Sistemas Operacionais

Processos e Threads

Prof. Dr. Paulo Ricardo Muniz Barros
paulobarros@feevale.br

Material adaptado de Prof. Dr. Gabriel Simões

FEEVALE DIGITAL

Conceitos: programa, processo?

Conceitos básicos

Programa

Conjunto de instruções numa linguagem de alto nível ou de máquina.

Processo

Um processo é uma instância de um programa de computador que está sendo executada. Um processo contém o código do programa e sua atividade atual. Dependendo do sistema operacional (SO), um processo pode ser feito de várias linhas de execução que executam instruções concorrentes, conhecidas como Threads.

Um programa de computador é uma coleção passiva de instruções, enquanto que um processo é a execução real dessas instruções. Vários processos podem ser associados com o mesmo programa. Abrir várias instâncias do mesmo programa, por exemplo, pode significar que mais de um processo está sendo executado.

Analogia

Exemplo: Preparação de bolo

• A receita →	• programa
• Os ingredientes →	• dados de entrada
• O cozinheiro →	• processador
• Atividade de preparar o bolo →	• processo

Caso o filho do cozinheiro venha a ser picado por uma abelha - interrupção e chaveamento p/ novo processo.

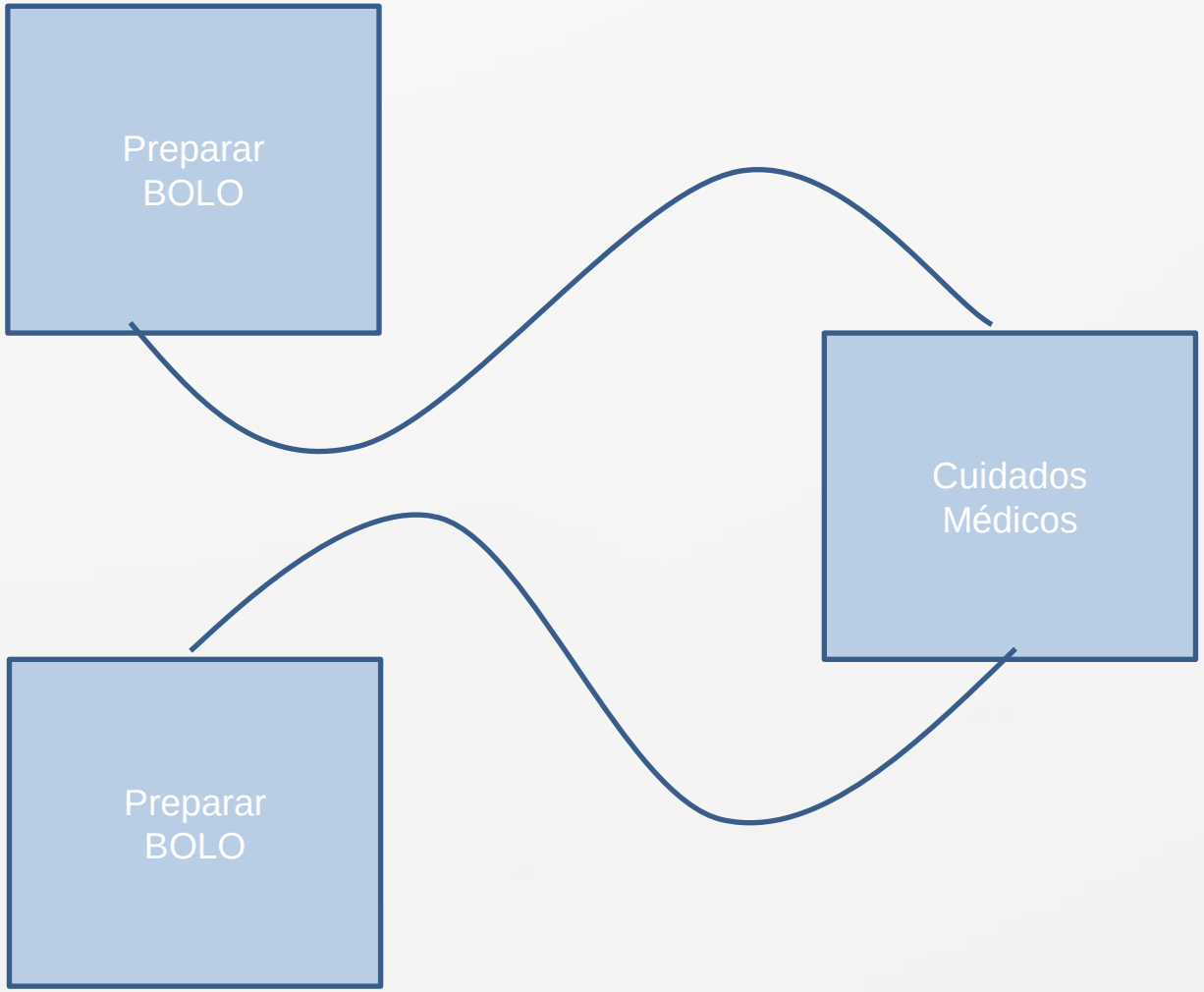
Analogia

Processo de fornecer cuidados médicos:

• Livro de primeiros socorros →	• programa
• Remédios →	• dados de entrada
• O cozinheiro →	• processador
• Atividade de cuidar da picada →	• processo

➡ Quando terminar, volta para o processo de preparar o bolo.

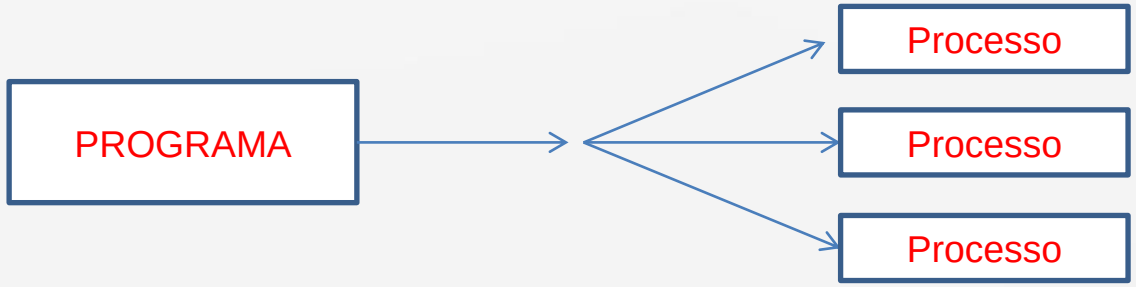
Analogia

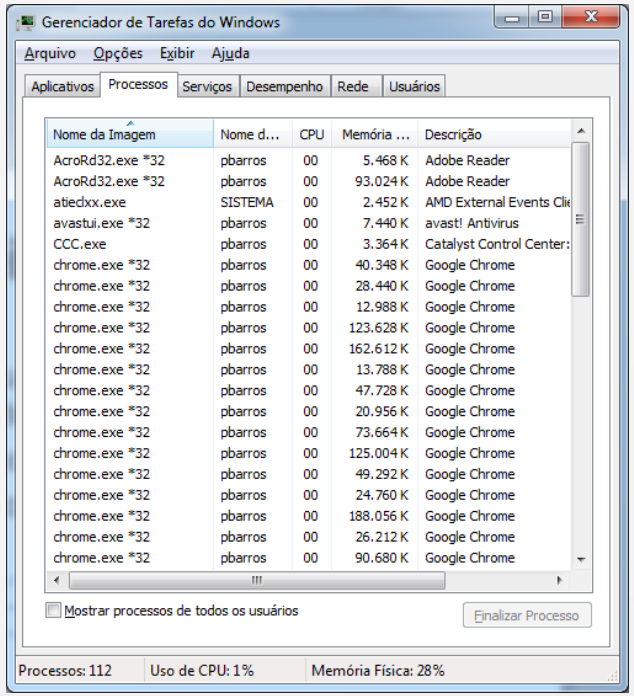


Conceitos básicos

Processo: O que o usuário enxerga como aplicação ou sistema, normalmente é composto por vários processos

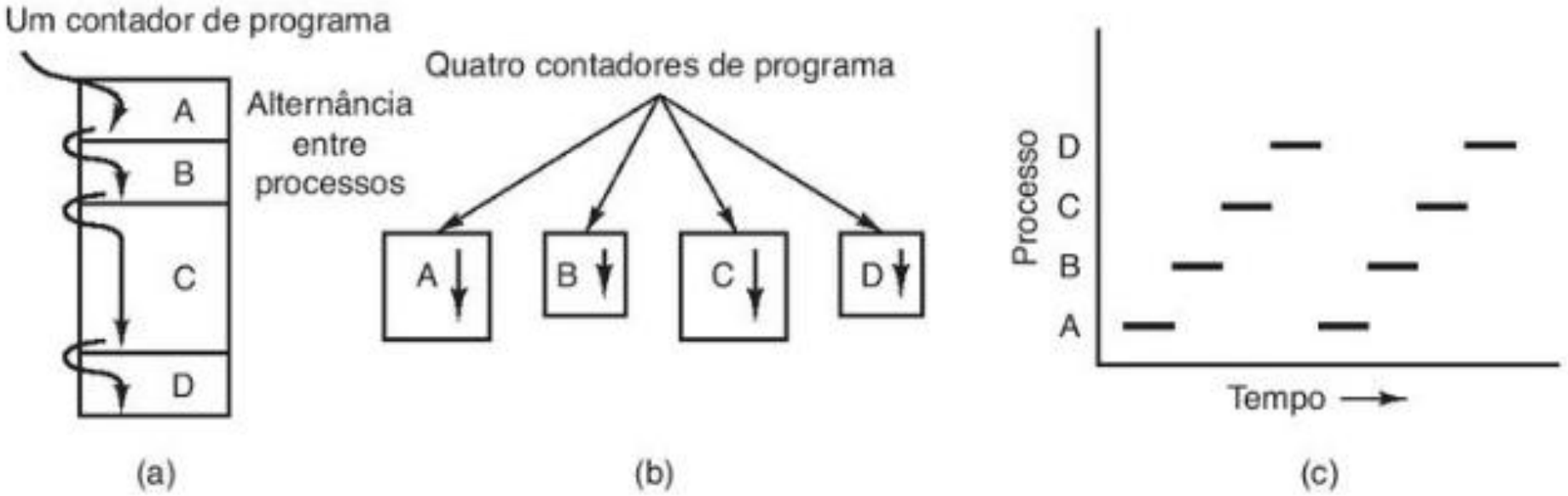
- Sequencial = 1 processo
- Concorrente = vários processos





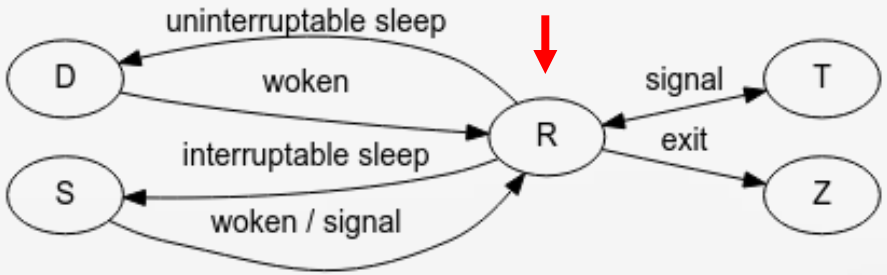
Multiprogramação

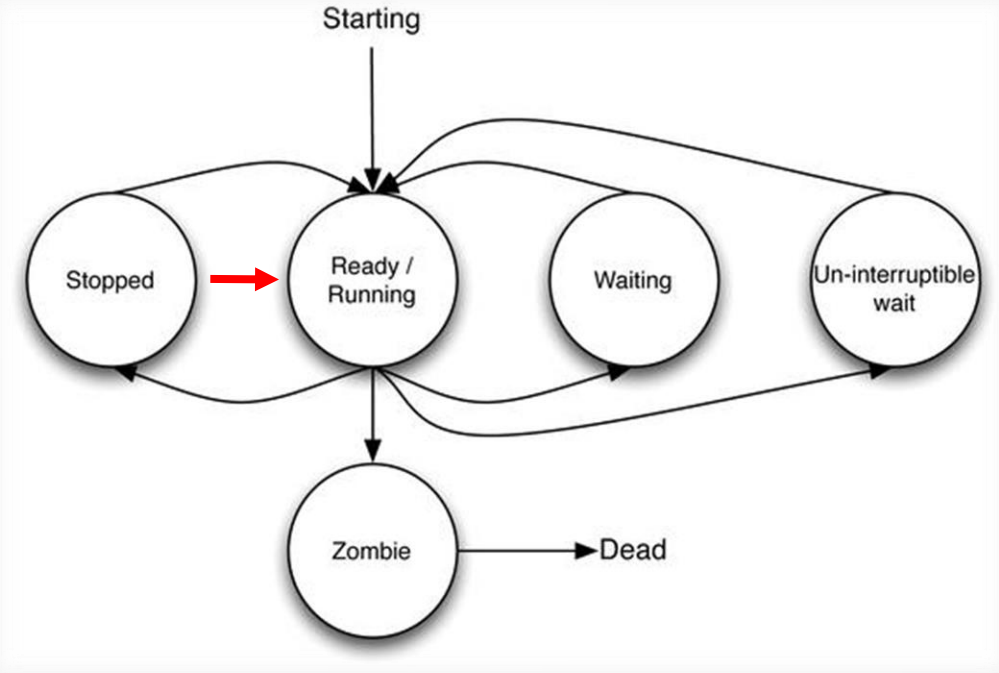
FIGURA 2.1 (a) Multiprogramação de quatro programas. (b) Modelo conceitual de quatro processos sequenciais independentes. (c) Apenas um programa está ativo de cada vez.



(FONTE: TANENBAUM,2016)

Estados de Processos

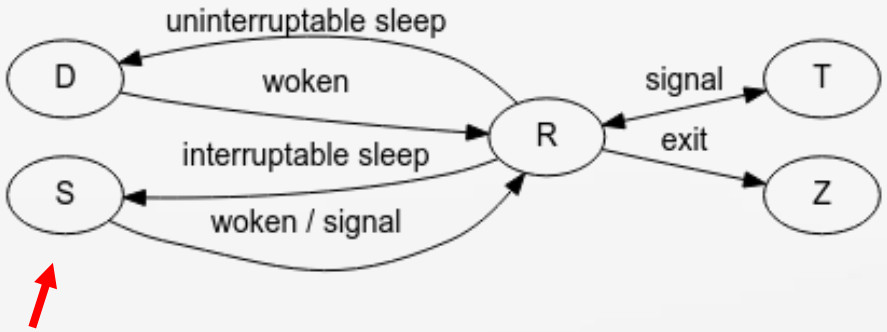


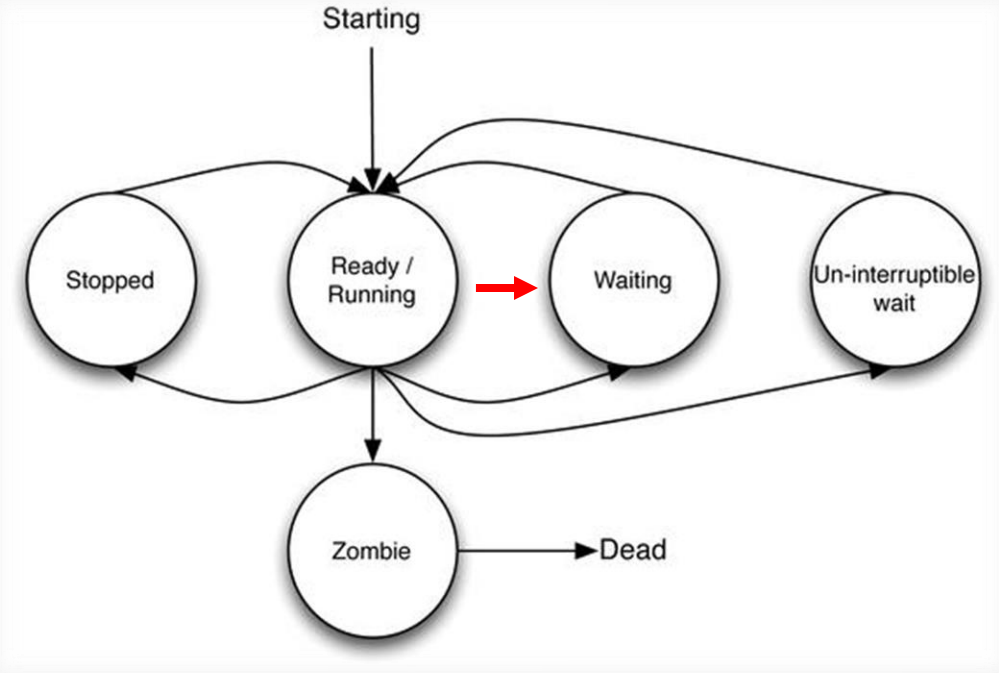


(FONTE: TANENBAUM,2016)

- R - em execução.

Estados de Processos

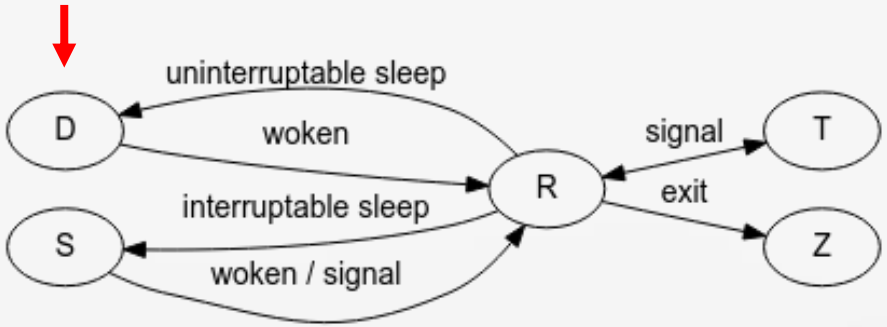


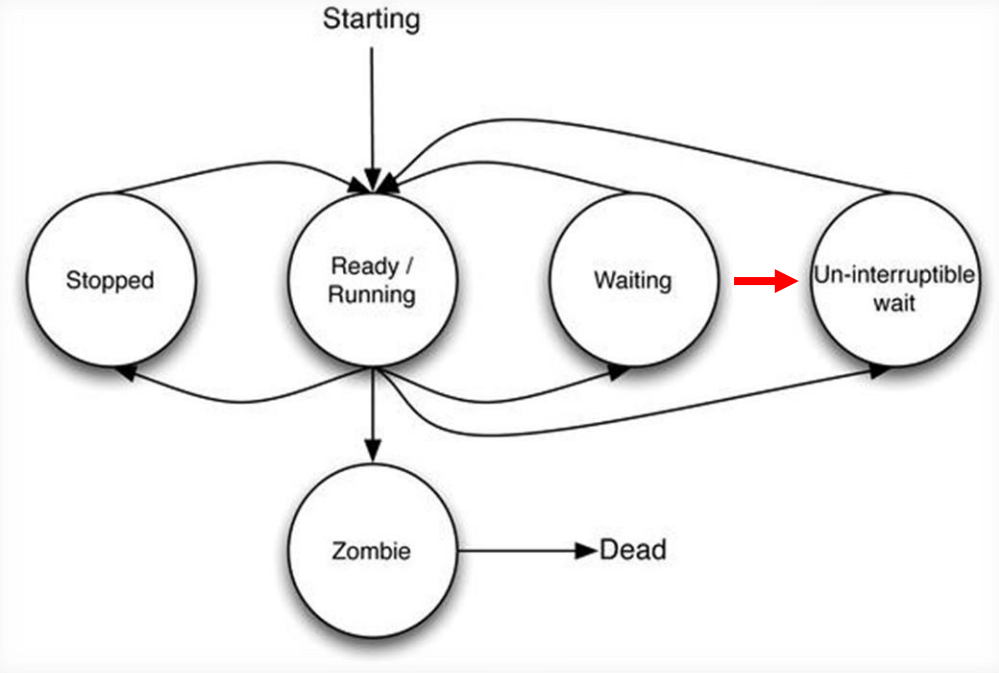


(FONTE: TANENBAUM,2016)

- S - dormindo ou interrompido.

Estados de Processos

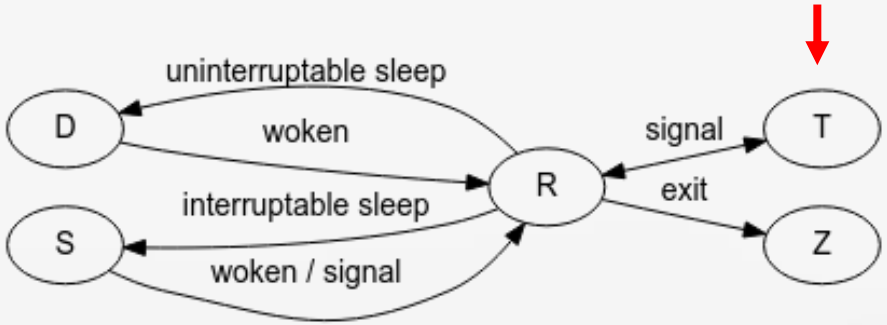


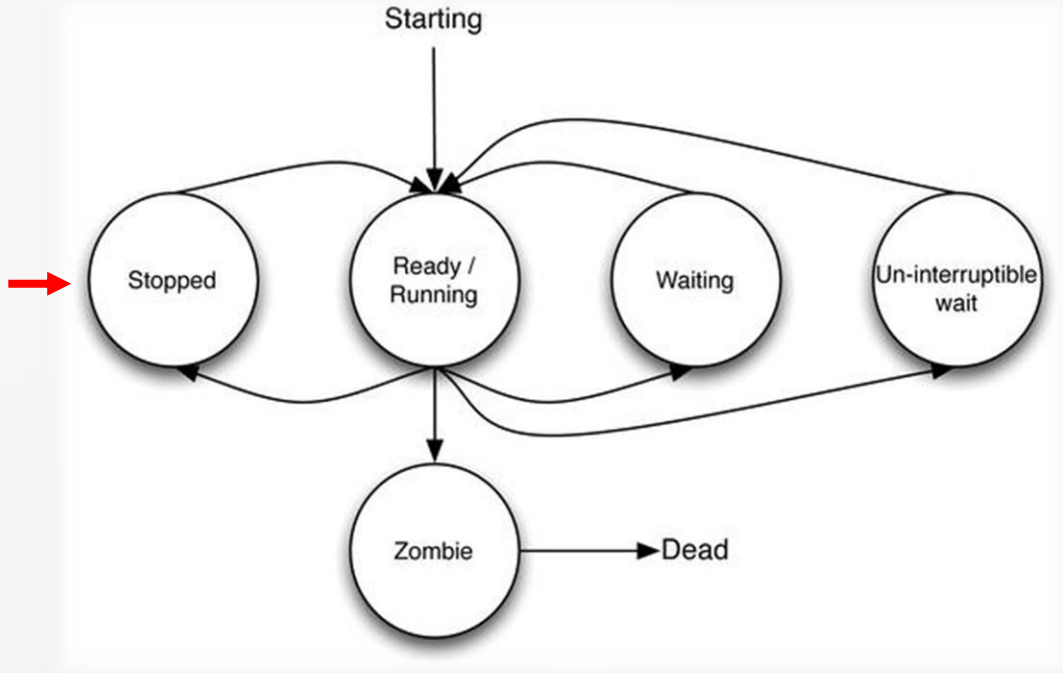


(FONTE: TANENBAUM,2016)

- D - dormindo, mas "ininterruptível".

Estados de Processos

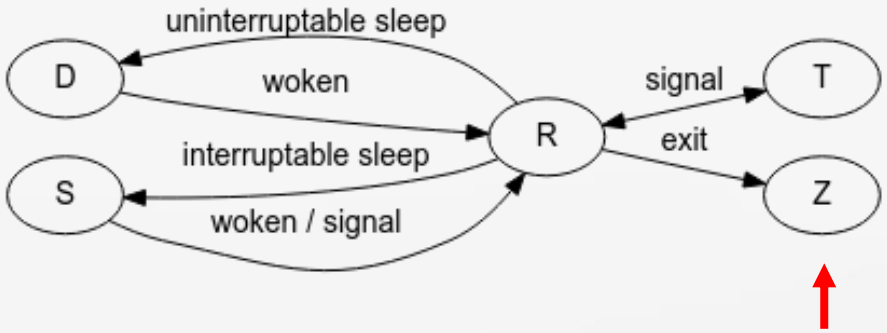


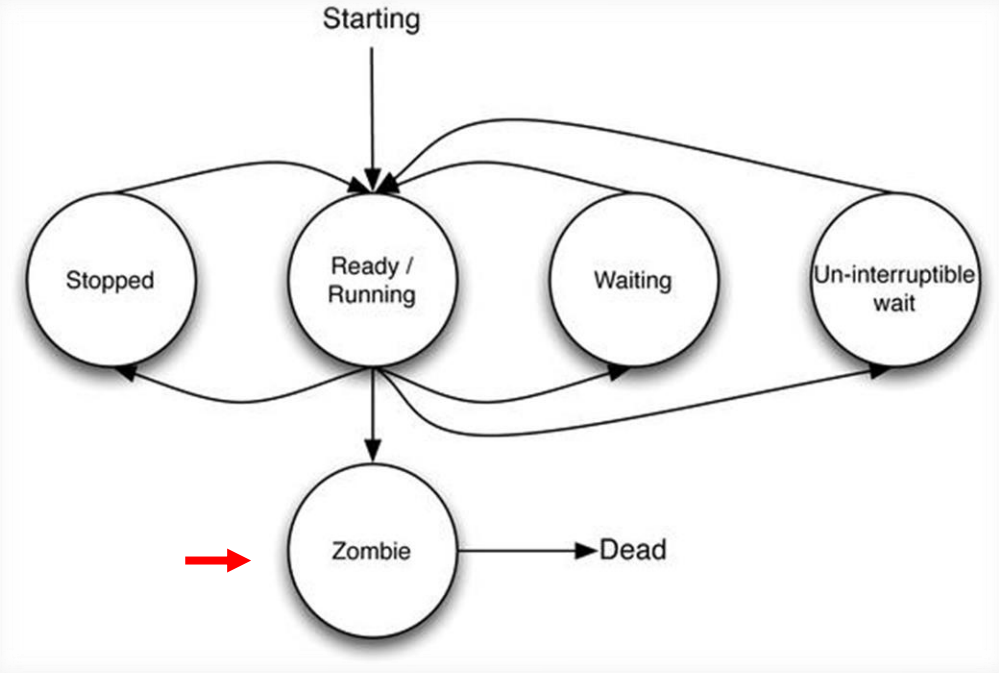


(FONTE: TANENBAUM,2016)

- `T - parado pelo controle de tarefas (CTRL-Z).`
- `t - parado pelo depurador.`

Estados de Processos





(FONTE: TANENBAUM,2016)

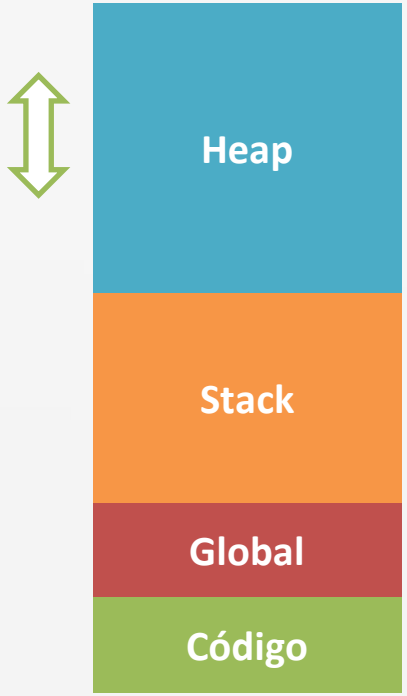
- Z - defunto/zumbi.
  Não responde as mensagens do SO.

Heap e Stack

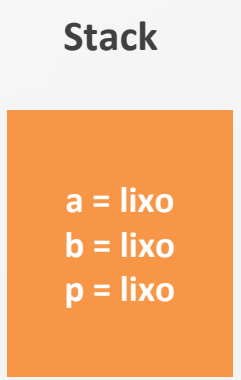
Stack é uma região de memória que armazena variáveis temporárias criadas por funções (incluindo a função main ()). Stack é uma estrutura de dados "LIFO" (última que entra é a primeira que sai), que é gerenciada e otimizada pela CPU. Toda vez que uma função declara uma nova variável, ela é "empurrada" para a Stack. Por outro lado, toda vez que uma função retorna, todas as variáveis empilhadas por essa função são liberadas (ou seja, são excluídas). Depois que uma variável da Stack é liberada, essa região de memória fica disponível para outras variáveis.

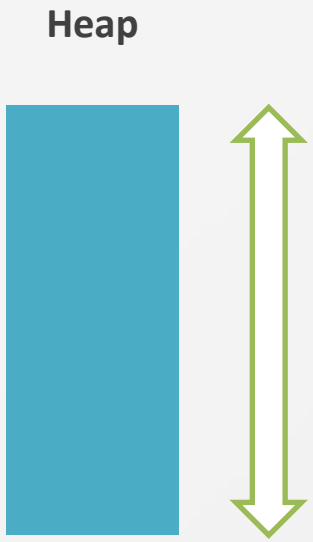
Já heap é uma região da memória que não é gerenciada automaticamente pela CPU. É uma região de memória flutuante e maior. Para alocar memória no heap deve-se usar malloc() ou calloc(), chamando primitivas de SO. Depois de alocar memória no heap, você é responsável por usar free() para desalocar a memória quando não precisar mais dela. Se você não conseguir fazer isso, seu programa terá o que é conhecido como vazamento de memória. Caso isso aconteça, a memória no heap ainda será reservada e não estará disponível para outros processos.

Heap e Stack



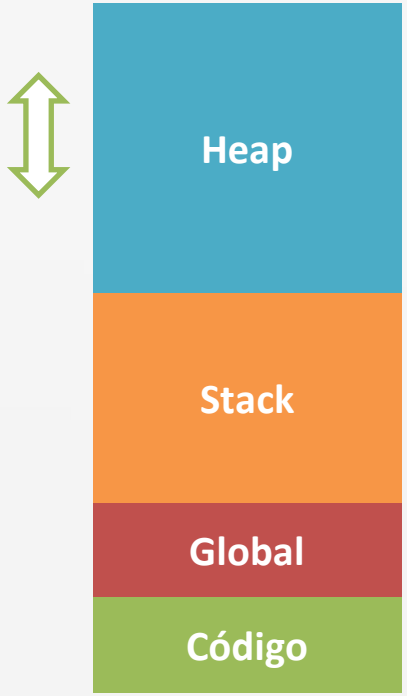

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     int a;
6     int b;
7     int *p;
8
9     a = 2;
10    b = 4;
11
12    p = (int *)malloc(sizeof(int));
13    *p = 10;
14
15    free(p);
16    return EXIT_SUCCESS;
17}
```



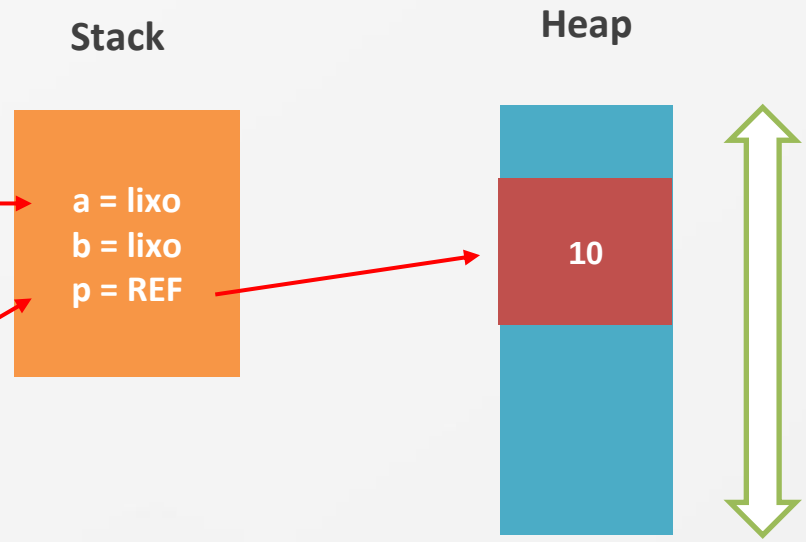


Fonte: Do autor.

Heap e Stack




```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     int a;
6     int b;
7     int *p;
8
9     a = 2;
10    b = 4;
11
12    p = (int *)malloc(sizeof(int));
13    *p = 10;
14
15    free(p);
16    return EXIT_SUCCESS;
17}
```



Fonte: Do autor.

Heap e Stack

- Verificando o tamanho da página:
  - `getconf PAGESIZE`
- Observando a lista de processos:
  - `htop`
- Observando os dados de memória de um processo:
  - `cat /proc/4226/statm`
- Verificando o tamanho limite de um stack:
  - `ulimit -s` {8MB no Ubuntu}

Threads

Um thread é um fluxo de execução através do código de processo com seu próprio contador de programa, registros de variáveis e uma pilha que contém o histórico de execução.

Um thread compartilha com seus threads pares informações como segmento de código, segmento de dados e arquivos abertos. Quando um thread altera um item de memória de segmento de código, todos os outros thread o veem.

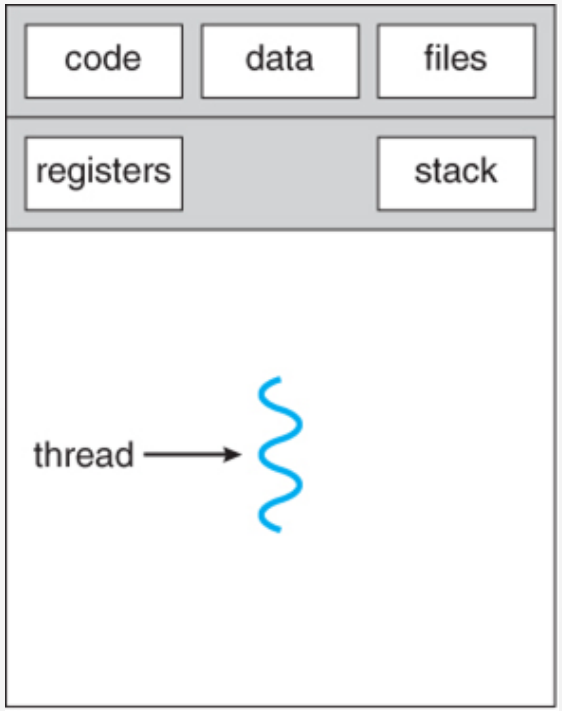
Um thread também é chamado de processo leve. Os threads fornecem uma maneira de melhorar o desempenho do programa por meio do paralelismo. Threads representam uma abordagem de software para melhorar o desempenho do sistema operacional, reduzindo a sobrecarga e melhorando o uso dos recursos de hardware.

Elementos compartilhados por Threads

Elementos do Processo	Elementos dos Threads
Espaço de Endereçamento	Contador de Programa
Variáveis globais	Registradores
Arquivos abertos	Stack
	Estado

Fonte: Do autor.

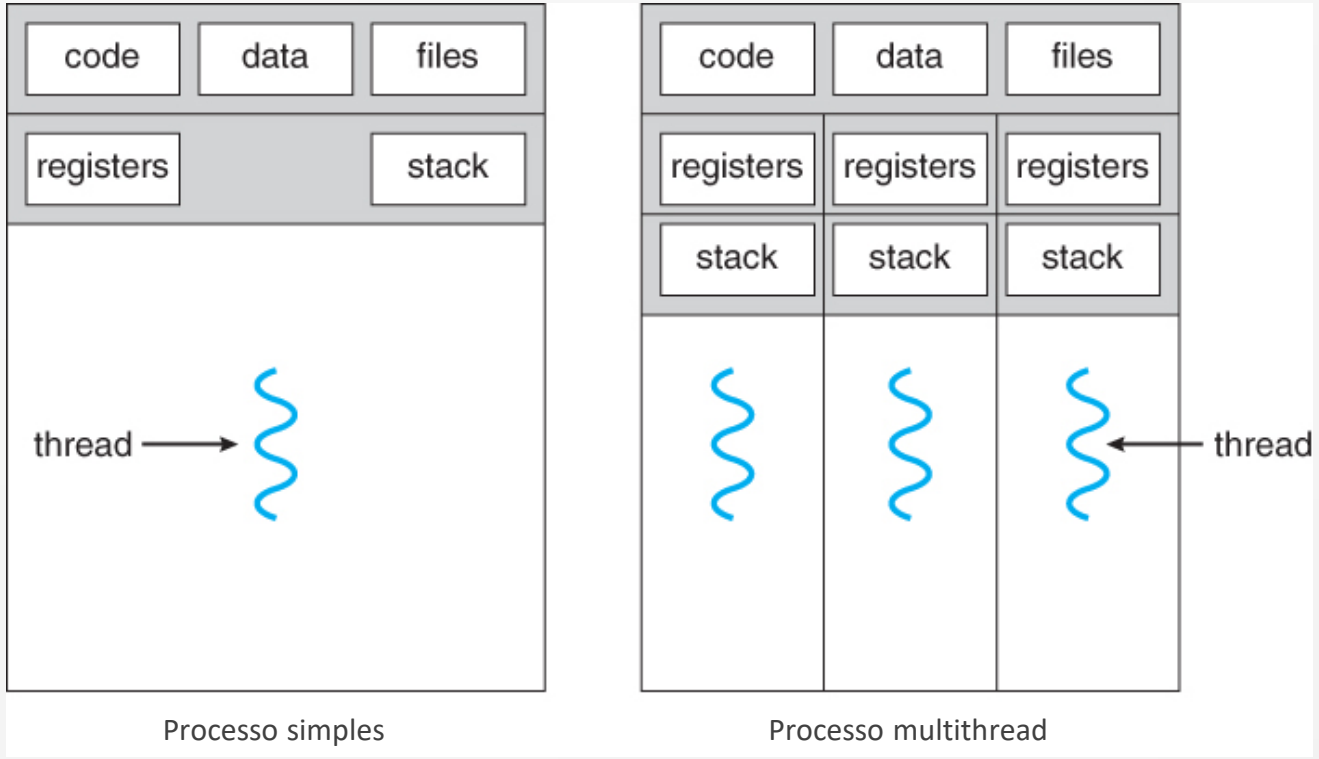
Processo Multithread



Processo simples

FONTE: SILBERSCHATZ, 2015

Processo Multithread



Processo simples

Processo multithread

FONTE: SILBERSCHATZ, 2015

Exemplo para criação de Threads

Para compilar um programa com threads é preciso apontar a lib que implementa POSIX Threads:

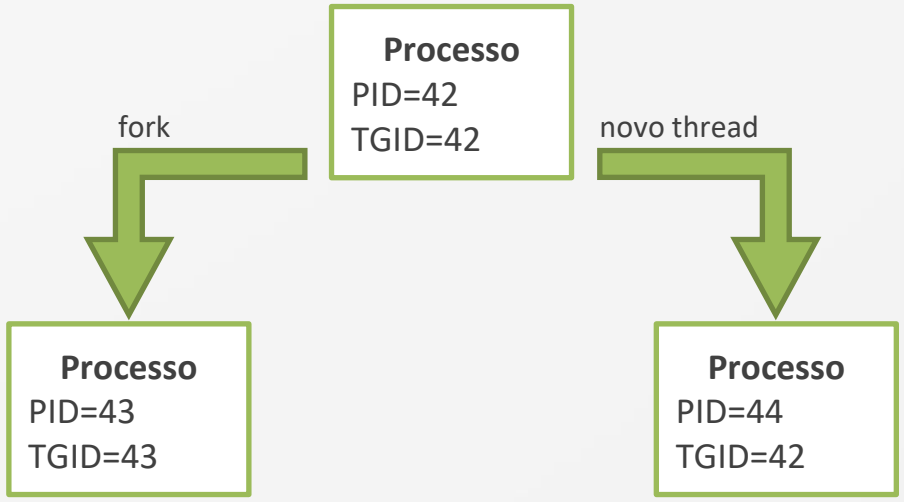
gcc fonte.c -o saida -lpthread

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <pthread.h>
4
5   void *threadPrint(void *vargp)
6   {
7           int *val = (int *)vargp;
8           for(int i = 0; i < *val; i++){
9                   printf("Valor: %d\n", *val);
10          }
11  }
12
13  int main()
14  {
15          pthread_t tid;
16          for (int i = 0; i < 10; i++){
17                  pthread_create(&tid, NULL, threadPrint, (void *)&i);
18          }
19          pthread_exit(NULL);
20          return EXIT_SUCCESS;
21  }
```

Fonte: Do autor.

PID e TGID

Como sabemos, os identificadores de processos (PID) são valores numéricos únicos que representam um processo em execução no SO. O Thread Group ID (TGID), representa um grupo de Threads ou tarefas. Note que no fluxo de exemplo, a seta a direita representa a criação de um novo thread em um processo existente. Perceba que um novo PID e uma nova representação de processo é criada, mas ao mesmo tempo o TGID não muda. Isso acontece em função do Linux reconhecer tanto processos como Threads como tarefas, tratando ambos da mesma maneira.



Fonte: Do autor.

Para listar os PIDs de threads vinculados a um processo, utilize:
ps -T -p <pid>

REFERÊNCIA

SILBERSCHATZ, Abraham. **Fundamentos de sistemas operacionais**. 9. Rio de Janeiro, LTC, 2015,recurso online. ISBN 978-85-216-3001-2.

TANENBAUM, Andrew S. **Sistemas Operacionais Modernos** - 3ª edição. Pearson, 2016. 674 ISBN 9788576052371.