Laboratório de Pesquisa em Redes e Multimídia

Escalonamento no Unix

Universidade Federal do Espírito Santo
Departamento de Informática

Sistemas Operacionais

Implementação do escalonador

- O escalonador mantém um array (chamado "qs") de 32 filas (*run queues*).
- Cada fila corresponde a quatro prioridades adjacentes
  - Fila 0 é utilizada para as prioridades entre 0 e 3
  - Fila 1, prioridades entre 4-7, ...
- Cada fila é implementada através de uma lista duplamente encadeada de *proc structures*
  - qs[0] aponta para o primeiro elemento da fila 0
- Uma variável global "*whichqs*" contém uma máscara de bits, onde cada bit representa uma das 32 filas
  - Um bit *i* é setado se houver algum processo na fila *i*
- Apenas processos prontos para executar (*ready*) são mantidos nestas filas do escalonador

Implementação do escalonador (cont.)

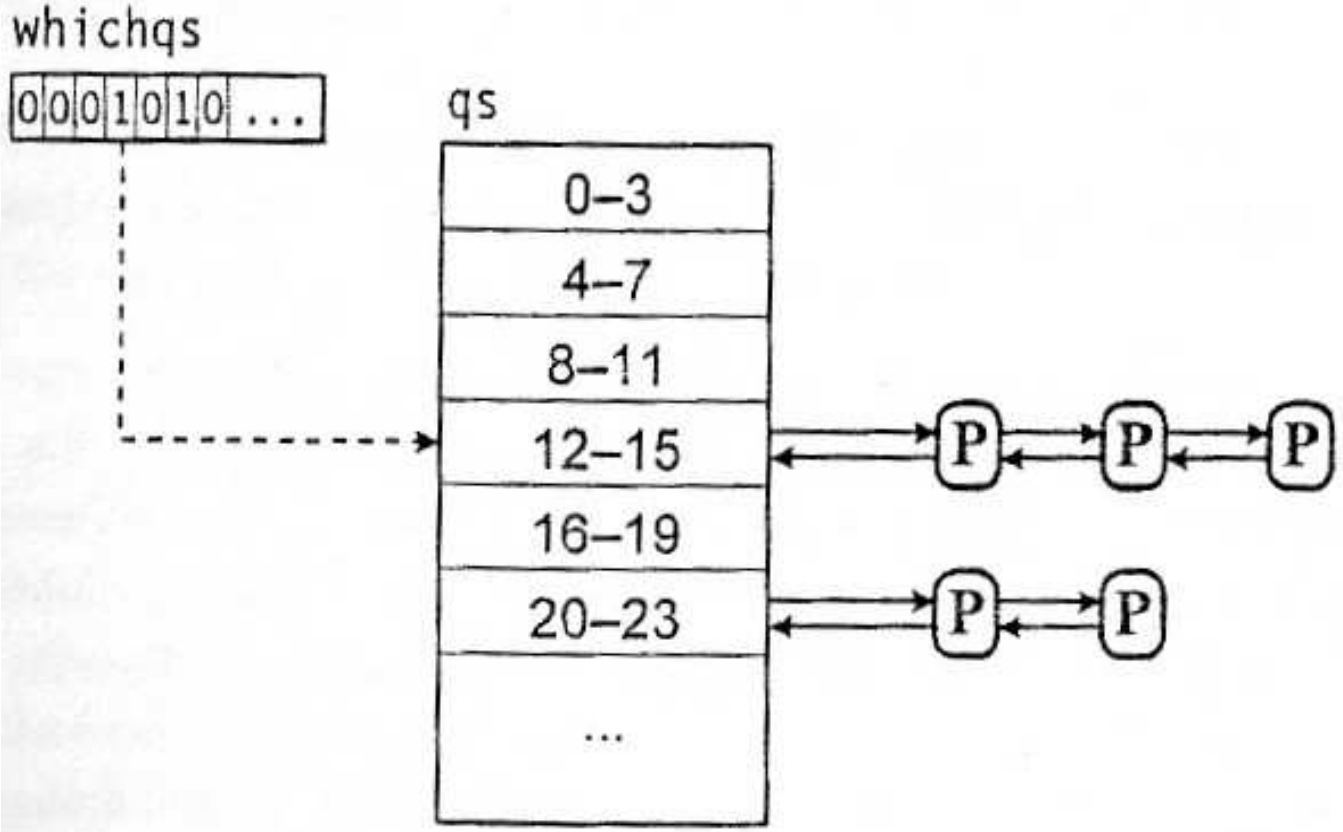


Figure 5-2. BSD scheduler data structures.

Implementação do escalonador (cont.)

- A rotina *swtch()* examina *whichqs* para encontrar o índice do primeiro bit "setado". Este índice corresponde à fila não vazia de maior prioridade.
- *swtch()* retira o primeiro processo da fila e realiza a devida troca de contexto. Para tal, é preciso acessar o PCB do processo, que contém o contexto de hardware do processo (registradores especiais e de uso geral).
  - O campo *p_addr* da *proc structure* aponta para as entradas da tabela de páginas referentes à *u area* do processo.
  - *swtch()* usa essa informação para acessar o contexto de hardware do processo.

Manipulação da *Run Queue* (1)

- O processo de maior prioridade é sempre aquele que detém a posse da CPU, a menos que o processo corrente já esteja executando em *kernel mode*.
  - Pode ser que quando um processo encontra-se no estado *kernel running* exista um outro processo em uma *run queue* de maior prioridade!
  - Razão: o kernel é não-preemptivo!
- Cada processo recebe um quantum fixo (100 ms no 4.3BSD)
  - A cada 100ms, ou seja, a cada 10 *CPU ticks*, o kernel invoca, via *callout*, a rotina *roundrobin()* para escalonar o próximo processo da mesma fila de onde saiu o processo corrente.
  - Se um processo mais prioritário entra em uma *run queue*, ele será escalonado antes, sem ter que esperar por *roundrobin().*
  - Se não houver mais nenhum processo na mesma fila de onde saiu o processo corrente (i.e., se existirem processos somente em filas de menor prioridade) o processo continua sendo executado, mesmo que o seu quantum expire.

Manipulação da *Run Queue* (2)

- Recálculo das prioridades:
  - Rotina *schedcpu()* recomputa as prioridades de todos os processos a cada segundo (ou seja, a cada 100 *CPU ticks*).
  - O *Clock Interrupt Handler*, por sua vez, recalcula a prioridade do processo corrente a cada 4 *CPU ticks*.
- Quatro situações em que pode ocorrer troca de contexto:
  - O quantum do processo corrente expirou
  - O recálculo de prioridades resulta em um outro processo se tornando mais prioritário.
  - O processo corrente (ou algum *Interrupt Handler*) acorda um processo mais prioritário (troca involuntária de contexto).
  - O processo corrente bloqueia ou finaliza (nesse caso, ocorre uma troca de contexto voluntária).
    - O kernel chama *swtch()* de dentro de *exit()* ou *sleep()*

Manipulação da *Run Queue* (3)

- Quando o sistema está em *kernel mode*, o processo corrente não pode ser preemptado imediatamente.
- O kernel liga um *flag* (*runrun*), indicando que existe um processo mais prioritário esperando para ser escalonado.
- Quando o processo está prestes a entrar em *user mode*, o kernel examina *runrun*. Se "setada", então o kernel transfere o controle para a rotina *switch()*, que inicia a troca de contexto.

Análise Final (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 .

Análise Final (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 possuem controle sobre as próprias prioridades. O mecanismo de *nice* é muito simplista e inadequado.
  - Como o kernel é **não preemptivo**, processos de maior prioridade podem ter que esperar um tempo significativo para ganhar a posse da CPU mesmo após terem sido feitos *runnnable* (isso é chamado de ***problema da inversão***)..

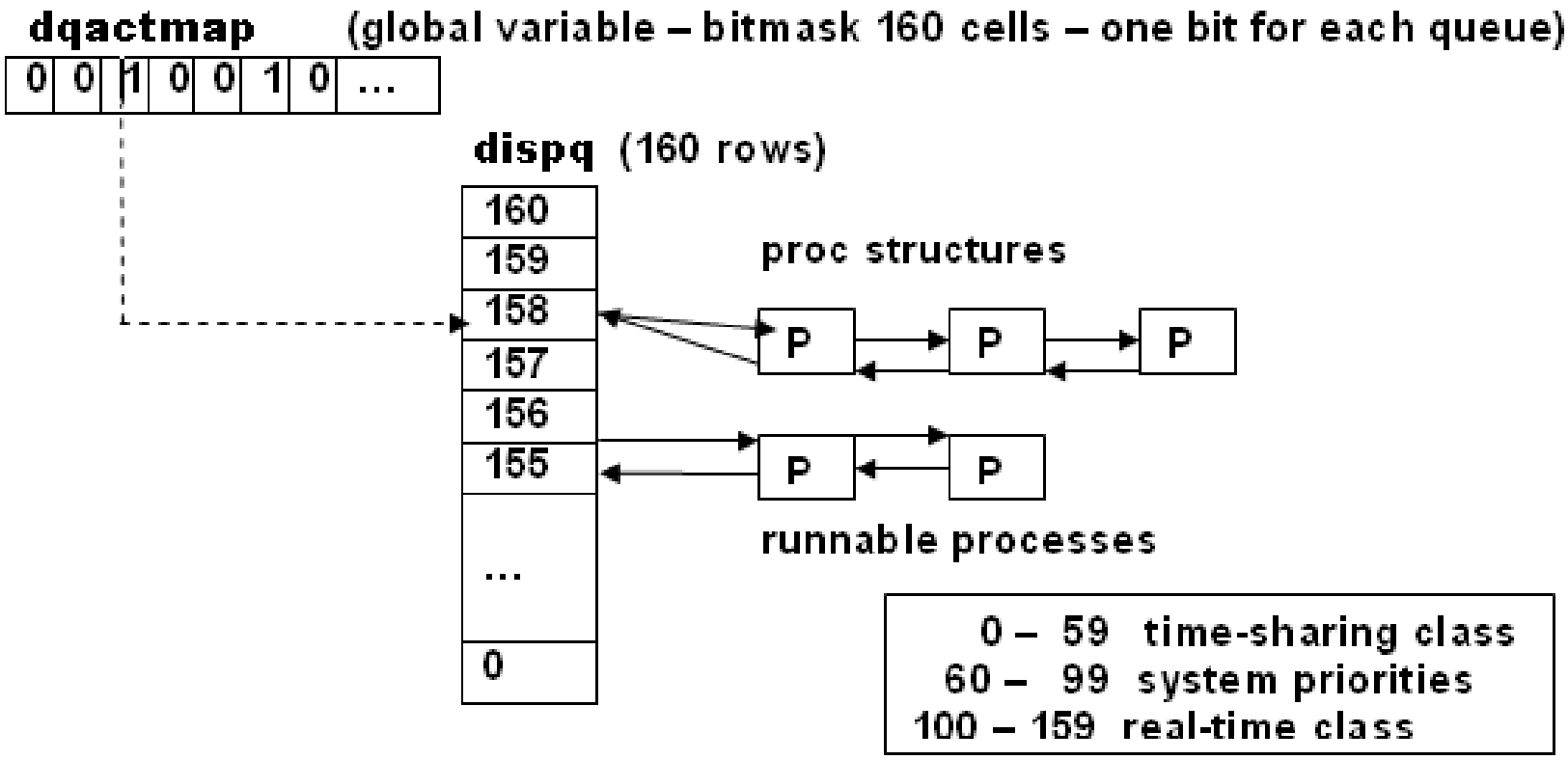
Escalonamento no SVR4 (1)

- 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

Escalonamento no SVR4 (2)

- Existem **rotinas independentes de classe** para fornecer:
  - Mudança de contexto
  - Manipulação da fila de processos
  - Preempção
- **Rotinas dependentes da 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.

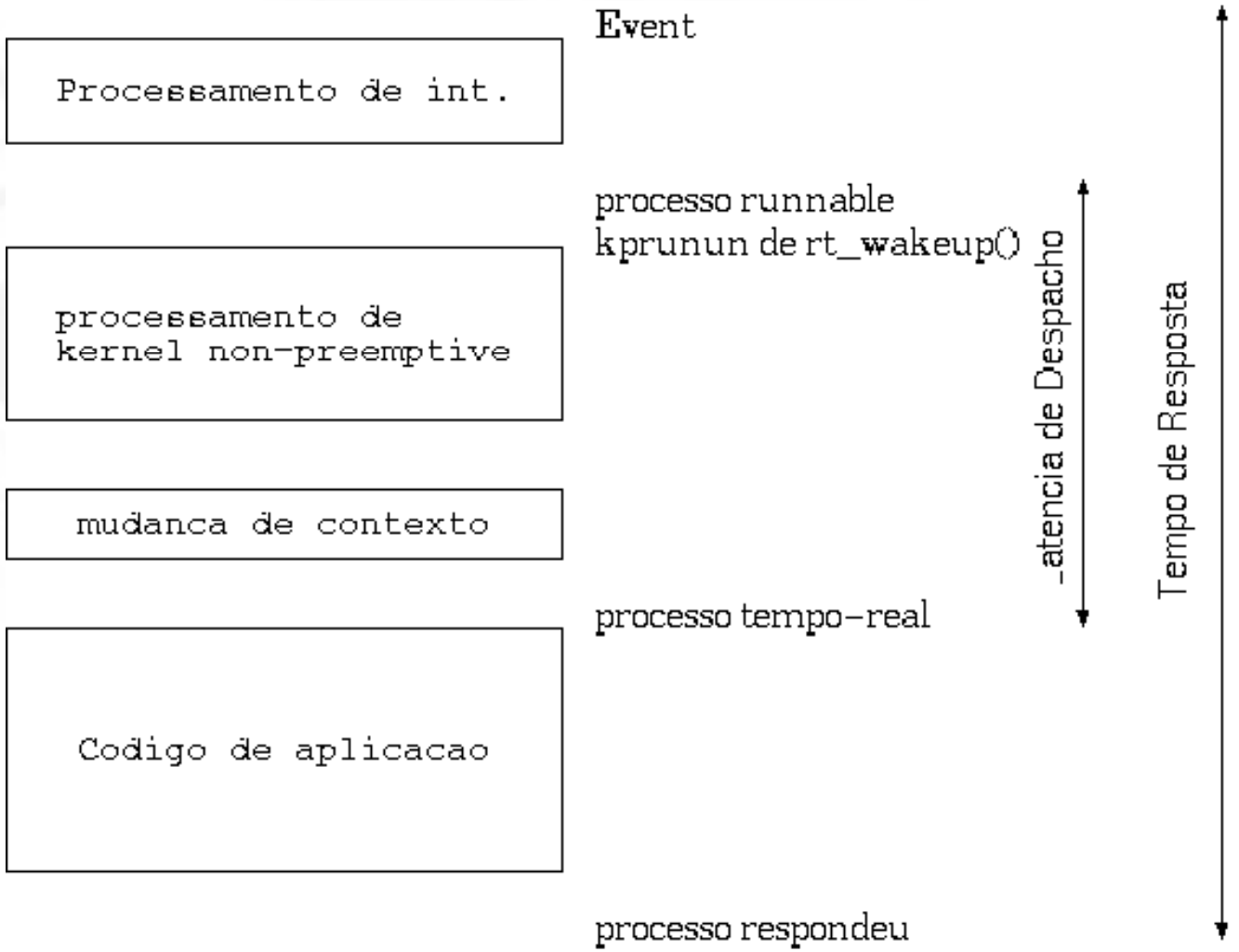
Escalonamento no SVR4 (3)



Escalonamento no 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 a flag *kprunrun*... caso ela esteja "setada":
    - Isto significa que um processo de tempo-real tornou-se pronto e precisa ser executado
    - O processo 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

Escalonamento no SVR4 (5)



Exercício

decay = ½ , p_nice = 0

p_usrpri	p_cpu	p_usrpri	p_cpu	p_usrpri	p_cpu
50	0 1 ... 100	50	0	50	0
62	50	50	0 1 ... 100	50	0
56	25	62	50	50	0 1 ... 100
53	12 13 ... 112	56	25	62	50
64	56	53	12 13 ... 112	56	25
57	28	64	56	53	12

decay

50 + p_cpu/4

Observações:

- Para sistemas muito carregados, o p_cpu se torna muito pequeno para cada processo

- O fator decay ainda faz com que ele caia mais ainda

-Fator decay dependente da carga do sistema, como implementado no 4.3BSD, é melhor do que valor fixo, como observado no SVR3.

- No caso do BSD, quanto maior a carga, menor o decaimento (decay tende a 1) => As prioridades dos processos recebendo CPU cycles decai mais rapidamente

Referências

- VAHALIA, U. Unix Internals: the new frontiers. Prentice-Hall, 1996.
  - Capítulo 5 (até seção 5.5)