

Conceito de Processo

- Um Sistema Operacional executa uma variedade de programas:
 - Sistemas de processamento em lotes (batch) – processa jobs
 - Sistemas de tempo compartilhado (time-shared) – roda processos de usuários ou tarefas (tasks)
- O livro texto usa os termos “job” e processo quase como sinônimos
- Processo: é um programa em execução
- Um processo inclui:
 - Contador de programa (PC)
 - Pilha
 - Segmento (área) de dados

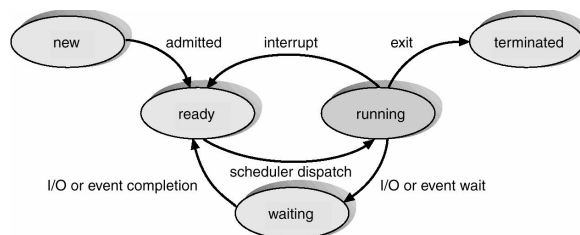
4.2

Módulo 4: Processos

- Conceito de Processo
- Escalonamento de processos
- Operações sobre processos
- Processos cooperantes
- Comunicação entre processos

4.1

Diagrama de Estados de Processos



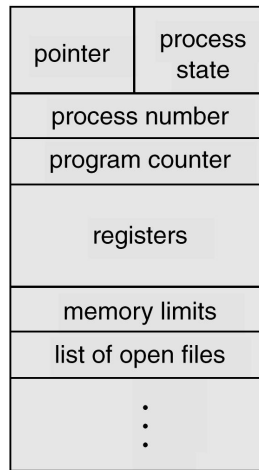
4.4

Estados de Processos

- Ao executar, processo muda de “estados”
 - New: processo está sendo criado
 - Running: instruções do processo estão executando
 - Waiting: processo está esperando ocorrência de algum evento
 - Terminated: processo terminou a execução

4.3

Process Control Block (PCB)



4.6

Process Control Block (PCB)

- PCB contém informações associadas a cada processo:
 - Estado do processo
 - Valor do PC (apontador de instruções)
 - Área para guardar valor dos registradores
 - Infos. para escalonamento de CPU (escalonamento processos)
 - Infos. Para gerenciamento de memória
 - Infos. De contabilidade dos processos
 - Status das operações de I/O (ex.: Infos. sobre arquivos usados)

4.5

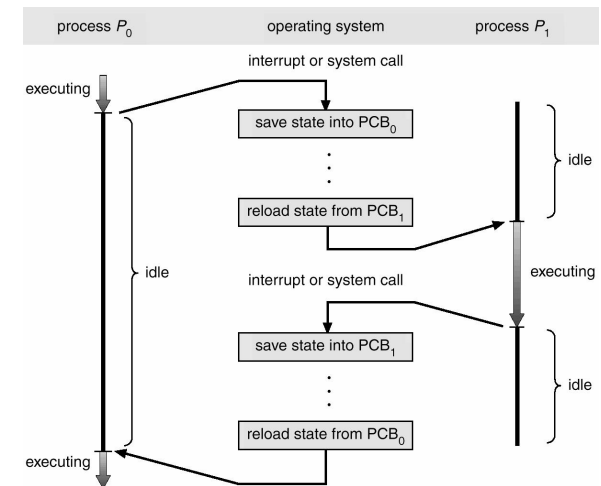
Filas para escalonamento de processos

- Fila de "jobs": conjunto de todos os processos no sistema
- Fila de prontos (ready queue): conjunto de todos os processos que residem na memória principal, prontos para executar mas não estão em execução
- Filas de dispositivos: conjunto de processos esperando por um dispositivo de I/O

“Os processos migram entre as diversas filas do sistema”

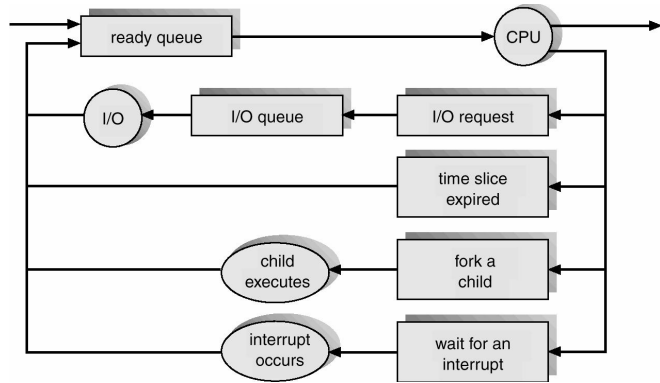
4.8

I roca de processos (ou context switch)



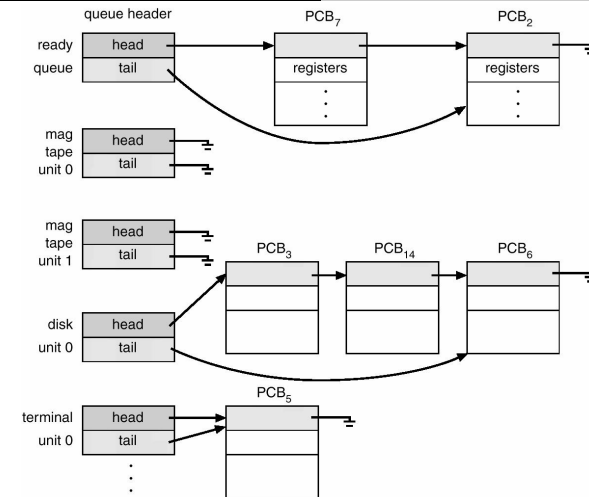
4.7

Representação do escalonamento de processos Sistemas Operacionais



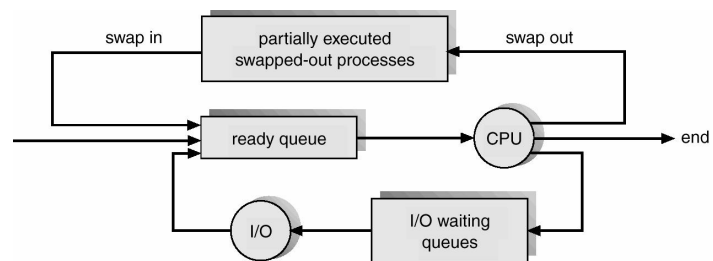
4.10

Fila de prontos e as diversas filas de dispositivos de I/O Sistemas Operacionais



4.9

Adição de um escalonador de médio prazo Sistemas Operacionais



4.12

Escalonadores Sistemas Operacionais

- Escalonador de longo prazo (Long-term scheduler) – seleciona qual processo deve ser colocado (buscado) para a fila de prontos
 - por exemplo:
 - * de uma fila de jobs batch
- Escalonador de curto prazo (short-term scheduler, CPU scheduler) – seleciona, da fila de prontos, que processo deve ser executado a seguir (i.e. alocado à CPU).

4.11

Trocas (chaveamentos) de Contexto

- Quando CPU troca processo em execução por outro, SO deve salvar o estado (contexto de execução) do processo anterior e carregar nos regs. da máquina o estado (guardado) do próximo processo
- Tempo de troca de contexto é “overhead”; a traca é útil mas não é o objetivo do sistema, objetivo é processar / computar jobs
- Tempo de troca de contexto é dependente do suporte de hardware

4.14

Escalonadores (Cont.)

- Escalonador de curto prazo é invocado muito frequentemente (milisegundos) ⇒ (deve ser rápido)
- Escalonador de longo prazo é invocado infrequentemente (segundos, minutos) ⇒ (pode ser lento)
- Escalonador de longo prazo controla o *grau de multiprogramação* do sistema
- Processos podem ser classificados em:
 - *I/O-bound processes* – delimitados pelo tempo de I/O, gasta mais tempo fazendo I/O do que computações, muitas pequenas rajadas (bursts) de CPU
 - *CPU-bound processes* – delimitados pelo tempo de CPU, gastam a maior parte do tempo fazendo computações; Poucas RAJADAS LONGAS de CPU

4.13

Criação de Processos (Cont.)

- Possibilidades para o espaço de endereçamento
 - Filho é duplicata do pai (espaço separado)
 - Filho tem programa carregado no espaço do pai (substituição)
- Exemplos do UNIX
 - Chamada a **fork** cria novo processo
 - Chamada **execve** usada **depois do fork** (no código do clone) para substituir programa no espaço do “clone”
 -

4.16

Criação de processos

- Processo pai cria “filhos” (ou clones), que por sua vez criam outros formando uma árvore de processos
- Compartilhamento de recursos possíveis:
 - Pai e filhos compartilham todos os recursos
 - Filhos compartilham subconjunto de recursos do pai
 - Pai e filhos não compartilham recursos
- Execuções possíveis:
 - Pai e filhos executam concorrentemente
 - Pai espera até filhos terminarem

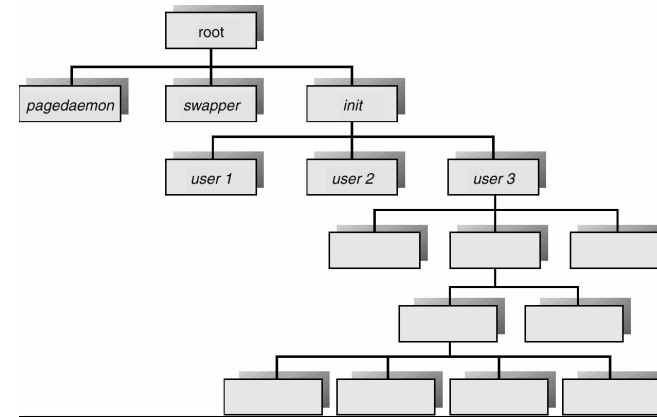
4.15

Término de Processos

- Processo executa chamada `exit` pedindo ao SO que termine o processo
 - Dado passado de filho para pai (via **wait**)
 - Recursos do processo são de-alocados pelo SO
- Pai pode terminar a execução de processos filhos (**abort**) devido:
 - Filho excedeu uso de recursos alocados
 - Tarefa atribuída ao filho não é mais necessária
 - Pai vai terminar... possibilidades são:
 - * SO termina filho também
 - * “Cascateamento” de terminos
 -

4.18

Uma árvore de processos em um sistema UNIX típico



4.17

Problema Produtor-Consumidor

- Paradigma (padrão de comportamento) para certos processos cooperantes
 - Processo produtor produz informação que é “consumida” por outro processo (consumidor)
- Comunicação entre os processos cooperantes utiliza um “buffer”:
 - *Bounded-buffer*: tem tamanho fixo
 - *Unbounded-buffer*: assume-se que o buffer não tem limite de tamanho.

4.20

Processos Cooperantes

- Processos *Independentes* não podem afetar ou serem afetados pela execução de outros processos
- Processos *Cooperantes* podem afetar a execução de outros processos cooperantes (ou ter execução afetada)
- Vantagens da cooperação entre processos:
 - Compartilhamento de informações
 - Aumento da velocidade de computação (speed-up)
 - Modularidade
 - Conveniência

4.19

Bounded-Buffer (Cont.)

- Consumer process

repeat

while $in = out$ **do** *no-op*;

$nextc := buffer[out]$;

$out := out + 1 \bmod n$;

...

consume the item in *nextc*

...

until *false*;

- Solution is correct, but can only fill up $n-1$ buffer.

4.22

Solução para probl. Bounded-Buffer com memória compartilhada

- Shared data

var n ;

type *item* = ... ;

var *buffer*. **array** $[0..n-1]$ **of** *item*;

$in, out: 0..n-1$;

- Producer process

repeat

...

produce an item in *nextp*

...

while $in + 1 \bmod n = out$ **do** *no-op*;

$buffer[in] := nextp$;

$in := in + 1 \bmod n$;

until *false*;

4.21

Threads (Cont.)

- Em uma tarefa com múltiplas threads, enquanto uma thread está bloqueada e esperando, uma segunda thread na mesma tarefa pode executar
 - Cooperação de múltiplas threads no mesmo job confere maior vazão (throughput) e melhoria de desempenho
 - Aplicações que requerem compartilhamento de dados se beneficiam ao utilizar threads (ex. buffer comum em padrão produtor-consumidor)
- Threads provêm um mecanismo que possibilita a um processo sequencial fazer uma chamada bloqueante ao SO e ao mesmo tempo obter paralelismo no processo
- Threads podem ser suportadas pelo núcleo do SO (ex. Mach, OS/2, Linux).
- Threads podem ser suportadas em espaço de endereços de usuário (User-level threads);
 - Como uma biblioteca que roda em espaço de usuário (ex. Projeto Andrew da CMU, C-Threads)
 - Suportado pela linguagem e respectivo compilador (ex. Modula, Java)
- Abordagem híbrida implementa ambos os métodos (ex. user-level e kernel-level threads em Solaris 2).

4.24

Threads

- Uma *thread* (ou *lightweight process*) é uma unidade básica de utilização de CPU
- Uma thread consiste de:
 - Apontador de instruções (PC)
 - Conjunto de registradores
 - Espaço de pilha
- Thread = processo de peso leve ou, também, linha de execução de um processo
- Uma thread compartilha com outras threads pares (peers):
 - A seção de código
 - A seção de dados
 - Os recursos do SO
 coletivamente tudo isso é conhecido como tarefa (task)
- Um processo tradicional (ou *heavyweight process*) é igual a uma tarefa com uma thread
- OBS: para executar com threads é necessário uma pilha por thread (uma só não é suficiente), ao contrário das áreas de código, vars. e heap que podem ser compartilhadas pelas threads pares.

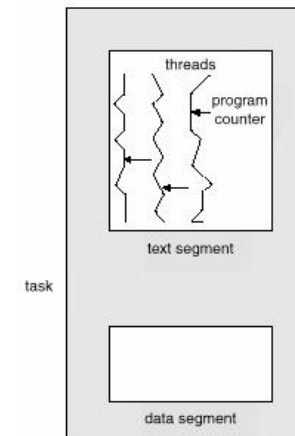
4.23

Suporte a Threads em Solaris 2

- Solaris 2 é uma versão do UNIX com suporte para threads pelo kernel ou em nível de usuário, multiprocessamento simétrico (SMP) e escalonamento de tempo real
- LWP (lightweight process) – intermediário entre user-level threads e kernel-level threads
- Utilização de recursos por tipo de thread:
 - Kernel thread: uma pequena estrutura de dados e uma pilha; troca de contexto entre threads pares não requer troca de infos. de acesso à memória (segmentos compartilhados) – relativamente rápido
 - LWP: PCB com valores de regs, infos. de contabilidade e memória; troca entre LWPs é relativamente lenta
 - User-level thread: precisa apenas uma pilha e uma apontador de instruções (PC) por thread. Chaveamento rápido entre threads pares (kernel vê somente os LWPs, estes é que suportam as threads user-level)

4.26

Múltiplas Threads dentro de uma tarefa



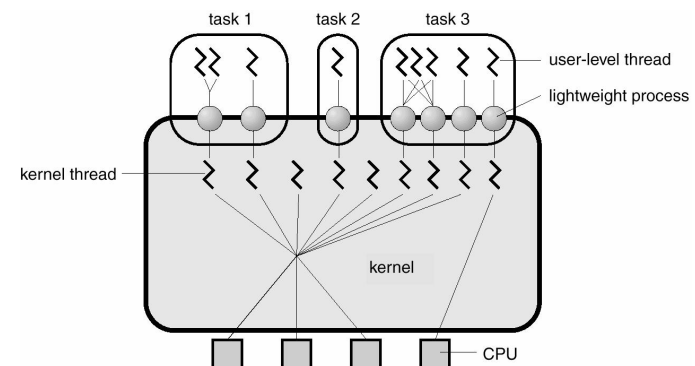
4.25

Comunicação entre Processos (IPC)

- São necessários mecanismos comunicação entre processos e sincronização de suas ações
- Sistema de mensagens: processos comunicam uns com os outros sem uso de variáveis compartilhadas
- Mecanismo de IPC por troca de mensagens provê duas operações:
 - **send(message)** – mensagens de tamanho fixo ou variável
 - **receive(message)**
- Se P e Q querem comunicar, eles precisam:
 - Estabelecer um “canal” de comunicação entre si
 - Trocar mensagens via send/receive
- Implementação de um link de comunicação:
 - Físico (e.g., memória compartilhada, barramento de hardware)
 - lógico (e.g., propriedades lógicas)

4.28

Solaris 2 Threads



4.27

Questões de Implementação

- Como os links são estabelecidos ?
- Um link pode ser associado a mais de dois processos ?
- Quantos links podem existir entre cada par de processos comunicantes ?
- Qual é a capacidade de um link ?
- O tamanho da mensagem é fixo ou variável ?
- O link é unidirecional ou bidirecional?