

# プロセス

田浦健次郎

# 用語の整理

- ▶ プログラム = 実行すべき命令が書かれているもの
  - ▶ Firefox, シェル (bash), ls, a.out, など
  - ▶ 実体としてはファイルとして存在している
- ▶ プロセス  $\approx$  プログラムが走っているもの
  - ▶ メニューでアプリを起動するアイコンをクリックしたり, コマンドプロンプトに ls と打ち込むたびに, プロセスが作られている
- ▶ たとえ話
  - ▶ プログラム  $\approx$  マニュアル
  - ▶ プロセス  $\approx$  マニュアルに従って働いている人

# プロセスの役割

- ▶ CPU を分け合うための抽象化
  - ▶ ユーザはプロセスを作る
  - ▶ 各プロセスは全力で走れば良い (他の人に CPU を譲る必要ない)
  - ▶ OS がプロセスに CPU を与えたり奪ったりする
- ▶ メモリを分け合うための抽象化 (アドレス空間)
  - ▶ 他のプロセスのメモリは覗けない, 壊せない

いずれも仕組みは後の週で

# プロセスを観察するコマンド

- ▶ Unix CUI
  - ▶ `ps` : 基本
  - ▶ `top` : 時々刻々表示
  - ▶ `pstree` : プロセスの親子関係も表示
  - ▶ `pgrep` : 色々な基準でプロセスを検索 ( $\approx$  `ps` + `grep`)
- ▶ Linux
  - ▶ `/proc/pid`
- ▶ Ubuntu GUI
  - ▶ システムモニタ
- ▶ Windows GUI
  - ▶ タスクマネージャ
  - ▶ リソースマネージャ

注:

- ▶ CUI (Character User Interface) 端末の中で字だけ出す
- ▶ GUI (Graphical User Interface) 窓を出して絵を出す

# ps auxww

- ▶ すべてのプロセスをコマンドライン含め表示

```
1 $ ps auxww
```

- ▶ ps と grep を組み合わせてトラブルシューティングする場面がよくある

```
1 $ ps auxww | grep ssh # ssh 走ってるか?  
2 $ ps auxww | grep tau # ユーザtau のプロセス
```

- ▶ `pgrep`  $\approx$  `ps + grep`

```
1 $ pgrep -f ssh  
2 $ pgrep -f tau
```

表記上の約束:

- ▶ \$はコマンドプロンプトのつもり
- ▶ \$以降 (下線) が入力すべきもの

# プロセス ID (PID)

- ▶ 存在しているプロセス全てに付けられている一意な識別子
- ▶ Linux では通常, 4194304 までの整数
- ▶ あるプロセスの PID を知らなくてはいけない場面はそう多くはないが, 外から強制終了 (kill) したい場合などに必要

# man コマンド

- ▶ 以降, コマンドや関数, システムコールの説明が出てきたら, 必要に応じて man コマンドで調べよ
- ▶ 例:

```
1 $ man ps  
2 $ man top  
3 $ man pstree
```

# Unix: プロセス関連のシステムコール

- ▶ `fork`

- ▶ プロセスを作る (コピーする)

- ▶ `execve`

- ▶ 現プロセスで指定のプログラムを実行する
  - ▶ 変種: `exec{v,l}p?e?` (引数の渡し方, 微妙な意味の違い)
  - ▶ 以下, 総称して `exec` と呼ぶ (実際には `exec` という名前の関数はない)

- ▶ `exit`

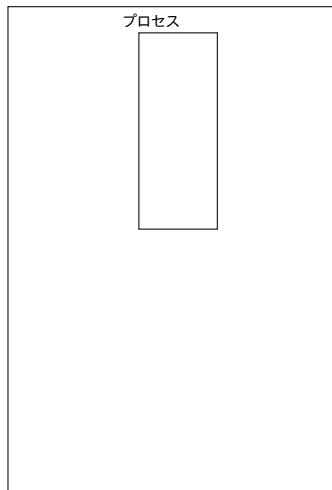
- ▶ 現プロセスを終了する
  - ▶ `_exit`

- ▶ `waitpid`

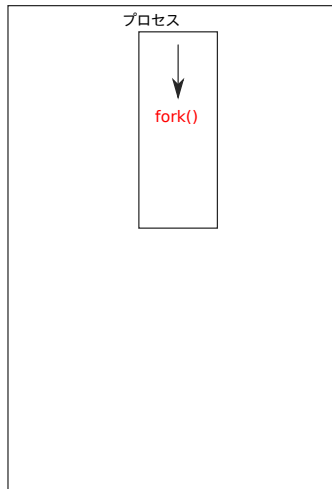
- ▶ 子プロセスの終了待ち + 処理
  - ▶ 変種: `wait`, `wait3`, `wait4`



# 子プロセスの生成～終了～処理

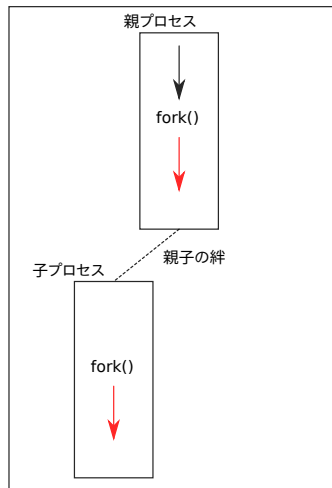


# 子プロセスの生成～終了～処理



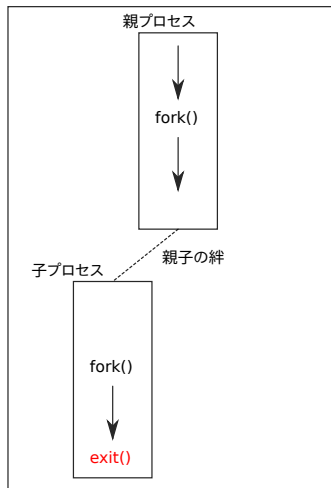
# 子プロセスの生成～終了～処理

1. fork ～ プロセスが複製される．親と子が両方、fork の続きを実行



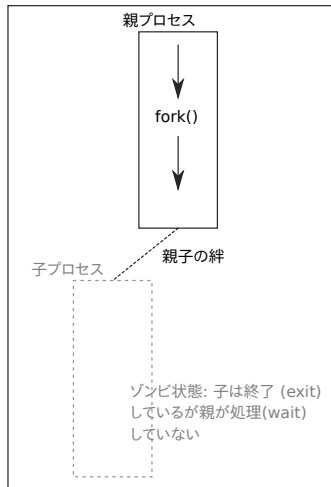
# 子プロセスの生成～終了～処理

1. fork ～ プロセスが複製される. 親と子が両方, fork の続きを実行
2. 子プロセスが終了する (exit を呼ぶ, main 関数が return するなど)



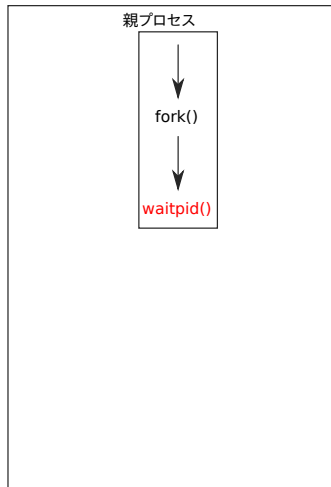
# 子プロセスの生成～終了～処理

1. fork ～ プロセスが複製される. 親と子が両方, fork の続きを実行
2. 子プロセスが終了する (exit を呼ぶ, main 関数が return するなど)
3. 親が処理する (wait, waitpid などと呼ぶ) まで, 子プロセスは「ゾンビ (プロセス番号だけが存在する) 状態」



# 子プロセスの生成～終了～処理

1. fork ～ プロセスが複製される. 親と子が両方, fork の続きを実行
2. 子プロセスが終了する (exit を呼ぶ, main 関数が return するなど)
3. 親が処理する (wait, waitpid など呼ぶ) ま  
で, 子プロセスは「ゾン  
ビ (プロセス番号だけが  
存在する) 状態」
4. 親が処理を終えるとすべ  
てがなくなる



# fork

- ▶ 呼び出したプロセスを複製
- ▶ `fork()` 続きが2 プロセス (親と子) で実行される
- ▶ 親と子で返り値だけが違う
  - ▶ 親: 子プロセスのプロセス番号
  - ▶ 子: 0
- ▶ 従って以下がテンプレート

```
1 pid_t pid = fork();
2 if (pid == -1) {
3     失敗 (子プロセスは作られていない)
4 } else if (pid == 0) {          /* child */
5     子プロセス
6 } else {                       /* child */
7     親プロセス
8 }
```

- ▶ 注: `pid_t` はプロセス番号 (process ID) の型; 実際は単なる整数

# exit

- ▶ `exit` を呼んだプロセスを, 指定した終了ステータス (`exit status`) で終了させる
- ▶ `exit status` : 0 .. 255 の整数

```
1 exit(status);  
2 ...
```

当然ながら `exit` 呼び出し以降 (上記の...) は実行されない

- ▶ `main` 関数が終了した場合も同じ効果 (従って `main` 関数の最後にわざわざ呼ばないのが普通)
  - ▶ `main` の返り値 (return value) が `exit status`
- ▶ `exit status` は親プロセスが取得可能



# waitpid

- ▶ 基本は子プロセスの終了待ち + 処理 (ゾンビ状態を解消; プロセス番号の回収)
- ▶ どの子プロセス (特定プロセス, どれでもよい, など) を対象とするか, 終了を待つ・待たない, などを指定可

```
1  int ws;  
2  pid_t pid = -1; /* -1 : どの子プロセスでも... */  
3  int options = 0; /* 0 : 終了するまで待つ */  
4  pid_t cid = waitpid(pid, &ws, options);  
5  if (cid == -1) {  
6      失敗;  
7  } else {  
8      ... ws に, 子プロセスcid に何が起きたかの情報 ...  
9  }
```

- ▶ 普通の終了: exit を呼んだ / main が return した
- ▶ 異常終了: segmentation fault など. 詳細は後の週で
- ▶ 停止, 再開: 詳細略
- ▶ 詳細は (これからいつも)man を参照

```
1  $ man waitpid
```

# man コマンドの落とし穴

- ▶ man コマンドでは、色々なものが検索できる
  - ▶ コマンド, システムコール, それ以外の関数 (ライブラリ) など
- ▶ 同じ名前の, コマンドと関数がある場合, 目当てでないほうが見つかってしまう場合がある
- ▶ システムコールだと思ったものがライブラリ関数ということもよくある
- ▶ `-s` オプションで, 種類 (正確にはマニュアルの「節」) を指定可能
  - ▶ `-s 1` : コマンド
  - ▶ `-s 2` : システムコール
  - ▶ `-s 3` : ライブラリ関数
- ▶ 例:

```
1 $ man -s 1 wait      # wait コマンド
2 $ man -s 2 wait      # wait システムコール
3 $ man -s 2 execv     # 見つからない
4 $ man -s 3 execv
```

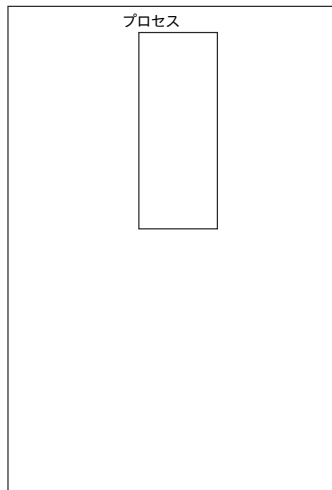
# ゾンビ (defunct)

- ▶ プロセス  $C$  がゾンビ (defunct)  $\equiv C$  が終了しているが、その親が (waitpid など)  $C$  の終了を確認していない
- ▶ 本質的には、 $C$  のプロセス番号を再利用できない状態
  - ▶ waitpid が「 $C$  が終了した」と親に知らせるまでは、 $C$  のプロセス番号を他のプロセスに再利用すると、プロセス番号からプロセスを一意に特定できなくなるので
- ▶ waitpid  $\approx$  お葬式; 子プロセスに「成仏」「輪廻転生」してもらおう
- ▶ 英語では、“the parent *reaps* the child” という

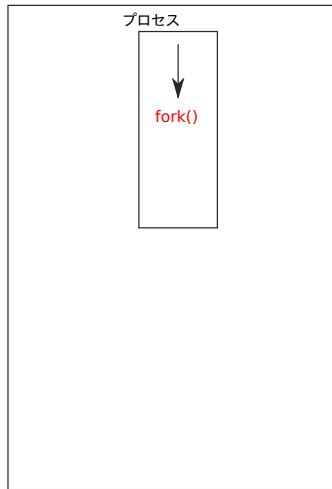
## 細かい注 waitpid, ゾンビにまつわる注

- ▶ Q. 子が終了する前に親が waitpid 等を呼んだら?
  - ▶ A. 子が終了するまで return しない (wait という名の通り)
- ▶ Q. 子が終了する前に親が終了できる?
  - ▶ A. できる
- ▶ Q. その場合, 誰がその子の葬式をする (その子はゾンビ状態のまま)?
  - ▶ A. あるプロセス  $C$  の親が,  $C$  より先に終了したら, 全プロセスの先祖 (init) が  $C$  の親をすることになっている

# 子プロセスの生成～exec

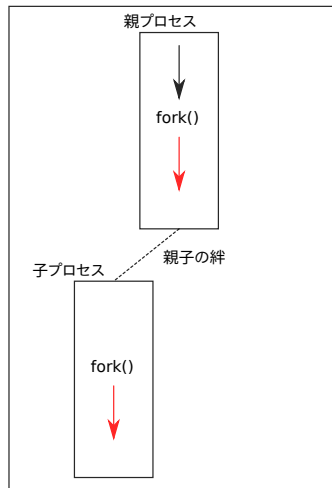


# 子プロセスの生成～exec



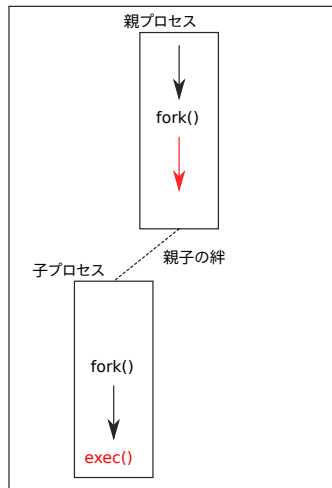
# 子プロセスの生成～exec

1. fork ～ プロセスが複製される．親と子が両方，fork の続きを実行



# 子プロセスの生成～exec

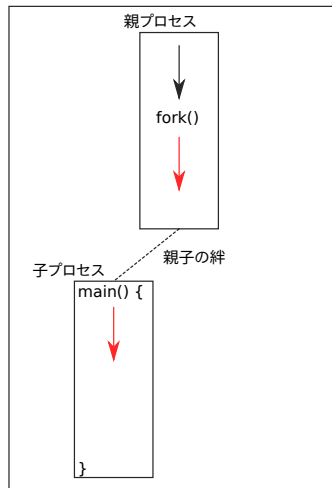
1. fork ～ プロセスが複製される．親と子が両方，fork の続きを実行
2. 子プロセスが exec を実行





# 子プロセスの生成～exec

1. fork ～ プロセスが複製される．親と子が両方，fork の続きを実行
2. 子プロセスが exec を実行



- ▶ 現プロセスで、指定したプログラムを実行する
- ▶ 以下は/bin/ls (いわゆる ls コマンド) を-l オプションで実行

```
1 char * const argv[] = { "/bin/ls", "-l", 0 };  
2 execv(argv[0], argv);  
3 ...
```

- ▶ 注:
  - ▶ exec を呼び出した続き (上記...以降) は実行され~~ない~~
  - ▶ 呼び出したプロセスが、今持っている状態をすべて忘れ、指定したプログラムを実行するだけの人に化~~ける~~
  - ▶ 特に、exec が子プロセスを作るわけではない

## (細かい注) exec の変種

- ▶ `exec{v,l}p?e?`
- ▶ ただし `execlpe` は存在しない
- ▶ つまり `execv`, `execve`, `execvp`, `execvpe`, `execl`, `execle`, `execlp`
- ▶ `execve` だけがシステムコール, 残りはその亜種
- ▶ `v` と `l` : 引数の渡し方 (`v` : 配列; `l` : 引数のリスト)
- ▶ `p` : 環境変数 `PATH` を参照してコマンドを検索する

```
1  execv("ls", argv);      // NG
2  execv("/bin/ls", argv); // OK
3  execvp("ls", argv);     // OK
4
```

- ▶ `e` : 子プロセスの環境変数を指定する (ない場合は親を引き継ぐ)