FACULDADE
SANTA TEREZINHA

SISTEMAS OPERACIONAIS

PROF CARLOS WAGNER
FACULDADE CEST

UNIDADE II – GERENCIAMENTO DE PROCESSOS (16h)

2.1 Modelos de processos

2.2 Escalonamento

2.3 Sincronização

2.4 Impasses

2.5 Gerenciamento de processos

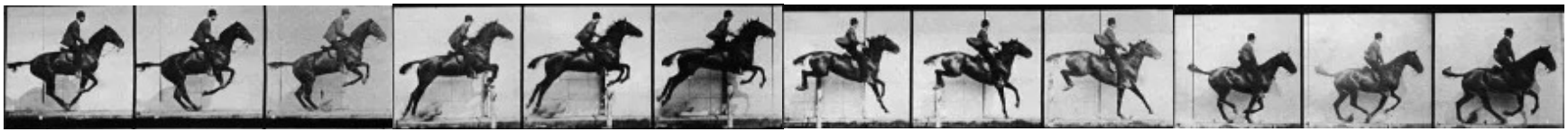
Conceitos

- Um processo é basicamente um programa em execução (TANEMBAUM, 2003)
- Informalmente, um processo é um programa em execção (SILBERCHATZ, 2004)
- Um programa é uma entidade inanimada; somente quando um processador lhe ‘sopra vida’ é que ele se toma a entidade ativa que chamamos de processo (DEITEL, 2005).

Modelo de Processos

- Em qualquer sistema de multiprogramação, a CPU muda de um processo para outro rapidamente, executando cada um por alguns milissegundos Em qualquer dado instante, a CPU está executando apenas um processo, mas em 1 segundo pode executar vários processos diferentes, o que dá a ilusão de paralelismo. Chamamos isso de pseudoparalelismo.
- Projetistas de sistemas operacionais desenvolveram o modelo conceitual de processos sequenciais para facilitar o suporte ao paralelismo

Modelo de Processos



https://www.youtube.com/watch?v=FYKZif9ooxs

Partes de um processo

Geralmente um processo é composto por várias partes;

Seção de texto	o código do programa
Contador do programa	O valor da atividade em curso e valores dos registradores do processador
Pilha	Dados temporários (parâmetros de métodos, endereços de retorno e variáveis locais)
Seção de dados	Contém as variáveis globais

Estas partes consistem no contexto do processo

Execução de um Processo

- Quatro eventos podem iniciar um processo:
  - Início do sistema;
  - Execução de uma chamada ao sistema de criação de processo por um processo em execução;
  - Uma requisição do usuário para criar um novo processo;
  - Início de um job em lote.

Execução de um Processo

- Dependendo de como o sistema operacional gerencia os processos, estes podem ser basicamente de 4 modelos:
  - Modelo dois estados;
  - Modelo três estados;
  - Modelo cinco estados;
  - Modelo seis estados;

Modelo de dois estados

- O modelo mais simples consiste apenas dos dois estados abaixo:
  - **Executando** – O processo está sendo executado
  - **Não Executando** – O estado está esperando para ser executado

E é só isso! Dois estados!

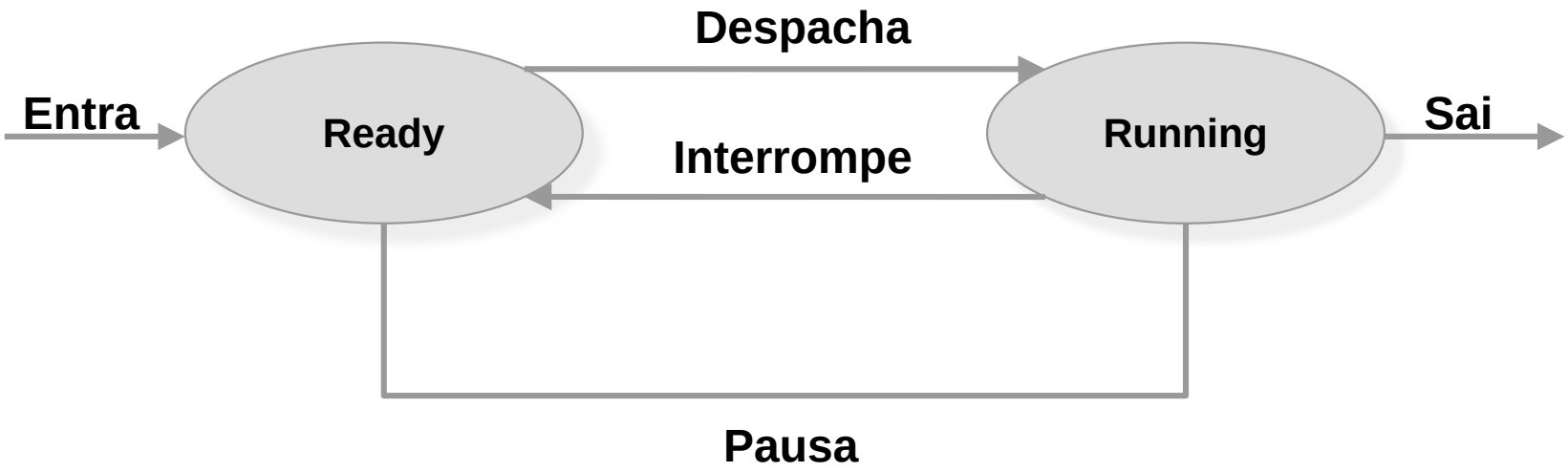
Fonte: https://freebiehive.com/khaby-lame-png/

2.1 Modelos de Processos

Modelo de dois estados

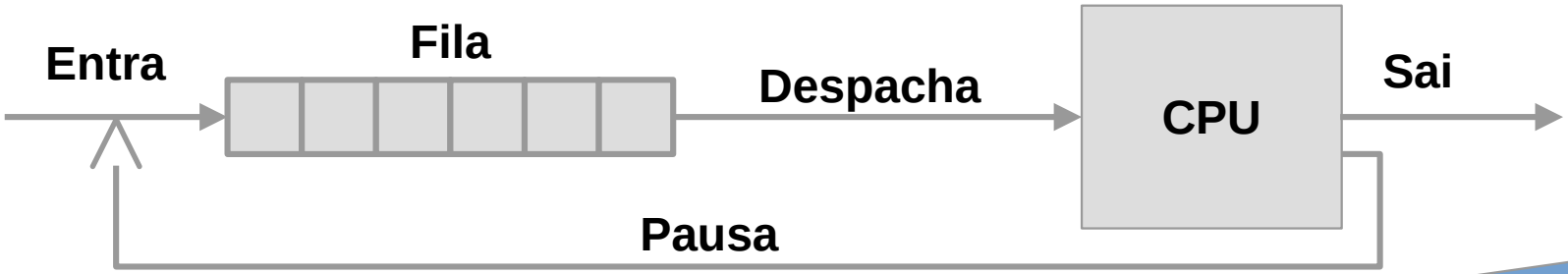
- Um processo de dois estados pode ser criado a qualquer tempo não importa se outro processo está sendo executado ou não.
- Primeiro, quando o S.O. cria um novo processo, ele também cria um PCB para o processo então o processo pode entrar no sistema em um estado de não execução. O sistema sabe e algum processo sai/deixa o sistema.
- Uma vez que o processo que está sendo executado atualmente será interrompido ou quebrado e o monitor do S.O. (um programa que chaveia o processador de um processo para outro), executará outro processo.
- Agora o “ex processo” (ou processo interrompido) é movido para o estado não executando e um dos outros processos se move para o estado executando e quando termina, sai do sistema.

Modelo de dois estados



Modelo de dois estados

Os processos que não estão sendo executados são mantidos numa espécie de fila, e esperam sua vez para executar. Na figura abaixo, há uma fila única na qual a seta "↑" é um ponteiro para o PCB (bloco de controle de processo), um bloco onde são armazenadas informações necessárias para a execução de cada processo.



Fonte: https://www.geeksforgeeks.org/two-state-process-model-in-operating-system/

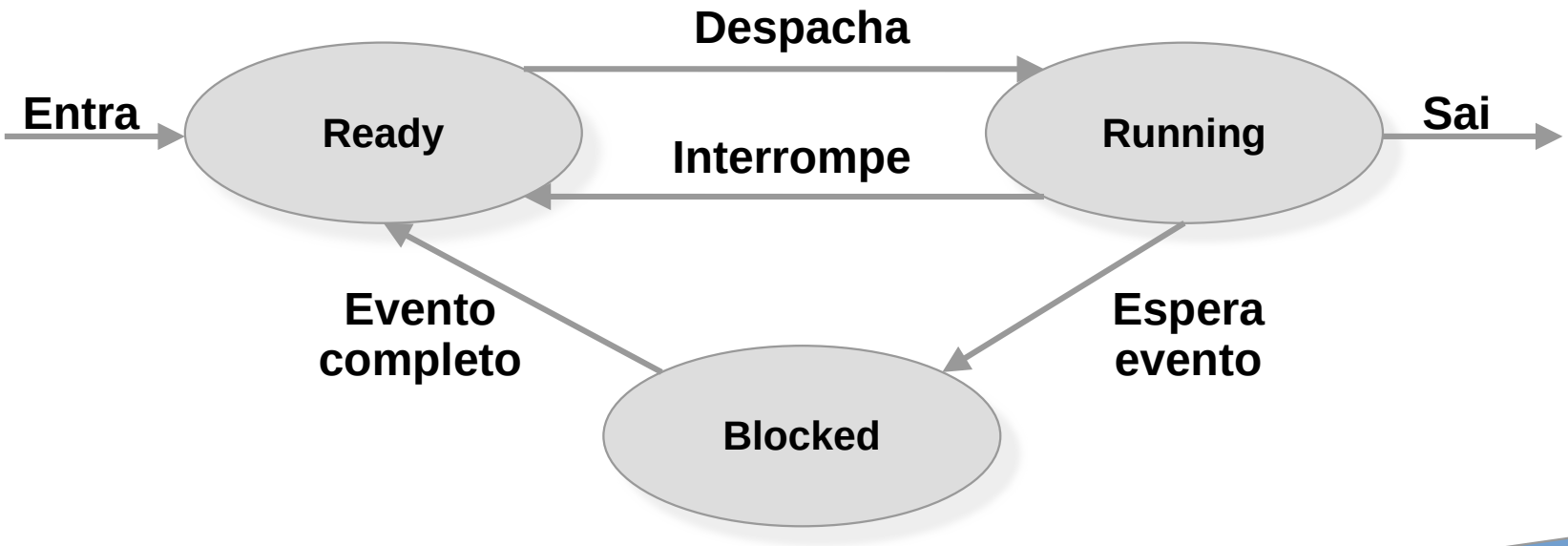
Modelo de três estados

O modelo de três estados considera a situação em que o processo esteja bloqueado, aguardando a conclusão de algum evento como uma operação de E/S

Dessa forma temos os seguintes estados:

- **Estado Pronto (Ready State)** – Um estado no qual um processo está pronto e esperando para sua execução
- **Estado Bloqueado (Blocked State)** – Um estado onde um processo não executa até ou a menos que um evento de processo ocorra, como a conclusão de uma operação de E/S.
- **Estado Executando (Running State)** – Um estado onde o o processo está sendo executado.

Modelo de três estados



2.1 Modelos de Processos

Modelo de três estados

O modelo de três estados permite a transição entre os estado do processo como segue.

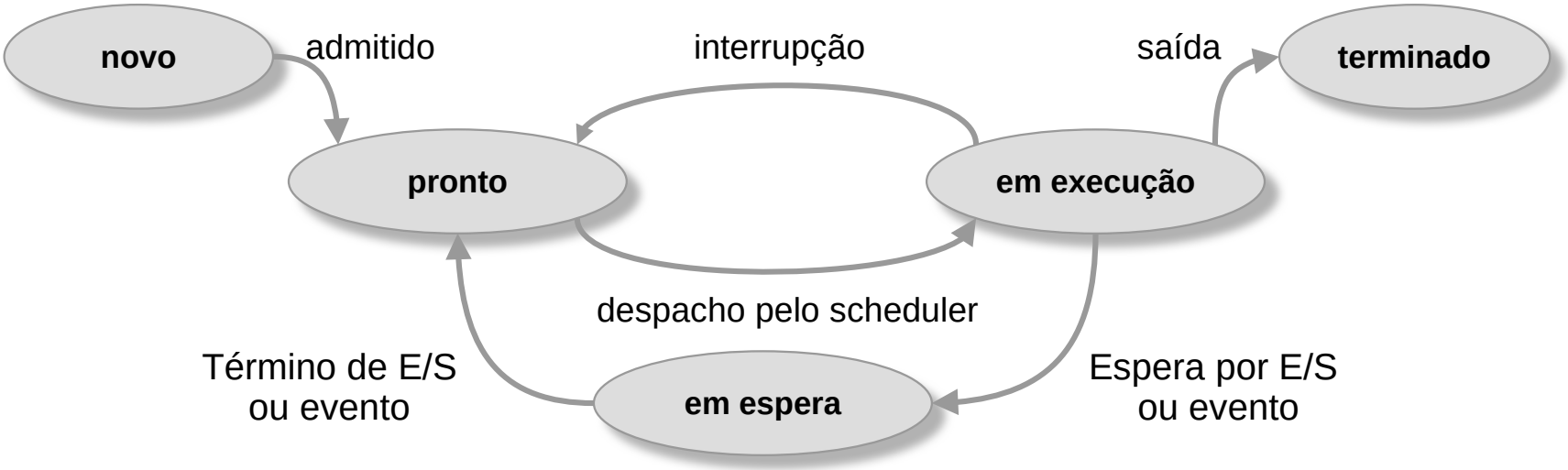
- **Ready -> Running** – Um processo muda de pronto para executando quando o S.O. seleciona um novo processo para executar e o sistema seleciona apenas um processo para executar no estado pronto
- **Running -> Ready** – Um processo passa do estado de execução para o estado pronto quando a condição de tempo limite é atendida ou outra pode ser devido a problemas de prioridade que foram atribuídos aos processos por um sistema operacional e um processo com prioridade mais alta entra no sistema. Essa transição é tratada por um agendador de processos.
- **Running -> Blocked/Waiting** – Um processo passa do estado de execução para o estado bloqueado quando a demanda do processo não é atendida, como um processo que exige maior memória ou exige alguns outros recursos que não estão disponíveis e, portanto, permite a execução do próximo processo em estado pronto. Essa transição também é tratada por um agendador de processos.
- **Blocked/Waiting -> Ready** – Um processo passa do estado bloqueado para o estado pronto por meio de um sinal do gerenciador de dispositivos de entrada ou saída de que a solicitação fornecida foi atendida e o processo pode continuar para execução.
- **Running -> Exit** – Um processo termina depois que é completamente executado.

O modelo de 5 estados

O modelo de 5 estados adiciona dois novos estados “não executando”: **pronto** e **em espera:**

- **Novo:** Um processo foi criado, mas ainda não foi admitido pelo S.O. para execução;
- **Pronto:** O processo está preparado para ser designado a um processador:
- **Em execução:** As instruções estão sendo executadas;
- **Em espera:** O processo está esperando pela ocorrência de algum evento (aguardando E/S ou algum sinal);
- **Terminado:** O processo terminou sua execução ou foi interrompido pelo S.O..

Execução de um Processo de 5 estados



Fonte: SILBERCHARTZ, 2004

Mudanças de estados do modelo de cinco estados

Null -> New: Um novo processo é criado para a execução;

New -> Ready: O sistema mudará o processo do estado novo para pronto e agora ele está pronto para execução. Aqui o sisetma pode indicar um limite tal que múltiplos processos não podem ocorrer, caso contrário podem ocorrer problemas de desempenho.

Ready -> Running: Agora o S.O. seleciopna um processo para executar e o sistema escolhe apenas um processo que está no estador pronto para execução.

Running -> Exit: O sistema termina o processo se o processo indica que está completado ou se ele foi interrompido.

2.1 Modelos de Processos

Mudanças de estados do modelo de cinco estados

Running -> Ready: A razão pela qual essa transição ocorre é quando o processo em execução atinge seu tempo máximo de execução para execução ininterrupta. Um exemplo disso pode ser um processo executado em segundo plano que realiza alguma manutenção ou outras funções periodicamente.

Running -> Blocked: Um processo é colocado no estado bloqueado se solicitar algo que está esperando. Da mesma forma, um processo pode solicitar alguns recursos que podem não estar disponíveis no momento ou pode estar aguardando uma operação de E/S ou aguardando a conclusão de algum outro processo antes que o processo possa continuar.

Blocked -> Ready: Um processo passa do estado bloqueado para o estado pronto quando ocorre o evento pelo qual estava aguardando.

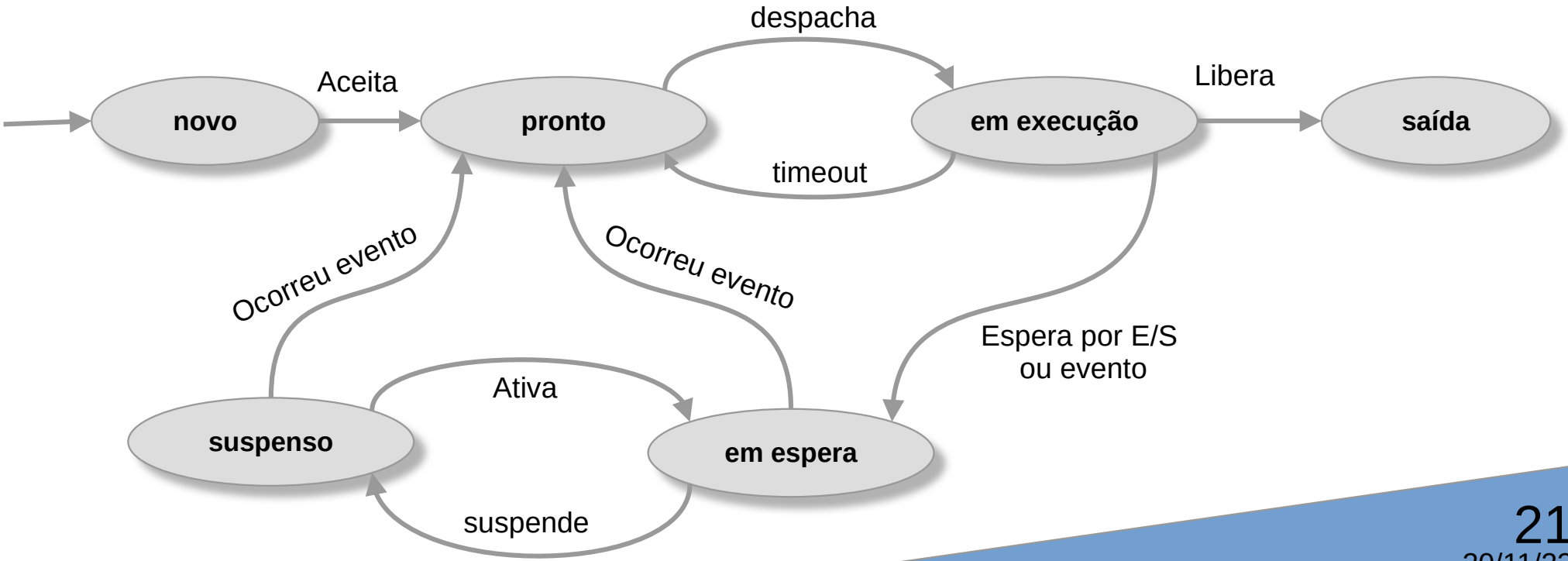
Ready -> Exit: Esta transição só pode existir em alguns casos porque, em alguns sistemas, um pai pode encerrar o processo filho a qualquer momento.

O modelo de 6 estados

Devido a cresncente complexidade dos processos e o fato de muitos sistemas não possuírem memória RAM suficiente para manter o gerenciamento apenas na memória principal, o modelo de 6 estados adiciona o estado **suspenso**, que é usado somente quando a fila de bloquados/esperando está cheia.

Suponha que, como discutimos no exemplo anterior, muitos processos intensivos de E/S sejam executados ao mesmo tempo e nossa fila de bloqueios/espera seja preenchida nesse momento, moveremos todos os outros processos para a memória secundária e carregaremos sempre que necessário ou quando o espaço fica vazio usando o conceito de memória virtual.

Execução de um Processo de 6 estados



2.1 Modelos de Processos

Mudanças de estados do modelo de seis estados

1. **Null → New:** Um novo processo é criado para execução.
2. **New → Ready:** O processo recém-criado que está pronto para ser executado na CPU e aguardando sua liberação é movido para cá no estado pronto. Pode haver vários processos nesses estados ao mesmo tempo.
3. **Ready → Running:** O processo Selecionado é movido para o estado Running onde receberá a CPU para executar suas tarefas. Pode haver no máximo um processo no estado Running por vez.
4. **Running → Exit:** O processo cuja execução for encerrada será encerrado e movido para o estado Exit.
5. **Running → Ready:** Quando o processo atingir seu limite máximo de tempo de execução ou quando um processo de alta prioridade chegar para execução, o processo atualmente presente no estado de execução será movido para o estado pronto.

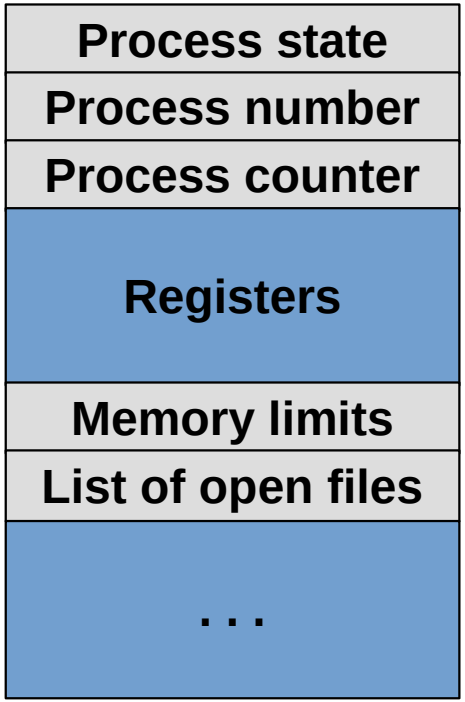
2.1 Modelos de Processos

Mudanças de estados do modelo de seis estados

6. **Running → Blocked:** Quando o processo está aguardando a ocorrência de algum evento, o processo será movido do estado Em execução para o estado Bloqueado.
7. **Blocked → Ready:** Um processo retornará ao estado pronto quando ocorrer o evento que ele estava esperando.
8. **Ready → Exit:** Isso só acontecerá quando o processo pai do processo atual for encerrado ou solicitar explicitamente o encerramento do processo filho.
9. **Blocked → Suspend:** Quando a fila Bloqueada for preenchida com os processos, alguns dos processos da fila bloqueada serão movidos para o estado Suspenso. O estado suspenso existe na memória secundária usando o conceito de memória virtual.
10. **Suspend → Blocked:** Quando o Espaço ficar disponível para o estado Bloqueado, o processo que foi colocado no estado Suspenso será movido de volta para o estado Bloqueado.
11. **Suspend → Ready:** Quando o evento ocorrer para o processo que estava aguardando no estado suspenso na memória secundária, ele será movido diretamente para o estado Pronto.

Execução de um Processo

- O sistema operacional gerencia o **bloco de controle de processo (PCB – Process Control Block),** uma estrutura especial que armazena informações importantes de cada processo. Tais informações incluem:
  - **Estado do processo:** pode ser novo, pronto, em execução, em espera, parado, etc;
  - **Contador de programa:** indica o endereço da próxima instrução a ser executada pelo processo;
  - **Registradores da CPU:** valores dos acumuladores, registradores índices, flags de verificação, além de valores de interrupções para que o processo continue corretamente quando retomar o controle da CPU.



PCB

Baseado em TANENBAUM, 2003

PCBs de alguns S.O.

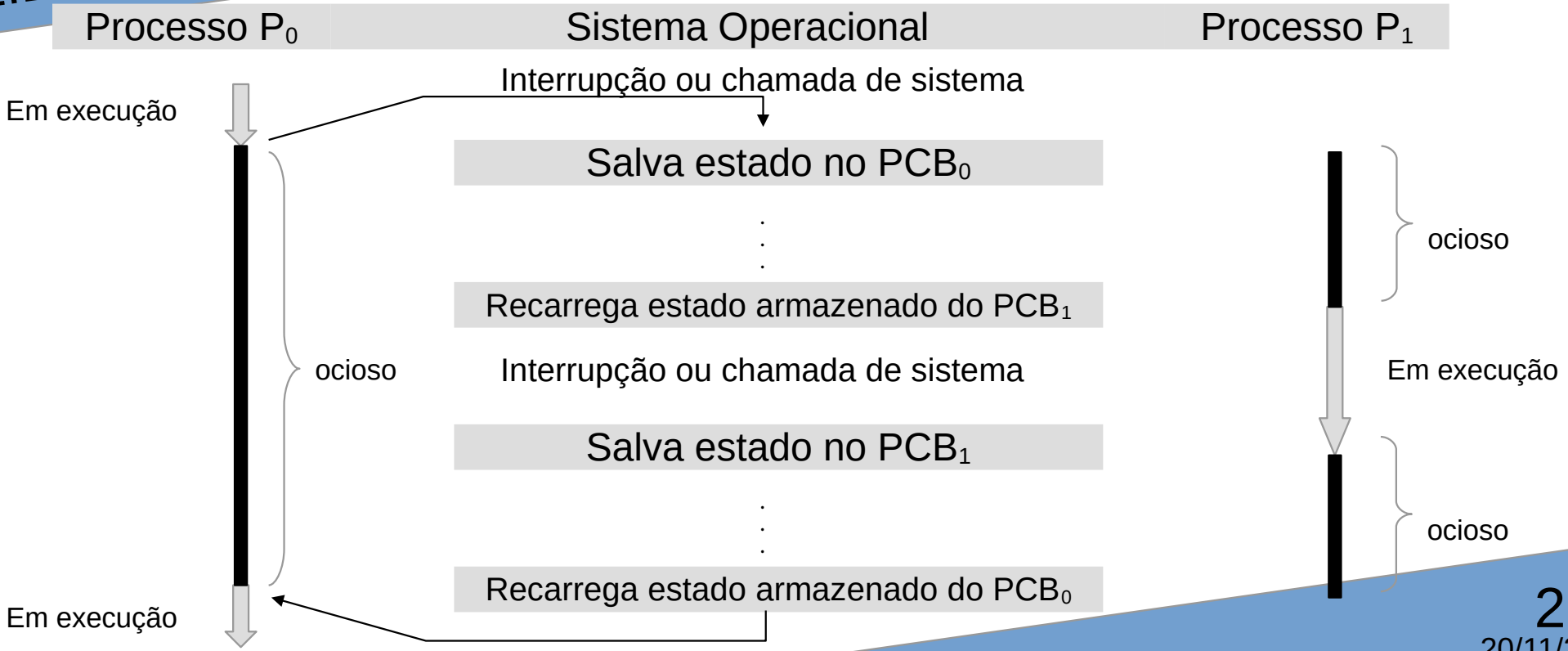
Linux - https://elixir.bootlin.com/linux/v6.5.1/source/arch/alpha/include/asm/thread_info.h

Windows - https://learn.microsoft.com/pt-br/windows/win32/api/winternl/nf-winternl-ntqueryinformationprocess?redirectedfrom=MSDN

Linux

Windows

2.1 Modelos de Processos



Fonte: SILBERCHARTZ, 2004

Execução de um Processo

- Todo processo criado recebe um número de identificação do S.O.. Este número é o **PID** (Process ID).
- Se um processo é criado por outro, o identificador do primeiro processo é o **PPID** (Parent PID).
- Se o **PID** recebido pelo processo recém criado for **negativo,** significa que algo errado aconteceu
- Ao criar um processo filho, este recebe inicialmente um **PID = zero** se tudo correr bem.
- Cada processo filho é responsável pelo valor de seus retornos.

Execução de um Processo

- Após a execução da trabalho, o processo deve ser encerrado por uma das seguintes razões:
  - **Saída normal (voluntária):** vai até o fim da execução;
  - **Saída por erro (voluntária):** não é possível executar por falta do recurso, por exemplo;
  - **Erro fatal (involuntário):** execução de instrução ilegal, referência de memória inexistente, divisão por zero;
  - **Cancelamento por outro processo (involuntário):** um segundo processo exeuta uma chamada ao sistema para terminar outro que estava em execução (kill, no Linux e TerminateProcess no Windows).

Hierarquias de Processos

- Alguns sistemas operacionais como o UNIX e derivados, permitem que processos criem outros processos, e como dizem no popular: “pai é quem cria”.
- Dessa forma, o processo criado por outro recebe o nome especial de filho e continua associado (dependente) ao processo que o criou.
- Processos filhos podem criar outros processos sucessivamente, criando um grupo de processos que são dependentes da existência do primeiro processo.
- No caso do Windows, os processos são independentes, mas estão associados através de um identificador especial (handle) para que o pai “controle” os filhos. Este identificador pode ser repassado para outro processo, quebrando a hierarquia.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
void main (int argc, char *argv[])
{
   int pid;
   /* ramifica um outro processo */
   pid = fork();

   if (pid < 0) { /* verifica se ocorreu algum erro */
      fprintf(stderr, "O fork falhou\n");
      exit(-1);
   } else if (pid == 0) { /* processo filho */
      printf("Oi, pai %d! Eu sou o seu filho %d\n", getppid(), getpid());
   } else { /* processo pai */
      /* O pai espera pela conclusão do filho */
     printf("Filho! É você? Sou eu, seu pai %d\n", getpid());
     wait(NULL);
     printf("O filho saiu\n");
     exit(0);
   }
}
```


Fonte: TANENBAUM, 2003

2.1 Modelos de Processos

Salve o arquivo como paiefilho.c

No prompt de comando digite:

```
$ gcc -o paiefilho paiefilho.c
```

Execute o programa com:

```
$ ./paiefilho
Filho! É você? Sou eu, seu pai 20080
Oi, pai 20080! Eu sou o seu filho 20081
O filho completou
```

THREADS

- O sistema operacional mantém armazenadas no PCB todas as estruturas necessárias para execução dos processos que são mantidos devidamente isolados para que não acessem indevidamente os recursos de outros processos.
- Isso permite que cada processo enxergue os recursos disponibilizados como sendo o seu próprio ambinete ou computador único.

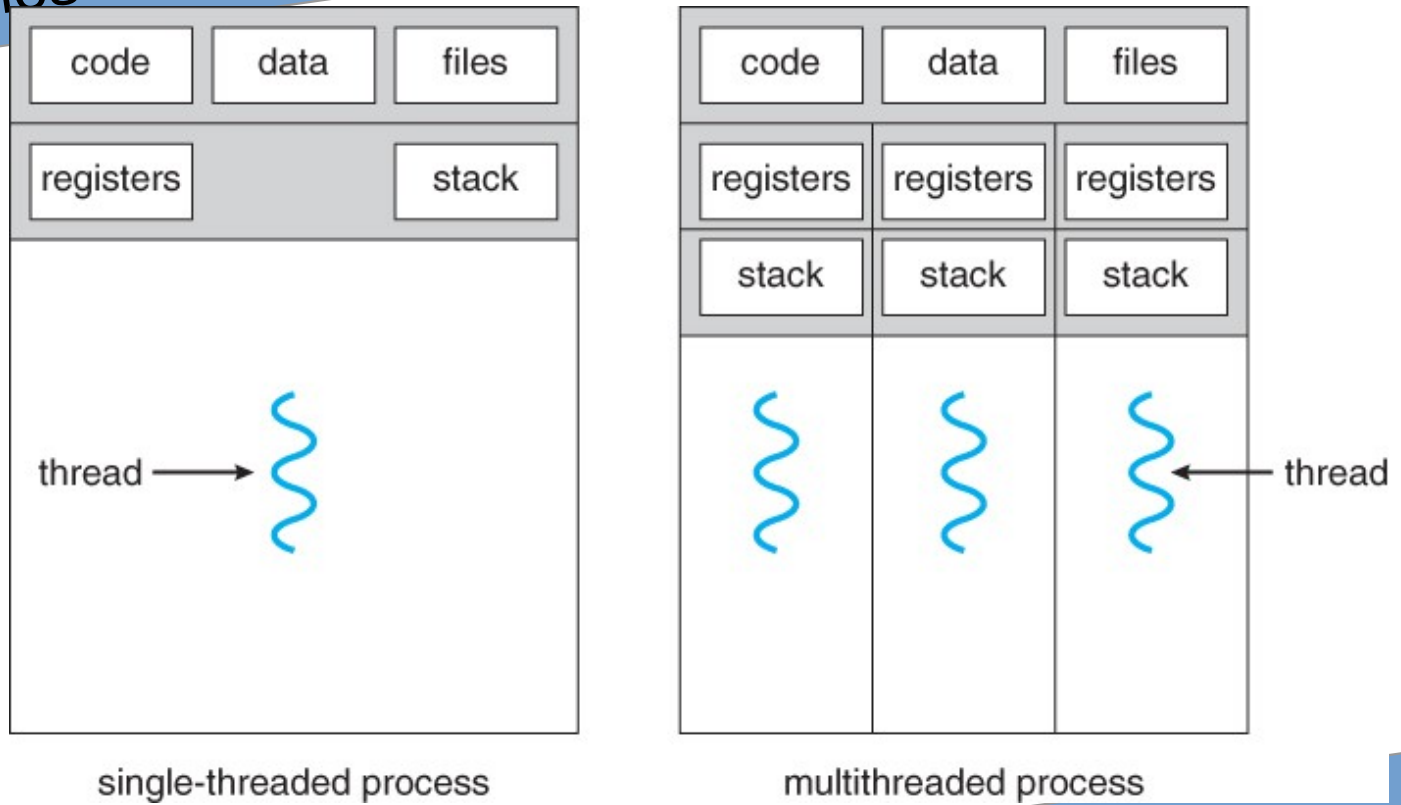
THREADS

- Uma thread (fio em inglês) é um fluxo separado de execução dentro de um processo. Seria um processo dentro de um processo ou alguns chamam de miniprocesso.
- Geralmente num programa existem várias threads sendo executadas ao mesmo tempo dentro do mesmo programa.
- Isso é desejado quando um programa precisa executar várias tarefas simultaneamente. Entretanto, algumas dessas atividades podem ficar bloqueadas e o uso de das threads ajuda a contornar essa questão, pois executam (quase) paralelamente.

THREADS

- As threads compartilham as informações dos PCBs dos processos que os criaram, além de terem suas próprias estruturas de armazenamento e controle;
- Threads são mais rápidas, fáceis e “baratas” para serem criadas;
- São boas para serem usadas em sistemas com muita E/S.
- São excelentes em sistemas multiprocessados ou multicore.

2.1 Modelos de Processos

- 

Fonte: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

THREADS

- Exemplos de uso de threads:
  - Editores de textos;
  - Servidores de rede;
  - Sistemas de backup;
  - Sistemas multimídia: editores de áudio/vídeo;
  - Planilhas eletrônicas...

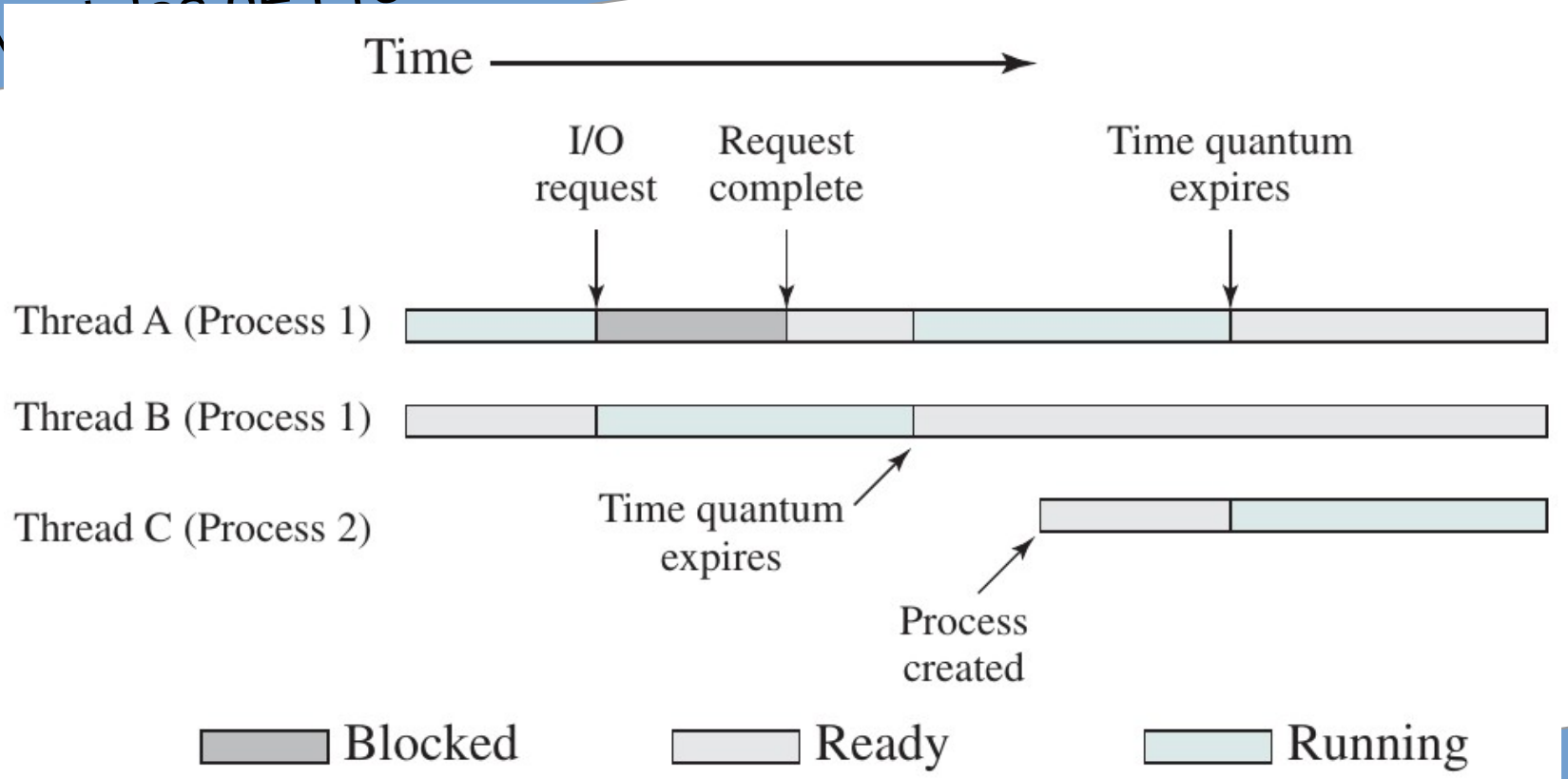
THREADS

- Existema basicamente três estados que uma thread pode assumir:
  - Running;
  - Ready;
  - Blocked.

THREADS

Há 4 operações básicas associadas à mudança de estados das threads [ANDE04]:

- **Spawn (gerar):** Quando um novo processo é gerado, uma thread é gerada junto com ele e internamente este processo pode gerar outras threads, disponibilizando um ponteiro de instruções e argumentos para a nova thread. A nova thread é disponibilizada com seu próprio contexto de registradores e espaço de pilhas e colocado na fila de pronto;
- **Block (Bloqueado):** Quando uma thread precisa esperar por um evento, ele será bloqueado, salvando seus resgistradores do usuário, contador de programas pilhas de ponteiros. Agora o processador pode mudar para outro processo na fila de pronto do mesmo processo ou outro.
- **Unblock (Desbloqueado):** Quando o evento que estava bloqueando a thread ocorre, a thread é mida para fila de pronto.
- **Finish (Finalizado):** Quando uma thread completa sua execução, seus registradores de contexto e pilhas são desalocados.


Time
I/O request
Request complete
Time quantum expires
Thread A (Process 1)
Thread B (Process 1)
Thread C (Process 2)
Time quantum expires
Process created
Blocked
Ready
Running

TIPOS DE THREADS

Existem duas categorias de implementação das threads:

- Threads a nível de usuário (user-level threads – ULTs);
- Threads a nível de kernel (kernel-level threads – KLTs).

Esta última também é referenciada como thread suportada pelo kernel ou processos peso-leve.

THREADS A NÍVEL DE USUÁRIO

São implementadas e gerenciadas pelo programa (processo) e o kernel não se ‘preocupa’ com sua existência.

Os programas podem implementar multithread usando bibliotecas apropriadas para gerenciamento das threads.

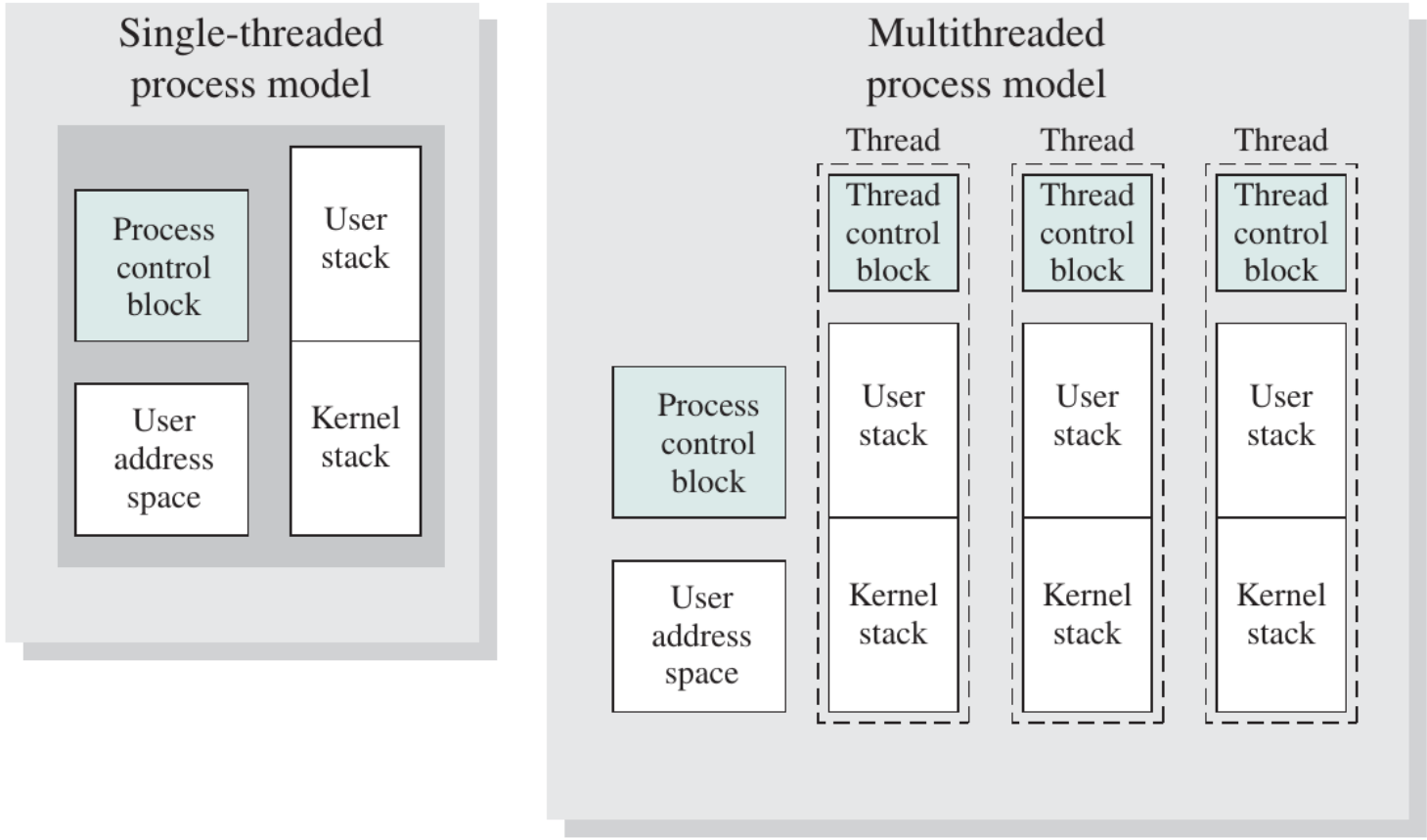
Estas bibliotecas tem código para criação e destruição das threads, passagem de mensagens e dados entre as threads, para escalonamento de execução e para salvar e restaurar o contexto das threads.

THREADS A NÍVEL DE USUÁRIO

Por ‘padrão’, uma aplicação inicia e começa sua execução com uma única thread. Esta aplicação e sua thread são alocadas para um único processo gerenciado pelo kernel e a qualquer momento durante a execução do programa (que está no estado de execução), a aplicação pode gerar (spawn) uma nova thread para ser executada dentro do mesmo processo.

Todo o processo de geração e criação das estruturas da nova thread é realizada pela biblioteca que ganha o estado de pronto.

O contexto é salvo quando o controle da thread é eviado para a biblioteca e restaurado quando volta novamente para a thread.



Modelos de Processos nono thread e multi thread
Fonte: Stallings (2012)

THREADS A NÍVEL DE KERNEL

Em um recurso KLT puro, todo o trabalho de gerenciamento de threads é feito pelo kernel. Não há código de gerenciamento de thread no nível do aplicativo, apenas uma interface de programação de aplicativo (API) para o recurso de thread do kernel. O Windows é um exemplo dessa abordagem.

O kernel mantém informações de contexto para o processo como um todo e para threads individuais dentro do processo. O agendamento pelo kernel é feito por thread. Esta abordagem supera as duas principais desvantagens da abordagem ULT. Primeiro, o kernel pode agendar simultaneamente vários threads do mesmo processo em vários processadores. Segundo, se um thread de um processo for bloqueado, o kernel poderá agendar outro thread do mesmo processo. Outra vantagem da abordagem KLT é que as próprias rotinas do kernel podem ser multithread.

THREADS A NÍVEL DE KERNEL

A principal desvantagem da abordagem KLT em comparação com a abordagem ULT é que a transferência de controle de um thread para outro dentro do mesmo processo requer uma mudança de modo para o kernel. Para ilustrar as diferenças, a abaixo mostra os resultados das medições feitas em um computador VAX uniprocessado executando um sistema operacional semelhante ao UNIX.

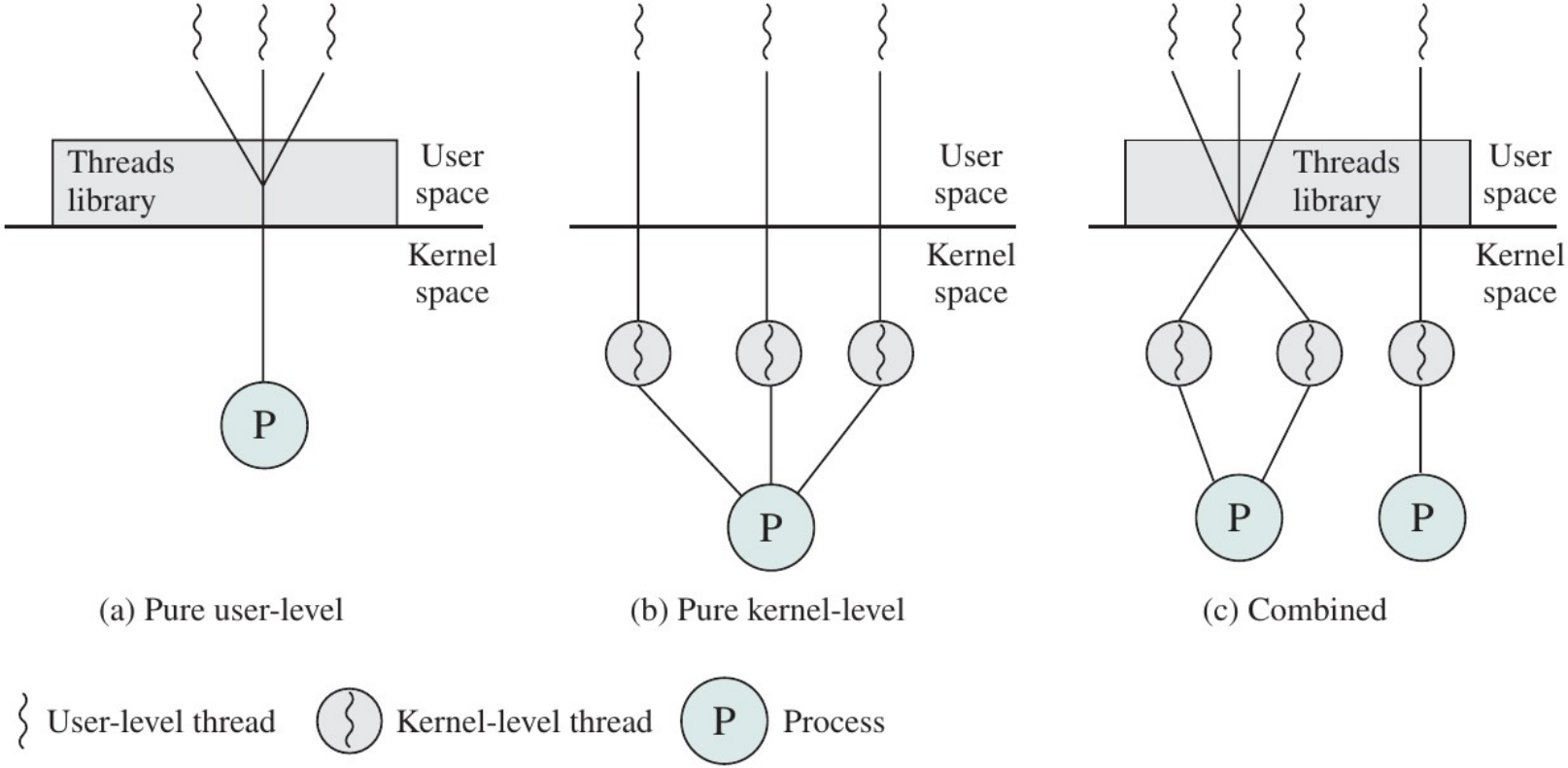
Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

THREADS A NÍVEL DE KERNEL

Em um recurso KLT puro, todo o trabalho de gerenciamento de threads é feito pelo kernel. Não há código de gerenciamento de thread no nível do aplicativo, apenas uma interface de programação de aplicativo (API) para o recurso de thread do kernel. O Windows é um exemplo dessa abordagem.

O kernel mantém informações de contexto para o processo como um todo e para threads individuais dentro do processo. O agendamento pelo kernel é feito por thread. Esta abordagem supera as duas principais desvantagens da abordagem ULT. Primeiro, o kernel pode agendar simultaneamente vários threads do mesmo processo em vários processadores. Segundo, se um thread de um processo for bloqueado, o kernel poderá agendar outro thread do mesmo processo. Outra vantagem da abordagem KLT é que as próprias rotinas do kernel podem ser multithread.

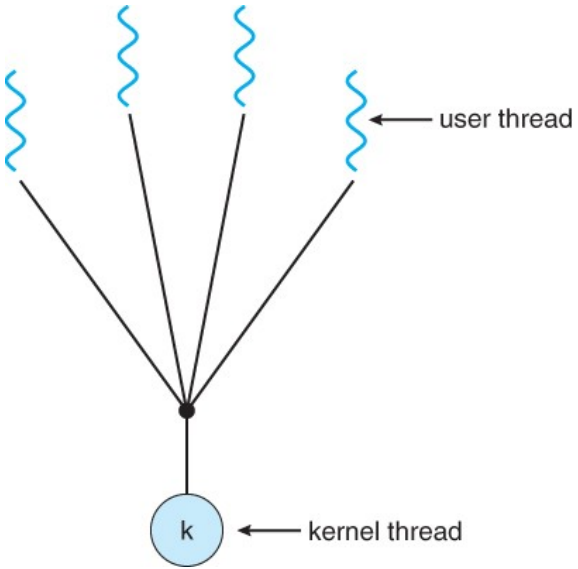
2.1 Modelos de Processos



Modelos de Processos multi thread
Fonte: Stalling (2012)

ESTRATÉGIAS DE MULTI THREAD

Modelo muitos para um:



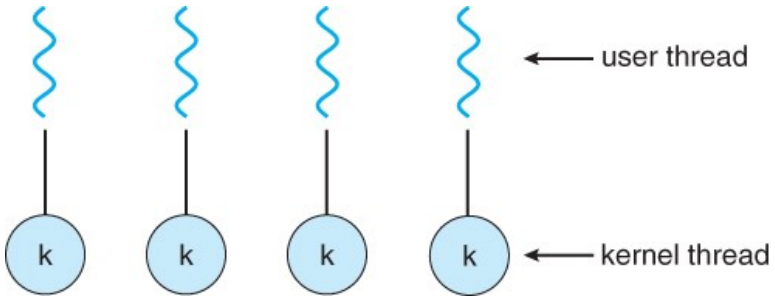
- No modelo muitos para um, muitos threads de nível de usuário são mapeados em um único thread de kernel.
- O gerenciamento de threads é feito pela biblioteca de threads no espaço do usuário, o que é muito eficiente.
- No entanto, se uma chamada de sistema de bloqueio for feita, todo o processo será bloqueado, mesmo que os outros threads do usuário possam continuar.
- Como um único thread de kernel pode operar somente em uma única CPU, o modelo muitos para um não permite que processos individuais sejam divididos em múltiplas CPUs.
- Threads verdes para Solaris e Threads GNU Portable implementam o modelo muitos para um no passado, mas poucos sistemas continuam a fazê-lo hoje.

Fonte: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

ESTRATÉGIAS DE MULTI THREAD

Modelo um para um:

- O modelo um para um cria um thread de kernel separado para lidar com cada thread de usuário.
- O modelo um para um supera os problemas listados acima, envolvendo o bloqueio de chamadas do sistema e a divisão de processos em várias CPUs.
- No entanto, a sobrecarga do gerenciamento do modelo um para um é mais significativa, envolvendo mais sobrecarga e desacelerando o sistema.
- A maioria das implementações deste modelo impõe um limite ao número de threads que podem ser criados.
- Linux e Windows de 95 a XP implementam o modelo um-para-um para threads.



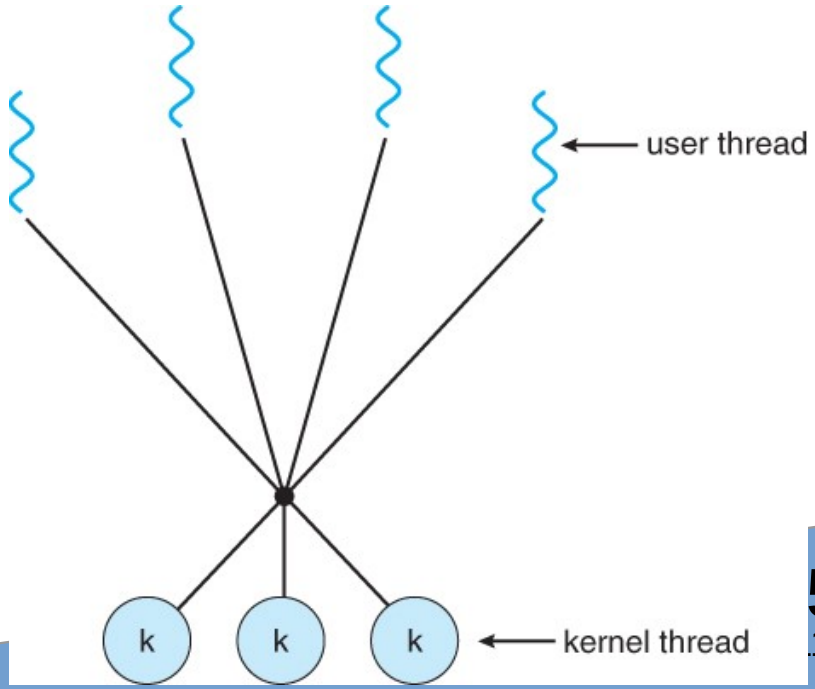
Fonte: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

ESTRATÉGIAS DE MULTI THREAD

Modelo muitos para muitos:

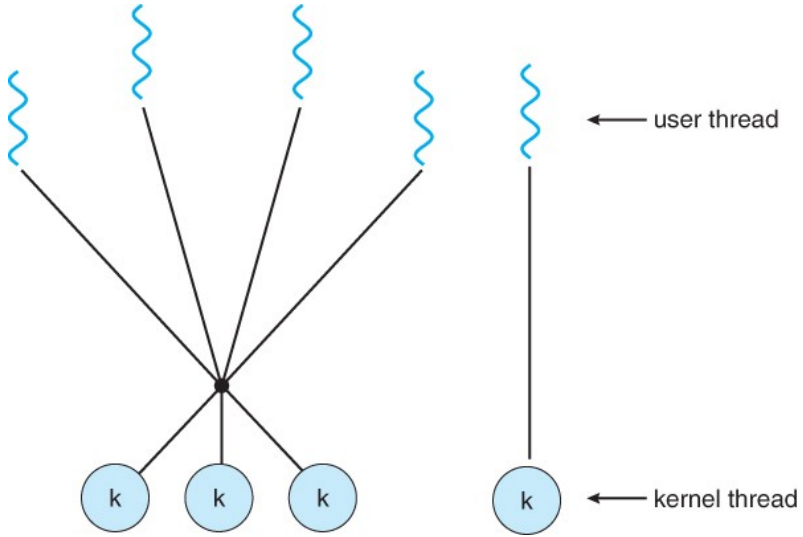
Fonte: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

- O modelo muitos para muitos multiplexa qualquer número de threads de usuário em um número igual ou menor de threads de kernel, combinando os melhores recursos dos modelos um para um e muitos para um.
- Os usuários não têm restrições quanto ao número de threads criados.
- O bloqueio de chamadas do sistema do kernel não bloqueia todo o processo.
- Os processos podem ser divididos em vários processadores.
- Processos individuais podem receber números variáveis de threads de kernel, dependendo do número de CPUs presentes e de outros fatores.



ESTRATÉGIAS DE MULTI THREAD

Modelo de dois níveis:



Fonte: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

- Uma variação popular do modelo muitos para muitos é o modelo de duas camadas, que permite operações muitos para muitos ou um para um.
- IRIX, HP-UX e Tru64 UNIX usam o modelo de duas camadas, assim como o Solaris antes do Solaris 9.

```
# Python program to illustrate the concept
# of threading
# importing the threading module
import threading


def print_cube(num):
    # function to print cube of given num
    print("Cube: {}" .format(num * num * num))


def print_square(num):
    # function to print square of given num
    print("Square: {}" .format(num * num))


if __name__ =="__main__":
```

```
18      # creating thread
19      t1 = threading.Thread(target=print_square, args=(10,))
20      t2 = threading.Thread(target=print_cube, args=(10,))
21
22      # starting thread 1
23      t1.start()
24      # starting thread 2
25      t2.start()
26
27      # wait until thread 1 is completely executed
28      t1.join()
29      # wait until thread 2 is completely executed
30      t2.join()
31
32      # both threads completely executed
33      print("Done!")
34
```

```
1  # Python program to illustrate the concept
2  # of threading
3  import threading
4  import os
5
6  def task1():
7     print("Task 1 assigned to thread: {}".format(threading.current_thread().name))
8     print("ID of process running task 1: {}".format(os.getpid()))
9
10 def task2():
11    print("Task 2 assigned to thread: {}".format(threading.current_thread().name))
12    print("ID of process running task 2: {}".format(os.getpid()))
13
14 if __name__ == "__main__":
15
```

```
16      # print ID of current process
17      print("ID of process running main program: {}".format(os.getpid()))
18
19      # print name of main thread
20      print("Main thread name: {}".format(threading.current_thread().name))
21
22      # creating threads
23      t1 = threading.Thread(target=task1, name='t1')
24      t2 = threading.Thread(target=task2, name='t2')
25
26      # starting threads
27      t1.start()
28      t2.start()
29
30      # wait until all threads finish
31      t1.join()
32      t2.join()
```


```python
# Python ThreadPool
import concurrent.futures

def worker():
   print("Worker thread running")

# create a thread pool with 2 threads
pool = concurrent.futures.ThreadPoolExecutor(max_workers=2)

# submit tasks to the pool
pool.submit(worker)
pool.submit(worker)

# wait for all tasks to complete
pool.shutdown(wait=True)

print("Main thread continuing to run")

```

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

// Let us create a global variable to change it in threads
int g = 0;

// The function to be executed by all threads
void *myThreadFun(void *vargp)
{
    // Store the value argument passed to this thread
    int *myid = (int *)vargp;

    // Let us create a static variable to observe its changes
    static int s = 0;

    // Change static and global variables
    ++s; ++g;

```

```
21      // Print the argument, static and global variables
22      printf("Thread ID: %d, Static: %d, Global: %d\n", *myid, ++s, ++g);
23  }
24
25  int main()
26  {
27      int i;
28      pthread_t tid;
29
30      // Let us create three threads
31      for (i = 0; i < 3; i++)
32      pthread_create(&tid, NULL, myThreadFun, (void *)&tid);
33
34      pthread_exit(NULL);
35      return 0;
36  }
```

Contato:

“

wagner.costa@cest.edu.br

”