

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ - ĐẠI HỌC QUỐC GIA

KHOA ĐIỆN TỬ VIỄN THÔNG



BÁO CÁO THỰC TẬP THIẾT KẾ HỆ THỐNG
NHÓM THỰC TẬP NHÚNG – DỰ ÁN gNode 5G VHT

Sinh viên thực hiện: Vũ Thị Hồng Nhung

Mã số sinh viên: 19021498

Giảng viên hướng dẫn: TS. Nguyễn Đăng Phú

Công ty: Tổng Công ty Công nghiệp Công nghệ cao Viettel

HÀ NỘI - 2022

LỜI CẢM ƠN

Đầu tiên, em xin được gửi lời cảm ơn tới Khoa Điện tử - Viễn thông, Trường Đại học Công nghệ - Đại học Quốc Gia Hà Nội tạo điều kiện cũng như cơ hội để em được đi thực tập trong môi trường doanh nghiệp.

Để hoàn thành kỳ thực tập nhúng tại Tổng Công ty Công nghiệp Công nghệ cao Viettel, em xin gửi lời cảm ơn chân thành nhất tới anh Tạ Quốc Việt – trưởng phòng phần mềm hệ thống, anh Tăng Thiên Vũ, anh Đậu Hồng Quân, anh Hoàng Đức Trường và anh Trần Nam Hải tại công ty cùng thầy hướng dẫn TS. Nguyễn Đăng Phú, những người tận tình giúp đỡ cũng như đã tạo điều kiện tốt nhất giúp em trong suốt quá trình thực tập và hoàn thành báo cáo này.

Mặc dù đã cố gắng hết sức mình để hoàn thiện kỳ thực tập nhưng do sự hạn chế về kiến thức, kinh nghiệm nên báo cáo thực tập không thể tránh khỏi những thiếu sót, em rất mong nhận được sự cảm thông, đóng góp ý kiến của các anh và thầy, cô!

Cuối cùng, em xin kính chúc các anh, chị tại công ty cùng thầy, cô sức khỏe dồi dào và gặt hái được nhiều thành công trong cuộc sống cũng như trong công việc.

Em xin chân thành cảm ơn!

Hà Nội, ngày 09 tháng 09 năm 2022

Sinh viên



Vũ Thị Hồng Nhung

TÓM TẮT

Tóm tắt: Báo cáo “THỰC TẬP THIẾT KẾ HỆ THỐNG – NHÓM THỰC TẬP NHÚNG – DỰ ÁN gNode 5G VHT” là báo cáo toàn bộ quá trình thực tập của em tại Tổng Công ty Công nghiệp Công nghệ cao Viettel – VHT trong khoảng thời gian tháng 6/2022 – tháng 9/2022.

Mục tiêu chung của kỳ thực tập vừa qua là làm quen với nền tảng chip nhúng, làm quen với quy trình thiết kế và phát triển một hệ thống nhúng Linux. Đồng thời, làm quen với môi trường làm việc thực tế liên quan đến sản phẩm nhúng, cụ thể là dự án gNodeB 5G – Viettel.

Báo cáo gồm 5 phần chính như sau:

A. PHÂN TÍCH YÊU CẦU

Yêu cầu lớn và tổng quát đặt ra trong báo cáo này là : “Đánh giá tính real-time của hệ thống nhúng Linux”. Trong phần này, báo cáo phân tích cụ thể các yêu cầu nhỏ như hệ thống nhúng Linux, Real-time và công cụ đo tính real-time của hệ thống.

B. LÝ THUYẾT VÀ TRIỂN KHAI

Phần lý thuyết đề cập một cách chi tiết về những khái niệm, vấn đề đặt ra trong phần phân tích yêu cầu bên trên. Đồng thời triển khai về hệ thống nhúng Linux cho mạch, Linux Kernel Real-time và lý thuyết về viết ứng dụng C.

C. KẾT QUẢ VÀ NHẬN XÉT

Mô tả các test case cho chương trình và đưa ra kết quả. Rút ra nhận xét và kết luận về tính chính xác giữa kết quả của các testcase và lý thuyết.

D. KẾT QUẢ THU ĐƯỢC SAU QUÁ TRÌNH THỰC TẬP

Tổng hợp lại các công việc chính và kết quả thu được, bài học kinh nghiệm rút ra sau quá trình thực tập tại công ty.

E. TÀI LIỆU THAM KHẢO

Các tài liệu, giáo trình, nguồn tham khảo đã phục vụ cho quá trình thực tập nhúng tại công ty.

MỤC LỤC

LỜI CẢM ƠN	2
TÓM TẮT	3
A. PHÂN TÍCH YÊU CẦU	9
I. Hệ thống nhúng Linux	9
1. Hệ thống nhúng	9
2. Linux.....	10
3. Hệ thống nhúng Linux	12
II. Real-time	14
III. Công cụ đo tính real-time của hệ thống	16
B. LÝ THUYẾT VÀ TRIỂN KHAI.....	19
I. Lý thuyết và triển khai hệ thống nhúng Linux trên thiết bị	19
1. Tìm hiểu về KIT Orange Pi Zero.....	19
2. Boot Pi với PiOs	21
2.1. Lý thuyết liên quan	21
2.2. Triển khai boot Pi với PiOs.....	22
2.2.1. Chuẩn bị các phụ kiện cần thiết	22
2.2.2. Cài hệ điều hành Raspbian	23
2.2.3. Cài đặt wifi và kết nối qua SSH.....	25
2.3 Nhận xét	28
3. Build Image bằng công cụ Yocto Project.....	28
3.1. Lý thuyết liên quan	28
3.2. Triển khai build image bằng Yocto Project	31
3.2.1. Triển khai build image	31
3.2.2. Boot bản image lên KIT Pi.....	33

3.3. Nhận xét	33
II. Lý thuyết và triển khai Linux Kernel Real-time.....	34
1. Lý thuyết về Linux Kernel Real-time.....	34
1.1. Scheduling với Real-Time	34
1.2. Locking	36
1.3. Signalling	36
1.4. Clocks và Cyclic Tasks.....	37
1.5. Đánh giá một hệ thống Real-time	38
2. Triển khai build Kernel Real-time.....	38
III. Lý thuyết và triển khai viết ứng dụng C.....	40
1. Lý thuyết về lập trình multi-thread.....	40
2. Lý thuyết về tối ưu code realtime	44
3. Triển khai yêu cầu	44
3.1. Thread SAMPLE	44
3.2. Thread INPUT	46
3.3. Thread LOGGING	46
3.4. Phần code tối ưu.....	48
3.4.1. Thread SAMPLE.....	48
3.4.2. Thread INPUT.....	49
3.4.2. Thread LOGGING	50
C. KẾT QUẢ VÀ NHẬN XÉT	52
I. Mô tả các test case	55
1. Test case 1	55
2. Test case 2	56
3. Test case 3	57

4. Test case 4	58
II. Nhận xét và đánh giá kết quả.....	59
1. Lần chạy ứng dụng ban đầu.....	59
2. Lần chạy ứng dụng ban đầu cùng ứng dụng disturb*10.....	59
3. Lần chạy ứng dụng được tối ưu.....	59
4. Lần chạy ứng dụng được tối ưu cùng disturb*10.....	59
5. Kết luận.....	60
D. KẾT QUẢ THU ĐƯỢC SAU QUÁ TRÌNH THỰC TẬP.....	60
E. TÀI LIỆU THAM KHẢO	61

MỤC LỤC BẢNG

<i>Bảng 1. 1 Thông số kỹ thuật của KIT Orange Pi Zero</i>	<i>20</i>
<i>Bảng 1. 2 Kết nối dây KIT Pi và USB2UART.....</i>	<i>23</i>
<i>Bảng 1. 3 So sánh sách lược lập lịch giữa Non-Real-time và Real-time.....</i>	<i>36</i>

MỤC LỤC HÌNH ẢNH

<i>Hình 1. 1 Mô hình kiến trúc hệ điều hành Linux</i>	<i>12</i>
<i>Hình 1. 2 Sơ đồ thuật toán cho thread SAMPLE.....</i>	<i>17</i>
<i>Hình 1. 3 Sơ đồ thuật toán cho thread INPUT</i>	<i>18</i>
<i>Hình 1. 4 Sơ đồ thuật toán cho ththread LOGGING.....</i>	<i>18</i>
<i>Hình 2. 1 Giao diện của KIT Orange Pi Zero</i>	<i>21</i>
<i>Hình 2. 2 Phần mềm VNC Viewer</i>	<i>23</i>
<i>Hình 2. 3 Kết nối thực tế KIT Orange Pi Zero và USB2UART</i>	<i>24</i>

<i>Hình 2. 4 Cấu hình KIT Pi qua cổng COM</i>	<i>24</i>
<i>Hình 2. 5 Cấu hình KIT Pi qua cổng COM</i>	<i>25</i>
<i>Hình 2. 6 Giao diện sau khi kết nối wifi và chạy lệnh ifconfig</i>	<i>26</i>
<i>Hình 2. 7 Cấu hình KIT Pi khi kết nối qua SSH</i>	<i>27</i>
<i>Hình 2. 8 Giao diện khi kết nối qua SSH</i>	<i>27</i>
<i>Hình 2. 9 Yocto Project.....</i>	<i>29</i>
<i>Hình 2. 10 Bitbake</i>	<i>30</i>
<i>Hình 2. 11 Poky.....</i>	<i>31</i>
<i>Hình 3. 1 Biểu đồ phân bố giá trị interval trên Kernel thường</i>	<i>55</i>
<i>Hình 3. 2 Biểu đồ phân bố giá trị interval trên Kernel Real-time</i>	<i>55</i>
<i>Hình 3. 3 Biểu đồ phân bố giá trị interval trên Kernel thường cùng file disturb.....</i>	<i>56</i>
<i>Hình 3. 4 Biểu đồ phân bố giá trị interval trên Kernel Real-time cùng file disturb.....</i>	<i>56</i>
<i>Hình 3. 5 Biểu đồ phân bố giá trị interval khi code tối ưu trên Kernel thường</i>	<i>57</i>
<i>Hình 3. 6 Biểu đồ phân bố giá trị interval khi code tối ưu trên Kernel Real-time</i>	<i>57</i>
<i>Hình 3. 7 Biểu đồ phân bố giá trị interval khi code tối ưu trên Kernel thường cùng file disturb</i>	<i>58</i>
<i>Hình 3. 8 Biểu đồ phân bố giá trị interval khi code tối ưu trên Kernel Real-time cùng file disturb</i>	<i>58</i>

A. PHÂN TÍCH YÊU CẦU

Bài báo cáo này với yêu cầu lớn và tổng quát là “Đánh giá trình real-time của hệ thống nhúng Linux”. Dưới đây là những phân tích yêu cầu:

I. Hệ thống nhúng Linux

Trong phần này, các khái niệm về hệ thống nhúng hoạt động dựa trên Linux sẽ được bóc tách và làm rõ như sau:

1. Hệ thống nhúng

Hệ thống nhúng (Embedded System) là một thuật ngữ để chỉ một hệ thống có khả năng tự trị được nhúng vào trong một môi trường hay một hệ thống mẹ. Đó là các hệ thống tích hợp cả phần cứng và phần mềm phục vụ các bài toán chuyên dụng trong nhiều lĩnh vực công nghiệp, tự động hoá điều khiển, quan trắc và truyền tin. Đặc điểm của các hệ thống nhúng là hoạt động ổn định và có tính năng tự động hoá cao.

Các hệ thống nhúng được thiết kế để thực hiện một số nhiệm vụ chuyên dụng chứ không phải đóng vai trò là các hệ thống máy tính đa chức năng. Một số hệ thống đòi hỏi ràng buộc về khả năng hoạt động thời gian thực để đảm bảo độ an toàn và tính ứng dụng; một số hệ thống không đòi hỏi hoặc ràng buộc chặt chẽ, cho phép đơn giản hóa hệ thống phần cứng để giảm thiểu chi phí sản xuất. Ngoài ra, hệ thống nhúng còn có tài nguyên giới hạn, các hệ thống nhúng bị giới hạn cả về phần cứng và phần mềm so với các hệ thống khác như máy tính cá nhân. Bên cạnh đó, hệ thống nhúng có yêu cầu chất lượng ổn định và độ tin cậy cao. Một lỗi của hệ thống nhúng có thể gây ra hậu quả lớn và khó khắc phục. Vì vậy việc phát triển hệ thống nhúng yêu cầu quy trình kiểm tra - kiểm thử rất cẩn thận.

Hệ thống nhúng là hệ thống máy tính được thiết kế có chức năng điều khiển và được đặt (nhúng, cài, gắn) trong một hệ thống lớn hơn. Như vậy, hệ thống

nhúng liên quan đến hệ thống máy tính, bao gồm cả phần cứng lẫn phần mềm được phát triển đồng bộ, thực hiện những chức năng điều khiển đối với một hệ thống lớn hơn mà hệ thống nhúng đó được cài vào. Có thể nói hệ thống nhúng là bộ não của hệ thống lớn hơn mà nó điều khiển. Hệ thống lớn là môi trường hoạt động của hệ thống nhúng. Có thể hình dung vị trí trung tâm quan trọng của hệ thống nhúng trong hệ thống lớn hơn gồm nhiều hệ thống con mà hệ thống nhúng điều khiển, kể cả tương tác giữa hệ thống lớn hơn đó với môi trường xung quanh. Nói cách khác, hệ thống nhúng là một dạng vi tính đặc biệt, là hệ thống chuyên dụng để ứng dụng các bộ xử lý chuyên dụng nhằm điều khiển hệ thống lớn hơn mà nó là thành phần.

Ngày nay, hệ thống nhúng rất gần gũi với cuộc sống của chúng ta, điều này được thấy rõ trong các sản phẩm như sau:

- Các thiết bị kết nối mạng: router, hub, gateway,...
- Các hệ thống dẫn đường trong không lưu, hệ thống định vị toàn cầu, vệ tinh
- Các thiết bị văn phòng: máy photocopy, máy fax, máy in, máy scan,...
- Dây chuyền sản xuất tự động trong công nghiệp, robots.
- Các thiết bị gia dụng: tủ lạnh, lò vi sóng, lò nướng,...

2. Linux

Linux là một hệ điều hành máy tính được phát triển từ năm 1991 dựa trên hệ điều hành Unix và bằng viết bằng ngôn ngữ C. Do Linux được phát hành miễn phí và có nhiều ưu điểm vượt trội nên Linux vẫn giữ được một chỗ đứng vững chắc trong lòng người dùng .

Vậy thử so sánh Linux và Windows. Ta sẽ thấy được như sau:

- Linux là hệ điều hành nguồn mở (sử dụng miễn phí). Trong khi đó hệ điều hành Windows là thương mại (phải mua bản quyền).

- Linux có quyền truy cập vào mã nguồn và thay đổi nó theo nhu cầu của người dùng. Trong khi đó điều này là không thể đối với Windows.
- Linux sẽ chạy nhanh và ổn định hơn Windows. Đặc biệt là với các hệ thống cấu hình yếu cũng như lỗi thời.
- Các bản phân phối Linux không thu thập dữ liệu người dùng. Trong khi Windows thu thập tất cả các chi tiết người dùng.
- Khi một ứng dụng bị treo Linux sẽ dễ dàng tắt nó đi hơn vì bạn chỉ cần sử dụng lệnh kill/pkill. Trong khi đó với Windows bạn sẽ rất khó để đóng nó, có khi bạn cần phải khởi động lại máy.
- Linux sử dụng chủ yếu với giao diện dòng lệnh. Windows thì giao diện người dùng, dễ sử dụng hơn Linux.
- Một số nhà cung cấp sẽ không hỗ trợ Driver cho Linux. Đây là điểm yếu hơn của Linux so với Windows.
- Linux có tính bảo mật cao vì dễ xác định lỗi và sửa lỗi. Trong khi Windows có lượng người dùng lớn và trở thành mục tiêu cho các nhà phát triển virus và phần mềm độc hại.

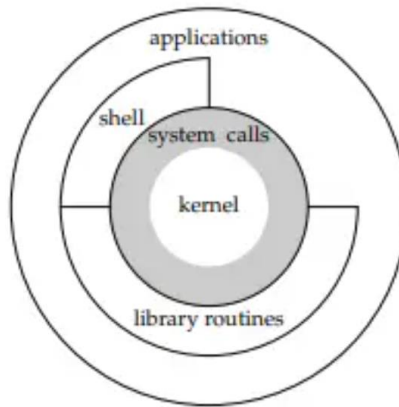
Tóm lại, hệ điều hành Linux sẽ có các ưu điểm như sau:

- Hoàn toàn miễn phí.
- Tính linh hoạt cao.
- Chạy ổn định trên các hệ thống có cấu hình yếu.
- Hướng dẫn sử dụng phong phú.
- Độ an toàn cao..

Bên cạnh đó, hệ điều hành này vẫn có các nhược điểm như:

- Các phần mềm còn hạn chế.
- Phần cứng ít được hỗ trợ.
- Đòi hỏi người dùng phải thành thạo, mất thời gian để làm quen.

Kiến trúc hệ điều hành linux được chia thành 3 thành phần là Kernel, Shell và Applications. Hình 1.1 dưới đây mô tả kiến trúc của hệ điều hành Linux:



Hình 1. 1 Mô hình kiến trúc hệ điều hành Linux

Cụ thể các thành phần như sau:

- Kernel: Là phần quan trọng nhất. Có thể nói nó là trái tim của hệ điều hành. Phần kernel chứa các module, thư viện để quản lý và giao tiếp với phần cứng và các ứng dụng.
- Shell: Shell là một chương trình có chức năng thực thi các lệnh từ người dùng hoặc từ các ứng dụng cũng như là những tiện ích yêu cầu chuyển đến cho Kernel xử lý.
- Applications: Là các ứng dụng và tiện ích mà người dùng cài đặt trên Server.

3. Hệ thống nhúng Linux

Hệ thống nhúng Linux (Embedded Linux) đơn giản chỉ là một hệ thống nhúng dựa trên nhân Linux, bản phân phối của hệ thống nhúng Linux có thể bao

gồm các phần như: khung phát triển cho hệ thống nhúng Linux, các ứng dụng phần mềm khác nhau được thiết kế riêng để sử dụng trong hệ thống nhúng.

Bốn thành phần quan trọng của hệ thống nhúng Linux gồm:

- Toolchain: chứa trình biên dịch và các công cụ cần thiết để tạo code cho thiết bị. Những thứ khác đều phụ thuộc vào toolchain.
- Bootloader: Nó cần thiết cho quá trình khởi tạo và tải, boot Linux kernel.
- Kernel: Như đã nói ở trên nó là trái tim của hệ thống, có nhiệm vụ quản lý tài nguyên và giao tiếp với hardware.
- Root filesystem: Chứa các thư viện và chương trình được chạy sau khi quá trình khởi tạo kernel hoàn thành.

Hệ thống nhúng Linux có sự phát triển vượt bậc là do có sức hấp dẫn đối với các ứng dụng không đòi hỏi thời gian thực như; các hệ server nhúng, các ứng dụng giá thành thấp và đòi hỏi thời gian đưa sản phẩm ra thị trường nhanh. Mặt khác Linux là phần mềm mã nguồn mở nên bất kỳ ai cũng có thể hiểu và thay đổi theo ý mình. Linux cũng là một hệ điều hành có cấu trúc module và chiếm ít bộ nhớ trong khi Windows không có các ưu điểm này.

Bên cạnh các ưu điểm trên thì Embedded Linux cũng có các nhược điểm sau:

- Embedded Linux không phải là hệ điều hành thời gian thực nên có thể không phù hợp với một số ứng dụng như điều khiển quá trình, các ứng dụng có các yêu cầu xử lý khẩn cấp.
- Embedded Linux thiếu một chuẩn thống nhất và không phải là sản phẩm của một nhà cung cấp duy nhất nên khả năng hỗ trợ kỹ thuật chưa cao.

II. Real-time

Phần lớn chúng ta đều quen thuộc với những hệ thống mà dữ liệu cần được xử lý trong khoảng thời gian xác định. Thử lấy một ví dụ như sau một máy bay sử dụng một dãy các phép đo bằng bộ cảm biến để các định vị trí hiện tại của máy bay thì nó cần xử lý thật nhanh và kịp thời. Thêm vào đó, các hệ thống khác lại đòi hỏi thời gian xử lý phải thật nhanh và kịp thời, nếu chỉ chậm trễ một tí tấc cũng có thể xảy ra sự cố nghiêm trọng. Trong thực tế cũng có nhiều hệ thống như vậy. Theo một định nghĩa nào đó, ta có thể hiểu sự kiện này đòi hỏi xử lý thời gian thực. Những hệ xử lý thời gian thực như thế thường là các hệ thống nhúng. Vì vậy, trong thực tế chúng ta thường gặp các hệ thống nhúng thời gian thực (Real-time Embedded System).

Hệ thống thời gian thực là hệ mà phải thỏa mãn tường minh thời gian hồi đáp được đặt ra, nếu không thỏa mãn thì gặp phải hậu quả nghiêm trọng, kể cả hỏng hóc hệ thống hoàn toàn. Hệ thống thời gian thực là hệ mà tính đúng đắn logic dựa trên cả hai yếu tố là tính đúng đắn của các đầu ra và tính thời gian của chúng. Nói cách khác, hệ thời gian thực là hệ mà tính chất thời gian là yếu tố quan trọng không kém gì tính đúng đắn của các đầu ra.

Các tác vụ thời gian thực là những tác vụ phải được thực hiện và hoàn thành trong một khoảng thời gian cụ thể. Lập trình thời gian thực về cơ bản là đảm bảo kiểm soát các luồng đầu vào trong thời gian thực được lên lịch khi cần thiết và hoàn thành nhiệm vụ trước deadline. Bất kỳ trở ngại nào đối với điều này thì đều trở thành vấn đề. Dưới đây là một số vấn đề hay gặp phải:

- Scheduling (Lập lịch): Các thread thời gian thực phải được lên lịch trước vậy nên ta phải có chính sách thời gian thực, `SCHED_FIFO` hoặc `SCHED_RR`. Ngoài ra, ta nên có các mức độ ưu tiên được gán theo mức độ giảm dần, bắt đầu từ mức độ ưu tiên ngắn nhất theo deadline.

- Scheduling latency (Độ trễ lập lịch): Kernel có thể lên lịch ngay cho các sự kiện khi xảy ra ngắt hoặc bộ đếm thời gian, và không chịu sự chậm trễ vô hạn.
- Priority inversion (Đảo ngược ưu tiên): Là hệ quả của việc lập lịch dựa theo độ ưu tiên, dẫn đến độ trễ vô hạn khi có thread có mức độ ưu tiên cao bị chặn trên mutex do một thread có mức độ ưu tiên thấp.
- Accurate timers (Bộ hẹn giờ chính xác): Nếu ta muốn quản lý thời gian trong khu vực thấp mili giây hoặc micro giây, ta cần phải có bộ hẹn giờ phù hợp. Bộ hẹn giờ có độ phân giải cao rất quan trọng và là một cấu hình tùy chọn trên hầu hết các kernel.
- Page fault (Lỗi trang): Lỗi trang trong khi thực thi một đoạn mã quan trọng sẽ làm đảo lộn mọi thời gian ước tính. Ta có thể tránh chúng bằng cách khóa bộ nhớ.
- Interrupts (Ngắt): Xảy ra vào những thời điểm mà ta không thể đoán trước được và có thể dẫn đến sự cố không mong muốn. Ta có hai cách để xử lý: Một là chạy các ngắt dưới thread kernel. Cách thứ hai là trên multi-core (đa lõi) để bảo vệ một hoặc nhiều CPU khỏi việc xử lý ngắt.
- Processor caches (Bộ nhớ đệm của bộ xử lý): Cung cấp một bufer giữa CPU và bộ nhớ chính giống như caches, là một nguồn không xác định, đặc biệt là trên các thiết bị đa lõi.
- Memory bus contention (Tranh chấp bộ nhớ bus): Khi thiết bị ngoại vi truy cập trực tiếp vào bộ nhớ thông qua kênh DMA (Direct Memory Access), nó sử dụng một phần của bộ nhớ bus, điều này là chậm truy cập từ lõi CPU.

Để phân loại hệ thống thời gian thực, người ta thường phân ra thành 2 loại dựa trên tiêu chí đánh giá hậu quả của việc không đáp ứng ràng buộc về thời gian của từng hệ thống.

- Hệ thống “Hard real-time”: nếu không đáp ứng yêu cầu thời gian thực, hệ thống sẽ sụp đổ hay gây ra hậu quả nghiêm trọng.
- Hệ thống “Soft Real-time”: nếu không đáp ứng yêu cầu thời gian thực, hệ thống sẽ bị suy giảm về chất lượng dịch vụ

Khi thiết kế và phát triển hệ thời gian thực ta cần xem xét một số vấn đề liên quan đến thiết kế hệ thống, có thể là:

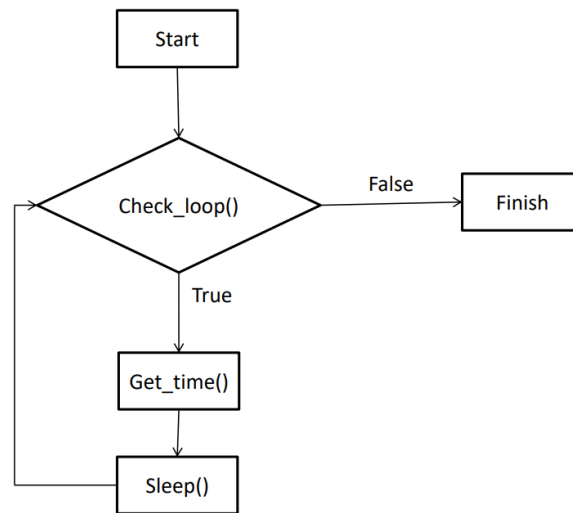
- Việc lựa chọn giữa phần cứng và phần mềm, hoặc kết hợp hợp lý giữa chúng theo giải pháp giá thành – hiệu quả.
- Việc quyết định xem sử dụng hệ điều hành thời gian thực (RTOS) thương mại hay tự thiết kế và xây dựng hệ điều hành chuyên dụng riêng.
- Việc lựa chọn ngôn ngữ phần mềm hợp lý để phát triển hệ thống.
- Việc cực đại hóa dung sai lỗi hệ thống và độ tin cậy thông qua thiết kế cân trọng và kiểm thử khắt khe, nghiêm ngặt, chính xác.
- Việc thiết kế và quản lý kiểm thử, đồng thời việc lựa chọn thiết bị kiểm thử và môi trường phát triển hệ thời gian thực

III. Công cụ đo tính real-time của hệ thống

Để xây dựng và phát triển các hệ thống nhúng đa phần kiểm tra ràng buộc thời gian thực của hệ thống dựa trên thực nghiệm tức là tiến hành đánh giá một số Test case (trường hợp kiểm thử) cụ thể. Với các hệ thống có cấu hình tương đối mạnh, có thể porting hệ điều hành thời gian thực thì chúng ta lại tin tưởng vào việc điều phối, lập lịch của hệ điều hành.

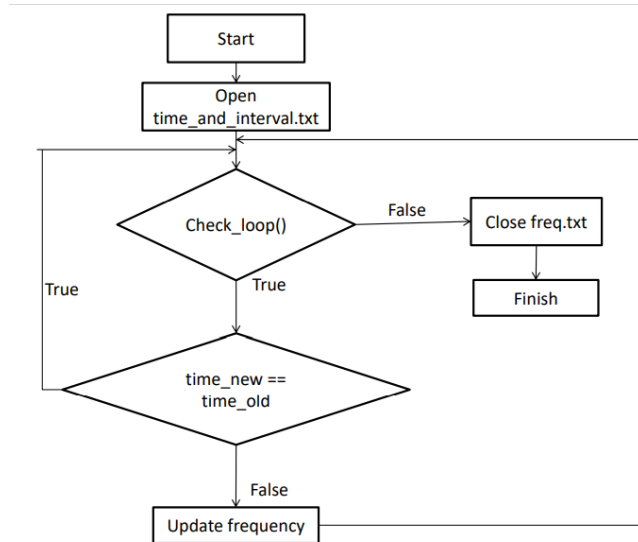
Cụ thể, chương trình mà ta thực hiện như sau: Đầu tiên cần lấy mẫu sau một khoảng thời gian T . Bên cạnh đó chương trình sẽ tính giá trị offset, offset ở đây là sự chênh lệch giữa thời gian hai lần lấy mẫu. Và khi so sánh T và kết quả offset thì ta sẽ thấy được độ trễ của Kernel.

Sơ đồ thuật toán được mô tả lần lượt trong các hình 1.2, 1.3 và 1.4 dưới đây như sau:



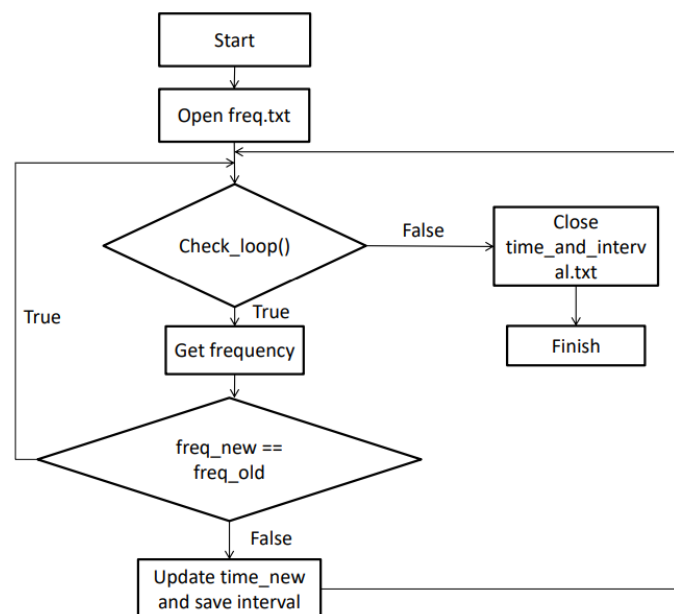
Hình 1. 2 Sơ đồ thuật toán cho thread *SAMPLE*

Thread **SAMPLE** thực hiện vô hạn lần đọc thời gian hệ thống hiện tại (chính xác đến đơn vị ns) vào biến T với chu kỳ X ns.



Hình 1. 3 Sơ đồ thuật toán cho thread INPUT

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.



Hình 1. 4 Sơ đồ thuật toán cho thread LOGGING

Thread **LOGGING** chờ khi biến T được cập nhật mới, giá trị interval (offset giữa biến T hiện tại và biến T của lần ghi trước) ra file có tên “**time_and_interval.txt**”.

B. LÝ THUYẾT VÀ TRIỂN KHAI

I. Lý thuyết và triển khai hệ thống nhúng Linux trên thiết bị

Trong phần này, ta tiến hành triển khai hệ thống nhúng trên KIT Orange Pi Zero như sau:

1. Tìm hiểu về KIT Orange Pi Zero

Máy tính nhúng Orange Pi Zero là dòng máy tính nhỏ gọn, đơn giản, hữu ích. Theo trang chủ của nhà sản xuất Orange Pi thì máy có thể chạy các hệ điều hành Android 4.4, Ubuntu, Debian. Orange Pi Zero sử dụng CPU AllWinner H2 SoC, được trang bị 512MB DDR3 SDRAM, giao tiếp Wifi, LAN, USB,...

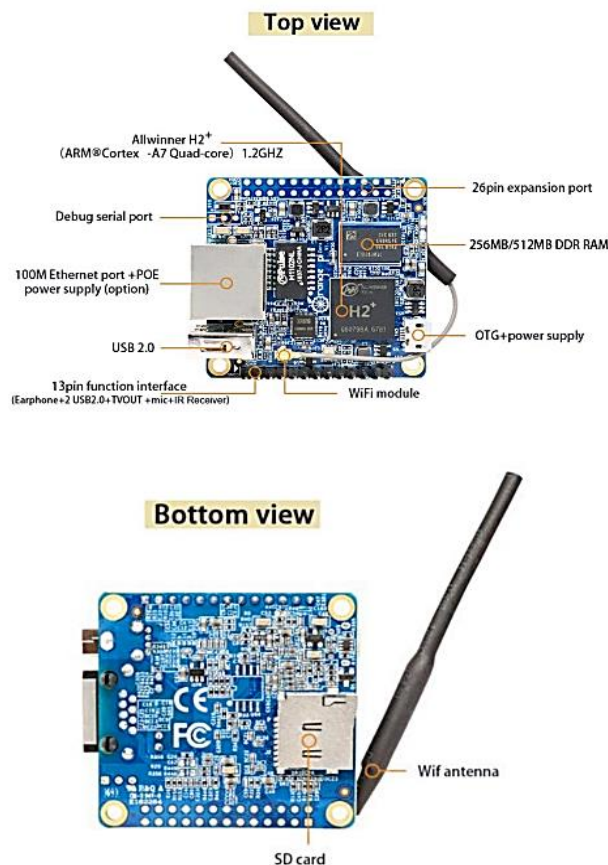
Các thông số kỹ thuật cụ thể được mô tả trong bảng 1.1 dưới đây:

CPU	H2 Quad-core Cortex-A7 H.265 / HEVC 1080P
GPU	Mali400MP2GPU@600MHz
Hỗ trợ	OpenGLES2.0
Bộ nhớ (SDRAM)	256MB/ 512MB DDR3 (chia sẻ với GPU) (256MB là tiêu chuẩn)
Lưu trữ trên máy	Thẻ TF (Tối đa 64GB) / NOR Flash (Mặc định Mặc định 2MB)
Mạng Onboard	10/100 M Ethernet RJ45 POE là mặc định tắt.

On WIFI	XR819, IEEE 802.11 b/ g/ n
Lỗ vào âm thanh	MIC
Đầu ra video	Hỗ trợ bảng bên ngoài thông qua 13pins
Nguồn	USB OTG có thể cung cấp nguồn điện
Cổng USB 2.0	Chỉ một cổng USB 2.0 HOST, một cổng USB 2.0 OTG
Nút	Nút nguồn
Thiết bị ngoại vi cấp thấp	26 đầu Pins, tương thích với Raspberry Pi B+ 13 đầu Pins, với 2x USB, chân IR, AUDIO (MIC, AV)
GPIO (1x3) pin	UART, ground
LED	LED dẫn điện và trạng thái dẫn
Hệ điều hành được hỗ trợ	Android, Lubuntu, Debian, Raspbian
Kích thước	48 mm x 46 mm
Khối lượng	26g

Bảng 1. 1 Thông số kỹ thuật của KIT Orange Pi Zero

Hình 1.2 dưới đây mô tả tương ứng giao diện của Orange Pi



Hình 2. 1 Giao diện của KIT Orange Pi Zero

Để triển khai hệ thống nhúng Linux thì có 2 hướng tiếp cận. Cách đầu tiên là boot Pi với PiOs và còn cách thứ hai là build bằng công cụ Yocto Project.

2. Boot Pi với PiOs

2.1. Lý thuyết liên quan

Armbian là phiên bản hệ điều hành Linux dành riêng cho các máy tính nhúng. Nó cung cấp nhiều hình ảnh dựng sẵn khác nhau cho một số bo mạch được hỗ trợ. Nó 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, Banana PI, trong số những máy tính khác.

2.2. Triển khai boot Pi với PiOs

2.2.1. Chuẩn bị các phụ kiện cần thiết

- KIT Orange Pi Zero
- Modul USB2UART PL2303 hoặc modul tương ứng và nguồn cấp



- Thẻ nhớ: MicroSD 16GB class 10 (Tối thiểu là 8GB)

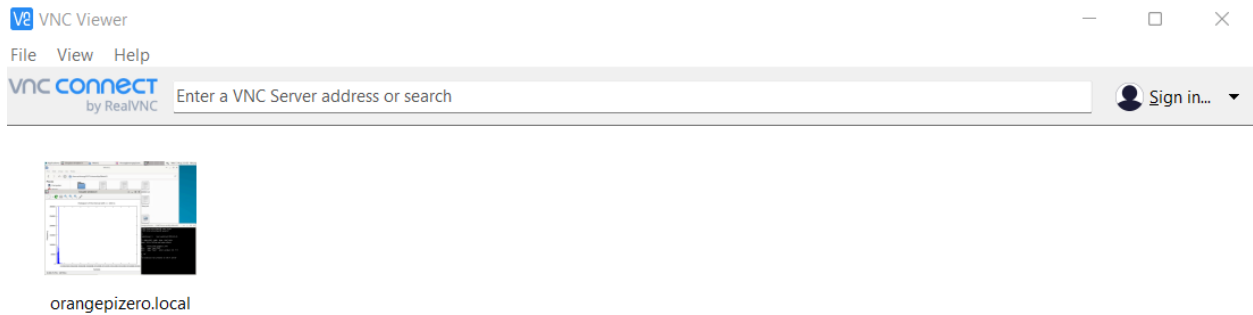


- Dây cắm: Loại cái – cái:



- Hệ điều hành Raspbian: <https://www.armbian.com/orange-pi-zero/>
- Phần mềm Win32DiskImager : <https://sourceforge.net/projects/win32diskimager/>
- Phần mềm Putty For Windows on Intel x86 để xem log cài đặt và remote từ xa: <https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>

- Phần mềm VNC Viewer để điều khiển máy tính từ xa, tức điều khiển PiOs của Orange Pi Zero)



Hình 2. 2 Phần mềm VNC Viewer

2.2.2. Cài hệ điều hành Raspbian

Bước 1: Format thẻ nhớ

Bước 2: Cài đặt Win32DiskImager. Bên cạnh đó ta cần giải nén Raspbian từ file đã tải về (theo đường link bên trên tương ứng), trở đường dẫn vào mục Image File, sau đó chọn Write.

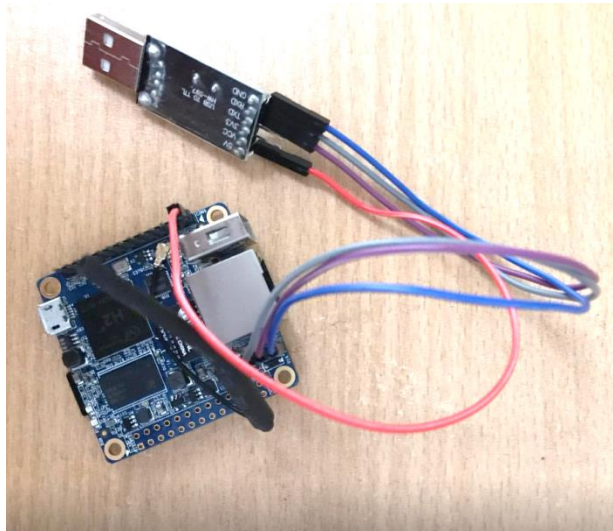
Bước 3: Chờ đợi thông báo hoàn thành, sau đó rút thẻ nhớ ra và cắm vào KIT Orange Pi Zero, thực hiện kết nối dây như dưới đây.

Bước 4: Thực hiện kết nối dây theo bảng như sau:

USB2UART	Orange Pi Zero
5V	+5V
TX	RX
RX	TX
GND	GND

Bảng 1. 2 Kết nối dây KIT Pi và USB2UART

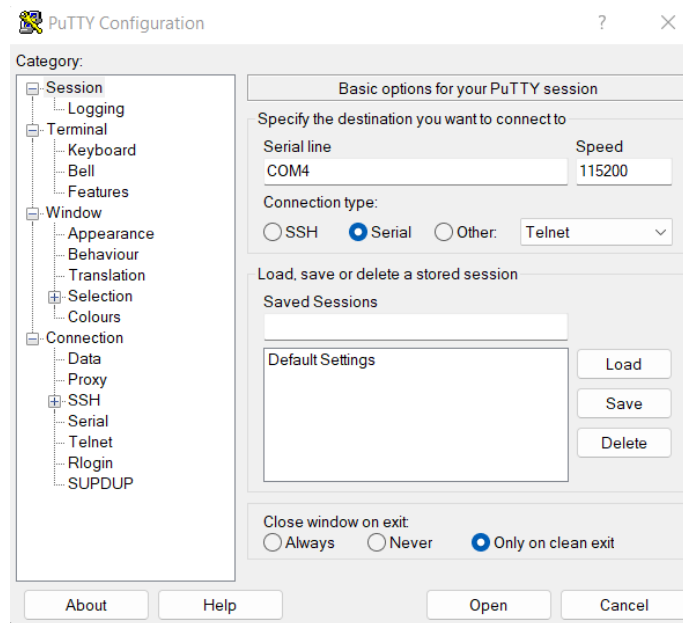
Hình ảnh thực tế kết nối như sau :



Hình 2. 3 Kết nối thực tế KIT Orange Pi Zero và USB2UART

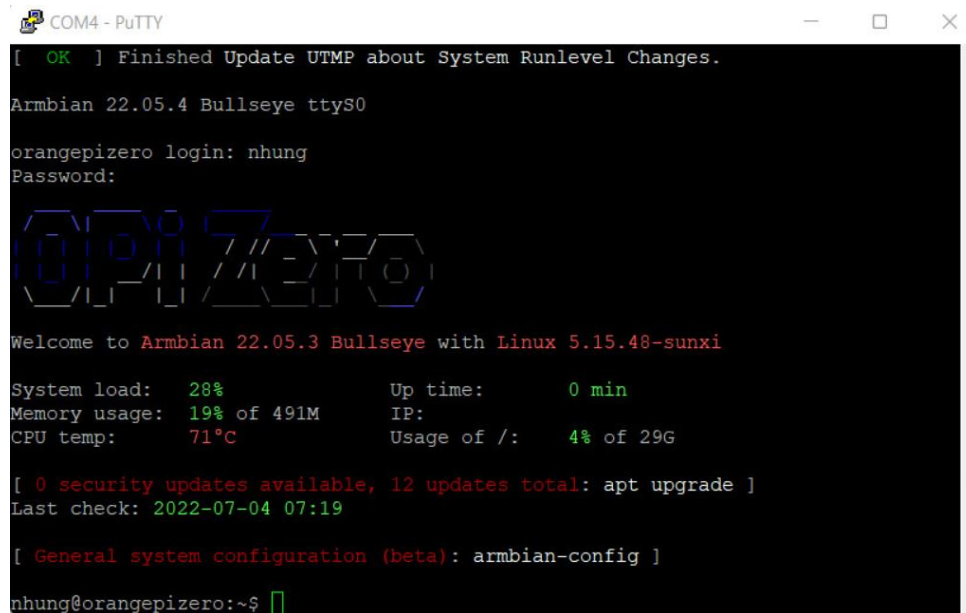
Bước 5: Cấu hình

Sau khi kết nối thì mở phần mềm Putty lên, sau đó chọn Serial tương ứng rồi chọn COM và tốc độ baud là 115200.



Hình 2. 4 Cấu hình KIT Pi qua cổng COM

Bước 6: Sau khi boot xong sẽ hiện thông báo đăng nhập, ta sử dụng tài khoản root và password là 1234 để đăng nhập. Sau đó là yêu cầu cần thay đổi password cũ và thay password mới, thiết lập thông tin.



```
COM4 - PuTTY
[ OK ] Finished Update UTMP about System Runlevel Changes.
Armbian 22.05.4 Bullseye ttyS0
orangezero login: hung
Password:
1234

Welcome to Armbian 22.05.3 Bullseye with Linux 5.15.48-sunxi

System load:  28%      Up time:    0 min
Memory usage: 19% of 491M  IP:
CPU temp:    71°C     Usage of /:  4% of 29G

[ 0 security updates available, 12 updates total: apt upgrade ]
Last check: 2022-07-04 07:19

[ General system configuration (beta): armbian-config ]
hung@orangezero:~$
```

Hình 2. 5 Cấu hình KIT Pi qua cổng COM

2.2.3. Cài đặt wifi và kết nối qua SSH

Bước 1: Chạy lệnh

```
1 | $ vi /etc/network/interfaces
```

Bước 2: Ấn Insert sau đó thêm dòng lệnh sau vào cuối

```
1 | auto wlan0
2 | iface wlan0 inet dhcp
3 | wpa_conf /etc/wpa_supplicant/wpa_supplicant.conf
```

Và ấn ESC và Shift + Z + Z để lưu lại.

Bước 3: Chạy lệnh

```
1 | $ vi /etc/wpa_supplicant/wpa_supplicant.conf
```

Bước 4: Ấn Insert sau đó thêm dòng lệnh sau vào cuối

```
1 | network={
2 |   ssid="tên wifi"
3 |   psk="mật khẩu"
4 | }
```

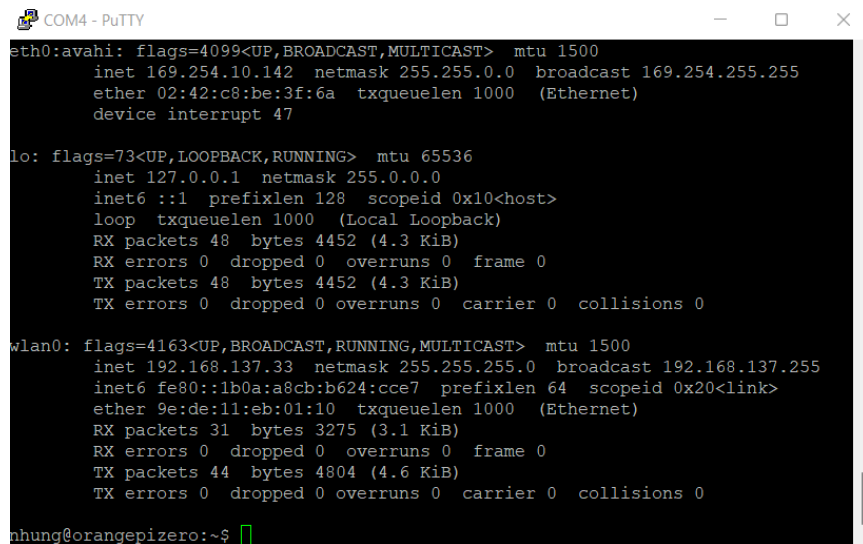
Trong đó tên wifi và mật khẩu wifi sẽ tương ứng với wifi mà ta đang sử dụng.

Sau đó ấn ESC và Shift + Z + Z để lưu lại.

Bước 5: Chạy lệnh sau để khởi động wifi

```
1 | $ ifconfig wlan0 up
2 | $ ifup wlan0
```

Thì thu được như sau:



```
COM4 - PuTTY
eth0:avahi: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 169.254.10.142 netmask 255.255.0.0 broadcast 169.254.255.255
    ether 02:42:c8:be:3f:6a txqueuelen 1000 (Ethernet)
    device interrupt 47

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 48 bytes 4452 (4.3 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 48 bytes 4452 (4.3 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

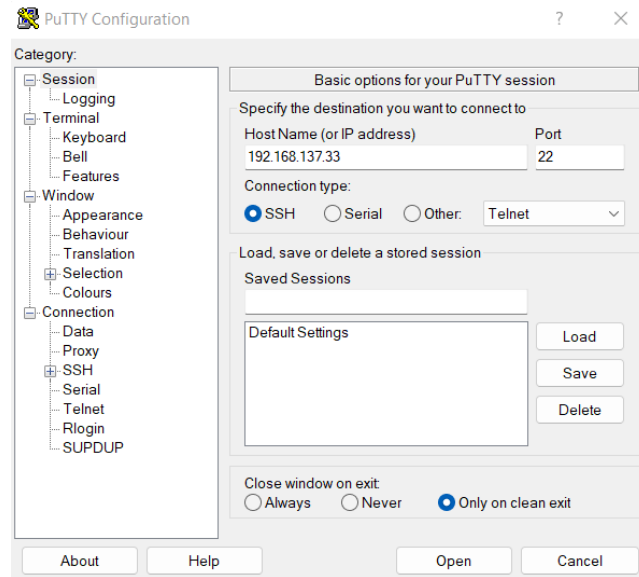
wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.137.33 netmask 255.255.255.0 broadcast 192.168.137.255
    inet6 fe80::1b0a:a8cb:b624:cce7 prefixlen 64 scopeid 0x20<link>
    ether 9e:de:11:eb:01:10 txqueuelen 1000 (Ethernet)
    RX packets 31 bytes 3275 (3.1 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 44 bytes 4804 (4.6 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

nhung@orangepizero:~$
```

Hình 2. 6 Giao diện sau khi kết nối wifi và chạy lệnh ifconfig

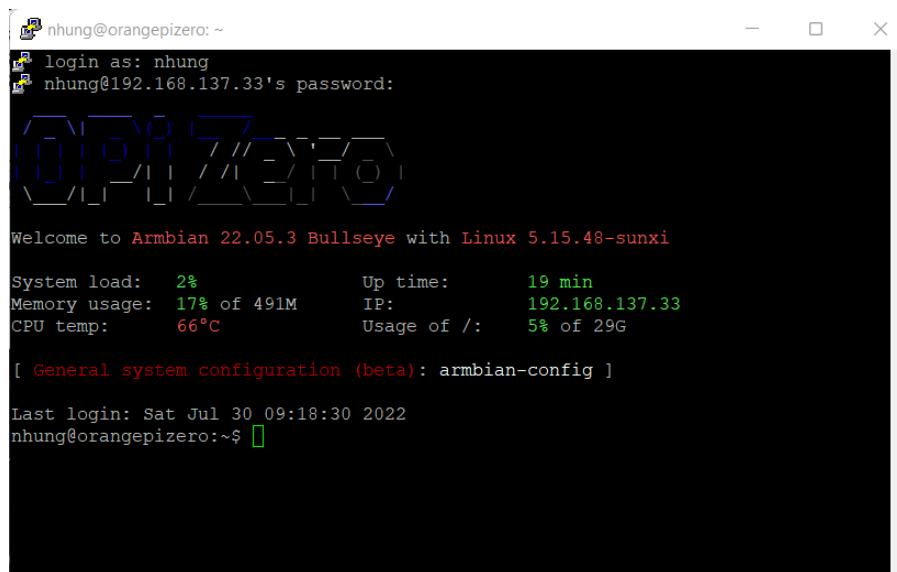
Bước 6: Khởi động lại Orange Pi Zero bằng lệnh reboot nếu bước 5 bị lỗi

Bước 7: Mở phần mềm Putty mới và login thông qua địa chỉ IP để thử remote Orange Pi Zero qua mạng



Hình 2. 7 Cấu hình KIT Pi khi kết nối qua SSH

Kết quả thu được:



Hình 2. 8 Giao diện khi kết nối qua SSH

2.3 Nhận xét

Đối với ưu điểm, có thể thấy:

- Dễ dàng để tiến hành thực hiện cài hệ điều hành.
- Các tính năng đầy đủ để phục vụ cho tìm hiểu và học tập.

Bên cạnh những ưu điểm, vẫn còn tồn tại các nhược điểm như sau:

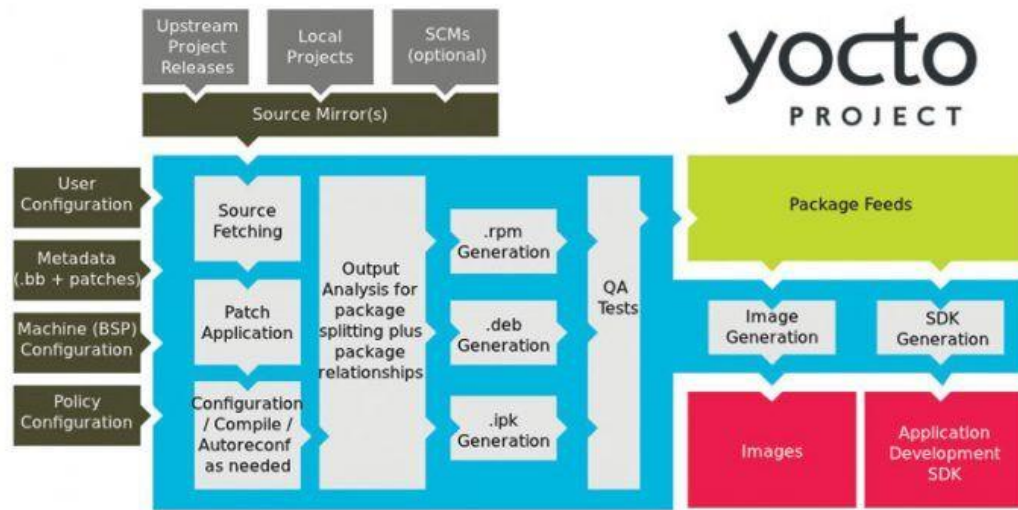
- Dung lượng tương đối lớn.
- Cộng đồng sử dụng KIT Orange Pi không đông đảo nên việc tìm hiểu, sửa lỗi tương đối khó khăn.
- Không hỗ trợ màn hình nên việc cài đặt thêm phần mềm VNC Viewer không có tác dụng nhiều.

3. Build Image bằng công cụ Yocto Project

3.1. Lý thuyết liên quan

Yocto hay còn có tên đầy đủ là 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. Một số điểm cần lưu ý về Yocto như sau:

- Yocto không phải là một bản distribution của hệ điều hành Linux
- Yocto không phải là build system
- Yocto không phải là framework
- Yocto không phải là tool

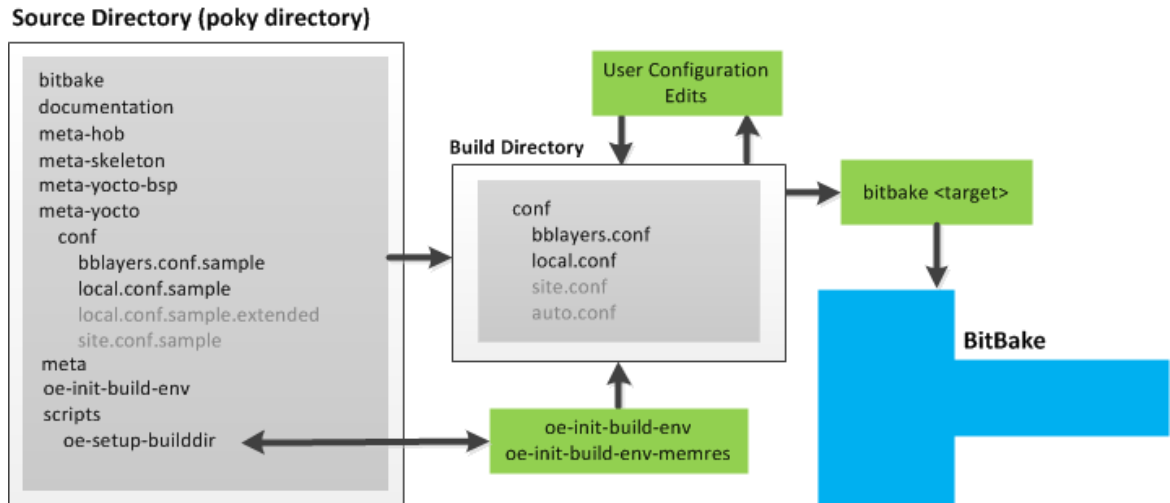


Hình 2. 9 Yocto Project

Yocto cho phép ta tạo ra một bản Linux có thể tùy biến được để người dùng có thể build cho nhiều mục đích khác nhau. Có thể nói Yocto sinh ra để giải quyết 2 vấn đề:

- Tìm xây dựng một distro phù hợp nhất với yêu cầu hệ thống sao cho ít tốn công sức nhất
- Tìm xây dựng một distro phù hợp nhất với yêu cầu hệ thống sao cho ít tốn công sức nhất

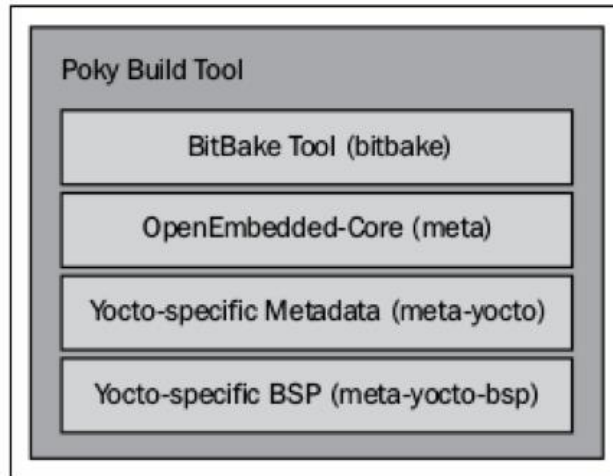
Và bản thân Yocto không có source, nó sẽ cung cấp các tool giống như một framework cho chúng ta phát triển tùy biến dễ dàng nhất có thể.



Hình 2. 10 Bitbake

Một công cụ cốt lõi của Yocto Project là Bitbake. Nó bao gồm một bộ thông dịch các script được viết trong các file recipe và thực hiện các lệnh trong đó. Nó mô tả lại và tự động hóa quá trình đưa ra một phần mềm vào một distro. Bitbake tạo ra một quy trình tự động là đầu vào là các file kịch bản, bitbake sẽ tự làm các công việc còn lại.

Yocto Project là được hợp thành từ nhiều dự án mã nguồn mở nhỏ hơn như Poky, BitBake và OpenEmbedded-Core. Ngoài các dự án chính này thì còn có nhiều dự án nhỏ khác nữa.



Hình 2. 11 Poky

Trong giai đoạn mới này, Poky (với tiền thân là một Linux distro) đã trở thành bản phân phối tham chiếu reference distribution của Yocto Project. 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. Poky là một giải pháp gần như là đầy đủ, nó cho phép tạo ra các Linux distro dành riêng cho phần cứng và tạo ra một bộ công cụ hỗ trợ phát triển phần mềm, hay còn gọi là SDK Software Development Kit dành riêng cho bản distro.

3.2. Triển khai build image bằng Yocto Project

3.2.1. Triển khai build image

Để build images cho KIT Orange Pi Zero bằng cách sử dụng Yocto Project, trước tiên ta cần cài đặt phần phụ thuộc cho Yocto như sau:

```
$ 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
```

Tiếp đó, ta cần git clone source code. Trong phần này, ta làm như sau:

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

Sau đó, mở rộng các module con bên ngoài như sau:

```
$ git submodule update --init
```

```
nhung@nhung-VirtualBox:~/orange-pi-distro$ git submodule update --init  
Submodule 'meta-openembedded' (git://git.openembedded.org/meta-openembedded) registered for path 'meta-openembedded'  
Submodule 'meta-qt5' (https://github.com/meta-qt5/meta-qt5.git) registered for path 'meta-qt5'  
Submodule 'poky' (git://git.yoctoproject.org/poky) registered for path 'poky'  
Cloning into '/home/nhung/orange-pi-distro/meta-openembedded'...  
Cloning into '/home/nhung/orange-pi-distro/meta-qt5'...  
Cloning into '/home/nhung/orange-pi-distro/poky'...  
Submodule path 'meta-openembedded': checked out 'a15d7f6ebcb0ed76c83c28f854d55e3f9d5b3677'  
Submodule path 'meta-qt5': checked out 'f22750291b224a3ee68456f0319ba18d428e197a'  
Submodule path 'poky': checked out '33e5b9e5adfc941ce65da8161c5cdf81fd1ed6c1'
```

Tiếp đó, với nguồn tệp source-me:

```
$ . source-me orange-pi-zero
```

```
nhung@nhung-VirtualBox:~/orange-pi-distro$ . source-me orange-pi-zero  
  
Command 'python2' not found, but can be installed with:  
  
sudo apt install python2  
  
OpenEmbedded requires 'python' to be python v2 (>= 2.7.3), not python v3.  
Please upgrade your python v2.  
Run 'bitbake <target>'  
Images:  
opiz-minimal  
opipcplus-minimal  
opipc-minimal  
opipc-qt5
```


Và xây dựng hình ảnh, build image như sau:

```
$ bitbake opiz-minimal
```

```
nhung@nhung-VirtualBox:~/orange-pi-distro/build$ bitbake opiz-minimal
/home/nhung/orange-pi-distro/poky/bitbake/lib/bb/fetch2/clearcase.py:159: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if command is 'mkview':
/home/nhung/orange-pi-distro/poky/bitbake/lib/bb/fetch2/clearcase.py:166: SyntaxWarning: "is" with a literal. Did you mean "=="?
  elif command is 'rmview':
/home/nhung/orange-pi-distro/poky/bitbake/lib/bb/fetch2/clearcase.py:170: SyntaxWarning: "is" with a literal. Did you mean "=="?
  elif command is 'setcs':
Parsing recipes: 100% |#####| Time: 0:03:14
Parsing of 1544 .bb files complete (0 cached, 1544 parsed). 2194 targets, 134 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION      = "1.40.0"
BUILD_SYS       = "x86_64-linux"
NATIVELSBSTRING = "ubuntu-20.04"
TARGET_SYS      = "arm-oe-linux-gnueabi"
MACHINE         = "orange-pi-zero"
DISTRO          = "opi"
DISTRO_VERSION  = "nodistro.0"
TUNE_FEATURES   = "arm armv7ve vfp neon vfpv4 callconvention-hard cortexa7"
TARGET_FPU      = "hard"
meta
meta-poky
meta-yocto-bsp  = "HEAD:33e5b9e5adfc941ce65da8161c5cdf81fd1ed6c1"
meta-oe        = "HEAD:a15d7f6ebcb0ed76c83c28f854d5e3f9d5b3677"
meta-qt5       = "HEAD:f22750291b224a3ee68456f0319ba18d428e197a"
meta-opi       = "master:0b7412f9ad645c6cfa72004892ff00cad921504"

Initialising tasks: 100% |#####| Time: 0:00:01
Sstate summary: Wanted 1082 Found 0 Missed 1082 Current 0 (0% match, 0% complete)
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
```

3.2.2. Boot bản image lên KIT Pi

Đầu tiên, ta cần flash bản image vừa build lên sdcard bằng lệnh sau:

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

Trong đó sdc là device, ta cần dùng lệnh dmesg - wH để kiểm tra.

Bước tiếp theo là nạp thẻ nhớ vào KIT Pi, và sử dụng phần mềm Putty để kết nối qua SSH.

3.3. Nhận xét

Đối với ưu điểm, có thể thấy:

- Dễ dàng để tiến hành thực hiện cài hệ điều hành.
- Có thể lựa chọn các option để bitbake khác nhau.
- Các tính năng đầy đủ để phục vụ cho tìm hiểu và học tập.

Bên cạnh những ưu điểm, vẫn còn tồn tại các nhược điểm như sau:

- Cần cài thêm package open-ssh để kết nối với KIT Pi.
- Tài liệu tham khảo, hướng dẫn cài đặt khá ít.
- Cộng đồng sử dụng KIT Orange Pi không đông đảo nên tìm sự hỗ trợ khá khó khăn như cần sử dụng Ubuntu 18.04 LTS và Python 2.7 để có thể build được image bằng công cụ yocto project

II. Lý thuyết và triển khai Linux Kernel Real-time

1. Lý thuyết về Linux Kernel Real-time

1.1. Scheduling với Real-Time

Kernel real-time tối ưu thiết kế kernel nhằm đặt độ trễ thấp nhất, thời gian đáp ứng của hệ thống và tạo ra tính xác định cho hệ thống. Từ đây thông qua bộ lập lịch thì ta có thể điều khiển thứ tự thực hiện luồng.

Trong linux luôn có một hàng đợi các thread đã sẵn sàng chạy và công việc của nó là lập lịch cho chúng trên CPU. Mỗi thread có một chính sách lập lịch (schedule policy) có thể được chia sẻ thời gian (time-shared) hoặc thời gian thực (real-time). Các thread chia sẻ thời gian có một giá trị nicess là tăng hoặc giảm quyền sử dụng của CPU. Các thread thời gian thực (real-time) có mức độ ưu tiên cao hơn thread sẽ đặt trước một thread có mức độ ưu tiên thấp hơn. Bộ lập lịch sẽ làm việc với các thread, không phải các process. Mỗi một thread được lập lịch bất kể nó đang chạy trong tiến trình nào.

Đối với giá trị nicess, một thread trở nên tốt hơn bằng cách giảm tải của nó lên trên hệ thống hoặc chuyển ngược lại hướng bằng cách tăng nó. Phạm vi của

giá trị nicess từ 19, thực sự tốt, cho đến -20, thực sự không tốt. Giá trị mặc định của nicess là 0, trung bình tốt.

Lập trình real-time có tính xác định (determinism) và tính fairness (công bằng). Cụ thể với tính xác định có nghĩa là các thread được thực hiện dựa vào độ ưu tiên. Còn đối với tính công bằng tức là mỗi thread có một khoảng thời gian thực hiện bằng nhau. Thread mang tính xác định sẽ được thực hiện trước các thread mang tính công bằng và mỗi thread mỗi khi nó được tạo ra sẽ được gán một sách lược lập lịch thuộc timeshare policy (non-real-time policy) hoặc real-time policy.

Time-shared polices (Non-Real-Time Policies)	Real-time policies
<ul style="list-style-type: none"> – SCHED_NORMAL (hay còn được gọi là SCHED_OTHER): thời gian động dựa trên nice value. Đây là chính sách mặc định và đa số các thread trong Linux đều dùng chính sách này. – SCHED_BATCH: tương tự như SCHED_NORMAL, ngoại trừ các thread được lập lịch với mức độ chi tiết hơn. Chúng chạy lâu hơn nên phải đợi lâu hơn cho đến khi được lập lịch lại. Mục đích là giảm số lượng chuyển đổi ngữ cảnh và giảm số lượng bộ nhớ cache của CPU bị xáo 	<ul style="list-style-type: none"> – SCHED_FIFO: Là quá trình First-In, First-Out theo thời gian thực. Khi bộ lập lịch chỉ định CPU cho tiến trình, nó mô tả vị trí hiện tại trong danh sách hàng đợi. Nếu không có tiến trình thời gian thực có mức độ ưu tiên cao hơn có thể chạy được thì tiến trình tiếp tục sử dụng CPU nếu nó muốn, ngay cả khi tồn tại các tiến trình có cùng mức độ ưu tiên có thể chạy được. Quyền ưu tiên tĩnh (từ 1 - 99), chỉ có thể mất CPU để có mức ưu tiên cao hơn hoặc phản ứng bị ngắt.

trộn. – SCHED_IDLE: Các thread chỉ được chạy khi không có thread nào từ bất kì chính sách khác đã sẵn sàng chạy. Đây là mức độ ưu tiên thấp nhất có thể.	– SCHED_RR: tương tự như SCHED_FIFO nhưng có lập lịch vòng tròn cho các nhiệm vụ có cùng mức ưu tiên. Khi bộ lập lịch chỉ định CPU cho tiến trình, nó mô tả tiến trình ở cuối danh sách hàng đợi. Chính sách này đảm bảo phân bổ thời gian công bằng thời gian CPU cho tất cả các quy trình thời gian thực có cùng mức ưu tiên – SCHED_DEADLINE: mức độ ưu tiên động dựa trên dealines.
---	--

Bảng 1. 3 So sánh sách lược lập lịch giữa Non-Real-time và Real-time

1.2. Locking

- Sử dụng pthread_mutex để khóa: Khóa mutex có tác dụng làm tránh tính xung đột giữa các thread. Nó đưa các thread vào trạng thái ngủ và nhường tài nguyên CPU cho các thread khác
- Kích hoạt tính kế thừa ưu tiên.
- Kích hoạt tính năng pthread_process_shared và pthread_process_robust.

1.3. Signalling

Khi cần đánh thức các thread đang ngủ bởi các khóa mutex, ta sẽ sử dụng các Conditional Variables (Biến điều kiện):

- Sử dụng pthread_cond objects để thông báo các tác vụ nếu tài nguyên được chia sẻ. Chúng có thể được liên kết với các pthread_mutex để cung cấp thông báo đồng bộ
- Không sử dụng các tín hiệu (chẳng hạn như POSIX timer hoặc hàm kill()). Chúng liên quan đến các bối cảnh không rõ ràng và có các hạn chế, không cung cấp bất kỳ đồng bộ hóa và khó để lập trình chính xác.
- Kích hoạt tính năng chia sẻ nếu biến điều kiện được truy cập bởi nhiều tiến trình trong cùng một chương trình
- Người gửi phải thông báo cho người nhận trước khi mở khóa liên kết với conditional variables

1.4. Clocks và Cyclic Tasks

Đối với CLOCK_MONOTONIC ta có

- Sử dụng các POSIX function cho phép xác định đặc tả clock
- Chọn CLOCK_MONOTONIC không thể thay đổi cũng như là đại diện cho một mốc thời gian kể từ tiến trình được khởi động. Điều này tốt hơn so với CLOCK_REALTIME. Về cơ bản, nếu không có sự thay đổi về mặt thời gian giữa lúc các tiến trình đang hoạt động. Tuy nhiên, CLOCK_REALTIME lấy thời gian thực mà thời gian thực có thể bị thay đổi người dùng, NTP.
- Sử dụng thời gian tuyệt đối. Tính toán thời gian tương đối dễ bị lỗi vì bản thân của việc thực hiện cần có thời gian.

1.5. Đánh giá một hệ thống Real-time

Thử đặt câu hỏi làm thế nào để có thể đánh giá một hệ thống thời gian thực?

Ta có thể sử dụng hai công cụ sau:

- Sử dụng công cụ cyclicttest (Một phần của rt-test package):
 - Đo lường, theo dõi độ trễ từ ngắt phần cứng đến userspace
 - Chạy ở mức độ ưu tiên để đánh giá
- Tạo tải cho hệ thống trong trường hợp xấu nhất:
 - Scheduling load: hackbench tool
 - Interrupt load
 - Serial/ network load
 - Các trường hợp tải khác nhau: stress-ng tool
 - Bộ điều khiển stress/ các thiết bị ngoại vi

2. Triển khai build Kernel Real-time

Đầu tiên ta cần tìm được bản patch real-time sao cho phù hợp với bản kernel mà ta đã build ở trên, cụ thể là bản patch-4.19-rt1.patch.

Sau đó, khi đã có được bản patch phù hợp thì ta cần tạo một bản meta-testlayer thông qua bitbake-layers, được thực hiện bằng lệnh sau:

```
bitbake-layers create-layer ../< meta-testlayer >
```

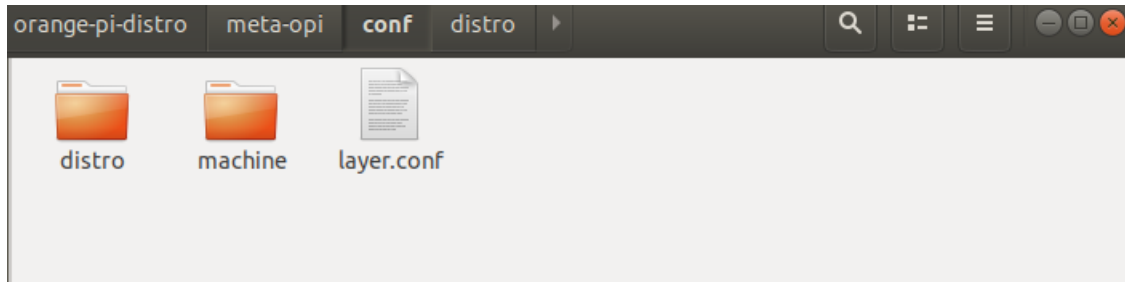
Sau đó, thực hiện add layers như sau:

```
bitbake-layers add-layer ../ meta-testlayer
```

Thêm layer vừa tạo, meta-testlayer vào trong bblayer.conf, bằng cách thêm đường dẫn:

```
/home/xxx/poky/meta-testlayer \
```

Tiếp đó, ta tìm tới folder recipes-kernel để được chứa file patch-4.19-rt1.patch. Sau đó, ta tìm đến theo đường dẫn /orange-pi-distro/meta-opi/conf thì ta thấy có layer.conf:



Tiếp đó, ta tìm theo đường dẫn meta-opi/recipes-kernel/linux/ và chỉnh sửa file linux-opiz_4.19.bb như dưới đây:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
```

```
SRC_URI += file://patch-4.19-rt1.patch
```

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

Cuối cùng, ta thực hiện lệnh sau để build:

```
$ bitbake opiz-minimal
```

Bước cuối cùng là ta tiến hành flash lại vào thẻ nhớ lần nữa:

```
$ sudo ./flash-sdcard.sh -d dev/dev/sdb -m orange-pi-zero -i opz-minimal
```

Tiến hành boot lên KIT Pi. Đăng nhập với user root và không cần password. Để kiểm tra kernel real-time hay không thì ta sử dụng lệnh uname -r như sau :

```
$ uname -r
```

```
orange-pi-zero login: root
root@orange-pi-zero:~# uname -r
4.19.0-rt1
```

Nếu kết quả thu được có chứa rt thì bản kernel mà ta đang có là bản kernel real-time.

III. Lý thuyết và triển khai viết ứng dụng C

1. Lý thuyết về lập trình multi-thread

Trong chương này ta sẽ tìm hiểu về luồng (thread) trong lập trình C, xây dựng các luồng với mục đích thu được thời gian hiện tại của hệ thống (ns), đồng thời xuất kết quả này ra file dạng .txt và tính toán sự chênh lệch giữa hai lần lấy mẫu.

Cụ thể trong phần này, ta sẽ nhắc đến các lý thuyết về lập trình multi thread, thread synchronization, sleep, shell script ...

Nếu trong một tiến trình có thể có nhiều nhánh thực hiện, các nhánh này có thể thực hiện độc lập và song song với nhau, khi đó ta gọi mỗi nhánh thực hiện trên là một luồng (thread). Thread là một thành phần của tiến trình, một tiến trình có thể chứa một hoặc nhiều thread và thread là một chuỗi điều khiển trong một tiến trình. Vậy nên nếu không có thread nào được tạo thêm thì tiến trình đó được gọi là **single-thread**, ngược lại nếu có thêm thread thì được gọi là **multi-thread**.

Ta thấy thao tác tạo ra thread nhanh hơn so với tạo tiến trình. Hơn thế nữa, thời gian hệ thống chuyển từ thread này sang thread khác nhanh hơn từ tiến trình này sang tiến trình khác. Và giao tiếp (đồng bộ) giữa các thread trong cùng một tiến trình có thể thực hiện dễ dàng hơn so với giữa các tiến trình do các thread cùng nhau chia sẻ tài nguyên của tiến trình đó. Bên cạnh đó, việc thiết kế, xây dựng

multi-thread cần tỉ mỉ cẩn thận bởi các thread có khả năng sử dụng tài nguyên chung của tiến trình và gỡ rối một tiến trình đa luồng phức tạp hơn so với tiến trình đơn luồng.

Để xây dựng tiến trình đa luồng trên Linux ta cần sử dụng thư viện pthread, tức là thêm `#include<pthread>` và khi biên dịch cần thêm tham số `-lpthread`.

- Tạo thread:

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

Return 0 on success, or a positive error number on error

- Kết thúc thread:

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

- Chờ một thread kết thúc

```
int pthread_join(pthread_t thread, void **retval);
```

Return 0 on success, or a positive error number on error

- Đồng bộ thread bằng mutex

Mutex cung cấp cơ chế khóa (lock) một đối tượng để đảm bảo tại một thời điểm chỉ có một thread thực hiện đoạn mã của mình. Có hai cách khởi tạo một mutex:

- Khởi tạo tĩnh:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- Khởi tạo động:

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t  
*attr);
```

Sau khi khởi tạo, mutex được khóa và mở bỏ hai hàm dưới đây:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Both return 0 on success, or a positive error number on error

Ngoài ra, để giải phóng một biến mutex ta sử dụng hàm:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Thread Synchronization (Biến điều kiện) cho phép một thread đăng ký đợi (đi vào trạng thái ngủ) cho đến khi một thread khác gửi một signal đánh thức nó dậy để chạy tiếp.

- Khai báo biến điều kiện: Có hai kiểu là tĩnh và động. Đối với khai báo tĩnh:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Còn khai báo động:

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
```

Khi không sử dụng ta có thể hủy bằng cách gọi hàm:

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

– Sleep:

Hàm **nanosleep()** thực hiện nhiệm vụ tương tự như **sleep()**, nhưng có một ưu điểm là nó cần chỉ định một khoảng thời gian ngủ (sleep interval):

```
int nanosleep(const struct timespec *request, struct timespec *remain);
```

Giống như **nanosleep()** thì **clock_nanosleep()** sẽ tạm dừng quá trình gọi cho đến khi một khoảng thời gian cụ thể trôi qua hoặc một tín hiệu đến.

```
int clock_nanosleep(clockid_t clockid, int flags,  
const struct timespec *request, struct timespec *remain);
```

*Return 0 successfully completed sleep,
or a positive error number on error or interrupted sleep*

– Shell script

Shell là một chương trình thông dịch lệnh của một hệ điều hành, cung cấp khả năng tương tác với hệ điều hành bằng cách gõ từng lệnh ở chế độ dòng lệnh, đồng thời trả lại kết quả thực hiện lệnh lại cho người sử dụng. Shell script cung cấp tập hợp các lệnh đặc biệt mà từ đó có thể tạo nên chương trình. Shell được bắt đầu khi người dùng đăng nhập hoặc khởi động terminal.

Vậy thử đặt câu hỏi là tại sao lại cần shell script? Bởi khi có shell script ta có thể tránh các công việc tự động hóa và lặp đi lặp lại, đồng thời

giám sát hệ thống và system admins sử dụng shell script để sao lưu thường xuyên.

Shell script có một vài ưu điểm như sau:

- Viết Shell script sẽ nhanh hơn;
- Lệnh và cú pháp hoàn toàn giống với lệnh được nhập trực tiếp trong dòng lệnh.

Bên cạnh đó, vẫn còn một vài nhược điểm:

- Tốc độ thực hiện chậm
- Không phù hợp cho các task lớn và phức tạp;
- Dễ xảy ra lỗi tốn kém, một lỗi duy nhất có thể thay đổi lệnh có thể gây hại.

2. Lý thuyết về tối ưu code realtime

Trong phần này, ta sẽ tối ưu thuật toán cho chương trình ban đầu bằng phương pháp lập lịch như sau:

- Thêm các schedule policy cho các thread
- Synchronization bằng mutex
- Sử dụng các signal
- Clock

3. Triển khai yêu cầu

3.1. Thread SAMPLE

Thread **SAMPLE** thực hiện vô hạn lần đọc thời gian hệ thống hiện tại (chính xác đến đơn vị ns) vào biến T với chu kỳ **X** ns.

```
void * sample_func(void *arg) {
```

```
clock_gettime(CLOCK_MONOTONIC, &rqst);

while(1) {

    rqst.tv_nsec += T;

    if(rqst.tv_nsec > 1000*1000*1000) {
        rqst.tv_nsec -= 1000*1000*1000;
        rqst.tv_sec++;
    }

    clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &rqst,
NULL);

    clock_gettime(CLOCK_MONOTONIC, &rtc);

}

return NULL;
}
```

3.2. Thread INPUT

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.

```
void * input_func(void *arg) {  
  
    file2 = fopen("freq.txt","r");  
  
    while(1) {  
  
        char buff[100];  
  
        fgets(buff,sizeof(buff),file2);  
  
        char *eptr;  
  
        T = strtol(buff,&eptr,10);  
  
    }  
  
    fclose(file2);  
  
    return NULL;  
  
}
```

3.3. Thread LOGGING

Thread **LOGGING** chờ khi biến T được cập nhật mới, giá trị interval (offset giữa biến T hiện tại và biến T của lần ghi trước) ra file có tên “**time_and_interval.txt**”.

```

void  *logging_func(void* arg) {

    file1 = fopen("time_and_interval.txt","a+");

    while(1) {

        if(rtc.tv_nsec != ot.tv_nsec ||rtc.tv_sec != ot.tv_sec ) {

            if(file1 == NULL) {

                printf("Unable to open file\n");

                return NULL;

            }

            long diff_sec = (long)rtc.tv_sec - (long)ot.tv_sec ;

            long diff_nsec;

            if(rtc.tv_nsec > ot.tv_nsec) {

                diff_nsec = rtc.tv_nsec - ot.tv_nsec;

            }

            else {

                diff_nsec = 1000000000 - ot.tv_nsec + rtc.tv_nsec;

                diff_sec = diff_sec - 1;

            }

```

```
if(ot.tv_nsec != 0) {  
  
    fprintf(file1, "%ld\n",diff_nsec);  
  
}  
  
ot.tv_nsec = rtc.tv_nsec;  
  
ot.tv_sec = rtc.tv_sec;  
  
}  
  
}  
  
fclose(file1);  
  
return NULL;  
  
}
```

3.4. Phần code tối ưu

3.4.1. Thread SAMPLE

```
void * sample_func(void *arg) {  
  
    struct sched_param sp;  
  
    sp.sched_priority = 70;  
  
    pthread_setschedparam(pthread_self(), SCHED_FIFO, &sp);  
  
    clock_gettime(CLOCK_MONOTONIC, &rqst);  
  
    while(1) {
```



```

    rqst.tv_nsec += T;

    if(rqst.tv_nsec > 1000*1000*1000) {

        rqst.tv_nsec -= 1000*1000*1000;

        rqst.tv_sec++;

    }

    clock_nanosleep(CLOCK_MONOTONIC,    TIMER_ABSTIME,    &rqst,
NULL);

    clock_gettime(CLOCK_MONOTONIC, &rtc);

    }

    return NULL;

}

```

3.4.2. Thread INPUT

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.

```

void * input_func(void *arg) {

    struct sched_param sp;

    sp.sched_priority = 10;

    pthread_setschedparam(pthread_self(), SCHED_FIFO, &sp);

```

```

    file2 = fopen("freq.txt","r");

    while(1) {

        char buff[100];

        fgets(buff,sizeof(buff),file2);

        char *eptr;

        T = strtol(buff,&eptr,10);

    }

    fclose(file2);

    return NULL;

}

```

3.4.2. Thread LOGGING

Thread **LOGGING** chờ khi biến T được cập nhật mới, giá trị interval (offset giữa biến T hiện tại và biến T của lần ghi trước) ra file có tên “**time_and_interval.txt**”.

```

void  *logging_func(void* arg) {

    struct sched_param sp;

    sp.sched_priority = 40;

    pthread_setschedparam(pthread_self(), SCHED_FIFO, &sp);

```

```

file1 = fopen("time_and_interval.txt", "a+");

while(1) {

    if(rtc.tv_nsec != ot.tv_nsec || rtc.tv_sec != ot.tv_sec ) {

        if(file1 == NULL)

            {

                printf("Unable to open file\n");

                return NULL;

            }

        long diff_sec = (long)rtc.tv_sec - (long)ot.tv_sec ;

        long diff_nsec;

        if(rtc.tv_nsec > ot.tv_nsec)

            {

                diff_nsec = rtc.tv_nsec - ot.tv_nsec;

            }

        else

            {

                diff_nsec = 1000000000 - ot.tv_nsec + rtc.tv_nsec;

                diff_sec = diff_sec - 1;

            }

```

```
        if(ot.tv_nsec != 0)
        {
            fprintf(file1, "%ld\n",diff_nsec);
        }

        ot.tv_nsec = rtc.tv_nsec;

        ot.tv_sec = rtc.tv_sec;

    }

}

fclose(file1);

return NULL;

}
```

C. KẾT QUẢ VÀ NHẬN XÉT

Trong phần này ta thực hiện chạy chương trình C đã triển khai trên KIT Orange Pi Zero. Các test case dưới đây là được thực hiện trên phần cứng là KIT Orange Pi Zero, chương trình được chạy trên Kernel thường và Kernel Real-time và chu kỳ lấy mẫu ta chọn là 100000 ns. Đồng thời ta chạy thêm file disturb.c, file này có mục đích là tạo ra các tiến trình, các tiến trình này sẽ là các vòng lặp lồng nhau. Từ đó, nó sẽ chiếm CPU khi nó đang chạy, khiến cho CPU không thể hoàn thành chương trình C trên Linux của chúng ta trong thời gian ngắn.

Cụ thể file disturb.c như sau:

```
#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

int disturb(int id)
{
    int i, j, temp;

    for(int j = 0; j < 200000; j++)
    {
        for(int i = 0; i < 200000; i++)
        {
            temp = temp + i;
        }
    }

    exit(0);
}

int main(int argc, char **argv)
{
    int i;
```

```
pid_t pid;

if(argc != 2)

{

    printf("Supply with one number (number of disturb to run)\n");

}

else

{

    for(int i = 0; i < atoi(argv[1]); i++)

    {

        pid = fork();

        if(!pid)

        {

            disturb(i);

        }

    }

    printf("\n");

}

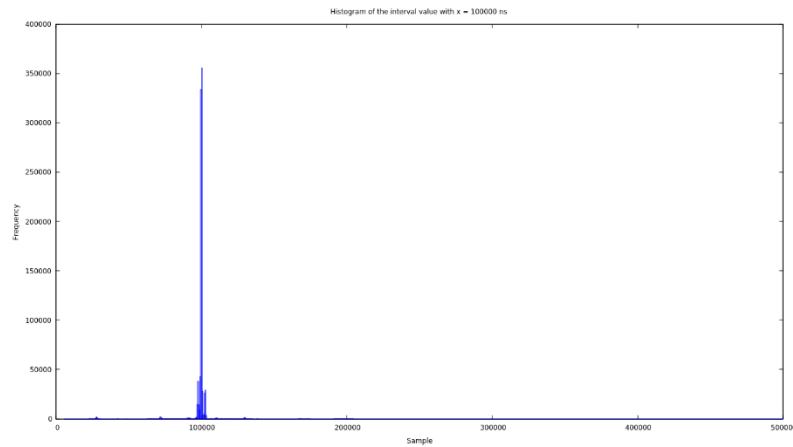
return 0;

}
```

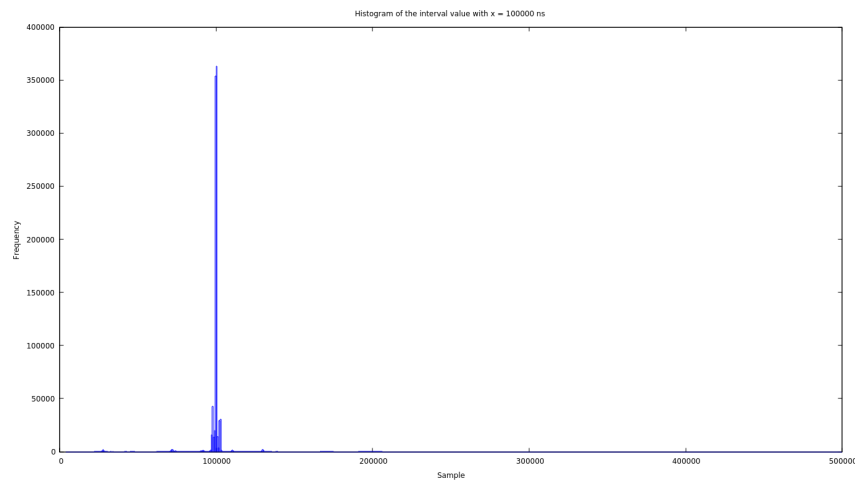
I. Mô tả các test case

1. Test case 1

Trong phần này, ta thực hiện chạy chương trình trên Kernel thường và Kernel Real-time nhưng chưa kèm disturb. Kết quả được thể hiện tương ứng trong hình 3.1 và 3.2 dưới đây:



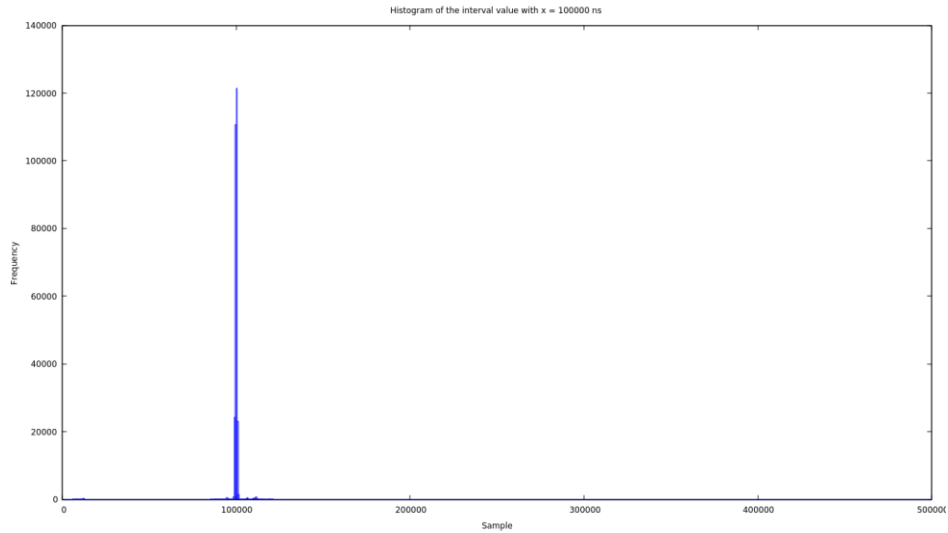
Hình 3. 1 Biểu đồ phân bố giá trị interval trên Kernel thường



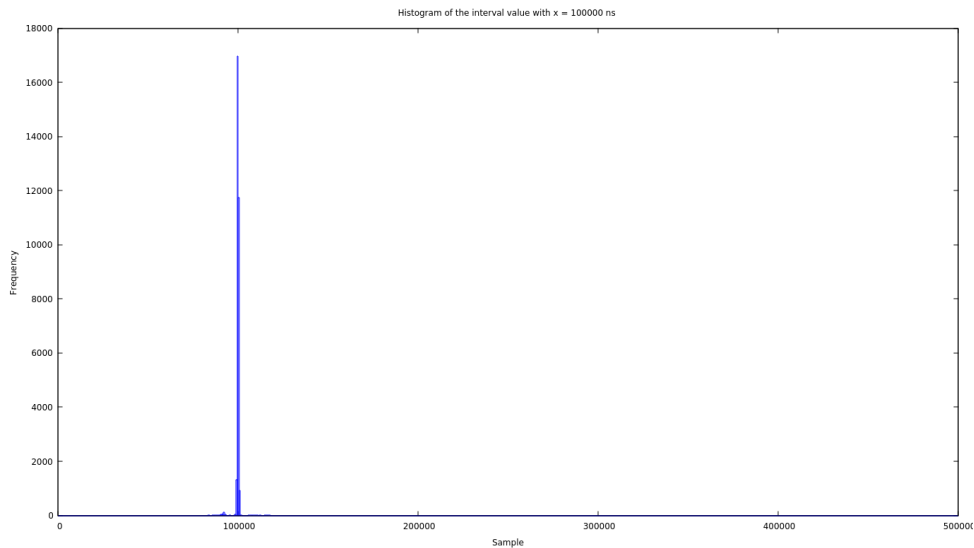
Hình 3. 2 Biểu đồ phân bố giá trị interval trên Kernel Real-time

2. Test case 2

Đối với test case thứ hai này, ta thực hiện chạy chương trình lên Kernel thường và Kernel Real-time và kèm theo file disturb. Dưới đây là hình 3.3 và hình 3.4 mô tả kết quả của test case tương ứng với Kernel thường và Kernel Real-time:



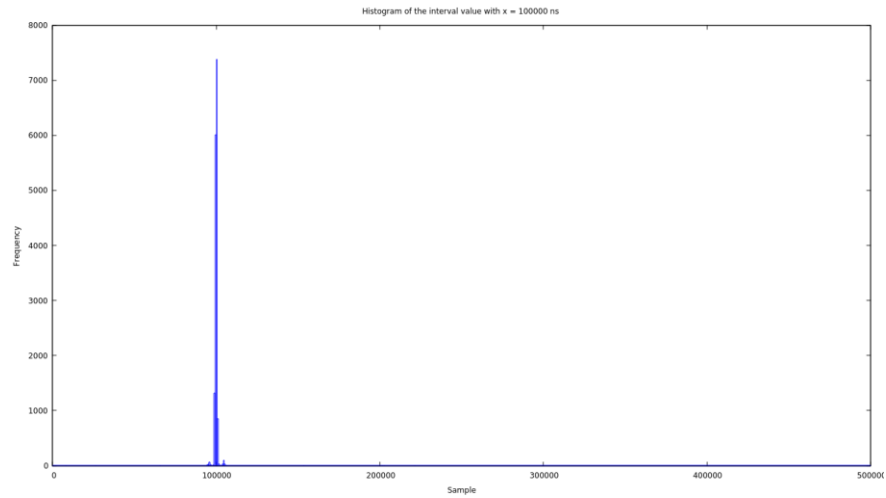
Hình 3. 3 Biểu đồ phân bố giá trị interval trên Kernel thường cùng file disturb



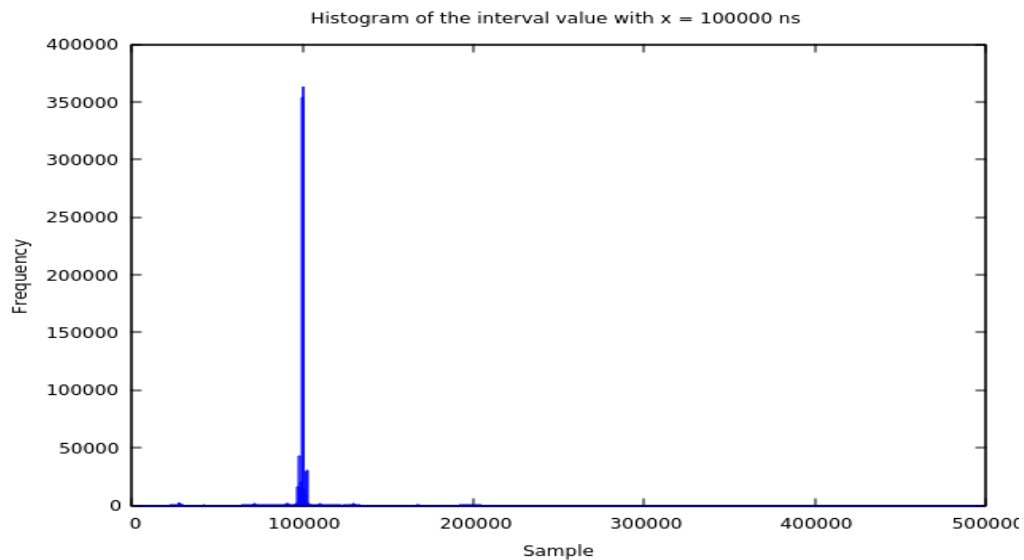
Hình 3. 4 Biểu đồ phân bố giá trị interval trên Kernel Real-time cùng file disturb

3. Test case 3

Đối với test case này thì ta chạy chương trình được tối ưu nhờ lập lịch trên Kernel thường và Kernel Real-time. Dưới đây là hình 3.5 và hình 3.6 mô tả kết quả của test case tương ứng với Kernel thường và Kernel Real-time:



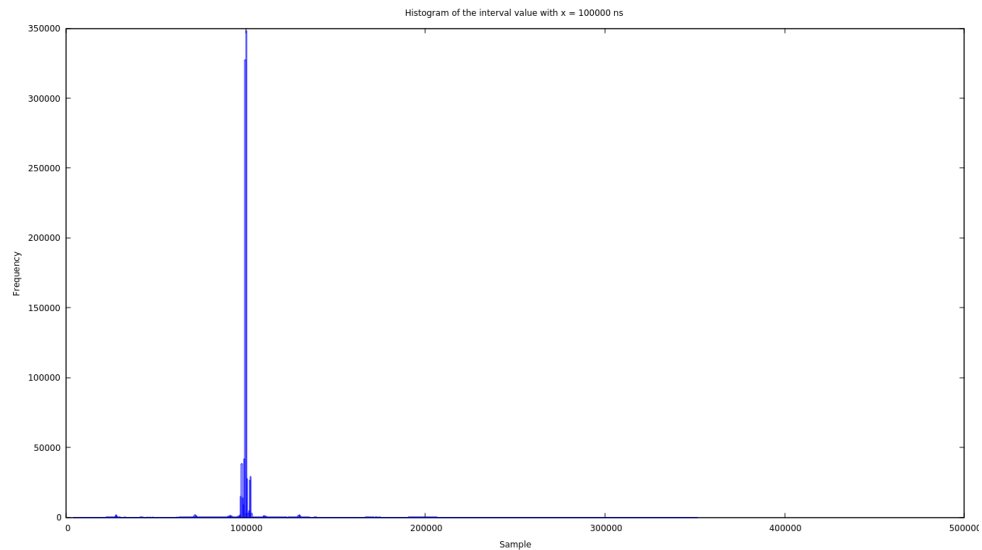
Hình 3. 5 Biểu đồ phân bố giá trị interval khi code tối ưu trên Kernel thường



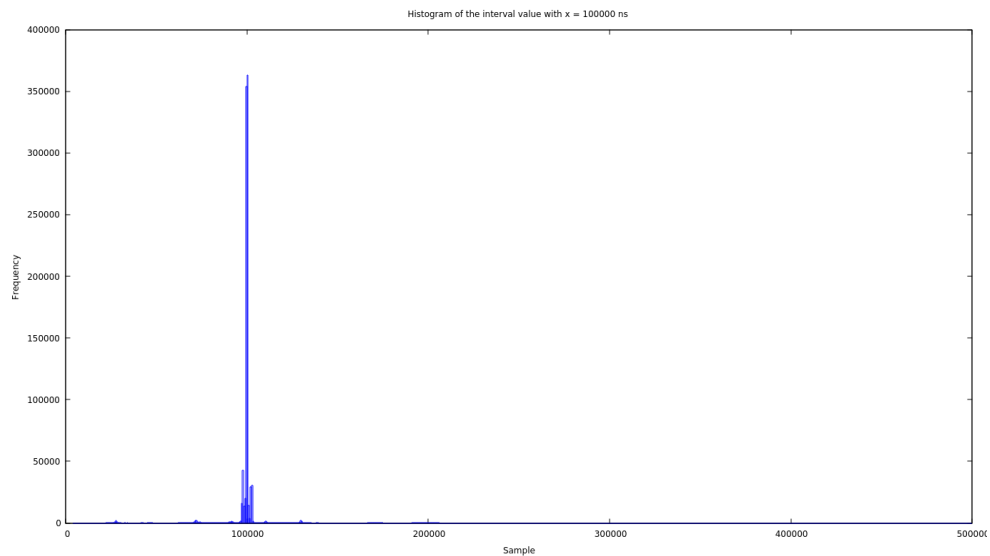
Hình 3. 6 Biểu đồ phân bố giá trị interval khi code tối ưu trên Kernel Real-time

4. Test case 4

Đối với test case này thì ta chạy chương trình được tối ưu nhờ lập lịch trên Kernel thường và Kernel Real-time và kèm theo file disturb. Kết quả được thể hiện tương ứng trong hình 3.7 và 3.8 dưới đây:



Hình 3. 7 Biểu đồ phân bố giá trị interval khi code tối ưu trên Kernel thường cùng file disturb



Hình 3. 8 Biểu đồ phân bố giá trị interval khi code tối ưu trên Kernel Real-time cùng file disturb

II. Nhận xét và đánh giá kết quả

Đối với mỗi test case ở phần trên, ta đưa ra nhận xét và đánh giá các kết quả thu được như sau:

1. Lần chạy ứng dụng ban đầu

Khi so sánh kết quả thu được giữa Kernel thường và Kernel Real-time ta thấy rằng khá giống với yêu cầu đã đặt ra. Thật vậy, điều này được thể hiện rõ trong hình 3.1 và hình 3.2 tương ứng với Kernel mặc định và Kernel Real-time.

2. Lần chạy ứng dụng ban đầu cùng ứng dụng disturb*10

Khi thực hiện chạy file disturb.c *10 và code trong phần này chưa được tối ưu bằng phương pháp lập lịch thì Kernel Real-time cho kết quả tương đối tốt hơn so với kernel thường. Cụ thể khi quan sát cả hai hình 3.3 và hình 3.4 thì kernel thường thì các giá trị interval vẫn nằm xung quanh giá trị 100000 ns. Còn đối với số lượng mẫu 100000 ns của Kernel Real-time giảm khi không có disturb.

3. Lần chạy ứng dụng được tối ưu

Khi so sánh kết quả thu được giữa Kernel thường và Kernel Real-time khi chương trình được tối ưu bằng phương pháp lập lịch thì ta thấy rằng khá giống với yêu cầu đã đặt ra. Thật vậy, ta có thể thấy điều này được thể hiện rõ trong hình 3.5 và hình 3.6 tương ứng với Kernel mặc định và Kernel Real-time.

4. Lần chạy ứng dụng được tối ưu cùng disturb*10

Với phần code tối ưu bằng phương pháp lập lịch này khi ta chạy đồng thời thêm file disturb.c *10 thì thấy rằng kết quả trên Kernel Real-time được cải thiện. Tại hình 3.7, kết quả khi thực hiện chương trình ở Kernel thường ta thấy kết quả đã cải thiện so với khi chưa tối ưu chương trình. Cụ thể khi quan sát hai hình 3.7 và 3.8 ta thấy số lượng giá trị interval tại giá trị 100000 ns ở Kernel Real-time cao hơn so với Kernel thường.

5. Kết luận

Từ các nhận xét trên ta có thể đưa ra đánh giá rằng kết quả thu mà ta thu được giống với lý thuyết đã nêu ra. Ta thấy được sự khác biệt khi chạy chương trình tối ưu cùng thêm file `disturb.c` trên Kernel mặc định và Kernel Real-time. Từ đây, ta có thể thấy nhờ có phương pháp lập lịch kết quả được cải thiện cả trên Kernel thường và Kernel Real-time.

D. KẾT QUẢ THU ĐƯỢC SAU QUÁ TRÌNH THỰC TẬP

Tổng hợp lại các công việc chính mà em được thực hiện trong kỳ thực tập như sau:

- Làm bài tập lập trình C trên Linux và thực hiện tối ưu bằng phương pháp lập lịch.
- Cài đặt hệ điều hành cho KIT Pi.
- Sử dụng công cụ Yocto Project để build kernel cùng các images liên quan để boot lên mạch.
- Porting Kernel Real-time cho KIT Pi.
- Tiến hành benchmarking hiệu năng của kernel, so sánh Kernel thường và Kernel Real-time.

Sau quá trình được thực tập tại VHT, em đã:

- Có cơ hội được làm việc với các nền tảng chip nhúng là máy tính Pi, cụ thể là KIT Orange Pi Zero.
- Tìm hiểu các kiến thức về lập trình C trên Linux, các cách tối ưu thuật toán, các kiến thức về Yocto Project và Kernel Real-time. Nói khác là được làm quen với quy trình thiết kế và phát triển một hệ thống nhúng Linux.

- Ngoài ra, được tiếp cận với môi trường làm việc thực tế tại Tổng Công ty Công nghiệp Công nghệ cao Viettel, Hòa Lạc.

E. TÀI LIỆU THAM KHẢO

- [1] Michael Kerrisk, “The Linux Programming Interface” (2010).
- [2] Frank Vasquez, Chris Simmonds , “Mastering Embedded Linux Programming Create fast and reliable embedded solutions with Linux 5.4 and the Yocto Project 3.1” (Dunfell)-Packt Publishing (2021).
- [3] Daniel P.Bovet & Marco Cesati, “Understanding the Linux Kernel”, 3rd Edition .
- [4] Nguyễn Trí Thành, “Giáo trình lập trình Linux nâng cao” (2014).
- [5] John Ogness, “A Checklist for Writing Linux Real-Time Applications” (2020).
- [6] Bootlin, “Understanding Linux real-time with PREEMPT_RT” (2022).