OS 制作

金田　航輝

１．研究概要

プログラムを制作する上でブラックボックス化しがちな OS の基本的な動作を学ぶために簡易的な OS の制作を行った。

２．研究の具体的内容

（１）環境構築

今回の制作では Linux 上で開発する必要があったので、Windows11 上で wsl を使用し、ubuntu を実行することで、OS を制作した。

さらに Linux 上で QEMU というエミュレータを使用し、制作した OS を実行した。

また統合開発環境として、Visual Studio Code を使用し、バージョン管理では git を使用した。

・wsl

Windows 上で Linux 環境を動作させる機能

・ubuntu

Linux を基にした OS の１種

（２）Hello World!

まず、他の OS の機能を使わずに画面にメッセージを表示するために、バイナリエディタでプログラムを制作した。（図１）

```
00 00 00 00 00 00 00 00 00 00 00 00 20 00 50 60
2E 72 64 61 74 61 00 00 1C 00 00 00 00 20 00 00
00 02 00 00 00 04 00 00 00 00 00 00 00 00 00 00
00 00 00 00 40 00 50 40 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
48 83 EC 28 48 8B 4A 40 48 8D 15 F1 0F 00 00 FF
51 08 EB FE 00 00 00 00 00 00 00 00 00 00 00 00
```

図１　バイナリの例

（３）EDK II

EDK II とは intel が UEFI とその周辺プログラムを実装し、それがのちにオープンソースとして公開されたものである。今回は EDK II を使用してメモリマップの表示を行った。（図２）

```
Hello, okako World!
map->buffer = 3FEA8A30, map->map_size = 00000840
All done
```

図２　画面に文字を表示する様子

（４）ブートローダー

EDK II では UEFI の開発ができるが、UEFI アプリには UEFI で決められた規格通りにしか制作できない。そのためカーネルというファイルを UEFI と分けて作ることでその制約を気にしないで制作することができる。

（５）描画

OS を作るうえでは文字を表示するためには、まず画面上のどのピクセルであってもどのような色であっても描画できる必要がある。（図３）

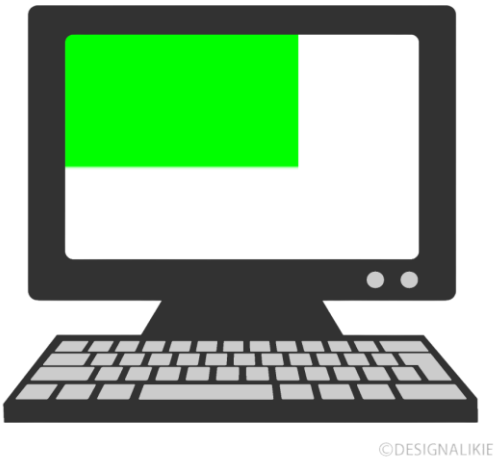

図３　画面に表示するイメージ

次にその機能を使用することによって文字の表示を行う。(図４)文字を描画するために、色を描画する場所を「１」、色を描画しない場所を「0」、とした２次元配列を使用して文字の描画を実現する。(図５)

図４　画面上に文字 AA を表示するイメージ

```
// #@@range_begin(font_a)
const uint8_t kFontA[16] = {
  0b00000000, //
  0b00011000, //    **
  0b00011000, //    **
  0b00011000, //    **
  0b00011000, //    **
  0b00100100, //   *  *
  0b00100100, //   *  *
  0b00100100, //   *  *
  0b00100100, //   *  *
  0b01111110, //  ******
  0b01000010, //  *    *
  0b01000010, //  *    *
  0b01000010, //  *    *
  0b11100111, // ***  ***
  0b00000000, //
  0b00000000, //
};
// #@@range_end(font_a)
```

図５　文字 A の配列

またすべての英数字に対してフォントを手入力するには時間があまりにもかかるので、東雲フォントというフォントを加工し、この OS でも使えるようにした。また printk という関数を作りログ用の文字を表示できるようにした。(図６)

```
printk: 3
printk: 4
printk: 5
printk: 6
printk: 7
printk: 8
printk: 9
printk: 10
printk: 11
printk: 12
printk: 13
printk: 14
printk: 15
printk: 16
printk: 17
printk: 18
printk: 19
printk: 20
printk: 21
printk: 22
printk: 23
printk: 24
printk: 25
printk: 26
```

図６　printk 関数を使用した出力例

(６)マウス入力

マウスの表示もまずは文字の表示と同じように専用の２次元配列を作りそれをピクセルを描画する関数で表示する。(図７)

```
const char mouse_cursor_shape[kMouseCursorHeight][kMouseCursorWidth + 1] = {
  "@           ",
  "@@          ",
  "@.@         ",
  "@..@        ",
  "@...@       ",
  "@....@      ",
  "@.....@     ",
  "@......@    ",
  "@.......@   ",
  "@........@  ",
  "@.........@ ",
  "@..........@",
  "@......@@@@@",
  "@.....@     ",
  "@....@@.@   ",
  "@...@ @.@   ",
  "@..@  @.@   ",
  "@.@   @.@   ",
  "@@    @.@   ",
  "@     @.@   ",
  "      @.@   ",
  "      @@@   ",
};
// #@@range_end(mouse_cursor_shape)
```

図７　マウスカーソルの配列の様子

マウスカーソルを動作させるプログラムは、マウスカーソルを初期の位置に描画する。

MouseCursor クラスのコンストラクタで行う。（図８）マウスカーソルの移動では MouseCursor クラス内の MoveRelative 関数で行う。（図９）では実際にマウスカーソルを表示している

```
mousecursor class
class MouseCursor {
 public:
  MouseCursor(PixelWriter*   writer,   PixelColor
erase_color,
              Vector2D<int> initial_position);
  void MoveRelative(Vector2D<int> displacement);

 private:
  PixelWriter* pixel_writer_ = nullptr;
  PixelColor erase_color_;
  Vector2D<int> position_;
};
```

図８　MouseCursor クラスの内部

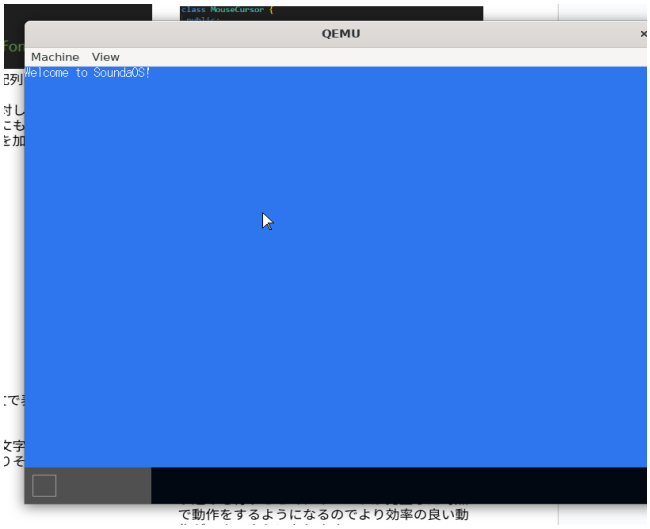

図９　マウスカーソルの表示

（７）割り込み

現在のマウスの動作は、ポーリング方式を採用している。しかしポーリング方式では、マウスの動作が固まることがある。しかし割り込みを行うことで、イベントが発生した時点で動作をするようになるので、より効率が良くなる。

（８）メモリ管理

OS が動作するに従い、メモリが必要になったり、不要になったりすることがある。しかし現在はメモリの確保、解放の機能がない。

そのため今回はページング処理を行いメモリの管理を行う。（図 10）

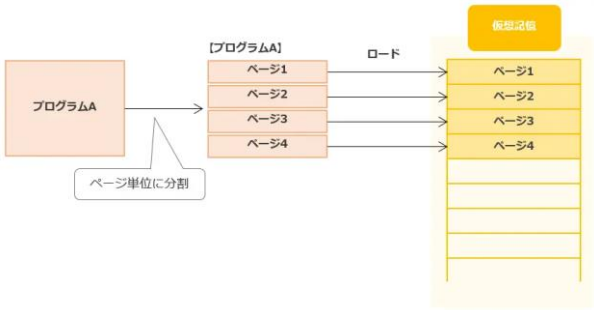

図 10　ページング処理のイメージ

（９）重ね合わせ処理

現在ではレイヤーの処理ができていない。

そのためマウスカーソルを文字の上などに動かすと文字が背景色で上書きされてしまう。その動作の回避のために重ね合わせ処理を制作する。（図 11）

図 11　レイヤーのイメージ

(10)ウィンドウ

ウィンドウは前回制作した重ね合わせ処理を応用することで作ることができる。イメージとしては、ウィンドウ用のレイヤーをマウスカーソルとデスクトップのレイヤーの間に表示する。(図 12)

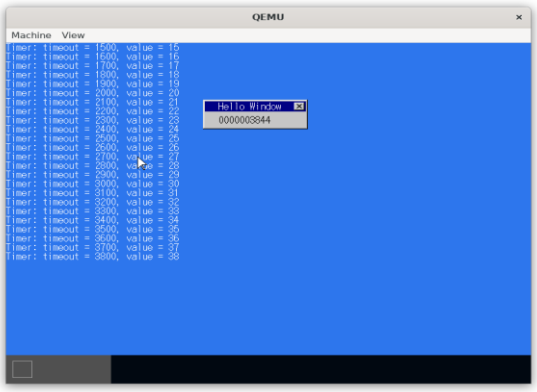

図 12　ウィンドウを表示している様子

(11)キー入力

キー押下時にイベントを処理するハンドラを作成し、押されたことを検出する。(図 13)

```
initilize keyboard
void  InitializeKeyboard(std::deque<Message>&
msg_queue) {
 usb::HIDKeyboardDriver::default_observer =
  [&msg_queue](uint8_t keycode) {
    Message msg{Message::kKeyPush};
    msg.arg.keyboard.modifier = modifier;
    msg.arg.keyboard.keycode = keycode;
    msg.arg.keyboard.ascii = ascii;
    msg_queue.push_back(msg);
  };
}
```

図 13　キー入力用ハンドラ

またより OS に近づけるために、テキストボックスウィンドウを表示し、キーボードを押下すると対応した値が入力されるようにし、またカーソルが出てくるようにする。(図 14)

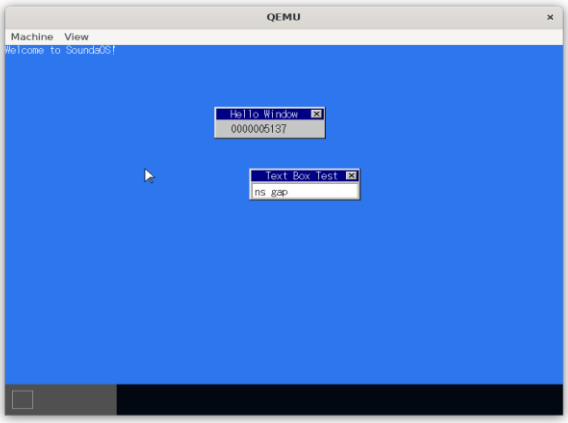

図 14　文字入力の様子

(12)コマンド

ターミナル上でコマンドを実行するために、エンターキーが押されると ExecuteLine 関数が実行される。次の NULL 文字までをコマンドと認識し、入力されたコマンドを実行する。(図 15)

```
void Terminal::ExecuteLine() {
  char* command = &linebuf_[0];
  char* first_arg = strchr(&linebuf_[0], '
');
  if (first_arg) {
    *first_arg = 0;
    ++first_arg;
  }

  if (strcmp(command, "echo") == 0) {
    if (first_arg) {
      Print(first_arg);
    }
    Print("¥n");
  } else if (command[0] != 0) {
    Print("no such command: ");
    Print(command);
    Print("¥n");
  }
}
```

図 15　ExecuteLine 関数

(13)ファイルシステム

まずファイルとは一般的に任意のバイト列に名前を名付けたものだ。そのファイルの操作/アクセス/検索のためのシステムをファイルシステムと呼ぶ。今回は試しやすさや、規格が公開されていて実装しやすいので FAT を扱う。

また確認用のコマンド「ls」を実装する。そのためのファイル名を取得するプログラムを(図 16)に示す。

```
void ReadName(const DirectoryEntry& entry, char* base, char* ext) {
  memcpy(base, &entry.name[0], 8);
  base[8] = 0;
  for (int i = 7; i >= 0 && base[i] == 0x20; --i) {
    base[i] = 0;
  }

  memcpy(ext, &entry.name[8], 3);
  ext[3] = 0;
  for (int i = 2; i >= 0 && ext[i] == 0x20; --i) {
    ext[i] = 0;
  }
}
```

図 16　ReadName 関数

(14)アプリケーション

今回は、ファイルシステムを使用し、OS 本体とは別のファイルを作成し、OS 側からアプリケーションを読み取り実行できるようにする。そのため ExecuteLine 関数とは別に ExecuteFile 関数を追加する。(図 17)

```
VoidTerminal::ExecuteFile(const fat::DirectoryEntry& file_entry) {
  auto cluster = file_entry.FirstCluster();
  auto remain_bytes = file_entry.file_size;

  std::vector<uint8_t> file_buf(remain_bytes);
  auto p = &file_buf[0];

  while (cluster != 0 && cluster != fat::kEndOfClusterchain) {
    const auto copy_bytes = fat::bytes_per_cluster < remain_bytes ?
      fat::bytes_per_cluster : remain_bytes;
    memcpy(p, fat::GetSectorByCluster<uint8_t>(cluster), copy_bytes);

    remain_bytes -= copy_bytes;
    p += copy_bytes;
    cluster = fat::NextCluster(cluster);
  }

  using Func = void ();
  auto f = reinterpret_cast<Func*>(&file_buf[0]);
  f();
}
```

図 17　ExecuteFile 関数

３．研究のまとめ

今回の OS 制作では、OS を作るということを通して OS の基本的な動作や、どのようなプログラムで作られているかを理解することができた。

また C 言語などのプログラミングの実習などでは、あまり使わないと思っていたポインタが、ほぼすべての関数に１つ以上使われていたことに驚いた。

またクラスを使用して、オブジェクト指向を意識して、プログラムすることの大切さが再び理解することができた。

今後はプログラムを制作するときは、伝わりやすさを考え、関数化、オブジェクト指向の意識をして伝わりやすいプログラムを書けるようにしたい。

参考文献

内田公太 ゼロからの OS 自作入門　743p.
株式会社マイナビ出版

mikanos (source code)
https://github.com/uchan-nos/mikanos

ページング方式とは-IT を分かりやすく解説
https://medium-company.com/