Chương I Giới thiệu chung

Thời gian thực (Real-Time) là gì?

Thời gian thực rất khó định nghĩa. Ý tưởng cơ bản của thời gian thực thể hiện ở chỗ, một hệ thống phải có những phản ứng thích hợp, đúng thời điểm với môi trường của nó. Nhiều người luôn nghĩ rằng, thời gian thực có nghĩ là thực sự nhanh, càng nhanh càng tốt, điều này là sai lầm. Thời gian thực có nghĩa "đủ nhanh" (fast enough) trong một ngữ cảnh, một môi trường mà hệ thống đang hoạt động. Khi chúng ta đề cập đến máy tính điều khiển động cơ ô tô, chúng ta cần nó chạy càng nhanh càng tốt.

Một ví dụ khác, khi chúng ta đề cập đến một nhà máy lọc dầu hoá học chẳng hạn, nhà máy được điều khiển bởi một hoặc một hệ thống máy tính. Các máy tính này có trách nhiệm điều khiển quá trình hoá học đồng thời phải phát hiện ra được các sự cố có thể xảy ra. Tuy nhiên, các phản ứng hay các quá trình hoá học thường có hằng số thời gian khá lớn từ hàng giây cho tới hàng phút là ít. Chính vì thế mày chúng ta có thể giả thiết rằng máy tính hoàn toàn có khả năng phản ứng lại các sự cố nghiêm trọng. Tuy nhiên, đặt vấn đề là nếu hệ thống máy tính đó đang trong quá trình in một bản báo cáo dài về các thông số sản lượng của tuần trước thì đột nhiên trục trặc xảy ra. Vậy thì nó mất bao nhiêu thời gian để có thể phản ứng lại các sự cố như thế?

Thực chất của việc tính toán thời gian thực không chỉ ở việc phản ứng đủ nhanh mà còn phải đáng tin cậy và chính xác. Máy tính điều khiển động cơ trong ô tô của bạn phải có thể điều chỉnh luồng nhiên liệu và thời gian đánh lửa một cách hợp lý trong mỗi vòng quay. Nếu không, động cơ sẽ không làm việc theo mong muốn. Máy tính trong nhà máy lọc dầu phải có thể phát hiện và phản ứng lại các điều kiện bất thường trong thời gian cho phép để có thể tránh được các thảm hoạ có thể xảy ra.

Như vậy, nghệ thuật của lập trình thời gian thực chính là việc thiết kế hệ thống sao cho nó có thể tiếp nhận một cách chính xác các ràng buộc về mặt thời gian trong suốt quá trình các sự kiện ngẫu nhiên và không đồng bộ xảy ra.

Thời gian thực và các dạng của nó

Về cơ bản, chương trình có tính thời gian thực phải có khả năng phản ứng lại các *sự kiện* trong môi trường mà hệ thống làm việc trong khoảng thời gian nhất định cho trước. Những hệ thống như vậy được gọi là hệ thống "điều khiển sự kiện" (hay hệ thống lái sự kiện – event-driven) và có thể được mô tả bằng thời gian trễ từ khi mà sự kiện xảy ra cho tới khi hệ thống có hoạt động phản ứng lại với sự kiện đó.

Thời gian thực, mặt khác, đòi hỏi một giới hạn cao hơn về thời gian trễ, được gọi là "*thời hạn lập danh mục*" (*scheduling deadline*). Một hệ thống thời gian thực có thể được chia làm 2 loại. Thời "*gian thực cứng*" và thời "*gian thực mềm*" (*hard* real-time và *soft* real-time, tôi xin gọi theo đúng từ nguyên bản trong tiếng Anh về sau này). Trong hệ thống hard real-time, hệ thống phải tiếp nhận và nắm bắt được scheduling deadline của nó tại mỗi và mọi thời điểm. Sự sai sót trong việc tiếp nhận deadline có thể dẫn đến hậu quả nghiêm trọng thậm chí chết người. Lấy ví dụ, máy hỗ trợ nhịp tim cho bệnh nhân khi phẫu thuật. Thuật toán điều khiển phụ thuộc vào thời gian nhịp tim của người bệnh, nếu thời gian này bị trễ, tính mạng của người bệnh sẽ bị ảnh hưởng.

Đối với khái niệm soft real-time, scheduling deadline có dễ thở hơn chút ít. Chúng ta mong muốn hệ thống phản ứng lại các sự kiện trong thời gian cho phép nhưng không có gì thực sự nghiêm trọng xảy ra nếu hệ thống thỉnh thoảng bị trễ. Lỗi về mặt thời gian có thể chỉ đơn giản là dẫn đến hậu quả giảm độ tin cậy của đối tượng đối với hệ thống mà không có hậu quả thê thảm nào khác xảy ra. Mạng lưới thu ngân tự động của ngân hàng là ví dụ rõ nhất cho soft real-time. Mạng rút tiền tự động ATM là hệ thống thời gian thực? Chẳng ai dám đặt cược cả. Khi bạn đưa thẻ ATM vào máy, bạn mong là máy sẽ phản ứng lại trong vòng 1 hay 2 giây. Nhưng nếu nó lâu hơn thế, điều tồi tệ nhất có thể xảy ra là… bạn sốt ruột và thấy khó chịu đối với cái máy đó.

Trên thực tế có rất nhiều hệ thống phối hợp cả 2 loại trên, trong đó, một phần nào đó của hệ thống làm việc dựa trên hard real-time, một số phần khác lại dựa trên soft real-time.

Chương sau của bài viết này sẽ nói về đặc tính đầu tiên và cũng quan trọng nhất của lập trình thời gian thực. Đó là Polling và Interrupt.

CHƯƠNG II:
Polling và Interrupt

Một hệ thống thời gian thực được gọi là "điều khiển sự kiện" có nghĩa là hệ thống đó phải có chức năng chính là phản ứng lại các sự kiện xảy ra trong môi trường của hệ thống. Vậy thì hệ thống phản ứng lại các sự kiện như thế nào. Hiện nay có hai phương pháp tiếp cận vấn đề này. Phương pháp đầu tiên là *Polling* hay *Vòng lặp Polling*.

```
int main (void)
{
sys_init();
while (TRUE)
{
        if (event_1)
            service_event_1();
        if (event_2)
            service_event_2();
            .
            .
        if (event_n)
            service_event_n();
    }
}
```

Hình 1 – Vòng lặp Polling

Hãy xem đoạn code trong hình 1. Chương trình được bắt đầu bằng một vài cài đặt ban đầu cho hệ thống rồi truy cập vào trong một vòng lặp vô hạn, trong đó, các sự kiện mà hệ thống có thể phản ứng lại được kiểm tra. Khi có một sự kiện xảy ra, dịch vụ phản ứng lại sự kiện đó được kích hoạt.

Tiến trình thực hiện của vòng lặp trên tỏ ra khá đơn giản và thích hợp với những hệ thống nhỏ không đòi hỏi quá gắt gao về mặt thời gian. Tuy nhiên, cũng có một số vấn đề cần bàn đến:

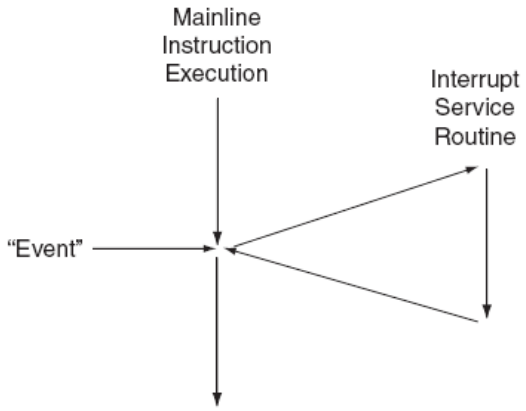
- Thời gian phản ứng lại sự kiện phụ thuộc rất lớn vào vị trí mà chương trình đang thực hiện trong vòng lặp. Lấy ví dụ: Nếu sự kiện event_1 xảy ra ngay trước câu lệnh if(event_1) thì thời gian phản ứng là rất ngắn. Nhưng nếu sự kiện event_1 xảy ra ngay sau khi câu lệnh kiểm tra đó, chương trình lúc này phải quét toàn bộ vòng lặp và quay trở về điểm đầu và thực hiện dịch vụ của sự kiện event_1.
- Và cũng là một hệ quả tất yếu, thời gian phản ứng cũng là một hàm của số lượng sự kiện được kích hoạt tại một thời điểm và sau đó là thời gian thực hiện các dịch vụ trong một lần quét vòng lặp của chương trình.

- Tất cả các sự kiện được chương trình đối xử một cách bình đẳng và không có sự ưu tiên nào cả.
- Khi có một đặc tính mới, do đó là dịch vụ mới, được thêm vào chương trình, thời gian phản ứng lại dài thêm ra.

Phương pháp tiếp cận thứ 2 là *ngắt* (*Interrupt*). Rất hữu dụng và cũng gây không ít khó khăn cho người lập trình. Ý tưởng của ngắt: sự xuất hiện của một sự kiện có thể "ngắt" tiến trình thực hiện của chương trình, "nhồi" thêm và thực hiện một tiến trình khác vào như hình 2. Khi tiến trình nhồi thêm được thực hiện xong, chương trình chính lại được thực hiện tiếp từ thời điểm bị ngắt. Tiến trình của sự kiện ngắt được thực hiện ngay lập tức mà không phải quan tâm đến chương trình chính. Những tiến trình như thế người ta gọi là "*chương trình con dịch vụ ngắt*" (*Interrupt Service Routine*) viết tắt *ISR*.

Các thế hệ vi xử lý hiện nay thường thực hiện 3 loại ngắt khác nhau

- Câu lệnh INT, hay nhiều khi còn được nhắc đến với cái tên là TRAP (bẫy). Nó như một câu lệnh để gọi một chương trình con đặc biệt. Chúng ta sẽ đề cập đến nó sau.
- Các trường hợp đặc biệt của bộ xử lý. Các điều kiện lỗi như lỗi chia 0, lỗi truy cập bất hợp pháp vào bộ nhớ có thể được điều khiển thông qua cơ chế ngắt.



Hình 2 - Ngắt

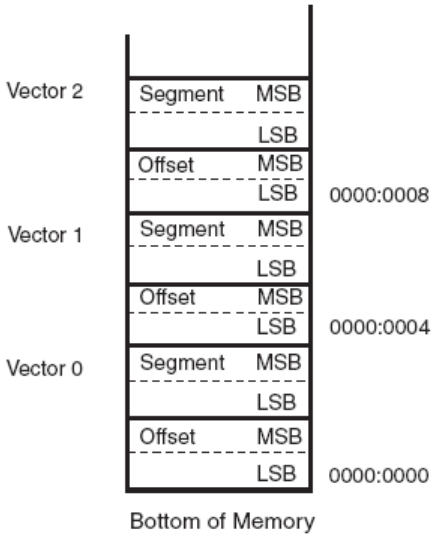
Hai loại ngắt kể trên đồng bộ với việc thực hiện lệnh. Trong đó, INT chính là một câu lệnh và các lỗi đặc biệt của bộ xử lý chính là kết quả trực tiếp của việc thực hiện lệnh.

Loại ngắt thứ 3 được tạo ra bởi sự kiện xảy ra bên ngoài bộ xử lý. Loại ngắt này được tạo ra bởi các I/O phần cứng và xảy ra không đồng bộ với việc thực hiện lệnh.

Các ngắt ngoài không đồng bộ này:

- Làm tối đa hoá hiệu suất và thông lượng của hệ thống máy tính
- Gây ra phần lớn các lỗi và rắc rối cho người lập trình

Hầu hết các bộ xử lý đều sử dụng lược đồ ngắt giống nhau. Hình 3 chỉ ra kiến trúc ngắt của Intel x86. 1kilo byte đầu tiên của bộ nhớ được giành cho *bảng véctơ ngắt* (*Interrupt Vector Table*). Mỗi một véctơ có 4 byte thể hiện địa chỉ (segment và offset) của chương trình con dịch vụ ngắt. Các véctơ này mang những ý nghĩa, chức năng khác nhau và được định nghĩa bởi kiến trúc của bộ xử lý. Ví dụ: véctơ 0 là lỗi chia 0, véctơ 3 là breakpoint (lệnh ngắt INT 1byte)…



Hình 3 - Ngắt - Bảng véctơ ngắt

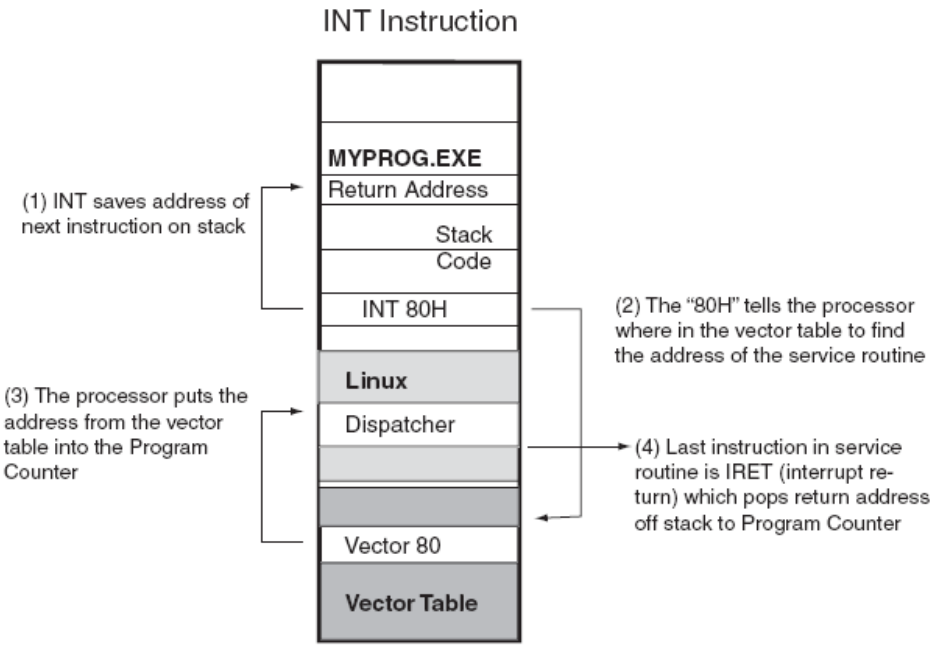
Một số véctơ được dành cho các ngắt ngoài. Trong PC, các véctơ 8 đến 15 và 0x70 đến 0x77 được dành cho phần cứng.

Tất cả các véctơ được truy cập thông qua lệnh ngắt INT 2byte, trong đó, byte thứ 2 chỉ ra số thứ tự của véctơ (ngắt). Phần mềm hệ thống thường thiết lập các quy ước liên quan đến nhiều véctơ này. Ví dụ: PC BIOS sử dụng một số ngắt cho các dịch vụ phần cứng và LINUX sử dụng ngắt INT 0x80 để gọi dịch vụ của kernel.

Sau đây là một ví dụ về việc sử dụng ngắt INT 0x80 của Linux.

- Bộ xử lý lưu lại giá trị hiện thời của thanh ghi bộ đếm chương trình Program Counter (PC) và Code Segment (CS) vào ngăn xếp stack cùng với từ điều khiển trạng thái bộ xử lý Proccesor Status Word (PSW).

- Byte thứ 2 trong câu lệnh INT là một chỉ số trong bảng véctơ ngắt để từ đó tìm được địa chỉ của chương trình con dịch vụ ngắt (ISR). Bộ xử lý nạp địa chỉ này vào thanh ghi PC và CS và việc thực hiện chương trình con được thực hiện từ điểm này.

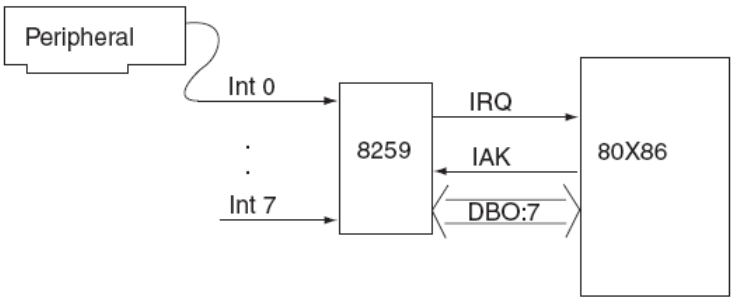


Hình 4 - Ngắt và hoạt động của ngắt

- Kết thúc của ISR là câu lệnh IRET (Interrupt Return). Nó giải phóng PC và CS để nạp lại giá trị của chương trình chính và thực hiện tiếp lệnh tiếp theo sau lệnh INT.

Lệnh INT cũng tương tự như lệnh gọi chương trình con CALL nhưng có đôi chút khác biệt: trong khi địa chỉ đích của lệnh CALL được nhúng vào trong câu lệnh đó thì với INT, ta không cần quan tâm đến địa chỉ của ISR. Địa chỉ của nó nằm trong bảng véctơ ngắt. Đây là một điểm thuận lợi cho việc truyền thông giữa chương trình được biên dịch và chương trình được tải, ví dụ như chương trình ứng dụng và hệ điều hành.

Các ngắt ngoài có cách thức thực hiện như thể hiện trong hình 5. Một thiết bị bên ngoài đưa ra một *"yêu cầu ngắt" Interrupt Request* (*IRQ*). Khi bộ xử lý phản ứng lại bằng một xác nhận "*chấp nhận ngắt*" *Interrupt Acknowledge* (*IAK*), thiết bị đó sẽ gửi số thứ tự của véctơ ngắt lên bus dữ liệu. Bộ xử lý sau đó sẽ thiết lập một lệnh ngắt INT với chỉ số véctơ ngắt đã được cung cấp.

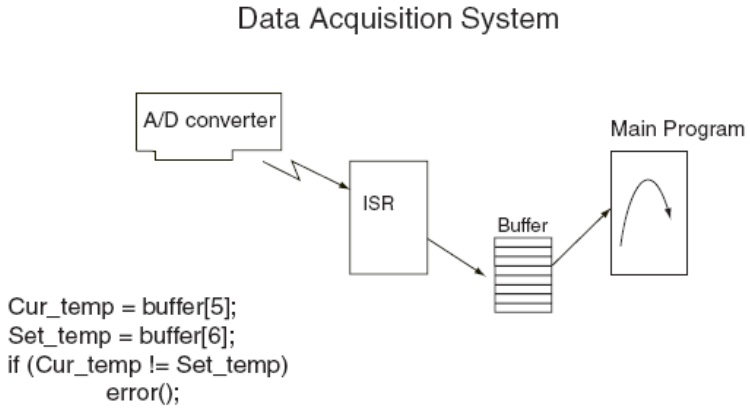


Hình 5 – Ngắt cứng

Trong thực tế, hầu hết các hệ thống đều sát nhập thêm một thiết bị ngoại vi đặc biệt được gọi là *bộ điều khiển ngắt Interrupt Controller* để quản lý các công việc chi tiết chẳng hạn như gửi chỉ số của véctơ ngắt lên bus dữ liệu vào đúng thời điểm cần thiết. Kiến trúc PC bao gồm 2 bộ điều khiển ngắt 8259, mỗi một bộ có thể quản lý được 8 ngắt. Mỗi một bộ 8259 có một cơ chế quản lý ngắt theo kiểu ưu tiên. Do đó, một thiết bị quan trọng sẽ được ưu tiên hơn thiết bị khác ít quan trọng hơn.

Ngắt có thể được kích hoạt hoặc bị vô hiệu hoá. Ở cấp độ của bộ xử lý, ngắt có thể được kích hoạt hoặc vô hiệu hoá thông qua câu lệnh STI và CLI. Các ngắt có thể được kích hoạt hoặc vô hiệu một cách có chọn lọc ở cả bộ điều khiển ngắt 8259 hay ở chính thiết bị đó. Trên thực tế, ***việc kích hoạt và vô hiệu ngắt chính là điểm mấu chốt để thiết kế và thực thi một phần mềm thời gian thực.***

Cũng không có gì đáng ngạc nhiên khi nói rằng ngắt không đồng bộ có những vấn đề đáng bàn của nó. Để ý một ứng dụng thu thập dữ liệu dựa trên bộ A/D đa kênh như trên hình 6. Cứ mỗi khi bộ chuyển đổi A/D thu thập một tập hợp dữ liệu trên các kênh, nó ngắt bộ xử lý. Chương trình con dịch vu ngắt đọc dữ liệu và cất vào bộ nhớ đệm, nơi mà chương trình khác (còn gọi là chương trình nền) sẽ tiếp tục xử lý.



Hình 6 – Ví dụ về ngắt

Hoạt động điều khiển ngắt cho phép chúng ta phản ứng lại A/D một cách nhanh chóng trong khi bộ nhớ đệm tách chương trình nền khỏi nguồn dữ liệu, ví dụ, chương trình nền không cần quan tâm đến dữ liệu được từ đâu mà có được.

Bây giờ hãy xem đến đoạn mã lệnh được ghi trong hình 6. Giả thiết chỉ là thí nghiệm, chúng ta cung cấp một tín hiệu biến đổi liên tục vào cả kênh 5 và 6. Đồng thời, giả thiết rằng chương trình sẽ không bị "fail" khi đang thực hiện đo tín hiệu đồng nhất.

Trong thực tế, chương trình như đã viết chắc chắn sẽ bị "fail" bởi vì một ngắt có thể xảy ra trong khi cập nhật biến Cur_temp và cập nhật biến Set_temp với kết quả là giá trị của biến Cur_temp được cập nhật từ tập hợp dữ liệu cũ trước đó, còn giá trị của biến Set_temp được cập nhật từ tập hợp dữ liệu hiện thời. Như vậy, khi tín hiệu đầu vào thay đổi theo thời gian và các tập hợp dữ liệu được tách rời nhau ở các thời gian xác định, giá trị các biến sẽ khác nhau và do đó, chương trình sẽ "fail".

Đây chính là bản chất của vấn đề lập trình thời gian thực. ***Cần phải quản lý các ngắt không đồng bộ để chúng không xảy ra vào những thời điểm không thích hợp.***

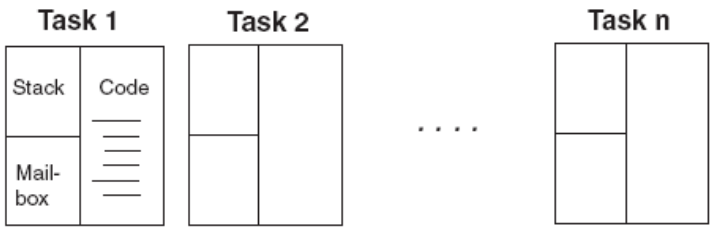
Có một giải pháp, dù không hay cho lắm, để giải quyết vấn đề này. Ta có thể dùng một lệnh vô hiệu hoá ngắt (CLI) trước khi cập nhật biến Cur_temp và kích hoạt ngắt bằng lệnh STI sau khi cập nhật biến Set_temp. Việc làm này giúp các ngắt tránh khỏi phiền phức của việc cập nhật liên tục như đã đề cập. Có vẻ như chúng ta đã sáng suốt khi sử dụng các lệnh CLI và STI như một chìa khoá cho một giải pháp đúng đắn, nhưng nếu chỉ đơn giản là rải các lệnh CLI và STI trong code của chương trình thì cũng chẳng khác gì việc sử dụng các lệnh "go to" và các biến toàn cục.

CHƯƠNG 3

CÁC TÁC VỤ – TASKS

Một giải pháp đưa ra có tên *Đa nhiệm*. Giải pháp này đã được chứng minh là một mô hình chuẩn cho các hệ thống điều khiển sự kiện và hệ thống sử dụng ngắt. Ý tưởng cơ bản của giải pháp này là chúng ta có thể phân chia một vấn đề lớn thành các nhánh nhỏ và đơn giản hơn để giải quyết. Mỗi một *vấn đề con – sub-problem* trở thành một *tác vụ - task*. Mỗi một tác vụ chỉ làm một việc đơn giản. Sau đó, chúng ta giả thiết rằng các tác vụ này chạy song song với nhau. Trên thực tế, các tác vụ không bao giờ chạy song song nếu chúng ta không có một hệ thống đa vi xử lý. Trong trường hợp đang xét, các tác vụ sẽ chia sẻ một bộ vi xử lý.

Cũng giống như các chương trình khác, một tác vụ bao gồm mã lệnh để thực hiện các chức năng mà tác vụ phải thực hiện (do người lập trình đã thiết kế). Mã lệnh được chứa trong một hàm tương tự như hàm main() trong ngôn ngữ lập trình C. Điều làm nên sự khác biệt của tác vụ chính là *ngữ cảnh – context* chứa trong *ngăn xếp – stack* của nó.



Hình 1 – Task là gì ?

Mỗi một tác vụ bao gồm :

- Mã nguồn chứa các chức năng của tác vụ

- Một *ngăn xếp – stack* để chứa ngữ cảnh của tác vụ

- Một *hộp thư – mail box* (tùy chọn) để phục vụ cho việc truyền thông với các tác vụ khác.

Chú ý rằng, đôi khi (nhiều khi khá hữu dụng) ta có thể tạo ra nhiều tác vụ từ một hàm chung. Như đã nói, điều làm cho một tác vụ có thể tách biệt và khác biệt với các tác vụ khác chính là ngăn xếp của nó. Đây thực tế chính là lập trình hướng đối tượng kiểu cổ điển. Ta có thể nghĩ rằng hàm tác vụ chính là việc định nghĩa một class. Và một tác vụ tạo ra từ hàm đó chính là một ví dụ về class.

Mặc dù có thể thấy các tác vụ là khá độc lập, nhưng về cơ bản thì chúng cũng cần phải hợp tác với nhau để thực hiện một mục đích chung đã được thiết kế sẵn cho hệ thống. Vì vậy, mỗi một tác vụ cần phải có một cơ chế truyền thông mà thông qua đó, chúng có thể kết nối, đồng bộ với các tác vụ khác. Trong trường hợp này, ta gọi cơ chế đó là *Hộp thư – mail box*.

Bảng 1 miêu tả cấu trúc mã nguồn của một tác vụ. Đối số data dùng để tham số hóa một tác vụ. Vai trò của nó cũng giống với các đối số argv và argc trong hàm main() với ngôn ngữ C. Đối số này thực sự quan trọng trong trường hợp nhiều tác vụ cùng được tạo ra từ một hàm. Sự *duy nhất* của tác vụ được thể hiện bởi giá trị của đối số này.

Bảng 1: Cấu trúc thông thường của một tác vụ

```
void task (void *data)
{
      init_task();

      while (TRUE)
      {
        Wait for message at task mailbox();
        switch (message.type)
        {
            case MESSAGE_TYPE_X:
              ...
              break;
            case MESSAGE_TYPE_Y:
              ...
              break;
     }
      }
}
```

Một tác vụ có thể được khởi động với một vài khởi tạo (có thể bao gồm khởi tạo đối số data). Sau đó, thông thường, tác vụ sẽ đi vào một vòng lặp không giới hạn. Tại một vài điểm trong vòng lặp, nó sẽ đợi "*một sự kiện nào đó xảy ra*", có thể, sự kiện đó là một bản tin được gửi tới mail box, hoặc chỉ đơn giản là tràn bộ định thời. Trong khi chờ sự kiện, tác vụ sẽ không làm gì cả và không sử dụng bộ vi xử lý. Một vài tác vụ khác nếu đã sẵn sàng hoạt động hoặc đang hoạt động sẽ sử dụng bộ vi xử lý.

Khi sự kiện mà tác vụ đang chờ xảy ra, tác vụ sẽ "*thức dậy*" và, ví dụ, nhận lấy bản tin, giải mã bản tin và hoạt động theo các yêu cầu đặt sẵn dựa trên một hệ thống các yêu cầu được phân định bởi câu lệnh switch(). Sau khi thực hiện xong yêu cầu, tác vụ lại quay trở lại trạng thái chờ sự kiện.

Có thể thấy rằng, tất cả các tác vụ đều giành phần lớn thời gian cho việc chờ một sự kiện nào đó xảy ra. Đây cũng chính là lý do để đa nhiệm có thể hoạt động.

Phần 4: Lập lịch trong hệ thời gian thực – Scheduling

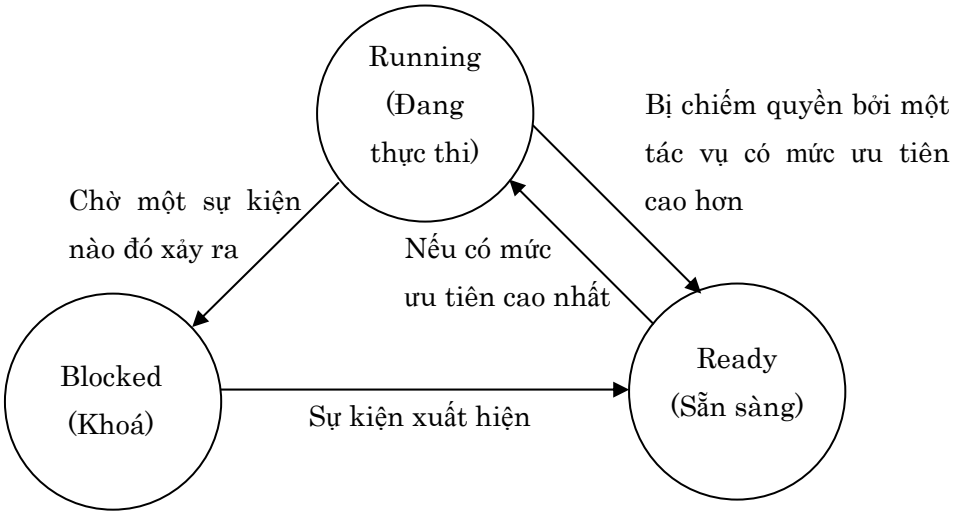
I. Scheduling:

Các tác vụ (task) hoạt động dưới sự giám sát của kernel thời gian thực. Chúng bao gồm:

- Một tập hợp các dịch vụ thực hiện các công việc như đồng bộ hoá và giao tiếp truyền thông giữa các tác vụ
- Một bộ lập lịch (scheduler) với chức năng khẳng định rằng chỉ có duy nhất tác vụ với mức ưu tiên cao nhất đang được thực thi.

Bộ lập lịch xét các tác vụ như những cái máy trạng thái (state machine). Tất cả các kernel đều có mô hình trạng thái của nó, tuy nhiên, thông thường thì các mô hình trạng thái nảy rất phức tạp. Hình 4.1 chỉ ra cho các bạn thấy một mô hình trạng thái mang tính khái niệm của tác vụ. Trong hình, ta thấy có các trạng thái:

- *Đang thực thi* (*Running*): chỉ có duy nhất một tác vụ là được nằm trong trạng thái này. Một tác vụ có thể tự động chuyển từ trạng thái Đang thực thi sang trạng thái Khoá (Blocked) bằng việc chờ đợi một sự kiện xảy ra. Trong một hệ thống có sự *chiếm quyền thực thi* (chúng ta sẽ đề cập đến nó sau), bộ lập lịch có thể bắt một tác vụ đang ở trạng thái Đang thực thi xuống trạng thái Sẵn sàng (Ready) nếu có một tác vụ với mức ưu tiên cao hơn chuyển đến trạng thái Sẵn sàng. Chúng ta gọi nó là *sự chiếm quyền thực thi* (*preemption*).
- *Sẵn sàng* (*Ready*): nếu một tác vụ đã sẵn sàng để hoạt động nhưng lại có mức ưu tiên thấp hơn tác vụ đang thực thi, tác vụ đó sẽ được chuyển đến trạng thái này và chờ. Tác vụ này sẽ được chuyển đến trạng thái Đang thực thi nếu nó trở thành tác vụ có mức ưu tiên cao nhất.
- *Khoá* (*Blocked*): một tác vụ bị khoá là tác vụ đang đợi một sự kiện nào đó xảy ra, ví dụ như một bản tin, tin nhắn được gửi đến hộp thư của tác vụ đó, hay thời gian chờ của tác vụ kết thúc….

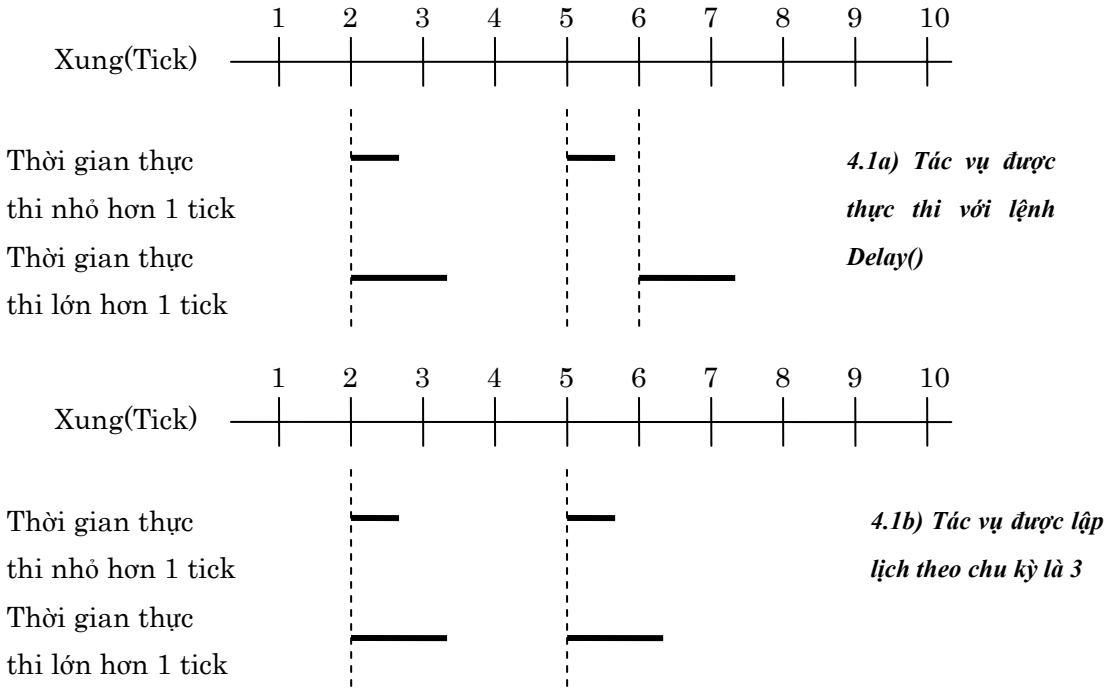


Hình 4.1 – Mô hình trạng thái của tác vụ

II. Lập lịch có chu kỳ:

Có rất nhiều tác vụ mà công việc của nó chỉ là thức dậy theo chu kỳ, làm một vài công việc nào đó và quay trở lại ngủ tiếp. Có một vài phương pháp để thực hiện các tác vụ kiểu này như trên hình 4.2. Trong tất cả các hệ điều hành, chúng ta đều có thể tìm thấy một lệnh gọi là hàm trễ Delay(), hoặc là một vài hàm có chức năng tương tự. Hàm này làm cho tác vụ bị khoá trong một thời gian xác định cho trước, thông thường thời gian này được biểu diễn bằng xung đồng hồ (clock tick). Hình 4.1a cho ta thấy việc thực hiện một tác vụ khi ta sử dụng lệnh Delay() đối với tác vụ có tính chu kỳ đó. Trong trường hợp này, khoảng thời gian trễ là 3 clock tick. Hoạt động của hệ thống phụ thuộc vào thời gian thực thi của tác vụ. Nếu thời gian thực hiện nhỏ hơn 1 tick thì tác vụ sẽ thức dậy sau mỗi 3 tick như mong muốn. Tuy nhiên, nếu tác vụ hoạt động quá 1 tick, khi đó, sau khi tác vụ gọi lệnh Delay(), nó vẫn sẽ bị khoá trong 3 clock tick. Thế nhưng, trong ví dụ này, tác vụ thực tế là sẽ thức dậy sau mỗi 4 xung clock tick. Đó không phải là điều chúng ta mong muốn.

Một phương án khác, không phải hệ thống nào cũng có, được trình bày ở hình 4.2b. Trong trường hợp này, bộ lập lịch sẽ đánh thức tác vụ vào đúng thời điểm thích hợp mà không quan tâm đến thời gian thực hiện tác vụ. Do đó, thay vì dùng hàm Delay(), một tác vụ có tính chu kỳ sẽ gọi hàm WaitTilNext(). Hàm này sẽ khóa tác vụ cho tới phiên thực hiện kế tiếp.



Hình 4.2 – Các tác vụ có tính chu kỳ

III. Lập lịch không theo chu kỳ:

Một số tác vụ phải phản ứng lại các sự kiện xảy ra ngẫu nhiên tại các thời điểm khác nhau. Một sự kiện có thể là việc một gói dữ liệu từ trên mạng được gửi đến nơi, việc một cái công tắc đóng lại để chỉ ra là bể nước đã đầy hoặc cũng có thể là sự kết thúc việc convert một tín hiệu tương tự sang số của bộ ADC và đang cần được đọc. Thông thường, các sự kiện không đồng bộ này được giao tiếp với máy tính thông qua các ngắt. Chương trình con dịch vụ ngắt phải có cách nào đó để kết nối sự xuất hiện của ngắt với tác vụ chịu trách nhiệm xử lý sự kiện.

IV. Lập lịch theo kiểu chiếm quyền thực thi và lập lịch không có chiếm quyền thực thi.

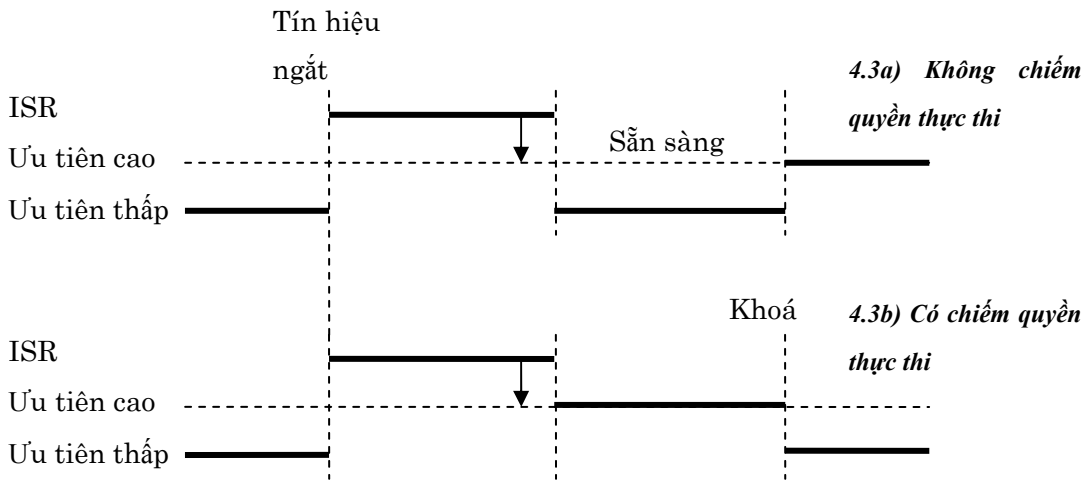
Có 2 phương thức cơ bản cho việc lập lịch một tác vụ: *chiếm quyền thực thi* và *không chiếm quyền thực thi*. Xét 2 tác vụ: tác vụ 1 có mức ưu tiên thấp hơn đang thực hiện và tác vụ 2 có mức ưu tiên cao hơn đang bị khoá để chờ một sự kiện xảy ra, sự kiện này được thông báo bởi một tín hiệu ngắt.

Hình 4.3a cho thấy những gì xảy ra trong hệ thống không có tính chiếm quyền ưu tiên. Chương trình con dịch vụ ngắt ISR làm cho tác vụ 2 với mức ưu tiên cao hơn chuyển từ trạng thái Khoá sang trạng thái Sẵn sàng. Tuy nhiên, đến khi ISR được thực hiện xong thì tác vụ 1 với mức ưu tiên thấp hơn vẫn sẽ được tiếp tục thực thi tại điểm nó bị ngắt. Sau đó, khi tác vụ 1 bị khoá để chờ sự kiện thì tác vụ 2 mới được chuyển sang trạng thái thực thi.

Hình 4.3b ứng với trường hợp của hệ thống có tính chiếm quyền ưu tiên. Điểm khác biệt ở đây là bộ lập lịch được gọi đến ở cuối chương trình con dịch vụ ngắt. Bộ lập lịch xác định tác vụ có mức ưu tiên cao đang ở trạng thái Sẵn sàng và chuyển nó lên trạng thái thực thi. Do đó, tác vụ với mức ưu tiên thấp đã bị chiếm quyền thực thi.

Một hệ thống không có tính chiếm quyền thực thi muốn rằng tất cả các tác vụ phải là "những công dân tốt" bằng cách tự nguyện trao trả bộ xử lý cho các tác vụ khác để chắc chắn một điều: các tác vụ đều có cơ hội để sử dụng bộ xử lý. Các thế hệ Windows trước kia đều là dạng này. Linux thì khác, nó là một hệ điều hành có tính chiếm quyền thực thi mặc dù các bản Linux chuẩn không quan tâm đến vấn đề thời gian thực bởi trong một thời gian dài, vấn đề chiếm quyền thực thi không được đề cập.

Các hệ thống có tính chiếm quyền thực thi cung cấp cho ta nhiều hơn thời gian phản ứng có thể dự đoán được bởi vì tác vụ có mức ưu tiên cao sẽ được xử lý ngay lập tức. Đây chính là điểm cốt lõi của thời gian thực: khả năng đảm bảo một thời gian lớn nhất để phản ứng lại một sự kiện. Trong hệ thống không chiếm quyền thực thi, chẳng có gì để đảm bảo thời gian một tác vụ nhường lại bộ xử lý cho các tác vụ khác. Mặt khác, trong một hệ thống có chiếm quyền ưu tiên, vấn đề tranh chấp tài nguyên hệ thống cũng đáng được quan tâm cẩn thận.



Hình 4.3 - Lập lịch: có và không có chiếm quyền thực thi

Hai phương án khác được tận dụng để xử lý các tác vụ có cùng mức ưu tiên. Trong phương thức lập lịch kiểu *vòng lặp robin*, một tác vụ sẽ được thực thi đến khi nào nó bị khoá (block) để chờ một sự kiện xuất hiện hoặc cũng có khi nó tình nguyện nhường (yield) bộ xử lý lại. Sự khác biệt giữa khoá và nhường là ở chỗ: trong trường hợp thứ 2, tác vụ sẽ quay trở lại trạng thái Sẵn sàng (ready).

Xét trường hợp trong danh sách Sẵn sàng có 3 tác vụ thứ tự lần lượt là A, B, C. Các tác vụ này có cùng mức ưu tiên như nhau. Tác vụ A đứng đầu danh sách và được chuyển đến trạng thái Thực thi. Khi tác vụ A nhường (yield) bộ xử lý, tác vụ B trở thành trạng thái thực thi và danh sách Sẵn sàng sẽ như sau:

B C A

Khi B nhường, C sẽ chuyển trạng thái và danh sách sẽ chuyển thành:

C A B

Như vậy, tất cả các tác vụ sẽ hoạt động thành một vòng tròn, chúng hoạt động theo kiểu nhường nhau. Các tác vụ có mức ưu tiên thấp hơn trong trạng thái Sẵn sàng sẽ không bao giờ được thực hiện cho đến khi tất cả các tác vụ trên bị khoá.

Nhát cắt thời gian là một biến thể của vòng lặp robin. Trong đó, nó quy định mỗi tác vụ sẽ nhận được một lượng thời gian nhất định hay còn gọi là nhát cắt thời gian. Việc làm này bảo vệ các tác vụ khỏi trường hợp chiếm dụng bộ xử lý quá lâu. Do đó, một tác vụ sẽ chạy cho đến khi nó bị khoá, nó tình nguyện nhường hay quá hạn về thời gian cho phép. Tuỳ thuộc vào hoàn cảnh và yêu cầu, các tác vụ sẽ có lượng thời gian cho phép bằng nhau hoặc khác nhau.

Xét trên khía cạnh nào đó, vòng lặp robin chỉ là một dạng khác của vòng lặp polling.

CHƯƠNG 5
CÁC DỊCH VỤ NHÂN – KERNEL SERVICES

Một nhân[1] (kernel – từ nay về sau xin được giữ nguyên bản bằng tiếng Anh) đa tác vụ được định nghĩa rất rộng bằng các dịch vụ mà nó cung cấp. Tập hợp các dịch vụ này cấu thành nên Giao diện Lập trình Ứng dụng (Application Programming Interface – API) cho phép người sử dụng có thể tận dụng các đặc tính của kernel. Khi mô tả các chức năng của một kernel đa tác vụ, người ta thường trình bày về API để chỉ ra các khái niệm được sắp xếp ra sao trong mã nguồn thực. Trong các phần còn lại của tài liệu, chúng ta chỉ xem xét API dưới dạng đơn giản và chỉ trên cơ sở ý tưởng để hiểu về các chức năng cơ bản. Việc thực hiện trên thực tế sẽ khác hơn về tiểu tiết so với mô hình được trình bày ở đây. Tuy nhiên, chúng vẫn thực hiện cùng một chức năng như nhau.

1. API cho tác vụ

Hãy xét API cho kernel bằng các dịch vụ cần thiết cho việc quản lý các tác vụ:

```
task_t *TaskCreate (void (*task)(void *data), void *data, int prior);
status_t TaskStart (task_t *task);
status_t TaskSuspend (task_t *task);
status_t TaskResume (task_t *task);
status_t TaskDelete (task_t *task);
```

Sẽ tuyệt đối không có điều gì xảy ra cho một hệ thống đa tác vụ cho tới khi chúng ta tạo ra một hoặc một số tác vụ. Để tạo ra một tác vụ, chúng ta sử dụng một hàm có tên hoặc có tên tương tự như `TaskCreate`. Ít nhất, chúng ta cũng cần phải cung cấp cho hàm này một số thông số như sau:

- Một con trỏ, trỏ tới hàm thực thi các mã lệnh của tác vụ (task)

- Một con trỏ, trỏ tới dữ liệu là đối số của hàm khi nó được gọi đến lần đầu tiên (data).

- Độ ưu tiên của tác vụ (prior).

Dịch vụ tạo tác vụ có thể trả về một con trỏ, trỏ tới *Khối điều khiển tác vụ – Task Control Block (TCB)* mà được định nghĩa ở đây là `task_t`. Đây là một cấu trúc dữ liệu chứa mọi thứ mà kernel cần để biết về một tác vụ. Con trỏ này sau đó có thể được sử dụng để làm đối số tới các dịch vụ quản lý tác vụ khác.

Chú ý rằng, trong quá trình thực thi lệnh ở trên, chúng ta giả sử rằng dịch vụ tạo tác vụ sẽ tự động định địa chỉ cho TCB và ngăn xếp stack. Đồng thời chúng ta cũng giả thiết rằng kích thước của ngăn xếp là cố định và được thiết lập ở đâu đó. Trong một số trường hợp, người sử dụng phải tự mình định địa chỉ cho TCB hoặc/và stack và kích thước của stack. Ngoài ra, cũng có nhiều đối số phụ có thể đi kèm ví dụ như tên của tác vụ bằng mã ASCII, giá trị thời gian lấy mẫu….

[1] Nhân hay Kernel trong một hệ điều hành là phần cốt lõi của chương trình, cư trú trong bộ nhớ và thực hiện hầu hết các nhiệm vụ điều hành chính. Ví dụ như quản lý các hoạt động vào ra, quản lý bộ nhớ…

Dịch vụ tạo tác vụ có thể hoặc không thể khởi động tác vụ. Nếu là không thể, một dịch vụ khởi động tác vụ riêng biệt được cung cấp. Một khi tác vụ được thực thi, nó có thể bị tạm dừng (treo – suspend). Khi bị treo, nó sẽ bị tách khỏi quá trình lập lịch hoạt động cho tới khi nó được hồi phục (resume) theo chu kỳ. Cuối cùng, nếu một tác vụ không còn cần thiết nữa, nó có thể bị xóa (delete) khỏi danh sách các tác vụ đang hoạt động. Nói chung, các hàm quản lý tác vụ ngoại trừ `TaskCreate` đều trả về kiểu `status_t` để chỉ ra rằng hàm đó có thực thi thành công hay không.

2. API của định thời

```
void Delay (unsigned int ticks);
void DelayUntil (time_t *time);
void WaitTilNext (void);
```

Mọi kernel đều có một hàm có tên gọi tương tự như `Delay` để khóa tác vụ được gọi trong một khoảng thời gian cố định tính theo số lần dao động của mạch thời gian. Một số hệ thống cũng có những biến thể của lệnh này như `DelayUntil` để khóa tác vụ được gọi cho tới một thời điểm cố định trong ngày. Kiểu dữ liệu time_t là kiểu unsigned long int.

`WaitTilNext` chỉ có trong hệ thống có hỗ trợ tác vụ theo chu kỳ. Hàm này khóa tác vụ được gọi cho tới lần gọi lần sau trong lịch trình lập lịch.

CHƯƠNG 6
TRUYỀN THÔNG GIỮA CÁC TÁC VỤ – Inter-task Communication

Mặc dù các tác vụ được xem như độc lập với nhau nhưng nhiệm vụ tổng quát của hệ thống yêu cầu các tác vụ phải có sự liên hệ với nhau, hợp tác với nhau. Do đó, thành phần cốt yêu của bất cứ hệ điều hành thời gian thực là tập hợp các dịch vụ truyền thông và đồng bộ.

Có một vài cơ chế đồng bộ và truyền thông hay được sử dụng, bao gồm:

- Semaphore[1]: Sử dụng cho việc đồng bộ hóa tín hiêu và khả năng tận dụng tài nguyên

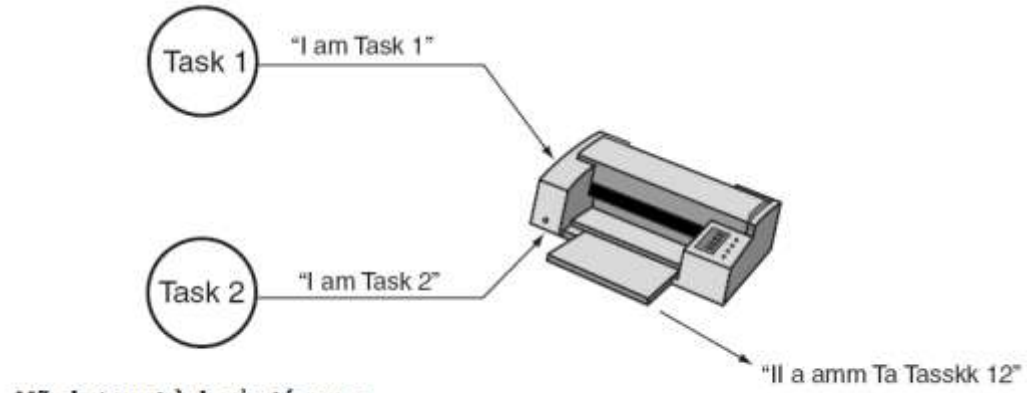
- Cờ sự kiện (Event Flag): Chỉ ra môt hoặc vài sự kiện đã xảy ra. Đây thực chất là mở rộng của Semaphore trong đó cho phép động bộ hóa trên các sự kiện hỗn hợp.

- Mailbox, hàng đợi và đường ống: Đây là cơ chế truyền thông dữ liệu giữa các tác vụ

Trên thực tế còn rất nhiều cơ chế khác ít được sử dụng rộng rãi như ADA và Monitor trong Java.

1. Semaphore

Hãy xét 2 tác vụ, mỗi một tác vụ có nhiệm vụ in một bản tin có nội dung "I am task n" (n là một số thứ tự của tác vụ) bằng một máy in chia sẻ như trong hình dưới. Nếu chúng ta không sử dụng bất cứ một cơ chế đồng bộ nào, kết quả có được từ máy in sẽ có thể là "II a amm tatasskk 12".

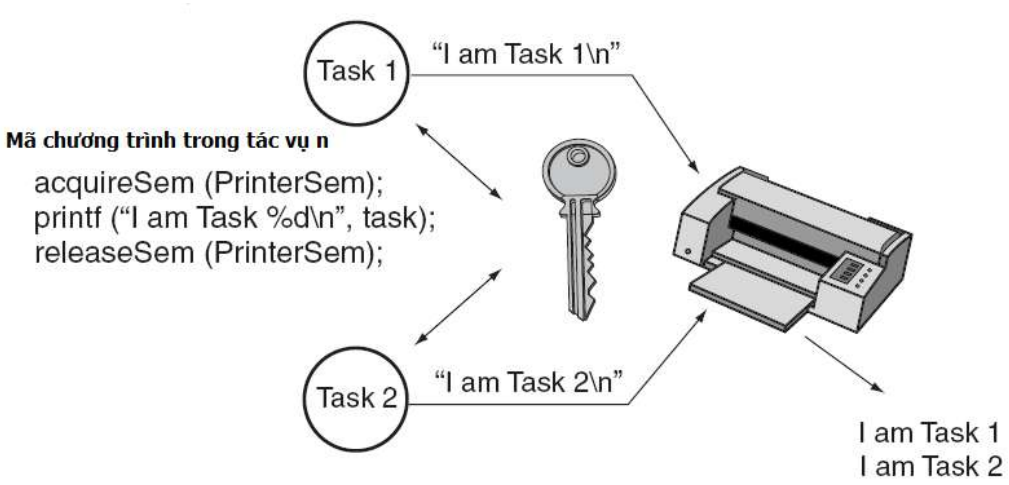


Hình 1. Chia sẻ tài nguyên

Điều cần thiết ở đây là phải có một cơ chế nào đó để với cơ chế này, máy in chỉ có thể được sử dụng bởi 1 tác vụ tại một thời điểm xác định.

[1] Semaphore: có nhiều cách định nghĩa và sử dụng, ví dụ như là cách sử dụng tín hiệu, thường là 2 lá cờ đặt ở vị trí thích hợp để thể hiện các tín hiêu chữ cái. Với lập trình, nó là cách để các chương trình đăng ký sử dụng tài nguyên dùng chung.

Semaphore hoạt động giống như một chiếc chìa khóa cho việc truy cập tới tài nguyên. Chỉ có tác vụ có chìa khóa này mới có quyền sử dụng tài nguyên. Để có thể sử dụng tài nguyên (là chiếc máy in trong trường hợp này), tác vụ cần *yêu cầu* (*acquire*) chìa khóa(semaphore) bằng cách gọi tới một dịch vụ thích hợp như trong hình 2. Nếu chìa khóa ở trạng thái sẵn sàng, tức là tài nguyên (máy in) hiện tại không được sử dụng bởi bất kỳ một tác vụ nào, tác vụ đó có thể được cho phép sử dụng tài nguyên. Sau khi sử dụng xong, tác vụ đó phải *trả lại* (*release*) semaphore cho các tác vụ khác có thể sử dụng.



Hình 2. Chia sẻ tài nguyên với Semaphore

Tuy nhiên, nếu máy in đang được sử dụng, tác vụ đó sẽ bị khóa cho tới khi tác vụ đang sử dụng máy in trả lại semaphore. Cùng một lúc có thể có nhiều tác vụ yêu cầu semaphore trong khi máy in đang hoạt động. Tất cả các tác vụ đó đều sẽ bị khóa. Các tác vụ bị khóa sẽ được xếp hàng theo kiểu hàng đợi theo thứ tự về mặt ưu tiên hoặc theo thứ tự thời gian mà chúng yêu cầu semaphore theo lệnh acquireSem. Cách thức sắp xếp thứ tự hàng đợi cho các tác vụ có thể được xây dựng trong kernel hoặc cũng có thể được cấu hình khi mà semaphore được tạo ra.

Lệnh acquireSem hoạt động như sau:

- Giảm giá trị của semaphore.

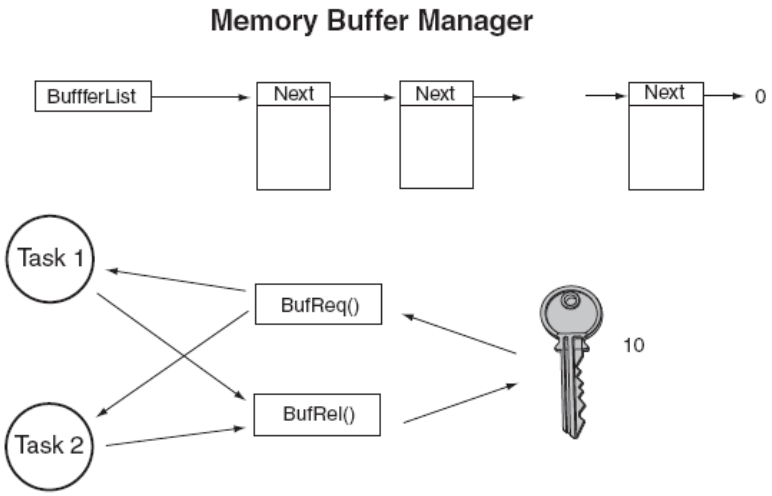
- Nếu kết quả giá trị lớn hơn hoặc bằng 0, điều này nói lên rằng tài nguyên là sẵn sàng, tác vụ có thể sử dụng tài nguyên ngay lập tức. Ngược lại nếu kết quả nhỏ hơn 0, tác vụ sẽ bị khóa và chờ đến khi tác vụ đang sử dụng tài nguyên thực hiện lệnh releaseSem.

Lệnh releaseSem tăng giá trị của semaphore. Nếu kết quả trả về bé hơn hoặc bằng 0, điều đó có nghĩa là có ít nhất một tác vụ đang đợi semaphore, do đó, tác vụ đó sẽ được chuyển vào trạng thái sẵn sàng.

Trong trường hợp máy in này, semaphore sẽ được gán mặc định ban đầu là 1 trong trường hợp hệ thống chỉ có 1 máy in được quản lý. Trường hợp này thông thường được gọi là semaphore *nhị phân* (*binary semaphore*) để phân biệt với các trường hợp tổng quát hơn

(*counting semaphore*), trong đó, semaphore được mặc định là một số bất kỳ nguyên và không âm.

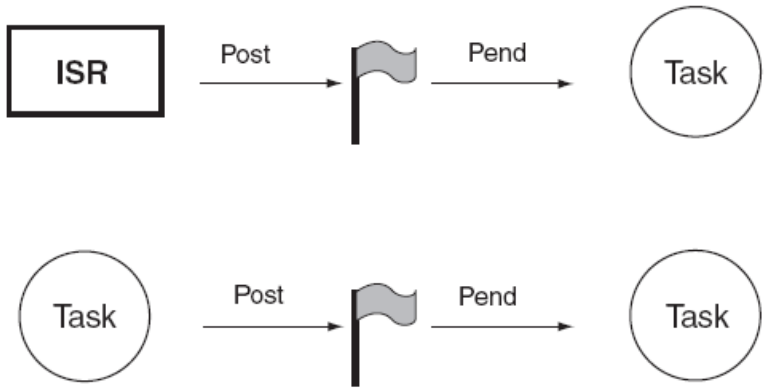
Xét một bộ định địa chỉ bộ nhớ động để quản lý số lượng bộ nhớ đệm cố định như trên hình 3. Ở đây, chúng ta khởi tạo cho semaphore một số lượng bộ nhớ đệm đang còn trống tại thời điểm ban đầu. Khi câu lệnh bufReq được gọi đến, nó trước tiên dành lấy semaphore, sau đó định địa chỉ cho bộ nhớ đệm. Trong 10 lần gọi lệnh bufReq đầu tiên, semaphore vẫn còn không âm, điều này làm cho các tác vụ yêu cầu semaphore vẫn có thể được hoạt động. Đến lần thực thi lệnh bufReq thứ 11, tác vụ yêu cầu sẽ bị khóa và chờ đến có một tác vụ khác gọi lệnh bufRel để giải phóng semaphore.



Hình 3. Chia sẻ hệ thống đa tài nguyên

Một số kenel sử dụng cả hai loại binary semaphore và couting semaphore vì trong một số trường hợp, binary semaphore có hiệu quả hơn. Binary semaphore đôi khi còn được gọi là *mutex* có nghĩa là *loại trừ lẫn nhau* (*mutual exclusion*).

Một semaphore đôi khi cũng có thể được sử dụng để tạo tín hiệu cho sự xuất hiện của một sự kiện như trong hình 4. Lấy ví dụ, làm thế nào mà hệ thống có thể nhận biết được sự xuất hiện của một ngắt? Tác vụ cần thông tin về sự xuất hiện của ngắt sẽ *treo* (*pend*) semaphore lên. Chương trình con dịch vụ ngắt (Internet Service Routine) sẽ phục vụ ngắt và sau đó *gửi* (*post*) semaphore lại. (Chú ý rằng thuật ngữ "pend" và "post" được sử dụng thường xuyên hơn các thuật ngữ "acquire" và "release").



Hình 4. Tạo tín hiệu cho sự kiện thông qua semaphore

Ở ví dụ trước, semaphore được khởi tạo bởi một giá trị khác 0 bởi vì tài nguyên là sẵn sàng để sử dụng. Ở đây, semaphore được khởi tạo là 0. Vì vậy, khi tác vụ đầu tiên treo (pend) semaphore, nó ngay lập tức bị khóa lại – sự kiện chưa được xảy ra. Khi một ISR gửi (post) lại semaphore, tác vụ đó được đánh thức và tiếp tục thực hiện.

Khi semaphore được sử dụng như một khóa tài nguyên, nhiều tác vụ có thể post hoặc pend nó. Tuy nhiên, trong trường hợp tạo tín hiệu hoặc đồng bộ hóa, semaphore thường được sử dụng bởi chỉ 1 ISR và 1 tác vụ.

Cơ chế tương tự như trên cũng có thể được sử dụng khi 1 tác vụ muốn tạo tin hiệu của một sự kiện tới một tác vụ khác.

2. Semaphore API

```
semaphore_t *SemCreate (unsigned int value);
status_t SemDelete (semaphore_t *sem);
status_t SemPost (semaphore_t *semaphore);
status_t SemPend (semaphore_t *semaphore, unsigned int timeout);
```

Các đối tượng semaphore cần phải được tạo ra từ dịch vụ có tên thường là SemCreate. Hàm này trả về một con trỏ, trỏ tới cấu trúc dữ liệu semaphore_t đại diện cho semaphore. Khi tạo ra một semaphore, chúng ta có thể gán cho nó một giá trị ban đầu nhằm phản ánh vai trò mà semaphore cần thực hiện. Nếu semaphore được dùng để quản lý tài nguyên thì giá trị ban đầu này chính là số lượng tài nguyên mà semaphore quản lý. Nếu semaphore được dùng để tạo tín hiệu cho sự kiện thì thông thường nó thường có giá trị ban đầu là Zero bởi vì sự kiện lúc đó chưa xảy ra. Semaphore có thể bị xóa nếu nó không còn tác dụng.

Các hàm cơ bản của semaphore là SemPost và SemPend. Khi post (gửi lên) một semaphore, đối số thường cần đến là chỉ số của chính semaphore. Chỉ số này chính là giá trị trả về khi thực hiện hàm SemCreate. Yêu cầu một semaphore (pend) thường cần 2 đối số. Thứ nhất là chỉ số của semaphore và thứ hai là thời gian quá hạn (timeout) tính theo xung đồng hồ hệ thống. Nếu đối số thời gian là một số khác 0 thì tham số này chính là thời gian lớn nhất mà

một tác vụ có thể đợi cho đến khi một semaphore được post lên. Nếu quá thời gian này mà vẫn chưa có semaphore nào thì hàm SemPend sẽ trả về một mã lỗi.

3. Các hộp thư – mailboxes

Ngoài việc tạo tín hiệu cho các sự kiện, các tác vụ còn cần phải chia sẻ dữ liệu cho nhau. Điều này được thực hiện thông qua cơ chế của mailbox. Một tác vụ (sender – bên gửi) gửi (post) một bản tin (message) tới hộp thư, trong khi một tác vụ khác (receiver – bên nhận) đang gửi yêu cầu (pend) tới hộp thư để đợi bản tin. Nếu không có bản tin nào được gửi tới hộp thư khi bên nhận gửi yêu cầu (pend) tới hộp thư thì tác vụ nhận sẽ bị khóa cho tới khi một bản tin được gửi (post) tới.

Trong nhiều trường hợp, bản tin là một con trỏ. Những gì mà nó trỏ tới phải được sự đồng ý qua lại giữa 2 tác vụ thành viên trong quá trình nhận/gửi. Ví dụ như người nhận và người gửi phải đồng ý với nhau về ngôn ngữ của một bức thư. Một số kernel sử dụng một lượng tối thiểu cấu trúc hệ thống cho các bản tin.

Tất nhiên là những gì mà con trỏ trỏ tới phải có thể truy cập được từ cả 2 phía nhận và gửi. Trong các hệ thống đặt ở chế độ bảo vệ như Linux, điều này (do thiết kế sẵn có) là không thể thực hiện được. Trong trường hợp này, bản tin sẽ được copy từ không gian của bên gửi và gửi vào hộp thư, và từ hộp thư nó lại được copy tới không gian của bên nhận.

Tùy thuộc vào các thực hiện, một hộp thư có thể chứa chỉ 1 bản tin. Nhưng nó cũng có thể chứa nhiều bản tin được xếp trong hàng đợi theo thứ tự giống với thứ tự mà chúng được gửi đi. Một kernel cho phép tùy chọn gửi đi một bản tin với "độ ưu tiên cao". Những bản tin kiểu này sẽ luôn được xếp lên đầu trong các hàng đợi của hộp thư. Đối với semaphore, số lượng các tác vụ có thể đợi trong hộp thư sẽ được sắp xếp trong hang đợi theo thứ tự của FIFO[2] hoặc theo thứ tự ưu tiên giữa chúng.

Xét trường hợp, điều gì xảy ra khi một bản tin được gửi đến 1 hộp thư chỉ có khả năng lưu giữ 1 bản tin, nhưng hộp thư này lại đang có một bản tin sẵn có trong nó? Có 2 khả năng thực hiện. Hoặc là hộp thư gửi dịch vụ để trả về một mã lỗi hoặc là tác vụ sẽ bị khóa cho tới khi nó được phép gửi (post) bản tin đi. Cũng có thể có khả năng là khi tác vụ gửi bản tin đi thì lại không có tác vụ nào đợi để nhận bản tin đó. Trong trường hợp này, bên gửi sẽ bị khóa cho tới khi một tác vụ khác nhận bản tin đó.

4. API của hộp thư – Mailbox API

```
mailbox_t *MbxCreate (void *message);
status_t MbxDelete (mailbox_t *mbx);
status_t MbxPost (mailbox_t *mbx, void *msg);
void *MbxPend (mailbox_t *mbx, unsigned int timeout, status_t *status);
```

[2] FIFO: First In First Out. Ta sẽ gặp rất nhiều khái niệm này khi làm việc với các hệ điều hành thời gian thực. Đối với các hệ điều hành này, FIFO dùng để liên kết giữa kernel chính (không có tính thời gian thực) của hệ điều hành với kernel thời gian thực (thường được chèn giữa kernel không thời gian thực và các thiết bị ngoại vi). Ta không thể trực tiếp truyền nhận thông tin giữa 2 kernel loại này vì bản chất của chúng khác nhau. Vì vậy cần phải có một cơ chế như FIFO trong các hệ điều hành này.

```
status_t MbxBroadcast (mailbox_t *mbx, void *msg);
```

API của hộp thư rất giống với API của semaphore. Trước khi làm bất cứ việc gì, chúng ta cần tạo ra một hộp thư. Dịch vụ tạo hộp thư, MbxCreate, sẽ trả về một con trỏ trỏ tới cấu trúc kiểu mailbox_t. MbxCreate cũng có thể cho phép tùy chọn gửi đi một bản tin khởi tạo tới mailbox khi nó được tạo ra.

Cũng giống như semaphore, các hoạt động cơ bản của hộp thư là pend và post. Đối số tới MbxPost là chỉ số của hộp thư và con trỏ trỏ tới bản tin. So với việc trả về một biến trạng thai như các hàm khác, MbxPend trả về một con trỏ trỏ tới bản tin nhận được và biến trạng thái được sử dụng như một đối số của hàm.

Một số hệ thống có một hàm tương tự như MbxBroadcast dùng để gửi bản tin đến tất cả các tác vụ đang yêu cầu bản tin trong hộp thư.

5. Hàng đợi và đường ống (Queues và Pipe)

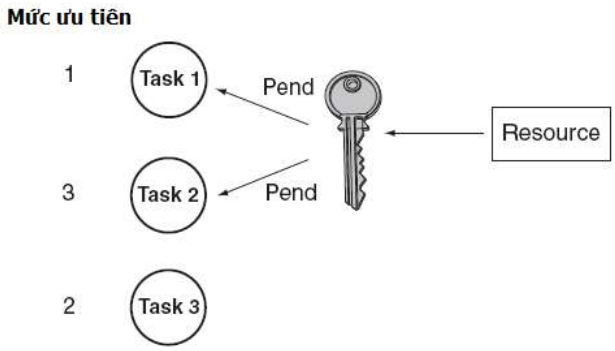
```
queue_t *QCreate (int qsize);
status_t QDelete (queue_t *queue);
status_t QPost (queue_t *queue, void *message);
status_t QPostFront (queue_t *queue, void *message);
void *QPend (queue_t *queue, unsigned int timeout, status_t *status);
status_t QFlush (queue_t *queue);

pipe_t PipeCreate (void);
status_t PipeDelete (pipe_t *pipe);
status_t PipeWrite (pipe_t *pipe, void *buffer, size_t len);
status_t PipeRead (pipe_t *pipe, void *buffer, size_t len);
```

Trong hầu hết các trường hợp, hàng đợi thường đơn giản là hộp thư có khả năng chứa nhiều thư. Khi tạo ra hàng đợi, chúng ta cần phải chỉ ra độ lớn (hay độ dài hàng đợi), là thông số chỉ ra số lượng bản tin chứa trong nó. Ngoài các lệnh gọi Post và Pend, hàng đợi luôn có dịch vụ để gửi bản tin có mức ưu tiên cao lên đứng đầu hàng, QPostFront(), hoặc để xóa toàn bộ các bản tin hiện thời đang nằm trong hộp thư, QFlush().

Dạng đường ống thì có khác một chút, trong khi hộp thư và hàng đợi di chuyển dữ liệu thành những đoạn rời rạc mà ta gọi là bản tin thì đường ống nói một cách chung chung là một chuỗi byte liên tục nối giữa 2 tác vụ. Một tác vụ đọc đường ống còn tác vụ kia ghi dữ liệu lên đường ống. Trên thực tế, các hệ thống Unix, gồm cả Linux, làm việc với các đường ống như các file thông thường cùng các hàm như read() và write() chuẩn. pipe_t chỉ là một mảng gồm 2 số nguyên dài trong đó nguyên tố Zero biểu thị việc kết thúc đọc của đường ống còn nguyên tố 1 biểu thị kết thúc ghi. Đường ống được tạo ra bằng lệnh pipe().

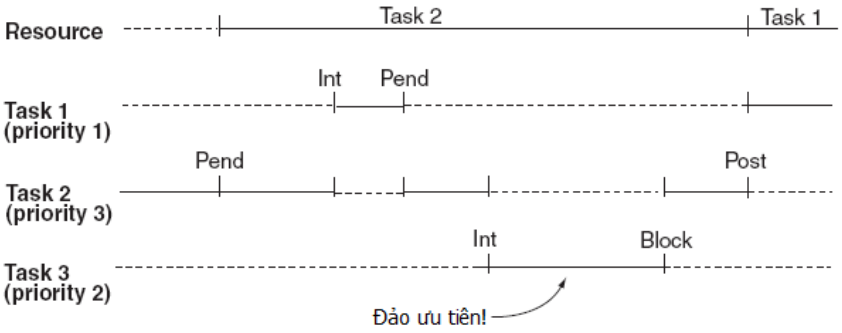
CHƯƠNG 7
CÁC VẤN ĐỀ NẢY SINH KHI GIẢI QUYẾT VẤN ĐỀ CHIA SẺ TÀI NGUYÊN – ĐẢO ƯU TIÊN

Trong chương trước, chúng ta đã sử dụng semaphore như một công cụ để giải quyết vấn đề tranh chấp tài nguyên. Tuy nhiên, phương pháp này có thể dẫn đến vấn đề khó thực thi khi hệ thống làm việc. Xét tình huống được trình bày trong hình 1. Tác vụ 1 và 2 yêu cầu sử dụng tài nguyên dùng chung được bảo vệ bởi semaphore. Tác vụ 1 có quyền ưu tiên cao nhất và tác vụ 2 có mức ưu tiên thấp nhất. Tác vụ 3 không cần sử dụng tài nguyên có mức ưu tiên trung bình.



Hình 1. Trường hợp đảo ưu tiên

Hình 2 trình bày về quá trình thực thi của hệ thống theo thời gian. Giả sử rằng tác vụ 2 đang làm việc và gửi yêu cầu sử dụng tài nguyên. Tài nguyên khi đó đang rỗi nên tác vụ 2 ngay lập tức sử dụng. Sau đó, một ngắt xuất hiện làm tác vụ 1 trở nên sẵn sàng hoạt động. Do tác vụ 1 có mức ưu tiên cao nhất nên nó chiếm quyền thực thi của tác vụ 2 và hoạt động cho tới khi nó gửi yêu cầu sử dụng tài nguyên.



Hình 2. Thời gian thực hiện – Đảo ưu tiên

Bởi vì tài nguyên đang được tác vụ 2 sử dụng nên tác vụ 1 sẽ bị khóa và tác vụ 2 lại nắm lại quyền điều khiển. Cho tới đây, mọi việc đang diễn ra theo đúng ý của chúng ta. Mặc dù tác vụ 1 có quyền ưu tiên cao hơn nhưng nó vẫn phải chờ cho đến khi tác vụ 2 kết thúc việc sử dụng tài nguyên.

Vấn đề nảy sinh trong trường hợp tác vụ 3 trở nên sẵn sàng hoạt động trong khi tác vụ 2 đang sử dụng tài nguyên. Do có quyền ưu tiên cao hơn tác vụ 2 nên tác vụ 3 sẽ chiếm quyền hoạt động của tác vụ 2. Trường hợp này gọi là "*Đảo ưu tiên*" (*Priority Inversion*) bởi vì tác vụ có quyền ưu tiên thấp hơn (tác vụ 3) lại có thể ngăn cản hoạt động của tác vụ có mức ưu tiên cao hơn (tác vụ 1).

Giải pháp thông thường để xử lý vấn đề này là tạm thời tăng mức ưu tiên của tác vụ 2 lên bằng với mức ưu tiên của tác vụ 1 ngay khi tác vụ 1 gửi yêu cầu sử dụng tài nguyên tới semaphore. Từ lúc này, tác vụ 2 sẽ không thể bị ngắt bởi các tác vụ có mức ưu tiên thấp hơn mức ưu tiên của tác vụ 1. Giải pháp này được gọi là "*Sự kế thừa mức ưu tiên*" – *priority inheritance*. Nếu kernel tạo ra sự khác biệt giữa một semaphore và một mutex[1] thì chức năng priority inheritance sẽ được tích hợp thành một thông số tùy chọn khi mutex được tạo ra.

Một giải pháp khác, được gọi là mức "*ưu tiên trần*" – *priority ceiling*, tăng mức ưu tiên của tác vụ 2 lên cao hơn tất cả các tác vụ có thể yêu cầu sử dụng mutex ngay khi tác vụ 2 nhận được mutex. Giải pháp này được coi là hữu hiệu hơn bởi vì nó loại trử được các ngắt không cần thiết. Không một tác vụ nào cần tài nguyên có thể cướp quyền điều khiển của tác vụ đang nắm giữ tài nguyên.

[1] Mutex: Mutual Exclusion – Sự loại trừ lẫn nhau. Đây là Binary semaphore Xem các khái niệm về mutex trong chương trước.