

システムプログラム 第5回

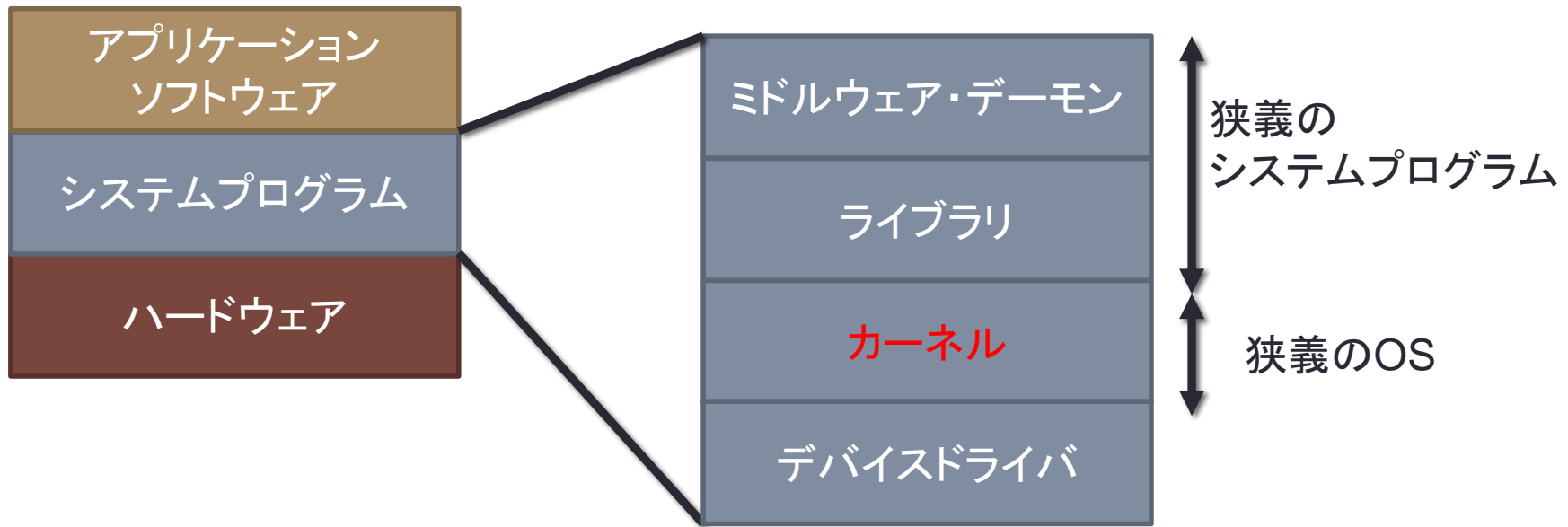
創域理工学部 情報計算科学科

松澤 智史

本日の内容

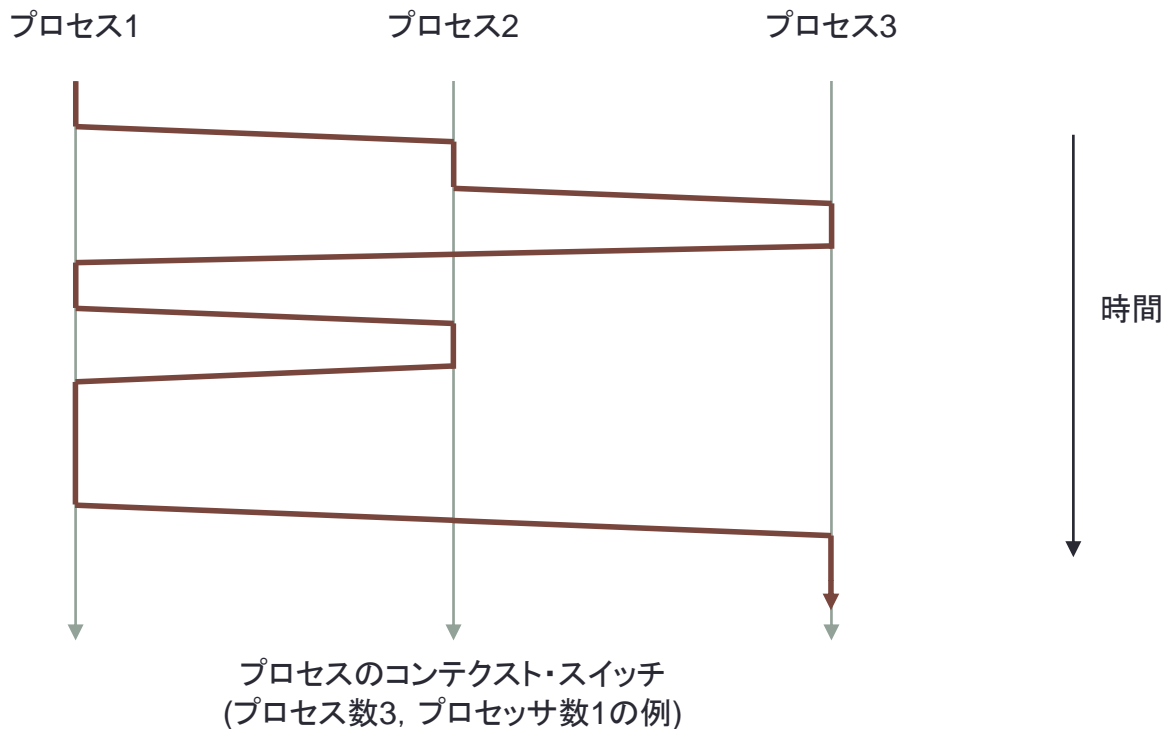
- システムプログラムの一部であるOSについて学ぶ
- 本日の内容はOSの機能の1つである「プロセス」を学ぶ

システムプログラム・・とは



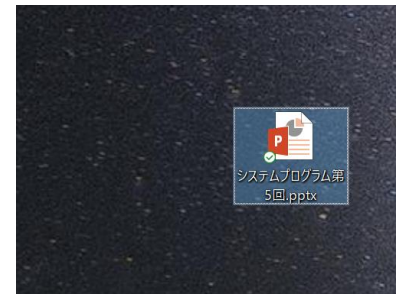
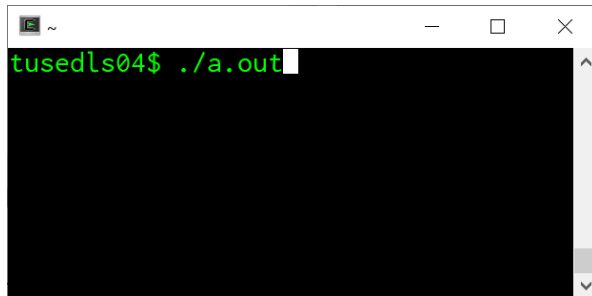
プロセス(第2回の復習)

- 実行中のプログラムをOS(カーネル)が抽象化したもの
- 各プロセスはハードウェアを独占的に使用しているように見える
- 多くのシステムでは「プロセスの数>実行するプロセッサの数」
- コンテキストスイッチにより並行動作
(あたかも独占的に動作しているように見せる)



プロセス

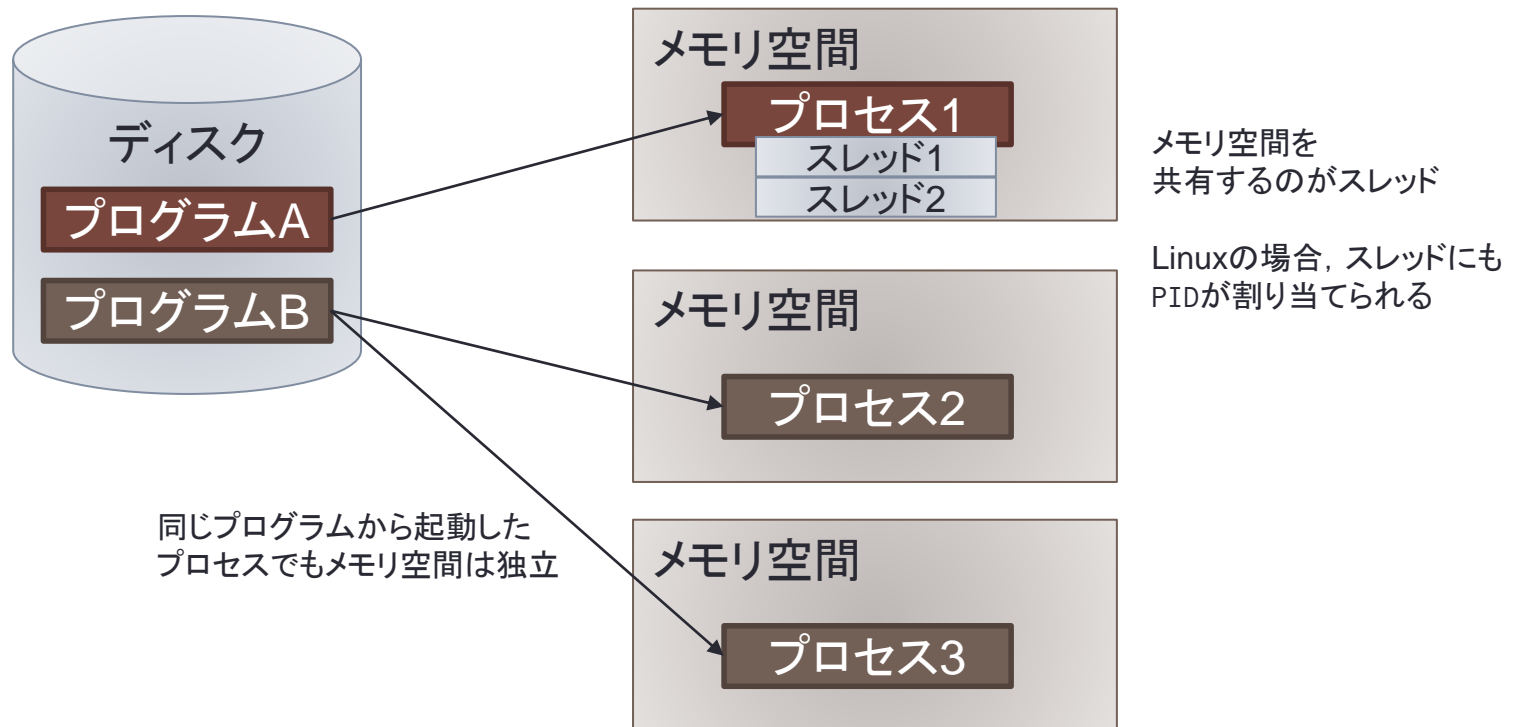
- プログラムの実行単位（タスク, ジョブなどと呼ばれることもある）
- プログラムとデータはメモリ上にあり, CPUによって逐次取り出して実行する
- 複数のプログラムを並行処理できる
 - 同じプログラムを並行処理はできない(Linuxカーネル2.4以前)
 - 同じプログラムの並行動作はスレッド(Linuxカーネル2.4以降)で行える



コマンドラインから実行, ダブルクリックで実行
→プロセス起動

プロセス(タスク)管理

- プログラムが実行されるとカーネルはプロセスを生成する
- カーネルはプログラムを主メモリに読み込み、独立したメモリ空間を割り当てる
- プロセスにはプロセスID(PID)という管理番号が割り当てられる



プロセスの確認

```
tusedls04$ ps
  PID TTY          TIME CMD
 23130 pts/0    00:00:00 bash
 23723 pts/0    00:00:00 ps
tusedls04$
```

ps コマンドで確認

```
tusedls04$ ps -aux
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root             1   0.0  0.0 191672 4740 ?        Ss   07:35   0:02 /usr/lib/systemd/syste
root             2   0.0  0.0     0     0 ?        Ss   07:35   0:00 [kthreadd]
root             3   0.0  0.0     0     0 ?        Ss   07:35   0:00 [ksoftirqd/0]
root             5   0.0  0.0     0     0 ?        Ss   07:35   0:00 [kworker/0:0H]
root             7   0.0  0.0     0     0 ?        Ss   07:35   0:00 [migration/0]
root             8   0.0  0.0     0     0 ?        Ss   07:35   0:00 [rcu_bh]
root             9   0.0  0.0     0     0 ?        Ss   07:35   0:06 [rcu_sched]
root            10   0.0  0.0     0     0 ?        Ss   07:35   0:00 [lru-add-drain]
root            11   0.0  0.0     0     0 ?        Ss   07:35   0:00 [watchdog/0]
root            12   0.0  0.0     0     0 ?        Ss   07:35   0:00 [watchdog/1]
root            13   0.0  0.0     0     0 ?        Ss   07:35   0:00 [migration/1]
root            14   0.0  0.0     0     0 ?        Ss   07:35   0:00 [ksoftirqd/1]
root            16   0.0  0.0     0     0 ?        Ss   07:35   0:00 [kworker/1:0H]
root            17   0.0  0.0     0     0 ?        Ss   07:35   0:00 [watchdog/2]
root            18   0.0  0.0     0     0 ?        Ss   07:35   0:00 [migration/2]
root            19   0.0  0.0     0     0 ?        Ss   07:35   0:00 [ksoftirqd/2]
root            21   0.0  0.0     0     0 ?        Ss   07:35   0:00 [kworker/2:0H]
root            22   0.0  0.0     0     0 ?        Ss   07:35   0:00 [watchdog/3]
root            23   0.0  0.0     0     0 ?        Ss   07:35   0:00 [migration/3]
root            24   0.0  0.0     0     0 ?        Ss   07:35   0:00 [ksoftirqd/3]
root            25   0.0  0.0     0     0 ?        Ss   07:35   0:00 [kworker/3:0H]
root            26   0.0  0.0     0     0 ?        Ss   07:35   0:00 [kworker/3:0H]
root            27   0.0  0.0     0     0 ?        Ss   07:35   0:00 [watchdog/4]
root            28   0.0  0.0     0     0 ?        Ss   07:35   0:00 [migration/4]
root            29   0.0  0.0     0     0 ?        Ss   07:35   0:00 [ksoftirqd/4]
root            30   0.0  0.0     0     0 ?        Ss   07:35   0:00 [kworker/4:0H]
root            31   0.0  0.0     0     0 ?        Ss   07:35   0:00 [kworker/4:0H]
root            32   0.0  0.0     0     0 ?        Ss   07:35   0:00 [watchdog/5]
root            33   0.0  0.0     0     0 ?        Ss   07:35   0:00 [migration/5]
root            34   0.0  0.0     0     0 ?        Ss   07:35   0:00 [ksoftirqd/5]
root            36   0.0  0.0     0     0 ?        Ss   07:35   0:00 [kworker/5:0H]
root            37   0.0  0.0     0     0 ?        Ss   07:35   0:00 [watchdog/6]
root            38   0.0  0.0     0     0 ?        Ss   07:35   0:00 [migration/6]
root            39   0.0  0.0     0     0 ?        Ss   07:35   0:00 [ksoftirqd/6]
root            41   0.0  0.0     0     0 ?        Ss   07:35   0:00 [kworker/6:0H]
root            42   0.0  0.0     0     0 ?        Ss   07:35   0:00 [watchdog/7]
root            43   0.0  0.0     0     0 ?        Ss   07:35   0:00 [migration/7]
root            44   0.0  0.0     0     0 ?        Ss   07:35   0:00 [ksoftirqd/7]
root            45   0.0  0.0     0     0 ?        Ss   07:35   0:00 [kworker/7:0H]
root            46   0.0  0.0     0     0 ?        Ss   07:35   0:00 [kworker/7:0H]
root            47   0.0  0.0     0     0 ?        Ss   07:35   0:00 [watchdog/8]
root            48   0.0  0.0     0     0 ?        Ss   07:35   0:00 [migration/8]
root            49   0.0  0.0     0     0 ?        Ss   07:35   0:00 [ksoftirqd/8]
root            51   0.0  0.0     0     0 ?        Ss   07:35   0:00 [kworker/8:0H]
root            52   0.0  0.0     0     0 ?        Ss   07:35   0:00 [watchdog/9]
root            53   0.0  0.0     0     0 ?        Ss   07:35   0:00 [migration/9]
root            54   0.0  0.0     0     0 ?        Ss   07:35   0:00 [ksoftirqd/9]
root            56   0.0  0.0     0     0 ?        Ss   07:35   0:00 [kworker/9:0H]
```

ps aux (axは全プロセス)
(uはプロセスの所有者表示)
-aux -axu でも良い

実験

```
tusedls04$ grep cores /proc/cpuinfo
cpu cores      : 1
cpu cores      : 1
cpu cores      : 1
cpu cores      : 1
cpu cores      : 1
cpu cores      : 1
cpu cores      : 1
cpu cores      : 1
cpu cores      : 1
cpu cores      : 1
tusedls04$
```

CPUの数を確認
tusedls04は10個のCPU

```
tusedls04$ ps aux |wc -l
274
tusedls04$
```

現在のプロセスの数
274

プロセスの数 >>>> CPUの数

並行処理(Concurrent)

並行処理

- 複数の処理が同時に走っており、
処理やイベントなどが起こるタイミングが任意である処理

例

あなたは台所で鍋の料理をしようとしている
あなたのすることは以下である

1. 鍋に水を入れ火をかける
2. お湯の温度があがる間に野菜を切る
3. まだお湯が沸いていないので、先に食卓にお皿を並べ始める
4. お皿を並べている途中にお湯が沸いたので、お皿並べを中断し、
切っておいた野菜を鍋に入れる
5. 野菜を鍋に入れたら、野菜が煮えるまでお皿並べの続きを再開する

このような処理が並行処理になる

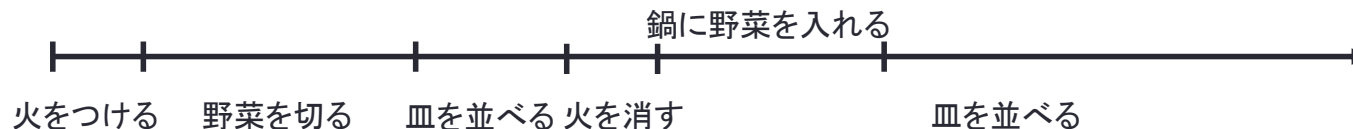
並行処理: 料理の例

タスク

- お湯をわかす→火をつける, 消す
- 野菜を切る
- 皿を並べる
- 野菜を煮込む→鍋に野菜を入れる

制約

- 「野菜を煮込む」タスクは, 「お湯をわかす」と「野菜を切る」というタスクが終わってからでないと実行できない
- 「火を消す」のタスクは最優先



並行処理: 計算機上(OS)の例

タスク(プロセス)

- Webブラウジング(Chrome, Safari等)
- 音楽再生
- ウイルススキャン

など

これらのプロセスの実行を適宜切り替えて,
同時に動いているように見せかける

余談:

プロセスは計算機上でプログラムを動かす実行単位
タスクは行うべき仕事という抽象的表現
計算機上でのタスクはプロセスと同義

並列処理(Parallel)

- 広い意味での並行処理の一部
- 並行処理は複数のタスクを1人で、**見せかけ上同時進行**しているように見える処理であることに対し
- 並列処理は実際に処理する人数がタスクと同じ数で行う(**実際に同時に行う**)処理

並列処理に関しては後の回で詳細に説明する

戻って・・・実験結果

```
tusedls04$ grep cores /proc/cpuinfo
cpu cores      : 1
cpu cores      : 1
cpu cores      : 1
cpu cores      : 1
cpu cores      : 1
cpu cores      : 1
cpu cores      : 1
cpu cores      : 1
cpu cores      : 1
cpu cores      : 1
tusedls04$
```

CPUの数を確認

tusedls04は10個のCPU

```
tusedls04$ ps aux |wc -l
274
tusedls04$
```

現在のプロセスの数

274

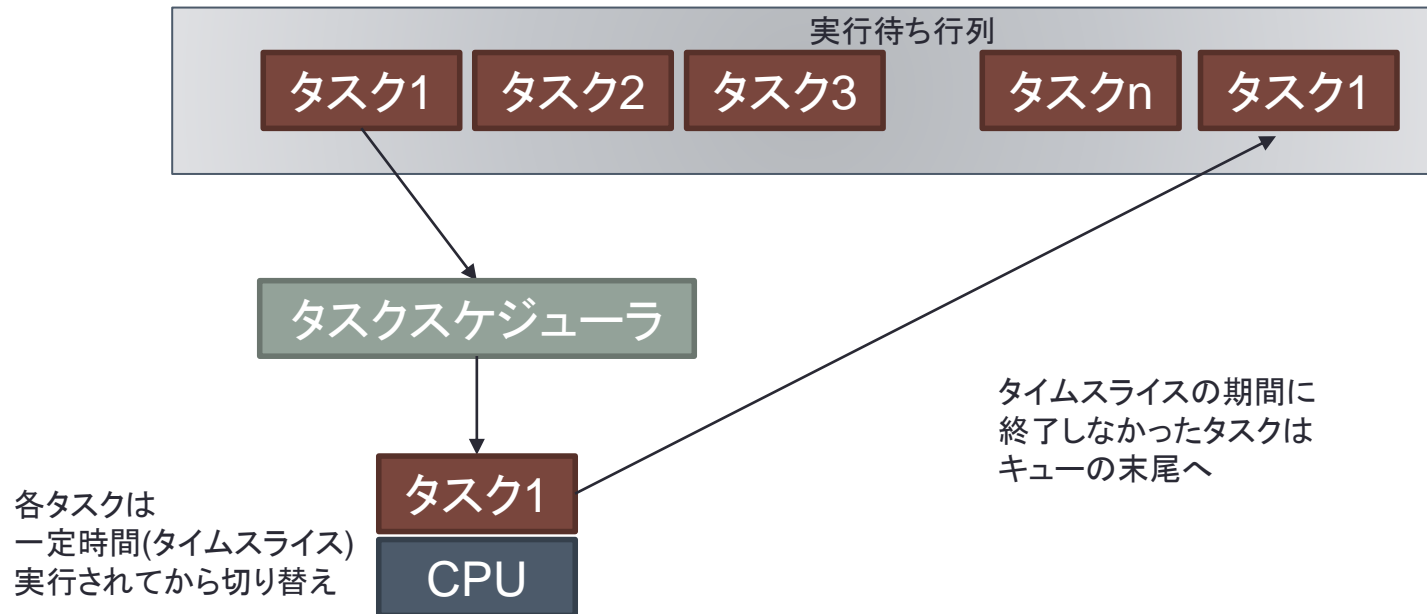
OSはCPUの数だけ並列処理させ、
他のプロセスは並行処理で見せかけの同時実行を行う
→では、どのタイミングで処理の切り替えを行う？

タスクスケジューリング

行うべきこと

「どのタスク(プロセス)」に「どのぐらいの時間実行」させて切り替えるか

最も原始的な方法: ラウンドロビン



ラウンドロビンの利点・欠点

- 利点

- 各タスクに均等の実行時間を与えることが可能
- 実装が簡単
- 各タスクが同じような粒度であれば最適

- 欠点

- 演算タスク
 - タイムスライス中ほぼCPUの処理を必要とする
- 対話型タスク:
 - 入力待ちの時間が発生する
 - 入力された際の反応速度が重要
- 同じタイムスライスでは不公平であり非効率
- タスクのスケジューリングには不向き

Linuxカーネルのタスクスケジューラ(1991~2003年)

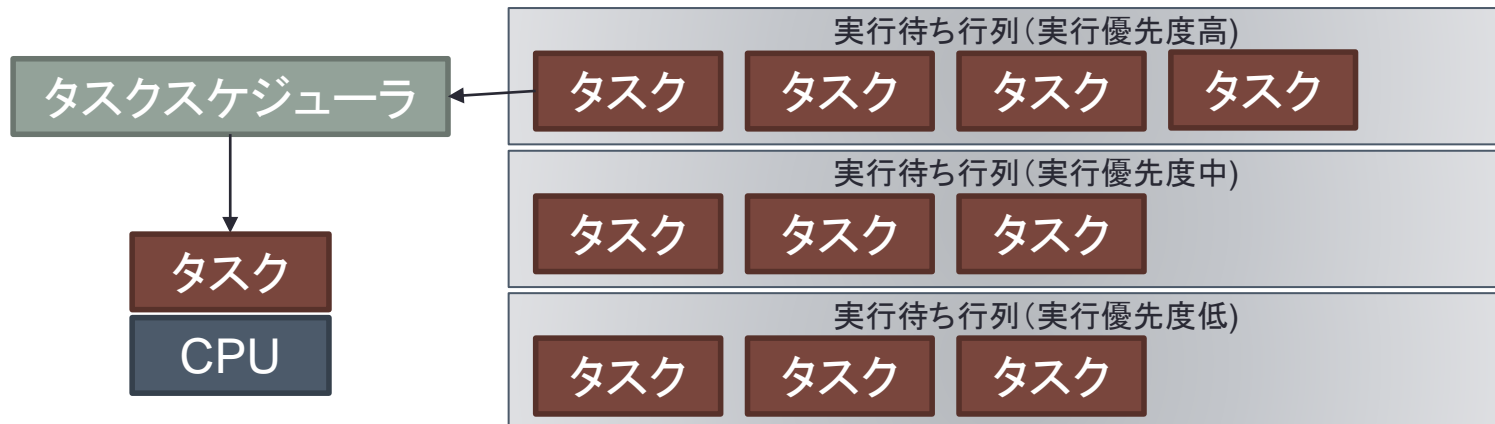
- nice値によるスケジューリング方式
 - nice値: タスクの実行優先度(-20~19) 小さい方が優先度が高い
 - 可変長タイムスライスを各タスクに持たせる
 - キューの中から**タイムスライスが最も多いタスクを選択**して実行する
 - 実行される時間は固定(tickの周期) tick:定期的に発生させるタイマ
 - 実行されたタスク(未終了タスク)は、以下の処理をしてキューに戻す
 - 実行された時間だけタイムスlicesを減らす
 - 残り時間の1/2とnice値 $\times (-1)$ の値を残りタイムスライスに加算
- 評価
 - 対話型タスクなどはタイムスタンプが長くなり、応答性が高くなる
 - 実行するタスクが少ない場合はうまくいく
 - タスク数が増えると、次の実行タスクを選ぶ処理、タイムスlicesの再計算処理などのオーバーヘッドが無視できなくなる

どちらもキュー全体を線形探索

Linuxカーネルのタスクスケジューラ(2003~2007年)

O(1) order one スケジューラ

- Linuxカーネル2.6.0(2003) – 2.6.22(2007)
- アルゴリズム
 - 実行優先度別のキューを複数作る
 - 実行タスクは**実行優先度の高いキューから選ばれる**(先頭から)
 - 実行時間は固定時間(tick)
 - 実行優先度は, nice値とタスクのスリープ時間で算出する



キュー(待ち行列)の中のタスクがなくなったら次のキューが最優先キューになる

Linuxカーネルのタスクスケジューラ(2007年~)

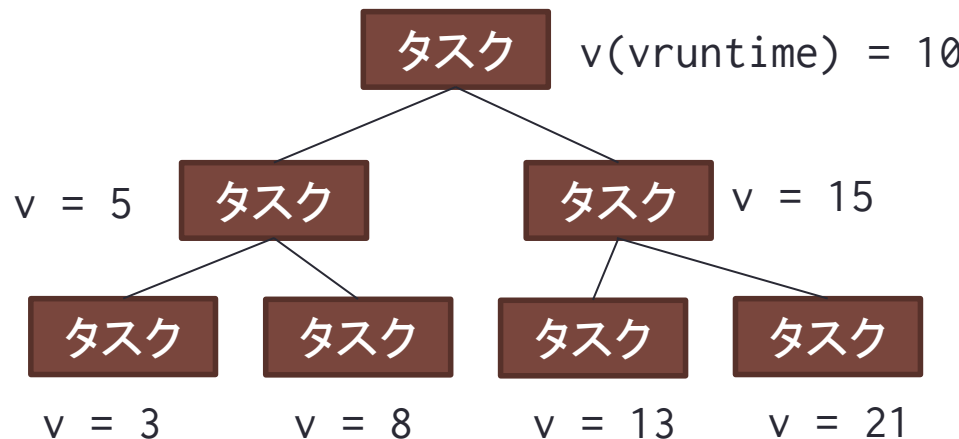
CFS(Completely Fair Scheduler)

- 各タスクがCPUで実行される時間を均一にする
- 実行タスクはCPU実行時間の少ないタスクが優先される
- CPU実行時間
 - 実際にCPUの実行に使った時間(≠CPUに割り当てられている時間)
 - 対話型などのスリープの多いタスクはあまり増加しない
 - 各タスクはvruntime(virtual run time)という実行時間総計の変数を持つ
 - 厳密にはvruntimeの値はnice値によって増減の度合いも異なる(nice値が低いほど増加しにくい)
 - キューではなくvruntimeをキーとした赤黒木(二分探索木の種類)で管理する
- タイムスライス(CPUに割り当てられる時間)
 - $\text{タイムスライス} = \text{最低実行時間(デフォルト7ms)} \times \text{タスクの重み}(1.25^{\text{nice}*(-1)}) \div \text{タスクの重みの合計}$
 - niceが低い(優先度高)ほど、総タスク数が少ないほど タイムスライスが増える

Linuxカーネルのタスクスケジューラ(2007年~) 続き

CFS(続き)

- 赤黒木
 - 二分探索木の種類
 - 最長のパスの長さが最短のパスの長さの2倍以内に収まる平衡木



- タスク追加・削除・実行タスク選択等にかかる時間は $O(\log n)$
(線形リストの場合, 実行タスク選択は1回で済むが, タスク追加で $O(n)$ かかる)

vruntimeの確認

```
tusedls13$ ps
  PID TTY          TIME CMD
11859 pts/1    00:00:00 bash
18426 pts/1    00:00:00 ps
tusedls13$ more /proc/11859/sched
bash (11859, #threads: 1)
-----
se.exec_start          : 26428013.474624
se.vruntime             : 26786.511689
se.sum_exec_runtime    : 379.944208
se.nr_migrations       : 41
nr_switches            : 1864
nr_voluntary_switches  : 1858
nr_involuntary_switches : 6
se.load.weight         : 1024
policy                 : 0
prio                   : 120
clock-delta            : 58
mm->numa_scan_seq      : 0
numa_migrations, 0
numa_faults_memory, 0, 0, 1, 0, -1
numa_faults_memory, 1, 0, 0, 0, -1
tusedls13$
```

プロセスID(PID)

vruntimeの値

他のスケジューラアルゴリズム

- EDF(Earliest Deadline First)
 - 〆切の近いタスクからやる 人間っぽい
 - CPU使用率100%になるとどれも間に合わない現象
- 多段フィードバックキュー(Multilevel Feedback Queue)
 - 複数のFIFOキューで構成されており, 先頭のFIFOが最も優先される

OSによるスケジューラの違い

- Linux
 - 多段フィードバックキュー
 - $O(1)$
 - CFS
- Windows
 - 多段フィードバックキュー (Windows Vista以降)
- MacOS
 - 多段フィードバックキュー

プロセスの親子関係

- プロセスは別の他のプロセス(親プロセス)から起動される
- すべてのプロセスの元のプロセスをinit プロセス(PID1 systemd)という
 - システム起動時にinitプログラムによって生成される
 - カーネルがディスクから読みだされ、システムを初期化し終わった直後に起動する

```
tusedls13$ pstree
systemd
├── ModemManager──2*[{ModemManager}]
├── NetworkManager──2*[{NetworkManager}]
├── VGAuthService
├── abrt-dbus──2*[{abrt-dbus}]
├── accounts-daemon──2*[{accounts-daemon}]
├── at-spi-bus-laun└──dbus-daemon
│   └──3*[{at-spi-bus-laun}]
├── atd
├── auditd└──audispd└──sedispatch
│   └──{auditd}└──{audispd}
├── automount──3*[{automount}]
├── avahi-daemon──3*[{avahi-daemon}]
├── chronyd
├── colord──2*[{colord}]
├── crond
├── cupsd
├── 2*[{dbus-daemon}]
├── dbus-launch
├── firewalld──{firewalld}
├── gssproxy──5*[{gssproxy}]
├── irqbalance
├── ksmctuned──sleep
├── libvirt└──16*[{libvirt}]
├── lightdm└──X──11*[{X}]
│   ├──lightdm└──lightdm-gtk-gre──11*[{lightdm-gtk-gre}]
│   └──lightdm──2*[{lightdm}]
├── lsm
├── lvm
├── master└──pickup
│   └──qmgr
├── mcelog
├── nscd──11*[{nscd}]
├── nslcd──5*[{nslcd}]
├── packagekitd──2*[{packagekitd}]
├── polkitd──6*[{polkitd}]
├── rngd
├── rpc.statd
├── rpcbind
├── rsyslogd──2*[{rsyslogd}]
├── rtkit-daemon──2*[{rtkit-daemon}]
├── smartd
├── sshd└──3*[{sshd}]
│   ├──sshd└──sshd──sfio-server
│   ├──sshd└──sshd──h
│   └──sshd
└── systemd-journal
```

pstreeコマンドで親子関係を確認

bashからpstreeが起動

プログラムで実験

```
tusedls04$ cat fork.c
#include <stdio.h>

int main(){
    long pid;
    pid = fork();

    printf("Hello %d\n", pid);
}
tusedls04$ gcc fork.c -o fork
tusedls04$ ./fork
Hello 8161
Hello 0
tusedls04$
```

表示順番が入れ替わる
こともある

- fork()でプロセス生成
fork()の返回值pidには
- 親プロセスには子プロセスのPID
 - 子プロセスは0 (-1)
 - -1の場合はfork()失敗

fork()の返回值の型は厳密にはpid_t型(types.h)

```
tusedls04$ cat fork.c
#include <stdio.h>

int main(){
    long pid;
    pid = fork();

    printf("Hello %d %d %d\n",
           pid, getpid(), getppid());
}
tusedls04$ gcc fork.c -o fork
tusedls04$ ./fork
Hello 8316 8315 7588
Hello 0 8316 8315
tusedls04$ ps
  PID TTY          TIME CMD
 7588 pts/0    00:00:00 bash
 8338 pts/0    00:00:00 ps
tusedls04$
```

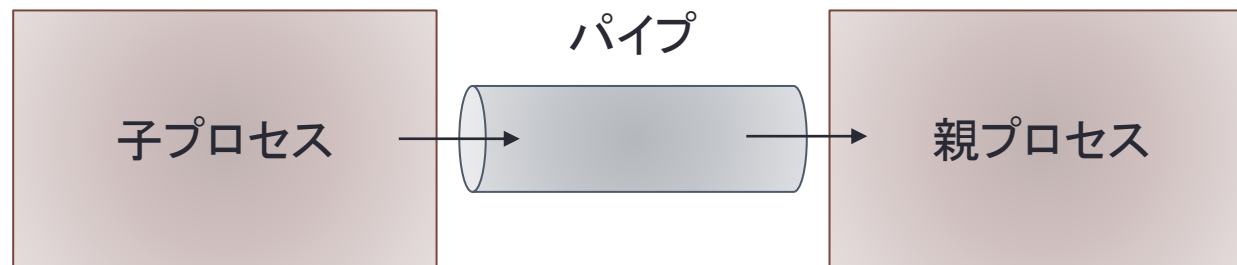
getpid()でプロセス番号を得る
getppid()で親プロセス番号を得る

この例では

親(PID 8315)プロセスが子プロセス(PID 8316)を生成
親(PID 8315)プロセスの親はbash(PID 7588)

プロセス間通信

- プロセスは独立したメモリ空間を割り当てられるため、値の共有ができない
- プロセス間の通信をサポートするパイプ(pipe)を使用すると送受信が可能
- パイプは一方方向（二つ作成して双方向の送受信は可能）



パイプの実験

```
tusedls04$ cat pipe.c
#include <stdio.h>
#include <string.h>

int main(){
    long pid;
    char buf[256];
    char str[256] = {"Hello"};
    int pi[2];

    pipe(pi); パイプ作成
    pid = fork();
    if(pid == 0) { //Child
        printf("Child(PID%d) to Parent(PID%d) > ", getpid(), getppid());
        scanf("%s", buf);
        write(pi[1], buf, strlen(buf)+1); パイプに送信
        strcpy(str, buf);
        printf("str=%s\n", str);
        close(pi[1]);
    } else { //Parent
        read(pi[0], buf, 256); パイプから受信
        printf("Parent(PID%d) : %s\n", getpid(), buf);
        printf("str=%s\n", str);
        close(pi[0]);
    }
}

tusedls04$ gcc pipe.c -o pipe
tusedls04$ ./pipe
Child(PID9836) to Parent(PID9835) > abcdefg 標準入力へ
str=abcdefg
Parent(PID9835) : abcdefg
str=Hello
tusedls04$
```

子プロセスのstrは変わっているが (メモリ空間が独立のため)
親プロセスのstrは変わっていない

まとめ

- プロセスはカーネルの機能の一つである
- プログラムの実行単位であり, プログラムが計算機資源を占有しているように見せかける
- プロセスへの割り当てメモリは独立(パイプで共有は可能)
- カーネルはプロセス単位で並行処理を行う
- 並行処理を行うための実行切り替えのアルゴリズムが複数ある
 - ラウンドロビン
 - nice値によるスケジューリング
 - $O(1)$ スケジューリング
 - CFS(Completely Fair Scheduler)
- プロセスは親子関係がある

質問あればどうぞ