Encontro 2 - Processos

Programas em execução. Entidade dinâmica. Acessa recursos de Hardware.

- **Algoritmo**: sequência de instruções que dada entrada chega em uma saída;
- **Programa**: algoritmo implementado em alguma linguagem de programação;
- **Processo**: Execução de uma versão binária de um programa.

Sistema operacional (SO) da suporte ao processo em sua criação, execução, gerencia dos processos e **arbitragem de recursos de hardware entre os processos.**

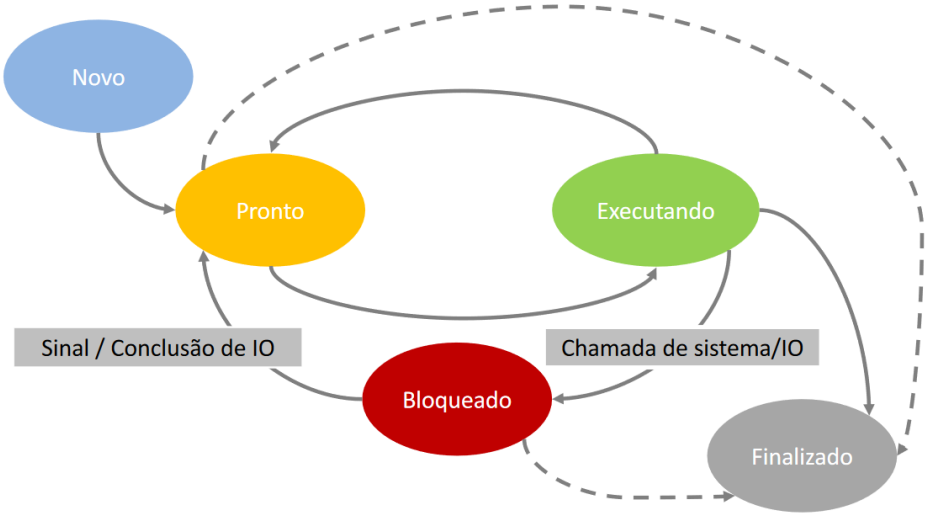
- Processo do ponto do SO:

- Código binário: gerado por compiladores;
- Recursos de hardware: Ram, disco, Entrada/Saída;
- Meta-dados de gestão: Id do processo(PID), identificador de prioridade, estatísticas de uso de recursos, entre outros.

Processo é representado no SO por um *Process Control Block* (PCB), que contém todas características do processo. Processos em *Foreground* são priorizados pelo SO/CPU. Exstem muitos processos ao mesmo tempo devido a: multi-usuários, logo cada usuário cria vários processos, também, SO tenta maximizar os recursos, tenta garantir zero ócio da CPU, possuir vários processos significa que sempre terá algo para se fazer.

- Estado e contexto:

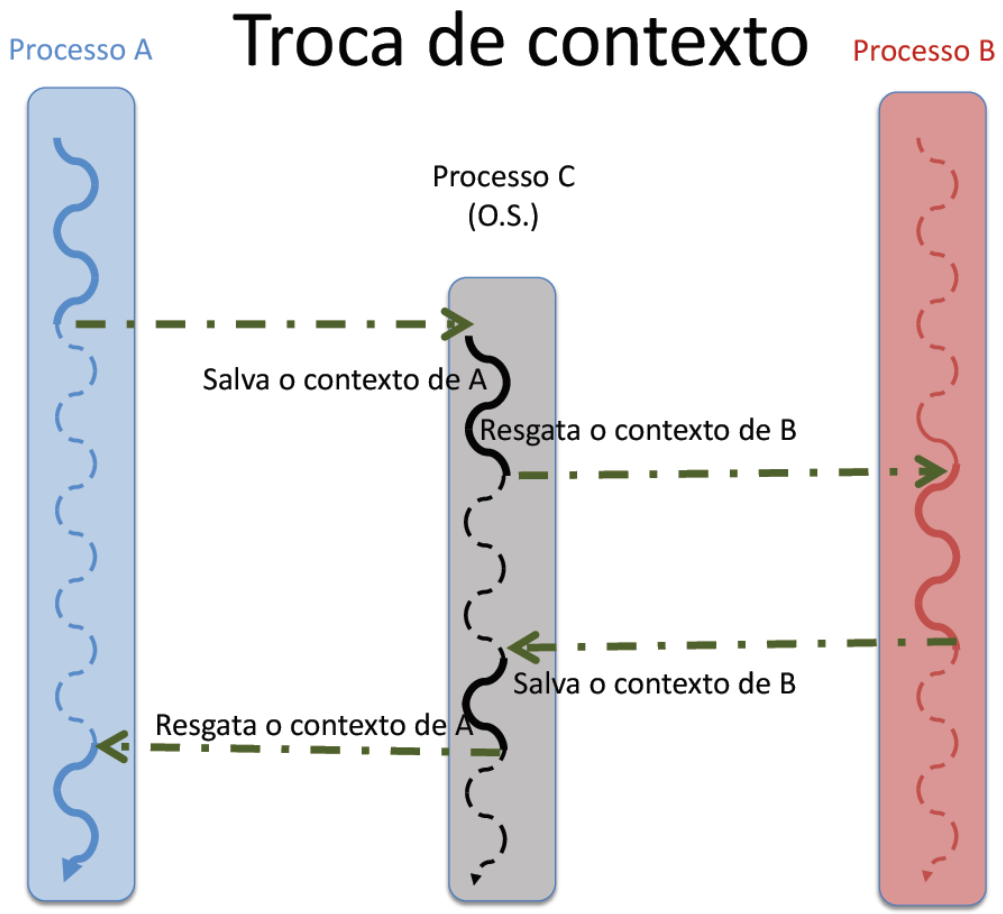
Ciclo de vida de um processo:



- **Novo**: processo criado, recursos alocados e disponíveis;
- **Pronto**: Pode ser executado a qualquer momento desde que a CPU esteja disponível;
- **Executando**: CPU liberada, um processo pronto começa a executar. Pode voltar para ao estado pronto por estar tempo demais executando e outros necessitam de acesso a CPU;

- **Bloqueado**: Espera de recursos de Hardware estar disponível. Recurso não disponível, acesso ao recurso leva tempo, operação terceirizada para outro dispositivo.
- **Finalizado**: Terminou de vez de executar (fim de código ou espontaneamente). Processo realiza chamadas indevidas (sem direito) a recursos, logo é finalizada forçadamente pelo SO.

- Troca de Contexto: (leva tempo)



Contexto: Toda descrição de recursos que o processo está/estava utilizando que caracteriza a execução do processo.

O sistema operacional também é um processo. Nem todos processos nascem iguais. Nem todos tem os mesmo direitos a recursos. Existem no mínimo dois níveis de direito:

1. Modo usuário: menos direitos;
2. Modo Kernel(núcleo): mais direitos e podem controlar outros processos. Podem mudar a prioridade, tipo de recursos que acessam, tipo de interação com outros processos de outros processos.

- Hierarquia de processos:

1. Boot do Hardware executa uma instrução inicial lida de algum setor específico do disco;
2. Esta instrução gera um processo de boot “processo boot” que testa recursos do hardware e chama outros processos do sistema;

3. Os processos do sistema gerenciam as funções necessários de hardware (que são outros processos);
4. Após todos *checks* do sistema é chamado um processos de login;
5. Quando o login for realizado novos processos são criados;
6. Usuário logado gerará mais processos para suprir suas demandas;
7. E, assim, obtém-se uma árvore de processos.

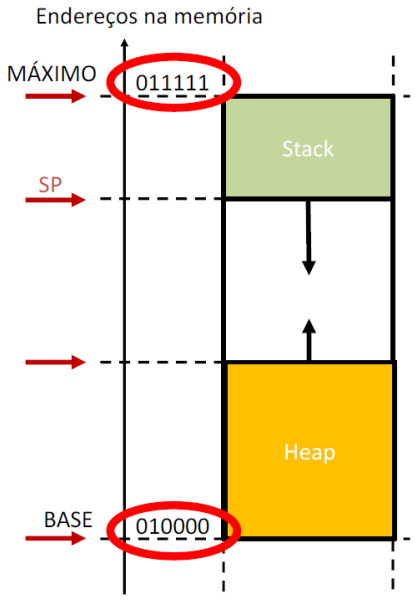
Encontro 3 - Implementação de processo:

- Processo do ponto de vista do OS:

1) Código executável;
2) Recursos de hardware utilizados (Ram, disco, E/S);
3) Metadados de gestão.

- Recursos acessados pelo processo:

- Memória
    - Segmento de código (próxima instrução)
    - Dividida em 2 partes (entre endereço base e máximo):
        - Espaço de *Heap*: Variáveis globais, variáveis alocadas dinamicamente, objetos criados durante execução. Cresce para cima, começa no base, soma deslocamento positivo para aumentar.
        - Espaço de pilha (*Stack*): Trechos de memória que são reservados para procedimentos (funções) e rotinas, para alocar as variáveis locais a este procedimento. Cada procedimento é empilhado para armazenar as variáveis do escopo. Cresce para baixo, começa no máximo, subtrai deslocamento positivo para aumentar.
        - *Stack* e *Heap* são alocados dinamicamente: No caso da *Stack* é para cada chamada de procedimento. No *Heap*, é para cada nova declaração de pointeiros, cada criador e destrutor de objetos. Tamanho do Heap costuma ser muito maior que o do *Stack*.
        - Mantém-se dois registradores com o endereço do topo do *Stack* (SP = *Stack Pointer*) e do topo do *Heap*;
        - Cuidado para o topo do *Heap* não ultrapassar o topo do *Stack* e vice-versa;



- Arquivos abertos:
    - Deve-se manter uma noção de quais arquivos estão abertos, ou foram utilizados pelo processo.
- Meta dados de escalonamento:
    - Qual ou quais cores de uma CPU que o processo está executando;
    - Executar mais de uma Thread.
- Meta dados:
    - PID: Número único associado ao dado processo;
    - Usuário associado ao dono do processo para conseguir restringir o que o processo pode fazer dadas permissões do usuário na máquina;
    - Prioridade;
    - Estatísticas sobre a execução do processo, para conseguir "prever o futuro" com base no histórico de utilização de recursos do processo.

- Tabela de processos (*Process Table*):

- Estrutura de dados que descreve os processos para podermos visualizar informações relevantes sobre os processos que estão em execução.;
- Em geral é uma tabela *Hash* onde o PID serve de chave;
- Tem que ser grande;
- Deve se manter na tabela também ponteiros de cada processo para seu processo pai e processos filhos (árvore).
- Possibilita acesso em tempo constante a cada processo.

- PCB (*Process Control Block*):

- Uma estrutura(ou classe) que contém informações relevantes para o funcionamento dos processos.
- Estas informações podem ser:
    - Prioridade de execução;
    - Tempo de execução restante;
    - Tempo de execução que utilizou no passado;

- Tempos de execução em modos de execução diferentes: *user*, *system*, *root*;
- Próximo processo pronto a ser executado (caso do minix);
- Nome do processo;
- Lista encadeada de processos que são acessíveis pelo processo;
- Descritor de arquivos abertos;
- Identificador do processo (PID);
- Estado do processo;
- "Familia": pais, filhos, irmãos, etc;
- Data de criação;

Obs: Um programa pode gerar mais de um processo, normalmente gera vários, para realizar a sua execução.

Encontro 4 - Programação Sistema

- getpid()

Função que retorna o PID(inteiro) do processo. Para utilizar esta função é necessário dar *include* nas bibliotecas <unistd.h> e <sys/types.h>.

- fork()

Função que cria um processo clone do processo que chamou-a. Todo contexto até a chamada do *fork* é copiado para o novo processo, gerando somente um novo PID no PCB para o processo criado. Após a criação do novo processo, o fluxo segue normalmente para ambos na instrução imediatamente posterior a chamado do *fork* (literalmente clones um do outro). Para controlar o fluxo de ambos processos após a chamada do *fork* (caso queira que eles tenham fluxos diferentes), deve-se trabalhar com o retorno que, para o processo novo(FILHO), é 0 e, para o processo antigo(PAI), é o número do PID do novo processo criado. Para utilizar esta função é necessário dar *include* nas bibliotecas <unistd.h> e <sys/types.h>.

```c
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>
#include <sys/types.h>

int fibo(int n) {
    if (n<=1)
        return n;
    else
        return fibo(n-2) + fibo(n-1);
}

int main(int argc, char** argv){
    int resultado = 0;
    int fk = -1;
    int arg = 5;
    int meu_pid = -1;
```

```
    fk = fork();

    if(fk != 0)
        printf("Processo pai PID = %d\n", getpid());
    else
        printf("Processo filho PID = %d\n", getpid());

    meu_pid = getpid();
    resultado = fibo(arg);
    printf("Fibo(%d) = %d\n", arg, resultado);
    printf("Resultado oferecido pelo processo de PID = %d\n", meu_pid);
    return 0;
}
```

- wait()

Função que realiza a espera de término de pelo menos um dos processos filhos. Para utilizar esta função é necessário dar *include* nas bibliotecas <unistd.h> e <sys/wait.h>.

- waitpid()

Função que realiza a espera de término de um dado filho pelo seu PID. Para utilizar esta função é necessário dar *include* nas bibliotecas <unistd.h> e <sys/ wait.h>.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(){
    pid_t pid1, pid2;
    FILE* arq;
    int status1, status2;

    arq = fopen("output.txt", "w");
    pid1 = fork();
    if(pid1 == 0) {
        pid2 = fork();
        if(pid2 == 0)
            fprintf(arq, "Eu sou ");
        else {
            wait(&status2);
            //waitpid(pid2, &status2, 0);
            fprintf(arq,"um bolinho ");
        }
    } else {
        wait(&status1);
```

```
        //waitdpid(pid1, &status1, 0);
        fprintf(arq, "de arroz!!");
    }
    fclose(arq);
    return 0;
}
```

- pipe

Canal de comunicação, bidirecional, entre processos que simula um cano, onde um processo escreve na entrada dados e outro processo consegue lê-los na saída. Usualmente se lê na entrada 0, e se escreve na entrada 1. Existe um mecanismo de sincronização para o uso dos pipes, onde um processo que quer ler algo, espera até ter algo escrito para ler o conteúdo.

- pipe()

Função que inicializa o canal de mensagem entre processos.

- write()

Função que escreve, usualmente no índice 1, no *pipe*.

- read()

Função que lê, usualmente no índice 0, do *pipe*.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>

int main() {
    int canal1[2];
    int canal2[2];
    int canal3[2];
    int nbytes;
    pid_t pid;
    char mensagem[80];
    int val=0;
    int tamanho=0;

    // Inicializa os pipes
    pipe(canal1);
    pipe(canal2);
    pipe(canal3);

    val = -1;
    pid = fork();
```

```
    if(pid != 0) {
        printf("Proc pai %d- Antes do envio de val = %d\n", getpid(), val
);
        val = 10;
        printf("Proc pai %d- Envio de val = %d\n", getpid(), val);
        write(canal1[1], &val, sizeof(int));

        // Espera mensagem do filho
        nbytes = read(canal2[0], &tamanho, sizeof(int));
        printf("Proc pai %d- Recebi do meu filho o tamanho da mensagem qu
e me será enviada (%d Bytes)\n", getpid(), tamanho);
        nbytes = read(canal3[0], mensagem, sizeof(char)*tamanho);
        printf("Proc pai %d- Recebi do meu filho a seguinte mensagem: %s\
n", getpid(), mensagem);
    } else {
        printf("Proc Filho %d- Antes de receber val do meu pai, val = %d\
n", getpid(), val);
        // Espera mensagem do pai
        nbytes = read(canal1[0], &val, sizeof(int));
        printf("Proc Filho %d- Depois de receber val do meu pai, val = %d
\n", getpid(), val);

        strcpy(mensagem, "Valeu paizao!!");
        tamanho = strlen(mensagem);
        write(canal2[1], &tamanho, sizeof(int));
        write(canal3[1], mensagem, sizeof(char)*tamanho);
    }
}
```

- execv()

Função que realiza a troca do binário do processo que realiza a chamada de execv para o binário, que tem seu nome passado como parâmetro.

É utilizado o fork-exec para que o novo processo criado passe a utilizar um binário já existente, um exemplo disso é o processo de *boot* do computador, onde o processo *init* realiza fork-exec para executar outras partes do sistema operacional.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    pid_t pid;

    pid = fork();
```

```
    if(pid != 0) {
        printf("Pai de PID: %d executando... Filho de PID: %d\n", getpid(
), pid);
    } else {
        execv("./pipe", NULL); // Chamada que troca o binário executado p
elo processo
        execv("./a.out", NULL); // Esta segunda chamada não será executad
a
    }
    return 0;
}
```

Encontro 5 - Escalonamento de CPU

- Conceitos e definições:

Escalonador é a parte mais central do OS, ele decide qual processo tem acesso a CPU em um dado instante. Visa otimizar o tempo de uso da CPU, tenta nunca deixar a CPU ociosa, sempre ocupa ela com algum processo. Existem vários critérios possíveis para otimização:

- **Eficiência**:
  - 100% de uso (CPU);
  - Maximizar o número de processos atendidos/segundo;
  - Minimizar o tempo de espera por parte do *user* (*turnaround*).
- **Equidade**:
  - Garantir acesso a todos processos.
- **Tempo de resposta**:
  - Ser rápido na tomada de decisão, para não tomar muito tempo da CPU que poderia estar sendo utilizado por outros processos.

Tudo depende de qual ponto de vista se deseja otimizar o OS, com foco no usuário, foco em atender a todo mundo.

- Níveis de escalonamento:

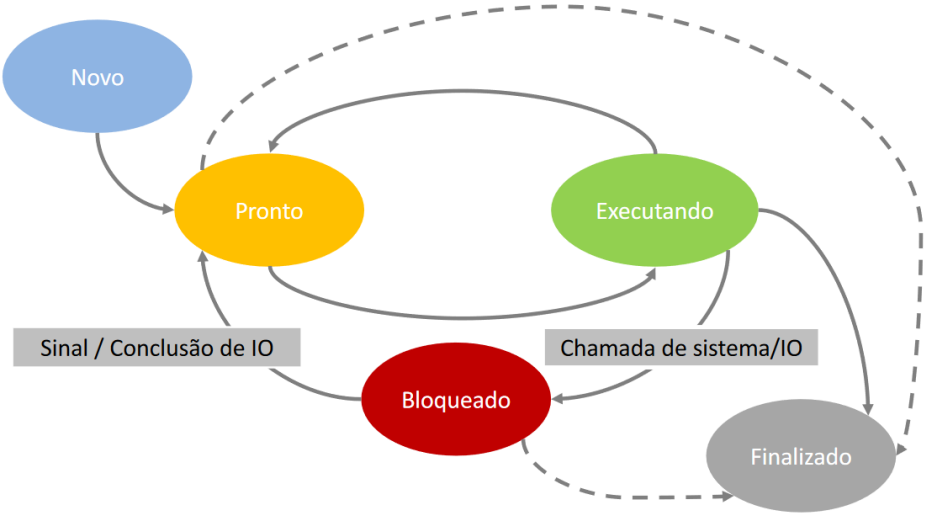
Existem vários níveis de escalonadores no OS, pois existem vários recursos de hardware e em vários níveis se reproduz a necessidade de serem escalonados. Escalas temporais diferentes, mas decisões a serem tomadas são parecidas. Distingue-se 3 níveis de escalonadores:

- **Curto prazo:**
  - Decide alocação da CPU;
  - Rápido;
  - Tempo de resposta na faixa dos **ms**.
- **Médio prazo:**
  - Decide gerenciamento de memória;

    - Swap (copiar processo da memória pro disco);
    - Tempo de resposta na faixa dos **100ms.**
- **Longo prazo:**
    - Criação de processos;
    - Acessos ao disco.
    - Tempo de resposta na faixa maior que **500ms**.

- Estados e escalonamento:

Quando se deve ser acionado o escalonador com base no diagrama de estados do processo:



1. Ao acontecer a transição de “Executando” para “Bloqueado”;
2. Ao acontecer a transição de “Executando” para “Pronto”;
3. Ao encerrar-se um processo. (Transição de “Executando” para “Finalizado”);
    - Nos casos 1,2 e 3 o processo para de executas e isso dispara o escalonador.
4. Ao acontecer a transição de “Bloqueado” para “Pronto”. (Processo liberado pode ser mais importante que o que está executando).
    - É o próprio escalonador que toma a iniciativa para ver se precisa ser trocado o processo em execução. (preempção)

- Preempção:

É o ato de interromper a execução de um processo para executar outro.

- Há dois tipos de escalonadores:

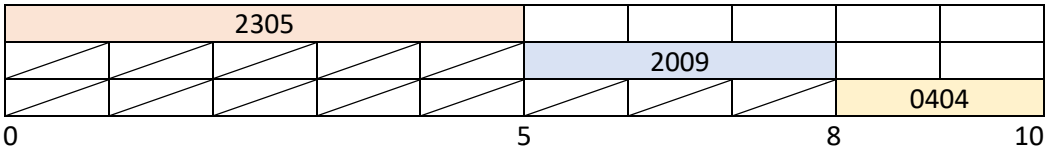
1. Preemptivos: que interrompem, caso seja necessário, o processo em execução (Caso 4). Neste caso quem realiza a preempção deve forçar o salvamento do contexto do processo que se encontra em execução (Sincronização).
2. Não-preemptivos: nunca interrompem o processo em execução, ele tem que deixar a CPU por vontade própria ou porque terminou. (Caso 1, 2 e 3). Sem preempção é mais fácil de implementar porque o próprio processo pode salvar seu estado, de forma “natural”.

- Escalonamento não-preemptivo: FIFO (*First In First Out*)

Primeiro que chega, primeiro que sai. Processo mais antigo na fila dos “prontos” é escalonado.

PID	Hora de chegada na fila	Duração
2305	0	5
0404	2	2
2009	1	3

Diagrama de Gantt:



Duração de tempo total utilizado da CPU é de 10 unidades de tempo.

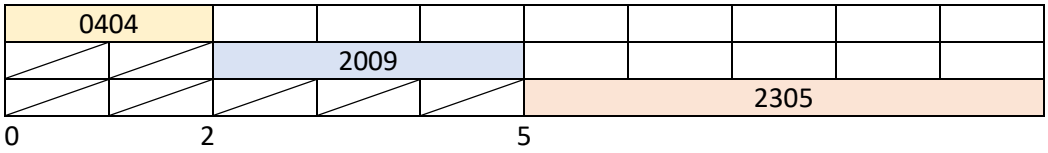
PID	Tempo de Espera
2305	0
2009	5
0404	8

Tempo médio de espera = (0+5+8)/3 = 4,33

Características

- Muito bom em tempo de tomada de decisão por ser um FIFO;
- Equidade: FRACA;
- Sensibilidade extrema a ordem de chegada. Considere o caso abaixo agora:

PID	Hora de chegada na fila	Duração
2305	2	5
0404	0	2
2009	1	3



PID	Tempo de Espera
2305	5
2009	2
0404	0

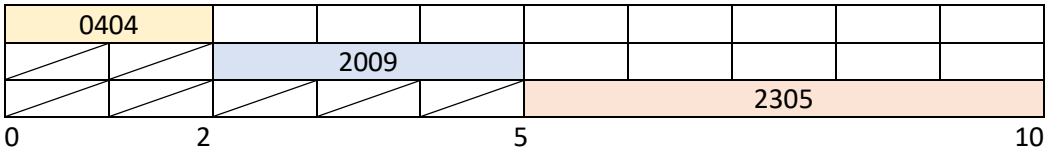
Tempo médio de espera = (5+2+0)/3 = 2,33

- Escalonamento não-preemptivo: SJF (*Shortest Job First*)

Não tem acesso a ordem de chegada, todos processos estão disponíveis em dado momento, o que decide é a duração do processo. Pressupõe-se que todos processos chegam no momento 0, no mesmo instante.

PID	Hora de chegada na fila	Duração
2305	0	5
0404	0	2
2009	0	3

Diagrama de Gant:



PID	Tempo de Espera
2305	5
2009	2
0404	0

Tempo médio de espera = (5+2+0)/3 = 2,33

Pode-se provar matematicamente que minimiza o tempo médio de espera do sistema. É utilizado para estimar o quanto os algoritmos heurísticos se aproximam deste valor ideal.

Para utilizar o SJF precisa-se saber a duração dos processos de antemão, ou seja, prever o futuro, não funciona sempre.

Características:

- Minimiza o tempo médio de espera dos processos;
- Precisa-se saber de antemão a duração dos processos;
    - Não acontece em sistema multi-usuários “abertos” usuais;
    - Acontece em sistemas de lotes ou em situações específicas (Jobs no Windows Server, meu robozinho do banco).
- Paliativo: estabelecer um modelo preditivo da duração futura, utilizando medições feitas no passado sobre a vida dos processos. (médias aritméticas e ponderadas sobre o tempo de execução antigo do processo)

- Escalonamento não-preemptivo: Prioridades

Têm-se várias categorias de processos e se dá uma prioridade diferente para cada um deles, o usuário pode estipular a prioridade sobre seus processos (comando nice no Linux, em C setpriority(), com limites de prioridade).

É necessário armazenar a prioridade do processo em seu PCB, ou utilizar algum outro parâmetro para priorizá-los. i.e: a duração de execução. Tem-se que tomar cuidado com o parâmetro de critério de priorização, pode-se tomar a decisão errada e ferrar tudo. Processos do kernel são mais prioritários que os do usuário. Processos de usuário podem ter prioridades diferentes (Esse word que to fazendo o resumo tem mais prioridade que meu joguinho aberto no background, pq to focadasso aqui). Processos com mesma prioridade tem que ser utilizado algum outro algoritmo para desempate, i.e FIFO.

Características:

- Perde-se a Equidade;
- Há risco de *starvation:* um processo não tem garantia de ter acesso a um dado recurso (CPU) se sempre tiver prioridade baixa, pois podem ficar entrando processos mais prioritários na fila;
    - Solução para isso é aumentar a prioridade dele caso o processo esteja há muito tempo esperando para ser executado.
- Há risco de inversão de prioridade: acontece quando um processo de baixa prioridade possui acesso exclusivo a um recurso crítico;
- Limite conceitual: faz sentido ter processos prioritários em um contexto não-preemptivo?
    - Se surgir um processo prioritário, de certa forma ele precisa passar na frente de um processo de baixa prioridade que está executando na CPU.

Encontro 6 - Escalonamento de CPU

- Problemas de escalonadores não preemptivos:

Sofrem de falta de equidade, e de eventuais problemas de *starvation.* Somente preemptar quando um processo mais prioritário fica pronto, ainda causa risco de *starvation.* Faz-se necessário uma “preempção automática” sistemática, com regularidade.

Todos estes problemas justificam os algoritmos preemptivos.

- Escalonamento preemptivo:

Interrompe a execução e um dado processo para dar lugar a outro. Ainda não resolvem todos problemas. Juntamente com o algoritmo de prioridades, há o risco de ao preemptar exista sempre um processo mais prioritário na espera, ou seja, *starvation*. O hardware, clock do pc, irá impor que, com uma certa regularidade, seja-se necessário a preempção.

- Escalonamento preemptivo: Round Robin (RR)

Mecanismo de permuta dos processos que estão prontos. É o mecanismo **mais equitativo possível,** nenhum processo tem direito a mais tempo (contíguo) que os outros.

- Define-se um **quantum** de tempo:
  - Tipicamente 10-100ms (deve ser considerado o tempo típico da troca de contexto)
- Aconteça o que acontecer, um processo em execução será interrompido após um **quantum;**
  - O processo será devolvido a fila dos processos prontos.
- O escalonador executará o próximo processo da fila dos prontos.

- Tamanho do quantum:

Se o **quantum** for muito grande, quase nunca um processo será interrompido, voltado a ser um algoritmo não-preemptivo. E se o **quantum** for muito pequeno pode gerar um certo conflito caso o tamanho se aproxime do tempo de troca de contexto (deve-se ter o tempo de troca de contexto no fator de 1/1000 ou menos para o tamanho do **quantum,** i.e. troca_de_contexto=0.01ms, logo **quantum**=10ms), ou o processo não terá tempo para "avançar" seu processamento.

Características:

- Tempo médio de espera é igualitário, mas maior. Entranto, ainda sim, alguns processos esperam muito mais em outros algoritmos. Nenhum irá ficar esperando eternamente;
- Seja **n** o número de processos prontos, cada um receberá em média (**1/n**)-ésimo da CPU para sua execução.
- Seja **q** a duração do quantum, um processo executa em média q/n unidades de tempo. E espera em média (n-1)*q/n = q(1-(1/n)) unidades de tempo.

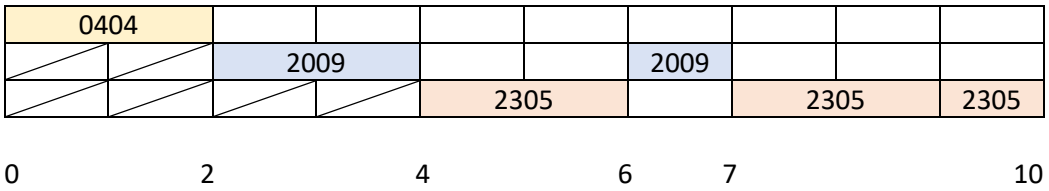
Exemplo RR*:

PID	Hora de chegada na fila	Duração
2305	0	5
0404	0	2
2009	0	3

Quantum = 2 unidades de tempo

*Tempo de troca de contexto = 0 unidades de tempo

Diagrama de Gant



PID	Tempo de Espera
2305	4+1
2009	2+2
0404	0

Tempo médio de espera = (0+4+1+2+2)/3 = 3 unidades de tempo

Comparando com os outros tipos de escalonadores estudados:

Escalonador	Tempo de Espera médio
FIFO (primeiro caso)	4,33
SJF	2,33
RR	3

RR não alcança o tempo ideal do caso do SJF mas ainda é melhor do que o algoritmo FIFO em muitos casos aleatórios.

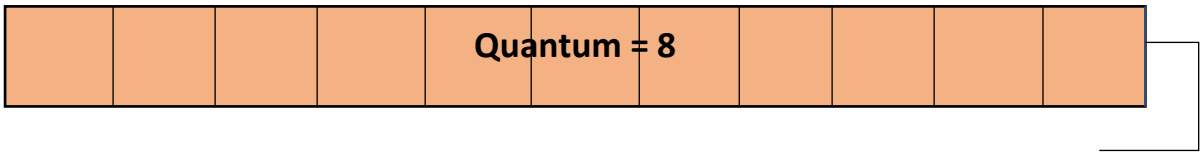
- Múltiplas filas de prioridade:

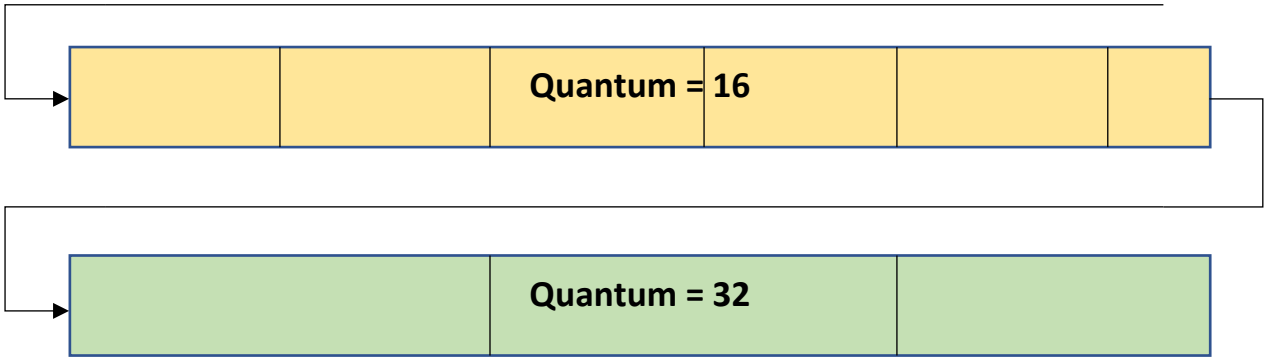
Para manter uma certa equidade, mas apenas entre processos que tem “necessidades” e “demandas” parecidas, usa-se múltiplas filas de processos prontos. Cada fila pode ter um algoritmo de escalonamento diferente e estes algoritmos serem preemptivos ou não (ou possuir um RR com quantums diferentes em cada fila), deve-se possuir um critério de desempate entre as filas. Pode-se também separar a utilização da CPU entre as filas, ou seja, a fila mais prioritária terá 80% de tempo da CPU enquanto que a menos prioritária, somente 20%.

- Múltiplas filas com realimentação:

Mecanismo mais sotisficado, além de ter várias filas com vários critérios em cada uma das filas, têm-se um mecanismo de *aging* (a ideia de que um processo que executa durante muito tempo, também muda de fila conforme o tempo passa).

Uma fila mais prioritária tem um quantum **menor,** e as demais filas possuem um quantum duas vezes maios que o quantum da fila anterior de prioridade maior, seguindo a ordem de prioridade.





Quando um processo realiza um certo número de ciclos de execução (Pronto -> Execução -> Pronto(volta pro fim da fila da prioridade dele)) sem encerrar, o mesmo processo perde sua prioridade e passará para a próxima fila de prioridade menor. Garantindo que o processo ainda execute e, por ter sua prioridade diminuída, consiga mais tempo de processamento na próxima execução.

Com isso processos rápidos tem chance de serem executados rapidamente, e se demorarem demais, perdem sua vez e tem sua prioridade diminuída.

Encontro 7 - Threads

- O que é uma *thread*?

O problema com nossa CPU e nossos processos é o "peso" do processo. Eles possuem contexto "grande", tempo de troca de contexto "grande", e isso gera perda de eficiência. O mínimo que se pode exigir de um processo, é a sequência de instruções que devem ser executadas (código binário, usualmente na cada das dezenas de MB), do lado dos dados de meta-gestão podemos diminuir os dados estatísticos do SO, mas não se ganha muita coisa, o melhor lugar para se perder "peso" do processo é a memória acessada pelo processo (que pode ser muito grande, casa dos GB).

O que os programas fazem é muito paralelo. Muitas aplicações hoje em dia com alto grau de paralelismo (navegador da web, varias abas utilizam do mesmo código fonte para renderizar coisas diferentes). O paralelismo é muito mais natural do que pensar de uma forma sequencial.

Se queremos manter o mesmo processo e compartilhar o mesmo espaço de memória das instruções entre vários fluxos de instruções precisamos adaptar/refinar o PCB (*struct proc*) para possibilitar o rastreamento de mais de um fluxo de instruções. **Basta replicar os PC para mais threads , e, também, replicar os SP para cada thread ter seu espaço de pilha**. Uma thread não é nada mais que vários PC no PCB para acompanhar os fluxos seguidos. Cada thread terá seu processo espaço de pilha, mas compartilharão o *heap*. i.e. Temos 2 threads para o processo X, logo terá que se ter uma *struct proc* da seguinte maneira:

```
struct proc {
    int* sp1;   // Stack pointer para a pilha da thread1
    int* sp2;   // Stack pointer para a pilha da thread2
    void* heap; // Único ponteiro de heap, pois este é compartilhado
    int* pc1;   // Endereço para próxima instrução da thread1
    int* pc2;   // Endereço para próxima instrução da thread2
    // etc...
}
```

- Memória compartilhada

O processo até agora tem seu espaço de memória privado, ninguém se mete, é um dos mecanismos básicos de segurança. Já nas *threads* tem o compartilhamento do *heap,* ou seja, todas variáveis que estão armazenadas lá, qualquer thread pode ler e modificar. No caso da leitura de variáveis não há problema algum, os threads podem ler em paralelo, já na escrita pode ocasionar em concorrência, i.e uma *thread* tentar escrever enquanto a outra estava lendo, ou até duas *threads* tentarem escrever no mesmo momento. As *threads* também possuem seu espaço privado na memória, que seria uma parte da pilha. Com isso variáveis locais são privadas (alocadas na pilha), enquanto que variáveis globais e ponteiros (alocadas no *heap*) serão compartilhadas entre *threads.* Cabe ao programador garantir que acessos conflitantes as variáveis compartilhadas não gerem problemas.

- *Threads* e Arquitetura

Recursos de hardware já vem com paralelismo embutido (CPUs *multi-core*, GPUs) Pensar de forma puramente sequencial cada vez menos faz sentido. Arquiteturas modernas trazem todas interfaces(bibliotecas) de programação para utilização de threads e ainda vão sendo criadas novas linguagens e paradigmas para programar nas arquiteturas. Linguagem Cuda da nvidia manipula *threads na GPU.*

- Modelos de *Threads*

Não é simples substituir um processo por uma *thread* para escalonar. No Linux são *threads* que são escalonadas não processos. Com isso se faz necessário dividir as threads em dois tipos: *Threads* de kernel/núcleo e *threads* de usuário. Existem 3 modelos de *threads,* dependendo da variável de ajuste (*thread* de kernel/núcleo):

1. Um por N: Constitui em uma *thread* de núcleo que hospeda/agrupa n *threads* de nível de usuário. O paralelismo entre as *threads* de nível de usuário é virtual, ou seja, simulado, elas competem entre si para utilizar a CPU, no caso do RR seria dentro do mesmo *quantum* de tempo. Há concorrência e não paralelismo.
2. Um por Um: há o mapeamento exato de uma *thread* de usuário para uma *thread* de núcleo. Ou seja, a unidade de escalonamento é a *thread* de usuário. Isto gera um problema caso haja muitas *threads(*dezenas de milhares) existindo, o escalonador terá que se virar para escalonar isto tudo. (Implementado no Linux)

3. N por M: N *threads* de nível de usuário vão ser mapeadas em M *threads* de núcleo (N>M). Vai ter, para um dado processo várias threads de núcleo que poderão ser escalonadas e em cada uma dessas *threads* (*lightweight process)* existirão várias *threads* de usuário. O processo passa a ser visto como um envelope que descreve todos recursos de hardware. Há um paralelismo onde o escalonador consegue escalonar mais de um fluxo de instruções (*threads* de núcleo, que são compostas por várias *threads* de usuário). Paralelismo físico, onde as *threads* de núcleo são escalonadas para os cores de processamento, e, também, paralelismo virtual enxergado pelo programador com a thread de núcleo sendo utilizada para chaveamento de processamento. Modelo caindo em desuso.

Encontro 8 - Programação concorrente

- Concorrência vs Paralelismo

Programas concorrentes são executados por um único núcleo de processamento (Paralelismo simulado). Programas paralelos são executados simultaneamente em mais de um núcleo de processamento. O que é comum entre as duas abordagens é a necessidade de projetar o programa de tal forma que processe o *input*/*output* sendo que a qualquer momento ele pode ser interrompido, e outro processo pode acessar os recursos disponíveis. Logo existe a necessidade de sincronizar as execuções entre os vários processos para evitar conflitos no uso dos recursos.

- Produtor/Consumidor

Produtor: processo que gera dados para serem utilizados por outros processos, usualmente um buffer na memória.

Consumidor: processo que consome os dados disponibilizados pelo Produtor para algum fim.

Para realmente funcionar, é necessário impor certos mecanismos que garantem que parte dos trechos de código quando são executados não haja interrupções até o final deste trecho. Existe condição de corrida entre os processos para acessar o buffer de dados.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int N=5; // Tamanho do buffer
int i=0; // Índice do buffer

int um_resultado(int arg) {
```

```
    return (arg*2 + 1);
}

int produce(int* buffer) {
    // Armazena em 'i' e incrementa (para não ser sobreescrito o que outr
o processo escreveu)
    // buffer é utilizado como uma fila
    buffer[i] = um_resultado(i);
    i = (i+1) % N;
    return 0;
}

int consume(int* buffer) {
    // Consome em 'i' e decrementa (para liberar a entrada para outro pro
cesso)
    int res = buffer[i];
    i = (i-1) % N;
    return res;
}

int main(void) {
    pid_t pid;
    int* the_buffer = (int*) malloc(sizeof(int)*N);
    int ii;
    int res;

    pid = fork();
    if(pid != 0) {
        // Pais geram os dados (Produtor)
        for(ii=0; ii<10; ii++) {
            produce(the_buffer);
            printf("Pai (produtor) produziu %d\n", the_buffer[i]);
        }
    } else {
        // Filhos consomem (Consumidor)
        for(ii=0; ii<10; ii++) {
            produce(the_buffer);
            printf("Filho (consumidor) consumiu %d\n", the_buffer[i]);
        }
    }
    return 0;
}
```

- Seção Crítica

Trecho de código que de forma garantida é executada de uma vez, até seu final, por um processo sem ser interrompido. Acontece apenas em escalonadores preemptivos. Deve respeitar 3 critérios:

1. Garantir que o que é executado nesta seção é executado só em um processo. Só em uma *thread*;
2. Não se pode impedir um processo de acessar esta seção crítica;
3. Não há postergação infinita, todo processo deve ter direito de acesso à seção crítica.

- Seção crítica em nível de *hardware*

1. Uma opção radical que não haja a preempção é por meio de suspensão das interrupções, inclusive do *clock*, é possível fazer quando hardware é simples, mas não é aconselhável para que em casos críticos onde possa-se querer recuperar o controle do processo pendurado.
2. Conjunto de instruções atômicas, ou seja, instruções que depois que começam a serem executadas não podem ser interrompidas, vai começar e terminar.

   Dois exemplos de instruções atômicas do *hardware*:

   1. *Test and Set*: testa o conteúdo de uma variável ou registrador e seta a variável ou registrador, é feito de forma atômica. Este tipo de bloqueio se chama espera ativa, o processo fica na CPU até poder continuar.

```
2. // Executado atomicamente
3. boolean test_and_set(boolen *target) {
4.     boolean rv = *target;
5.     *target = true;
6.     return rv;
7. }
8. while(test_and_set(&lock)); // Espera até retornar
9.     // Roda o Seção Crítica
10.    // …
11.    // Fim Seção Crítica
12.    lock = false // Libera para outros processos após SC
```

   2. *Compare and Swap*: muito parecido com o *test and set*, somente com um valor a mais para comparar o valor antigo do registrador com o valor de referência passado, e o novo valor que o registrador deve assumir. Caso o valor do registrador atual seja igual ao esperado é atualizado para o valor novo. Retorna valor antigo. Também é espera ativa.

```
int compare_and_swap(int* old_value, int expected, int new_value) {
    int tmp = *old_value;
    if(*old_value == expected)
        *old_value = new_value;
    return tmp; // Retorna o valor original
}
while(compare_and_swap(&lock, 0, 1) != 0); // Espera
```

```
    // Agora lock == 1, e  valia 0 antes
    // Roda o Seção Crítica
    // …
    // Fim Seção Crítica
    lock = 0; // Libera para outros processos após SC
```

- Locks e semáforos

Por mais que exista em nível de hardware algum tipo de suporte, vão se querer estruturas mais elaboradas para facilitar a programação concorrente. Internamente utilizarão o suporte do hardware mas para o usuário final isto será transparente e terá uma interface mais sofisticada.

- O Lock Mutex

Básico para qualquer interface de programação concorrente, funcionamento é muito parecido com o test_and_set mas o nível de abstração é muito maior e tem uma variável interna com dois valores possíveis, livre ou ocupado, true ou false, 1 ou 0. Parecido com uma classe com dois métodos. As chamadas são atômicas.

```
// Mutex (Funções atômicas)
get() { // Teste a variável interna
    while(!available) ; // Espera ativa
    available = false;
}
release() { // Libera o recurso
    available = true;
}
```

- Semáforos

É um mutex copiado, mais poderoso, controla o recurso e ainda mais controla uma fila de espera de até N acessos concorrentes a este recurso. Controla o acesso concorrente de até N instâncias de dados recurso, cada vez que se pede o recurso tem esta fila de espera até encher. Quando chega a N o tamanho da fila nenhum processo pode entrar mais nela e tem que esperar liberar uma vaga. Possibilita contabilizar quantos recursos podem ser utilizados em concorrência ou contar quantos processos podem acessar determinado recurso em concorrência.

Estrutura de dados abstrata para representar um semáforo:

```
int S; // Recursos disponíveis no momento
init (int N) {
    // Inicia a fila, seta em máximo
    S = N;
```

```
}
wait() {  // P()
    // Alguém esta solicitando o recurso (Feito de forma atômica)
    // caso seja maior que zero tem recurso sobrando e pode ser utilizado
    while( S <= 0) ;
    S--;
}
signal() {  // V()
    // Libera que parou de utilizar o recurso, libera o recurso
    S++;
}
```

Semáforo binário: quando N=2, determina quando determinado recurso pode ser utilizado ou não, equivalente ao Mutex.

- DeadLocks

Dois processos A e B e dois recursos em exclusão mútua (um arquivo, variável compartilhada) A acessa o arquivo e quer acessar a variável enquanto B acessa a variável e quer acessar o arquivo, com isso eles entram em *deadlock,* ou seja, **um fica esperando o outro liberar o recurso para utilizar o outro, bloqueando assim os recursos que estão sendo utilizados**. O problema de fato acontece quanto um número grande de processos entram em *deadlock,* tornando assim difícil decidir qual ação ou ações deve ser tomada para liberá-los.

Existem 4 critérios para um *deadlock* ocorrer:

1. Exclusão mútua:
2. Retenção e espera: um processo entra numa seção crítica e enquanto está na seção crítica ele fica de posse de outro recurso.
3. Não preempção: Se não houver preempção não irá ocorrer problema de *deadlock*.
4. Espera circular: Mesmo conceito da retenção e espera mas em mais processos. Do processo A depender do recurso utilizado por B e B depender do recurso utilizado por C e C depender do recurso acessado por A, fechando assim um ciclo. Pode-se ter N processos nesse ciclo, mas tem que ser um ciclo.

- Prevenir Deadlocks:

1. Projetar o sistema para que deadlocks não aconteçam, basta usar um escalonamento não preemptivo. Pode-se também abrir mão de um dos critérios definidos acima.
2. Podemos detectar caso haja algum deadlock e tomar alguma ação para corrigi-lo. o SO terá que manter novas estruturas de dados de recursos que os processos podem acessar e quais estão utilizando-os, realizando assim controle sobre o tipo de acesso das threads e processos que estão acessando os recursos, aumentando o uso de memória devido a maior quantidade de informações que devem ser armazenadas. Também terá uma tabela de recursosXprocessos. Pode ser utilizado para um subconjunto de recursos e processos declarados, construindo um grafo e, detectar ciclos no grafo. O SO terá que decidir qual

processo terá de liberar o recurso para, assim, liberar os demais, sacrificando/interrompendo um processo para isto. Como opção mais radical, o SO pode também encerrar todos processos participantes do *deadlock.*

3. (opção mais utilizada) Deixar o programador(usuário) se virar... Ele que programe direito para não dar *deadlock* nos programas dele. O programador deve entender que caso utilize *threads* para programar, tem o risco destas *threads* entrarem em *deadlock* e cabe a ele não deixar isto acontecer.

- Exemplos de Deadlocks:

1. Processo A está na seção crítica de um recurso e solicita outro recurso que o processo B está na seção crítica e, por sua vez, este solicita o recurso que o processo A está utilizando. Um processo está imobilizando um recurso enquanto solicita outro recurso que está imobilizado pelo outro. A espera por B e B espera por A;
2. Jantar dos filósofos: Filósofos tem duas atividades, pensar (não fazem nada, só pensam) ou comer utilizando dois garfos. Eles estão sentados em uma mesa redonda e do lado de cada prato tem dois garfos. Entre dois filósofos, sempre tem um garfo. Uns pensam, outros querem comer e para comer pegam dois garfos. Toda vez que quer comer tenta pegar os dois garfos e quando para de comer, libera os garfos. Tem conflito nos casos em que dois filósofos vizinhos tentam comer ao mesmo tempo. Tem o caso crítico quando um filósofo tenta comer e pega um dos garfos, mas o outro garfo está em uso pelo vizinho ao lado, logo não consegue comer nem pensar. E, por final, o caso do *deadlock* absoluto, quando no mesmo momento todos filósofos tentam pegar os garfos, mas como estão pegando no mesmo tempo, somente conseguem pegar um garfo e o outro se encontra com outro filósofo, logo ninguém consegue comer nem pensar.