OS 第 1 回課題レポート

1510151 柳 裕太

2017 年 6 月 10 日

1　課題 1: 和訳

1.1　OS のインターフェイス

オペレーティングシステム (以下、OS) の仕事は、コンピュータを複数のプログラムで共有したり、ハードウェアハードウェア単体のサポートより便利なサービスを提供することである。そのOS は低級ハードウェアを管理・取り出しを行い、それにより、例えば、ワードプロセッサが自身のディスクハードウェアが使用中かどうか考慮する気にする必要がなくなる。多重のハードウェアもまた、多くのプログラムにコンピュータを同時に共有及び実行 (あるいは実行と見せかける) することを許している。最終的に、OS は制御された相互作用する方法を提供しており、それによってこれらはデータを共有できたり、あるいは共に仕事をすることができるのである。

単一の OS はユーザにインターフェースを介してプログラム群を提供する。よいインターフェースをデザインすることは難しいことがわかる。一方に、我々はインターフェースをより簡単に正しく実行するために、シンプルで精密なものにしたがる。もう一方に、我々はより洗練された特徴をアプリケーションに提供するよう誘惑されるかもしれない。この緊張を解くトリックは、インターフェースをもっと普遍的に提供できるようにするために、ほんの少しメカニズムに依拠するデザインにすることである。

この本は単一の OS を実態のある例として、OS のコンセプトを説明するために利用する。xv6 という OS は、ケン・トンプソンとデニス・リッチーによる UNIXOS の基本的なインファーフェースを提供し、できるだけ UNIX 内部のデザインを模倣したものとなっている。UNIX はメカニズムもよく内包した限定的なインターフェースで、驚くべき汎用性を提供する。このインターフェースはにおいて成功してきており、現代の OS —BSD, Linux, MacOS, Solaris, そして更に、限定的では在るが、Microsoft Windows も—は UNIX のようなインターフェースを所持している。xv6 を理解することは、これらのシステムやその他を理解するためのよいスタート地点となるのである。

図 0-1 に記載されたとおり、xv6 は伝統的な kernel の形式をとっており、プログラムを走らせるためのサービスのような特別なプログラムがある。それぞれの走っているプログラム (プロセスと呼ぶ) は、指示/データ/スタックを内蔵したメモリを保持している。この指示はプログラムの計算が実装されている。データは計算における変数である。スタックは、プログラム処理の指示が構成されてある。

プロセスがカーネルサービスを呼び出す必要がある時、まず OS のインターフェース内にて手続きを呼び出す。この手続きのことを”システムコール”と呼ぶ。このシステム指示がカーネル内部

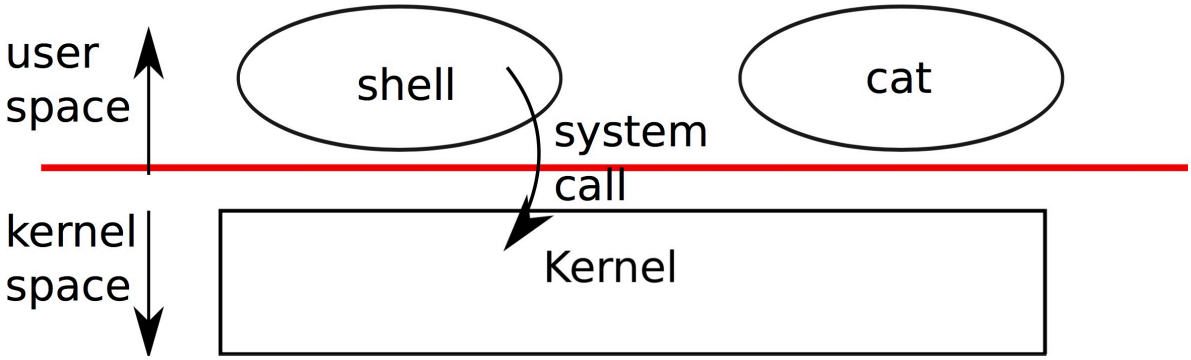

図 1　1 つのカーネルと 2 つのユーザプロセス

に入り、カーネルがサービスと結果返しを行う。それゆえプロセスはユーザスペースとカーネルスペースとの処理のやりとりを代行しているのである。

カーネルは CPU のハードウェア保護装置を、ユーザスペースにて各プロセス実行が自身のメモリにのみアクセスするのを確実化させるために使う。このカーネルはこれらの保護を通過するために求められるハードウェアの権限と共に実行するため、ユーザのプログラムらがわざわざこれらの権限を使わずに実行できる。ユーザプログラムがシステム指示を呼び出した際に、ハードウェアが権限のレベルと、カーネル内の既定の関数の処理実行開始を通知する。

カーネルが提供するシステム指示収集はユーザープログラムが参照するインターフェースである。xv6 のカーネルはのサービスのサブセットと UNIX カーネルがもたらすシステム指示を提供する。その指示が以下の表の通りである。

この先のチャプターにて xv6 のサービス—プロセス、メモリ、ファイル記述子、パイプ、そしてファイルシステム—そしてこれらについてコードやスニペット、そしてシェルがいかにこれらを使役するかという議論を用いて説明する。そのシェルがシステムコールを使役する方法を見ることで、いかに彼らが注意深くデザインされているのか物語ってくれるだろう。

シェルはユーザからや、実行されたものからのプログラムを読む一般的なプログラムで、そして伝統的な UNIX 的システムにおける最初のユーザインターフェースである。シェルはユーザプログラムという事実があり、カーネルの一部ではなく、システムコールインターフェースの力を表す例であり、すなわちシェルにおいては特異な要素はないのである。これはシェルは簡単に結果のような他のものに置き換えることができることもまた意味しており、また現代の UNIX システムは 1 種の選択するシェルを保持しており、それぞれが自身のユーザインターフェースとスクリプトの特徴を併せ持っている。xv6 のシェルは UNIX ボーンシェルの本質の単純なインプリメンテーションである。このインプリメンテーションは 7850 行目にて説明がある。

1.2　プロセスとメモリ

xv6 のプロセスはユーザスペースのメモリ (指示、データ、そしてスタック) と、毎プロセスのカーネルへの内部状態を含んでいる。xv6 ははタイムシェアリングを提供し、これが実行待ちのプロセス下におけるセット内の使用可能な CPU に判別しやすく切り替えることができる。プロセスが実行されていないときに xv6 は CPU レジスタを保持し、次にプロセスが実行されるときにそれ

システムコール	説明
fork()	プロセス作成
exit()	進行中プロセスを終了
wait()	子プロセスが終了するまで待機
kill(pid)	pid(プロセス ID) のプロセスを終了
getpid()	進行中プロセスの pid を返す
sleep(n)	n 秒間スリープする
exec(filename, *argv)	ファイルをロードし、実行
sbrk(n)	プロセスのメモリを n バイト増やす
open(filename, flags)	ファイルを開く (flags は読込/書込を表す)
read(fd, buf, n)	ファイルを buf 内へ開き、n バイト詠込む
write(fd, buf, n)	ファイルを開き、n バイト詠込む
close(fd)	fd にて指定した開いたファイルを閉じる
dup(fd)	fd を複製する
pipe(p)	パイプを開き、p 内の fd 群を返す
chdir(dirname)	カレントディレクトリを変更する
mkdir(dirname)	新規ディレクトリ作成
mknod(name, major, minor)	デバイスファイルを作成する
fstat(fd)	開いたファイルの情報を返す
link(f1, f2)	f1 のファイル名を f2 に変更する
unlink(filename)	ファイル削除

らを戻す。カーネルはプロセス識別子あるいは”pid”と、それぞれのプロセスを結びつけている。

プロセスは”fork”というシステムコールで新たなプロセスを作る時がある。fork は新たなプロセスを作り、それは子プロセスと呼ばれ、呼び出されるメモリは全く同一のものとなり、呼び出し元のプロセスを親プロセスと呼ぶ。fork は親子両方共を返す。親では、fork は子の pid を、子では 0 を返す。例えば、以下のプログラム片を考える。

```
1   int pid;
2   pid = fork();
3   if(pid > 0){
4     printf("parent: child=%d\n", pid);
5     pid = wait();
6     printf("child %d is done\n", pid);
7   } else if(pid == 0){
8     printf("child: exiting\n");
9     exit();
10  } else {
11    printf("fork error\n");
12  }
```

この exit システムコールは呼び出したプロセスを実行停止させ、メモリや開いたファイルなどといったリソース放棄を引き起こす。この wait システムコールは現プロセスの終了した子プロセスの pid を返し、もし対象プロセスの終了した子がない場合は、wait では終了するまで待機する。例えば、これらの出力では

```
1  parent: child=1234
2  child: exiting
```

はもしかしたら順番が前後しているかもしれなく、これは親あるいは子のどちらが先に printf コールを得たかに依存する。子が終了し親の wait システムコールの返り値を得ることで、親が以下を出力する

```
1  parent: child 1234 is done
```

親と子はそれぞれ異なるメモリとレジスタで実行されていることに注意しなければならず、1 つの変数を変更することが他所へ影響を及ぼすことはない。

この exec システムコールは呼び出したプロセスのメモリを、新たなファイルシステムに所蔵されたファイルから読み込んだメモリイメージ移動させる。このファイルは必ず特定のフォーマットを保持していなければならず、また内部には指示、データ部分、何処から指示が始まるか、等を指定ファイルに明記する必要があり、xv6 は ELF フォーマットを使用しており、詳細は Chapter2 にて記されている。exec が成功した時、何も呼び出したプログラムには値を返さず、そのかわりに ELF ヘッダ内にて、実行ファイルから読み込み、明示された開始ポイントを返す。exec は 2 つの引数があり、実行可能なファイルの名前と文字列を渡す。以下に例を示す。

```
1  char *argv[3];
2  argv[0] = "echo";
3  argv[1] = "hello";
4  argv[2] = 0;
5  exec("/bin/echo", argv);
6  printf("exec error\n");
```

このフラグメントは呼び出したプログラムを、引数の一覧である"echo hello"をある/bin/echo のプログラムのインスタンスへ送っている。多数のプログラムは最初の引数を無視しており、旧来のプログラムの名前として扱っている。

この xv6 のシェルは上位の呼び出しをユーザに変わってプログラムを走らせている。メインのシェルの構造は単純で、main(8001) を参照のこと。メインのループはコマンドライン上で"getcmd"によって入力を読み込んでいる。それから fork と呼ばれ、シェルの往路セスのコピーを作成する。親のシェルは wait と呼ばれるコールで、しばらく子プロセスがコマンドを走らせる。例えば、もしユーザが"echo hello"とプロンプトで打った場合、runcmd は"echo hello"を引数として呼び出すことになるだろう。runcmd(7906) は実際のコマンドを走らせる。"echo hello"に関しては、こちらは exec(7926) を呼び出すだろう。もし exec が完了したらその後は子供は runcmd の代わりに echo による指示を実行するだろう。いくつかの点において echo は exit を呼び出し、これが親に main(8001) における wait の値を返すのを引き起こすだろう。あなたはどうして fork と exec が別々ではなく、単独の呼び出しに統一されないのか気にするかもしれないが、我々はこ

のあとプロセス作成とプログラム読み込みが別々であることが、クレバーなデザインであることを知ることになるだろう。

xv6 は多くののユーザスペースのメモリを暗に割り当てており、例えば fork はメモリを親のメモリから子へコピーするために求められるだけ割り当てており、そして exec は実行できるファイルを保有するために必要なメモリを割り当てる。1 つのプロセスはあるときにより多くのメモリを要す。なぜならばランタイム (おそらくは、malloc のため) が、sbrk(n) を呼び出してメモリを n バイト増やし、sbrk が新規メモリの位置を返せられるようにする必要があるからである。

xv6 はユーザを他所から来たユーザから守るという概念は提供しておらず、UNIX の部分においては、すべての xv6 のプロセスは root として処理が行われる。