SE05 - Tìm hiểu về tiến trình

I. File thực thi trên Linux

- File thực thi trên Linux là các file nhị phân chứa các chương trình được viết bằng ngôn ngữ lập trình và được biên dịch để có thể chạy được trong môi trường Linux. Các file này chứa các lệnh dưới dạng mã máy được CPU hiểu và thực thi ngày khi chạy chương trình. Hệ thống Linux nhận dạng các file thực thi này bằng bit chỉ quyền thực thi của file (x).
- Các file nhị phân được đề cập ở trên tuân theo một cấu trúc cụ thể và một trong những cấu trúc phổ biến nhất là ELF (Executable and Linkable Format), được sử dụng rộng rãi cho các file thực thi, file đối tượng, thư viện liên kết động.
- Lý do ELF được sử dụng rộng rãi là do tính linh hoạt, khả năng mở rộng và hỗ trợ đa nền tảng của cấu trúc này.
- Cấu trúc một file ELF:
  - Header: được đặt ở đầu file, chứa các dữ liệu mô tả cấu trúc và đặc tính của file. Ta có thể xem phần header của file ELF bằng lệnh *readelf -h [tên file]*.

```
home@home-pc:~/NetBeansProjects/library_demo/shared$ readelf -h main
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1140
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14232 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

  - Ý nghĩa của các trường thông tin trên

Magic	Các byte đầu tiên xác định file có cấu trúc ELF và chứa thông tin cần thiết để tiến trìnhor có thể diễn giải file
Class	Xác định kiến trúc của file là 64-bit hoặc 32-bit
Data	Mô tả cáchlưu trữ dữ liệu mà file sử dụng. Thường là big endian hoặc little endian
Version	Phiên bản ELF sử dụng
OS/ABI	Xác định hệ điều hành và Application Binary Interface (Giao diện nhị phân ứng dụng). Trường này định nghĩa cách các phương thức và cấu trúc dữ liệu của chương trình được truy

	cập
ABI Version	Phiên bản ABI
Type	Xác định loại file.
Machine	Xác định kiến trúc cần để chạy file
Version	Xác định phiên bản của file
Entry point address	Cho biết địa chỉ mà chương trình sẽ bắt đầu thực thi khi được khởi chạy.
Start of program headers	Vị trí bắt đầu của program header
Start of section headers	Vị trí bắt đầu của section header
Flags	Chứa các flag cần thiết
Size of this header	Kích thước của ELF header
Size of program header	Kích thước của mỗi program header
Number of program headers	Số lượng program header
Size of section headers	Kích thước của mỗi section header
Number of section headers	Số lượng section header
Section header string table index	Chỉ mục của section table đại diện bảng tên section

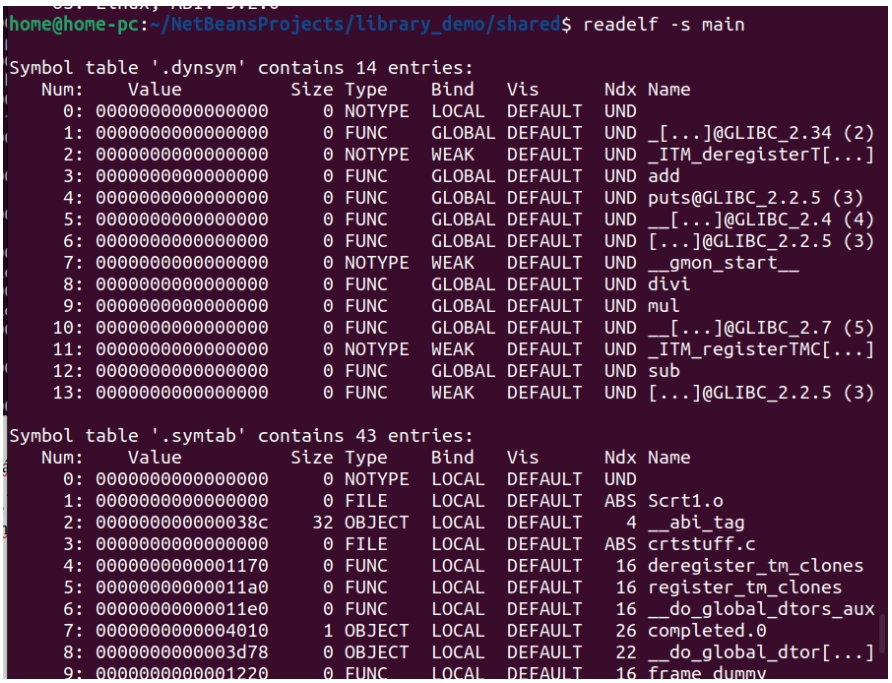
- Program header table:
  - Lưu trữ các thông tin về các segment. Mỗi segment được tạo thành từ một hoặc nhiều section. Kernel dùng các thông tin này tại thời điểm run time để tạo tiến trình và ánh xạ các segment vào memory.
  - Để chạy một chương trình, kernel tải ELF header và Program header table vào memory. Sau đó kernel tiếp tục tải các nội dụng được chỉ định trong phần LOAD của Program header table. Cuối cùng, quyền kiểm xoát được trả lại cho chính file thực thi.
  - Dùng lệnh readelf với tham số -l để xem Program header table của file

```
ome@home-pc:~/NetBeansProjects/library_demo/shared$ readelf -l main

lf file type is DYN (Position-Independent Executable file)
ntry point 0x1140
here are 13 program headers, starting at offset 64

rogram Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000800 0x0000000000000800  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x000000000000036d 0x000000000000036d  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x0000000000000174 0x0000000000000174  R      0x1000
  LOAD           0x0000000000002d70 0x0000000000003d70 0x0000000000003d70
                 0x00000000000002a0 0x00000000000002a8  RW     0x1000
  DYNAMIC        0x0000000000002d80 0x0000000000003d80 0x0000000000003d80
                 0x0000000000000200 0x0000000000000200  RW     0x8
  NOTE           0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  NOTE           0x0000000000000368 0x0000000000000368 0x0000000000000368
                 0x0000000000000044 0x0000000000000044  R      0x4
```

- Section Header Table
  - Lưu trữ thồn tin về các section. Các thông tin này được sử dụng lúc liên kết động, trước khi chương trình được khởi chạy. Linker sẽ sẽ liên kết file nhị phân với các thư viện chia sẻ cần thiết bằng cách tải chúng vào bộ nhớ.
  - Dùng lệnh readelf với tham số -S để xem Section Header Table của file

```
home@home-pc:~/NetBeansProjects/library_demo/shared$ readelf -S main
There are 31 section headers, starting at offset 0x3798:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.pr[...] NOTE             0000000000000338  00000338
       0000000000000030  0000000000000000   A       0     0     8
  [ 3] .note.gnu.bu[...] NOTE             0000000000000368  00000368
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             000000000000038c  0000038c
       0000000000000020  0000000000000000   A       0     0     4
  [ 5] .gnu.hash         GNU_HASH         00000000000003b0  000003b0
       0000000000000024  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           00000000000003d8  000003d8
       0000000000000150  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           0000000000000528  00000528
       00000000000000eb  0000000000000000   A       0     0     1
  [ 8] .gnu.version      VERSYM           0000000000000614  00000614
       000000000000001c  0000000000000002   A       6     0     2
  [ 9] .gnu.version_r    VERNEED          0000000000000630  00000630
       0000000000000050  0000000000000000   A       7     1     8
  [10] .rela.dyn         RELA             0000000000000680  00000680
       00000000000000c0  0000000000000018   A       6     0     8
  [11] .rela.plt         RELA             0000000000000740  00000740
       00000000000000c0  0000000000000018  AI       6    24     8
```

- Symbol table
  - Chứa các thông tin về các định nghĩa và tham chiếu symbol bởi chương trình hoặc thư viện liên kết động. Bảng này được dùng bởi linker để phân giải các tham chiếu symbol giữa các tệp đối tượng và tạo file thực thi hoặc thư viện liên kết động.
  - Dùng lệnh readelf với tham số -s để xem bảng symbol của file:



```
home@home-pc:~/NetBeansProjects/library_demo/shared$ readelf -s main

Symbol table '.dynsym' contains 14 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _[...]@GLIBC_2.34 (2)
     2: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterT[...]
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND add
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (3)
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __[...]@GLIBC_2.4 (4)
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (3)
     7: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     8: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND divi
     9: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND mul
    10: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __[...]@GLIBC_2.7 (5)
    11: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMC[...]
    12: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND sub
    13: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND [...]@GLIBC_2.2.5 (3)

Symbol table '.symtab' contains 43 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS Scrt1.o
     2: 000000000000038c    32 OBJECT  LOCAL  DEFAULT    4 __abi_tag
     3: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
     4: 0000000000001170     0 FUNC    LOCAL  DEFAULT   16 deregister_tm_clones
     5: 00000000000011a0     0 FUNC    LOCAL  DEFAULT   16 register_tm_clones
     6: 00000000000011e0     0 FUNC    LOCAL  DEFAULT   16 __do_global_dtors_aux
     7: 0000000000004010     1 OBJECT  LOCAL  DEFAULT   26 completed.0
     8: 0000000000003d78     0 OBJECT  LOCAL  DEFAULT   22 __do_global_dtor[...]
     9: 0000000000001220     0 FUNC    LOCAL  DEFAULT   16 frame_dummy
```

- Ý nghĩa các trường trên
  - .dynsym: chứa thông tin về các symbol động - được tham chiếu bởi dynamic linker tại thời điểm runtime. Bảng này được dùng bởi dynamic linker để phân giải các tham chiếu symbol giữa các thư viện liên kết động tại thời điểm runtime.
  - .symtab: chứa thông tin vê các symbol tĩnh được định nghĩa hoặc tham chiểu bởi file đối tượng hoặc thư viện tĩnh. Bảng này được dùng bởi linker để phân giải các tham chiếu symbol giữa các file đối tượng tại thời điểm liên kết.

Num	Chỉ mục của symbol.
Value	Giá trị hoặc địa chỉ của symbol tùy vào loại symbol.
Size	Kích thước symbol theo byte.
Type	Loại symbol, có thể là hàm, đối tượng hoặc section,…
Bind	Tính ràng buộc của symbol, có thể là cục bộ (LOCAL), yếu (WEAK) hoặc toàn cục (GLOBAL)
Vis	Khả năng hiển thị của symbol, mặc định (DEFAULT), được bảo vệ(PROTECTED) hoặc ẩn (HIDDEN).
Ndx	Chỉ mục của section chứa symbol
Name	Tên symbol.

- Khác nhau giữa file thực thi trong các kiến trúc máy tính khác nhau:

	32bit	64bit
Kích thước địa chỉ	32 bits	64 bits
Bộ nhớ tối đa có thể truy cập	4 GB	16 hexabytes
Cách truyền tham số	Sử dụng ngăn xếp	Sử dụng thanh ghi

của hàm		
Cơ chế gọi hệ thống	sử dụng lệnh int 0x80	sử dụng lệnh syscall
ABI	System V	System V, GNU
Kích thước ELF header	52 bytes	64 bytes
Khả năng tương thích	Một số hệ thống 64bit có hỗ trợ tương thích với file 32bit thông qua một lớp tương thích hoặc trình giả lập với hiệu suất bị giảm.	Không có khả năng tương thích ngược với hệ thống 32bit.

- Build chương trình với kiến trúc chỉ định bằng cách sử dụng cờ xác định kiến trúc khi biên dịch chương trình
  - -m32 để biên dịch thành file thực thi cho kiến trúc 32bit
  - -m64 để biên dịch thành file thực thi cho kiến trúc 64bit
- Trong trường hợp cần build file thực thi khác kiến trúc với hệ thống đang sử dụng, cần phải đảm bảo hệ thống được cài đặt đầy đủ các công cụ và thư viện cần thiết.

II. Tiến trình trên Linux

- Tiến trình là một chương trình đang thực thi một cách tuân tự.
- Khi một chương trình hoặc lệnh được thực thi, một đại diện cho có được hệ thống cung cấp cho tiến trình, bao gồm tất cả các dịch vụ và tài nguyên có thể được sử dụng bởi tiến trình.
- Tiến trình và quyền truy cập của nó đến các tài nguyên liên quan trực tiếp đến user khởi chạy tiến tình. Ví dụ: hai user với quyền truy cập khác nhau khi chạy cùng một chương trình có thể có được hai kết quả khác nhau
- Khi chương trình được tải lên memory và trở thành tiến trình, nó có thể được chia làm 4 phần: stack, heap, text và data.

Stack	Chứa các dữ liệu tạm thời như tham số của hàm, địa chỉ trả về và biến cục bộ
Heap	Bộ nhớ được cấp phát động cho một tiến trình trong thời điểm runtime.
Text	Chứa các thông tin về hoạt động hiện tại đại diện bằng giá trị của Program Counter và nội dung của các thanh ghi của bộ xử lý.
Data	Chứa các biến toàn cục và biến static

- Trong Linux, tiến trình có thể khởi chạy bằng 2 cách:
  - Foreground Process:
    - Mặc định tất cả tiến trình được khởi chạy trên nền, nhận input từ bàn phím hoặc chuột,… và xuất output ra màn hình, loa,…
    - Khi tiến trình tốn nhiều thời gian khi chạy trên nền, không tiến trình nào khác có thể được khởi động từ command prompt đang chạy tiến trình đó.
  - Background Process:
    - Tiến trình được chạy nền mà không cần input. Do đó các tiến trình khác có thể thực hiện song song với tiến trình được chạy nền mà không cần đợi nó hoàn thành.
    - Để chạy nền một lệnh, thêm & vào lệnh cần chạy.

- Dùng lệnh top để xem thông tin các tiến trình đang hoạt động theo thời gian thực

```
home@home-pc:~$ top

top - 18:59:30 up  1:46,  1 user,  load average: 0.14, 0.12, 0.15
Tasks: 354 total,   1 running, 353 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.4 us,  0.4 sy,  0.0 ni, 99.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  15770.5 total,   4425.6 free,   1493.2 used,   9851.7 buff/cache
MiB Swap:   1956.0 total,   1956.0 free,      0.0 used.  13434.4 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
      1 root      20   0  166780  11824   8200 S   0.0   0.1   0:02.13 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par+
      5 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 slub_fl+
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 netns
      8 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
      9 root      20   0       0      0      0 I   0.0   0.0   0:02.09 kworker+
     10 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_perc+
```

Ý nghĩa các trường trên:

PID	tiến trình ID
USER	Tên hoặc ID của người dùng khởi chạy tiến trình
PR	Độ ưu tiên của tiến trình, được tính toán dựa trênn giá trị "nice" của tiến trình và độ ưu tiên của bbọ lập lịch tiến trình .Độ ưu tiên tỷ lệ nghịch với giá trị này.
NI	Giá trị "nice" của tiến trình, thể hiện cho độ ưu tiên so với các tiến trình khác trên cùng hệ thống,
VIRT	Kích thước bộ nhớ ảo của tiến trình, là dung lượng bộ nhớ ảo mà mà tiến trình đang sử dụng.
RES	Dung lượng bộ nhớ vật lý mà tiến trình sử dụng. Cho biết tiến trình thực sự sử dụng bao nhiêu bộ nhớ trên RAM
SHR	Kích thước bộ nhớ chia sẻ của tiến trình, là dung lượng bộ nhớ dùng chung với các tiến trình khác.
S	Trạng thái của tiến trình. Có thê là R (running), S (sleeping), Z (zombie), hoặc khác.
%CPU	Phần trăm CPU mà tiến trình sử dụng. Cho biết tiến trình sử dụng bao nhiêu phần tài nguyên của bộ sử lý.
%MEM	Phần trăm bộ nhớ vật lý mà tiến trình sử dụng. Cho biết tiến trình sử dụng bao nhiêu phần bộ nhớ vật lý của hệ thống.
TIME+	Tổng thời gian CPU sử dụng bởi tiến trình dưới dạng giờ, phút và giây. Cho biết thời gian procces sử dụng CPU kể từ lúc tiến trình được khởi chạy
COMMAND	Tên và tham số truyền vào của lệnh khởi chạy tiến trình. Cho biết tiến trình đang thực hiện công việc gì.

- Trạng thái tiến trình:
  - Running: tiến trình đang chạy hoặc đã chuẩn bị để chạy (đang chờ để được cấp phát một phần tài nguyên CPU).
  - Waiting: tiến trình đang đợi một event hoặc tài nguyên. Linux phân biệt waiting tiến trình thành hai dạng: interruptible và uninterruptible. Interruptible waiting process có thể bị gián đoạn trong khi uninterruptible waiting process đợi trực tiếp trên phần cứng và không thể bị gián đoạn dưới bất kỳ trường hợp nào.

    - Stopped: tiến trình đã bị dừng, thường là do nhận dược tín hiệu. Một tiến trình đang được debbug có thể đnag trong trạng thái này.
    - Zombie: tiến trình đã bị hủy nhưng thông tin của chúng vẫn còn tồn tại trong trang thái tiến trình hoặc bảng tiến trình.
- Phân biệt tiến trình:
    - Parent/Child Process: Một tiến trình có thê tạo ra nhiều tiến trình khác, được gọi là child tiến trình. tiến trình tạo ra các child process được gọi là parent process. Quan hệ này được gọi là tiến trình hierarchy. Child tiến trình được kế thừa các thuộc tính của parent process tạo ra nó, bao gồm các biến môi trường, các file được mở, thư mục làm việc hiện tại,… Child process còn có thể nhận một bản copy của code và data segment và có thể tùy chỉnh độc lập với bản gốc. Ngoài ra, một process user A cũng có thể tạo một child process user B bằng cách thiết lập lại uid hoặc gid của child process với điều kiện user A phải là admin hệ thống hoặc được cấp quyền super user.
    - Orphan Process: sau khi hoàn thành nhiệm vụ, child tiến trình sẽ bị hủy và phát tín hiệu đến parent process để tiếp tục công việc. Nhưng trong một số trường hợp, parent process bị hủy trước child process, lúc này đã trở thành orphan process.
    - Daemon Process: các background process liên quan đến hệ thống thường được chạy dưới quyền của root hoặc là các service được gọi bởi các tiến trình khác.

III. Quản lý tiến trình trên Linux

- Mọi tiến trình đều chạy một phần ở chế độ người dùng và một phần ở chế độ hệ thống. Mỗi khi một quá trình thực hiện một system call, nó sẽ hoán đổi từ chế độ người dùng sang chế độ hệ thống và tiếp tục thực thi.
- Các chế độ này được hỗ tợ bởi phần cứng theo cách khác nhau nhưng nhìn chung có một cơ chế an toàn để chuyển từ chế độ người dùng sang chế độ hệ thống và ngược lại. Trong Linux, tiến trình không thể dừng các tiến trình các tiến trình đang chạy để có thể chạy. Mỗi tiến trình sẽ từ bỏ CPU mà nó được gán vào khi phải đợi môt sự kiện, lúc này, một tiến trình khác sẽ được chọn để chạy.
- Các tiến trình luôn phải gọi các system call nên thường sẽ nằm trong trạng thái đợi. Dù vậy, nếu một tiên stình được chạy cho đến khi vào trạng thái chờ thì vẫn có khả năng nó sử dụng một lượng lớn tài nguyên CPU do vậy Linux sử dụng cơ chế lập lịch ưu tiên để quản lý các tiến trình. Trong cơ chê này, mỗi quy tình chỉ được phép chạy trong một khoảng thời gian nhỏ (time-slice) - 200 milli giây và khi thời gian này hết, một tiến trình khác sẽ được chọn để chạy và quy tình ban đầu sẽ được đưa vào trạng thái đợi một lúc cho đến khi có thể chạy lại.
- Linux sử dụng một thuật toán tính toán độ ưu tiên để lựa chọn tiến trình nào được phép chạy trong các tién trình đã ở trạng thái sẵn sàng. Khi một tiến trình được chọn để chạy, Linux sẽ lưu lại trạng thái của tiến tình hiện tại, các thanh ghi cụ thể của bộ xử lý và một số thông tin khác. Sau đó trạng thái của tiến trình sẽ được khôi hục để tiếp tục chạy.
- Một số thông tin của tiến trình được lưu trữ để bộ lập lịch có thể phân bổ tài nguyên CPU một cách công bằng giữa các tiến trình đang chạy trên hệ thống:
    - **policy**
        - Chính sách của bộ lập lịch sẽ được áp dụng cho tiến trình. Có hai loại tiến trình được xác định bởi chính sách này là tiến trình bình thường và tiến trình thời gian thực. Các tiến trình thời gian thực có độ ưu tiên cao hơn các tiến trình khác và sẽ luôn được chạy trước nếu đã sẵn sàng.
        - Các tiến trình thời gian thực có thể có hai loại chính sách là round robin và FIFO.

- **priority**
  - Độ ưu tiên được bộ lập lịch gán cho tiến trình. Đây cũng là tổng thời gian theo tick mà tiến trình được phép chạy mỗi lần chạy.
- **rt_priority**
  - Độ ưu tiên được bộ lập lịch gán cho các tiến trình thời gian thực
- **counter**
  - Tổng thời gian (theo tick) còn lại mà tiến trình được phép chạy. Được set bằng với priority lúc bắt đầu chạy và giảm dần với mỗi clock tick.

- Bộ lập lịch được chạy từ nhiều vùng bên trong kernel, nó được chạy sau khi tiến trình hiện tại được đặt vào hàng đợi dừng và đội lúc cũng được chạy khi kết thúc việc gọi hệ thống, ngay trước khi tiến trình thoát khởi chế độ hệ thống. Mỗi lần bộ lập lịch chạy, nó sẽ thực hiện các công việc sau:
  - **kernel work**
    - Bộ lập lịch chạy các trình xử lý nửa dưới và xử lý hàng đợi tác vụ.
  - **current process**
    - Nếu **policy** tiến trình hiện tại là round robin, nó sẽ được đặt vào cuối hàng đợi được chạy.
    - Nếu tiến tình thuộc loại INTERRUPTIBLE trang thái của nó sẽ được đặt thành RUNNING.
    - Nếu tiến trình bị time out trạng thái của nó sẽ được đặt thành RUNNING.
    - Nếu tiến trình đang chạy thì trạng thái của nó sẽ được giữ nguyên.
    - Nếu tiến trình không ở trong trạng thái RUNNING sẽ bị loại bỏ khởi hàng đợi được chạy, tức tiến trình này sẽ không được xem xét khi bộ lập lịch lựa chọn tiến trình tiếp theo được khở chạy
  - **process selection**
    - Bộ lập lịch dựa trên độ ưu tiên của mỗi tiến trình để lựa chọn tiến trình phù hợp nhất để chạy. Với tiến trình thường độ ưu tiên là priority còn với tiến trình thời gian thực thì độ ưu tiên là priority + 1000. Nếu nhiều tiến trình có cùng độ ưu tiên, tiến trình có vị trí cao nhất trong hàng đợi được chạy sẽ được chọn và tiến trình hiện tại sẽ được đặt vào cuối hàng đợi này.
  - **swap process**
    - Nếu tiến trình được chọn để chạy không phải tiến trình hiện tại, tiến trình hiện tại sẽ bị tạm dừng và tiến trình được chọn sẽ được khởi chạy.

- Trong hệ thống đa bộ xử lý, mỗi bộ xử lý sẽ chạy một bộ định thời riêng biệt.

IV. Thư mục /proc

- Hệ thống tệp "/proc" là mộtt hệ thống giả tệp (pseudo-filesystem) trong Linux được sử dụng để hiển thị thông tin về hệ thống và quá trình của nó trong một cấu trúc giống như tệp hệ thống phân cấp.
- Hệ thống tệp này cung cấp một cách thức cho người dùng và các chương trình để truy cập và thao tác thông tin về hệ thống và các tiến trình trên hệ thống thông qua sự tương tác với các tệp và thư mục trong hệ thống tệp này.
- Các tệp và thư mục trong "/proc" không phải là các tệp và thư mục thực sự trên đĩa cứng, mà là các tệp và thư mục ảo được tạo ra động bởi kernel khi cần thiết.
- Hệ thống tệp "/proc" cung cấp một số lượng lớn thông tin về hệ thống, bao gồm thông tin về các quá trình, cấu hình hệ thống, các thiết bị phần cứng và nhiều hơn nữa.

- Một số tệp và thư mục được sử dụng phổ biến nhất trong "/proc" bao gồm:
  - /proc/cpuinfo: thông tin về CPU trên hệ thống
  - /proc/meminfo: thông tin về việc sử dụng bộ nhớ trên hệ thống
  - proc/[id]: thông tin về các tiến trình trên hệ thống.
- Trong hệ thống tập tin "/proc" của Linux, mỗi tiến trình trên hệ thống được đại diện bằng một thư mục có tên giống với PID của nó.
- Bên trong thư mục cho mỗi tiến trình, có nhiều tệp và thư mục con cung cấp thông tin về tiến trình và trạng thái của nó:
  - cmdline: chứa các tham số dòng lệnh được sử dụng để khởi động tiến trình.
  - cwd: chứa thư mục làm việc hiện tại của tiến trình.
  - environ: chứa các biến môi trường cho tiến trình.
  - maps: chứa thôn tin các vùng nhớ đã được map và quyền quyền truy cập đến chúng.
  - stat: chứa các thông tin khác nhau về tiến trình như pid, ppid, trang thái,…
  - statm: chứa các thông tin về sử dụng bộ nhớ của tiến trình.
- Nội dung của các thư mục "/proc/PID" được tạo ra động bởi kernel và luôn thay đổi khi tiến trình chạy. Do đó, Nhiều tiện ích hệ thống và công cụ giám sát sử dụng thông tin trong thư mục "/proc/PID" như một công cụ mạnh mẽ để cung cấp thông tin thời gian thực về hành vi của từng tiến trình trên hệ thống.
- Một số rủi ro khi sử dụng /proc:
  - Vô tình sửa đổi các tham số hệ thống: Một số tệp trong hệ thống tập tin "/proc" có thể được sửa đổi để thay đổi hành vi của hệ thống, nếu việc sửa đổi không đúng cách có thể gây ra hậu quả không mong muốn và có thể gây hại cho hệ thống.
  - Lộ thông tin: Hệ thống tập tin "/proc" có thể tiết lộ thông tin nhạy cảm về hệ thống và các tiến trình của nó.
  - Tiêu thụ tài nguyên: Theo dõi các tiến trình bằng cách sử dụng hệ thống tập tin "/proc" có thể tiêu thụ nhiều tài nguyên hệ thống, đặc biệt là nếu nhiều tiến trình được theo dõi cùng một lúc.

V. Interprocess Communication

- Các tiến trình giao tiếp với nhau và với kernel phối hợp các hoạt động. Linux hỗ trợ một số cơ chế Inter-Process Communication (IPC) - Giao tiếp liên tiến trình hỗ trợ cho việc giao tiếp này.

1. Signals

- Tín hiệu là một trong những phương pháp giao tiếp liên tiến trình được sử dụng bởi các hệ thống UNIX. Chúng được dùng để báo hiệu các sự kiện không đồng bộ đến các tiến trình, ngoài ra còn được sử dụng bởi shell để báo hiệu các lệnh điều khiển cho các tiến trình con của chúng.
- Để xem các loại tín hiệu bằng lệnh kill -l

```
home@home-pc:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

- Ví dụ:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void handler(int sig) {
    printf("\nReceived signal %d\n", sig);
}

int main() {
    // Register the signal handler for SIGINT
    signal(SIGINT, handler);

    // Wait for the signal to be received
    printf("Waiting for signal...\n");
    pause();

    return 0;
}
```

Sau khi chạy chương tình trên và gửi một signal đến chương trình (Ctrl + C) ta được kết quả:

```
home@home-pc:~/NetBeansProjects$ ./demo
Waiting for signal...
^C
Received signal 2
```

2. Pipe

- Chuyển hướng output của lệnh này làm input của lệnh tiếp theo, được sử dụng bằng cách thêm | vào giữa hai lệnh.
- Ví dụ in ra thông tin các tệp tại thư mục làm việc hiện tại có tháng được sửa cuối cùng là tháng 3:

```
home@home-pc:~$ ls -l | grep Mar
drwxr-xr-x 2 home home 4096 Mar 23 01:21 Desktop
drwxr-xr-x 2 home home 4096 Mar 24 23:25 Downloads
drwxr-xr-x 2 home home 4096 Mar 23 01:21 Music
-rw------- 1 home home 2053 Mar 27 22:58 Passwords.kdbx
drwxr-xr-x 3 home home 4096 Mar 27 19:52 Pictures
drwxrwxr-x 3 home home 4096 Mar 23 01:48 Programs
drwxr-xr-x 2 home home 4096 Mar 23 01:21 Public
drwx------ 4 home home 4096 Mar 23 03:30 snap
drwxr-xr-x 2 home home 4096 Mar 23 01:21 Templates
drwxr-xr-x 2 home home 4096 Mar 23 01:21 Videos
drwxrwxr-x 3 home home 4096 Mar 24 23:06 vmware
```

3. Message Queues

- Hàng đợi lời nhắn cho phép các tiến trình ghi lời nhắn, thứ có thể được đọc bởi các tiến trình khác. Hệ thống hỗ trợ sử dụng hàng đợi này qua các lời gọi: msgget(), msgsnd() và msgrcv().
- Để sử dụng hàng đợi lời nhắn, một tiến trình sẽ tạo hoặc kết nối đến một hàng đợi lời nhắn có sẵn xử dụng lời gọi msgget(). Lời gọi hệ thống này trả về một định danh của hàng đợi lời nhắn mà tiến trình có thể dùng để gửi và nhận lời nhắn.
- Sau khi đã kết nối thành công, tiến tình có thể gửi và nhận lời nhắn sử dụng msgsnd() và msgrcv().
- Ví dụ:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MESSAGE_SIZE 50

struct message {
   long mtype;
   char mtext[MESSAGE_SIZE];
};

int main() {
//Create the message queue
key_t key = ftok("/tmp", 'a');
int msgid = msgget(key, 0666 | IPC_CREAT);

//Send a message to the message queue
struct message msg;
msg.mtype = 1;
snprintf(msg.mtext, MESSAGE_SIZE, "Hello, world!");
msgsnd(msgid, &msg, MESSAGE_SIZE, 0);

//Receive a message from the message queueC
  msgrcv(msgid, &msg, MESSAGE_SIZE, 1, 0);
```

```
   printf("Received message: %s\n", msg.mtext);

//         Clean up the message queue
msgctl(msgid, IPC_RMID, NULL);

return 0;
}
```

4. Shared Memory

- Trong, bộ nhớ chia sẻ là một cơ chế giao tiếp giữa các tiến trình cho phép nhiều tiến trình chia sẻ một khu vực bộ nhớ. Bộ nhớ chia sẻ có thể được sử dụng để trao đổi lượng dữ liệu lớn giữa các tiến trình mà không gây ra tình trạng chồng chéo dữ liệu giữa các tiến trình.
- Bộ nhớ chia sẻ trong Linux được triển khai bằng cách sử dụng các lời gọi hệ thống shmget(), shmat() và shmdt().
- Lời gọi hệ thống shmget() được sử dụng để tạo một khu vực bộ nhớ chia sẻ mới hoặc truy cập vào một khu vực bộ nhớ chia sẻ hiện có và trả về một định danh của bộ nhớ chia sẻ mà có thể được sử dụng cho các lời gọi hệ thống khác.
- Lời gọi hệ thống shmat() được sử dụng để gắn kết với khu bộ nhớ chia sẻ, làm cho nó có thể truy cập được từ một tiến trình. Nó trả về một con trỏ đến bộ nhớ chia sẻ đó, có thể được sử dụng để đọc từ hoặc ghi vào bộ nhớ chia sẻ.
- Lời gọi hệ thống shmdt() được sử dụng để tách khỏi một bộ nhớ chia sẻ, giải phóng các tài nguyên được sử dụng bởi bộ nhớ chia sẻ.
- Các bộ nhớ chia sẻ có một khóa duy nhất và được xác định bằng một số nhận dạng bộ nhớ chia sẻ. Các bộ nhớ chia sẻ có thể được truy cập bởi nhiều tiến trình cùng một lúc, và các thay đổi được thực hiện trên bộ nhớ chia sẻ bởi một tiến trình sẽ ngay lập tức được hiển thị với tất cả các tiến trình khác đã gắn kết với đoạn bộ nhớ đó.

- Ví dụ:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024

int main() {
   // Create the shared memory segment
   key_t key = ftok("/tmp", 'a');
   int shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);

   // Attach to the shared memory segment
   char *shmaddr = shmat(shmid, NULL, 0);

   // Write a message to the shared memory segment
```

```
    snprintf(shmaddr, SHM_SIZE, "Hello, world!");

    // Read the message from the shared memory segment
    printf("Received message: %s\n", shmaddr);

    // Detach from the shared memory segment
    shmdt(shmaddr);

    // Destroy the shared memory segment
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

5. **Socket**
   - Socket còn có thể được dùng để giao tiếp giữa các tiến trình ngoài được dùng để giao tiếp trong hệ thống mạng
   - Để sử dụng Socket cho IPC, tiến trình trước hết cần tạo và liên kết đến một socket bằng local address
   - Khi các tiến trình đã liên kết với cùng một socket, chúng có thể gửi nhận dữ liệu giống như giao tiếp qua mạng

6. **So sánh ưu khuyết điểm của các loại IPC**

	Ưu điểm	Nhược điểm
Signal	Đơn giản và hiệu quả	Không hỗ trợ trao đổi dữ liệu giữa các tiến trình
Pipe	Dễ sử dụng và hiệu quả	Chỉ có thể giao tiếp một chiều
Message queues	Đáng tin cậy và linh hoạt	Kém hiệu quả hơn các cách khác
Shared memory	Nhanh và hiệu quả	Yêu cầu đồng bộ cao
Sockets	Linh hoạt và mạnh mẽ	Phức tạp và yêu cầu đồng bộ cao

7. **Các vấn đề về bất đồng bộ.**
   - Race condition: xảy ra khi nhiều tiến trình truy cập một tài nguyên được chia sẻ mà không được đồng bộ hóa đúng cách. Ví dụ khi nhiều tiến trình ố gắng cập nhật cùng biến chia sẻ cùng một lúc. Tùy thuộc vào thứ tự của các cập nhật được thực hiện, một hoặc nhiều trong số các cập nhật có thể bị mất, dẫn đến kết quả không chính xác.
   - Deadlock: Deadlock xảy ra khi hai hoặc nhiều tiến trình đang đợi nhau trả về một tài nguyên hệ thống và chúng đều không thể tiếp tục dẫn đến tro chương trình
   - Để tránh các vấn đề trên, cần có một phương pháp đồng bộ thật sự tốt, ví dụ như lock hoặc semaphores.
     - Lock là một nguyên tắc đồng bộ hóa cho phép chỉ một tiến trình hoặc luồng truy cập vào một tài nguyên chia sẻ tại một thời điểm. Khi một tiến trình hoặc luồng cố gắng truy cập vào tài nguyên chia sẻ, nó trước tiên cố gắng lấy khóa, và nếu khóa đã được giữ bởi một tiến trình hoặc luồng khác, nó sẽ bị chặn cho đến khi khóa được giải phóng.
     - Semaphore là một nguyên tắc đồng bộ hóa được sử dụng để kiểm soát truy cập đến tài nguyên chia sẻ bởi nhiều tiến trình hoặc luồng. Semaphore duy trì một bộ đếm đại diện cho số tiến

trình được phép truy cập vào tài nguyên chia sẻ tại một thời điểm. Khi một tiến trình muốn truy cập vào tài nguyên chia sẻ, nó trước tiên cố gắng giảm bộ đếm. Nếu bộ đếm lớn hơn không, tiến trình được phép truy cập vào tài nguyên chia sẻ, và bộ đếm được giảm. Nếu bộ đếm bằng không, tiến trình bị chặn cho đến khi một tiến trình khác giải phóng Semaphore và tăng bộ đếm.

VI. Gọi một tiến trình thực thi từ một tiến trình khác.

- Có thể gọi lừoi gọi hệ thống folk() và exec() để tạo một tiến trình từ một tiến trình khác.
- Hàm: int c pid_t fork(void);

Tạo một tiến trình mới bằng cách nhân bản proccess gọi nó, trong trường hợp gọi từ một chương trình, process được tạo sẽ chứa đoạn code giống với chương tình tọa ra nó.
Giá trị trả về: ID của tiến trình được tạo với parrent proccess và 0 với child process. Trong trường hợp có lỗi xảy ra, hàm trả về -1.

- Các hàm: exec()

Dùng để thay thế image của process hiện tại bằng một image khác.
Chỉ trả về -1 khi có lỗi xảy ra.

- Ví dụ:

main.c
~/NetBeansProjects

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    pid_t pid;
    int status;

    pid = fork();

    if (pid == 0) {
        // Child process
        printf("Start child process %d\n", status);
        execl("/bin/ls", "ls", "-l", NULL);
        return 1;
    } else {
        // Parent process
        waitpid(pid, &status, 0);
        printf("Child process exited with status %d\n", status);
    }

    return 0;
}
```

- Kết quả sau khi build và chạy:

```
home@home-pc:~/NetBeansProjects$ gcc -o main main.c
home@home-pc:~/NetBeansProjects$ ./main
Start child process 4096
total 944
drwxrwxr-x 4 home home   4096 Apr 10 17:10 file_io_2
drwxrwxr-x 3 home home   4096 Apr  3 22:10 find-text-in-file
drwxrwxr-x 3 home home   4096 Apr  3 22:11 find_text_length
drwxrwxr-x 5 home home   4096 Apr 13 01:28 library_demo
-rwxrwxr-x 1 home home  16136 Apr 15 02:12 main
-rw-rw-r-- 1 home home    490 Apr 15 02:11 main.c
drwxrwxr-x 5 home home   4096 Apr  7 18:42 pointer
-rwxrwxr-x 1 home home 925079 Apr 15 02:11 SE05.odt
Child process exited with status 0
```

VII. Một số thuật toán sắp xếp có độ hiệu quả cao

1. Merge Sort
   - Là một thuật toán sắp xếp sử dụng phương pháp chia để trị.
   - Ý tưởng cơ bản của Merge Sort là chia mảng đầu vào thành hai nửa, sau đó sắp xếp chúng rồi trộn lại thành mảng đã sắp xếp.
   - Độ phúc tạp tệ nhất là O(nlog(n)).
   - Merge Sort có tính ổn định - bảo toàn thứ tự tương đối của các phần tử bằng nhau.
   - Nhược điểm là yêu cầu sử dụng bộ nhớ để lưu trữ các mảng tạm thời được sử dụng trong quá trình trộn. Thuộc tính này quan trọng trong một số ứng dụng, chẳng hạn như sắp xếp cơ sở dữ liệu theo nhiều tiêu chí khác nhau.
   - Merge Sort hiệu quả khi tính ổn định là quan trọng và sử dụng bộ nhớ không phải là một vấn đề.

2. Quick Sort
   - Là một thuật toán sắp xếp sử dụng phương pháp chia để trị.
   - Ý tưởng cơ bản của Quick Sort là chọn một phần tử pivot từ mảng, chia mảng thành hai mảng con dựa trên phần tử pivot đó, và sau đó đệ quy sắp xếp các mảng con.
   - Lựa chọn phần tử pivot có thể ảnh hưởng đáng kể đến hiệu suất của thuật toán. Lý tưởng nhất, pivot nên là phần tử trung vị trong mảng, vì điều này dẫn đến các phân vùng cân bằng nhất. Tuy nhiên, tìm phần tử trung vị có thể tốn nhiều thời gian tính toán.
   - Thuật toán có độ phức tạp trung bình là O(nlog(n)). Thuật toán có độ phức tạp thời gian tệ nhất là O(n^2), xảy ra khi phần tử pivot được chọn không tốt và dẫn đến các phân vùng không cân bằng.
   - Quick Sort không có tính ổn định - không bảo toàn thứ tự tương đối của các phần tử bằng nhau.
   - Quick Sort thường nhanh hơn các thuật toán khác và sử dụng bộ nhớ đệm hiệu quả hơn.
   - Quick Sort hiệu quả khi hiệu suất trung bình quan trọng hơn hiệu suất tệ nhất và tính ổn định.

3. Heap Sort
   - Heap Sort là một thuật toán sắp xếp phổ biến sử dụng cấu trúc dữ liệu cây nhị phân heap.
   - Một cây nhị phân heap là một trường hợp đặc biệt của cấu trúc dữ liệu cây nhị phân, trong đó khóa của nút gốc được so sánh với các con của nó và được sắp xếp một cách phù hợp.c
   - Ý tưởng cơ bản của Heap Sort là trước tiên xây dựng một cây nhị phân từ mảng đầu vào, sau đó lặp đi lặp lại lấy ra phần tử tối đa (hoặc tối thiểu) từ cây và đặt nó vào cuối mảng đầu ra. Quá trình này được lặp lại cho đến khi tất cả các phần tử đã được lấy ra và mảng đầu ra đã được sắp xếp.
   - Độ phúc tạp tệ nhất là O(nlog(n)).
   - Heap Sort không có tính ổn định - không bảo toàn thứ tự tương đối của các phần tử bằng nhau.
   - Một nhược điểm của Heap Sort là nó không hiệu quả trong việc sử dụng bộ nhớ đệm so với một số thuật toán sắp xếp khác, chẳng hạn như Quick Sort, do việc sử dụng cấu trúc dữ liệu heap nhị phân. Ngoài ra, nó yêu cầu bộ nhớ phụ để lưu trữ heap, điều này có thể gây lo ngại đối với các mảng lớn.
   - Heap Sort hiệu quả khi dữ liệu đầu vào đã có dạng của một heap nhị phân hoặc khi tính ổn định không quan trọng.