TRABALHO 1:
UM PACOTE DE THREADS

RAUL H.C. LOPES

1. Descrição

Implemente um pacote de threads para o Linux. Seu pacote deverá contemplar políticas de escalonamento de CPU e sincronização de processos. Use o pacote *QuickThreads*, fornecido no mesmo *tarball* que este documento. Não será permitido o uso de nenhum procedimento da *pthreads* ou de qualquer outra *library* que não seja *QuickThreads*.

2. Roteiro

(1) Leia os documentos *README* e *doc/qt.ps* do pacote *QuickThreads*.
(2) Veja exemplo de um *thread scheduler* contido no mesmo pacote (em *stp.c*) e exemplo de uso em *meas.c*.
(3) Proponha uma política de escalonamento multi-nível que implemente ao menos *FCFS*, *SJB* e *RR*. Sua política multi-nível deverá evitar *starvation* e deverá permitir o escalonamento de processos do seu *kernel* em tempo real.
(4) Estabeleça um conjunto de primitivas de sincronização para seu pacote, baseadas em comunicação entre processos.
(5) Implemente os exemplos definidos na seção 3 para validar o seu trabalho.

2.1. **O que entregar.** O seu trabalho consistirá em um artigo escrito em LaTeX (não é *noweb*), fontes do escalonador, dos programas de validação e do *test driver*, além de *Makefile*.

3. Testes para validação

A validação do seu escalonador será realizada através de um conjunto de testes sobre problemas clássicos de sincronização de processos. Você escolherá dentre três tipos de testes para validar seu pacote de threads.

3.1. **Obrigatório.** Todos os pacotes deverão implementar input/output padrão (equivalente de *stdio*), usando um processo concorrente com o escalonador para realizar I/O assíncrono. Siga o roteiro:

(1) Use a primitiva **_clone** para criar um processo, denominado *assyncio* que compartilhe memória com seu escalonador.
(2) Todo input/output de threads do seu pacote deverá realizado através de *assyncio*.
(3) Invente um mecanismo de sincronização e comunicação entre os processos de seu escalonador e o processo *assyncio*.

Todos os pacotes implementarão seu próprio pacote de memória dinâmica, substituindo a dupla *malloc/free*. Implemente ao menos:

- um processo no pacote, que será escalonado em tempo real, e que alocará ao início um grande bloco de memória do Linux;
- esse processo aceitará pedidos de alocação ou liberação de memória através de primitivas *request(n)* (pedido de n bytes de memória), e *release* (pedido de liberação de bloco;
- este processo será escalonado em tempo real, sendo dirigido por evento;
- ele poderá implementar política SJN (atende primeiro quem requisita menos) ou FCFS, ou os dois: comparações serão valorizadas.
- use algoritmos como *best fit* para implementar a alocação de memória.

3.2. **Opcional nível 0.** Implemente uma árvore B concorrente em disco (como especificada no TP0) para validar seu pacote.

3.2.1. *Mínimo.* As seguintes restrições devem ser seguidas:

- Todo input/output para disco deve ser mapeada em memória, usando algoritmos de paginação em memória virtual.
- Um processo (criado via **_clone**) será responsável pela realização de acesso a disco.
- Um processo de tempo real do seu escalonador resolverá eventuais problemas de *page-fault*.

3.2.2. *Peso.* Esta alternativa tem peso 3 na avaliação.

3.2.3. *Férias.* Mesmo que você nao implemente esta alternativa agora, você poderá fazê-lo até o dia 30/04 e o resultado poderá ser usado para corrigir sua nota, desde que você não tenha sido reprovado no período normal. IMPORTANTE: só será realizada correção de nota para quem tiver sido aprovado no período normal.

3.3. **Opcional nível 1.** Implemente uma árvore B concorrente, armazenada em memória, como especificada no TP0.

3.3.1. *Restrições.* A seguintes restrições devem ser seguidas:

- Todo input/output padrão será realizado através de *assyncio*.
- A alocação de memória dinâmica será realizada pelo seu pacote de threads.

3.3.2. *Peso.* Esta alternativa tem peso dois na avaliação.

3.4. **Opcional nível 2.** Implemente programas para resolver os seguintes problemas:

(1) **Roller Coaster**: Existe um linha de ônibus com C carros circulando, todos realizando o mesmo trajeto e obedecendo às restrições:
   - todos os passageiros embarcam no mesmo ponto;
   - um carro só sai do ponto se tiver p passgeiros embarcados;
   - cada passageiro é modelado por um processo que eventualmente requisita entrada em um carro;
   - cada carro é modelado por um processo que demora um tempo randômico maior do que zero para dar uma volta completa na linha.

   Escreva um program que simula a sincronaizacao de um número arbitrário de carros e passageiros.

(2) **Jantar dos canibais**: Numa tribo de canibais todos comem do mesmo caldeirão, onde são cozinhados picadinhos de missionários metidos (ok... todo missionário é metido!) Cada canibal vive em três estados: dorme, caça missionários, come. Se um canibal encontra o caldeirão vazio (cada caldeirão serve c porções), ele acorda o cozinheiro, que vive em dois estados: dorme, cozinha. Implemente um simulador para essa vida tribal chata. Considere melhorar o padrão de vida tribo, criando um bordel onde são admitidos no máximo b canibais normais. Considere também a existência de canibais tarados: cada um vale por dois normais. Admita que com um bordel na área o cozinheiro também pode melhorar seu padrão de vida. Lembre-se: você deve definir e implementar condições de exclusão mútua, *liveness* e ausência de *deadlock*, inclusive no bordel. Ah! Quanto mais realístico, melhor.

(3) **Acesso concorrente a fila**: Dada uma lista encadeada de itens, as seguintes operações são admissíveis:
   - **insert**: insere item ao final da lista;
   - **delete**: exclui item qualquer;

• **search**: procura item.

Implemente um processo que mantém um lista com essas operações e que maximiza o acesso concorrente a ela. Implemente simulador que teste esse pacote.

3.4.1. *Restrições.* A seguintes restrições devem ser seguidas:

- Todo input/output padrão será realizado através de *assyncio*.

3.4.2. *Peso.* Esta alternativa tem peso um na avaliação.

4. 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, vale a velha regra: se durante a execução deste trabalho 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 multiplicado por 1.5.
- 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...

4.1. **Linguagem de programação.** Você poderá usar *C/C++* e, eventualmente, linguagens de script.

5. 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.

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

tp1.tar.gz

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

```
> gzip -dc tp1.tar.gz | tar -xf -
> make pdf
> make compile
> make consist
> make perform
```

5.2. **Entrega.** O prazo final para entrega do trabalho é 05/04/2003, 9 horas, via e-mail, que conterá:

- **Subject**: OS
- **corpo da mensagem**: identificção dos autores;
- **attachment**: o arquivo *tp1.tar.gz*

Cuidado! Não erre. Não há tempo para novas submissões.