PONTIFÍCIA UNIVERSIDADE CATÓLICA DE SÃO PAULO
Centro de Ciências Exatas e Tecnologia

Sistemas Operacionais I & II
Prova P1 - 2005

1) **[1,0 ponto]** Escolha APENAS uma das seguintes questões e responda:
   a) O que é modo kernel ou modo supervisor? O que é modo usuário? Quais as diferenças entre esses dois modos? Porque eles são necessários?

Processadores modernos tem dois modos de execução, o modo kernel e o modo usuário. Além das instruções gerais, existem instruções privilegiadas que só podem ser usadas no modo kernel. Essas instruções privilegiadas ajudam o processador a acessar informações sensíveis (por exemplo, limpar o cache) e a manipular operações vitais (por exemplo, operações de I/O). No modo usuário, apenas instruções gerais podem ser executadas. Se uma instrução privilegiada for executada no modo usuário, o hardware não executa a instrução, mas ao invés disso, trata a instrução como ilegal e gera um *trap* no sistema operacional. Um sistema operacional normalmente executa no modo kernel, pois precisa ter acesso a todos os componentes de hardware e nenhum usuário pode afetar a execução do SO.

   b) O que é um contexto? Apresente uma descrição detalhada de todas as atividades de uma troca de contexto.

Um processo precisa de alguns recursos do sistema para executar com sucesso. Esses recursos do sistema incluem o ID do processo, registradores, áreas da memória (para instruções, variáveis locais e globais, pilha e etc.), várias tabelas (tabela de processos), e um program counter para indicar a próxima instrução a ser executada. Ele constitui o ambiente ou contexto de um processo. Os passos de uma troca de contexto do processo A para o processo B são os seguintes:

- **Suspende a execução de A**
- **Transfere o controle para o escalonador de CPU.**
- **Salva o contexto de A em seu PCB (process control block) e outras tabelas**
- **Carrega o contexto de B (a partir do seu PCB) nos registradores e etc.**
- **Retoma a execução das instruções de B a partir do *program counter* de B**

2) [1,0 ponto] Escolha APENAS uma das seguintes questões e responda:

a) Defina condição de corrida. Responda primeiro a questão e use um exemplo para ilustrar a sua resposta.

Condição de corrida é uma situação em que mais do que um processo ou thread estão executando e a acessando concorrentemente um dado compartilhado, e o resultado final depende da ordem de execução. Por exemplo, dois processos atualizando o saldo de uma mesma conta conforme o código apresentado abaixo:

```
Depositar() {
  READ(Arq_Contas, Reg_Cliente);
  READLN(Valor_Deposito);
  Reg_Cliente.Saldo =   Reg_Cliente.Saldo-Valor_Deposito;
  WRITE(Arq_Contas,Reg_Cliente);
}
```

```
Retirar() {
  READ(Arq_Contas, Reg_Cliente);
  READLN(Valor_Deposito);
  Reg_Cliente.Saldo =   Reg_Cliente.Saldo-Valor_Deposito;
  WRITE(Arq_Contas,Reg_Cliente);
}
```

b) Explique os modelo de threads one-to-one, many-to-one, and many-to-many.

O modelo one-to-one (um-para-um) mapeia cada thread de usuário em um thread de kernel. Fornece maior concorrência dos que o modelo muitos-para-um, permitindo que outro thread execute quando um thread efetuar uma chamada bloqueante ao sistema; também permite ter múltiplos threads executando em paralelo em multiprocessadores.

O modelo many-to-one (muito-para-um) mapeia muitos threads de usuário em um thread de kernel. A gerencia de threads é realizada no especo de endereçamento de usuário,sendo assim eficiente, mas o processo inteiro é bloqueado se um dos threads realizar uma chamada bloqueante ao sistema.

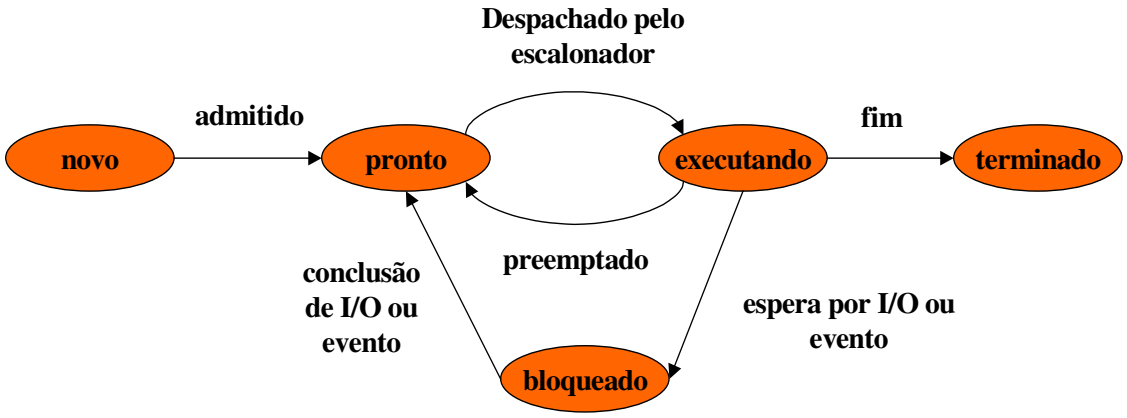
O modelo many-to-many (muitos-para-muitos) multiplexa muitos threads de usuário com um número menor ou igual de threads de kernel. O número de threads de kernel pode ser específico para determinada aplicação ou determinada máquina. Este modelo evita a criação excessiva de threads.

3) [1,0 ponto] Apresente o diagrama de estados de um processo, incluindo todas as transições e uma breve descrição de cada estado e cada transição.

À medida que um processo executa, ele muda de estado. Estado de um processo é definido pela sua atividade corrente. Cada processo pode estar em um dos seguintes estados, conforme ilustrado na figura:

- **Novo: o processo está sendo criado**
- **Executando: as instruções do processo estão sendo executadas**
- **Suspenso ou Bloqueado: o processo está esperando que algum evento ocorra (tal como o término de uma operação de E/S ou o recebimento de um sinal)**

- **Pronto: o processo está esperando para ser atribuído a um processador**
- **Terminado: o processo terminou sua execução**



4) [1,0 ponto] Quais são as condições necessárias para a ocorrência de um Deadlock? Liste essas condições e forneça uma descrição breve de cada uma delas.

Deadlock pode surgir se as seguintes quatro condições ocorrerem simultaneamente:

- **Exclusão Mútua: apenas um processo por vez pode utilizar o recurso**
- **Hold-and-wait: existem processos que alocaram recursos e estão bloqueados esperando por outro processo**
- **Não Preempção: Os recursos não podem ser preemptados (tomados dos processos). Um processo só libera os recursos espontaneamente, após ter terminado de usá-lo.**
- **Espera circular: Existe um conjunto {P1, P2, ..., Pn} de processos bloqueados, tal que P1 está esperando por um recurso alocado a P2, que está esperando por um recurso alocado a P3, ..., e Pn está esperando por um recurso alocado a P1.**

5) [1,0 ponto] Qual a função principal de um escalonador de processos? Defina escalonamento preemptivo, não-preemptivo e cooperativo.

O escalonador de processos é parte do Sistema Operacional responsável pelo escalonamento (procedimento de seleção) de processos. Ou seja, decidir qual dos processos prontos para execução deve ser alocado à CPU.

Escalonamento Preemptivo: um algoritmo de escalonamento é dito preemptivo quando o sistema operacional pode interromper um processo em execução para que outro processo utilize o processador.

Escalonamento Não-Preemptivo: Nos primeiro Sistemas Operacionais, onde predominava tipicamente processamento em Batch, o escalonamento era do tipo não-preemptivo. Neste escalonamento, quando

um processo (ou Job) ganha o direito de utilizar a CPU, nenhum outro processo pode lhe tirar esse recurso.

Escalonamento Cooperativo: No escalonamento cooperativo alguma política não-preemptiva deve ser adotada. A partir do momento que um processo está em execução, este voluntariamente libera o processador, retornado para a fila de pronto.

6) [2,0 pontos] Considere os seguintes conjuntos de processos, com a duração da fase de uso de CPU dada em milissegundos:

Processo	Duração da fase de uso da CPU	Prioridade
P_1	10	3
P_2	8	1
P_3	4	3
P_4	7	4
P_5	5	2

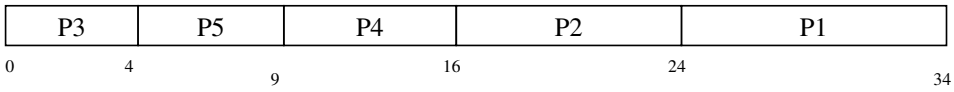
Supõe-se que os processos entrem na fila de processos prontos na ordem P1, P2, P3, P4 e P5, todos no tempo 0.

a) Desenhe quatro diagramas de Gantt ilustrando a execução desses processos usando os algoritmos de alocação FIFO, MP, por prioridade de forma não-preemptiva (um número de prioridade menor indica uma prioridade mais alta), e Round-Robin (quantum = 2).

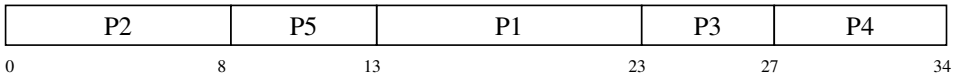
b) Qual é o tempo de processamento médio para os algoritmos de alocação do item a?

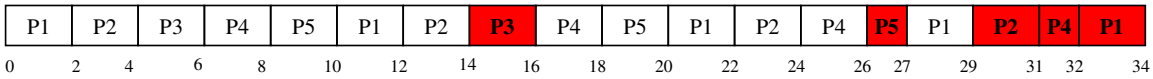
FIFO (First-In First-Out)

MP (Menor Primeiro)



Prioridade (Não Preemptivo)





	FIFO	MP	Prioridade	RR
P1	10	34	23	34
P2	18	24	8	31
P3	22	4	27	16
P4	29	16	34	32
P5	34	9	13	27
Média	22,6	17,4	21,0	28,0

7) [3,0 pontos] O problema dos leitores e escritores (Courtouins *et al*, 1971) modela o acesso concorrente a um banco de dados. É aceitável que haja mais de um processo lendo o banco de dados ao mesmo tempo, mas se um processo estiver escrevendo no banco de dados (ou seja, modificando-o), nenhum outro processo, nem mesmo os leitores, poderão ter acesso a ele enquanto o escritor não terminar. Uma solução para este problema é utilizar semáforos para sincronizar os acessos concorrentes dos leitores e escritores ao banco de dados. Considerando os semáforos ***db*** (acesso ao banco de dados) e ***mutex*** (acesso exclusivo a região crítica), apresente uma solução em pseudo-código C para este problema.

```
typedef int semaphore; / * use your imagination * /

semaphore mutex = 1; / * controls access to 'rc' * /
semaphore db = 1; / * controls access to the data base * /
int rc = 0; / * # of processes reading or wanting to * /

void reader(void)
```

```
{
        while (TRUE) { / * repeat forever * /
                down(&mutex); / * get exclusive access to 'rc' * /
                rc = rc + 1; / * one reader more now * /
                if (rc == 1) down(&db); / * if this is the first reader ... * /
                up(&mutex); / * release exclusive access to 'rc' * /
                read_data_base(); / * access the data * /
                down(&mutex); / * get exclusive access to 'rc' * /
                rc = rc - 1; / * one reader fewer now * /
                if (rc == 0) up(&db); / * if this is the last reader ... * /
                up(&mutex); / * release exclusive access to 'rc' * /
                use_data_read(); / * noncritical region * /
        }
}

void writer(void)
{
        while (TRUE) { / * repeat forever * /
                think_up_data(); / * noncritical region * /
                down(&db); / * get exclusive access * /
                write_data_base(); / * update the data * /
                up(&db); / * release exclusive access * /
        }
}
```

obs: utilize as operações sobre semáforos denominadas **down(&<semáforo>)** e **up(&<semáforo>)**.