Aula 2

Gerência de Processos

Gerência de Processos

- O que são Processos e *Threads*?
- Porque são necessários?
- Como são implementados?
- Como são controlados?
  - **Escalonamento**

Referências:

- Capítulo 4: 4.1 a 4.5

Concorrência

Porque executar vários programas simultaneamente?

- Permitir que vários usuários usem a máquina ao mesmo tempo
- Multiprocessadores: completar a tarefa mais rapidamente
- Muitas coisas acontecem ao mesmo tempo:
  - E/S
  - Tempo passa
  - Múltiplos usuários
  - ... e o SO tem que coordenar estas atividades

Multiprocessamento resolve estes problemas

É implementado usando-se **processos** e ***threads***

Processos

Um processo é um **fluxo de controle seqüencial** e seu **espaço de endereçamento**

Informalmente, um processo é a execução de um programa junto com dados usados por ele

Pontos importantes:

- Execução seqüencial
- Tudo o que pode interagir com o processo (espaço de endereçamento):
  - Registradores
  - Memória usada
  - Arquivos

Threads

Um processo tem duas partes:

- Ativa: fluxo de controle
- Passiva: espaço de endereçamento

Um ***thread*** consiste somente no fluxo de controle:

- Também chamado **processo leve** (diet?)

Porque?

- Espaço de endereçamento custa caro
- Muitas vezes processos usam dados compartilhados, e usar-se vários threads no mesmo espaço de endereçamento é mais eficiente

Nomenclatura

Processo (Unix):

Espaço de endereçamento + fluxo de controle

Thread:

Fluxo de controle

Tarefa (*task*):

Espaço de endereçamento

- Um processo é uma tarefa com um único *thread*
- Uma tarefa pode conter vários *threads*
- Um *thread* pertence a uma única tarefa

Confusão: frequentemente se usa processo ou tarefa quando se quer dizer *thread*

Implementação

Tarefa contém:

- Espaço de endereçamento virtual
- Ambiente (arquivos, nomes, etc)

Thread contém:

- Registradores (incluindo PC)
- Pilha de execução (*stack*)

Todos os threads de uma tarefa acessam a mesma memória

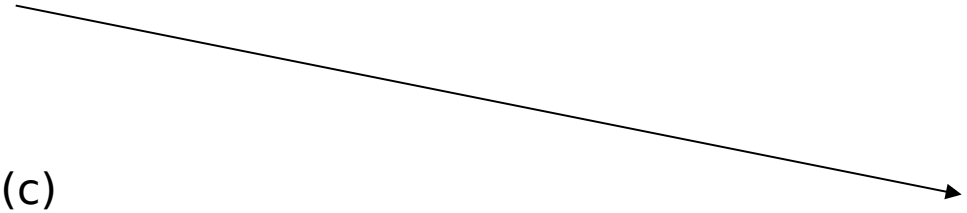
Até a pilha de execução de um *thread* pode ser acessada por outro *thread*

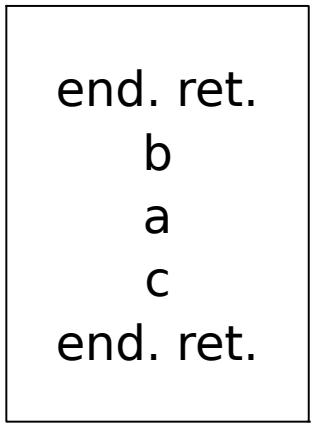
Pilhas de Execução

```
p2()
{
};

p1(c)
int c;
{
  int a,b;
  p2();
};

main()
{
p1(c);
};
```





Pilha de execução do processo

Diversas combinações

Uma tarefa, um thread:

- SOs simples como o DOS: um usuário faz uma coisa por vez

Uma tarefa, vários threads:

- Núcleo do SO: coordena diversas atividades, mas proteção não é necessária

Várias tarefas, um *thread* por tarefa:

- Unix original
- Diversos usuários com um programa cada

Várias tarefas, vários threads por tarefa:

- Caso mais comum

Vários *threads*, uma CPU?

Cada thread tem a ilusão de ter uma CPU dedicada, mas só existe uma CPU.

O **Escalonador** é o programa que controla qual thread executa a cada instante:

```
while (true) {
espera_evento();
salva_thread_corrente();
escolhe_novo_thread();
carrega_novo_thread();
}
```

Escalonador

Cada thread é representado por um **PCB** – *Process Control Block* – que contém:

- Estado do processo: *Running*, *Ready* e *Blocked*;
- Número do processo (*thread*);
- *Program Counter*;
- Registradores;
- Localização da pilha de execução;
- Priodidades de execução;
- etc...

Escalonador

```
while (true) {
espera_evento();
salva_thread_corrente();
escolhe_novo_thread();
carrega_novo_thread();
}
```

Eventos podem causar reescalonamento:

Internos:

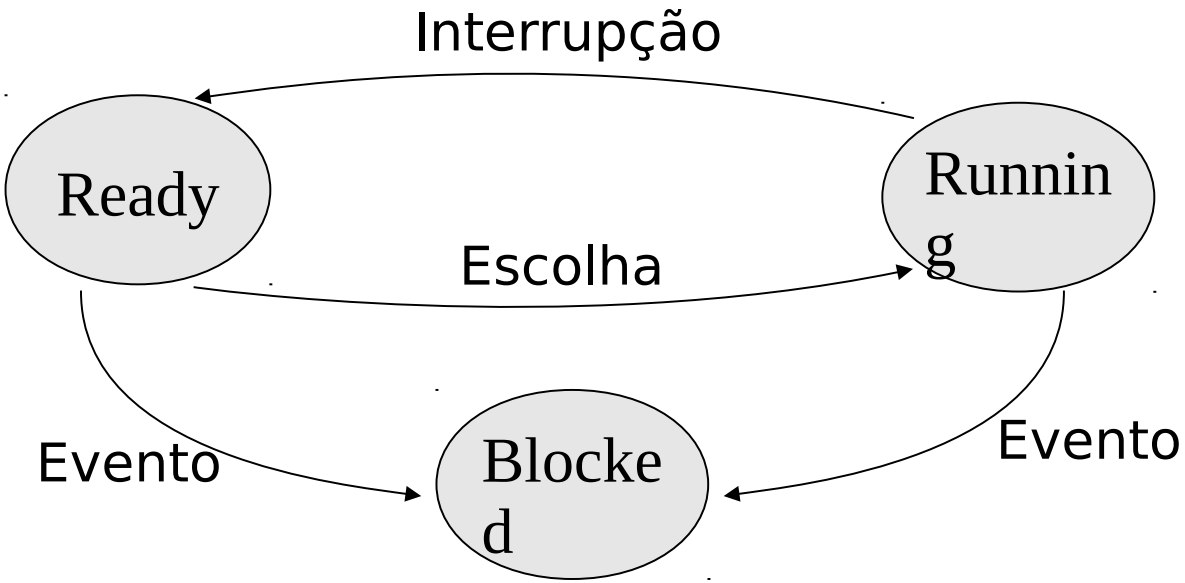
- *Thread* espera por E/S
- Thread espera por outro thread
- Thread resolve “dormir”

Externos:

- Interrupções de E/S
- Timer: o thread corrrente já executou demais

Escalonador

O escalonador altera o estado do *thread*:



- *Ready*: Pronto para executar
- *Running*: Executanto
- *Blocked*: Esperando por algum evento

Escalonador

```
while (true) {
espera_evento();
salva_thread_corrente();
escolhe_novo_thread();
carrega_novo_thread();
}
```

Salva o estado do *thread* corrente no **PCB**

- Registradores;
- PC;
- Localização da pilha de execução

Escalonador

```
while (true) {
espera_evento();
salva_thread_corrente();
escolhe_novo_thread();
carrega_novo_thread();
}
```

Escolher um thread entre os que estão *Ready*. Várias políticas de escalonamento existem:

- FIFO
- Round-robin
- Priodidades (fixas ou dinâmicas)
- ...

Escalonador

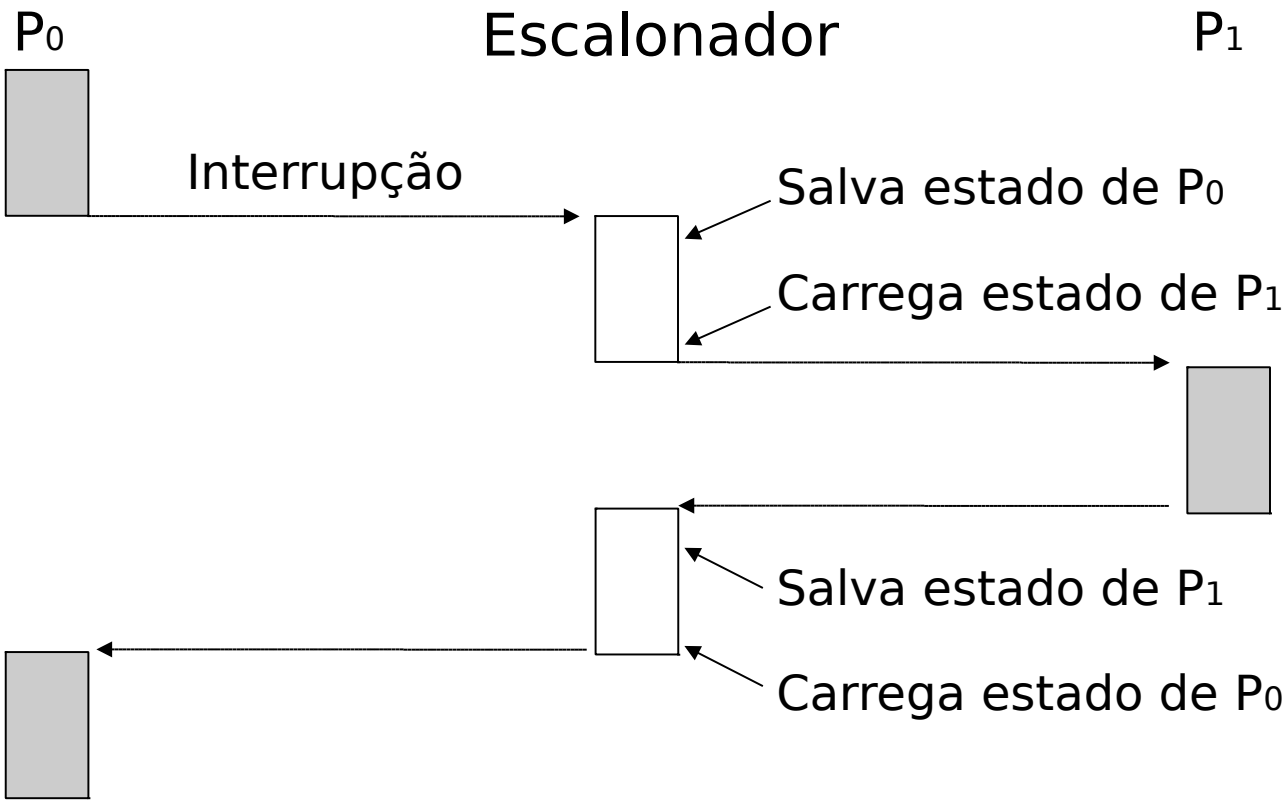
```
while (true) {
espera_evento();
salva_thread_corrente();
escolhe_novo_thread();
carrega_novo_thread();
}
```

Carrega o estado *thread* corrente do PCB e coloca nos registradores

Salvar o estado do thread antigo e carregar o estado do próximo é chamado **troca de contexto**:

- *Overhead* alto
- Necessário para proteção
- Ocorre com frequência muito alta
- Tem que ser otimizado

Escalonador



Resumo

Concorrência: **Processos** e ***threads***

- Processo: fluxo de controle + espaço de endereçamento
- Thread: fluxo de controle

Implementação via **Escalonador**:

- Estado do thread: *Running*, *Ready*, *Blocked*
- Escalonador decide qual thread será executado:
  - Troca é causada por eventos;
  - Escolha de novo thread depende de política de escalonamento;
  - Troca de contexto