TRABALHO 0: CONCORRÊNCIA EM LEITURA/ESCRITA

RAUL H.C. LOPES

1. Introdução

O obejtivo do trabalho consiste em estudar variações sobre o problema de *readers-and-writers*. Para implementar este trabalho você deverá dominar os conceitos de:

- comunicação e sincronização via rendez-vous;
- comunicação e sincronização via monitores e objetos protegidos;
- concorrência no problema de leitua concorrente, escrita exclusiva;
- árvore B.

As seções a seguir detalham o trabalho, que deve ser implementado em Ada. O trabalho demandará dedicação sua em:

- conceber um algoritmos para trabalhar com árvore que maximizem o acesso concorrente;
- especificar os teste de consistência e desempenho dos mesmos.

Se você se sente desanimado por este trabalho ou pelo curso de **SO**, lembre-se que você está se formando em **Ciência da Computação** e que como tal sua habilidade principal deveria ser estudar algoritmos. Eu divido as atividades de cientistas da computação em três grupos:

- Investigação de algoritmos e computabilidade.
- Engenharia de artefatos, quando o profissional usa blocos prontos para produzir utilidades do dia-a-dia, uma atividade nobre e útil.
- "Engenharia" de nomes, quando o "profissional" se dedica à atividade de criar nomes novos para bois velhos, ou, como Dijkstra disse uma vez, criar nomes para montros que não consegue enfrentar na esperança de que, devidamente nomeado, o montro se submeta; os gregos faziam isso há 2.000 anos, muitos livros de IA fazem isso até hoje.

Eu ofereço cursos direcionados para o primeiro time. Respeito o segundo grupo, mas aviso que, dado que este é um curso de **Ciência**

da Computação, investigação de algoritmos é o preço a pagar pelo título que adquirirão.

Leia a última seção deste texto sobre as regras de execução e prazos de entrega do trabalho. A próxima seção descreve o tabalho.

2. O tabalho

Implemente ao menos os seguintes algoritmos para acesso concorrente a uma árvore B:

- pesquisa de um item;
- inserção de novo item;
- exclusão de item;
- lista de itens em ordem crescente.

Use uma das variantes de árvore B apresentadas em [3], que contém dicas excelentes sobre otimização de espaço e desempenho. Mas, note que os algoritmos ali apresentados são seqüenciais.

2.1. Os tipos de arquivos a indexar.

Sua árvore indexará arquivos estruturados em registros, cada um composto de strings separados pelo caracter ':' (dois pontos). Arquivo indexado e índice residirão em disco.

2.2. Os programas a implementar.

Implemente no mínimo:

- Pacote para acesso concorrente a árvore B que use tasks e rendez-vous;
- Programa interativo para teste do pacote acima;
- Pacote para acesso concorrente a árvore B que use tasks e objetos protegidos;
- Programa interativo para teste do pacote que usa objetos protegidos.

Os programas interativos terão as seguintes características:

- **Linha de commando:** com dois argumentos: o nome do arquivo a indexar e o nome do índice a gerar;
- ***stdin*:** lotes de operações. Um lote de operações é uma seqüência não vazia de linhas terminada por asterisco ($*$). Cada lote define um conjunto de seqüências de operações que devem ser processados concorrentemente. Cada seqüência de operações é apresentada em uma linha de *stdin*. Repetindo:
  - lotes são separados por asterisco ($*$): o fim de um lote é marcado por uma linha contendo apenas asterisco seguido de *newline*;
  - lotes são processados concorrentemente;
  - lotes são compostos por seqüências de operações;

– cada seqüência de operações é terminado por *newline* (uma seqüência por linha de *stdin*);
– as operações de uma seqüência são processadas seqüencialmente (mas, concorrentemente com as operações das outras seqüências do mesmo lote;)
– o processamento de novo lote só começa depois que o lote anterior foi inteiramente processado.
– Uma seqüência de operações contém operações separadas por asterisco.
– Uma operação pertence a um dos seguintes tipos:
  S:<key>
  Encontre registro com chave <*key*>. Seu programa deverá produzir no output:
  * caso encontre a chave no índice:
    `S:<key>:<reg><nl>`
  * caso não encontre a chave no índice:
    `S:<key>:<nl>`

  onde <*key*> é a chave procurada, <*reg*> é seu registro no arquivo, <*nl*> é *newline*.
  D:<key>
  Exclua chave <*key*> do índice.
  Seu programa não produzirá no output.
  I:key:reg
  Adicione chave $< key >$ ao índice e registro $< reg >$ ao arquivo de dados.
  Seu programa não produzirá output.
  L:<key>
  Lista em ordem crescente, para todas as chaves maiores do que <*key*>, a chave e o registro associado. Os pare chave e registro serão listados no *stdout*como:
  `<key>:<reg><nl>`

2.3. **Otimizando...** Seu programa poderá otimizar:

- *throughput*;
- *turnaround time*;
- *response time.*

2.4. **Testes.** Especifique testes que validem seu programa em termos de ausência de erros e desempenho. Implemente programas que executem os seus testes e coletem os resultados.

2.5. **O que entregar.** O seu trabalho consistirá em um arquivo(s) LaTeX que conterá(ão) e documentará(ão):

- o fonte de cada programa/pacote;
- a especificação completa de todos os testes usados para testar a consistência de seus programas, incluindo programas auxiliares (o uso de ferramentas para auxílio à automatização de teste será valorizado;)
- os resultados dos testes de consistência;
- especificação dos testes de desempenho e resultados obtidos.

Você deverá entregar um *tarball* contendo *makefile* e fontes do LaTeX.

3. Regras para execução do trabalho

O trabalho substitui parte do peso das provas, logo valem as seguintes observações:

- Este trabalho poderá ser executado em grupos de até 2 alunos. No entanto, valem as seguintes observações:
  - Os próximos tabalhos dependem deste;
  - Se durante a execução deste trabalho ou do próximo ficar claro que um dos elementos do grupo não trabalhou (famoso parasita no vácuo do bobo), eu anulo o trabalho do grupo.

  Moral da história: se seu colega de grupo se candidatar a parasita no seu vácuo (ou em outra posição mais delicada e incômoda), afaste-o do grupo e eu terei prazer em exterminá-lo.
- Trabalhos individuais valerão mais: terão peso 1.4 na nota final, reduzindo o peso da médias das provas em 1.4.
- Você pode trocar idéias com colegas, mas cuidado com o nível de troca. Plágio neste curso é considerado falta gravíssima: $d > 1$ plágios anulam 2^d trabalhos/provas.
- O trabalho é seu e não do professor. Eu darei referências e dicas, mas cabe a você desenvolver algoritmos, implementar programas, definir casos de teste, suar, apanhar, sofrer...

3.1. **Linguagem de programação.** Os pacotes de árvore B concorrente deverão ser implementados em Ada.

3.2. **O produto.** O produto a ser entregue consiste em documento LaTeX de onde será possível extrair todos os fontes necessários para compilar os programas e executar os testes. O seu documento deverá conter:

- descrição dos algoritmos de cada programa, com referências para as fontes dos mesmos;

- fontes dos programas;
- especificações de testes, bem como suas implementações e resultados.
- especificações de testes de desempenho, bem como suas implementações e resultados obtidos;
- comparações entre resultados.

Você poderá apresentar sua documentação distribuída por vários documentos LaTeX. Mas, garanta que as referências cruzadas entre documentos estão corretas e aprenda a usar o pacote *hyperref* do LaTeX para criar links entre documentos. Aliás, se seu Linux/FreeBSD tiver o pacote *teTeX-doc* instalado, você poderá ver documentação sobre os pacotes LaTeX disponíveis na sua instalação em:

/usr/share/texmf/doc/index.html

Qualidade de documento (que contribui para a nota final), inclui o uso competente do LaTeX para gerar, por exemplo: referências cruzadas, referências bibliográficas (use *bibtex*), equações e tabelas decentemente formatadas. Por exemplo, índice remissivo de símbolos principais dos programas é extremamente bem-vindo. O documento ideal deveria ter a cara do TeXbook [2].

3.3. **Testes.** É bem-vindo e recomendado o uso de ferramentas de auxílio a teste automático como: *aunit*, *expect*, *dejagnu*, etc.

Importante: testes bem especificados são baseados em condições de correção formal de algoritmos: testes de loops, por exemplo, garantem progresso, término, segurança (propriedade invariante.) Em resumo, validação por teste nao elimina a necessidade de conhecer os fundamentos de prova formal de correção de programas. A avaliação da qualidade dos seus testes levará isso em conta.

3.4. **Arquivo submetido.** Você submeterá um aquivo de nome

tp0.tar.gz

Eu usarei os seguintes comandos para obter seus programas compilados e seu documento em formato *PDF*:

```
> gzip -dc tp0.tar.gz | tar -xf -
> make pdf
> make compile
> make consist
> make perform
```

a seqüência acima gerará ao menos:

- programa de nome

  rvbtree

  que testará o pacote de *B-tree*, usando rendez-vous.
- programa de nome

  pobtree

  que testará o pacote de *B-tree*, usando objetos protegidos.
- arquivo *PDF* de nome

  article.pdf

  que conterá o documento básico sobre o trabalho, contendo, possivelmente, links para outros documentos.
- arquivo com resultados de teste de consistência em formato a ser especificado.
- arquivo com resultados de teste de desmpenho em formato a ser especificado.

3.5. **Data de entrega.** O prazo final para entrega do trabalho será definido em acordo com a turma. Em princípio, proponho 14/02/2003. É importante que as seguintes restrições sejam obedecidas:

- Haverá 3 trabalhos práticos que poderão ser feitos em grupo, e um trabalho individual. Além disso, o último trabalho só poderá ser feito por quem fizer o segundo.
- O último trabalho deve ser entregue ao final da última semana de aula.
- entre a entrega de cada trabalho deverão transcorrer, ao menos, 10 dias.

Importante: são 4 trabalhos e talvez isso pareça carga excessiva. Mas, ninguém é obrigado a fazer qualquer um deles.

3.6. **Novos detalhamentos.** Fique atento à minha página para: novos detalhes sobre os trabalhos e dicas sobre o uso de ferramentas úteis, como: LaTeX, ferramentas de teste, *make*, programas para extração automática de fontes de programas contidos em documentos LaTeX.

4. A atribuição de nota

Cada pacote implementado (juntamente com seu programa de teste) poderá receber até 4.6 pontos, independentemente de desempenho. Além disso, o trabalho poderá ganhar 1.0 por ausência total de erros e 0.4 por melhor desempenho geral. Total: 10.6 pontos em 10!

4.1. **Nota básica por pacote.** A nota básica por pacote atribuirá pontos de acordo com a tabela 1.

Detalhes dos itens avaliados seguem:

Verificação de consistência	1.8
Verificação de desempenho	0.6
Documentação	0.2
Qualidade de pacote	0.4
Desempenho em concorrência	1.6

TABLE 1. Itens básicos por pacote

- **Verificação de consistência:** Levará em conta qualidade dos testes especificados para deteção de erros. Serão valorizados baseados em condições de correção dos algoritmos. A especificação deverá conter:
  - o teste realizado e sua justificativa;
  - o resultado esperado;
  - resultado obtido.
- **Verificação de desempenho:** Cada trabalho deve apresentar um conjunto de testes verificando o desempenho do mesmo em diferentes circunstâncias.
- **Documentação:** Este item, embora tendo pouco peso aparente, um grande peso implícito. Fatos:
  - Só serão corrgidos algoritmos e programas que estejam documentados. Se o aluno usar um algoritmo revolucionário e não apontar claramente para isso, poderá nem ter o trabalho corrigido.
  - Testes não especificados serão considerados como nulos.

  Além disso, a documentação deve providenciar acesso fácil a instruções de uso dos programas e todas as suas características importantes.
- **Qualidade do pacote:** Qualidade do pacote como implementação de um ADT.
- **Desempenho em concorrência:** Cada pacote participará de competição para avaliação de desempenho em concorrência em quatro classes:
  - **best throughtput;**
  - **best turnaround time;**
  - **best response time;**
  - **variance in response time.**

  Cada classe avaliará o pacote nos itens:
  - Busca concorrente;
  - Busca e inserção concorrentes;
  - Busca e exclusão concorrentes;

Consistência geral	1.0
Best hacker prize	0.4
Especificação inconsistente	−1.0 cada

TABLE 2. Itens adicionais de pontuação

– Lista concorrente de todos os ítens.

Em cada item avalidado (quatro itens vezes quatro classes dando 16 itens), o vencedor ganha 0.1.

4.2. **Itens adicionais.** Pontos poderão ser adicionados ou retirados de acordo com a tabela 2.

Esses itens estão detalhados a seguir:

- **Consistência geral:** Todo trabalho que contiver os dois pacotes implementados e respectivos programas de teste recebe 1.0 de bônus. No entanto, cada erro detectado descontará 0.1 desse bônus.
- **Best hacker prize:** O vencedor de todas a classes de ganha 0.4.
- **Especificação inconsistente:** cada caso de teste especificado na documentação que apresente resultado obtido nos testes do *Castelo* diferentes do que estiver contido na documentação retira 1.0 da nota final do trabalho. Moral da história: coloca na documentação só o que você testou.

4.3. **Crime e punição.** Serão anulados:

- trabalhos enquadrados como plágios, com conseqüências graves para outros trabalhos porque $d > 1$ plágios ao longo semestre anulam 2^d trabalhos;
- trabalhos em grupo em que um dos componentes do grupo seja enquadrado como parasita no vácuo;
- trabalhos que apresentem qualquer erro no processo de descompactação, compilaçao, ou geração de documentos: siga estritamente as regras da seção 3.4.

REFERENCES

[1] John Bentley and Robert Sedgewick, *Ternary tree*, Dr. Dobb's (1998).
[2] Donald E. Knuth, *TeX: The program*, Addison-Wesley, 1986.
[3] ______, *The art of computer programmimg, volume 3: Sorting and searching*, Addison-Wesley, 1998.