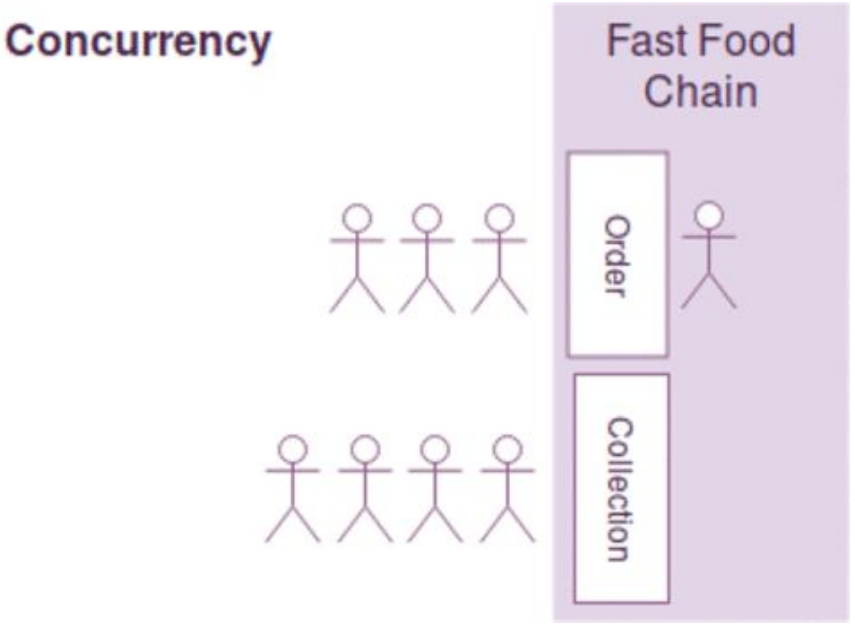
URI | UNIVERSIDADE REGIONAL INTEGRADA DO ALTO URUGUAI E DAS MISSÕES

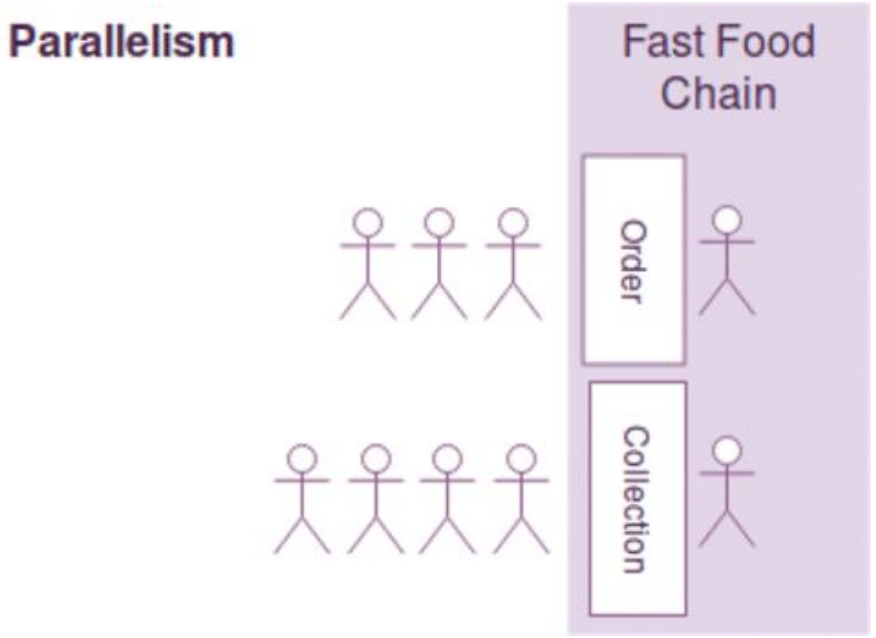
30-769

Linguagem de Programação IV

MSc. Fernando Schubert

CONCORRÊNCIA VS PARALELISMO

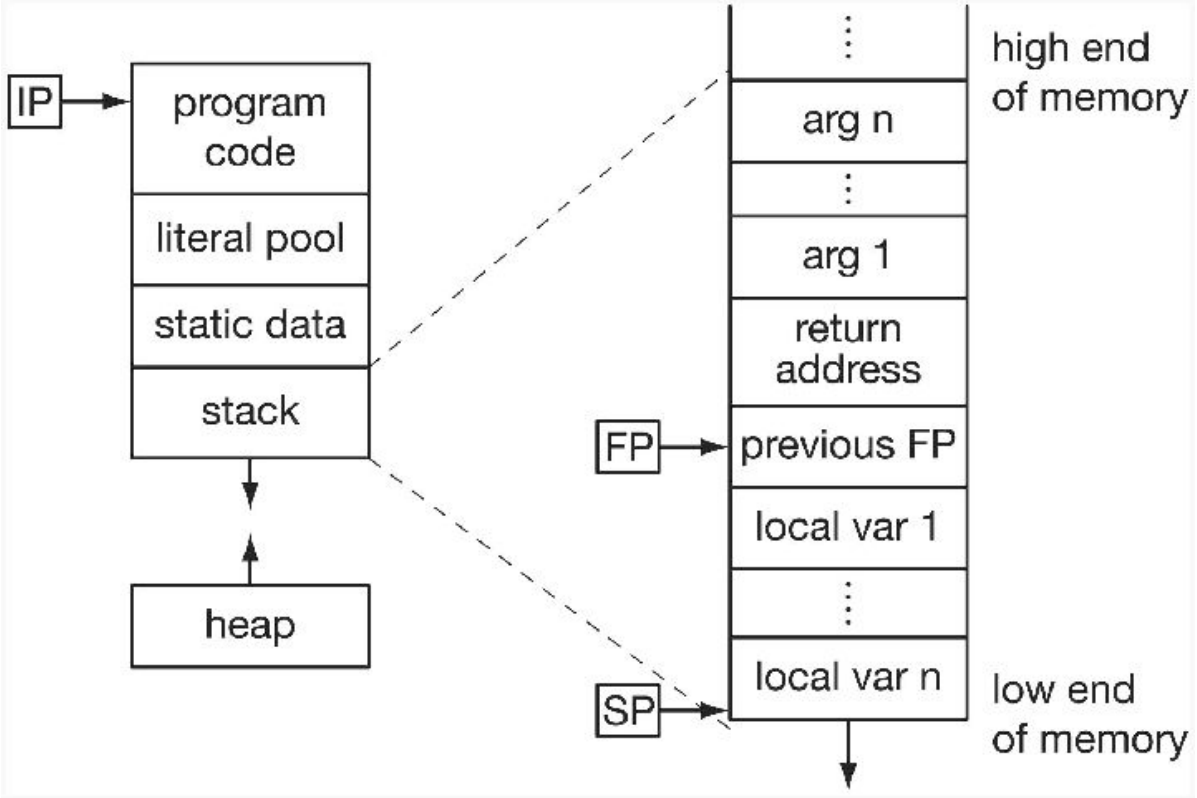




PROCESSOS

- Abstração usada pelo S.O. para designar a execução de um programa.
- É caracterizado por uma thread de execução, um estado corrente e um conjunto associado de recursos do sistema.
- Um processo é um programa individual em execução (uma instância de um programa rodando em um computador).
- É também referenciado como “tarefa” (task) ou mesmo “job”.
- O processo é uma entidade ativa (i.e., é um conceito dinâmico), ao contrário do programa.
- Cada processo é representado no SO por estruturas de controle (ex: bloco de controle de processo).

PROCESSOS


IP
program code
literal pool
static data
stack
heap
FP
SP
arg n
arg 1
return address
previous FP
local var 1
local var n
high end of memory
low end of memory

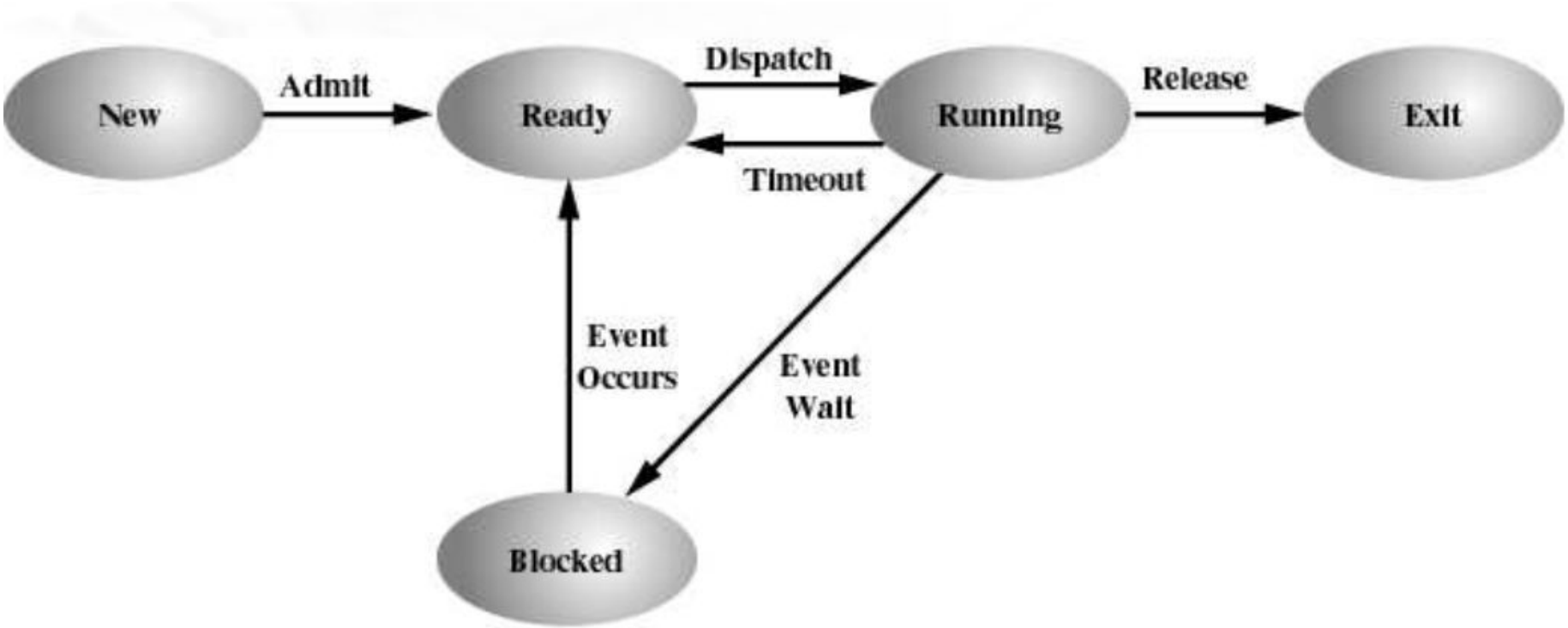
PROCESSOS

- A multiprogramação pressupõe a existência de vários processos disputando o processador.
- Execução do escalonamento
  - Tarefa de alternar a CPU entre dois processos
  - O tempo depende muito do hardware
  - Velocidade da memória, no. de registradores, instruções especias de carga de registradores. 1 a 1000 microseg.
- Troca de contexto

ESTADO DE PROCESSOS

- Durante a sua execução, um processo passa por diversos estados, refletindo o seu comportamento dinâmico, isso é, a sua evolução no tempo.
  - Exemplos de estados:
    - New: recém criado.
    - Ready: pronto para execução.
    - Running: em execução.
    - Blocked: esperando por um evento.
    - Exit: processo terminado.
- Apenas um único processo pode estar no estado “running” num dado processador, num dado instante.

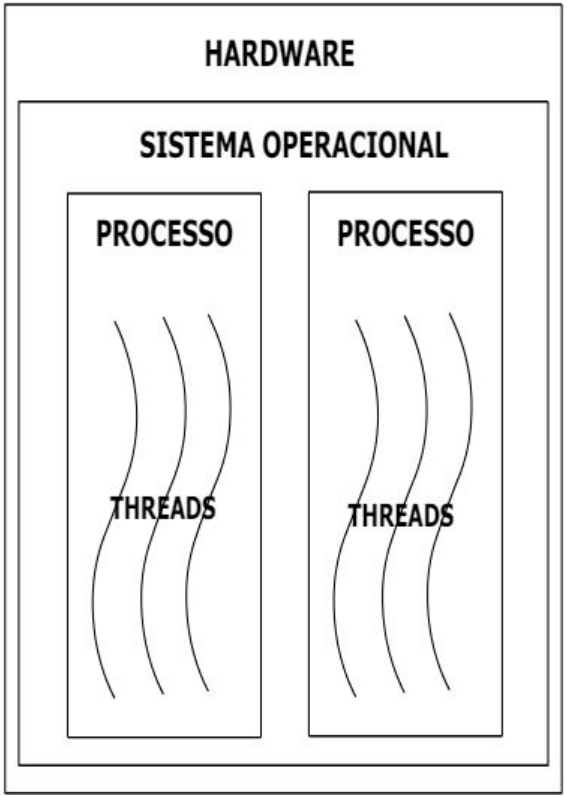
ESTADO DE PROCESSOS



THREADS

- **Unidade de Execução**: Uma thread é a menor unidade de processamento que pode ser executada por um sistema operacional. Ela representa uma sequência de instruções que o CPU pode processar de forma independente.
- **Compartilhamento de Recursos**: Threads dentro de um mesmo processo compartilham o mesmo espaço de memória e recursos, como arquivos abertos e variáveis globais. Isso permite comunicação e troca de dados entre threads de maneira eficiente.
- **Execução Concorrente**: Threads permitem a execução concorrente de tarefas dentro de um processo. Em sistemas multicore, várias threads podem ser executadas em paralelo, o que melhora o desempenho de aplicações que podem ser divididas em tarefas independentes.
- **Criação e Gerenciamento**: Threads podem ser criadas e gerenciadas pelo sistema operacional ou por bibliotecas de threads (como POSIX threads em sistemas Unix). O gerenciamento inclui a criação, sincronização e término das threads.
- **Multithreading e Desempenho**: O uso de múltiplas threads (multithreading) pode melhorar o desempenho de aplicações, permitindo que tarefas sejam realizadas simultaneamente. No entanto, isso também requer mecanismos de sincronização para evitar condições de corrida e outros problemas de concorrência.

THREADS


HARDWARE
SISTEMA OPERACIONAL
PROCESSO
THREADS
PROCESSO
THREADS

DIFERENÇAS ENTRE PROCESSO E THREAD

	Processo	Thread
Independência e Isolamento	São instâncias independentes de um programa em execução. Cada processo possui seu próprio espaço de memória (isolado), o que significa que não compartilham diretamente recursos como variáveis ou arquivos abertos, a menos que usem mecanismos de comunicação entre processos (IPC).	São unidades de execução dentro de um processo. Múltiplas threads de um mesmo processo compartilham o mesmo espaço de memória e recursos do processo, o que facilita a comunicação e o compartilhamento de dados entre elas.
Overhead (Sobrecarga)	Criar e gerenciar processos é mais pesado em termos de tempo e recursos, porque o sistema operacional precisa alocar memória e recursos separadamente para cada processo.	Criar e gerenciar threads é mais leve, pois elas compartilham o espaço de memória e recursos do processo pai, reduzindo a sobrecarga em comparação com a criação de processos.
Comunicação	A comunicação entre processos (IPC) é mais complexa e lenta, pois envolve mecanismos como pipes, filas de mensagens, ou memória compartilhada.	Como threads dentro de um mesmo processo compartilham o mesmo espaço de memória, a comunicação entre elas é mais rápida e simples.
Falhas	Se um processo falha, ele geralmente não afeta outros processos, pois cada um tem seu próprio espaço de memória isolado.	Se uma thread falha (por exemplo, causando um erro de segmentação), pode afetar todo o processo, pois todas as threads compartilham o mesmo espaço de memória.
Execução	Cada processo pode conter uma ou mais threads. Processos são executados de forma independente e o sistema operacional gerencia sua troca através de um mecanismo de escalonamento.	São executadas de forma concorrente dentro do mesmo processo. Em sistemas multicore, múltiplas threads podem ser executadas simultaneamente em diferentes núcleos.

DIFERENÇAS ENTRE PROCESSO E THREAD

Processo

- É pesado
- Espaço de endereçamento próprio
- Comunicação interprocesso é cara e limitada
- Troca de contexto é custosa

Thread

- É leve
- Espaço de endereçamento compartilhado
- Comunicação interprocesso é barata
- Troca de contexto é barata

fork()

- A chamada de sistema `fork` é usada para criar um novo processo em sistemas Linux e Unix, chamado de processo filho, que é executado simultaneamente com o processo que faz a chamada `fork()` (processo pai).
- Após a criação de um novo processo filho, ambos os processos executarão a próxima instrução após a chamada do sistema `fork`.
- O processo filho usa o mesmo contador de programa (PC), os mesmos registradores da CPU e os mesmos arquivos abertos que são usados no processo pai.
- A chamada `fork()` não recebe parâmetros e retorna um valor inteiro.
- Abaixo estão os diferentes valores retornados pela função `fork()`.
    - Valor negativo: A criação do processo filho foi malsucedida.
    - Zero: Retornado para o processo filho recém-criado.
    - Valor positivo: Retornado para o processo pai ou chamador. O valor contém o ID do processo do processo filho recém-criado.

fork() E join()

- O fork especifica que uma designada rotina deve iniciar sua **execução concorrentemente** com o **invoker**
- **join** permite que o invoker aguarde pelo término da rotina invocada

```
function F return is ...;
procedure P;
      ...
      C:= fork F;
      ...
      J:= join C;
      ...
end P;
```

- Depois do fork, P e F executarão concorrentemente. No ponto do join, P aguardará até que F tenha terminado (se não o tiver feito)

wait()

- Uma chamada para `wait()` bloqueia o processo que a invoca até que um de seus processos filhos termine ou até que um sinal seja recebido.
- Após o processo filho terminar, o processo pai continua sua execução após a instrução da chamada do sistema `wait()`.
- O processo filho pode terminar devido a qualquer uma das seguintes razões:
  - Ele chama `exit()`;
  - Ele retorna (um `int`) a partir da função `main`;
  - Ele recebe um sinal (do sistema operacional ou de outro processo) cuja ação padrão é terminar.