Apostila de Exercícios - Concorrência e Distribuição

Baseada no Conteúdo Completo do Curso

Sessão 1 — Teoria de distribuição, concorrência e SO (mín. 10)

Questão 1 - Definições e contrastes

Explique, com exemplos reais:

(a) concorrência vs. paralelismo

(b) multiprogramação vs. multitarefa

(c) computação distribuída vs. paralela (memória compartilhada)

(d) preempção vs. cooperação.

Questão 2 - Caminho de uma system call

Descreva o caminho completo de uma chamada `write(fd, buf, n)` feita por um processo em modo usuário: transição de modo, papel da Tabela de System Calls, salvamento/restauração de contexto, e retorno ao usuário. Onde entram PC, SP e PSW?

Questão 3 - Interrupção vs. exceção vs. trap

Diferencie os três. Dê um exemplo típico de cada e como o SO reage (qual muda o PC? quem atualiza o PSW?).

Questão 4 - Context switch

Liste tudo que precisa ser salvo e restaurado num comutador de contexto entre dois processos/threads (inclua registradores gerais, PC, SP, PSW e metadados do agendador). Explique por que o custo do context switch afeta a granularidade de tarefas.

Questão 5 - Amdahl e Gustafson

(a) Enuncie a Lei de Amdahl e interprete o parâmetro "fração sequencial".

(b) Dê um exemplo em que a Lei de Gustafson descreve melhor o speedup do que Amdahl.

(c) Explique por que "mais threads" pode piorar o tempo.

Questão 6 - Afirmações — certo ou errado? Justifique.

(a) "Em monoprocessadores não há concorrência verdadeira."

(b) "Locks previnem deadlocks."

(c) "Busy-waiting é sempre pior que bloqueio."

(d) "Preempção garante fairness."

Questão 7 - Condições de Coffman

Dê um exemplo concreto que satisfaça as 4 condições de deadlock. Mostre duas formas diferentes de quebrar cada condição (total de 8 estratégias, duas por condição).

Questão 8 - Modelo de memória

(a) Por que `volatile` não torna código thread-safe?

(b) O que são data races e happens-before?

(c) Dê um exemplo de out-of-order execution que quebra uma suposição intuitiva do usuário.

Questão 9 - Caches e compartilhamento falso

Explique false sharing. Proponha um microbenchmark que evidencie a queda de performance e uma técnica para mitigá-la.

Questão 10 - Escalonamento

Compare FIFO, Round-Robin e CFS (ou outro justo). Qual política favorece throughput? Qual reduz latência interativa? Como o quantum afeta mix de cargas CPU-bound vs I/O-bound?

Questão 11 - PC, PSW, SP em detalhe

Dado um estouro de pilha em modo usuário:

(a) qual registrador detecta a violação?

(b) que bit no PSW muda?

(c) como o SO decide qual handler invocar?

(d) como o PC é atualizado para o handler e depois retorna?

Questão 12 - IPC e distribuição

Compare pipes, sockets, memória compartilhada e RPC/REST quanto a: latência, cópia de dados, facilidade de depuração e semântica de erro (falhas parciais em sistemas distribuídos).

Questão 13 - Prioridades e inversão de prioridade

Explique o fenômeno, dê um cenário com três tarefas (alta, média, baixa) e proponha duas soluções (ex.: herança de prioridade).

Questão 14 - Seção crítica e granularidade

Mostre como lock coarse-grained vs fine-grained muda: (i) desempenho, (ii) complexidade, (iii) risco de deadlock. Proponha uma métrica para decidir o ponto ótimo.

Questão 15 - Sinais (signals)

Em um processo multithread, quem recebe um `SIGINT`? Como controlar entrega a uma thread específica? Por que handlers devem ser async-signal-safe?

Sessão 2 — Fluxo em grafos e "linguagem join/fork" (mín. 10)

Convenção da linguagem (DSL) para este bloco

- `A -> B` = B começa após A (sequência).
- `fork {X, Y, Z}` = inicia X, Y, Z em paralelo (após o passo corrente).
- `join {X, Y} -> W` = W só inicia após X e Y terminarem.
- `start` e `end` delimitam o fluxo.

Questão 16 - Do texto ao grafo

Construa o grafo (ASCII ou desenhado) para:

```
start -> A -> fork {B, C} -> join {B, C} -> D -> end
```

Questão 17 - Do grafo ao texto

Para o grafo:

```
start
  |
  v
  A
 /\
v  v
B  C
 \/
  v
  D
  |
  v
 end
```

Escreva a descrição na DSL.

Questão 18 - Múltiplo join

Represente: A dispara B, C, D em paralelo; E só começa quando B e D terminarem; F só começa quando C terminar; G só começa quando E e F terminarem; depois end.

Questão 19 - Várias barreiras

Transforme a DSL em grafo:

```
start -> fork {A, B, C} -> join {A, B} -> E -> join {E, C} -> F -> end
```

Questão 20 - "Essa saída é possível?" #1

Para `start -> A -> fork {B, C} -> join {B, C} -> D -> end`, liste todas as sequências lineares possíveis de início/fim das tarefas (considere apenas ordem relativa que respeite dependências). Diga se A, C, B, D (como ordem de término) é válida.

Questão 21 - Corrida de dados no grafo

No fluxo:

```
start -> A -> fork {B, C} -> join {B, C} -> D -> end
```

suponha que B e C escrevem na mesma variável global sem sincronização. Explique como o grafo permite a condição de corrida e proponha duas correções no grafo (não no código).

Questão 22 - Caminhos críticos

Dado o grafo:

```
start -> A(2ms) -> fork {B(6ms), C(1ms)} -> join {B, C} -> D(3ms) -> end
```

(a) Qual o makespan mínimo?

(b) Qual o caminho crítico?

(c) E se C dependesse de B?

Questão 23 - Deadlock no grafo?

Construa um grafo com dois joins que, se implementado ingenuamente com dois locks, pode causar deadlock (pista: esperas circulares entre "join X" e "join Y"). Depois, reescreva o grafo para evitar.

Questão 24 - Granularidade das tarefas

Reescreva `start -> A -> B -> C -> D -> end` para maximizar paralelismo sem violar dependências dadas: B depende de A; C depende de A; D depende de B e C.

Questão 25 - Fusão e fissão

Dado:

```
start -> fork {A, B} -> join {A, B} -> fork {C, D} -> join {C, D} -> end
```

reorganize em uma única fork e uma única join equivalentes (assumindo recursos suficientes). Justifique.

Questão 26 - "Essa saída é possível?" #2

Para `start -> fork {A, B, C} -> join {A, B, C} -> end`, avalie se é possível a ordem de início: B começa, depois C, depois A, e de término: C termina, depois A, depois B. Explique por quê.

Questão 27 - Dependência condicional

Modele em DSL: A calcula x; se x > 0, executa B e C em paralelo e depois D; caso contrário, executa apenas E, e então D. (Use um nó de decisão textual, ex.: `if (x>0) { ... } else { ... }`).

Questão 28 - Pipeline em grafo

Modele um pipeline em 3 estágios P1, P2, P3 processando 5 itens com overlap (cada estágio dura 1ms). Esboce um grafo/cronograma que mostre o tempo total.

Questão 29 - Barreiras parciais

Escreva a DSL em que A dispara B, C, D, E. F deve aguardar somente B, D. G aguarda C, E. H aguarda F, G. Desenhe o grafo correspondente.

Questão 30 - Map-Reduce em DSL

Modele: Map de T1..T8 em paralelo, depois um Reduce único, e por fim Persist. Mostre o grafo e a DSL.

Sessão 3 — Programação (pthreads, create/join/exit, vetores, multifunções, promise/future, move) + análise de código (mín. 10)

Questão 31 - pthreads — soma em duas metades

Dado um `int v[10]`, crie duas threads que somam `v[0..4]` e `v[5..9]`. Passe struct com arr, inicio, fim; retorne a soma parcial via `pthread_exit` e combine na main. (Compilar com `-pthread`.)

Questão 32 - Particionamento por ID

Generalize o exercício anterior para N threads e vetor de tamanho M. Cada thread calcula seu intervalo a partir do id (i) usando divisão inteira. Não use globais. Teste casos em que M não é múltiplo de N.

Questão 33 - Verificação de primalidade em paralelo

Dado um vetor de `long int`, crie K threads; cada uma processa um chunk e imprime "id, número, primo?". Garanta que as linhas não se embaralhem (sugestão: mutex). Meça o tempo total.

Questão 34 - Produtor-Consumidor com condicional

Um chef (produtor) coloca inteiros numa fila de tamanho 5; três clientes (consumidores) retiram. Use `pthread_mutex_t` + `pthread_cond_t`. Termine com um poison pill para encerrar consumidores.

Questão 35 - Barreira com pthread_barrier_t

Execute 4 threads que fazem: fase1 (cálculo), barreira, fase2 (usa resultados parciais). Substitua depois por uma barreira manual via mutex+cond e discuta diferenças.

Questão 36 - Análise "essa saída é possível?" #pthread

Dado o código:

```c
int x = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void* A(void*) { x = 1; printf("A:%d\n", x); return NULL; }
void* B(void*) { printf("B:%d\n", x); x = 2; return NULL; }

int main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, A, NULL);
  pthread_create(&t2, NULL, B, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
}
```

(a) Liste saídas possíveis e improváveis.

(b) Reescreva para garantir que B sempre leia x==1 usando mutex ou barreira.

Questão 37 - Retorno seguro via pthread_exit

Mostre o bug ao retornar um int como `(void*)soma` em 64-bits. Corrija com `intptr_t` ou alocação dinâmica. Explique prós/contras de cada.

Questão 38 - Promise/Future — fatorial

Implemente: `std::promise<unsigned long long>` passado para uma thread que calcula n!. A main faz outras coisas e depois chama `future.get()`. Trate exceções se n for grande (ex.: lançar em `set_exception`).

Questão 39 - Promise/Future — várias tarefas

Lance 4 threads, cada uma com sua `promise<int>`; some os resultados quando cada future ficar pronto (não bloqueie tudo de uma vez). Dica: `wait_for` com polling leve ou when_all (se quiser usar biblioteca auxiliar), ou então desenhe uma fila de futures.

Questão 40 - std::async vs promise

Reescreva o exercício 38 usando `std::async` (lazily vs eagerly). Explique quando async evita boilerplate e quando promise é necessário.

Questão 41 - std::move em concorrência

Dado:

```cpp
std::promise<int> p;
auto f = p.get_future();
std::thread t(worker, std::move(p)); // move aqui
```

(a) Explique por que precisa de `std::move`.

(b) Mostre o erro típico ao passar p por cópia.

(c) Demonstre use-after-move incorreto da promise na main.

Questão 42 - Race condition sutil

Código:

```cpp

```

```
struct Data { int* v; int n; };
void* sum(void* arg) {
  Data* d = (Data*)arg;
  int s = 0;
  for (int i = 0; i < d->n; ++i) s += d->v[i];
  pthread_exit((void*)(intptr_t)s);
}
int main() {
  int n = 1000000;
  int* v = (int*)malloc(n*sizeof(int));
  for (int i=0;i<n;i++) v[i]=1;
  Data d{v, n};
  pthread_t t;
  pthread_create(&t, NULL, sum, &d);
  free(v); // <-- feito aqui!
  void* r; pthread_join(t, &r);
  printf("%ld\n", (long)r);
}
```

(a) Explique o bug.

(b) Corrija sem cópia do vetor (sugestões: lifetime, join antes do free, contagem de referências, smart pointers em C++).

Questão 43 - Falso compartilhamento

Implemente um microbenchmark com duas threads incrementando contadores em structs adjacentes. Meça throughput; depois alinhe cada contador a um cache line (ex.: `std::hardware_constructive_interference_size` ou padding manual) e compare.

Questão 44 - Análise "essa saída é possível?" — fork

Dado:

c

```
pid_t pid = fork();
if (pid == 0) { printf("C"); }
else { printf("P"); }
printf("X");
```

Liste todas as saídas possíveis e explique o porquê da ordem (considere buffering e escalonamento). Em seguida, force ordem determinística sem IPC pesado (pista: fflush, waitpid).

Questão 45 - Pipeline com threads

Três threads formam um pipeline: T1 lê e normaliza dados, T2 processa, T3 escreve. Use três filas thread-safe (ou uma por aresta) com mutex+cond. Garanta encerramento limpo (poison pill encadeado). Meça latência por item e throughput.

Questão 46 - Mapeamento de ID por partição

Escreva uma função que, dado `thread_id`, `num_threads`, `n_elems`, retorne `(start, end)` do bloco. Prove que cobre [0, n_elems) sem sobreposição, mesmo quando `n_elems < num_threads`.

Questão 47 - std::future_status

Lance uma tarefa com promise/future. Na main, faça uma UI loop que periodicamente chama `future.wait_for(50ms)` para atualizar uma barra de progresso. Saia com cancelamento quando demorar demais (simule com timeout). Mostre como sinalizaria à thread para parar.

Questão 48 - Revisão de código — mutex desnecessário

Dado:

```cpp
std::mutex m;
int acc = 0;
void worker(int n) {
  for (int i=0;i<n;i++) {
    std::lock_guard<std::mutex> lg(m);
    acc += i*i; // trabalho curto
  }
}
```

(a) Explique por que esse lock é gargalo.

(b) Otimize com acumulação local e uma única fusão protegida.

(c) Mostre o impacto teórico via Amdahl.

Questão 49 - Thread-safe logging

Implemente um logger com uma única thread consumidora que escreve no arquivo, enquanto N produtores apenas enfileiram mensagens. Feche corretamente (flush + poison pill). Discuta como evitar priority inversion.

Questão 50 - Comparativo de abordagens

Resolva a mesma tarefa (somar um vetor grande) de três formas:

(a) pthreads "puro",

(b) `std::thread` + `std::future` (via async),

(c) thread pool simples.

Compare: linhas de código, complexidade de sincronização, reuso, e desempenho para tamanhos pequenos vs. grandes.

Notas Importantes

Esta apostila foi desenvolvida para cobrir de forma abrangente todos os tópicos fundamentais de concorrência e distribuição, seguindo a estrutura curricular completa do curso. As questões foram elaboradas para:

- **Testar compreensão teórica profunda** dos conceitos de sistemas operacionais e concorrência
- **Avaliar capacidade de análise** de grafos de fluxo e transformações DSL
- **Verificar habilidades práticas** de implementação com diferentes APIs (pthreads, std::thread, futures)
- **Desenvolver pensamento crítico** na identificação de problemas de concorrência e suas soluções

Dicas para Resolução

1. **Sessão 1**: Foque na compreensão dos mecanismos internos do SO e nas implicações teóricas
2. **Sessão 2**: Pratique a conversão bidirecional entre representações gráficas e textuais
3. **Sessão 3**: Implemente os códigos e teste as saídas para validar sua compreensão

Recursos Necessários

- Compilador C/C++ com suporte a pthreads (`-pthread`)
- Conhecimento de conceitos de sistemas operacionais
- Familiaridade com APIs de concorrência modernas (C++11 threads, futures)
- Capacidade de análise de desempenho e debugging

Esta apostila representa o estado da arte em exercícios de concorrência e distribuição, cobrindo desde fundamentos teóricos até implementações práticas avançadas.