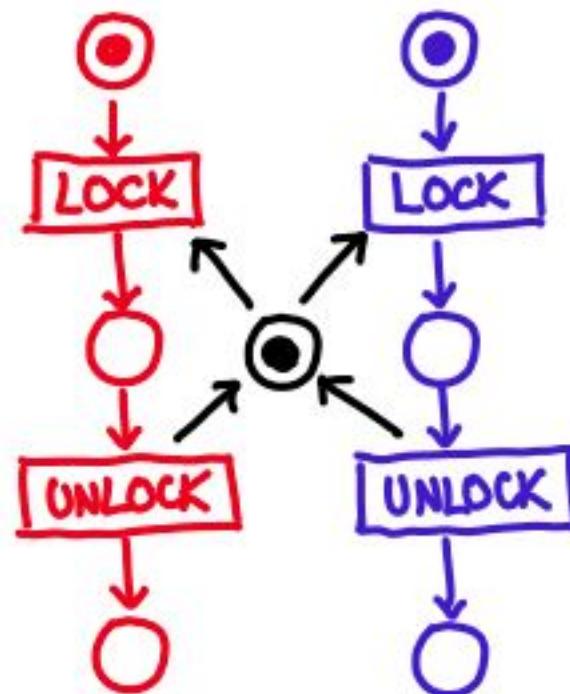


Capítulo 2

Gerenciamento de Processos, Threads e Escalonamento



Roteiro:

- Processos
- Escalonamento
- Threads
- Comunicação e sincronização inter-processo
- Problemas clássicos de sincronização e comunicação

O que é um Processo?

- Um programa em execução
- Uma atividade concorrente (do usuário ou do sistema) em andamento ou suspensa
- Utiliza recursos (dispositivos de E/S) causando mudanças em seu estado
- Seu contexto é gerenciado pelo núcleo
- sua execução controlada pelo escalonador

O que caracteriza um processo?

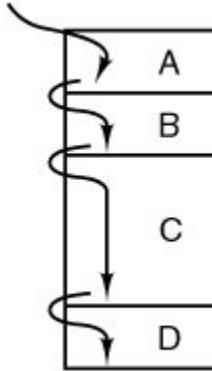
Cada processo consiste de:

- a) Espaço de endereçamento na memória principal
 - com segmentos de código (text), de dados, heap e pilha, regiões de memória compartilhada
- b) Informações de Controle
 - mantidas em tabelas no núcleo, por exemplo: PiD, o estado do processo, lista de arquivos abertos, nível de prioridade, etc.
- c) Credenciais
 - Identificação do usuário e grupo do processo
- d) Variáveis do ambiente (environment variables)
- e) Contexto da CPU
conteúdo dos registradores de propósito geral, SP, PC, IR, PSW

Processos

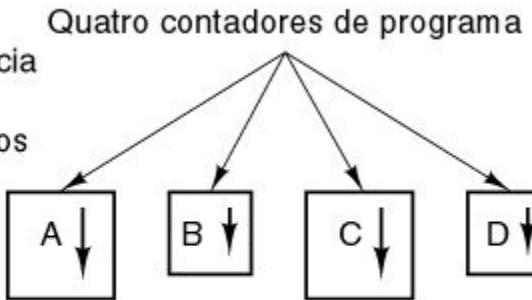
O Modelo de Processo

Um contador de programa



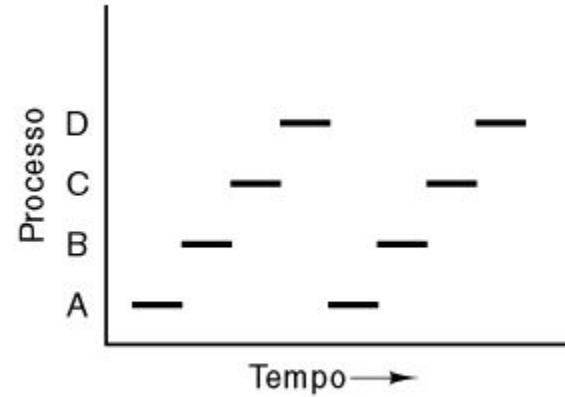
(a)

Alternância entre processos



(b)

Quatro contadores de programa



(c)

- Considere a multiprogramação de 4 programas:
 - O contador de programa (PC) alternadamente assume endereços de cada programa
 - Conceitualmente são 4 processos sequenciais independentes
 - Somente um programa está ativo a cada momento

Criação de Processos

Principais eventos que levam à criação de processos:

1. Ao iniciar o sistema operacional (o init)
2. Usuário executa comando/ inicia programa através da shell
3. Atendimento de uma requisição específica (p.ex. processo **inet** cria processo para tratar requisição de rede: ftp, rsh, etc.)
4. Ativação de um processo em momento pré-determinado (através do cron)
5. Processamento de um job de uma fila de jobs

Em todos os casos, um processo pai cria um novo processo usando `fork()`

Término de Processos

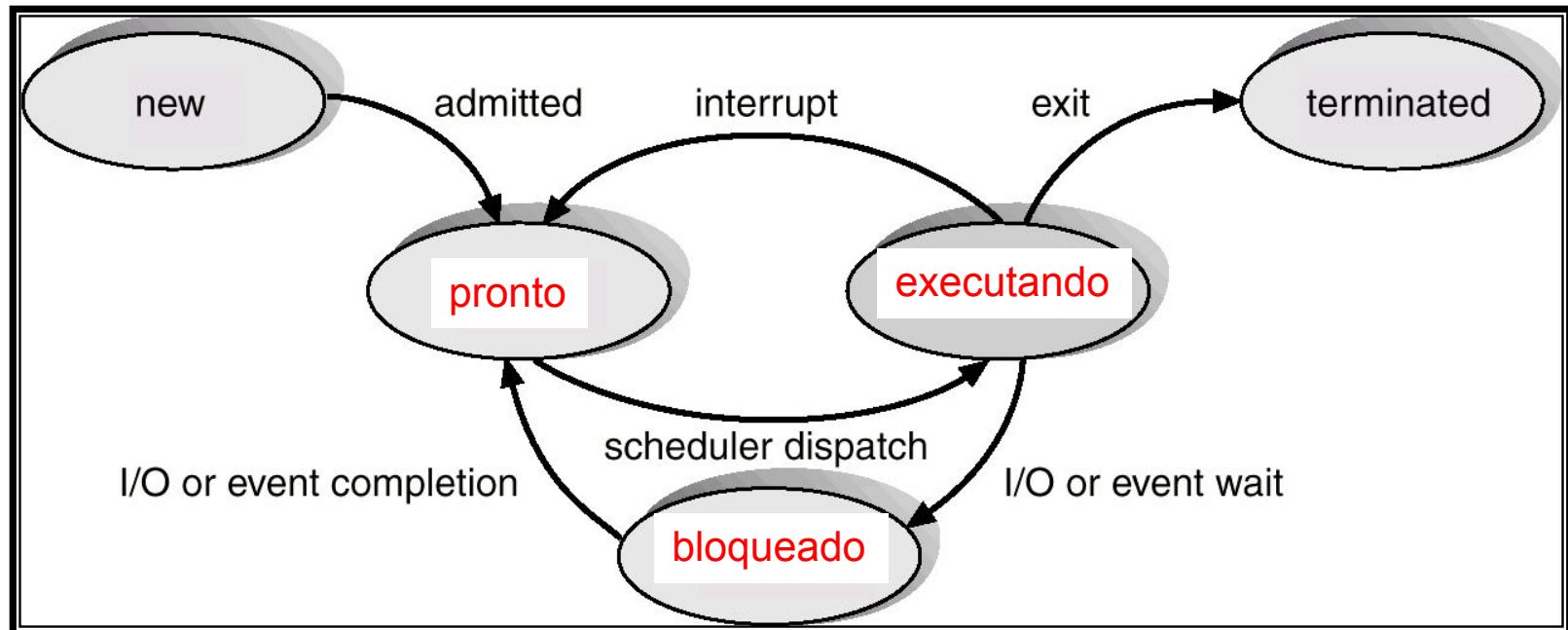
Condições que levam ao término de processos:

1. Saída normal (voluntária) através de **exit**
2. Saída por erro (voluntária)
3. Erro fatal (involuntário)
4. Cancelamento por um outro processo (involuntário), através de um sinal.

Hierarquias de Processos

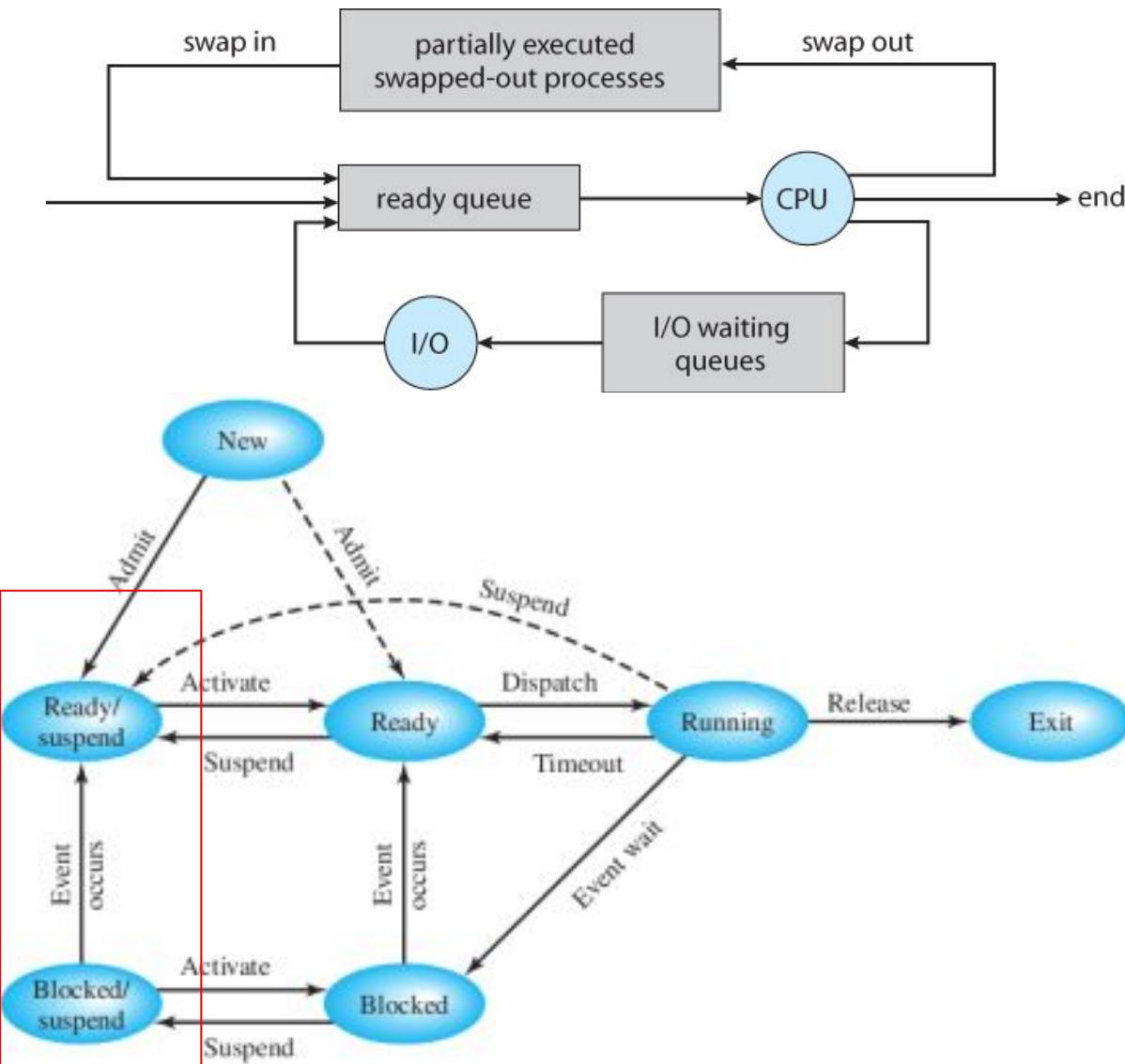
- Processo pai cria um processo filho, processo filho pode criar seu próprio processo, etc.
- Forma-se uma hierarquia de processos
 - UNIX chama isso de “grupo de processos”
- Windows não possui o conceito de hierarquia de processos
 - Todos os processos são criados no mesmo nível

Máquina de Estados dos Processos

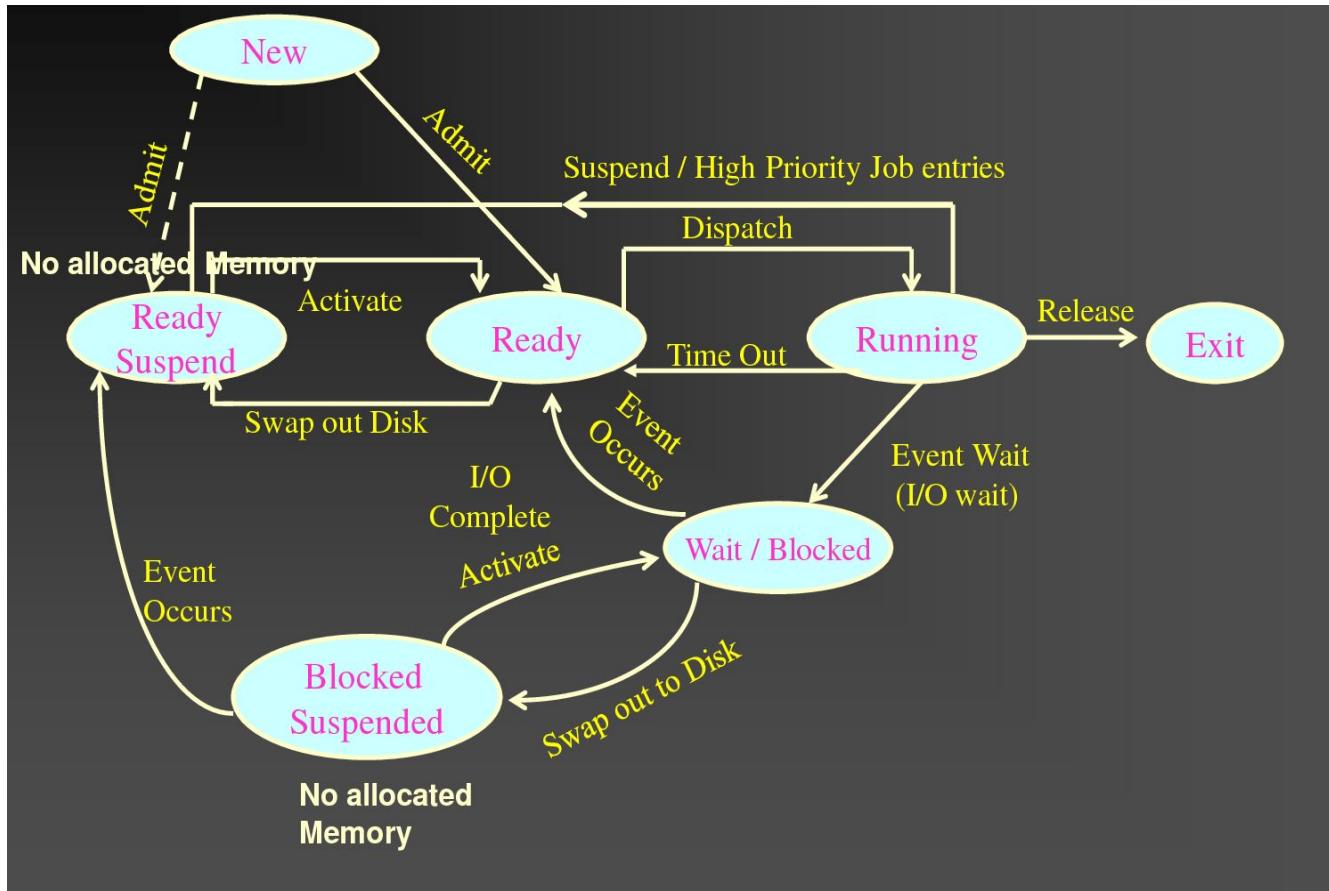


- Ao longo de sua execução, um processo pode assumir os seguintes estados:
 - **new**: processo foi criado (núcleo aloca entrada na tabela de proc.)
 - **Running/executando**: instruções sendo executadas.
 - **Waiting/bloqueado**: Processo aguarda por algum evento ou recurso para prosseguir.
 - **Ready/pronto**: Processo aguarda por alocação do processador.
 - **terminated**: Processo deu exit(), núcleo desaloca entrada

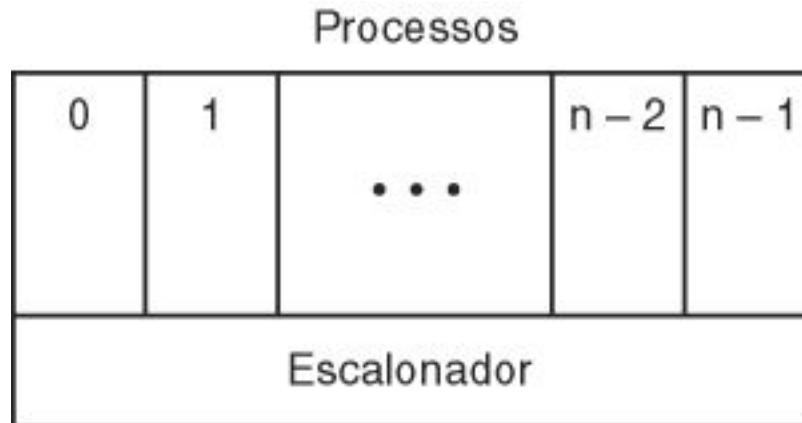
Máquina de Estados de Processos (com estados suspensos)



Estados de Processos (um pouco mais próximo da realidade)



Escalonador



- Camada mais básica do núcleo do SO gerencia execução dos processos
 - trata interrupções, troca de contexto, escalonamento
- Acima daquela camada estão os processos concorrentes

Processos daemon

Definição: Um **daemon** é um processo que executa em background e não têm um terminal associado

- Muitos deles são criados no momento do boot (existem dezenas de kernel daemons executando periodicamente)
- São responsáveis por iniciar ou executar serviços do sistema
- Não há processo pai que tenha controle sobre eles
- Exemplos:
 - **inetd** (internet service daemon) fornece serviços de Internet. Para cada serviço configurado, ele escuta por pedidos de conexão de clientes. Os pedidos são atendidos através da criação de um processo que executa o protocolo apropriado,
 - **cron** um serviço que executa comandos agendados em crontab.
 - **cupsd** é um spooler de impressão que trata dos pedidos de impressão no sistema.
 - **sshd** fornece login remoto seguro e facilidades de execução.
-

Zombies e orfãos

Um processo zombie é um processo que completou, mas que ainda apresenta uma entrada numa tabela de processos.

Um processo filho é dito **órfão** quando este permanece em execução mesmo depois de o seu processo pai ter sido terminado (ou concluído sem ter executado `wait()`)

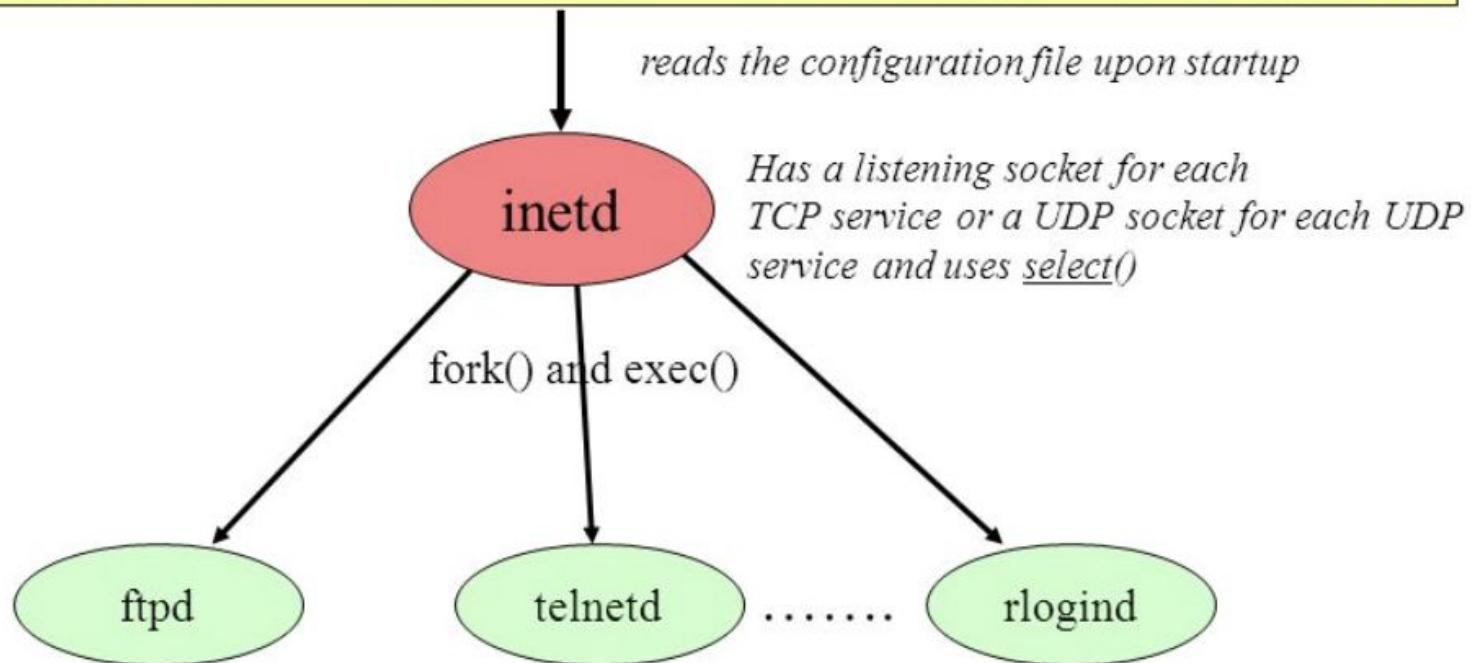
órfãos são identificados, e `init` passa a ser o seu novo processo pai. Termina-se o orfão com o sinal **SIGHUP**

Alguns daemons ligados à comunicação por rede

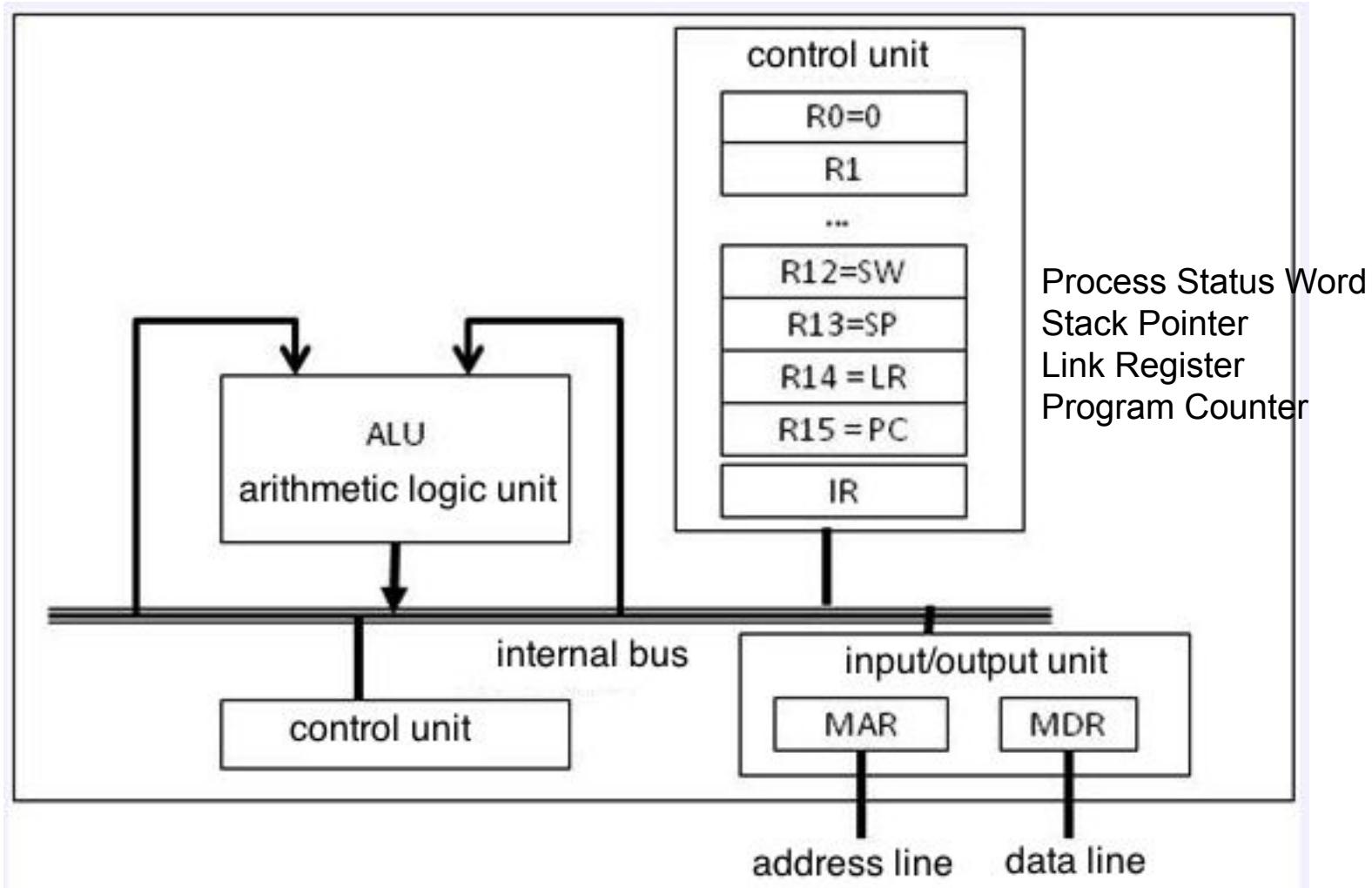
/etc/inetd.conf file

ftp	stream	tcp	nowait	root	/usr/sbin/in.ftpd	in.ftpd
telnet	stream	tcp	nowait	root	/usr/sbin/in.telnetd	in.telnetd
name	dgram	udp	wait	root	/usr/sbin/in.tnamed	in.tnamed
shell	stream	tcp	nowait	root	/usr/sbin/in.rshd	in.rshd
login	stream	tcp	nowait	root	/usr/sbin/in.rlogind	in.rlogind
tftp	dgram	udp	wait	root	/usr/sbin/in.tftpd	in.tftpd -s /tftpboot

reads the configuration file upon startup



CPU e Registradores



Obs:

Process Status Word = 16 bits que refletem o estado atual da CPU

Link Register = contém o endereço de retorno qdo a chamada de subrotina for concluída (é mais eficiente do que guardar endereços de retorno no registro de ativação da pilha).

Implementação de Processos

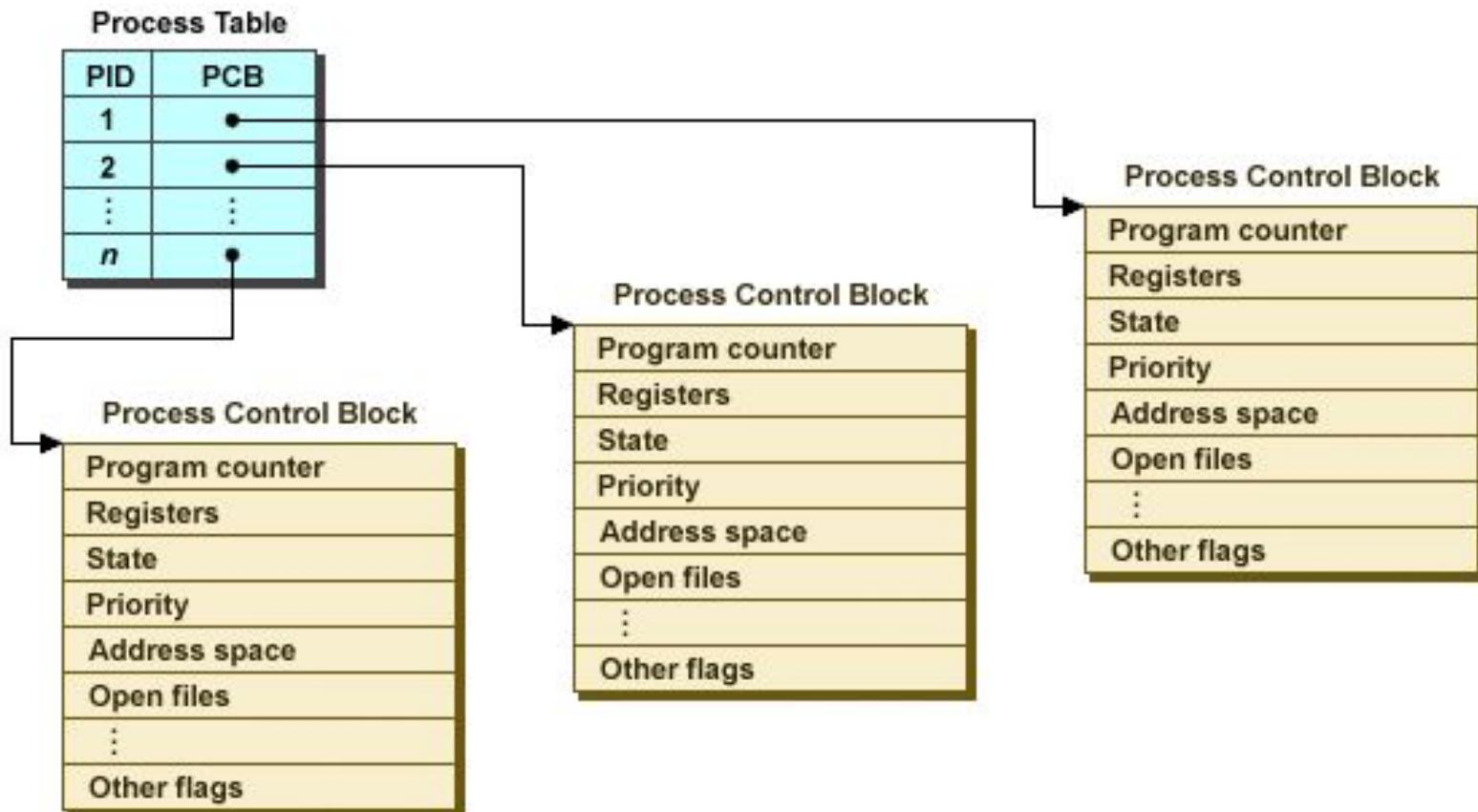
A cada processo estão associadas informações sobre o seu estado de execução (o seu *contexto de execução*),

Gerenciamento de processos	Gerenciamento de memória	Gerenciamento de arquivos
Registradores Contador de programa Palavra de estado do programa Ponteiro de pilha Estado do processo Prioridade Parâmetros de escalonamento Identificador (ID) do processo Processo pai Grupo do processo Sinais Momento em que o processo iniciou Tempo usado da CPU Tempo de CPU do filho Momento do próximo alarme	Ponteiro para o segmento de código Ponteiro para o segmento de dados Ponteiro para o segmento de pilha	Diretório-raiz Diretório de trabalho Descritores de arquivos Identificador (ID) do usuário Identificador (ID) do grupo

Fig.: Contexto de cada processo

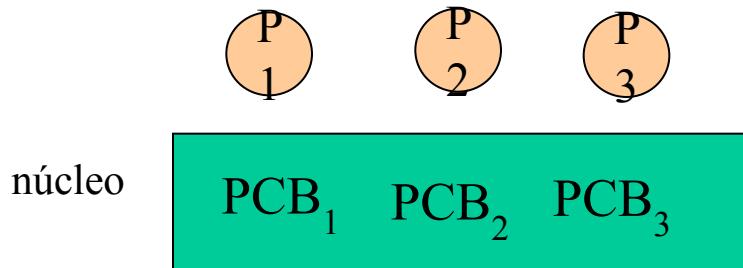
Essas e outras informações são armazenadas na Tabela de Processos e na Área U

Process Table e PCB



O Process Control Block (PCB) é uma estrutura no núcleo que contém o estado de cada processo, é referenciado por uma entrada na Tabela de

Process Control Block (PCB)



PCB contém as informações que são necessárias para gerenciar os processos e viabilizar a troca de contexto entre eles.

Para ser capaz de reiniciar um processo interrompido (ou esperando) o estado anterior em que deixou a CPU precisa ser restaurado;

Carrega-se a CPU (e o Mapeador de Memória) com os dados de contexto armazenados no PCB

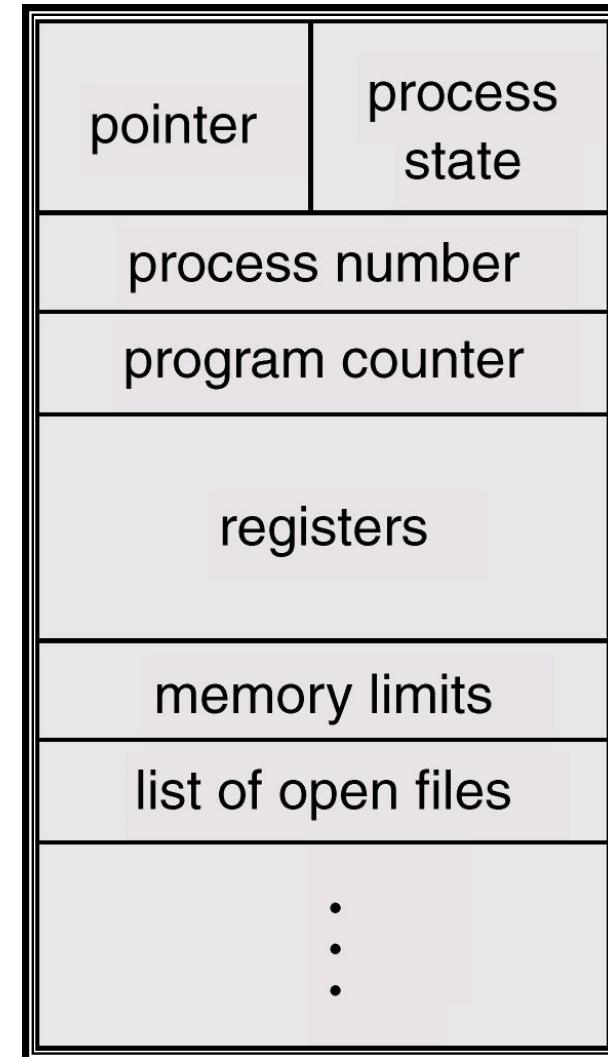


Imagen de um processo

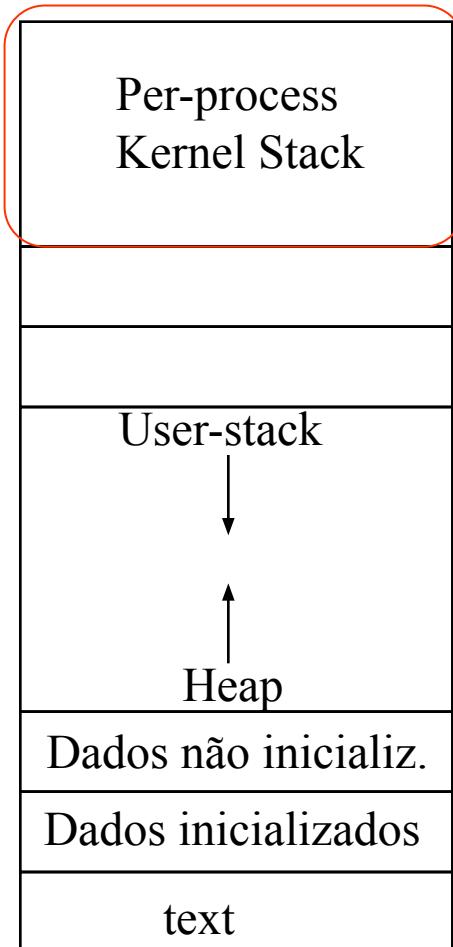


Imagen do processo na Memória
(espaço de endereçamento)

Pilha de procedimentos do kernel?

Sim, pois o tratamento de uma interrupção (ou `sys_call`) pode envolver vários procedimentos do núcleo, e tudo precisa ser salvo quando chegar a próxima interrupção.
Obs: acesso a essa área apenas no modo kernel.

Imagen do executável no Disco

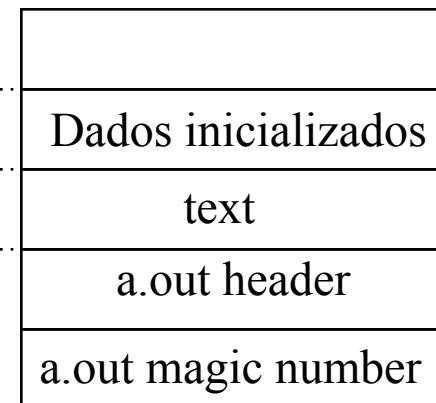


Tabela de Processos e Área U

O núcleo contém uma tabela de processos com uma entrada que descreve o estado de cada processo ativo no sistema.

Cada processo também possui uma **Área u** (na área de memória do processo) com informações adicionais que controlam a operação de um processo.

A entrada da tabela de processos e a área u fazem parte do contexto de um processo.

Área U (U area) de um processo

A área U é uma extensão da entrada na tabela de processos que fica no espaço de memória do processo.

Tabela de processos contém:

Estado do processo

UserID

A área U contém:

Ponteiro para a entrada na tabela de processos

Descritores de Arquivos e todos os arquivos abertos

Diretório corrente e diretório raiz

Parametros de I/O

Limites de tamanho do processo

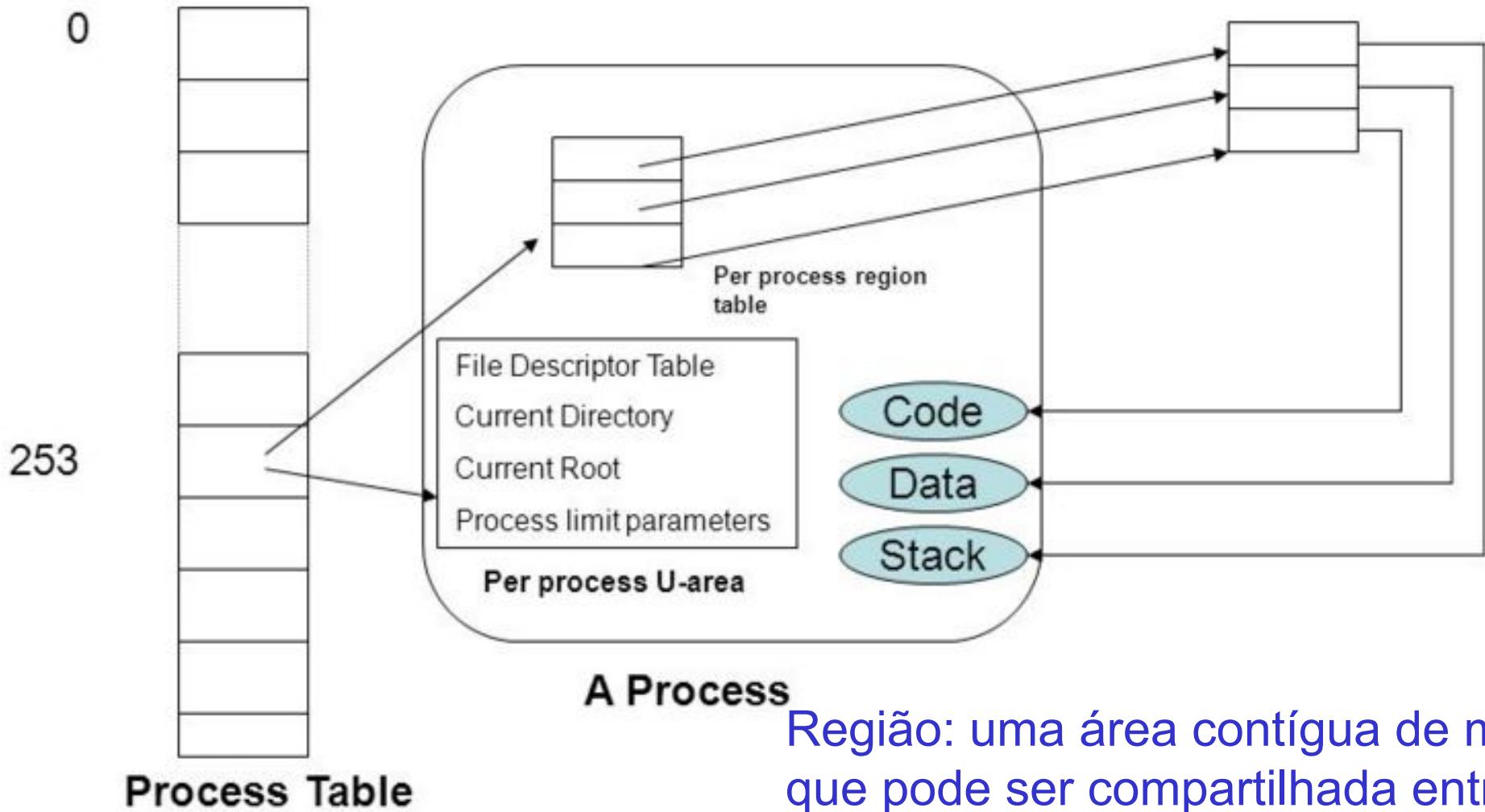
como o processo vai reagir aos sinais.

o "terminal de login" associado ao processo, se houver um.

O Núcleo tem acesso a área U do processo em execução, mas não dos demais processos

Área U

kernel region table



Região: uma área contígua de memória que pode ser compartilhada entre processos

Estado do processo na área U

- Os IDs de usuário reais e efetivos - privilégios permitidos ao processo, como direitos de acesso a arquivos.
- o tempo que o processo passou sendo executado no modo de usuário e no modo de kernel.
- Como o processo vai reagir aos sinais.
- o "terminal de login" associado ao processo, se houver um.
- O campo Error registra os erros obtidos durante as recentes chamadas de sistema.
- O diretório atual e a raiz atual descrevem o ambiente do sistema de arquivos do processo.
- A tabela de descritores de arquivos abertos.
- Os campos de limite restringem o tamanho de um processo e o tamanho de um arquivo que ele pode gravar.

Region Table

- Unix divide o espaço de endereço virtual de um processo em regiões lógicas.
- Uma região é uma área contígua do espaço de endereço virtual que é tratada como um objeto distinto a ser compartilhado ou protegido.
- Assim, texto, dados e pilha geralmente formam regiões separadas de um processo.
- Mas processos podem compartilhar uma região.
- Por exemplo, vários processos podem executar o mesmo programa, e para tal compartilham uma cópia da região de texto.
- vários processos podem cooperar para compartilhar uma região comum de memória compartilhada.
- A Kernel Region Table armazena todas as regiões ativas no sistema.

Tabela de Processos (proc)

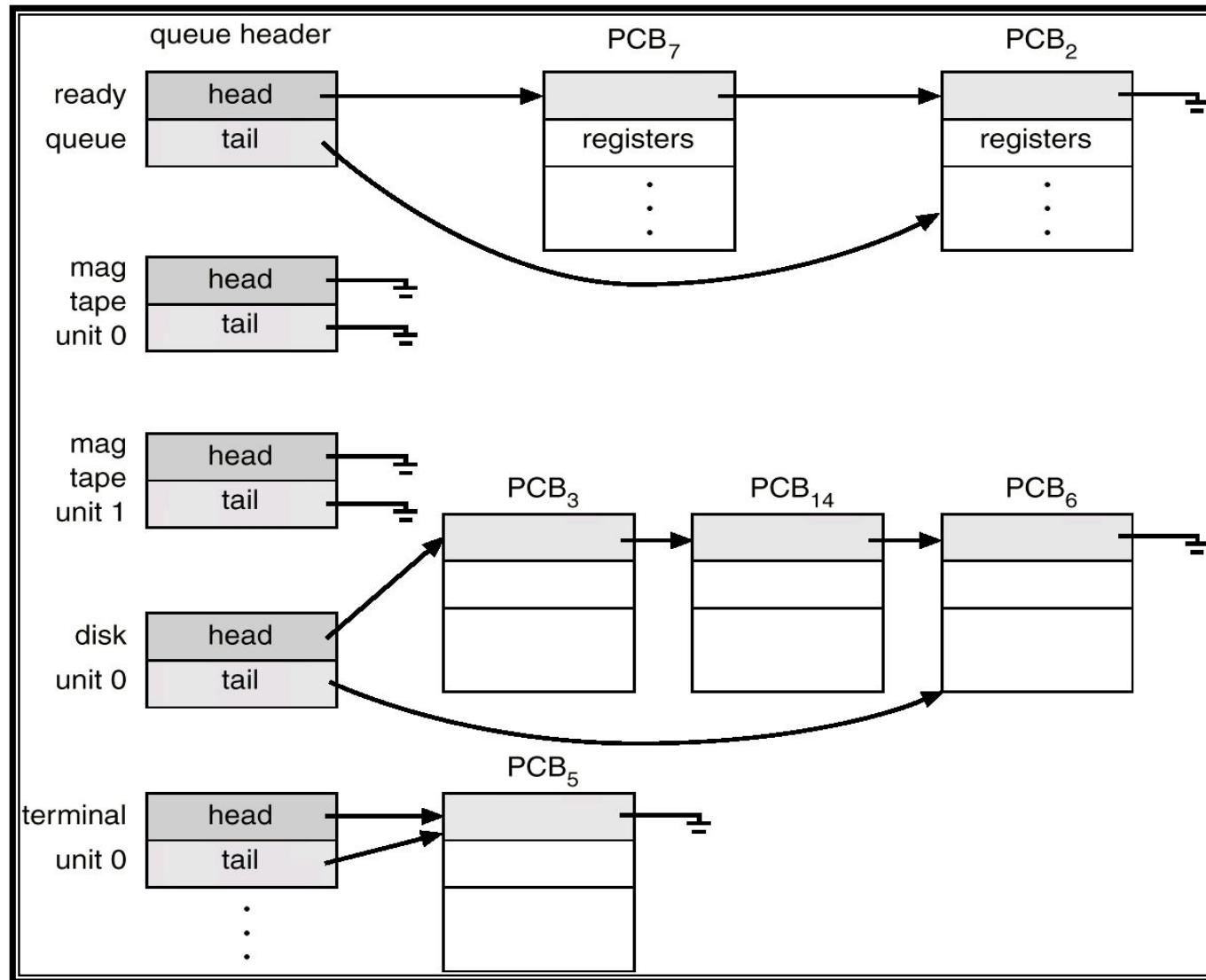
Além do PCB, o núcleo gerencia uma tabela de processos, com informações adicionais por processo;

É uma área no núcleo com informações sobre todos os processos, tais como:

- PID
- Endereço do PCB do processo
- Estado do processo
- Ponteiros entre processos nas filas de prontos/bloqueados (usados pelo escalonador)
- Ponteiros para os processos pai, filho e irmão
- Informação para o tratamento de sinais (máscaras, etc.)
- Informação para gerenciamento de memória,
- Informações estatísticas
- etc.

Obs1: Conjuntamente, o PCB e a entrada na Tabela de Processos contém todas as informações necessárias para a gerência dos processos.

Filas dos prontos e de espera por E/S



Troca de Contexto

Consiste de salvar o estado dos recursos em uso (especialmente estado dos registradores da CPU) no PCB do processo interrompido,

E após tratar a interrupção, carregar a CPU com o estado salvo (PC, registradores, stack pointer, PSW, etc.) do processo que irá continuar

A troca de contexto precisa ser:

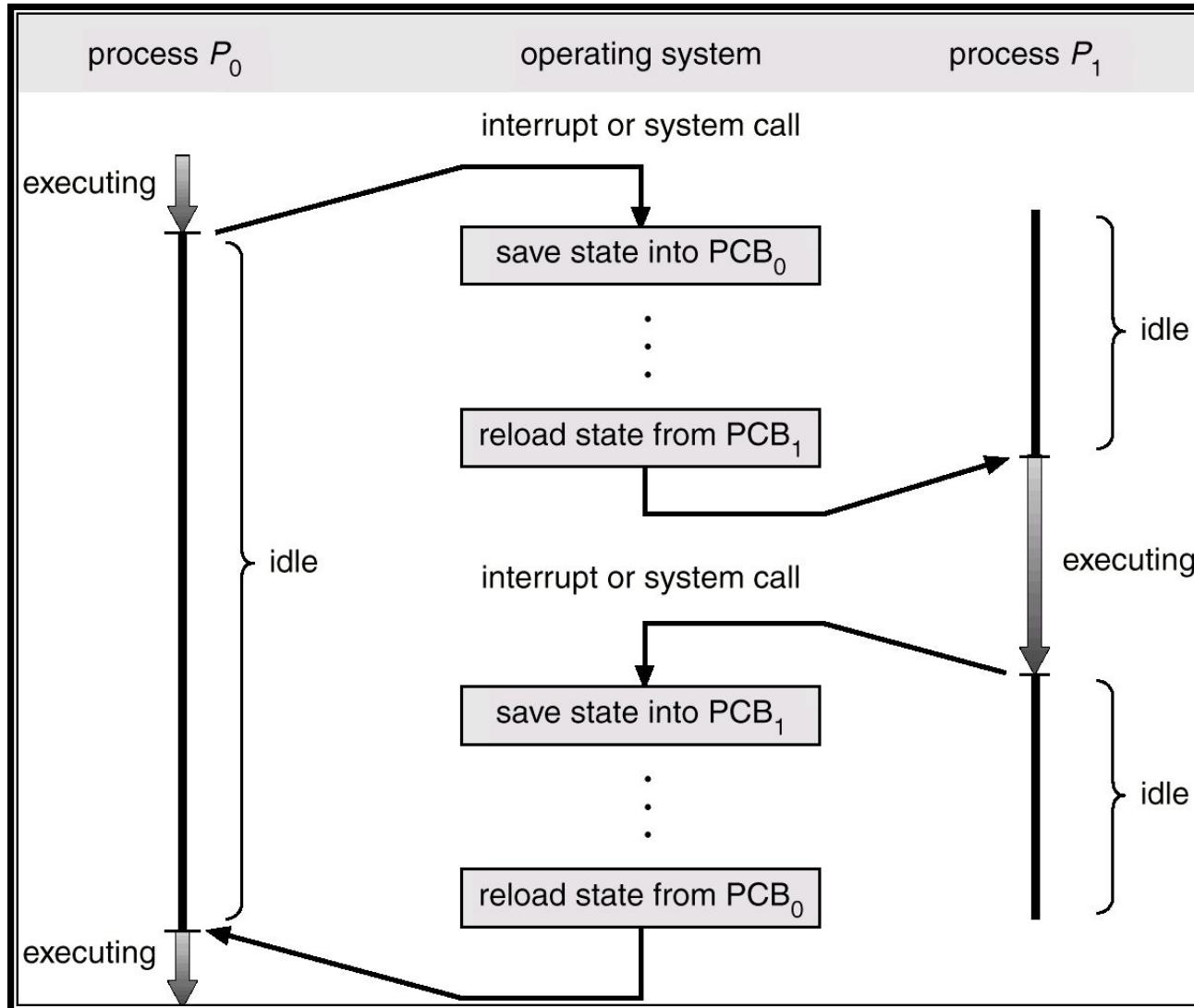
1. **Completa e consistente**
2. **Muito rápida**

Para garantir 1) o núcleo não pode ser interrompido durante o salvamento e re-carregamento do contexto, isto é, precisa-se **garantir a atomicidade** da operação

O salvamento/carregamento do contexto é realizada por um tratador de interrupção genérico, ou **tratador de interrupção de primeiro nível**. Este geralmente é programado em assembler.

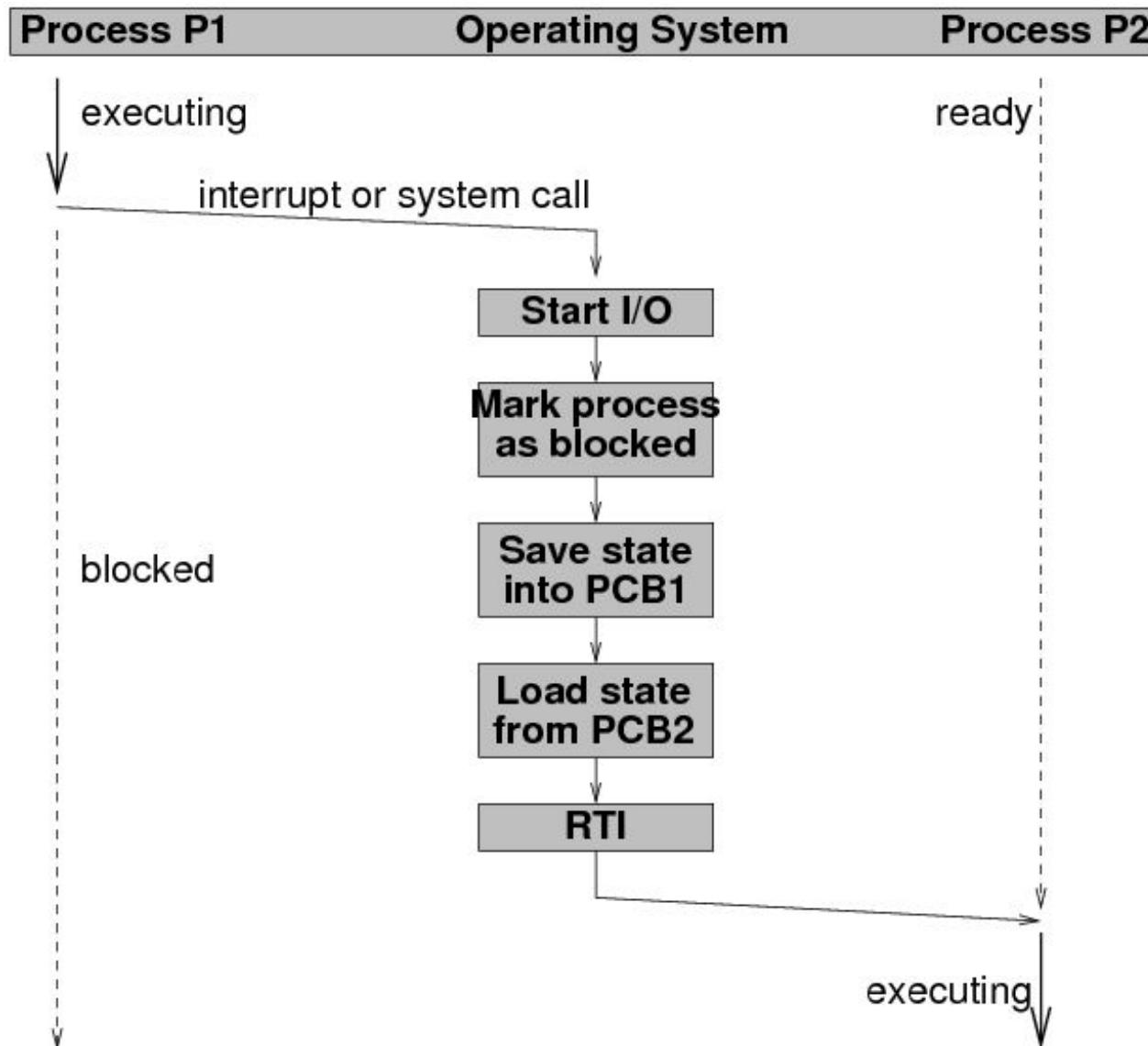
Troca de Contexto

Ocorre em todos sistemas multiprogramados. É essencial para a multiplexação da CPU



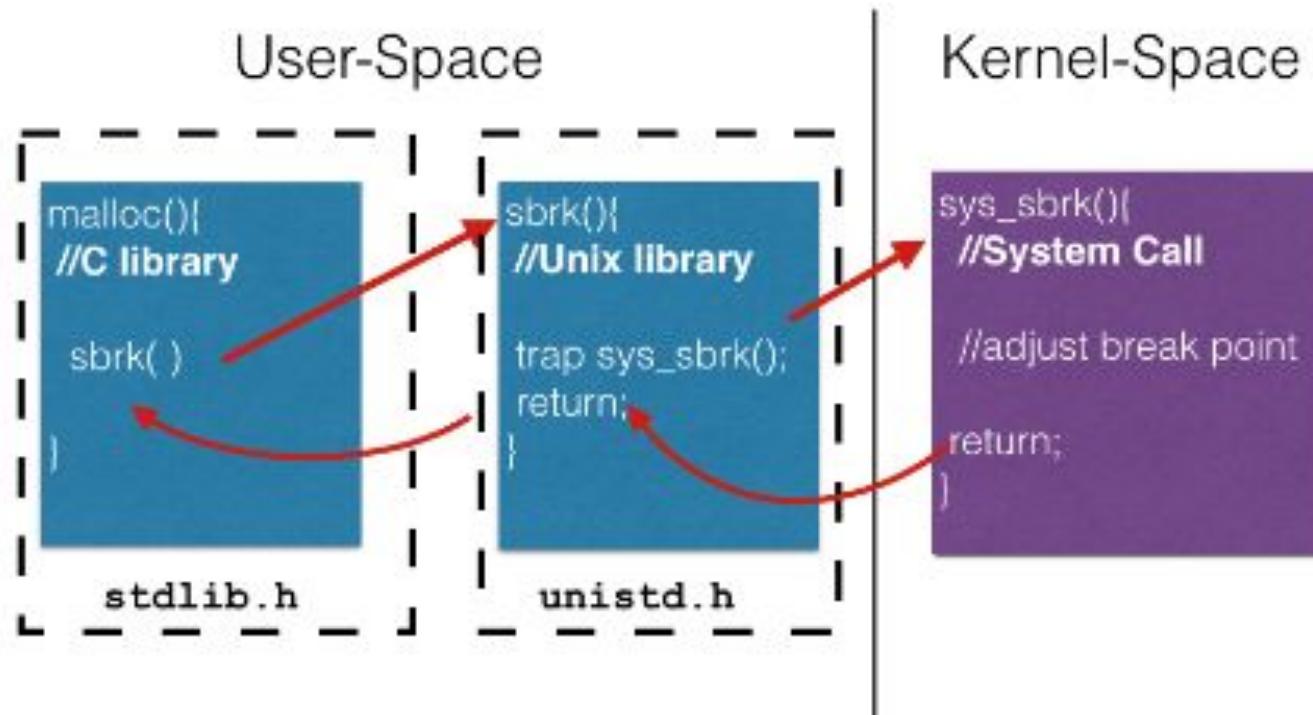
Troca de Contexto: P1 \rightarrow P2

A interrupção pode ser de hardware ou software (system call)

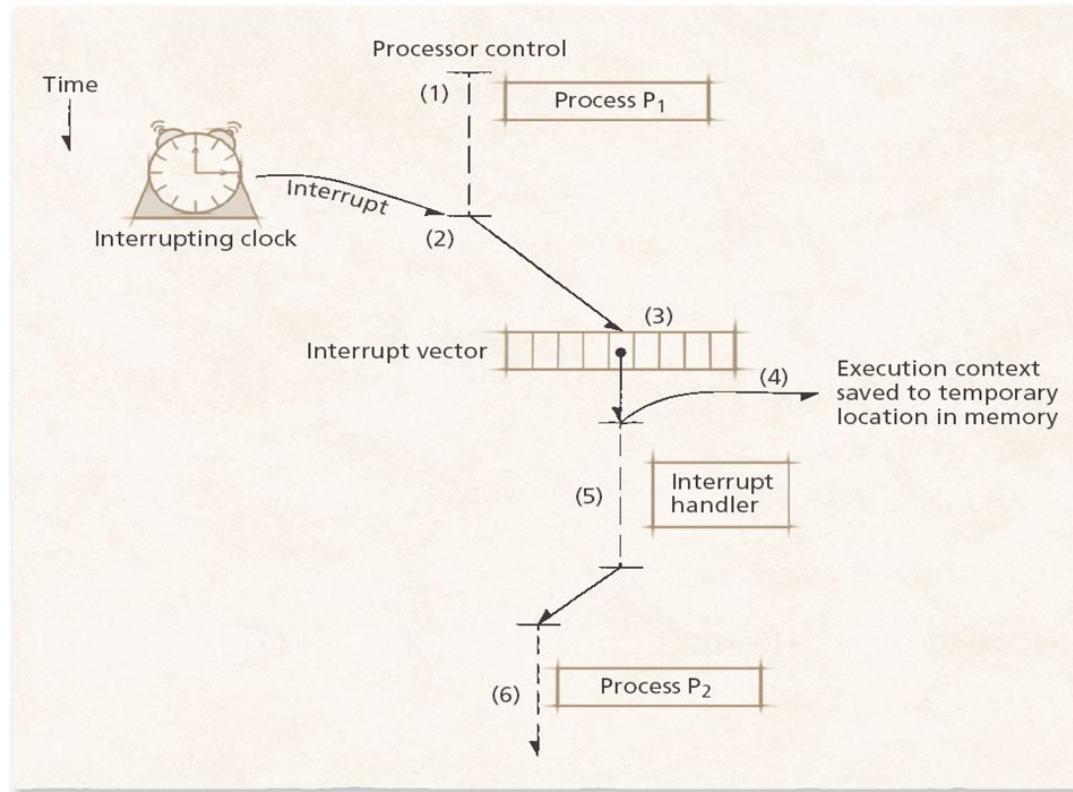


Exemplo de Chamada de Sistema em Unix

Chamando a função malloc()



Tratamento de Interrupções



- Para desviar o controle de execução (para o tratador da interrupção), o contexto precisa ser salvo.
- Ao receber uma interrupção, o HW carrega o endereço contido na entrada do vetor de interrupção.
- Mas o tratamento da interrupção é executado no mesmo fluxo de controle do processo que estava executando.

Interrupções vs. Exceções

O conjunto de interrupções depende da arquitetura do sistema.

A especificação da Intel Architecture IA-32 distingue dois tipos de sinais que um processador pode receber:

- Interrupções
 - Notificam o processador que um evento ocorreu, e/ou que o estado de um recurso (p.ex. dispositivo de E/S)mudou
 - Gerado por um dispositivo externo ao processador
- Exceção
 - Indica a ocorrência de um erro, de hardware ou causado por uma instrução sendo executada
 - Classificados como faults, traps or aborts

Interrupções são gatilho para troca de contexto

São 4 tipos de interrupção reconhecidos pela Intel Architecture IA-32::

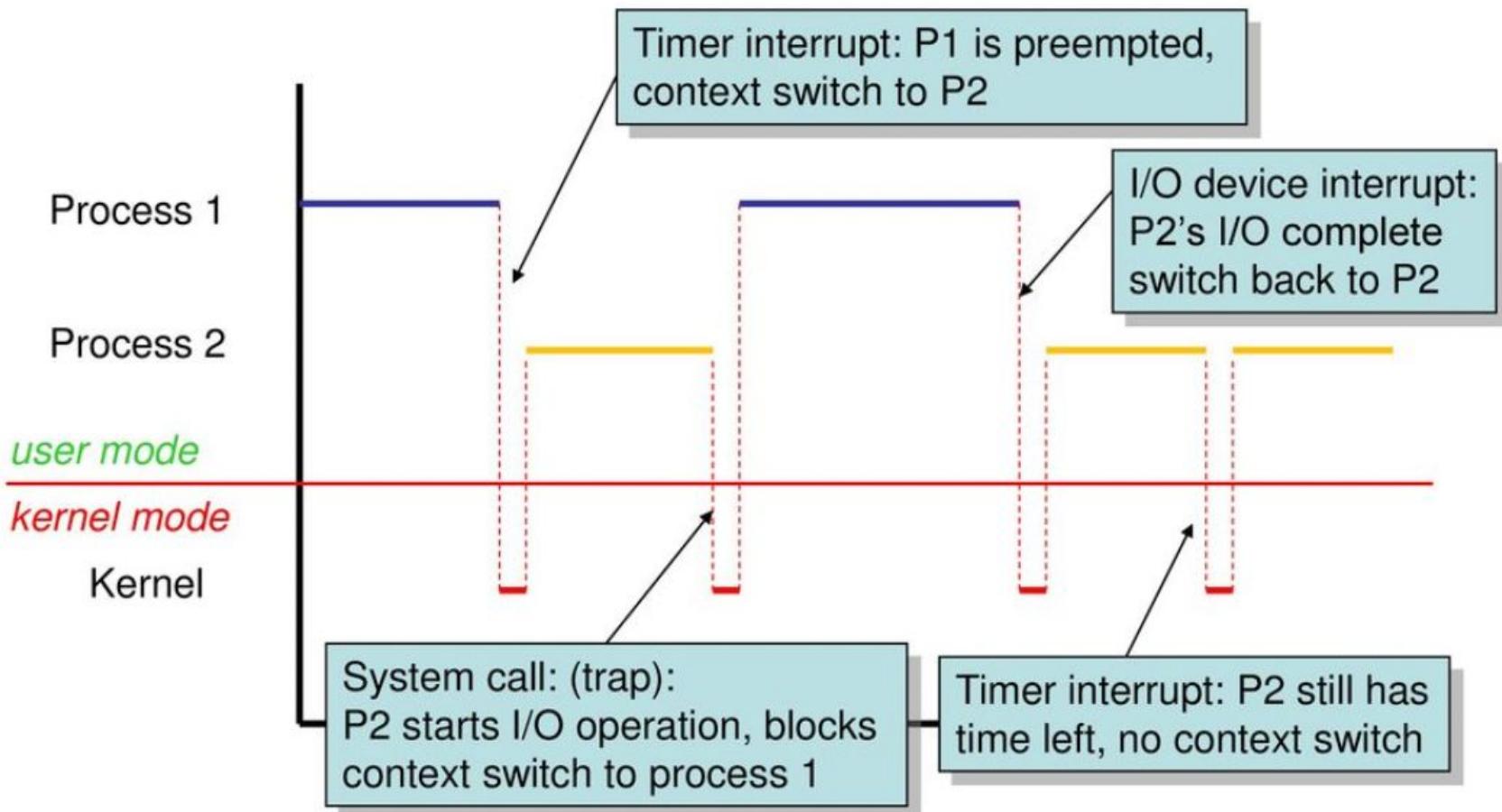
<u>Tipo</u>	<u>Descrição para cada tipo</u>
E/S	Iniciados pelo HW, notificam o processador de que o estado do dispositivo de E/S mudou (p.ex. E/S finalizada)
Timer	evento periódico para agendamento de uma ação e/ou monitoramento de desempenho. Por exemplo, tempo do processo atual expirou.
Inter-CPU	Usados em sistemas multiprocesadores, para comunicação e sincronização entre processadores
Trap	exceção (divisão por zero, segmentation fault, etc.) ou chamada de sistema

Lista de Interrupções

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode. ¹
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction. ²
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory. ³
18	#MC	Machine Check	Error codes (if any) and source are model dependent. ⁴
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction ⁵
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT <i>n</i> instruction.

<https://images.app.goo.gl/SYip86RwyB32QPZm9>

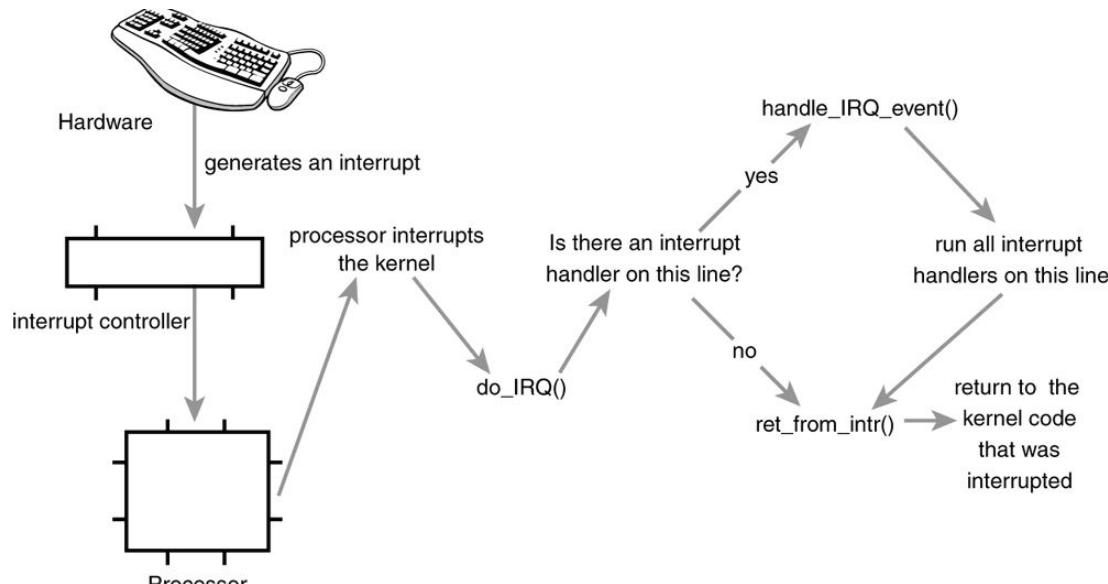
Exemplos de Interrupções em Trocas de Contexto



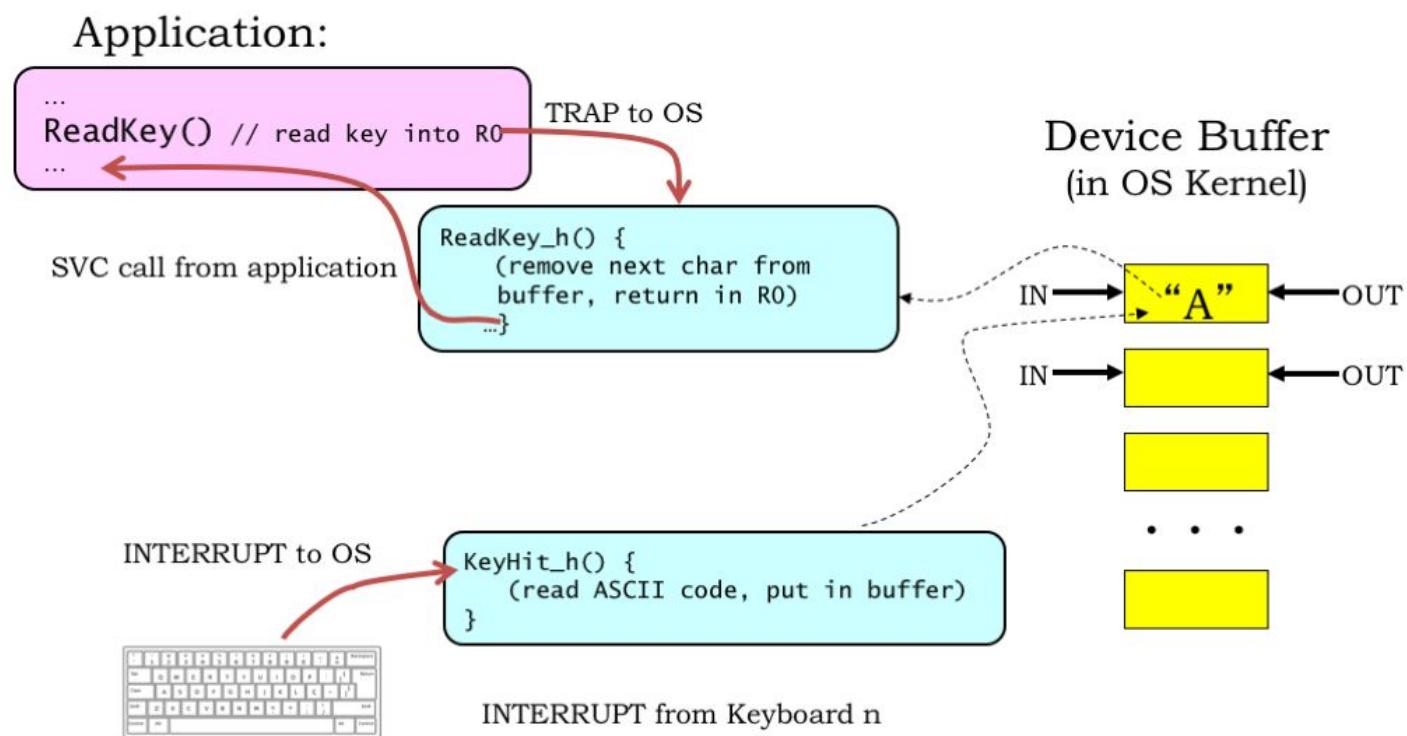
Interrupções

Sequência de ações realizadas pelo núcleo quando ocorre uma interrupção

1. O hardware empilha o contador de programa etc.
2. O hardware carrega o novo contador de programa a partir do vetor de interrupção.
3. O procedimento em linguagem de montagem salva os registradores.
4. O procedimento em linguagem de montagem configura uma nova pilha.
5. O serviço de interrupção em C executa (em geral lê e armazena temporariamente a entrada).
6. O escalonador decide qual processo é o próximo a executar.
7. O procedimento em C retorna para o código em linguagem de montagem.
8. O procedimento em linguagem de montagem inicia o novo processo atual.



Interruções de Software & Hardware



<https://images.app.goo.gl/b7xPSMzdnnndFyr7p8>

Tratamento de Interrupções em detalhes

Interrupções de HW ou SW:

- I/O Interrupt (I/O Device)
- Segmentation Fault -> Error
- System Call -> Trap
- send message -> Trap
- Clock Interrupt

First Level Int. Handler (FLIH), em Assembly

1. desabilita interrupções
2. salva contexto em tabela de processos/PCB
3. cria nova pilha temporária no kernel
4. carrega no PC o end. do Vetor de Interrupções
5. habilita interrupções

Tratador de interrupção específico():

1. trata a interrupção (p.ex. Escreve/le dados de buffer do driver)
2. se algum processo pode ser desbloqueado então chama escalonador
3. retorna

Dispatcher, em Assembly:

1. desabilita interrupções
2. carrega o contexto na CPU & mapeamento de memória do processo a ser executado
3. habilita interrupções

Scheduler():

- insere o processo desbloqueado na fila de prontos
- Escolhe próximo processo
- retorna

Tratamento de Interrupções

Etapas executadas quando ocorre uma interrupção

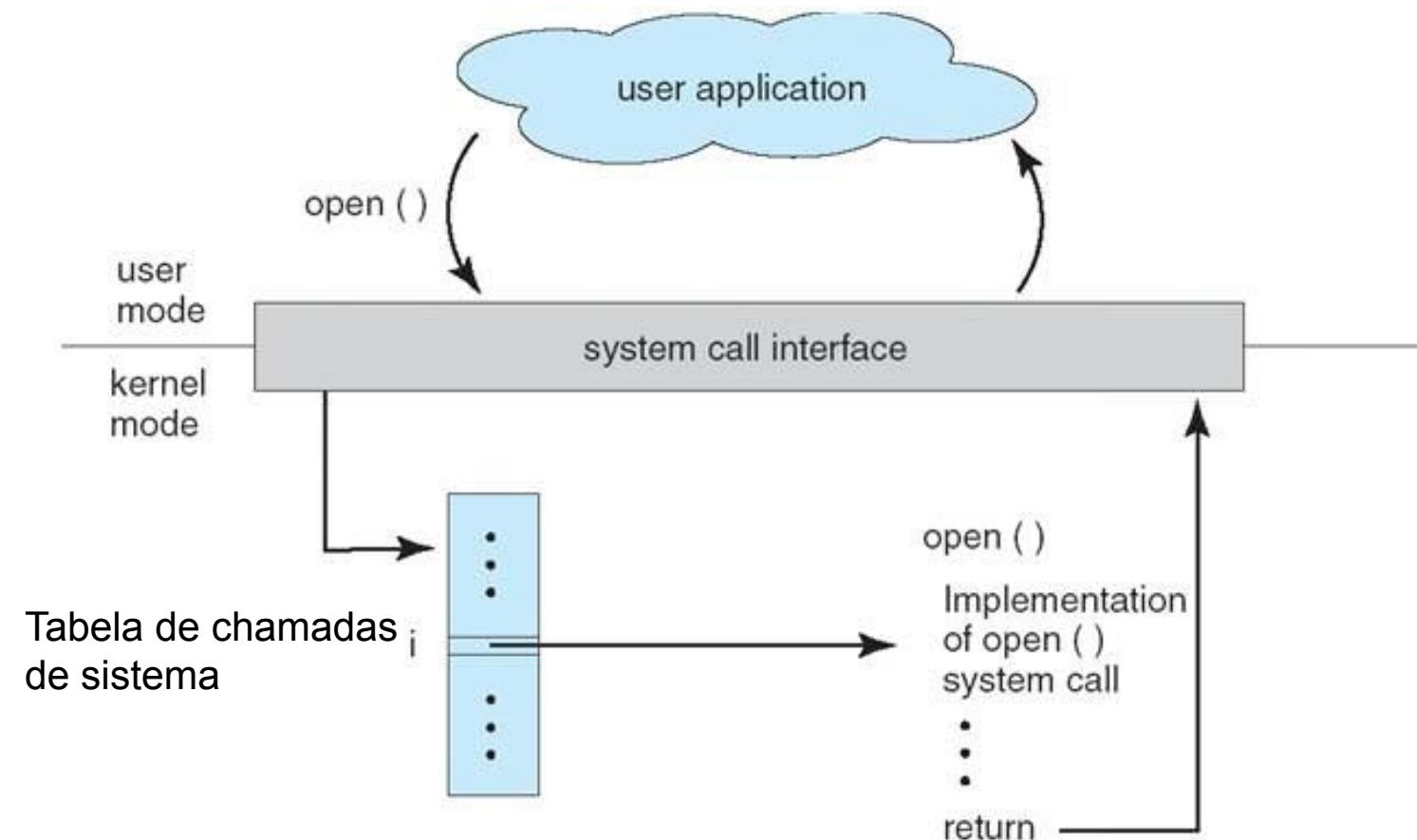
1. O hardware empilha o contador de programa etc.
2. O hardware carrega o novo contador de programa a partir do vetor de interrupção.
3. O procedimento em linguagem de montagem salva os registradores.
4. O procedimento em linguagem de montagem configura uma nova pilha.
5. O serviço de interrupção em C executa (em geral lê e armazena temporariamente a entrada).
6. O escalonador decide qual processo é o próximo a executar.
7. O procedimento em C retorna para o código em linguagem de montagem.
8. O procedimento em linguagem de montagem inicia o novo processo atual.

Vetor de Interrupção:

- Localizado em endereço baixo de memória (núcleo)
- Uma entrada para cada tipo de interrupção (trap, clock, E/S)
- Cada entrada contém endereço de um procedimento tratador da interrupção (Interrupt Service Routine -ISR) que fará a ação necessária para tratar a interrupção

Chamada de Sistema

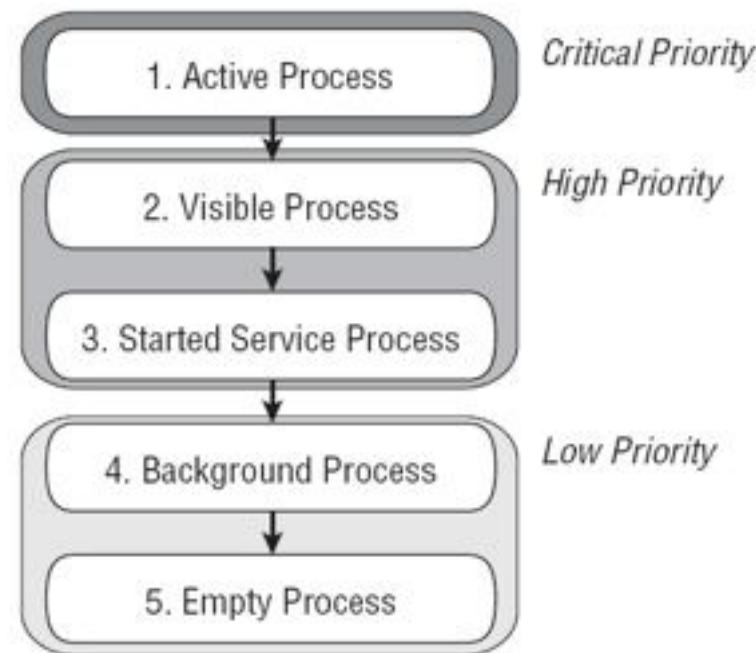
Mecânica de uma chamada de sistemas



Processos em Android



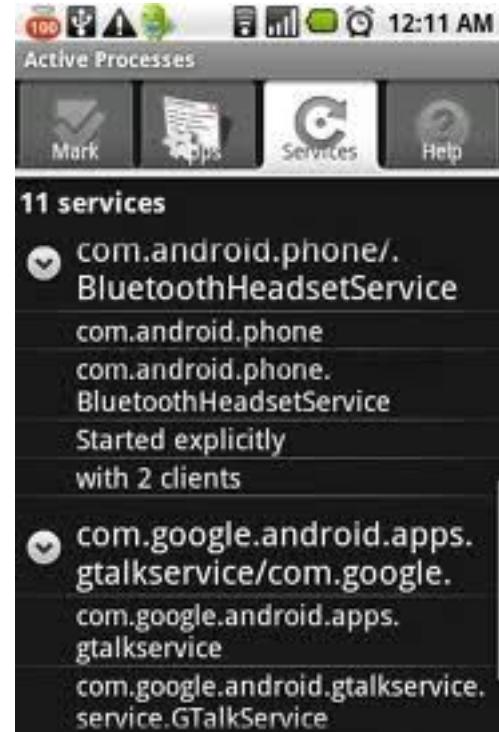
- Cada aplicativo Android executa em um processo separado, e tem a sua própria instância de Dalvik (máquina virtual Java)
- O Dalvik é uma VM otimizada, de forma que muitas instâncias de Dalvik podem executar em um smartphone
- Dalvik acessa os recursos de HW do dispositivo através de chamadas ao Linux kernel
- Os executáveis que rodam na VM Dalvik (.dex) usarem pouca memória RAM
- Processos com baixa prioridade podem ser terminados
- Aplicativos podem interagir com "Serviços" que são processos executando em background



Fonte: <http://mobworld.wordpress.com>
Memory Management in Android

Visualizar/Gerenciar Processos Unix e Android

```
dhcpdip52:~ endler$ ps -eld
UID PID PPID  P CPU PRI NI   SZ RSS WCHAN S ADDR TTY      TIME CMD
 0 142 127  4100 0 31 0 603440 876 -  S 73a9d20 ??    0:09.24 /Library/Dropbox
501 310 100  4004 0 63 0 4911556 8112 -  S 5f09540 ??    0:15.41 /System/Library/
501 7825 7819  4000 0 42 0 636264 36412 -  S 7336d20 ??    3:37.87 /Applications/Fi
501 7953 7819  4000 0 42 0 412808 3600 -  S 70b6a80 ??    0:00.42 /Applications/Fi
dhcpdip52:~ endler$
```



ps, kill, etc.

Active Processes Viewer/killer

Escalonamento de Processos



Escalonamento

- É uma função essencial de um sistema operacional multiprogramado/ multitarefa
- Define qual dos processos no estado pronto vai ser o próximo a ganhar o controle da CPU (precisa estar carregado na memória principal)
- Também gerencia a/s filas de processos prontos à espera da CPU
- Assim, aumenta-se a utilização da CPU, que nunca fica ociosa (a espera por E/S não impede o progresso de algum outro processo)

Escalonamento

É a tarefa de escolher o próximo processo a ser executado pela CPU

- e determinar até quando o processo escolhido executa
- e gerenciar as filas de processos prontos para executar e
- atribuir prioridades

Trata-se da tarefa de mais alta prioridade e que faz parte do nível mais baixo do Sistema Operacional.

Escalonamento

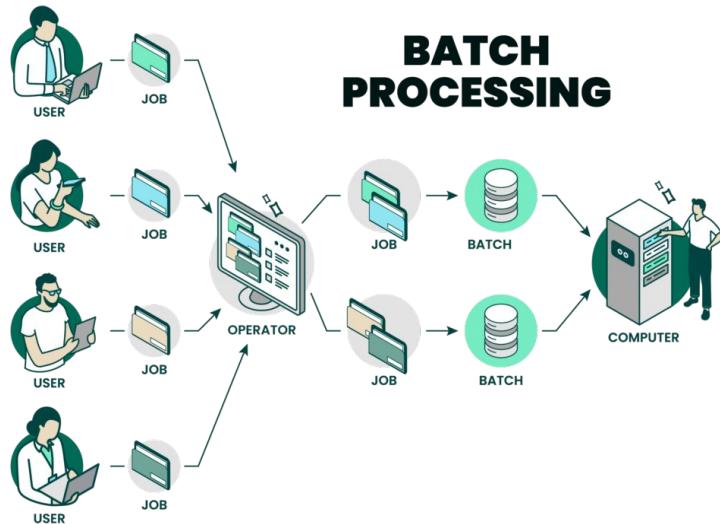
- Em sistemas multi-programados (multi-tarefa), a cada instante um ou mais processos podem estar no estado *pronto*, e.g.:
 - Processos do sistema vs processos de usuários
 - Processos curtos vs longos/eternos
 - processos interativos (IO-bound) vs intensivos em CPU (CPU-bound) ou
 - jobs em lote (batch)

O Escalonador é um serviço do kernel responsável por:

- gerenciar a/s fila/s de prontos, ajustando prioridades dos processos, trocando-os de fila, etc.
- escolher qual dos processos prontos vai ser o próximo a usar CPU (de acordo com as prioridades dos processos)

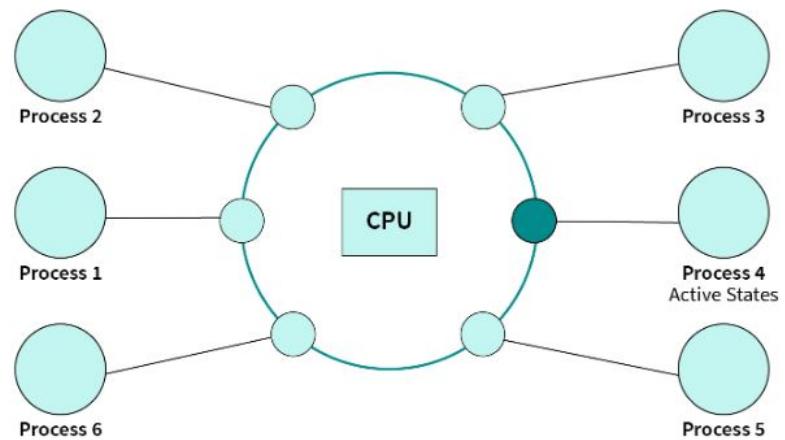
Políticas de Escalonamento

Sistemas Batch



obs: assume-se tempo de CPU conhecido

Sistemas Interativos:



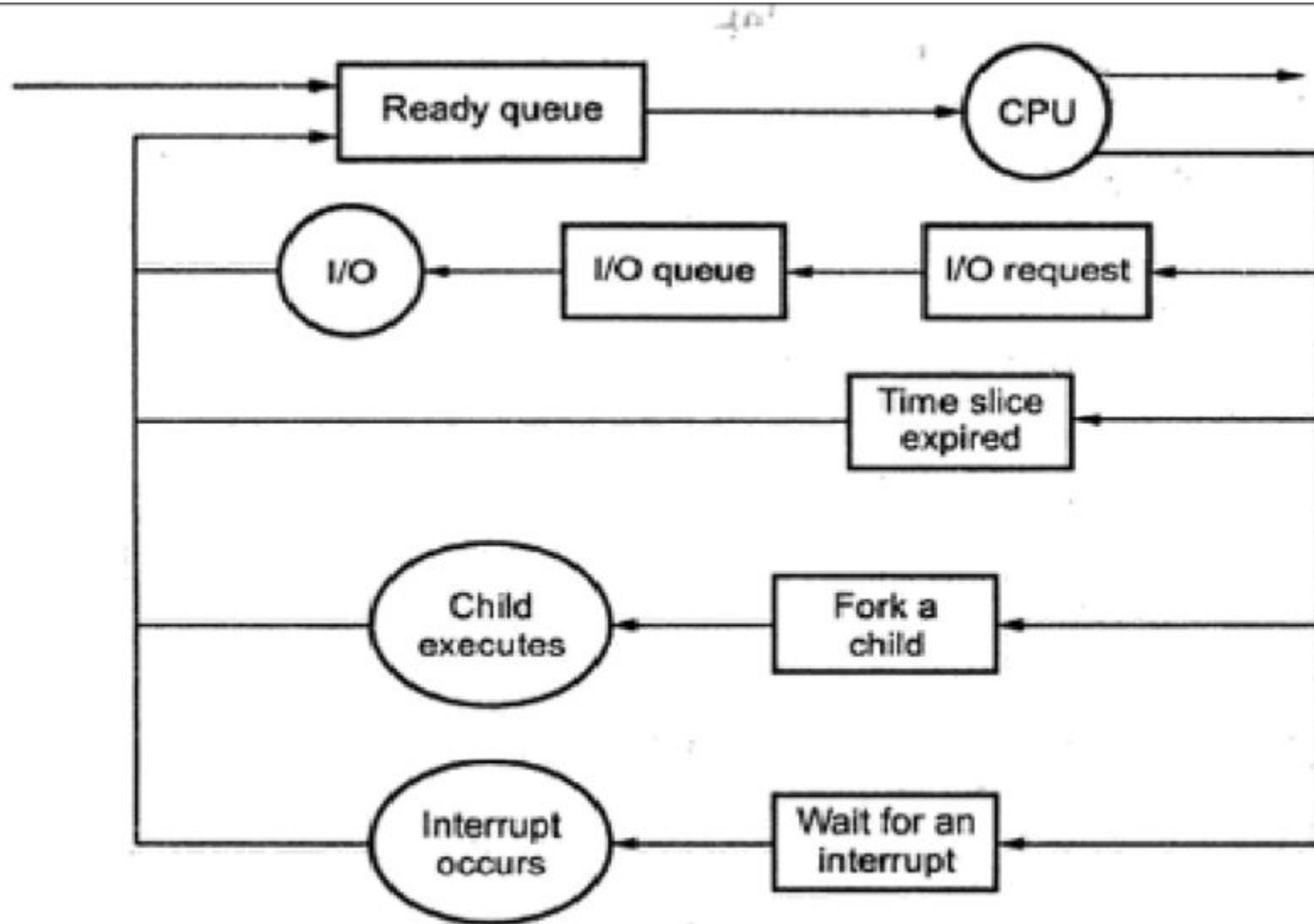
obs: com compartilhamento de tempo

Possíveis objetivos do Escalonamento

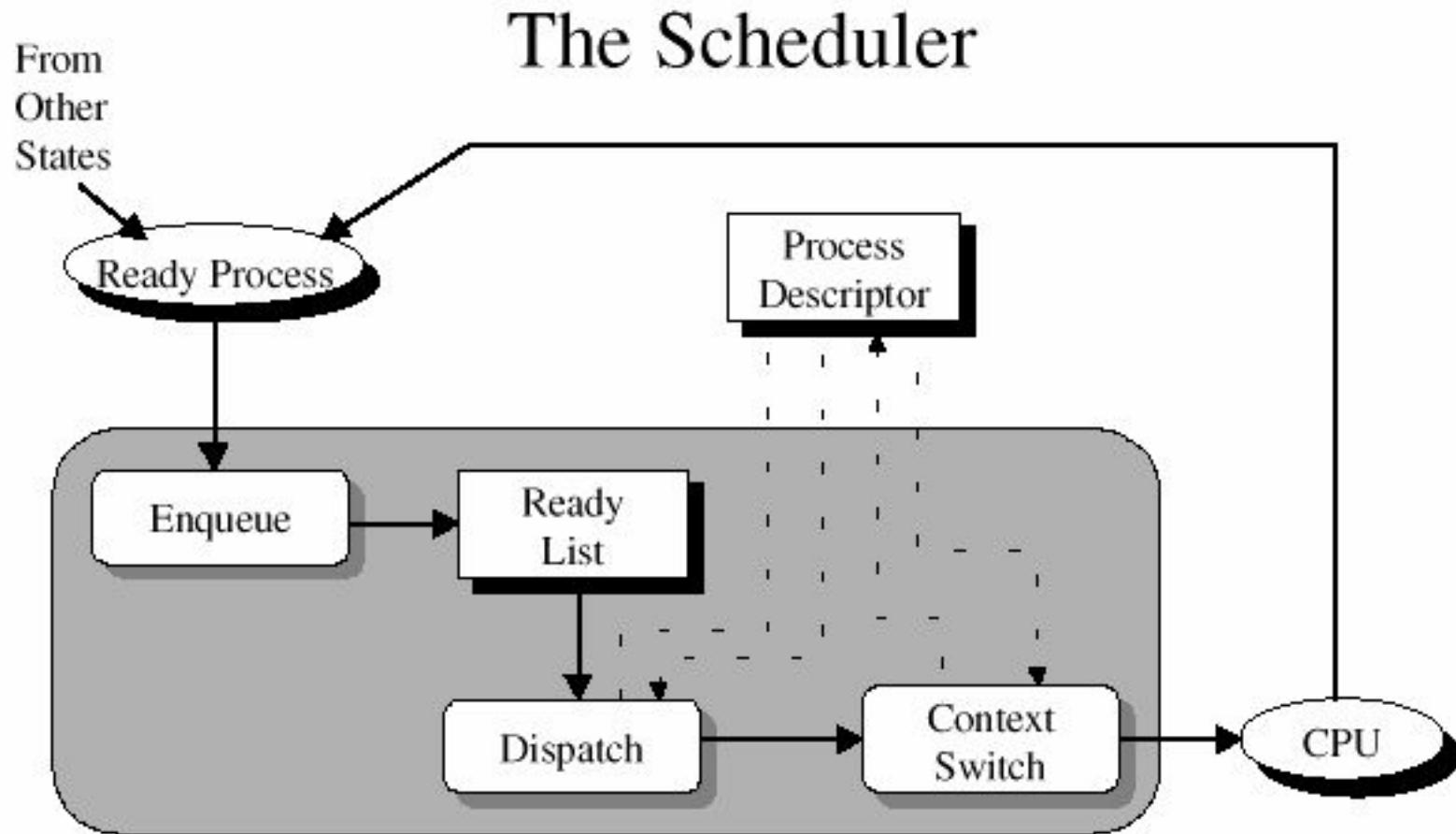
Algoritmos de escalonamento podem ter diferentes objetivos:

- Garantir **justiça** (*fairness*): cada processo ganha mesmo tempo de CPU
- Aumentar eficiência: **maximizar a utilização de CPU** (perto de 100%)
- **Minimizar o tempo médio de resposta** (para sistemas interativos)
- Minimizar de tempo médio de permanência dos processos no sistema (para processamento batch: o Δt entre início-fim)
- **Maximizar vazão**: aumentar o número de processos concluídos por unidade de tempo

Escalonamento (gerenciamento de filas de espera)



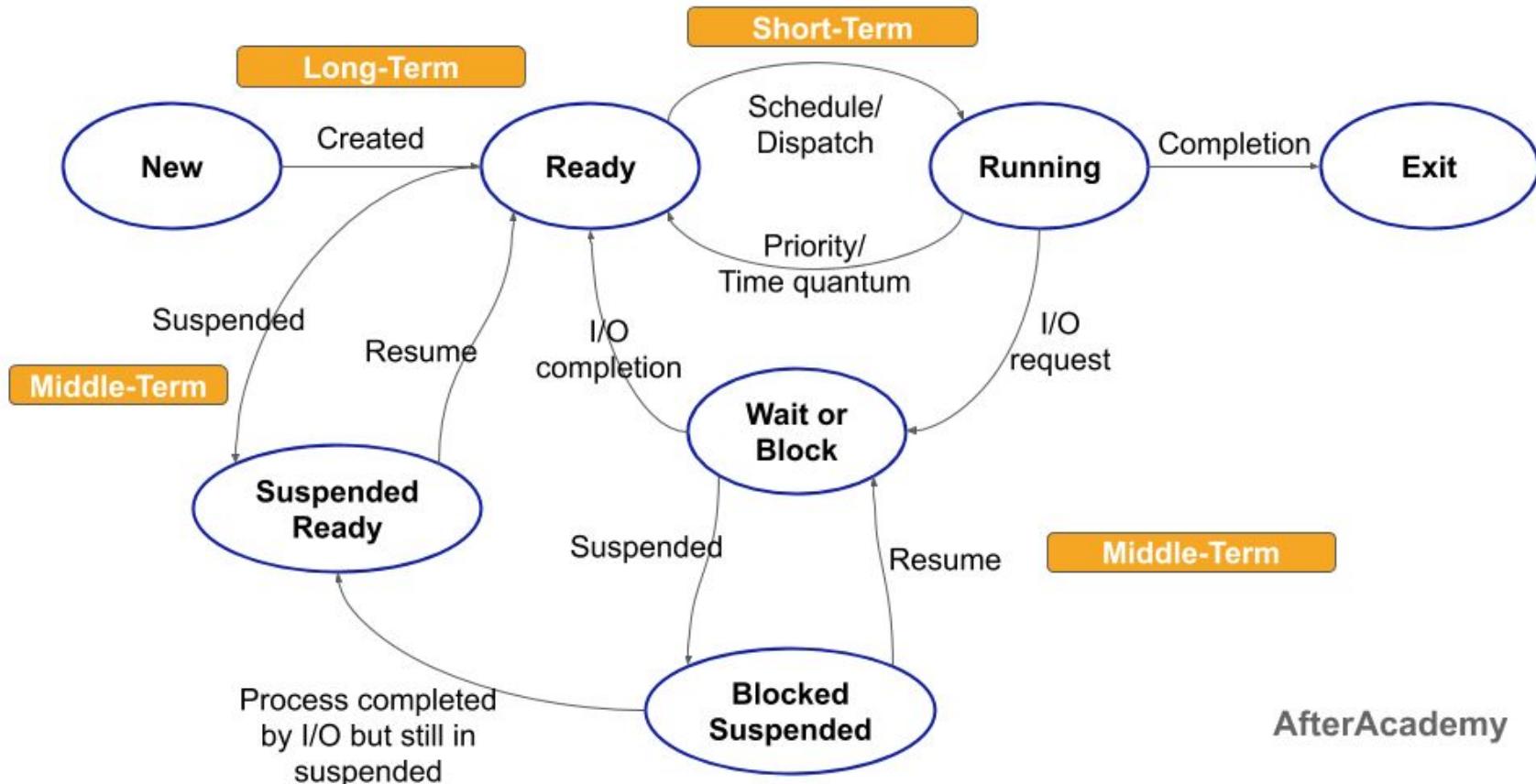
O Ciclo do Escalonador (curto prazo)



Níveis de escalonamento

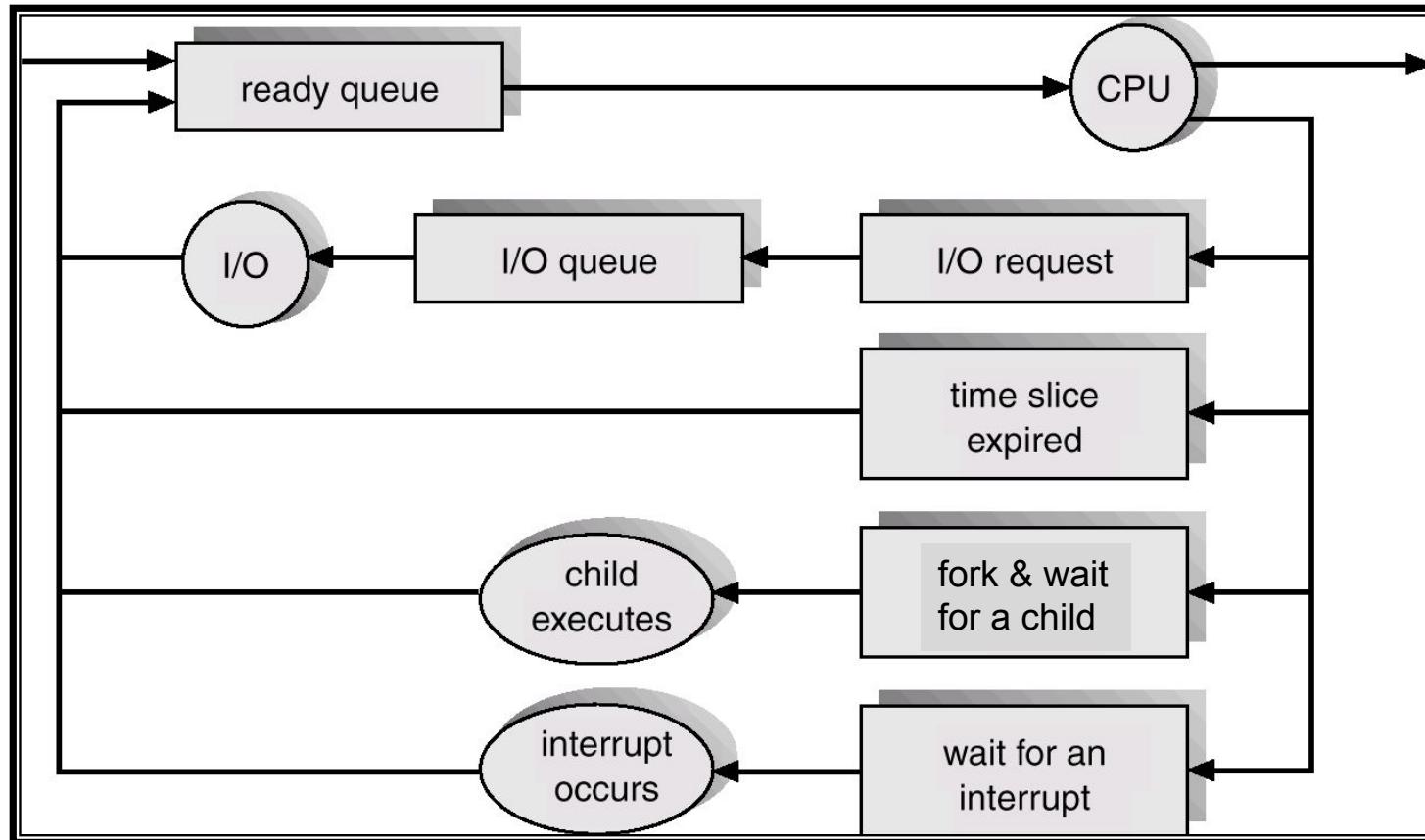
1. Escalonamento de longo prazo (Job Scheduler)
 - Determina quando um novo processo é inserido no sistema para execução
 - ao ser criado, processo recebe uma prioridade (vai para uma fila dos prontos)
2. Escalonamento de médio prazo
 - Escolhe o processo fora da memória (e.g. “swapped out”) a ser carregado para a memória principal (RAM)
 - ajuda a manter um equilíbrio entre I/O bound e CPU bound, e controla o grau de multi-programação
3. Escalonamento de curto prazo (“dispatching”)
 - Escolhe um dos processos da fila de prontos para usar a CPU.

Escalonamento de longo, médio ou curto prazo



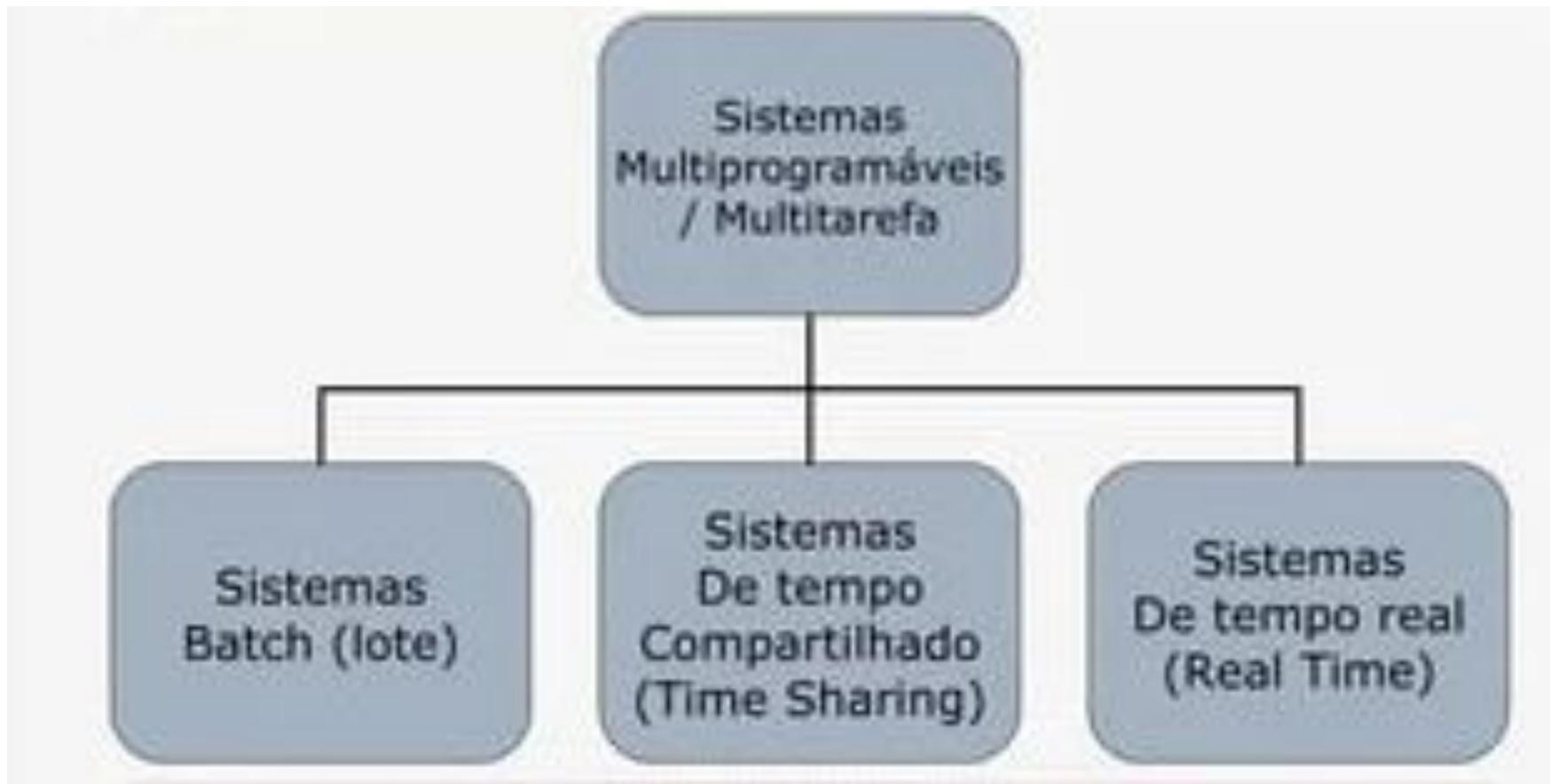
Fonte: <https://images.app.goo.gl/iuZXnloGxmC3CsKu8>

Escalonamento de Curto Prazo



ações e eventos que fazem um processo entrar na fila dos prontos

Tipos de sistemas operacionais



Sistemas multi-tarefa: de tempo compartilhado, em lote e tempo real

sistemas de tempo compartilhado otimizam o uso de recursos, sobretudo, a CPU,

- permitem que vários processos realizem suas tarefas alternadamente em uma única CPU. (cada um ganha quantum de tempo)
- tempo compartilhado é adotado em sistemas onde múltiplos usuários precisam acessar e interagir com o computador ao mesmo tempo.

sistemas de processamento em lote (modo batch)

- conjunto de tarefas/jobs/processos são executadas em lotes, ou seja, em grupos.
- As tarefas não têm interação humana.
- Essas tarefas são organizadas em filas de espera e são processadas de forma sequencial, uma após a outra.
- gerencia-se a execução dessas tarefas na ordem em que estão no lote

sistemas de tempo real: os processos são escalonados sempre de forma a que a E/S ocorra no prazo necessário para controlar aparelho do mundo real

Escalonamento

Tipos de sistemas e objetivos do escalonamento

All systems

Fairness - giving each process a fair share of the CPU

Policy enforcement - seeing that stated policy is carried out

Balance - keeping all parts of the system busy

Batch systems

Throughput - maximize jobs per hour

Turnaround time - minimize time between submission and termination

CPU utilization - keep the CPU busy all the time

Interactive systems

Response time - respond to requests quickly

Proportionality - meet users' expectations

Real-time systems

Meeting deadlines - avoid losing data

Predictability - avoid quality degradation in multimedia systems

Parâmetros dos algoritmos de escalonamento

Sistemas em lote (Batch)

- Taxa de execução (throughput): máximo número de jobs executados por unidade de tempo;
- Turnaround time (tempo de retorno): tempo total até o processo estar concluído;
- Tempo de espera: tempo gasto na fila de prontos;
- Eficiência: CPU deve estar 100% do tempo ocupada.

Sistemas interativos

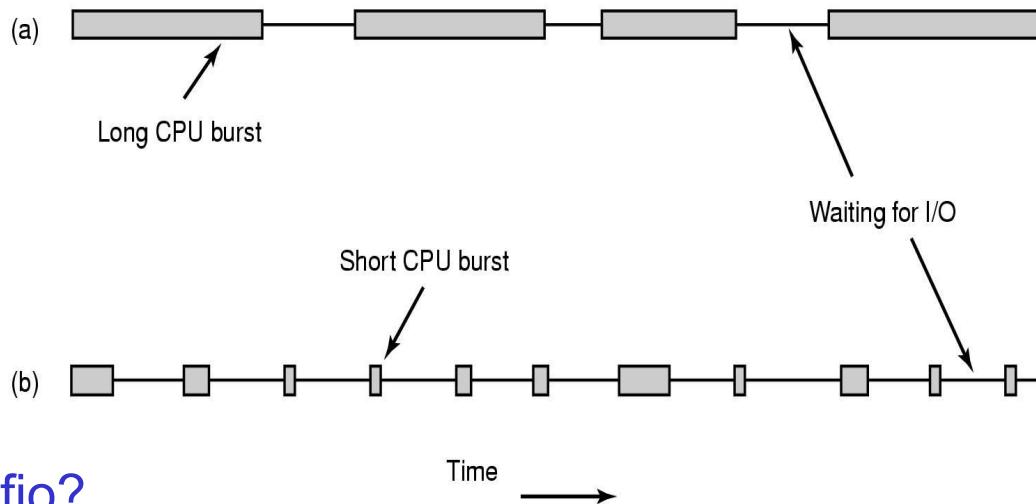
- Tempo de resposta: tempo esperando para iniciar execução;
- Justiça no atendimento dos usuários.

Sistemas de tempo real:

- Atender os prazos de E/S

Política/Algoritmo de Escalonamento

- Mesmo em sistemas interativos os processos têm perfis diferentes de rajadas de processamento (CPU burst) e de frequência de espera de E/S
- Processos com E/S frequente (ditos, **I/O bound**) devem receber uma prioridade maior. Assim, consegue-se atender um conjunto maior de processos, e com justiça



Principal desafio?

- como definir essas prioridades se o comportamento futuro de cada processo não é previsível (i.e. de quanto será a sua proxima rajada de CPU? e quais serão os períodos de E/S frequente?)

Formas de implementar o escalonador

1. “modo embutido” no tratamento a uma interrupção
 - Ao final do tratamento da interrupção, o procedimento para escalonamento é chamado
 - Executa como parte do fluxo de controle do processo que estava em execução
2. “modo autônomo” - executa como um processo independente – de máxima prioridade
 - Escalonador possui sua própria CPU (em arq multi-core) e entra em ação sempre que for necessário
 - é executado periodicamente para verificar se deve trocar o processo do usuário sendo executado
 - Ocorre uma alternância entre o processo escalonador e os demais processos
 -

Tipos de Escalonamento

Dentre as políticas de escalonamento, existem duas macro-categorias:

Diferença com relação ao grau de interferência nos processos:

- **preemptivo**: escalonador pode interromper o processo em execução, e escolhendo outro processo para executar
- **não-preemptivo**: escalonador só é chamado quando processo é bloqueado (chamada de sistema), ou terminar
- **ao método de seleção do próximo processo:**
 - Uso da função Prioridade(processo)
 - Regra de desempate (para processos com mesma prioridade)
 - Escolha aleatória
 - Cronológica (FIFO)
 - Cíclica (*Round Robin*)
 - Outra informação sobre a tarefa (p.ex. prazos para a E/S)

Escalonamento com prioridades

- Prioridade estática: não muda ao longo da execução do processo;
- Prioridade dinâmica: muda em tempo de execução:
 - Escalonador aumenta ou diminui dependendo do recente consumo de CPU
 - Próprio processo pode aumentar/diminuir sua prioridade (Chamada **nice** do Unix)
- Implementando níveis de Prioridades:
 - várias filas de pronto F , uma para cada nível de prioridade
 - Sejam processos P e Q:
 - P em F_i sempre executa antes de Q em F_j se $i > j$
 - Ordenação de P, Q na mesma F segue critério específico

Escalonamento Preemptivo: como clock ticks são tratados?

- Interrupção clock tick é o 2º. mais prioritário (e ocorre a cada 10 mseg, $(10^{-2}$ segundos)
- Tarefas do tratador:
 - Re-arma o clock (se necessário)
 - Atualiza estatísticas sobre uso de CPU do processo corrente
 - Re-cálculo de prioridades e tratamento de quantum expirado
 - Envia um sinal SIGXCPU para processo corrente, caso seu quantum tenha expirado
 - Atualiza contador time-of-the-day
 - Executa funções agendadas do kernel (*callouts*)
 - Trata alarmes
- Algumas das tarefas apenas executadas apenas em *major clock ticks* (p.ex. cada 4 ou 10 ticks)

Algoritmo geral para um escalonador preemptivo para multi-core

```
Scheduler() {  
    do { // existe alguma CPU livre  
        Pegue o processo P mais prioritário de ready_a;  
        Seja cpu uma CPU livre;  
        if (cpu != NIL) Aloca_CPU(P,cpu);  
    } while (cpu != NIL);  
    do { // todas CPUs estão em uso  
        Pegue o processo P mais prioritário de ready_a;  
        Pegue o processo Q em execução de menor prioridade;  
        if (Priority(P) > Priority(Q)) troca Q por P;  
    } while (Priority(P) > Priority(Q));  
    if (self->Status.Type != 'running') Preempt(P,self);  
}
```

Callouts e Alarmes

callout = função que o kernel deve executar em um momento futuro, por exemplo:

- retransmissão de pacotes de rede
- Funções de gerenciamento do escalonador ou do gerente de memória
- Polling de dispositivos que não emitem interrupções

Obs: São mantidos em uma fila ordenada (a qualquer momento podem surir novos)

alarmes = solicitações “me acorda” feitos por processos ao kernel, para:
profiling, processos de tempo real, o tempo que o processo usou em user mode.

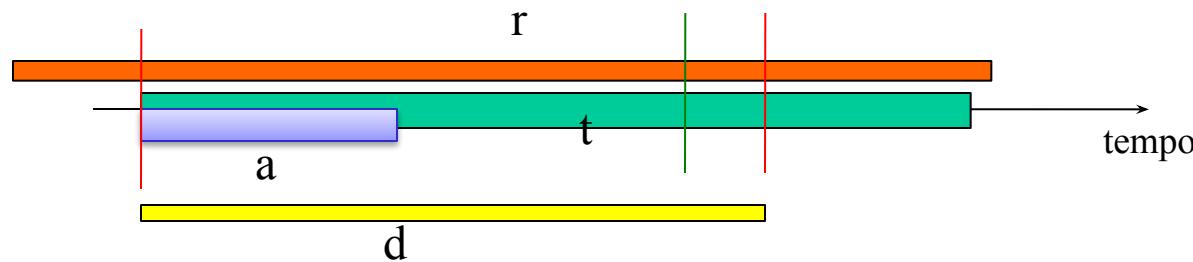
Possíveis Parâmetros de Escalonadores

- Parâmetros Internos (dados coletados pelo sistema)
 - Tipo do processo (sistema vs usuário)
 - quantidade de memória usada
 - Tempo total de CPU requisitado
 - Percentual da fatia de tempo utilizada
 - Tempo total de permanência no sistema
- Parâmetros Externos
 - Prazo para término de procesamento (Deadline)
 - Prioridade do usuário:
 - Administrador vs usuário normal
 - em função da importância de usuários na empresa,
 - em função do valor desembolsado para usar o sistema

Função Prioridade

- Possíveis parâmetros:

- a = tempo de serviço alcançado
- r = tempo de permanência no sistema
- t = tempo total de serviço
- d = Periodicidade (para tempo real)
- deadline (explícito ou definido pelo período)
- e = prioridade externa
- Quantidade de memória requisitada (p/ processamento em lotes)



Algoritmos de escalonamento

Nome, Modo decisão, Priorid., Desempate

FIFO: não-preemptivo $P = r$ randomico

SJF: não-preemptivo $P = -t$ cron./randomico

SRT: preemptivo $P = -(t-a)$ cron./randomico

RR: preemptivo $P = 0$ cíclico

ML: preemptivo $P = e$ cíclico

não-preemptivo $P = e$ cronológico

- n níveis de prioridade fixa
- nível P é atendido quando as filas n a P+1 estão vazias

SJF= Shortest Job First; SRT = Shortest Remaining Time; RR= RoundRobin;

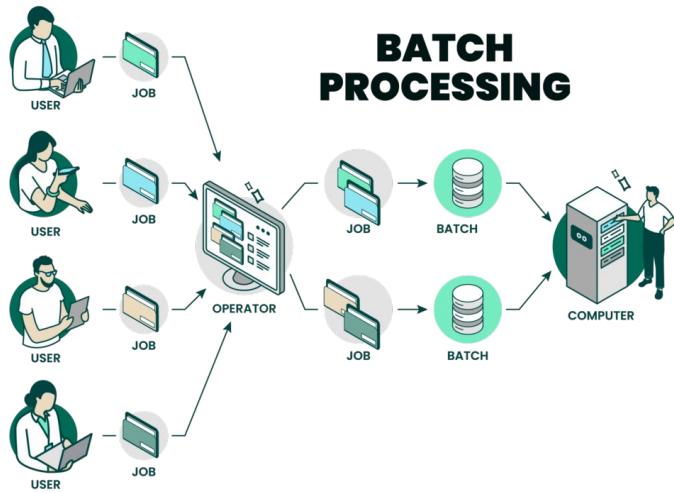
ML = Multi-level

a = serviço alcançado; r = permanência no sistema, t = tempo total de serviço

e = prioridade externa

Políticas de Escalonamento

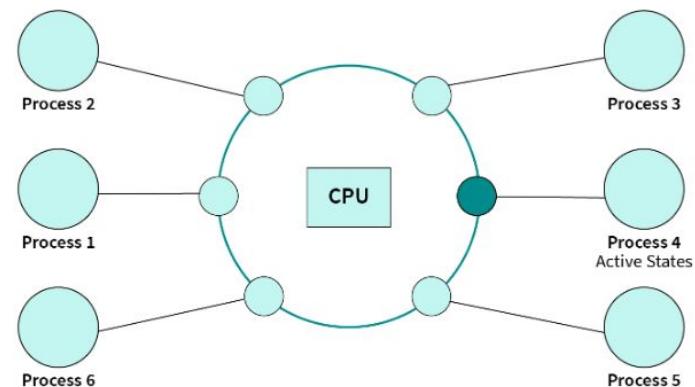
Sistemas Batch



- FCFS
- SJF
- SRTN

obs: assume-se tempo de CPU conhecido

Sistemas Interativos:



- Round-Robin
- Prioridades
- Multi-Level
- Multi-Level com feedback

First Come First Serve (FCFS/FIFO)

Política não-preemptiva muito simples para sistemas em lote: Processos são executados na CPU seguindo a ordem de chegada;

Ordem de chegada

Tempo de rajada de CPU

A

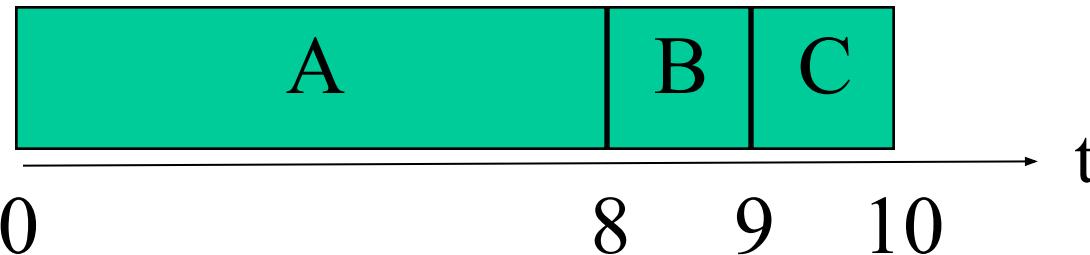
8

B

1

C

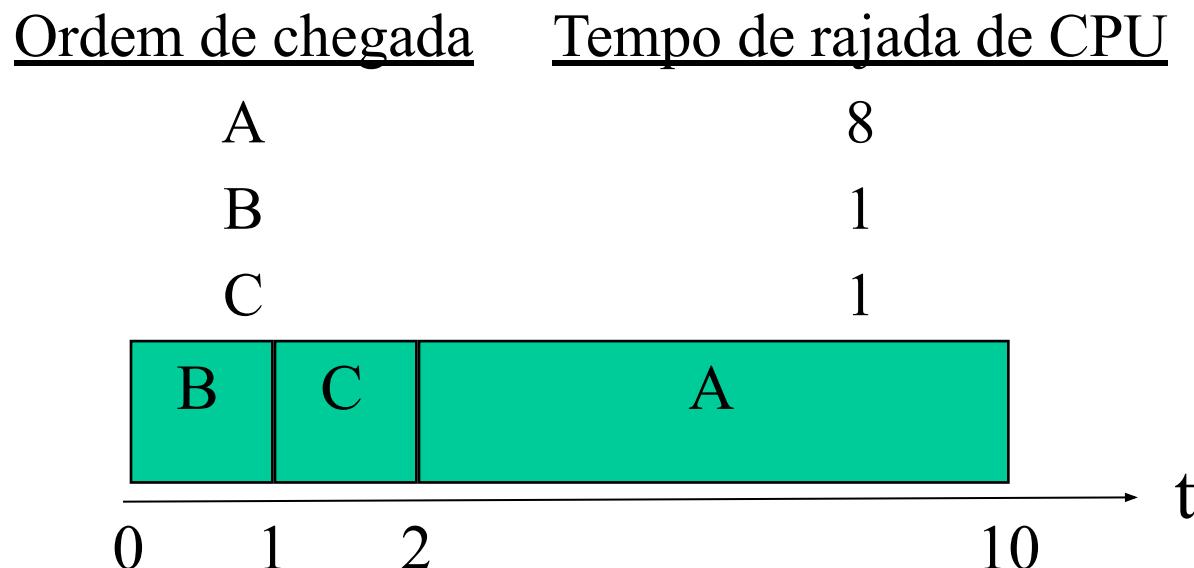
1



Tempo médio de espera do conjunto {A,B,C}
 $(0 + 8 + 9) / 3 = 5.7$

Shortest Job First (SJF)

- Escalonamento para sistemas em lote/batch
- Todos os processos são ordenados segundo os tempos esperados de CPU
- O processo de menor rajada de CPU é executado primeiro
- Assume-se: que é possível prever o tempo total de CPU de cada processo;

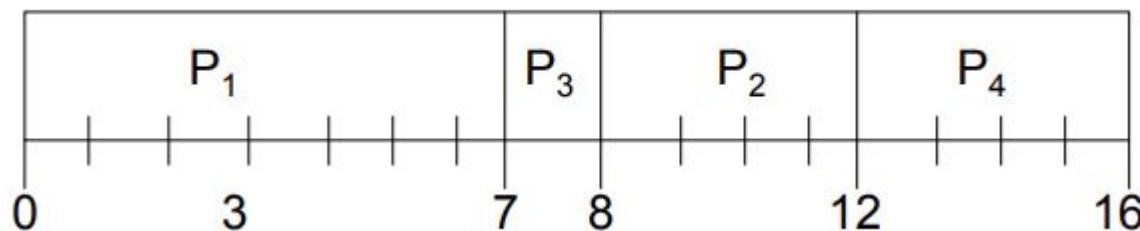


Tempo médio de espera ótimo para o conjunto {A,B,C}:

$$(0 + 1 + 2) / 3 = 1$$

Shortest Job First (SJF) não preemptivo

	Chegada	Rajada de CPU
P1	0	7
P2	2	4
P3	4	1
P4	5	4



Tempo médio de Espera de {P1,P2,P3,P4}: $0+6+3+7=16/4$
=> P3 entrou na frente de P2 por ter uma rajada menor
=> P2 e P4 estavam com mesma prioridade (extensão da rajada), mas P2 chegou primeiro

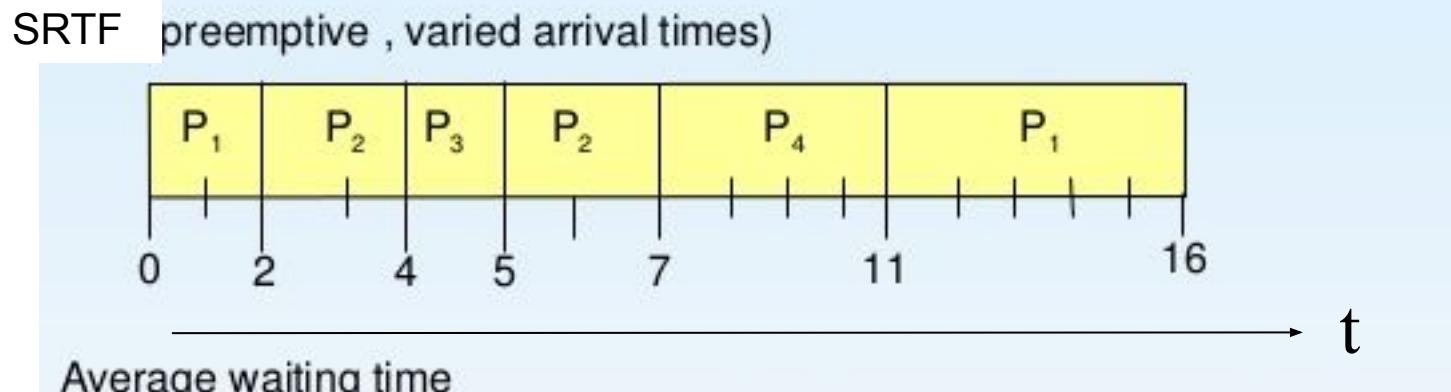
Shortest Remaining Time First (SRTF)

Menor Tempo remanescente

- É a variante preemptiva do escalonamento SJF (para sistemas batch)
- A cada novo processo que chega, o escalonador estima sua rajada de CPU;
- Se estimativa de rajada de CPU do novo processo for menor do que o tempo remanescente do processo em execução, esse é interrompido e o novo processo é executado. Senão, o processo anterior continua a executar.
- Processos são colocados na fila de prontos na **posição proporcional ao seu tempo de execução remanescente** – i.e. processo com o menor tempo remanescente de CPU no início da fila, 2o. menor tempo, na 2a. posição da fila, etc. .

Shortest Remaining Time First (SRTF)

Processos	Tempo de Chegada	Rajada de CPU
P1	0	7
P2	2	4
P3	4	1
P4	5	4



$$\begin{aligned} &= [(11-2)+1+0+(7-5)]/4 = 12/4 \\ &= 3 \end{aligned}$$

$$\text{Average turn-around time} = (16 + 7 + 5 + 11)/4 = 9.75$$

Cálculo do “Waiting Time” para processo P =

Tempo de finalização(P) - Tempo de Chegada(P) - Duração rajada CPU(P)

SJF

SJF pode ser ideal para minimizar o tempo médio de espera, mas principal dificuldade é saber a extensão da próxima rajada de CPU

- Para escalonamento (a longo prazo) em um sistema batch o usuário mesmo pode indicar o limite máximo ao submeter a sua tarefa
- Mas SJF não pode ser implementado para escalonamento a curto prazo, pois não há como saber a extensão da próxima rajada
- Mas pode-se tentar estimar a sua extensão
- Espera que a rajada seguinte, τ_{n+1} , seja semelhante ao tamanho da rajada anterior t_n e da estimativa das rajadas do passado, τ_n
- Usa-se uma média exponencial dos tempos onde o valor de $\alpha > 0.5$ confere mais peso ao recente do que ao passado

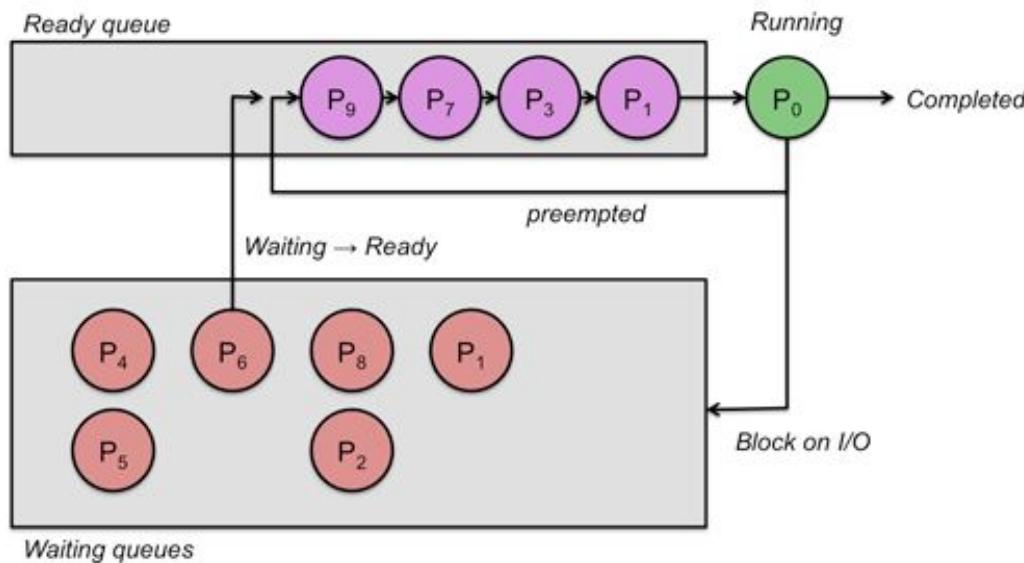
$$\tau_{n+1} = \alpha * t_n + (1-\alpha) * \tau_n$$

Escalonamento *Round Robin*

- Processos se revezam em uma fila circular
- Cada processo recebe um quantum de tempo (Δt) de CPU. Após esgotar o tempo, processo é interrompido e colocado no final da fila.
- Objetivo do escalonamento: justiça no atendimento de processos interativos

Como escolher o quantum Δt ?

- se Δt for pequeno (2 mseg): a troca de contexto pode ser uma parte considerável do quantum
- se muito grande (50 mseg): o tempo de resposta do sistema pode ser muito grande (para usuários interativos)



Escalonamento *Round Robin*

Seja a política RoundRobin com quantum de tempo de 4

Considere processos abaixo na fila com seus tempos de CPU:

- P1: 20
- P2: 12
- P3: 8
- P4: 16
- P5: 4

Obs:

Escalonamento RR tem tempo médio de espera alto, pois prioriza justiça.

$$P1 = 40$$

$$P2 = 32$$

$$P3: 8 + (28-16) = 20$$

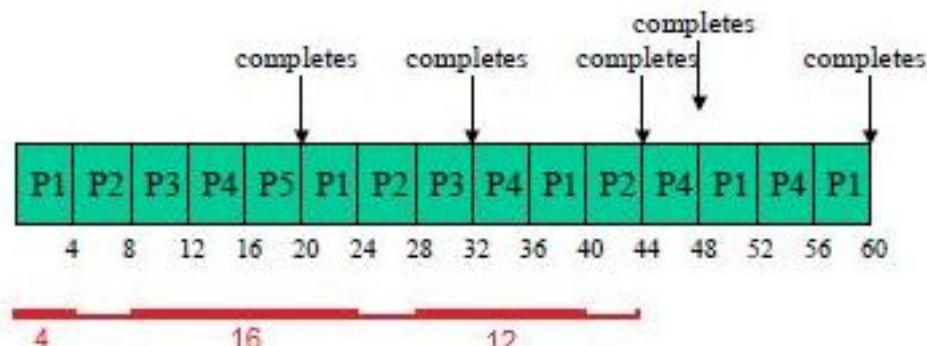
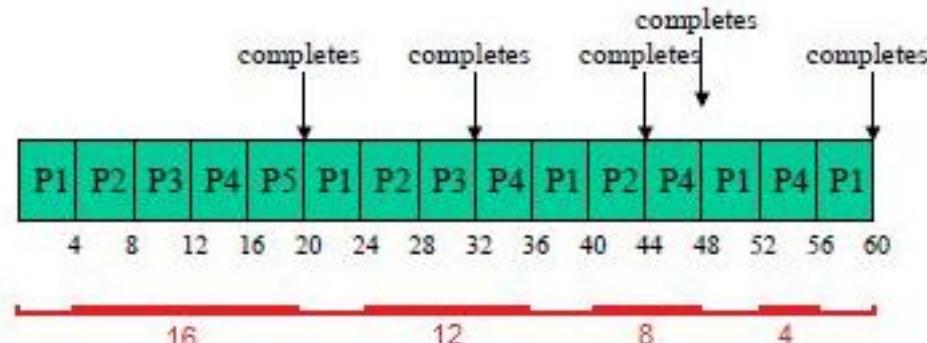
$$P4: 12 + 16 + 8 + 8 = 44$$

$$P5 = 16$$

Tempo médio de espera:

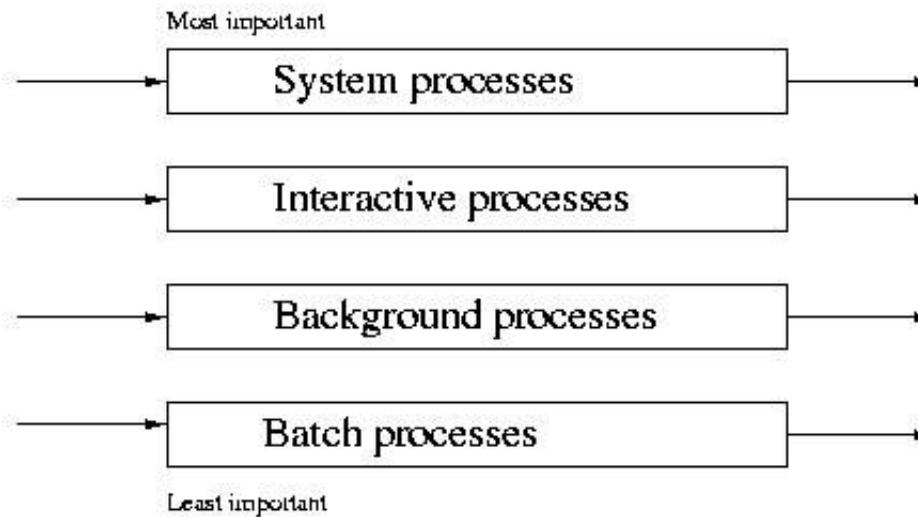
$$(40+32+20+44+16)/5 = 152/5 = 30.4$$

Diagrama de Gantt (com tempos de espera)



Escalonamento com Prioridades em múltiplos níveis (Multi-Level)

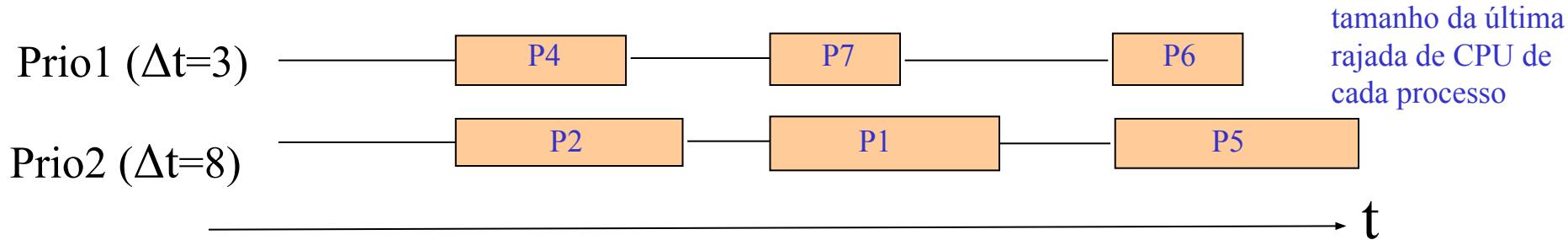
- Para sistemas com mix de processos interativos e batch
- Processos são classificados segundo seu tipo, sua importância, etc. e cada classe tem sua própria fila de prontos. (prioridade fixa)



- Para cada nível, pode se usar um escalonamento específico. Por exemplo, FCFS para maior prioridade , RR para nível 2, etc.
- Sempre executa todos os processos em um nível, antes de considerar os processos de níveis mais baixos.
- Para evitar o *problema de inanição* (= alguns processos nunca ganham a vez), pode-se definir períodos de tempo máximos para cada categoria: por exemplo, 70% para prioridade 1, 20% para prioridade 2, etc.

Escalonamento com prioridades dinâmicas

- Variante da política de múltiplos níveis
- Mas as prioridades podem ser dinâmicas, e processos podem migrar entre as filas de prontos.
- Um processo é promovido para fila mais prioritária se seu comportamento recente for de menor rajada de CPU (isto é, se ele se bloquear antes que termine o seu quantum de tempo, Δt)



Principal problema: como estimar a proxima rajada de CPU de cada processo, para posicioná-lo no nível de prioridade correto.

Algoritmos com prioridades dinâmicas

Princípio: Para cada processo, *prever a próxima rajada de processamento, a partir da última rajada de CPU e da estimativa anterior.*

Exemplo: Seja T_0 a estimativa anterior de tempo CPU necessário e T_1 o tempo de CPU recentemente gasto (última rajada).

Então, a estimativa do próximo quantum, T_2 , deve ser ajustada por média ponderada:

$$T_2 = \alpha * T_1 + (1-\alpha) * T_0.$$

Se $\alpha > 0.5$ dá-se mais importância para o comportamento mais recente, e $\alpha < 0.5$ maior importância para o comportamento mais no passado

Algorítmos de escalonamento

Múltiplos níveis com feedback - MLF (*Multi-level with feedback*):

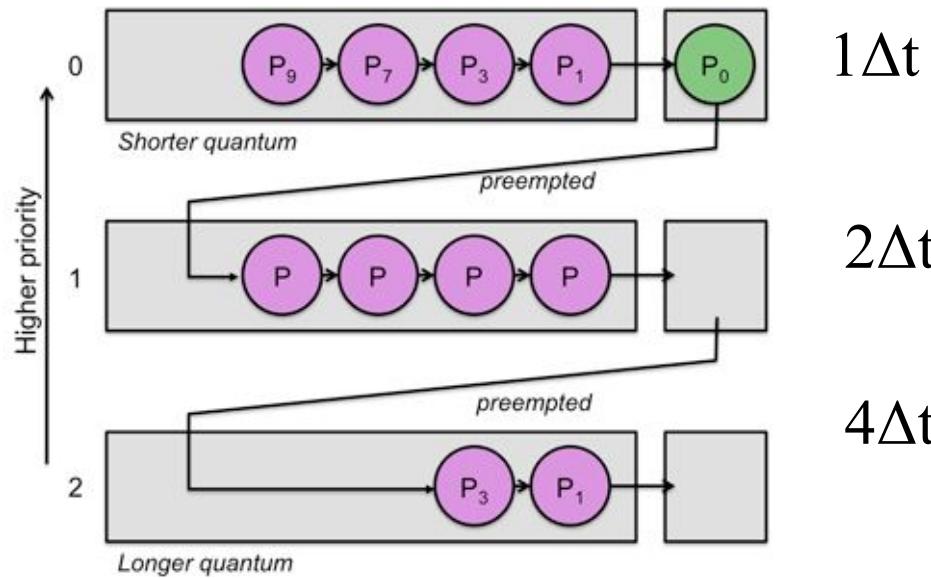
- Similar ao ML, mas com uma prioridade que varia dinamicamente
- Todo processo começa no nível de prioridade mais alto n
- Cada nível de prioridade P prescreve um tempo máximo (fatia de tempo) t_P
- t_P aumenta à medida que P diminui
- geralmente:

$$t_n = \Delta t \quad (\text{constante})$$

$$t_{P-1} = 2 \times t_P$$

Filas em múltiplos níveis com feedback

Idéia: dar maior prioridade para processos que precisam de fatia (ou quantum) de tempo (Δt) menor. Se um processo usa a CPU durante todo seu quantum Δt , cai para o nível de prioridade abaixo. Quando não utiliza todo seu Δt , se mostra mais I/O-bound e portanto é promovido a uma fila mais prioritária.



- Processos que permanecem muito tempo “presos” em suas filas sem acesso a CPU, são promovidos “artificialmente” para níveis mais prioritários (mecanismo de envelhecimento/aging)
- Isso evita starvation (inanição).

Filas em múltiplos níveis com feedback

A fila de prontos é dividida em várias filas menores com base nas rajadas esperadas de uso da CPU. Os processos não são estacionários em uma fila, mas podem migrar entre as filas.

Preempção: se aparecer um processo de uma fila mais prioritária e enquanto um processo menos prioritário está executando na CPU, esse é interrompido

Este escalonamento é mais flexível e genérico, mas envolve vários parâmetros/decisões, e por isso é complexo de se configurar.

- número de filas
- qual é escalonamento dentro de cada uma das filas
- método para determinar a movimentação de um processo para a fila mais prioritária e para a menos prioritária
- método para escolher em qual fila é posicionado um novo processo que acabou de chegar

Como é o escalonamento no Linux?

- O escalonador do kernel 2.4 divide o tempo em épocas e, varre toda a lista para escolher o melhor processo
- em cada época, cada processo tem permissão para executar até sua fatia de tempo (calculada no início da época)
- Se um processo não usar toda a sua fatia de tempo, metade da fatia de tempo restante é adicionada à nova fatia de tempo para permitir que ela seja executada por mais tempo na próxima época.
-
- Quando um processo esgota seu quantum de tempo, ele é preemptado e substituído por outro processo executável.
- Um processo pode ser selecionado várias vezes na mesma época, desde que o seu quantum não tenha sido esgotado:
-
-
- A época termina quando todos os processos executáveis tiverem esgotado seus quanta; nesse caso, o escalonador recomputa as durações de quantum de tempo de todos os processos para a nova época começa.
-

Como é o escalonamento no Linux?

Linux usa uma variante de fila multinível com feedback;

- O processo é preemptado quando:
 - outro processo previamente suspenso com maior prioridade é reativado;
 - o processo abandona voluntariamente a CPU fazendo uma chamada de sistema bloqueante, ou chamando a syscall `sched_yield()`;
 - o processo diminui/aumenta voluntariamente a sua prioridade chamando `setpriority()`
 - O processo em tempo real está pronto para ser executado.

Cada processo tem um quantum de tempo de base ,

- Os usuários podem alterar o quantum de tempo de base dos seus processos utilizando as chamadas de sistema `nice()` e `setpriority()`

Algoritmos de escalonamento

Função de prioridade do MLF:

Definir a prioridade $P = n - i$ para um a :

prioridade / tempo de serviço alcançado

$$n \quad a < \Delta T$$

$$n-1 \quad a < \Delta T + 2T$$

$$n-2 \quad a < \Delta T + 2T + 4T$$

...

...

$$n-i \quad a < (2^{i+1} - 1)T$$

- Ache o menor i tal que $a < (2^{i+1} - 1)T$:

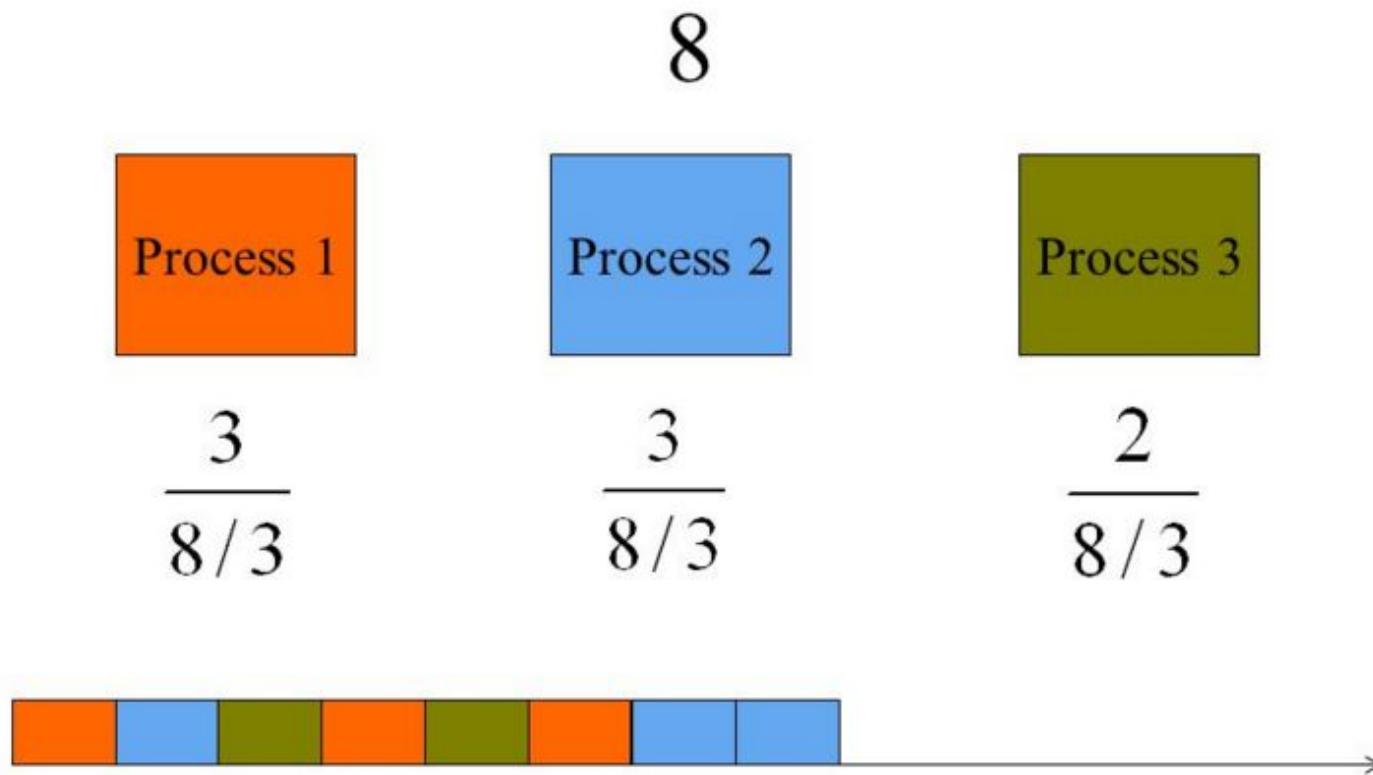
$$i = \lfloor \lg_2(a/T+1) \rfloor$$

- Defina $P = n - i = n - \lfloor \lg_2(a/T+1) \rfloor$

Outras políticas de escalonamento

Escalonamento garantido

- A cada processo é atribuída uma cota de quantums, que devem ser utilizadas em cada ciclo de cotas (p.ex. 8)
- Muito simples, só é feito para processos do usuário (e não de sistema)



Outras políticas de escalonamento

Rate-monotonic scheduling (RMS) é um algoritmo de atribuição estática de prioridades a processos periódicos para sistemas operacionais de tempo real (RTOS).

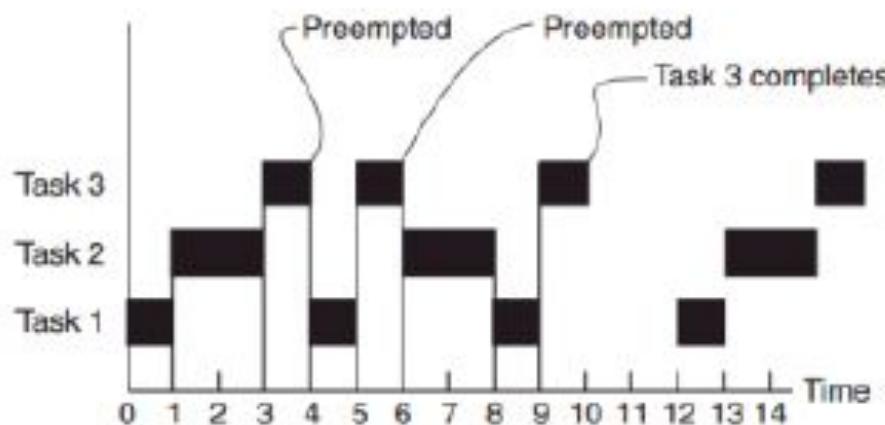
As prioridades estáticas são atribuídas de acordo com a duração do ciclo dos processos, sendo que aqueles com ciclos menores ganham maior prioridade.

Esses sistemas operacionais têm políticas preemptivas e precisam dar garantias determinísticas sobre tempos de resposta. A análise rate-monotonic é usada para determinar exatamente quais processos podem ser escalonados juntos.

Outras políticas de escalonamento

Exemplo de Rate-monotonic scheduling (RMS)

Task	Execution Time	Period	Priority
T1	1	4	High
T2	2	6	Medium
T3	3	12	Low

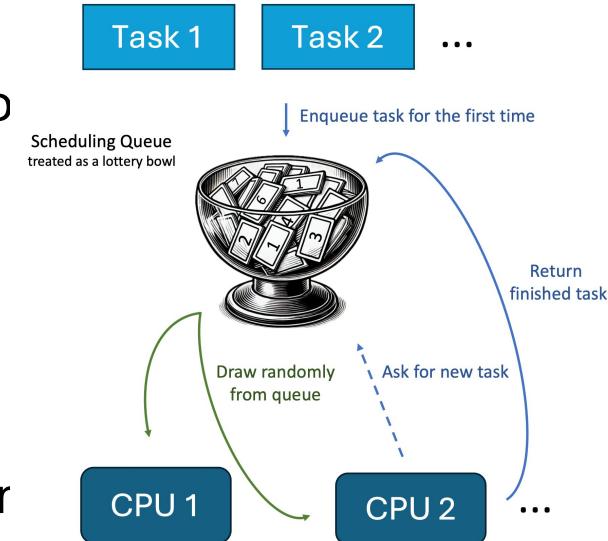


Escalonamento por sorteio (Lottery Scheduling)

Idéia:

atribuir certo conjunto de bilhetes de loteria a cada processo. Processos mais prioritários recebem um número maior de bilhetes

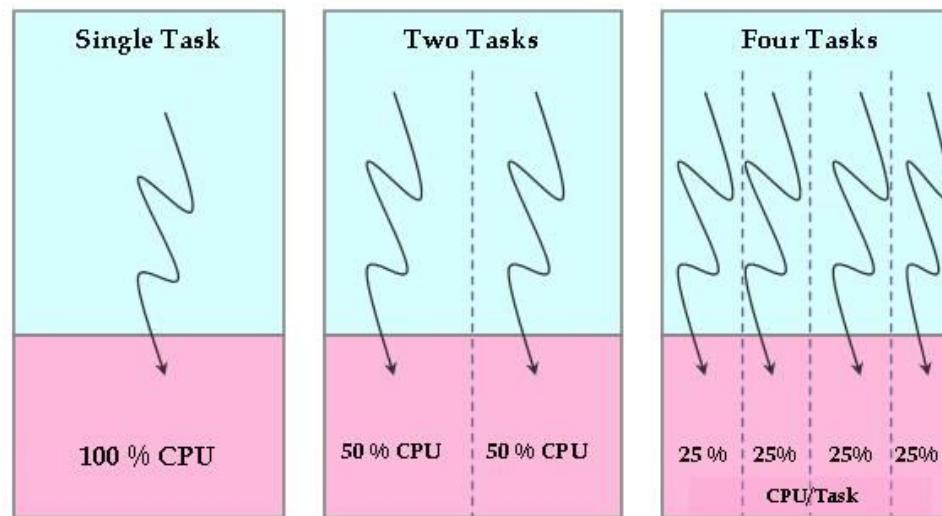
- O escalonador sorteia um bilhete de loteria, e o processo com o bilhete “sorteado” pode executar no próximo quantum de tempo
- Cada processo irá executar em uma frequência proporcional ao número de bilhetes que possui
- Vantagens:
 - simplicidade e distribuição dos tickets reflete bem as prioridades mútuas
 - Processo pode transferir tickets para processos filho, ou usuário deve distribuir os seus tickets entre seus processos



Completely Fair Scheduler (CFS)

Idéia:

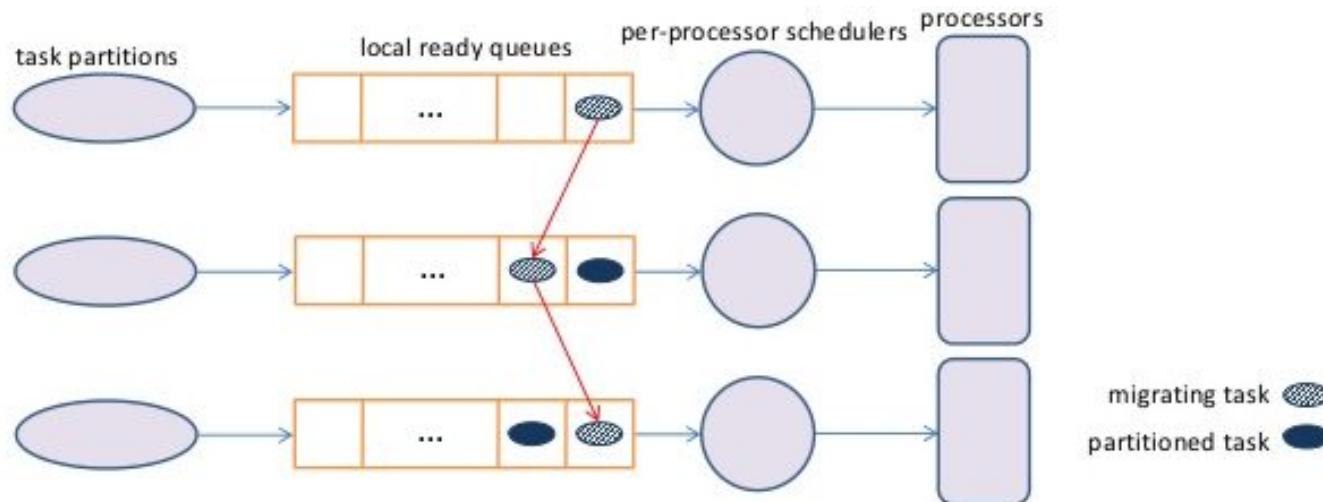
- O período de agendamento é a unidade de tempo de uma CPU multitarefa ideal e,
- dentro desse período, cada tarefa/processo executável deve ter algum tempo para ser executada na CPU.



Ideal Precise Multi-tasking CPU – Each task runs in parallel and consumes equal CPU share

Outras políticas de escalonamento

Escalonamento em arquiteturas multi-processador



Fonte: S. Alshar (Resource Sharing in Uni/Multiprocessor Embedded Systems)

- Processos são previamente alocados a um dos processadores, e entram na fila de prontos do processador específico.
- Alguns processos também precisam executar em um determinado processador
- Outros podem executar em qualquer processador e estão livres para migrar de um para outro durante a execução (migração)
- Processadores podem puxar ou empurrar processos entre si a depender do desbalanço de carga.

Escalonamento Rate Monotonic (RM)

Usado para escalarar um conjunto de processos P_i cíclicos (que sempre precisam ser executados a cada t_i unidades de tempo).

Para sistemas de tempo real

- Escalonamento preemptivo
- Atribui uma prioridade (estática) maior para o processo com menor ciclo t_i
- a prioridade é relativa entre os processos

Assuma 2 processos

P_1 com $(t_{1,c}, c_1) = (50, 20)$ e

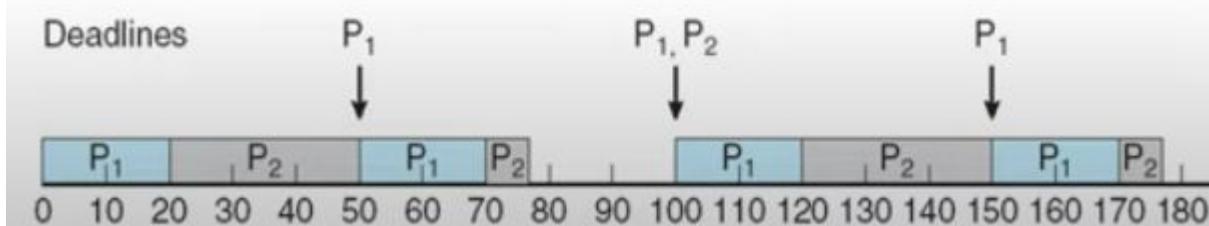
P_2 com $(t_{2,c}, c_2) = (100, 35)$

Então, P_1 recebe uma prioridade maior do que P_2

Utilizacão de CPU

$U_1 = 20/50 = 0.4$ e $U_2 = 35/100 = 0.35$. Então $U_1 + U_2 = 0.75 < 1$

Assuma que P_1 e P_2 chegam ao sistema no mesmo momento



Escalonamento Rate Monotonic (RM)

Escalonamento RM é ótimo:

se com RM não for possível escalar um conjunto de processos sem ferir prazos, então não existe nenhum outro escalonamento de prioridade estática que funcione.

Um conjunto de processos consegue ser escalonado com RM se eles satisfazem a seguinte equação

RM impõe um limite sobre a utilização total de CPU: $N * 2^{1/n} - 1$

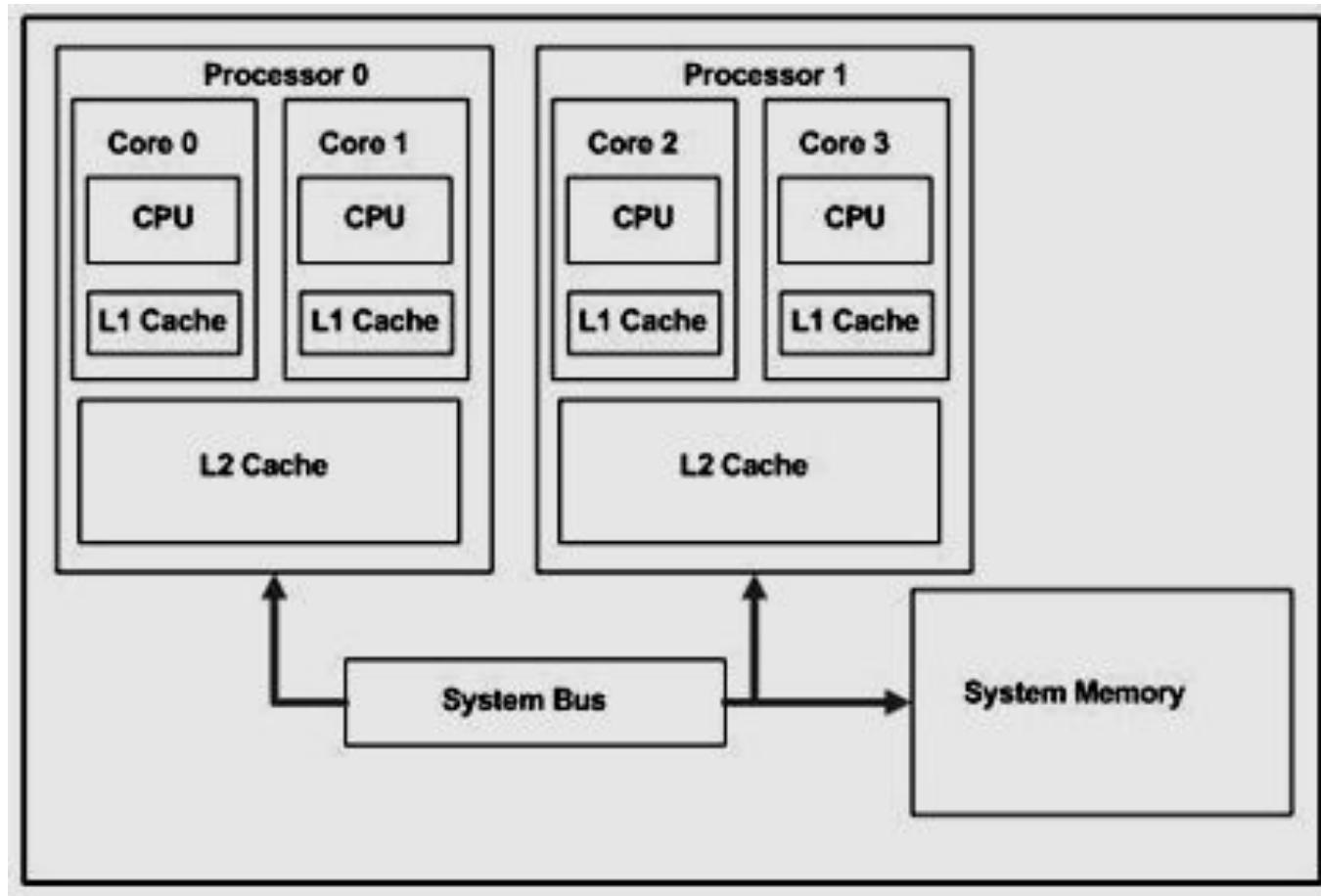
Isso limita aprox, $U_i = 68\%$ para cada processo

Significa que o conjunto de processos está deixando uma “margem de manobra muito pequena” nos ciclos dos processamentos conjuntos

Comparação

- Sistemas de processamento em lote
 - FIFO, SJF, SRT:
 - FIFO é o mais simples
 - SJF/SRT possuem tempos médios de *turnaround* (# processos/tempo) menores
- Sistemas interativos
 - RR puro ou MLF (c/ RR por prioridade) são apropriados
 - A escolha do quantum de tempo q determina o overhead
 - Quando $q \rightarrow \infty$, RR se aproxima de FIFO
 - Quando $q \rightarrow 0$, overhead de troca de contexto $\rightarrow 100\%$
 - Quando $q >>$ overhead de troca de contexto, e n processos executam, então o desempenho é $\approx 1/n$ CPU velocidade

Sistemas Multi-processador



Escalonamento em Sistemas Multi-processador

- Mais complexo, vários processadores devem ser mantidos ocupados
- Multiprocessamento assimétrico = um processador é o mestre (executa código do núcleo) e os demais executam em modo usuário
- Multiprocessamento simétrico (SMP) = cada processador tem uma fila de prontos própria e escalona processos dessa fila
- *Afinidade ao processador*: sistemas com SMP tentam manter os processos/threads sempre no mesmo processador, para evitar que o Memory Cache precise ser invalidado
- Maioria dos SOs (Linux, Mac OSX, Windows,) adotam SMP

Escalonamento em Sistemas Multi-processador

- Balanceamento de carga é um objetivo importante em um sistema multi-processador
- Envolve a migração de processos entre processadores
- *Migração push*: periodicamente, transfere-se processos de processadores mais ocupados para processadores menos ocupados
- *Migração pull*: processadores desocupados “roubam” processos da fila de prontos de processadores ocupados

Obs: a migração de processos conflita com a afinidade ao processador (o custo com a invalidação de caches pode superar os benefícios do balanceamento de carga).

Por isso, migração só é realizada se o desbalanço passar de um limiar

Shortest Job First - SJF

Princípio básico usado em sistemas de lotes:

Idéia: o job mais curto tem prioridade

- Usado quando o tempo de execução de cada tarefa é conhecido de antemão
- Objetivo: maximizar a taxa de saída/finalização (*throughput*)
- SJF sempre produz a menor média de tempo de saída!

8	4	4	4
A	B	C	D

(a)

4	4	4	8
B	C	D	A

(b)

Tempos de finalização:

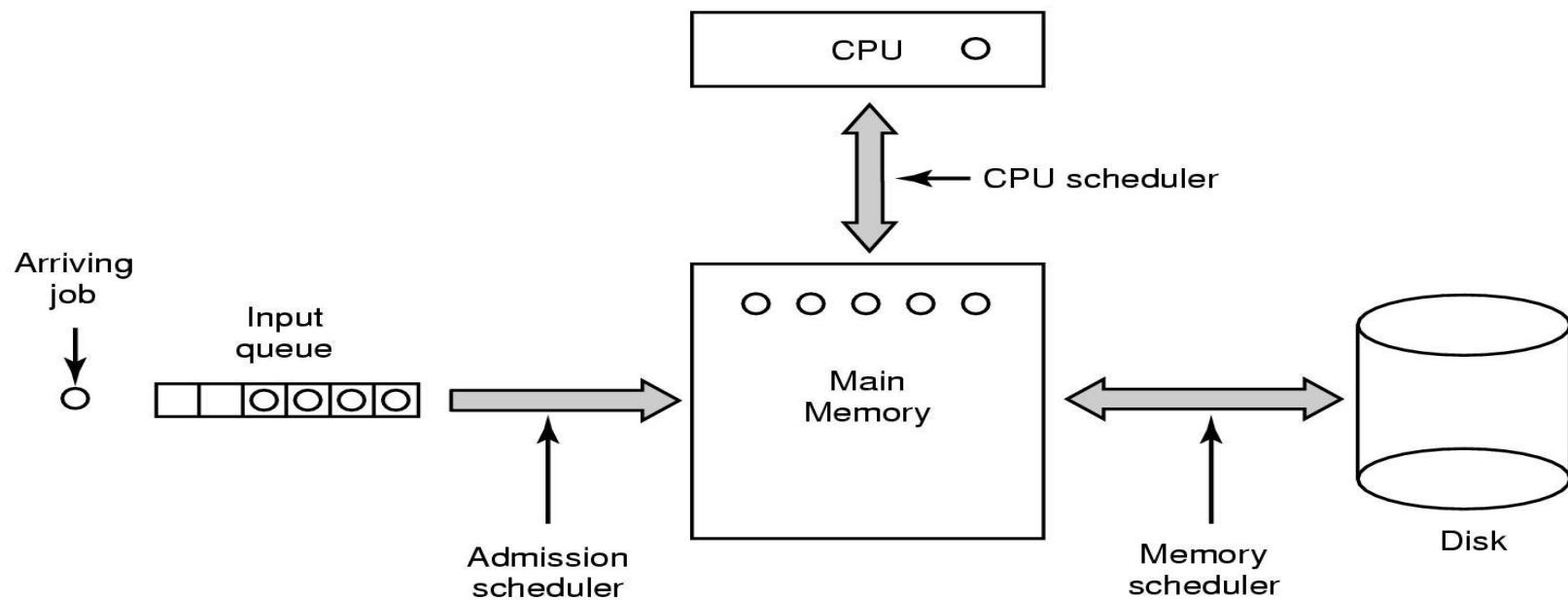
$$(a) \quad (8+12+16+20)/4 = 14$$

$$(b) \quad (4+8+12+20)/4 = 11.$$

Escalonamento em três níveis

Sistemas em Lote geralmente possuem escalonadores em 3 níveis:

- E. de admissão scheduler: faz um mix adequado entre jobs mais curtos e mais demorados (e leva em conta prioridade dos usuários)
- E. de memória: maior prioridade para jobs menores
- E. de CPU: poderia adotar a SJF



Escalonamento por Prioridade

Prioridade pode ser um mix de:

- Prioridade do usuário dono do job
- Prioridade de acordo com tarefa executada pelo job (tarefas de sistema são mais prioritárias do que de usuário)
- Prioridade, de acordo com os recursos requisitados/usados pelo job

Prioridade de job pode ser estática ou dinâmica.

Quando dinâmica:

- Pode diminuir a medida que o job vai usando a CPU, para dar chance a jobs novos
- Pode diminuir se job usa todo o seu quantum de tempo (CPU-bound) e aumentar se job requisita E/S antes de quantum terminar (CPU-bound)

Escalonamento pode usar uma única fila ou várias filas.

Por prioridade

- Nesse caso, SJF pode ser um fator de desempate

<u>Job</u>	<u>Tempo CPU</u>	<u>Prioridade</u>
A	8	2
B	1	1
C	1	3

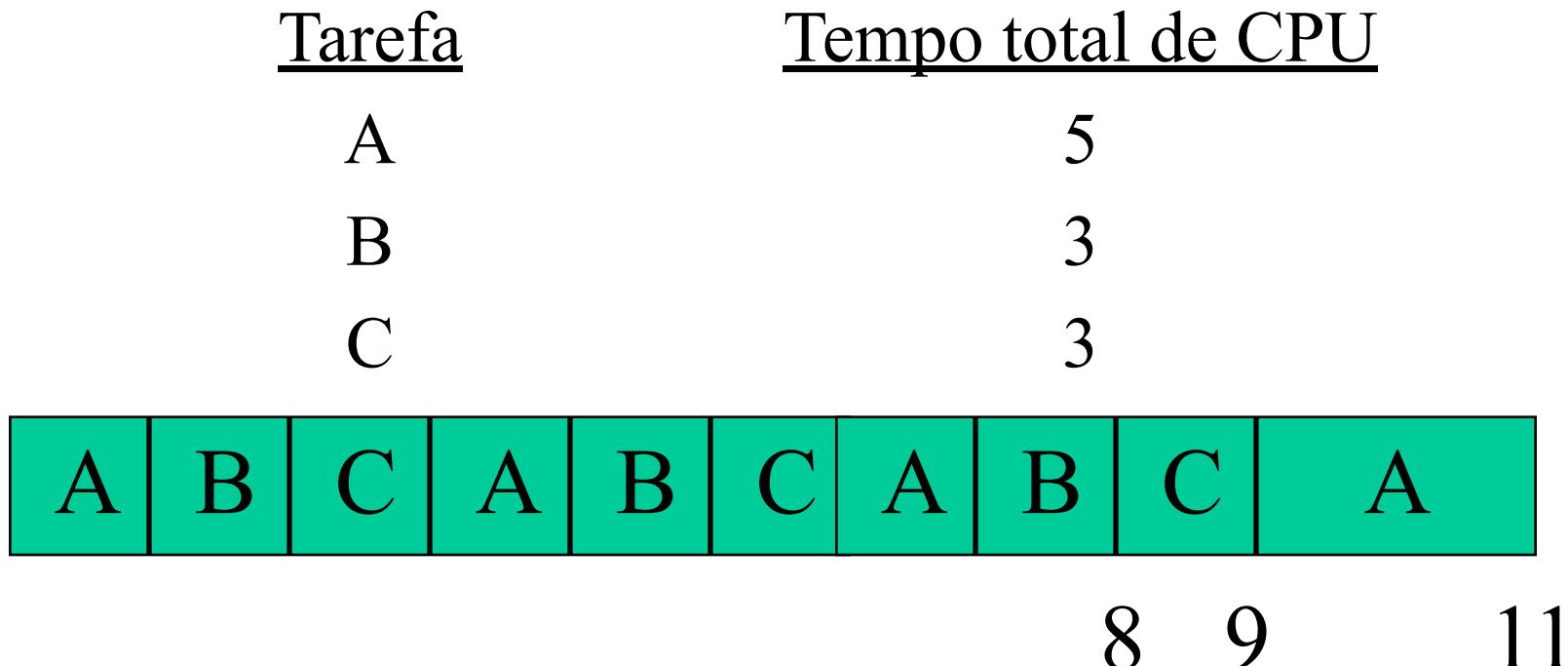


0 1 9 10

Tempo médio de espera $(0 + 1 + 9) / 3 = 3.3$

Round Robin

- Escalonamento preemptivo
- Cada processo recebe um quantum de tempo de CPU. Após esgotar o tempo, é colocado no final da fila.
- Objetivo: justiça no atendimento

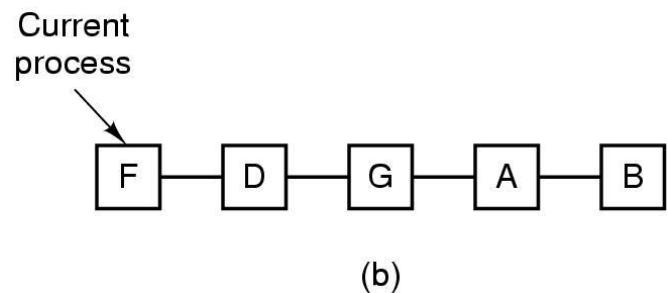
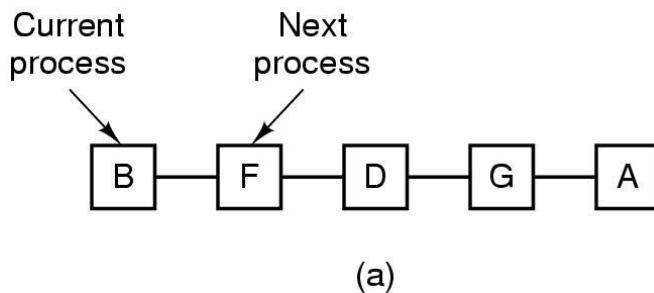


$$\text{Tempo médio de saída} = (8 + 9 + 11) / 3 = 9.3$$

Scheduling in Interactive Systems

Round Robin Scheduling

- Basic technique & simple algorithm, for time-sharing
- Each process gets a time interval (quantum) for the CPU
- Process is preempted if it runs until the end of its quantum
- Adequate for scheduling processes with same priority
- Main issue: how large should be the quantum?
 - Too small (20 msec): context-switch may be considerable portion of quantum
 - Too large (500 msec): system response time may be too large (for interactive users)



- (a) list of runnable processes while B is executing
(b) list of runnable processes after B uses up its quantum

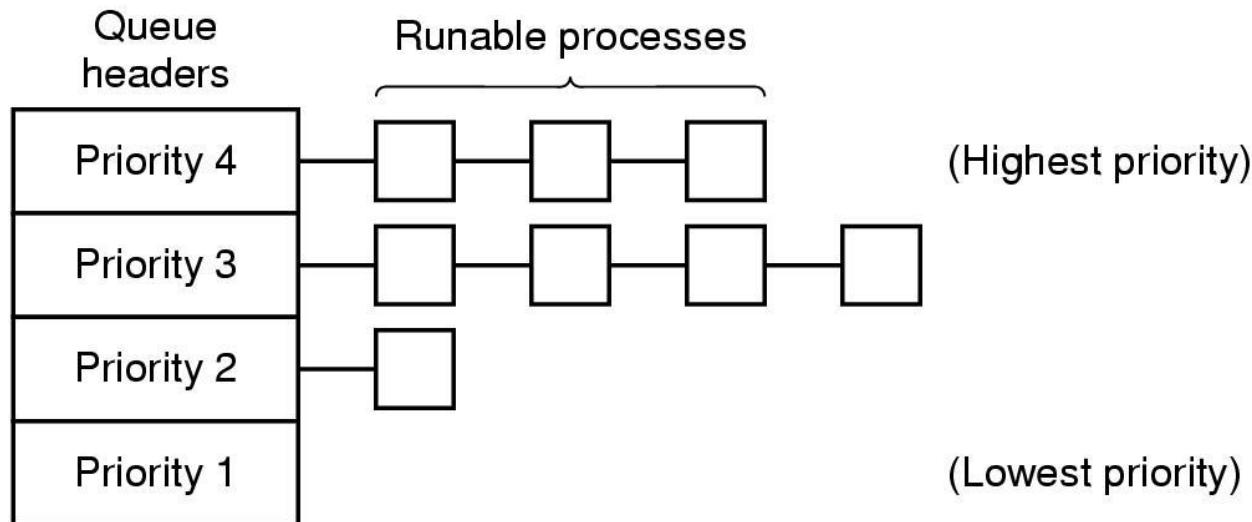
Scheduling in Interactive Systems

Usually, processes are grouped into priority classes and using Round-Robin within each class:

Algorithm: Consider highest priority class with some process: then schedule them in RR. But if queues N (and larger) are empty, then select processes in queue N-1.

Example: Minix

In addition, each priority class may have a specific quantum (High priority – small quantum, low priority - larger quantum)



Outras políticas de escalonamento

Escalonamento garantido

- cada um dos n usuários recebe aproximadamente $1/n$ dos ciclos de CPU
- Muito simples, e só é feito para processos do usuário (e não de sistema)

Escalonamento por sorteio (*lottery scheduling*)

- Sorteio *quase* aleatório de processos (todos ou em cada nível de prioridade)
- Vantagem: simplicidade e distribuição uniforme de valores sorteados geralmente garante igualdade de chances
- Para garantir justiça, impõe-se um limite no número de vezes que um processo pode ser sorteado em determinado período

Scheduling in Real-Time Systems

Schedulable real-time system

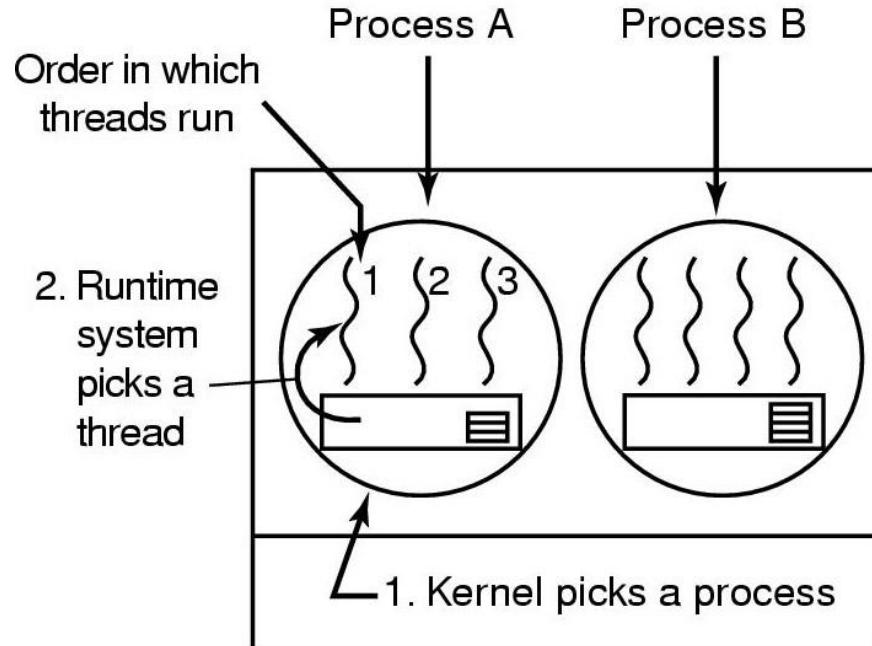
- Given
 - m periodic events
 - event i occurs within period P_i and requires C_i seconds
- Then the load can only be handled if

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

Policy versus Mechanism

- Separate what is allowed to be done with how it is done
 - a process knows which of its children threads are important and need priority
- Scheduling algorithm parameterized
 - mechanism in the kernel
- Parameters filled in by user processes
 - policy set by user process

Thread Scheduling (1)



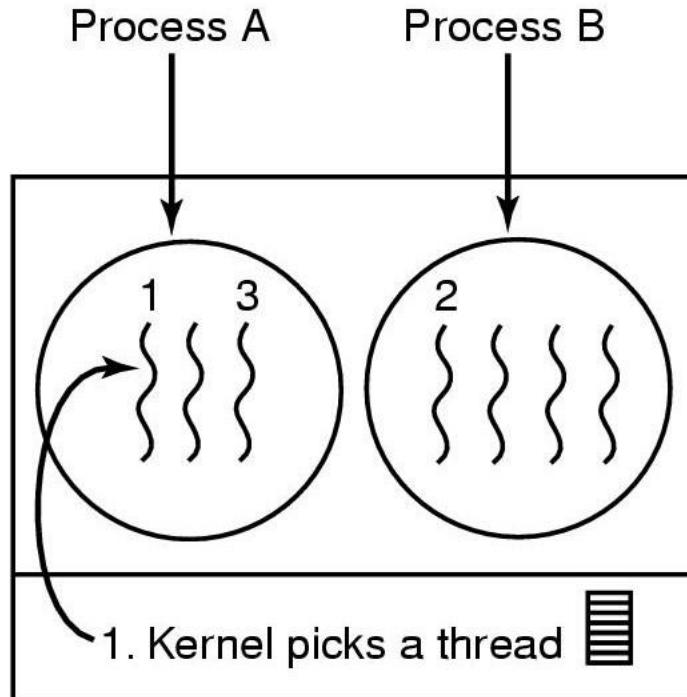
Possible: A1, A2, A3, A1, A2, A3

Not possible: A1, B1, A2, B2, A3, B3

Possible scheduling of user-level threads

- 50-msec process quantum
- threads run 5 msec/CPU burst

Thread Scheduling (2)



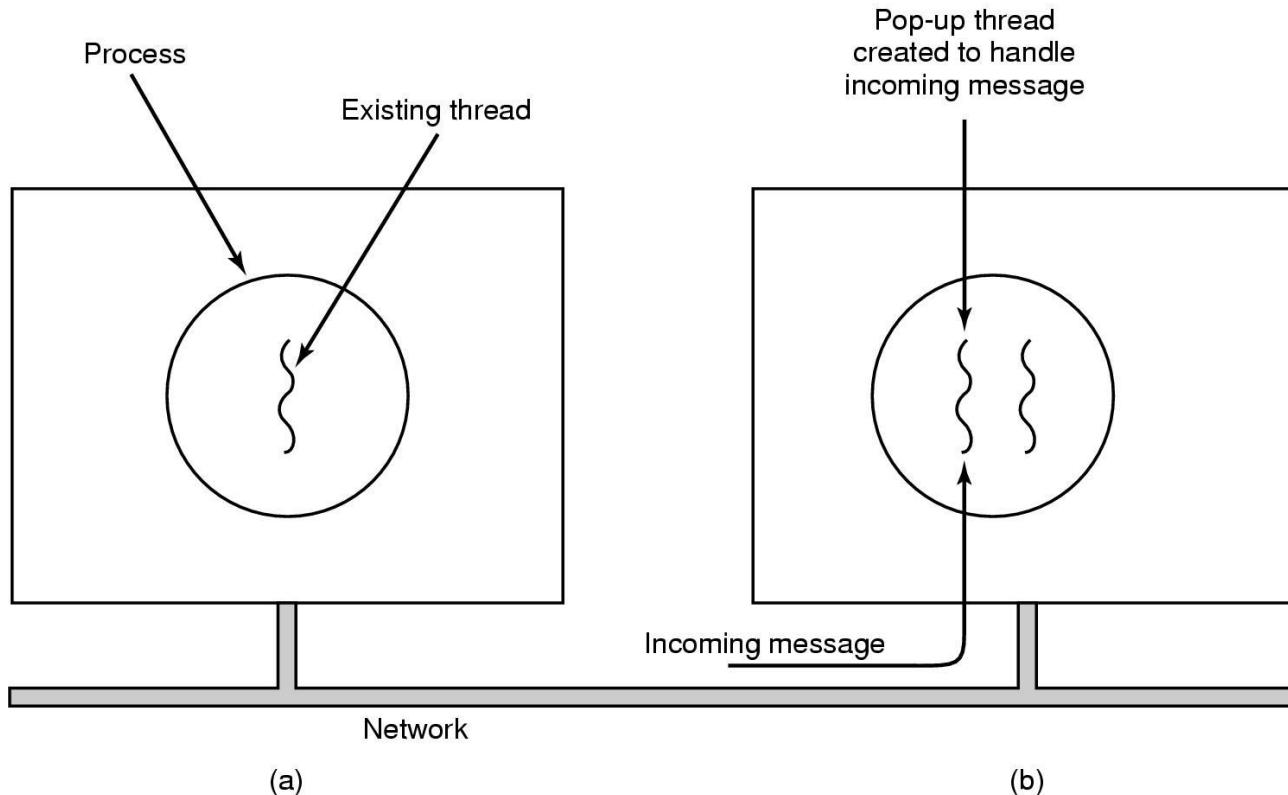
Possible: A1, A2, A3, A1, A2, A3

Also possible: A1, B1, A2, B2, A3, B3

Possible scheduling of kernel-level threads

- 50-msec process quantum
- threads run 5 msec/CPU burst

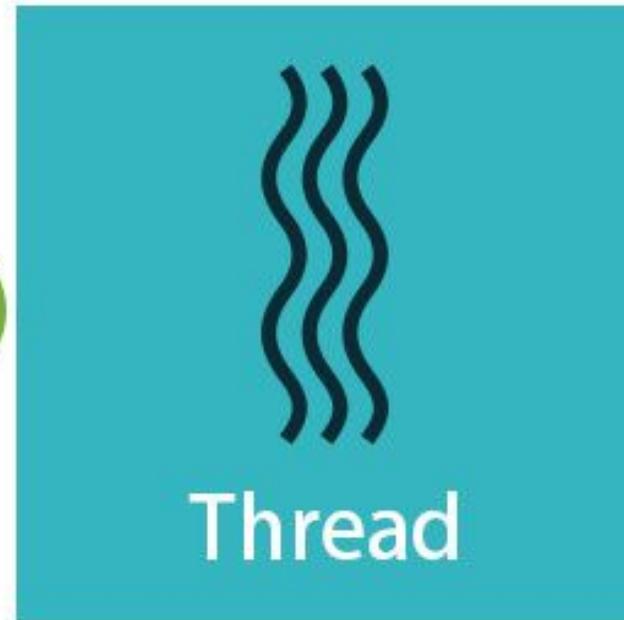
Pop-Up Threads



- Creation of a new thread when message arrives
 - (a) before message arrives
 - (b) after message arrives



vs



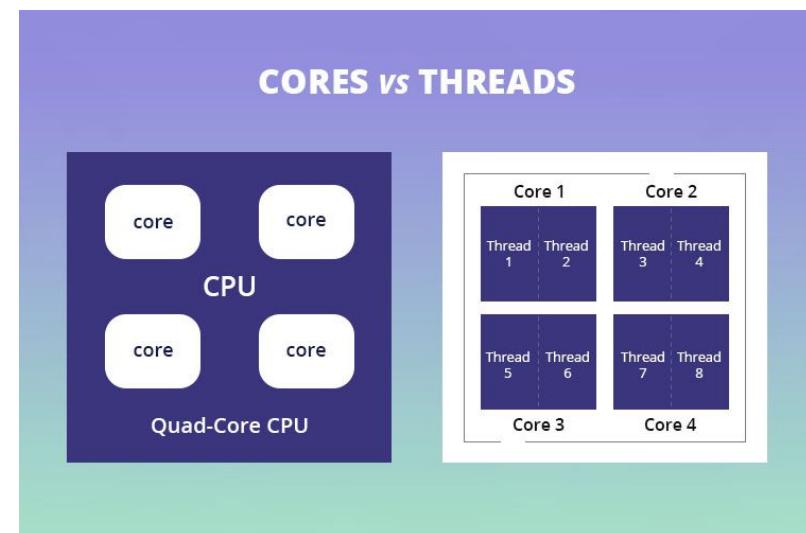
Limitações do Modelo de Processos

- Várias aplicações precisam executar tarefas inherentemente concorrentes, compartilhando estruturas de dados.
- Ex: servidores, monitores de transações de bancos de dados, clientes de Email, serviços de rede, etc.
- Poderia-se criar processos filhos para as tarefas concorrentes, mas
 - processos requerem a alocação de muitos recursos pelo Sistema Operacional e
 - ocupam espaços de memória distintos tornando a comunicação/sincronização custosa.
- Arquiteturas multi-core permitem a execução paralela com compartilhamento de memória.
- Então, por que não permitir processos com várias linhas internas de execução concorrentes?

Threads

Threads são “processos leves”

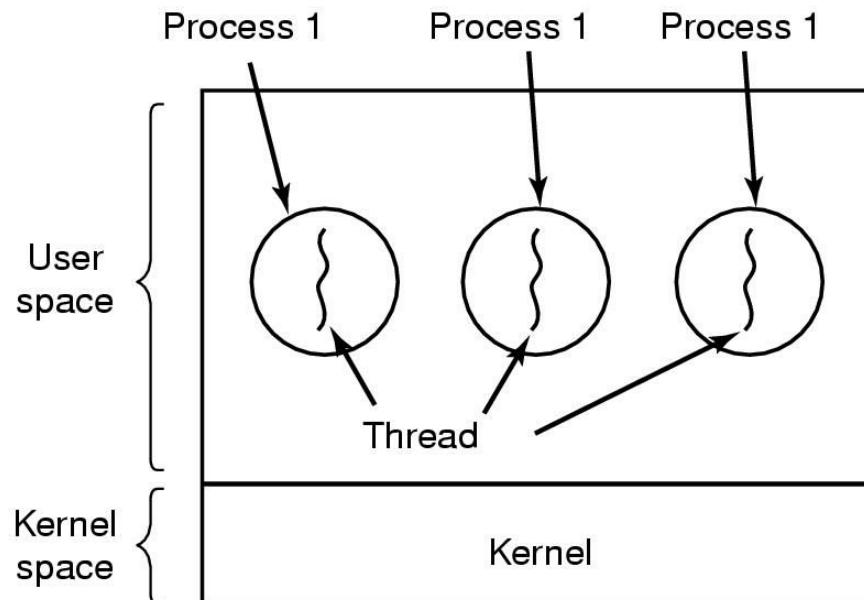
- Uma linha de execução independente
- Vinculada a um processo,
- Possui sua própria pilha de execução
- Pode coexistir com outras threads do mesmo processo
- Compartilha dados globais com membros do processo
- Também possuem vários estados: em execução; prontos, bloqueados, finalizado, etc.
- Se arquitetura for multi-core, podem executar em paralelo.



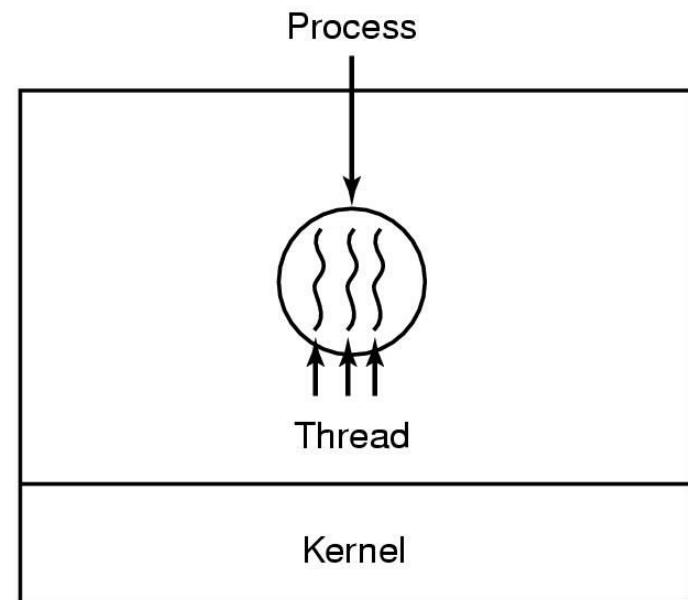
Threads

Thread = linha de execução independente (e concorrente) dentro de mesmo processo

- Múltiplas threads são necessárias quando:
 - mais de uma tarefa deve ser executada concorrentemente, e
 - é necessário compartilhar variáveis/dados do processo (e.g. uma cache em um servidor de arquivos, conexões em um servidor Web; buffers internos, etc.)



(a)



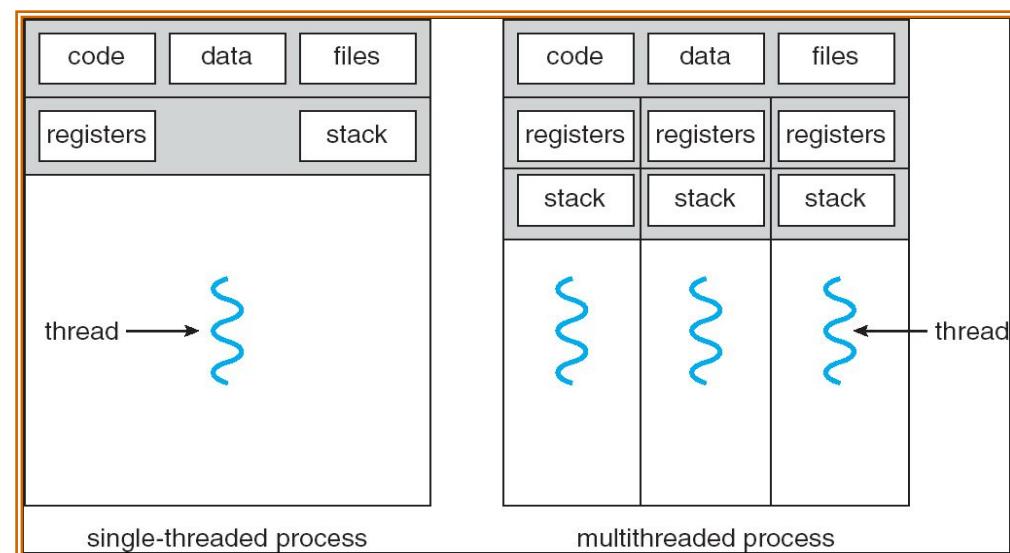
(b)

Thread

Thread: uma unidade básica de utilização da CPU (fluxo de controle dentro do processo)

Composta por um:

- ID
- contador de programa
- conjunto de registradores
- Uma pilha
- Compartilha recursos e espaço de endereçamento com outros threads do mesmo processo, assim como descritores de arquivos, temporizadores, tratadores de sinais
-



Processos com uma ou mais threads

Exemplo: Editor de texto

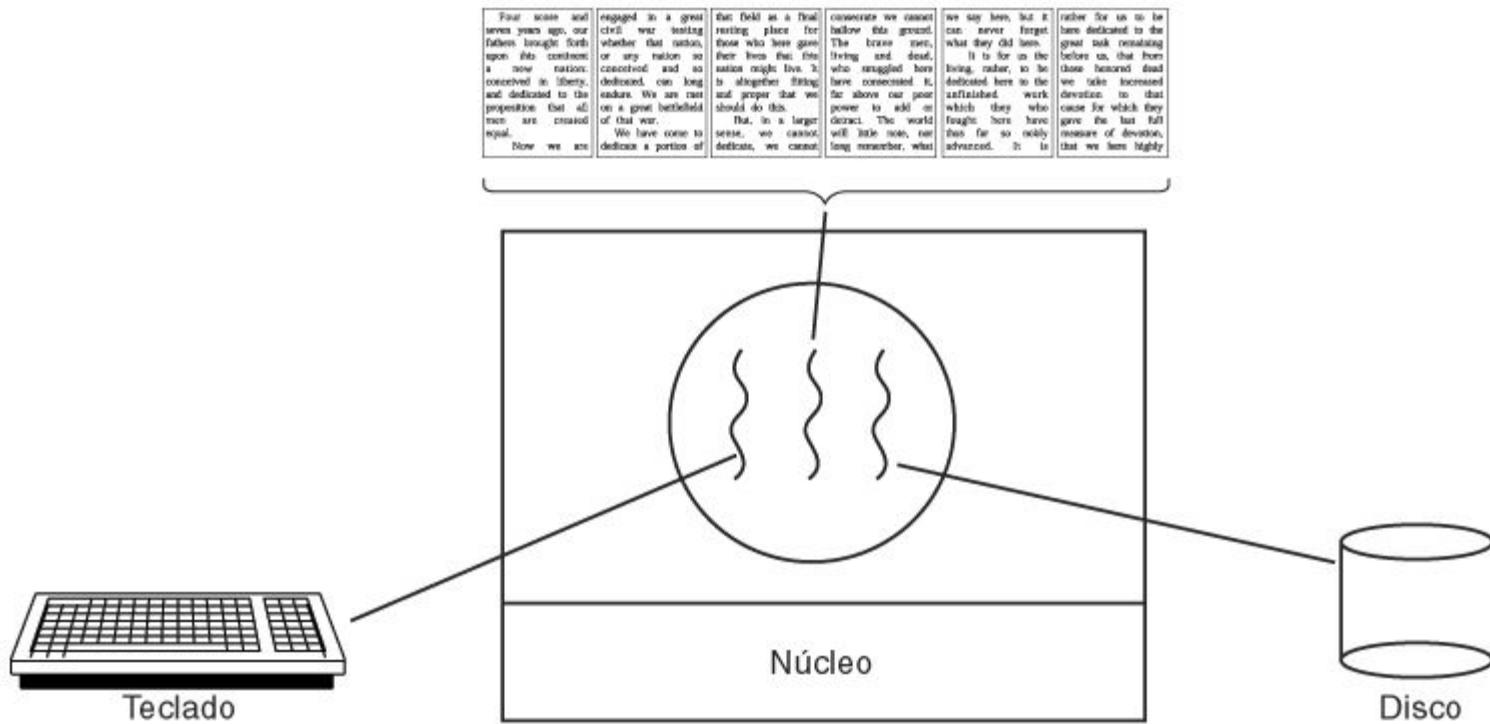
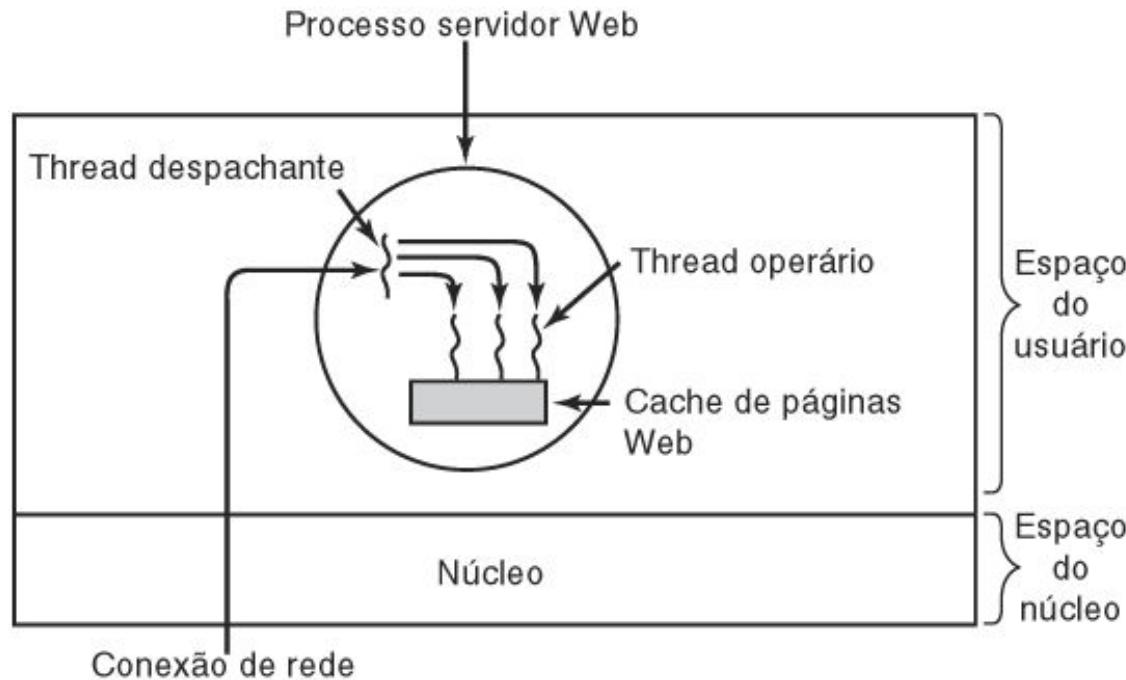


Fig.: Um Editor de texto com três threads (corretor ortográfico online, entrada do teclado, cópia de segurança)

Exemplo:

Um servidor web com múltiplas threads: clientes não precisam esperar o resultado da requisição



```
while (TRUE) {  
    get_next_request(&buf);  
    handoff_work(&buf);  
}
```

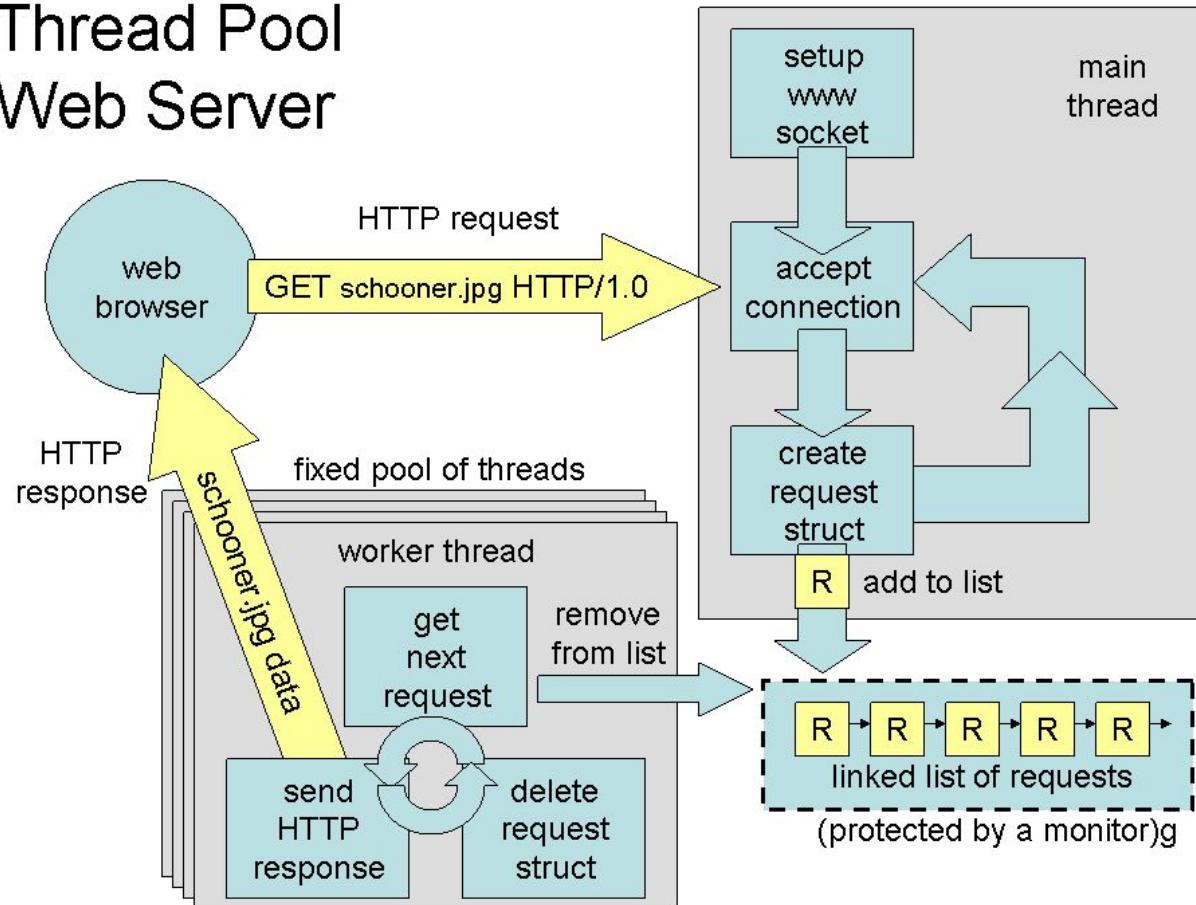
(a)

```
while (TRUE) {  
    wait_for_work(&buf)  
    look_for_page_in_cache(&buf, &page);  
    if (page_not_in_cache(&page))  
        read_page_from_disk(&buf, &page);  
    return_page(&page);  
}
```

(b)

Exemplo: Servidor com Pool de threads

Thread Pool Web Server



Cada thread (do pool de threads) executa um procedimento que pega uma nova requisição R, da lista processa-a e gera uma resposta

Threads: Principais Características

- Na criação da thread, passa-se um ponteiro de um procedimento a ser executado;

```
int pthread_create(pthread_t *thread,
pthread_attr_t *attr, void *(*start_routine)
(void *arg), void *arg);
```

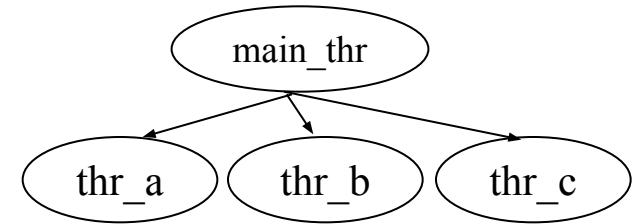
- Se duas threads executam o mesmo procedimento, cada uma terá a sua própria cópia das variáveis locais;
- As threads podem acessar todos os dados globais do programa e o heap (memória alocada dinamicamente) do processo em que executam.
- No acesso a dados globais, não há qualquer mecanismo automático de sincronização. O programador da aplicação precisa cuidar disso (usando mutexes/semáforos)
-

Criação de Threads: Exemplo

```
#include <stdio.h>
#include <pthread.h>
/* Function prototypes for thread routines */
void *sub_a(void *);
void *sub_b(void *);
void *sub_c(void *);
thread_t thr_a, thr_b, thr_c;

void main() {
    thread_t main_thr;
    main_thr = thr_self();
    printf("Main thread = %d\n", main_thr);
    if (thr_create(NULL, 0, sub_b, NULL, THR_SUSPENDED|THR_NEW_LWP, &thr_b))
        fprintf(stderr,"Can't create thr_b\n"), exit(1);
    if (thr_create(NULL, 0, sub_a, (void *)thr_b, THR_NEW_LWP, &thr_a))
        fprintf(stderr,"Can't create thr_a\n"), exit(1);
    if (thr_create(NULL, 0, sub_c, (void *)main_thr, THR_NEW_LWP, &thr_c))
        fprintf(stderr,"Can't create thr_c\n"), exit(1);

    printf("Main Created threads A:%d B:%d C:%d", thr_a, thr_b, thr_c);
    printf("Main Thread exiting...\n");
    thr_exit((void *)main_thr);
}
```



Diferenças Processos vs Threads

Difference	Process	Thread
Resource Allocation	Allocate new resources each time we run a program.	Share resources of process.
Resource Sharing	In general, resources are not shared. The code may be shared for the same program.	Share code, heap, data area except stack.
Address	Have a separate address space	Share address space
Communication	Communicate through IPC.	Communicate freely with modifying shared variables.
Context Switching	Generally slower than thread.	Generally faster than process.

Usando POSIX threads (pthread)

Pthread é um padrão de fato para threads em Unix

- #include pthread.h
- Compile seu programa com -lpthread
gcc -o test test.c -lpthread
- Recomenda-se verificar valores de retorno das principais chamadas

POSIX Threads

`pthread_create` – Cria uma thread

`pthread_exit` – Conclui uma thread

`pthread_join` – Espera que uma thread termine

`pthread_yield` – Libera a CPU para executar outra thread

`pthread_attr_init` – Inicializa atributos do thread

`pthread_attr_destroy` – Destrói atributos da thread

Atributos de threads:

- Normalmente, os atributos default são suficientes
- Só podem ser modificados na inicialização da thread
- Exemplos: Tamanho da pilha, Endereço da pilha, Set/Get Scheduling Policy, Preservação da thread depois de seu término, Prioridade

POSIX Threads

- `pthread_create()` – Cria uma thread
- `pthread_exit()` – Conclui uma thread
- `pthread_join()` – Espera que uma thread termine
- `pthread_yield()` – Libera a CPU para executar outra thread
- `pthread_attr_init()` – Inicializa atributos do thread
- `pthread_attr_destroy()` – Destroi atributos da thread

Atributos de threads:

- Normalmente, os atributos default são suficientes
- Só podem ser modificados na inicialização da thread
- Exemplos: Tamanho da pilha, Endereço da pilha, Set/Get Scheduling Policy, Desconexão da thread, Prioridade, etc.

Exemplo:

```
pthread_attr_t      attr;
pthread_t          thread;
rc = pthread_attr_init(&attr);
rc = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
rc = pthread_create(&thread, &attr, proc, NULL);
rc = pthread_attr_destroy(&attr);
```

Sincronização

join	Fazer uma thread esperar até que os outras terminem
mutex	evitar inconsistências de dados devido a condições de corrida.
cond var	para suspender a execução e liberar o processador até que alguma condição seja verdadeira.

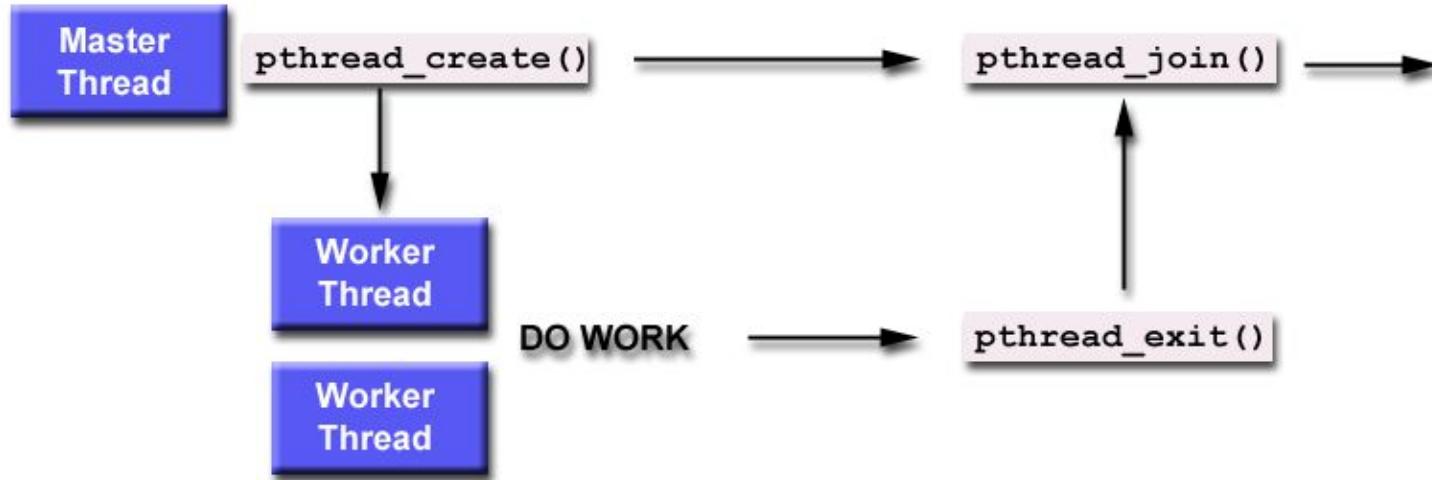
pthread_t : o tipo thread

Operações:

```
int pthread_create(pthread_t *thread,  
                    const pthread_attr_t *attr,  
                    void * (*start_routine)(void *),  
                    void *arg);  
  
int pthread_join(pthread_t thread, void **status);  
int pthread_detach();  
void pthread_exit();
```

- **pthread_create** main thread cria uma thread filho
- **pthread_exit** libera os recursos alocados à thread
- **pthread_join** faz a main thread esperar pelo término da thread filho
 - status = valor retornado no exit
- **pthread_detach** transforma a thread em uma thread de background (daemon) e desvincula o objeto **pthread_t** à sua execução. Não poderá fazer um **join** nessa thread

Criação e Sincronização entre Threads



```
void main() {
    pthread_t tid;
    int status;
    pthread_create(&tid, NULL, thread_worker, NULL);
    ...
    pthread_join(tid, (void*) &status);
    printf("Return value is: %d\n", status);
}
```

```
int pthread_join( pthread_t tid, void* status )
```

// a thread invocadora é bloqueada até que a thread **tid** termina

- **tid** A threadID pela qual deseja-se esperar;
- **status** o valor de retorno da thread executando o exit()

Exemplo de Uso de Threads

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

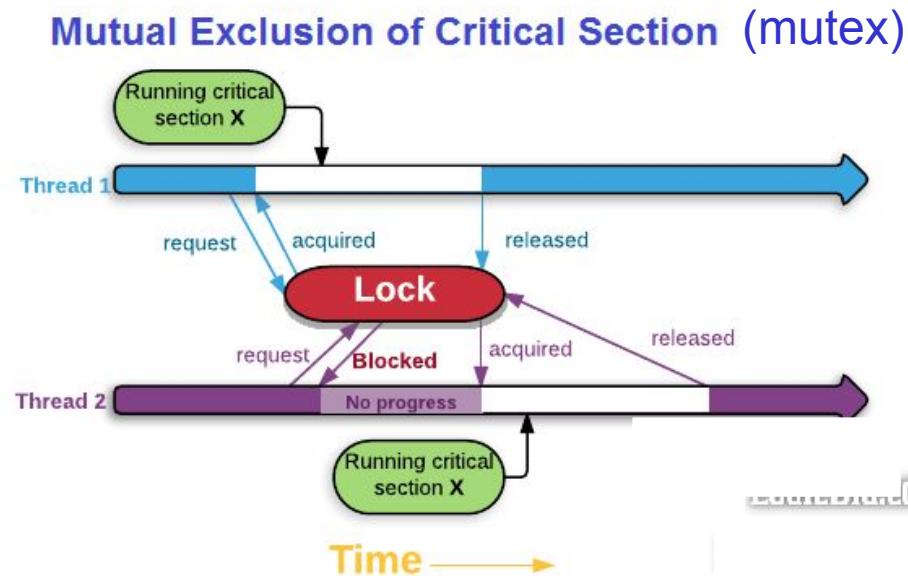
void *PrintHello(void *threadid)
{
    int tid = (int) threadid;
    printf("\n%d: Hello World!\n", tid);
    ... /* do other things */
    pthread_exit(NULL); /*not necessary*/
}

int main()
{
    pthread_t threads[NUM_THREADS];
    int t;
    for(t=0; t < NUM_THREADS; t++)
    {
        printf("Creating thread %d\n", t);
        pthread_create(&threads[t], NULL, PrintHello, (void *)t);
    }

    for(t=0; t < NUM_THREADS; t++)
        pthread_join(threads[t],NULL); /* wait for all the threads to
terminate*/
}
```

Sincronização entre Threads

- Como threads acessam variáveis compartilhadas, é necessário coordenar o acesso a estas usando mecanismos de sincronização
- Todos os pacotes de thread (e pthread) oferecem mecanismos tais como: semáforos, mutexes, barreiras de sincronização e variáveis de condição, etc.



Mutexes (Mutual Exclusion)

Através de **lock** e **unlock**, pode-se garantir que uma única thread execute um determinado código por vez.

- Escopo do mutex precisa estar visível para todas as threads.

Tipo: **pthread_mutex_t**

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                      const pthread_mutexattr_t *attr);  
int pthread_mutex_destroy(pthread_mutex_t *mutex);  
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);  
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Exemplo Mutex

```
void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

main()
{
    int rc1, rc2;
    pthread_t thread1, thread2;
    if((rc1(pthread_create( &thread1, NULL, &functionC, NULL))) printf("Creation failed: %d\n", rc1);
    if((rc2(pthread_create( &thread2, NULL, &functionC, NULL))) printf("Creation failed: %d\n", rc2);

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    exit(0);
}

void *functionC()
{
    pthread_mutex_lock( &mutex1 );
    counter++;
    printf("Counter value: %d\n",counter);
    pthread_mutex_unlock( &mutex1 );
}
```

Variável de Condição

- É uma primitiva de sincronização
- permite que threads aguardem por uma sinalização de outras threads mesmo que estejam dentro de uma região crítica (protégida por um mutex)
- Tipo `pthread_cond_t`

```
int pthread_cond_init(pthread_cond_t *cond,  
                      const pthread_condattr_t *attr);  
int pthread_cond_destroy(pthread_cond_t *cond);  
int pthread_cond_wait(pthread_cond_t *cond,  
                     pthread_mutex_t *mutex);  
int pthread_cond_signal(pthread_cond_t *cond);  
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Exemplo Variável de Condição (½)

```
#include <pthread.h>

pthread_mutex_t count_mutex      = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t condition_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_cond  = PTHREAD_COND_INITIALIZER;

int count = 0;
#define COUNT_DONE 10
#define COUNT_HALT1 3
#define COUNT_HALT2 6

main()
{
    pthread_t thread1, thread2;

    pthread_create( &thread1, NULL, &functionCount1, NULL);
    pthread_create( &thread2, NULL, &functionCount2, NULL);
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    exit(0);
}
```

Exemplo Variável de Condição (2/2)

```
void *functionCount1()
{
    for(;;)
    {
        pthread_mutex_lock( &condition_mutex );
        while( count >= COUNT_HALT1 && count <=
COUNT_HALT2 )
        {
            pthread_cond_wait( &condition_cond,
&condition_mutex );
        }
        pthread_mutex_unlock( &condition_mutex );

        pthread_mutex_lock( &count_mutex );
        count++;
        printf("Counter value functionCount1:
%d\n",count);
        pthread_mutex_unlock( &count_mutex );

        if(count >= COUNT_DONE)  return(NULL);
    }
}
```

```
void *functionCount2()
{
    for(;;)
    {
        pthread_mutex_lock( &condition_mutex );
        if( count < COUNT_HALT1 || count >
COUNT_HALT2 )
        {
            pthread_cond_signal( &condition_cond );
        }
        pthread_mutex_unlock( &condition_mutex );

        pthread_mutex_lock( &count_mutex );
        count++;
        printf("Counter value functionCount2:
%d\n",count);
        pthread_mutex_unlock( &count_mutex );

        if(count >= COUNT_DONE)  return(NULL);
    }
}
```

```
Counter value functionCount1: 1
Counter value functionCount1: 2
Counter value functionCount1: 3
Counter value functionCount1: 4
Counter value functionCount2: 5
Counter value functionCount2: 6
Counter value functionCount2: 7
Counter value functionCount1: 8
Counter value functionCount1: 9
Counter value functionCount1: 10
Counter value functionCount2: 11
```

Exemplo: Produtor-Consumidor em pthreads: buffer de capacidade 1

```
#define MAX 100                                /* Numbers to produce */
pthread_mutex_t the_mutex;                      /* mutex semaphore */
pthread_cond_t consumer_c, producer_c;          // 2 condition variables for
                                                // consumer, and producer
int buffer = 0;

void* producer(void *ptr) {
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex);          /* protect buffer */
        while (buffer != 0)                      /* If there is something
                                                    in the buffer then wait */
            pthread_cond_wait(&producer_c, &the_mutex);
        buffer = rand() +1;
        pthread_cond_signal(&consumer_c);        /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release the buffer */
    }
    pthread_exit(0);
}
```

Produtor-Consumidor em pthreads

```
void* consumer(void *ptr) {
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex);           /* protect buffer */
        while (buffer == 0)                      /* If there is nothing in
the buffer then wait */
            pthread_cond_wait(&consumer_c, &the_mutex);
        printf("content of the buffer is %d\n", &buff);
        buffer = 0;                            /* consume the item */
        pthread_cond_signal(&producer_c); /* wake up producer*/
        pthread_mutex_unlock(&the_mutex); /* release the buffer */
    }
    pthread_exit(0);
}
```

Produtor-Consumidor em pthreads

```
int main(int argc, char **argv) {
    pthread_t pro, con;

    // Initialize the mutex and condition variables
    pthread_mutex_init(&the_mutex, NULL);
    pthread_cond_init(&consumer_c, NULL); /* Initialize consumer cond variable */
    pthread_cond_init(&producer_c, NULL); /* Initialize producer cond variable */

    // Create the threads
    pthread_create(&con, NULL, consumer, NULL);
    pthread_create(&pro, NULL, producer, NULL);

    pthread_join(&con, NULL);
    pthread_join(&pro, NULL);
...
}
```

Barreira de Sincronização

```
# include <pthread.h>
#define THREADS 10
#define MAX 100000
pthread_barrier_t barr; // defining a synchronization barrier
double initial_matrix[ROWS][COLS];
double final_matrix[ROWS][COLS]

void * entry_point(void *arg) {
    int rank = (int)arg;
    int r, col;
    // do some work in parallel
    for (r = rank * MAX/THREADS; r < (rank+1)*MAX/THREADS;
r++)
        for(col = 0; col < COLS; ++col)
            dotProduct(r, col, initial_matrix, final_matrix);

    // Synchronization point
    int rc = pthread_barrier_wait(&barr);

    // do other work after all threads have synchronized
    for (r = rank * MAX/THREADS; r < (rank+1)*MAX/THREADS;
r++)
        for(col = 0; col < COLS; ++col)
            dotProduct(r, col, final_matrix, initial_matrix);
}
```

```
int main(int argc, char **argv)
{
    pthread_t thr[THREADS];

    // Barrier initialization
    if(pthread_barrier_init(&barr, NULL, THREADS))
    {
        printf("Could not create a barrier\n");
        return -1;
    }

    for(int i = 0; i < THREADS; ++i)
        pthread_create(&thr[i], NULL, &entry_point, (void*)i);
    for(int i = 0; i < THREADS; ++i)
        pthread_join(thr[i], NULL);

    double det = Determinant(initial_matrix);
    return 0;
}
```

Gerenciamento de Threads

Ao contrário de processos, threads compartilham a mesma região de memória e outros recursos

Per process items

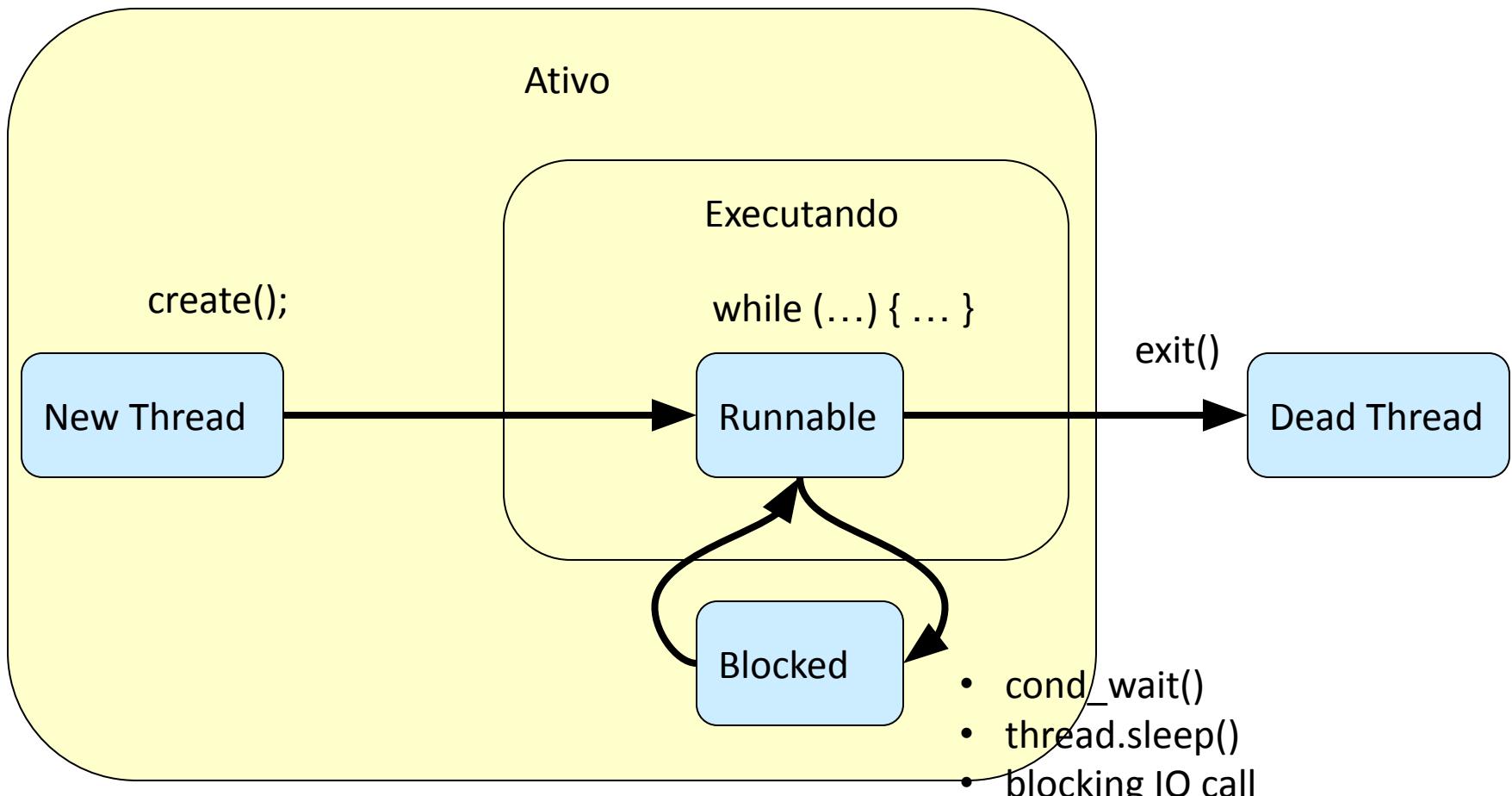
Address space
Global variables
Open files
Child processes
Pending alarms
Signals and signal handlers
Accounting information

Per thread items

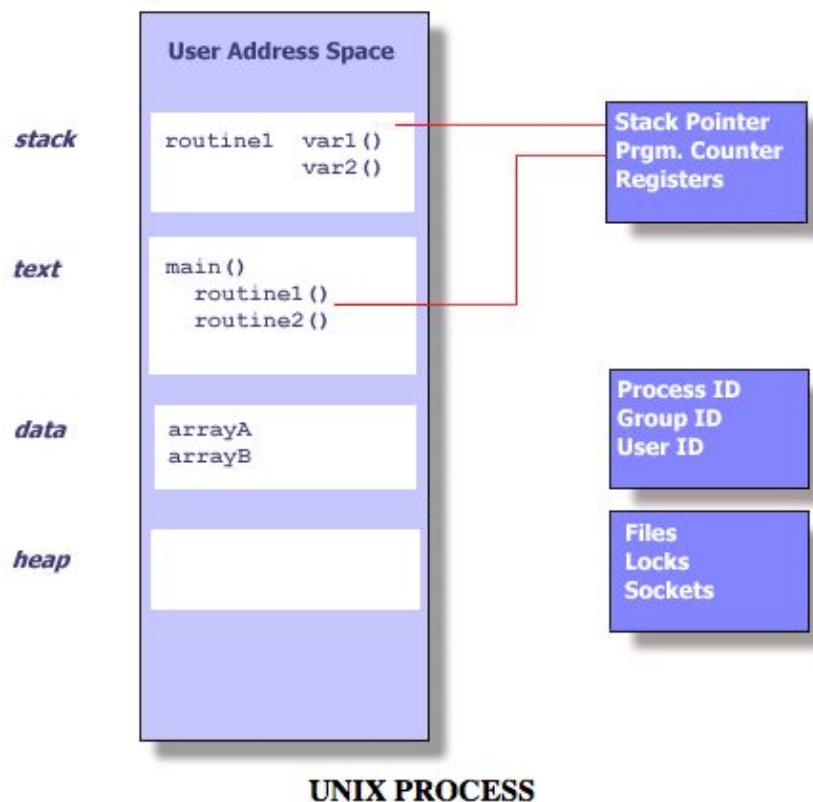
Program counter
Registers
Stack
State

- Cada thread possui sua própria pilha e contexto de CPU (conteúdo de PC, SP, registradores, PSW, etc.)
- Uma thread pode estar nos estados: *executando, esperando e pronta*

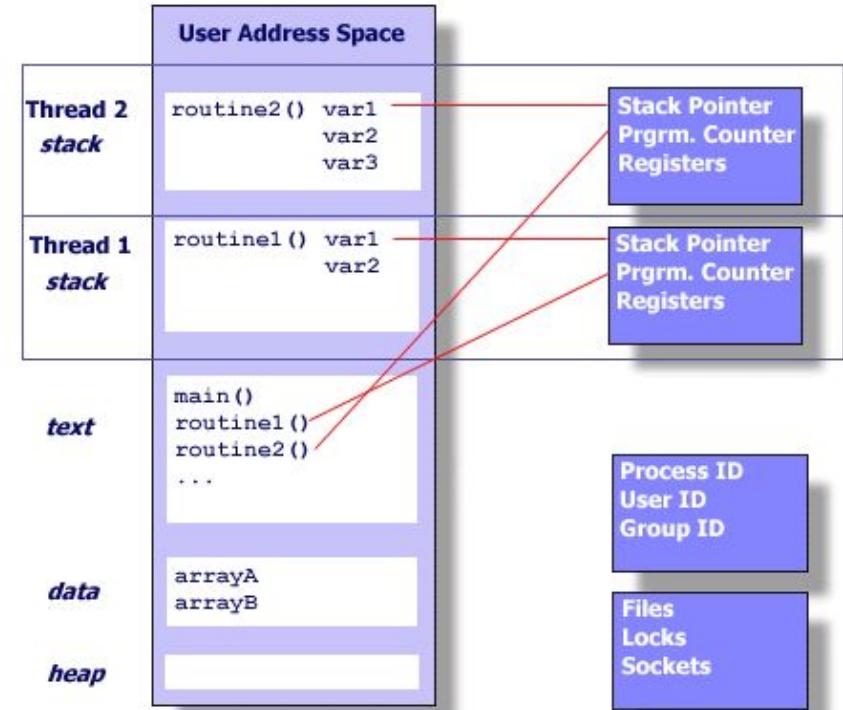
Diagrama de estados de threads



Gerenciamento processos vs threads



UNIX PROCESS



THREADS WITHIN A UNIX PROCESS

O Descriptor de Thread

Para cada thread, o kernel (ou biblioteca de threads) mantém a seguinte informação, que é mantida independente dos descritores de processos (PCBs)

Contexto:

program counter (PC) /* próxima instrução */
process status word (PSW) /* resultado da operação, ex: carry-bit */
stack pointer (SP) /* pilha de execução do thread */
registers /* conteúdo dos registradores da CPU */
state /* blocked, running, ready */
priority
host_process /* processo hospedeiro ou kernel */
thread_Id /* identificador do thread */
processID /* processo ao qual a thread pertence */



Formas de Implementar Threads

Threads em modo kernel (1-para-1):

- thread é a unidade de escalonamento do núcleo
- A biblioteca de chamadas de sistema inclui operações para criar/gerenciar threads
- Algoritmo de escalonamento (de threads) é implementado pelo núcleo
- Exemplos:
 - Windows NT/XP/2000
 - Solaris (anterior à vers. 9)
 - Linux: LinuxThreads ou "Native Posix Thread Library (NPTL)"
 - Apple: Multiprocessing Services
 - NetBSD, FreeBSD
 - Lightweight Processes (LWP)

Threads em modo usuário (N-para-1):

- todas as threads do processo são mapeadas para única thread do núcleo (do processo)
- Se qq thread do processo fizer system-call, todo processo é bloqueado (não aproveita paralelismo de arquiteturas multi-core)
- Política de escalonamento é implementadas na biblioteca de threads
- Exemplos:
 - GNU Portable threads,
 - Thread manager (Apple),
 - Netscape Portable Runtime,
 - State Threads, LWP (SunOS)
 - POSIX P-threads, C-threads

Bibliotecas de Threads

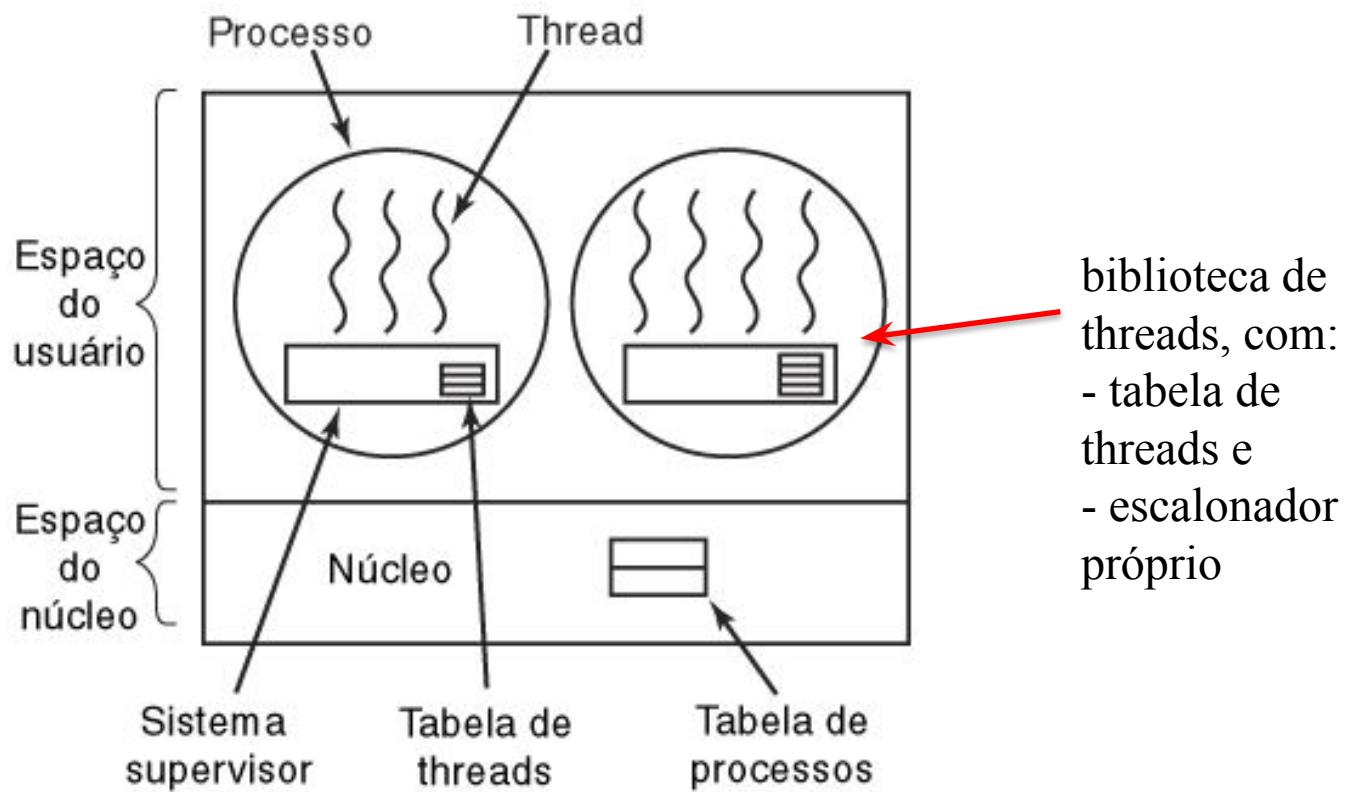
Fornece ao programador uma API para a criação e gerenciamento de threads

As 2 abordagens de implementar uma biblioteca de threads:

1. Biblioteca no espaço do usuário sem suporte ao kernel
 - a. Chamar uma função da biblioteca faz chamada de função local no espaço do usuário e não em uma chamada de sistema
2. Biblioteca no nível do núcleo com suporte direto ao sistema operacional.
 - a. threads são gerenciadas e controladas pelo núcleo

POSIX Pthreads (pode ser fornecida no nível usuário ou núcleo)

Threads em Modo Usuário



Biblioteca de threads em modo usuário:

- Escalonamento das threads de acordo com as necessidades do programa de aplicação. Quando uma thread requisita E/S, bloqueia todo o processo.
- Exemplos: POSIX P-threads, C-threads

Threads em modo Kernel (*Lightweight Process*)

Kernel chaveia entre threads,
independente do processo ao
qual pertencem:

Vantagens:

As próprias funções do kernel
podem ser concorrentes;

Principais problemas:

- Troca de contexto entre threads
precisa considerar proteção de
memória (limites de processo)
- Cada troca de contexto (entre os
LWP) requer um TRAP para o
núcleo (troca modo usuário para
modo supervisor)

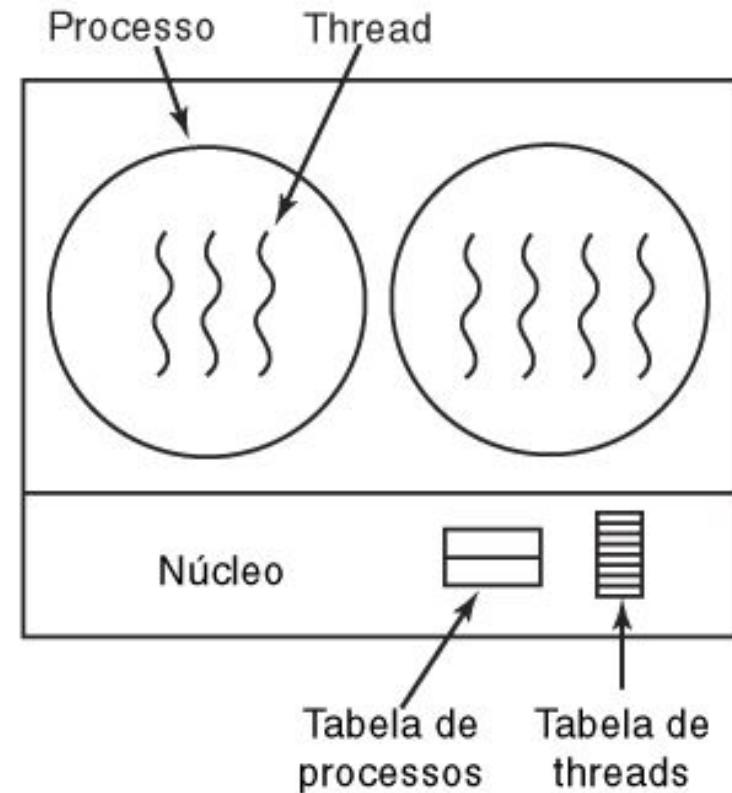


Fig.: Threads gerenciadas pelo núcleo

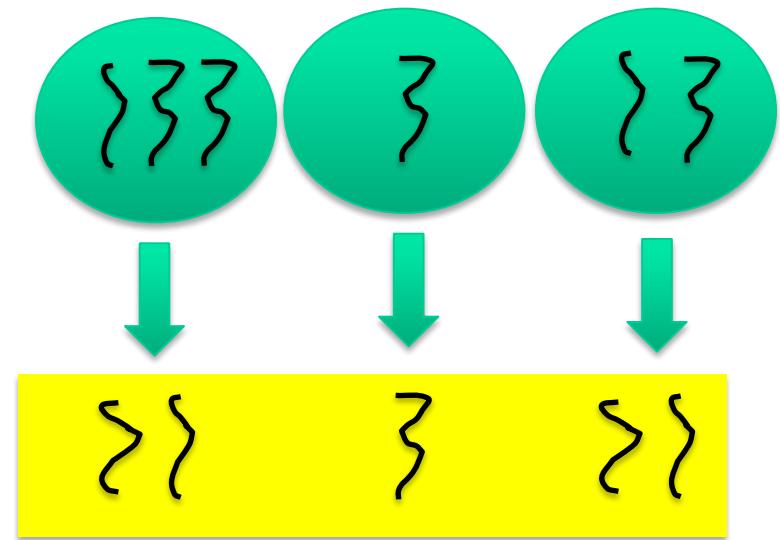
Mais informações em:

<http://www.ibiblio.org/pub/Linux/docs/faqs/Threads-FAQ/html/ThreadLibs.html>

Threads em Modo Híbrido

Threading híbrido (N-paro-M):

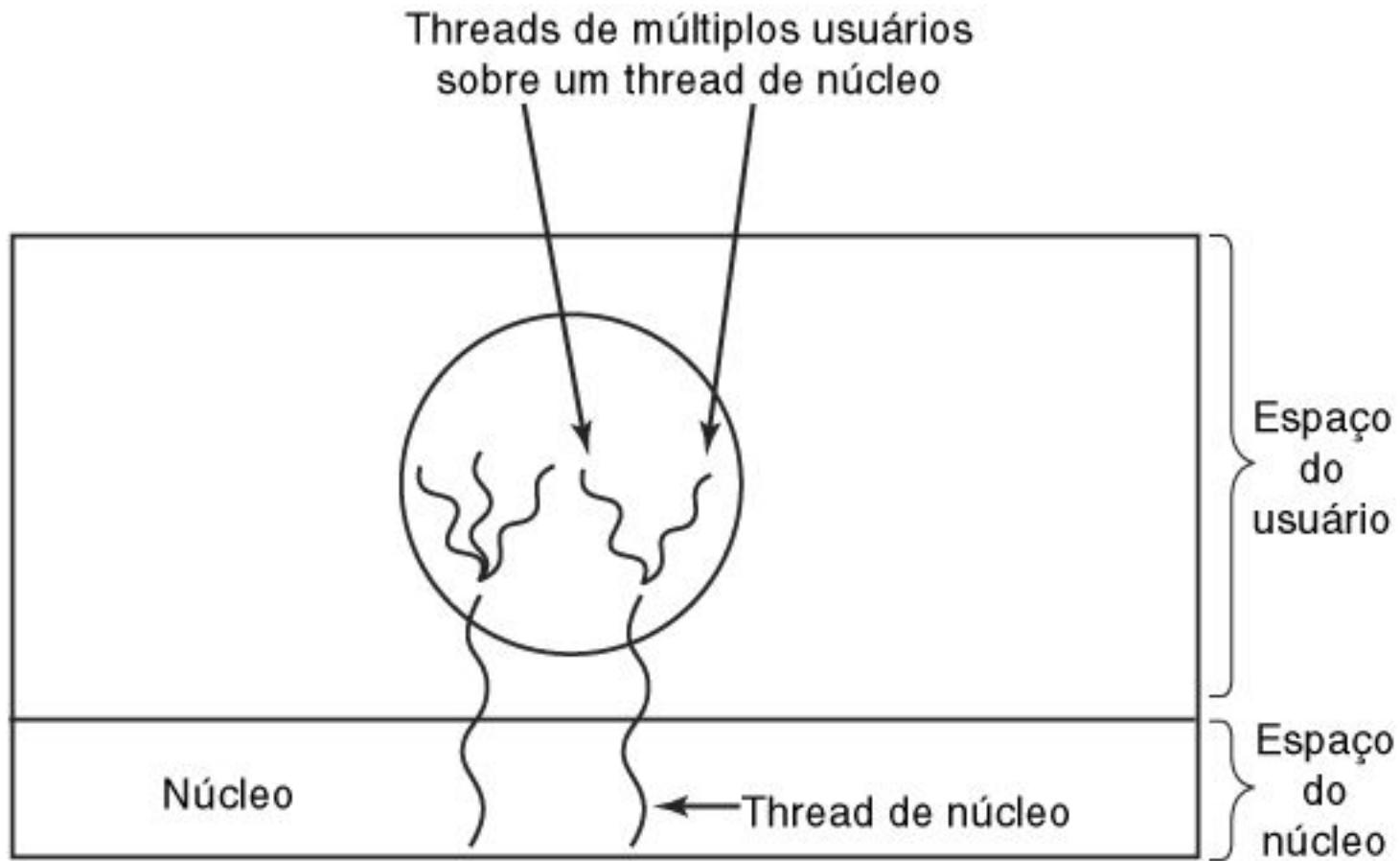
- N threads de um processo são mapeadas em M threads do núcleo
- Mais difícil de implementar (pois os escalonadores do modo usuário e do modo kernel precisam se coordenar)
- troca de contexto entre threads intra-processo não requer troca de contexto no inter-processo
- Mas quando uma thread do modo kernel realiza uma chamada bloqueante, todos os threads no modo usuário correspondentes também entram em estado de espera.
- Exemplos:
 - Microsoft Windows7,
 - IRIX
 - HP-UX
 - Tru64 UNIX
 - Solaris 8



Mais informações em:

<http://www.ibiblio.org/pub/Linux/docs/faqs/Threads-FAQ/html/ThreadLibs.html>

Implementações Híbridas



Multiplexação de threads de usuário sobre
threads de núcleo

Algumas Implementações POSIX Threads (pthreads)

POSIX Threads = modelo de programação, coleção de interfaces que permitem criar, controlar e efetuar o escalonamento, a comunicação e a sincronização entre threads.

Threads em modo kernel:

- **Native POSIX Threading Library (NPTL)**
- **LinuxThreads (para Linux)**
- **Win32 Phtreads**

Threads em modo usuário:

- **FSU Pthreads (SunOS 4.1.x, Solaris 2.x, SCO UNIX, FreeBSD and Linux)**
- **LPW (SunOS 3/4, mips-ultrix, 386BSD, HP-UX and Linux)**
- **pcthreads**
- **pthreads**

Mais informações em:

<http://www.ibiblio.org/pub/Linux/docs/faqs/Threads-FAQ/html/ThreadLibs.html>

Threads em modo núcleo: vantagens e desvantagens

Vantagens:

- se uma thread de um processo bloqueia, outras threads do mesmo processo podem prosseguir
- Núcleo pode ter concorrência

Desvantagens:

- Troca de contexto é menos eficiente (pois requer troca entre modos: usuário → núcleo →usuário)
- Núcleo fica mais complexo (pois precisa implementar tabelas de threads e de processos)
- Desenvolvedor de aplicação tem menos controle sobre o escalonamento das threads de seu processo, já que é o núcleo que faz isso.

Linux Threads

- Usa o conceito mais genérico de task (um processo capaz de compartilhar alguns recursos)
- Chamada `clone()` possibilita variados graus de compartilhamento entre tasks, através do uso dos seguintes flags

Flag	Significado
<code>CLONE_FS</code>	Informação do File System é compartilhada
<code>CLONE_VM</code>	Mesmo espaço de memória é compartilhado
<code>CLONE_SIGHAND</code>	Tratadores de sinal são compartilhados
<code>CLONE_FILES</code>	Conjunto de arquivos abertos é compartilhado

- Chamando `clone()` sem qualquer um das flags setadas equivale a um processo
- Chamando `clone()` com todos os flags setados equivale a criar uma thread, pois todos esses recursos serão compartilhados

Outras questões de projeto

Implementação de threads precisa estar coerente com semântica de algumas system-calls. Por exemplo:

- O que ocorre se processo com >1 threads executa um FORK?, O processo filho deverá ter o mesmo número de threads do pai?
- O que acontece se uma thread executa um close(fd) e arquivo fd ainda está em uso por outra thread?
- Em algumas bibliotecas *malloc* não é reentrante (possui estado). Portanto, pode haver problemas se >1 thread executam esta chamada concorrentemente.
- O que acontece se uma thread faz chamada de sistema, e antes que seja capaz de ler variável global *errno*, outra thread faz outra chamada a sistema e sobre escreve *errno* incial?

Conclusão: quando suporte a threads é incluido no núcleo, a semântica de algumas chamadas de sistema precisa ser re-definida (e bibliotecas re-implementadas).

Scheduler Activations

- Goal – mimic functionality of kernel threads
 - gain performance of user space threads
- Avoids unnecessary user/kernel transitions
- Kernel assigns virtual processors to each process
 - lets runtime system allocate threads to processors
- Problem:
 - Fundamental reliance on kernel (lower layer) calling procedures in user space (higher layer)

Conclusão

- Processo (thread) são as abstrações de uma execução concorrente em um sistema
- Processos possuem um espaço de endereçamento isolado/protegido dos demais processos e pelo menos uma thread
- Threads são fluxos concorrentes compartilhando memória
- Os estados executando, pronto, ou bloqueado se aplicam igualmente a processos e a threads;
- O núcleo gerencia o ciclo de vida, a comunicação, a sincronização e o escalonamento entre processos
- A maioria dos S.O. faz escalonamento preemptivo com vários níveis de prioridades.
- As prioridades são definidas a partir de critérios de uso dos recursos do sistema e de aspectos relacionados ao tipo de aplicação ou do usuário

Listas de Exercícios

1. Suponha que você precisa desenvolver um programa concorrente que faz E/S do terminal (teclado/monitor), da interface de rede (TCP/IP) e é também um cliente de banco de dados, e onde dados intermediários são guardados em uma cache. Você implementaria com múltiplos processos ou como um processo multi-threaded. Justifique.
2. Seja um núcleo que não dá suporte nativo a threads. Então, nessa implementação, o que o núcleo precisa garantir ao fazer a troca de um P1.t1 para um P2.t2 ?
3. Descreva o escalonamento preemptivo de múltiplos níveis com feedback. O que exatamente é esse feedback?
4. Explique o que significa a execução de um processo em determinado ponto estar I/O- e CPU-bound? E por que é interessante dar uma prioridade maior a processos E/O-bound?