オペレーティングシステム

2022/11/22

海谷 治彦

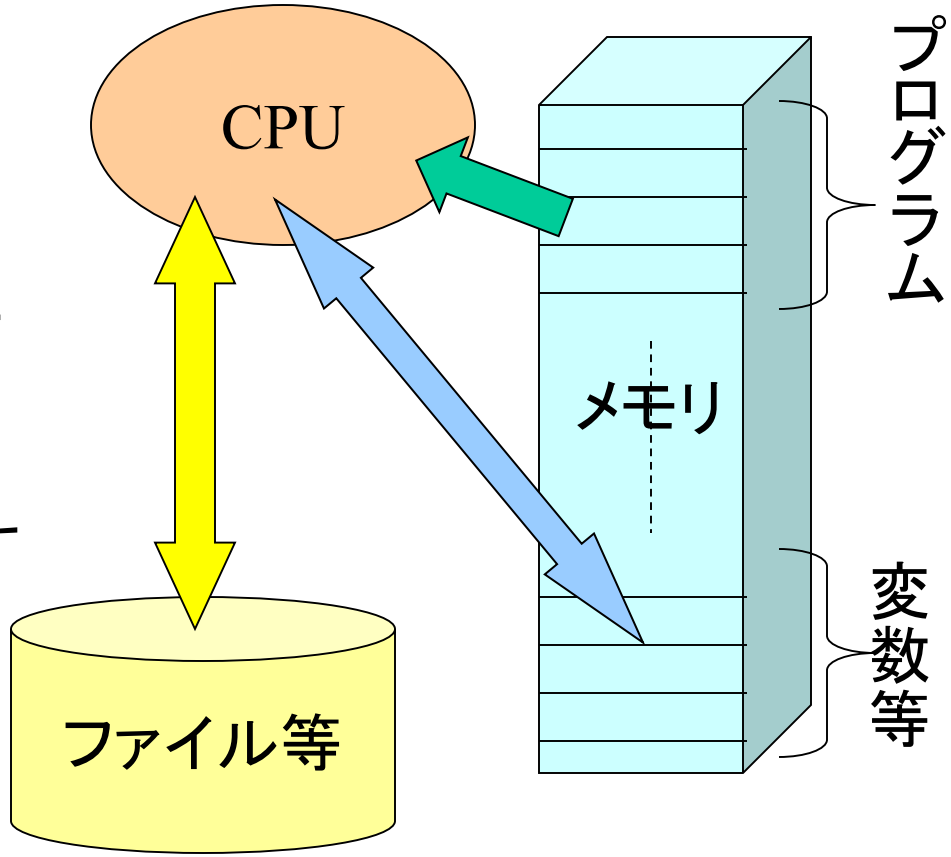
目次

- プロセスとその管理
- プログラムの実行制御とプロセス
- プロセスの実現
- プロセススケジュール
- 同アルゴリズム
- スレッド
- プロセスの生成と消滅
- shellについて
- プロセス間通信

復習

プログラムの処理の流れ

- プログラムがメモリに読み込まれる.
- 計算に必要なメモリも確保される.（変数等のため)
- CPUがプログラムを順に読んで，計算をする.
- 必要ならば，デバイス(ファイル等)にアクセスする.



復習

プロセス (タスク)

- 処理中のプログラム.
- プログラムのインスタンス.
- タスクとも呼ばれる.
- 前頁の「メモリに読み込まれたプログラムとデータ」に対応する概念.
- 1つのプログラムをもとに複数のプロセスが発生するため，プログラムとは概念的に区別される.
- 以下の対比でイメージを得て！

	書き物	実行
ソフトウェア	プログラム	プロセス
音楽	譜面	演奏
ゲーム	ルール(ブック)	実際のプレイ

復習

Linuxでの実際

- プロセスが計算を実行するためには資源が必要，少なくともメモリとCPUは必要.
- 1つのプログラムをもとに多数のプロセスが生成されている.
- 無論，実行可能なプログラムは多数ある.
- 沢山のプロセスが同時に動いている(ように見える).
- プロセスの生成と消滅が繰り返されている.

  OSを通して業務(アプリの実行)するので当たり前か.
- プロセスの寿命はまちまち.

  ls は一瞬で終わるが，httpd (ウエブサーバー)は何日も動いている．OSは当然，動きっぱなし.

復習

単一プロセスの管理

- プロセスの実体は,
  - プログラムを読み込んだCS (Code Segment)
  - 計算結果等を読み書きするDS (Data Segment)
  - レジスタの中身 (主に一般用)
  - アドレス変換テーブル

  と言える.
- OSはファイルから読み込んだプログラムに基づき，これらを生成・設定する.
- OSは全プロセスの上記情報を覚えておく.
- より具体的なプロセス生成，終了の手順は後日に.

何故プログラムとプロセスは別？

- 昔は逐次的にプログラムを実行することも多かったので，プロセスとプログラムを区別する必要があまり無かった.

- 現代では，書き物(譜面)としてのプログラムをCPUが同時に複数実行(演奏)することも珍しくない.
- よって，プログラムとプロセスの概念を分ける必要がある.

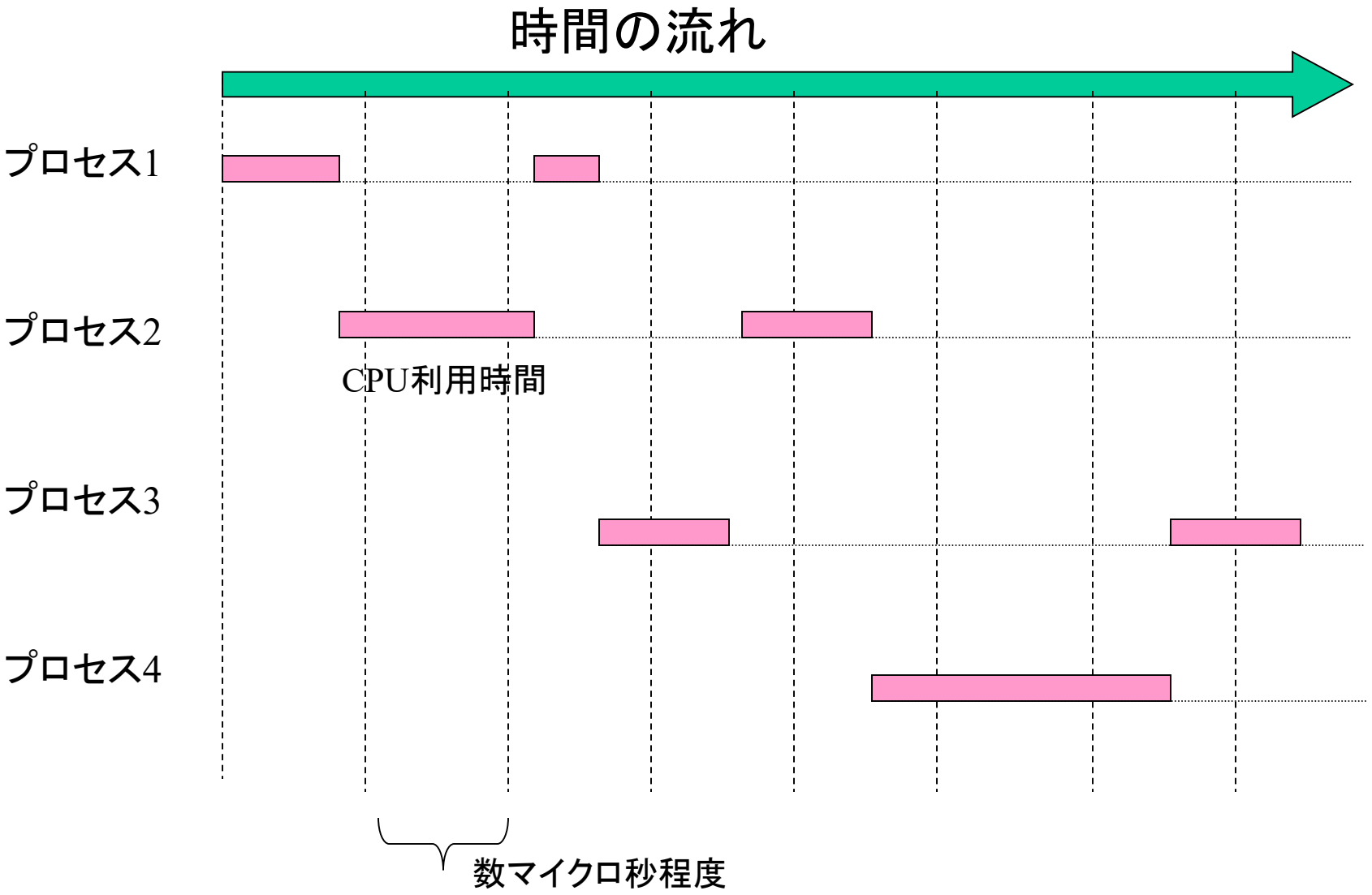
実例

```
kaiya     3242     1  0 May18 ?        00:00:00 /usr/libexec/gvfsd-burn --spawne
kaiya     3246     1  0 May18 ?        00:00:00 gnome-terminal
kaiya     3247  3246  0 May18 ?        00:00:00 gnome-pty-helper
kaiya     3248  3246  0 May18 pts/0    00:00:00 bash
root      3415  2054  0 May18 ?        00:00:00 /sbin/dhclient -d -4 -sf /usr/li
root      3449     2  0 May18 ?        00:00:00 [loop0]
root      3450     2  0 May18 ?        00:00:00 [jbd2/loop0-8]
root      3451     2  0 May18 ?        00:00:00 [ext4-dio-unwrit]
ntp       3485     1  0 May18 ?        00:00:00 ntpd -u ntp:ntp -p /var/run/ntpd
apache    3828  2451  0 May18 ?        00:00:00 /usr/sbin/httpd
apache    3829  2451  0 May18 ?        00:00:00 /usr/sbin/httpd
apache    3830  2451  0 May18 ?        00:00:00 /usr/sbin/httpd
apache    3831  2451  0 May18 ?        00:00:00 /usr/sbin/httpd
apache    3832  2451  0 May18 ?        00:00:00 /usr/sbin/httpd
apache    3833  2451  0 May18 ?        00:00:00 /usr/sbin/httpd
apache    3834  2451  0 May18 ?        00:00:00 /usr/sbin/httpd
apache    3835  2451  0 May18 ?        00:00:00 /usr/sbin/httpd
apache    3836  2451  0 May18 ?        00:00:00 /usr/sbin/httpd
postfix   7329  2417  0 16:29 ?        00:00:00 pickup -l -t fifo -u
root      7425  2298  0 16:50 ?        00:00:00 sshd: kaiya [priv]
kaiya     7429  7425  0 16:50 ?        00:00:00 sshd: kaiya@pts/1
kaiya     7430  7429  0 16:50 pts/1    00:00:00 bash
kaiya     7494  7430  1 16:52 pts/1    00:00:00 ps -ef
[kaiya@kaiya2014 ~]$ ls -l /usr/sbin/httpd
-rwxr-xr-x. 1 root root 341712 Apr  4 08:55 /usr/sbin/httpd
[kaiya@kaiya2014 ~]$
```

たくさんの httpd プロセスが動いているし,
二つの bash というプログラムも見える.

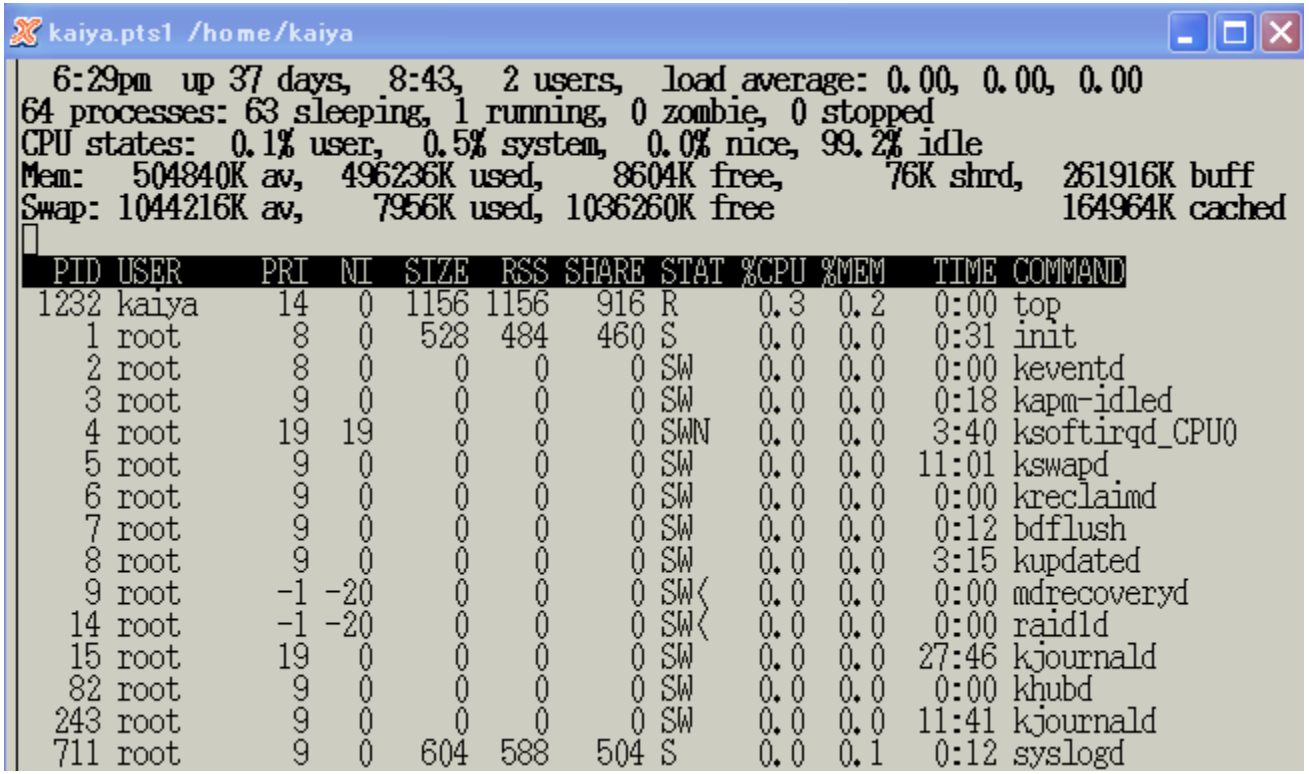
復習’

時分割の考え方 (図7.4改)



復習

Linuxでのプロセスの観察2



top というコマンドでも観察できる.

各プロセスは状態をもっている

```
[kaiya@linux2001 ~]% ps x
  PID TTY      STAT   TIME COMMAND
13909 ?        S      0:07 Xvnc :1 -desktop X -auth /home/s
13921 ?        S      0:00 twm
31408 pts/0    S      0:00 -csh
31796 ttyp0    S      0:00 -bin/tcsh
31943 ttyp0    T      0:00 vi a.c
31949 ttyp0    R      0:00 ps x
[kaiya@linux2001 ~]%
```

ここの部分が現在の状態を示す.

struct task_struct { // 無論，抜粋です include/linux/sched.h
volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
}

プロセスの状態

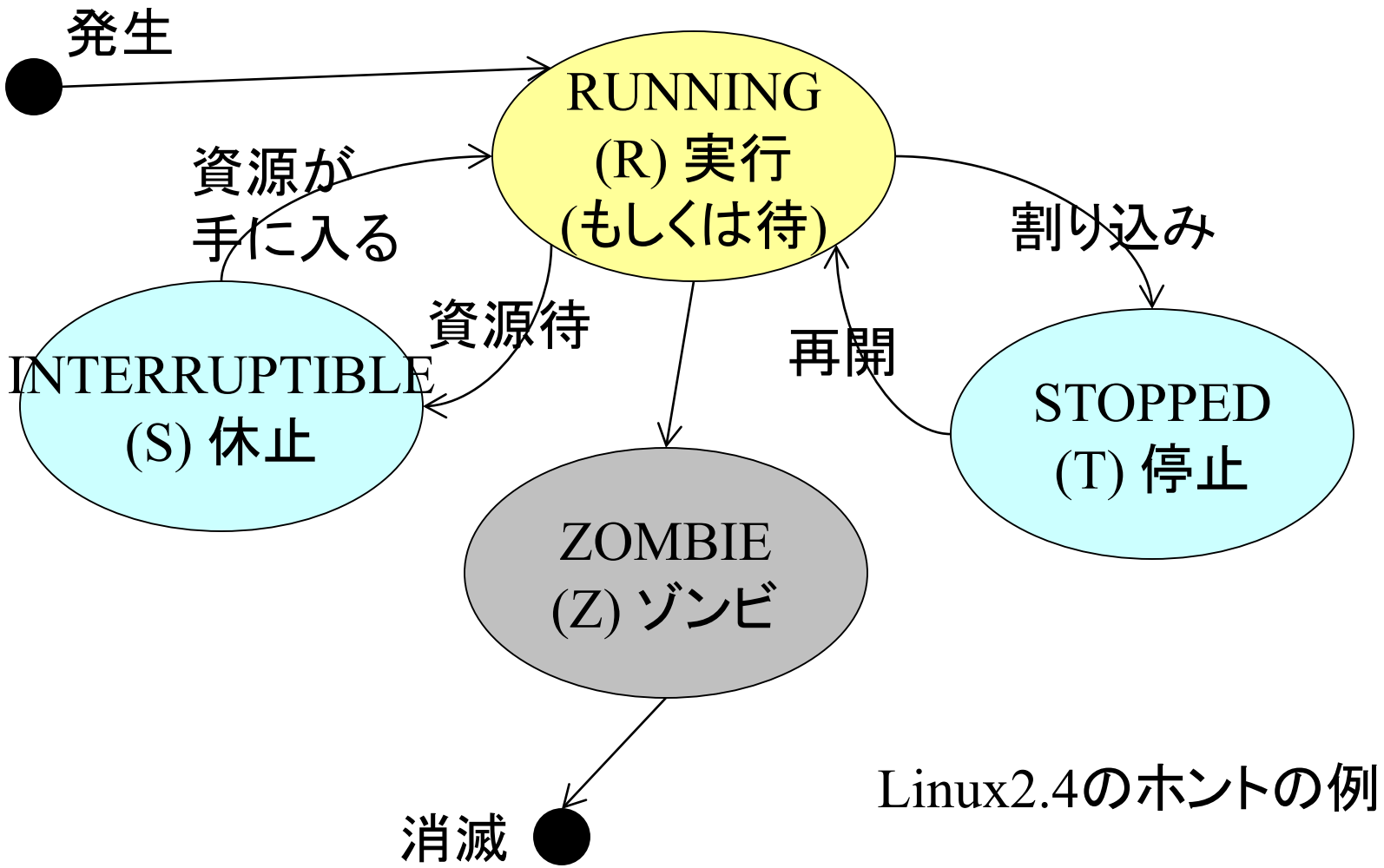
- そもそもCPUは1個程度なので，ある瞬間に実行されているプロセスも1個程度.
- 資源確保の関係等で，プロセスは常に実行状態とは限らない.
  - 例えば，diskの書き込み待ちとか.
- よって，各プロセスは状態変数 state を持ち，個々のプロセスの状態をkernelが知ることができる.
- とりうる状態は，6個くらいらしいが，主なものを次項に示す.

状態の値

- (R) TASK_RUNNING 実行中もしくは実行待ち
- (S) TASK_INTERRUPTIBLE ある条件が成り立つのを待っている状態，例えば必要な資源が空くのを待っているとか.
- (Z) TASK_ZOMBIE プロセスは終了しているが，完全に削除されていない状態.
- (T) TASK_STOPPED 外部からの割り込み等でプロセスが停止している状態

実際の値は include/linux/sched.h の85行目あたり.

1つのプロセスの状態遷移



Linux2.4のホントの例

プロセススケジュール

- プロセスはそれぞれの事情で休止したり停止したりしているが，いつかは条件がそろい実行可能となる．
- 複数の実行可能プロセスがあった場合，どれから実行するかを決めるのはOSである．
- この決め方のアルゴリズムがいくつかあり，OSの授業では定番のネタとなっている．

アルゴリズム紹介 1/2

- First Come First Served (FCFS)
  - プロセスが生成された順番にCPUを割り当てる.
- Shortest Processing Time First
  - 短いものから片づける
- Priority Scheduling
  - 予めプロセスに優先度をつける
  - UNIXでも一部採用されている
- Round Robin
  - 短いタイムスパンで公平に切り替える，TSS的.
  - 今時のOSの定番.

アルゴリズム紹介 2/2

- PriorityとRound Robinの組み合わせ
- Dynamic Dispatching
  - I/Oの利用頻度に基づき，優先度を変更する.
  - 今時のPC等には不向きだが，大型汎用コンピュータでは有効なのかもしれない.
- Multilevel Feedback Queue
  - 優先度の異なる複数の実行待ちの行列を準備する.

アルゴリズム全体の雰囲気

- 多くのプロセスをトータルで短時間で終わらせることに注視している.
  - 銀行の夜間一括処理的なものをスコープとしている.
- 今時の対話的な処理の場合「処理の終わり」が明確でないため，実は Round Robin 以外，あまり役に立たない.
- プロセスの応答性能を考慮したアルゴリズムが重要である.

復習

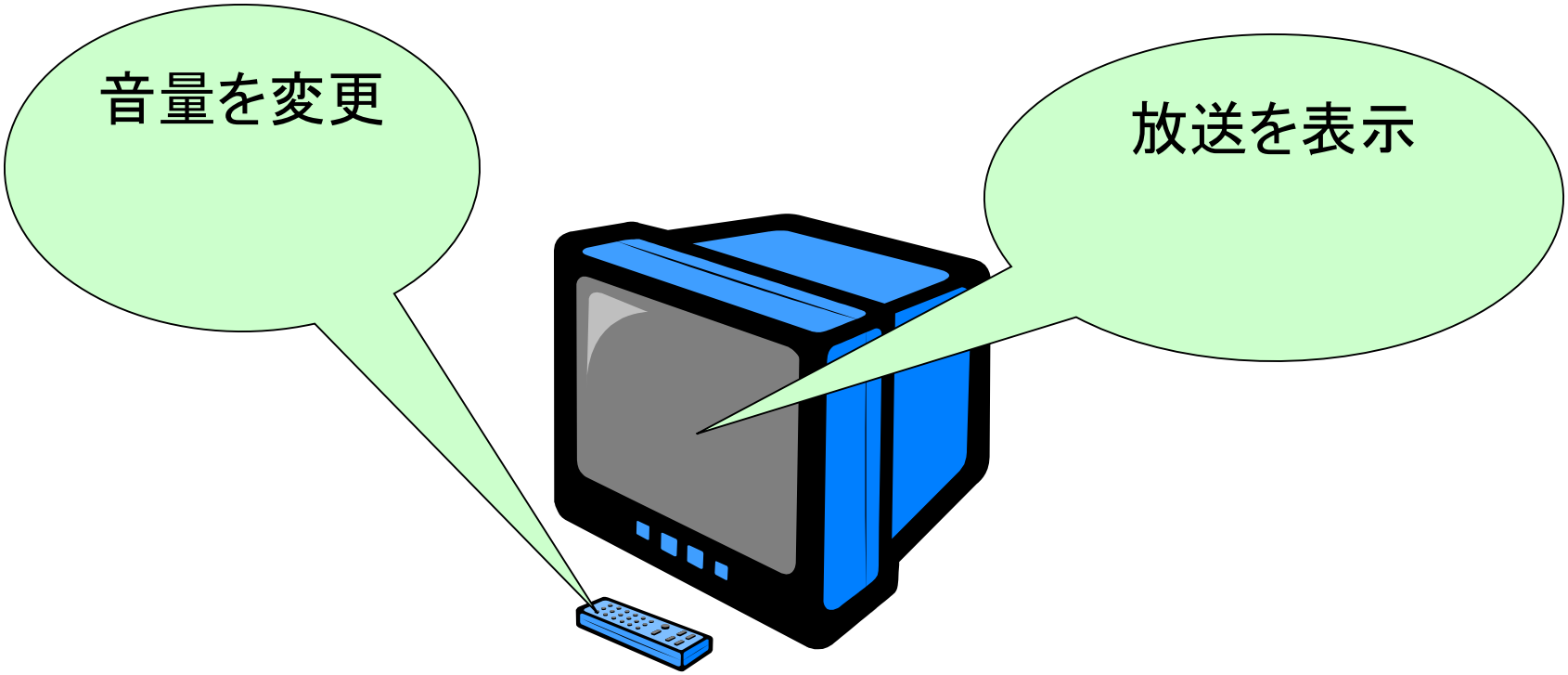
プロセス切り替え方式・今昔

- I/Oイベント駆動型
  - 実行中のプロセスが入出力に入ったら，他のプロセスに実行権を移す手法.
  - あるプロセスの事情が他のプロセスに影響を与えるため，「完全なマルチプロセスでは無い」といわれる.
  - 実現が簡単な古い方法.
  - Windows3.1で採用されていた，古い型の大型機でも.
  - 教科書の「4.2 マルチプログラミング」はコレに相当.
- プリエンプティブ型
  - Preemptive
  - 各プロセスの事情に関係なく，一定間隔でプロセスを切り替える.
  - CPUのタイマーが大きな役割を果たす.
  - 技術的に実現は上記より難しい.

並列動作: スレッドの前ふり

- 複数の処理が同時に動くこと.
- 自然界？ではあたりまえ.
- ハードウェアを伴う機器制御でもあたりまえ.

例: テレビの制御



音量変更中に，放映(画，音)が停止したら，やっぱ怒るよねぇ.

並列と並行 (用語の話)

- **並列 (parallel)**
  - 複数の仕事を同時に行う.
- **並行 (concurrent)**
  - 潜在的に同時進行可能な処理を論理的に表現したモデルやプログラムの性質.
- よって, あるプログラム言語では並行処理を記述できるが, それが並列に実行されるかはマシン次第.
  - シングルコアでは並列処理は無理.

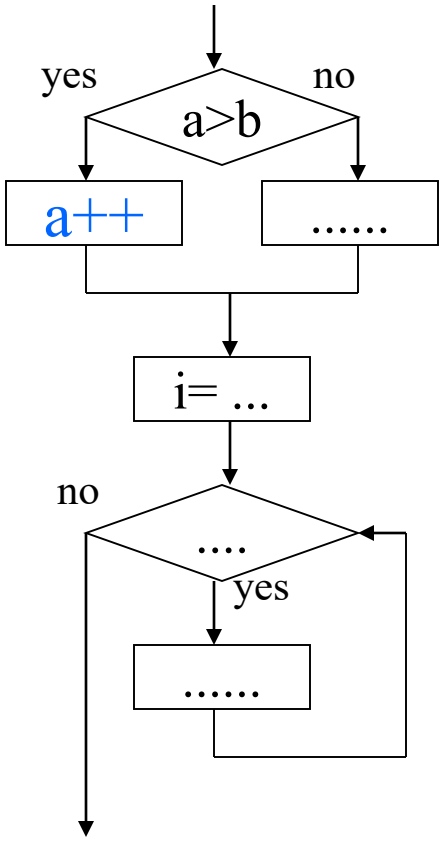
プロセス VS スレッド

- プロセス (もしくはタスク)
  - OSレベルで，独立した処理単位
  - ファイル等は共有できるが変数等は共有できない.
  - 詳細はすでに結構話してます.
- スレッド
  - 1つのプログラム内での異なる処理の流れ.
  - 1つのプログラム内なので，当然，変数等も共有できる.
  - 1つのプログラム内なので，プログラミングできます.

いままでやってきたプログラム

```
main(){
  int a, b, i;
  .....
  if(a>b){
      a++;
  }else{
      ........
  }
  for(i=... ){
      .....
  }
}
```

所謂
フローチャート



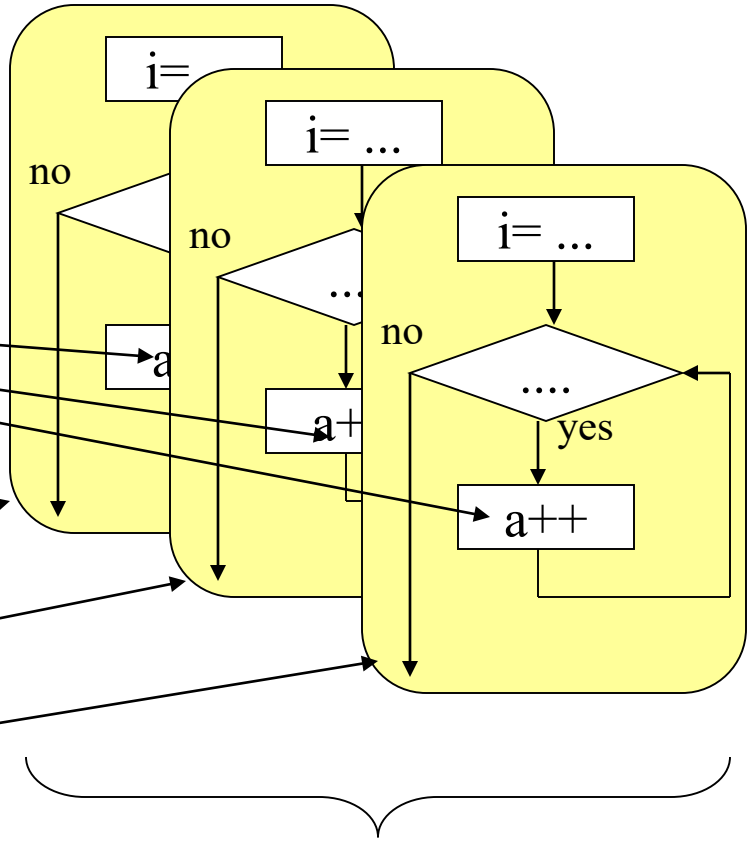
いままで学んできたプログラムは，ある一瞬には，ある1つの命令しか実行されていなかった.

処理の流れは一本である.

スレッドを利用したプログラム

```
class MyThread extends Thread{
static int a=0;
... run(){
    for(int i....){
        a++;
    }
}

.... main(....){
   .....
   new MyThread().start();
   new MyThread().start();
   new MyThread().start();
....
}
}
```



独立並行して動くことができる.
変数も共有できる.

スレッド・プログラミング

- Javaではとても気軽にスレッドを用いたプログラムを記述できる.
- C, C++等でも，スレッドを使うためのライブラリが完備されている.
- 他，新しい目の言語では大抵，スレッドが使える.
  - Ruby 等

余談 HTについて

- Intel hyper threading (HT)
- Core i7等に採用されている最新？技術.
- 1個のCPU(というかコア)で二つ(以上)の命令を同時に実行するための技術.
- 結果，物理的なCPU(コア)よりも多くのCPUがあるように振舞えるハードウェアの技術.

- 今回のスレットとはニュアンスが異なりますので，注意してください.

プロセスの生成

- 一般的にLinux/UNIXでは，すでに存在するプロセスの複製をつくり，複製の内容を作り変えることで，新しいプロセスを生成する.

- この複製もとになっているプロセスを通常，「親プロセス」と呼ぶ.

プロセスの親子関係の例

```
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 Aug27 ?        00:00:05 init
root           2     1  0 Aug27 ?        00:00:00 [kflushd]
root           3     1  0 Aug27 ?        00:00:01 [kupdate]
root           4     1  0 Aug27 ?        00:00:00 [kpiod]
root           5     1  0 Aug27 ?        00:00:04 [kswapd]
root           6     1  0 Aug27 ?        00:00:00 [mdrecoveryd]
root          47     1  0 Aug27 ?        00:00:00 [khubd]
root         547     1  0 Aug27 ?        00:00:02 /usr/sbin/sshd
root         940   547  0 23:18 ?        00:00:00 /usr/sbin/sshd
kaiya        941   940  0 23:18 pts/0    00:00:00 -csh
kaiya       1013   941  0 23:23 pts/0    00:00:00 ps -ef
kaiya        968   941  0 23:19 pts/0    00:00:00 vi a.c
root         538     1  0 Aug27 ?        00:00:00 inetd
root         983   538  0 23:22 ?        00:00:00 in.rlogind
root         984   983  0 23:22 pts/1    00:00:00 login -- kaiya
kaiya        985   984  0 23:22 pts/1    00:00:00 -bash
kaiya       1012   985  3 23:23 pts/1    00:00:00 emacs Foo.java
```

ある日，あるマシンのプロセスを抜粋 (ps –ef)

読み方

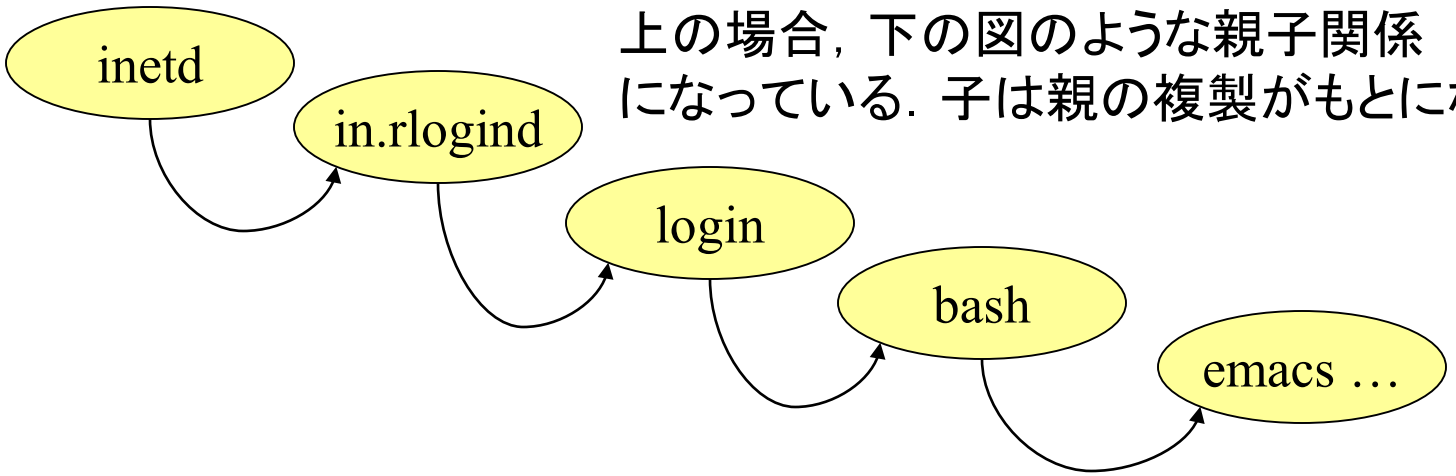
```
UID          PID  PPID  C STIME TTY          TIME CMD
root         538     1  0 Aug27 ?        00:00:00 inetd
root         983   538  0 23:22 ?        00:00:00 in.rlogind
root         984   983  0 23:22 pts/1    00:00:00 login -- kaiya
kaiya        985   984  0 23:22 pts/1    00:00:00 -bash
kaiya       1012   985  3 23:23 pts/1    00:00:00 emacs Foo.java
```

1行が1プロセス

自プロセスの番号

親プロセスの番号

プロセスのもとになったコマンド名



上の場合，下の図のような親子関係になっている．子は親の複製がもとになっている．

最初のプロセス

- 複製をもとにプロセスが生成されると，最初にタネになるプロセスがないとはじまらない.
- Linuxには以下の2つのタネになるプロセスがある.
  - プロセス0 Swapper, 初期化プロセス等とよばれ，カーネル内の変数等の初期化をする.
  - プロセス1 Init ほとんどすべてのプロセスの先祖となる

最初のプロセスの実際

- プロセス0 Swapper
  - init/main.c の中の，start_kernel(void)関数が実体.
- プロセス1 init
  - init/main.c の，init(void * unused)関数が実体.
  - init/main.c の中の一番最後に記述されている.
- ゼロからLinuxが起動するあたりの話は時間に余裕があればやります.

```
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Aug27 ?        00:00:05 init
root         2     1  0 Aug27 ?        00:00:00 [kflushd]
root         3     1  0 Aug27 ?        00:00:01 [kupdate]
root         4     1  0 Aug27 ?        00:00:00 [kpiod]
root         5     1  0 Aug27 ?        00:00:04 [kswapd]
```

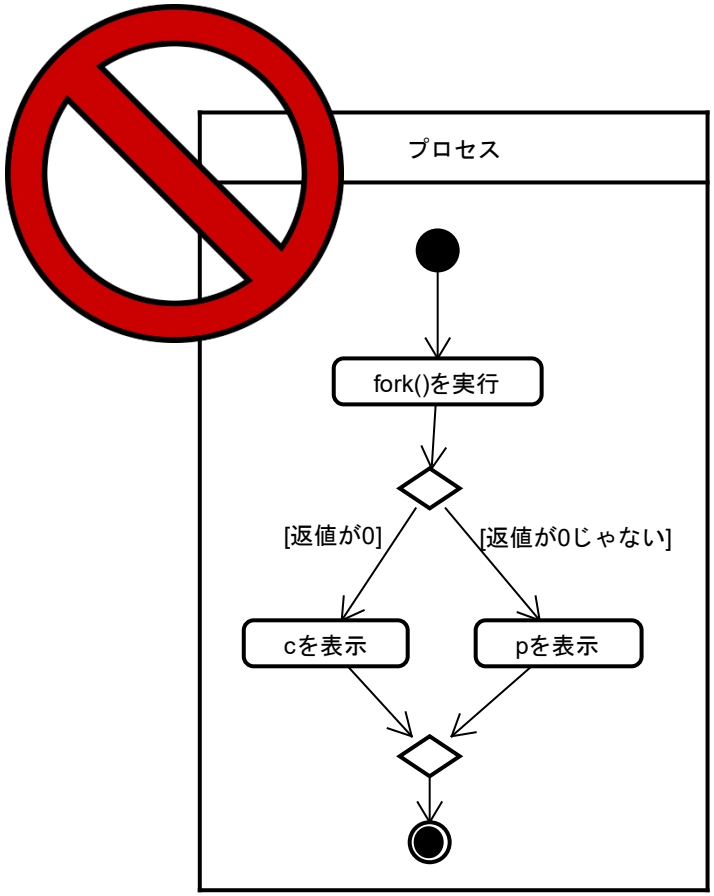
どうやって複製を作るか？

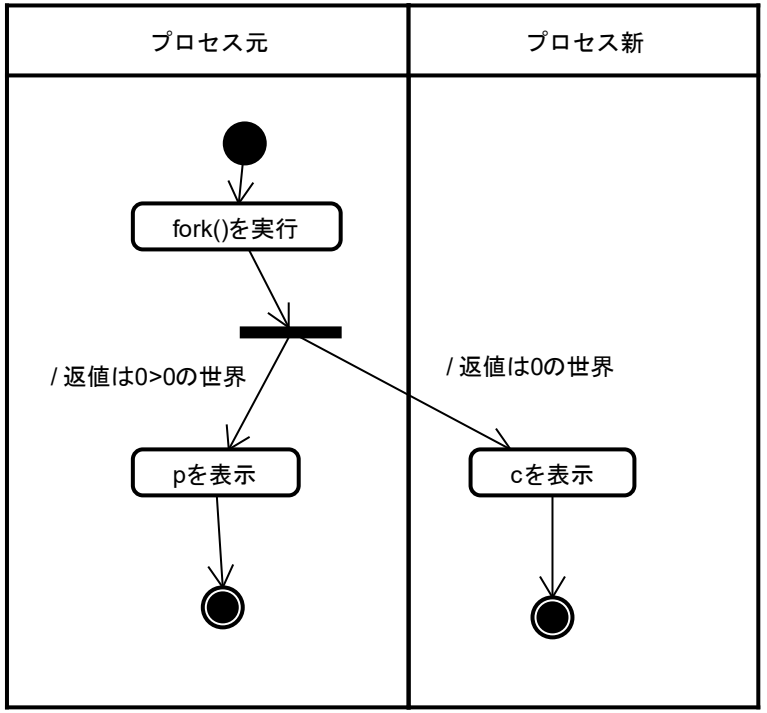
- forkシステムコールを利用
  - 実際に複製を作成する関数.
  - man fork 参照
- cloneシステムコールを利用
  - 親と一部のデータを共有する子プロセスを作成する関数.
  - 上記のforkより処理が軽い.
  - 本講義ではとりあず扱わない.

fork1.cのサンプルプログラム (抜粋)

```
 1| void showchar(char c){
 2|   // 省略
 3| }
 4|
 5| main(int argc, char* argv[]){
 6| pid_t ch;
 7|   if((ch=fork())==0){ // child
 8|     showchar('c');
 9|   }else if(ch>0){ // parent
10|     showchar('p');
11|   }
12|
13| }
```

動作のフロー


プロセス
fork()を実行
[返値が0]
[返値が0じゃない]
cを表示
pを表示


プロセス元
プロセス新
fork()を実行
/ 返値は0>0の世界
/ 返値は0の世界
pを表示
cを表示

fork()関数の実行

- この実行が行われた時点でプロセスのコピーが作成される.
- 実行後，自分がコピー(子供)かオリジナルかはfork()の返り値でわかる.
  - 返り値=0: 子供
  - 返り値>0: オリジナル，値は子供のプロセスID
  - それ以外: fork()失敗.
- 前述の例では，if文の最初の条件が成り立った分岐は子の処理の流れ，次の分岐がオリジナルの流れとなる.
- 分岐した流れは併合することはない.

fork1.cの説明

- 単純に自分の複製を作成するプログラム.
- プログラム自体は文字cを1秒おきに20個画面に表示するダケの関数 showchar(c)を実行しているだけ.
- しかし親プロセスと子プロセスで異なる文字を表示するため，プロセスが複製されたことがわかる.

fork1.c の実行と観察

- 単にコンパイルすれば動きます.
  - WSL2でも動きました.
- 動作させて二つの文字が表示されるのを確認する.
- 同時にpsコマンド(ps –lx)で同じ名前のプロセスが存在し,
- 親子関係があるのを確認する.

shellの実体 プロセス複製器

```
[kaiya@linux2001 fork]$ /usr/bin/cal
    October 2003
Su Mo Tu We Th Fr Sa
            1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

[kaiya@linux2001 fork]$
```

bashの例

```
[kaiya@linux2001 ~/fork]% /bin/ls
a.out  forktest.c
[kaiya@linux2001 ~/fork]%
```

tcshの例

コマンド名(プログラム名)をいれるとプログラムが実行されるのは，シェルといわれるプロセス複製プログラムと対話していることになる.

文献1 p.81, shellは自分で作れる！

fork2.c 簡単なshell

- 文字入力をコマンドとみたてて，その実行を行うプログラム.
- bashやtcshも基本的にはこの構成.
- プロセス生成・消滅機構の簡単な例.
- 観察事項
  - 確かに他のコマンドを呼び出せるかを確認.
    - コマンドはフルパスで書く必要がある.
  - 呼び出されたコマンドともとのプログラムに親子関係があるかを ps –xl 等で確認.
    - 親が10秒待つようにコードをかいてある.

fork2.c の概要

```
 1| main(int argc, char* argv[]){
 2| pid_t ch; char buf[100];
 3|
 4|   while(fgets(buf, 100, stdin)!=NULL){
 5|     buf[strlen(buf)-1]='¥0';
 6|     if((ch=fork())==0){ // child
 7|       execl(buf, buf, NULL); // execveを呼ぶ
 8|     }else if(ch>0){ // parent
 9|       sleep(10);
10|       printf("done %d¥n", ch);
11|       wait(0);
12|     }
13|   }
14|
15| }
```

ライブラリ関数 execl

- 実行中のプロセスを他のプログラムに作り変える関数.
- システムコール execveを簡易に使えるようにしたもの. (フロントエンド)
- 詳細はマニュアルを参照.
- execlpとexeclvとか仲間の関数が多数ある.

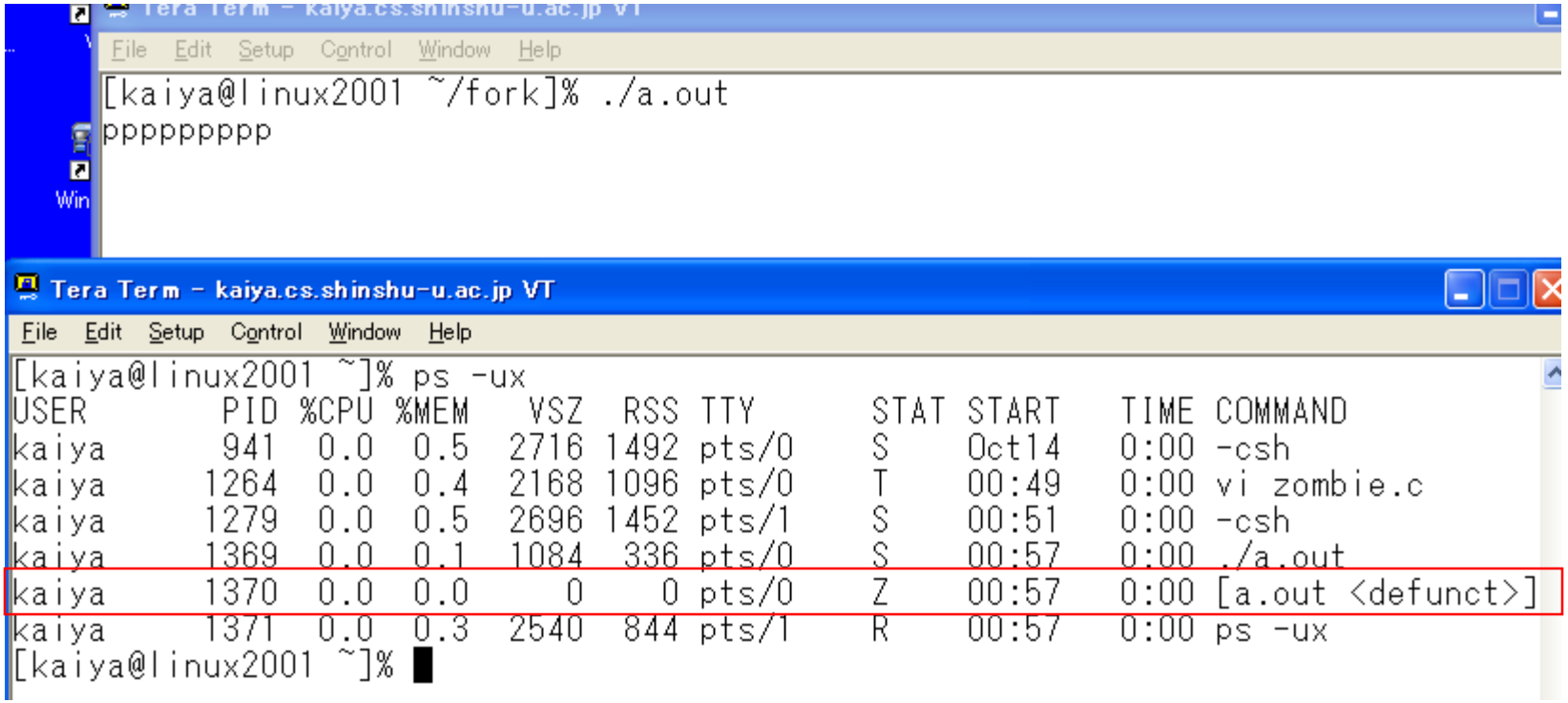
システムコール wait

- 子プロセスの実行終了を待つための関数.
- 同時に子プロセスの利用していた資源の解放も行う.
  - コレによって子プロセスは完全に消滅する.
  - コレをしないとゾンビ(後述)が残る場合がある.

プロセスの消滅とゾンビ

- 計算が終わるとプロセスも消滅し，カーネル内から削除される‥‥はずである.
- しかし，(死んだ)子供の情報に親がアクセスする場合をUNIXは想定しているので，計算が終わったのにプロセスのデータが残っているという状態が起こる.
- この状態を，ゾンビ状態という.

ゾンビの例



詳細は zombie.c を参照.

プロセス間通信

- Inter Process Communication (IPC)
- プロセス同士はデータを交換するための通信路を結ぶことができる.
- 異なるマシン上のプロセス同士でも, この仕組みによって情報交換が可能となっている.
  - ウエブサイトとブラウザが通信できるのも, この技術のおかげ.
- 初期に扱ったパイプは, この技術の一種である.

復習

パイプ&フィルタ モデル

- UNIX/Linux等のコマンド言語は，複数のコマンドを接続して，より複雑な処理を「その場で」(on the fly)構成することができる.
- このような処理構成法をパイプ&フィルタ モデルと呼ぶ.
- 処理対象のデータが行で区分けされたテキストでないと，うまく機能しない場合が多い.
- パイプは | で表現する場合が多い.
- 個々のコマンドがフィルタの役目をする.
- 話は簡単で，
  - 1個前のコマンドの出力を次のコマンドの出力とする.
  - だけ.
- GUIでは，このような，臨機応変な対応が容易ではない.

復習’

例

拡張子が .txt のファイルだけ列挙しました.

```
E:¥>ls *.txt
bus.txt  log.txt  oh.txt  output1.txt  output2.txt  tel.txt  todo.txt
```

拡張子をとりました.

```
E:¥>ls *.txt | sed -n s/.txt//p
bus
log
oh
output1
output2
tel
todo
```

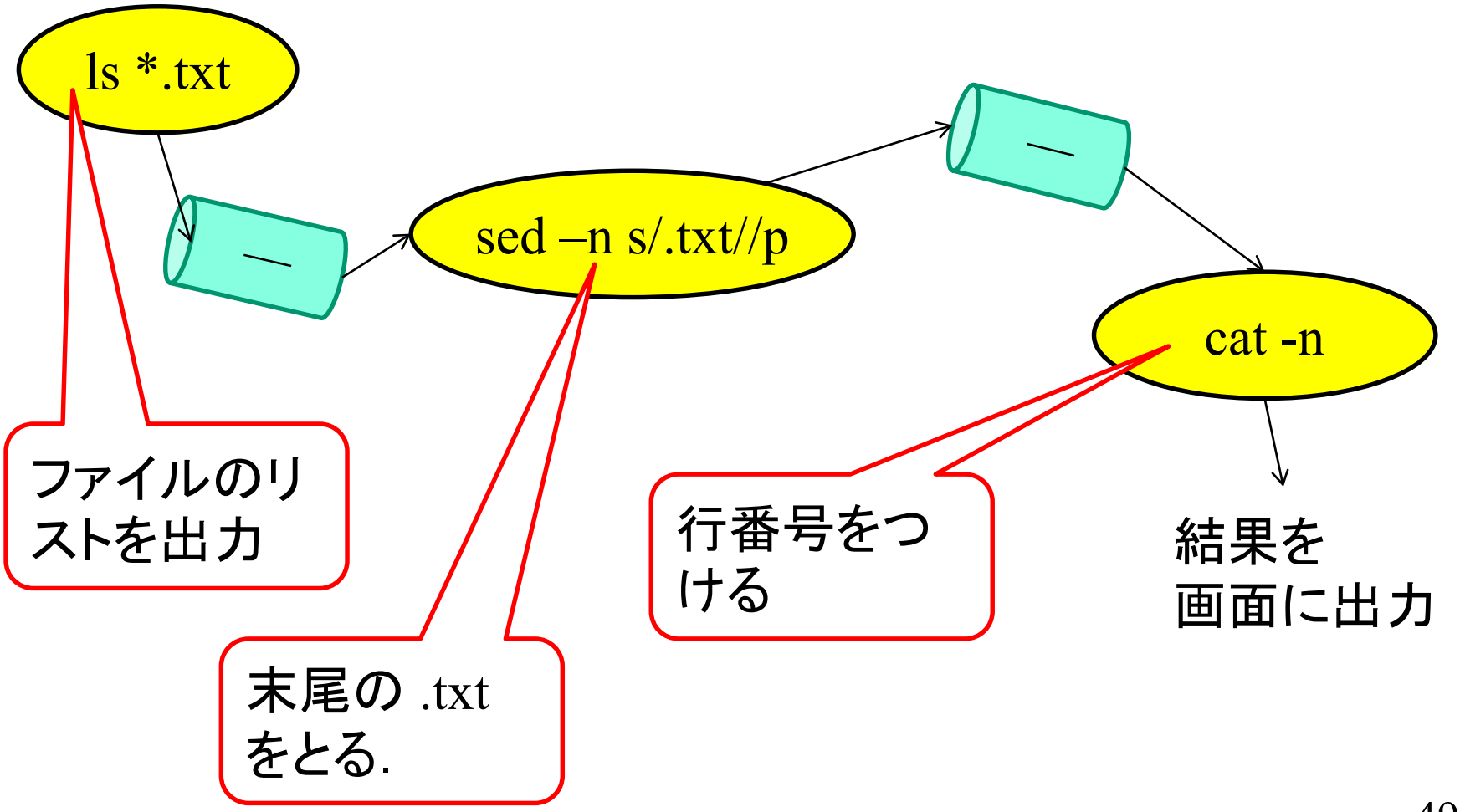
行番号をつけてみました

```
E:¥>ls *.txt | sed -n s/.txt//p | cat -n
     1  bus
     2  log
     3  oh
     4  output1
     5  output2
     6  tel
     7  todo
```

復習’

概念説明

```
ls *.txt | sed -n s/.txt//p | cat -n
```



本日は以上

アンケートのほう，
よろしくご提出ください

本日は以上

アンケートのほう，
よろしくご提出ください