HAW Hamburg - IE6

Operating Systems Lab 3

Date	07.06.2023
Group2	Thishan Warnakulasooriya Celestine Machuca
Supervisor	Prof. Dr.-Ing. Holger Gräßner

Execution of a TCP server via internet super daemon

In this task we create a C program that saves the received text data via a network, to a logfile in the beagle board. We integrated our C program into the server program that was created utilizing the internet super daemon service.

inetd (internet service daemon)[1] is a super-server daemon on many Unix systems that provides Internet services. For each configured service, it listens for requests from connecting clients. Requests are served by spawning a process that runs the appropriate executable, but simple services such as echo are served by inetd itself. External executables, which are run on request, can be single- or multi-threaded.

Below we have documented the sub-steps we took for the configuration and the execution of this task.

Task a

- A C program that reads data from stdin and then saves it into a log file in the beagle board(figure 1).

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
  const char * filename = argv[1];
  FILE *fp = fopen(filename, "a");
  if (fp == NULL) {
    printf("Cannot open file %s\n", filename);
    exit(1);
  }
  char str[4096];
  while (fgets(str, sizeof str, stdin)) {
    fputs(str, fp);
    fflush(fp);
  }
  fflush(fp);
  fclose(fp);
  return 0;
}
```

Figure 1 : C Program for saving received data into logfile

[1] inetd (https://en.wikipedia.org/wiki/Inetd, accessed on 11.06.23)

- Then we added our service into the inetd.conf file in the beagle board and configured our program's reading and writing paths.

```
#inetd.conf
# Packages should modify this file by using update-inetd(8)
#
# <service_name> <sock_type> <proto> <flags> <user> <server_path> <args>
#
#:INTERNAL: Internal services
#discard            stream  tcp     nowait  root    internal
#discard            dgram   udp     wait    root    internal
#daytime            stream  tcp     nowait  root    internal
#time        stream  tcp     nowait  root    internal

#:STANDARD: These are standard services.
telnet      stream  tcp    nowait  root    /usr/sbin/tcpd  /usr/sbin/in.telnetd
ftp         stream  tcp    nowait  root    /usr/sbin/tcpd  /usr/sbin/in.ftpd
celeste-net    stream  tcp    nowait  root    /home/e6bs/ThishanCeleste/lab3task lab3task
/home/e6bs/ThishanCeleste/output.txt
```

Figure 2 : Modified inetd.conf

- we then modified the /etc/services file to add the defined service as follows

```
#services
# Local services

celeste-net      9999/tcp
```

Figure 3 : Modified services file

- In order to now reboot the configured minted.conf, we could either restart the Beagle board or send a kill command to reboot only the inetd service.
- We found the pid of the inetd service and then sent the following command to reboot the service.

```
Sudo kill -HUP 2243
```

- Then we used both telnet service and a python script to communicate and send messages to the beagle board using the lab computer.

test.py

```python
import socket
server_ip = '141.22.14.100'
server_port = 9999
def sendStringToServer(ip, port, string):
        print("trying to send bs")
        client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        client_socket.connect((ip, port))
        client_socket.sendall(string.encode())
        client_socket.close()
        print("i think it worked :s")
sendStringToServer(server_ip, server_port, 'Hello Wofghjkjhgrld\n')
```

Figure 4 : Python script that send out message to a port

```
group3@bslabmaster-OptiPlex-3050:~$ telnet 141.22.14.100 9999
Trying 141.22.14.100...
Connected to 141.22.14.100.
Escape character is '^]'.
Hello My name is Pierre Thishan.
How are you doing ?
Ok bye bye
^]
telnet> quit
Connection closed.
group3@bslabmaster-OptiPlex-3050:~$ 
```

Figure 5 : Terminal output when using telnet to send messages

- Thereafter we checked the logfile under the configured path in the beagle board to verify that the messages we sent via network has been successfully transferred and written.

```
e6bs@beagle8: ~/ThishanCeleste          group3@bslabmaster-OptiPlex-3050: ~
  GNU nano 2.2.6                                          File: output.txt

Hello World
Hello World
Hello Wofghjkjhgrld
Hello Wofghjkjhgrld
Hello there!!
Hello there!!
Hello there!!
Hello there!!
Hello Wofghjkjhgrld
Hello Wojefkefjkefkjefkjekfjkef celeste hgrld
gkjrjgkrkgrkgjkrgjkjgkrkgjkrgkrjg
^X
^[9
efdjlkfjefkdddkdkdkd
dkkdkdk
ldldlld
ldldlld
ldldldldl
Hello My name is Pierre Thishan.
How are you doing ?
Ok bye bye
```

Figure 6 : Logfile saved in beagle board

Task b

Here we used the **ps** and **netstat** system commands to observe the beagle board's operating systems' reaction during the external access.

- Using **ps -aux | grep inetd** we found the pid of inetd service.

```
e6bs      2841  0.0  0.2   4436  1052 pts/0    R+   03:30   0:00 ps -aux
e6bs@beagle8:~/ThishanCeleste$ ps -aux | grep inetd
warning: bad ps syntax, perhaps a bogus '-'?
See http://gitorious.org/procps/procps/blobs/master/Documentation/FAQ
root      2343  0.0  0.1   1952   652 ?        Ss   01:01   0:00 /usr/sbin/inetd
e6bs      2844  0.0  0.1   3524   716 pts/0    S+   03:31   0:00 grep inetd
e6bs@beagle8:~/ThishanCeleste$
```

Figure 7 : Terminal output for ps command

- When the **netstat** command was run, we could see that the beagle board has an active connection with the lab computer through tcp.

```
e6bs@beagle8:~/ThishanCeleste$ netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 beagle8.etech.haw-h:ssh 141.22.14.15:57370      ESTABLISHED
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node   Path
unix  7      [ ]         DGRAM                    7461     /dev/log
unix  3      [ ]         STREAM     CONNECTED     7931     /var/run/dbus/system_bus_socket
unix  3      [ ]         STREAM     CONNECTED     7921     /var/run/dbus/system_bus_socket
unix  3      [ ]         STREAM     CONNECTED     7652
unix  3      [ ]         STREAM     CONNECTED     7645
unix  3      [ ]         STREAM     CONNECTED     7641
unix  3      [ ]         STREAM     CONNECTED     7728     @/tmp/.X11-unix/X0
unix  3      [ ]         STREAM     CONNECTED     7884     /var/run/dbus/system_bus_socket
unix  3      [ ]         STREAM     CONNECTED     7897
unix  3      [ ]         STREAM     CONNECTED     7930
unix  3      [ ]         STREAM     CONNECTED     7920
unix  2      [ ]         DGRAM                    8044
unix  3      [ ]         STREAM     CONNECTED     7648     @/tmp/.X11-unix/X0
unix  3      [ ]         STREAM     CONNECTED     7653
unix  3      [ ]         STREAM     CONNECTED     7642
unix  2      [ ]         DGRAM                    7882
unix  2      [ ]         DGRAM                    7886
unix  3      [ ]         STREAM     CONNECTED     7898     /var/run/dbus/system_bus_socket
unix  3      [ ]         STREAM     CONNECTED     7508
unix  3      [ ]         STREAM     CONNECTED     8007
unix  2      [ ]         DGRAM                    7664
unix  3      [ ]         STREAM     CONNECTED     7643
unix  2      [ ]         DGRAM                    7795
unix  3      [ ]         STREAM     CONNECTED     7259
unix  3      [ ]         STREAM     CONNECTED     7258
unix  3      [ ]         STREAM     CONNECTED     8008
unix  3      [ ]         STREAM     CONNECTED     7727
unix  3      [ ]         STREAM     CONNECTED     7644
unix  3      [ ]         STREAM     CONNECTED     7883
unix  3      [ ]         DGRAM                    4886
unix  3      [ ]         STREAM     CONNECTED     7509
unix  3      [ ]         DGRAM                    4887
```

Figure 8 : Terminal output of netstat command

Task c

Here we used the network analyzing tool **wireshark** to observe the establishing phase, receiving data, and the disconnection phase of the server we implemented in the beagle board.

- First we had to establish a connection with the beagle board again with graphical interfaces enabled. Then we opened the GUI of wireshark application to analyze the network of the beagle board for the following connection calls from the lab computer.

Figure 9 : Terminal output of sent messages using telnet service

- In the above figure we first establish a connection to the server through the port we specified before (9999), and then after sending a few text strings, we close the connection with the server.
- Below we have the screen dump taken throughout this process.

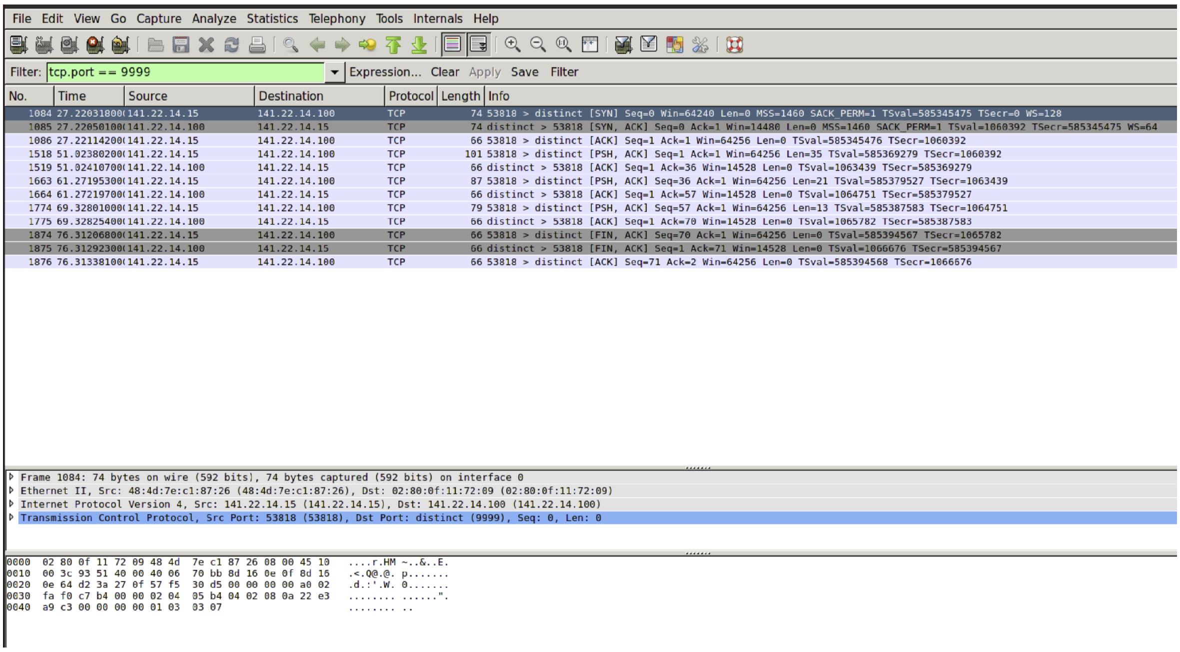

Figure 10 : Screenshot of the wirshark console

Message queues

- Design two processes transferring data uni-directionally by a message queue.

A program was made to make use of the default behavior msgrcv() without the parameter IPC_NOWAIT by using the default blocking behavior that waits until a message has arrived to the queue

```
// implement message queue
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
//include fork and wait
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#define MAX_TEXT 100

struct msg_buffer
{
    long message_type;
    char message_text[MAX_TEXT];
} message;

void writeTestProccess(int msgid, struct msg_buffer *message, char *text)
{
    message->message_type = 1;
    strcpy(message->message_text, text);
    msgsnd(msgid, message, strlen(message->message_text) + 1, 0);
}

#define __USE_BLOCKING

void readTestProccess(int msgid, struct msg_buffer *message)
{
    #ifdef __USE_BLOCKING
        //blocking until message arrive
        msgrcv(msgid, message, MAX_TEXT, 1, 0);
        printf("Data Received is : %s \n", message->message_text);
    #endif
```

```
    #ifdef __USE_NON_BLOCKING
        //non blocking
        if (msgrcv(msgid, message, MAX_TEXT, 1, IPC_NOWAIT) == -1)
        {
            printf("No data\n");
        }
        else
        {
            printf("Data Received is : %s \n", message->message_text);
        }
    #endif
}

int main(int argc, char **argv)
{
    key_t key = ftok("/some/path", 65);        // create unique key
    //create message queue
    int msgid = msgget(key, 0666 | IPC_CREAT | IPC_EXCL); // create message
queue and return id
    if (msgid == -1)
    {
        printf("Message queue already exist\n");
        msgid = msgget(key, 0666);
    }
    else
    {
        printf("Message queue created\n");
    }
    //create child process
    pid_t pid = fork();
    if (pid == 0)
    {
        //child process
        //read all message from message queue
        for(int i = 0 ; i < 10 ; i++)
        {
            readTestProccess(msgid, &message);
        }
    }
    else
    {
        //parent process
        //
        sleep(5);
```

```
        int times = atoi(argv[1]);
        printf("times: %d\n", times);
        for (int i = 0; i < times; i++)
        {
            printf("writing message to message queue %d\n", i);
            char msg[MAX_TEXT];
            sprintf(msg, "Hello from parent process %d", i);
            writeTestProccess(msgid, &message, msg);
        }
        printf("Data has been sent\n");
    }
}
```

Figure 11 : C program

- Below is the console output when we ran the program in a personal computer.

```
celeste@debian-vm:~/lab-3$ ./task2 10
Message queue already exist
times: 10
writing message to message queue 0
writing message to message queue 1
writing message to message queue 2
writing message to message queue 3
writing message to message queue 4
writing message to message queue 5
writing message to message queue 6
writing message to message queue 7
writing message to message queue 8
writing message to message queue 9
Data has been sent
Data Received is : Hello from parent process 0
Data Received is : Hello from parent process 7
Data Received is : Hello from parent process 8
Data Received is : Hello from parent process 9
```

Figure 13 : Console output

- Check the blocking features by doing parameter variation.

The __USE_NON_BLOCKING flag was set thereby using the non-blocking behavior. As it can be seen from the output the behavior is just repeated 10 times with no detection of messages as expected

```
void readTestProccess(int msgid, struct msg_buffer *message)
{
```

```c
    #ifdef __USE_BLOCKING
        //blocking until message arrive
        msgrcv(msgid, message, MAX_TEXT, 1, 0);
        printf("Data Received is : %s \n", message->message_text);
    #endif

    #ifdef __USE_NON_BLOCKING
        //non blocking
        if (msgrcv(msgid, message, MAX_TEXT, 1, IPC_NOWAIT) == -1)
        {
            printf("No data\n");
        }
        else
        {
            printf("Data Received is : %s \n", message->message_text);
        }
    #endif
}
```

Figure 14 : C program

```
celeste@debian-vm:~/lab-3$ ./task2 10
Message queue already exist
No data
No data
No data
No data
No data
No data
No data
No data
No data
No data
times: 10
writing message to message queue 0
writing message to message queue 1
writing message to message queue 2
writing message to message queue 3
writing message to message queue 4
writing message to message queue 5
writing message to message queue 6
writing message to message queue 7
writing message to message queue 8
writing message to message queue 9
```

Figure 15 : Console output

Shared memory

- Design two processes transferring data unidirectionally by shared memory.

//task3.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <pthread.h>
#include <time.h>

#define SHM_NAME "/shm_example"
#define MUTEX_NAME "/mutex_example"

typedef struct {
    pthread_mutex_t mutex;
    size_t block_size;
    char data[];
} shared_data;

int main() {
    int shm_fd;
    shared_data *sh_data;
    size_t block_sizes[] = { 1024, 2048, 4096, 8192 , 16384, 32768,
65536, 131072, 262144, 524288 };
    size_t num_sizes = sizeof(block_sizes) / sizeof(size_t);
    char *temp_buf;
    struct timespec start, end;
    long long elapsed_ns;

    for (size_t i = 0; i < num_sizes; ++i) {
        size_t size = block_sizes[i];

        /* Create shared memory object and set its size */
        shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
```

```
        if (shm_fd == -1) {
            perror("Shared memory");
            return -1;
        }

        if (ftruncate(shm_fd, sizeof(shared_data) + size) != 0) {
            perror("Shared memory size");
            return -1;
        }

        /* Map shared memory object */
        sh_data = (shared_data *)mmap(NULL, sizeof(shared_data) +
size, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
        if (sh_data == MAP_FAILED) {
            perror("Memory mapping");
            return -1;
        }

        /* Initialize mutex */
        pthread_mutexattr_t attrmutex;
        pthread_mutexattr_init(&attrmutex);
        pthread_mutexattr_setpshared(&attrmutex,
PTHREAD_PROCESS_SHARED);
        pthread_mutex_init(&sh_data->mutex, &attrmutex);

        /* Allocate temp buffer */
        temp_buf = malloc(size);
        memset(temp_buf, 0, size);

        pid_t pid = fork();

        if (pid < 0) {
            perror("Fork");
            return -1;
        } else if (pid == 0) { /* Child process */
            clock_gettime(CLOCK_REALTIME, &start);
            pthread_mutex_lock(&sh_data->mutex);
            memcpy(sh_data->data, temp_buf, size);
            pthread_mutex_unlock(&sh_data->mutex);
            clock_gettime(CLOCK_REALTIME, &end);
            elapsed_ns = (end.tv_sec - start.tv_sec) * 1000000000LL +
(end.tv_nsec - start.tv_nsec);
```

```
            printf("Child wrote %zu bytes in %lld ns\n", size,
elapsed_ns);
            exit(0);
        } else { /* Parent process */
            wait(NULL); /* Wait for child to finish writing */
            clock_gettime(CLOCK_REALTIME, &start);
            pthread_mutex_lock(&sh_data->mutex);
            memcpy(temp_buf, sh_data->data, size);
            pthread_mutex_unlock(&sh_data->mutex);
            clock_gettime(CLOCK_REALTIME, &end);
            elapsed_ns = (end.tv_sec - start.tv_sec) * 1000000000LL +
(end.tv_nsec - start.tv_nsec);
            printf("Parent read %zu bytes in %lld ns\n", size,
elapsed_ns);
        }

        /* Clean up */
        munmap(sh_data, sizeof(shared_data) + size);

        if (shm_unlink(SHM_NAME) != 0) {
            perror("Unlink shared memory");
            return -1;
        }

        free(temp_buf);
    }


    return 0;
}
```

Figure 16 :

- We then compiled the program and executed it

```
gcc task3.c -o task3 -lrt -lpthread
./task3
```

- We then checked the output and it was as expected

```
celeste@debian-vm:~/lab-3$ ./task3
Child wrote 1024 bytes in 4799 ns
Parent read 1024 bytes in 2455 ns
Child wrote 2048 bytes in 7013 ns
Parent read 2048 bytes in 3266 ns
Child wrote 4096 bytes in 13345 ns
Parent read 4096 bytes in 3897 ns
Child wrote 8192 bytes in 24186 ns
Parent read 8192 bytes in 6662 ns
Child wrote 16384 bytes in 34715 ns
Parent read 16384 bytes in 8005 ns
Child wrote 32768 bytes in 38933 ns
Parent read 32768 bytes in 7343 ns
Child wrote 65536 bytes in 62698 ns
Parent read 65536 bytes in 23154 ns
Child wrote 131072 bytes in 110277 ns
Parent read 131072 bytes in 35828 ns
Child wrote 262144 bytes in 154079 ns
Parent read 262144 bytes in 53821 ns
Child wrote 524288 bytes in 335860 ns
Parent read 524288 bytes in 137538 ns
```

Figure 17 : Console output

- Synchronise the data exchange by use of an appropriate handshake.

For the effects of synchronization, as we had in effect two processes writing to the same memory block, synchronization was required. We implemented it by using a mutex.

```
/* Initialize mutex */
pthread_mutexattr_t attrmutex;
pthread_mutexattr_init(&attrmutex);
pthread_mutexattr_setpshared(&attrmutex, PTHREAD_PROCESS_SHARED);
pthread_mutex_init(&sh_data->mutex, &attrmutex);
```

```
//enter critical section
pthread_mutex_lock(&sh_data->mutex);
memcpy(sh_data->data, temp_buf, size);
pthread_mutex_unlock(&sh_data->mutex);
//exit critical section
```

- Make the program do a measurement of data transfer rates at different data block sizes.

  The program was iterated throughout the following block sizes = `{ 1024, 2048, 4096, 8192 , 16384, 32768, 65536, 131072, 262144, 524288 };`

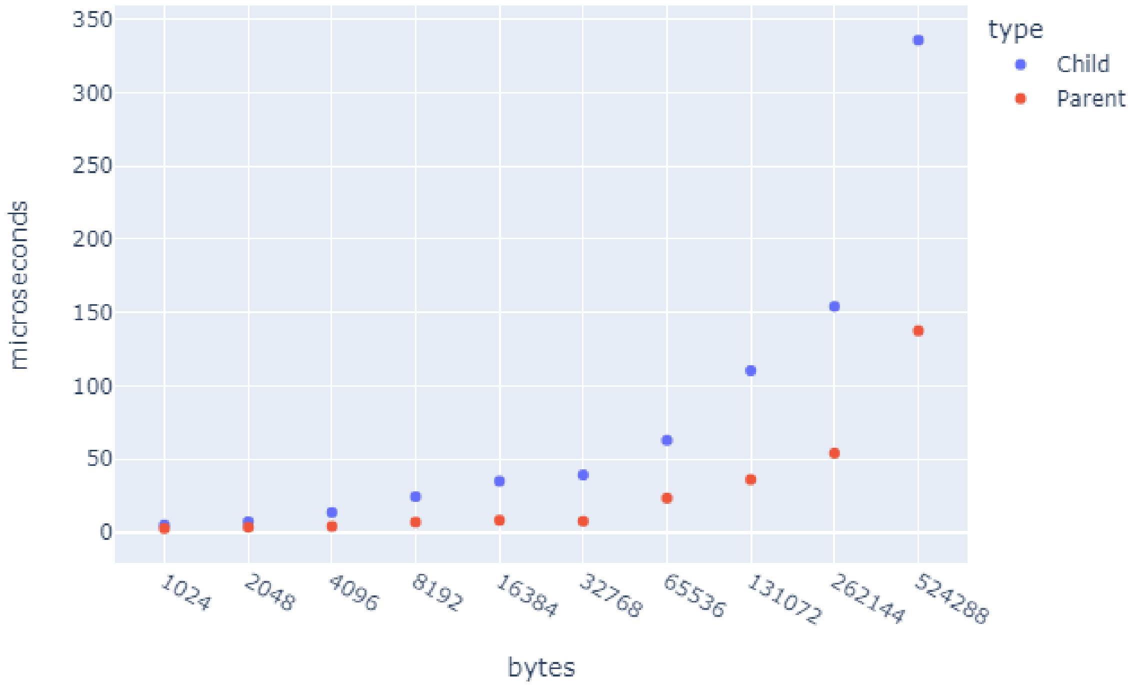

FIgure 18 : Plots of child and parent processes for different block sizes

- Compare and discuss your results!

As the block size increases the time also increases, it is hard to draw conclusions but it can be noted that a higher linear increase.

- Run this measurement on the lab PC as well as on the beagle board.

The results couldn't be compared with the beagle board due to a simple lack of lab time, we apologise for the inconvenience.