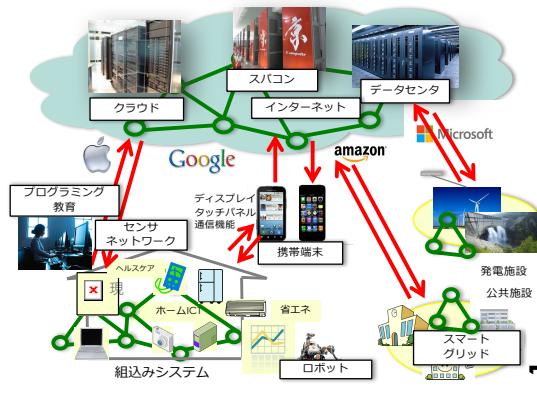


システムプログラミング プロセス2

芝浦工業大学
情報工学科
菅谷みどり

システムプログラミング



プログラム例 (execve)



```
#include <stdio.h>
2
3 extern char **environ;
4
5 main()
6 {
7     char *argv[2];
8
9     argv[0] = "/bin/ls";
10    argv[1] = NULL;
11
12    execve(argv[0], argv, environ);
13 }
```

execve は、プロセスの実行を上書きするシステムコールである

- `/bin/ls` のプログラムがロードされ、`ls` の `main` から実行が開始される (`/bin/ls` を実行するプログラムとなる)
- 引数、環境変数はそれぞれ `argv` 形式で渡す必要がある
- 環境変数は現在の環境変数そのまま、`argv` は `NULL` を渡している

システムプログラミング講義

27

Copy-on-write

- ・ コンピュータプログラミングにおける最適化戦略の一種類

システムプログラミング

22

プログラム例2 (execve) 引数



```
#include <stdio.h>
2
3 extern char **environ;
4
5 main()
6 {
7     char *argv[3];
8
9     argv[0] = "/bin/ls";
10    argv[1] = "/";
11    argv[2] = NULL;
13
14    execve(argv[0], argv, environ);
15 }
```

実行結果

```
% ./a.out
bin dev home lhome lost+found mnt proc sbin usr work boot etc initrd lib
misc opt root tmp var
%
システムプログラミング講義
```

28

演習

- 子プロセスに別のプログラム（コマンド）を実行させるように変更してみよう



組込みシステム講義

29

Deep Understanding

- https://qiita.com/tajima_taso/items/d3c8d516fe8995613c67



システムプログラミング

33

プログラム実行のためのライブラリ関数

・ プログラムの実行

- execve は引数や環境変数を argv 形式で渡す必要がある
- プログラムをサーチしてくれないため、もう少し簡単に使えるように以下のライブラリ関数が用意されている

```
int execl(const char *path, const char *arg, ...);  
int execlp(const char *file, const char *arg, ...);  
int execle(const char *path, const char *arg, ..., char * const envp[]);  
int execv(const char *path, char *const argv[]);  
int execvp(const char *file, char *const argv[]);
```

・ execl, execlp, execle

- コマンド引数の渡し方が execve とは異なり、これらのライブラリ関数の引数として、コマンド引数のように文字列を並べて渡す
- execl("/bin/ls", "ls", "/", NULL);
- execlp("ls", "ls", "/", NULL);

組込みシステム講義

31



プロセスで学んだこと

- Fork()というシステムコールで新しいプロセス（処理単位）を作成することができる
- 一つのプロセスが子プロセスを無限（資源量の上限あり）に作成することができる
 - CPUが一つにもかかわらず、プロセスの生成は無限
 - 仮想化
- V1 で学んでないこと
 - OS 側での仕事
 - プロセスのスケジューリング、割り込み、シグナル制御、優先度の制御、プロセスの切り替え、状態管理、メモリ管理
 - プロセス同士のデータの受け渡し

システムプログラミング

34

プロセスに関するOS処理

システムプログラミング

35



Address space of the child

- Two possibilities for the address space of the child relative to the parent:
 - The child may be **an exact duplicate of the parent**, sharing the same program and data segments in memory.
 - Each will have their own PCB, including program counter, registers, and PID. This is the behavior of the fork system call in UNIX.
- The child process may have a new program**
 - loaded into its address space, with all new code and data segments. This is the behavior of the spawn system calls in Windows. **UNIX systems implement this as a second step, using the exec(v) system call.**

2019/11/8

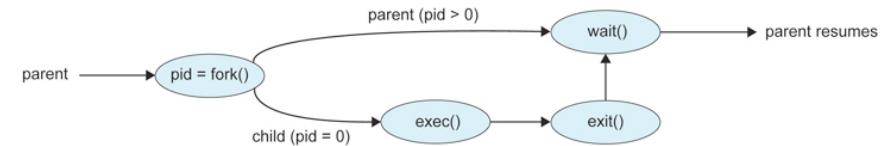
38



Visual Sequence of a process creation

Fork system call

- Returns the PID of the process child to each process
 - 0 to child process
 - Non-zero child PID to the parent
- return value indicates which process is which.

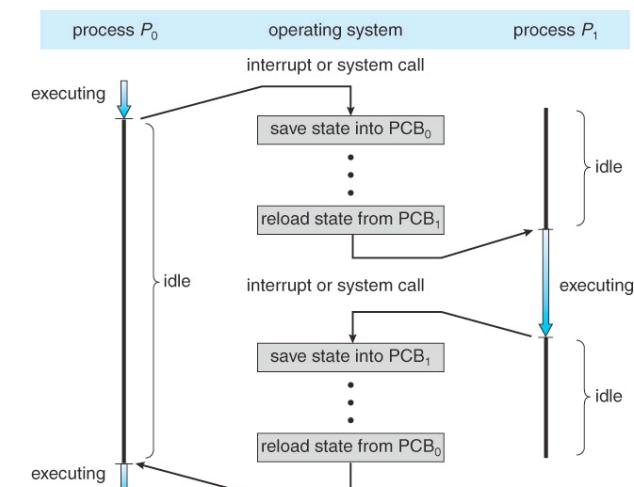


2019/11/8

36



Diagram showing CPU switch from process to process



2019/11/8

39



For Multi-Tasking

- Is it sufficient for achieving multi-tasking OS
 - That provides the mechanism to switching process mechanism?

2019/11/8



40

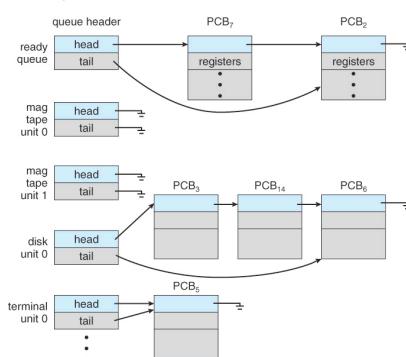


41

Scheduling Queues

- All processes are stored in the job queue
 - Processes in the Ready state are placed in the ready queue.
 - Processes waiting for a device to become available or to deliver data are placed in device queues. There is generally a separate device queue for each device.
 - Other queues may also be created and used as needed.

2019/11/8



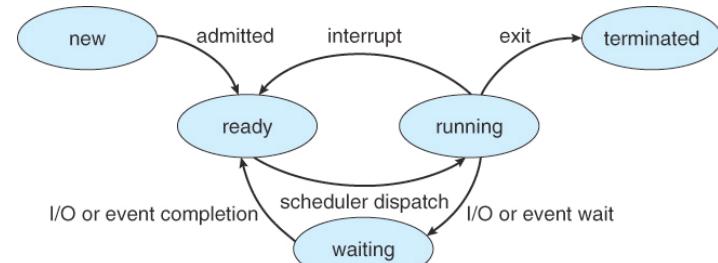
43



44

Process State

- Process may be in one of 5 states
 - **New** - The process is in the stage of being created.
 - **Ready** - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
 - **Running** - The CPU is working on this process's instructions.
 - **Waiting** - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur
 - **Terminated** - The process has completed.



2019/11/8

Process Scheduling

- **The two main objectives of the process scheduling system**
 - are to keep the CPU busy at all times
 - to deliver "acceptable" response times for all programs, particularly for interactive ones.
- **The process scheduler must meet these objectives**
 - by implementing suitable policies for swapping processes in and out of the CPU.

2019/11/8

Schedulers

- **A long-term scheduler**
 - is typical of a batch system or a very heavily loaded system. It runs infrequently, (such as when one process ends selecting one more to be loaded in from disk in its place), and can afford to take the time to implement intelligent and advanced scheduling algorithms.
- **The short-term scheduler, or CPU Scheduler,**
 - runs very frequently, on the order of 100 milliseconds, and must very quickly swap one process out of the CPU and swap in another one.
- **Some systems also employ a medium-term scheduler.**
 - When system loads get high, this scheduler will swap one or more processes out of the ready queue system for a few seconds, in order to allow smaller faster jobs to finish up quickly and clear the system.
 - An efficient scheduling system will select a good process mix of CPU-bound processes and I/O bound processes.

2019/11/8

45



Context Switch

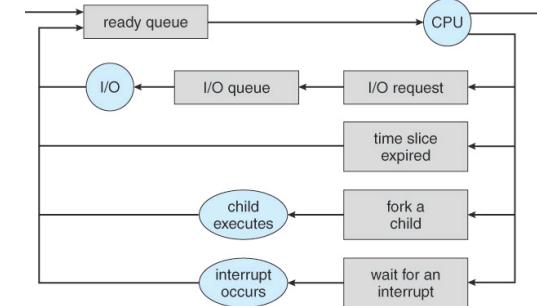
- **Whenever an interrupt arrives,**
 - the CPU must do a state-save of the currently running process, then switch into kernel mode to handle the interrupt, and then do a state-restore of the interrupted process.
- **A context switch occurs**
 - when the time slice for one process has expired and a new process is to be loaded from the ready queue.
 - This will be caused by a timer interrupt, which will then cause the current process's state to be saved and the new process's state to be restored.
 - Saving and restoring states involves saving and restoring all of the registers and program counter(s),
 - as well as the process control blocks described above.
- **Context switching happens**
 - VERY VERY frequently, and the overhead of doing the switching is just lost CPU time, so context switches (state saves & restores) need to be as fast as possible. Some hardware has special provisions for speeding this up, such as a single machine instruction for saving or restoring all registers at once.

2019/11/8

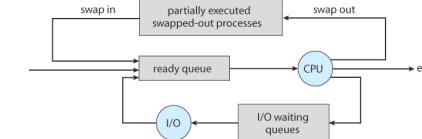
47



Queuing-diagram representation of process scheduling



- **Addition of a medium-term scheduling to the queuing diagram**



2019/11/8

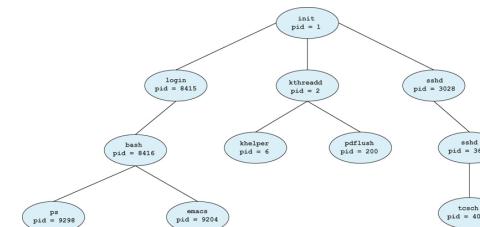
46



Operation on Processes

Process Creation

- Processes may create other processes through appropriate system calls, such as **fork** or **spawn**.
- The process which does the creating is termed the **parent** of the other process, which is termed its **child**. Each process is given an integer identifier, termed its **process identifier**, or PID. The parent PID (PPID) is also stored for each process.



2019/11/8

ps –el will list complete information for all processes

49



プロセスとスレッドの目的

CPUの仮想化

- 物理的なCPU数は固定、少數
 - ラップトップ、スマート
 - サーバー
- CPUは1つか2つにもかかわらず、数十、数百のプログラムを立ち上げることができる
- 個々のプログラムを書く人が明示的な「譲り合い」をする必要がない

システムプログラミング



50



Process Creation in Windows

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    // allocate memory
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // create child process
    if (!CreateProcess(NULL, // use command line
                      "C:\\WINDOWS\\system32\\mspaint.exe", // command line
                      NULL, // don't inherit process handle
                      NULL, // don't inherit thread handle
                      FALSE, // disable handle inheritance
                      0, // no creation flags
                      NULL, // use parent's environment block
                      NULL, // use parent's existing directory
                      &si,
                      &pi))
    {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }
    // parent will wait for the child to complete
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    // close handles
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

2019/11/8

Figure 3.12 Creating a separate process using the Win32 API.

51

Process Termination

- Processes may request their own termination**
 - by making the `exit()` system call, typically returning an int.
- child code:**
 - `int exitCode; exit(exitCode);` // return `exitCode`; has the same effect when executed from `main()`
- parent code:**
 - `pid_t pid; int status pid = wait(&status);`
 - // `pid` indicates which child exited.
 - // `exitCode` in low-order bits of `status`
 - // macros can test the high-order bits of `status` for why it stopped
- Processes may also be terminated by the system for a variety of reasons, including:**
 - The inability of the system to deliver necessary system resources.
 - In response to a KILL command, or other unhandled process interrupt.
 - A parent may kill its children if the task assigned to them is no longer needed.

2019/11/8



52



Inter-process Communication

Independent Processes

- operating concurrently on a systems are those that can neither affect other processes or be affected by other processes.

Cooperating Processes

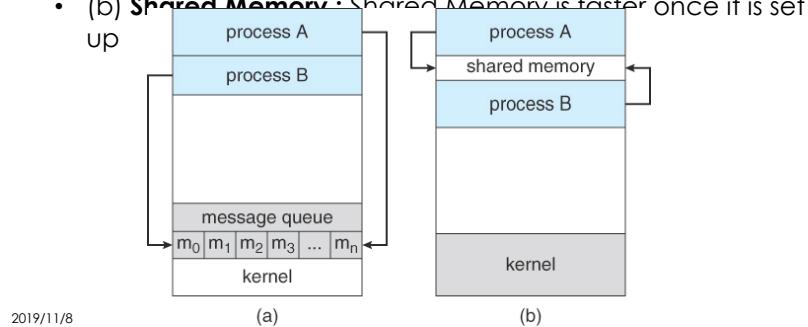
- are those that can affect or be affected by other processes. There are several reasons why cooperating processes are allowed:
 - Information Sharing
 - Computation speedup
 - Modularity
 - Convenience

2019/11/8

53

Two Communication models:

- Cooperating process require some type of inter-process communication, which is most commonly one of two types:
 - (a) **Message passing** : Message Passing requires system calls for every message transfer, and is therefore slower, but it is simpler to set up and works well across multiple computers.
 - (b) **Shared Memory** : Shared Memory is faster once it is set up



2019/11/8

(a)

(b)

54

Example

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main (void)
{
    int segment_id; /* identifier */
    char *shared_memory; /* pointer */
    const int size = 4096; /* size */

    segment_id =
shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

2019/11/8

```
/* attach */
shared_memory = (char *) shmat
(segment_id, NULL, 0);

/* write a message */
sprintf(shared_memory, "Hi there!");

/* print out the string */
printf("%s\n", shared_memory);

/* detach */
shmdt(shared_memory);

/* remove */
shmctl(segment_id, IPC_RMID,
NULL);

return 0;
```

56

Shared-Memory Systems

- In general the memory to be shared in a shared-memory system
 - is initially within the address space of a particular process, which needs to make system calls in order to make that memory publicly available to one or more other processes.
- Other processes
 - which wish to use the shared memory must then make their own system calls to attach the shared memory area onto their address space.
- Generally a few messages
 - must be passed back and forth between the cooperating processes first in order to set up and coordinate the shared memory access.

2019/11/8

55

Message-Passing Systems

- Message passing systems must support at a minimum system calls for "send message" and "receive message".
- A communication link must be established between the cooperating processes before messages can be sent.
- There are three key issues to be resolved in message passing systems as further explored in the next three subsections:
 - Direct or indirect communication (naming)
 - Synchronous or asynchronous communication
 - Automatic or explicit buffering.

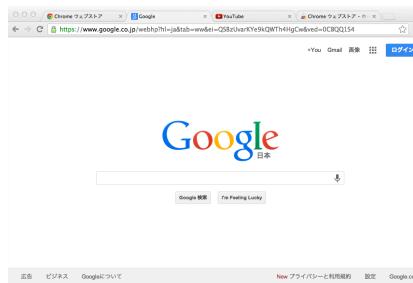
2019/11/8

59

(Coffee break)

Multiprocess architecture –Chrome browser

- Most contemporary web browser provide tabbed browsing
 - Allows a single instance of a web browser application to open several websites at the same time, with each site in a separate tab.
 - To switch between the different sites, a user need only click on the appropriate tab.



2019/11/8

KonokaScript



60

Summary

In this session, we learn

- The concept of process – a program in execution, which forms the basis of all computation.
- The various features of processes, including scheduling, creation and termination, and communication.

2019/11/8

62

Google's Chrome

- Chrome identifies three different types of processes:
 - Browser, renderers, and plug-ins.
- **The browser process**
 - Is responsible for managing the user interface as well as disk and network I/O. A new browser process is created when Chrome is started. Only one browser process is created.
- **Renderer process**
 - Contain logic for rendering web pages. Thus, they contain the logic for handling HTML, Javascript, images and so forth.
 - As a general rule, a new renderer process is created for each website opened in a new tab, and so several renderer process may be active at the same time/
- **A plug-in process**
 - Is created for each type of plug-in (such as Flash or QuickTime) in use. Plug-in process contain the code for the plug-in as well as additional code that enables the plug-in to communicate with associated renderer process and the browser process.

2019/11/8

KonokaScript



61

リダイレクション, パイプ (発展)

組込みシステム講義

63

リダイレクション、パイプ

・ 概要

- リダイレクション：標準入出力をファイルにする機能

```
% ls *.c > a.txt
```

```
% wc < a.txt
```

38 38 374

・ 解説

- リダイレクションにより、入出力をファイルに対して行うという処理は、ls, wc とは無関係に行われる
- プロセスの標準入出力は、ファイルディスクリプタの0,1,2 に割り当てられている
- これらのファイルディスクリプタの先が何かは、プロセスは関知しない

組込みシステム講義



Execve とファイルディスクリプタ

・ Execve

- プロセスのメモリ空間を上書き、main 関数から実行を開始する
- ただし、ファイルディスクリプタを含め、その他の属性は変更しない
- ファイルディスクリプタの付け替えを自由にできるようにしたい

組込みシステム講義

ファイルディスクリプタの付け替え

・ dup, dup2 システムコール

```
int dup(int oldfd);  
int dup2(int oldfd, int newfd);
```

・ どちらも指定されたファイルディスクリプタの複製を作成する

- 呼び出し後は、古いファイルディスクリプタも dup, dup2 により作成された新しいファイルディスクリプタも、同じものとして利用できるようになっている
- いづれも close により閉じるまで、有効である

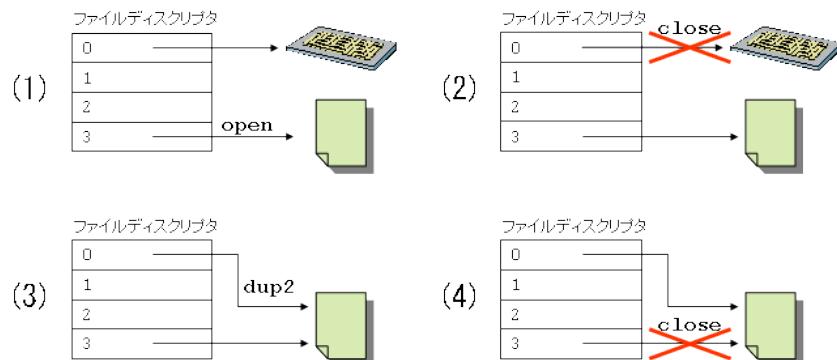
・ dup, dup2

- dup: ファイルディスクリプタとして使用していない最小の値を使用
- dup2: newfd として指定された値を使用するという点が異なる

組込みシステム講義



dup2 のリダイレクション設定



1. ファイルをオープンし、ファイルディスクリプタ 3を得る
2. 標準入力（ファイルディスクリプタ 0）をクローズする
3. dup2 により、ファイルディスクリプタ 3の複製を、ファイルディスクリプタ 0として作成する
4. ファイルディスクリプタ3 をクローズすることで、ファイルはファイルディスクリプタ 0からのみアクセスできるようになる

組込みシステム講義

プログラム例

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5 void
6 main(int argc, char *argv[])
7 {
8     int file_fd;
9     if (argc != 2) {
10         printf("Usage: %s
file_name\n", argv[0]);
11         exit(1);
12     }
13     file_fd = open(argv[1],
O_RDONLY);
14     if (file_fd < 0) {
15         perror("open");
16         close(file_fd);
17         exit(1);
18 }
```

組込みシステム講義



68

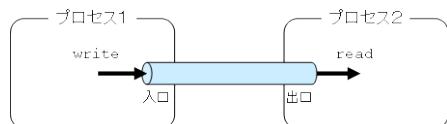
パイプ

・ パイプ機能

- あるコマンドの出力を、別のコマンドの入力とする
 - E.g. cat /proc/cpuinfo | less
 - E.g. ls *.c | wc

・ パイプの構造

- 入口、出口があり、入口から書いたデータを出口から読み出すことができる。
 - データの流れの方向は、入口から、出口の単方向であり、入口から書いた順番で出口から読み出される。



組込みシステム講義

70

実行結果と解説

・ プログラムの内容

- 第1引数に指定されたファイルを、wcコマンドの標準入力とするプログラム

・ Dupによるリダイレクション

- file_fd = open()にてファイルのオープン
 - ファイルディスクリプタ3を得る
- 19行目でファイルディスクリプタ0をクローズ
 - 標準入力（ファイルディスクリプタ0）をクローズする
- dup2の実行
 - オープンしたファイルを、ファイルディスクリプタ0からもアクセスできるようにする
- close(file_fd)
 - オープンしたファイルは、ファイルディスクリプタ0からのみアクセスできるようになる
- execvpにより、wcを起動
 - ファイルディスクリプタ0は、12行目でオープンしたファイルとつながっているため、wcの標準入力はそのファイルから読み込まれる

組込みシステム講義



69

pipeシステムコール

・ システムコール

```
int pipe (int filedes[2]);
```

・ 実行内容

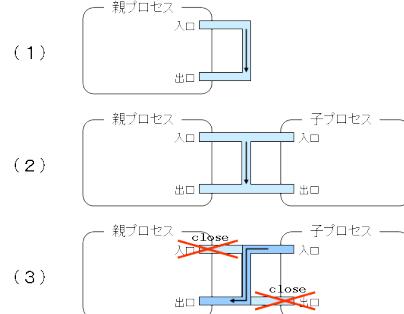
- 入力と出力それぞれの口（ファイルディスクリプタ）を持つパイプが1本作られる
- ファイルディスクリプタは、filedes配列に格納される。
 - filedes[0]がパイプの出口、filedes[1]がパイプの入口のファイルディスクリプタになる
- filedes[0]から読み、filedes[1]に書く

組込みシステム講義



71

pipe の実行手順



1. pipe システムコールの実行
-入力と出力それぞれの口（ファイルディスクリプタ）を持つパイプが1本作られる。
2. この状態で fork を呼ぶと、パイプ（入出力の口）が両方とも共有された状態になる。
3. 一方のプロセスが入力を、もう一方が出力の口を閉じると、プロセスからプロセスへデータを送れる（プロセス間通信）状態になる。

組込みシステム講義

72



プログラム例

```

23 void
24 do_parent()
25 {
26     char c;
27     int count, status;
28
29     printf("this is parent.\n");
30
31     close(pipe_fd[1]);
32
33     while ((count = read(pipe_fd[0],
34 &c, 1)) > 0) {
35         putchar(c);
36     }
37
38     if (count < 0) {
39         perror("read");
40         exit(1);
41     }
42
43     if (wait(&status) < 0) {
44         perror("wait");
45         exit(1);
46     }
47 }
```

組込みシステム講義

73



プログラム例（続き）

```

48 main()
49 {
50     int child;
51
52     if (pipe(pipe_fd) < 0) {
53         perror("pipe");
54         exit(1);
55     }
56
57     if ((child = fork()) < 0) {
58         perror("fork");
59         exit(1);
60     }
61
62     if (child)
63         do_parent();
64     else
65         do_child();
66 }
```

実行結果

```
% ./a.out
this is parent.
this is child.
Hello, dad!!.
```

組込みシステム講義

74

Dup を用いた書き換え

```

22 void
23 do_parent()
24 {
25     char c;
26     int count, status;
27
28     printf("this is parent\n");
29
30     close(pipe_fd[1]);
31
32     close(0);
33     dup2(pipe_fd[0], 0);
34     close(pipe_fd[0]);
35
36     close(1);
37     dup2(pipe_fd[1], 1);
38     close(pipe_fd[1]);
39
40     while ((*p)
41         putchar(*p++));
42
43 }
```

組込みシステム講義

75



Dup を用いた書き換え（続き）

```
44
45 main()
46 {
47     int child;
48
49     if (pipe(pipe_fd) < 0) {
50         perror("pipe");
51         exit(1);
52     }
53
54     if ((child = fork()) < 0) {
55         perror("fork");
56         exit(1);
57     }
58
59     if (child)
60         do_parent();
61     else
62         do_child();
63 }
```

組込みシステム講義



76

簡単なプログラムの実行

- 以下のライブラリ関数は、単純なプロセスの実行や、パイプを使用してのプロセスの実行を簡単に用意されている

```
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
int system (const char * string);
```

組込みシステム講義



78

メモリ領域確保

- 動的メモリ領域の確保

```
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

- システムコールの違い

- malloc: 確保される領域の初期値は不定
- calloc: 確保した領域を 0 に初期化する
- free: 確保されたメモリを解放する

組込みシステム講義



77

練習課題

- system(3)

• System(3) は、シェルによるコマンドの実行を行ってくれるライブラリ関数である。以下のように、シェル（/bin/sh）のコマンドラインを文字列で渡すと実行してくれる。

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 main()
5 {
6     system("ls *.c | wc");
7 }
```

- fork, execve (又は相当のライブラリ関数), waitなどを使用して、system(3)相当の機能を持つ関数 mysystem を作りなさい。
- マニュアルページに記述してあるシグナルについては、無視してよい

組込みシステム講義



79

同期, 非同期

システムプログラミング

80

81

ブロッキング/ノンブロッキング

・プロセスの終了

- ・カーネルは親にシグナルを送って通知
- ・子の終了は非同期（親の実行中いつでも起こり得る）<シグナルはいつでも送られる
- ・親は、実行中にシグナルを
 - ・無視する：ブロックする
 - ・受け取る：シグナルハンドラで受け取る

- #include <sys/wait.h>

- 1.pid_t wait(int *statloc);

- 2.pid_t waitpid(pid_t pid, int *statloc, int options);

- 1: 子プロセスが終了するまで、呼び出し側をブロックする
- 2: 正常終了は受け取る

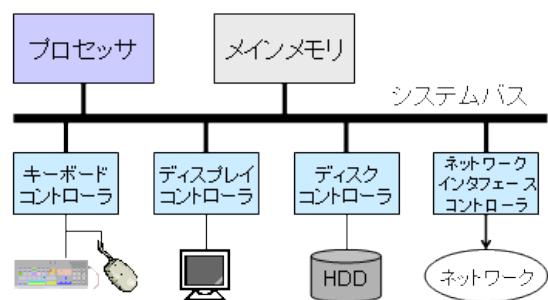
・ブロッキング：待つ

・ノンブロッキング：待たない（割り込み受ける）

システムプログラミング

(復習) 入出力機器

- ・割り込み
 - ・入出力機器からのイベント通知の仕組み
- ・入出力機器
 - ・PCの場合、コンピュータにI/Oコントローラを通じて接続されている



82

83

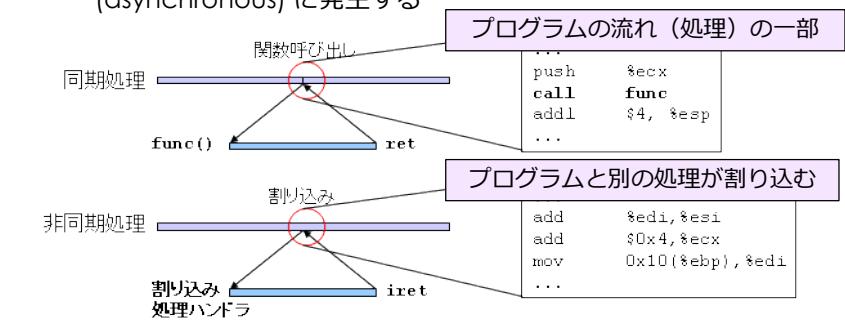
(復習) 同期処理と非同期処理

・関数呼び出しは同期処理

- ・明示的に関数を呼び出し、呼び出しされた関数で処理が行われ、関数呼び出しから帰ってくる

・割り込みは非同期処理

- ・本来のプログラムの流れとは無関係に、非同期的(asynchronous)に発生する



いつ割り込まれるかわからない

演習3：

- ・ ブロッキング、ノンブロッキングの実装の性能を比較する
 - ・ 仕様：
 - wait(), waiitpid() 部分が異なる実装の forkプログラムを作成する
 - 例に従って比較して、考察を述べる
 - ・ 例)
 - time ./fork_wait
 - 0.016u 0.104s 0:00.28 39.2% 0+0k 0+0io 0pf+0w
 - time ./fork_waitpid 10
 - 0.000u 0.008s 0:00.12 0.0% 0+0k 0+0io 0pf+0w
 - ・ **real/user/sys**
 - キャッシュに乗ると早いので一度目と二度目以降の結果が異なるので注意