Laboratório de Pesquisa em Redes e Multimídia

Escalonamento no Unix

Universidade Federal do Espírito Santo
Departamento de Informática

Sistemas Operacionais

Introdução

- Unix é um S.O. multiprogramado de tempo compartilhado:
- A concorrência é emulada intercalando processos com base na atribuição de uma fatia de tempo (*time slice* ou *quantum*).
- O escalonador (*scheduler*) determina qual processo receberá a posse da CPU em um dado instante.
- 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”

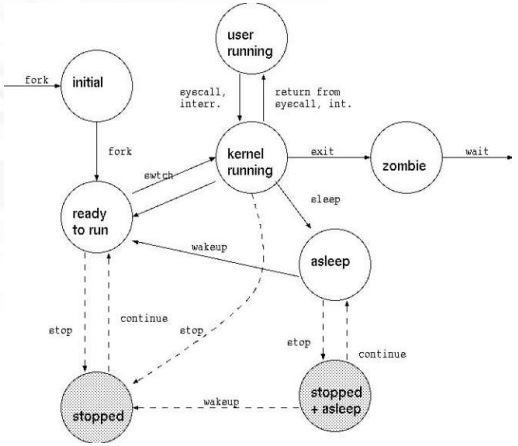
Projeto do Escalonador

- 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* (um processo não deve ficar eternamente esperando pela posse da CPU).
- Os objetivos acima podem ser conflitantes!
  - Aplicações que concorrem pela CPU podem apresentar características diversas

Objetivos do Escalonador (cont.)

- Processos interativos (*shells*, editores, interfaces gráficas, etc.)
  - I/O BoundPassam grande parte do tempo esperando por interações
  - Requisito: reduzir os tempos de resposta médios e a variância (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 (*turnarround*).
- Processos de tempo real
  - Requisito: limites garantidos nos tempos de resposta.
  - Ex: aplicações de vídeo.
- As funções do kernel – tais como gerência de memória, tratamento de interrupções e gerência de processos
  - Devem ser executadas prontamente, sempre que requisitadas.

Máquina de Estados do Unix



Modos de Operação da CPU

- Finalidade principal: proteção!
- 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.
- Processos de usuário rodam em *user mode*; logo, não podem – acidental ou maliciosamente –, corromper outro processo ou mesmo o kernel.
- Processos do S.O. rodam em kernel mode.

Custo da Troca de Contexto

- O projeto do escalonador deve considerar que a traca de contexto é uma operação custosa:
  - O contexto 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, por exemplo: atualizar cache de dados, instruções e tabela de tradução de endereços; se houver pipeline de instruções, ele deve ser "esvaziado", etc.
- Esses fatores podem influenciar a implementação ou até mesmo na própria escolha da política de escalonamento a ser adotada.

Escalonamento Tradicional

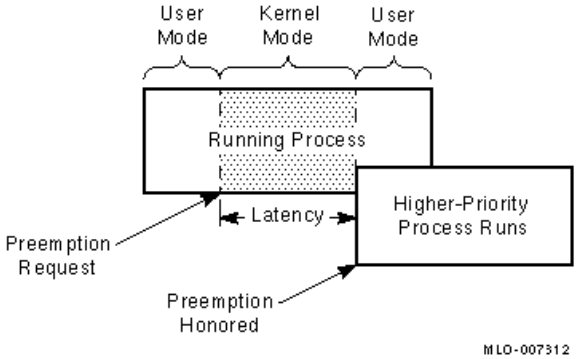
- Usado nos antigos sistemas Unix SVR3 e 4.3BSD
- Projetado para lidar com os seguintes requisitos:
  - Tempo compartilhado
  - Ambientes interativos
  - Processos *background* (batch) e *foreground* rodando simultaneamente
    - Melhorar os tempos de resposta para os usuários interativos, e
    - Garantir, 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.

Escalonamento Tradicional (cont.)

- O escalonador sempre seleciona o processo com a prioridade mais alta dentre aqueles no estado *ready*
- Processos com a mesma prioridade:
  - "***Preemptive round robin***": escalonamento circular com preempção
  - O *quantum* possui um valor fixo, tipicamente de 100 ms.
- A chegada de um processo de mais alta prioridade na fila de prontos DEVE forçar a preempção do processo em execução
- **O kernel do Unix tradicional é não-preemptivo**
  - ... Se o processo em execução estiver no estado *kernel running*, ele NÃO é preemptado
  - 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).
  - E se o *quantum* do processo acabar enquanto ele estiver em *kernel running?*

Escalonamento Tradicional (cont.)

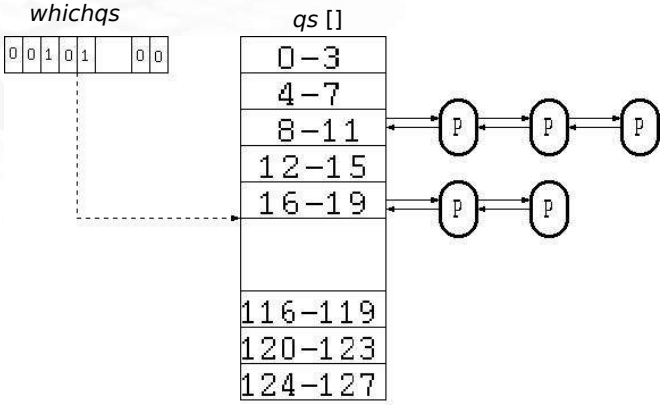
- Kernel não-preemptivo



Prioridades dos Processos – 4.3BSD

- 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

Implementação das Filas – 4.3BSD



Implementação 4.3BSD (cont.)

- Sempre que o escalonador for acionado, é executada a rotinha *swtch()*
- A rotina *swtch()* examina *whichqs* para encontrar a *run queue* de maior prioridade (primeiro bit "setado")
- *swtch()* retira o primeiro processo da fila e realiza a devida troca de contexto.
  - Para tal, é preciso acessar o contexto de hardware do processo (registradores especiais e de uso geral), que está na uArea
- O processo de maior prioridade é sempre aquele que detém a posse da CPU.... a menos que o processo corrente **esteja executando em *kernel mode***.
  - Se um processo mais prioritário se tornar *ready* e o processo em execução estiver no estado *kernel running,* uma flag é setada (*runrun*)
  - Quando o processo em execução estiver prestes a renornar para *user running*, o kernel examina a flag. Se ela está "setada", então o kernel transfere o controle para a rotina *switch()*, que inicia a troca de contexto.

Prioridades Dinâmicas

- 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_pri*** é usado pelo escalonador para decidir qual processo escalonar
- Quando o sistema está em ***user mode***

  $p\_pri = p\_usrpri$
- *p_usrpri* depende dos valores de *p_cpu* e *p_nice*

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

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

- **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 ≤ +39 **=>** 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.

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

- ***p_cpu*** (uso recente da CPU)
  - O valor inicial é 0
  - A cada interrupção de relógio (1 *tick* = 10ms) a rotina de tratamento incrementa *p_cpu* do processo em execução (até o máximo de 127)
  - A cada 100 *ticks (1 s)* os *p_cpu*s de todos os processos são reduzidos por um fator de decaimento (*decay factor*)
- **Recálculo das prioridades – 4.3BSD**
  - A cada 4 ticks, é recalculada a prioridade do processo em execução
  - A cada 100 ticks, após a aplicação do decay factor, são recalculadas as prioridades de todos os processo
    - No 4.3BSD o *decay* depende da carga do sistema
    - No SVR3 o decay é ½

Implementação 4.3BSD (cont.)

- Sempre que um processo é selecionado para rodar, ele recebe um quantum fixo de 100 ms
  - A cada 100ms (10 *ticks*) o kernel invoca a rotina *roundrobin()* para escalonar o próximo processo da mesma fila de onde saiu o processo corrente.
  - 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.
  - Se um processo mais prioritário entra em uma *run queue*, há preempção
    - Ele será escalonado antes, sem ter que esperar por *roundrobin()*.

Implementação BSD (cont.)

- 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()*

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*

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)..

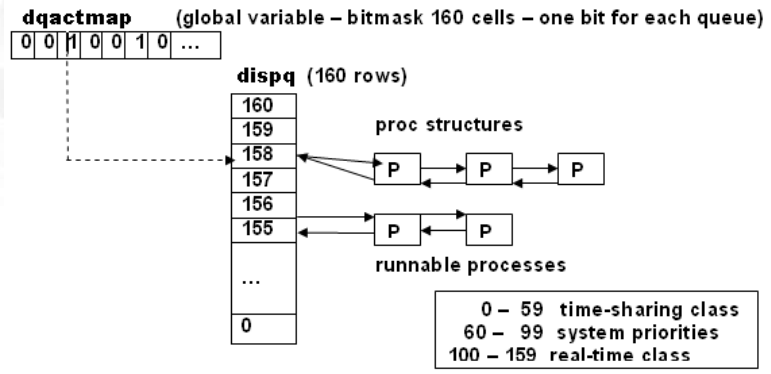
Implementação - SVR4 (1)

- 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.

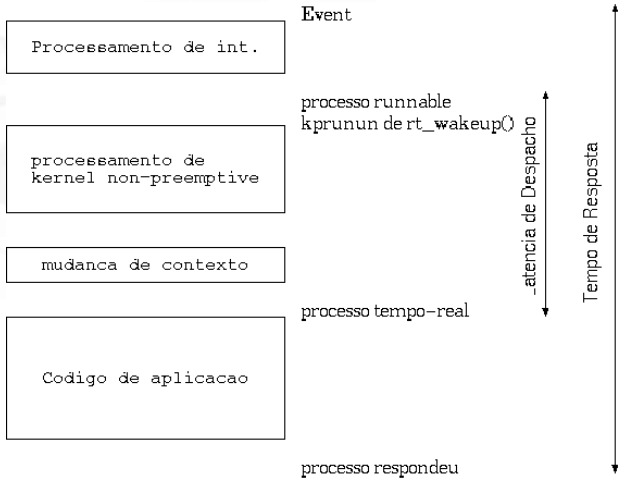
Implementação - SVR4 (3)



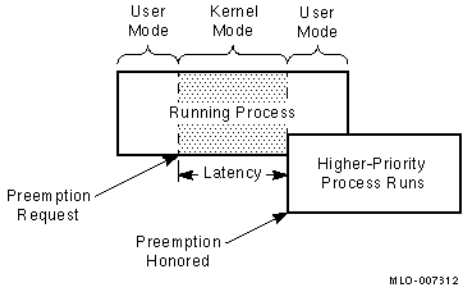
Implementação - SVR4 (4)

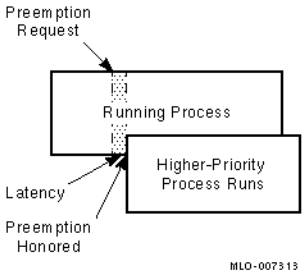
- Processos de tempo real exigem tempos de resposta limitados
- **Preemption points** são definidos em pontos do kernel onde
  - Todas as estruturas de dados do kernel encontram-se estáveis
  - O kernel está prestes a iniciar alguma computação longa
- Em cada preemption point
  - O kernel verifica a 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

Implementação - SVR4 (5)



Kernel preemptivo x Kernel não-preemptivo





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