

## Unix: Processos e o Kernel

### O Kernel

- É um programa especial, uma parte privilegiada do sistema operacional, que roda diretamente sobre o hardware.
- Ele implementa o modelo de processos do sistema.
  - O kernel não é um processo!
- O kernel oferece uma série de serviços aos processos (de usuários ou de sistema)
- No Unix, o código do kernel reside em um arquivo em disco normalmente nomeado como */vmunix* ou */unix* (no Ubuntu, */boot/vmlinuz...*)
- Programa *bootstrapping* carrega o kernel para a memória na inicialização do sistema.

### A Abstração “Processo”

- Processo é uma entidade que executa um programa e que provê um ambiente de execução para ele.
- Processos têm um ciclo de vida:
  - São criados pelas SVCs *fork* ou *vfork* e rodam até que sejam terminados (por meio da primitiva *exit*).
- Durante a sua execução um processo pode rodar um ou mais programas.
  - A primitiva **exec** é invocada para rodar um novo programa.
- Processos no Unix possuem uma hierarquia
  - Cada processo tem um processo pai (*parent process*) e pode ter um ou mais processos filhos (*child processes*).

### A Abstração “Processo” (cont.)

- O processo *init* (PID 1) localiza-se no topo da hierarquia de processos de usuário do Unix.
  - Primeiro processo de usuário a ser criado, quando o sistema é iniciado.
  - Executa o programa */etc/init*.
- Todos os processos descendem do processo *init*, à exceção de alguns poucos, como o *swapper* e o *pagedaemon*.
  - Esses também são criados na inicialização do sistema e ajudam o kernel na execução das suas tarefas
  - Diferentemente do *init*, esses processos são implementados dentro do próprio kernel, ou seja, não há um programa binário regular para eles.
- Se, ao terminar, um processo tiver processos filhos ativos, estes tornam-se órfãos e são herdados pelo processo *init*.

## Imagem do Processo

- *Espaço de endereçamento do usuário*
  - Texto (código), dados, *user stack*, regiões de memória compartilhada, etc.
- *Informações de controle*
  - Armazenadas em estruturas mantidas pelo kernel
  - *u area* e *proc structure*

## Processo e Memória Virtual

- A maioria das implementações do UNIX usa memória virtual
  - Endereços utilizados no programa não referenciam diretamente memória física
  - Cada processo possui seu “espaço de endereçamento virtual”
    - Endereços virtuais são traduzidos para uma localização física na memória principal (e.g. tabela de páginas)
  - Processos só podem acessar endereços dentro deste espaço
- O Espaço de endereçamento é dividido em:
  - **Process Space**
  - **System Space** ou *kernel space*
    - Parte fixa do espaço de endereçamento virtual mapeia o “kernel text” e as estruturas de dados do kernel

## System (Kernel) Space

- Além de código e dados do usuário, parte do espaço de endereçamento virtual de cada processo refere-se a código e dados do kernel (que, afinal, roda em benefício dos processos).
- Esta porção é conhecida como *system space* ou *kernel space*, e só pode ser acessada em *kernel mode*.
- Como existe apenas uma instância do kernel todos os processos são mapeados em um único *kernel address space*.
- O kernel mantém algumas estruturas de dados globais e algumas específicas para cada processo (“per-process objects”).
- Essas últimas contêm informações que permitem ao kernel acessar o espaço de endereçamento de qualquer processo.

## Estruturas de Controle

- Proc Structure
  - Contém todas as informações que o kernel possa precisar quando um processo NÃO está *running*
  - **Encontra-se no *system space***
- u Area
  - Contém informações necessárias apenas quando o processo está *running*
  - **Encontra-se no *process space***

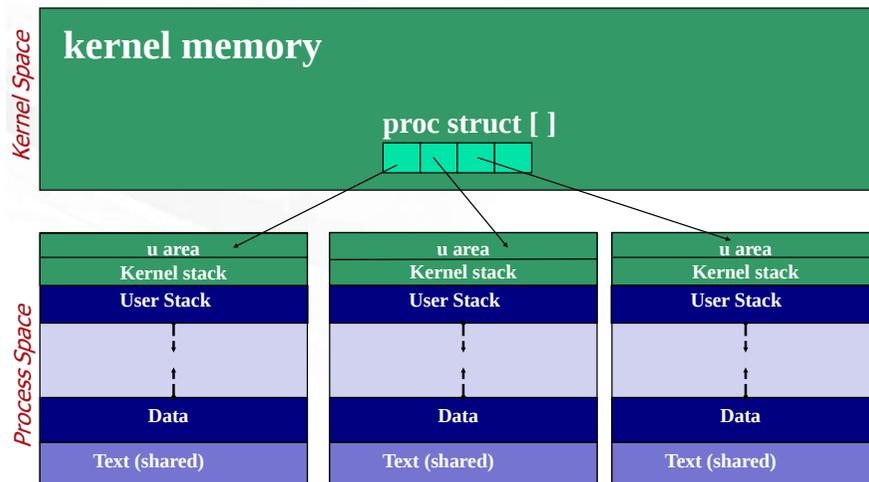
## Proc Structure

- Contém todas as informações que o kernel possa precisar quando um processo NÃO está *running*
- Campos da proc structure
  - Identificadores: PID, *process group*, *process session*, etc.
  - Informações de hierarquia (*p\_pid*)
  - Estado do processo corrente
  - Ponteiros para linkar o processo em filas
  - Prioridade de escalonamento
  - Informações para manipulação de sinais (ignorados, bloqueados, postados, etc.)
  - Informações para gerenciamento da memória
  - Localização do mapa de endereços para a *u area* do processo

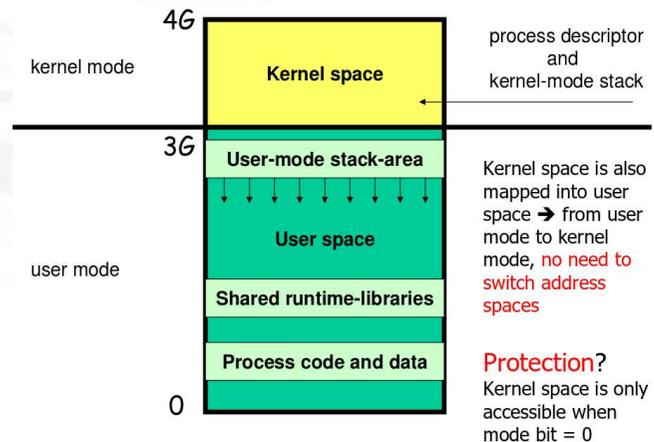
## U area

- Contém informações necessárias apenas quando o processo está *running*
- É normalmente mapeada sempre num local fixo do espaço de endereçamento virtual do processo.
- Campos da u area:
  - Hardware context* (no UNIX é chamado de “Bloco de controle de processo” !!)
  - Real e effective UID e GID* (credenciais do usuário)
  - Argumentos, valores de retorno e status de erros da SVC corrente
  - Handlers* de sinais e informações relacionadas
  - Informações do header do processo, como tamanho das áreas de texto, dados e pilha
  - Tabela de descritores de arquivos abertos
  - Estatísticas de uso da CPU
  - Ponteiro para a *proc structure*

## Kernel Space x Process Space



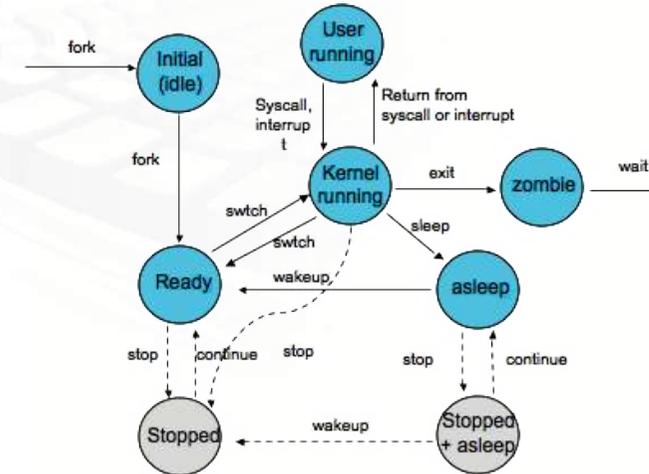
## E o Linux



## User Mode x Kernel Mode

- Com finalidade principal de proteção, a maioria dos computadores atuais fornece pelo menos dois modos de operação.
- O Unix requer do hardware a implementação de apenas 2 modos de operação:
  - user mode* (menos privilegiado)
  - kernel mode* (com mais privilégios).
- Processos de usuário rodam em *user mode*
  - Processos não podem - acidental ou maliciosamente - corromper outro processo ou mesmo o kernel
- Acesso ao *kernel space* só pode ser feito em *kernel mode*
  - O processo só consegue acessar o *kernel space* via SVC (portanto, acesso controlado).

## User Mode/Kernel Mode x Máquina de Estados



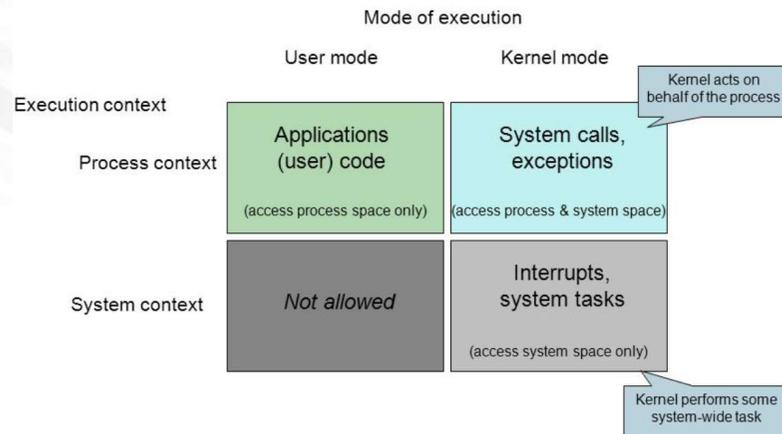
## Process Context

- As funções do kernel podem ser executadas em dois contextos:
  - Process context*
  - System context*
- Em *process context* o kernel age em benefício do processo corrente (ex: ao executar uma SVC).
- Neste contexto, o kernel pode acessar e modificar o espaço de endereçamento, a *u area* e a *kernel stack* do processo.
- O kernel pode ainda bloquear o processo corrente se este deve esperar por um recurso ou atividade de algum dispositivo.

## System Context

- Tarefas como atender a interrupções e re-computar prioridades não são executadas em benefício de um processo em particular.
  - Atividades genéricas como estas (*system-wide tasks*) são ditas executadas em *system context*.
- Quando executando em *system context*, o kernel **não pode acessar o espaço de endereçamento** (incluindo *u area* ou a *kernel stack*) do processo corrente.
- Além disso, o kernel **não pode bloquear o processo** corrente se estiver em *system context* pois poderia estar bloqueando um processo que não tem nada a ver com a tarefa sendo executada.
- Em algumas situações pode até nem ter mesmo um processo para bloquear (todos podem estar esperando por I/O).

## Resumindo ...

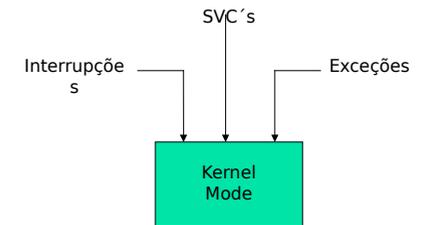


## Executando em Kernel Mode

- Existem 3 tipos de eventos que fazem o sistema entrar em *kernel mode*:
  - As chamadas ao sistema (SVCs/traps/interrupções de software)
  - As interrupções provenientes dos dispositivos periféricos
  - As exceções
- Na ocorrência de uma delas, uma sequência especial de instruções é executada para por o sistema em *kernel mode* e passar o controle para o kernel.

HW alterna para kernel mode utilizando a kernel stack do processo:

- HW salva PC e Status (e possivelmente outros "estados" na kernel stack)
- SW (rotina assembly) salva outras informações necessárias para seguir o tratamento do evento



## Executando em Kernel Mode (cont.)

- Interrupções
  - Eventos assíncronos causados por dispositivos periféricos (disco, relógio, interface de rede, etc).
  - São tratadas em *system context* pois não são causadas ou provenientes do processo corrente
    - Não pode haver acesso ao *process space*
    - Não pode bloquear (esperar por eventos) pois isso bloquearia um processo arbitrário e inocente

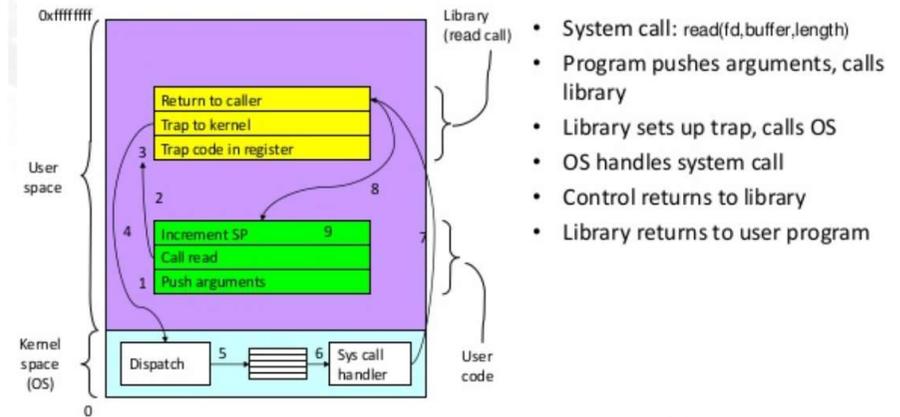
## Executando em Kernel Mode (cont.)

- Exceções
  - Eventos síncronos ao processo em execução, causados por eventos relacionados ao próprio processo
    - Divisão por zero, acesso a endereço ilegal, *overflow*, etc.)
  - A rotina de tratamento da exceção (*exception handler*) roda em *process context*
    - Pode acessar o espaço de endereçamento do processo (incluindo a *U area*) e bloquear, se necessário.

## Executando em Kernel Mode (cont.)

- SVC's
  - Ocorrem quando o processo executa uma instrução especial do processador (*trap*, *syscall*)
  - São eventos síncronos ao processo em execução
  - Assim como as exceções, a SVC roda em **process context**:
    - Pode acessar o espaço de endereçamento do processo (incluindo *U area*), além de poder bloquear o processo, se necessário.

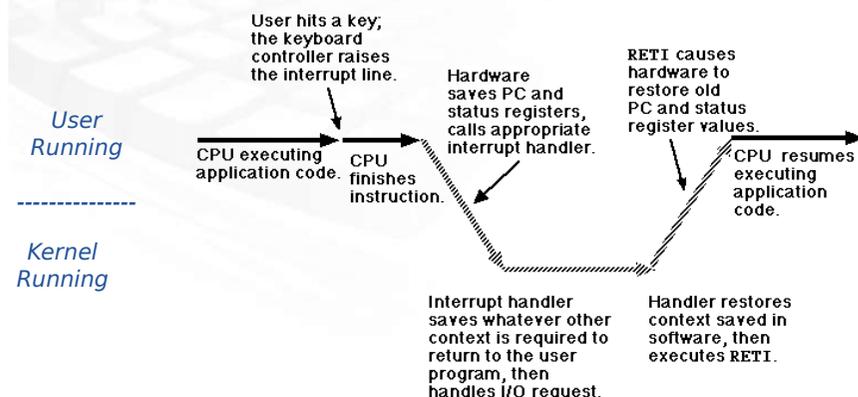
## Execução de uma SVC



## Manipulação de Interrupções

- Interrupções são eventos gerados assincronamente à atividade regular do sistema.
  - O sistema não sabe em que ponto no fluxo de instruções a interrupção ocorrerá.
- *Interrupt Handler* ou *Interrupt Service Routine* é o nome dado à rotina de tratamento da interrupção.
- O Interrupt Handler roda em *kernel mode* e *system context*.
  - Logo, o **handler não pode** acessar o *process space*, já que ele não tem nada a ver com a interrupção. Idem bloqueá-lo.
  - O tempo de servir a interrupção é descontado do quantum do processo em execução (*time-slice*).

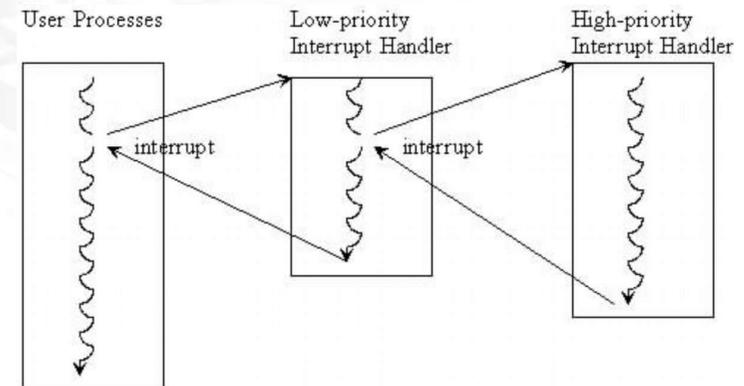
## Manipulação de Interrupções (cont.)



## Manipulação de Interrupções - IPL

- Interrupções podem ocorrer enquanto uma outra está sendo atendida
  - Necessidade de se priorizar as interrupções
  - Ex: interrupção de relógio é mais prioritária do que a interrupção de interface de rede (cujo processamento dura vários *ticks* do relógio)
- A cada tipo de interrupção está associado a um **Interrupt Priority Level - IPL**
  - O número de níveis de interrupção varia de acordo com o padrão Unix e com as arquiteturas de hardware.
  - Ex: Unix BSD - IPL's variam de 0 a 31.

## Manipulação de Interrupções - IPL



## Manipulação de Interrupções - IPL

(cont.)

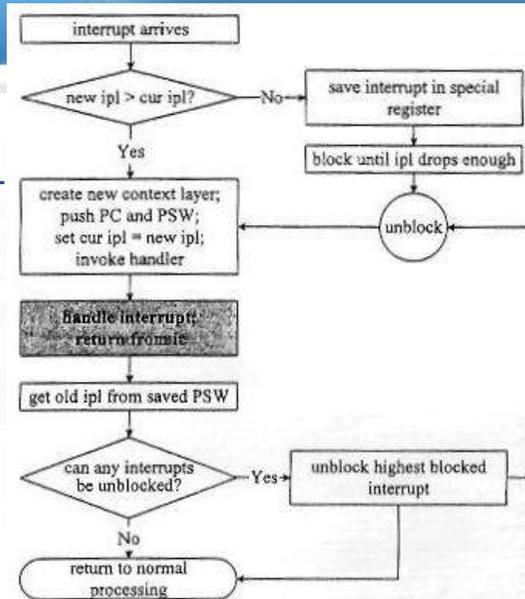


Figure 2-5. Interrupt handling.

## Sincronização

- O kernel do Unix é reentrante, o que significa que num dado momento vários processos podem estar ativos no kernel.
- Como todos os processos compartilham uma única cópia das estruturas de dados do kernel, é necessário impor alguma forma de sincronização para prevenir que o kernel não seja corrompido.

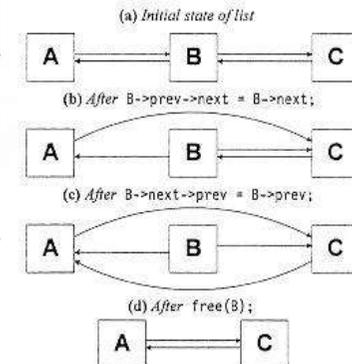


Figure 2-6. Removing an element from a linked list.

## Sincronização (cont.)

- Como o kernel tradicional do Unix é **não-preemptivo**
  - Um processo rodando no estado *kernel running* (i.e. CPU em *kernel mode*) não pode ser preemptado (substituído por um outro processo), mesmo que o seu quantum expire.
- Quando o completar a atividade em kernel mode (ex: acabou o tratamento da interrupção, ou a execução da SVC finalizou), então o processo em execução poderá sofrer preempção
  - Se seu quantum expirou OU um processo mais prioritário tornou-se pronto
- Nesse caso, existe a garantia de que o kernel está num estado consistente.
- Em resumo, um kernel não-preemptivo é uma solução ampla para a maioria dos problemas de sincronização. Porém, três situações ainda persistem:
  - Operações bloqueantes, interrupções e multiprocessamento.**

## Sincronização x Operações Bloqueantes

- Uma operação bloqueante é quando, ao fazer uma SVC o processo é bloqueado “no meio”... até que a operação se complete (ex: I/O)
- Quando o evento ocorrer, o processo será desbloqueado e, ao renornar ao estado kernel running, completará a execução da SVC que causou o bloqueio do mesmo
- Alguns objetos (dentro do kernel) podem precisar ser protegidos entre operações bloqueantes, requerendo mecanismos adicionais... Ex:
  - Operação de leitura de um bloco de disco para um buffer do *buffer cache* (outros processos não podem acessar o buffer enquanto o I/O não se completa).
- Solução: uso de *locks* e *wanted flag*

## Sincronização x Interrupções

- Por ser não-preemptivo, o kernel é protegido de preempção por um outro processo. Entretanto, um processo manipulando estruturas de dados do kernel pode ser interrompido por um dispositivo.
- Nessa situação, se o *Interrupt Handler* tentar acessar estruturas do kernel elas podem ficar em estado inconsistente (intencionalmente ou não).
- Solução: bloquear interrupções ao acessar estruturas críticas do kernel.
  - OBS: bloquear uma interrupção significa bloquear todas as interrupções de mesmo nível ou menor.

```
int x = splbio();           /*inibe interrupções de disco */
modify disk buffer cache; /* região crítica */
splx(x);                   /* restaura valor prévio do IPL */
```

## Sincronização x Multiprocessadores

- No caso de mais de um processador os problemas de sincronização são muito mais complexos já que a premissa de não-preempção do kernel deixa de existir.
- Dois processos podem executar em *kernel mode* em diferentes processadores e executarem uma mesma função do kernel concorrentemente.
- Assim, a qualquer hora que o kernel acessar uma estrutura de dados global esta deve ser protegida de acesso por outros processadores (o próprio mecanismo de *locking* deve ser protegido de múltiplos acessos).

## Referências

- VAHALIA, U. Unix Internals: the new frontiers. Prentice-Hall, 1996.
  - Capítulo 2 (até seção 2.5)