Lista 3 – Threads, Escalonamento e Sincronismo de processos

1. (0.5p) Quando programamos com threads, podemos controlar o tempo de uma thread no CPU. Por exemplo, a API `POSIX Pthreads` contém a função `pthread_yield` para fazer uma thread "desistir" do CPU. Por qual motivo desejaríamos fazer isso? Conseguem pensar em um exemplo prático onde essa ação seria importante, por exemplo, em um sistema web? Explique cada detalhe.

2. **(2.0p) IPC.** Você conhece o jogo de adivinhação "quente ou frio"? Implemente o programa `adivinha.c` que gera um número aleatório entre 1 e 100. Este programa deve criar um processo filho que tentará adivinhar o número sorteado. Para isso, a cada rodada o processo filho deve comunicar seu palpite ao processo pai, que apenas pode responder se o número sorteado é maior, menor ou igual ao palpite do filho.
   Requisitos
   A. O jogo termina quando o processo pai informar ao filho que o número sorteado é igual ao palpite dele.
   B. O processo pai só pode sortear o número após a criação do filho.
   C. O processo filho deve sortear seu próprio número que determinará o palpite inicial.
   D. O processo filho não pode trapacear para ler a memória do processo pai e ver o número. A única informação que ele deve receber do pai é a citada no item A
   E. A solução deve utilizar alguma técnica de IPC para resolver o problema. **Sugestão**: Comece o testando a solução de memória compartilhada.

3. **(0.5p)** Na pasta `utils` são apresentadas três versões de um mesmo programa: Uma serial `prog-serial.c`, uma "paralela" com múltiplos processos `prog-processos.c` e uma "paralela" com múltiplas threads `prog-threads.c`. O programa possui duas funções, uma que usa a CPU de forma intensiva (CPU bound) e outra que simula o uso intensivo de operações de E/S (I/O bound). Compile e execute cada uma das versões conforme as orientações no cabeçalho dos arquivos. Anote e compare o desempenho das três versões. Explique as diferenças observadas.

4. **(0.5p)** Nos códigos do exercício anterior, altere o número de processos (`prog-processos.c`) e threads (`prog-threads.c`) para 100, recompile e execute novamente os programas. Observe o que ocorre com o tempo de execução das duas versões.

5. **(2.0p)** Escreva o programa `criathreads.c` que crie uma cadeia sequencial de `N` de threads além da thread principal: A thread `N` deve ser criada pela thread `N-1`, que por sua vez é criada pela thread `N-2` e assim por diante. Cada thread deve imprimir seu `TID` (consulte a função `pthread_self`) e o `TID` da thread que a criou (no caso de threads, cabe ao programador passar essa informação adiante). As impressões na tela devem ocorrer em ordem inversa de criação, ou seja, a thread `N` deve ser a primeira a imprimir, logo seguirá a thread `N-1` e assim até completar o conjunto. O processo mais antigo deverá mostrar seu `PID`.

Exemplo da saída do programa:

```
Sou a thread com TID  3 criada pela thread com TID  2

Sou a thread com TID  2 criada pela thread com TID  1

Sou a thread com TID  1 criada pelo processo pai com PID <pid>
```

6. **(3.0p) Escalonadores modernos**
   A. Explique a diferença entre escalonamento preemptivo e não-preemptivo.
   B. O problema da condição de corrida, o qual é ocasionado pela falta de sincronismo entre dois ou mais processos concorrentes que compartilham dados em comum, pode causar inconsistência de dados tornando imprevisível o comportamento destes processos. Isto é um comportamento altamente não desejado para o programador de sistemas operacionais. Se o problema da condição de corrida poderia ser facilmente evitado utilizando um kernel do tipo não-preemptivo (por exemplo, desabilitando o uso de interrupções durante a execução dos processos), quais são então as principais razões para a maioria dos sistemas operacionais modernos utilizarem um kernel do tipo preemptivo (*e.g.*, sistemas embarcados, dispositivos inteligentes, Linux, Windows, macOS, sistemas distribuídos ou em nuvem)?
   C. Cite possíveis cenários onde o uso de um kernel preemptivo se faz evidente na prática hoje em dia.

7. **(2.0p) Escalonadores modernos** – Os seguintes processos são escalonados utilizando um algoritmo de escalonamento round-robin (RR) do tipo preemptivo e baseado em prioridades. A cada processo é atribuída uma prioridade, onde um número alto indica uma prioridade relativa alta. O escalonador executará primeiro o processo de mais alta prioridade. Para processos com a mesma prioridade, será utilizado um escalonador round-robin com um quantum de 10 unidades. Se um processo é preemptado por um processo de prioridade maior, o processo preemptado é colocado no final da fila.

Processo	Prioridade	Burst Time (ms)	Chegada (ms)
P1	8	15	0
P2	3	20	0
P3	4	20	20
P4	4	20	25
P5	5	5	45
P6	5	15	55

   A. Mostre a ordem de escalonamento dos processos utilizando um diagrama de Gantt
   B. Qual é o tempo de execução (turnaround time) de cada processo?
   C. Qual é o tempo de espera (waiting time) de cada processo?

8. **(2.0p)** Considere o seguinte conjunto de processos (tempo do CPU burst em milissegundos). Assumir que os processos chegaram na ordem P1, P2, P3, P4, P5, todos no instante de tempo 0:

Processo	Burst Time (ms)	Prioridade

P1	2	2
P2	1	1
P3	8	4
P4	4	2
P5	5	3

A. Desenhe quatro diagramas de Gantt ilustrando a execução destes processos utilizando os seguintes algoritmos de escalonamento: FCFS, SJF, prioridade não-preemptiva (um número grande implica uma alta prioridade), e RR (quantum = 2 ms).
B. Qual é o tempo de execução (turnaround time) de cada processo para cada um dos algoritmos de escalonamento no item anterior?
C. Qual é o tempo de espera (waiting time) de cada processo para cada um destes algoritmos de escalonamento?
D. Qual dos algoritmos resulta no menor tempo de espera médio (average waiting time) (incluindo todos os processos)?

9. **(1.0p)** Muitos dos algoritmos de escalonamento da CPU são parametrizados. Por exemplo, o algoritmo RR requer um parâmetro para indicar o tamanho da unidade de tempo (time quantum). O algoritmo de filas multinível com retroalimentação (Multilevel feedback queues) requer parâmetros para definir o número de filas, o algoritmo de escalonamento para cada fila, o critério utilizado para mover processos entre as filas, etc. Desta maneira, estes algoritmos são na verdade conjuntos de algoritmos (por exemplo, o conjunto de algoritmos RR para todos os tamanhos de quantum, etc.). Um determinado conjunto de algoritmos pode incluir um outro (por exemplo, o algoritmo FCFS é igual ao algoritmo RR com tempo de quantum infinito). Que relação (se houver) existe entre os seguintes pares de conjuntos de algoritmos?
   A. Prioridade e SJF
   B. Multilevel feedback queues e FCFS
   C. Prioridade e FCFS
   D. RR e SJF

10. **(1.0p)** Você está implementando um escalonador RR (Round-Robin). Você precisa escolher um quantum para um sistema onde a troca de contexto leva 10 µs (microsegundos). Comente sobre as características de performance decorrentes das escolhas dos seguintes quantums, justificando suas respostas:
    A. 20µs
    B. 50ms
    C. 1s

Questões – Sincronização de processos

2. **(2.0p)** Escreva o programa `abc123.c` que cria dois processos filhos. Estes processos executam em paralelo e escrevem ambos em um mesmo arquivo alternando entre a escrita de um número e de uma letra por linha. O processo que escreve os números escreverá um número para cada letra entre `A` e `Z`. Após terminarem, o processo pai

mostra o conteúdo do arquivo no terminal. O programa deve ser livre de condições de corrida.

Indicações

- Assumir que a API `POSIX` para gerenciamento de arquivos não é *thread-safe*, portanto, deverão ser implementadas medidas de sincronismo adequadas.
- Utilizar a implementação `POSIX Pthreads` de semáforos sem nome (`sem_init`, `sem_wait`, `sem_post`, `sem_destroy`).
- Exemplo de execução:

Linha	**./abc123.out** # run
1	1 A
2	2 B
...	…
25	25 Y
26	26 Z

3. **(2.0p)** A implementação da solução de Peterson mostrada abaixo apresenta o problema de condição de corrida (*race condition* em inglês). Identifique a linha do código onde ela ocorre, e explique o motivo da solução funcionar corretamente apesar disso. Assumir um sistema multiprocessado que executa instruções conforme a sequência do código, isto é, ele não realiza otimizações envolvendo reordenação de instruções.

   Sugestão

   Revisar a Seção 6.3 do livro de Silberschatz, ed. 10 (2018).

```
int turn;
boolean flag[2];
int i; // i representa o processo atual
int j; // j representa o outro processo

while (true) {
  flag[i] = true;
  turn = j;
  while (flag[j] && turn == j);
  /* secao critica */
  flag[i] = false;
  /* secao nao critica */
}
```

4. **(2.0p)** O programa `syncerror.c` apresenta um erro de sincronismo. Identifique este erro apontando:
   A. As linhas do código onde o erro acontece
   B. O motivo pelo qual o erro ocorre
   C. Uma versão modificada chamada `no-syncerror.c` que corrija o erro e explique qual será o conteúdo de `arq1.txt` após o fim da execução do programa (`1-5? 6-10?`)

   Indicações

- As alterações realizadas na sua solução proposta devem ser claramente identificadas, listando todas as linhas alteradas

11. **(4p)** Escreva um programa em C para encontrar o número com maior quantidade de divisores entre 1 e 100.000. O programa deve executar com um número de threads definido pelo usuário, e a carga de trabalho deve ser dividida de forma aproximadamente igual entre as threads no que diz respeito à quantidade de números verificados. Por exemplo, se fossem 4 threads, cada uma deveria verificar aproximadamente 25.000 números.
    - A thread original deve inserir os 100.000 números numa fila. Cada thread deve continuamente retirar um número da fila para calcular sua quantidade de divisores, de maneira que nenhuma thread fique ociosa até que a fila esteja vazia.
    - Neste exercício, você deve criar sua própria implementação **thread-safe** de uma fila (implemente também outras estruturas de dados e algoritmos thread-safe, caso julgue necessário).
    - Não serão aceitas implementações de fila criadas por terceiros, mesmo que devidamente acreditadas. Além disso, seu programa deve ser livre de condições de corrida. A saída do programa deve ser o número encontrado.

5. **(2.0p)** No seguinte fragmento de código temos duas variantes de uma mesma função chamada `somatorio` que é acessada por inúmeras threads, onde `soma` é uma variável com escopo de acesso global, portanto qualquer função do programa pode acessá-la e alterar seu valor.

```
int soma = 0; /* Variavel global */

void *somatorio1(void *param) {
    int incremento = atoi(param);

    soma = soma + incremento;

    pthread_exit(0);
}

void *somatorio2(void *param) {
    int incremento = atoi(param);

    pthread_mutex_lock(&mutex);
    soma = soma + incremento;
    pthread_mutex_unlock(&mutex);

    pthread_exit(0);
}
```

A. Escreva o programa `soma-multithread.c`, onde o processo `main` cria 4 threads, cada thread chama a função `somatorio1` e no final `main` imprime o valor da variável `soma` depois da execução das várias threads.
B. Se rodar o programa muitas vezes, produzirá o mesmo valor final de `soma` sempre? Argumente a sua resposta.
C. Se existir uma ordem para a execução das threads, explique qual é e de que maneiras poderá influenciar esta ordem na solução. **Sugestão:** Imprimir a `TID` de cada thread durante sua execução utilizando a chamada de sistema `pthread_self()`.
D. Explique o porquê do uso do `mutex` na função `somatorio2`, quais problemas ele soluciona e, de que maneira se modifica a resposta na execução da `main` com esta função?

6. **(2.0p) Deadlocks** – Responda e argumente:
   A. Qual a relação entre deadlock e condição de corrida?
   B. Qual a relação entre deadlock e condição de exclusão mútua?
   C. Como identificar o deadlock em um software. Indique as características.
   D. Quais arquiteturas de software são mais suscetíveis a apresentarem problemas de deadlock?
   E. Que consequências teriam os deadlocks em um processo de qualidade de software?

7. **(1.0p) O problema produtor-consumidor** (Problema do buffer limitado). Dois processos compartilham um buffer comum de tamanho fixo. Um deles, o produtor, insere informações no buffer, e o outro, o consumidor, as retira dele. O problema surge quando o produtor quer colocar um item novo no buffer, mas ele já está cheio. A solução é o produtor ir dormir, para ser despertado quando o consumidor tiver removido um ou mais itens. De modo similar, se o consumidor quer remover um item do buffer e vê que este está vazio, ele vai dormir até o produtor colocar algo no buffer e despertá-lo. O pseudo-código deste problema é:

```
#define N  100 /* Tamanho do buffer */

int count = 0; /* Numero de itens no buffer */

void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();
        if(count == N)
            sleep();
        insert_item(item);
        count = count + 1;
        if(count == 1)
            wakeup(consumer);
    }
}
```

```
void consumer(void) {
    int item;

    while(TRUE) {
        if(count == 0)
            sleep();
        item = remove_item();
        count = count - 1;
        if(count == N - 1)
            wakeup(producer);
        consume item(item);
    }
}
```

Se esse código for executado por threads concorrentemente, ou seja, tivermos uma thread executando a função `producer` e outra thread executando `consumer`, ocorrerá algum problema? Para responder à questão considere os seguintes pontos:

A. Existe uma região de memória compartilhada no código.
B. Se existe algum problema descreva-o com os conceitos vistos no capítulo.
C. Caso ocorra algum problema no código anterior, implemente uma solução chamada `producer-consumer-threadsafe.c` e anexe o código. Explique a sua solução a partir do código, explicitando o que cada nova linha faz para resolver o problema.

8. **(2.0p)** Implemente um programa cujo objetivo é manter duas threads executando em um laço infinito. As threads devem exibir alternadamente na tela T0, T1, T0, T1, ... Resolva o problema utilizando a solução de Peterson para garantir a sincronização, implemente e execute. Comente sobre o resultado obtido.

9. **(2.0p)** Implemente o exercício anterior, substituindo a solução de Peterson por semáforos.