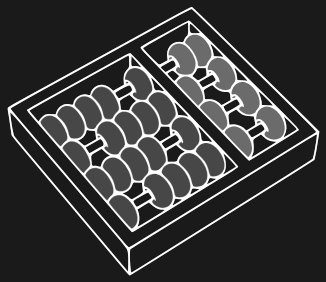


UNICAMP

Threads

Prof. Carlos A. Astudillo

castudillo@ic.unicamp.br

Sistemas Operacionais (MC504A)

2º Semestre 2023

Adaptado de Dave Eckhard

Recapitulando

- Cobrimos os capítulos 1–3 de OS:P+P
- Não é exatamente o mesmo, revisem suas anotações em aula
- Este tema é aproximadamente o capítulo 4 de OS:P+P

Objetivos da aula

- Threads (registradores escalonáveis)
- Por que threads?
- Diferentes versões
- Cancelamento
- Condições de corrida

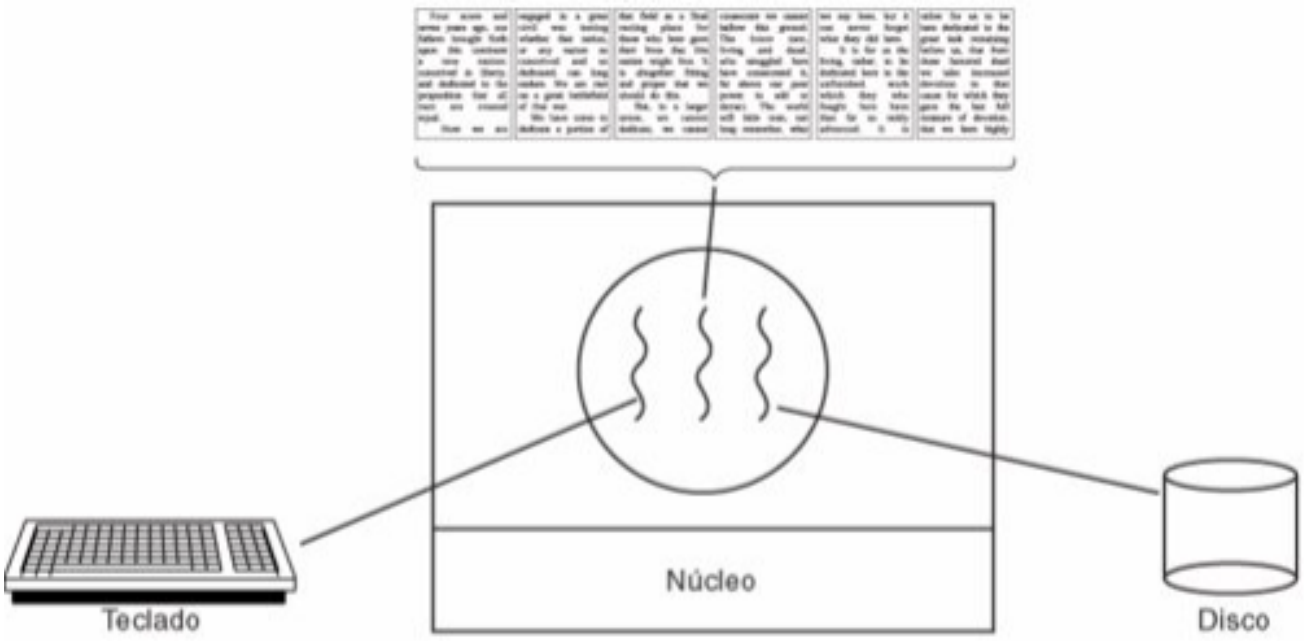
Threads ou Fluxos de Execução

Definição

- Uma *thread* é uma linha ou fluxo de execução de código que executa em paralelo com outras *threads* do mesmo processo, compartilhando seu espaço de endereçamento.
- Na prática uma *thread* é equivalente a um "mini-processo" dentro de um processo.
- Isto permite que várias ações sejam executadas em paralelo por um mesmo processo.

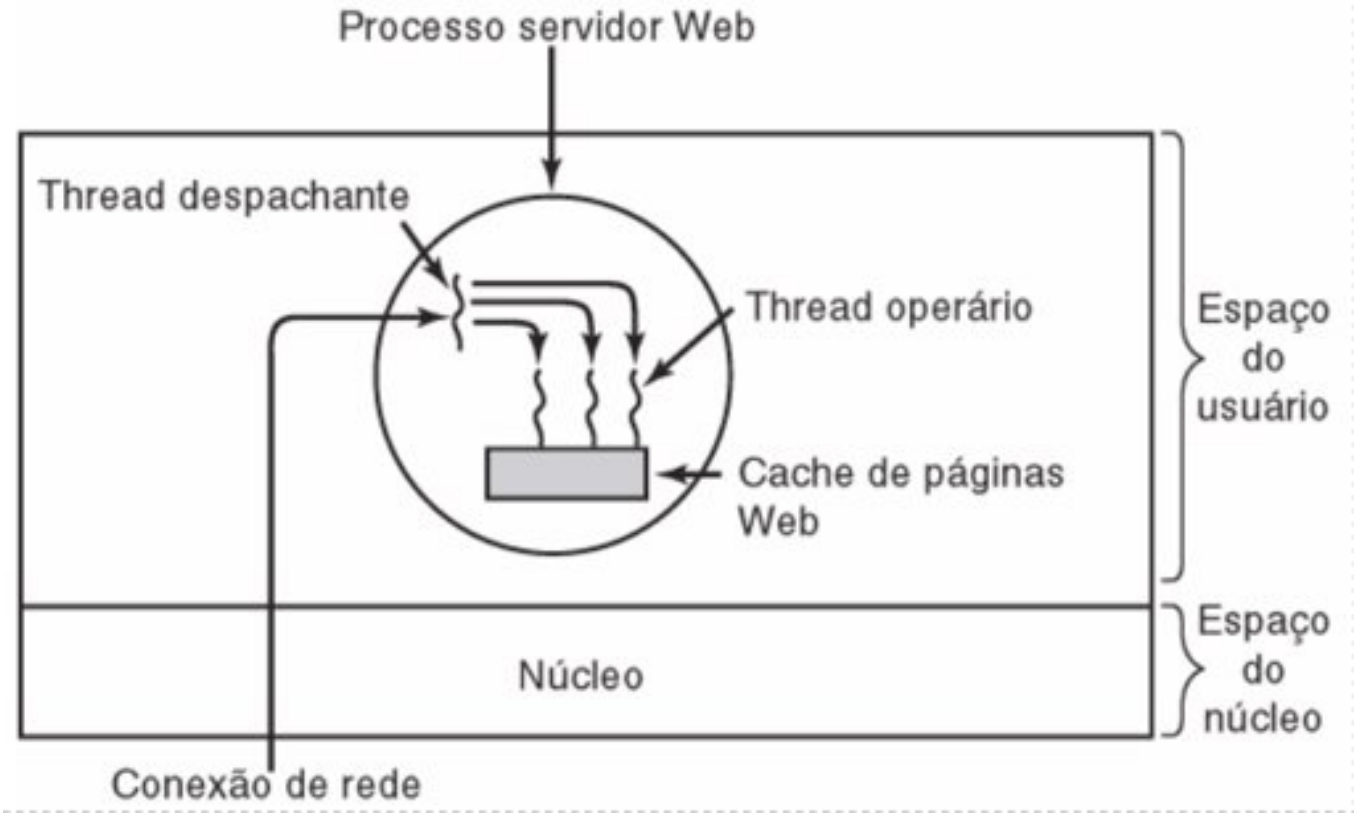
Aplicações de Threads

Processador de texto



Aplicações de Threads

Servidor eeb



Por que threads?

- Em um programa muitas vezes é necessário executar mais de uma atividade ao mesmo tempo.
  - ex.: aguardar a entrada de dados do usuário e reproduzir um som enquanto aguarda
- Uma thread é muito mais leve que um processo comum.
  - Ganho de performance na criação e destruição de threads se comparada a processos (10 a 100x)
- Quando uma aplicação tem atividade I/O bound e CPU bound as threads podem acelerar a execução, pois não concorrerão por recurso.

Por que threads?

- Compartilham o acesso às estruturas de dados
- Tempo de resposta (sensibilidade)
- Mais velocidade em processadores múltiplos

Acesso compartilhado a estruturas de dados

- Servidor de banco de dados para várias filiais de bancos
  - Verificar que múltiplas regras são seguidas
    - Balance de contas
    - Limite diário de retiros
    - etc.
  - Operações de múltiplas contas
  - Muitos acessos, cada um modifica uma fracção do banco de dados
- Servidor para um jogo de múltiplos jogadores
  - Muitos jogadores
  - Acesso (e atualização) do estado do mundo
    - Escanear múltiplos objetos
    - Atualizar um ou mais objetos

Acesso compartilhado a estruturas de dados

- Thread por jogador
  - Os objetos do jogo estão dentro de um mesmo espaço de memoria
  - Cada thread pode acessar e atualizar os objetos do jogo
  - Acesso compartilhado a objetos do sistema operacional (arquivos)
- Mudar entre threads é barato
  - Armazenar n registradores
  - Carregar n registradores

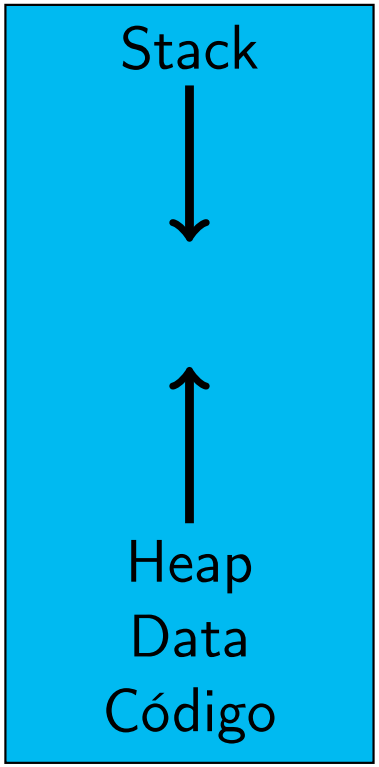
Tempo de resposta

- Cancelar uma ação vs. processamento massivo
- Botão cancelar vs. descomprimindo um arquivo gigante
  - Gerenciar o botão do mouse durante uma operação de 10 segundos
    - Mapear (x, y) para a área de "Cancelar"
    - Mudar a cor, animar o botão, fazer um som
    - Verificar que ao soltar do botão aconteça na área correta da tela
  - . . . sem que o descompressor entenda um click do mouse
  - E parar o descompressor é uma tarefa separada
    - Os threads permiten que o usuário possa registrar distintas intenções enquanto continuam executándo-se

Múltiplos processadores

- Mais CPUs não podem ajudar num processo de um thread só
- Desfocando cor em photoshop
  - Dividir a imagem em regiões
  - Um thread desfocando em cada região pela CPU
  - Pode (as vezes) dar um aumento de velocidade linear

Processo de um thread



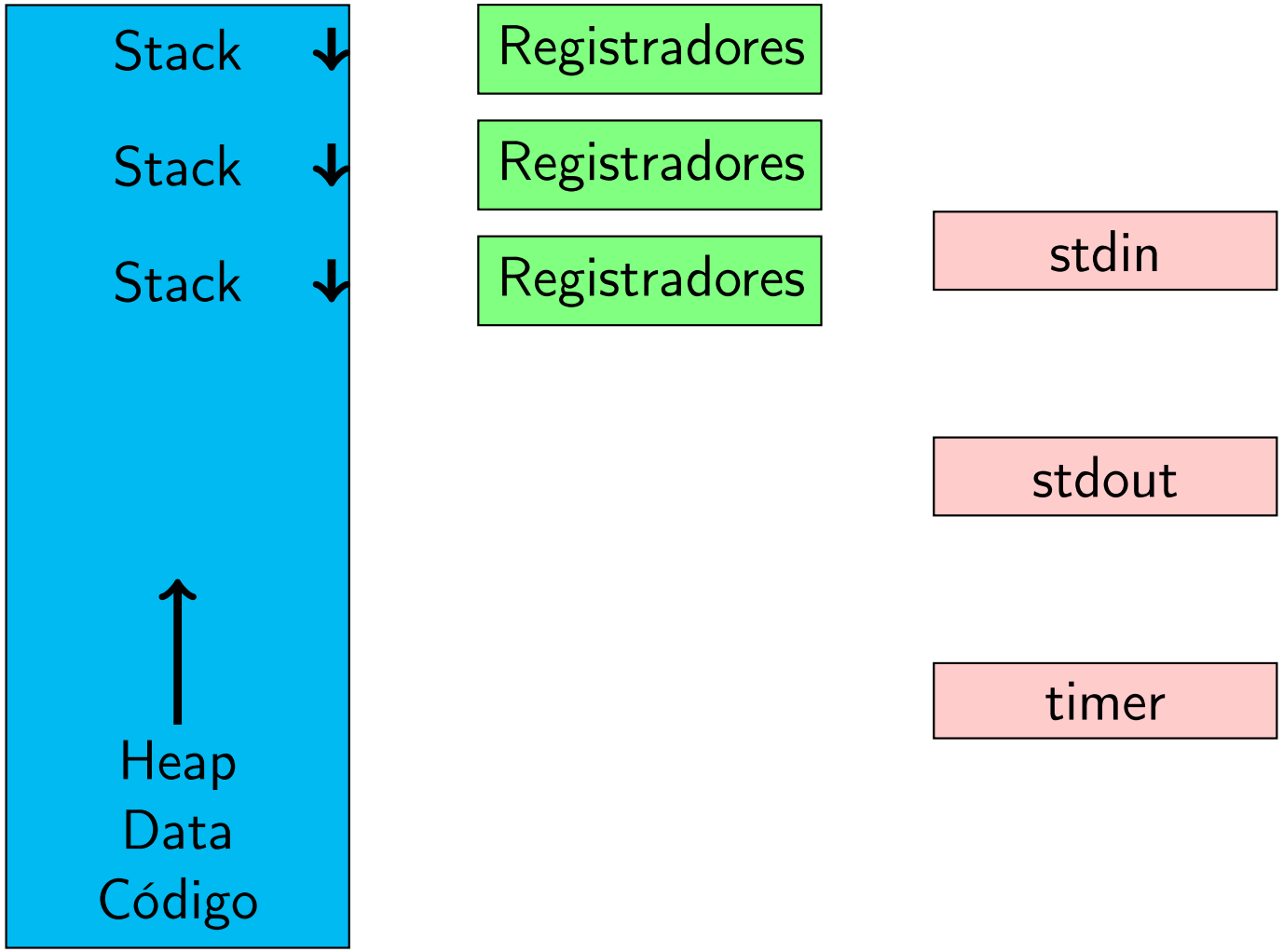
Registradores

stdin

stdout

timer

Processo de múltiplos threads



Em português?

(Para o exemplo anterior)

- Três stacks
  - Três conjuntos de variáveis locais
- Três conjuntos de registradores
  - Três ponteiros do stack
  - Três %eax, etc.
- Três escalonadores (*schedulers*)
- Três potenciais interações ruins
  - A/B, A/C, B/C
  - O padrão se complica se existem mais threads

Modelo de Thread

Processo: um espaço de endereço e uma única linha de controle
Threads: um espaço de endereço e múltiplas linhas de controle

- Modelo de Thread
  - Recursos particulares (PC, registradores, pilha)
  - Recursos compartilhados (espaço de endereço – variáveis globais, arquivos, etc)
- Modelo de Processo
  - Agrupamento de recursos (espaço de endereço com texto e dados do programa; arquivos abertos, processos filhos, tratadores de sinais, alarmes pendentes etc)
  - Execução

Múltiplas execuções no mesmo ambiente do processo – com certa independência entre as execuções

Tipos de threads

Terminologia: `threads:processo`

- Espaço de usuário $(N : 1)$
- Threads do kernel $(1 : 1)$
- Muitos a muitos $(M : N)$

Threads de usuário $(N : 1)$

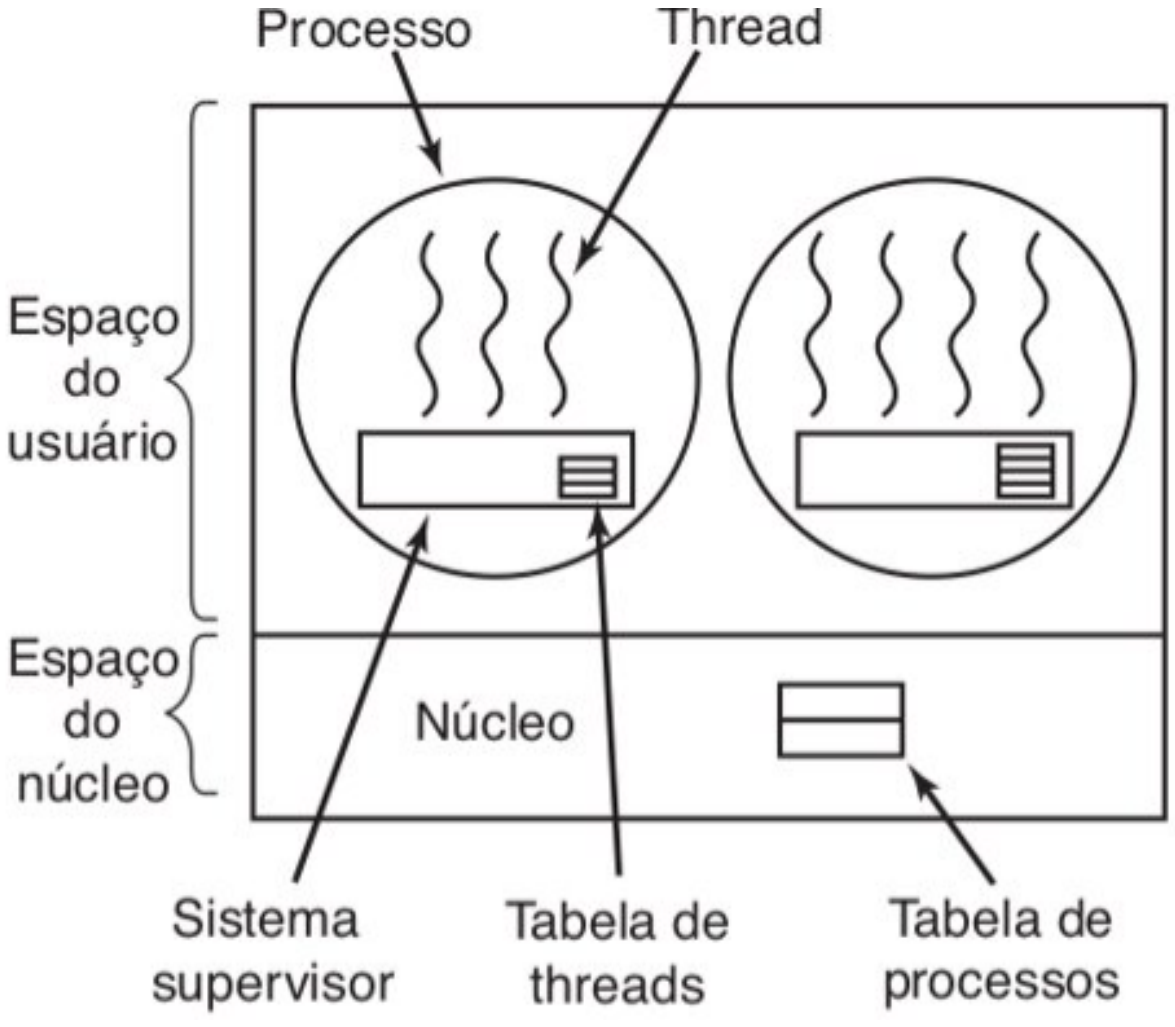
- Threads internos
  - A biblioteca de threads agrega os threads a um processo
  - A mudança de threads *só troca os registradores*
    - Um código simples em `asm`
    - Pode apenas chamar a `yield` (entrega a CPU, e se move ao final da lista)

Threads de usuário $(N : 1)$

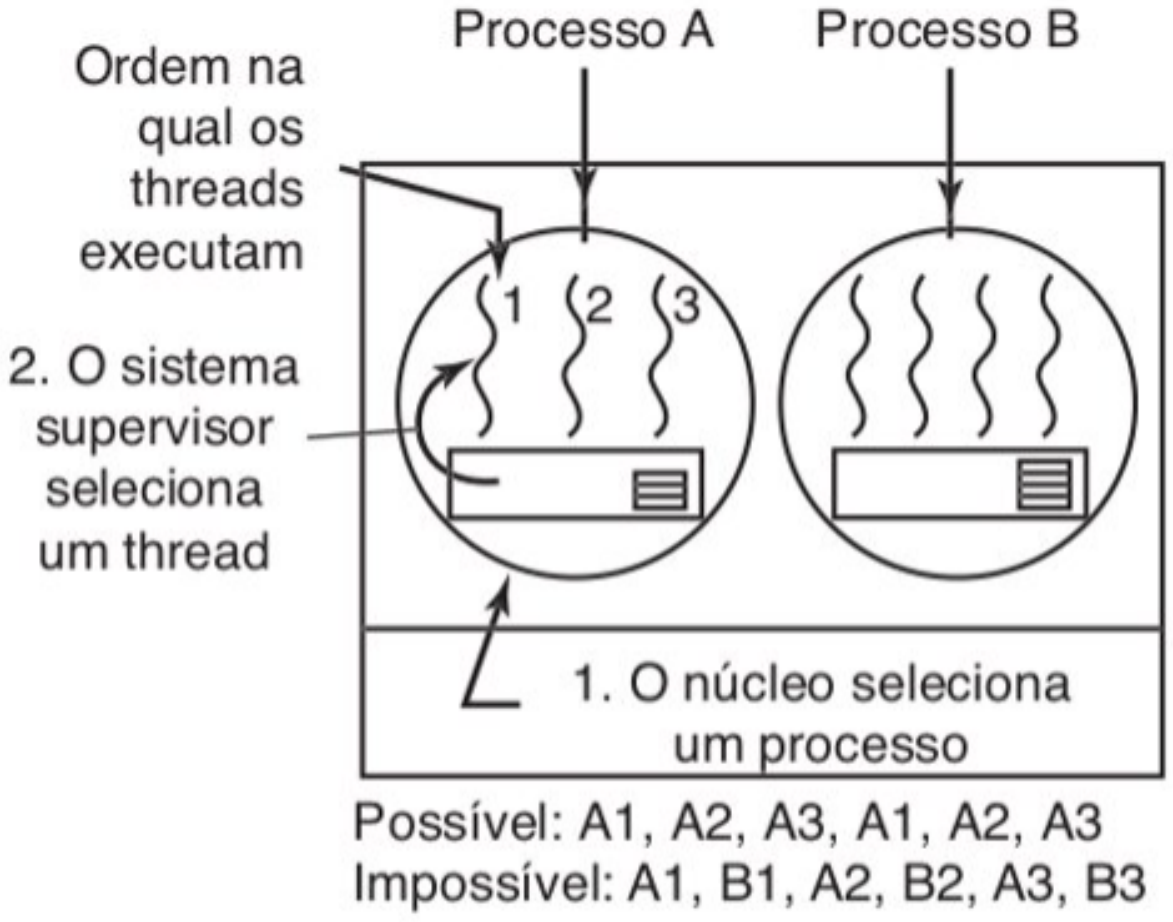
- 👍 Não precisa mudança no sistema operacional
- 👎 Qualquer chamada ao sistema bloqueia todos os threads
  - O processo faz uma chamada ao sistema
  - O kernel bloqueia o processo
  - Chamadas especiais que não bloqueiem podem ajudar

  Escalonado cooperativo: insuficiente, esquisito
  - Há que inserir chamadas manuais a `yield`
- 👎 Não pode ir mais rápido em máquinas de múltiplos processadores

Threads de usuário ($N : 1$)



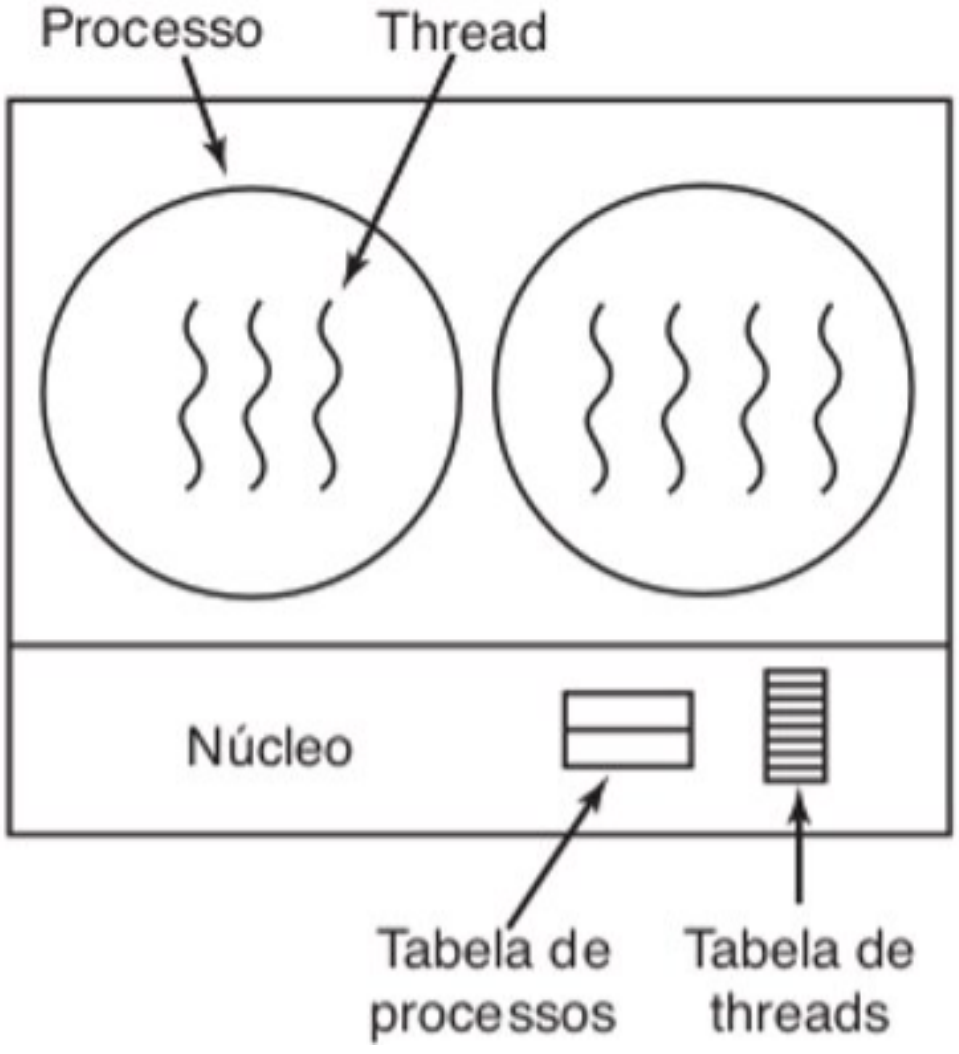
Threads de usuário $(N : 1)$



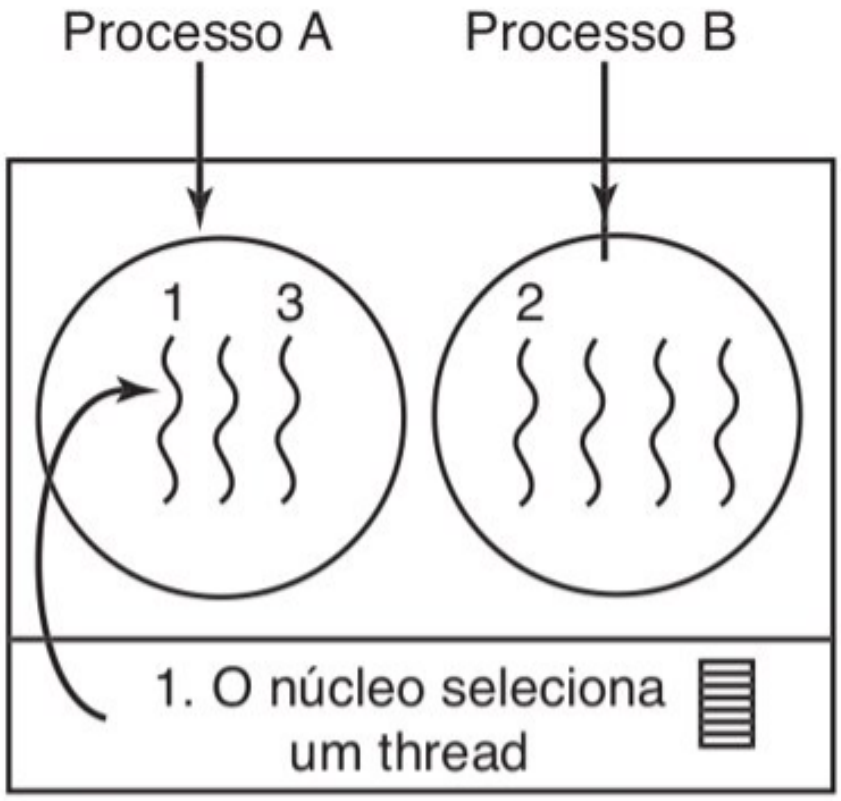
Threads do kernel (1 : 1)

- Threads suportados pelo sistema operacional
  - O sistema operacional conhece a correspondência thread-processo
  - As regiões de memoria são compartilhadas, e as referencias são contadas
- Cada thread é sagrado
  - O conjunto de registradores é gerenciado pelo kernel
  - Existe um stack do kernel quando o thread está executando código de kernel
  - Escalonamento real (desencadeada, *triggered*, pelo timer)
- Características
  - 👍 O programa se executa mais rápido em multiprocesador
  - 👍 Threads que monopolizam o CPU não obtém todo o tempo dele
  - 👎 As bibliotecas do espaço de usuário devem de re-escrever-se para que sejam seguras nestes threads
  - 👎 Requere mais memoria do kernel
    - 1 PCB $\Rightarrow$ 1 TCB + N tCB
    - 1 k-stack $\Rightarrow$ N k-stacks

Threads do kernel (1 : 1)



Threads do kernel (1 : 1)

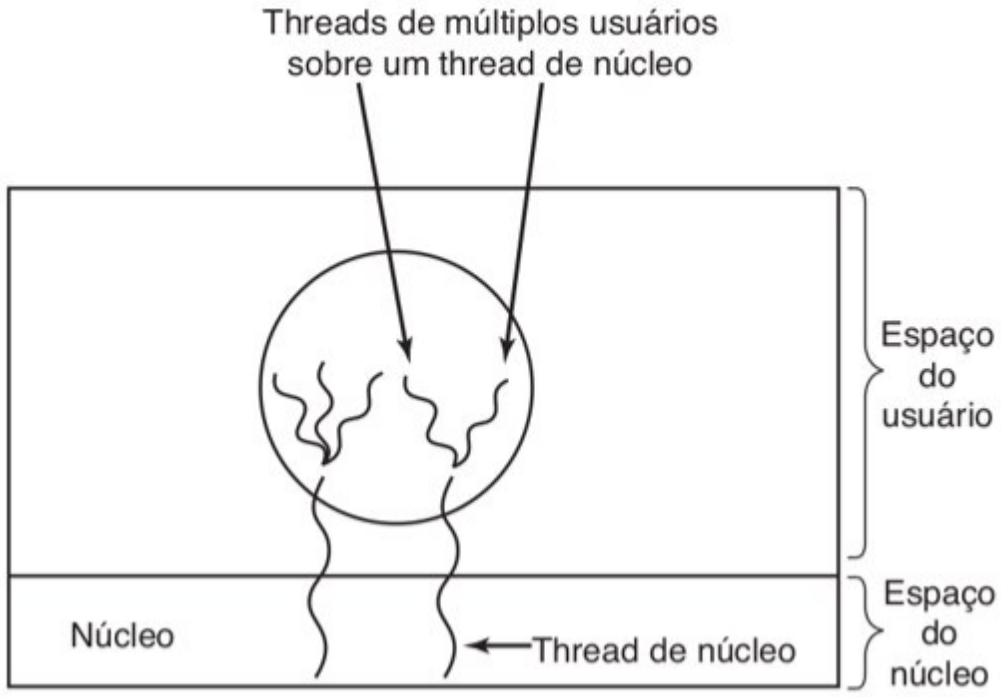


Possível: A1, A2, A3, A1, A2, A3
Também possível: A1, B1, A2, B2, A3, B3

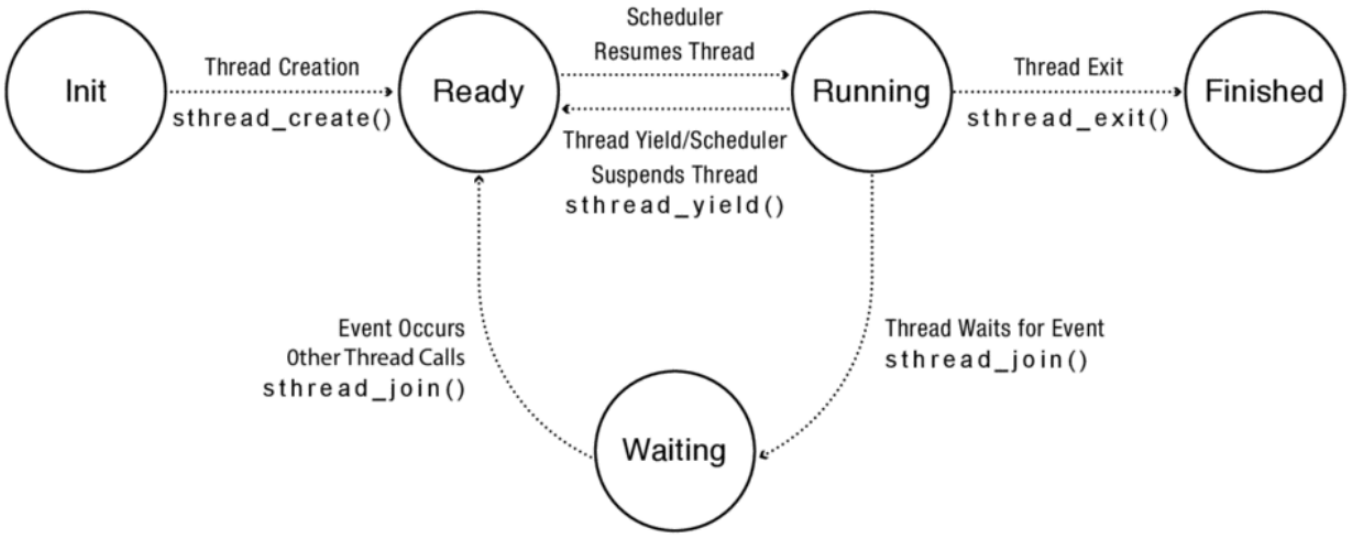
Muitos a muitos ($M : N$)

- É um terreno médio
  - O sistema operacional oferece threads de kernel
  - M threads de usuário compartilham N threads de kernel
- Padrões para compartilhar
  - Dedicado
    - O thread de usuário x é dono do thread do kernel y
  - Compartilhado
    - Um thread de kernel por cada CPU (hardware)
    - Cada thread de kernel executa o seguinte thread de usuário que seja executável
  - Muitas variações
- Características
  - Excelente quando o escalonador (*scheduler*) funciona como se espera

Muitos a muitos ($M : N$)



Ciclo de vida dos threads



Cancelamento de threads

- Cancelamento de threads
  - Quando?
  - Não queremos o resultado de este cálculo/processo/<*insere algo interessante aqui*>
  - Cancelamos a computação
- Dois tipos: assíncrono e diferido

Cancelamento de threads

Assíncrono

- Imediata
- Deter a execução agora
  - Executar 0 instruções mais (pelo menos no espaço de usuário)
  - Liberar o stack, os registradores
  - Não mais thread
- Difícil de recuperar recursos (garbage-collector), entenda-se por recursos: arquivos abertos, dispositivos, etc.
- Difícil de manter a consistência das estruturas de dados

Cancelamento de threads

Diferido

- Por favor parese . . .
- Quase que escrevemos: "*Prezado thread #42, por favor pare sua execução. Que saudade de você, o seguinte ciclo nos vemos. Beijos e abraços, o usuário.*"
- Os threads devem verificar-se para ser cancelados
- Ou devemos definir pontos de cancelamento seguros
  - Depois de chamar `close()` está bem que me pare

Olá Mundo

```
#include <stdio.h>
#include "thread.h"

static void go(int n);

#define NTHREADS 10
static thread_t threads[NTHREADS];

int main(int argc, char **argv) {
    int i;
    long exitValue;

    for (i = 0; i < NTHREADS; i++){
        thread_create(&(threads[i]), &go, i);
    }
    for (i = 0; i < NTHREADS; i++){
        exitValue = thread_join(threads[i]);
        printf("Thread %d returned with %ld\n",
                        i, exitValue);
    }
    printf("Main thread done.\n");
    return 0;
}

void go(int n) {
    printf("Hello from thread %d\n", n);
    thread_exit(100 + n);
    // Not reached
}
```

```
% ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

Condições de corrida

- O que pensamos do código?

```
ticket = next_ticket++; // 0 => 1
```

- O que acontece na realidade (geração de código[1])

```
ticket = temp = next_ticket; // 0
++temp; // 1 mas invisível
next_ticket = temp;
```

- Lembram das condições de corrida dos processos?
  - O que é uma condição de corrida?
  - Revisem as slides e notas da aula do kernel

[1] Vem a importância de aprender compiladores?

Ley de Murphy (para *threads*)

- O mundo arbitrariamente pode entrelaçar outra execução
  - Multiprocessador
    - N threads executando instruções **simultaneamente**
    - Mas é claro, os resultados podem estar entrelaçados
  - Processador único
    - Só um thread executando-se por vez
    - Mas N threads são executáveis, e o contador (timer) conta para zero
- O mundo decidirá a forma mais dolorosa de entrelaçar
  - ☹ Um num milhão, e acontece cada minuto

O que se espera

H_0		H_1	
`tkt = tmp = n_tkt;`	0		
`++tmp;`	1		
`n_tkt = tmp;`	1		
		`tkt = tmp = n_tkt;`	1
		`++tmp;`	2
		`n_tkt = tmp;`	2

- H_0 tem um ticket em 0
- H_1 tem um ticket em 1
- ☺ O resultado em `n_tkt` é 2, e nosso chefe está feliz

Mas! o que acontece...

H_0		H_1	
`tkt = tmp = n_tkt;`	0		
		`tkt = tmp = n_tkt;`	0
`++tmp;`	1		
		`++tmp;`	1
`n_tkt = tmp;`	1		
		`n_tkt = tmp;`	1

- H_0 tem um ticket em 0

  H_1 tem um ticket em 0, também

☹ O resultado em `n_tkt` é 1, e nosso chefe está "super feliz"

O que aconteceu?

- Cada thread fez algo razoável
  - ...assumindo que nenhum outro thread estava modificando esses objetos
  - ...quer dizer, assumindo exclussão mutua
- O mundo é cruel (se algo pode dar errado, isso vai dar errado)
  - Qualquer confusão no escalonador acontecerá tarde ou cedo
  - O que se espera que não aconteça, vai acontecer
  - O que não esperava acontecer, ...

O hack da shell-script #!

- O que é um shell script?
- É um arquivo de varias instruções (dependentes da shell)

```
#!/bin/sh
echo "Que bom é saber sistemas operacionais!"
sleep 10
exit 0
```

- Ou uma condição de corrida esperando acontecer . . .

O hack da shell-script #!

- O que é #!?
  - Um hack
  - Chamado shebang, hash-bang, temhbang, pound-bang, hash-pling[2]
- Quando dizemos
  `execl("/foo/script", "script", "arg1", 0);`
- O arquivo executável `/foo/script` começa
- Quando encontramos `#!/bin/sh`, o kernel re-escreve a chamada
  `execl("/bin/sh", "/foo/script", "script", "arg1", 0);`
- O shell faz `open("/foo/script", O_RDONLY, 0);`

[2] http://en.wikipedia.org/wiki/Shebang_%28Unix%29

A invenção do `setuid`

- Set user identity
- Patente U.S. #4 135 240
  - Dennis M. Ritchie
  - 16 de janeiro de 1979
- Conceito
  - Um programa é armazenado com certos privilegios de armazenamento
  - Quando se executa, tem duas identidades
    - A identidade do quem invoca
    - A identidade do programa mesmo
  - Pode mudar identidades a vontade
    - Abrir arquivos como o quem invocou
    - Abrir outros arquivos como o dono do programa

Exemplo de `setuid`: imprimir um arquivo

- Objetivo
  - Cada usuário pode enfileirar arquivos
  - Os usuários não podem excluir os arquivos de outros usuários
- Solução
  - O diretório das filas é do usuário `printer`
  - `setuid` como programa `enfileirar-arquivo`
    - Criar um arquivo de filas como o usuário `printer`
    - Copiar os dados do usuário (`user`) como o usuário `user`
  - `setuid` como programa `eliminar-arquivo`
    - Permite eliminar arquivos que um enfileirou
  - O usuário `printer` medeia o acesso à fila do usuário `user`

Condições de corrida

Processo 0	Processo 1
`ln -s /bin/lpr /tmp/lpr`	
	`run /tmp/lpr`
	`setuid` a usuário `printer`
	`start "/bin/sh /tmp/lpr ..."`
`rm /tmp/lpr`	
`ln -s /my/exploit /tmp/lpr`	
	`script = open("/tmp/lpr");`
	`execute /my/exploit`

O que aconteceu?

- A intenção: atribuir privilégios ao conteúdo do programa
- O que aconteceu na realidade?
  - Primeiro, o nome foi mapeado aos privilégios
    - nome $\Rightarrow$ arquivo, arquivo $\Rightarrow$ privilégios
  - Depois o nome do programa foi ligado a um arquivo diferente
  - Então, o nome foi mapeado aos conteúdos do arquivo
    - nome $\Rightarrow$ outro arquivo, outro arquivo $\Rightarrow$ outro conteúdo
- Como se resolve o problema?

Solucionando condições de corrida

- Analisar a sequencia de operações cuidadosamente
- Encontrar uma subsequencia que deve ser ininterrompida
  - Seção crítica
- Utilizamos um mecanismo de sincronização
  - Continuará ...

Pontos importantes

- Threads: o que? e por que?
- Tipos de threads
- Razones dos distintos modos de threads
- Condições de corrida
  - Certifique-se de entender esta parte

Leituras adicionais

- `setuid` demysitified
  - Hao Chen, David Wagner, Drew Dean
  - `http://www.cs.berkeley.edu/~daw/papers/setuid-usenix02.pdf`
- Cancel button problem
  - Attentiveness: Reactivity at scale
  - Gregory S. Hartman
  - `http://repository.cmu.edu/dissertations/15/`