Marisangila Alves, MSc
marisangila.alves@udesc.br
marisangila.com.br

UDESC
UNIVERSIDADE DO ESTADO DE SANTA CATARINA
JOINVILLE
CENTRO DE CIÊNCIAS TECNOLÓGICAS

UDESC
Universidade do Estado de Santa Catarina

2025/2

Sistemas Operacionais

Processos

Sumário

Definição

O que é um processo?

> *Processo é uma abstração de um programa em execução.*
>
> (TANENBAUM, 2010)

> *(TANENBAUM, 2010) Um processo é apenas uma instância de um programa em execução, incluindo os valores atuais do contador do programa, registradores e variáveis.*

Atenção!

Processo $\neq$ Programa

- Programa: Entidade estática;
- Processo: Seu estado muda a medida em que avança sua execução.

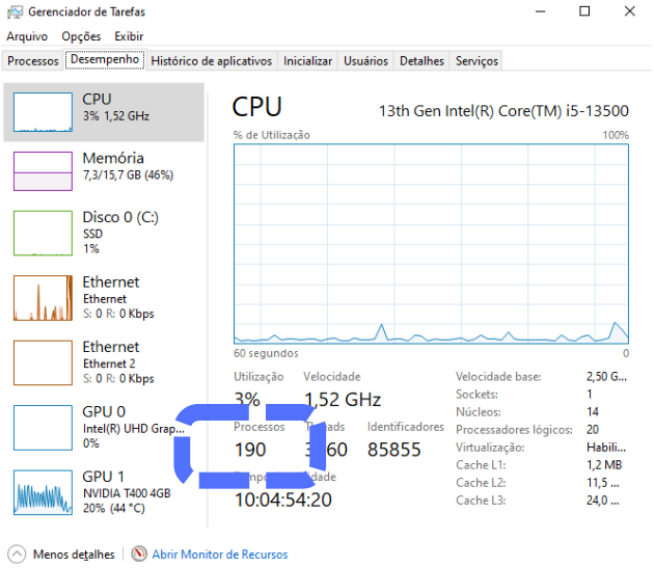
Nota: (Analogia:)

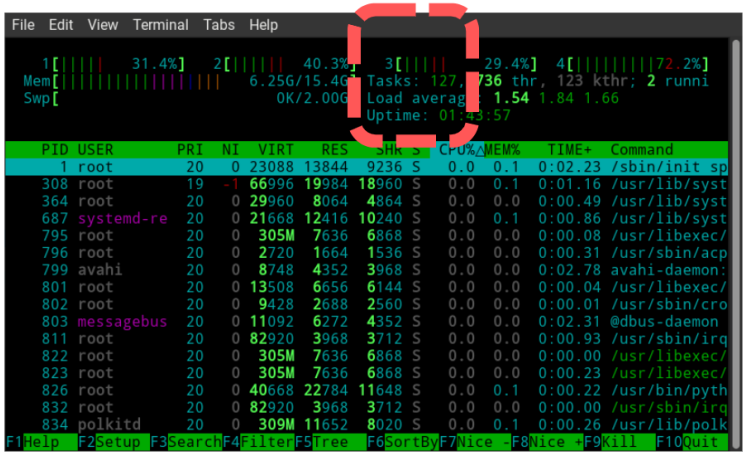
Lembrando de Programação Orientada a Objetos: **Classe está para programa, assim como processo está para objeto!**

Nota: (Analogia:)

Ainda mais abstrata: Podemos pensar tambem em receitas e cozinheiros (Maziero, 2019).

1. Pode-se dizer que um programa é o equivalente de uma "receita de torta" dentro de um livro de receitas (um diretório) guardado em uma estante (um disco) na cozinha (o computador).
2. Essa receita de torta define os ingredientes necessários (entradas) e o modo de preparo (programa) da torta (saída).
3. A ação de "executar" a receita, providenciando os ingredientes e seguindo os passos definidos na mesma, é a tarefa propriamente dita.
4. A cada momento, o cozinheiro (o processador) está seguindo um passo da receita (posição da execução) e tem uma certa disposição dos ingredientes e utensílios em uso (as entradas e variáveis internas da tarefa).


Gerenciador de Tarefas
Arquivo Opções Exibir
Processos Desempenho Histórico de aplicativos Inicializar Usuários Detalhes Serviços
CPU
3% 1,52 GHz
Memória
7,3/15,7 GB (46%)
Disco 0 (C:)
SSD
1%
Ethernet
Ethernet
S: 0 R: 0 Kbps
Ethernet
Ethernet 2
S: 0 R: 0 Kbps
GPU 0
Intel(R) UHD Grap...
0%
GPU 1
NVIDIA T400 4GB
20% (44 °C)
CPU
13th Gen Intel(R) Core(TM) i5-13500
% de Utilização
100%
60 segundos
0
Utilização
3%
Velocidade
1,52 GHz
Processos
190
Identificadores
85855
10:04:54:20
Velocidade base: 2,50 G...
Sockets: 1
Núcleos: 14
Processadores lógicos: 20
Virtualização: Habili...
Cache L1: 1,2 MB
Cache L2: 11,5 ...
Cache L3: 24,0 ...
Menos detalhes
Abrir Monitor de Recursos


File Edit View Terminal Tabs Help
1[|||||| 31.4%] 2[|||||| 40.3%] 3[|||||| 29.4%] 4[||||||||72.2%]
Mem[|||||||||||||||| 6.25G/15.4G] Tasks: 127, 736 thr, 123 kthr; 2 runni
Swp[0K/2.00G] Load averag 1.54 1.84 1.66
Uptime: 01:43:57
PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
1 root 20 0 23088 13844 9236 S 0.0 0.1 0:02.23 /sbin/init sp
308 root 19 -1 66996 19984 18960 S 0.0 0.1 0:01.16 /usr/lib/syst
364 root 20 0 29960 8064 4864 S 0.0 0.0 0:00.49 /usr/lib/syst
687 systemd-re 20 0 21668 12416 10240 S 0.0 0.1 0:00.86 /usr/lib/syst
795 root 20 0 305M 7636 6868 S 0.0 0.0 0:00.08 /usr/libexec/
796 root 20 0 2720 1664 1536 S 0.0 0.0 0:00.31 /usr/sbin/acp
799 avahi 20 0 8748 4352 3968 S 0.0 0.0 0:02.78 avahi-daemon:
801 root 20 0 13508 6656 6144 S 0.0 0.0 0:00.04 /usr/libexec/
802 root 20 0 9428 2688 2560 S 0.0 0.0 0:00.01 /usr/sbin/cro
803 messagebus 20 0 11092 6272 4352 S 0.0 0.0 0:02.31 @dbus-daemon
811 root 20 0 82920 3968 3712 S 0.0 0.0 0:00.93 /usr/sbin/irq
822 root 20 0 305M 7636 6868 S 0.0 0.0 0:00.00 /usr/libexec/
823 root 20 0 305M 7636 6868 S 0.0 0.0 0:00.23 /usr/libexec/
826 root 20 0 40668 22784 11648 S 0.0 0.1 0:00.22 /usr/bin/pyth
832 root 20 0 82920 3968 3712 S 0.0 0.0 0:00.00 /usr/sbin/irq
834 polkitd 20 0 309M 11652 8020 S 0.0 0.1 0:00.26 /usr/lib/polk
F1Help F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice - F8Nice + F9Kill F10Quit

```
1 htop
```

Multiprogramação

Multiprogramação é a troca rápida de processos.

- Múltiplos processos são mantidos na memória principal.
- Otimização de recursos computacionais.
- Máquinas monoprocessadas **ou** multiprocessadas.
- Multiprogramação:
  - Interrupção;
  - Proteção entre processos.

Sistemas Batch:

- Execução sequencial de tarefas em lotes (*jobs*).
- Leitura linear dos dados no disco.
- Apenas um processo por vez (monoprocessado).
- Baixa utilização do processador devido ao tempo de espera por I/O.

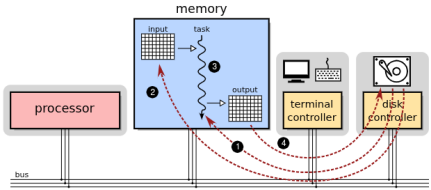


Figura 1: Sistemas Monotarefa (Maziero, 2019).

O HD permitiu carregar e armazenar múltiplos processos que não cabem na memória principal (OLIVEIRA; CARISSIMI; TOSCANI, 2001).

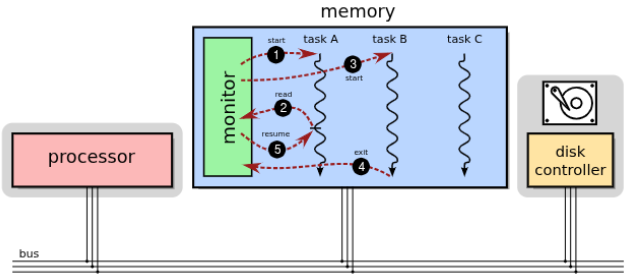


Figura 2: Sistemas Multitarefa (Maziero, 2019).

Processos de usuário vs processos do sistema

- A maioria executa programas dos usuários.
- Alguns realizam tarefas do próprio sistema (***daemons***).
- Exemplo: *spooling* de impressão (usuário envia ao disco, *daemon* envia à impressora).

- **Ciclo de processador:** quando usa a CPU.
- **Ciclo de E/S:** quando aguarda operações de entrada/saída.
- Alternância ocorre por chamadas de sistema.
- O primeiro ciclo de um processo é sempre de processador.

Quanto ao uso de recursos:

- **CPU-bound:** dependem principalmente do processador (ex.: algoritmo de busca).
- **I/O-bound:** dependem principalmente de E/S (ex.: cópia de arquivos).

- A utilização da CPU pode ser expressa por:

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

- Onde:
  - p: fração do tempo que um processo fica esperando dispositivos de E/S.
  - n: número de processos em memória ao mesmo tempo.
- Essa fórmula é uma simplificação e funciona melhor em cenários teóricos.
- Na prática, a utilização da CPU é influenciada por diversos fatores:
  - Prioridade dos processos.
  - Política de agendamento do sistema operacional.
  - Presença de outros recursos limitantes.

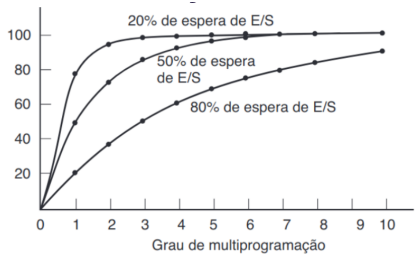
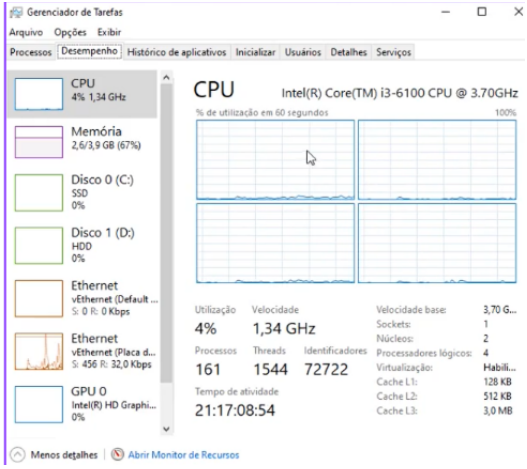
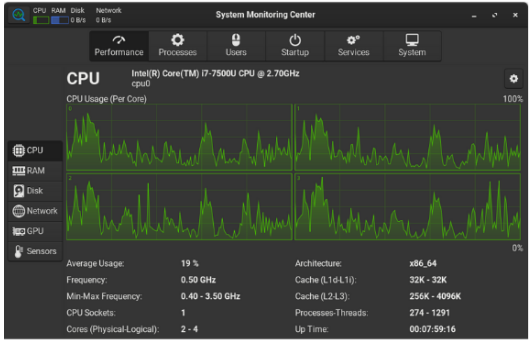


Figura 3: Grau de Multiprogramação (TANENBAUM, 2010).

Monitoramento da CPU





- Só CPU-bound → gargalo no processador.
- Só I/O-bound → CPU ociosa.
- Multiprogramação busca equilibrar ambos.
- Não basta apenas multiprogramação:
  - *Time Sharing* ou sistemas de tempo compartilhado;
  - Preempção por tempo.

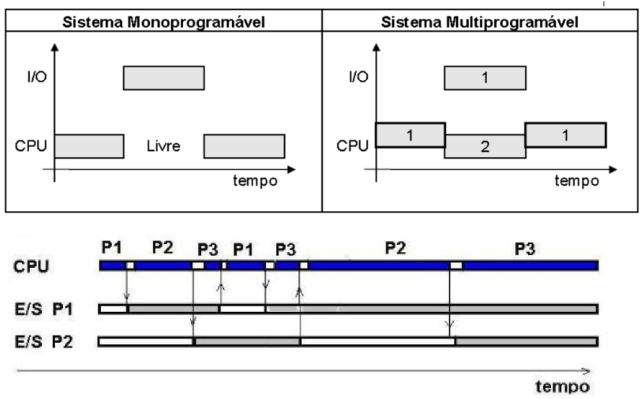


Figura 4: Monoprogramável versus Multiprogramável.

Máquinas Monoprocessadas e Multiprocessadas

- **Máquinas Monoprocessadas:**
  - Possuem apenas uma CPU.
  - Apenas **um processo** executa por vez.
  - O sistema alterna rapidamente entre processos.
  - Essa alternância cria a **ilusão de paralelismo** (**pseudoparalelismo**) (TANENBAUM, 2010).
- **Máquinas Multiprocessadas:**
  - Possuem duas ou mais CPUs trabalhando em conjunto.
  - Permitem a execução de **vários processos ao mesmo tempo**.
  - Existe **paralelismo real**.

Estados de Um Processo

Figura 5: Estados de um processo (Maziero, 2019).

Criação

1. Inicialização do sistema;
2. Chamada de sistema;
3. Solicitação de um usuário; e
4. Início de uma tarefa em lote.

Inicialização do Sistema:

- Criação dos primeiros processos pelo sistema operacional.

Tipos de processos:

- **Interativos:** interação direta com o usuário, executados em primeiro plano (*foreground*).
- **Não interativos:** serviços do sistema, executados em segundo plano (*background*).
- ***Daemons*:** processos do sistema, sem vínculo direto com o usuário (ex.: impressão, rede, logs).

Chamadas de Sistemas

- Chamada de sistema para criação de processo por outro processo em execução.
- Exemplos:
  - `fork()` no Unix.
  - `CreateProcess()` no Windows.
- Variáveis de sistema são copiadas.
- Técnica de otimização: **copy-on-write**.

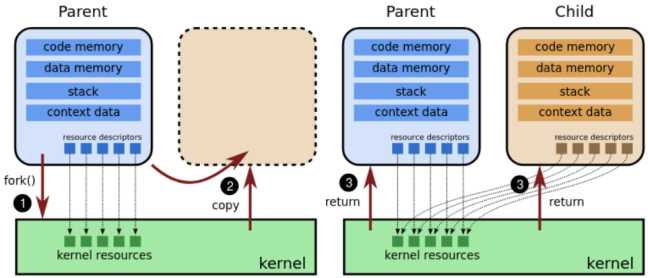


Figura 6: Hierarquia de Processo (Maziero, 2019).

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
int main ()
{
  int retval ;
  printf ("Ola, sou o processo %5d\n", getpid()) ;
  retval = fork () ;
  printf ("[retval: %5d] sou %5d, filho de %5d\n", retval,
   ↪  getpid(), getppid()) ;
  if ( retval < 0 )    // erro no fork()
  {
```

```
14      perror ("Erro") ;
15      exit (1) ;
16    }
17    else
18      if ( retval > 0 )  // sou o processo pai
19        wait (0) ;
20      else               // sou o processo filho
21        sleep (5) ;
22
23    printf ("Tchau de %5d!\n", getpid()) ;
24    exit (0) ;
25  }
```

Código 1: Exemplo de uso da chamada de sistema `fork`.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
int main (int argc, char *argv[], char *envp[])
{
  int retval ;
  printf ("Ola, sou o processo %5d\n", getpid()) ;
  retval = fork () ;
  printf ("[retval: %5d] sou %5d, filho de %5d\n", retval,
   ↪  getpid(), getppid()) ;
  if ( retval < 0 )       // erro no fork ()
    perror ("Erro: ") ;
```

```
14    else
15      if ( retval > 0 )      // sou o processo pai
16        wait (0) ;
17      else                   // sou o processo filho
18      {
19        execve ("/bin/date", argv, envp) ;
20        perror ("Erro") ;
21      }
22    printf ("Tchau de %5d!\n", getpid()) ;
23    return EXIT_SUCCESS;
24  }
```

Código 2: Exemplo de uso da chamada de sistema execv.

Solicitação de Usuário:

- Um usuário pode solicitar a criação de um novo processo.
- O processo é associado a uma sessão de trabalho.
- Exemplos:
  - Login e senha no *shell*.
  - Identificação por **PID único**.

Nota:

- O shell (`bash`, `zsh`, etc.) ou o gerenciador de janelas não são mágicos:
- Eles são processos em execução.
- Quando você pede para executar outro programa, eles chamam uma syscall `execv` para carregar o novo binário.
  1. Abrir o binário ELF.
  2. Carregar suas seções (`.text`, `.data`, etc.) para memória.
  3. Criar uma nova pilha inicial.
  4. Definir o ponteiro de instrução para `_start` (ou `main`, indiretamente).
  5. Iniciar a execução do novo programa.

Hierarquia

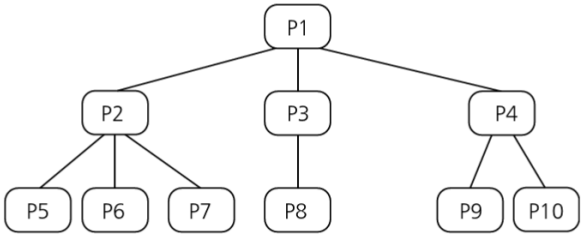
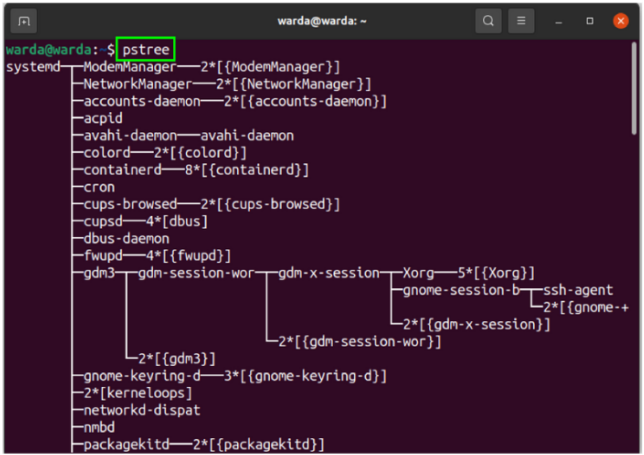


Figura 7: Hierarquia de Processo (OLIVEIRA; CARISSIMI; TOSCANI, 2001).

- Processos podem criar outros processos.
- Várias gerações podem ser criadas.
- Quando um processo é encerrado, todos os seus subprocessos são destruídos.
- **Windows:** permite a transferência de descendência.
- **UNIX:** não permite transferência de descendência.
- Existe uma **árvore de processos**, cujo processo raiz é o `init`.


warda@warda: ~
warda@warda:~$ pstree
systemd─┬─ModemManager───2*[{ModemManager}]
├─NetworkManager───2*[{NetworkManager}]
├─accounts-daemon───2*[{accounts-daemon}]
├─acpid
├─avahi-daemon───avahi-daemon
├─colord───2*[{colord}]
├─containerd───8*[{containerd}]
├─cron
├─cups-browsed───2*[{cups-browsed}]
├─cupsd───4*[dbus]
├─dbus-daemon
├─fwupd───4*[{fwupd}]
├─gdm3─┬─gdm-session-wor─┬─gdm-x-session─┬─Xorg───5*[{Xorg}]
│ │ │ ├─gnome-session-b─┬─ssh-agent
│ │ │ │ └─2*[{gnome-+
│ │ │ └─2*[{gdm-x-session}]
│ │ └─2*[{gdm-session-wor}]
│ └─2*[{gdm3}]
├─gnome-keyring-d───3*[{gnome-keyring-d}]
├─2*[kerneloops]
├─networkd-dispat
├─nmbd
├─packagekitd───2*[{packagekitd}]

```
1 pstree
2 pstree -p
3 ps -ejH
```

PCB (Process Control Block)

Gerenciamento de processo	Gerenciamento de memória	Gerenciamento de arquivo
Registros	Ponteiro para informações sobre o segmento de texto	Diretório-raiz
Contador de programa	Ponteiro para informações sobre o segmento de dados	Diretório de trabalho
Palavra de estado do programa	Ponteiro para informações sobre o segmento de pilha	Descritores de arquivo
Ponteiro da pilha		ID do usuário
Estado do processo		ID do grupo
Prioridade		
Parâmetros de escalonamento		
ID do processo		
Processo pai		
Grupo de processo		
Sinais		
Momento em que um processo foi iniciado		
Tempo de CPU usado		
Tempo de CPU do processo filho		
Tempo do alarme seguinte		

Figura 8: Descritor ou bloco de controle do processo(PCB) (TANENBAUM, 2010).

Clique aqui!

Acesse `struct task_struct`

O PID (Process ID) é um identificador único atribuído pelo sistema operacional a cada processo.
Para visualizar todos os processos do usuário em execução:

```
1          ps aux
```

- **Área de Código:** contém as instruções executáveis do programa, em seção de leitura apenas.
- **Área de Dados:** armazena variáveis globais, estáticas e dados dinâmicos alocados durante a execução.
- **Área de Pilha:** gerencia a execução de funções, guardando endereços de retorno, parâmetros e variáveis locais (*LIFO*).
- **Contador de Programa:** registrador que aponta para a próxima instrução a ser executada.

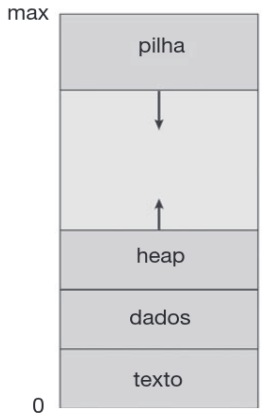
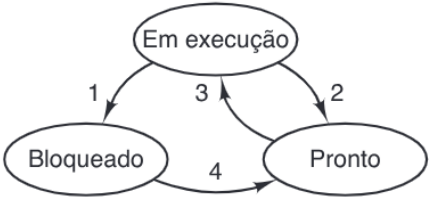


Figura 9: Processo na memória (SILBERSCHATZ; GALVIN; GAGNE, 2001).

Dispatcher e Scheduler

- Política : Decidir como fazer;
- Mecânismo : Execução da decisão;
- ***Scheduler* (Escalonador):** decide qual processo deve usar a CPU.
- ***Dispatcher:*** realiza a troca de contexto e entrega a CPU ao processo escolhido.
- Portanto, em resumo, o *scheduler* escolhe, o *dispatcher* executa a escolha.

Execução



1. O processo é bloqueado aguardando uma entrada
2. O escalonador seleciona outro processo
3. O escalonador seleciona esse processo
4. A entrada torna-se disponível

Figura 10: Diagrama de estados de um processo (TANENBAUM, 2010).

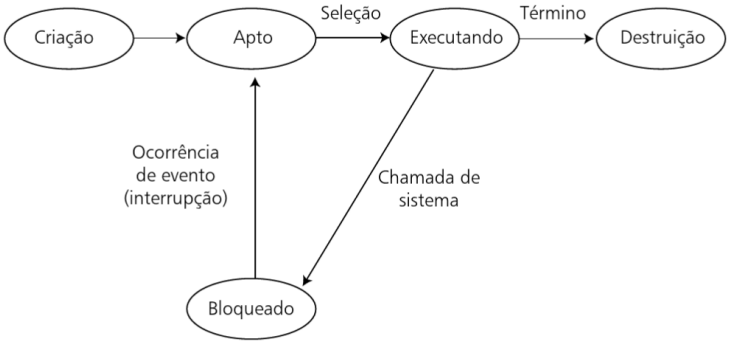


Figura 11: Diagrama de estados de um processo - 5 estados (OLIVEIRA; CARISSIMI; TOSCANI, 2001).

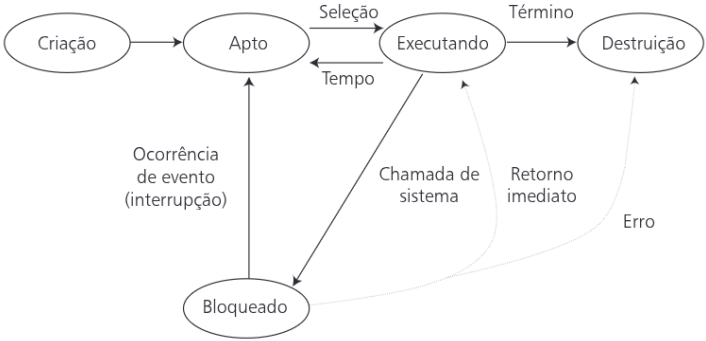


Figura 12: Diagrama de estados de um processo - Novos caminhos (OLIVEIRA; CARISSIMI; TOSCANI, 2001).

- **Nova:** a tarefa está sendo preparada para executar.
- **Pronta:** a tarefa está esperando pelo processador.
- **Executando:** a tarefa está executando suas instruções.
- **Suspensa:** a tarefa aguarda algum evento externo.
- **Terminada:** a tarefa encerrou ou foi abortada.

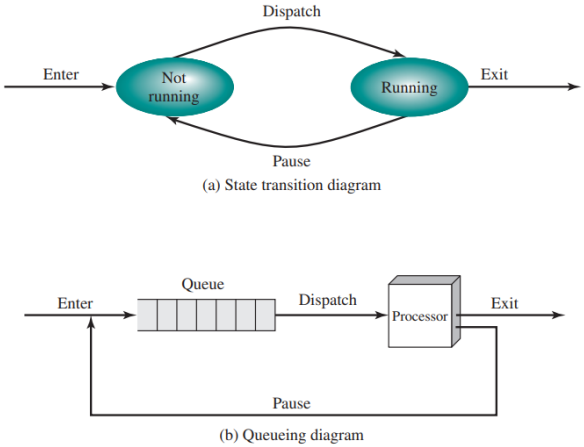


Figura 13: Estados de um processo (SILBERSCHATZ; GALVIN; GAGNE, 2001).

Na prática no kernel Linux:

- *R (Running):* processo em execução ou pronto para executar na CPU.
- *S (Sleeping):* processo em espera, aguardando algum evento (estado mais comum).
- *D (Uninterruptible Sleep):* processo esperando I/O, não pode ser interrompido.
- *T (Stopped):* processo parado, geralmente por um sinal (ex.: Ctrl+Z).
- *Z (Zombie):* processo terminou, mas sua entrada ainda está na tabela de processos até que o pai colete seu status.
- *I (Idle):* thread inativa, usada em kernels mais recentes (aparece no lugar de S para certas threads).
- *X (Dead):* processo morto (estado raro, geralmente não visível para o usuário).

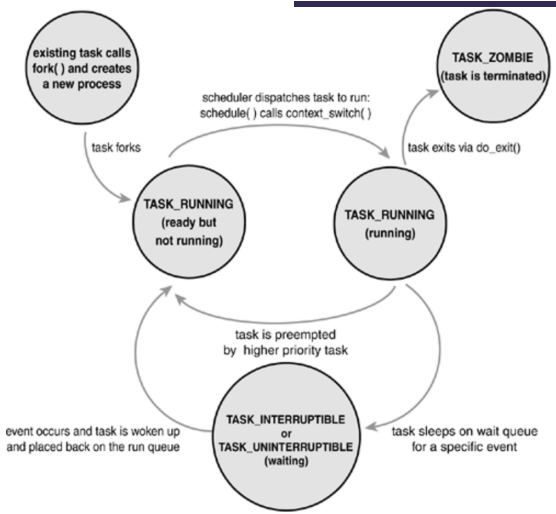
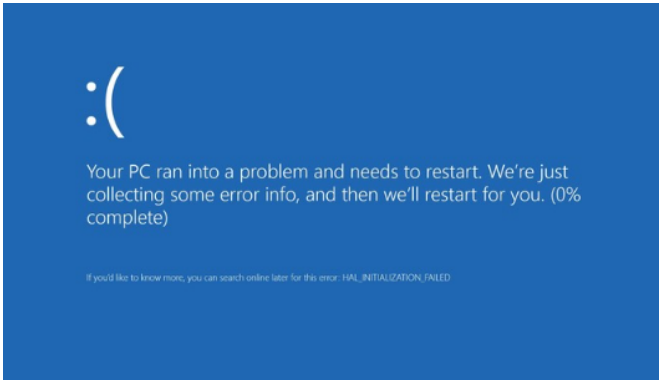


Figura 14: Fluxo de processos do kernel Linux (Love, 2005).

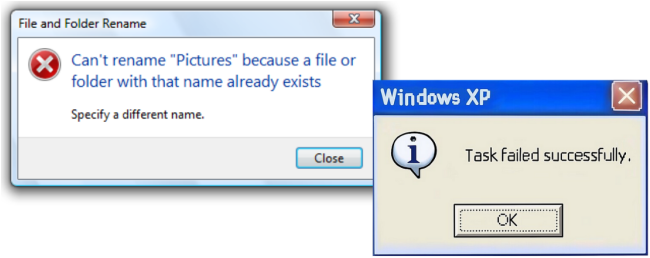
Término

- Saída normal (voluntária):
  - Exemplo: *log off* do usuário.

- Erro fatal (involuntário):



- Saída por erro (voluntária):



- Encerrado por outro processo (voluntário):
  - Chamada de Sistema:
  - `Kill` - Unix ou `TerminateProcess` - Windows.

Proteção

- É natural que o compartilhamento de recursos entre processos possa interferir na execução correta dos processos.
- Para evitar problemas, existem mecanismos de proteção:
  - Modos de operação;
  - Interrupção;
  - Proteção de periféricos, memória e processador.

- Uma interrupção é um sinal enviado ao processador que interrompe a execução atual.
- Permite que o processador responda rapidamente a eventos de entrada e saída.
- Exemplos:
  - Pressionar uma tecla.
  - Receber dados através de uma placa de rede.

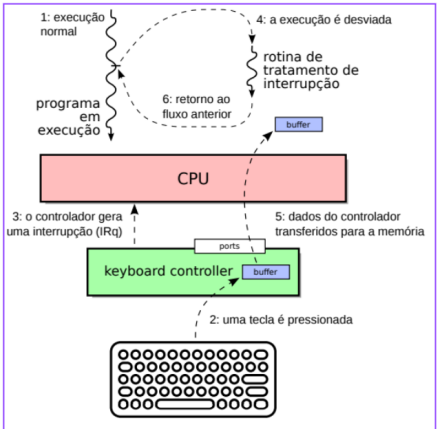


Figura 15: Tratamento de Interrupções (Maziero, 2019).

- Permite que controladores de periféricos chamem atenção do processador.
- Tratador de interrupção: rotina executada quando ocorre interrupção; retorna à execução original.
- Tipos de interrupção:
  - Hardware: evento externo, imprevisível.
  - Software (trap): gerada pelo próprio programa; usada em chamadas de sistema.
  - Exceção: gerada pelo processador por erros (divisão por zero, acesso inválido à memória).

- Ao ocorrer interrupção:
  - Registradores salvos (geralmente na pilha)
  - Execução desviada para tratador
  - Retorno ao programa via instrução de retorno de interrupção
- Prioridade: interrupções mais importantes podem interromper tratadores em execução.
- Vetores de interrupção: tabela na memória com endereços dos tratadores.
- Exemplos de Exceções:
  - Divisão por zero (Division by Zero)
  - Falha de segmentação (Segmentation Fault)
  - Breakpoint / Debug
  - Overflow aritmético (Arithmetic Overflow)
  - Instrução inválida (Invalid Opcode)

- O SO depende do hardware para implementar proteção.
- Modos de operação:
  - Supervisor: sem restrições, executa todas as instruções.
  - Usuário: restrições; instruções privilegiadas só em modo supervisor.
- Tentativa de instrução privilegiada em modo usuário → interrupção → SO em modo supervisor → processo abortado.
- Processos de usuário executam em modo usuário, SO em modo supervisor.
- Ao ligar ou resetar, processador inicia em modo supervisor e executa código de inicialização do SO em ROM.

- Instruções de E/S são privilegiadas: acesso direto por usuário gera interrupção.
- Processos de usuário devem usar chamadas de sistema para E/S.
- Tipos de interrupções:
  - Periféricos: conclusão de operação de E/S.
  - Hardware de proteção: captura operações ilegais.
  - Software (trap): chamada de sistema do usuário.
- Todas alternam processador para modo supervisor para execução do SO.

- Interrupções podem ser causadas por:
  - Hardware: E/S, temporizador, periféricos
  - Software (trap): chamadas de sistema
  - Exceções: divisão por zero, falha de segmentação, breakpoint/debug
- Registradores são salvos para preservar execução do programa.
- Vetores de interrupção determinam qual rotina deve ser executada.

Proteção de Memória e Controle de Processos I

- Evitar que usuários substituam rotinas do SO ou corrompam memória.
- Isolamento de memória: processo não acessa memória de outro.
- Registradores de limite:
  - Inferior: início da área do processo
  - Superior: fim da área do processo
  - Acesso ilegal gera interrupção → SO aborta processo.
- E/S mapeada em memória → proteção também cobre periféricos.
- Temporizador (timer): previne monopolização do processador, gera interrupções periódicas.
- Instruções privilegiadas: apenas SO pode modificar registradores de limite e controlar interrupções.
- Base do mecanismo: interrupções + modos de operação → sistemas seguros.

Threads

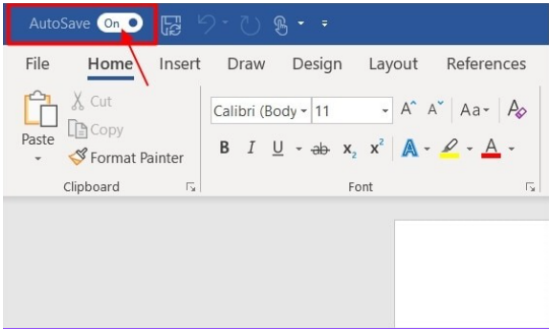
- Uma thread é definida como sendo um **fluxo de execução independente**.
- Um thread, às vezes chamado de **processo leve (lightweight process)**, é uma unidade básica de utilização de CPU;
  - Compreende um **ID de thread**, um **contador de programa**, um **conjunto de registradores** e uma **pilha**.

- Compartilha com outros threads pertencentes ao mesmo processo:
  - **Seção de código**, **seção de dados**, e outros **recursos do sistema operacional**, tais como **arquivos abertos** e **sinais**.
- Um **processo tradicional** ou **pesado (heavyweight)**, tem um **único fluxo de controle**.
- **Processos com múltiplos threads** podem realizar mais de uma tarefa por vez.
- **Processos** oferecem isolamento e segurança, enquanto as **threads** permitem colaboração eficiente no mesmo contexto.

- **Capacidade de resposta**
- **Compartilhamento de recursos**
- **Desempenho:** economia de tempo e recursos
  - Criar um thread é de **10 a 100 vezes mais rápido** que criar um processo.
- **Paralelismo (Multiprocessador):** permite executar mais de uma tarefa ao mesmo tempo.

- **Threads de usuário:**
  - Associadas a aplicativos que usam bibliotecas para gerenciar concorrência.
- **Threads de kernel:**
  - Parte da infraestrutura do sistema operacional.
  - Permitem um gerenciamento mais robusto e eficiente.

- Aplicativos **GUI** normalmente executam múltiplas tarefas ao mesmo tempo em um único processo.
- Exemplo: **salvamento automático** no Word.



- Editor de texto com backup local utiliza **três threads**:
  - **Thread 1:** interage com o usuário.
  - **Thread 2:** reformata o documento quando solicitado.
  - **Thread 3:** escreve periodicamente os conteúdos da RAM para o disco.

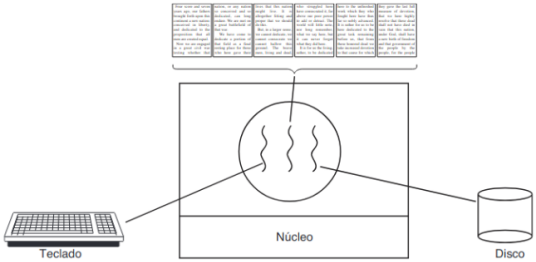


Figura 16: Editor de Texto (TANENBAUM, 2010).

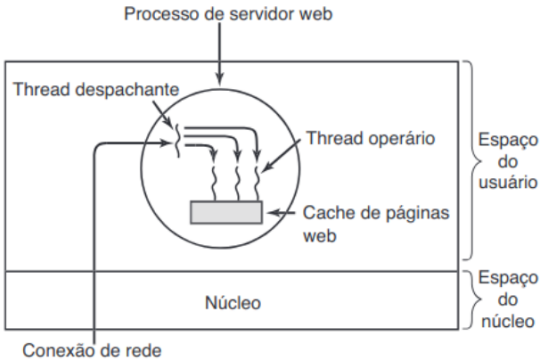


Figura 17: Servidor para Websites (TANENBAUM, 2010).

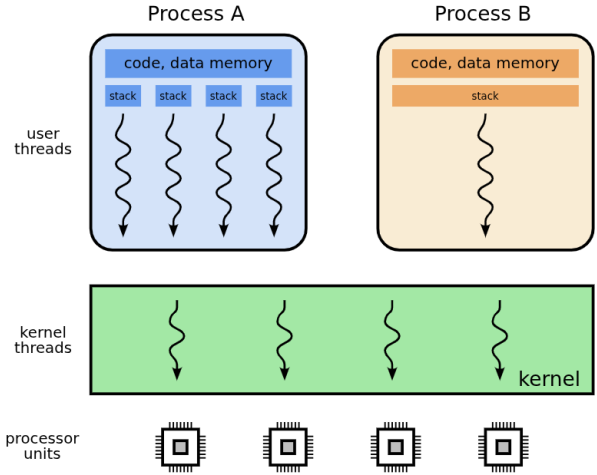


Figura 18: Threads de espaço de usuário e kernel (Maziero, 2019).

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define NUM_THREADS 16
void *threadBody (void *id)
{
  long tid = (long) id ; // ID da thread
  printf ("t%02ld: Olá!\n", tid) ;
  sleep (3) ;
  printf ("t%02ld: Tchau!\n", tid) ;
  pthread_exit (NULL) ;
}

int main (int argc, char *argv[])
{
  pthread_t thread [NUM_THREADS] ;
  long i, status ;
  for (i=0; i<NUM_THREADS; i++)
  {
    printf ("Main: criando thread %02ld\n", i) ;
    status = pthread_create (&thread[i], NULL, threadBody, (void *) i) ;
```

```
23      if (status)
24        perror ("pthread_create") ;
25    }
26    printf ("Main: fim\n") ;
27    pthread_exit (NULL) ;
28  }
```

Código 3: Exemplo de criação de `threads`.

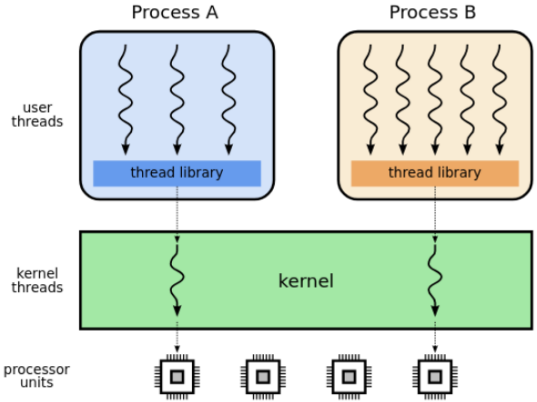


Figura 19: Modelo N:1 (Maziero, 2019).

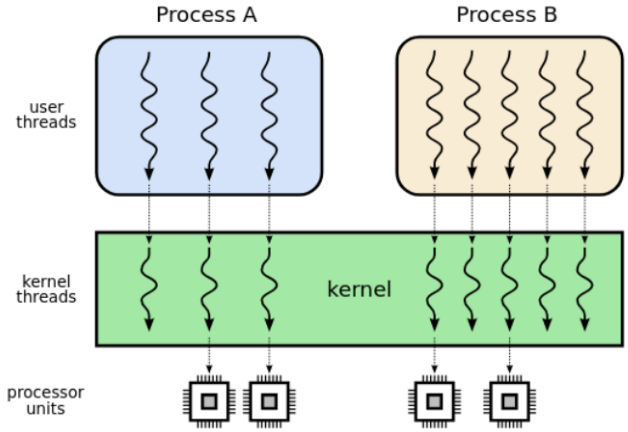


Figura 20: Modelo 1:1 (Maziero, 2019).

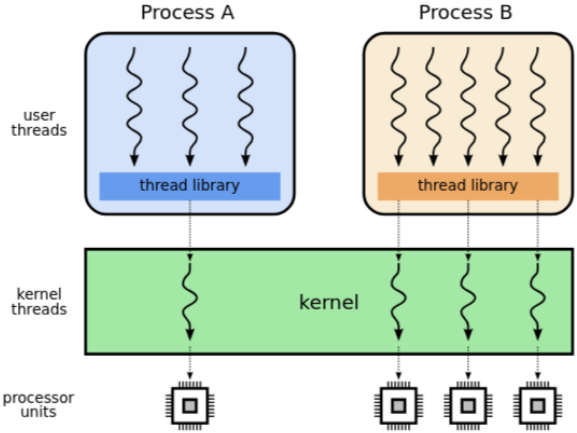


Figura 21: Modelo N:M - *threads pool* (Maziero, 2019).

Modelo	N:1	1:1	N:M
Resumo	N *threads* do processo mapeados em uma *thread* de núcleo	Cada *thread* do processo mapeado em uma *thread* de núcleo	N *threads* do processo mapeados em M< N *threads* de núcleo
Implementação	no processo (biblioteca)	no núcleo	em ambos
Complexidade	baixa	média	alta
Custo de gerência	baixo	médio	alto
Escalabilidade	alta	baixa	alta
Paralelismo entre *threads* do mesmo processo	não	sim	sim
Troca de contexto entre *threads* do mesmo processo	rápida	lenta	rápida
Divisão de recursos entre tarefas	injusta	justa	variável, pois o mapeamento *thread*→ processador é dinâmico
Exemplos	GNU Portable Threads, Microsoft UMS	Windows, Linux	Solaris, FreeBSD KSE

Figura 22: Comparação (Maziero, 2019).

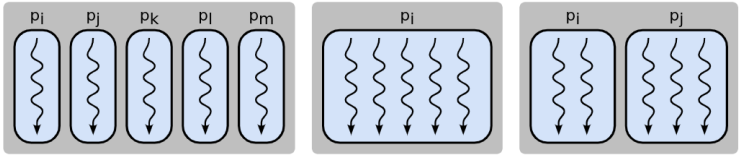


Figura 23: Comparação (Maziero, 2019).

Característica	Com processos	Com *threads* (1:1)	Híbrido
Custo de criação de tarefas	alto	baixo	médio
Troca de contexto	lenta	rápida	variável
Uso de memória	alto	baixo	médio
Compartilhamento de dados entre tarefas	canais de comunicação e áreas de memoria compartilhada.	variáveis globais e dinâmicas.	ambos.
Robustez	um erro fica contido no processo.	um erro pode afetar todas as *threads*.	um erro pode afetar as *threads* no mesmo processo.
Segurança	cada processo pode executar com usuários e permissões distintas.	todas as *threads* herdam as permissões do processo onde executam.	*threads* com as mesmas permissões podem ser agrupadas em um mesmo processo.
Exemplos	Apache 1.*, PostGres	Apache 2.*, MySQL	Chrome, Firefox, Oracle

Figura 24: Comparação (Maziero, 2019).

Capítulo 2 (OLIVEIRA; CARISSIMI; TOSCANI, 2001)

Bibliografia

OLIVEIRA, R. S.; CARISSIMI, A. S.; TOSCANI, S. S. **Sistemas Operacionais**. 2. ed. Porto Alegre: Sagra-Luzzatto, 2001.

STALLINGS, William. **Operating Systems: Internals and Design Principles**. 6. ed. Upper Saddle River, NJ: Prentice-Hall, 2009.

TANENBAUM, Andrew S. **Sistemas Operacionais Modernos**. 3. ed. São Paulo: Pearson, 2010.

SILBERSCHATZ, Abraham; GALVIN, Peter; GAGNE, Greg. **Sistemas Operacionais: Conceitos e Aplicações**. Rio de Janeiro: LTC, 2001.

TANENBAUM, Andrew S.; WOODHULL, Albert S. **Sistemas Operacionais: Projeto e Implementação**. 2. ed. Porto Alegre: Bookman, 2000.

MAZIERO, Carlos Alberto. **Sistemas Operacionais: Conceitos e Mecanismos [recurso eletrônico]**. Curitiba: DINF - UFPR, 2019. ISBN 978-85-7335-340-2.

Estes slides estão protegidos por uma licença Creative Commons

Este modelo foi adaptado de Maxime Chupin.

Marisangila Alves, MSc
marisangila.alves@udesc.br
marisangila.com.br

UDESC
UNIVERSIDADE DO ESTADO DE SANTA CATARINA
JOINVILLE
CENTRO DE CIÊNCIAS TECNOLÓGICAS

UDESC
Universidade do Estado de Santa Catarina

2025/2

Sistemas Operacionais

Processos