ĐẠI HỌC CÔNG NGHỆ - ĐHQGHN 

**KHOA ĐIỆN TỬ VIỄN THÔNG**

----- □ & □ -----



**Thực tập thiết kế hệ thống nhúng**

**Nhóm thực tập Nhúng- Dự án gNode 5G VHT**

**TẬP ĐOÀN CÔNG NGHIỆP VIỄN THÔNG QUÂN ĐỘI**

**TỔNG CÔNG TY CÔNG NGHIỆP CÔNG NGHỆ CAO VIETTEL**

**Báo cáo thực tập**

Sinh viên thực hiện:

**Đỗ Thái Vũ - 19021540**

**Hà Nội, tháng 9 năm 2022**

**LỜI NÓI ĐẦU**

*Lời đầu tiên cho em xin gửi lời cảm ơn đến Khoa Điện tử Viễn thông – Trường Đại học Công nghệ - Đại học Quốc gia Hà Nội đã tạo điều kiện cho em được đi thực tập và viết báo cáo này, đây là một cơ hội tốt để em có thể thực hành và áp dụng những kiến thức đã được học trên nhà trường cũng như giúp em có cơ hội được trải nghiệm cơ hội làm việc trực tiếp trong môi trường doanh nghiệp.*

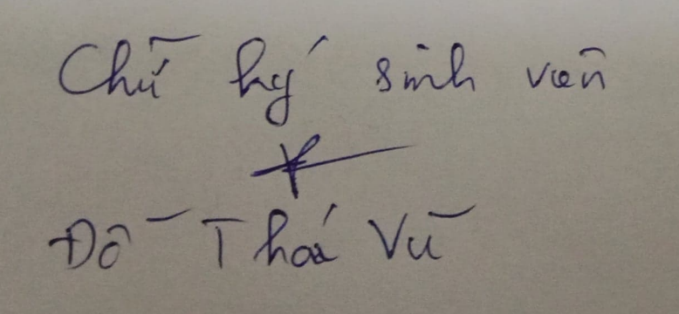
*Em xin gửi lời chân thành cảm ơn đến Trung tâm Kỹ Thuật Công Nghệ - Khối 2 – Tổng Công Ty Công nghiệp Công nghệ cao Viettel, các anh chị trong dự án gNodeB 5G đã tạo điều kiện cho em được thực tập tại đơn vị, đặc biệt là anh Tạ Quốc Việt, anh Tăng Thiên Vũ, anh Đậu Hồng Quân, anh Hoàng Đức Trường và anh Trần Nam Hải thời gian qua đã tận tình chỉ bảo, hướng dẫn cũng như chia sẻ cho em những kinh nghiệm quý báu trong suốt thời gian thực tập tại đơn vị.*

*Em xin gửi lời cảm ơn đến thầy Lưu Mạnh Hà trong suốt thời gian qua đã giúp đỡ em để em có thể hoàn thiện tốt báo cáo thực tập này.*

*Do hiểu biết hạn hẹp của mình, báo cáo của em còn nhiều thiếu xót, sai lầm, mong các anh và thầy chỉ báo vào giúp đỡ.*

*Em xin cảm ơn mọi người.*

                                                                            Sinh viên thực tập



# MỤC LỤC

[CHƯƠNG 1: HỆ THỐNG NHÚNG LINUX 4](#_Toc114010861)

[1. Tìm hiểu về Hệ thống nhúng Linux 4](#_Toc114010862)

[1.1 Giới thiệu về Hệ thống nhúng 4](#_Toc114010863)

[1.2 Giới thiệu về Hệ điều hành Linux 5](#_Toc114010864)

[1.2.1 Giới thiệu chung về Linux 5](#_Toc114010865)

[1.2.2 Cấu trúc của Linux 5](#_Toc114010866)

[1.3 Giới thiệu về Hệ thống nhúng Linux 6](#_Toc114010867)

[1.3.1 Giới thiệu chung về Hệ thống nhúng Linux 6](#_Toc114010868)

[1.3.2 Cấu trúc của hệ thống nhúng Linux 7](#_Toc114010869)

[2. Tìm hiểu về Real-Time 10](#_Toc114010870)

[3. Kiểm tra tính Real-Time của hệ thống 11](#_Toc114010871)

[Chương II : Lý thuyết và triển khai 12](#_Toc114010872)

[1. Lý thuyết và triển khai hệ thống nhúng trên trên thiết bị (Orange Pi) 12](#_Toc114010873)

[1.1 Giới thiệu phần cứng (Orange Pi 3) 12](#_Toc114010874)

[1.2 Armbian OS cho Orange Pi 12](#_Toc114010875)

[1.2.1 Giới thiệu về Armbian OS 12](#_Toc114010876)

[1.2.2 Một số công cụ trong quá trình Boot Armbian lên Orange Pi 13](#_Toc114010877)

[1.2.3 Tiến hành Boot lên Orange Pi 15](#_Toc114010878)

[1.2.4 Ưu-Nhược điểm của Armbian 15](#_Toc114010879)

[1.3 Build Image bằng Yocto Project 16](#_Toc114010880)

[1.3.1 Giới thiệu về Yocto Project 16](#_Toc114010881)

[1.3.2 Poky 16](#_Toc114010882)

[1.3.3 Bitbake 17](#_Toc114010883)

[1.3.4 Build Image cho Orange Pi bằng Yocto Project 18](#_Toc114010884)

[1.3.5 Build Image cho Orange Pi bằng Xunlong github 19](#_Toc114010885)

[2. Lý thuyết về Kernel Real-Time 21](#_Toc114010886)

[2.1 Scheduling 21](#_Toc114010887)

[2.1.1 Timeshare Policy 21](#_Toc114010888)

[2.1.2 Real-Time Policy 22](#_Toc114010889)

[2.2 Scheduling latency 22](#_Toc114010890)

[2.3 Priority 23](#_Toc114010891)

[3. Xây dựng Kernel Real-Time cho Orange Pi 23](#_Toc114010892)

[4. Lý thuyết C Programing 26](#_Toc114010893)

[4.1 Multithread, sleep, mutex 26](#_Toc114010894)

[4.1.1 Multithread 26](#_Toc114010895)

[4.1.2 Timer and Sleep 27](#_Toc114010896)

[4.1.3 Mutex 27](#_Toc114010897)

[4.2 Tối ưu hóa chương trình bằng phương pháp lập lịch 28](#_Toc114010898)

[4.3 Triển khai 29](#_Toc114010899)

[CHƯƠNG III: KẾT QUẢ VÀ NHẬN XÉT KẾT QUẢ 31](#_Toc114010900)

[1. Thực hiện chạy kết quả 31](#_Toc114010901)

[1.1 Trường hợp 1 31](#_Toc114010902)

[1.2 Trường hợp 2 34](#_Toc114010903)

[1.3 Trường hợp 3 35](#_Toc114010904)

[1.4 Trường hợp 4 37](#_Toc114010905)

[2. Nhận xét kết quả đạt được 41](#_Toc114010906)

[CHƯƠNG IV: KẾT QUẢ ĐẠT ĐƯỢC TRONG THỜI GIAN THỰC TẬP 42](#_Toc114010907)

# CHƯƠNG 1: HỆ THỐNG NHÚNG LINUX

## 1. Tìm hiểu về Hệ thống nhúng Linux

### 1.1 Giới thiệu về Hệ thống nhúng

- Hệ thống nhúng (Embedded system) là từ ngữ chuyên ngành để chỉ hệ thống có khả năng tự trị được nhúng vào trong môi trường (phần cứng). Mỗi hệ thống nhúng thường được phục vụ cho bài toán toán chuyên dụng trong nhiều lĩnh vực công nghiệp như tự động hóa, đo đạc thông tin, điều khiển kỹ thuật…

- Hệ thống nhúng được tích hợp cả phần mềm và phần cứng tùy vào mục đich chuyên dụng nào đó. Do đó hệ thống nhúng chỉ thực hiện được một vài chức năng nhất định. Hệ thống nhúng rất đa dạng và phong phú về thể loại, nó có mặt trong hầu hết các lĩnh vực trong cuộc sống:

+ Lĩnh vực dân sự:

* Các thiết bị thông minh trong gia đình : Điều hòa, tivi, máy giặt …
* Các thiết bị phục vụ công nghiệp : Hệ thống điều khiển trong nhà máy, hệ thống dây truyền, robot …
* Hệ thống giao thông vận tải : Hệ thống đèn điều khiển, Xe tự lái…
* Hệ thống viễn thông.

+ Lĩnh vực quân sự:

* Các thiết bị quân sự : Hệ thống giám sát tên lửa, thiết bị điều khiển máy bay…
* Các hệ thống dò tin và chống nhiễu.

### 1.2 Giới thiệu về Hệ điều hành Linux

#### 1.2.1 Giới thiệu chung về Linux

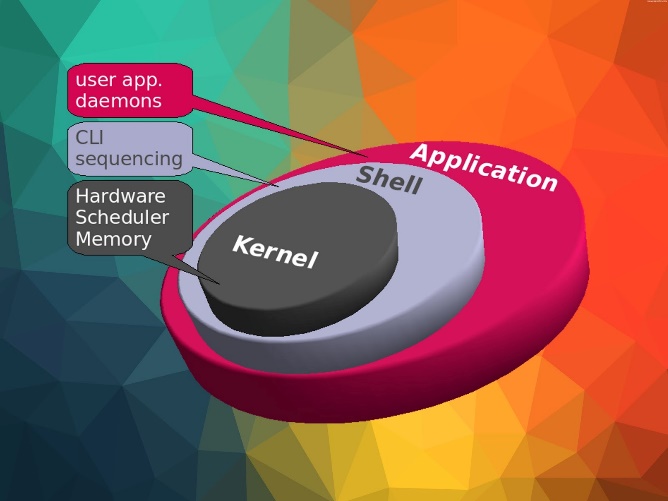
- Linux là một hệ điều hành máy tính được phát triển (bởi nhà khoa học Linus Torvalds vào năm 1991) trên hệ điều hành Unix được viết bằng C và được phát hành miễn phí.

- So với các hệ điều hành hiện nay như Windows , MS-DOS thì Linux tỏ ra phù hợp với môi trường doanh nghiệp với các ưu điểm :

* Mã nguồn mở, không mất tiền mua bản quyền.
* Đa người dùng, tính linh hoạt cao.
* Có thể sử dụng trên nhiều loại máy tính khác nhau.
* Tính bảo mật cao.
* Có cộng đồng hỗ trợ lớn.

#### 1.2.2 Cấu trúc của Linux

- Về cơ bản, cấu trúc của Linux gồm 3 phần là : Kernel, Shell, Application



Hình 1.1: Cấu trúc cơ bản của Linux

* Application : Là các chương trình và ứng dụng tiện ích mà người dùng cài đặt trên hệ điều hành.
* Shell : Là chương trình có tác dụng thực thi các lệnh (command) của người dùng từ Application chuyển đến Kernel để xử lý. Đây là cầu nối trung gian giữa Kernel và Application. Shell có nhiệm vụ phiên dịch các tập lệnh gửi từ Application và gửi đến Kernel để thực thi.
* Kernel : “Nhân” của Linux, đây là phần quan trọng nhất của Linux. Kernel chứa phần cứng (Hardware), bộ lập lịch (Scheduler), bộ nhớ (Memory), ngoài ra còn các Module, thư viện (Library) để quản lý và giao tiếp được với phần cứng và ứng dụng liên quan.

### 1.3 Giới thiệu về Hệ thống nhúng Linux

#### 1.3.1 Giới thiệu chung về Hệ thống nhúng Linux

- Sau khi đã hiểu về định nghĩa “Hệ thống nhúng” và “Linux”, thì bạn đọc đã tưởng tượng đôi phần về Hệ thống nhúng Linux. Chúng được hiểu đơn giản là hệ thống nhúng dựa trên Linux Kernel (Nhân), bản phân phối của hệ thống nhúng Linux bao gồm các phần như : khung hệ thống, các ứng dụng phần mềm khác nhau được thiết kế riêng trên hệ thống.

-Về phần mềm, các bản phân phối sử dụng công cụ (tool) để phát triên Hệ thống nhúng:

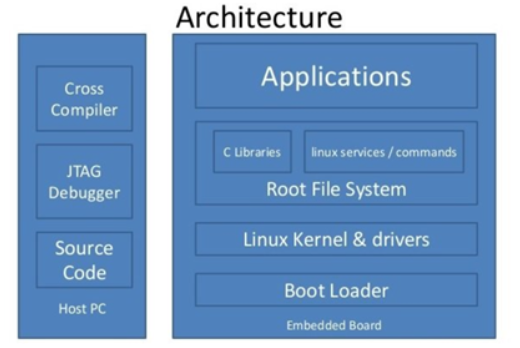
* Trình biên dịch chéo (Cross-Compile)
* Trình gỡ lỗi (Debug)
* Trình duyệt đặc biệt

- Về phần cứng, các bản phân phối hỗ trợ một số vi xử lý 32 bit hoặc 64 bit có kiến trúc phổ biến như :

* X86
* ARM
* MIPS
* SH
* PPC/PowerPC

#### 1.3.2 Cấu trúc của hệ thống nhúng Linux

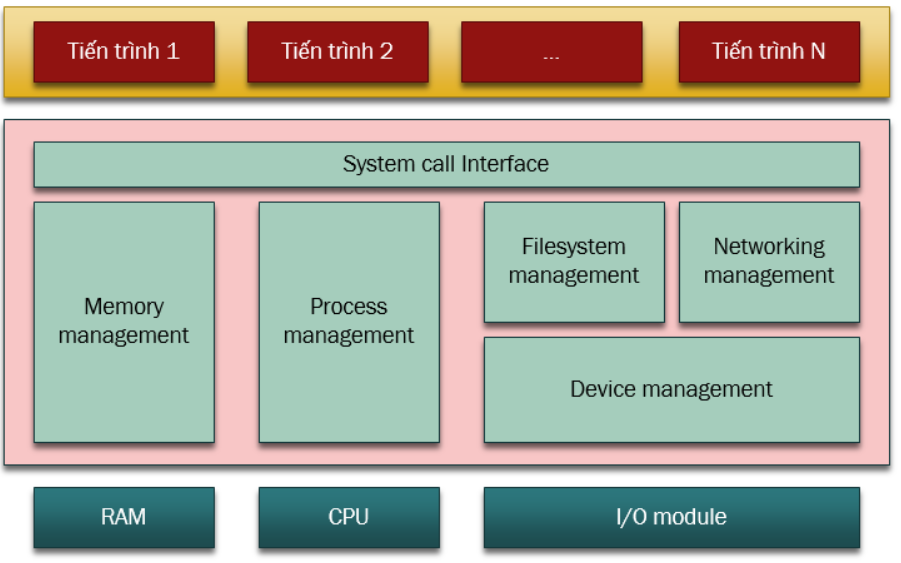
- Hình 2 biểu diễn một số thành phần chính của Hệ thống nhúng Linux



Hình 1.2: Thành phần của Hệ thống nhúng Linux

- Các thành phần của hệ thống nhúng Linux :

1. Linux Kernel : “Nhân” của hệ thống và các driver liên quan.



Hình 1.3: Kiến trúc của Linux Kernel dưới góc độ quản lý

Linux Kernel chia làm 3 phần gồm User-Space, Kernel-Space, Hardware, trong đó nó cung cấp một số dịch vụ giao tiếp với phần cứng :

+ Process management: có nhiệm vụ quản lý các tiến trình, bao gồm các công việc:

* Tạo/hủy các tiến trình.
* Lập lịch cho các tiến trình. Đây thực chất là lên kế hoạch: CPU sẽ thực thi chương trình khi nào, thực thi trong bao lâu, tiếp theo là chương trình nào.
* Hỗ trợ các tiến trình giao tiếp với nhau.
* Đồng bộ hoạt động của các tiến trình để tránh xảy ra tranh chấp tài nguyên.

+ Memory management: có nhiệm vụ quản lý bộ nhớ, bao gồm các công việc:

* Cấp phát bộ nhớ trước khi đưa chương trình vào, thu hồi bộ nhớ khi tiến trình kết thúc.
* Đảm bảo chương trình nào cũng có cơ hội được đưa vào bộ nhớ.
* Bảo vệ vùng nhớ của mỗi tiến trình.

+ Device management: có nhiệm vụ quản lý thiết bị, bao gồm các công việc:

* Điều khiển hoạt động của các thiết bị.
* Giám sát trạng thái của các thiết bị.
* Trao đổi dữ liệu với các thiết bị.
* Lập lịch sử dụng các thiết bị, đặc biệt là thiết bị lưu trữ (ví dụ ổ cứng).

+ File system management: có nhiệm vụ quản lý dữ liệu trên thiết bị lưu trữ (như ổ cứng, thẻ nhớ). Quản lý dữ liệu gồm các công việc: thêm, tìm kiếm, sửa, xóa dữ liệu.

+ Networking management: có nhiệm vụ quản lý các gói tin (packet) theo mô hình TCP/IP.

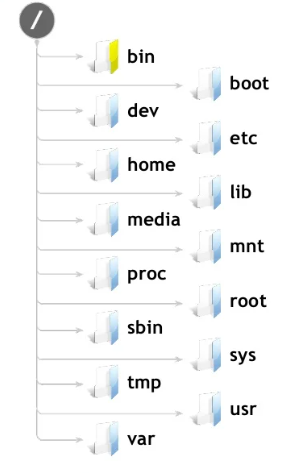
+ System call Interface: có nhiệm vụ cung cấp các dịch vụ sử dụng phần cứng cho các tiến trình. Mỗi dịch vụ được gọi là một system call.

1. Toolchain

Do phần cứng của Hệ thống nhúng không đủ tài nguyên để tự xây dựng các file bin nên chúng được biên dịch thông qua trình biên dịch chéo (Cross-Compile) , GNU Debugger…

1. Root File System

Linux Kernel hoạt động với tệp root. Hình 1.4 dưới đây chỉ các thư mục liên kết đến root ( “/” )



Hình 1.4: Thư mục Root

## 2. Tìm hiểu về Real-Time

- Hệ điều hành Thời gian thực ( Real-Time Operating System) là hệ điều hành thường được nhúng trong vi xử lý nhằm tối ưu thời gian xử lý các task và đa nhiệm.

* Linux không phải là hệ điều hành Real-Time.

- Hệ thống nhúng Thời gian Thực (Real-Time System) là hệ thống có các ràng buộc thời gian cố định, được xác định rõ rang, có độ trễ thấp. Quy trình xử lý được thực hiện trong các rang buộc đó, ở đây yếu tố thời gian thực cũng quan trọng không kém với tính chính xác kết quả đầu ra.

- Dưới đây là sự so sánh giữa RTOS (FreeRTOS) và Non-Reatime (Linux)

|  |  |
| --- | --- |
| Non real-time | Real-time Operating System |
| - Các task không có ràng buộc về thời gian  - Không phân chia thành các phần chi tiết  -Thời hạn cho các task có thể là vài phút, vài giờ hoặc thậm chí là vài ngày | - Các task có ràng buộc về thời gian  - Chia thành 2 loại là Soft-Realtime và Hard-Realtime  - Thời hạn cho các task theo thứ tự giây |

- Có thể thấy RTOS đòi hỏi ràng buộc thời gian chặt chẽ hơn Non-Real Time. Ngoài ra trong RTOS cũng chia ra thành 2 loại Hard-Realtime và Soft-Realtime.

* Trong khi Hardtime cần phải tối ưu cả phần mềm và phần cứng thì Soft-Realtime chỉ cần tối ưu phần mềm. Dù không được tốt như Hard-Realtime xong chất lượng của Soft-Realtime vẫn chấp nhận được.
* Kernel Realtime là một bản Soft-Realtime vì nó sử dụng bản kernel thông thường và patching kernel realtime.

## 3. Kiểm tra tính Real-Time của hệ thống

- Để kiểm tra tính Real-Time của hệ thống, chúng ta sẽ chạy chương trình trên 2 môi trường là Non-Realtime và Realtime. Chương trình được đặt ra là so sánh thời gian lấy mẫu (tại một thời gian nhất định), kiểm tra sự sai khác của chúng thông qua chương trình C.

-Chương trình được chia ra thành 3 thread:

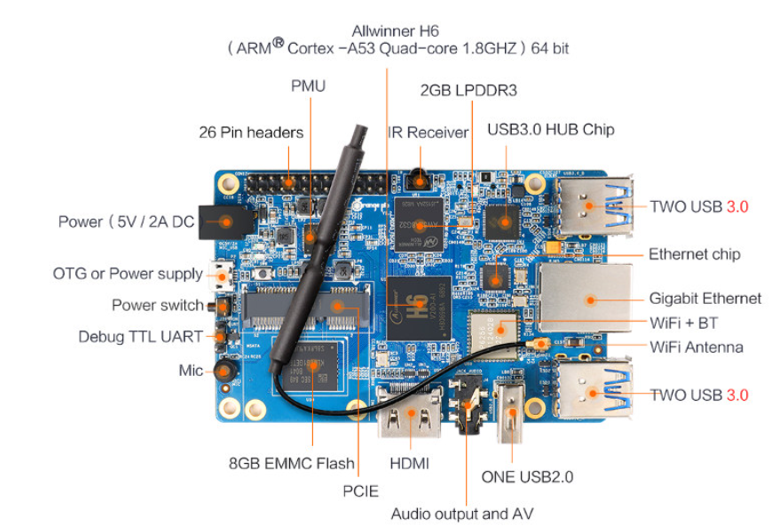
* Thread SAMPLE : thực hiện vô hạn lần nhiệm vụ sau với chu kì X ns. Nhiệm vụ là đọc thời gian hệ thống hiện tại (chính xác đến đơn vị ns) vào biến T.
* Thread INPUT kiểm tra file “freq.txt” để xác định chu kỳ X (của thread SAMPLE) có bị thay đổi không?, nếu có thay đổi thì cập nhật lại chu kỳ X. Người dùng có thể echo giá trị chu kỳ X mong muốn vào file “freq.txt” để thread INPUT cập nhật lại X.
* Thread LOGGING chờ khi biến T được cập nhật mới, thì ghi giá trị biến T và giá trị interval (offset giữa biến T hiện tại và biến T của lần ghi trước) ra file có tên “time\_and\_interval.txt”.

# Chương II : Lý thuyết và triển khai

## 1. Lý thuyết và triển khai hệ thống nhúng trên trên thiết bị (Orange Pi)

### 1.1 Giới thiệu phần cứng (Orange Pi 3)

- Orange Pi là một loại máy tính nhúng được phát triển bởi Shenzhen Xunlong Software CO, có nhiều phiên bản khác nhau, phù hợp vào nhu cầu của người dùng. Nó tích hợp các phần cứng của máy tính như CPU, Memory, RAM, cổng USB, cổng Ethernet, ô thẻ nhớ SD …



Hình 2.1: Orange Pi 3

- Thông số kỹ thuật quan trọng của Orange Pi 3:

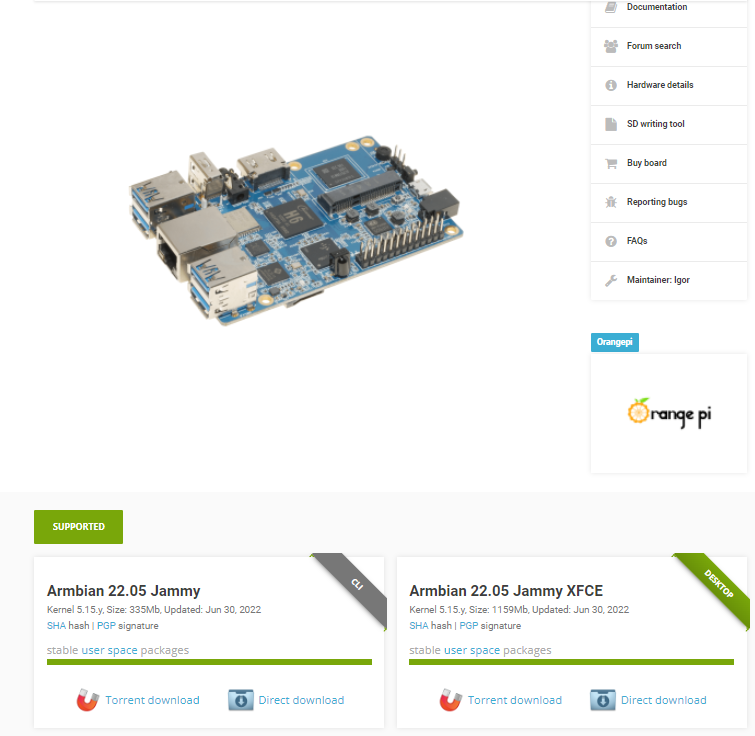
* RAM: 2GB
* Bộ nhớ trong 8GB EMMC Flash
* Chip H6 64bits ARM Cortex A53
* Wifi và Bluetooth chip AP6256 với Wifi IEEE 802.11 a/b/g/n/ac và BT phiên bản 5.0
* 26 Pin headers
* Power 5A/2V DC

### 1.2 Armbian OS cho Orange Pi

#### 1.2.1 Giới thiệu về Armbian OS

- Armbian OS là hệ điều hành Linux dành riêng cho các máy tính chạy chip ARM , dựa trên Debian và Ubuntu và tích hợp môi trường máy tính để bàn nhẹ (XFCE) để có thể cung cấp một hệ thống được tối ưu hóa cho các máy tính hiệu suất thấp, chẳng hạn như Raspberry Pi, Orange Pi …

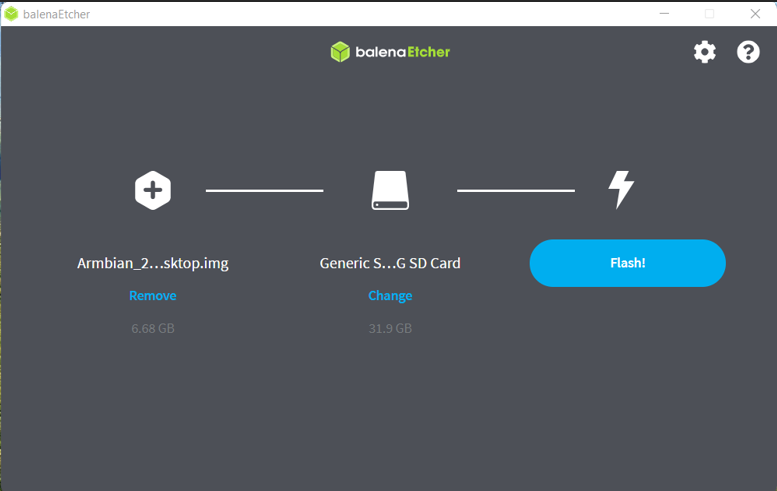
- Mỗi máy tính nhúng khác nhau sẽ có một bản image OS riêng biệt, ta cần boot image lên mạch



Hình 2.2: Armbian OS cho Orange Pi 3

#### 1.2.2 Một số công cụ trong quá trình Boot Armbian lên Orange Pi

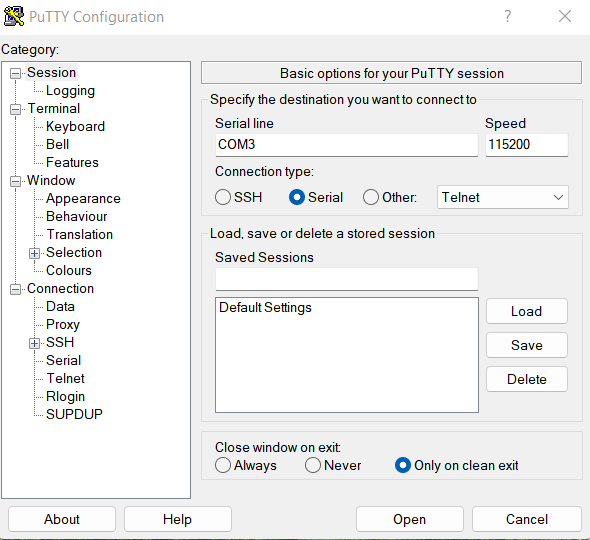
* balenaEtcher
* - Đây là phần mềm giúp ghi dữ liệu file OS .image lên thẻ nhớ.



Hình 2.3: Phần mềm balenaEtcher

* puTTy

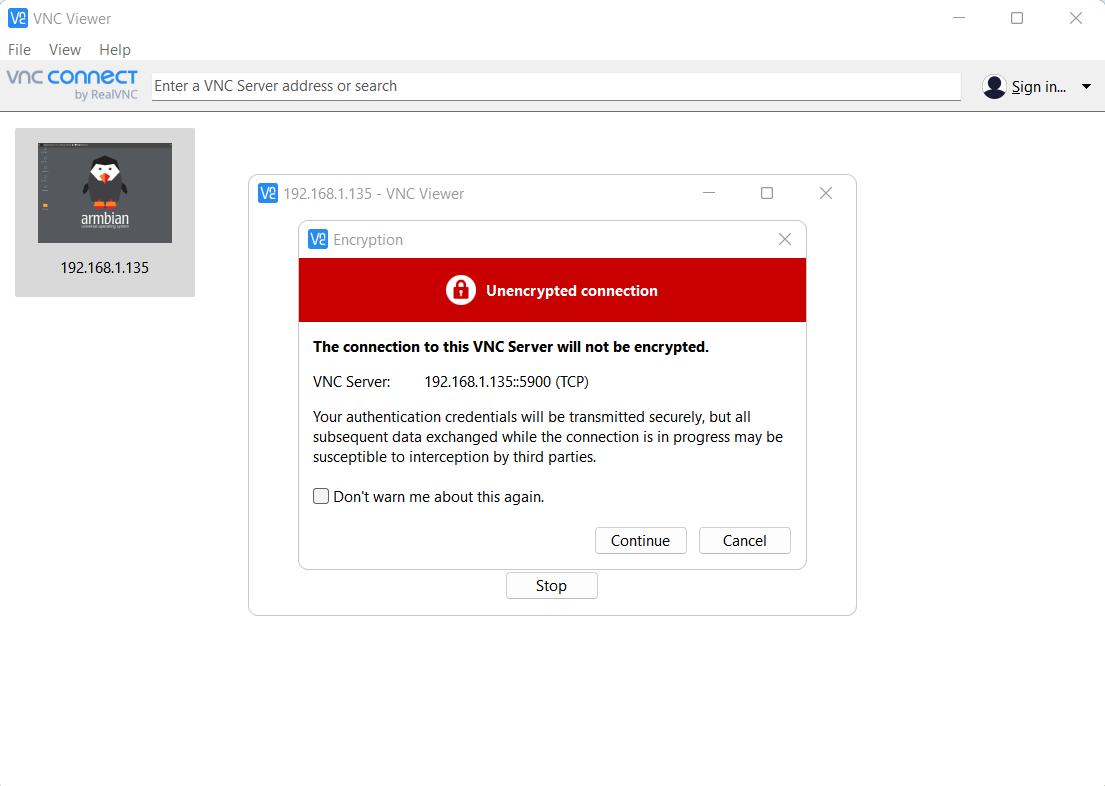
- puTTy giúp kết nối giữa PC (máy tính , laptop) với Orange Pi qua điều khiển SSH (Secure Shell). Với lần đầu tiên kết nối với Pi, bạn có kết nối qua Serial để thiết lập wifi bằng lệnh “nmtui-connect”.



Hình 2.4: Phần mềm puTTy

* VNC Viewer

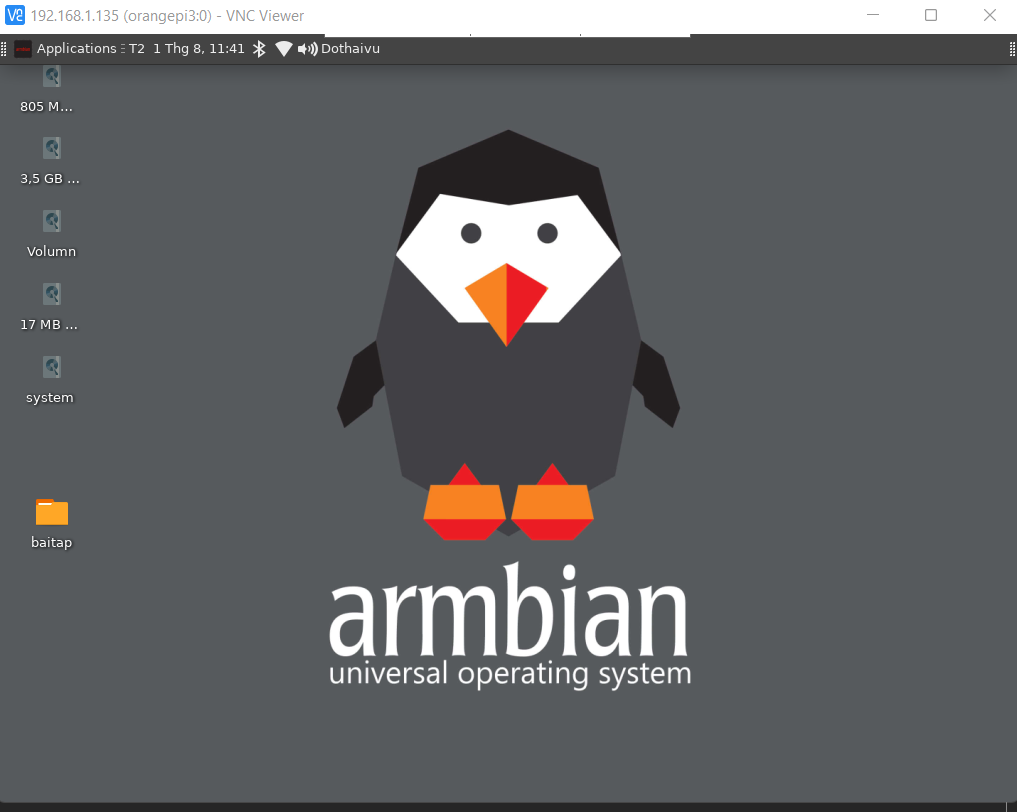
- Đối với một số Armbian OS XFCE, thì VNC Viewer là phần mềm giúp điều khiển máy tính (Armbian OS của Opi) từ xa bằng giao diện màn hình.



Hình 2.5: Phần mềm VNC Viewer

#### 1.2.3 Tiến hành Boot lên Orange Pi

* Bước 1: Sau khi tải được Armbian OS về máy thì ghi file này lên thẻ nhớ SD qua balenaEtcher.
* Bước 2: Sau khi nạp thành công, kết nối PC với OPi qua puTTy bằng SSH. Sử dụng lệnh “ipconfig” để lấy địa chỉ ip để dễ remote các lần sau.
* Bước 3: Đăng nhập lần đầu trong orange pi
* Bước 4: Config VNC Server bằng lệnh “sudo dpkg-reconfigure x11-common”,
* Bước 5: Điều khiển Orange Pi qua VNC Viewer hoặc puTTy với địa chỉ ip mà ta đã tìm ra được trước ở bước 2.



Hình 2.6 GUI của Armbian

#### 1.2.4 Ưu-Nhược điểm của Armbian

- Ưu điểm lớn mà người dùng có được là Orange Pi có chức năng như 1 máy tính thật sự, như giao diện người dùng, khả năng tự biên dịch,… Tuy nhiên đối với một hệ thống tối ưu, một số Application có sẵn trên OS là thừa thãi, khiến vi xử lý có thể chậm hơn.

- Vì vậy, chúng ta có thể tự build một OS phù hợp cho mỗi người dùng khác nhau, trong đó có thể kể đến Yocto Project.

### 1.3 Build Image bằng Yocto Project

#### 1.3.1 Giới thiệu về Yocto Project

- Yocto Project là một dự án mã nguồn mở có mục tiêu là cung cấp các công cụ giúp xây dựng các hệ thống nhúng hoạt động trên hệ điều hành Linux. Yocto Project là được hợp thành từ nhiều dự án mã nguồn mở nhỏ hơn như Poky, BitBake và OpenEmbedded-Core. Trong đó Poky là một bản phân phối tham chiếu - “reference distribution” của Yocto project.

- Một số điểm mạnh của Yocto:

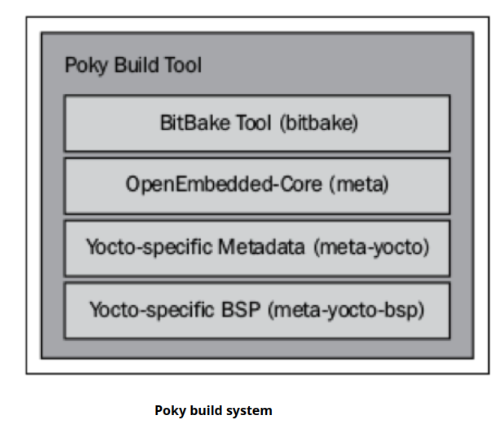
* Thuận tiện và có trợ giúp từ cộng đồng Linux.
* Mã nguồn mở và miễn phí.

- Một vài nhược điểm:

* Một số yocto project đã lỗi phiên bản, hoặc lỗi phiên bản phần mềm liên quan như python.
* Một số board mới xuất hiện thì chưa thể có Yocto Project được.

#### 1.3.2 Poky

- Poky bao gồm OpenEmbedded Build System(BitBake + OpenEmbedded-Core) và một bộ các metadata mặc định có sẵn (bao gồm các recipes, các file cấu hình…) giúp chúng ta có thể bắt đầu build các Linux software packages của riêng mình.



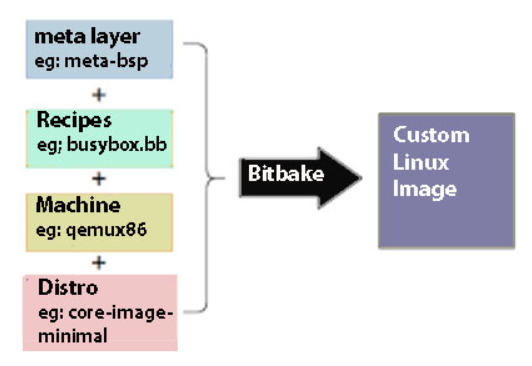
HÌnh 2.7: Poky

- Các Linux software packages có thể build bao gồm:

* Bootloader
* Linux kernel
* Root filesystems
* Toolchains & Software Development Kits (SDKs)

#### 1.3.3 Bitbake

- Bitbake là một công cụ cốt lõi của Yocto project được viết bằng ngôn ngữ python. Nó là một task runner(công cụ thực hiện các task) với tasks ở đây là các hàm mà Bitbake cần chạy theo thứ tự để thực hiện các recipe (công thức tạo phần mềm) cần thực hiện.



Hình 2.8: Bitbake

- Bitbake thực hiện các loại metadata sau.

* Recipes: files có đuôi ".bb" chứa các thông tin về việc build một phần mềm, nó bao gồm việc tải source code ở đâu, làm sao để build và cài đặt phần mềm đó.
* Append: files có đuôi ".bbappend" chứa thêm các lệnh cho một file Recipes. files Append cho ta khả năng bảo trì nâng cấp phần mềm dễ dàng hơn.
* Include: files có đuôi ".inc" chứa thông tin dùng chung của các file Recipes cho phép việc chia sẻ thông tin giữa các files Recipes.
* Classes: files có đuôi "bbclass" chứa các thông tin cơ bản về việc build phần mềm như nào.
* Configuration: files có đuôi ".conf" định nghĩa các biến cấu hình dùng trong các files Recipes khác.

#### 1.3.4 Build Image cho Orange Pi bằng Yocto Project

- Sau khi tìm hiểu về Yocto Project, ta tiến hành build Image cho Opi:

* Bước 1: Cài đặt môi trường và Dependencies:

sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib \ build-essential chrpath socat cpio python3 python3-pip python3-pexpect \xz-utils debianutils iputils-ping python3-git python3-jinja2 libegl1-mesa \ libsdl1.2-dev pylint3 xterm

* Bước 2: Clone Yocto

git clone https://github.com/Halolo/orange-pi-distro.git

* Bước 3: Sao chép các module bên ngoài và cập nhật

git submodule update --init

* Bước 4: Chạy . source-me <machine>

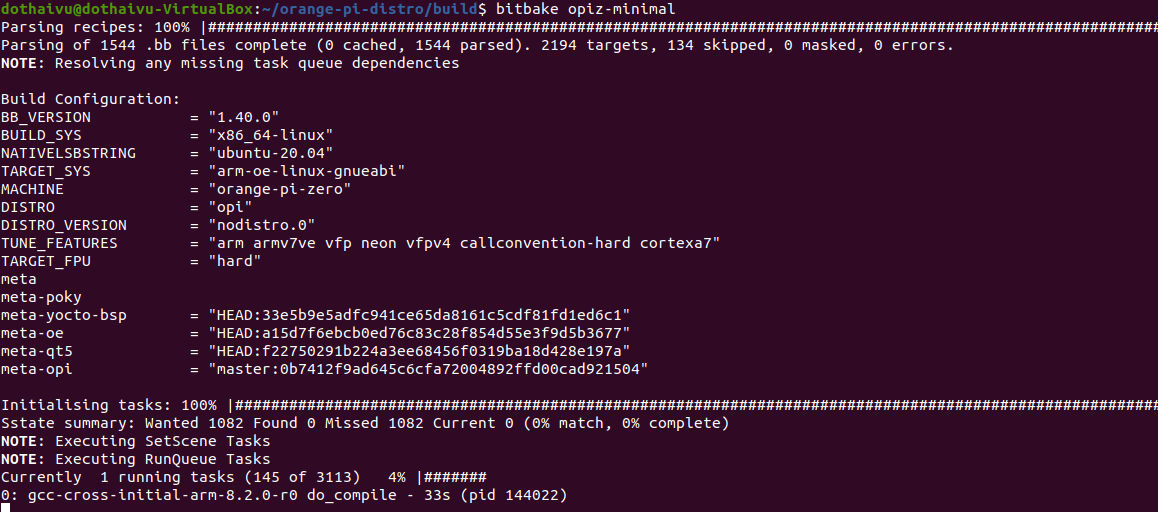
. source-me orange-pi-zero

* Bước 5: Build bằng bitbake<target>

bitbake opiz-minimal

* Bước 6: Ghi nó lên thẻ nhớ

sudo .flash-sdcard.sh -d dev/sdc -m orange-pi-zero -i opiz-minimal



Hình 2.9 Build Image bằng Yocto

#### 1.3.5 Build Image cho Orange Pi bằng Xunlong github

- Một số mạch Orange Pi đời mới chưa có Yocto Project, thì có thể tự build image bằng cách clone từ Xunlong – Công ty sản xuất board Orange Pi

- Sau đây là cách build Image cho Orange Pi 3:

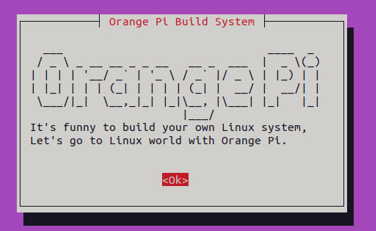
* Bước 1: Clone source code trên github Xunlong, chọn bản thích hợp với môi trường. Ở đây, tôi chọn bản phù hợp với môi trường Ubuntu 22 LTS

git clone -b next https://github.com/orangepi-xunlong/orangepi-build.git

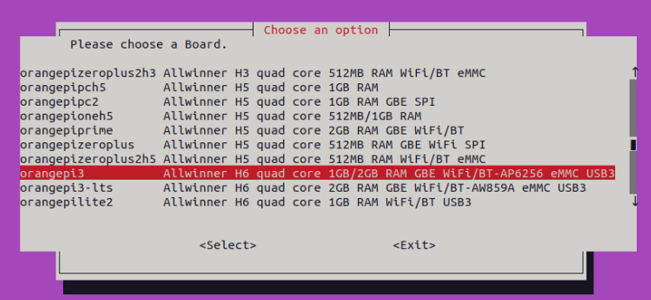
* Bước 2: Chạy file build.sh

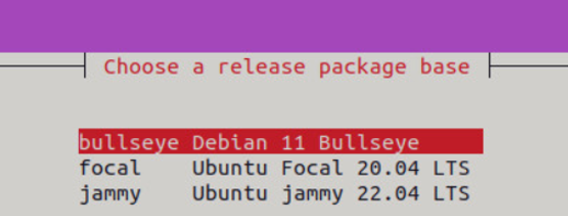
cd orangepi-build

./build.sh



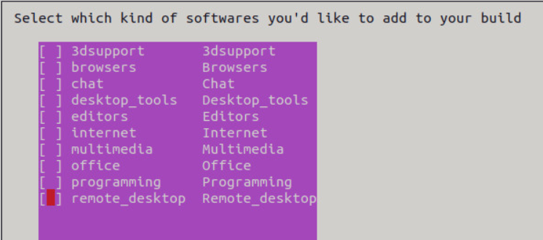
* Bước 3: Lựa chọn cấu hình của KIT và môi trường cài đặt



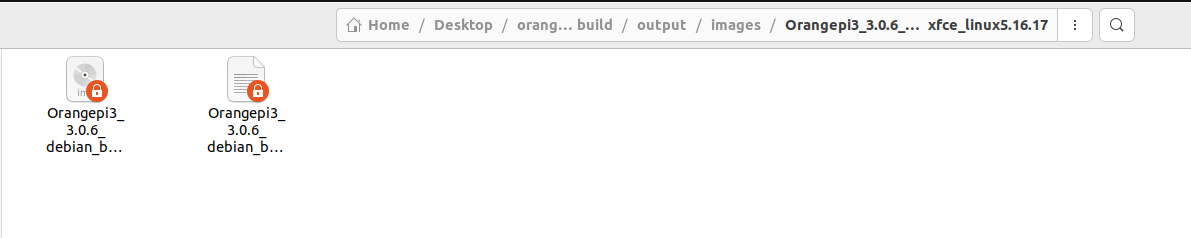


Hình 2.10: Lựa chọn cấu hình và môi trường cho OPi

* Bước 4: Chọn các application bạn muốn build trong Image



* Bước 5: Kiểm tra Output và ghi lên thẻ nhớ SD



Hình 2.11: Output khi build bằng Xunlong github

## 2. Lý thuyết về Kernel Real-Time

- Kernel real-time là một bản vá kernel (kernel patch) mà nó bao gồm các công cụ cần thiết để tạo được việc hoàn thành trước deadline của một task yêu cầu real-time.

- Nó tối ưu thiết kế kernel để đạt được thời gian trễ thấp nhất, thời gian đáp ứng của hệ thống cũng như tạo ra sự xác định(determinism) của hệ thống. Kernel real-time cho ta hoàn toàn quyền điều khiển việc thứ tự thực hiện của các thread thông qua bộ lập lịch.

- Ở kernel real-time các task mà đã được lập lịch có độ ưu tiên cao thì sẽ chắc chắn được thực hiện trước những task khác nói cách khác là có quyền preemptive điều này có thể sẽ không được thực hiện ở kernel thường.

- Các vấn đề liên quan về lập lịch

* Scheduling (lập lịch)
* Scheduling latency (lập lịch độ trễ)
* Priority (độ ưu tiên)

### 2.1 Scheduling

- Lập lịch có hai tính chất chính là tính công bằng (fairness) và xác định (determinism).

+ Tính công bằng là mỗi thread sẽ có một khoảng thời gian thực hiện bằng nhau.

+ Tính xác định là các thread sẽ được thực hiện dựa trên độ ưu tiên, và sẽ “mạnh” hơn tính công bằng.

+ Một thread khi được tạo sẽ được gán một schedule policy thuộc timeshare policy hoặc real-time policy.

#### 2.1.1 Timeshare Policy

Các Timeshare Policy mang tính chất công bằng (fairness). Một số policy là:

* SCHED\_NORMAL hoặc (SCHED\_OTHER): khi tạo một thread thì đây sẽ là policy mặc định cho thread.
* SCHED\_BATCH: Giống với SCHED\_NORMAL chỉ khác chỗ thread sẽ chạy lâu hơn đồng thời cũng phải chờ lâu hơn khi muốn thực hiện lại thread đó.
* SCHED\_IDLE: chỉ chạy thread khi không có thread nào đang chạy.

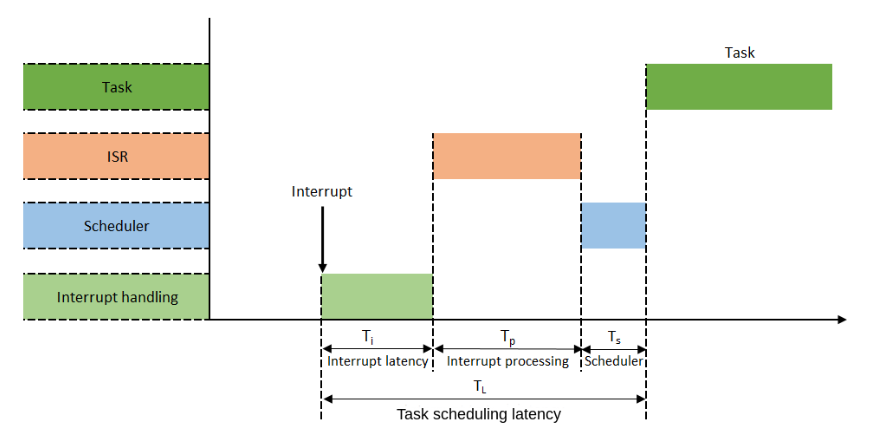
#### 2.1.2 Real-Time Policy

Các real-time policy có mục đích tạo ra tính xác định. Thread Real-time-policy luôn luôn thực hiện trước các thread timeshare. Để tạo ra code real-time thì ta phải sử dụng các policy lập lịch mang tính real-time như là:

* SCHED\_FIFO: sử dụng thuật toán First In First Out có nghĩa là thực hiện các thread có độ ưu tiên cao tới các thread có độ ưu tiên thấp, thread nào có cùng độ ưu tiên thì thread nào đến trước thì sẽ được thực hiện trước.
* SCHED\_RR: sử dụng thuật toán Round Robin, thuật toán này có nghĩa là bộ lập lịch sẽ thực hiện các thread có cùng độ ưu tiên theo thứ tự xoay vòng, mỗi thread sẽ được thực hiện trong một khoảng thời gian nhất định được xác định trước (mặc định là 100ms).

### 2.2 Scheduling latency

Các thread cần phải được lên lịch ngay khi nó phải thực hiện một công việc nào đó. Tuy nhiên vẫn có khả năng xảy ra độ trễ tại thời điểm đánh thức bộ đếm thời gian của hệ thống đến thời điểm luồng bắt đầu chạy.



Hình 2.12: Scheduling latency

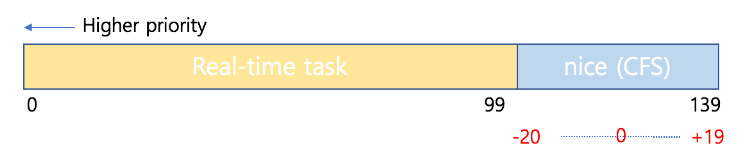
### 2.3 Priority

- Lập lịch theo chỉ số ưu tiên của tiến trình (priority-base) là một phương pháp phổ biến ở các hệ điều hành, bao gồm cả linux. Về cơ bản, thread có độ ưu tiên cao hơn thì sẽ được thực hiện trước, nếu độ ưu tiên bằng nhau cần được lập lịch để chia sẻ thời gian với nhau tùy vào chính sách lập lịch của mỗi tiến trình.

- Hệ điều hành sẽ cung cấp 2 loại priority: nice value (cho normal process) và real-time priority (cho real-time process).

* Nice value : Có giá trị từ -20 đến +19, độ lớn càng nhỏ thì time slice (thời gian cố định được cung cấp cho 1 tiến trình) càng lớn.
* Real-time priority : Có giá trị từ 0 đến 99, 0 là giá trj có độ ưu tiên lớn nhất.

- Hình 2.13 dưới đây thể hiện real-time task luôn có hệ số priority ưu tiên để được lập lịch trước so với normal task.



Hình 2.13: Priority và nice value

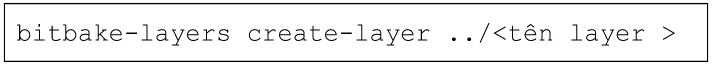
## 3. Xây dựng Kernel Real-Time cho Orange Pi

* Bước 1: ta phải tìm bản patch real-time phù hợp với phiên bản kernel thường đã build từ trước, có thể tìm [ở đây](https://mirrors.edge.kernel.org/pub/linux/kernel/projects/rt/).

[svtt2.VHT-SVTT2] ➤ scp patch-4.19-rt1.patch [dothaivu@192.167.120.236](mailto:dothaivu@192.167.120.236):

dothaivu@192.167.120.236's password:

* Bước 2: Sau khi tạo poky, tạo 1 layer



lưu ý tên layer nên bắt đầu với "meta-..."



[dothaivu@SRV236-REDHAT build]$ bitbake-layers create-layer ../meta-mylayer

NOTE: Starting bitbake server...

Add your new layer with 'bitbake-layers add-layer ../meta-mylayer'

[dothaivu@SRV236-REDHAT build]$ bitbake-layers add-layer ../meta-mylayer

* Bước 3: Chuyển meta-mylayer về poky

[dothaivu@SRV236-REDHAT orange-pi-distro]$ ls

build \_config.yml COPYING.MIT flash-sdcard.sh meta-mylayer meta-openembedded meta-opi meta-qt5 poky README.md source-me

[dothaivu@SRV236-REDHAT orange-pi-distro]$ cp -r ./meta-mylayer/ ./poky/

* Bước 4: Thực hiện thêm đường dẫn layer vừa tạo vào file bblayer.conf

[dothaivu@SRV236-REDHAT orange-pi-distro]$ cd build/conf/

[dothaivu@SRV236-REDHAT conf]$ ls

bblayers.conf local.conf sanity\_info templateconf.cfg

[dothaivu@SRV236-REDHAT conf]$ vi bblayers.conf

Add đường dẫn layer vào , ta có nội dung của file bblayer.conf

# POKY\_BBLAYERS\_CONF\_VERSION is increased each time build/conf/bblayers.conf

# changes incompatibly

POKY\_BBLAYERS\_CONF\_VERSION = "2"

BBPATH = "${TOPDIR}"

BBFILES ?= ""

BBLAYERS ?= " \

/home/dothaivu/orange-pi-distro/poky/meta \

/home/dothaivu/orange-pi-distro/poky/meta-poky \

/home/dothaivu/orange-pi-distro/poky/meta-yocto-bsp \

/home/dothaivu/orange-pi-distro/poky/meta-mylayer \

"

# OPI

BBLAYERS += " \

/home/dothaivu/orange-pi-distro/meta-openembedded/meta-oe \

/home/dothaivu/orange-pi-distro/meta-qt5 \

/home/dothaivu/orange-pi-distro/meta-opi \

"

- Set up to the Build

+ Tìm đến file recipes-kernel để chứa file patch-4.19.rt1.patch

[dothaivu@SRV236-REDHAT]$ cp patch-4.19-rt1.patch orange-pi-distro/meta-opi/recipes-kernel/linux/linux-opi/linux-opiz

+ Di chuyển đến meta-opi/conf để tạo file layer.conf

[dothaivu@SRV236-REDHAT]$ cd orange-pi-distro/meta-opi/conf/

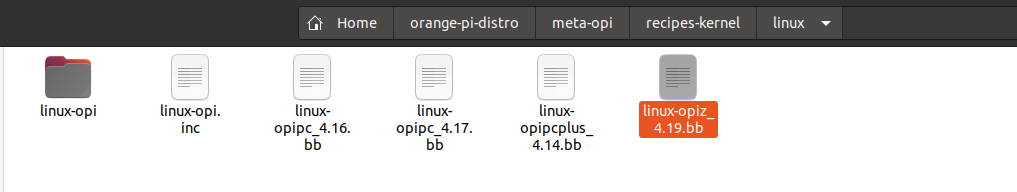
[dothaivu@SRV236-REDHAT conf]$ ls

layer.conf

[dothaivu@SRV236-REDHAT conf]$ vi layer.conf

- Ở đây do đã build kernel thường trước đó nên ta không cần phải chỉnh sửa thêm file layer.conf nữa.

- Chuyển đến thư mục meta-opi/recipes-kernel/linux/ để chỉnh sửa file linux-opiz\_4.19.bb như sau:

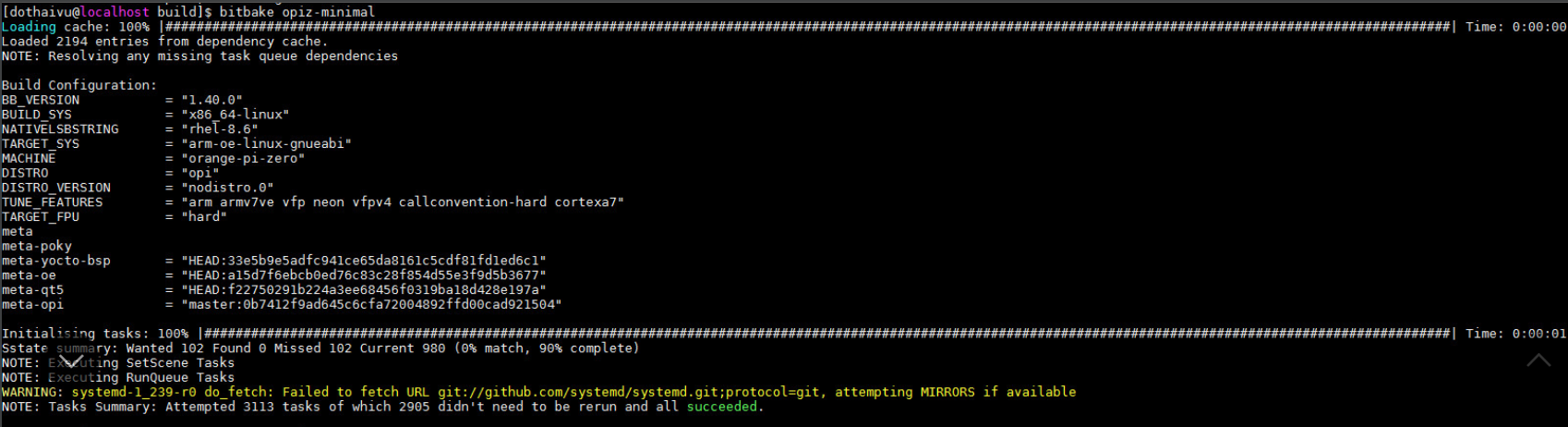


FILESEXTRAPATHS\_prepend := "${THISDIR}/${PN}:"

SRC\_URI += <file://patch-4.19-rt1.patch>

PRINC := "${@int(PRINC) + 1}"

- Cuối cùng thực hiện bitbake opiz-minimal lại lần nữa.



Hình 2.14: Build Kernel Real-Time cho OPi

## 4. Lý thuyết C Programing

### 4.1 Multithread, sleep, mutex

#### 4.1.1 Multithread

- Giống như các quy trình, các luồng là một cơ chế cho phép ứng dụng thực hiện nhiều nhiệm vụ đồng thời. Một quy trình duy nhất có thể chứa nhiều luồng.

- Tất cả các luồng này đều thực thi độc lập giống nhau và tất cả chúng đều chia sẻ cùng một bộ nhớ chung, bao gồm cả dữ liệu được khởi tạo, dữ liệu chưa khởi tạo và phân đoạn đống.

- Multithread có thể chia tiến trình giống hàm fork(), nhưng nó có ưu điểm:

* Có thể chia sẻ thông tin giữa các thread (luồng) nhanh và dễ.
* Thread creation nhanh hơn process creation-typically 10 lần hoặc tốt hơn.

- Các thao tác của multithread tương ứng với các câu lệnh phổ biến dưới đây

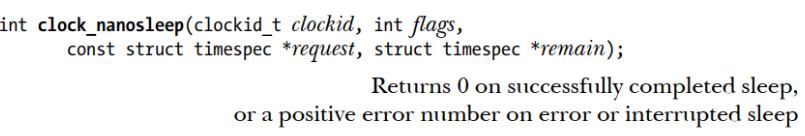
* pthread\_create(&thread,NULL,func,&arg)
* pthread\_exit(void \*retval)
* pthread\_cancel()
* pthread\_self(void)
* pthread\_join(&thread,void \*\*retval)

#### 4.1.2 Timer and Sleep

- Một vài lệnh thiết lập thời gian trong đồng hồ POSIX

* clock\_gettime () :lấy giá trị hiện tại của đồng hồ;
* clock\_getres () : trả về độ phân giải của đồng hồ;
* clock\_settime () : cập nhật đồng hồ.

- Trong chương trình C, có nhiều cách để đưa tiến trình vào trạng thái sleep nhưng khi tín hiệu phân phối ở tốc độ cao thì ta có sử dụng lệnh clock\_nanosleep() để độ chia nhỏ nhất thêm chính xác.



Hình 2.15: Hàm clock\_nanosleep

#### 4.1.3 Mutex

- Để đồng bộ hóa các Thread, tránh xung đột khi các thread cùng truy cập đến condition variable tại một thời điểm, ta sử dụng khóa Mutex.

- Mutex được hiểu là đối tượng (hoặc cờ hiệu) mang 2 trạng thái là đang được sử dụng và chưa được sử dụng ( trạng thái sẵn sàng).

- Khi đó thread có toàn quyền truy cập và thiết lập giá trị cho các biến tại một thời điểm.

- Dưới đây là một số thao tác cần biết khi làm việc với mutex

* Cách khai báo mutex cấp tốc :

Pthread\_mutex\_t a\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

+ Lưu ý : Đối tượng mutex này không thể bị khóa 2 lần bởi cùng 1 luồng. Trong luồng nếu bạn đã gọi hàm khóa mutex này và thực hiện khóa mutex 1 lần nữa, bạn sẽ rơi vào trạng thái khóa chết (deadlock).

* Khóa

int rc = pthread\_mutex \_lock( & a\_mutex);

rc = 1 thì có lỗi ; rc = 0 thì khóa lại thành công.

+ Mở khóa để trả lại tài nguyên cho luồng khác.

* Tháo khóa

int rc = pthread\_mutex \_unlock( & a\_mutex);

rc = 1 thì có lỗi ; rc = 0 thì mở khóa lại thành công.

* Hủy mutex ( nên hủy sau khi dùng xong)

rc = pthrea\_mutex\_destroy (& a\_mutex)

### 4.2 Tối ưu hóa chương trình bằng phương pháp lập lịch

- Như đã nêu ở phần 2.1 về Schuduling, ta sẽ thiết lập độ ưu tiên priority trong các thread khác nhau.

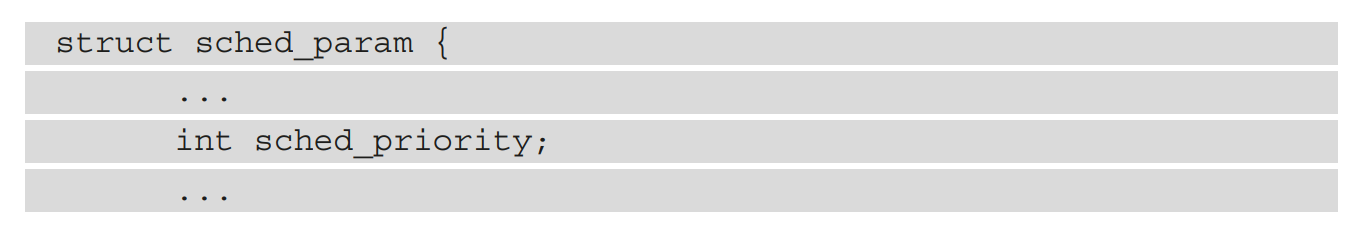
- Code tối ưu

struct sched\_param sp;

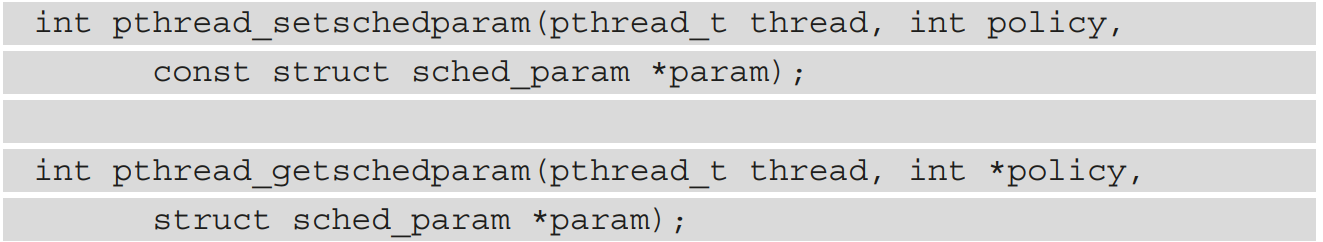
sp.sched\_priority = 10;

pthread\_setschedparam(pthread\_self(), SCHED\_FIFO, &sp);

Trong đó

* Sched\_param là một struct mà 
* sched\_priority có thể hiểu là một int thể hiện mức độ ưu tiên như đã nói ở trên.

- Lệnh pthread\_setschedparam(pthread\_self(), SCHED\_FIFO, &sp); dùng để set policy cho thread:



Trong đó :

+ pthread\_self() chỉ về chr chính nó.

+ SCHED\_FIFO là 1 policy đã nói ở phần trên

+&sp trỏ về sched\_param mà ta vừa tạo.

### 4.3 Triển khai

4.3.1 Các Thread trong chương trình

* Thread get\_time

void \*getTime(void \*args)

{ struct sched\_param sp;

sp.sched\_priority = 70;

pthread\_setschedparam(pthread\_self(), SCHED\_FIFO, &sp);

clock\_gettime(CLOCK\_MONOTONIC, &request);

while (1)

{

request.tv\_nsec += freq;

if(request.tv\_nsec > 1000000000) {

request.tv\_nsec -= 1000\*1000\*1000;

request.tv\_sec++;

}

clock\_nanosleep(CLOCK\_MONOTONIC, TIMER\_ABSTIME, &request, NULL);

clock\_gettime(CLOCK\_MONOTONIC, &tp);

}

return NULL;}

* Thread get\_freq

void \*getFreq(void \*args)

{ struct sched\_param sp;

sp.sched\_priority = 10;

pthread\_setschedparam(pthread\_self(), SCHED\_FIFO, &sp);

FILE \*fp;

fp = fopen("freq.txt","r");

unsigned long new\_freq = 0;

fscanf(fp,"%lu",&new\_freq);

while(1)

{

//pthread\_mutex\_lock(&mutex);

if(new\_freq != freq)

{

freq = new\_freq;

}

}

fclose(fp);

}

* Thread save\_time

void \*save\_time(void \*args)

{

/\* struct sched\_param sp;

sp.sched\_priority = 40;

pthread\_setschedparam(pthread\_self(), SCHED\_FIFO, &sp);\*/

FILE \*file ;

file = fopen("time\_and\_interval.txt","a+");

//save time

while(1)

{

long diff\_sec = tp.tv\_sec - tmp.tv\_sec ;

long diff\_nsec;

if(tp.tv\_nsec != tmp.tv\_nsec || tp.tv\_sec != tmp.tv\_sec)

{

if(tp.tv\_nsec > tmp.tv\_nsec)

{

diff\_nsec = tp.tv\_nsec - tmp.tv\_nsec;

}

else

{

diff\_nsec = 1000000000 + tp.tv\_nsec - tmp.tv\_nsec ;

diff\_sec = diff\_sec - 1;

}

fprintf(file,"\n%ld",diff\_nsec);

tmp.tv\_nsec =tp.tv\_nsec;

tmp.tv\_sec = tp.tv\_sec;

}

}

fclose(file);

return NULL;

}

# CHƯƠNG III: KẾT QUẢ VÀ NHẬN XÉT KẾT QUẢ

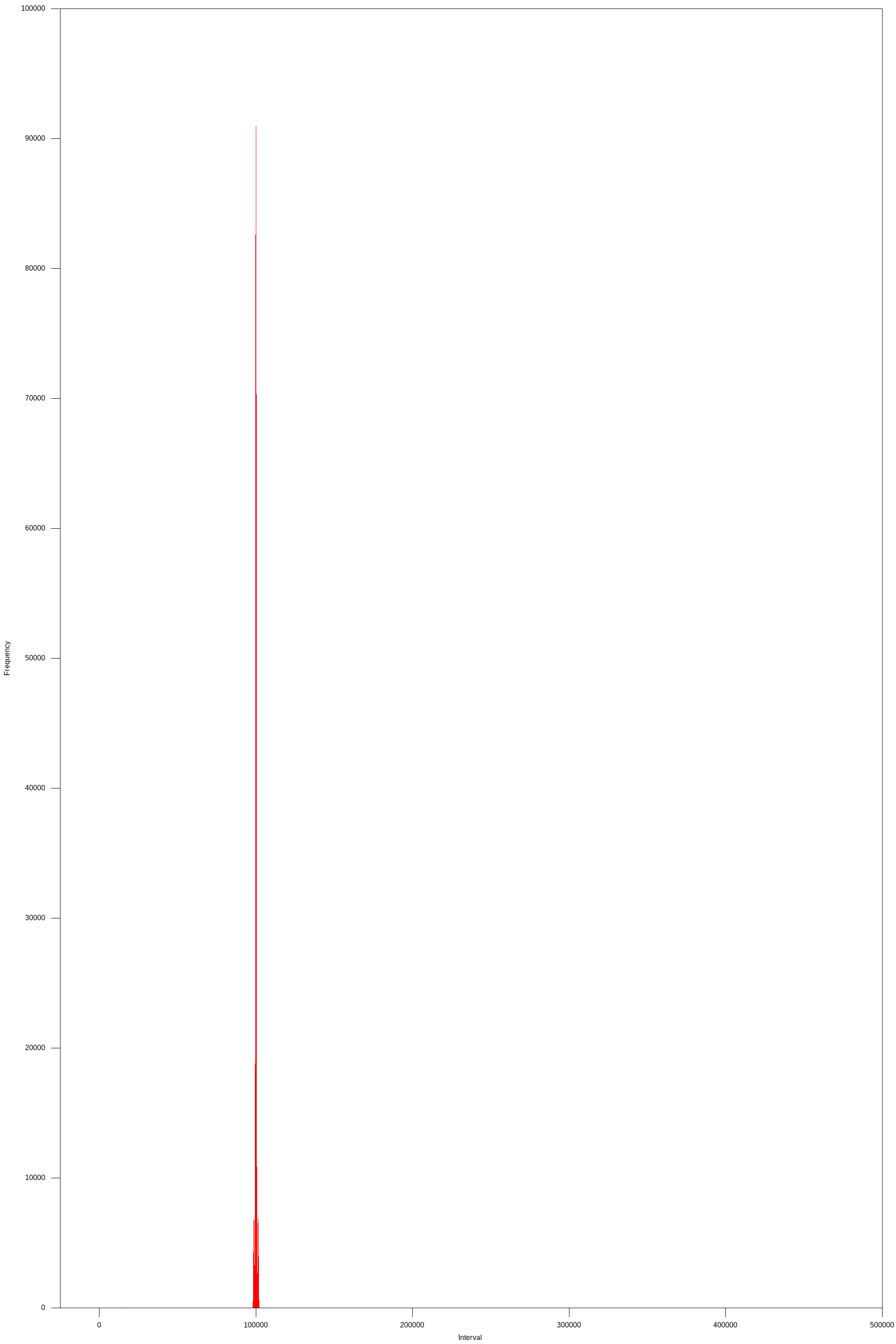
## 1. Thực hiện chạy kết quả

### 1.1 Trường hợp 1

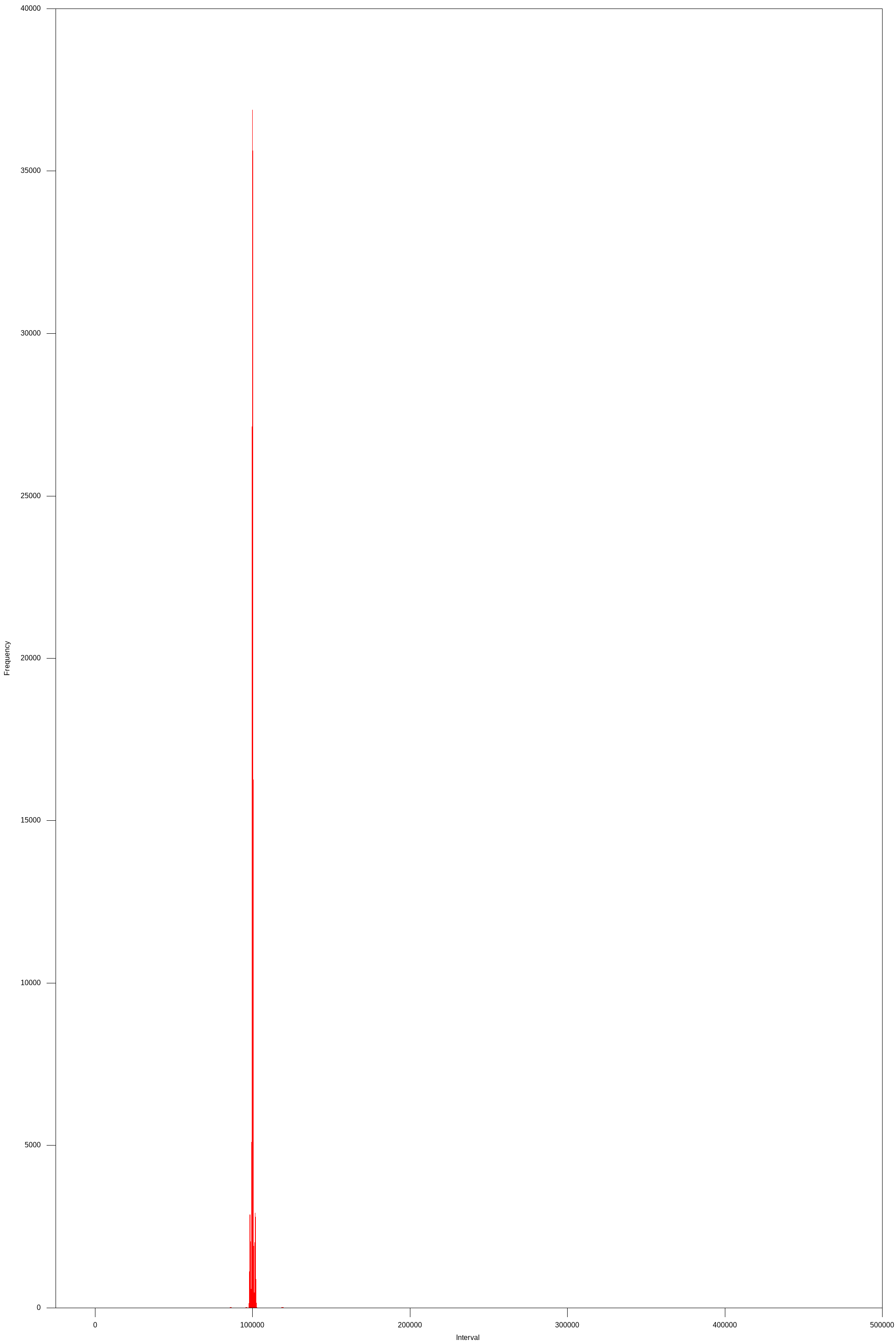
1.1.1. Mô tả

* Chạy file C code thường trên Kernel thường và Kernel real-time, không kèm theo file disturb
* Chu kỳ lấy mẫu: 100000ns

1.1.2 Kết quả



Hình 3.1: Kernel thường, code thường, không có disturb



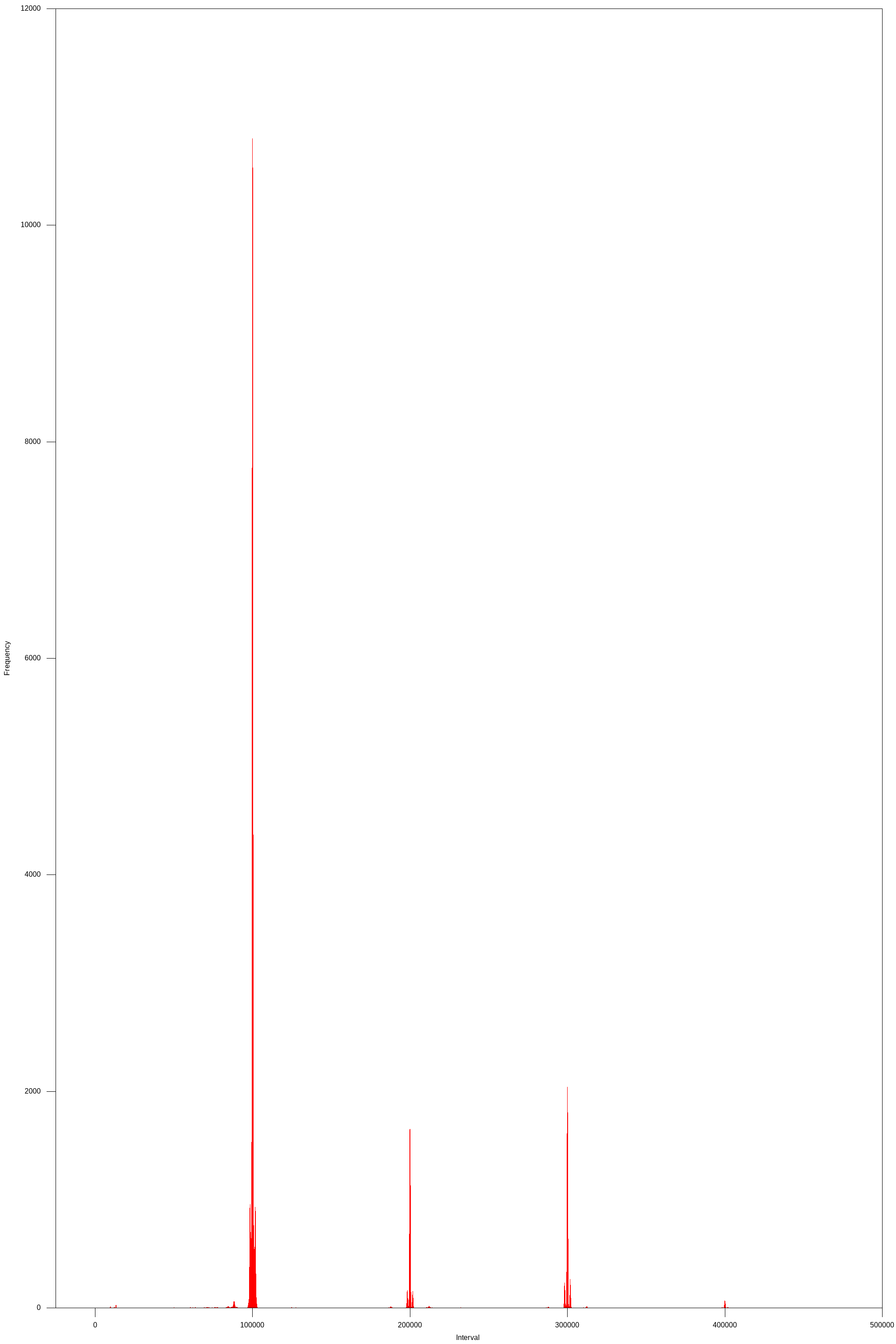
Hình 3.2: Kernel Realtime, code thường, không có disturb

### 1.2 Trường hợp 2

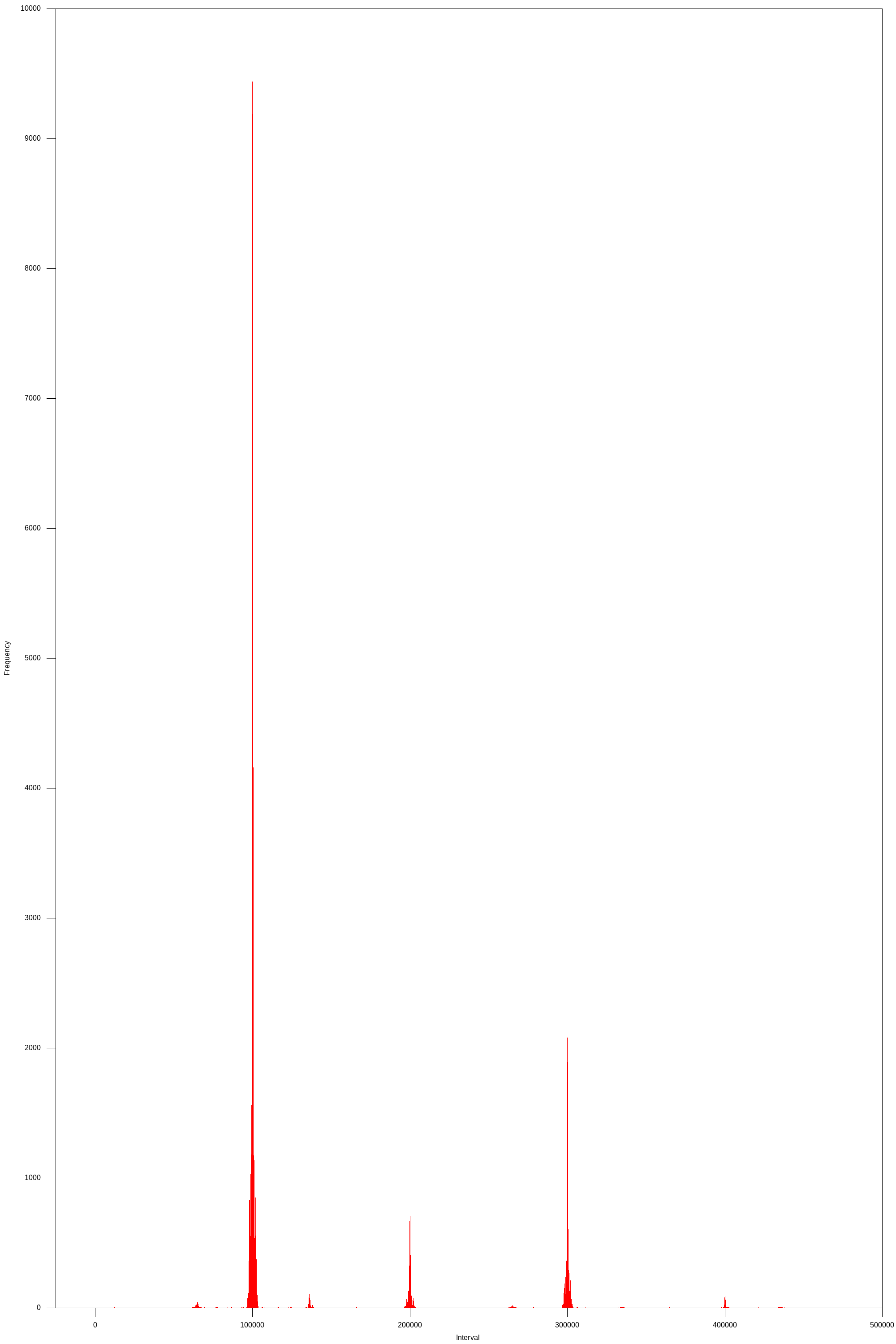
1.2.1 Mô tả

* Chạy file C code thường trên Kernel thường và Kernel real-time, có kèm theo file disturb 10
* Chu kỳ lấy mẫu: 100000ns

1.2.2 Kết quả



Hình 3.3: Kernel thường, code thường, có disturb 10



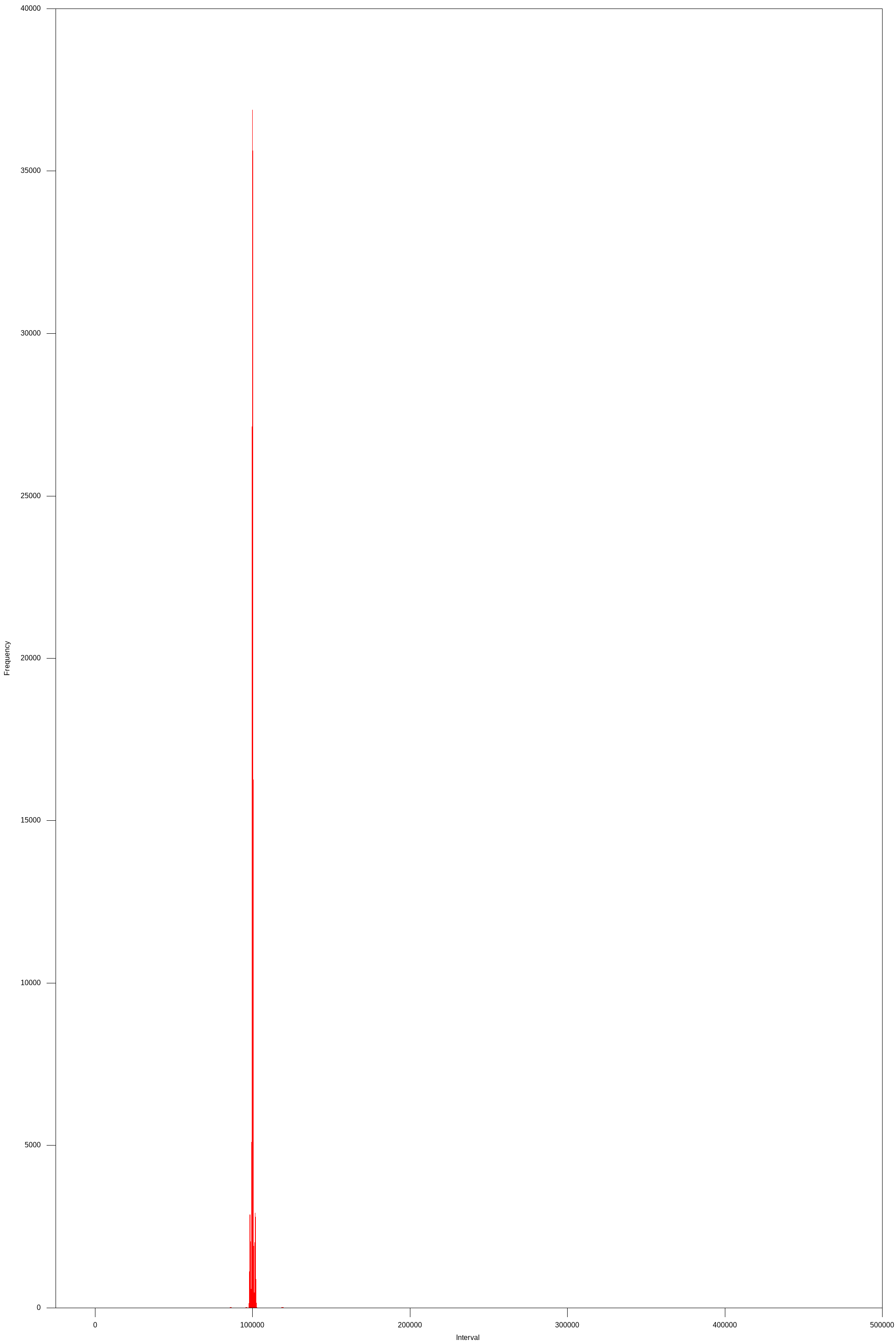
Hình 3.4: Kernel Realtime, code thường, có disturb 10

### 1.3 Trường hợp 3

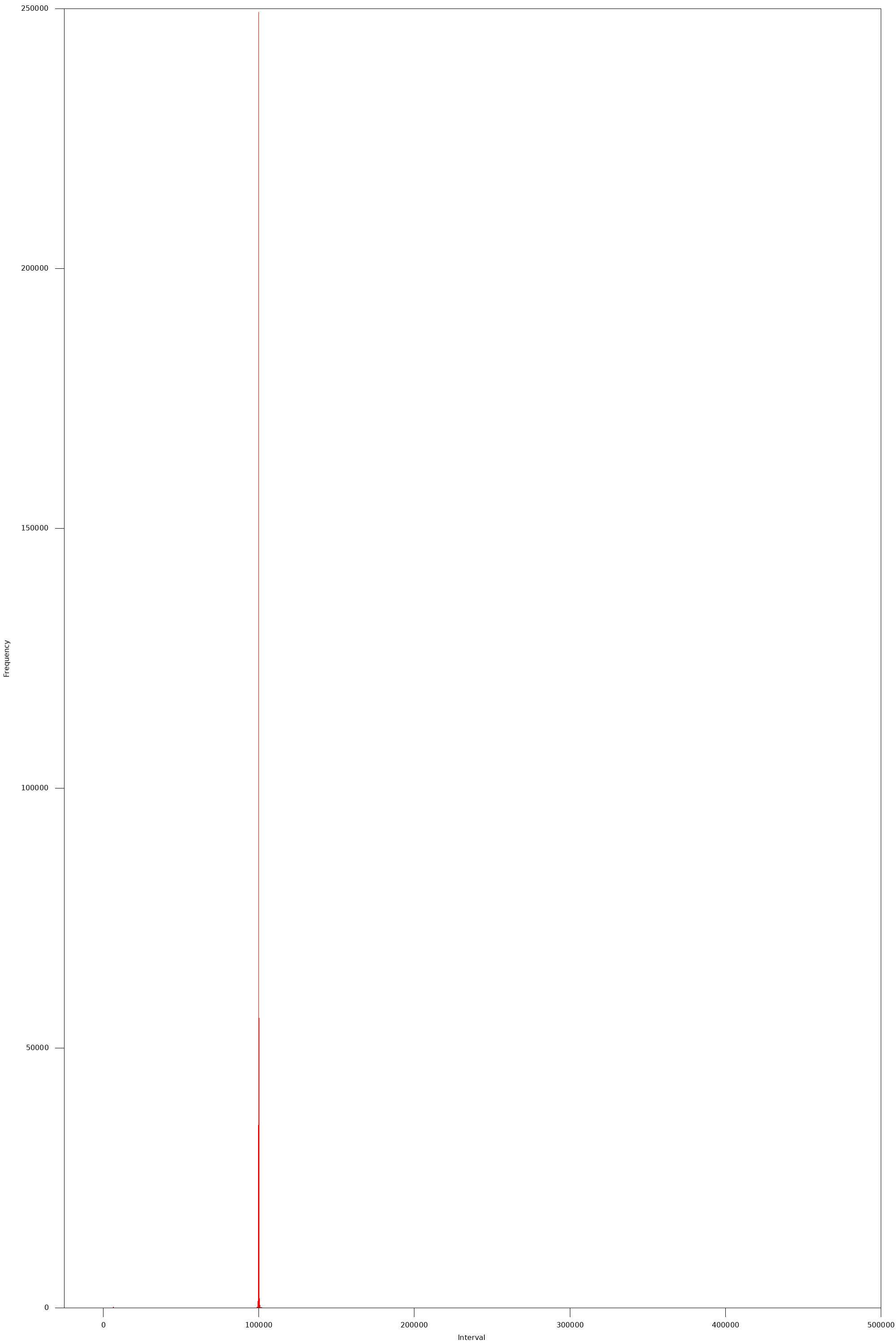
1.3.1 Mô tả

* Chạy file C code tối ưu trên Kernel thường và Kernel real-time,không có kèm theo file disturb
* Chu kỳ lấy mẫu: 100000ns

1.3.2 Kết quả



Hình 3.5: Kernel thường, code tối ưu( lập lịch cả 3 thread), có disturb 10



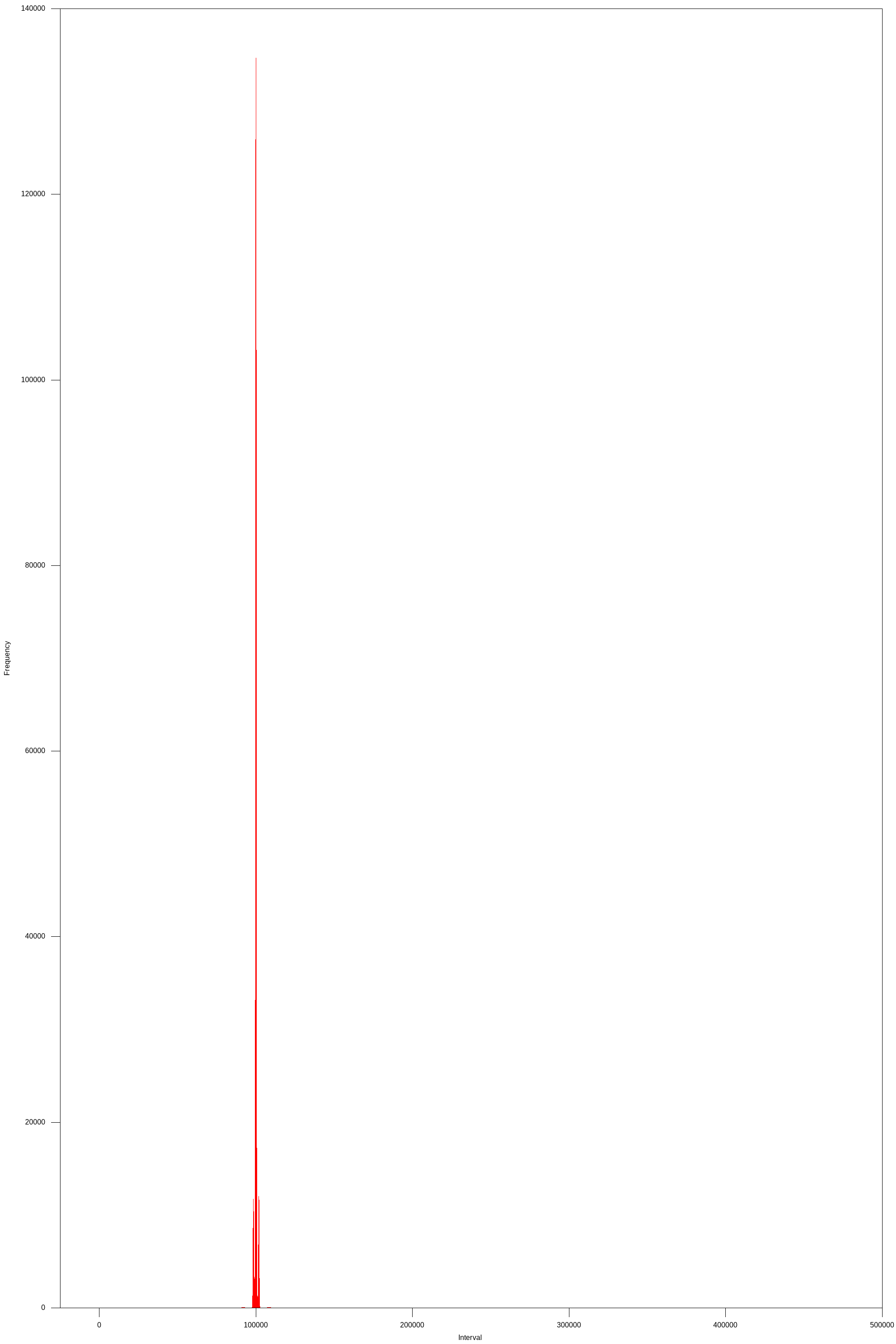
Hình 3.6: Kernel Realtime, code tối ưu (lập lịch cho 3 thread), không disturb

### 1.4 Trường hợp 4

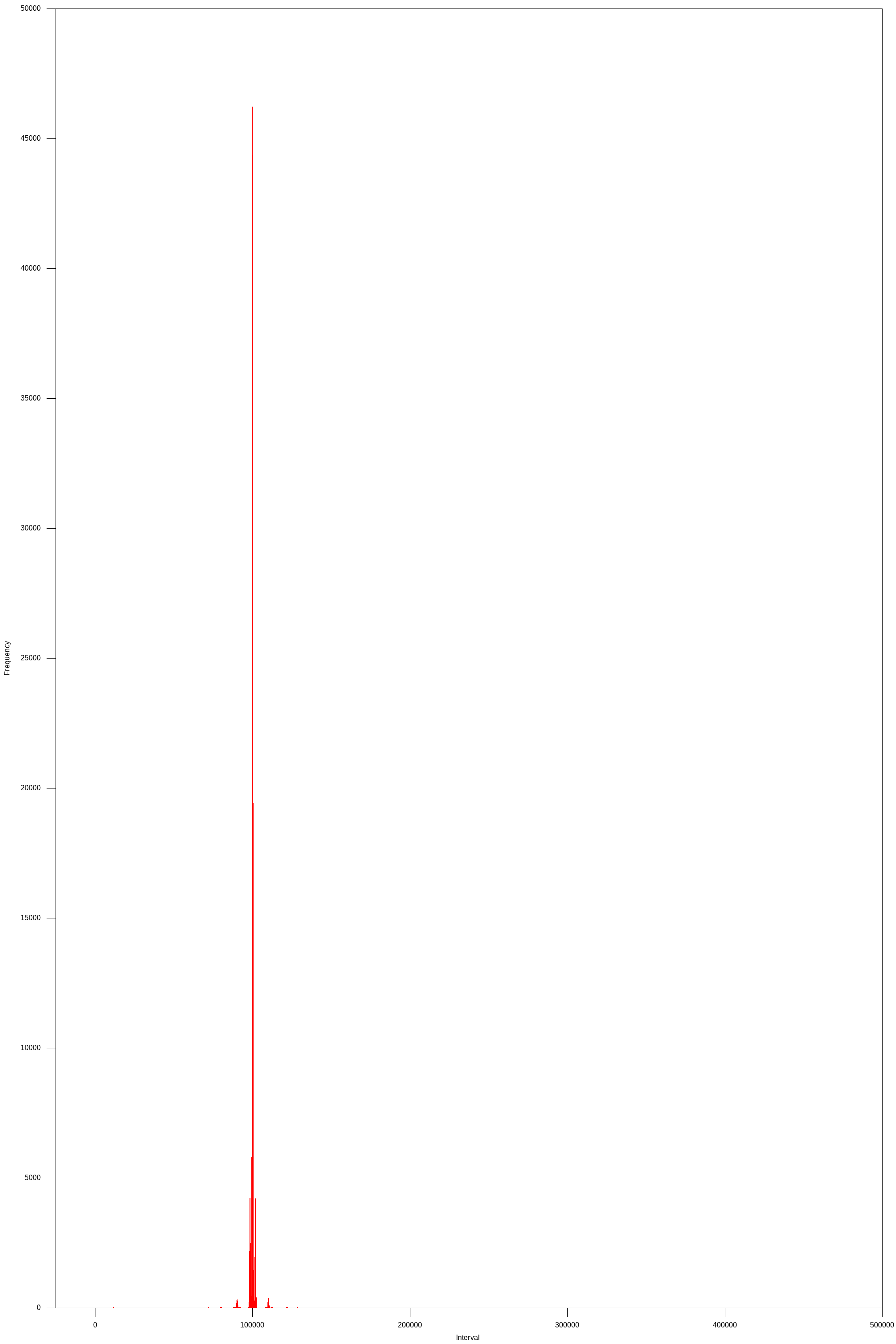
1.4.1 Mô tả

* Chạy file C code tối ưu trên Kernel thường và Kernel real-time, có kèm theo file disturb 10
* Chu kỳ lấy mẫu: 100000ns

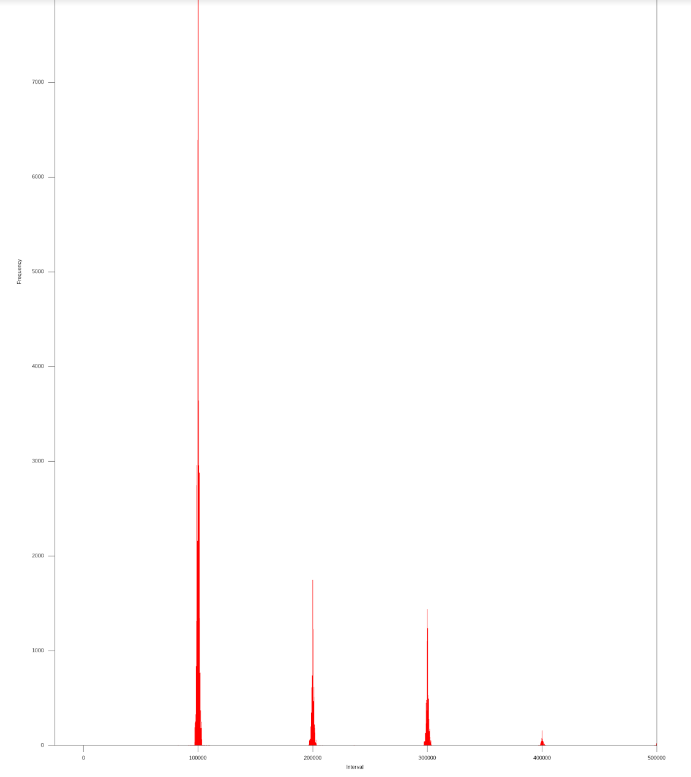
1.4.2 Kết quả



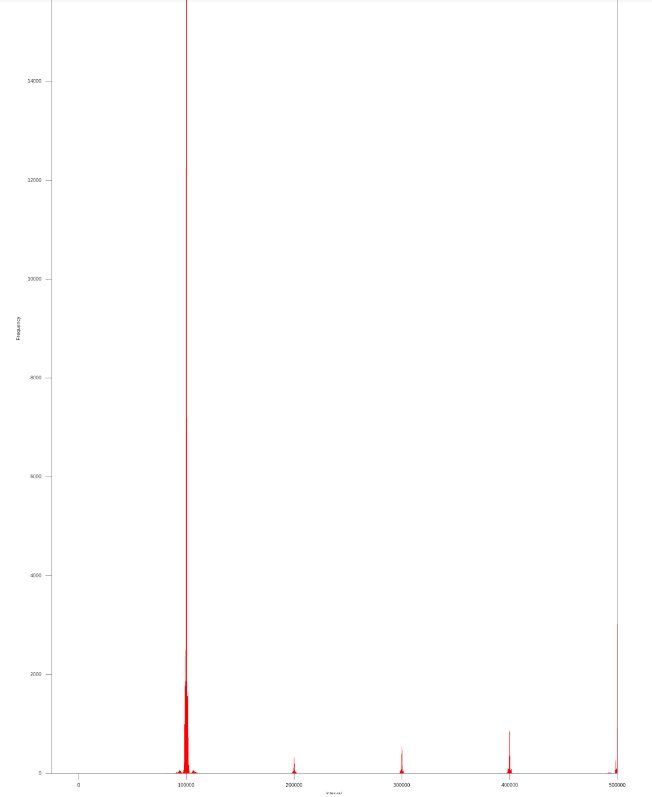
Hình 3.7: Kernel thường, code tối ưu( lập lịch cả 3 thread), có disturb 10



Hình 3.8: Kernel Realtime, code tối ưu( lập lịch cho 3 thread), có disturb 10



Hình 3.9: Kernel thường, code tối ưu( chỉ lập lịch cho Sample Thread), có disturb 10



Hình 3.10: Kernel Realtime, code tối ưu( lập lịch mỗi sample thread), có disturb 10

## 2. Nhận xét kết quả đạt được

* Kết quả khi chạy code thường so với code tối ưu chỉ có khác biệt đáng kể khi có xuất hiện chương trình disturb chiếm CPU, còn lại thì không có sự khác biệt đáng của kết quả khi nó đều cho ra kết quả tốt khi chạy chương trình giữa kernel thường và kernel real-time.
* Khi chạy code tối ưu có disturb thì kernel realtime có kết quả tốt hơn kernel thường, chúng có kết quả tập trung vào 1 mốc còn kernel thường bị phân tán.
* Với trường hợp lập lịch cho cả 3 thread thì kết quả đạt được ở kernel real-time tốt hơn kernel thường vì số mẫu kết quả đúng đạt được của kernel real-time là lớn hơn so với kernel thường khi chạy chương trình trong cùng khoảng thời gian, cùng số lượng disturb.
* Kernel real-time tốt hơn kernel thường trong việc thực hiện các chương trình yêu cầu tính real-time (yêu cầu khắt khe về thời gian hoàn thành).

# CHƯƠNG IV: KẾT QUẢ ĐẠT ĐƯỢC TRONG THỜI GIAN THỰC TẬP

-Được tiếp cận với môi trường làm việc thực tế, làm việc trên thiết bị của công ty.

- Những thứ học được:

+ Hiểu biết thêm kiến thức về ngành nhúng.

+ Cách build kernel bằng Yocto project, và cách thêm các chương trình, tool có sẵn vào image.

+ Cách thêm chương trình custom của mình vào image, cách thêm bản patch cho kernel.

+ Cách lập lịch cho thread để tối ưu code để đạt được kết quả mong muốn.

-Khó khăn gặp phải

+ Khó khăn khi tìm bản patch phù hợp để build kernel real-time, gặp nhiều lỗi không phù hợp với phiên bản.

+ Khó khăn khi tìm toolchain( bộ gcc binary) của kernel realtime để biên dịch code.