

Este arquivo retrata todo o conteúdo presente, até o momento, sobre conceitos e fundamentos do Sistema Operacional, sendo aplicado como material de estudo para provas semestral. Gostaria que você dissecasse cada questão mencionada implicando com todos os pontos principais de perguntas, detalhadamente, buscando facilitar e simplificar o entendimento sobre os assuntos, e o ato de revisar este material.

Questão 1 — O que é um processo e qual a diferença em relação a um programa?

Definição formal (clara):

- **Programa:** arquivo estático com código e dados (instruções armazenadas em disco).
- **Processo:** instância em execução de um programa — inclui o código carregado na memória, pilha, heap, registradores, descritores de arquivos e informações de controle gerenciadas pelo sistema operacional

Explicação passo a passo:

1. O programa é o “texto” (ex.: `editor.exe`) guardado no disco.
2. Quando você executa esse programa, o SO cria um processo: aloca memória, inicializa pilha e heap, abre arquivos, atribui PID, coloca o processo na fila de prontos.
3. O processo é o que efetivamente corre na CPU; pode haver vários processos executando o mesmo programa (ex.: abrir duas janelas do navegador)

Exemplo prático:

- O arquivo `calc.exe` é o programa; cada vez que você abre calculadora nasce um processo `calc` com seu próprio estado

Pontos que o professor costuma cobrar:

- diferença “estático × dinâmico”; mencionar recursos (memória, descritores de arquivos); citar PID e estado do processo (pronto, executando, bloqueado)

Dica de memorização:

- programa = receita
- processo = cozinheiro executando a receita

Resposta pronta (curta, para entrega):

Um **processo** é a execução de um programa em memória, contendo o código, dados, pilha, registradores e recursos alocados (descritores de arquivos, PID, etc.). Já o **programa** é apenas o conjunto de instruções armazenado em disco. Enquanto o programa é estático, o processo é o seu estado dinâmico em execução.

Questão 2 — O que compõe o contexto de um processo e sua importância na troca de processos (troca de contexto)?

Componentes do contexto (detalhado):

- **Registradores da CPU:** incluindo o contador de programa (Program Counter / PC) e registradores gerais.
- **Ponteiro de pilha (stack pointer)** e estado da pilha.
- **Mapeamento de memória / espaço de endereçamento:** tabelas de páginas, limites de segmentos.
- **Informações de I/O:** descritores de arquivos abertos, sockets.
- **Informações de gerenciamento:** PID, prioridade, estado (pronto, executando, bloqueado), contadores de tempo, sinalizações.
- **Informação de contexto de kernel (se houver):** tabelas de página, TLS, ponteiros para estruturas do SO

Por que é importante (troca de contexto):

- **Troca de contexto** é o ato de salvar o contexto do processo A e carregar o contexto do processo B para que a CPU possa continuar B exatamente onde ele parou. Sem salvar corretamente, A não poderia voltar a executar do mesmo ponto. A troca tem custo (overhead) — muitos contextos switches diminuem eficiência

Exemplo prático:

- CPU interrompe processo para servir uma interrupção ou para executar outro processo. Antes de mudar, o SO grava registradores e o PC de A; quando voltar a A, restaura esses valores

Pontos que o professor pode questionar:

- O que o SO salva em usuário x O Kernel Mode. O custo de troca de contexto; diferença entre troca leve (thread) e pesada (processo)

Dica de memorização:

- Contexto = “foto” do estado do processo (o SO tira foto antes de trocar)

Resposta pronta (curta):

O **contexto** de um processo engloba seus registradores (incl. contador de programa), pilha, espaço de endereçamento, descritores de I/O e informações administrativas (PID, prioridade, estado). Na troca de contexto, o sistema salva o contexto do processo atual e carrega o do próximo para que a CPU retome a execução corretamente; esse processo é essencial, mas tem overhead.

Questão 3 — O que são threads e qual a vantagem de usar múltiplas threads num mesmo processo?

Definição e natureza:

- **Thread:** unidade básica de execução (linha de execução) dentro de um processo. Um processo pode conter uma ou várias threads.
- Threads compartilham o mesmo espaço de memória (heap, variáveis globais, descritores de arquivo) mas cada thread tem **própria pilha** e registradores

Vantagens principais:

- **Concorrência interna:** permite que diferentes tarefas do mesmo programa ocorram simultaneamente (I/O + cálculo).
- **Melhor utilização de CPU em sistemas multicore:** threads podem executar em núcleos diferentes.
- **Menor custo para criar e alternar** (comparado a processos).
- **Maior responsividade:** ex.: interface não travar enquanto uma thread faz cálculo pesado

Exemplos práticos:

- Navegador: uma thread por aba/renderização, outra para rede, outra para interface.
- Servidor web: thread por conexão (ou pool de threads)

Riscos a considerar (resumidos):

- sincronização necessária (mutex, semáforos), risco de *race conditions* e deadlocks

Pontos que o professor pode cobrar:

- diferença entre thread e processo; stacks separados; exemplos de problemas de concorrência

Dica de memorização:

- processo = empresa
- threads = funcionários que trabalham na mesma sala (mesmo espaço/recursos)

Resposta pronta (curta):

Threads são linhas de execução dentro de um processo que compartilham o espaço de memória. Múltiplas threads melhoram desempenho e responsividade, permitem paralelismo em múltiplos núcleos e têm baixo custo de criação/troca comparado a processos.

Questão 4 — Por que threads são mais leves que processos e quais os riscos do compartilhamento de memória?

Por que são mais leves (explicação técnica):

- Criar e alternar entre threads exige menos operações: não é necessário duplicar o espaço de endereçamento, tabelas de arquivos ou outras estruturas pesadas; threads compartilham dados do processo pai. Consequentemente, **overhead de criação e troca** (context switch) é menor

O que é compartilhado vs individual:

- **Compartilhado:** heap, memória global, descritores de arquivo, endereçamento.
- **Individual:** pilha da thread, registradores (incl. PC), estado de execução.

Riscos do compartilhamento de memória:

- **Race conditions:** duas threads escrevem/leem do mesmo dado sem sincronização → resultados inconsistentes.
- **Corrompimento de dados:** uma thread sobrescreve estrutura usada por outra.
- **Deadlocks:** uso incorreto de locks pode levar threads a esperarem indefinidamente umas às outras.
- **Priority inversion:** thread de alta prioridade bloqueada por thread de baixa prioridade que detém um recurso

Como mitigar:

- Usar **mutexes, semáforos, monitores, locks**
- Projetar seções críticas curtas; evitar deadlocks (ordenar aquisição de locks, timeouts)

Pontos de prova frequentes:

- Pedir para explicar *stack vs heap* em relação a threads.
- Exemplos de race condition e como corrigi-la (ex.: usar mutex).

Resposta pronta (curta):

Threads são “leves” porque compartilham o mesmo espaço de memória e recursos do processo, reduzindo custo de criação e troca. Os riscos incluem race conditions, corrupção de dados e deadlocks, por isso é necessário usar mecanismos de sincronização.

Questão 5 — Papel do escalonador de processos e por que é necessário?

O que faz o escalonador (scheduler):

- Decide **qual processo/thread** será executado pela CPU e por quanto tempo.
- Mantém filas (ready queue), escolhe próxima tarefa, realiza preempção quando necessário

Objetivos do escalonador (métricas):

- **Utilização da CPU** (alta);
- **Throughput** (nº de processos concluídos por unidade de tempo);
- **Turnaround time** (tempo desde submissão até conclusão);
- **Waiting time** (tempo na fila de prontos);
- **Response time** (tempo até primeira resposta, importante para sistemas interativos);
- **Justiça / ausência de starvation**

Por que é necessário:

- Em sistemas multitarefa a CPU é recurso limitado; sem escalonador processos poderiam monopolizar, gerando ineficiência e má resposta para processos interativos

Tipos básicos de escalonamento:

- **Preemptivo** (pode interromper processos) × **Não-preemptivo**;
- Exemplos: FCFS, SJF, Priority, Round Robin

Pontos para prova:

- saber comparar objetivos (ex.: SJF minimiza turnaround médio; RR fornece melhor tempo resposta em sistema interativo).

Resposta pronta (curta):

O escalonador é a parte do SO que decide qual processo executa na CPU e por quanto tempo. Ele garante uso eficiente da CPU, balanceando throughput, tempos de resposta e justiça entre processos.

Questão 6 — Funcionamento do Round Robin e como garante justiça

Como funciona (passo a passo):

1. Existe uma **fila de prontos** (FIFO).
2. Cada processo no topo recebe um **quantum** (fatia de tempo fixa).
3. Se o processo terminar antes do quantum, libera a CPU; se não, é **preemptado** ao final do quantum e colocado no fim da fila.
4. Repete-se circularmente.

Justiça (fairness):

- Todos os processos recebem fatias iguais de CPU em ordem cíclica — nenhum fica eternamente sem executar (evita starvation).
- Justiça depende do tamanho do quantum: quantum muito grande reduz preempção → pior justiça; quantum muito pequeno aumenta overhead de context switch

Trade-offs e observações importantes:

- **Overhead:** context switches frequentes se quantum for muito pequeno.
- **Degeneração para FCFS:** se quantum muito grande, RR se comporta parecido com FCFS.
- **Boa escolha:** quantum alguns vezes maior que tempo de troca de contexto e pequeno relative ao tempo de execução interativo

Exemplo numérico (mini):

- Processos A, B, C; quantum = 10ms. A usa 15ms → executa 10ms (volta pra fila), B 8ms (termina), etc. Ordem de execução mantém fairness

Pontos de prova:

- explicar efeito do tamanho do quantum; calcular sequência de execução dada carga

Resposta pronta (curta):

No Round Robin cada processo recebe um **quantum** fixo de CPU e, se não terminar, volta ao fim da fila. Assim, todos recebem fatias iguais em ordem circular, garantindo justiça e reduzindo espera indefinida.

Questão 7 — O que é comunicação entre processos (IPC) e por que é necessária?

Definição clara:

- **IPC (Inter-Process Communication)** refere-se aos mecanismos que permitem troca de dados e sincronização entre processos distintos

Por que é necessária (motivação):

- Processos isolados não compartilham memória; muitas aplicações são compostas por módulos/processos que precisam cooperar (cliente-servidor, produtor-consumidor). IPC possibilita modularidade, escalabilidade (processos distribuídos) e cooperação entre programas

Propriedades da IPC:

- **Síncrona vs Assíncrona** (bloqueante vs não bloqueante);
- **Direta vs Indireta** (comunicação direta entre processos vs via mailbox);
- **Local vs Remota** (pipes/sockets locais ou comunicação por rede)

Exemplo prático:

- Um editor envia dados para o processo de impressão por meio de IPC (pipe ou socket).

Pontos que caem em prova:

- definir IPC, comparar técnicas, explicar diferença entre sincronização e comunicação (IPC pode prover ambos)

Resposta pronta (curta):

IPC são mecanismos (pipes, sockets, memória compartilhada, mensagens) que permitem processos trocarem dados e se sincronizarem. São essenciais para que módulos distintos cooperem sem compartilhar diretamente o espaço de endereço.

Questão 8 — Cite e explique duas técnicas de IPC (detalhar: memória compartilhada e passagem de mensagens)

Técnica 1 — Memória Compartilhada

O que é:

- Área de memória que dois ou mais processos mapeiam no seu espaço de endereçamento para leitura/escrita direta

Como funciona (resumido):

- O SO cria um segmento compartilhado; processos mapeiam-no e acessam como se fosse memória local

Vantagens:

- Muito rápida (sem cópias entre espaços de usuário e kernel). Ideal para grandes volumes de dados

Desvantagens / necessidade extra:

- Não fornece sincronização por si só — é preciso usar **mutexes**, **semáforos**, **spinlocks** para evitar race conditions

Exemplo:

- Compartilhar uma grande tabela de dados entre processos de análise.

Técnica 2 — Passagem de Mensagens (Message Passing)

O que é:

- Processos trocam mensagens (envio/recebimento). Implementações comuns: pipes, FIFOs, message queues, sockets

Como funciona (resumido):

- Processo A faz `send(mensagem)` e processo B faz `receive()`. Pode ser bloqueante ou não

Vantagens:

- Simples de usar, mais segura (sem compartilhamento direto de memória); facilita comunicação entre processos em máquinas diferentes (sockets)

Desvantagens:

- Potencialmente mais lenta (cópias, overhead do kernel) e pode ter limites de tamanho por mensagem

Comparação rápida (quando usar o quê):

- **Memória compartilhada** → alta performance, grande volume, ambos processos na mesma máquina;
- **Mensagens** → simplicidade, segurança, comunicação remota.

Pontos de prova:

- citar necessidade de sincronização para memória compartilhada; tipos de pipes e sockets; blocking/non-blocking

Resposta pronta (curta):

Duas técnicas comuns de IPC são: (1) **Memória compartilhada**, onde processos mapeiam uma mesma região de memória para comunicação rápida (requer sincronização externa); e (2) **Passagem de mensagens**, em que processos trocam mensagens via pipes, filas ou sockets, oferecendo mais isolamento e segurança à custa de maior overhead.

Questão 9 — O que é um deadlock e exemplo simples de ocorrência

Definição clara:

- **Deadlock** é situação em que dois ou mais processos ficam permanentemente bloqueados porque cada um espera por um recurso que outro possui; nenhum pode progredir.

Exemplo simples (clássico):

- Processo A: ocupa impressora (recurso X), solicita scanner (recurso Y) → aguarda.
- Processo B: ocupa scanner (Y), solicita impressora (X) → aguarda.
Resultado: A e B ficam presos

Diferença entre deadlock e starvation:

- **Deadlock:** situação de impasse circular onde nenhum prossegue.
- **Starvation:** processo não progride por não receber tempo/recurso, mas não há necessariamente impasse circular (pode ser por prioridades)

Como detectar (resumo):

- Grafos de alocação de recursos (Resource Allocation Graph) — ciclo indica possível deadlock (em sistema com instância única do recurso)

Pontos de prova:

- pedir exemplo real; explicar detecção vs prevenção vs recuperação; mencionar Banker's algorithm como técnica de avoidance

Resposta pronta (curta):

Deadlock é um impasse em que processos ficam bloqueados esperando recursos uns dos outros (ex.: A segura X e pede Y; B segura Y e pede X). Nessa situação nenhum dos processos avança, exigindo medidas de prevenção, detecção ou recuperação.

Questão 10 — Quatro condições necessárias para deadlock (Coffman) — explicar cada

As quatro condições (com explicação e como quebrá-las):

1. Exclusão mútua

- *O que:* o recurso não pode ser compartilhado simultaneamente (ex.: impressora).

- *Como quebrar:* utilizar recursos compartilháveis quando possível (ex.: permitir acesso compartilhado) — nem sempre aplicável.

2. Retenção e espera (hold and wait)

- *O que:* processo detém pelo menos um recurso e espera por outro que está ocupado.
- *Como quebrar:* obrigar processos a requisitar todos os recursos de uma vez (alocação atômica) ou liberar antes de requisitar novos.

3. Não-preempção (no preemption)

- *O que:* recursos não podem ser retirados de um processo forçadamente — só são liberados voluntariamente.
- *Como quebrar:* permitir preempção de recursos (o SO retira e aloca a outro), ou restaurar estado.

4. Espera circular (circular wait)

- *O que:* existe um ciclo de processos $P_0 \rightarrow P_1 \rightarrow \dots \rightarrow P_n \rightarrow P_0$, onde cada um espera um recurso do próximo.
- *Como quebrar:* impor uma ordenação total de recursos e exigir que processos requisitem recursos em ordem crescente (prevent circular waiting).

Observação:

- **Todas** as quatro condições devem existir ao mesmo tempo para haver um deadlock. Remover ou impedir qualquer uma evita deadlock.

Pontos de prova:

- pedir para explicar cada condição e dar exemplo de como prevenir (ex.: resource ordering).

Resposta pronta (curta):

Segundo Coffman, as quatro condições são: exclusão mútua, retenção e espera, não-preempção e espera circular. Se as quatro ocorrerem simultaneamente, pode surgir deadlock; eliminar qualquer uma delas impede o deadlock.

Resumo prático para revisão (flashcards rápidos)

- Processo vs Programa: **execução** × arquivo.
- Contexto: **registradores + PC + pilha + descritores + estado**.
- Thread: **linha de execução**, compartilha heap; cada thread tem **pilha própria**.
- Threads leves: menos overhead; risco = **race conditions**.
- Scheduler: decide **quem e quanto tempo** executa.
- Round Robin: **quantum**, preemptivo, justo, tradeoff com overhead.
- IPC: **memória compartilhada** (rápida + sincronização) e **mensagens** (seguro + overhead).
- Deadlock: impasse; quatro condições de Coffman.