情報処理 (計算機) システム Ⅱ ・第 11 回

2023 年 6 月 23 日

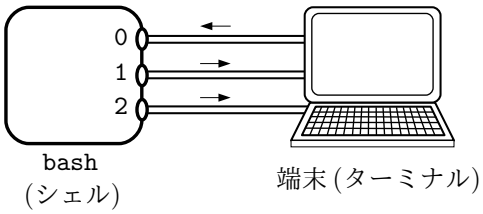
今回の内容

11.1 シェルと端末エミュレータ、ディスプレイサーバ

Unix 系 OS (たとえば Linux) の環境では、端末エミュレータ (`gnome-terminal` など) を起動すると、端末エミュレータは自動的に**シェル (shell)**と総称されるプログラム (たとえば `bash`) を起動します。端末エミュレータのウィンドウに

```
y220000@s01612h001:~$
```

のようなコマンドプロンプト (入力を促す文字列) を表示させているのは、このシェルと呼ばれるプログラムです。シェルのプロセスと端末エミュレータのプロセスとは、**疑似端末**[1]と呼ばれる通信経路でつながっています。シェルのプロセスは、ちょうど `open` を呼び出してファイルを開いた場合と同じように、この疑似端末と情報のやり取りを行う通信の口を開いた状態で起動されています。



端末の例: DEC 社製 VT220

図 1: 1980 年頃のシェルと端末 (ターミナル) の関係

1980 年くらいまでは、基本的には文字を表示することしかできないディスプレイとキーボードの組からなる、図 1 のような**端末 (ターミナル)**と呼ばれる入出力装置を計算機に接続し、それをユーザプログラムが (もちろんカーネルのデバイス管理機構を介して) 使用するのが一般的でした。現在では、GUI[2] を備えた計算機が主流となり、物理的な端末に代って、端末エミュレータと呼ばれるソフトウェアが使用されるようになっています。疑似端末は、Unix 系の OS がこのようなソフトウェアのために提供している仕組みで、シェルなどのユーザプログラムから見ると、あたかも物理的な端末に対して入出力を行っているかのように見えます。シェルが疑似端末に送った文字列は端末エミュレータが受け取り、端末エミュレータは、GUI を管理しているディスプレイサーバに対して、必要な画面の描画処理を依頼することで、文字列を画面に表示します。逆に、キーボードの

[1] Unix 系の OS に含まれる機能です。

[2] Graphical User Interface (グラフィカルユーザインタフェース)

操作をディスプレイサーバが検知すると、この情報が端末エミュレータに送られ、端末エミュレータが文字列として疑似端末に送り、それをシェルが読み取ることになります。

シェル、端末エミュレータ、ディスプレイサーバの関係は、図 2 のようなものとなります。

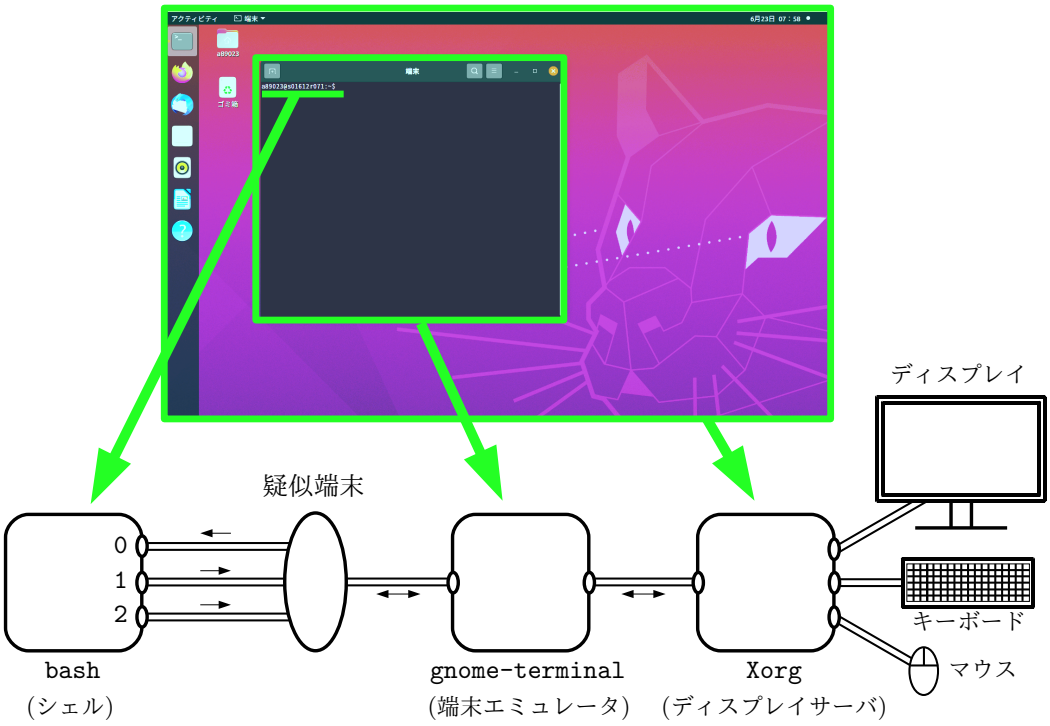


図 2: シェルと端末エミュレータ、ディスプレイサーバの関係

メモ

11.2 標準入力、標準出力、標準エラー出力

シェルが起動した時には、0 番から 2 番までの 3 つのファイル記述子が疑似端末に接続されていて、0 番は読み込み用に、1 番と 2 番は書き込み用にオープンされた状態になっています。通常、プロセスの 0 番から 2 番までのファイル記述子は常に疑似端末やファイルなどに接続された状態になっており、それぞれ次の表のような役割が与えられています。

ファイル記述子	名称	役割
0	標準入力	通常、(疑似) 端末を介してキーボード (あるいはファイルなど) に接続されていて、そのプロセスが入力データを読み込むのに使用される (`read(0, ...)` を呼び出す)
1	標準出力	通常、(疑似) 端末を解してディスプレイ画面 (あるいはファイルなど) に接続されていて、そのプロセスが出力データを書き出すのに使用される (`write(1, ...)` を呼び出す)
2	標準エラー出力	通常、(疑似) 端末を解してディスプレイ画面 (あるいはファイルなど) に接続されていて、そのプロセスがエラーメッセージを書き出すのに使用される (`write(2, ...)` を呼び出す)

C 言語の標準入出力ライブラリ関数である `scanf` や `getchar` は、そのプロセスの標準入力から、システムコール `read` により文字などの情報を読み取ります。また、`printf` や `puts` は、プロセスの標準出力に対して `write` を行います。標準エラー出力に対する出力をしたい場合は、ヘッダファイル `stdio.h` 内で宣言されている変数 `stderr` を参照して、たとえば

```
fprintf(stderr, "Error: %s というファイルが見つかりません\n", filename);
```

のように出力先を指定します。ヘッダファイル `stdio.h` には、同様に `stdin` や `stdout` という変数も宣言されていて、それぞれ、標準入力や標準出力を入出力先として指定するために使用することができます[3]。

メモ

11.3 シェルによるコマンドの実行

シェルのプロセスは、その標準入力 (0 番のファイル記述子) からコマンドを読み込み、読み込んだコマンドを実行していきます。このコマンドには、内部コマンドと外部コマンドの 2 種類があります。内部コマンドは、シェル自身によってシェルの内部で実行されるコマンドで、新たなプログラム (プロセス) が起動されることはありません。たとえば `bash` でよく使用される内部コマンドとして次の表のようなものがあります。表中の**環境変数**とは、Unix 系 OS のプロセスの属性の一つで、プロセスごとに環境変数名とその値 (文字列) の組を複数設定することができます。

[3] `stdin`、`stdout`、`stderr` は、`fscanf`、`fprintf`、`getc`、`fputc`、`fgets`、`fputs`、`fread`、`fwrite`、`fflush` などの標準入出力ライブラリ関数において、それぞれ、標準入力、標準出力、標準エラー出力を入出力先として指定するものですが、これら自身はファイル記述子 (0 以上の整数) を表すものではありませんので注意が必要です。

内部コマンド	働き
cd［パス名］	シェルのカレントディレクトリを変更する (［ ］部分を省略するとカレントディレクトリをユーザのホームディレクトリに変更する)
exit［終了コード］	シェルを終了する (［ ］部分を省略すると最後に実行したコマンドの終了コードがシェル自身の終了コードとなる)
変数名=値	シェルの内部変数の値を設定する
export［変数名］	シェルの内部変数とその値を、外部コマンドを実行する際の環境変数として使用するように指定する (［ ］部分を省略すると現在の設定が表示される)
history［ 個数 ］	これまでに実行したコマンドの履歴を表示する (［ ］部分を省略すると保存されているすべての履歴を表示する)
help［コマンド名］	シェルの内部コマンドの使用法を表示する (［ ］部分を省略すると内部コマンドの一覧を表示する)

メモ

外部コマンドの実行　内部コマンド以外のコマンドは、外部コマンドとして実行されます。Unix 系 OS でよく使用される ls、pwd、cp、mv、rm、mkdir、rmdir、cat などのコマンドは、すべてこのような仕組みで実行される外部コマンドです[4]。たとえば、端末エミュレータのウィンドウで、キーボードから

```
y220000@s01612h001:~$ cc -o myprog myprog.c
```

のように入力してエンターキーを押すと、シェルは、標準入力から「cc -o myprog myprog.c」という文字列 (と改行文字) を読み取りますが、cc という内部コマンドは存在しないため、シェルは cc という名前のプログラムファイルを**コマンドサーチパス**と呼ばれるディレクトリ群内で探索します。コマンドサーチパスは PATH という環境変数に、探索対象のディレクトリのパス名を、探索順に「:」で区切って並べた文字列として保持されています。

シェルは PATH に登録されている各ディレクトリを順に調べて、cc という名前の (実行可能な) プログラムファイルを探します。たとえば、PATH に登録されていた /usr/bin というディレクトリで cc というプログラムファイルを見つけた場合、シェルは、/usr/bin/cc というプログラムファイルを起動します。

[4] pwd など、使用頻度の高い一部の外部コマンドは、シェルの内部コマンドとしても用意されていることがあります。

また、シェルに対して

```
y220000@s01612h001:~$ ./myprog
```

のように / を含むパス名で指定して、プログラムファイルを直接実行することもできます。この場合はコマンドサーチパスは使われません。

シェルがプログラムファイル (外部コマンド) を起動する際には (特にユーザが指定しない限り) シェルが open しているファイル記述子を起動したプロセスに引き継ぎます。このため、新しいプロセスは、図 3 のように、シェルが使用していた疑似端末と接続された状態で実行が開始されることになり、疑似端末を介してキーボードからの入力を受け取ったり、ディスプレイに文字を出力したりすることができます。

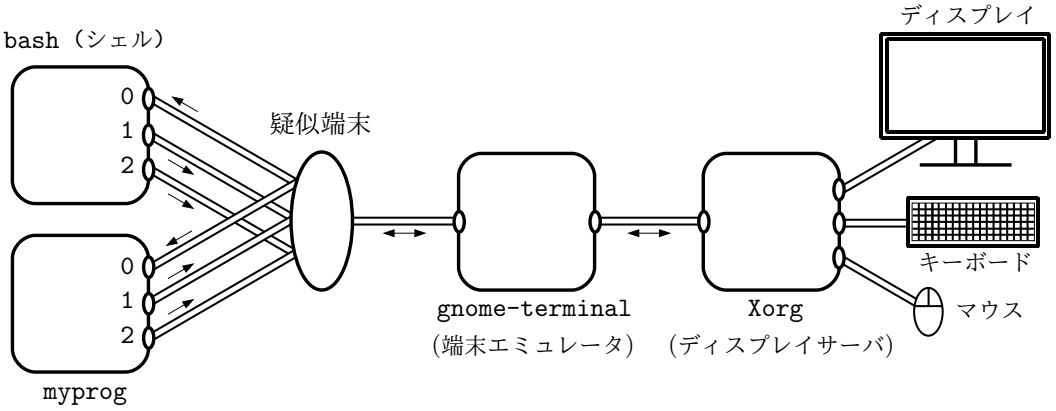


図 3: myprog とシェル、端末エミュレータ、ディスプレイサーバの関係

シェルはプログラムファイル (外部コマンド) を起動すると、起動したプロセスが終了するまでイベント待ち状態になります。起動したプロセスが終了すると、シェルは、疑似端末にプロンプトの文字列を送って、ユーザが次のコマンドを入力するのを待ちます。

メモ

コマンドサーチパス設定上の注意　ユーザが自分で書いた C プログラム myprog.c を

```
y220000@s01612h001:~$ cc -o myprog myprog.c
```

のようにコンパイルすると、カレントディレクトリに myprog という名前のプログラムファイルが作成されますが、この myprog を

```
y220000@s01612h001:~$ myprog
```

のようにして実行することはできません。これは、セキュリティ上の理由から、PATH には、通常、相対パスを登録しないからです。カレントディレクトリを表す . を PATH に登録しておけば、単に myprog と入力するだけで、シェルは ./myprog を実行してくれるはずですが、もし、そうしておくと、たとえば、あるディレクトリ /some/where に悪意のあるプログラムファイル[5]が ls という名前で置かれていて、たまたま、そのディレクトリに cd した後、/usr/bin/ls を実行するつもりで

```
y220000@s01612h001:/some/where$ ls
```

と入力してエンターキーを押すと、その悪意のあるプログラムが実行されてしまうことになります。このため、PATH には信頼のおけるディレクトリの絶対パス名のみを登録する慣習になっています。

情報処理 (計算機) システム Ⅱ ・第 11 回・終わり

[5]たとえば、実行したユーザのすべてのファイルを削除してしまうようなプログラムです。