EA872 Laboratório de Programação de Software Básico
Atividade 11

Vinícius Esperança Mantovani **RA 247395**
Entrega (limite): 08/11/2023, 13hs.

Exercício 1)

Inicialmente, foi criada uma função, chamada “initThread” que passou a conter todo o processo de resposta a requisições, anteriormente contido na “main”. Agora, entre as modificações internas a este processo, temos o seguinte:

```
   char requisition[10];
   char resource[50];
   char connection[50];
```

Foram criadas as três strings acima para armazenar os dados vindos do parser, que serão modificados por outras threads.

Além disso, foi colocada parte do processo dentro de uma região crítica tratada com um mutex, conforme se segue:

```
pthread_mutex_lock(&mutex_parse);
   yy_scan_string(buf_entrada);
   yyparse();
   strcpy(requisition, reqs);
   strcpy(resource, ress);
   strcpy(connection, conn);
   memset(ress, 0, sizeof(ress));
   memset(reqs, 0, sizeof(reqs));
   memset(conn, 0, sizeof(conn));
   pthread_mutex_unlock(&mutex_parse);
```

Temos ainda, também, um mutex que controla a contagem de threads criadas, para não permitir a criação de threads excessivas. Conforme:

```
   pthread_mutex_lock(&mutex_contagem);
   cont_threads--;
```

```
   pthread_mutex_unlock(&mutex_contagem);
```

Vale destacar que algumas variáveis do programa foram transformadas em variáveis globais:

```
extern char reqs[10];
extern char ress[50];
extern char conn[50];
int cont_threads = 0; //conta o numero de subprocessos criados para
atender requisicoes
int status; //usado na chamada wait
char webspace[100];
```

E, foram criadas novas variáveis (os mutex), também globais:

```
pthread_mutex_t mutex_parse = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_contagem = PTHREAD_MUTEX_INITIALIZER;
```

Foi criada uma estrutura para ser passada como parâmetro na chamada pthread_create, conforme o que se passa:

```
struct paramsThread{
   int soquete_mensagem;
};
```

Ela funciona como uma maneira de passar o soquete para dentro da função de inicialização da thread. Isso é feito após a instanciação da estrutura:

```
struct paramsThread *soq_struct = malloc(sizeof(struct paramsThread));
//struct usada para passar como parametro na chamada create
```

Então, foram feitas, ainda, alterações na main, conforme se segue:

```
pthread_mutex_lock(&mutex_contagem);
        //analisa se ha disponibilidade de subprocessos para atender a
uma requisição
       if(cont_threads >= MAX_THREADS){
           pthread_mutex_unlock(&mutex_contagem);
           //printf("maximo de processos, tente novamente!\n");
           // printf("numero de threads no if: %d\n", cont_threads);
           read(soquete_msg, buf_entrada, TAMANHO_BUF);
              printaEXC(argv[1], "/excesso.html", "keep-alive", fd_reg,
soquete_msg);
           free(soq_struct);
           close(soquete_msg);

       } else{
           //printf("entra no else\n");
```

```
            cont_threads++;
            pthread_mutex_unlock(&mutex_contagem);
```

Aqui, percebe-se que o bloco “if-else” foi colocado dentro de uma proteção por mutex, para que não houvesse problema com a contagem de threads.

Ainda, foi colocada a chamada “pthread_create” dentro do bloco else.

```
            } else{

//printf("entra no else\n");

            cont_threads++;
            pthread_mutex_unlock(&mutex_contagem);

            // printf("numero de threads no else: %d\n", cont_threads);

            //write(1, "\nchega aqui\n", 12);
            if((socks.revents & POLLIN))
            {

                int ret = pthread_create(&thread_id[cont_threads], NULL,
initThread, (void *)soq_struct);
                // sleep(1);
                if(ret){
                    printf("ERRO NO PTHREAD_CREATE");
                    exit(-1);
                }

            }
```

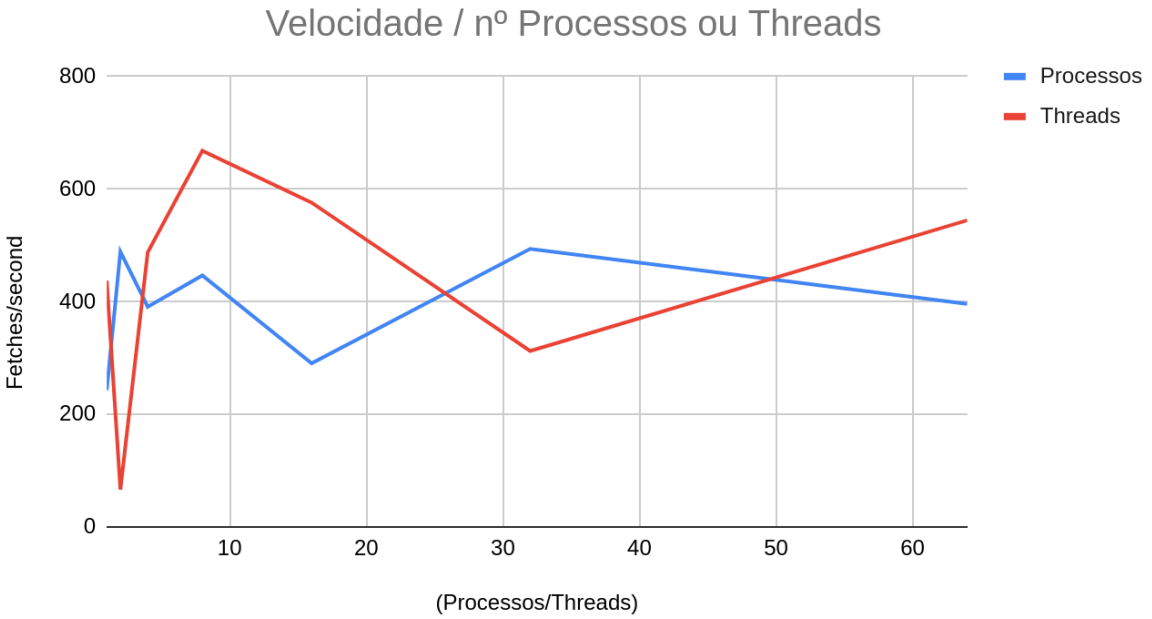
Exercício 2)

Atenção: o uso de -fetches foi substituído pelo uso de -seconds, por conta da oscilação muito grande no tempo de execução, chegando a demorar tempos muito altos!

Vale destacar, de início, que o tempo usado para análise foi de 4 segundos e, foram repetidos os testes 5 vezes.

a) A seguir, é apresentada a tabela e o gráfico com os valores de fetches/second em função do número de processos:

processes	threads	
fetches/second	fetches/second	processes/threads
243,000	437,5	1
489,000	66,44980	2
391,000	487,747	4
446,750	668,25	8
290,500	575,998	16
493,750	312,50	32
396,250	544,75	64



b) Neste item, foi feita uma análise da versão inicial do servidor, para a qual foram obtidos resultados muito próximos para todas as quantias de processos em paralelo. Isso porque, o programa tem três chamadas "sleep", o que acaba sendo maior que as variações possíveis entre números de requisições respondidas por segundo para números diferentes de paralelos.

fetches/second	nº paralelo
0,249	1
0,249005	2
0,24912	4
0,24993	8
0,249875	16
0,249876	32
0,249906	64

c) Analisando inicialmente os resultados para o teste dos programas multi-thread e multi-processos, nota-se que, em média, o servidor multi-thread foi mais rápido na resposta de requisições quando comparado ao servidor multi-processos. Com exceção dos casos com duas e com trinta e duas threads/processos em que o multi-processos foi mais eficiente, nos demais o servidor multi-threads foi melhor em velocidade. Desse modo, como não houveram mudanças ruins na mudança do código multi-processos para multi-threads, temos, conforme o esperado, um desempenho melhor no caso do servidor multi-threads. Isso ocorre, por conta de a transição entre duas threads, uma em execução e uma aguardando para executar (aguardando pelo tempo na CPU), é menos custosa em tempo que a transição entre dois processos nessa mesma condição. Isso porque, um processo contém um espaço de endereçamento virtual, threads, descritores de arquivos, semáforos e gerenciadores de sinal, enquanto que uma thread contém menos informações, sendo elas: uma pilha, valores de registradores e sua prioridade própria. Dessa forma, a transição de processos exige uma transição de um número maior de informações que o número envolvido na transição de threads e, por consequência, esse número maior leva mais tempo para ser transitado, enquanto que o número menor relacionado às threads é transitado em menos tempo. Assim se impõe uma velocidade maior ao multi-thread.

Analisando, agora, os resultados dos testes relacionados ao servidor mono-processo, nota-se que o número de respostas por segundo é extremamente pequeno (aproximadamente 0,25), muito menor que o dos demais servidores e, os valores obtidos para os testes diferentes não variaram significativamente. Isso ocorreu, porque o tempo de “sleep” de alguns trechos do código do servidor é extremamente maior que as variações causadas pelo número de requisições feitas em paralelo. Deste modo, praticamente não oscila o tempo, implicando que também não oscila praticamente o número de fetches/second do programa, independentemente do número de requisições paralelas.

d) Tratando do caso de mono-processo, uma melhoria possível seria reduzir o tempo necessário de chamadas “sleep” do programa. Isso porque, elas atrasam a resposta do servidor ao cliente. Ademais, no caso dos servidores multi-threads e multi-processos, uma melhoria possível seria alterar a lógica do parser, de maneira a permitir que eles trabalhassem apenas com variáveis locais, permitindo que o “yyparse” não fosse uma região crítica e, portanto, reduzindo o tempo de espera de threads pelo tempo de execução na CPU.

e) Fez-se a melhoria no código do servidor mono-processo, retirando todas as chamadas “sleep”, por meio de uma chamada “poll” como a feita nos demais servidores. O resultado obtido foi o seguinte:

fetches/second	nº paralelo
134,9990	1
101,5000	2
105,0000	4
88,9978	8
77,4999	16
103,2500	32
316,2800	64

Assim, pode-se notar uma pequena variação no número de requisições respondidas por segundo entre os casos com 1, 2 e 4 processos em paralelo. Então, em sequência, nota-se, para 8 e 16 requisições uma queda, que é, logo, recuperada para uma quantia de 32 requisições em paralelo, sendo que, para 64 temos um número muito grande de requisições respondidas por segundo.