Simulando reservas de assentos em um cinema para avaliar a eficiência e desafios relacionados ao multithreading

Arthur Cenci
CCET – Centro de Ciências Exatas e Tecnológicas
Unioeste – Universidade Estadual do Oeste do Paraná
Cascavel, Brasil
arthur.silva14@unioeste.br

Kauan Campos
CCET – Centro de Ciências Exatas e Tecnológicas
Unioeste – Universidade Estadual do Oeste do Paraná
Cascavel, Brasil
kauan.campos@unioeste.br

Matheus Lopes
CCET – Centro de Ciências Exatas e Tecnológicas
Unioeste – Universidade Estadual do Oeste do Paraná
Cascavel, Brasil
matheus.lopes8@unioeste.br

Marcio Oyamada
CCET – Centro de Ciências Exatas e Tecnológicas
Unioeste – Universidade Estadual do Oeste do Paraná
Cascavel, Brasil
marcio.oyamada@unioeste.br

I. Introdução

Desde o início dos anos 2000, a indústria de processadores tem adotado arquiteturas multicore para atender à crescente demanda por desempenho. Cada núcleo atua como um processador independente, permitindo a execução real de tarefas em paralelo. Para tirar proveito desse paralelismo, a programação multithreading fornece mecanismos que aumentam a eficiência do uso da CPU e promovem a concorrência na execução de processos.

Embora o multithreading seja frequentemente associado ao hardware moderno, seu conceito é anterior aos processadores multicore, estando ligado às políticas de escalonamento e troca de contexto dos sistemas operacionais. Threads são unidades básicas de execução dentro de um processo, compartilhando código, dados e recursos do sistema, o que possibilita a execução simultânea de tarefas. No entanto, esse modelo também introduz desafios relacionados à concorrência de acesso a recursos compartilhados.

Um dos principais problemas enfrentados é a condição de corrida, que ocorre quando dois ou mais threads acessam dados compartilhados de forma não sincronizada, resultando em comportamentos imprevisíveis. Outro desafio é a seção crítica, um trecho de código onde o acesso simultâneo por múltiplas threads deve ser evitado para garantir consistência.

Diante disso, mecanismos como **exclusão mútua**, **progresso** e **espera limitada** são essenciais para controlar o acesso concorrente às seções críticas. Estes conceitos garantem que os processos compartilhem recursos de forma segura e eficiente.

Este artigo tem como objetivo simular o processo de reserva de assentos em uma sala de cinema, utilizando três abordagens: uma implementação sequencial, uma com multithreading sem sincronização e outra com multithreading sincronizado. Serão discutidos os desafios enfrentados sem o uso de sincronização, as soluções aplicadas e a análise de desempenho de cada abordagem. Com isso, busca-se demonstrar na prática os benefícios e cuidados necessários ao se trabalhar com multithreading

II. O problema da reserva de assentos

O problema consiste em simular as reservas de uma sala de cinema, utilizando-se de três funções diferentes para esta tarefa. A primeira é sequencial, ou seja, uma thread fará o papel de um cliente buscando reservar assentos em um cinema até ele não ter mais assento disponíveis. A função dois busca reservar os assentos com uma função multithreading, onde cada thread representa um cliente buscando reservar um assento; as buscas e reservas na função dois acontecem de modo concorrente, isto é, como se os clientes tentassem armazenar ao mesmo tempo sem uma forma de controle, o que abre precedentes para condições de corrida.

A função três busca também reservar assentos com uma função multithreading. No entanto, essa função se preocupa em sincronizar os acessos de cada thread ou cliente a matriz que representa o cinema. A comparação entre essas 3 abordagens visa observar o impacto do uso do multithreading na eficiência do programa e o impacto da concorrência (com e sem sincronismo) na integridade do programa e de seus resultados.

III. IMPLEMENTAÇÃO DO PROBLEMA

A função principal do código implementa um menu de opções ao usuário, assim sendo, ele pode escolher entre as opções: ”Mostrar todos os assentos”, ”Reservar Lugares (Sequencial)”, ”Reservar Lugares (multithreading sem sincronização)”, ”Reservar Lugares (multithreading com sincronização)” e ”Sair”.

A primeira função desenvolvida foi a de alocação sequencial, como pode ser visto logo abaixo, na figura 1. Ela recebe a matriz de assentos e enquanto o número de assentos ocupados for menor que o número total de assentos ela, aleatoriamente, aloca um novo assento e registra em um arquivo “ocupantes.txt”.

```c
// Alocação sequencial
void alocar_lugar_sequencialmente(char assentos[FILAS][COLUNAS]){
    while(assentos_ocupados < NUM_ASSENTOS){
        int fila = rand() % FILAS;
        int coluna = rand() % COLUNAS;
        if(assentos[fila][coluna] == 'O') continue;
        assentos[fila][coluna] = 'O';
        assentos_ocupados++;
        registrar_ocupante_assento(fila, coluna, pthread_self());
    }
}
```

Figura (1) - Função que aloca posições na matriz

A segunda função desenvolvida foi a de alocação multithreading, como pode ser visto posteriormente, na figura 2. Ela converte o tipo genérico no tipo da matriz e enquanto o cinema não estiver todo ocupado e a thread não romper o seu limite de tentativas ela tenta alocar um assento livre e por fim registra a reserva em um arquivo chamado ocupantes.txt.

```c
// Alocação paralela sem sincronização

void* alocar_lugar_paralelamente(void* arg){
    char (*assentos)[FILAS][COLUNAS] = arg;
    int tentativas = 0;
    const int MAX_TENTATIVAS = NUM_ASSENTOS * 10;

    while(assentos_ocupados < NUM_ASSENTOS && tentativas < MAX_TENTATIVAS) {
        int fila = rand() % FILAS;
        int coluna = rand() % COLUNAS;
        tentativas++;

        if((*assentos)[fila][coluna] == 'O') continue;

        // Adiciona delay para forçar condição de corrida
        usleep(100);

        (*assentos)[fila][coluna] = 'O';
        assentos_ocupados++;
        registrar_ocupante_assento(fila, coluna, pthread_self());
    }
    pthread_exit(0);
}
```

Figura (2) - Função multithreading que aloca posições na matriz de forma concorrente

A terceira função implementa o multithreading. Além disso, ela se preocupa com o sincronismo entre threads, a fim de que não ocorram condições de corrida na seção crítica da função. Ela também transforma o tipo genérico para o tipo da matriz de assentos, reservando, após isso, os assentos enquanto o número de assentos ocupados for menor que o de assentos totais.

Dentro do *while* é onde fica implementado parte do semáforo. Antes de entrar na região crítica de acesso a estrutura da matriz é chamada a função *sem_wait,* que incrementa a variável *semid* e, no caso de *semid* ser menor que 0, ele bloqueia a thread, tirando ela de execução. Logo abaixo é feita uma verificação de, no caso do número de assentos ocupados for maior que o número total de assentos, a *thread* em execução libera a *thread* em espera e aumenta o valor de *semid.*

Essa verificação evita que uma thread entre no *sem_wait* sem assentos livres pra serem alocados. Após isso, o algoritmo verifica se o assento está ocupado e, caso não esteja, o assento é ocupado e registrado no arquivo “ocupantes.txt”. Com a reserva feita, o semáforo é incrementado, liberando a próxima *thread* para execução. Tudo o que foi descrito pode ser visto na figura 3 abaixo.

```c
// Alocação paralela com sincronização
void* alocar_lugar_paralelamente_sync(void* arg){
    char (*assentos)[FILAS][COLUNAS] = arg;
    while(1){
        int fila = rand() % FILAS;
        int coluna = rand() % COLUNAS;

        sem_wait(&semid);
        if (assentos_ocupados >= NUM_ASSENTOS){
            sem_post(&semid);
            break;
        }
        if((*assentos)[fila][coluna] == 'L'){
            (*assentos)[fila][coluna] = 'O';
            assentos_ocupados++;
            registrar_ocupante_assento(fila, coluna, pthread_self());
        }
        sem_post(&semid);
    }
    pthread_exit(0);
}
```

Figura (3) - Função multithreading que aloca posições na matriz de forma sincronizada por um semáforo

Para cada reserva é feito um registro, gravando o nome da *thread* que reservou o assento. Esse registro é posto em um arquivo chamado “ocupantes.txt”, e a função que realiza esse registro pode ser vista logo abaixo, na figura 4.

```c
// Utilizado para registrar assentos no arquivo
void registrar_ocupante_assento(int fila, int colunas, pthread_t tid){

    sem_wait(&sem_n_registros);
    FILE *f = fopen("ocupantes.txt", "a");
    if (f == NULL) return;
    fprintf(f, "Ocupante do assento [%d][%d]: Thread %lu\n", fila, colunas, tid);
    fclose(f);

    // Incrementa o contador de registros
    n_registros++;
    sem_post(&sem_n_registros);
}
```

Figura (4) - Função que registra a reserva de assento em um arquivo

Para garantir que a variável n_registros represente de fato o número de registros efetuados é usado um semáforo na função, assim sendo, a função evita concorrência sobre a variável, o que poderia ocorre caso a função fosse chamada pela função de alocação multithreading sem sincronização.

Ao término da função, existe uma variável global chamada *n_registros* que é incrementada, a fim de que sejam encontradas inconsistências geradas pela concorrência sem sincronismo, e essa variável é, posteriormente, utilizada na função de checagem de métricas. Caso o número de registros feitos seja diferente do número de assentos ocupados, duas ou mais threads diferentes reservaram o mesmo assento. Além disso, a variável ocupados_verificados utiliza o retorno da função contar_ocupados(), que percorre a matriz contando assentos ocupados, o que garante o número de assentos de fato ocupados. A função que faz a checagem de algumas das métricas que serão analisadas pode ser vista na figura 5.

```c
// Verifica a integridade e escreve métricas
void checarMetricas(char assentos[FILAS][COLUNAS], const char* metodo, double tempo){

    double throughput = NUM_ASSENTOS / tempo;
    int ocupados_verificados = contar_ocupados(assentos);

    printf("\nMétricas - %s:\n", metodo);
    printf("Tempo de execução: %.6f segundos\n", tempo);
    printf("Vazão (assentos/s): %.2f\n", throughput);
    printf("Assentos ocupados: %d\n", assentos_ocupados);
    if (n_registros != NUM_ASSENTOS || ocupados_verificados != NUM_ASSENTOS)
        printf("Inconsistência detectada nos dados!\n");
    else
        printf("Integridade verificada com sucesso\n");
}
```

Figura (5) – Função que checa métricas de eficiência

Por fim, é importante salientar que o código é controlado por uma série de variáveis e constantes globais, como o número de linhas, colunas, threads, assentos ocupados e registros de reservas, e todas elas podem ser vistas na figura 6.

```c
#define FILAS 25
#define COLUNAS 25
#define NUM_THREADS 3
#define NUM_ASSENTOS (FILAS * COLUNAS)

int assentos_ocupados = 0;
int n_registros = 0; // Contador de registros
```

Figura (6) – Variáveis e constantes

Além das variáveis e constantes exibidas acima, há a variável de semáforo, inicializada como 1 na função *main* e declarada como global, e o vetor threads de tamanho *NUM_THREADS*.

IV. ANÁLISE DOS RESULTADOS OBTIDOS

Neste capítulo será feita a análise sobre os resultados obtidos com a execução de cada uma das três abordagens construídas. Vale a pena ressaltar que as análises foram feitas sobre sobre uma matriz de 25 linhas por 25 colunas e com 3 threads tentando reservar os assentos ao mesmo tempo

O resultado obtido após a execução das reservas por meio da função de alocação sequencial pode ser visto na figura 7.

```
--- CINEMA ---

1. Mostrar lugares disponiveis
2. Reservar lugar (sequencial)
3. Reservar lugar (multithreading sem sync)
4. Reservar lugar (multithreading com sync)
5. Sair...
2

Métricas - Sequencial:
Tempo de execução: 0.010498 segundos
Vazão (assentos/s): 59535.15
Assentos ocupados: 625
Integridade verificada com sucesso
```

Figura (7) – Resultado da execução da reserva sequencial de assentos

Os resultados exibidos na figura 7 podem ser complementados pela ferramenta perf, que foi utilizada para exibir métricas sobre a execução do código. As métricas da ferramenta podem ser vistas na figura 8.

```
Performance counter stats for './multiOF':

          11,01 msec task-clock                #    0,001 CPUs utilized
              4      context-switches          #  363,248 /sec
              3      cpu-migrations            #  272,436 /sec
             64      page-faults               #    5,812 K/sec
     23.072.879      cycles                    #    2,095 GHz
      9.198.710      stalled-cycles-frontend   #   39,87% frontend cycles idle
     16.950.340      instructions              #    0,73  insn per cycle
                                               #    0,54  stalled cycles per insn
      3.683.063      branches                  #  334,466 M/sec
        402.372      branch-misses             #   10,92% of all branches

    11,513205255 seconds time elapsed

     0,001014000 seconds user
     0,010828000 seconds sys
```

Figura (8) - Exibição de estatísicas sobre o código sequencial a partir da ferramenta perf

A partir dos resultados revelados, então, é possível ter dimensão do tempo de demora na alocação de assentos, além de outros vários dados que serão usado na comparação entre abordagens mais a frente.

Seguindo, então, o resultado obtido a partir da função de reservas com multithreading assíncrono pode ser visto na figura 9.

```
--- CINEMA ---

1. Mostrar lugares disponiveis
2. Reservar lugar (sequencial)
3. Reservar lugar (multithreading sem sync)
4. Reservar lugar (multithreading com sync)
5. Sair...
3

Métricas - Multithread sem sync:
Tempo de execução: 0.039774 segundos
Vazão (assentos/s): 15713.78
Assentos ocupados: 627
Quantidade incorreta de assentos alocados
```

Figura (9) - Resultado da função de alocação de assento por multithreading sem sincronização

É possível observar que o código multithread sem sincronia é mais lento que o sequencial e não gera resultados corretos, como apontado pelo resultado na figura 9. Para corroborar com os fatos, há ainda mais métricas na figura 10, reforçando as observações anteriores.

```
Performance counter stats for './multiOF':

          23,31 msec task-clock                #    0,003 CPUs utilized
            704      context-switches          #   30,200 K/sec
            155      cpu-migrations            #    6,649 K/sec
             92      page-faults               #    3,947 K/sec
     59.867.008      cycles                    #    2,568 GHz
     27.847.539      stalled-cycles-frontend   #   46,52% frontend cycles idle
     28.868.518      instructions              #    0,48  insn per cycle
                                               #    0,96  stalled cycles per insn
      6.340.755      branches                  #  272,007 M/sec
        868.206      branch-misses             #   13,69% of all branches

    7,927787739 seconds time elapsed

    0,001224000 seconds user
    0,022049000 seconds sys
```

Figura (10) - Exibição de estatísicas sobre o código multithreading assíncrono a partir da ferramenta perf

Os motivos pelos quais o código é menos eficiente podem ser vistos ao compararmos os resultados nas figuras 8 e 10. É possível ver o aumento, principalmente, no número de instruções executadas, trocas de contexto feitas, migrações de CPU e número de ciclos utilizados, ou seja, a criação e o gerenciamento de *threads* envolvem um alto custo computacional.

O sistema operacional precisa alocar recursos, fazer troca de contexto e outras diversas operações, e para tarefas pequenas e rápidas como reservar um assento, o tempo gasto com gerenciamento das *threads* pode ser maior do que o tempo da própria tarefa.

Por fim, os resultados obtidos na execução das reservas com multithreading com sincronização podem ser vistos na figura 11.

```
--- CINEMA ---

1. Mostrar lugares disponiveis
2. Reservar lugar (sequencial)
3. Reservar lugar (multithreading sem sync)
4. Reservar lugar (multithreading com sync)
5. Sair...
4

Métricas - Multithread com sync:
Tempo de execução: 0.014683 segundos
Vazão (assentos/s): 42566.23
Assentos ocupados: 625
Integridade verificada com sucesso
```

Figura (11) – Resultado da função de alocação de assento por multithreading com sincronização

Já em um primeiro momento, vê-se que a execução foi mais rápida que a alocação sem sincronização, além de obter sucesso, o que não foi visto no primeiro método, ou seja, a integridade da matriz não foi agredida. Os dados da figura 12 corroboram com as afirmações.

```
Performance counter stats for './multiOF':

          26,32 msec task-clock                #    0,002 CPUs utilized
            624      context-switches          #   23,712 K/sec
            106      cpu-migrations            #    4,028 K/sec
             97      page-faults               #    3,686 K/sec
     49.730.035      cycles                    #    1,890 GHz
     19.299.456      stalled-cycles-frontend   #   38,81% frontend cycles idle
     28.023.666      instructions              #    0,56  insn per cycle
                                               #    0,69  stalled cycles per insn
      6.097.442      branches                  #  231,704 M/sec
        740.220      branch-misses             #   12,14% of all branches

    13,325266249 seconds time elapsed

     0,002955000 seconds user
     0,022656000 seconds sys
```

Figura (12) - Exibição de estatísicas sobre o código multithreading síncrono a partir da ferramenta perf

A partir das figuras 11 e 12, pode-se inferir que as reservas sincronizadas diminuíram as trocas de contexto, migrações de CPU e número de ciclos, ainda que o número de instruções não tenha sofrido grandes mudanças.

A diferença entre os resultados de cada método pode ser explicada pela sincronização, que ajuda a evitar desperdício de tempo e esforço, ou seja, a versão sem sincronização causa muita contenção e retrabalho, enquanto a sincronizada é controlada e eficiente. Mesmo com o "custo" do semáforo, a sincronização evita escritas duplicadas no mesmo assento, condições de corrida e tentativas inúteis de reservar lugares já ocupados.

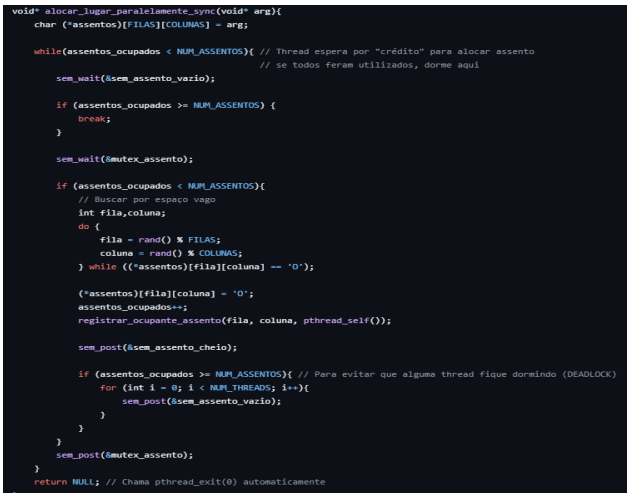
Durante a implementação, foram utilizados diferentes sistemas operacionais para testes das implementações. O Windows, por não ter um suporte nativo para a linguagem C, foi o que mais apresentou problemas para a codificação do problema e, é claro, para a execução do código, já que muitas das bibliotecas C presentes no Linux, simplesmente não estão presentes no Windows.

O MacOS, mesmo sendo um sistema Unix, também apresentou algumas inconsistências e erros, impossibilitando a execução do código inicialmente feito em Linux. Por isso, a fim de realizar uma experimentação, foi implementada uma versão alternativa da reserva de assento com utilização de *multithreading com sync.* As duas versões podem ser vistas e comparadas nas figuras logo abaixo.

```c
void* alocar_lugar_paralelamente_sync(void* arg){
    char (*assentos)[FILAS][COLUNAS] = arg;
    while(1){
        int fila = rand() % FILAS;
        int coluna = rand() % COLUNAS;

        sem_wait(&semid);
        if (assentos_ocupados >= NUM_ASSENTOS){
            sem_post(&semid);
            break;
        }
        if((*assentos)[fila][coluna] == 'L'){
            (*assentos)[fila][coluna] = 'O';
            assentos_ocupados++;
            registrar_ocupante_assento(fila, coluna, pthread_self());
        }
        sem_post(&semid);
    }
    pthread_exit(0);
}
```

Figura (12) – Implementação inicial. Foi codificada, testada e verificada em sistema Linux



```
void* alocar_lugar_paralelamente_sync(void* arg){
    char (*assentos)[FILAS][COLUNAS] = arg;

    while(assentos_ocupados < NUM_ASSENTOS){ // Thread espera por "crédito" para alocar assento
                                             // se todos feram utilizados, dorme aqui
        sem_wait(&sem_assento_vazio);

        if (assentos_ocupados >= NUM_ASSENTOS) {
            break;
        }

        sem_wait(&mutex_assento);

        if (assentos_ocupados < NUM_ASSENTOS){
            // Buscar por espaço vago
            int fila,coluna;
            do {
                fila = rand() % FILAS;
                coluna = rand() % COLUNAS;
            } while ((*assentos)[fila][coluna] == 'O');

            (*assentos)[fila][coluna] = 'O';
            assentos_ocupados++;
            registrar_ocupante_assento(fila, coluna, pthread_self());

            sem_post(&sem_assento_cheio);

            if (assentos_ocupados >= NUM_ASSENTOS){ // Para evitar que alguma thread fique dormindo (DEADLOCK)
                for (int i = 0; i < NUM_THREADS; i++){
                    sem_post(&sem_assento_vazio);
                }
            }
        }
        sem_post(&mutex_assento);
    }
    return NULL; // Chama pthread_exit(0) automaticamente
```

Figura (13) – Implementação em MacOS

Percebe-se, pela figura 13, o código visivelmente maior. Tecnicamente, também, a maior diferença é a utilização de mais de um semáforo na implementação. O resultado também é, performaticamente falando, bem inferior ao resultado da implementação em Linux.

Numa breve análise da tabela logo abaixo, que contém o resumo das métricas do resultado de cada solução, vemos que, ainda que a sequencial nos ofereça uma solução consistente, isto é, sem erros na alocação dos assentos do cinema, ela é quase três vezes mais lenta que a solução de multithreading adequada.

Os testes foram feitos em matrizes de quinhentas linhas por quinhentas colunas, totalizando duzentos e cinquenta mil assentos disponíveis para serem alocados. O multithreading sincronizado implementado em MacOS é mais lento que o feito para Linux e a solução sequencial, que aqui nos serve como base de comparação, ainda que ofereça uma solução consistente.

Apesar de ser ligeiramente mais rápido, o multhreading dessincronizado oferece uma solução não-íntegra, isto é, com erros de alocação. As condições de corrida geradas pelo acesso não-controlado das threads em seção crítica fizeram certos assentos não serem alocados ou serem alocados duas vezes, fazendo com que eles acabassem liberados.

Métrica	sequncial	Multithreading s/ sinc	Multithreading c/ sinc (MacOS)	Multithreading c/ sinc (linux)
Tempo médio de execução	4.0849s	3.5816s	4.7271s	1.4084s
Vazão média (assentos/s)	61.770	69.800	52.885	177.554
Total de inconsistências	0	33	0	0
Integridade dos Dados	Garantida	Corrompida	Garantida	Garantida
Speedup vs sequencial	1.0x	1.14x	0.86x	2.9x

Com os resultados, vemos que, com uma solução bem implementada, o uso de multithreading pode favorecer as buscas por uma solução ideal, isto é, uma solução mais rápida que um código single-thread e que não tenha inconsistências/falhas.

Conclusão

A realização deste projeto permitiu uma análise prática e detalhada dos desafios e impactos do uso de multithreading em aplicações concorrentes. Foi possível observar que a implementação sem sincronização, apesar de apresentar maior velocidade em alguns testes, compromete a consistência dos dados, gerando condições de corrida e registros incorretos. Já a versão com sincronização, mesmo que um pouco mais lenta, demonstrou eficiência na integridade dos dados e melhor controle de concorrência, evitando falhas como alocação duplicada de assentos.

A versão sequencial, apesar de estável, demonstrou menor desempenho em relação ao tempo de execução e à taxa de vazão, comprovando a superioridade do uso de threads quando bem sincronizadas. Assim, o projeto evidenciou a importância do uso correto de mecanismos de sincronização como semáforos para garantir a consistência e a segurança de sistemas concorrentes.

Recomendação de trabalhos futuros

Uso de mutexes e outras técnicas de sincronização: Comparar desempenho e integridade com outras formas de controle além de semáforos.

Interface gráfica (GUI): Implementar uma interface visual para facilitar a simulação da reserva de assentos.

Referências

[1] *Práticas da matéria de Sistemas Operacionais. Professor M. S. Oyamada. Unioeste – campus Cascavel. Cascavel, 2025.*

[2] M. S. Oyamada, SOPratica3Sincronizacao. Cascavel, 2025.

[3] M. S. Oyamada, SOPratica4Threads. Cascavel, 2025.

[4] M. S. Oyamada, Aula5Threads. Cascavel, 2025.

[5] Canal O Programador W, Linguagem C – Threads. Disponível em: https://www.youtube.com/watch?v=jHRUxUMT4dc.

[6] Canal O Programador W, Linguagem C – Threads e Mutex. Disponível em: https://www.youtube.com/watch?v=qFLpRBuoCG0.

[7] SILBERSCHATZ, A. et al. **Operating System Concepts, 9th Edition**. [s.l.] John Wiley & Sons, 2012.