Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencia de la Computación
R-322 - Sistemas Operativos 1

Nombre y Apellido: **Legajo:**

1) Se debe diseñar la lógica de un servidor TCP que gestiona R recursos globales usando un pool de 4 hilos con las siguientes características.

- Arquitectura: No hay un hilo dedicado a accept(). Todos comparten un único descriptor epoll_fd.
- Sincronización A: Las modificaciones a los recursos deben ser atómicas.
- Sincronización B: Si un cliente solicita más recursos de los disponibles, el hilo debe bloquearse (mediante variables de condición) hasta que haya suficientes.
- Desconexiones: Si un cliente se desconecta sin devolver recursos estos se pierden (el servidor solo detecta el EOF y limpia el socket).
- Eventos: Usar EPOLLONESHOT.
- Signals: El sistema no puede caerse al recibir un SIGPIPE.

El servidor acepta los siguientes comandos.

Comando	**Acción**	**Respuesta esperada**
TAKE N	Pide N recursos	OK <libres> (Espera si no hay suficientes)
GIVE N	Devuelve N recursos	OK <libres>
STATUS	Consulta estado	OK <libres> <usados>
QUIT	Cierra conexión	BYE (y se cierra el socket)

Se puede asumir disponible y ya implementadas las siguientes funciones:

- int setup_server_socket(int port); (Retorna socket de escucha configurado)
- void set_nonblocking(int fd);
- int parse_client_request(int fd, int *action, int *amount); (Retorna 1 si hay comando, 0 si EAGAIN, -1 si error/desconexión. Action devuelve CMD_TAKE, CMD_GIVE, etc.)
- void send_response(int fd, const char *format, ...);

Al final del código responder:

a) ¿Qué problema ocurriría si no se usara EPOLLONESHOT?

b) Variables de condición: En su función que implementa el comando GIVE, ¿utilizó pthread_cond_signal o pthread_cond_broadcast? Justifique su decisión.

c) Deadlocks: Según las 4 condiciones de Coffman ¿Existe riesgo de interbloqueo en su función que implementa el comando TAKE combinada con el bucle epoll?

2) Implemente un cliente para el servidor del ejercicio 1 utilizando fork() y pipe().

a) El proceso pare debe: conectarse al servidor vía TCP, lee comandos desde el pipe enviados por el hijo, enviar los comandos al servidor a través del socket, leer las respuestas del servidor y mostrarlas en stdout.

b) El proceso hijo debe: leer comandos desde stdin, enviarlos al padre a través del pipe, terminar cuando se ingresa QUIT.

Se puede asumir:

- int connect_to_server(const char *ip, int port);

3) Implemente un cliente en Erlang que se conecte al servidor del ejercicio 1. El cliente debe:

a) Conectarse al sevidor TCP en la dirección y puerto especificados.

b) Permitir enviar comandos al servidor.

c) Recibir y mostrar las respuestas del servidor.

d) Manejar correctamente la desconexión.

Usar gen_tcp:connect/3 para la conexión, gen_tcp:send/2 para enviar comandos, gen_tcp:recv/2 para recibir respuestas.

Principales funciones a implementar

a) start(ServerIP, Port) - Inicia el cliente y se conecta al servidor.

b) send_command(Command) - Envía un command al servidor.

c) loop(Socket) - Bucle principal que recibe y procesa respuestas.

d) stop() - Cierra la conexión correctamente.

Comportamiento esperado, izquierda cliente C, derecha cliente Erlang

$./cliente 127.0.0.1 8080	1> cliente:start("127.0.0.1", 8080)
Cliente iniciado	Conectado al servidor
TAKE 3	2> cliente:send_command("TAKE 3")
OK 7	OK 7
STATUS	3> cliente:send_command("STATUS")
OK 7 3	OK 7 3
GIVE 1	4> cliente:send_command("GIVE 1")
OK 8	OK 8
QUIT	4> cliente:stop()
BYE	BYE
Conexión cerrada	Conexión cerrada