EA872 - Laboratório de Programação de Software Básico

Atividade 9

Gerenciamento de processos em um sistema multiprogramado
Aplicações práticas

1. Objetivos

Familiarização com chamadas de sistema (*system calls*) no ambiente UNIX. Implementação de processos num sistema multiprogramado. Introdução às chamadas de sistema para manipulação e sincronização de processos. Nesta atividade serão aplicados os conceitos de concorrência e de multiprogramação no servidor web em desenvolvimento.

2. Aplicação de técnicas de multiprogramação em um servidor WEB

- Nesta etapa, a atividade relacionada ao desenvolvimento do servidor WEB consiste na introdução de concorrência no servidor. Cada requisição recebida por ele será tratada por um processo independente de maneira a aumentar sua eficiência e atender um maior número de requisições/seg.
- Deverá ser definido um número máximo (N) de processos-filho que serão criados para tratar as requisições recebidas da rede. Cada processo-filho trata uma única requisição e termina (*exit*).
- Sempre que o servidor receber uma requisição de um navegador web e já tiver criado o número máximo de processos-filho pré-definido, deverá enviar uma mensagem (página html) de indisponibilidade ao navegador e sugerindo que o usuário tente mais tarde.
- Caso ainda não tenha atingido o limite de N processos-filho, o servidor deverá criar mais um para atender exclusivamente a requisição recém-chegada e terminá-lo após o atendimento (*exit*).
- Deverá ser feito um controle preciso do número de processos-filho por meio de incremento/decremento em uma variável.
- O aluno deverá fazer as alterações no servidor web desenvolvido até agora para atender os requisitos acima, observando as particularidades descritas a seguir.

2.1. Sobre fechamento de conexões no HTTP 1.1

O protocolo HTTP versão 1.1 sempre mantém uma conexão TCP/IP ativa mesmo após atendida a requisição que a originou. Isto permite que uma mesma conexão atenda múltiplas requisições, eliminando assim o *overhead* de abertura e fechamento de conexões. Caso o cliente não deseje manter a conexão ativa, a mensagem HTTP de requisição terá o parâmetro *Connection* com o valor *Close*. Por outro lado, não é recomendado que o servidor HTTP 1.1 permita que conexões permaneçam ativas indefinidamente, pois conexões são recursos limitados de comunicação. Neste caso, o servidor pode (e deve) encerrar uma conexão caso a mesma fique inativa por um período de tempo (5 segundos, por exemplo). Veja o início do Cap. 9 da RFC9112 disponibilizada em uma atividade anterior.

Nesta atividade, ao tratar uma requisição, o processo deve inspecionar o parâmetro *Connection*. Caso seu valor seja *Close*, o processo deve atender a requisição e encerrar a conexão imediatamente (chamada de sistema *close*). Caso contrário, o processo deve atender a requisição e mantê-la aberta por um intervalo de tempo (vamos ignorar por enquanto o campo *Keep-alive*, se ele existir na requisição). O problema que surge é: como e onde controlar este tempo?

Uma resposta seria controlar este tempo na chamada *read*. Mas como *read* é bloqueante e não possui um parâmetro de *timeout*, a única forma de evitar o bloqueio é controlar o *timeout* através de outra chamada de sistema e, para isso, temos a chamada ***select***. A ideia é simples: ao invés de bloquear um processo indefinidamente em uma chamada *read*, bloqueia-se o mesmo por um período desejado usando uma chamada *select*. Esta chamada permite ao programador verificar se há dados para ler em um descritor de arquivo de leitura (ou socket) ou se um descritor (ou socket) está pronto para escrita. Se uma destas condições ocorrer a chamada retorna dizendo quantos estão prontos. Mas se não houver nenhum pronto, ela só fica bloqueada esperando por um tempo que o programador define (e não indefinidamente como faz a chamada *read*). Ao final deste tempo ela retorna com um valor 0 (que indica *timeout*) e o programa pode considerar que os descritores de leitura e escrita não ficaram prontos no prazo e tomar suas providências. A chamada *select* permite passar até três listas de descritores de arquivo (que podem ser sockets) e um valor de *timeout*. As listas são descritores monitorados para fins de leitura, escrita e

condições especiais. Funções utilitárias (iniciando por FD_) permitem manipular estas listas. A chamada retorna:

- o número de descritores de arquivo fornecidos à chamada que estejam prontos para leitura ou escrita, ou apresentem condições especiais;
- 0, em caso de *timeout*;
- -1, em caso de erro.

Veja exemplo de uso da chamada *select* no programa `teste_select.c` abaixo. Consulte a página de manual referente à chamada *select* para detalhes sobre sua utilização. Este programa está disponível na página da disciplina.

```
/* Programa teste_select.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
  int n, c;
  long int tolerancia = 5;
  fd_set fds;
  struct timeval timeout;
  int fd;

  fd = 0;              /* O que faz esta linha e as demais? Comente cada uma. */
  FD_ZERO(&fds);       /*  */
  FD_SET(fd, &fds);    /*  */
  timeout.tv_sec = tolerancia;  /*  */
  timeout.tv_usec = 0; /*  */

  n = select(1, &fds, (fd_set *)0, (fd_set *)0, &timeout);
  /* O que faz a linha acima? */

  if(n > 0 && FD_ISSET(fd, &fds))   /*  */
    {
      c = getchar();  /* O que acontecerá de diferente aqui? */
      printf("Caractere %c teclado.\n", c);
    }
  else if(n == 0)    /*  */
        {
        printf("Nada foi teclado em %ld s.\n", tolerancia);
        }
       else perror("Erro em select():");   /*  */
  exit(1);   /*  */
}
```

2.2. O problema das interrupções em chamadas de sistema

Em alguns sistemas UNIX o envio de um sinal a um processo bloqueado em uma chamada de sistema faz com que esta chamada de sistema seja interrompida e retorne antes do gerenciador de sinais ser invocado. Neste caso a chamada retorna um valor negativo (-1), que significa erro, e o código de erro (*errno*) é EINTR (*chamada a função interrompida*). Por exemplo, um processo bloqueado em uma chamada *accept* que recebe um sinal de término de processo-filho pode apresentar este comportamento, isto é, retornar com erro antes da hora. Para evitar que isso comprometa a lógica do programa que manipula sinais, podemos tentar "proteger" as chamadas de sistema, chamando-as novamente caso constatemos que elas retornaram com erro e o código do erro é EINTR.

3.Atividades para realização durante a aula

As questões a seguir devem ser respondidas em um relatório em arquivo PDF ou diretamente no editor do Moodle.

1. (1 pt) Execute o programa `teste_select.c` , entenda seu funcionamento e o explique detalhadamente, não deixando de lado as linhas que estão com comentários vazios. Observe que só um comentário simples não é suficiente para explicar o que uma determinada linha está

fazendo e como está impactando o programa. Explique bem como entendeu o funcionamento de cada uma.

2. (1 pt) Explique por que podemos ou não eliminar a variável `tolerancia` , setar a estrutura `timeout.tv_sec` com o valor desejado e usar essa estrutura diretamente na linha com printf do programa `teste_select.c`. Faça e documente um teste que corrobore seus argumentos.

3. (1 pt) Reescreva a parte do código de seu servidor web de maneira a incluir a chamada de sistema *select* adotando um tempo de timeout igual a 3 + (último dígito de seu RA) segundos. Mostre como ficou essa parte do código (entregue só essa parte) e documente um teste (com telnet, por exemplo) que mostre o impacto da inclusão de *select*. Para facilitar a documentação, inclua chamadas a *printf* que indiquem o que o programa fez ou está fazendo em um determinado instante.

4. Atividades para entrega na próxima semana

1. (4 pts) Implemente uma nova versão de seu servidor que contemple todos os requisitos de multiprogramação que foram discutidos neste roteiro, incluindo a proteção contra erro do tipo EINTR e o uso da nova chamada *poll* (em substituição a *select*) para definir prazo de espera em conexões abertas. Atente para as pequenas diferenças entre *poll* (mais moderna) e *select*. Apresente, em um arquivo zip separado, os arquivos com os códigos-fonte do seu servidor devidamente documentados, especialmente nas partes novas que foram introduzidas nesta atividade. Entre os arquivos deverá haver um chamado README.txt, onde você deverá escrever as instruções completas de compilação do pacote, incluindo aquelas relativas aos analisadores léxico e sintático, para que possamos fazer testes independentes com o código. Inclua também os arquivos que compõem o webspace a ser usado para testes.

2. (3 pts) Faça diversos testes com esta nova versão do servidor e documente-os em seu relatório. Tais testes deverão contemplar pelo menos os seguintes casos: envio e recebimento de respostas de várias requisições simultâneas dentro do limite que o servidor suporta (N) e envio de requisições e recebimento de respostas acima da capacidade de tratamento do servidor (N). Dicas: você pode escolher um valor baixo de N (mas N>1) para facilitar os testes de superação do limite e você pode fazer cada processo-filho ficar suspenso por alguns segundos, via chamada *sleep*, de maneira a lhe dar tempo de submeter outras requisições “simultâneas”. Você pode também abrir vários shells em vários terminais para facilitar a submissão de requisições quase simultâneas.

