

Sistemas Operacionais I

Profa. Kalinka Regina Lucas Jaquie Castelo Branco
kalinka@icmc.usp.br

Universidade de São Paulo

Setembro de 2020

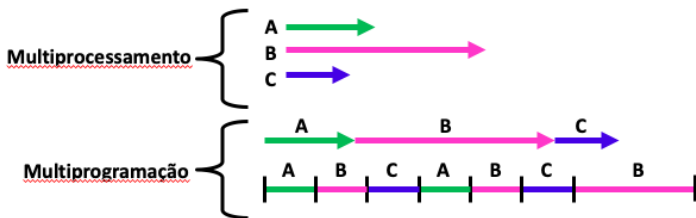
- Definição de antes: um único contexto de execução único - Descreve sua representação
- Fornece a abstração de: Uma única sequência de execução que representa uma tarefa programada separadamente - Também é uma definição válida!
- *Threads* são um mecanismo de simultaneidade
- Proteção é um conceito ortogonal - Um domínio de proteção pode conter um ou vários *threads*.

- Os sistemas operacionais devem lidar com várias coisas ao mesmo tempo (VCMT) - Processos, interrupções, manutenção do sistema em segundo plano
- Os servidores em rede devem lidar com VCMT - Múltiplas conexões tratadas simultaneamente
- Programas paralelos devem lidar com VCMT - Para alcançar melhor desempenho
- Programas com interface de usuário geralmente devem lidar com VCMT - Para alcançar a capacidade de resposta do usuário durante a computação
- Os programas ligados à rede e ao disco devem lidar com VCMT - Para ocultar a latência da rede / disco; Etapas de sequência no acesso ou comunicação

- *Threads* são uma unidade de simultaneidade fornecida pelo sistema operacional
- Cada *thread* pode representar uma coisa ou uma tarefa

- Simultaneidade trata de lidar com várias coisas ao mesmo tempo (VCMT)
- Paralelismo é fazer várias coisas simultaneamente
- Exemplo: dois *threads* em um sistema de núcleo único
 - ... executar simultaneamente ...
 - ... mas não em paralelo
- Cada *thread* lida ou gerencia uma coisa ou tarefa separada ...
- Mas essas tarefas não são necessariamente executadas simultaneamente!

- Multiprocessamento: vários núcleos
- Multiprogramação: vários trabalhos/processos
- *Multithreading*: vários *threads* / processos
- O que significa executar dois *threads* simultaneamente?
 - O **Escalonador** é livre para executar *threads* em qualquer ordem e intercalação



- Imagine o seguinte programa

```
main() {
    ComputePI("pi.txt");
    PrintClassList("classlist.txt");
}
```

- Qual é o comportamento aqui?

- Imagine o seguinte programa

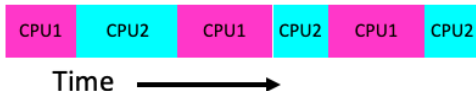
```
main() {
    ComputePI("pi.txt");
    PrintClassList("classlist.txt");
}
```

- Qual é o comportamento aqui?
- O programa nunca imprimiria a lista de aulas
- Por quê? ComputePI nunca terminaria

- Versão do programa com *threads* (sintaxe flexível):

```
main() {
    create_thread(ComputePI, "pi.txt");
    create_thread(PrintClassList, "classlist.txt");
}
```

- *create_thread*: gera um novo *thread* executando o procedimento fornecido
- Deve se comportar como se outra CPU estivesse executando o procedimento dado
- Agora, você realmente veria a lista de classes



De volta aos números que todos deveriam saber de Jeff Dean

Lidar com E/S em thread separadas, evita bloquear outros processos

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

- *Thread* está em um dos três estados a seguir:
 - EXECUTANDO - executando
 - PRONTO - elegível para execução, mas não em execução no momento
 - BLOQUEADO - inelegível para execução
- Se um *thread* está esperando que uma E/S termine, o SO o marca como BLOQUEADO
- Quando a E/S finalmente termina, o sistema operacional o marca como PRONTO

- Versão do programa com *threads* (sintaxe flexível):

```
main() {
    create_thread(ReadLargeFile, "pi.txt");
    create_thread(RenderUserInterface);
}
```

- Qual é o comportamento aqui?
- Ainda responde à entrada do usuário
- Ao ler o arquivo em segundo plano

- Você sabe como compilar um programa C e executar o executável - Isso cria um processo que está executando aquele programa
- Inicialmente, este novo processo tem um *thread* em seu próprio espaço de endereço - Com código, globais, etc. conforme especificado no executável

P: Como podemos fazer um processo *multithread*?

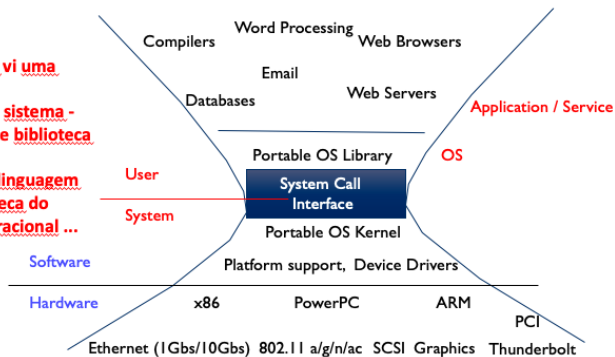
- Você sabe como compilar um programa C e executar o executável - Isso cria um processo que está executando aquele programa
- Inicialmente, este novo processo tem um *thread* em seu próprio espaço de endereço - Com código, globais, etc. conforme especificado no executável

P: Como podemos fazer um processo *multithread*?

R: Assim que o processo é iniciado, ele emite chamadas de sistema para criar novos *threads* Esses novos *threads* são parte do processo: eles compartilham seu espaço de endereço.

"Mas, eu nunca vi uma syscall!"

- **Chamada de sistema - problemas de biblioteca de SO**
- **Runtime da linguagem usa a biblioteca do sistema operacional ...**



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

- thread é criada executando *start_routine* com *arg* como seu único argumento.
- return é implícito a chamada *pthread_exit*

```
void pthread_exit(void *value_ptr);
```

- Termina thread e torna *value_ptr* disponível para qualquer adesão bem sucedida.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

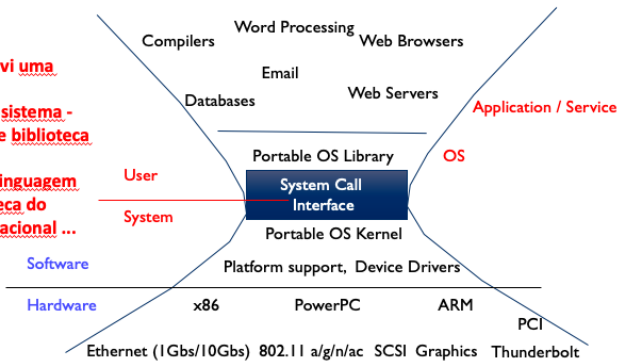
- Suspende a execução de uma thread chamada *at*; e que a thread alvo termine.
- Com o retorno de um valor não NULL *value_ptr* o valor é passado para *pthread_exit()* pela thread terminando e torna disponível a localização referenciada por *value_ptr*.

[man pthread](#)

<https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html>

"Mas, eu nunca vi uma syscall!"

- Chamada de sistema - problemas de biblioteca de SO
- Runtime da linguagem usa a biblioteca do sistema operacional ...



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

- thread é criada executando `start_routine` com `arg` como seu único argumento.
- return é implícito a chamada `pthread_exit`

```
void pthread_exit(void *value_ptr);
```

- Termina thread e torna `value_ptr` disponível para qualquer adesão bem sucedida.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- Suspende a execução de uma thread chamada `at` e que a thread alvo termine.
- Com o retorno de um valor não NULL `value_ptr` o valor é passado para `pthread_exit()` pela thread terminando e torna disponível a localização referenciada por `value_ptr`.

[man pthread](#)

<https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html>

- O que acontece quando `pthread_create(...)` é chamado em um processo?

Library:

```
int pthread_create(...) {
    Faça algo normal como uma fn..

    asm code ... syscall # into %eax
    put args into registers %ebx, ...
    special trap instruction
```

Kernel:

```
get args from regs
dispatch to system func
Do the work to spawn the new thread
Store return value in %eax
```

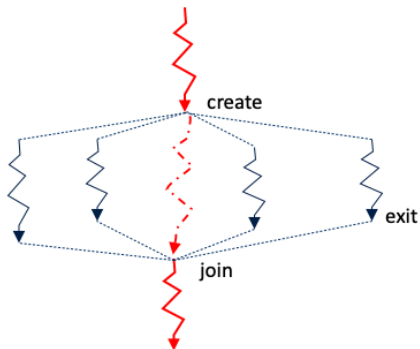
```
get return values from regs
Faça algo normal mais um pouco como uma fn..
};
```

Exemplo: O que há em uma consulta?

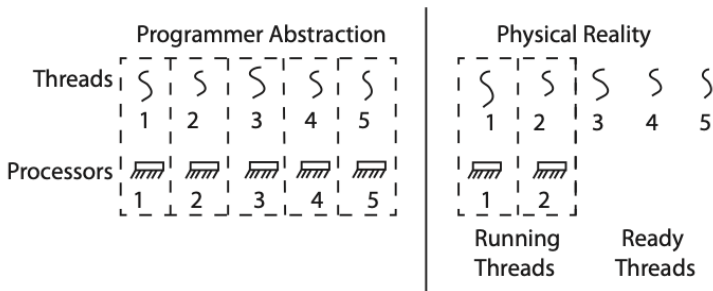
Sistemas
Operacionais

|

Profa.
Kalinka
Branco

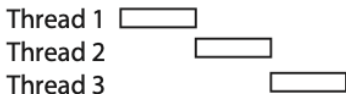


- O thread principal cria (bifurca - *fork*) uma coleção de sub-threads, passando-os argumentos para trabalhar ...
- ... E depois se junta (*join*) a eles, coletando resultados.



- Ilusão: número infinito de processadores
- Realidade: *Threads* executam com “velocidade” variável
- Os programas devem ser projetados para funcionar com qualquer programação.

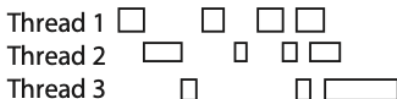
Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
$x = x + 1;$	$x = x + 1;$	$x = x + 1$	$x = x + 1$
$y = y + x;$	$y = y + x;$	$y = y + x$
$z = x + 5y;$	$z = x + 5y;$	thread is suspended
.	.	other thread(s) run	thread is suspended
.	.	thread is resumed	other thread(s) run
.	thread is resumed
		$y = y + x$
		$z = x + 5y$	$z = x + 5y$



a) One execution



b) Another execution



c) Another execution

- Não determinismo:
 - O escalonador pode executar *threads* em qualquer ordem
 - O escalonador pode alternar tópicos a qualquer momento
 - Isso pode tornar o teste muito difícil
- Tópicos Independentes
 - Nenhum estado compartilhado com outros tópicos
 - Condições determinísticas e reproduzíveis
- Tópicos de cooperação
 - Estado compartilhado entre vários tópicos
- **Meta: Correção por Projeto**

Continuemos com **Comunicação** entre Processos ...