オペレーティングシステム演習

第6回（2009.05.21）

割り込みベクトルのトレース

割り込み処理の目的

- 目的
  - 周辺機器からの信号を受け付ける。
    - キー入力、マウス処理、通信データなど、コンピュータ側が予測できない「処理要求」を処理する。
    - ハードウェア割り込み
  - システムコールを発行する。
    - 「割り込み」がOSの処理の受付窓口
    - ソフトウェア割り込み
  - 異常事態に対処する
    - 割り算エラーなどに対処する。

割り込みの種類

- テキストP42
  - タイマー割り込み
    - レポートでは、「内部割込み」としたものがあったが、設計によって異なる。
    - タスク切り替え
      - マルチタスク処理で、一つのタスクが「持ち時間」を終了したような場合
- カーネルの入り口
  - システムコール
    - ユーザプログラムから制御を切り替えるため、割り込みを発行する。(プロセスIDが手渡される。)

トレースとは

- プログラムの「流れ」を辿ることである！
  - 他人の考え方を理解する。
  - 「名作」を読もう！

- 良質なプログラムは・・・
  - if-else/switch-case/while-doなどで
    - 流れを制御する。
  - 共通の処理の本体は、「関数」で定義されている。
  - 受け渡しする「引数」が必要最小限にまとめられている。
    - 構造体を活用している。
  - コメントが多くて、読みやすい

辿る手順（例えば・・・）

- 「糸口」をつかむ。
  - 「関数」の名前を、最初に調べる。
    - 好きな人ができたら、その人の名前を知りたいと思う。
  - 「関数」は、Javaでのメソッドと同じ。
- 名前がわかったら、次は場所を調べる。
  - どの「ファイル」で定義されているか、調べる。
    - 好きな人が、どこに住んでいるか興味を持つ
  - メソッドが所属するclassを調べる。
  - ⇒　この時に、活躍するのがgrepコマンド

トレースの方法（再掲）

- lsで、ファイルの一覧を取得
- grepで、プログラム内部の記述を探す。
  - grep　put_irq_handler　*.c
  - そのディレクトリにない場合には、
    - grep　irq_table　*/*.c
- リダイレクション（　＞　）
  - 標準出力した内容を、ファイルに落とす。
  - grep do_read *.c > searchread
- moreコマンド
- パイプ　｜　標準出力を標準入力に切り替える
  - grep　do_read　*.c｜　more

トレースの方法(2)

- “do_chroot”を探しているとする。

  grep do_chroot *.c
  - と入力する。
  - 誤って、*.cを忘れた場合
    - [ctrl]+Dと入力する。
      - 標準入力(パイプ)からの入力と見なされているため、キー入力が「入力」になってしまった。そこで、[Ctrl]+Dを入力すると、標準入力パイプのEOF(End of File)を送信して、閉じることができる。
  - 結果が表示されたら、どのファイルにあるかわかる。

  - 目的とするファイルが見つかったら、そのファイルを開く。
  - vi でファイルを開いたら、/do_chrootと入力して、文字列を探す。

検索した結果が、膨大になったら・・・

- リダイレクション（　＞　）
  - 標準出力した内容を、ファイルに落とす。
  - grep do_read *.c > searchread
    - リダイレクションするファイルが存在していないことを予め確認しておく。
    - 存在しているファイルにリダイレクトすると、上書きされてしまう。
  - ファイルに落としてから、ゆっくりファイルの中身を参照すればよい。

- moreコマンド
- 長いファイルを読むためのコマンド

- リダイレクションでファイルに落とさずに、直接閲覧することもできる。
  - ⇒「パイプ」を使用する。
- パイプ　｜　標準出力を標準入力に切り替える
  - grep　do_read　*.c｜　more

moreの使い方

- more　ファイル名と入力する
  - [SPACE]　次の画面を表示する
  - b　前の画面を表示する
  - h　使い方の説明を表示する
  - q　moreを終了する

パイプ(pipe)とは

- テキスト P26
  - パイプの前のコマンドの「標準出力」を
  - パイプの後のコマンドの「標準入力」につなぐ。

  command1 | command2
  - 二つのコマンドを、縦棒（ | ）でつなぐだけ。

- UNIXが提供するIPC(プロセス間通信）機能の一つ
  - テキストP114

- ソースコードでは・・・
  - /dev/usr/fs　read.cのread.c　I_PIPEの名称でチェックすると、パイプ処理の「分岐」が見つかる。

最初の課題

- 自分の学籍番号の下２桁を、FSに対する割り込み処理番号として、どの関数がコールされるか名称を調べる。
- その名称が、どのファイルで定義されているかを調べる。
- 「本体定義」されている部分の行番号を調べる。

C言語の「宣言」と「定義」・「コール」

- 型の宣言

  extern int func_A(char, int ・・・);
  - extern :外部宣言（別の部分で定義していることを明示している。）
  - 何のためにexternを書くか？
    - 他のファイル（あるいは、そのファイルの下の方）で定義されている関数を引用するのに、型を通知している。
- 定義部分

  int func_A(char, int ・・・)

  {

  　　関数の定義部分

  }
  - { }内部で、関数本体を定義している。
- 関数コール部分:
  - y = func_A(x);
    - 制御が関数に移り、return文で元のプログラムに戻る。

コールしているところから、「定義部分」に制御が移る

間接参照

- int　*a, b;
  - bは、整数型の変数（数値の入れ物）
  - aは、「入れ物」のアドレスを示す。
    - a = malloc(sizeof(int));
    - *a = 10;

    - 「入れ物」の場合には、メモリを割り付けて使う。
  - サイズが変わったり、用途によって別の場所を示す場合に使い分ける。

構造体

```
struct   dmapA  {
        int   A;
        char B;
        struct dmap *next;
} dmapB;
```

- dmapAという「構造体名」で、dmapBという変数を定義する場合。

- 構造体とは・・・
  - 複数の種類データを「一まとめ」に扱う。
  - データのコンテナ（乗り合いバス）のようなもの
  - Structという予約語で定義されている

- 上の例では、整数型のAと、文字型のB、そして、「自分自身の型を示す「ポインタ」を宣言している。

  - 自分自身へのポインタ ⇒ チェーン構造を表現するのに使える。

「情報隠蔽」(Information Hiding)

- 無駄なパラメータを外に見せない
  - プログラムの「カプセル化」/「モジュール化」
- 適切な「構造体」を定義すると、構造体へのポインタを手渡しするだけで、情報の受け渡しが完了する。
  - ⇒ この流れで、C++のClassがデザインされ、methodや、attributeが「クラス」内部にカプセル化された。
    - Classと、instanceが分けられた。
- 呼び出しが単純になる。
  - student -> submit (report);
  - 「学生」インスタンスの「提出」methodを、「レポート」を引数として実行する。
  - プログラムを読んでいて、何をやっているか理解しやすい。
  - Studentの属するClass(元構造体)や、reportの属するClassには、それぞれのクラスに必要なパラメータが定義されている。

今日の課題

- /usr/src/fsの中にあるファイルでトレースを続ける。

- 前回のトレースで、
  int SYSVEC
- までたどり着いた。その続きをトレースする。
  - /usr/src/fs内部のファイル処理
  - /usr/src/kernelに処理を手渡すまで

宿題

- 今週も、宿題はありません。
- 授業時間内に「課題」の空欄を埋めて、授業時間内に提出して下さい。
  - 提出したら帰って構いません。