Laboratório de Pesquisa em Redes e Multimídia

Escalonamento no Unix

Universidade Federal do Espírito Santo
Departamento de Informática

Sistemas Operacionais

Introdução (1)

- Unix é um sistema de tempo compartilhado:
  - Permite que vários processos sejam executados concorrentemente.
  - Processos ativos competem pelos diversos recursos do sistema (memória, periféricos e, em particular, a CPU).
- A concorrência é emulada intercalando processos com base na atribuição de uma fatia de tempo (time slice ou quantum). Cada processo que recebe a CPU é executado durante a fatia de tempo a ele atribuída.
- O escalonador (*scheduler*) é o componente que determina qual processo receberá a posse da CPU em um determinado instante.

Introdução (2)

- 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* (um processo não deve ficar eternamente esperando pela posse da CPU).
- Os objetivos acima podem ser conflitantes!
  - O escalonador deve prover uma maneira eficiente de balancear esses objetivos, minimizando overhead.

Introdução (3)

- Troca de Contexto (*Context Switch*)
  - Ocorre sempre que o escalonador decide entregar a CPU a um outro processo.
  - Operação custosa
    - O contexto de execução de hardware é salvo no respectivo PCB (*u area*).
    - Os valores dos registradores do próximo processo a ser executado são carregados na CPU
    - Adicionalmente, tarefas específicas a cada arquitetura de hardware podem ser realizadas. Exemplos :
      - Atualizar *cache* de dados, instruções e tabela de tradução de endereços
      - Se houver *pipeline* de instruções, ele deve ser "esvaziado"
  - Em resumo, esses fatores confirmam o alto custo de uma operação de troca de contexto.
  - Isso pode ter influência na implementação ou até mesmo na própria escolha da política de escalonamento a ser adotada.

Rotina de Interrupção do Relógio (1)

- O relógio (*clock*) é um elemento de hardware que interrompe a CPU em intervalos de tempos fixos.
  - *CPU tick*, *clock tick* ou *tick*.
  - Algumas máquinas requerem que o S.O. rearme o relógio após cada interrupção; em outras, o relógio rearma-se sozinho.
- A maioria dos computadores suporta uma variedade de intervalos:
  - Unix tipicamente configura o *CPU tick* em 10 ms.
  - A constante HZ (definida no arquivo param.h) armazena a freqüência desejada para o *CPU tick*.
    - Ex: HZ=100 implica em um intervalo de *tick* de 10 ms.
- As funções do kernel são usualmente medidas em números de *CPU ticks* ao invés de segundos ou milisegundos.

Rotina de Interrupção do Relógio (2)

- A interrupção de relógio é a mais prioritária, só perdendo para a interrupção de falha de energia.
- Uma "Rotina de Interrupção do Relógio" (*Clock Interrupt Handler*) é executada em resposta a uma interrupção do relógio
  - Deve ser **rápida** e suas tarefas devem ser mínimas
- Tarefas típicas do *Clock Interrupt Handler*:
  - Reiniciar o relógio, se necessário
  - Atualizar as estatísticas de uso da CPU p/ o processo corrente
  - Realizar funções relacionadas ao escalonamento (recalcular prioridades, tratar evento de fim de fatia de tempo do processo corrente, etc.)
  - Manipular sinais (ex: enviar o sinal SIGXCPU para o processo corrente, caso ele tenha excedido a sua cota de uso de CPU)
  - Atualizar o *time-of-day* e outros *timers* relacionados
  - Acordar processos de sistema quando apropriado (ex: *swapper* e *pagedaemon*)
  - Manipular *callouts* e alarmes (vide adiante)

Rotina de Interrupção do Relógio (3)

- Algumas das tarefas do *Clock Interrupt Handler* não precisam ser efetuadas necessariamente a cada *tick*, podendo ser realizadas em intervalos de tempo maiores, múltiplos de um *tick*.
- A maioria dos sistemas Unix materializa esse conceito de intervalos maiores através da definição de um *major tick.*
- Um *major tick* ocorre então a cada "*n*" *CPU ticks*. Certas tarefas somente podem ser executadas nos *major ticks*.
  - Ex: 4.3BSD recomputa as prioridades a cada 4 ticks (n = 4)
  - Ex: SVR4 manipula alarmes e acorda processos de sistema a cada segundo, se necessário (n = 10)

Callouts (1)

- São registros de funções que devem ser invocadas pelo kernel num tempo futuro (i.e., após decorridos um número *n* de *CPU ticks*).

- Definição de um *callout* no Unix SVR4:

```
int to_ID =  timeout (void (*fn)(), caddr_t arg, long delta);
```

  fn() – função do kernel a ser invocada ("*callout funtion*")
  arg – argumento a ser passado para fn()
  delta – intervalo de tempo definido em número de *CPU ticks*

- Para cancelar um *callout* deve-se fazer referência ao identificador usado na sua criação:

```
void untimeout (int to_ID);
```

- *Callout* são invocadas pelo kernel em *system context*:
  - fn() não pode bloquear o processo em execução
  - fn() não pode acessar o espaço de endereçamento do processo corrente

Callouts (2)

- *Callouts* são adequados para a realização de tarefas periódicas:
  - Retransmissão de pacotes em protocolos de comunicação
  - Algumas funções do escalonador e do gerente de memória
  - *Polling* de dispositivos periféricos que não suportam interrupção
- *Callouts* são consideradas operações normais de kernel
  - A *Rotina de Interrupção do Relógio* não invoca diretamente os *callouts*. A cada *CPU tick*, ela verifica se existe algum *callout* a executar.
    - Se SIM, ela "seta" um *flag* indicando que o *callout handler* deve ser executado
  - Quando o sistema retorna à sua prioridade base de interrupção ele verifica esta *flag* e:
    - Se *flag* setado então o kernel invoca o *callout handler*
    - O *callout handler*, por sua vez, invoca cada *callout function* que deve ser executado
- Cada *callout* só será efetivamente executado quando todas as interrupções pendentes forem atendidas.

Callouts (3)

- *Callouts* pendentes podem ser ordenados por "tempo até disparar", como no caso do Unix 4.3BSD.

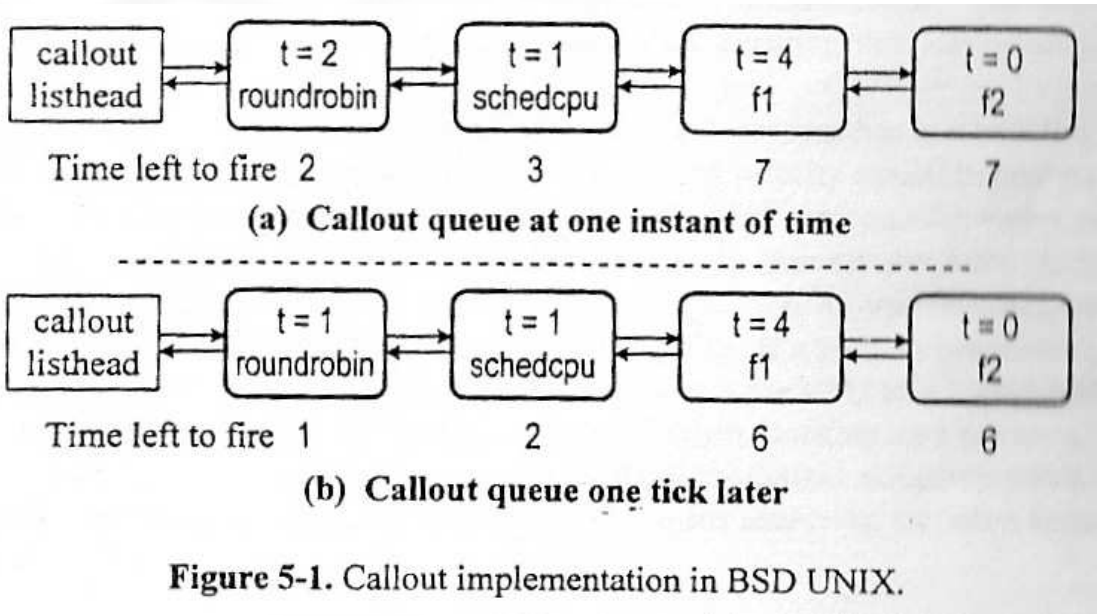


Figure 5-1. Callout implementation in BSD UNIX.

roundrobin(): buscar outro processo com a mesma prioridade
schedcpu(): recomputar a prioridade.

Alarmes (1)

- Processos podem solicitar ao kernel que este lhes envie um alarme (um sinal) após transcorrido um certo intervalo de tempo.
- O Unix provê três tipos de alarmes:
  - *Real-time alarm*, *profiling alarm* e *virtual-time alarm*
- *Real time alarm*
  - Mede o tempo real decorrido
  - O kernel notifica o processo requisitante através do sinal SIGALRM
- *Profiling alarm*
  - Mede a quantidade de tempo que um processo está sendo executado (i.e., o seu tempo corrente de execução).
  - O kernel notifica o processo através do sinal SIGPROF
- *Virtual-time alarm*
  - Monitora o tempo gasto pelo processo em *user mode*
  - O kernel notifica o processo através do sinal SIGVALRM

Alarmes (2)

- Unix BSD
  - SVC: setitimer()
    - Permite a um processo solicitar qualquer tipo de alarme
    - Intervalos em microsegundos, convertido internamente pelo kernel em número de CPU ticks
- Unix System V
  - SVC: alarm()
    - Processo pode invocar apenas apenas o *real-time alarm*
    - Intervalo de tempo em segundos
- Unix SVR4
  - SVC: hrtsys()
    - Intervalo pode ser especificado em microsegundos (maior resolução)

Alarmes (3)

- A alta resolução dos alarmes de tempo real não implica necessariamente em uma alta precisão
  - O kernel envia o sinal SIGALRM ao processo (o que fica registrado nas suas estruturas de controle)
  - O processo não vai responder ao sinal até que ele seja escalonado para executar (entre no estado *running*)
- *Timers* de alta resolução quando utilizados por processos com alta prioridade são menos susceptíveis a atrasos de escalonamento
  - Se o processo estiver executando em *kerne mode* o problema persiste

Objetivos do Escalonador (1)

- Sendo um sistema **de tempo compartilhado**, o Unix deve alocar a CPU de maneira **justa** para todos os processos.
- Naturalmente, quanto maior for a **carga no sistema, menor** será a **porção de CPU** dada a cada processo e, portanto, eles executarão mais lentamente
- É função do escalonador **garantir** uma boa **performance** para todos os processos, considerando todas as expectativas de carga do sistema.
- Aplicações que concorrem pela CPU podem apresentar **características diversas** entre si, implicando em **diferentes requisitos** de escalonamento.
  - Ex: aplicações interativas, aplicações *batch* e aplicações de tempo real

Objetivos do Escalonador (2)

- Processos interativos (*shells*, editores, interfaces gráficas, etc.)
  - Passam grande parte do tempo esperando por interações
  - Interações devem ser processadas rapidamente; do contrário, os usuário observarão um alto tempo de resposta.
  - Requisito: reduzir os tempos médios (e a variância) entre uma ação de usuário e a resposta da aplicação de forma que o usuário não detecte/perceba este atraso (50-150 ms)
- Processos *batch* (que rodam em *background*, sem interação com o usuário)
  - Medida da eficiência do escalonamento: 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.
- Processos de tempo real
  - 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.

Objetivos do Escalonador (3)

- as funções do kernel – tais como gerência de memória, tratamento de interrupções e gerência de processos – devem ser executadas prontamente sempre que requisitadas.
- Num S.O. bem comportado:
  - Todas as aplicações devem sempre progredir
  - Nenhuma aplicação deve impedir que as outras progridam, exceto os casos em que o usuário explicitamente permita isso
  - O sistema deve ser sempre capaz de receber e processar entradas interativas de usuário para que o mesmo possa controlar o sistema
    - Ex: "Ctrl+Alt+Del"

Escalonamento Tradicional (1)

- Usado nos antigos sistemas Unix SVR3 e 4.3BSD. Projetado para lidar com os seguintes requisitos:
  - Tempo compartilhado
  - Ambientes interativos
  - Processos *background* (batch) e *foreground* rodando simultaneamente
- A política de escalonamento adotada objetiva:
  - Melhorar os tempos de resposta para os usuários interativos, e
  - Garantir, ao mesmo tempo, que processos *background* não sofram *starvation*
- O escalonamento tradicional do Unix é baseado em prioridades dinâmicas
  - A cada processo é atribuída uma prioridade de escalonamento, que é alterada com o passar do tempo.
  - O escalonador sempre seleciona o processo com a prioridade mais alta dentre aqueles no estado pronto-para-execução (*ready/runnable process*).

Escalonamento Tradicional (2)

- 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.
- Esse esquema previne a ocorrência de *starvation*.

- O valor da prioridade de um processo é calculado com base em dois fatores:
  - *Usage factor* – é uma medida do padrão de uso recente da CPU pelo processo.
  - *Nice value* – valor numérico relacionado ao uso da SVC *nice*.
- É a alteração dinâmica do *usage factor* que permite ao kernel variar dinamicamente a prioridade do processo

Escalonamento Tradicional (3)

- O escalonador tradicional usa o esquema de "*preemptive round robin*", isto é, escalonamento circular com preempção, para aqueles processos com a mesma prioridade.
- O *quantum* possui um valor fixo, tipicamente de 100 ms.
- A chegada de um processo de mais alta prioridade na fila de prontos força a preempção do processo executando em *user mode* (força uma troca de contexto), mesmo que este último não tenha terminado o seu quantum.
- O kernel do Unix tradicional é não-preemptivo.
  - Um processo executando em *kernel mode* nunca é interrompido por um outro processo.
- Um processo pode voluntariamente perder a posse da CPU ao bloquear-se esperando por algum recurso
- Pode haver preempção do processo quando este retorna de *kernel mode* para *user mode*.

Prioridades dos Processos (1)

- 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*)
- Por ordem decrescente de prioridade...
  - Swapper
  - Controle de dispositivos de E/S orientados a bloco
  - Manipulação de arquivos
  - Controle de dispositivos de E/S orientados a caractere
  - Processos de usuário

Prioridades dos Processos (2)

- 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* controlável pelo usuário (SVC *nice*)
- O escalonador usa o campo *p_pri* para decidir qual processo escalonar. Se o processo roda em modo usuário então *p_pri = p_usrpri*
- Quando um processo acorda após bloquear em uma SVC, *p_pri* é aumentado, recebendo o valor da prioridade de kernel referente ao motivo de espera – *sleep priorities*).

Prioridades dos Processos (3)

- *Sleep priority*
  - Após um processo ter sido bloqueado (por exemplo, dentro de uma SVC), quando ele for acordado o kernel seta o valor de *p_pri* como sendo o mesmo valor da prioridade do recurso/evento pelo qual o processo esteve esperando
    - Ex: 28 para terminais, 20 para disco
  - Com a prioridade (maior) de kernel recém adquirida o processo 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*, restaurando o valor original da prioridade do processo em modo usuário (antes dele ter feito a SVC).

Prioridades dos Processos (4)

- A prioridade do processo em modo usuário (*p_usrpri*) depende de dois fatores:
  - Uso recente da CPU (*p_cpu*)
  - Fator *p_nice*=[0-39], que é controlado pelo usuário (via SVC *nice()*)
    - Default: *p_nice* = 20
    - Quanto MAIOR o *p_nice*, MAIOR o *p_usrpri* (=> menor prioridade).
    - O comando *nice(x)* : -20 ≤ x ≤ +39 **=>** p_nice = p_nice + x
      - Usuários comuns só podem chamar o comando nice() com valores positivos (o que diminui a prioridade final do processo).
      - Comando nice com valores negativos (que aumentam o valor de *p_usrpri*, diminuindo a prioridade final do processo), somente pelo *superuser*.
        - *normal user*: Δ↑ *p_nice* ➔ Δ↓ prioridade
        - *superuser*: Δ↓ *p_nice* ➔ Δ↑ prioridade
    - Processos *background* recebem automaticamente grandes valores de *p_nice* (por isso são menos prioritários).

Prioridades dos Processos (5)

- Algoritmo
  - Na criação do processo: *p_cpu* = 0
  - A cada *tick*: rotina de interrupção do relógio incrementa *p_cpu* do processo em execução (até o máximo de 127), de modo a refletir o seu uso relativo de CPU.
  - A cada 100 *CPU ticks* (isto é, a cada segundo): via *callout*, o kernel invoca a rotina *schedcpu()*, que reduz o *p_cpu* de todos os processos de um "fator de decaimento" (*decay factor*) e recomputa a prioridade *p_usrpri* de todos os processos segundo a fórmula:

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

  - PUSER = 50, que é a prioridade base dos processos de usuário
- No SVR3
  - *Decay* possui valor fixo igual a ½.
- No 4.3BSD
  - Fórmula usada: $$decay = \frac{2 \times load\_average}{(2 \times load\_average + 1)}$$
  - *load_average*: número médio de processos aptos a executar (*ready*) no último segundo.

Prioridades dos Processos (6)

- Observações:
  - Quanto mais um processo ocupa a CPU, mais seu *p_cpu* aumenta, maior será seu *p_usrpri* e, consequentemente, menor será a sua prioridade.
  - Quanto mais um processo espera para ser escalonado, menor será o seu *p_cpu* e, consequentemente, menor será o de *p_usrpri.* Com isso, a sua prioridade diminui menos do que os processos que usam muito a CPU e que tiveram um maior aumento de *p_cpu*.
- Em função do recálculo de prioridades feito pela rotina *schedcpu()* pode haver troca de contexto
  - Isto acontece quando o processo em execução fica com prioridade mais baixa do que qualquer outro processo pronto para executar (*ready*), considerando-se os novos valores de *p_usrpri* de todos os processos.
  - Isso previne *starvation* e favorece processos *I/O bound*.

Prioridades dos Processos (7)

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

Implementação do escalonador (1)

- 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 (2)

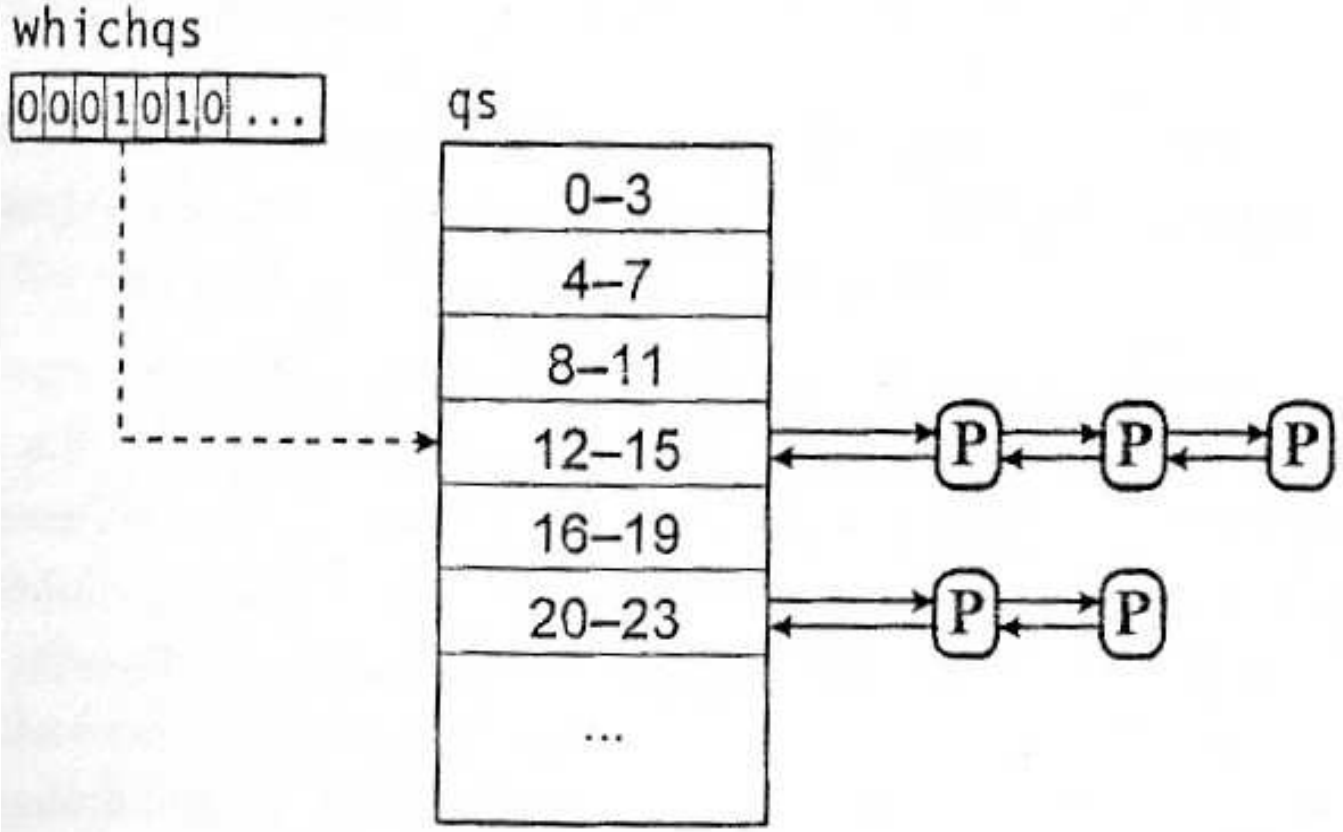


Figure 5-2. BSD scheduler data structures.

Implementação do escalonador (3)

- 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 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***)..

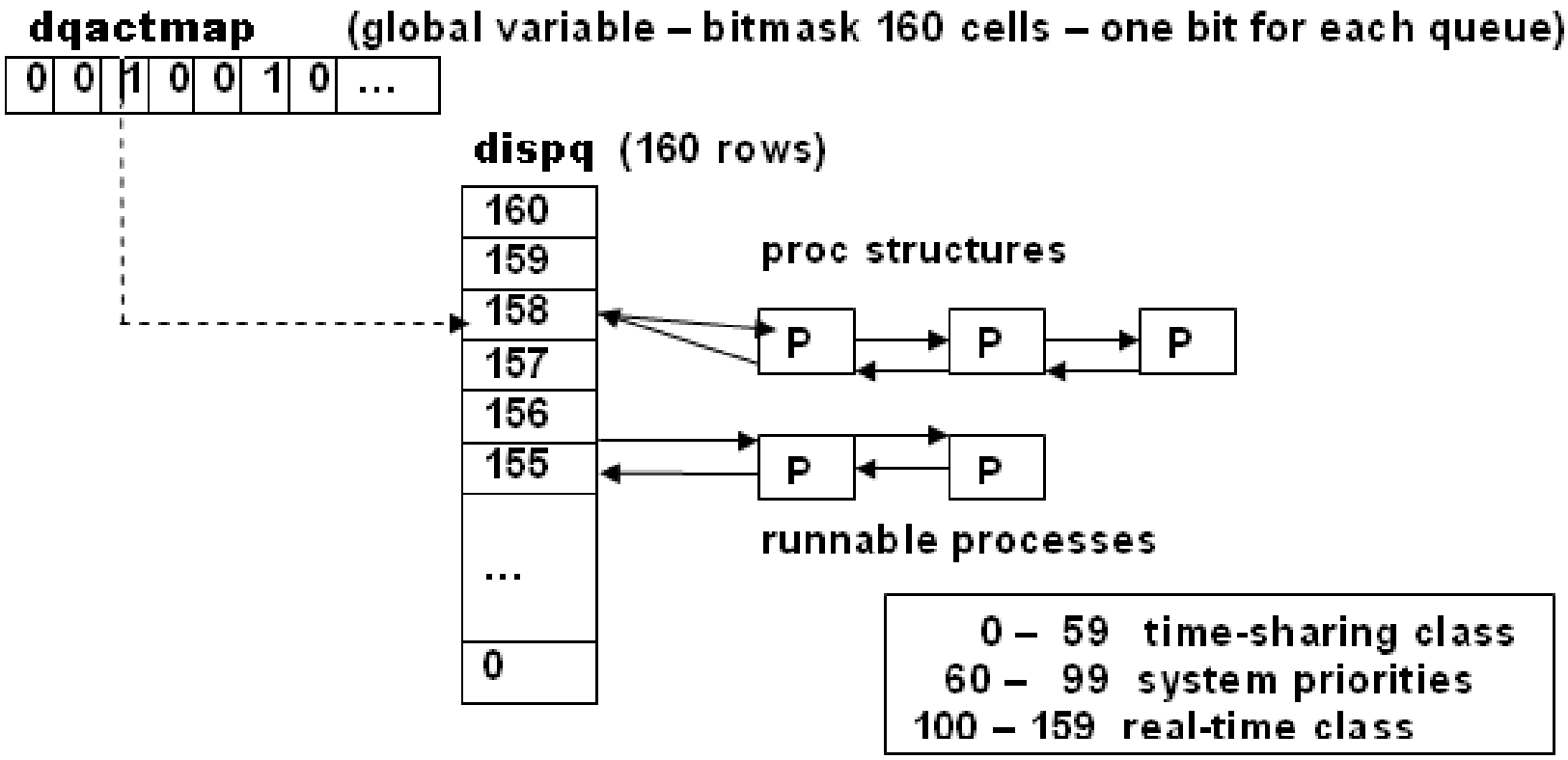
Escalonamento no SVR4 (1)

- 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.

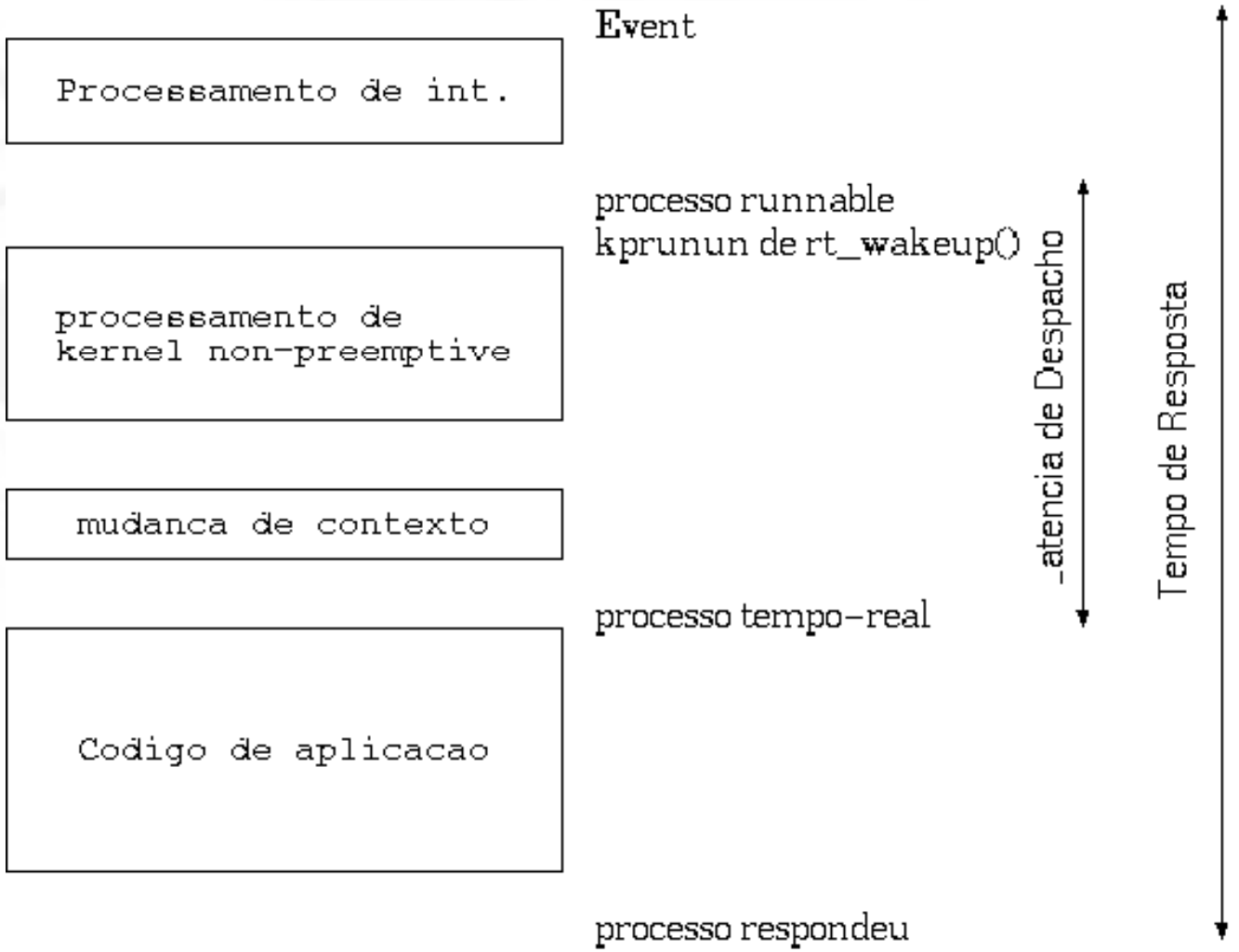
Escalonamento no SVR4 (3)



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)



Referências

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