Aula 3 - Processos e Threads

Sistemas Operacionais
Ciência da Computação
IFB - Campus Taguatinga

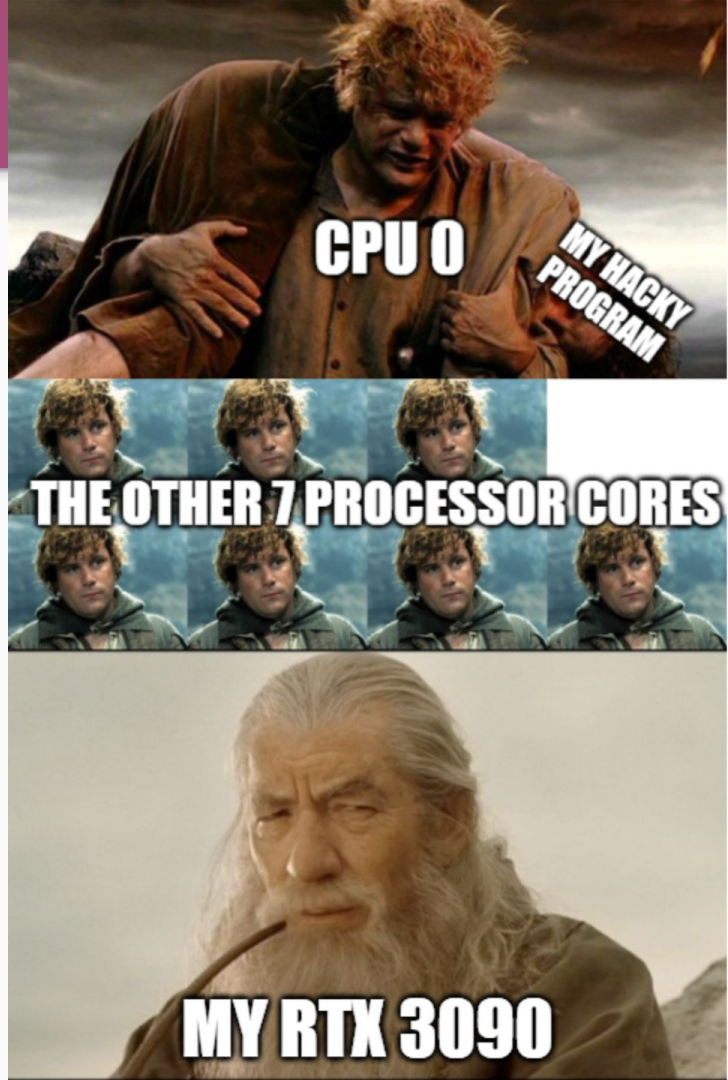
Professor João Victor de A. Oliveira

Na aula passada

- Conceitos de Sistemas Operacionais
  - Processos
  - Espaços de Endereçamento
  - Arquivos
  - Entrada/Saída
  - Proteção
  - Interpretador de Comandos (Shell)
- Estrutura de Sistemas Operacionais

Hoje

- Conceitos básicos
  - Processos
  - Threads



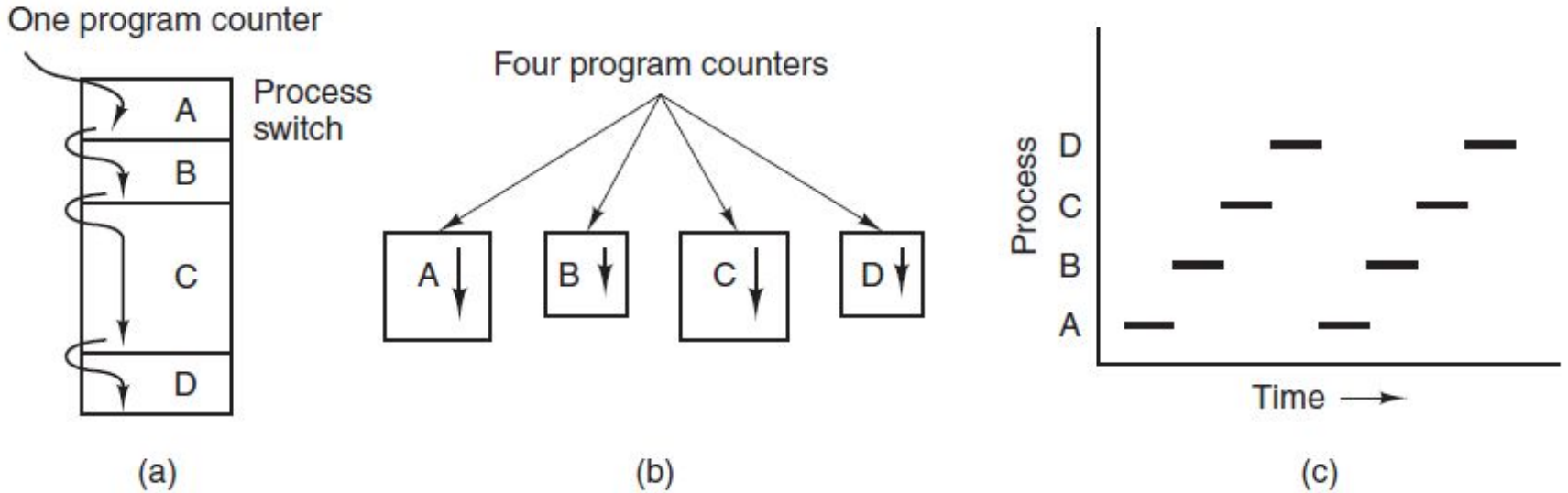
Processos

- Processo é uma abstração de um programa em execução
  - Em um sistema de **multiprogramação**, a CPU muda de um processo para outro rapidamente
    - CPU executa cada processo por dezenas ou centenas de milissegundos
      - Um processo por vez
  - No curso de 1s ela pode trabalhar em vários deles, dando a ideia de paralelismo (**pseudoparalelismo)**

Modelo de Processo

- Todos os softwares executáveis no computador são organizados em uma **série de processos sequenciais**, ou, simplesmente, **processos**
  - **Processo é uma instância de um programa em execução**
    - Inclui valores atuais do contador de programa, registradores e variáveis
    - Conceitualmente, cada processo tem sua própria CPU virtual
      - Na verdade a CPU real troca a todo momento de processo em processo
      - Esse mecanismo de trocas rápidas é chamado de **multiprogramação**

Modelo de Processo



Criação de Processos

- **4 eventos principais para a criação de processos**
  1. **Inicialização do sistema**
     - Interação com o usuário (primeiro plano) ou realizando alguma função sem essa interação (segundo plano)
       - Exemplo de processos em segundo plano: **Daemons**
  2. **Chamada de sistema de criação de processo por um processo em execução**
     - Processos criam e se comunicam com outros processos para cumprir uma dada tarefa
  3. **Solicitação de um usuário para criar um novo processo**
     - Dando um duplo clique em um ícone de programa ou digitando o nome de um programa no terminal
  4. **Início de uma tarefa em lote**
     - fila de processos a serem executados em sequência, por exemplo

Término de Processos

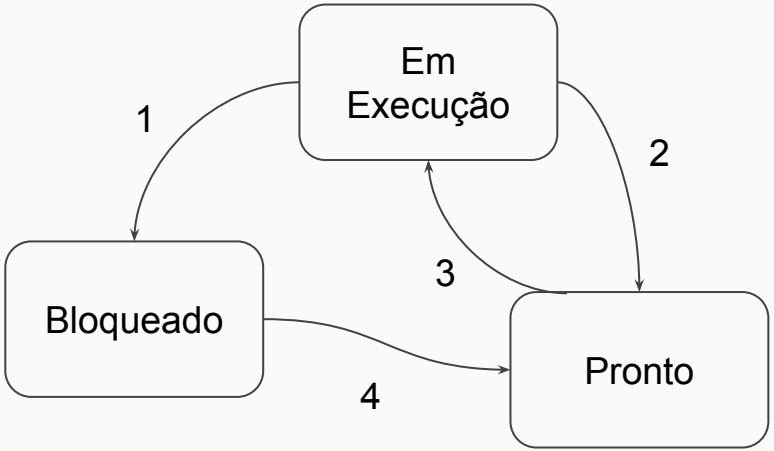
- Um processo termina normalmente por uma das condições a seguir:
  1. **Saída normal (voluntária)**
  2. **Erro Fatal (involuntário)**
     - ex.: se um usuário digita: gcc foo.c, mas o programa foo.c não existe, o compilador anuncia esse fato e termina a execução
  3. **Saída por erro (voluntária)**
     - Um processo pode dizer ao SO que gostaria de lidar sozinho com determinados erros
     - Neste caso o processo é sinalizado (interrompido), em vez de terminado
  4. **Morto por outro processo (involuntário)**
     - Chamada de sistema para matar um processo: **kill**
       - Necessária a devida permissão de um processo para matar outro processo

Hierarquia de Processos

- Em alguns sistemas, quando um processo cria outro, costumam ser associados de alguma maneira **(pai e filho)**
  - Um processo filho pode em si criar mais processos formando uma **hierarquia de processos**
- **UNIX**
  - Processo **INIT**
    - Presente na imagem da inicialização do sistema
    - Todos processos no sistema inteiro pertencem a uma única árvore, com init na sua raiz

Estados de processos

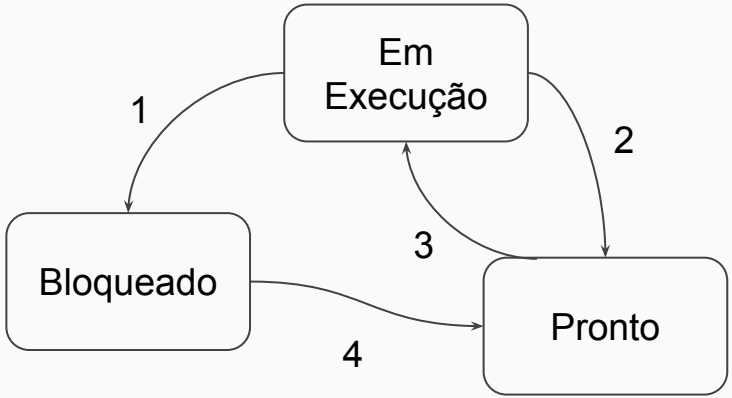
- **Em Execução**
  - Realmente usando a CPU naquele instante
- **Pronto**
  - Executável, temporariamente parado para deixar outro processo ser executado
- **Bloqueado**
  - Incapaz de ser executado até que algum evento externo aconteça



Estados de processos

4 transições possíveis

1. **O SO descobre que um processo não pode continuar sua execução**
   - Ex.: um processo lê de um pipe ou de um arquivo especial (STDIN, por exemplo) e não há uma entrada disponível, o processo é automaticamente bloqueado
2. **O escalonador decide que o processo em andamento foi executado por tempo suficiente**
3. **Processo pronto para execução é colocado em execução**
4. **Um evento externo, pelo qual o processo estava esperando, acontece.**



Implementação de Processos

- Modelo de Processos é implementado pelo SO a partir de uma **tabela de processos**
  - Cada entrada (ou **blocos de controle de processo)** registra informações sobre um processo
    - Ex.: Contador de Programa, ponteiro de pilha, alocação de memória, estado dos arquivos abertos, escalonamento...
    - Ou seja, armazena tudo que deva ser salvo quando o processo **é trocado do estado em execução para pronto ou bloqueado**, de forma que seja possível reiniciá-lo mais tarde como se não tivesse sido parado

Implementação de Processos

- Campos de uma entrada típica na tabela de processos
  - **Gerenciamento de Processo**
    - Registros, Contador de Programa, Palavra de estado do programa, Ponteiro da pilha, Estado do processo, Prioridade, Parâmetros de escalonamento, ID do Processo, Processo Pai, Grupo de Processo, Sinais, Momento em que um processo foi iniciado, Tempo de CPU usado e/ou do processo filho, Tempo do alarme seguinte...
  - **Gerenciamento de Memória**
    - Ponteiro para informações sobre o segmento de texto
    - Ponteiro para informações sobre o segmento de dados
    - Ponteiro para informações sobre o segmento de pilha
  - **Gerenciamento de Arquivo**
    - Diretório raiz, Diretório de trabalho, Descritores de arquivo, ID do usuário, ID do Grupo

Criação de um Processos no Unix

- fork( )
  - Cria um novo processo, chamado-o de processo filho
  - Ambos os processos (pai e filho) irão executar a próxima linha de comando
    - De fato, o processo filho será uma cópia **exata** do espaço de endereçamento do processo pai
  - Processo pai e processo filho serão diferenciados pelo retorno da função:
    - **Se fork( ) retorna um valor negativo:** erro ao criar o processo filho
    - **Se fork( ) retorna zero:** o programa está sendo executado pelo processo filho
    - **Se fork( ) retorna um valor positivo:** o programa está sendo executado pelo processo pai e o valor positivo é o ID do processo filho
  - Exemplo: **fork.c**
- exec( ):
  - Para ler em casa: https://www.geeksforgeeks.org/difference-fork-exec/

Chamada de Sistema Wait

- pid_t **wait**(int ***wstatus**)
  - Bloqueia um processo pai até que algum de seus processos filhos seja terminado ou envie um sinal
    - Após o término de um processo filho, o pai irá continuar sua execução a partir da instrução de chamada de sistema wait( )
  - Quando um processo filho é finalizado, a execução da chamada wait permite que o sistema libere os recursos associados ao processo filho
    - Se o processo wait não foi realizado por um processo pai, o processo filho (morto) é considerado pelo sistema como um ***zombie (ou defunct)*** até que o processo pai seja encerrado
  - **Maiores informações:**
    - No terminal execute o seguinte comando:**man wait**

Exercício 1

Quantos “Hello”s serão impressos no programa ao lado?

```c
#include<stdio.h>
#include<unistd.h>

#define N 4
int main(){

    fork();
    fork();
    fork();
    printf("hello\n");

    return 0;

}
```

Exercício 2

Crie um programa em C que gere um *zombie*… 🧟

Modelando a Multiprogramação

- Um processo médio realiza computação apenas 20% do tempo em que está na memória
  - **Então com cinco processos ao mesmo tempo na memória, a CPU deve estar ocupada o tempo inteiro?**
    - Esse modelo seria irrealisticamente otimista
      - Presume que todos os cinco processos jamais estarão esperando por uma E/S ao mesmo tempo
  - **Modelo mais realista:** examinar o uso da CPU a partir de um **ponto de vista probabilístico**

Modelando a Multiprogramação

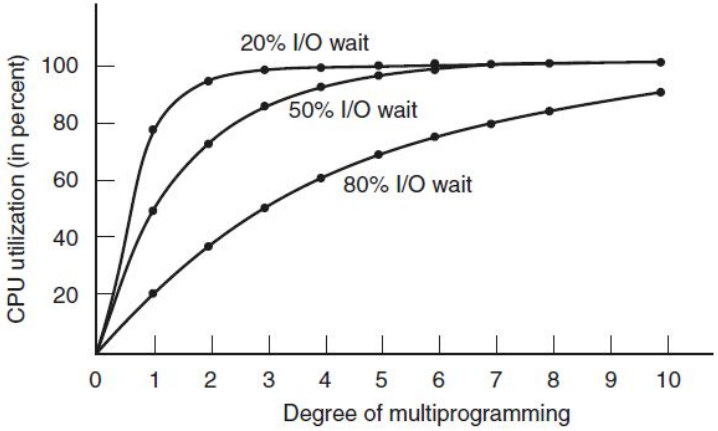
- **Ponto de vista probabilístico**
  - Suponha que um processo passe uma **fração *p*** de seu tempo esperando que os dispositivos de E/S sejam concluídos
  - Com ***n*** processos na memória ao mesmo tempo a probabilidade de que todos os processos ***n*** estejam esperando E/S é p^n
  - **Logo a utilização da CPU é dada pela Fórmula:**

$$Utilização\ da\ CPU = 1 - p^n$$

Utilização da CPU como uma função do número de processos na memória

- Número de processos na memória: Grau de Multiprogramação
  - Se os processos passam **80% do tempo** esperando por dispositivos de E/S pelo menos 10 processos devem estar na memória ao mesmo tempo para que a CPU não desperdice menos de 10% de uso
- Este modelo simples presume que todos os processos são independentes
  - Pode ser usado para realizar previsões específicas do desempenho da CPU

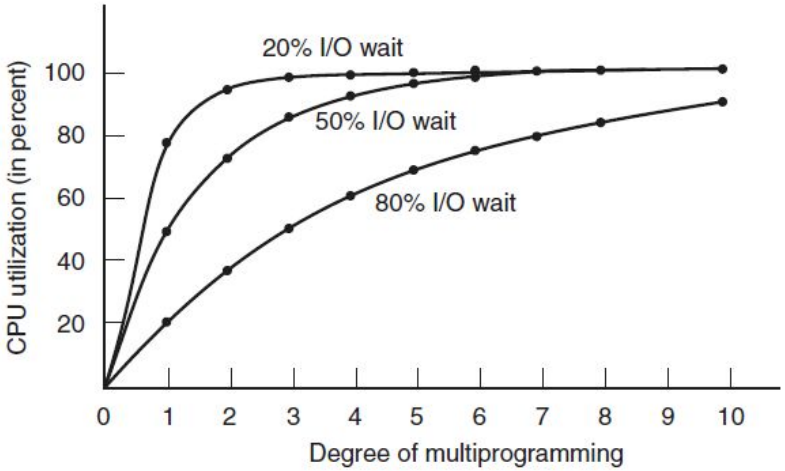
Utilização da CPU = 1 - p^n



Utilização da CPU como uma função do número de processos na memória

- Ex.: Suponha um Computador de 8GB de memória com as características:
  - SO e suas tabelas ocupam 2 GB
  - Cada programa de usuário ocupa 2GB (3 programas de usuário na memória simultaneamente)
  - Com uma espera média de 80% temos que a utilização de CPU é de $1 - 0{,}8^3$ ou em torno de 49%
  - Acrescentando +8GB, aumentamos o **grau de multiprogramação** de 3 para 7
    - Aumenta a utilização da CPU em 30%

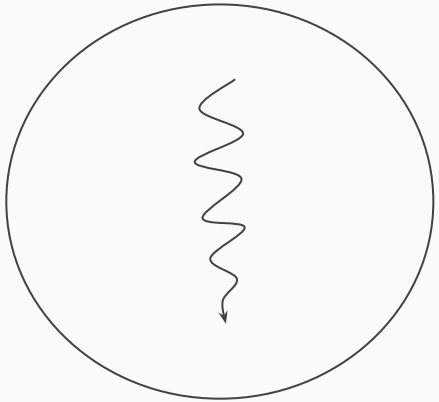
Utilização da CPU = $1 - p^n$

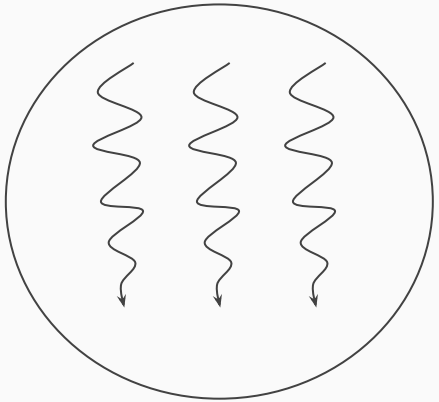


Threads

Threads

- SO Tradicionais: um processo tem um **espaço de endereçamento** e uma ***única thread de controle*** *(linha de execução)*
  - Diferentemente dos processos tradicionais, as ***threads*** *tem a capacidade de compartilhar um mesmo espaço de endereçamento do processo que faz parte e todos os seus dados entre eles*





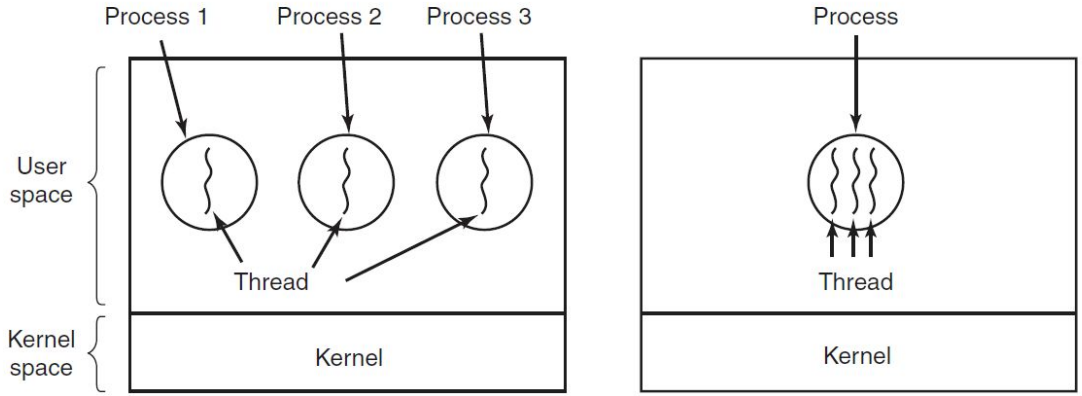
Utilização de Threads

- **Threads são mais leves que processos**
  - **Não precisa armazenar todas as informações que um processo tipicamente armazena**
  - Mais fácil e mais rápido de criar e destruir em comparação aos processos
    - Em alguns sistemas, criar um thread é algo de 10 a 100 vezes mais rápido do que criar um processo
- **Threads resulta em ganho de desempenho**
  - Principalmente quando há uma computação substancial e também E/S substancial
    - Threads permitem que essas atividades se sobreponham, acelerando desse modo a aplicação
- **Threads são úteis em sistemas com múltiplas CPUs**
  - Paralelismo real é possível nesse caso, visto que cada thread poderia ser executada em diferentes CPUs

Modelo de Thread Clássico

- Threads compartilham um espaço de endereçamento e outros recursos
- Processos compartilham memórias físicas, discos impressoras e outros recursos
    - Como threads tem algumas das propriedades dos processo, às vezes eles são chamados de **processos leves**
    - Termo **Multithread** usado para descrever a situação de permitir múltiplos threads no mesmo processos

Modelo de Thread Clássico



(a) Três processos tradicionais

Embora em ambos casos tenhamos três threads, em (a) cada um deles irá operar em um espaço de endereçamento diferente

(b) Único processo com três threads de controle

Todos compartilham o mesmo espaço de endereçamento

Quando executado em uma única CPU, os threads se revezam executando

- Parecem executar em paralelo, cada um em uma CPU com um terço da velocidade da CPU real

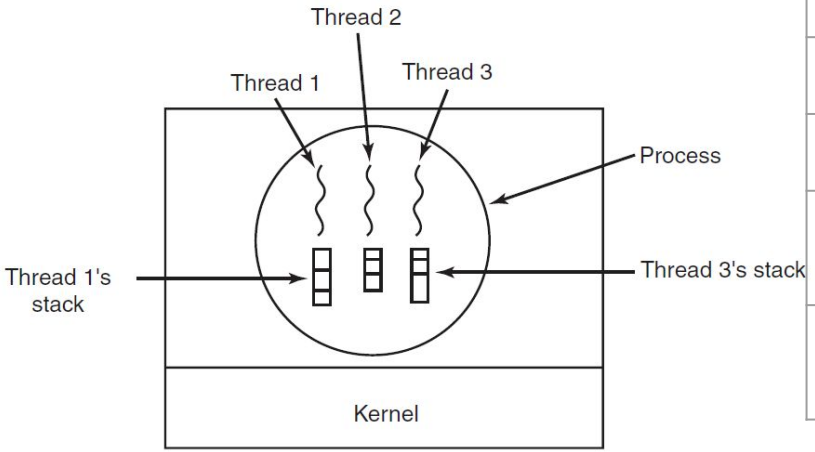
Modelo de Thread Clássico

- **Threads de um mesmo processo tem o mesmo espaço de endereçamento**
  - Logo também compartilham as mesmas variáveis globais
  - Um thread pode ler, escrever ou mesmo apagar a pilha de outro thread
- **Não há proteção entre threads, pois:**
  - É impossível
  - Não seria necessário
- Um processo é sempre propriedade de um único usuário, que presumivelmente criou múltiplos threads de maneira que eles possam cooperar e não lutar entre si

Itens por processo	Itens por thread
Espaço de endereçamento	Contador de Programa
Variáveis globais	Registradores
Arquivos abertos	Pilha
Processos Filhos	Estado
Alarmes Pendentes	
Sinais e Tratadores de Sinais	
Informações de Contabilidade	

Modelo de Thread Clássico

- **Uma thread (assim como um processo) pode estar em qualquer um de vários estados:** em execução, bloqueado, pronto, ou concluído
- Cada thread tem sua própria **pilha**
  - Contém uma estrutura para cada rotina chamada mas não retornada
  - Essa estrutura contém variáveis locais das rotinas e o endereço de retorno para usar quando a chamada de rotina for encerrada



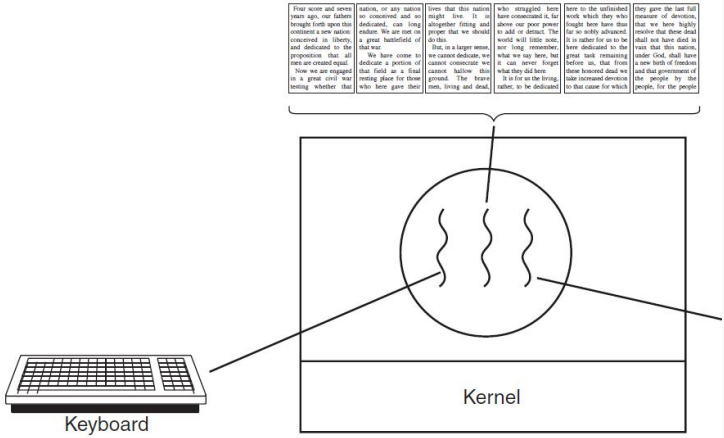
Itens por processo	Itens por thread
Espaço de endereçamento	Contador de Programa
Variáveis globais	Registradores
Arquivos abertos	Pilha
Processos Filhos	Estado
Alarmes Pendentes	
Sinais e Tratadores de Sinais	
Informações de Contabilidade	

Exemplos de utilização de Threads - Processador de Texto com 3 Threads

- Processadores de Texto
  - Exibem o documento que está sendo criado em uma tela **formatada exatamente como aparecerá na página impressa**
  - **O que acontece quando um usuário subitamente apaga uma frase da página 1 de um livro de 800 páginas?**
    - Imagine também que, após isso, ele quer fazer **outra mudança na página 600** e digita um comando dizendo ao processador de texto para ir diretamente até aquela página
    - Qual o problema que o processador de texto deverá resolver?

Utilização de Threads - Processador de Texto com 3 Threads

- O processador de texto agora é forçado a reformatar o livro inteiro até a página 600 **(algo difícil)**
  - A princípio, ele não sabe qual será a primeira linha da página 600 até ter processado todas as páginas anteriores
  - Pode gerar um atraso substancial antes que a página 600 seja exibida
  - Resulta em um usuário infeliz =\
- Threads podem ajudar aqui!!!
  - Poderia ser escrito um programa com 2 threads
    - Thread que interage com o usuário
    - Lida com a reformatação em segundo plano
  - O que acontece se tivermos um salvamento automático nesse processador de texto???



Utilização de Threads - Processador de Texto com 3 Threads

- Processadores de Texto com 3 threads
  - Thread que interage com o usuário
  - Lida com a reformatação em segundo plano
  - Escreve o conteúdo do arquivo presente na RAM para o disco periodicamente
- Esses três processos compartilham de uma memória comum
  - Têm acesso ao documento que está sendo editado
  - Note que com 3 processos isso é basicamente impossível!
    - Por que?

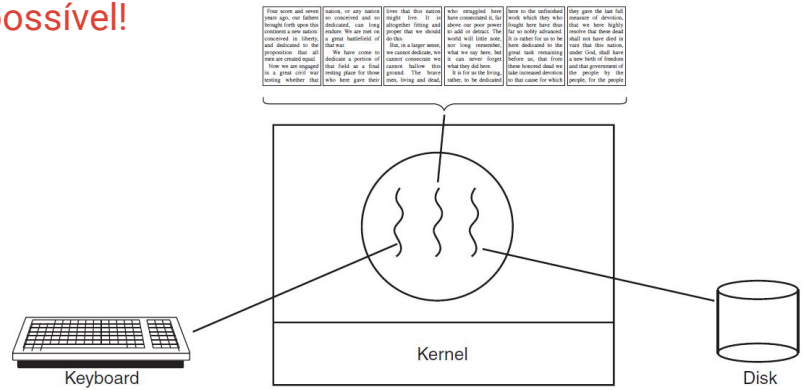
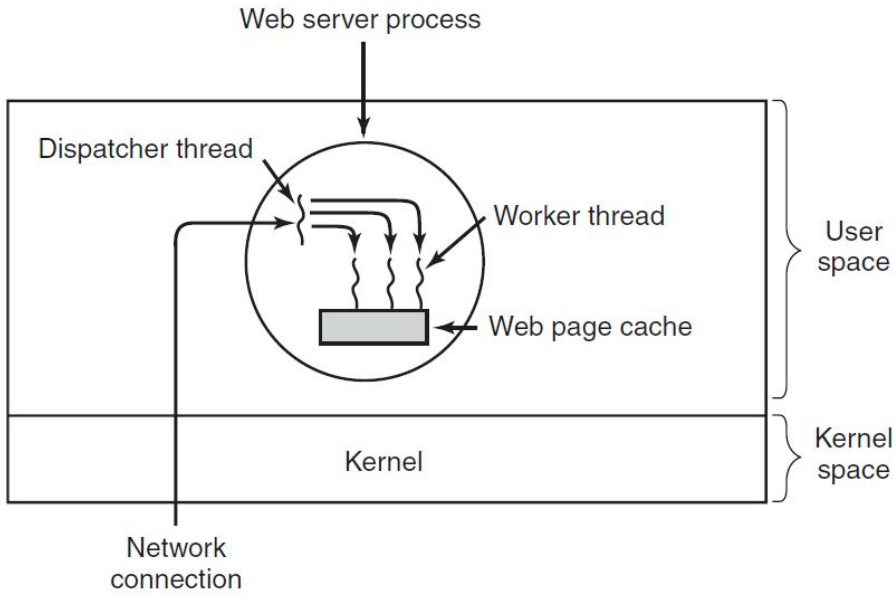


Figure 2-7. A word processor with three threads.

Servidor de web - Multithread



- **Thread despachante**
  - Lê as requisições de trabalho
- **Thread operário**
  - recebe uma solicitação e verifica se a solicitação pode ser satisfeita a partir do **cache** da página da web
    - Se o conteúdo da página está na memória principal

```
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)

Modelo de Thread Clássico

- **Processos normalmente começam com um único thread**
  - Esse thread tem a capacidade de criar novos threads, chamando uma rotina de biblioteca como **thread_create**
    - Um de seus parâmetros é o **nome de uma rotina** para a nova thread executar
  - Quando a thread tiver terminado o trabalho pode concluir sua execução chamando uma rotina de biblioteca como **thread_exit**
    - Além disso, uma thread pode **esperar pela saída de outro thread** através da rotina **thread_join**
      - bloqueia a thread que a executou até que um thread específico tenha terminado
  - Outra chamada de thread comum é a **thread_yield**
    - **Thread abre mão voluntariamente da CPU para deixar outro thread ser executadas**
      - visto que não há uma interrupção de relógio para forçar a multiprogramação de threads

Complicações

- **Se um processo pai tem múltiplas threads, o filho não deveria tê-los também?**
  - O que acontece se uma thread no pai estava bloqueado em uma chamada **read** do teclado?
    - Dois threads estão agora bloqueados no teclado, um no pai e outro no filho?
    - Quando uma linha é digitada, ambas threads recebem uma cópia? Apenas o pai? apenas o filho?
- O que acontece quando uma thread fecha um arquivo enquanto outro ainda está lendo dele?
- E se um thread alocar mais memória e, no meio do caminho, há um chaveamento de threads, e outro também começa a alocar mais memória
  - Memória provavelmente será alocada duas vezes
- **Programas multithread devem ser projetados com cuidado para funcionarem corretamente!!**

Threads POSIX

- Padrão para threads portáteis:
  - IEEE 1003.1c
  - Pacote **Pthreads**
    - Suporte na maioria dos sistemas UNIX
    - Define mais de 60 chamadas de função

Chamada de Thread	Descrição
Pthread_create	Cria um novo thread
Pthread_exit	Conclui a chamada de thread
Pthread_join	Espera que um thread específico seja abandonado
Pthread_yield	Libera a CPU para que outro thread seja executado
Pthread_attr_init	Cria e inicializa uma estrutura de atributos do thread
Pthread_attr_destroy	Remove uma estrutura de atributos do thread

Exemplo de programa usando threads

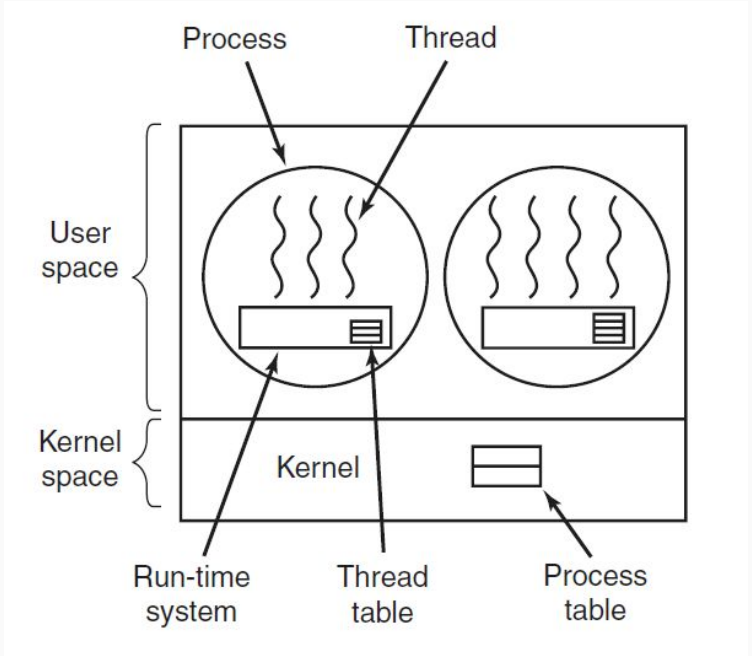
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS     10

void *print_hello_world(void *tid)
{
        /* This function prints the thread's identifier and then exits. */
        printf("Hello World. Greetings from thread %d\n", tid);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        /* The main program creates 10 threads and then exits. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Main here. Creating thread %d\n", i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

                if (status != 0) {
                        printf("Oops. pthread_create returned error code %d\n", status);
                        exit(-1);
                }
        }
        exit(NULL);
}
```

Implementando threads no espaço do usuário

- O pacote de threads está inteiramente no espaço do usuário
  - Núcleo não tem conhecimento sobre threads
  - Para o núcleo ele está gerenciando processos comuns de um único thread
- **Principal Vantagem:** pode ser implementado em SO's sem suporte aos threads
  - Threads, nesse caso, são implementados por uma biblioteca

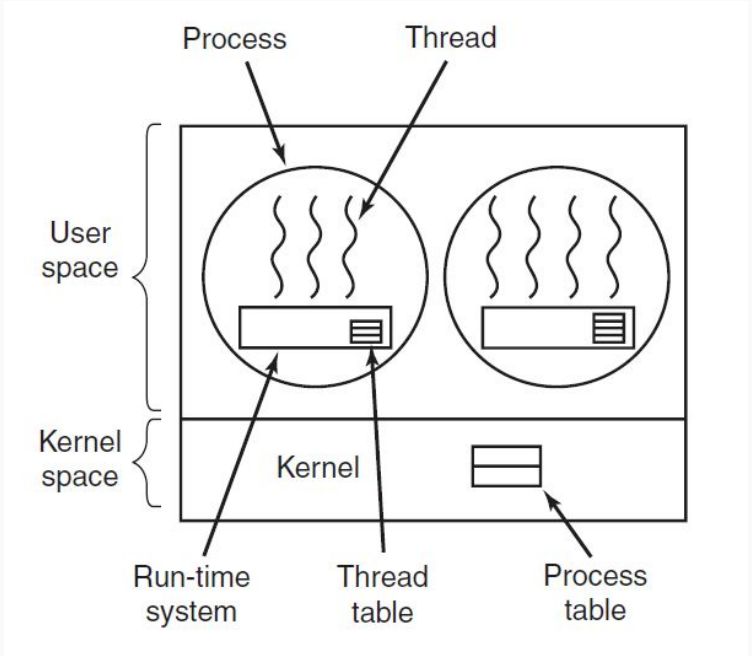
Implementando threads no espaço do usuário

- Cada processo precisa da sua **própria tabela de threads privada**
  - De forma a controlá-los naquele processo
  - Controla apenas propriedades de cada thread
    - Exs.: Contador de programa, ponteiro de pilha, registradores, estado e assim por diante
  - Tabela de threads gerenciada pelo **sistema de tempo de execução**
    - Formada por um conjunto de rotinas que gerencia as threads
      - Pthread_create, pthread_exit, pthread_join, pthread_yield e etc.



Implementando threads no espaço do usuário

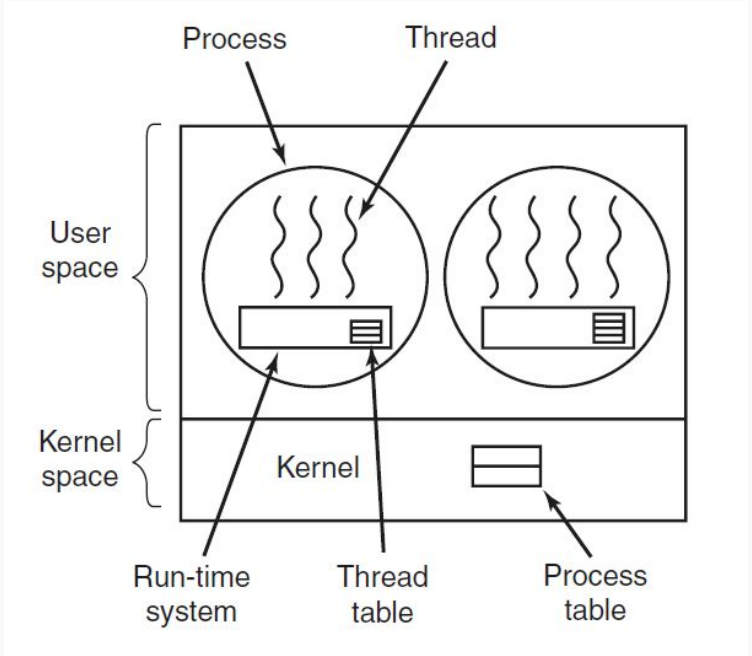
- **Chaveamento de threads** no espaço do usuário é extremamente rápido
  - Pelo menos em comparação ao modo núcleo
    - Não precisa desviar o controle para o núcleo
    - Criar threads, substituir a execução de um thread a outra é bem eficiente no espaço do usuário
    - Escalonamento entre threads de usuário é bastante rápido
      - Aliás, permite que cada processo implemente seu próprio algoritmo de escalonamento!



Ex.: Thread Coletora de lixo

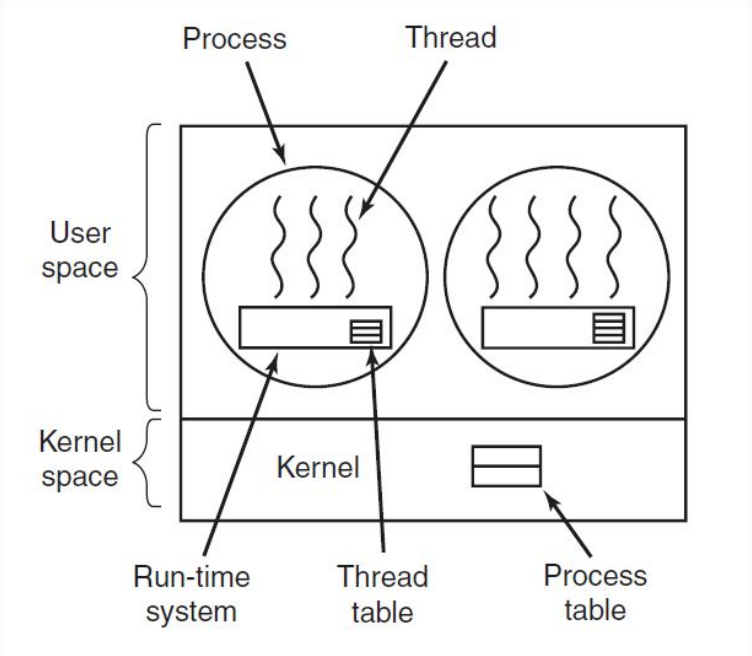
Implementando threads no espaço do usuário

- **Problemas importantes**
  - **Chamadas de sistemas bloqueantes**
    - Uma thread bloqueada pararia todo o processo?
    - **Possível soluções**
      - Modificar chamadas de sistema para que não sejam bloqueantes
        - Tira a vantagem de que as threads de usuário possam ser executadas em qualquer SO
        - Mudança na semântica de chamadas de sistema afeta muitos programas já existentes



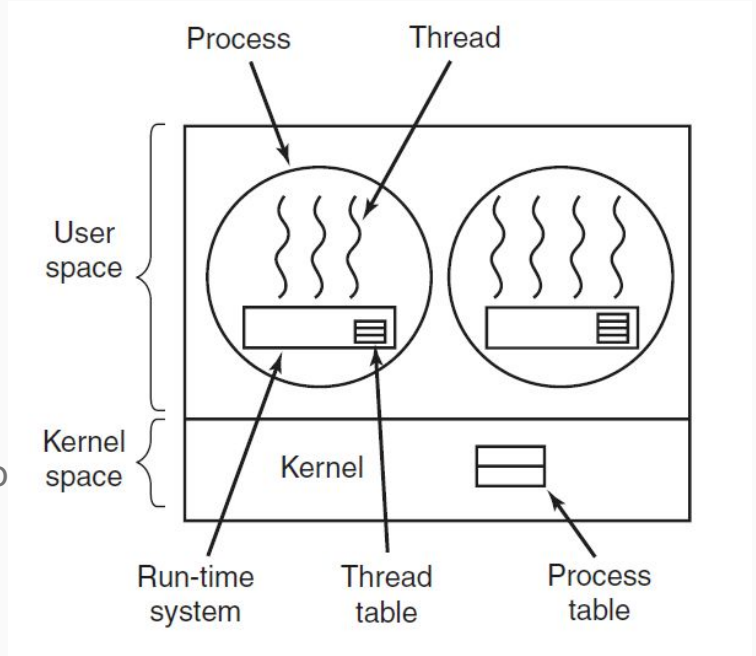
Implementando threads no espaço do usuário

- **Problemas importantes**
  - **Chamadas de sistemas bloqueantes**
    - Uma thread bloqueada pararia todo o processo?
    - **Possível soluções**
      - Dizer antecipadamente se uma chamada será bloqueada
        - Ex.: chamada de sistema **select**
        - Permite a quem chama saber se um *read* futuro será bloqueado
        - Um código colocado em torno de uma chamada de sistema para verificação é chamado de **jacket ou wrapper**



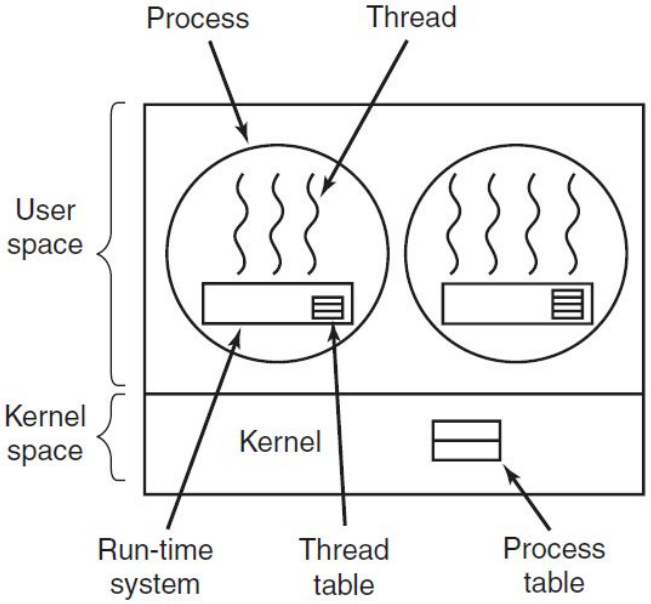
Implementando threads no espaço do usuário

- **Problemas importantes**
  - **Faltas de página**
    - Nem todo o programa pode estar na memória principal ao mesmo tempo
    - Em uma **falta de página** o processo será bloqueado enquanto as instruções necessárias estão sendo buscadas e lidas no disco
      - Se ocorrer uma falta de página na thread, o núcleo bloqueará o processo inteiro
  - **Se uma thread inicia a execução, nenhuma outra será executada**
    - A não ser que esta saia voluntariamente
    - Não há interrupções de relógio em modo usuário



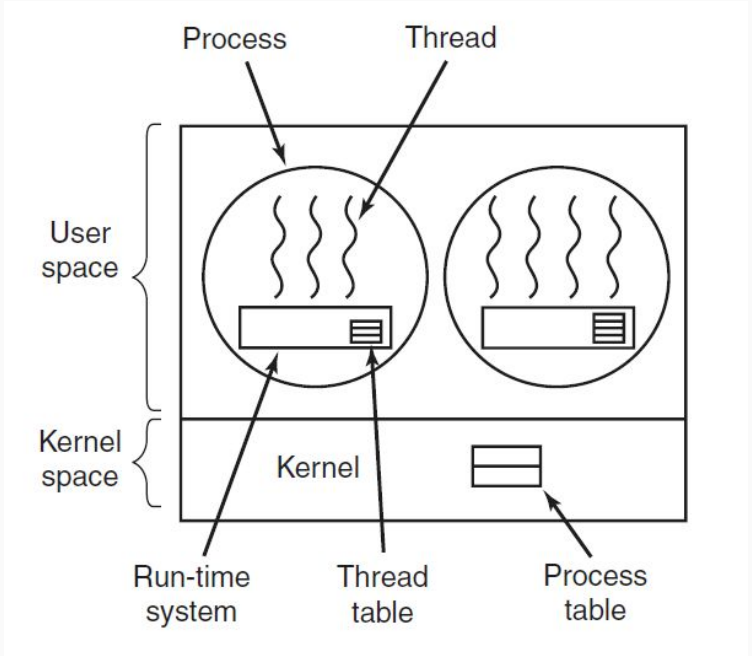
Implementando threads no espaço do usuário

- **Problemas importantes**
  - **Se uma thread inicia a execução, nenhum outro será executado**
    - **Possível solução:** obrigar o sistema de tempo de execução a solicitar um sinal de relógio (interrupção) a cada segundo, para dar a ele controle
      - Sobrecarga total poderia ser substancial



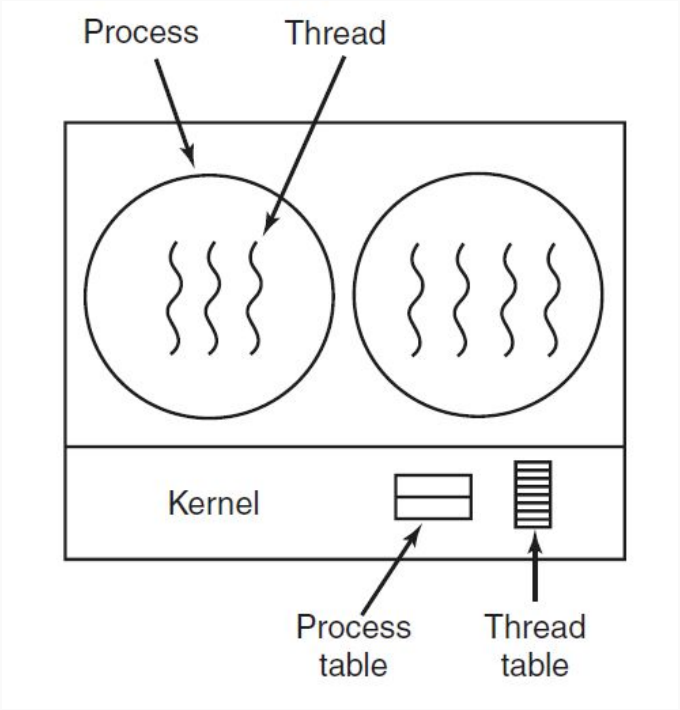
Implementando threads no espaço do usuário

- **Problema mais devastador**
  - Programadores geralmente desejam threads precisamente em aplicações nas quais elas **são bloqueados com frequência**
    - Ex.: Servidor web com múltiplos threads
  - Esses threads estão constantemente fazendo chamadas de sistema



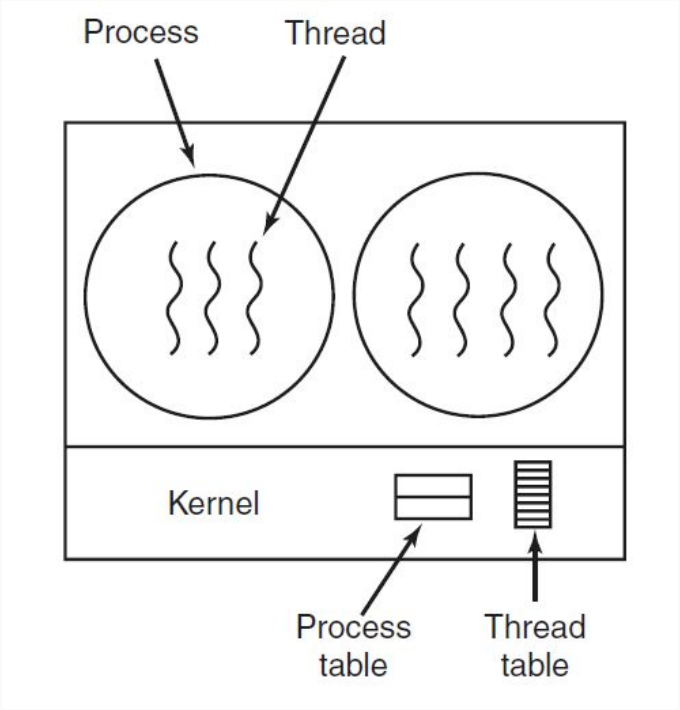
Implementando threads no núcleo

- Considere que agora o núcleo tome conhecimento dos threads
  - Não é mais necessário um **sistema de tempo de execução em cada processo**
  - Não há uma tabela de thread em cada processo
    - **O núcleo terá uma tabela que controla todos threads do sistema**
- Quando um thread quer criar um novo ou destruir um thread existente, isso será feito através de chamadas de núcleo



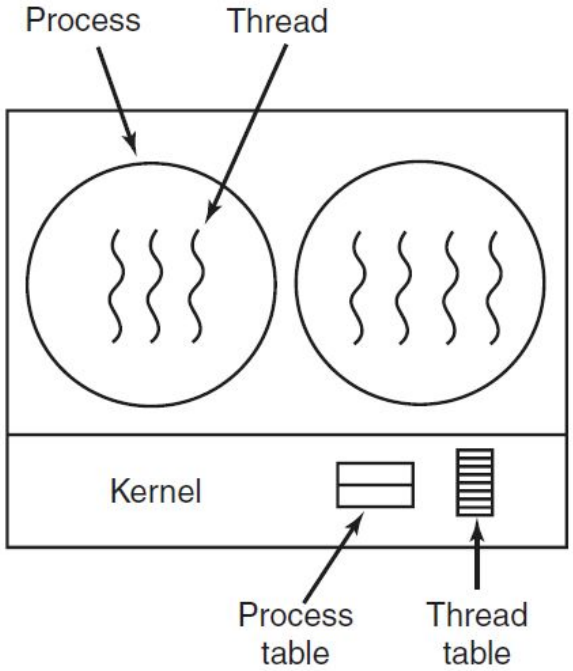
Implementando threads no núcleo

- Quando um thread for bloqueado:
  - Núcleo tem a opção de executar outro thread do mesmo processo ou algum de um outro processo diferentes
- Custo relativamente maior em se criar e destruir threads
  - **Solução ambiental: reciclar threads!**
    - Quando uma thread é destruído, ele é marcado como não executável
      - Mas suas estruturas de dados de núcleo não são afetadas
    - Quando uma nova thread precisa ser criado, um antigo é reativado, evitando parte da sobrecarga



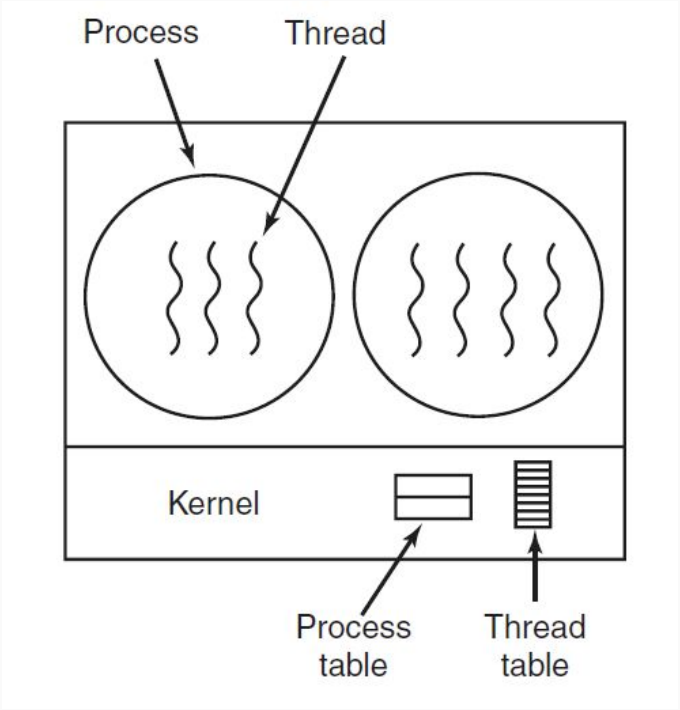
Implementando threads no núcleo

- Threads de núcleo não exigem quaisquer chamadas de sistema novas e não bloqueantes
  - Além disso, se uma thread em um processo provoca uma falta de página, não impede que outros threads do processo sejam executados
- **Desvantagens**
  - Chamadas de sistemas são custosas!
    - Se operações de thread (criação, término, etc.) forem frequentes, ocorrerá uma sobrecarga ainda maior
  - Não resolve problemas do tipo:
    - O que acontece quando um processo de múltiplas threads é bifurcado (*fork*)?



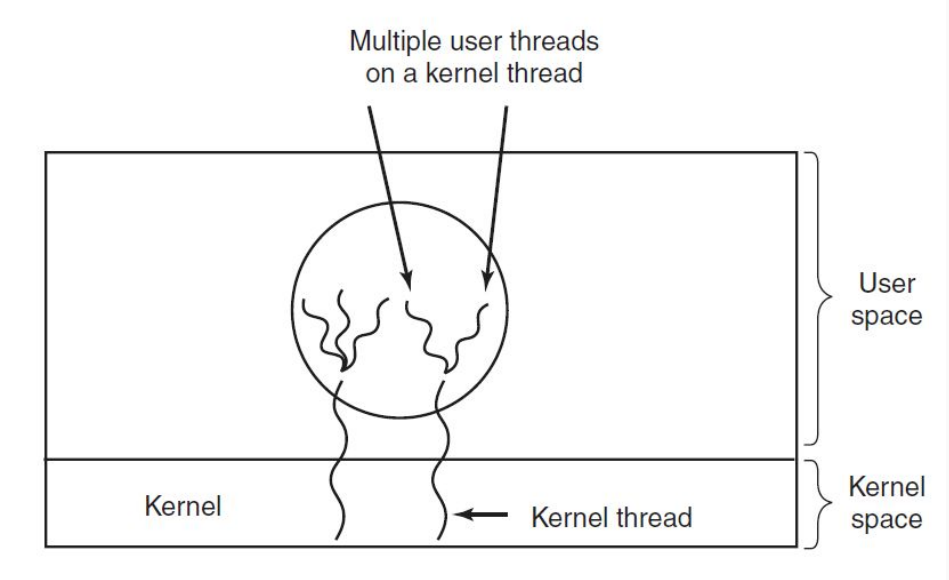
Implementando threads no núcleo

- **Desvantagens**
  - Não resolve problemas do tipo:
    - Quando chegar um sinal ao processo, qual thread deve cuidar dele?
      - O que acontece quando dois threads tem um interesse no mesmo sinal?
    - Dentre outros...



Implementando threads híbridas

- **Objetivo:** combinar as vantagens de threads de usuário com threads de núcleo
  - **Uma maneira:** usar threads de núcleo e então multiplexar os de usuário em alguns ou todos eles



Na próxima aula

- Comunicação de Processos e threads