

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL- INSTITUTO DE INFORMÁTICA
DEPARTAMENTO DE INFORMÁTICA APLICADA - INF01151 - SISTEMAS OPERACIONAIS II
PROFESSOR: Marcelo Johann - SEMESTRE: 2012-1 – AVALIAÇÃO GVGO - 18/12/2012

Nome: _____ Cartão: _____

1. (2.0) Faça um resumo geral sobre os dois temas do trabalho e a discussão que foi realizada em aula.

2. Notas do professor:

- a) (2.0) sua a participação na discussão desse trabalho: XXXX (Calculada pelo Prof.)
- b) (2.0) nota sobre o material apresentado pelo grupo: 2.0

3. (2.0) Responda às quatro questões abaixo:

- a) Quais a diferença das implementações de memória compartilhada distribuída entre o Chorus, Amoeba e Mach?
 - b) Como funciona e onde são usadas as DHT – *Distributed Hash Tables*?
 - c) Para usar WebServices quais são os sistemas e elementos básicos necessários, quem os programa?
 - d) Desenhe o relacionamento entre *Cliente*, *Servidor*, *IDL Description* e *IDL Compiler*, *Stub*, *Skeleton* e *ORB* em um sistema baseado em CORBA e diga o fornecedor de cada um desses elementos, ou seja quem escreveu o código dele?
4. (2.0) Dê uma nota ao seu desempenho nessa atividade do GVGO, justificando sua resposta: _____
5. (sem valor) Qual a sua opinião sobre a realização das discussões nesse trabalho?

Nome: José Reny Jorge Joss

Cartão: 180121

1. (1.0) Quem deve construir o código dos STUBs para fazer *RPC*? (Marque apenas uma alternativa)

- O programador do servidor O sistema operacional O *Hardware*
 Um gerador automático O programador da aplicação

2. (0.8) Marque as alternativas corretas:

A troca de mensagens é um mecanismo mais simples para programação concorrente se comparado ao compartilhamento de memória;

Um servidor é quase sempre composto por uma estrutura assim:

while (1) { recv(&msg); switch(msg.type) {case1: pthread_create(service_1,msg); break; case 2: ... } }

O uso de multicomputadores (*grids* ou *clusters*) é atualmente uma tendência, entre outros fatores, porque há limites físicos para construir uma única CPU de altíssimo desempenho, e quando isso é possível ainda assim a CPU única ainda seria cara;

O aspecto mais importante no desenvolvimento de um sistema distribuído é o seu desempenho, não importando se para isso tivermos que usar modelos de programação diferentes para comunicação e acesso a recursos remotos, com a consciência de onde eles se encontram;

3. (1.0) Sobre comunicação por troca de mensagens, identifique qual das afirmativas abaixo é falsa:

Primitivas de envio de mensagens não bloqueantes podem ser implementadas fazendo com que o S.O. copie a mensagem a ser enviada ou então avisando o usuário quando o *buffer* for liberado através de uma interrupção do processo do usuário, desviando o controle dele para um ponto de tratamento.

Somente primitivas de **envio** de mensagens podem ser bloqueantes ou não bloqueantes

Tanto primitivas *bufferizadas* quanto não *bufferizadas* correm o risco de perder mensagens recebidas se o receptor não está lendo elas suficientemente rápido.

É possível identificar processos servidores no sistema deixando que eles escolham endereços de um espaço grande, e localizando-os por *broadcast* na rede, mas isso consome banda significativa da rede.

Os protocolos de TCP e UDP são representados no mesmo nível do modelo OSI/ISO, no entanto UDP não oferece garantia de entrega e ordenamento das mensagens.

4. (1.0) Qual é o principal problema que limita o uso de *RPC*?

O principal problema com o uso de RPC é a sincronia de parâmetros por referência a um procedimento de outra máquina, pois RPC não suporta automaticamente esta forma de programação de parâmetros, demandando que os parâmetros sejam passados por valor. Isso torna as

5. (1.2) Observe as sequências de acessos à memória e valores lidos ou escritos, e marque na tabela:

a	$P_1 \xrightarrow{w(x)1}$	b		$P_1 \xrightarrow{w(x)1}$	$P_2 \xrightarrow{w(x)2}$	$P_3 \xrightarrow{r(x)1 \quad r(x)2}$	$P_4 \xrightarrow{r(x)2 \quad r(x)1}$	$P_2 \xrightarrow{w(x)2}$	$P_3 \xrightarrow{r(x)1 \quad r(x)2}$	$P_4 \xrightarrow{r(x)1 \quad r(x)2}$	<i>mensagens mais pesadas e afeta o desempenho do sistema</i>
		seqüência a	seqüência b								
	válida	inválida		válida	inválida						
Em modelo de consistência estrita		X									
Em modelo de consistência seqüencial		X			X						
Em modelo de consistência causal	X					X					

	seqüência a		seqüência b	
	válida	inválida	válida	inválida
Em modelo de consistência estrita		X		
Em modelo de consistência seqüencial		X	X	
Em modelo de consistência causal	X		X	

6. (1.5) Cite os três modelos básicos de semântica de compartilhamento de arquivos e diga sucintamente o que os caracteriza (e diferencia).

- Semântica UNIX: os arquivos podem ser lidos e editados simultaneamente por mais de um processo, porém é necessário atentar para a sobreposição de informações, visto que uma processos podem alterar informações de outros no mesmo endereço.
- Semântica de rede: arquivos podem ser lidos e editados, porém as alterações em um arquivo só são visíveis a outros processos a partir do momento que o arquivo é fechado.
- Semântica restritiva: arquivos só podem ser criados, lidos e excluídos. Não é permitida a edição.

1.0 7. (1.5) Caracterize o conceito de algoritmo distribuído, dizendo no que ele se distingue de um algoritmo centralizado, justificando sua necessidade, e mostrando também qual o principal desafio para seu desenvolvimento.

Um algoritmo distribuído não tem diferentes máquinas e apresenta problemas como sincronismo temporal e espera para o envio de mensagens.

Um algoritmo centralizado não possui problemas com sincronismo temporal, visto que a relojo é utilizado pelos programas, não sempre a mesma, ou seja, não haverá desordem temporal. Durante o envio de mensagens, algoritmos centralizados dispendem menor tempo aguardando para enviar ou receber mensagens, devido a proximidade física.

Para o sincronismo temporal na algoritmos distribuídos são necessárias técnicas que sincronizem as diferenças de tempo entre as diferentes máquinas e a espera no envio de mensagens. → (folha anexa)

0.5 8. (1.0) Como se faz para implementar o algoritmo de sincronização de relógio lógico de Lamport, com ordenação total de eventos no sistema? Elabore sua resposta em forma de algoritmo, determinando os testes e ações que devem ser implementadas, e quando isto deve ser executado, além de especificar regras que devem ser válidas para o sistema.

1) é remetente

1.1) prepara mensagem e anexa o tempo local na mensagem.

2) é receptor

2.1) recebe a mensagem

2.2) analisa o seu tempo local com o tempo da sua mensagem.

2.2.1) se tempo local < tempo mensagem, então o tempo local é atualizado para o tempo da mensagem + uma unidade de tempo

2.2.2) se tempo local > tempo mensagem, então o tempo local segue o mesmo.

9. (1.0) Em um sistema de votação com fantasmas para atualização de arquivos replicados, até quantas máquinas podem estar fora do ar para continuarmos implementando leituras e escritas corretas?

Podem estar fora do ar até $n-2$ máquinas, visto que uma demais estará fazendo leituras e escritas e a outra (fantasma) irá utilizar as alterações feitas nos arquivos, mantendo a consistência do sistema.

Nome: joão mig. f. gomes Cartão: 180171

2.2 1. (3,0) Dentre as afirmações abaixo, assinale aquelas que estão corretas, e somente elas:

- [V] Em um sistema que modela *threads* em seu *kernel*, quando se usa mapeamento de *threads* N:1, uma *thread* de usuário bloqueada causará o bloqueio de todas as *threads*;
- [V] Em um sistema operacional que não possui o conceito de *threads*, ainda é possível programar usando diferentes linhas de execução para um mesmo processo;
- [V] Em um programa Java, novas *threads* que são criadas sempre executam apenas o código que está escrito dentro da rotina declarada como "*public void run()*";
- [V] A chamada de sistema *Test And Set* executa uma operação atômica que testa e muda o valor de uma variável, sem possibilidade de interrupção, e, portanto, é um recurso que o sistema operacional oferece para implementar procedimentos de sincronismo de forma mais simples.
- [F] Toda solução para exclusão mútua necessita de operações *Test And Set* implementadas pelo *hardware* (processador);
- [V] Um grafo de fluxo de processos que não é propriamente aninhado pode ser implementado em C sobre um sistema operacional que implemente o padrão POSIX utilizando *threads* e semáforos;
- [V] Um grafo de fluxo de processos que não é propriamente aninhado pode ser implementado em C sobre um sistema operacional UNIX utilizando as chamadas *fork*, *execve* e *join*;
- [F] Condição de corrida é a indefinição da ordem em que instruções de diferentes processos vão executar, em decorrência do fato de que os processos podem perder a CPU a qualquer momento.
- [V] O término da fatia de tempo é um evento que passa um processo para o estado de apto a usar a CPU, ou seja, o coloca na fila de "prontos";
- [V] O sincronismo através de semáforos exige que os vários trechos concorrentes tenham acesso às mesmas variáveis, pois é assim que acessam o semáforo para se sincronizarem, e portanto, somente pode ser utilizado para sincronismo de *threads* dentro de um mesmo processo;
- [F] Programação concorrente somente confere maior velocidade ao programa quando se utilizam máquinas com múltiplos processadores, ou múltiplos *cores*, ou quando o processador tem *hyperthreading*.
- [V] O padrão POSIX define semáforos que são "*kernel-persistent*", isto é, uma vez criados, eles são mantidos após a morte do processo, como o valor que foi deixando, o que requer cuidado na inicialização;
- [V] O padrão POSIX define segmentos de memória compartilhada modelados como entradas no sistema de arquivo. Após a morte do processo que os criou, esses segmentos "de memória" continuam existindo;
- [V] Um trecho de código em Java que faz: "*while (condição) semaforo.P();*" é um exemplo típico de *busy waiting*, o que é fácil perceber pela construção '*while*', que permanece continuamente testando a condição.
- [F] Um programa concorrente com condição de corrida apresentará sempre um resultado diferente a cada vez que for executado novamente, o que consiste um problema de correção;

2. (2,0) **Produtor e Consumidor com Semáforos** - Escreva o código simplificado para n processos, produtores e consumidores, que operam sobre um *buffer* circular de tamanho máximo *N*. Você precisa garantir exclusão mútua no acesso ao *buffer*, e controlar o número de itens e de espaços existentes. Todos os controles devem ser feitos com semáforos. Você deve declarar todas as variáveis, os semáforos utilizados e seus valores iniciais. Não esqueça de representar os laços do programa, e considere semáforos diferentes para exclusão mútua dos escritores e dos leitores.

~~3. (1,0) Produtor e Consumidor com Monitores~~ - Escreva o código simplificado para n processos, produtores e consumidores, que operam sobre um *buffer* circular de tamanho máximo N , fazendo todo o sincronismo com monitores, em uma linguagem que implemente diferentes variáveis de condição.

~~4. (1,0) Produtor e Consumidor com Monitores~~ – Adapte a sua solução da questão anterior para que seja implementada em Java. Você pode abstrair detalhes de sintaxe, tratamento de exceções e usar nomes de classes e métodos que não sejam exatamente os mesmos, mas deve considerar as principais particularidades semânticas da linguagem.

~~05~~ ~~5. (1,0)~~ Escreva um esboço de implementação das operações de P e V de semáforos no *kernel*. Suponha que já existam rotinas para inserção e remoção de descritores de processos em filas, e não esqueça de declarar valores iniciais das variáveis.

~~06~~ ~~6. (1,0)~~ Compare a implementação de um programa concorrente feito com *threads* e feito com processos. Cite as vantagens e desvantagens de cada uma, bem como as principais decisões de projeto e estrutura de código em cada caso.

~~7. (1,0) Semáforos Contadores com Semáforos Binários.~~ Semáforos binários são o mesmo que *mutexes*: objetos que somente podem estar em dois estados, livres ou bloqueados (1 ou 0), não contando com números positivos o número de operações de liberação, mas bloqueando um número qualquer de processos em sua fila. Imagine que vocês precise agora implementar Semáforos Contadores como os conhecemos, usando apenas semáforos binários, sem acesso ao Kernel ou outras formas de sincronismo, sem qualquer outra estrutura centralizadora e sem espera ativa (*busy waiting*). Observe a proposta de implementação abaixo (<http://www.cs.umd.edu/~shankar/412-Notes/10-BinarySemaphores.html>), em um código semelhante a Pascal, e diga se a implementação é correta, ou explique por que não é.

Counting Semaphore construct: Semaphore S initially K Implementation using Binary Semaphores	record S { integer val initially K, // value of S or # of processes waiting on S BinarySemaphore wait initially 0, // wait here to wait on S BinarySemaphore mutex initially 1 // protects val }
P(S) { P(S.mutex); if S.val <= 0 then { S.val := S.val - 1; V(S.mutex); aqui: P(S.wait); } else { S.val := S.val - 1; V(S.mutex); } }	V(S) { P(S.mutex); if S.val < 0 then { V(S.wait); S.val := S.val + 1; V(S.mutex); } }