modo kernel a partir do ponto de bloqueio.
- Quando a SVC é finalmente completada, imediatamente antes de retornar ao modo usuário, o kernel faz *p_pri = p_usrpri*

# Escalonamento Tradicional (cont.)

- O kernel do Unix tradicional é **não-preemptivo**
  - A chegada de um processo de mais alta prioridade na fila de prontos força a preempção do processo em execução SE ele está executando em *user mode*
  - Um processo executando em *kernel mode* nunca é preemptado (nunca perde a posse da CPU para um outro processo).
  - Quando o sistema for retornar para *user mode*, é verificado se há um processo mais prioritário na fila de prontos. Se sim, ocorre a preempção (troca de contexto).

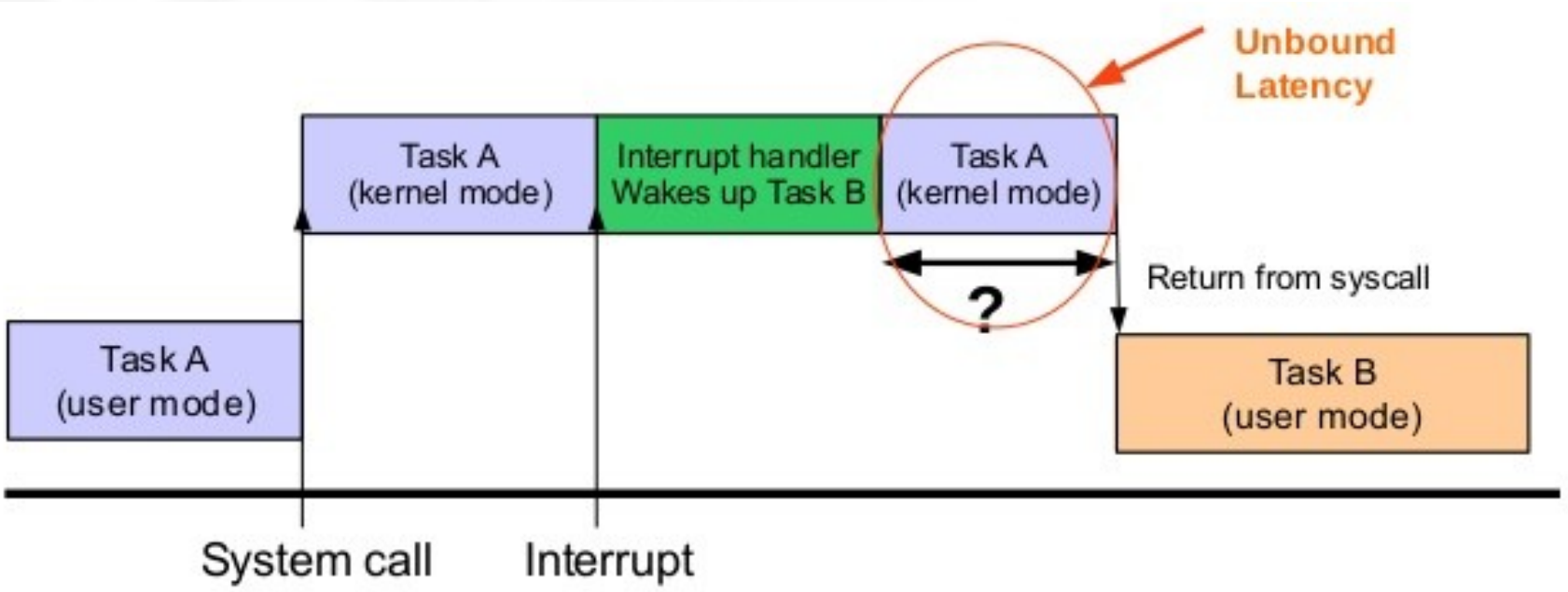
# Escalonamento Tradicional (cont.)

- Kernel não-preemptivo



# Escalonamento Tradicional (cont.)

- Kernel não-preemptivo



# Implementação BSD - Análise (1)

- Vantagens:
  - O algoritmo de escalonamento tradicional do Unix é simples e efetivo, sendo adequado para:
    - Sistemas de tempo compartilhado (*time sharing*)
    - Mix de processos interativos e *batch.*
  - Recomputação dinâmica das prioridades previne a ocorrência de *starvation*.
  - A abordagem favorece processos I/O bound, que requerem *bursts* de CPU pequenos e pouco freqüentes .

# Implementação BSD - Análise (2)

- Deficiências:
  - **Baixa escalabilidade**: se o número de processos é muito alto, torna-se ineficiente recalcular todas as prioridades a cada segundo;
  - Não existe a **garantia de alocação da CPU** para um processo específico ou então para um grupo de processos;
  - Não existe garantias de tempos de resposta para aplicações com característica de **tempo-real**.
  - Aplicações não podem controlar as suas prioridades.
    - O mecanismo de *nice* é muito simplista e inadequado.
  - Como o kernel é **não-preemptivo**, processos de maior prioridade podem ter que esperar muito tempo para ganhar a posse da CPU (problema da inversão)..

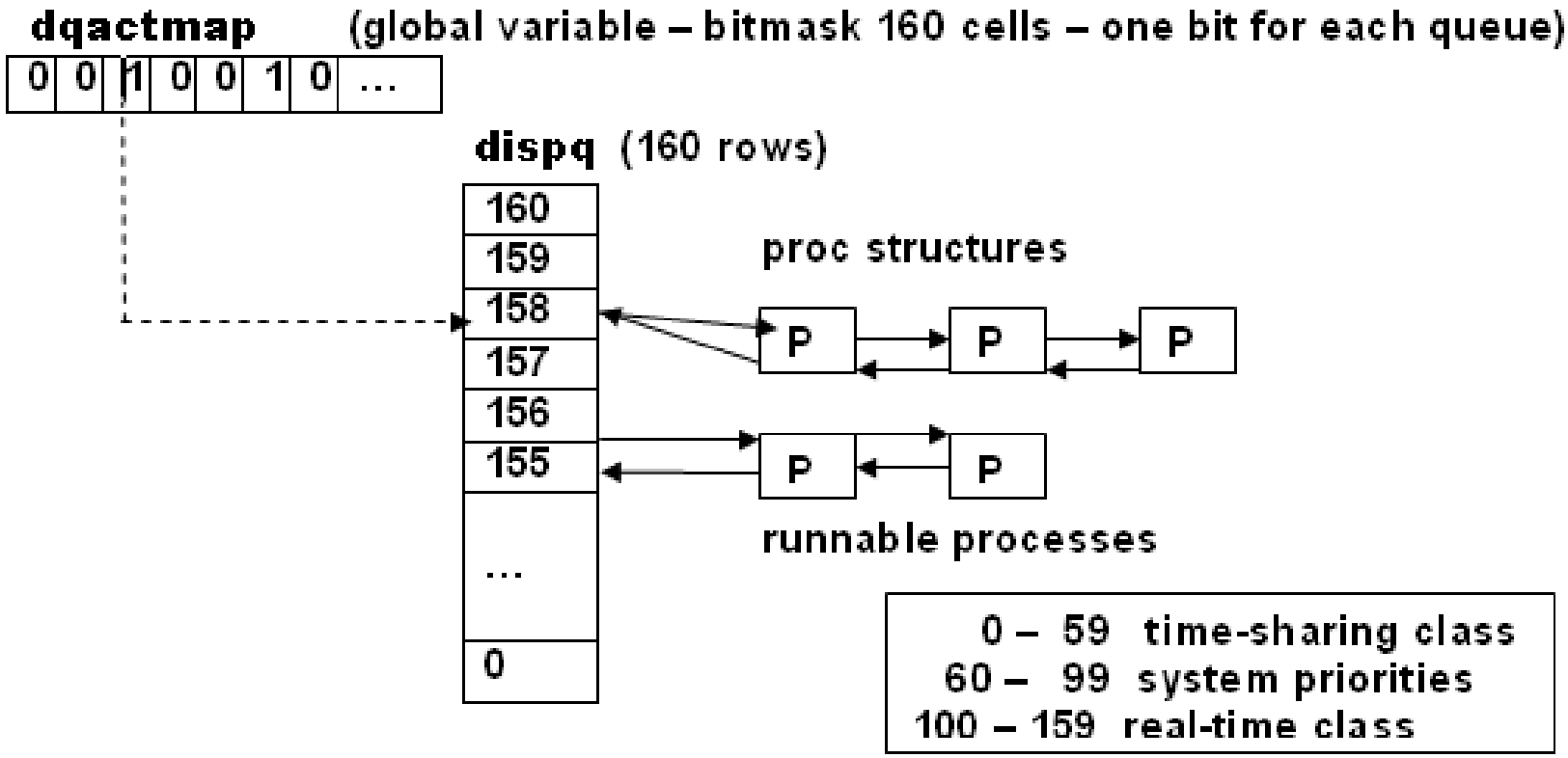
# Implementação - SVR4 (1)

- S.O. reprojetado
  - Orientação a objeto
- Objetivos de projeto do escalonador no SVR4:
  - Suportar mais aplicações, incluindo tempo-real
  - Permitir às aplicações maior controle sobre prioridade e escalonamento
  - Permitir a adição de novas políticas de uma forma modular
  - Limitar a latência de despacho para aplicações dependentes do tempo
- Classes de escalonadores
  - Classes oferecidas originalmente: time-sharing e tempo-real
  - É possível criar novas classes tratando outros tipos de processos

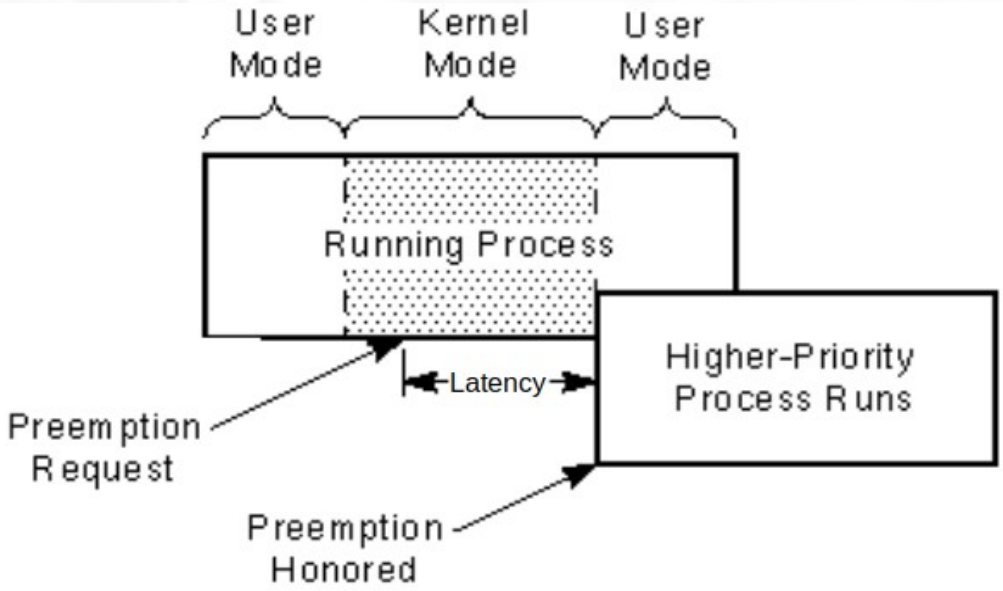
# Implementação - SVR4 (2)

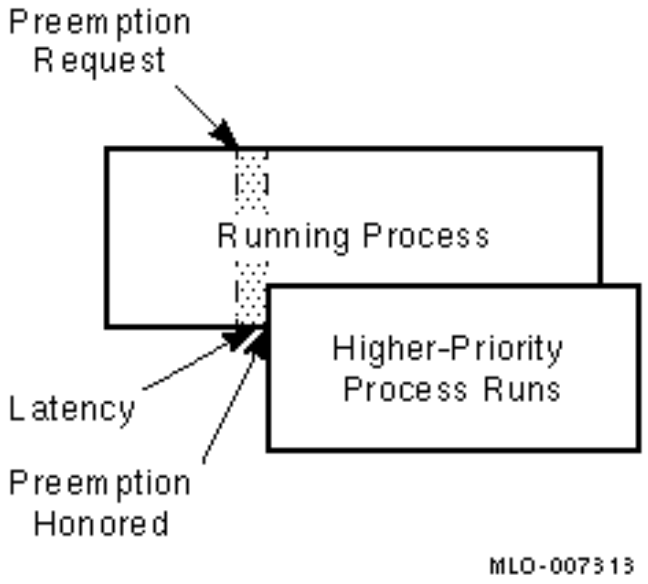
- Existem **rotinas independentes de classe** para fornecer:
  - Mudança de contexto
  - Manipulação da fila de processos
  - Preempção
- **Rotinas dependentes de classe**
  - Funções virtuais implementadas de forma específica por cada classe (herança)
  - Recomputação de prioridades
    - real-time class – prioridades e quanta fixos
    - time-sharing class – prioridades variam dinamicamente
      - Processos com menor prioridade têm maior quantum
      - Usa *event-driven scheduling*: prioridade é alterada na resposta a eventos.

# Implementação - SVR4 (3)



# Kernel preemptivo x Kernel não-preemptivo



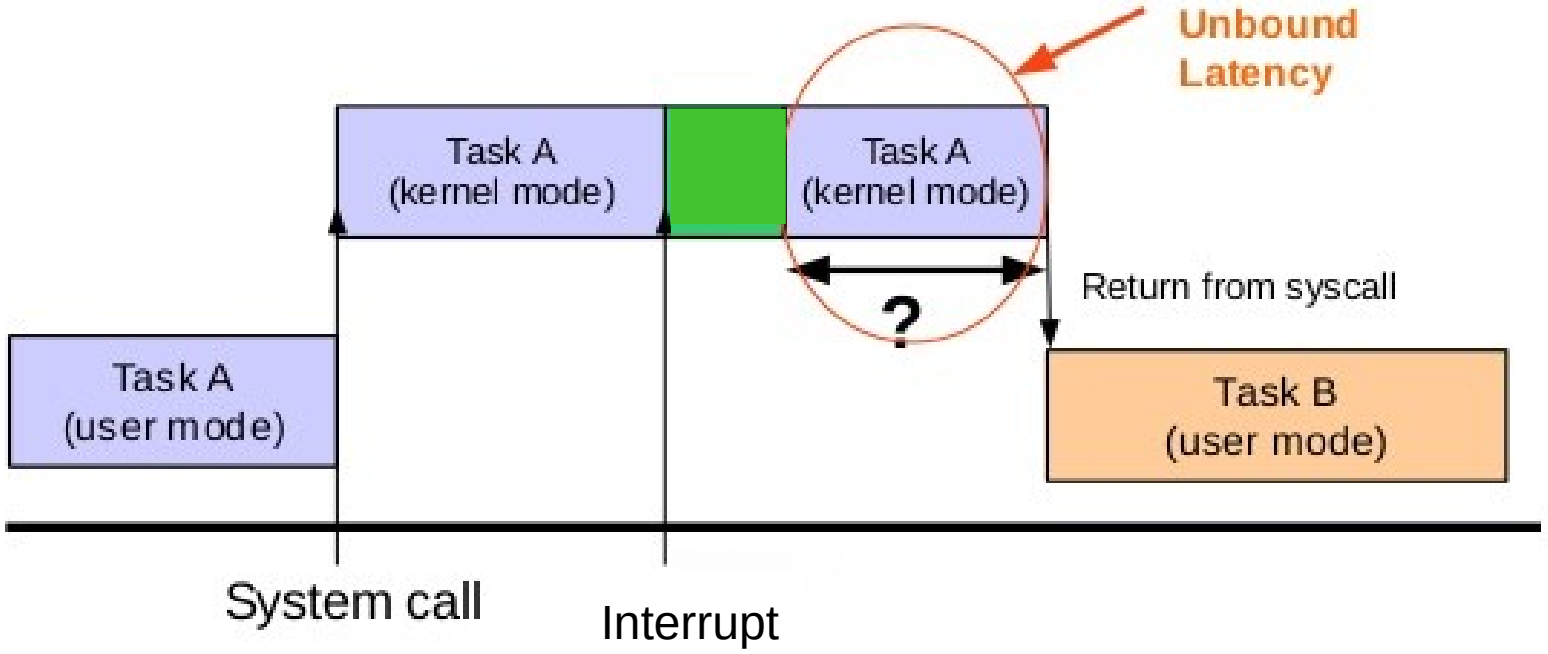



# Implementação - SVR4 (4)

- Processos de tempo real exigem tempos de resposta limitados
- **Preemption points** são definidos em pontos do kernel onde
  - Todas as estruturas de dados do kernel encontram-se estáveis
  - O kernel está prestes a iniciar alguma computação longa
- Em cada preemption point
  - O kernel verifica se um processo de tempo-real tornou-se pronto e precisa ser executado
    - O processo corrente é então preemptado
- Os limites nos tempos máximos que um processo de tempo-real precisa esperar são definidos pelo maior intervalo entre dois **preeption points** consecutivos

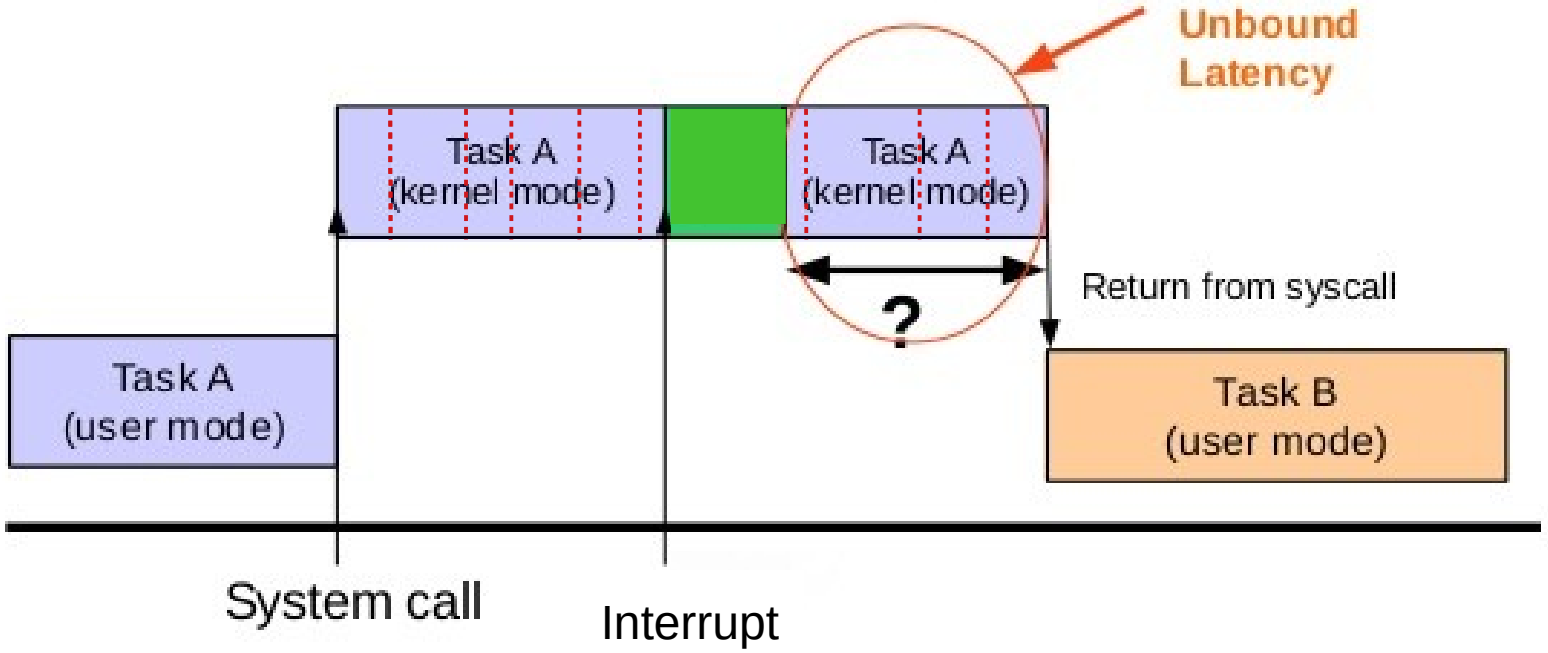
# Implementação - SVR4 (5)

- Latency of Non-Preemptive configuration



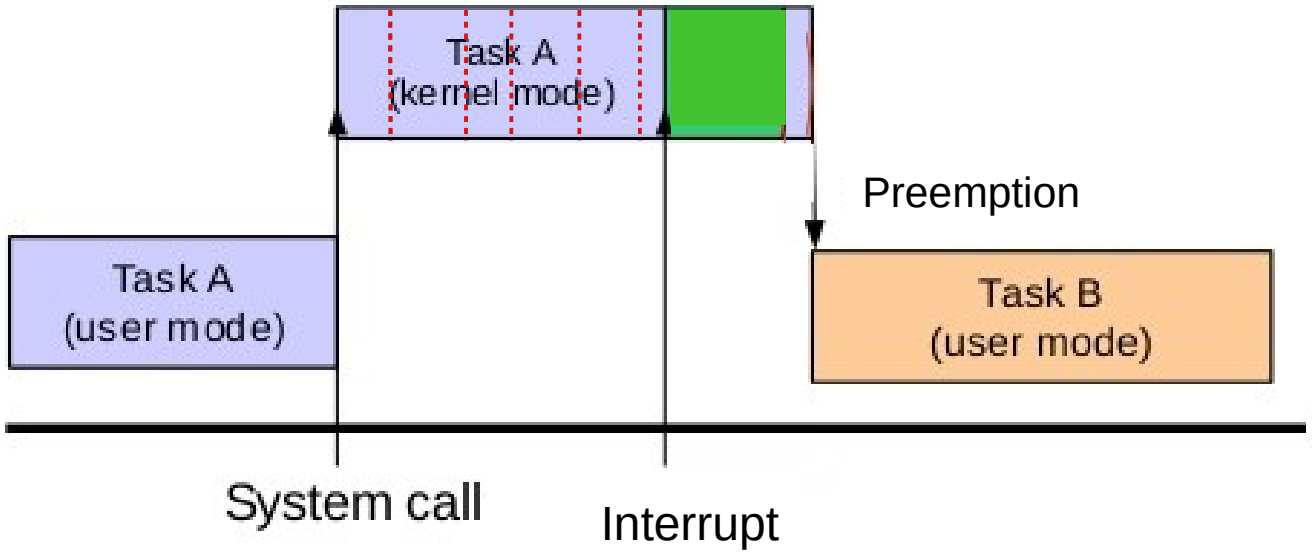
# Implementação - SVR4 (5)

- Latency of Non-Preemptive configuration



# Implementação - SVR4 (5)

- Latency of Non-Preemptive configuration



# E o LINUX?

- Processos de tempo real exigem tempos de resposta limitados
- **Preemption points** são definidos em pontos do kernel onde
  - Todas as estruturas de dados do kernel encontram-se estáveis
  - O kernel está prestes a iniciar alguma computação longa
- Em cada preemption point
  - O kernel verifica se um processo de tempo-real tornou-se pronto e precisa ser executado
    - O processo corrente é então preemptado
- Os limites nos tempos máximos que um processo de tempo-real precisa esperar são definidos pelo maior intervalo entre dois **preeption points** consecutivos

# POSIX 1003.1b Real-time extensions

- Priority Scheduling
- Real-Time Signals
- Clocks and Timers
- Semaphores
- Message Passing
- Shared Memory
- Asynchronous and Synchronous I/O
- Memory Locking

# Referências


- **VAHALIA, U. Unix Internals: the new frontiers. Prentice-Hall, 1996.**
  - **Capítulo 5 (até seção 5.5)**
- R. S. de Oliveira, A. S. Carissimi e S. S. Toscani, ''Sistemas Operacionais'', 4ª Edição (série didática da UFRGS), Editora Sagra-Luzzato, 2010.
  - Seção 4.5.5
- **REFERÊNCIAS EXTRA**
  - “A complete guide to Linux process scheduling”. Msc Thesis (2015)
    - https://trepo.tuni.fi/bitstream/handle/10024/96864/GRADU-1428493916.pdf
  - “Making Linux do hard real time”
    - https://events.static.linuxfound.org/sites/events/files/slides/rtmux_1.pdf