システムソフトウェア

2020年度

第1回 (10/1)

月曜7-8限・木曜7-8限(Zoom)

講義担当：渡部卓雄 (Takuo Watanabe)

http://titech-os.github.io

e-mail: takuo∅c.titech.ac.jp

本日のメニュー

- 導入
  - 講義の目的と講義内容の説明
  - 講義計画と評価方法の説明
  - 参考図書など
- OSの役割と機能
- システムコール

講義の目的

- オペレーティングシステム(OS)に関する基本的概念と基本的アルゴリズムを習得する.
  - OSの基本構造，プロセス管理，メモリ管理，ファイルシステム，デバイス管理など
- OSに関係する諸理論と実装技術をバランスよく学習する.

なぜOSについて学習するのか

- プログラミング言語処理系（コンパイラ等）に関する技術と同様に，計算機科学(Computer Science)における最も重要な成果が集約されている.
- 日々最新の技術が投入され発展しつつある.
- 計算機システム・ソフトウェア分野以外でも，OSに関する知識は重要である.

本講義で仮定する知識

- 関連講義
  - 手続き型プログラミング基礎
  - 手続き型プログラミング発展
  - システムプログラミング
  - アセンブリ言語
  - コンピュータアーキテクチャ
  - （コンパイラ構成）
- その他
  - Unixの（コマンドラインによる）操作
  - 数理論理学

教材：xv6

- MITで開発された教育用OS
  - カーネルとユーザランド（アプリケーション）計1万行程度でUnixの基本的な機能を提供
  - x86/RISC-Vで動作
    - 今年度は主にRISC-V版を使用する
- ソースコード
  - RISC-V版
    - https://github.com/mit-pdos/xv6-riscv/
  - x86版
    - https://github.com/mit-pdos/xv6-public/

講義で扱うプログラムの例 (syscall.c)

```
108 static uint64 (*syscalls[])(void) = {
109 [SYS_fork]    sys_fork,
110 [SYS_exit]    sys_exit,
111 [SYS_wait]    sys_wait,
     ...
129 [SYS_close]   sys_close,
130 };
131
132 void
133 syscall(void)
134 {
135   int num;
136   struct proc *p = myproc();
137
138   num = p->tf->a7;
139   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
140     p->tf->a0 = syscalls[num]();
141   } else {
142     printf("%d %s: unknown sys call %d\n",
143             p->pid, p->name, num);
144     p->tf->a0 = -1;
145   }
146 }
```

配列，構造体，共用体，ポインタ（関数ポインタ含む）等が駆使されたコードを読む（書く）必要があるので，復習しておくこと．

講義で扱うプログラムの例 (swtch.S)

```
 8 .globl swtch
 9 swtch:
10         sd ra, 0(a0)
11         sd sp, 8(a0)
12         sd s0, 16(a0)
13         sd s1, 24(a0)
14         sd s2, 32(a0)
15         sd s3, 40(a0)
16         sd s4, 48(a0)
17         sd s5, 56(a0)
18         sd s6, 64(a0)
19         sd s7, 72(a0)
20         sd s8, 80(a0)
21         sd s9, 88(a0)
22         sd s10, 96(a0)
23         sd s11, 104(a0)
24
```

```
25         ld ra, 0(a1)
26         ld sp, 8(a1)
27         ld s0, 16(a1)
28         ld s1, 24(a1)
29         ld s2, 32(a1)
30         ld s3, 40(a1)
31         ld s4, 48(a1)
32         ld s5, 56(a1)
33         ld s6, 64(a1)
34         ld s7, 72(a1)
35         ld s8, 80(a1)
36         ld s9, 88(a1)
37         ld s10, 96(a1)
38         ld s11, 104(a1)
39
40         ret
```

スケジュール(10/1)

	日付	時限	
1	10/1(木)	7-8	導入・OSの役割と機能
2	10/5(月)	7-8	OSの構成
3	10/8(木)	7-8	メモリ管理(1)
4	10/15(木)	7-8	メモリ管理(2)
5	10/19(月)	7-8	割込みとシステムコール(1)
6	10/22(木)	7-8	割込みとシステムコール(2)
7	10/26(月)	7-8	並行性制御(1)
8	10/29(木)	7-8	並行性制御(2)
9	11/2(月)	7-8	スケジューリング(1)
10	11/5(木)	7-8	スケジューリング(2)
11	11/9(月)	7-8	ファイルシステム(1)
12	11/12(木)	7-8	ファイルシステム(2)
13	11/16(月)	7-8	並行性制御(3)
14	11/19(木)	7-8	並行性制御(4)
期末試験	未定	未定	

成績評価について

- 期末試験(50%)
  - 必須．実施方法については未定
- 課題(10%+40%)
  - 小課題1回+大課題1回
    - プログラム作成と実験
  - 提出物（レポート，プログラム等）により評価
    - レポートはLaTeXで書くこと.
- 自由課題(+α)
  - 自由参加．特にOSや関連分野に興味がある人，高得点が欲しい人向け.
  - 内容・方法については個別に対応します.

遠隔講義のサポート

- Zoomによる遠隔講義を基本とするが，学習支援のため以下のサービスを用いる
- Web (Github): titech-os.github.io
  - 課題プログラム等の配布
- Slack： titech-os.slack.com
  - アナウンス，質問・コメント
- OCW-i
  - 講義資料(スライド等)の配布，課題提出
- sli.do
  - 匿名での質問・アンケート等

教科書

- 特に指定しない
- xv6については，同OSと一緒に配布されている解説書を読むこと（英文）
  - xv6, a simple, Unix-like teaching operating system
    - Russ Cox, Frans Kaashoek, Robert Morris
  - RISC-V版 (Aug. 31, 2020)
    - https://pdos.csail.mit.edu/6.828/2020/xv6/book-riscv-rev1.pdf

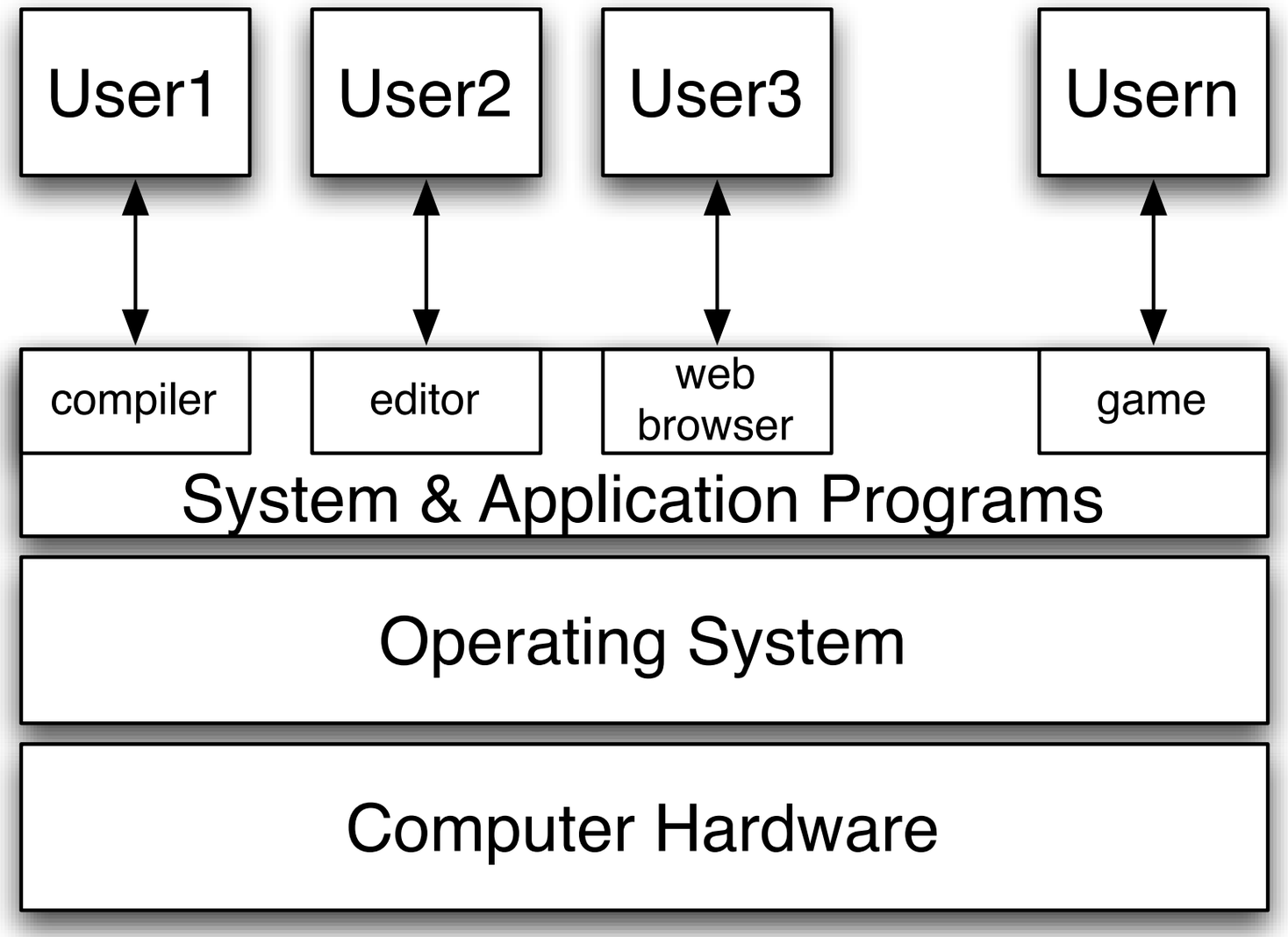
OSの役割と機能

OSの役割

- ユーザの視点
  - コンピュータを便利に使うためのソフトウェア
- ハードウェアの視点
  - 計算資源(CPU，メモリ，ディスクなど)を適切に利用するためのソフトウェア

- ユーザとハードウェアの間に位置し，抽象化と保護機能を提供するソフトウェア

OSの位置づけ


User1
User2
User3
Usern
compiler
editor
web browser
game
System & Application Programs
Operating System
Computer Hardware

OSが見せる「幻想」(1)

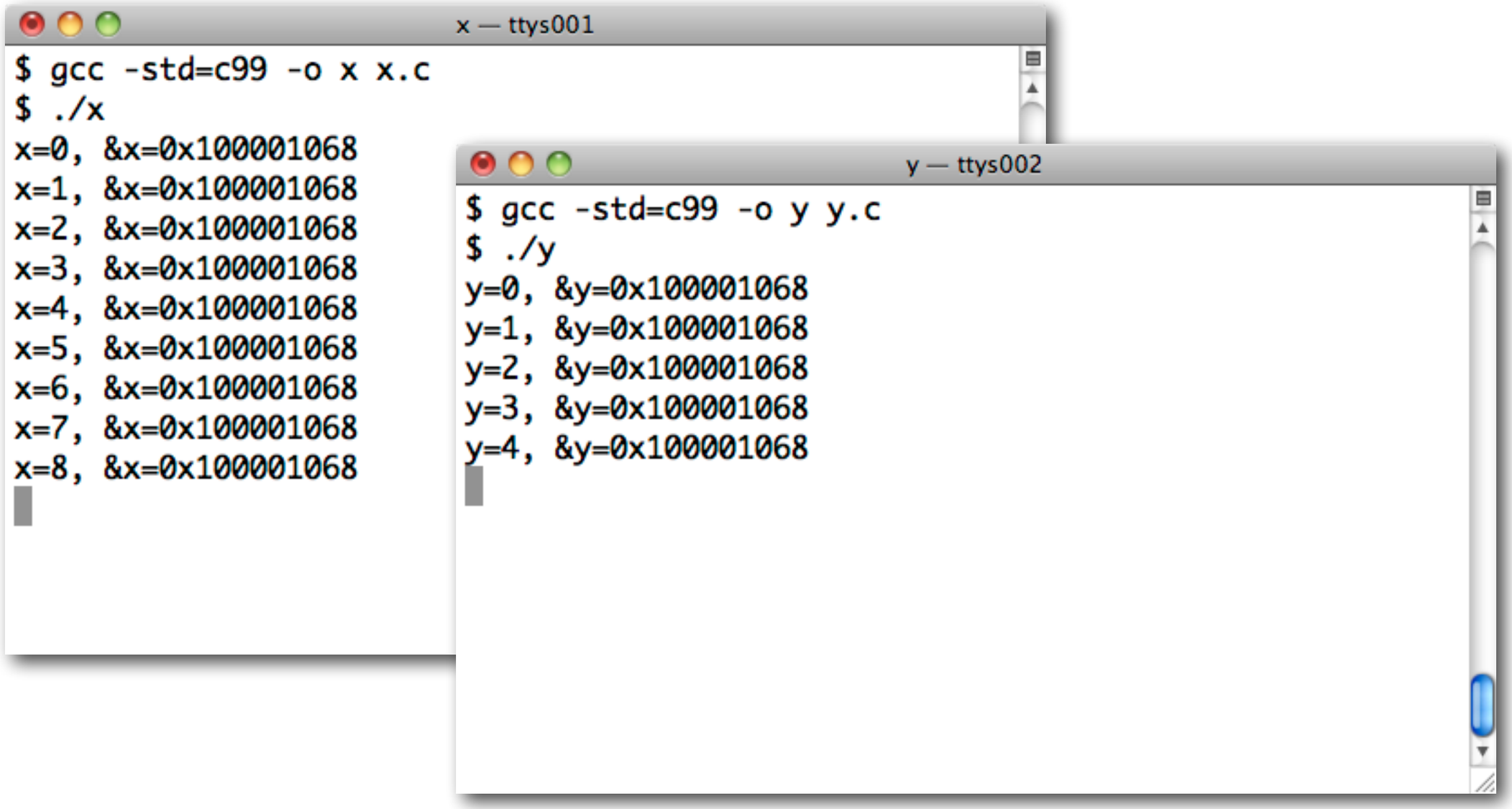
(1) 以下のような２つのプログラムを用意する.

```c
#include <stdio.h>
int x = 0;
int main (void) {
  while (x < 10000) {
    printf("x=%d, &x=%p\n",
           x++, &x);
    for (int i = 100000000;
         i > 0; i--);
  }
  return 0;
}
```

x.c

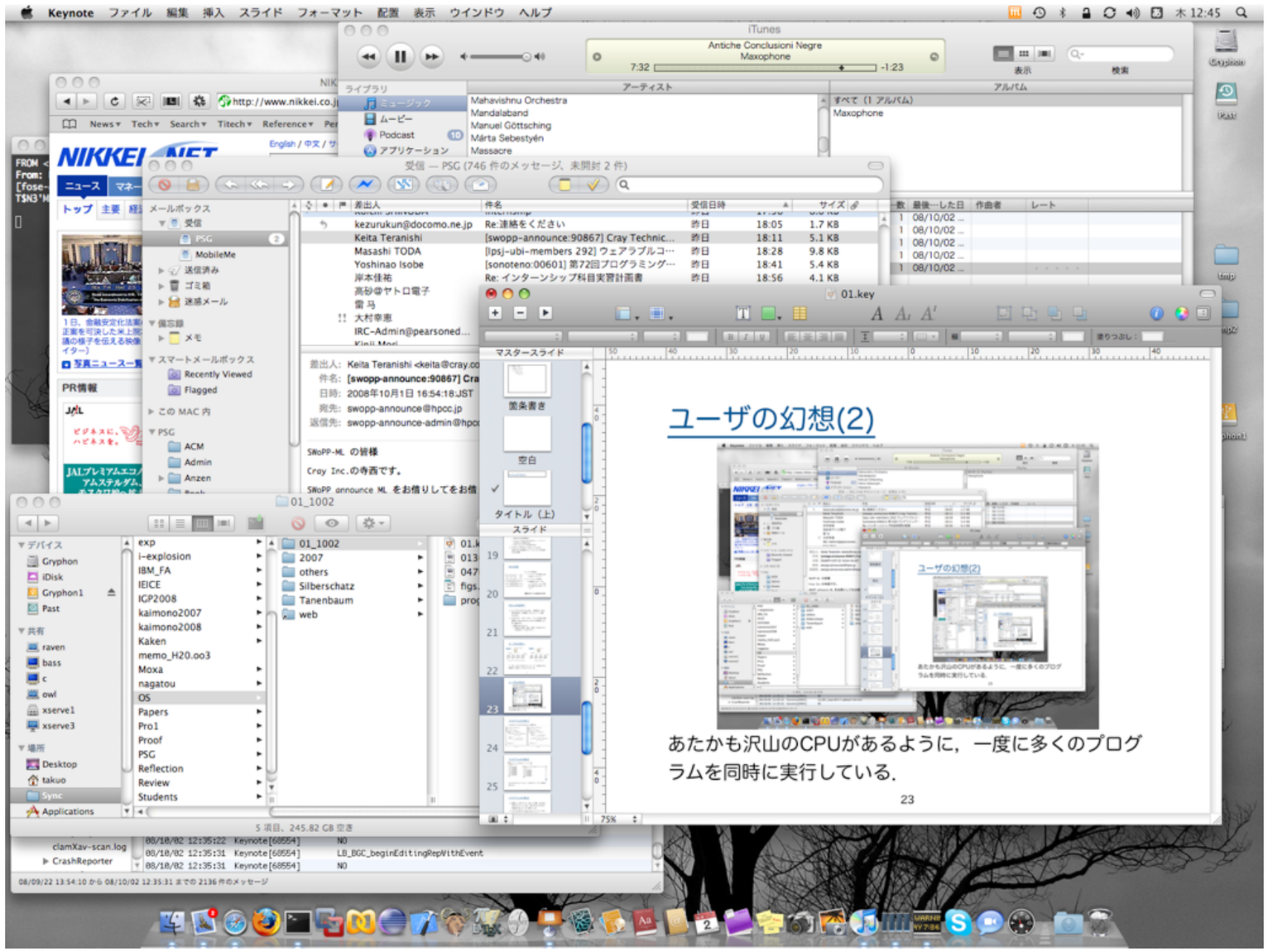
```c
#include <stdio.h>
int y = 0;
int main (void) {
  while (y < 10000) {
    for (int i = 100000000;
         i > 0; i--);
    printf("y=%d, &y=%p\n",
           y++, &y);
  }
  return 0;
}
```

y.c



(2) それぞれ別々のウィンドウで同時に実行する．ここで変数 x, y のアドレス(ポインタ値)がたまたま同じになっているが，実行するとそれぞれの値はきちんと 1 ずつ増えている．

OSが見せる「幻想」(2)

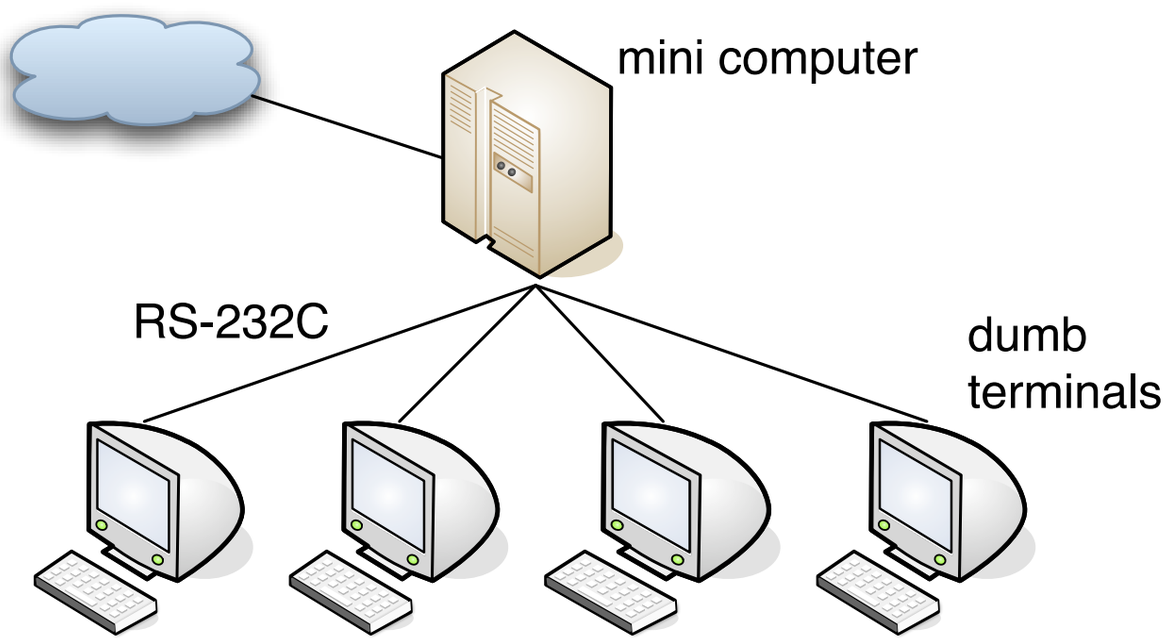


あたかも沢山のCPUがあるように，一度に多くのプログラムを同時に実行している.

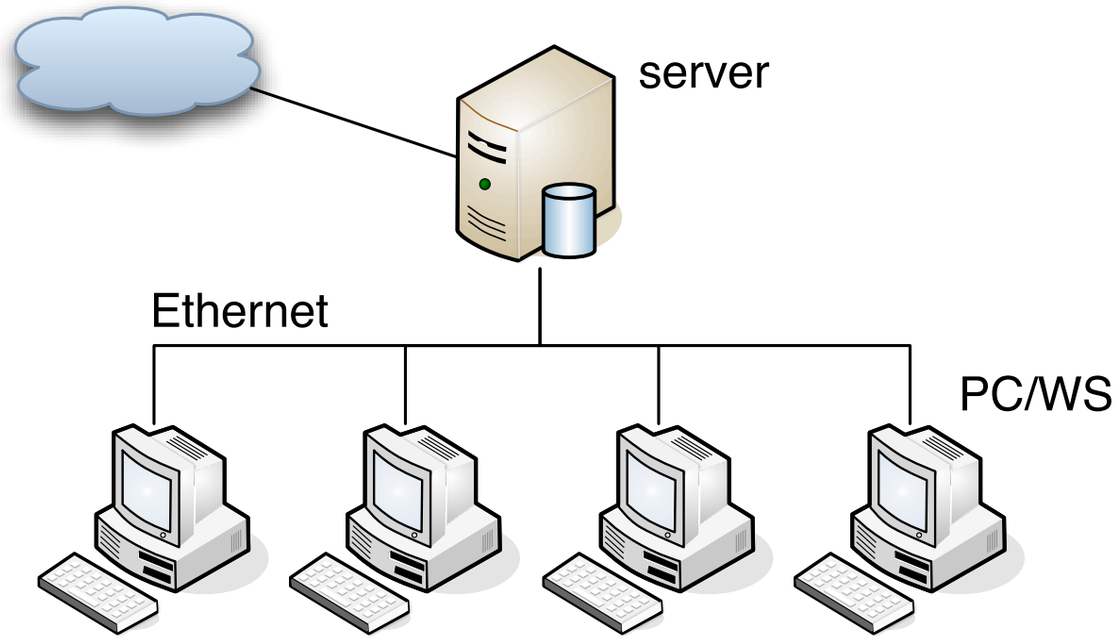
何が起ったのか(1,2)

- 各プログラムは，あたかも自分がCPUを占有しているかのように動作できる.
  - CPU(コア)数よりも実行しているプログラムの数が多い．CPUコアは一度に1つのプログラムしか実行できないのではなかったのか？
- 各プログラムは，あたかも自分がメモリを占有しているかのように動作している.
  - 同時に実行している複数のプログラムで，同じメモリアドレスを別のものとして使っている.

OSが見せる「幻想」(3)



タイムシェアリングシステム (TSS)



LAN上のPC/WS

ユーザは，あたかも自分がコンピュータやディスクを占有しているようにみえている.

TSS (Time Sharing System)

DEC VAX-11/780

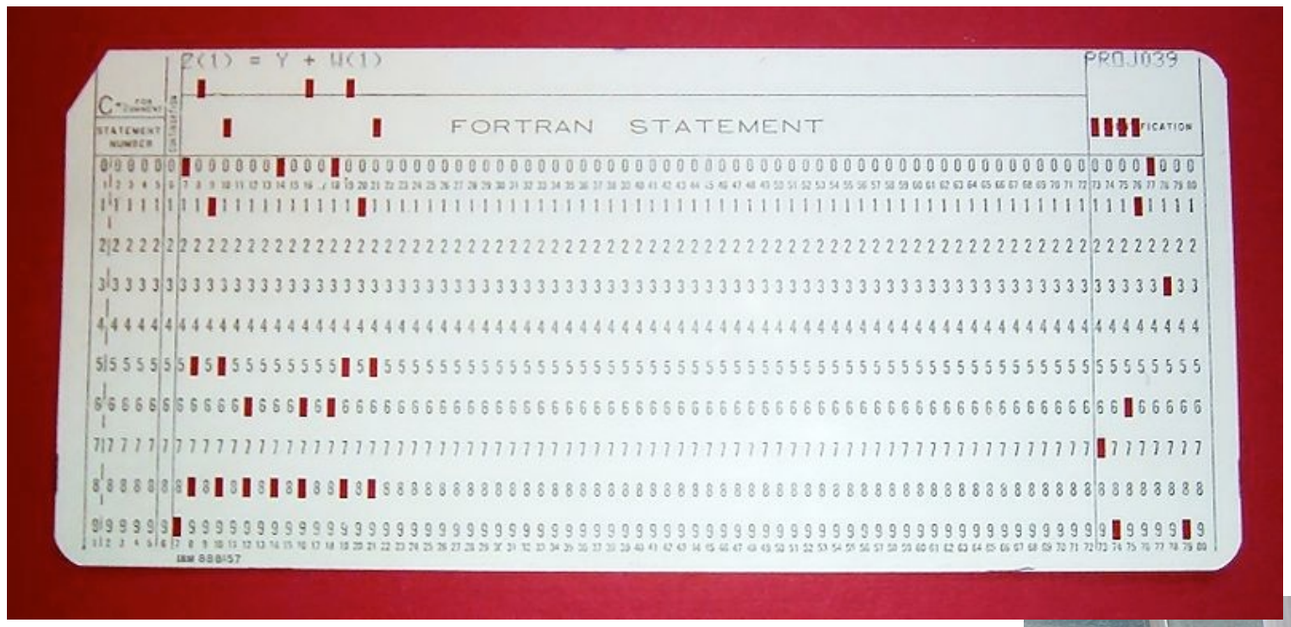
シリアル回線
(9600bps)

DEC VT-100

1台のコンピュータに数台〜数十台の端末がシリアル回線によって接続されている．TSSは，各端末のユーザが見かけ上同時にコンピュータを使えるような仕組みである．

写真はWikipediaより

閑話：バッチ処理(1)



カードパンチャーによってカードにプログラム1行分の穴をあける．プログラムはこのカードの束である．

カードの束をカードリーダで読み取ってコンパイル・実行を行う．結果はラインプリンタに出力される．

写真はWikipediaより

閑話：バッチ処理(2)

プログラムをパンチカードに打つ前に，左のようなコーディングシートに手書きでプログラムを書き，紙上デバッグをしっかり行う必要があった.

大学のプログラミング演習は，(1)紙上コーディングして(2)パンチカードを作成し, (3)担当者にカードの束を渡して(4)結果のプリントアウトを待つという形で行われた.　（私より上の世代）

写真はWikipediaより

何が起ったのか(3)

- CPUやメモリ，ハードディスク，プリンタ，ネットワーク等の計算資源を多くのユーザで共有している.
- 他のマシン(ファイルサーバ)上のハードディスクに，ネットワークを介してあたかも自分のPCに直接つながっているかのようにアクセスできる.
  - ネットワークに接続されたプリンタ等も同様

OSによる抽象化(abstraction)

- ユーザやプログラムに対し，仮想的な(ある意味理想の)計算機システムという「幻想」をみせる
- そのために，CPU, メモリ，ディスク等の計算資源(computational resource)を抽象化した形で提供する.
  - ハードウェアに依存した詳細の隠蔽
  - ハードウェアの物理的制約の緩和
  - ハードウェアの共有

OSによる保護(protection)

- 1つのプログラムがCPUを長時間独占できないようにする.
- 1つのプログラムが他のプログラムが使っているメモリ領域を勝手に読んだり破壊できないようにする.
- 適切な権限のない人(プログラム)が計算資源を使うことができないようにする.

OSの機能が不十分な場合(1)

```
#include <stdio.h>
int x = 0;
int main (void) {
  while (x < 10000) {
    printf("x=%d, &x=%p\n",
           x++, &x);
    for (int i = 100000000;
         i > 0; i--)
      yield(); // 切替操作
  }
  return 0;
}
```

- プログラム内で明示的に他のプログラムへの切替え操作を実行する必要がある.
  - これを間違いなくやるのはかなり大変.
  - さぼれば自分がCPUを占有できる！

OSの機能が不十分な場合(2)

- 異なるプログラムの間で，変数や関数のアドレスが重ならないよう意識する必要がある.
  - 他のプログラムが使っているメモリ領域を勝手に読んだり書き込んだりしないよう，注意してプログラムを作る必要がある.
  - 悪意をもって，他のプログラムが使っているメモリ領域を破壊するようなプログラムを防ぐことができない.

OSの機能が不十分な場合(3)

- 様々なデバイスを複数のプログラムで共有し，適切に使うようプログラムを書くのはかなり大変
  - ディスクコントローラ
  - USBコントローラ
    - キーボード，マウス，USBメモリ他
  - ネットワークコントローラ
  - グラフィックコントローラ
    - ディスプレイ

OSの役割(まとめ)

- 以下を行う機能を提供することで，コンピュータハードウェアを使いやすくする.
  - 抽象化
    - ハードウェアに依存した詳細の隠蔽
    - ハードウェアの物理的制約の緩和
    - ハードウェアの共有
  - 保護

システムコール

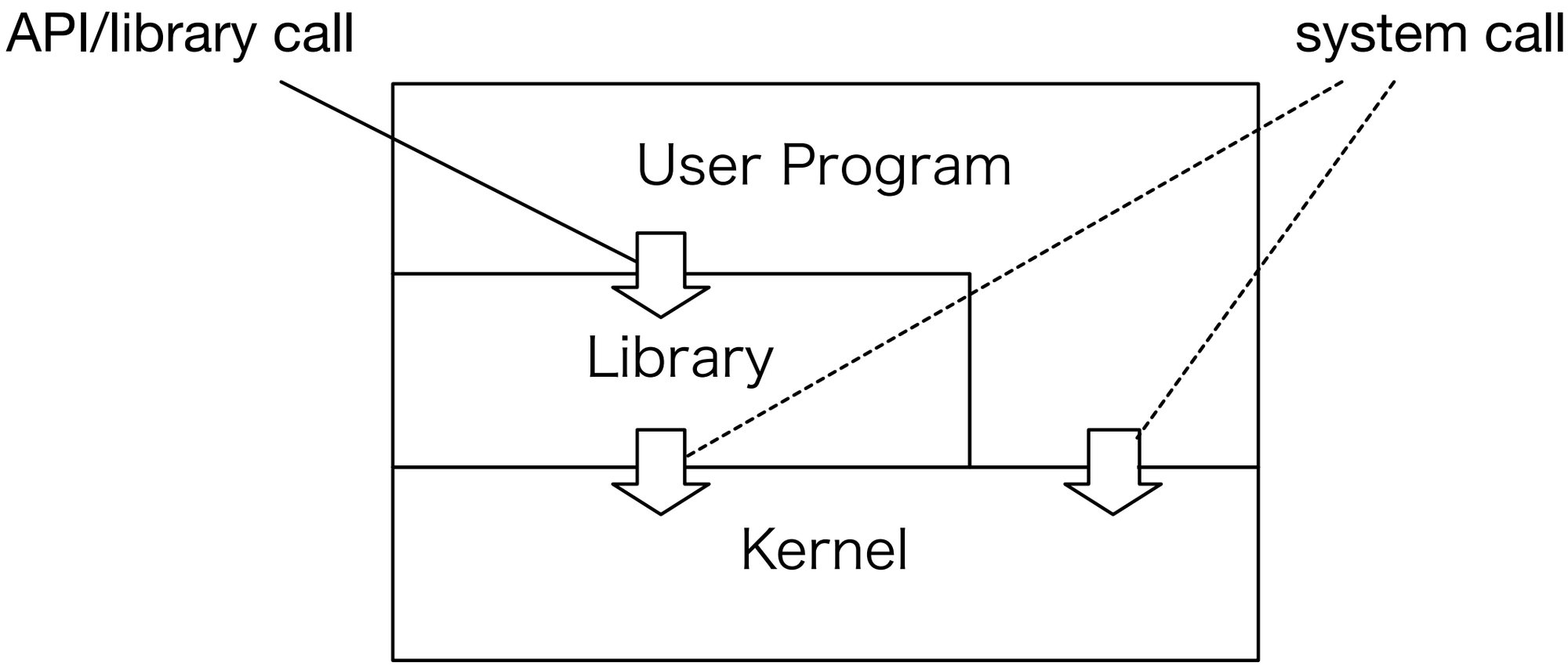
OSが提供するサービスの利用

- ユーザプログラムが意味のある動作をするためには，OSが提供するサービスを利用する必要がある．
  - 入出力，プロセスに関する諸操作，メモリに関する諸操作，ファイルに関する諸操作，通信
- OSが提供するサービスを利用する方法
  - システムコール
  - API/ライブラリ

システムコール vs. ライブラリ

- システムコール
  - カーネルが提供するサービスを直接呼び出す機構
    - 例： write, fork, exec, sbrk, etc.
- ライブラリ
  - システムコールによるサービスや，その他よく用いられる処理を使いやすい形で提供する関数やクラス等の集まり
    - 例：printf, malloc, qsort

システムコール vs. ライブラリ



システムコール

- OSカーネルが提供する最も基本的なサービス
- 割り込み命令を用いて呼び出される
  - 通常の関数呼び出しとは異なるメカニズム
- カーネルモードで実行される
  - 呼び出しの際，ユーザモードとカーネルモードの切替が行われる

システムコールの種類

- プロセス制御
- 入出力，通信
- ファイル管理
- デバイス管理
- OS自体の制御・管理

xv6のシステムコール(1/2)

fork()	プロセスを作成する
exit()	現プロセスを終了する
wait()	子プロセスの終了を待つ
kill(pid)	pidで指定したプロセスを終了させる
getpid()	現プロセスのidを得る
sleep(n)	n秒休む
exec(filename, *argv)	現プロセスで指定したプログラムを実行する
sbrk(n)	現プロセスのメモリ領域を拡張する
open(filename, flags)	ファイルをオープンする
read(fd, buf, n)	ファイルからbufにnバイト読む

xv6のシステムコール(2/2)

write(fd, buf, n)	bufの内容をファイルにnバイト書く
close(fd)	開いているファイルを閉じる
dup(fd)	ファイルディスクリプタの複製を作る
pipe(p)	パイプを作る
chdir(dirname)	カレントディレクトリを変更する
mkdir(dirname)	ディレクトリを作る
mknod(name, major, minor)	デバイスファイルを作る
fstat(fd, fstatp)	ファイルの情報を得る
link(f1, f2)	ファイルのリンクを作る
unlink(filename)	ファイルのリンクを消去する

xv6のライブラリ

strcpy(s, t)	文字列をコピーする
strcmp(p, q)	文字列を比較する
strlen(s)	文字列の長さを計る
memset(dst, c, n)	指定したメモリの内容をcで埋める
strchr(s, c)	文字列中の文字の位置を調べる
gets(buf, max)	標準入力から文字列を一行読み込む
stat(n, st)	ファイルの情報を得る
memmove(vdst, vsrc, n)	メモリの内容をコピーする
malloc(nbyte)	指定したサイズのメモリを確保する
free(ap)	mallocで確保したメモリを解放する
atoi(s)	文字列を10進整数とみなして数値に変換する
printf(fd, s, ...)	書式付きで出力する

入出力システムコール(1)

- int read(int fd, char *buf, int size)
  - ファイルディスクリプタfdが指す入力から，最大size個の文字（バイトデータ）をbufに読み込む
    - bufが指す先にはsizeバイト以上の書き込み可能なメモリが確保されていなければならない
  - 返値は読み込んだ文字（バイト）数
- int write(int fd, char *buf, int size)
  - ファイルディスクリプタfdが指す出力に，bufから最大size個の文字（バイトデータ）を出力する
  - 返値は出力できた文字（バイト）数
    - 通常はsizeと同じ

ファイルディスクリプタ

- ファイル等の入出力先への参照を表す非負整数値で，入出力を行うシステムコールはこの値を使う．
  - 0, 1, 2は以下の入出力先に割り当てられている
    - ０：標準入力
    - １：標準出力
    - ２：標準エラー出力
  - ファイルディスクリプタは空いている番号のうち小さいものから割り当てられる

入出力システムコール(2)

- int open(char *filename, int flags)
  - filenameで指定されたファイルをflags（次頁参照）にしたがってオープンする
  - 返値：オープンしたファイルに割り当てられたファイルディスクリプタ，-1（エラー）
- int close(int fd)
  - ファイルディスクリプタfdで指定されたファイルをクローズする
  - fdは空き番号となる
  - 返値：0（成功），-1（エラー）

openのフラグ

- 以下を | (bitwise-or)で組み合わせて用いる
  - O_RDONLY：読み出し専用
  - O_WRONLY：書き込み専用
  - O_RDWR：読み書き両用
  - O_CREATE：ファイルがない場合は作成する

- 例：open("foo", O_WRONLY | O_CREATE)
  - ファイルfooを書き込み専用でオープンする．もし当該ファイルがない場合は新たに作成する.

例題：ucopy.c (xv6)

```c
#include "types.h"
#include "stat.h"
#include "fcntl.h"
#include "user.h"

#define BUFSIZE 512
char buf[BUFSIZE];

int toupper(int c) {
    return c >= 'a' && c <= 'z' ? c - 'a' + 'A' : c;
}

int main(int argc, char *argv[]) {
    char *prog = argv[0];
    char *sfile = argv[1], *dfile = argv[2];
    int sfd = open(sfile, O_RDONLY);
    int dfd = open(dfile, O_WRONLY | O_CREATE);
    int n;
    while ((n = read(sfd, buf, sizeof(buf))) > 0) {
        int i;
        for (i = 0; i < n; i++)
            buf[i] = toupper(buf[i]);
        write(dfd, buf, n);
    }
    close(sfd);
    close(dfd);
    exit();
}
```

例題：ucopy.c (macOS)

```c
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define BUFSIZE 512
char buf[BUFSIZE];

int toupper(int c) {
    return c >= 'a' && c <= 'z' ? c - 'a' + 'A' : c;
}

int main(int argc, char *argv[]) {
    char *prog = argv[0];
    char *sfile = argv[1], *dfile = argv[2];
    int sfd = open(sfile, O_RDONLY);
    int dfd = open(dfile, O_WRONLY | O_CREAT, 0666);
    int n;
    while ((n = read(sfd, buf, sizeof(buf))) > 0) {
        int i;
        for (i = 0; i < n; i++)
            buf[i] = toupper(buf[i]);
        write(dfd, buf, n);
    }
    close(sfd);
    close(dfd);
    exit(EXIT_SUCCESS);
}
```

例題の動作

- ファイルをコピーする．その際，アルファベットの小文字(a～z)を大文字に変更する．

```
$ ucopy README README1
```

実行時エラーへの対処

- open
  - openできない場合はファイルディスクリプタの代わりに -1 を返す
- read
  - 返値は読むことができたバイト数
  - エラーの場合 -1
- write
  - 返値は書くことができたバイト数
  - エラーの場合 -1
    - 第3引数より小さい場合はエラーとすることもある

エラー処理(1)

コマンドライン引数の検査

```
int main(int argc, char *argv[]) {
    char *prog = argv[0];
    if (argc != 3) {
        printf(2, "usage: %s sfile dfile\n", prog);
        exit();
    }
    char *sfile = argv[1], *dfile = argv[2];
    ...
}
```

- 関数mainの引数
  - argc : コマンドライン引数の数＋1
  - argv : コマンドライン引数
    - argv[0] はプログラム（実行形式）自体のファイル名

エラー処理(2)

ファイルがオープン可能か否かの検査を追加

```
int main(int argc, char *argv[]) {
    ...
    int sfd = open(sfile, O_RDONLY);
    if (sfd < 0) {
        printf(2, "%s: cannot open %s\n", prog, sfile);
        exit();
    }
    int dfd = open(dfile, O_WRONLY | O_CREATE);
    if (dfd < 0) {
        printf(2, "%s: cannot open %s\n", prog, dfile);
        close(sfd);
        exit();
    }
    int n;
    ...
}
```

エラー処理(3)

読み書きエラーのチェックを追加

```
int main(int argc, char *argv[]) {
    ...
    while ((n = read(sfd, buf, sizeof(buf))) > 0) {
        int i;
        for (i = 0; i < n; i++)
            buf[i] = toupper(buf[i]);
        if (write(dfd, buf, n) < n) {
            printf(2, "%s: write error: %s\n", prog, dfile);
            close(sfd);
            close(dfd);
            exit();
        }
    }
    if (n < 0){
        printf(2, "%s: read error: %s\n", prog, sfile);
        close(sfd);
        close(dfd);
        exit();
    }
    ...
}
```

エラー処理を加えたコード

```
#include "types.h"
#include "stat.h"
#include "fcntl.h"
#include "user.h"

#define BUFSIZE 512
char buf[BUFSIZE];

int toupper(int c) {
    return c >= 'a' && c <= 'z' ? c - 'a' + 'A' : c;
}

int main(int argc, char *argv[]) {
    char *prog = argv[0];
    if (argc != 3) {
        printf(2, "usage: %s sfile dfile\n", prog);
        exit();
    }
    char *sfile = argv[1], *dfile = argv[2];
    int sfd = open(sfile, O_RDONLY);
    if (sfd < 0) {
        printf(2, "%s: cannot open %s\n", prog, sfile);
        exit();
    }
```

```
    int dfd = open(dfile, O_WRONLY | O_CREATE);
    if (dfd < 0) {
        printf(2, "%s: cannot open %s\n", prog, dfile);
        close(sfd);
        exit();
    }
    int n;
    while ((n = read(sfd, buf, sizeof(buf))) > 0) {
        int i;
        for (i = 0; i < n; i++)
            buf[i] = toupper(buf[i]);
        if (write(dfd, buf, n) < n) {
            printf(2, "%s: write error: %s\n", prog, dfile);
            close(sfd);
            close(dfd);
            exit();
        }
    }
    if (n < 0) {
        printf(2, "%s: read error: %s\n", prog, sfile);
        close(sfd);
        close(dfd);
        exit();
    }
    close(sfd);
    close(dfd);
    exit();
}
```

プロセス

- 実行中のプログラム
  - プログラムとプロセスの違い
    - プログラム：（ファイルに格納されている）命令の列（あるいはプログラムテキスト）
    - プロセス：メモリ上にコピーされたプログラムと，それを実行するために割り当てられたメモリやCPUなどの計算資源に関する諸情報
  - OSによって，複数のプロセスが（見かけ上）同時に実行することができる
    - 一つのプログラムに対し，それを実行する複数のプロセスが存在し得る.

プロセス関連のシステムコール(1)

- int fork()
  - 現在実行中のプロセスの分身を作る
    - fork実行後は，同じプログラム（のforkの呼び出しの続き）を実行するプロセスがもう一つできる.
      - 新しくできたプロセスはforkを実行したプロセスの子プロセスという（forkを実行した側は親プロセス）.
  - 返値
    - 親プロセス：子プロセスのプロセスID
      - プロセスID：各プロセスに付与されるユニークな番号
    - 子プロセス：0
    - −1（エラー）

プロセス関連のシステムコール(2)

- int exit(int *)
  - 実行中のプロセスを停止する
    - xv6では必ずこれを行うこと
  - 返値：-1（エラー）
    - 成功した場合は呼び出し側に戻ることはない
- int wait()
  - 子プロセスの停止を待つ
  - 返値：停止した子プロセスのID，-1（エラー）

プロセスの生成と終了

- 一つの例外を除いて，全てのプロセスはforkによって生成される
  - 最初のプロセス(init)だけはカーネルから起動される
- init以外の各プロセスは，その親プロセスがwaitを実行して終了を待つ
- 各プロセスはexitで終了し，（waitで待っている）親プロセスに終了が通知される
  - exitを実行後，親プロセスにwaitで看取られるまでの状態をゾンビ(zombie)という

プロセス関連のシステムコール(3)

- int exec(char *file, char **argv)
  - 現在実行中のプロセスを，fileで指定されたプログラムの実行に置き換える
  - argv: プログラムとそのコマンド行引数
    - argv[0]：プログラム名
    - argv[1], ... ：コマンド行引数
  - 返値：-1（エラー）
    - 成功した場合はこの関数の呼び出し側に戻ることはない

プロセス関連のシステムコール(4)

- int getpid()
  - 返値：実行中のプロセスのプロセスID
    - プロセスID：プロセスに付与された番号
- int kill(int pid)
  - 指定したプロセスIDをもつプロセスを停止する
  - 返値：0（成功），-1（エラー）

入出力システムコール(3)

- int pipe(int *p)
  - プロセス間通信に用いるバッファ（パイプ）を作成し，その読み出し，書き込み用ファイルディスクリプタをp[0], p[1]にそれぞれセットする.
  - pは長さ２以上の配列（を指すポインタ）
  - 返値：０（成功），-1（失敗）

入出力システムコール(4)

- int dup(int fd)
  - ファイルディスクリプタfdをコピーする．正確には，fdが指している（入出力を表す）構造体をコピーし，コピーを指すファイルディスクリプタを返値とする.
  - forkでfd（が指す構造体）が２つのプロセスによって共有された場合に，例えばどこまで読み書きをしたかといった情報も共有されてしまう．それを避けるためにdupを用いる.

例題：hellopipe.c

```
#include "types.h"
#include "stat.h"
#include "fcntl.h"
#include "user.h"

int main() {
    int p[2];
    char *argv[2] = { "wc", 0 };
    pipe(p);                            // パイプ作成
    if (fork() == 0) {                  // 子プロセス
        close(p[1]);                    // パイプの出力側は不要なので閉じる
        close(0);                       // 標準入力を閉じる
        dup(p[0]);                      // パイプの入力側の複製が標準入力になる
        close(p[0]);                    // パイプの入力側を閉じる
        exec("wc", argv);               // wcコマンドを実行
    }
    else {                              // 親プロセス
        close(p[0]);                    // パイプの入力側は不要なので閉じる
        write(p[1], "Hello, World!\n", 14);
        write(p[1], "System Software is Fun.\n", 24);
        close(p[1]);
        wait();                         // 子プロセスの終了を待つ
    }
    exit();
}
```

まとめ

- 講義についての案内
- OSの役割と機能
  - 計算資源の抽象化と保護機能
- システムコール