Mini Project Operating System

Artjoms Tarasovs

Filip Sismis

UCN, University College of Northern Denmark

Computer science

dmaj0920

26. April 2021

University College of Northern Denmark,

Computer Science Academic Profession Programme

dmaj0920 group C
Mini Project Operating System

Project participants:

Artjoms Tarasovs

Filip Sismis

Supervisiors:

Karsten Jeppesen

Submission date: 26/04/2021

--------------- ---------------

Artjoms Tarasovs Filip Sismis

Preface

The group chose Linux Mint as operating system for this mini project. The main purpose of this report is to show reader some insights to Linux Mint distribution. And insides of Linux based operating system. In particular: thread model, memory management, scheduling details and I/O model.

Contents

1. Introduction

Linux distribution is operating system made from software collection that is based on the Linux kernel. Typical Linux distribution consists of a Linux kernel, GNU tools and libraries, desktop environment, a window system, a window manager, and some additional software. Linux Mint is one of many Linux distributions. It is based on the Ubuntu which was based on Debian.

Linux Mint is commonly recommended as an introduction to the Linux distributions. It's very simple to install and use out of the box since it comes with variety of free and open-source applications and is able to provide full multimedia support right after the installation if you choose to do so. It comes in multiple versions with different desktop environments and those are Cinnamon, MATE and xfce. Linux Mint is designed mostly for desktops. With its low minimum hardware requirements, it can be run also on older machines. Linux Mint distribution is community driven projects which is both free of cost and open source. All of the versions are freely available on the Linux Mint official website. [1]

Minimal hardware requirements:

- 1GB RAM (2GB recommended)
- 15 GB of hard-drive space (20 recommended)
- Screen of 1024x768 resolution (on lower resolutions some windows needs to be adjusted if they don't fit the screen)

2. Thread model

Linux has a unique implementation of threads. To the Linux kernel, there is *no* concept of a thread. Linux implements all threads as standard processes. The Linux kernel does not provide any special scheduling semantics or data structures to represent threads. Instead, a thread is merely a process that shares certain resources with other processes. Each thread has a unique `task_struct` and appears to the kernel as a normal process (which just happens to share resources, such as an address space, with other processes). The `task_struct` is a process descriptor inside of each process that is represented in Linux kernel as a C-language structure. The structure is quite large and offers all the information about one particular process as: process id, state, parent process, children, processor registers, etc. [2]

Linux approach to threads contrasts with other operating system such as Microsoft Windows which has an explicit kernel support for threads. Sometimes threads are called "lightweight processes". Name lightweight processes sums up the difference between Linux and other systems. Other systems view threads as abstraction to provide lighter and quicker execution than heavy process. While Linux sees threads as a manner of sharing resources between processes. [2]

This view of threads also changes the scheduling of the multi-threaded processes in the Linux kernel. As stated before Linux treats threads as processes so Linux turns process scheduling into thread scheduling by treating a scheduled process as if it was single-threaded. When a process is multi-threaded with N-amount of threads then equal N-amount of scheduling actions is required to cover the threads.[3]

3. Scheduling

Process scheduling is how the operating system assigns tasks to processors, then a running process can perform the task. Linux has a modular approach to processor scheduling, that different algorithms can be used to schedule different process types. One of the algorithms that was used for scheduling is Classic preemptive scheduling. It was popularized in Unix and not only Linux adopted it. Another one is Completely fair scheduling (CFS) which became part of the Linux kernel in 2007 and has been the default scheduler for Linux since then. It is used for scheduling normal processes as opposed to real-time. Therefore, it is named SCHED_NORMAL. CFS is fit for interactive applications typically found in a desktop environment. But also, can be configured as SCHED_BATCH for use in heavy batch workloads commonly found on high volume web servers. [3]

As the documentation of the kernel states most of the CFS design can be summed up in single sentence: CFS basically models and ideal precise multitasking CPU on real hardware. Which means that in such system, with n running processes, each process would have 1/n amount of CPU-power while running constantly. That is impossible to run more than one execution flow on a single core processor so CFS tries to mimic this perfectly fair scheduling. The CFS doesn't just assign a timeslice to a process but the scheduler calculates how long a task should run as a function of the total number of currently runnable processes. So instead of just setting a nice value to a task, the scheduler uses it to weight a proportion of the CPU the process is to receive. CFS achieves fairness by letting a task run for a period, proportional to its weight divided by the total weight of all runnable processes.[4]

4. Memory management

When Linux uses system RAM, it creates a virtual memory layer to then assigns processes to virtual memory. Virtual memory is a combination of both RAM and *swap space*. This extra abstraction layer is here so that each running process does not overlap and try to use memory already being used by another process. This also means that virtual memory can be expanded beyond the physical RAM capacity, which can be useful in a pinch even if it is not very efficient.[5]

As mentioned before, a process is allowed access to a section of RAM by the operating system; the process has no control over memory allocation. As a result, the developers that work on applications, do not really worry about working with memory.[5]

Usually, any file or part of a file system is mapped using the system command *mmap,* and is referred to as a memory mapped file. If a memory page file does not have any file associated with it, it is referred to as anonymous memory and utilities allocated using the "malloc" function. Using the way file mapped memory and anonymous memory are allocated, the operating system can have processes using the same files working with the same virtual memory page thus using memory more efficiently. [5]

Sometimes it happens that a process is occupying memory that is needed for another one. In this case, the OS uses the OOM (out of memory) killer. This utility chooses a process and reallocates its memory pages to other processes. [5]

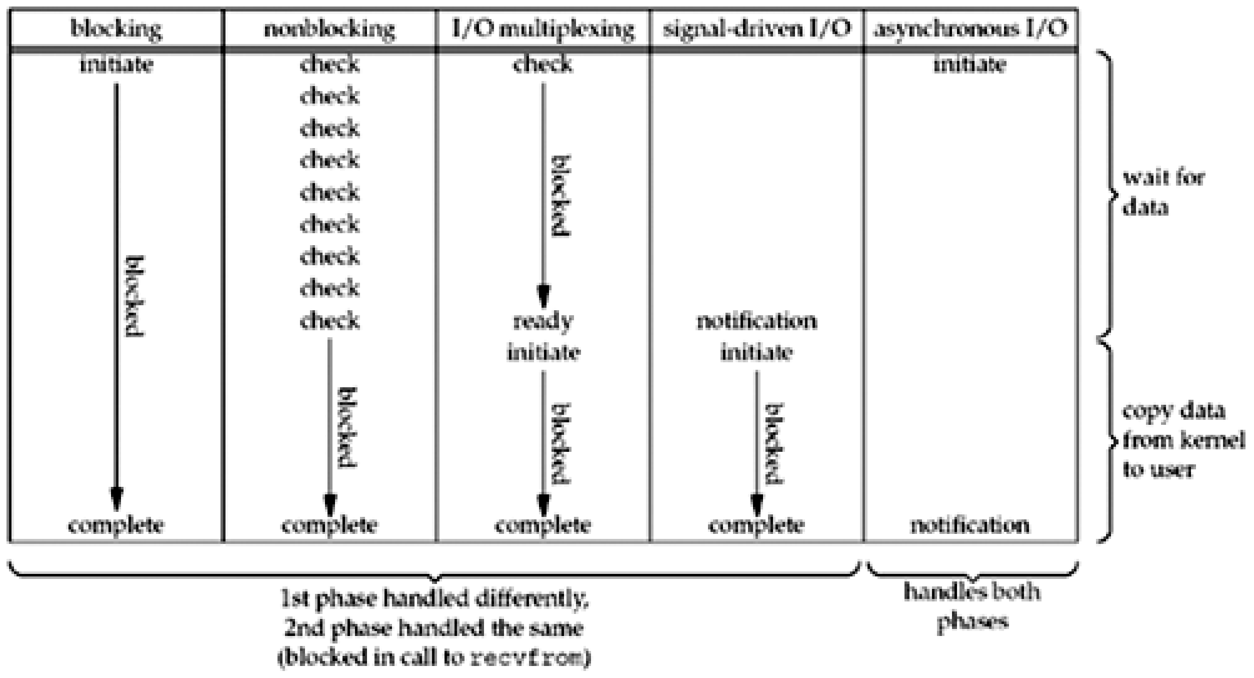
Swap space - a section of your hard drive designated as available for use in case usable RAM runs out.

5. I/O Model

There are five main I/O models under Linux:

1. Blocking I/O
2. Nonblocking I/O
3. I/O multiplexing
4. Signal Driven I/O
5. Asynchronous I/O

These models can be dived into four categories and those are Blocking I/O, Non-blocking I/O, Synchronous I/O and Asynchronous I/O. The first four models are synchronous because the actual I/O operation blocks the process until the I/O operation completes which can be seen in the figure below that the second phase of copying data from kernel to user is identical in the all of the first models. [6] Comparison of these five models can be seen here:

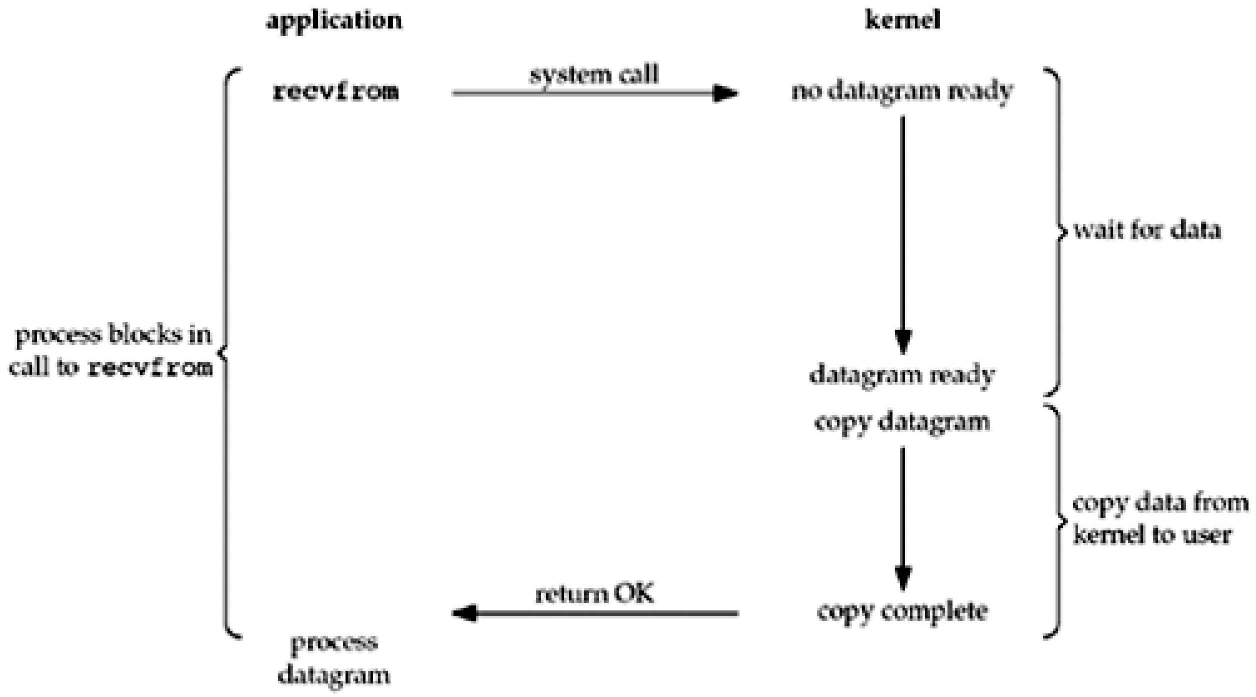


[6]

Comparison of five I/O models

5.1. Blocking I/O

The process will block until the data copy is complete. The application calls an IO function, causing the application to block and wait for the data to be ready. After the data is ready, copy from the kernel to user space, the IO function returns a success indication. The blocking IO model diagram is as follows:

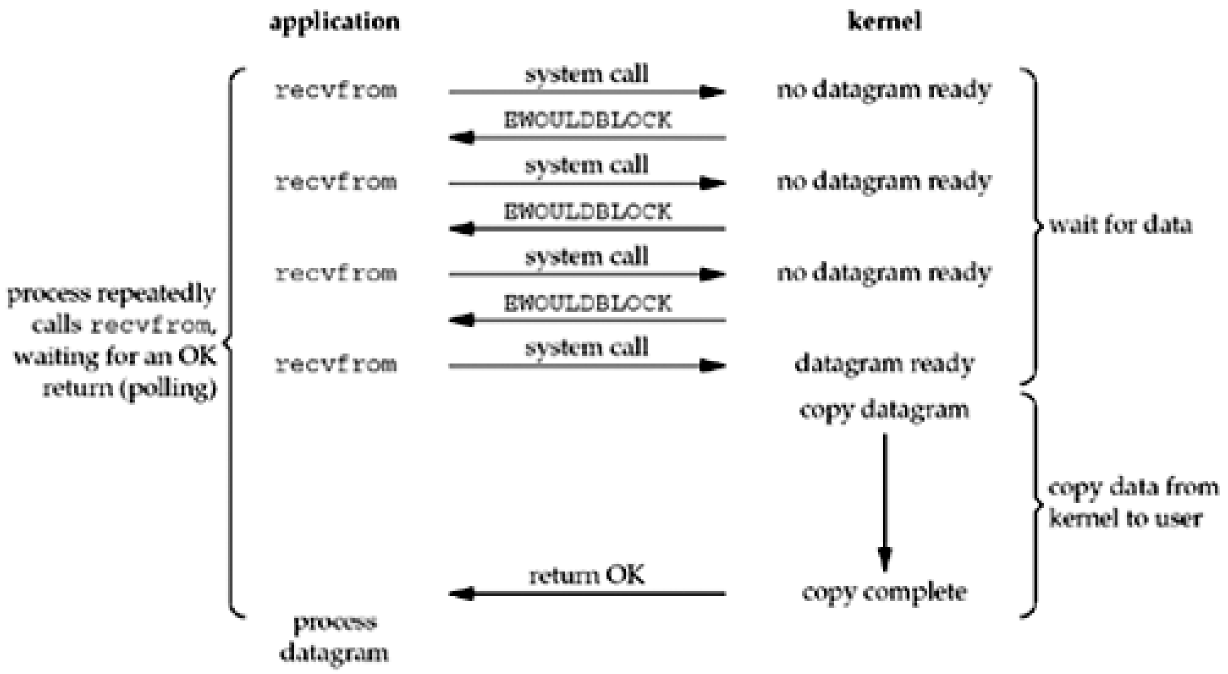


[6]

Blocking I/O model

5.2. Nonblocking I/O

Through the process repeatedly calling the IO function, the process is blocked during the data copy process. The model diagram is as follows:

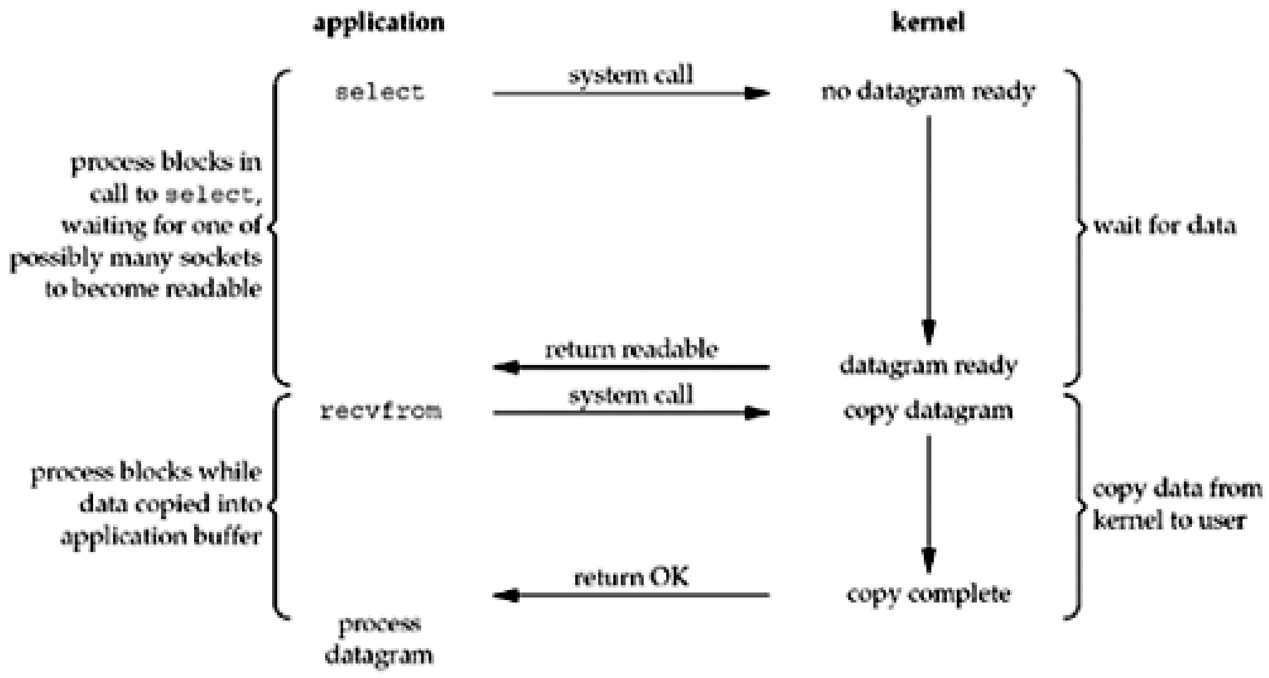


[6]

Nonblocking I/O model

5.3. I/O multiplexing

Mainly select and epoll. A thread can listen to multiple IO ports, and distribute to a specific thread for processing when the socket has read and write events. The model looks like this:

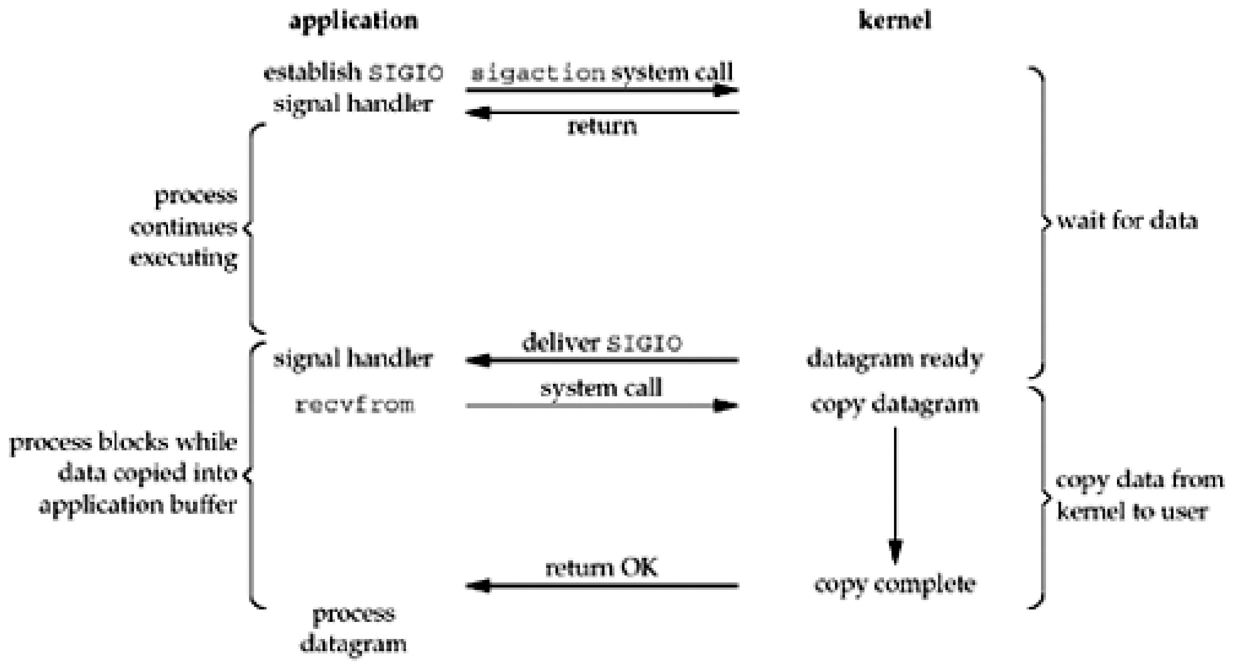

application
kernel
select
system call
no datagram ready
process blocks in call to select, waiting for one of possibly many sockets to become readable
wait for data
return readable
datagram ready
recvfrom
system call
copy datagram
process blocks while data copied into application buffer
copy data from kernel to user
return OK
copy complete
process datagram

[6]

I/O multiplexing model

5.4. Signal Driven I/O

Signal-driven I/O: First we allow the Socket to signal drive IO and install a signal handler that keeps the process running without blocking. When the data is ready, the process receives a SIGIO signal, which can be called by the I/O operation function in the signal handler. The process is shown below:

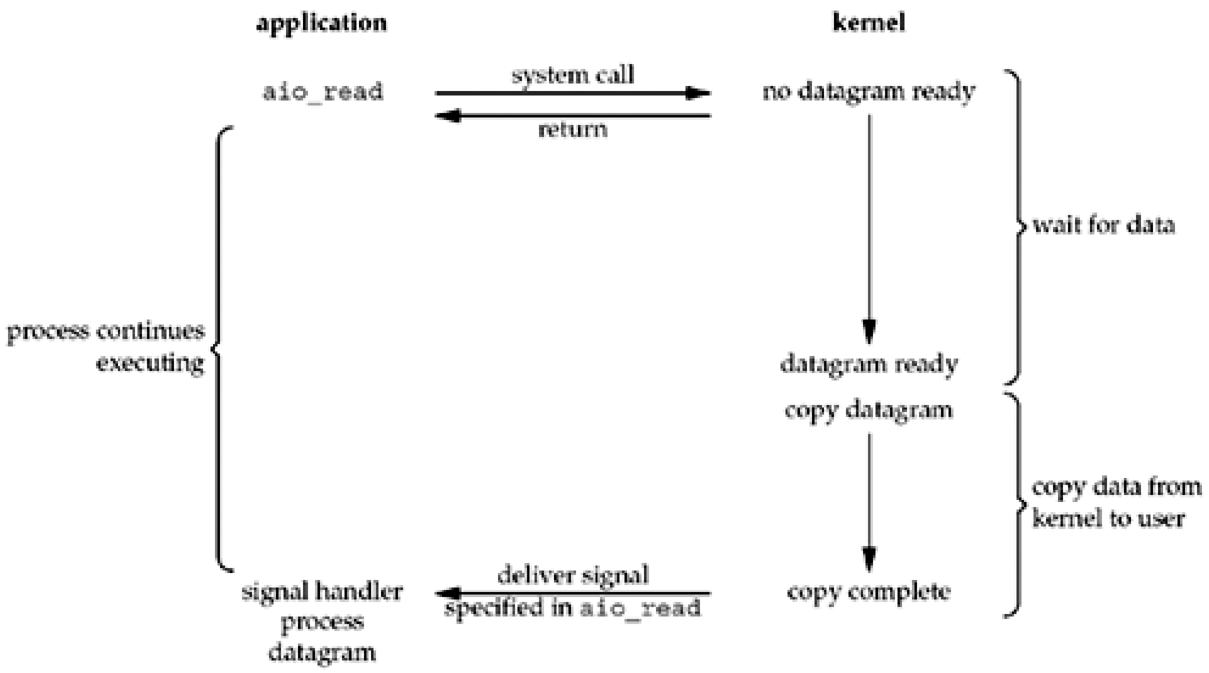


[6]

Signal driven I/O model

5.5. Asynchronous I/O

Asynchronous IO is not executed sequentially relative to synchronous IO. After the user process makes the aio_read system call, regardless of whether the kernel data is ready, it will be returned directly to the user process, and then the user state process can do other things. When the socket data is ready, the kernel directly copies the data to the process and then sends a notification from the kernel to the process. In both phases of IO, the processes are non-blocking. The asynchronous process is shown below:


application
kernel
aio_read
system call
return
no datagram ready
wait for data
process continues executing
datagram ready
copy datagram
copy data from kernel to user
deliver signal specified in aio_read
copy complete
signal handler process datagram

[6]

Asynchronous I/O model

References

[1] Official Linux Mint website: https://linuxmint.com/ (Accessed April 2021)

[2] Robert Love: *Linux Kernel Development, 2nd Edition.* 2005

[3] Marty Kalin: Completely fair process scheduling in Linux: https://opensource.com/article/19/2/fair-scheduling-linux (Accessed April 2021)

[4] Wolfgang Mauerer: *Professional Linux Kernel Architecture*. Wrox, 2008

[5] Overview of the Memory management from ServerSuit: https://serversuit.com/community/technical-tips/view/how-does-linux-handle-ram.html (Accessed April 2021)

[6] https://www.programmersought.com/article/17295141509/ (Accessed April 2021)