コンピュータリテラシ＃6 – コンピュータシステムとOS

久野 靖 (電気通信大学)

2017.5.19

1　今回の目標

今回の目標は次の通りです。

- コンピュータの一般的な構造について理解する — コンピュータの組み立てや設置を必要とする研究もありますので予備知識として。
- オペレーティングシステム (OS) とその役割について理解する — ソフトウェアの基本構造を知ることは計測実験などで必要になります。
- Unix を題材にプロセスやコマンドインタプリタなどの概念を理解する — プロセスについて知ることはソフト的な実験ではまず基本的な前提となります。

2　コンピュータシステムの構造

2.1　ハードウェアの一般的な構成

コンピュータの動作はすべてプログラムによって定められていますが、プログラムが動いて実際に仕事をするためには、CPU をはじめとする物理的な装置が必要です。この、目にみえるコンピュータの装置のことを**ハードウェア** (hardware) と呼びます。ハードウェアの中核部分はプログラムを動作させる **CPU** とプログラムやデータを保持する**メモリ**ですが、このほかに**入出力装置** (I/O devices) が必要です。その役割は、ユーザや他のコンピュータとやりとりしたり、長期的なデータ保持などの機能を提供することです。主要な入出力装置を挙げておきます。

- キーボード、マウスなど — 人間がシステムに指示やデータを渡す**入力装置** (input devices)
- ディスプレイ、プリンタなど — システムが人間に情報を提示する**出力装置** (output devices)
- ディスク、フラッシュメモリ — 長期的にデータを保持する **2 次記憶装置** (secondary storage)
- ネットワークインタフェース — 他のシステムとデータをやりとりする**インタフェース** (interface)

これらの装置の配置のされ方は、コンピュータの種類や用途によってさまざまですが、普段目にする PC などを例に描いたものが図 1 です。

中核となるのが VLSI 技術で作られた CPU です。今日の CPU の多くは、内部に**コア** (core) と呼ばれる独立した CPU の機能を果たす部分が複数入っていて、これらが並列に動作することで性能を向上させています。これを**マルチコア** (multi-core) と呼びます。そしてメモリ (これも VLSI 技術で作られます) は、CPU と密接にやりとりする必要があるので、専用の配線で CPU とつながっています。

コンピュータのケースを開けると、中には**マザーボード** (mother board) と呼ばれる基板が入っていて、その上には CPU・メモリとバスが搭載されています。バスはただの配線なのですが、その上にたくさん端子のついたソケットが数個くっついています。そして、CPU とメモリ以外の要素 (つまり入出力のための回路) はこのソケットに基盤を差し込むことで接続します。こうしておけば、このソケットを抜き差しするだけで、さまざまな入出力装置を増設したり外したりできるわけです。

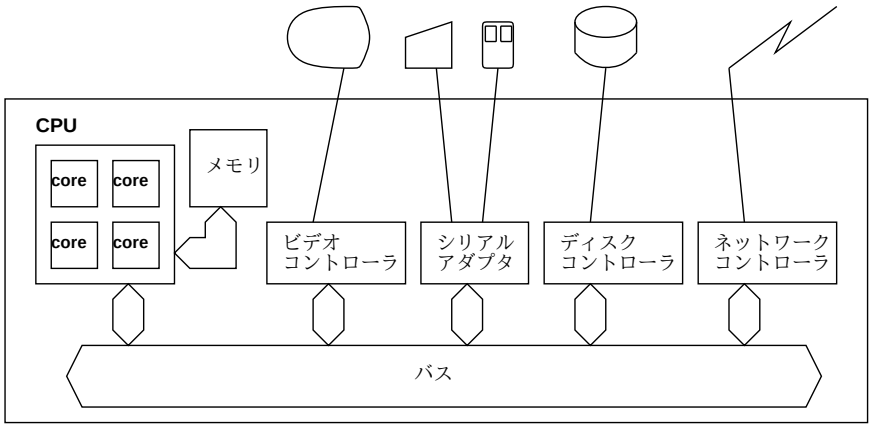


図 1: PC などのハードウェアの構成

この、抜き差しする基盤の反対側には、また別のさまざまな形のソケットがついていて、ここと実際の入出力装置をケーブルやコネクタで接続します。基盤の上に載っている回路はコントローラ、アダプタなどと呼ばれ、バス上の信号と各入出力装置の間の橋渡しを行う機能を持ちます。このようにすることで、CPU からはどの入出力装置でもバス上の同じ信号で制御でき、それぞれの装置に固有の信号の送受はコントローラにまかせることができるのです。

PC などのシステムにはおおむね、ビデオコントローラ (画面)、シリアルアダプタ (キーボード、マウス)、ディスクコントローラ、ネットワークコントローラなどが備わっています

2.2 ソフトウェアの一般的な構成

ハードウェアは以上だとして、それを動かすソフトウェアはどのような構造になっているのでしょうか? あなたがたとえばブラウザで Web を見ているとき、コンピュータ上で動いているのはブラウザだけでしょうか? ブラウザを動かした状態でさらに別のプログラムを動かしたり、ブラウザの窓の位置を変更したりする時は、「ブラウザではない何か」を使って操作をしていますね?

この「何か」の部分は、使っているソフトが違っても同じように使え、さまざまなソフトを使う手助けとなってくれます。そしてその手助けも、プログラムを動かす、窓を操作する、など沢山の方面に渡っています。つまり一般に、ソフトウェアには次の 2 種類があるわけです。

- **アプリケーションソフト** (application software) — ユーザが各々の仕事を実行するためのソフト。
- **システムソフト** (system software) — アプリケーションを使って仕事をする上で必要な手助けや土台となったり、コンピュータを使う上で必要となる仕事を行うソフト。基本ソフトとも呼ぶ。

具体的には、どのような「手助け」が必要でしょうか? それはこれから見ていきますが、その前にシステムソフトを次のように分類しておきます (この分類は人により違うかも知れません)。

- **オペレーティングシステム** (operating systems、OS) — アプリケーションが動く下ざさえとなる機能を提供するひとまとまりのソフトウェア。
- **ミドルウェア** (middleware) — データベース管理システム、Web サーバなど、(OS と同様の意味で) アプリケーションが動く基盤となるが、OS とは独立に開発・提供されているもの。
- **言語処理系** (language processors) — プログラムを記述して動かすためのソフトウェア。
- **ユティリティ** (utilities) — ファイルの操作やデータ形式の変換など、(特定目的に特化した「アプリケーションソフト」とは対照的に) 汎用的な作業を手助けする。

今回はおもに Unix を題材として、まず OS について見ていきます。

2.3 OS とその働き exam

上で OS の役割りは「アプリケーションが動く下ささえ」と書きましたが、それは具体的にはどういうことでしょうか? もう少し具体的に考えてみましょう。

コンピュータのハードウェア命令は、メモリとレジスタの間のデータ転送や四則演算など、ごく基本的な機能しか提供しません。ですから、多くのプログラムで必要とするような、画面の表示、文字の入力などの機能の実現には、かなり長いコードが必要です。

それを各アプリケーションを作る人が毎回用意するのは大変ですし、個々のアプリケーションが勝手に入出力機器を制御しはじめると、入力の混乱や画面の破壊など、様々な問題が起きそうです。ですから、以下は OS の重要な役割りだと言えます (図 2)。

- 多くのプログラムが必要とする標準的機能を一括して提供する

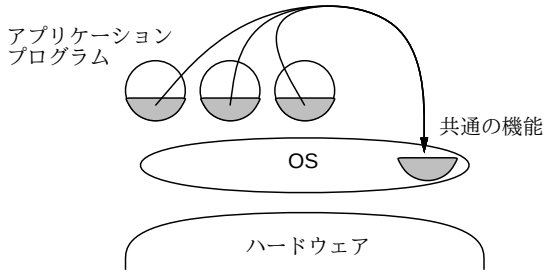


図 2: OS による標準的機能の提供

次に、現在のコンピュータシステムでは、ある窓で計算をさせながら、別の窓では待ち時間に Web を見るなど、複数プログラムを並行して動かす**マルチタスク** (multitask) 機能が使われます (図 3)。

- 複数のプログラムが並行して動作するのを管理する

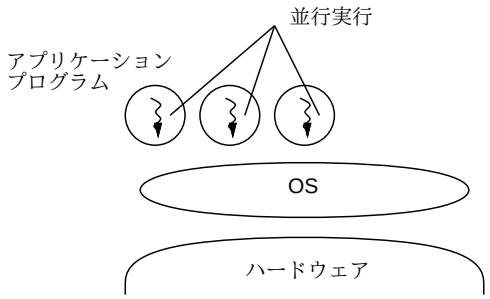


図 3: OS によるマルチタスク機能の提供

とすると、あるプログラムを動かすときに、そのつど CPU を止めてプログラムをメモリに書き込むわけには行きません (それでは現在進行中の別の仕事も止まってしまいます)。ですから、以下も OS の基本的な役割りです (図 4)。

- ユーザが指定したプログラムを読み込んで実行開始させる

さて、そうして並行動作している複数のプログラムが同じメモリ番地やディスク上の領域を使おうとしたら、やはり混乱が起きます。ですから、以下のことも OS の重要な仕事です。

- コンピュータのメモリを各プログラムにうまく割り当てて調整する
- 入出力装置に対するアクセス (読み/書き) を管理する

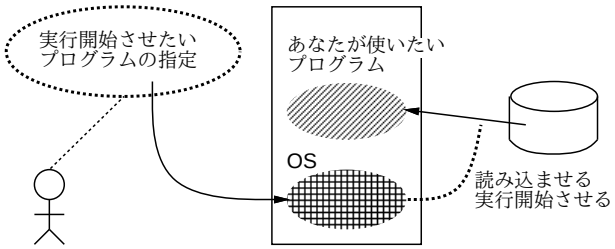


図 4: OS によるプログラムの実行開始

見かたを変えると、コンピュータに備わった入出力装置、メモリ、CPU などはすべて、数が限られた貴重な**資源** (resource) だと言えます。そこで、OS の働きは次のようにまとめることができます。

- コンピュータ内部の各種資源を管理する

つまり、動作中のコンピュータでは常に OS が動き、ハードウェアと一体となりすべてを管理しています。そして、あなたやあなたのプログラムがコンピュータを使おうとする時、必要な資源はすべて OS に頼んで使わせてもらい、それによって仕事ができるのです (図 5)。

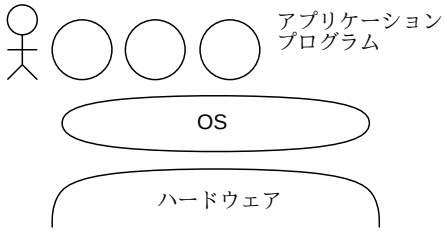


図 5: OS とユーザ、アプリケーションの関係

3 プロセスとその観察・操作

3.1 マルチタスクとプロセス exam

まず OS で一番目だつ機能であるマルチタスク、つまり複数のプログラムを並行して走らせる機能について見てみます。どのようにしてそんなことが可能なのでしょう?

コンピュータのメモリ上には命令の列 (プログラム) が置かれていて、CPU はプログラムカウンタ (PC) が指している番地から順に命令を取り出しては実行して行きます (図 6)。

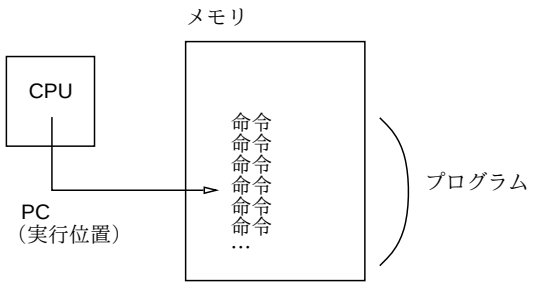


図 6: CPU による命令の実行

そこで、複数のプログラムを並行して動かすには、それらをメモリに一緒に入れます。そして、CPU についている**タイマー** (timer) と呼ばれる機能を動かした状態で、まずプログラム A の実行を始めます。タイマーは一定時間たつと CPU に信号を送ります。CPU はタイマーから信号を受け取ると、プログラム A の実行を一時中断してプログラム B の実行に切り替わります。またしばらくすると実行

はプログラム C に、そして次は A に戻ります。このようにすると、複数のプログラムが「小刻みに切り替わりながら」実行されますが、CPU は非常に高速なので、ユーザにとってはすべてのプログラムが同時に動いているように見えます。

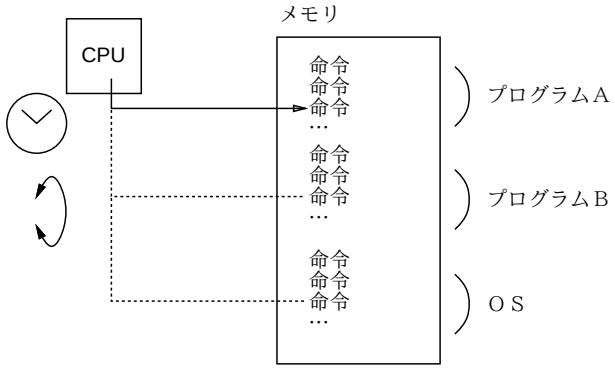


図 7: マルチタスク機能の実現

正確には、A〜C のうち 1 つは OS で、タイマー信号が来ると CPU は常に OS の実行に切り替わります。OS は各プログラムの使用時間や優先順位を調べ、次に実行するプログラムを選択し、実行開始させます。このため、OS は各プログラムへの CPU 割り当てを自由に制御できるのです。

一般に、この「動いている状態のプログラム」のことを**プロセス** (process) と呼びます。現在は CPU を複数搭載したシステム [1] も多く使われていますが。そのようなシステムでも、動かしたいプログラム数 (プロセス数) は CPU 数より多いのが普通ですから、上で述べた「小刻みな切り替わり」があることは同じです。こうして、CPU は 1 個、メモリは 1 枚しかなくて、その上で直接プログラムを走らせるなら 1 個だけしか走らせられなくても、OS がその上にプロセスという「プログラムが走るいれもの」を作り出すことで、多数のプログラムを並行して走らせられるのです。

一般に「実際にはないけれどソフトウェア (OS など) の働きによって役に立つものを作り出す」ことを**仮想化** (virtualizaiton) と言い、コンピュータシステムでは多く使われる考え方です。そして、プロセスは OS の働きによって作り出れた「仮想化された CPU とメモリ」なのです。[2]

3.2 ps — Unix でのプロセス観察 exam

Unix では、**ps**(process status) コマンドによって、現在動作中のプロセスを観察できます (指定するパラメタによって、表示するプロセスの範囲や詳しさを制御できます)。

Unix システムの系統ごとに ps のパラメタ指定は違っています。ここでは sol の OS である GNU/Linux の場合について、基本的なもののみを説明します。なお以下で「端末」というのはコマンドを打ち込んでいる「窓」に相当するものと思ってください。

- `ps` — 簡潔な表示。使っている端末から動かしたプロセスのみ
- `ps -f` — 各プロセスについてのより詳しい表示
- `ps -A` — 他人のものも含めたすべてのプロセス表示
- `ps -u` ユーザ名 — 指定ユーザの全プロセス表示

たとえば sol で `ps` を実行した結果を次に示します。

```
...$ ps
   PID TTY          TIME CMD
 49299 pts/77   00:00:00 tcsh
 84487 pts/77   00:00:00 ps
```

[1] 1 つの CPU 中に複数のコアが入っているマルチコアのシステムもありますし、さらに CPU チップを複数搭載した**マルチプロセサ** (multiprocessor) と呼ばれるシステムもあります。

[2] OS のうちでプロセスを作り出す機能の部分を**プロセス管理**と呼びます。

PID というのは**プロセス ID** で、システム内の各プロセスにつけられた固有番号です。そして TTY が端末番号で、現在自分は pts/77 という端末から使っていることが分かります。TIME というのはそのプロセスがどれくらい CPU 時間を使用したかの累計、CMD はコマンド名です。

これを見ると、現在の端末からは 2 つのプロセスを動かしていると分かります。1 つは ps のプロセスですが、これは「現在自分は ps のプログラムを動かして自分のプロセスを調べているので、その調べるという作業をしているプログラムである ps も当然調べた中に入っている」ということです。[3]

もう 1 つの tcsh は**コマンドインタプリタ** (command interpreter) といい、ユーザからコマンドを受け付け、対応するプログラムを動かす働きのプログラムです (「コマンドを解釈してくれる」からこのような名前がついている)。コンピュータが行う全ての動作はプログラムによって実行されるので、コマンドを打ち込んだらそのコマンドが動作する、ということ自体もそれをおこなうプログラムがあるから起きてくれる、というわけです。なお、Unix ではコマンドインタプリタのことを「貝殻のようにユーザを護ってくれている」というイメージから**シェル** (shell) とも呼びます。

プロセスが 2 つしか見えないのではつまらないので、全部のプロセスを見てみましょうか。sol の上で合計いくつくらいプロセスがあると思いますか?

```
...$ ps -A
   PID TTY          TIME CMD
     1 ?        00:01:42 init
     2 ?        00:00:04 kthreadd
     3 ?        00:14:24 migration/0
(途中略…)
130173 ?        00:00:00 httpd
130175 pts/126  00:00:00 a.out
131069 ?        00:00:00 Web Content
```

この例をやってみた時は 2725 個のプロセスが表示されました。[4]sol は多数のユーザが共同で使い、またさまざまなサービスも提供しているシステムなので、常時このように多くのプロセスが動作しているのです。なお、TTY が?のプロセスは端末から切り離されて単独で動き続けているもので、その中には CPU 使用時間が長いものも含まれています。

しかし 2 千以上では見ても分かりませんから、今度は自分が動かしているプロセスを見ることにします (皆様も自分の ID を指定してください)。

```
...$ ps -u ka002689
   PID TTY          TIME CMD
 49298 ?        00:00:00 sshd
 49299 pts/77   00:00:00 tcsh
 67891 ?        00:00:00 sshd
 67896 pts/162  00:00:00 tcsh
 68171 pts/77   00:00:00 ps
```

このときはもう 1 つ別のマシンから sol に入っていたので、pts/162 の tcsh が増えています。あと、sshd というのが 2 つありますがこれは何でしょうか? 外部からネットワーク経由で (より正確には SSH プロトコルで)sol に接続すると、その接続は sshd というプログラムにつながり、そこで認証処理が行われて OK だと、新しい端末を 1 つ割り当てて tcsh を起動します。この処理をやっているのが sshd のプロセスです。このプロセスは使用を終わるまでずっと待っていて、終ったら接続を切ります。ためしに、pts/162 の側を exit してから再度見てみます。

[3]鏡に向かって写真を撮ったら撮っている自分も写るようなものだと思ってください。
[4]実際は画面で数えられるわけはないので、ファイルに保存して数えましたが、そういう話題はまた後で。

```
...$ ps -u ka002689
   PID TTY          TIME CMD
 49298 ?        00:00:00 sshd
 49299 pts/77   00:00:00 tcsh
 93685 pts/77   00:00:00 ps
```

確かに tcsh と sshd が 1 つずつ減りました。ところで sshd は TTY が?ですが、これは sshd がログイン処理の後に端末を割り当てるので sshd 自体は端末の中で動作していないことによります。では最後に、より詳しい表示を見るため、オプションに-f を追加します。

```
...$ ps -f -u ka002689
UID         PID   PPID  C STIME TTY          TIME CMD
ka002689  49298  49213  0 12:33 ?       00:00:00 sshd: ka002689@pts/77
ka002689  49299  49298  0 12:33 pts/77  00:00:00 -tcsh
ka002689 103662  49299 17 14:55 pts/77  00:00:00 ps -f -u ka002689
```

いくつかフィールドが増えています。まず PPID は「Parent PID」つまりそのプロセスを作り出したプロセスの ID になります。よく見ると、sshd が tcsh を作り出し、そして tcsh が ps を作り出していますね。C は CPU 使用率で、そのプロセスが実行している全時間のうち CPU を使っていた割合 (%) です。ps は少し CPU を使っていますが、あとはほとんど使っていないことが分かります。また CMD のところはコマンド名だけでなくオプションまで表示されています。

このように、ps を使うことで現在のプロセス群の動作状態をかなり詳しく調べることができます。ここで説明した以外のオプションについては、man コマンドなどで調べることができます。

3.3 シェルによるプロセスの生成 exam

ここまでプロセスの観察について見て来ましたが、プロセスを作るのにはどうすればいいのでしょう? まず外部から sol にログインするとそれに対応するシェル tcsh のプロセスができることが分かりました。次に、シェルに対してコマンド「ps」や「emacs &」などを打ち込むと、そのコマンドを実行するためのプロセスがそのつど生成されます (図 8)。

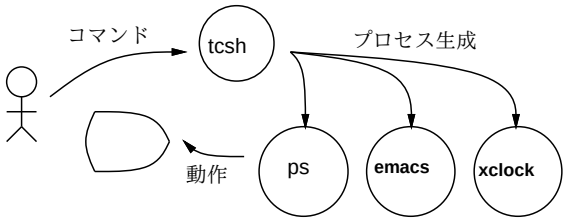


図 8: コマンドインタプリタ

毎回 ps を実行するごとに、PID が異ることに気づきましたか? つまり、ps のプロセスは 1 回表示を行うだけで終わってしまい、必要のつど新たに作られています。一方、tcsh は同じままです。この様子を図 9 に示します。つまり、tcsh はずっと動いていますが、コマンドの方は利用者がコマンドを打つたびにそれを実行する新しいプロセスが tcsh によって作られるのです。[5]

図の点線は、tcsh が子プロセスの完了を待つ箇所です。通常は一つコマンドを打ったらその完了を待って次のコマンドを打つちますね。ただし時間が掛かったり別の窓として動きはじめるコマンドの場合には、待ちたくないですね。そのときはコマンドの最後に「&」をつけて「待たずにすぐ次のコマンドを打ちたい」と指示します。つまり、&をつけると新しいプロセスができるのではなく、常

[5] tcsh が終るのは、利用者が exit という特別なコマンドを打ち込んだ時だけです。ということは、exit というのは他のコマンドのように新しいプロセスとして実行されるのではなく、tcsh 自身によって実行されることになります。このような「特別な」コマンドがいくつか存在します。

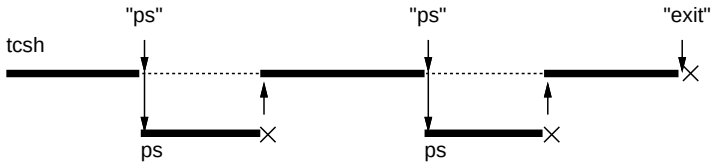


図 9: tcsh によるコマンドの発行と待ち合わせ

に新しいプロセスはできて、ただ&をつけないとそのプロセスの完了を待つので、次のコマンドを打つときには先のプロセスは消滅しているわけです。

3.4 kill によるプロセスの操作 exam

プロセスは生成されたあとは仕事が終れば自分で終了しますし、ブラウザなど対話的プログラムであれば終了メニューなどによって終了します。しかし時には、プロセスがおかしくなって外部から操作する必要が生じることもあります。そのときは **kill** コマンドを使って、プロセスに次のような「信号」を送ります (PID は `ps` で調べたものを指定します)。

- `kill -STOP PID` — プロセスの実行を一時凍結する
- `kill -CONT PID` — 凍結したプロセスの実行を再開する
- `kill -TERM PID` — プロセスに「終わってほしい」と信号する
- `kill -KILL PID` — プロセスを強制終了させる

なお、コマンド実行中に`^C` を押すとそのコマンドを実行しているプロセスに`-TERM` 相当のシグナルが、`^Z` を押すと`-STOP` 相当のシグナルが、それぞれ送られます。[6]

演習 1 プロセスの生成や操作について、次の内容から 1 つ以上 (できれば全部) 試してみなさい。

a. 「`xclock -update 1 &`」により秒針つきの時計を画面に表示し、その PID を調べなさい。次に、`kill` を使ってこのプロセスを操作してみなさい。操作に先立ち、調べるべきことがらを複数考えてから試すこと (たとえば、凍結してから再開した場合に時計は「遅れたまま」になるかどうかなど)。

b. Emacs エディタを動かし、同様に操作してみなさい。操作に先立ち、調べるべきことがらを複数考えてから試すこと (たとえば、凍結した状態で Emacs の窓の上に別の窓を重ねて隠すと、上の窓をどけたときにその部分の表示は見えるかとか、凍結した状態で文字を打ち込むとその文字は入るかとか、その後解凍した時どうなるかなど)。

c. プロセスの親子関係について (PID と PPID の関係を見て) 確認したあと、「プログラムを起動すると、そのプロセスの PPID は起動したシェルになる。そのプログラムを動かしたままログアウトすると、起動したシェルが消滅するが、動かしたままのプログラムの PPID はどうなるか」を調べる実験をおこないなさい。動かしたままのプログラムとしては「`sleep 秒数 &`」を推奨するが、そのほかのものを使ってもよい。実験に先立ち、どうなるかについての想定とその理由を書き留めておき、結果と照合すること。

4 シェルの制御機能

4.1 リダイレクションとパイプ exam

リダイレクション (redirection) とは、データの流れを切り替える機能です。Unix では各プログラムは、**標準入力** (stdin)/**標準出力** (stdout)/**標準エラー出力** (stderr) という 3 つのチャネル (データの

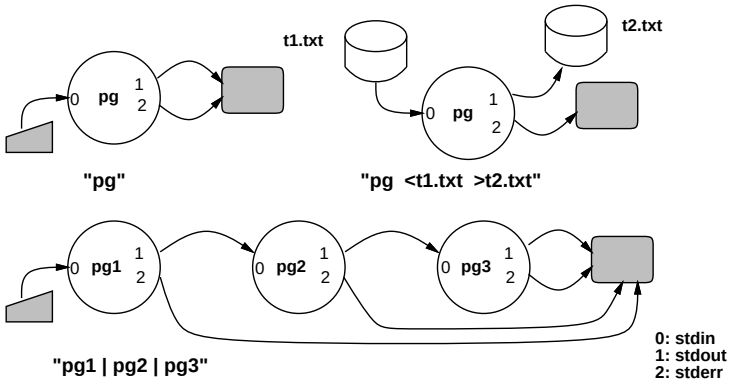


図 10: リダイレクションの考え方

通り口) を使ってデータを読み書きするのが標準になっています。通常状態では、標準入力はキーボード入力、標準出力と標準エラー出力は画面につながります (図 10 左)。

しかし先の例のように、出力をファイルに保存したいとき、あるいは入力があらかじめ用意されているのでファイルから読ませたいときは次のように指定することでおこなえます。

コマンド <入力ファイル
コマンド >出力ファイル
コマンド >>出力ファイル

3 番目は「出力の追加」で、これを使うと既にあるファイルの末尾に出力内容を追加できます。そして図 10 右のように、入力と出力のリダイレクションは併用もできます。

ところで、stderrって何でしょう? 実は各プログラムは通常の出力は stdout に送りますが、エラーメッセージだけは stderr に送ります。なぜかというと、たとえば長い時間掛かる処理をファイルに保存するようにリダイレクションを使った時、何か問題が起きてプログラムがそれを教えてあげようとメッセージを出してもファイルに入ってしまったらユーザがメッセージを見るのは翌朝だったりして不便だからです。そこで、stdout をリダイレクションしても stderr はもとのまま画面につながっていて、メッセージがすぐ見られるようにしているわけです。

ところで、上でプログラムと書いたのは実行している状態のプログラムですから、実際にはプロセスです。つまり stdin、stdout、stderr はプロセスごとに接続先を制御できます。そこでシェルでは

コマンド | コマンド | コマンド

のように「|」を使って複数のコマンドを区切ったときは、その前側のプロセスの stdout を後側のプロセスの stdin につなぐ、という制御が実行されます。これを**パイプライン** (pipeline) と呼びます (図 10 下)。stderr はこの場合も画面のままです。

ですから先程の例に戻ると、多数の行がある `ps -A` の出力をゆっくり見るためには、ファイルにリダイレクトする代わりに「`ps -A | less`」としてもよいのです。`less` は `man` コマンドで呼び出される「1 画面ずつ見るためのプログラム」であり、ファイル名を指定したらそのファイルを、指定しなければ stdin の内容を 1 画面ずつ表示してくれます。[7]

シェルの機能の中でパイプラインは代表的なものですが、ほかにもコマンドの実行され方を制御する方法が多数用意されています。ここでは主要なもの 3 つを説明します。

- コマンド `;` コマンド — 前のコマンドに続けて後のコマンドを実行する。たとえば `ps` を実行する前に日時も表示させたい場合は「`date; ps`」と 1 行に打つことができる。
- コマンド `&` — コマンドを並列に実行する (終了まで待たない)。

[6]「相当の」というのは、いちおう区別のためにシグナル番号は違えてあるけれど機能的には同じという意味です。
[7] `less` の中では `n`(次の画面)、`p`(前の画面)、`q`(終了する) などのコマンドが使えます。

- (コマンド…) — コマンドをサブシェルで実行する。サブシェルというのは、シェル自身のそっくりコピーなのでコマンドが動作する様子は同じだが、それらの実行や入出力は 1 つにまとめられて行われる。たとえば次のようにすることで、1 分間をおいて 2 回プロセスをファイルに記録できる (そしてその間待っていなくてよい)。

```
...$ (date ; ps -f ; sleep 60 ; ps -f) >rec.txt &
```

4.2 ジョブコントロール

ジョブ (job) とは、コンピュータ用語としては、プログラムやプロセスよりも大きな (それらがいくつか連なった)「ひとまとまりの仕事」という意味で使われます。

一方で Unix では、1 行のコマンド行で複数のプログラムを起動したり、それらの間で入出力のやりとりを指定するなど、かなり複雑なことができます。しかし、複数のプログラムということは複数のプロセスから成るわけなので、それらを ps で探したりして操作するのは面倒です。

そこで、コマンド行で指定したひとまとまりのプロセス全体 (ジョブ) に対して停止や再開などの処理ができる機能が作られました。これが Unix の**ジョブコントロール**機能です。ジョブコントロール的にはジョブは次の 3 つの状態を持ちます。

- **フォアグラウンド** (foreground) — 端末入出力 (キー入力/ 画面出力) ができる通常の状態
- **バックグラウンド** (background) — プロセスは実行しているが端末入出力はできない
- **中断** (suspend) — プロセスが実行を中断している (凍結されている) 状態

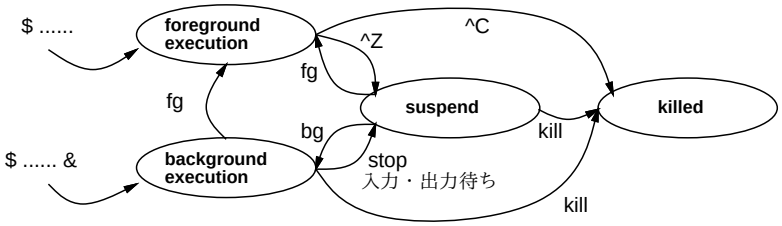


図 11: ジョブコントロールの状態遷移

これらの間の遷移を図 11 にまとめます。通常コマンド (&なし) で実行開始したジョブはフォアグラウンド状態です。この状態で終了したり Ctrl-C で止めるこもありますが、実行中に Ctrl-Z を打つと中断状態になります。次に&をつけて実行開始したジョブはバックグラウンド状態です。この状態では端末入出力はできません。このジョブも自然に終了することもありますが、止めるには「kill」を使います。また「fg」コマンドを使ってフォアグラウンドに移すことや、「stop」コマンドを使って中断にすることもできます。また、ジョブ中のどれかのプログラムが端末入出力を行おうとすると自動的に中断になります。最後に中断状態のジョブは、終了させるのには kill、フォアグラウンド実行に移すには fg、バックグラウンド実行に移すには bg を使います。

現在あるジョブを調べるには jobs というコマンドを使います。そしてその番号をもとに、ジョブを操作するコマンド fg、bg、stop、kill では「%番号」という形でジョブを指定します (kill で%を指定しない場合は先にやった PID を指定する方の機能になります。例を見てみましょう。

```
...$ emacs &  ← emacs をバックグラウンドで
[1] 33893     ←ジョブ番号と PID を表示
...$ man ps & ← man コマンドをバックグラウンドで
[2] 36457     ←ジョブ番号と PID を表示
...$          ← [RET] だけを打つ
[2]  + 中断 (tty 出力) man ps ← man は出力を試み中断
```

```
...$ xclock -update 1  ←xclockをフォアグラウンドで
(&を忘れたのでコマンドが打てない)
^Z     ←Ctrl-Zでフォアグラウンドのxclockを中断
中断
...$ jobs    ← ジョブ一覧を確認
[1]    実行中です     emacs
[2]  - 中断 (tty出力) man ps
[3]  + 中断           xclock -update 1
...$ bg %3  ←時計を動かすためバックグラウンドに
[3]  xclock -update 1 &  ←確認のためコマンド行表示
...$ fg %2  ←manの表示を見るためフォアグラウンドに
(manの画面になる。しばらく見てからqで終了)
man ps       ← さっきfgした対象のコマンド行が表示される
...$ jobs    ← ジョブ一覧を確認
[1]  + 実行中です     emacs
[3]  - 実行中です     xclock -update 1
...$ kill %3
...$         ← [RET]だけを打つ
[3]    終了           xclock -update 1
...$         ←emacsを^X^Cで終了してから[RET]を打つ
[1]    終了           emacs
...$         ←ジョブはなくなった状態
```

演習 2 ここまでに出て来た「`ps -A >test.txt`」「`ps -A | less`」「`(date ; ps -f ; sleep 60 ; ps -f) >rec.txt &`」および上のジョブコントロールの使用例をそのままやって見なさい。納得できたら、次の課題から 1 つ以上試してみなさい。

a. stdout をリダイレクトしても stderr は画面に出ることを確認できるような例を考案して実行し確認しなさい。また stderr も stdout と一緒にリダイレクトする方法を `man tcsh` で探してこちらも確かにそうなっていることを確認しなさい。

b. コマンド「`tee` ファイル」は、stdin から入って来た文字をそのまま stdout にコピーするが、ただしそのデータを指定したファイルにも記録する。「`ps -A | tee test2.txt | less`」で 1 画面ずつ表示している途中で `q` で `less` を終了したとき、表示された量とファイルに記録した量は同じか違うか。違うとしたらその差はどれくらいか、また何回か実行したとき変化するか。1 つのパイプライン内で `tee` を複数回使った場合はどうか。などの問題を検討しなさい。これをもとに、この現象の理由を推理してみなさい。[8]

c. ジョブがバックグラウンドで実行中に、端末入出力が必要になると、そこで中断するはずである。実際にそのようなことが起こる例を考案して実行し確認しなさい。できれば、そのようなことが複数回起きる (例: バックグラウンド実行→出力要求 (中断) → fg →出力→ Ctrl-Z → bg →バックグラウンド実行→入力要求 (中断)、など) 例だとより望ましい。

いずれの課題も、実際にやったことや画面表示の記録 (長い場合は途中を適宜省く) をきちんと示すこと。

最後の課題をやるときにキーボードから入力するコマンドが必要であるが、それには `cat` を使えばよい。`cat` は「`cat` ファイル…」で指定された 1 つ以上のファイルを順次 stdout に出力するが、ファイル名の代わりに「`-`」を指定するとそこでは stdin からの入力を出力する (または 1 つもファイルを

[8] 出力の量が多くないとうまく調べられない場合は、`(ps -A; ps -A; ps -A)` のようにして複数回 `ps` を動かすとよい。

指定しない場合も stdin からの入力を出力する。キーボード入力を終らせるには Ctrl-D を打てばよい。次の例を参照。

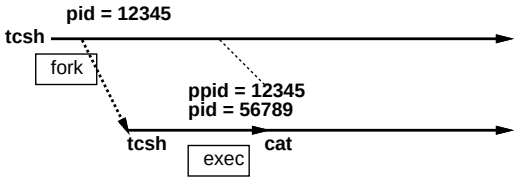
```
...$ cat >test3.txt
aaa
bbbbbb
^D      ← ctrl-D を打ったところ
...$    ← test3.txt には aaa と bbbbbb が入っている
```

演習 3 ps で PPID フィールドを表示させることで、あるプロセスがどのプロセスによって作り出されたが分かる。これを利用して、次のことを検討してみなさい。単に憶測を書くのでなく、まず仮説を立て、次にそれを確認するテスト実行を行いその結果を示して検討すること。

a. 「(sleep 10; sleep 11; sleep 12) &」のような複合コマンドがどのようにして作られているのかを検討する。

b. 「(ps -f | cat | cat | cat | cat) &」のようなパイプラインがどのようにして作られているのかを検討する。

c. シェルが一般に「;」「|」「(...)」「&」などの機能をどう実現しているか検討する。



なお、Unix 内部でプロセスを作る際には、(1)1 つのプロセスが親子の 2 つに分裂する (プログラム名は同じまま)、(2) あるプロセスが別のプログラムに「変身」する (PID は同じまま)、という 2 つの機能を組み合わせて実行しています (図)。(1) は分かりやすいですが、(2) は「なぜか tcsh であるべきプロセスが cat になっている」のような分かりにくい結果をもたらします。

本日の課題 6A

本日の課題は「演習 1」「演習 2」「演習 3」に含まれる小問 (合計で 9 個) の中から 1 つ以上を選択し、結果をレポートとして報告して頂くことです。LMS の「レポート # 6」の入力欄に直接入力してください (別途作成したものをコピーペーストで貼っても構いません)。以下の内容がこの順に含まれるようにしてください。

- 冒頭に題名「コンピュータリテラシレポート # 6」、学籍番号、氏名、提出日付を書く。(グループでやったものはグループのメンバー全員の氏名も別途書く。)
- 課題の再掲を書く (どんな課題であるかをレポートを読む人が分かる程度に要約する)。
- レポート本体の内容 (やったこととその結果) を書く。
- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。
- 以下のアンケートに対する回答。

Q1. OS の機能やプロセスについてどれくらい知っていましたか。新たに知って面白かったことは何ですか。

Q2. ps によるプロセスの観察や kill による操作などについてどのように思いましたか。

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

なお、課題はグループでやって構いません。その場合も、(メンバー氏名を明記した上で) レポートは必ず各自で執筆してください。レポート文面が同一 (コピー) と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。