Curso de Engenharia de Computação
Sistemas Operacionais

Processos e Threads – Parte II

Slides da disciplina Sistemas Operacionais
Curso de Engenharia de Computação
Instituto Mauá de Tecnologia – Escola de Engenharia Mauá
Prof. Marco Antonio Furlan de Souza

Processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Lembrando …**
  - Um processo é uma **instância** de um **programa** em **execução**;
  - Cada **processo** é **independente** dos **demais processos**;
  - Pode-se afirmar que um **processo** é um **agrupamento** de **recursos relacionados**:
    - Cada **processo** possui seu **próprio espaço** de **endereçamento**, **contendo** o **código** (texto) e **dados**;
    - Possui, também, **outros recursos próprios** tais como **arquivos abertos**, **processos-filho**, **alarmes** pendentes, **manipuladores** de **sinais** etc.
  - Ou seja, o **ambiente de execução** de **um processo** é **isolado** dos **demais processos** pelo **sistema operacional**.

Estados de um processo

INSTITUTO MAUÁ DE TECNOLOGIA

- **Conceitos**
  - **Nem todos** os **processos** em execução de uma sistema operacional **possuem** a **mesma velocidade** e **prioridade**;
  - Assim, **cabe** ao **sistema operacional dividir** o **tempo** de **processamento** da **CPU entre** esses **processos** de modo que **todos** consigam ter um **desempenho satisfatório**;
  - Por exemplo, imagine que o **sistema** tem apenas **dois processos** em execução, **P** que está **executando** um **jogo online** e **Q** que está **imprimindo** um **documento**;
  - O **processo P** precisa de **muito mais CPU** que o **processo Q**; no entanto, **não** se pode **deixar Q** totalmente **parado** e **nem interromper P frequentemente**;
  - Desse modo, **cabe** ao sistema **operacional coordenar** seus **processos** para que todos **tenham vez** no uso da **CPU sem afetar** fortemente seu **desempenho**.

Estados de um processo

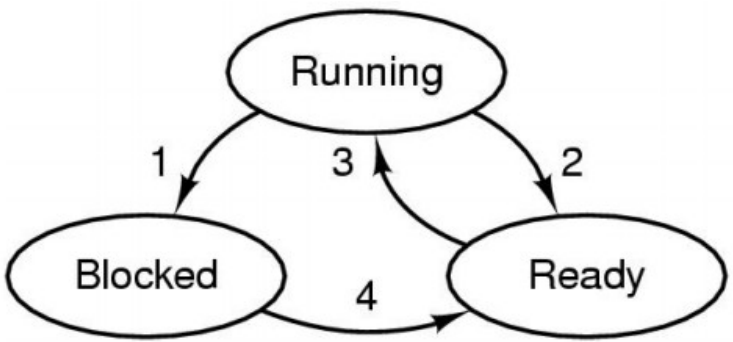
INSTITUTO MAUÁ DE TECNOLOGIA

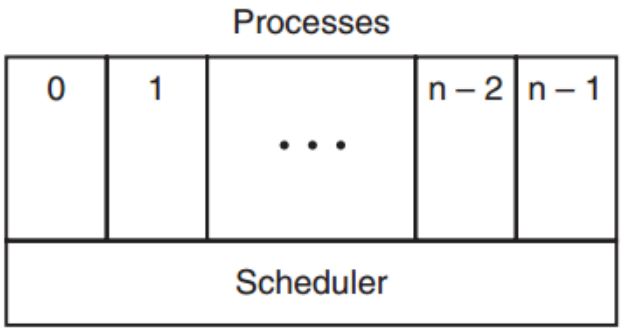
- **Conceitos**
  - O **componente** do **sistema operacional** que **agenda** a **execução** dos **processos** é o agendador (*scheduler*);
  - Para poder **coordenar adequadamente** os processos, todo **sistema operacional** permite que seus **processos** sejam **colocados** em "**estados**" pelo seu **agendador** – **situações de execução distintas** de um **processo** – no caso do Linux, até o usuário final pode alterar o estado de seus processos;
  - Os **estados** em que os **processos** podem **se encontrar** são:
    - **Executando**: isto é, **utilizando** a **CPU** naquele instante;
    - **Pronto**: **pode ser executado** – **parado temporariamente** para que outro processo possa executar;
    - **Bloqueado**: **incapaz** de **executar** até que algum **evento externo ocorra**.

Estados de um processo

INSTITUTO MAUÁ DE TECNOLOGIA

- **Diagrama de estados de um processo**





- Considerar que o processo está no **estado executando** *(running)*.
- Se ele precisar **ler** a **entrada** de um **arquivo**, então o **processo vai** para o **estado bloqueado** (*blocked*) – **transição 1**;
- Quando a **entrada termina**, o **processo vai** para o **estado pronto** (*ready*) – **transição 4**;
- **Quando** o **agendador** (*scheduler*) do **sistema operacional desejar**, ele **moverá** o **processo** para o **estado executando** – **transição 3**;
- **Se necessário**, o **sistema operacional colocará** o **processo** no **estado pronto** para **atender outro processo** distinto – **transição 2**.

Implementação de processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Tabela de processos**
  - Para implementar o **modelo** de **processo**, o **sistema operacional** mantém uma **tabela** (vetor de estruturas), denominada de **tabela de processos**, com **uma entrada por processo** (essas entradas também são chamadas de **blocos de controle de processo**);
  - Cada **entrada contém informações** sobre o **estado** do **processo**, incluindo seu **contador** de **programa** (**PC**), **ponteiro** de **pilha** (**SP**), **alocação de memória**, o **estado** de seus **arquivos abertos**, e outras **informações do contexto de execução** e de **agendamento** e demais informações sobre o **processo** e que **deve** ser **salvo quando** o **processo** é **comutado** do **estado de execução para estado pronto ou estado bloqueado**, para que **possa** ser **reiniciado mais tarde**.

Implementação de processos

INSTITUTO MAUÁ DE TECNOLOGIA
MAUÁ

- **Tabela de processos**
  - **Principais campos** para armazenar informações de processos

Gerenciamento de processo	**Gerenciamento da memória**	**Gerenciamento de arquivos**
• Registradores • Contador de programa (**PC**) • Registrador de estado do programa (**PSW**) • Ponteiro de pilha (**SP**) • Estado do processo • Prioridade • Parâmetros de agendamento • ID do processo (**PID**) • Processo-pai • Grupo de processo • Sinais • Momento que foi iniciado • Tempo de CPU utilizado • Tempo de CPU utilizado pelos filhos • Momento do próximo alarme	• Ponteiro para informação do segmento de texto • Ponteiro para informação do segmento de dados • Ponteiro para informação do segmento de pilha	• Diretório raiz • Diretório de trabalho • Descritores de arquivo • ID do usuário (UID) • ID do grupo (GID)

Implementação de processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Como diversos processos utilizam uma mesma CPU?**
  - Existe em uma **parte** da **memória** em que se **armazenam endereços** de **procedimentos de interrupção** *(interrupt service procedure)* que **respondem** às **interrupções** – **vetor de interrupções**;
  - Uma **interrupção** é um **evento** que **altera** o **fluxo** de **execução normal** de um **programa** e pode ser **gerado** por **dispositivos** de **hardware** ou até **mesmo pela** própria **CPU**;
  - **Exemplo**: suponha que o processo 3 do usuário esteja em execução e que uma **interrupção** de **disco** tenha **acontecido** – **neste caso**, os **dados** do **processo 3** tais **como PC**, **PSW**, **um** ou **mais registradores são empilhados** na **pilha atual pelo hardware de interrupção**;
  - Depois, cabe a um **procedimento** de **interrupção** ser **executado** para **tratar** a **interrupção**.

Implementação de processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Como diversos processos utilizam uma mesma CPU?**
  - **Quando** se executa uma **interrupção**, os **registradores** do **processo** que estavam no **topo** da **pilha** são **salvos** na **entrada** da **tabela** de **processo do processo** que estava **atualmente** em **execução** e que foi **interrompido**;
  - Em **seguida**, as **informações** do **processo armazenadas** no **topo** da **pilha** pela **interrupção** são **removidas** e o **ponteiro** da **pilha** e este é **definido** para **apontar** para uma **pilha temporária usada** pelo **manipulador** de **processos**;
  - **Depois**, entra em **execução** um **procedimento** (normalmente escrito em **C**) que executa **código específico** da **interrupção**;
  - Pode ser que este código **altere** o **estado** de **algum processo** para **pronto**. De qualquer forma, **após** o **término** deste **procedimento**, o **agendador** é **chamado** para **determinar** qual **processo** será **executado** em **seguida**;
  - **Depois**, uma **rotina** se **encarrega** de **carregar** os **registradores** e **mapa** da **memória** para o **novo processo escolhido** pelo **agendador** e o **coloca** em **execução**.

Modelagem de multiprogramação

INSTITUTO MAUÁ DE TECNOLOGIA

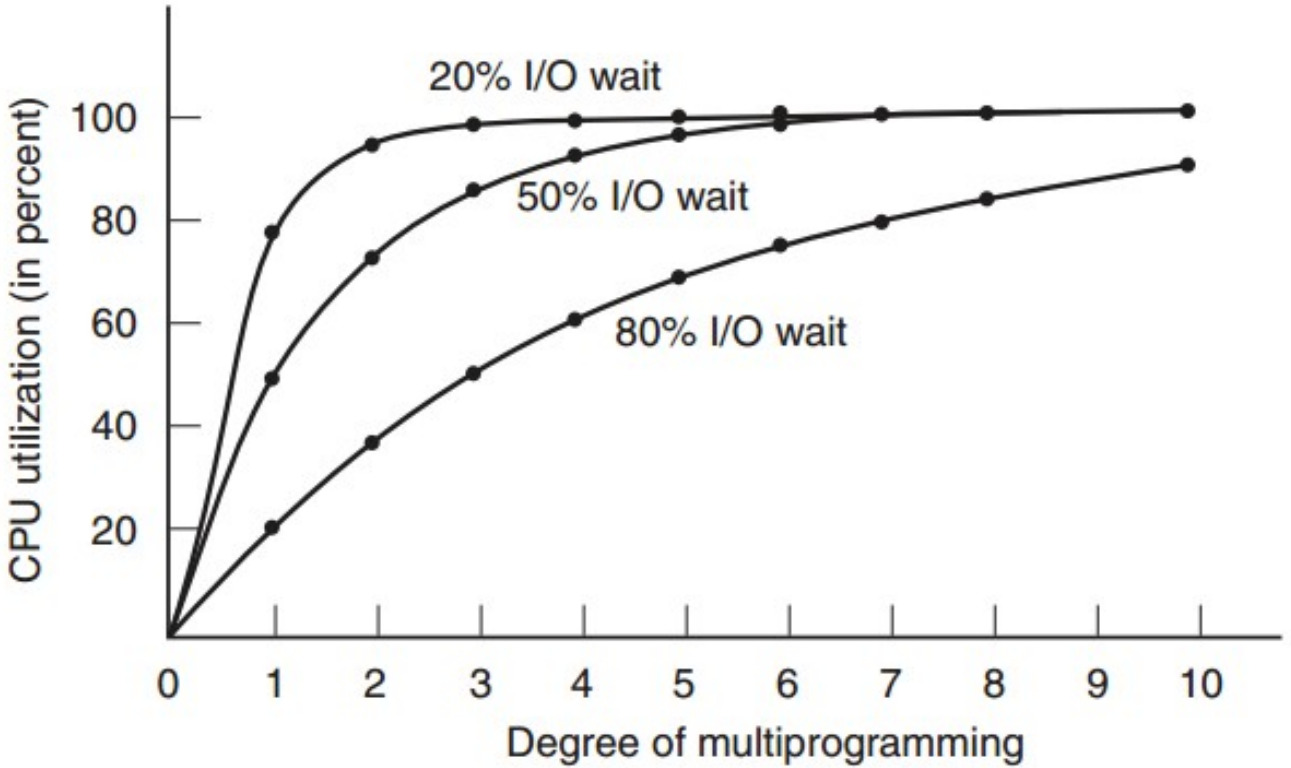
- **Como a multiprogramação afeta o consumo da CPU?**
  - Se existem ***n* processos** em **execução** na **memória** e supondo que eles são **idênticos** e, ainda, supondo que a **probabilidade** deles serem **interrompidos** (por algum evento de E/S) seja ***p***, então a **utilização da CPU** pode ser **calculada** como a **probabilidade** que os **processos** estão todos em **execução**:

$$\text{Utilização}_{\text{CPU}} = 1 - p^n$$

  - Esta é uma **simplificação pois processos** possuem algum **tipo** de **interação** e a **forma correta** de modelá-los **matematicamente** é pela **teoria das filas**;
  - Com esta fórmula é possível traçar curvas de utilização da CPU de acordo com a probabilidade de interrupção (veja o próximo slide).

Modelagem de multiprogramação

- **Como a multiprogramação afeta o consumo da CPU?**



Threads

INSTITUTO MAUÁ DE TECNOLOGIA

- **Conceitos**
  - ***Thread*** (“linha”) representa **em computação** “**linha de processamento**”;
  - Um **processo comum** apresenta **uma** e **única linha** de **processamento** – as **instruções** do processo que são **executadas** do seu **início até** seu **término**;
  - Um **processo multithreaded** é aquele que, **internamente**, é capaz de **executar** diversas **threads simultaneamente** – **similar** à **execução simultânea** de diversos **processos** – e com **compartilhamento** de **recursos** do **processo** que as **executou**;
  - Assim, deste ponto para frente, o nome **thread representará** o **conceito** de um “**processo leve**” que pode ser **executado** dentro do **espaço** de **endereçamento** de um **processo** e com **compartilhamento** de seus **recursos**.

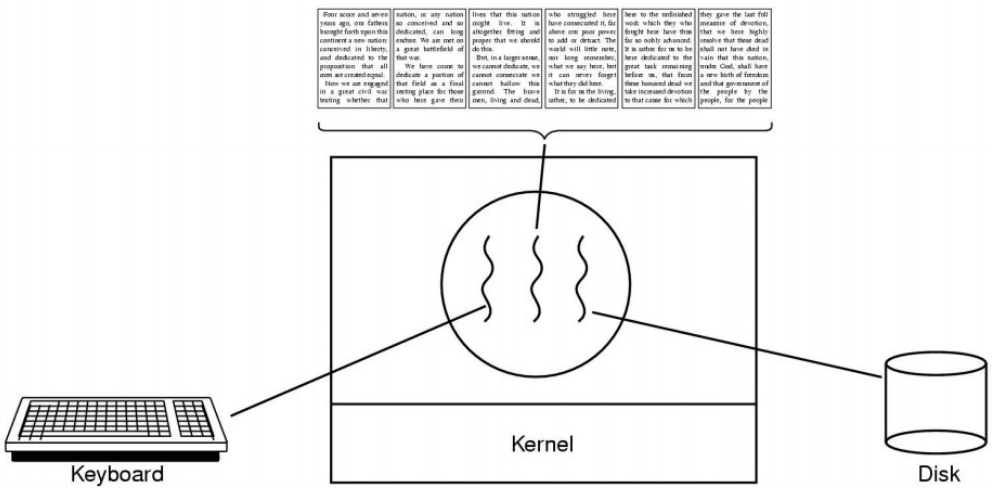
Threads

INSTITUTO MAUÁ DE TECNOLOGIA

- **Por que utilizar threads?**
  - Existem **diversas aplicações** em que é necessário **executar** uma série de **atividades** sobre um **mesmo conjunto** de **recursos**. **Como processos não compartilham recursos**, uma **implementação multithreaded** é **mais simples neste caso**;
  - **Threads** são mais "**leves**" que processos – para se **manter threads em** um **processo**, **gasta-se menos memória** e é **mais rápido criar** e **destruir threads** do **que processos**;
  - O uso de **threads melhora** o **desempenho** de um **processo** no **sentido que**, quando uma **thread** do **processo bloqueia** para **tratar interrupções** de E/S, suas outras **threads** podem **continuar** a **execução**, **aproveitando melhor** o **tempo** de **CPU**;
  - **Sistemas operacionais** atuais **conseguem aproveitar threads** de um **processo** em **múltiplas CPUs**, **melhorando** ainda **mais** o **desempenho** do processo.

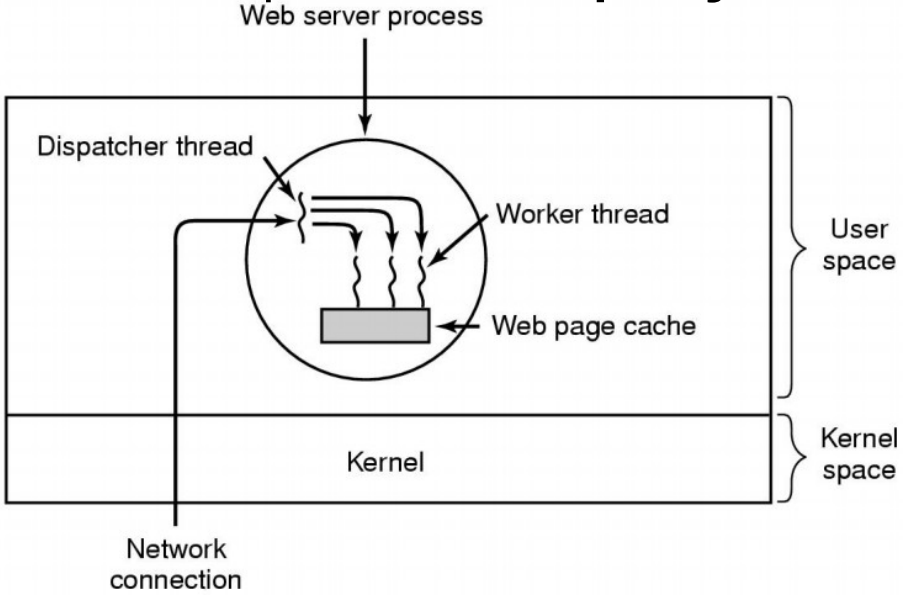
Threads

- **Exemplos de utilização de threads**
  - **Editor de texto**
    - Thread para **interagir** com o **usuário**;
    - Thread que **formata** o **texto** em **segundo plano**;
    - Thread que **salva** o **texto automaticamente** em **segundo plano**.



Threads

- **Exemplos de utilização de threads**
  - **Servidor Web**
    - **Thread** para **receber as requisições HTTP** e **despachar** para **outras threads** o **envio** de **páginas** para o **usuário** (***dispatcher***);
    - **Thread** que **trata** a **requisição** e **envia página** ao **usuário** (***worker***).



Dispatcher

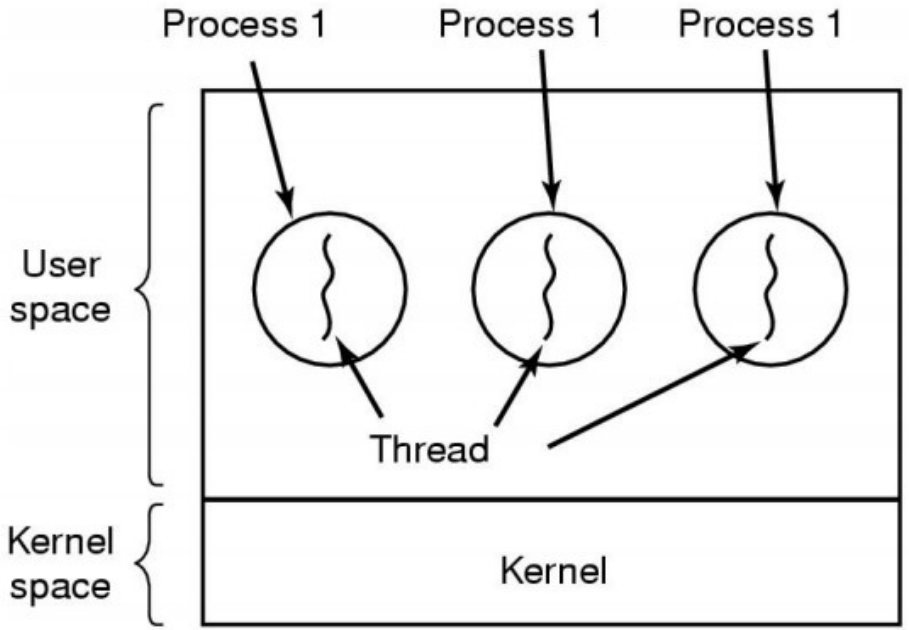
```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

Worker

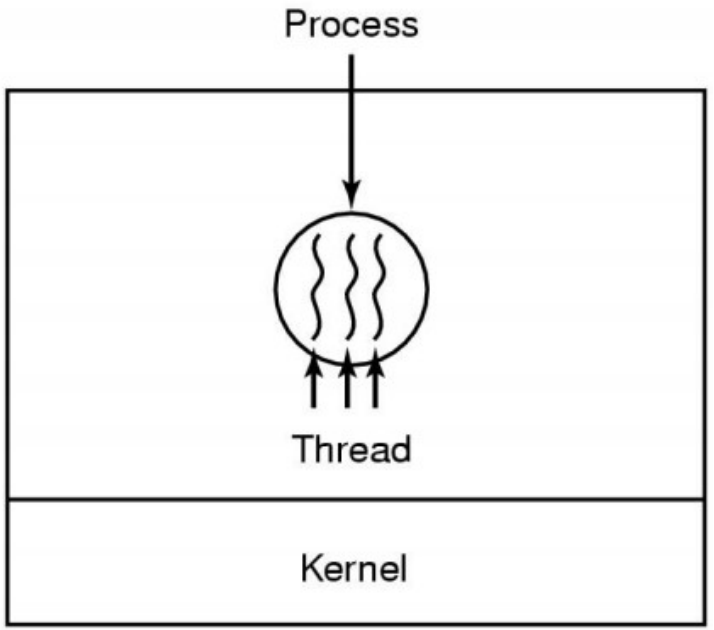
```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

Modelos de threads

- **Modelo clássico**
  - **Threads** são **executadas** em um mesmo **ambiente** de **processo**:



Três processos (threads isoladas)



Um processo com três threads

Modelos de threads

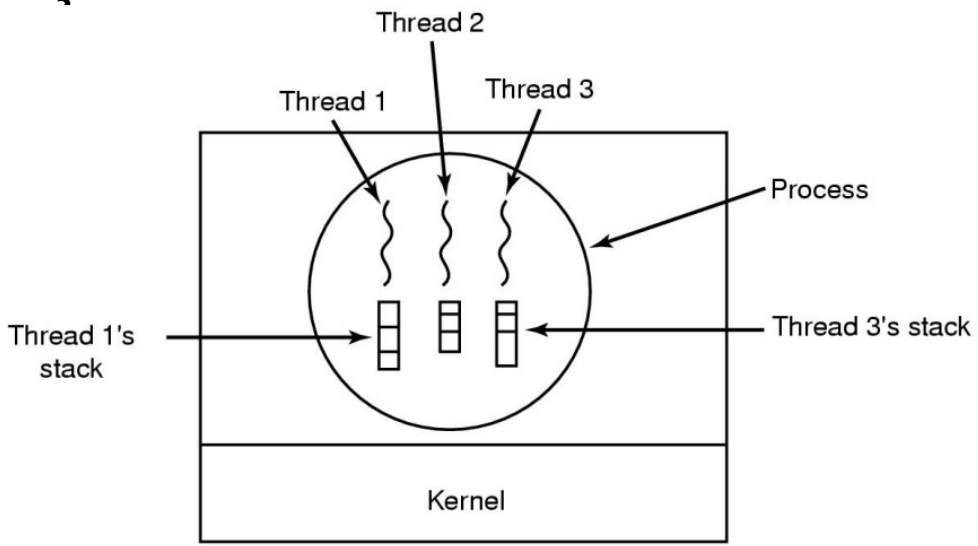
INSTITUTO MAUÁ DE TECNOLOGIA

- **Modelo clássico**
  - Uma **thread conta** com os **dados** a **seguir**, parte **recebidos** do **processo** que a criou e **parte** da **própria thread**:

Do processo	**Específico da thread**
• Espaço de endereçamento • Variáveis globais • Arquivos abertos • Processos-filho • Alarmes pendentes • Sinais e manipuladores de sinais • Outras informações do processo	• Contador de programa • Registradores • Pilha • Estado (executando, bloqueado, pronto ou terminado)

Modelos de threads

- **Modelo clássico**
  - Uma **thread** pode estar em um dos **estados** a **seguir**: **executando**, **bloqueada**, **pronta** ou **terminada** (como em processos);
  - Cada **thread** possui sua própria **pilha**: armazena as **variáveis locais** de **funções executadas** e não terminada dentro da **thread**:



Modelos de threads

INSTITUTO MAUÁ DE TECNOLOGIA

- **Modelo clássico**
  - **Funções “típicas”** para **manipular threads**:
    - `thread_create`: **cria** uma **thread** a partir de uma **função passada** como **argumento** e que será **executada nela**;
    - `thread_exit`: **termina** a **execução** de uma thread que concluiu seu “trabalho”;
    - `thread_join`: **bloqueia** a **thread** que **executou** esta **função** até que a **execução** da **thread passada** como **argumento** a esta **função** tenha **terminado**;
    - `thread_yield`: permite que uma **thread voluntariamente libere CPU** para **outra**(s) **thread**(s). Esta **função** é **importante** pois não existe **comutação** de **threads** por **interrupção**, como nos processos.

Modelos de threads

INSTITUTO MAUÁ DE TECNOLOGIA

- **Modelo clássico**
  - **Problemas com threads**
    - Se um processo possui muitas threads em execução, um processo-filho dele também as terá:
    - E se uma thread do processo-pai for bloqueada, a mesma thread no processo-filho também será bloqueada?
    - Em um mesmo processo, o que acontece com uma thread que fecha um mesmo arquivo que outra thread ainda está lendo?
    - E se uma thread verifica que precisa alocar mais memória e então ocorre uma comutação entre threads e outra verifica que também precisa alocar mais memória, é alocada duas vezes mais memória que o necessário?
  - Para **lidar** estes **problemas**, é **necessário estudar** com **cuidado** a biblioteca de **threads** que se está **utilizando** – de qualquer modo, **programar** um **sistema multithreaded** é **mais complicado** que o **similar multiprocesso** …

Implementações de threads

INSTITUTO MAUÁ DE TECNOLOGIA

- **Threads POSIX**
  - O **IEEE** definiu um **padrão** para **threads** – padrão **IEEE 003.1c**;
  - A **biblioteca** mais conhecida que **implementa** este padrão é a biblioteca **Pthreads** (em C);
  - Diversos sistemas UNIX e Linux implementam esta biblioteca;
  - Algumas funções (~ 60 no total):

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Implementações de threads

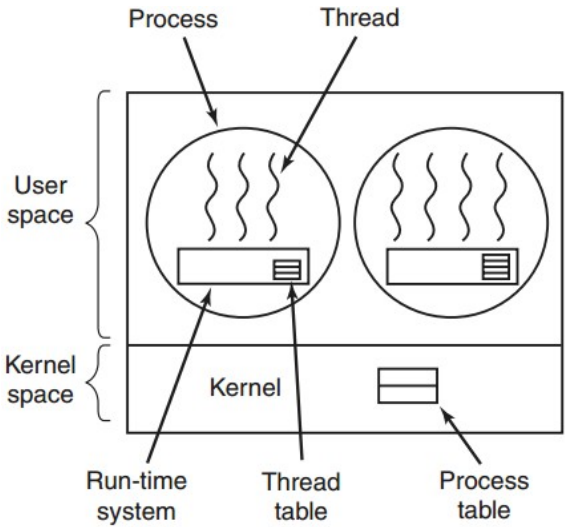
INSTITUTO MAUÁ DE TECNOLOGIA

- **Threads POSIX - Exemplo**

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
/* Para compilar: gcc -Wall -Wextra hello_world_pthread.c -o hello_world_pthread -lpthread */
#define NUMBER_OF_THREADS 10
void *print_hello_world(void *tid) {
    printf("Hello World. Greetings from thread %d\n", *((int *)tid));
    pthread_exit(NULL);
}
int main(void) {
    pthread_t threads[NUMBER_OF_THREADS];
    int status;
    for (int i = 0; i < NUMBER_OF_THREADS; i++) {
        printf("%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(EXIT_FAILURE);
        }
    }
    exit(EXIT_SUCCESS);
}
```

Implementações de threads

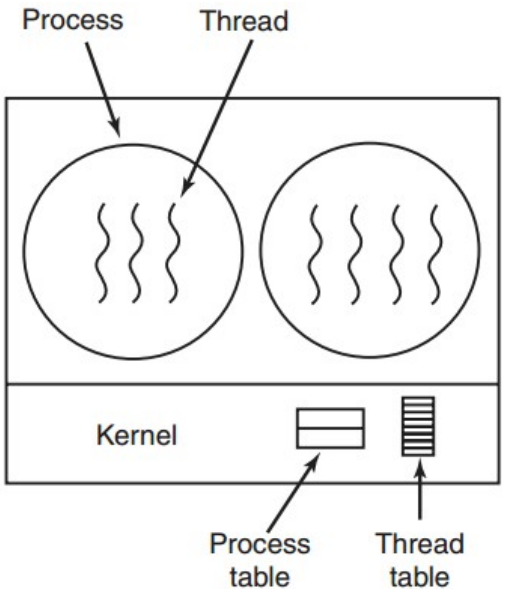
- **Possibilidades de implementação de threads**
  - **No espaço** do **usuário**
    - Neste caso, as **threads** são **independentes** do **kernel**;
    - As threads são **controladas** em um **processo** por uma **tabela** de **threads** contendo contador de programa, ponteiro de pilha, registadores, informações de estado etc;
    - **Vantagens**: **(1)** podem ser **implementadas** em sistemas operacionais que **não suportam threads; (2)** é mais eficiente (não comuta para o kernel); **(3)** Agendadores eficientes;
    - **Desvantagens**: **(1)** se uma thread gerar uma **interrupção** de E/S, as **demais** ficam **bloqueadas** – a **menos que** a chamada de **E/S** seja **não bloqueante**; **(2)** se uma thread não liberar CPU, as demais ficam bloqueadas; **(3) aplicações** que **usam muito chamadas** de **sistema** (como servidor WEB) com este tipo de implementação apresentam desempenho sofrível.



Implementações de threads

INSTITUTO MAUÁ DE TECNOLOGIA

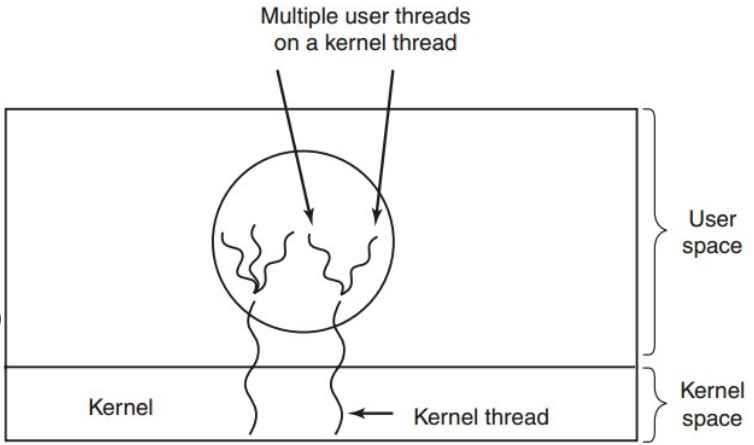
- **Possibilidades de implementação de threads**
  - **No kernel**
    - Neste caso, **não** é **necessário implementar** o **mecanismo** de **execução** de **threads** no espaço do usuário – ele está no kernel (tabela de threads e algoritmos) – a biblioteca de threads é mais simples;
    - Mas a **biblioteca** de **threads** terá de fazer **chamadas** de **sistema** ao **kernel** para usar threads – esta comutação **consome tempo**! Técnicas como **reciclagem** de **threads** melhoram o desempenho;
    - O **problema** de **chamadas** de **sistema bloqueantes** é **minimizado** – o kernel detecta outras threads não afetadas pela chamada e coloca-as para executar;
    - Alguns **problemas**: (1) um **processo** que possui **várias threads executa** um **fork** – o que **acontece** com as **threads** do **kernel**?; (2) um **processo** recebe um **sinal** – e se uma ou mais **threads** registram-se para tratar o sinal. **Qual delas trata**?



Implementações de threads

INSTITUTO MAUÁ DE TECNOLOGIA

- **Possibilidades de implementação de threads**
  - **Abordagem híbrida**
    - Uma forma híbrida possível é **combinar threads** do **espaço** do **usuário** com **threads** do **kernel**;
    - O **programador** pode **determinar** quantas **threads** de **kernel** deseja **utilizar** e quantas **threads** do **espaço** do usuário **utilizará** e **então multiplexar** estas **últimas** em **threads** do **kernel** – as **threads** do **espaço** do **usuário** se "**revezam**" no **uso** das **threads** de **kernel**;
    - As **threads** de **kernel** são **agendadas** no **kernel** e as **threads** do **espaço** do **usuário** são apenas **criadas**, **destruídas** e **agendadas** no **espaço** do **usuário**.



Implementações de threads

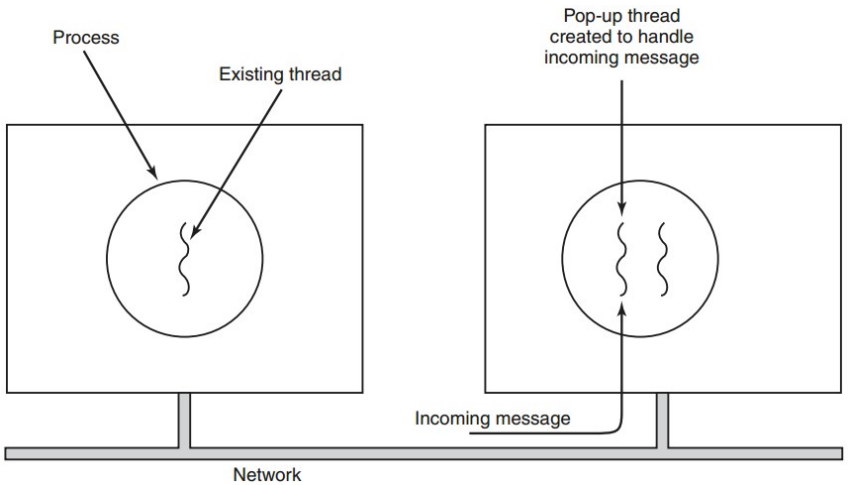
INSTITUTO MAUÁ DE TECNOLOGIA

- **Possibilidades de implementação de threads**
  - **Ativações do agendador**
    - É uma **estratégia** que visa **minimizar** as **chamadas** de **sistema** ao **kernel** quando uma **implementação** de **threads** no **espaço** de **usuário** é **utilizada**;
    - Nessa **estratégia**, o **kernel atribui** um certo número de **processadores virtuais** aos **processos**, que **criam threads** nesses **processadores**;
    - Dessa forma, o **kernel** "**sabe**" quando uma **thread** em algum **processo executou** uma **instrução bloqueante** e "**avisa**" o **agendador** das **threads** do **processo** que isso ocorreu (**upcall**);
    - Então, o **agendador** de **threads** do processo no espaço do usuário **bloqueia** a **thread** sem a **necessidade** de **realizar chamadas** de sistema ao kernel – aumentando assim o desempenho do sistema;
    - Essa estratégia também cobre o tratamento de interrupções:.

Implementações de threads

INSTITUTO MAUÁ DE TECNOLOGIA

- **Threads Pop-up**
  - Em **sistemas distribuídos** é muito comum a utilização de threads;
  - A **abordagem comum** para tratar **requisições** em um **servidor**, por exemplo, é **bloquear** um **processo** ou **thread** até o **momento** que **chega** uma **requisição** quando o **processo** ou **thread** é **desbloqueado** e então **trata** a **requisição**;
  - Uma **forma diferente** de **tratar requisições** é **criar** uma **thread nova** para **tratar** uma **nova requisição** que chegou;
  - Assim, uma **nova thread** é **criada** para **tratar** a **requisição**, que é **mais rápido** do que **alocar** alguma **thread existente** (restaurar registradores, pilha etc).



Utilizando threads

INSTITUTO MAUÁ DE TECNOLOGIA

- **Problemas ao se passar para programação multithreaded**
  - **Variáveis globais**: **duas threads acessando** a **mesma variável** global podem **gerar valores inconsistentes**. Uma **thread** `T1` obtém o **acessa** uma **variável global** `V`, mas no **momento** que vai **obter** seu **valor**, **outra thread** `T2` **acessa** e **altera** o **valor** de `V`. Então a **thread** `T1` **utiliza** um **valor inconsistente** em **relação** aquele **original**;
  - **Funções (de biblioteca) não reentrantes**: bibliotecas que não foram preparadas para serem executadas quando uma outra cópia está em execução;
  - **Sinais**: qual thread tratará um sinal ou alarme;
  - **Pilha**: diversas threads necessitam de diversas pilhas. Diferentemente da pilha de processo, que se esgotar o kernel aloca mais espaço, quando uma pilha de uma thread esgota, quem alocará mais espaço?

Referências bibliográficas

TANENBAUM, Andrew S. **Sistemas operacionais modernos**. 3. ed. São Paulo: Pearson, 2013. 653 p.