Baiscs of Linux and Kernel

Linux

Linux is a Unix-like computer operating system assembled under the model of free and open source software development and distribution

- Linux is Free
- Portable to any Hardware Platform
- Linux is scalable
- Linux is secure
- The Linux OS and most Linux applications have very short debug-times

What's a Kernel?

The kernel is a program that constitutes the central core of a computer operating system. It has complete control over everything that occurs in the system.

- Controls and mediates access to hardware.
- Implements and supports fundamental abstractions:
  - Processes, files, devices etc.
- Schedules / allocates system resources:
  - Memory, CPU, disk, etc.
- Enforces security and protection

Tasks Of Kernel

- Process Management

In charge of creating , destroying and handling process.

Communication between process.

The scheduler,which controls how process share the CPU,is part of process management.

- <u>Memory Management</u>

Kernel builds virtual addressing space for any & all processes on top of the available resources.

Virtual memory makes the system appear to have more memory than it actually has by sharing it between competing processes as they need it.

- File Systems

Kernel builds a structured file system on top of the unstructured hardware & the resulting file abstraction is heavily used throughout the whole system

- Device Control

Every system operation maps to a physical device.All devices control operations are performed by code that is specific to the device being addressed.That code is called a Device Driver.

Kernel must have embedded in it a device driver for every single peripheral present on a system,from hard drive to the keyboard & tape drive

- <u>Networking</u>

Networking mustbe managed by operating system,because most network operations are not specific to a process.

The packets must be collected,identified & dispatched before a process takes care of them.

Loadable Modules

- One of the good features of Linux is the ability to extend at runtime the set of features offered by the kernel. This means that you can add functionality to the kernel(and remove functionality as well) while the system is up and running.
- Each piece of code that can be added to the kernel at runtime is called a **Module.**

Device Drivers

- In computing, a Device driver is a computer program that operates or controls a particular type of device that is attached to a computer.
- Application calls the Driver's functions,and the driver functions performs operations on the device and gives results back to the application.
- Linux Device Drivers are broadly divided in 3 categories.
  - Character Device Drivers
  - Block/Storage Device Drivers
  - Network Device Drivers

- Character Device Drivers

Relate to devices through which the system transmits & receives data character by character

Do not support random access

Unbuffered i/o routines

Mice,keyboard,serial modems are examples of character device drivers

- <u>Block Device Drivers</u>

Block devices correspond with devices that move data inform of Blocks

Block devices support random access and seeking,and generally use buffered input & output.

The operating system allocates a data buffer to hold a single block each for i/o.

- <u>Network Device Drivers</u>

Network communication devices are controlled using network device drivers

Token ring drivers,ATM drivers,Wi-Fi drivers are example of Network Device Drivers

Hello World

- #include <linux/init.h>
- #include <linux/module.h>
- MODULE_LICENSE("Dual BSD/GPL");
- static int hello_init(void)
- {
- printk(KERN_ALERT "Hello, world\n");
- return 0;
- }
- static void hello_exit(void)
- {
- printk(KERN_ALERT "Goodbye, cruel world\n");
- }
- module_init(hello_init);
- module_exit(hello_exit);

Building & Running Module

- Compiling module

Obj-m +=Hello.o

- Building module

make

- Loading module

insmod Hello.ko

- Unloading Module

rmmod Hello

Kernel Symbol Table

- Names of functions & variables are called symbols.In a compiled program all the symbols used in program & their addresses are kept in a symbol table.Same is with kernel image also.
- Symbols within loadable modules are local only they can be made global using macros

```
EXPORT_SYMBOL(name);
EXPORT_SYMBOL_GPL(name);
```

Module Parameters

- At the time of inserting the Module we can pass arguments to Module.
- Using MACRO module_param(name,type,permission); we can pass arguments to the module.

Concurrency in the Kernel

- Linux systems run multiple processes,more than one of which can be trying to use our driver at the same time.
- Most devices are capable of interrupting the processor.
- Linux can run on symmetric multiprocessor (SMP) systems.
- As a result, Linux kernel code, including driver code, must be reentrant—it must be capable of running in more than one context at the same time.

Concurrency in the Kernel contd...

- Data structures must be carefully designed to keep multiple threads of execution separate, and the code must take care to access shared data in ways that prevent corruption of the data.

The current process

- Linux maintains a the structure of the kind "task_struct" for every process that runs in the system. This structure defined in "linux/sched.h"
- It maintains various data of a process like the process id, name, memory related information etc. While writing a module if we want to get information about the current process that is running in the kernel, we need to read the "task_struct" of the corresponding process.

The current process contd...

- The kernel provides a easy way to do this by providing a macro by the name "current", which always returns a pointer to the "task_struct" of the current executing process.
- Following statement will print PocessID & name

printk(KERN_INFO "The process is \"%s\" (pid %i)\n",current->comm, current->pid);

Thank You

Linux Device Driver..

(2^{nd} Presentation)

Character Device Driver

-->A simple characer device driver which takes input from user side using command line argument from application program and reverses it using character device driver.

-->File operations include open,write,read and close.

<u>Introducing ioctl()</u>

-->Input/output control(ioctl) is a common operation or a system call,available in most driver categoriesIf there is no other system call that meets a particular requirement, then ioctl() is the one to use..

-->If there is no other system call that meets a particular requirement, then ioctl() is the one to use..

-->Practical examples include volume control for an audio device, display configuration for a video device, reading device registers, and so on....

-->In our example we are exchanging value of variables between user space and kernel space using ioctl().

Defining the ioctl() commands

-->The Linux header file /usr/include/asm/ioctl.h defines macros that must be used to create the ioctl command number. These macros take various combinations of three arguments: .

#define GET_VAR _IO(int type, int number)

#define GET_VAR _IOR/_IOW/_IORW(int type, int number, data_type)

-->type--an 8-bit integer selected to be specific to the device driver.

-->number--an 8-bit integer 'command number' 'Within a driver, distinct numbers should be chosen

-->data_type--The name of a type used to compute how many bytes are exchanged between the client and the driver.

-->IO(int type, int number) - used for a simple ioctl that sends nothing but the type and number, and receives back nothing but an (integer) retval.

_IOR(int type, int number, data_type) - used for an ioctl that reads data from the device driver. The driver will be allowed to return sizeof(data_type) bytes to the user.

_IOW(int type, int number, data_type) - similar to _IOR, but used to write data to the driver.

_IORW(int type, int number, data_type) - a combination of _IOR and _IOW. That is, data is both written to the driver and then read back from the driver by the client.

Process

→Running instance of a program is called a process in Linux, every process has a parent process.

→Each process in a Linux system is identified by its unique process ID, sometimes referred to as pid.

→ Process IDs are 16-bit numbers that are assigned sequentially by Linux as new processes are created

→There is a process known as 'init' that is the very first process that Linux kernel creates after system boots up.

→All the process there-on are children of this process either directly or indirectly.

→The only time it terminates is when the Linux system is shut down.

→The init process always has process ID 1 associated with it.

Process vs Threads

1. Threads share the address space of the process that created it; processes have their own address.

- 2.Threads have direct access to the data segment of its process; processes have their own copy of the data segment of the parent process.
- 3.Threads can directly communicate with other threads of its process; processes must use interprocess communication to communicate with sibling processes.
- 4.Threads have almost no overhead; processes have considerable overhead.
- 5.New threads are easily created; new processes require duplication of the parent process.
- 6.Threads can exercise considerable control over threads of the same process; processes can only exercise control over child processes.
- 7.Changes to the main thread (cancellation, priority change, etc.) may affect the behavior of the other threads of the process; changes to the parent process does not affect child processes.

Process

Process creation in Linux

- Process is created using fork( ) & exec() system call
- creates new process by copying old
- both copies then proceed running
- – old copy resumes (after “fork( )”)
- – so does new
- new copy is not functionally different

FORK example

```
#include <stdio.h>
main()
{
printf("How many times do you see this line\n");
fork();
printf("How about this one\n");
}
---------------------------------------------------------------------------------
---
ips@EICPU1200:~/Desktop$ gcc -o fork fork.c
einfochips@EICPU1200:~/Desktop$ ./fork
How many times do you see this line
How about this one
How about this one
```

FORK example

```
#include <stdio.h>
main()
{
printf("How many times do you see this line,process id=%d\n",getpid()");
fork();
printf("How about this one , process id=%d  \n",getpid());
}
-----------------------------------------------------------------------------------
ips@EICPU1200:~/Desktop$ gcc -o fork fork.c
einfochips@EICPU1200:~/Desktop$ ./fork

How many times do you see this line,process id=4936
How about this one , process id=4936
How about this one , process id=4937
```

FORK ... self-identify?

```
#include <stdio.h>
main()
{
printf("How many times do you see this line,process id=%d\n",getpid()");
result=fork();
printf("How about this one, process id=%d--i got--%d\n",getpid(),result);
}
```

```
ips@EICPU1200:~/Desktop$ gcc -o fork fork.c
einfochips@EICPU1200:~/Desktop$ ./fork
How many times do you see this line,process id=4936
How about this one, process id=4936--i got--4937
How about this one, process id=4937--i got--0
```

FORK example……

```
#include <stdio.h>
main()
{
printf("How many times do you see this line,process id=%d\n",getpid()");
result=fork();
if(result==0)
printf("This is child process\n");
else
printf("Parent process still continues.....\n");
}
```

```
ips@EICPU1200:~/Desktop$ gcc -o fork fork.c
einfochips@EICPU1200:~/Desktop$ ./fork
This is parent process
Parent process still continues.....
This is child process
```

Wait()

- Now there is another rule we must learn: when the parent dies before it **wait()**s for the child, the child is reparented to the **init** process (PID 1). This is not a problem if the child is still living well and under control. However, if the child is already defunct, we're in a bit of a bind. See, the original parent can no longer **wait()**, since it's dead. So how does **init** know to **wait()** for these *zombie processes*?
- On some systems, **init** periodically destroys all the defunct processes it owns. On other systems, it outright refuses to become the parent of any defunct processes, instead destroying them immediately.

Exec

The exec() family of functions replaces the current process image with a new process image.

```
#include <unistd.h>
```

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
 int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

Example:

```
execl("/bin/ls","-l",NULL);
```

Exec........

```
#include <stdio.h>
#include <unistd.h>

main()
{
int result;
printf("This is parent process\n");
result=fork();
if(result==0)
{
execl("/bin/ls","-l",NULL);
}
else
{
printf("Parent process still continues.....\n");
}
}
```

Threads

→A thread is a sequence of instructions to be executed with in a program. Threads, like processes, are a mechanism to allow to program to do more than one thing at a time.

→Conceptually, a thread exists within a process. Threads are a finer-grained unit of execution than processes.

→When you invoke a program, Linux creates a new process and in that process creates a single thread, which runs the program sequentially. That thread can create additional threads; all these threads run the same program in the same process, but each thread may be executing a different part of the program at any given time.

Threads

Threads in the same process share:

- Process instructions
- Most data
- open files (descriptors)
- signals and signal handlers
- current working directory
- User and group id

Each thread has a unique:

- Thread ID
- set of registers, stack pointer
- stack for local variables, return addresses
- signal mask
- priority
- Return value: errno

Why Threads are Required?

Suppose there is a process, that receiving real time inputs and corresponding to each input it has to produce a certain output. Now, if the process is not multi-threaded ie. if the process does not involve multiple threads, then the whole processing in the process becomes synchronous. This means that the process takes an input processes it and produces an output.

The limitation in the above design is that the process cannot accept an input until its done processing the earlier one and in case processing an input takes longer than expected then accepting further inputs goes on hold.

Threads

Creating thread

int **pthread_create**(pthread_t ****thread***, const pthread_attr_t ****attr***, void *(**start_routine*) (void *), void ****arg***);

thread: - returns the thread id.

attr*:* argument points to a *pthread_attr_t* structure, If *attr* is NULL, then the thread is created with default attributes.

void * (*start_routine) - pointer to the function to be threaded. Function has a single argument: pointer to void.

***arg -** pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

Threads

pthread_join- wait for termination of another thread

int **pthread_join**(pthread_t th, void **thread_return);

th - thread suspended until the thread identified by th terminates,
thread_return - If thread_return is not NULL, the return value of th is stored in the location pointed to by thread_return.

pthread_exit- terminate the calling thread

void **pthread_exit**(void ***retval**);

retval - Return value of thread.

Threads

pthread_detach- wait for termination of another thread

int **pthread_detach**(pthread_t thread);

The pthread_detach() function marks the thread identified by thread as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another
thread to join with the terminated thread.

RACE CONDITION

If multiple process concurrently access and modify a variable, then value of that variable depends upon the order of execution of those processes. In this case the programs may give unexpected result. this is known as race condition.Various synchronization mechanisms are used to avoid race condition.

Mutex

- A mutex is a special lock that only one thread may lock at a time. If a thread locks a mutex and then a second thread also tries to lock the same mutex, the second thread is blocked, or put on hold.
- Only when the first thread unlocks the mutex is the second thread unblocked—allowed to resume execution.

Mutex

- A typical sequence in the use of a mutex is as follows:

- Create and initialize a mutex variable
- Several threads attempt to lock the mutex
- Only one succeeds and that thread owns the mutex
- The owner thread performs some set of actions
- The owner unlocks the mutex
- Another thread acquires the mutex and repeats the process
- Finally the mutex is destroyed

Why Pthreads?

Following table compares timing results for the fork() subroutine and pthread_create() subroutine. Timing reflects 50,000 process/thread creations were performed with the time utility, and units are in seconds, no optimization flags.

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5

IPC (Inter Process Communication)

In **computing**, **inter-process communication** (**IPC**) is a set of methods for the exchange of data among multiple threads in one or more processes.

Linux supports the classic Unix IPC mechanisms of signals, pipes and semaphores and also the System V IPC mechanisms of shared memory, semaphores and message queues.

The types of inter process communication are:

Signals - Sent by other processes or the kernel to a specific process to indicate various conditions.

Pipes - Unnamed pipes set up by the shell normally with the "|" character to route output from one program to the input of another.

FIFOS - Named pipes operating on the basis of first data in, first data out.

Message queues - Message queues are a mechanism set up to allow one or more processes to write messages that can be read by one or more other processes.

Semaphores - Counters that are used to control access to shared resources. These counters are used as a locking mechanism to prevent more than one process from using the resource at a time.

Shared memory - The mapping of a memory area to be shared by multiple processes.

PIPEs

Pipes are an inter-process communication mechanism that is provided in all flavors of UNIX. A "pipe" defines one-way flow of data between processes. All data written to a pipe by a program is routed by the Kernel to another process, which can then access it and read the data.

pipe() function returns a pair of file descriptors.

One of these descriptors is connected to the write end of the pipe, and the other is connected to the read end

pipe() is useful using with **fork()**

Message Queues

(1)Creating message queue

int msgget(key_t ***key***, int ***msgflg***);

msgget() returns the message queue ID on success, or -1 on failure
key*:* is a system-wide unique identifier describing the queue
msgflg *:*should be set to the permissions with IPC_CREAT

```
key = ftok("/home/filename", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);
```

A message queue can also be created using command:

$ ipcmk –Q

$ ipcs –q

------ Message Queues --------

- key msqid owner perms used-bytes messages 0x33ec1686 65536 user 644 0 0

Message Queues

(2) **Sending to the queue**

Each message is made up of two parts, which are defined in the template structure struct msgbuf, as defined in *sys/msg.h*:

```
struct message {
        long mtype;
        char mtext[1]; } msg ;
```

To pass the information to a message queue

int **msgsnd**(int ***msqid***, const void *__*msgp*__, size_t ***msgsz***, int ***msgflg***);

msqid: is the message queue identifier returned by **msgget()**

msgp: is a pointer to the data you want to put on the queue

msgsz: is the size in bytes of the data

msgflg: allows you to set some optional flag parameters, otherwise 0

Message Queues

<u>(3)Receiving from the queue</u>

int **msgrcv**(int ***msqid***, void ****msgp***, size_t ***msgsz***, long ***msgtyp***, int ***msgflg***);

msqid: is the message queue identifier returned by **msgget()**
msgp: is a pointer to the data you want to get from the queue
msgsz: is the size in bytes of the data

<u>Msgtyp</u>	**<u>Effect on msgrcv()</u>**
Zero	Retrieve the next message on the queue, regardless of its *mtype*.
Positive	Get the next message with an *mtype equal to* the specified *msgtyp*.
Negative	Retrieve the first message on the queue whose *mtype* field is less than or equal to the absolute value of the *msgtyp* argument.

Message Queues

(4)<u>Destroying a message queue</u>

int **msgctl**(int ***msqid***, int ***cmd***, struct msqid_ds ****buf***);

msqid*:* is the message queue identifier returned by **msgget()**

cmd:the operation to be performed by msgctl() is specified in cmd and is one of:

IPC_STAT: Gather information about the message queue and place it in the structure pointed to by buf.

IPC_SET:

IPC_RMID: Remove the message queue specified by msqid and destroy the data associated with it.

buf*:* can be set to NULL for the purposes of IPC_RMID.

queue.c (~/Programs) - gedit

Open Save Undo

clientip.c queue.c read.c

```c
#include <string.h>
#include <sys/msg.h>

int main()
{
int msqid;
int mykey=ftok("chardev.c", 'B');
    struct message {
                  long type;
                  char text[40];
                 } msg;
long msgtyp = 0;

    msqid = msgget((key_t)mykey, 0666 | IPC_CREAT);

    msg.type = 1;

    strcpy(msg.text, "This is message 1");
    msgsnd(msqid, (void *) &msg, sizeof(msg.text), IPC_NOWAIT);

    strcpy(msg.text, "This is message 2");
    msgsnd(msqid, (void *) &msg, sizeof(msg.text), IPC_NOWAIT);

    msg.type = 2;

    strcpy(msg.text, "This is message 1 of type 2");
    msgsnd(msqid, (void *) &msg, sizeof(msg.text), IPC_NOWAIT);

    return 0;
}
```

C Tab Width: 4 Ln 29, Col 5 INS

read.c (~/Programs) - gedit

clientip.c | queue.c | read.c

```c
#include <stdio.h>
#include <sys/msg.h>

int main()
{
int msqid;

int mykey=ftok("chardev.c", 'B');
struct message {
    long type;
    char text[40];
               } msg;

    long msgtyp = 1;

    msqid = msgget((key_t)mykey, 0666);

    msgrcv(msqid, (void *) &msg, sizeof(msg.text), msgtyp, MSG_NOERROR | IPC_NOWAIT);

    printf("%s \n", msg.text);

    printf("Mykey= %d\n",mykey);

    return 0;
}
```

C | Tab Width: 4 | Ln 23, Col 1 | INS

Shared Memory

A segment of memory that is shared between processes.

(1)Creating the segment and connecting:

int **shmget**(key_t ***key***, size_t ***size***, int ***shmflg***);

shmget() returns an identifier for the shared memory segment
Key: is a system-wide unique identifier should be created with ftok
Size: is the size in bytes of the shared memory segment.
Shmflg: should be set to the permissions with IPC_CREAT

```
key_t key;
int shmid;

key = ftok("/home/filename", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
```

Shared Memory……

<u>(2)Attach me—getting a pointer to the segment</u>

void *shmat(int ***shmid***, void ****shmaddr***, int ***shmflg***);

shmat() returns a void pointer to the shared memory segment
shmid: is the shared memory ID we got from the call to **shmget()**
shmaddr: which you can use to tell **shmat()** which specific address to use but we can just set it to 0 and let the OS choose the address.
shmflg: can be set to SHM_RDONLY if you only want to read from it, 0 otherwise.

<u>complete example of how to get a pointer to a shared memory segment:</u>

```
key_t key;
int shmid;
char *data;

key = ftok("/home/filename", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
data = shmat(shmid, (void *)0, 0);
```

Shared Memory……

(3) **Reading and Writing**

Reading:

→printf("shared contents: %s\n", data);

→for (d = data; *d != NULL; d++)

putchar(*s);

putchar('\n');

Writing:

→printf("Enter a string: ");

gets(data);

Shared Memory……

<u>(4)Detaching from and deleting segments</u>

int shmdt(void ****shmaddr***);

Shmaddr: is the address you got from **shmat()**

When you detach from the segment, it isn't destroyed. Nor it is removed when *everyone* detaches from it.

destroy it using a call to **shmctl()**

shmctl(shmid, IPC_RMID, NULL);

→you can destroy the shared memory segment from the command line using the **ipcrm** Unix command

Concurrency

What are concurrency issues? Well, since you have multiple processes modifying the shared memory segment, it is possible that certain errors could crop up when updates to the segment occur simultaneously.

This *concurrent* access is almost always a problem when you have multiple writers to a shared object.

memset

memset -- Fill a region of memory with the given value

Synopsis

void * memset (void * s, int c, size_t count);

Arguments

s-- Pointer to the start of the area.

c--The byte to fill the area with

count--The size of the area.

Ftok

key_t **ftok**(const char* pathname,int *proj_id*);

The **ftok** function uses the identity of the file named by the given *pathname* (which must refer to an existing, accessible file) and the least significant 8 bits of *proj_id* (which must be nonzero) to generate a **key_t** type System V IPC key, suitable for use with **msgget, semget**, or **shmget.**

On success the generated **key_t** value is returned. On failure -1 is returned

IPC Mechanisms

In Linux

INDEX

IPC Mechanisms

- FIFO(Named PIPE)
- Semaphore
- Socket
- Signals
- Message queue vs PIPE

FIFO(Named PIPE)

- A first-in, first-out (FIFO) file is a pipe that has a name in the filesystem
- Any process can open or close the FIFO; the processes on either end of the pipe need not be related to each other. FIFOs are also called named pipes.
- A FIFO special file (a named pipe) is similar to a pipe, except that it is accessed as part of the file system.
- It can be opened by multiple processes for reading or writing. When processes are exchanging data via the FIFO, the kernel passes all data internally without writing it to the file system.
- Thus, the FIFO special file has no contents on the file system; the file system entry merely serves as a reference point so that processes can access the pipe using a name in the file system.

FIFO

- (1) **Creating a FIFO**
- int **mkfifo** (const char ***filename**, mode_t **mode**)
- **mkfifo**() makes a FIFO special file with name **filename**.
- **Mode**: specifies the FIFO's permissions

OR

- int **mknod**(const char ***path**, mode_t **mode**, rdev_t dev_**identifier**);
- Creates a FIFO special file (named pipe), with the path name specified in the path argument.
- The first byte of the mode argument determines the file type of the special file:
- **S_IFCHR**----Character special file
- **S_IFFIFO**----FIFO special file

FIFO

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#define FIFO_NAME "MYFIFO"

int main(void)
{
    char s[300];
    int num, fd,data;

    mknod(FIFO_NAME, S_IFIFO | 0666, 0);

    printf("waiting for readers...\n");
    fd = open(FIFO_NAME, O_WRONLY);
    printf("got a reader--type some stuff\n");

    data=gets(s);
    {
        if ((num = write(fd, s, strlen(s))) == -1)
            perror("write");
        else
            printf("I wrote %d bytes\n", num);

    }

    return 0;
}
```

FIFO

fifo1.c (~/Programs) - gedit

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#define FIFO_NAME "MYFIFO"

int main(void)
{
    char s[300];
    int num, fd;

    mknod(FIFO_NAME, S_IFIFO | 0666, 0);

    printf("waiting for writers...\n");
    fd = open(FIFO_NAME, O_RDONLY);
    printf("got a writer\n");

    do {
        if ((num = read(fd, s, 300)) == -1)
            perror("read");
        else
            {
            s[num] = '\0';
            printf(You reads %d bytes: \"%s\"\n", num, s);
            }
    } 
    while (num > 0);

    return 0;
}
```

FIFO

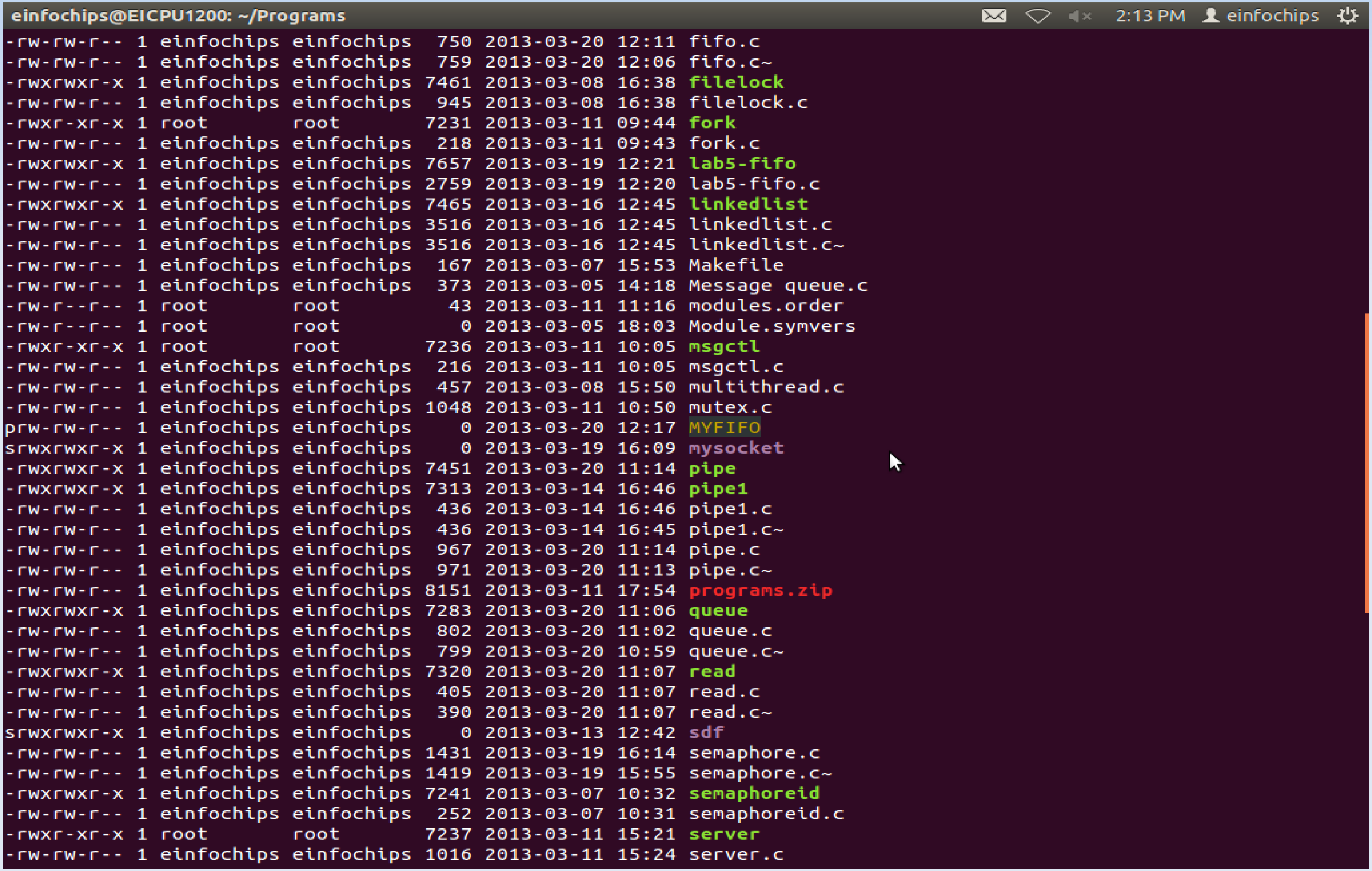
2:17 PM einfochips

```
einfochips@EICPU1200:~/Programs$ gcc -o fifo fifo.c
fifo.c: In function ‘main’:
fifo.c:23:9: warning: assignment makes integer from pointer without a cast [enabled by default]
einfochips@EICPU1200:~/Programs$ ./fifo
waiting for readers...
got a reader--type some stuff
Hello
I wrote 5 bytes
einfochips@EICPU1200:~/Programs$
```

FIFO

```
einfochips@EICPU1200:~/Programs$ gcc -o fifo1 fifo1.c
einfochips@EICPU1200:~/Programs$ ./fifo1
waiting for writers...
got a writer
You reads 5 bytes: "Hello"
You reads 0 bytes: ""
einfochips@EICPU1200:~/Programs$
```

FIFO


einfochips@EICPU1200: ~/Programs
2:13 PM einfochips
-rw-rw-r-- 1 einfochips einfochips 750 2013-03-20 12:11 fifo.c
-rw-rw-r-- 1 einfochips einfochips 759 2013-03-20 12:06 fifo.c~
-rwxrwxr-x 1 einfochips einfochips 7461 2013-03-08 16:38 filelock
-rw-rw-r-- 1 einfochips einfochips 945 2013-03-08 16:38 filelock.c
-rwxr-xr-x 1 root   root   7231 2013-03-11 09:44 fork
-rw-rw-r-- 1 einfochips einfochips 218 2013-03-11 09:43 fork.c
-rwxrwxr-x 1 einfochips einfochips 7657 2013-03-19 12:21 lab5-fifo
-rw-rw-r-- 1 einfochips einfochips 2759 2013-03-19 12:20 lab5-fifo.c
-rwxrwxr-x 1 einfochips einfochips 7465 2013-03-16 12:45 linkedlist
-rw-rw-r-- 1 einfochips einfochips 3516 2013-03-16 12:45 linkedlist.c
-rw-rw-r-- 1 einfochips einfochips 3516 2013-03-16 12:45 linkedlist.c~
-rw-rw-r-- 1 einfochips einfochips 167 2013-03-07 15:53 Makefile
-rw-rw-r-- 1 einfochips einfochips 373 2013-03-05 14:18 Message queue.c
-rw-r--r-- 1 root   root   43 2013-03-11 11:16 modules.order
-rw-r--r-- 1 root   root   0 2013-03-05 18:03 Module.symvers
-rwxr-xr-x 1 root   root   7236 2013-03-11 10:05 msgctl
-rw-rw-r-- 1 einfochips einfochips 216 2013-03-11 10:05 msgctl.c
-rw-rw-r-- 1 einfochips einfochips 457 2013-03-08 15:50 multithread.c
-rw-rw-r-- 1 einfochips einfochips 1048 2013-03-11 10:50 mutex.c
prw-rw-r-- 1 einfochips einfochips   0 2013-03-20 12:17 MYFIFO
srwxrwxr-x 1 einfochips einfochips   0 2013-03-19 16:09 mysocket
-rwxrwxr-x 1 einfochips einfochips 7451 2013-03-20 11:14 pipe
-rwxrwxr-x 1 einfochips einfochips 7313 2013-03-14 16:46 pipe1
-rw-rw-r-- 1 einfochips einfochips 436 2013-03-14 16:46 pipe1.c
-rw-rw-r-- 1 einfochips einfochips 436 2013-03-14 16:45 pipe1.c~
-rw-rw-r-- 1 einfochips einfochips 967 2013-03-20 11:14 pipe.c
-rw-rw-r-- 1 einfochips einfochips 971 2013-03-20 11:13 pipe.c~
-rw-rw-r-- 1 einfochips einfochips 8151 2013-03-11 17:54 programs.zip
-rwxrwxr-x 1 einfochips einfochips 7283 2013-03-20 11:06 queue
-rw-rw-r-- 1 einfochips einfochips 802 2013-03-20 11:02 queue.c
-rw-rw-r-- 1 einfochips einfochips 799 2013-03-20 10:59 queue.c~
-rwxrwxr-x 1 einfochips einfochips 7320 2013-03-20 11:07 read
-rw-rw-r-- 1 einfochips einfochips 405 2013-03-20 11:07 read.c
-rw-rw-r-- 1 einfochips einfochips 390 2013-03-20 11:07 read.c~
srwxrwxr-x 1 einfochips einfochips   0 2013-03-13 12:42 sdf
-rw-rw-r-- 1 einfochips einfochips 1431 2013-03-19 16:14 semaphore.c
-rw-rw-r-- 1 einfochips einfochips 1419 2013-03-19 15:55 semaphore.c~
-rwxrwxr-x 1 einfochips einfochips 7241 2013-03-07 10:32 semaphoreid
-rw-rw-r-- 1 einfochips einfochips 252 2013-03-07 10:31 semaphoreid.c
-rwxr-xr-x 1 root   root   7237 2013-03-11 15:21 server
-rw-rw-r-- 1 einfochips einfochips 1016 2013-03-11 15:24 server.c

Semaphores

- A semaphore is a primitive used to provide synchronization between various processes or between the various threads in a given process.
- Semaphores can be thought of as simple counters that indicate the status of a resource. This counter is a protected variable and cannot be accessed by the user directly.
- If counter is greater than 0, then the resource is available, and if the counter is 0 or less, then that resource is busy or being used by someone else

Semaphores

sem_open | **sem_init**

sem_wait/sem_trywait/sem_timedwait

sem_post

sem_getvalue

sem_wait/sem_trywait/sem_timedwait

Named Semaphore

Memory based /Unnamed Semaphore

sem_close

sem_unlink

sem_destroy

Semaphores

(1)<u>Creating a Semaphore</u>

-->int **sem_init**(sem_t ***sem**, int **pshared**, unsigned int **value**);

- Initializes an unnamed semaphore pointed to by **sem** to **value**.
- **pshared** indicates whether the semaphore is shared among processes (if pshared is zero, the semaphores is not shared; otherwise it is shared).
- Returns 0 on success and -1 on error
- sem_t sem;

sem_init(&sem, 0, 1);

-->sem_t ***sem_open**(const char ***name**, int **oflag**, mode_t **mode**, unsigned int **value**);

- Initializes the named **name** semaphore to **value**
- **Oflag is ORed between O_CREAT & O_EXCL,mode is for permission**

Semaphores

(2)<u>Waiting on a Semaphore</u>

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);

- If the value of the semaphore pointed to by sem is greater than 0, **sem_wait()** decreases it by 1 and returns immediately. Otherwise, the process is blocked.
- **sem_trywait()** is similar to sem_wait() except that if the semaphore value is not greater than 0, it returns an error.
- **sem_timedwait()** specifies a time limit pointed to by abs_timeout. If the decrement cannot proceed and time limit expires, the function returns an error.
- All these functions return 0 on success and -1 on error.

Semaphores

(3)Incrementing a Semaphore

int **sem_post**(sem_t ***sem**);

- Increments the value of the semaphore pointed to by sem

If the value becomes greater than 0, a process may be unblocked.

- Returns 0 on success and -1 on error.

Example

```
sem_t sem;
...
// do critical stuff
sem_post(&sem);
```

Semaphores

(4)<u>Getting the Value of a Semaphore</u>

- int **sem_getvalue**(sem_t ***sem**, int ***sval**);

- Gets the value of the semaphore pointed to by sem and places it in the location pointed to by sval.

Returns 0 on success and -1 on error.

Example:

```
sem_t sem;
...
int value;
sem_getvalue(&sem, &value);
```

Semaphores

(5)Destroying a Semaphore

int **sem_destroy**(sem_t ***sem**);

- Destroys the unnamed semaphore pointed to by sem (only used for

semaphores initialized by sem_init())

*int **sem_close**(sem_t ***sem**);*

- Disassociates the named semaphore pointed by sem from the

process (only used for semaphores created by sem_open())

int **sem_unlink**(const char ***name**);

- Removes the named name semaphore

Semaphores Example

In the following program, we have 3 threads.

Each thread needs to enter the critical section 3 times, where it sleeps for a random amount of time.

We initialize the semaphore to 2 so that at most 2 threads can be in the critical section at the same time.

We can see in the output that this is indeed the case.

einfochips@EICPU1200: ~/Programs

```
einfochips@EICPU1200:~/Programs$ gcc -pthread semaphore.c
einfochips@EICPU1200:~/Programs$ ./a.out
Thread A enters and sleeps for 4 seconds...
Thread C enters and sleeps for 1 seconds...
Thread C incremented counter= 1
Thread C leaves the critical section
Thread C enters and sleeps for 3 seconds...
Thread A incremented counter= 2
Thread A leaves the critical section
Thread A enters and sleeps for 3 seconds...
Thread C incremented counter= 3
Thread C leaves the critical section
Thread C enters and sleeps for 4 seconds...
Thread A incremented counter= 4
Thread A leaves the critical section
Thread A enters and sleeps for 0 seconds...
Thread A incremented counter= 5
Thread A leaves the critical section
Thread B enters and sleeps for 1 seconds...
Thread C incremented counter= 6
Thread C leaves the critical section
Thread B incremented counter= 7
Thread B leaves the critical section
Thread B enters and sleeps for 3 seconds...
Thread B incremented counter= 8
Thread B leaves the critical section
Thread B enters and sleeps for 1 seconds...
Thread B incremented counter= 9
Thread B leaves the critical section
einfochips@EICPU1200:~/Programs$
```

Sockets

- A socket is a bidirectional communication device that can be used to communicate with another process on the same machine or with a process running on other machines.
- struct sockaddr_un

```
{
unsigned short sun_family;  /* AF_UNIX */
char sun_path[108];
}
```

Sockets

Call comparison at server side & client side

Server side	**Client side**
socket()	socket()
bind()	
listen()	connect()
accept()	

Sockets

(1) **Create a socket (Server & Client)**

- int **socket**(int **domain**, int **type**, int **protocol**);

creates an endpoint for communication and returns a descriptor

- **domain:**communication domain in which the socket should be created.Some of address families are AF_INET (IP), AF_INET6 (IPv6), AF_UNIX (local channel, similar to pipes), AF_ISO (ISO protocols), and AF_NS (Xerox Network Systems protocols).

type:type of service

SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET,

SOCK_RAW, SOCK_RDM, SOCK_PACKET

protocol —indicate a specific protocol to use in supporting the sockets operation

- On success, a **socket descriptor** for the new socket is returned,On error, **-1** is returned

Sockets

(2)<u>**Assigning a name to a socket**</u>

- int **bind**(int **sockfd**, const struct sockaddr ***addr**,socklen_t **addrlen**);

bind() assigns the address specified by addr to the socket referred to by the file descriptor sockfd.

- **sockfd:** file descriptor of socket
- **addr:** a pointer to a sockaddr structure
- **addrlen:**the size, in bytes, of the address structure pointed to by addr.

- On success, **zero** is returned. On error, **-1** is returned

Sockets

(3) **Listen for incoming connections**

- int **listen**(int **sockfd**, int **backlog**);
- **listen**() marks the socket referred to by sockfd as a passive socket, that is, as a socket that will be used to accept incoming connection requests using accept(2).
- **Sockfd:** file descriptor of socket
- **backlog:** number of incoming connections that can be queued
- On success, zero is returned.On error, -1 is returned.

Sockets

<u>At client side</u>

- int **connect**(int **sockfd**, const struct sockaddr ***addr**, socklen_t **addrlen**);
- The **connect**() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr.
- **addrlen**: size of addr
- If the connection succeeds, zero is returned. On error, -1 is returned.

Sockets

(4) **Accept a connection from a client**

int **accept**(int **sockfd**, struct sockaddr ***addr**, socklen_t ***addrlen**);

- It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket.The newly created socket is not in the listening state. The original socket sockfd is unaffected by this call.
- **Sockfd:** file descriptor of socket created using socket()
- **addr:**a pointer to a sockaddr structure
- **addrlen:**the size, in bytes, of the address structure pointed to by addr.
- On success, these system calls return a nonnegative integer that is a **descriptor** for the accepted socket. On error, **-1** is returned

Sockets

(5) **Close the connection**

- int **shutdown** (int **sockfd**, int **how**);
- The shutdown() call causes all or part of a full-duplex connection on the socket associated with sockfd to be shut down.

sockfd: file descriptor of socket

how: allows us to tell the socket what part of the full-duplex connection to shut down:

SHUT_RD:Disables further receive operations.

SHUT_WR:Disables further send operations.

SHUT_RDWR:Disables further send and receive operations.

- **Close(sockfd)**

closes a file descriptor

Sockets

- struct sockaddr_in {
- sa_family_t sin_family; /* address family: AF_INET */
- in_port_t sin_port; /* port in network byte order */
- struct in_addr sin_addr; /* internet address */
- };

- /* Internet address. */
- struct in_addr {
- uint32_t s_addr; /* address in network byte order */
- };

Sockets

- struct hostent {
-     char *h_name;   /* official name of host */
-     char **h_aliases;   /* alias list */
-     int   h_addrtype;   /* host address type */
-     int   h_length;   /* length of address */
-     char **h_addr_list;   /* list of addresses */
- }
- #define h_addr h_addr_list[0] /* for backward compatibility */

atoi & htons

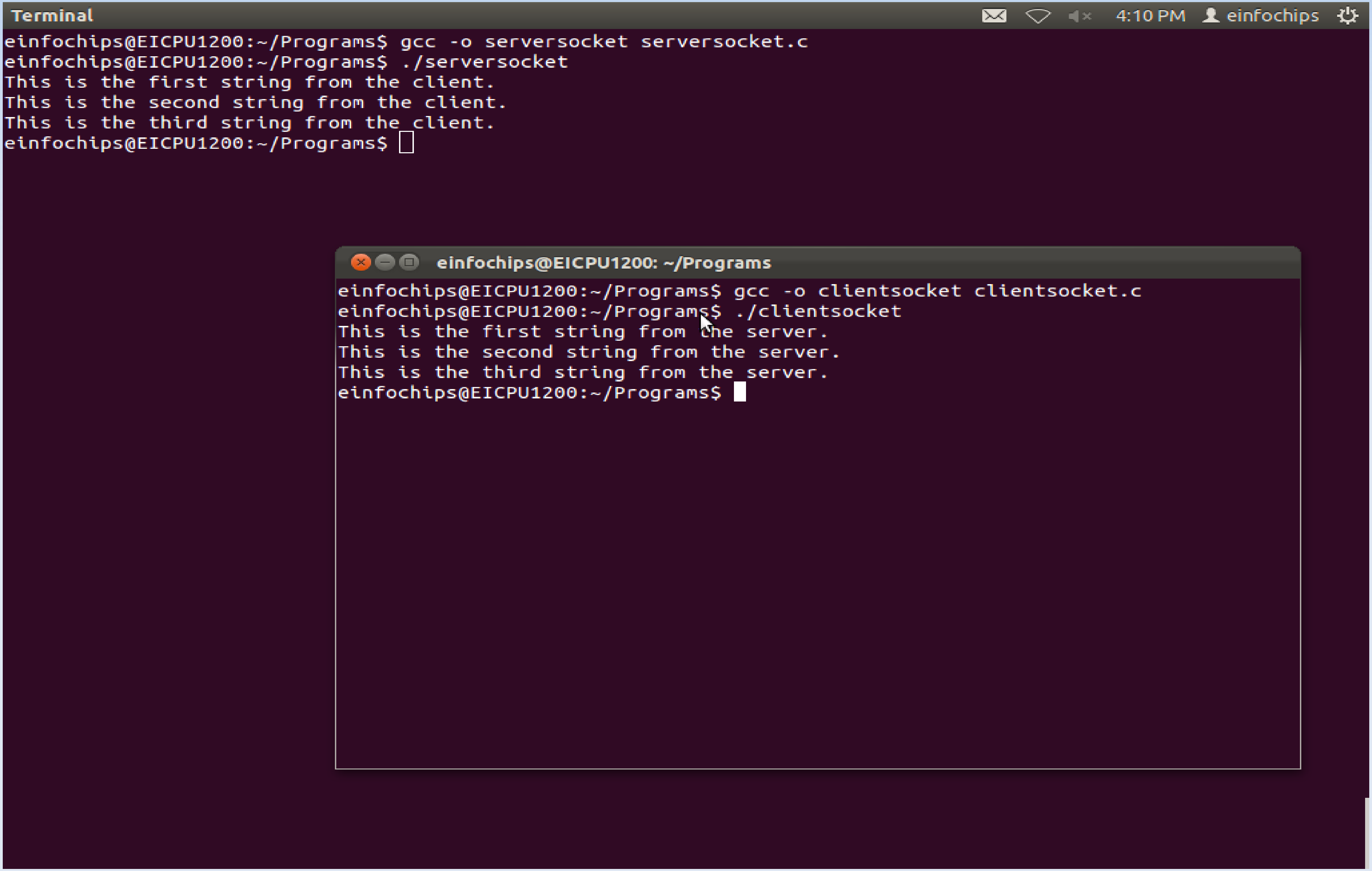
int **atoi** (const char * **str**);

- Convert ASCII string to integer

uint16_t **htons**(uint16_t **hostshort**);

- The htons() function converts the unsigned short integer **hostshort** from host byte order to network byte order.

Sockets

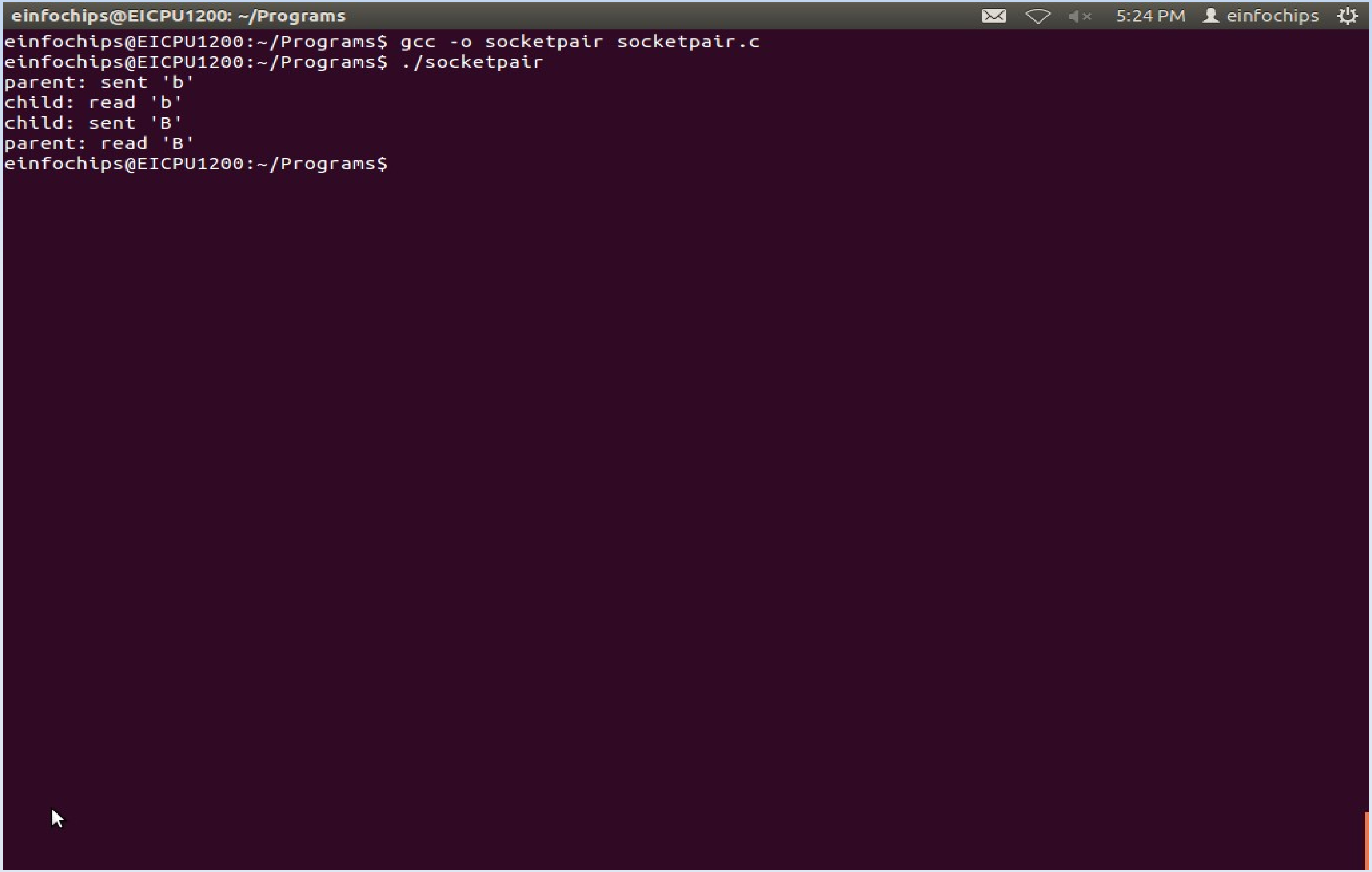

Terminal
4:10 PM
einfochips
einfochips@EICPU1200:~/Programs$ gcc -o serversocket serversocket.c
einfochips@EICPU1200:~/Programs$ ./serversocket
This is the first string from the client.
This is the second string from the client.
This is the third string from the client.
einfochips@EICPU1200:~/Programs$
einfochips@EICPU1200: ~/Programs
einfochips@EICPU1200:~/Programs$ gcc -o clientsocket clientsocket.c
einfochips@EICPU1200:~/Programs$ ./clientsocket
This is the first string from the server.
This is the second string from the server.
This is the third string from the server.
einfochips@EICPU1200:~/Programs$

Sockets

```
einfochips@EICPU1200: ~/Programs                                    2:59 PM  einfochips
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0p\222\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1544392, ...}) = 0
mmap2(NULL, 1554968, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x3aa000
mmap2(0x520000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x176) = 0x520000
mmap2(0x523000, 10776, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x523000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7779000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb77798d0, limit:1048575, seg_32bit:1, contents:0, read_exec_on
ly:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
mprotect(0x520000, 8192, PROT_READ)     = 0
mprotect(0x8049000, 4096, PROT_READ)    = 0
mprotect(0xd91000, 4096, PROT_READ)     = 0
munmap(0xb777a000, 57845)               = 0
socket(PF_FILE, SOCK_STREAM, 0)         = 3
unlink("mysocket")                      = 0
bind(3, {sa_family=AF_FILE, path="mysocket"}, 10) = 0
listen(3, 5)                            = 0
accept(3, {sa_family=0xd9 /* AF_??? */, sa_data="\0\0\0\0X\253x\267\1\0\0\0\0\0"}, [2]) = 4
fcntl64(4, F_GETFL)                     = 0x2 (flags O_RDWR)
brk(0)                                  = 0x8353000
brk(0x8374000)                          = 0x8374000
fstat64(4, {st_mode=S_IFSOCK|0777, st_size=0, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7788000
_llseek(4, 0, 0xbfe24318, SEEK_CUR)     = -1 ESPIPE (Illegal seek)
send(4, "This is the first string from th"..., 42, 0) = 42
send(4, "This is the second string from t"..., 43, 0) = 43
send(4, "This is the third string from th"..., 42, 0) = 42
read(4, "This is the first string from th"..., 4096) = 42
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7787000
write(1, "This is the first string from th"..., 42This is the first string from the client.
) = 42
read(4, "This is the second string from t"..., 4096) = 85
write(1, "This is the second string from t"..., 43This is the second string from the client.
) = 43
write(1, "This is the third string from th"..., 42This is the third string from the client.
) = 42
exit_group(10)                          = ?
einfochips@EICPU1200:~/Programs$ ^C
einfochips@EICPU1200:~/Programs$
```

SOCKETPAIR

- int **socketpair**(int **domain**, int **type**, int **protocol**, int **sv[2]**);
- The socketpair() call creates an unnamed pair of connected sockets in the specified domain, of the specified type, and using the optionally specified protocol.
- The descriptors used in referencing the new sockets are returned in sv[0] and sv[1]. The two sockets are indistinguishable.
- On success, **zero** is returned. On error, **-1** is returned,


einfochips@EICPU1200: ~/Programs
5:24 PM
einfochips
einfochips@EICPU1200:~/Programs$ gcc -o socketpair socketpair.c
einfochips@EICPU1200:~/Programs$ ./socketpair
parent: sent 'b'
child: read 'b'
child: sent 'B'
parent: read 'B'
einfochips@EICPU1200:~/Programs$

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>

int main(void)
{
    int sv[2]; /* the pair of socket descriptors */
    char buf; /* for data exchange between processes */

    if (socketpair(AF_UNIX, SOCK_STREAM, 0, sv) == -1)
     {
        perror("socketpair");
        exit(1);
     }
    if (!fork())     /* child */
     {
        read(sv[1], &buf, 1);
        printf("child: read '%c'\n", buf);
        buf = toupper(buf);       /* make it uppercase */
        write(sv[1], &buf, 1);
        printf("child: sent '%c'\n", buf);
     }
    else           /* parent */
     {
        write(sv[0], "b", 1);
        printf("parent: sent 'b'\n");
        read(sv[0], &buf, 1);
        printf("parent: read '%c'\n", buf);
        wait(NULL);     /* wait for child to die */
     }
    return 0;
}
```

Signal

- Signals are mechanisms for communicating with and manipulating processes in Linux.
- A signal is a special message sent to a process. Signals are asynchronous; when a process receives a signal, it processes the signal immediately, without finishing the cur-rent function or even the current line of code.
- Each signal has a current disposition, which determines how the process behaves when it is delivered the signal.

Signal

int **sigaction** (int **signum**, const struct sigaction *__act__, struct sigaction *__oldact__);

signum: (Input) A signal from the list defined in

act: A pointer to the **sigaction structure** that describes the action to be taken for the signal.

If act is a **NULL** pointer, signal handling is unchanged.

oldact: If oldact is non-NULL, the previous action is saved in oldact.

Signal

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
};
```

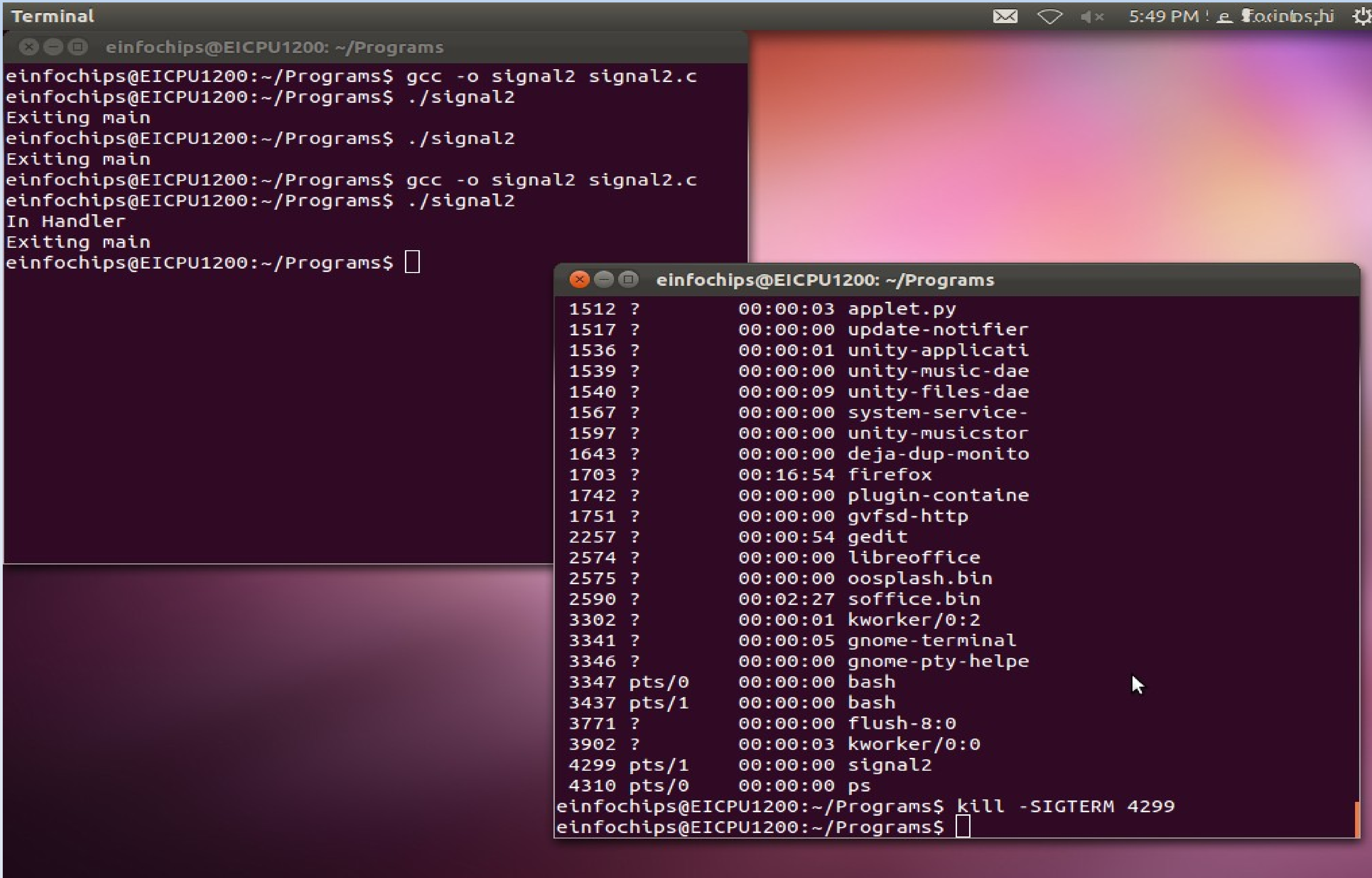
sa_handler: SIG_DFL, SIG_IGN or pointer to a function.

sa_mask: Additional set of signals to be blocked during execution of signal-catching function.

sa_flags: Special flags to affect behaviour of signal.

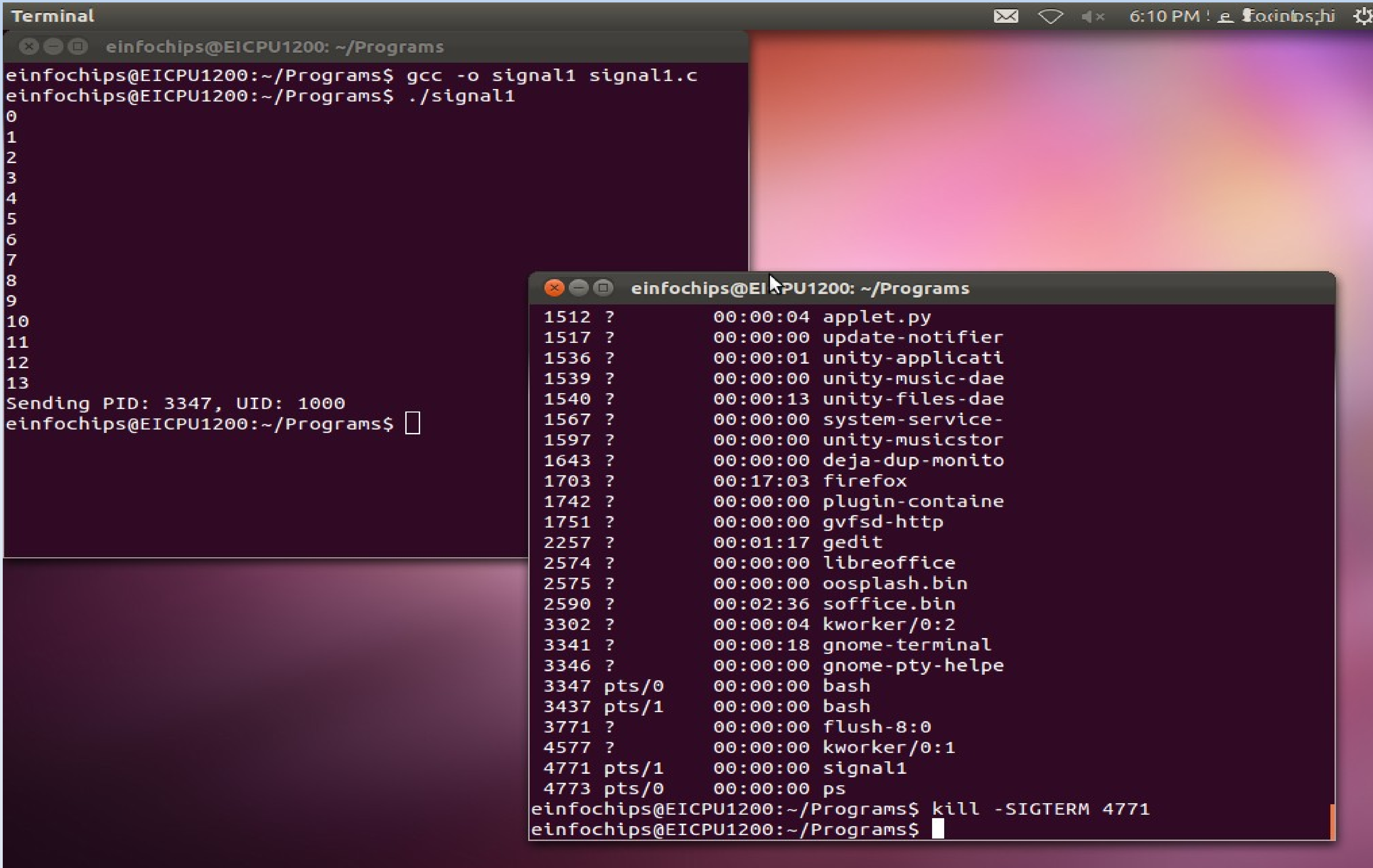
sa_sigaction: This is an alternative way to run the signal handler

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

static void handler (int sig)
{
    printf ("In Handler\n");
}

int main ()

{
    struct sigaction act;

    memset (&act, '\0', sizeof(act));

    act.sa_handler = &handler;
    act.sa_flags = 0;

    if (sigaction(SIGTERM, &act, NULL) < 0)
    {
        perror ("sigaction");
    }

    sleep (10);
    printf("Exiting main\n");

    return 0;
}
```


Terminal
5:49 PM
einfochips@EICPU1200: ~/Programs
einfochips@EICPU1200:~/Programs$ gcc -o signal2 signal2.c
einfochips@EICPU1200:~/Programs$ ./signal2
Exiting main
einfochips@EICPU1200:~/Programs$ ./signal2
Exiting main
einfochips@EICPU1200:~/Programs$ gcc -o signal2 signal2.c
einfochips@EICPU1200:~/Programs$ ./signal2
In Handler
Exiting main
einfochips@EICPU1200:~/Programs$
einfochips@EICPU1200: ~/Programs
1512 ? 00:00:03 applet.py
1517 ? 00:00:00 update-notifier
1536 ? 00:00:01 unity-applicati
1539 ? 00:00:00 unity-music-dae
1540 ? 00:00:09 unity-files-dae
1567 ? 00:00:00 system-service-
1597 ? 00:00:00 unity-musicstor
1643 ? 00:00:00 deja-dup-monito
1703 ? 00:16:54 firefox
1742 ? 00:00:00 plugin-containe
1751 ? 00:00:00 gvfsd-http
2257 ? 00:00:54 gedit
2574 ? 00:00:00 libreoffice
2575 ? 00:00:00 oosplash.bin
2590 ? 00:02:27 soffice.bin
3302 ? 00:00:01 kworker/0:2
3341 ? 00:00:05 gnome-terminal
3346 ? 00:00:00 gnome-pty-helpe
3347 pts/0 00:00:00 bash
3437 pts/1 00:00:00 bash
3771 ? 00:00:00 flush-8:0
3902 ? 00:00:03 kworker/0:0
4299 pts/1 00:00:00 signal2
4310 pts/0 00:00:00 ps
einfochips@EICPU1200:~/Programs$ kill -SIGTERM 4299
einfochips@EICPU1200:~/Programs$

signal1.c (~/Programs) - gedit

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

volatile int count = 0;

static void hdl (int sig, siginfo_t *siginfo, void *context)
{
    count=1;
    printf ("Sending PID: %ld, UID: %ld\n",(long)siginfo->si_pid, (long)siginfo->si_uid);
}

int main (int argc, char *argv[])
{
    int i=0;
    struct sigaction act;
    memset (&act, '\0', sizeof(act));

    /* Use the sa_sigaction field because the handles has two additional parameters */
    act.sa_sigaction = &hdl;

    /* The SA_SIGINFO flag tells sigaction() to use the sa_sigaction field, not sa_handler. */
    act.sa_flags = SA_SIGINFO;

    if (sigaction(SIGTERM, &act, NULL) < 0)
    {
        perror ("sigaction");
    }
    while(!count)
    {
    printf("%d\n",i++);
    sleep(1);
    }
    return 0;
}
```


Terminal
6:10 PM
einfochips@EICPU1200: ~/Programs
einfochips@EICPU1200:~/Programs$ gcc -o signal1 signal1.c
einfochips@EICPU1200:~/Programs$ ./signal1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
Sending PID: 3347, UID: 1000
einfochips@EICPU1200:~/Programs$
1512 ? 00:00:04 applet.py
1517 ? 00:00:00 update-notifier
1536 ? 00:00:01 unity-applicati
1539 ? 00:00:00 unity-music-dae
1540 ? 00:00:13 unity-files-dae
1567 ? 00:00:00 system-service-
1597 ? 00:00:00 unity-musicstor
1643 ? 00:00:00 deja-dup-monito
1703 ? 00:17:03 firefox
1742 ? 00:00:00 plugin-containe
1751 ? 00:00:00 gvfsd-http
2257 ? 00:01:17 gedit
2574 ? 00:00:00 libreoffice
2575 ? 00:00:00 oosplash.bin
2590 ? 00:02:36 soffice.bin
3302 ? 00:00:04 kworker/0:2
3341 ? 00:00:18 gnome-terminal
3346 ? 00:00:00 gnome-pty-helpe
3347 pts/0 00:00:00 bash
3437 pts/1 00:00:00 bash
3771 ? 00:00:00 flush-8:0
4577 ? 00:00:00 kworker/0:1
4771 pts/1 00:00:00 signal1
4773 pts/0 00:00:00 ps
einfochips@EICPU1200:~/Programs$ kill -SIGTERM 4771
einfochips@EICPU1200:~/Programs$

Signal Block

- A signal may be blocked, which means that it will not be delivered until it is later unblocked. Between the time when it is generated and when it is delivered a signal is said to be pending.
- In a traditional single-threaded application, sigprocmask(2) can be used to manipulate the signal mask.
- int **sigprocmask**(int **how**, const sigset_t ***set**, sigset_t ***oldset**);
- how: SIG_BLOCK

  SIG_UNBLOCK
- set: set of mask
- oldset: previous value of the signal mask is stored

Signal Block

int **sigemptyset(**sigset_t ***set);**

- **sigemptyset** initializes the signal set given by set to empty, with all signals excluded from the set.

int **sigaddset**(sigset_t ***set**, int **signum**);

- **sigaddset** add signal signum from set.

Message queue vs PIPE

- Pipes, once closed, require some amount of cooperation on both sides to reestablish them, message queues can be closed and reopened on either side without the coorporation of the other side.
- Default maximum size of queue is 16384 bytes,for PIPE it is 65536 bytes