情報科学研究部用テキスト

テキストエディタを作ろう

Go Suzuki

目次

コンパイラ，OS の呼称方法

Microsoft C は Microsoft 製の最近の C++ コンパイラの C 言語部分を示す．つまり，`extern \"C\"` で指定した Visual C++(2005 以降) である．

BSD は実装上 BSD のシステムコール・ヘッダが利用できる OS，つまり，FreeBSD, NetBSD, OpenBSD, DragonflyBSD, macOS(Darwin) を示す． 古い BSD については当てはまらない事項がある可能性がある．

ライセンス

1. ファイルを読み書きしよう 楽

1.1. 標準ライブラリのセキュリティを考慮した正しい使い方

普段標準ライブラリをのほほ〜んと使っていないだろうか． 危険でないプログラムを作るにはしっかりとエラーハンドリングを意識して実装することが重要である．この章では，Visual C++ しか実装していない `fscanf_s`，`fopen_s` などの話も混ぜつつ，エラーハンドリングを見ていこう．

危険なプログラム例として次のプログラムを挙げる．

危険な使い方

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

```
int main(){
    FILE * fp;
    fp = fopen("hoge.txt", "r");
    char buf[32]; // 現代の C 言語では関数の頭で宣言しなくても良い.
    fscanf(fp, "%s", buf);
    puts(buf);
    return 0;
}
```

1.1.1. 戻り値 /error_no を意識して

普段何気なく `fopen` や，`fprintf` などを使っていないだろうか．プログラム例では戻り値を無視しまっているが，非常に重要なものである．

(1) 戻り値は適切かを確認しよう

`fopen` はファイルを開くのに失敗すると，`NULL` を返す．開こうとしたファイルが存在しないことはよくあることで，正しくチェックすべきである．

`fopen` の戻り値を確認する．

```
// ファイル先頭
#include <stdlib.h>
//fp = fopen("hoge.txt", "r") == NULL を修正
if((fp = fopen("hoge.txt", "r")) == NULL) {
    puts("ERROR: Cannot open.");
    exit(EXIT_FAILURE);
}
```

普通 `NULL` は `0` であるが，これは標準 C には定義されていないので，正しく `NULL` と書くべきである．if 文の中で，エラーメッセージを表示して，`exit` 関数を呼び出している．エラーコードは `EXIT_FAILURE` である．`1` を入れる文献もあるが，同じ理由でしっかりと識別子を書くべきである．

`fscanf` についても同様に，チェックをしてみよう．open はできたものの書き込みに失敗することはたくさんある．

`fscanf` の戻り値を確認する．

```
// ファイル先頭
```

```
#include <stdlib.h>
//fscanf(fp, "%s", buf) を修正
if(fscanf(fp, "%s", buf) == EOF) {
    fprintf(stderr, "File write error.");
}
```

(2) エラーの内容を確認する

さて，ただファイルを開けないとエラー表示するだけではユーザに不親切である．エラーの情報が分かることが望ましい．そのためには，`errno.h` を用いる．

`errno` を確認する．

```
// ファイル先頭
#include <errno.h>
// 修正したものをさらに修正
if((fp = fopen("hoge.txt", "r")) == NULL) {
    puts("ERROR: Cannot open.");
    perror("file open error");
    exit(EXIT_FAILURE);
}
```

`perror` はエラーを出力する関数である．通常 `stderr` のバッファに上記の例では以下のように出力される．

`perror` の出力

```
file open error: File Not Found
```

エラーに関する文字列を返す関数 `strerror` も存在する．

`strerror`

```
char *strerror(int errnum);
int strerror_r(int errnum, char *buf, size_t buflen);   // XSI
char *strerror_r(int errnum, char *buf, size_t buflen); // GNU
```

`strerror` はそのままバッファへのポインタを返す．しかし，この関数はスレッドセーフでない場合があり，マルチスレッド環境での使用は推奨されない．一方，`strerror_r` は XSI/GNU 拡張である．`buf` にエラーメッセージを格納する．それぞれで定義が違うのは二つの拡張

によって使い方が異なる. XSI は成功すると 0 を返し, GNU は `buf` をそのまま返すか，エラーメッセージが格納された静的なバッファへのポインタを返す． 全く互換性がないことに注意しよう.

Microsoft C では，全く異なるアプローチ（C オプション標準なやり方）でこの問題に対処していることを後述する.

さらに，あなたが BSD や Windows のプログラマであり，やる気があるならば Exit Code をそのエラーにあったものに変えると良いだろう.（Linux では標準の Exit Code が存在しない．）

`exit` の Exit Code を変える

```
// ファイル先頭
#include <stdlib.h>
#ifdef _WIN32
#include <winerror.h>
#elif defined(__FreeBSD__) || (defined(__NetBSD__) || \
        defined(__OpenBSD__) || defined(__DragonflyBSD__) || \
        defined(__APPLE__)
#define EXIT_RETURN
#include <sysexits.h>
#endif
// 修正したものをさらに修正
if((fp = fopen("hoge.txt", "r")) == NULL) {
    printf("ERROR: Cannot open.\n");
    perror("file open error");
    if(errno == ENOENT)
#ifdef _WIN32
        exit(ERROR_FILE_NOT_FOUND);
#elif defined(EXIT_RETURN)
        exit(EX_OSFILE);
#else
        exit(EXIT_FAILURE);
#endif
    else
        exit(EXIT_FAILURE);
}
```

大抵のプログラマはこれを怠るが，普通のことなので，気にする必要はない.

(3) 最近の言語のエラーハンドリング

さて，あなたが C 言語のプログラマでない場合，すなわち最近の言語のプログラマである場合，事はもっと簡単である．

まず，ジャンプ例外機能のある言語だとどうだろうか．マルチスレッドのことなんか考えずにもっと簡潔に書くことができる．

ジャンプ例外機能の例： Ruby

```
begin
    file = File.open("hoge.txt", "r")
rescure => error
    p error
    exit(1) # 環境依存
end
```

ジャンプ例外機能の例： C#

```
try{
    var file = new System.IO.StreamReader("hoge.txt");
}catch(Exception ex){
    System.Console.Error.WriteLine(ex);
    System.Environment.Exit(1); // 環境依存
}
```

ファイルが存在するか否かを調べるために `fopen` を用いることがある． これは C 言語では正しい使い方であるが， ジャンプ例外機能のあるプログラミング言語では例外で条件分岐するのはやめるべきである． あくまで， エラーが起こった時のために使うべきである． 基本的に，ファイルの存在可否を調べる関数が備わっているはずであり， そちらを使うべきである． 一般的に，ジャンプ例外が発生した場合，プログラムのパフォーマンスはとても悪くなってしまう．ループで複数のファイルが存在するか否かを調べると， 例外を使った場合とそうでない場合で実行速度が大幅に異なる．また，可読性の観点からもやめるべきである．

次に， 直和型（Union）のある言語の場合を見てみよう． ファイルの読み書きで使われる直和型とは，例えば，Rust の `Result<File>` や Haskell の `IO モナド` である．特徴としては，普通はエラーハンドリングをしなければならない点である．C 言語やジャンプ例外機能のある言語は簡単にエラーハンドリングを無視することができる．一方で，これらの言語は意識してエラーハンドリングを無視しなければならない．例えば，Rust は `unwrap` 関数を用いて，

Union のある言語でエラーハンドリングを無視する例： Rust

```
let mut f = File::open("hoge.txt").unwrap();
```

と書かねばならない．この時，エラーが発生すれば panic を起こしてプログラムは即時強制終了される．プログラマが注意しなくても安全なプログラムが書ける一方で，気軽にプログラムが書けないことが欠点だろうか．しかし，モナドの力を使えば，簡単にエラー処理を呼び出し元に任せることができる．Rust だと `?` を付けることがそれに該当するだろう．

では，エラーハンドリングをするとどう書けるだろうか．様々なやり方があるが，Rust の場合は，`match` を用いると手続きのように書けて分かりやすいだろう．

Union のある言語での例： Rust

```
match File::open("hoge.txt") {
    Ok(file) => ..
    Err(ex) => {
        eprintln!("{}", ex);
        process::exit(1); // 環境依存
    }
}
```

他にも便利な関数が沢山あるので，使いこなすと楽しく書けるだろう．

1.1.2. ファイルの読み書きの際に気を付けること

では実際にファイルを読み込んでみよう．

(1) `fopen`

`fopen` にはモードがある．読み込みなら `r`，書き込みなら `w`，そして追記は `a` である．`+` は読み込みも書き込みもする場合に使われ，`w+` は既存のファイルを削除し，`r+` は既存のファイルが存在しないとエラーになる． そして，`a+` はあっても無くても良い． もし， テキストエディタではなくバイナリエディタを使うのであれば，`b` を用いる． バイナリモードであれば，CTRL+Z を `EOF` として認識しなくなる違いがある．（つまりファイルの最後まで読み込む．）

(2) `fscanf` のバッファオーバーラン

`fscanf` では書式文字列 `%s` を用いて文字列の読み取りをしているが，これは危険である．変数 `buf` は 32 バイト分の文字列を読み取ることができるが， これを超えた場合は， スタック上の他の変数を上書きしてしまう場合がある． つまり，`fscanf` には最大何文字読み出して良い

かを渡していないから，32 バイト分の文字列を超える大きさの文字列を読み出してしまう可能性がある．そして，その文字列を変数 `buf` に書き込むが，この時に，変数 `buf` の領域を超える場所に書き込んでしまう．これは，他の変数等を書き換えてしまう可能性がある．そのため，読み込むバイト数を指定するべきである．

読み込むバイト数を指定する

```
//fscanf(fp, "%s", buf); を修正
fscanf(fp, "%31s", buf);
```

また，C11 の Annex K Bounds-checking interfaces では，`fscanf_s` を利用することができる．（Microsoft C が対応．）

`fscanf_s` を利用する

```
//fscanf(fp, "%s", buf); を修正
fscanf_s(fp, "%s", buf, _countof(buf));
```

また，`fgets` が利用できるならば（複雑な処理でなければ），`fgets` を使った方が性能が良くなるので，検討の余地はあるだろう．

(3) お片付けは忘れずに

使ったリソースをこまめに使用後に片付けすることを心がけよう．ファイルの読み書きでは `fclose` を使用する．今回の例では，プログラム終了後，OS が片づけることになるが，大規模なプログラムを書くと，ファイルを頻繁に読み書きすることがあるだろう．そういう場合に，片付けしなければ，簡単にハードウェア資源を消費しつくしてしまう．

`fclose` を利用する

```
//return 文の前に挿入
fclose(fp);
```

ファイルの読み書き中にエラーハンドリング等で，`exit` 関数を実行する場合は，`atexit` 関数を利用すると良い．この関数は，プログラムの終了時に呼び出す関数を登録するものである．

`atexit` を活用して出来たプログラム

```
#include <stdio.h>
#include <stdlib.h>
#ifdef _WIN32
```

```
#include <winerror.h>
#elif defined(__FreeBSD__) || (defined(__NetBSD__) || \
        defined(__OpenBSD__) || defined(__DragonflyBSD__) || \
        defined(__APPLE__)
#define EXIT_RETURN
#include <sysexits.h>
#endif
FILE * fp = NULL;

void on_exit() {
    if(fp != NULL)
        fclose(fp);
}

int main(){
    if((fp = fopen("hoge.txt", "r")) == NULL) {
        printf("ERROR: Cannot open.\n");
        perror("file open error");
        if(errno == ENOENT)
    #ifdef _WIN32
            exit(ERROR_FILE_NOT_FOUND);
    #elif defined(EXIT_RETURN)
            exit(EX_OSFILE);
    #else
            exit(EXIT_FAILURE);
    #endif
        else
            exit(EXIT_FAILURE);
    }
    atexit(on_exit); // 終了時に on_exit を呼び出す
    char buf[32]; // 現代の C 言語では関数の頭で宣言しなくても良い.
    fscanf(fp, "%31s", buf);
    puts(buf);
    // ...
    return 0;
}
```

1.1.3. 補講 malloc もエラーチェックをしよう

もう一つ，特にエラーチェックしたい関数がヒープアロケーション関係の関数である．malloc，calloc，realloc 等のアロケーションをする関数は，利用可能な空きメモリがない場合に NULL を返す．

NULL と malloc

```
char * p = (char *)malloc(sizeof(char) * 32);
if(p == NULL) {
    fputs("malloc failed.", stderr);
    exit(EXIT_FAILURE);
}
```

64bit 環境の贅沢にメモリが使える環境であれば大丈夫だが，マイコンや，16bit，8bit 環境であれば，アロケーションに失敗する可能性は高まる．それらの環境では特に気を付けるべきである．

そして，片付けとして必ず free 関数を呼び出すことを忘れてはいけない．アロケーションをたくさんして，不要になったのに解放しないことをメモリリークと呼ぶ．連続稼働するサーバプログラムや IoT 機器等では，メモリリークするプログラムを混入させないことが非常に重要である．

free

```
free(p);
```

さらに，一度解放した領域を利用したり，もう一度解放させてはならない．前者を use-after-free，後者を double-free と呼ぶ．use-after-free は，解放後の領域は malloc によってまた別の場所で使われている可能性があるからである．その領域を上書きすれば，どこか別の関数で想定外の値の変化として障害が起きてしまう．一方，double-free は実装によってダメになる仕組みがあるので，ネットで調べてみると良い．double-free は侮ってはいけない．double-free を利用した権限昇格の脆弱性の例もあり，気を付けるべきである．

さて，対処法としては一つ，free の後に必ず NULL を代入することが挙げられるだろう．ポインタを別の場所にコピーしていた場合防げないが，マシにはなるだろう．

NULL-after-free

```
free(p);
p = NULL;
```

1.1.4. 補講C11 の Annex K 拡張

さて，今まで本書では C11 の Annex K 拡張を利用してきたが，利用は推奨されない．なぜなら，Annex K 拡張を正しく実装したコンパイラは存在せず（提案した Microsoft Visual C++ でさえ！），将来のバージョンの C 言語で使用を削除することが提言されているからである．GNU C の実装者の提出したレポート[*1]では，書きづらいことや，長さを渡すときに，`sizeof` や `strlen` の使い方を誤ったりすることがあることを指摘している．（Annex K に限らず，`sizeof` や `strlen` の使い方を間違える人が多いでしょと思ったりもしますが ...）そのレポートでは，代替案としてアドレスサニタイザや Intel MPX（最新の CPU ではもはや使えない）を推奨しているが，オーバーヘッドが大きい．GCC の人たちはもっと真面目に考えるべきではないだろうか．なので，セキュアに動くプログラムを作りたければ，C++ や OS の API の利用を検討した方が良さそうである．

1.2. 標準ライブラリの先にあるもの

さて，標準ライブラリの関数の先にあるものを考えたことがあるだろうか．例えば，ファイルの読み書きをする時には，もちろん記憶装置にデータを読みに行く必要がある．そんな大変な処理は誰がしてくれるのでしょう -- そう，OS です．

1.2.1. Operating System, Operation System

Operating System はハードウェアを抽象化し，リソースを管理するプログラムのことだ．例えば，SSD 上のファイル読み込みをする際，あなたは SSD のデータシートを読んだだろうか．そうではなくて，標準 C ライブラリの仕様書（あるいは入門書）を読んで実装したはずだ．そのように実装すれば，あなたの SSD が HDD に変わったって，データの読み書きができる．さらに，NFS などを用いてネットワーク上のアイテムにアクセスすることができるはずである．このハードウェアの違いを吸収することが OS の一つ目の役割であるハードウェアの抽象化である．

そして，OS はリソースを管理する．リソースとは，ファイルだけでなく，プロセスやメモリも含まれる．あなたのプログラムでメモリを読み書きする時，あなたのプログラムは他のプログラムのメモリを読み書きすることができるだろうか．リソースを OS がうまく管理することで，効率よくマシンを動かすことができ，あなたのプログラムのメモリの中のデータを他のプログラムから守ることができる．

1.2.2. カーネル，Kernel

OS の核の部分がカーネル，kernel である． カーネルは OS とその上で動くプログラムが動くための基本的なメカニズムを提供する． カーネルはユーザプログラムや OS のプログラムと

---

1 Carlos O'Donell, Martin Sebor:Updated Field Experience With Annex K — Bounds Checking Interfaces, http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1969.htm

ハードウェアの仲介をする役割を持つ．一般的に，プログラムが動作する情報の集合体である process，プロセスの管理，プロセスとプロセスがやり取りするための Inter-Process Communication, IPC，プロセス間通信の機能，そしてハードウェアにアクセスするための手段を提供する．

一つ，重要なのがカーネルランドと，ユーザランドである．カーネルランドは全てのハードウェアにアクセスできる特権モードで実行されるプログラムの範囲をさす．カーネルは通常カーネルランドで実行される．一方，ユーザランドは非特権モードで実行される．すなわちユーザランドでハードウェアを触りたい場合，カーネルに処理を依頼する必要がある．

1.2.3. システムコール，system call，API

さて，例えばファイルの読み書きには記憶装置を操作することが不可欠である．ユーザランドで動く標準ライブラリはどのようにして処理を依頼しているか，それはシステムコールを利用している．システムコールと言われて難しく感じるかもしれないが，普通の関数呼び出しとそう変わりはない．やり方は CPU によって異なるが，例えば，システムコール用の命令があったり，割り込みを発生させたりする．（Intel x86 上で動作する OS はその昔，システムコール用の命令があったにもかかわらず，速いからという理由で例外割り込みを用いてシステムコールを実現していた．）システムコールはカーネルによって異なる．したがって，標準 C ライブラリはカーネル &OS 間の差異を埋めているのである．また，システムコールや標準ライブラリなどの関数の集合のことを API と呼ぶ．開発者は OS（カーネル）やライブラリの API を用いてプログラミングを行っている．

カーネルの持つシステムコールを呼び出した時，ユーザランドから，カーネルランドに切り替えてカーネルが実行する．すなわち，CPU の動作モードが変更されて処理が実行される．システムコールを沢山呼ぶと，オーバーヘッドが発生し速度が低下してしまう．したがって，システムコールが少ないほど，処理の速い OS となる．例えば，Linux ではファイルの読み込みを一つのシステムコールでできる `readfile` を追加した．従来 `open`，`read`，`close` と 3 つのシステムコールを呼ぶことでできていた処理を，1 つのシステムコールで行うことができ，実行速度の向上が期待できる．

1.2.4. システムコールを覗いてみよう

では，ファイルを開いて読むだけのプログラムのシステムコールを呼ぶ様子を覗いてみよう．次のようなプログラムを用意しよう．

hoge.c

```
#include <stdio.h>

int main(void) {
```

```
    char buf[16];
    FILE * fp = fopen("hoge.c", "r");
    fgets(buf, sizeof(buf), fp);
    fclose(fp);
    return buf[0];
}
```

(1) Linux

このプログラムをコンパイルして，`strace` と共に実行してみよう．

shell

```
$ strace a.out
```

次のように表示されたはずである．

`strace a.out` with linux-5.4 and clang-10.0

```
.... いろいろ
openat(AT_FDCWD, "hoge.c", O_RDONLY)    = 3
fstat(3, {st_mode=S_IFREG|0664, st_size=163, ...}) = 0
read(3, "#include <stdio.h>\n\nint main(voi"..., 4096) = 163
close(3)                                 = 0
```

`strace` は呼ばれたシステムコールを表示してくれるプログラムである．`openat`，`fstat`，`read`，`close` というシステムコールが呼ばれてるのが確認できる． それぞれは英語の意味の通りの動作をする関数であり，Linux のシステムコールである．https://linuxjm.osdn.jp/html/LDP_man-pages/man2/ を見ると，Linux にある全てのシステムコールのドキュメントを見ることができる．興味があればそれを見ると良いだろう．（実際に，C 言語から直接呼び出してみても良いだろう．）

(2) XNU(macOS)

このプログラムをコンパイルして，`dtruss` と共に実行してみよう．

shell

```
# dtruss -c ./a.out
```

次のように表示されたはずである．

sudo dtruss -c ./a.out with Darwin 19.6.0 and Clang 12.0

```
.... いろいろ
open_nocancel("hoge.c\0", 0x0, 0x1B6)  = 3 0
fstat64(0x3, 0x7FFEE6C8B9A8, 0x0)  = 0 0
read_nocancel(0x3, "#in...\n\0", 0x1000)  = 163 0
lseek(0x3, 0xFFFFFFFFFFFFFF6C, 0x1)  = 15 0
close_nocancel(0x3)  = 0 0
.... いろいろ
```

`dtruss` は呼ばれたシステムコールを表示してくれるプログラムである．`open_nocancel`，`read_nocancel`，`close_nocancel` というシステムコールが呼ばれてるのが確認できる．それぞれは英語の意味の通りの動作をする関数であり，XNU(FreeBSD) のシステムコールである．https://github.com/apple/darwin-xnu/blob/master/bsd/kern/syscalls.master を見ると，XNU にある全てのシステムコールの一覧を見ることができる．Apple のドキュメントで検索すれば出てくるので，それを使ってみても良いだろう．

(3) NT(Windows)

このプログラムをコンパイルして，NtTrace と共に実行してみよう．

NtTrace は github においてあり，ビルドに Visual Studio 2017 以降が必要である．プロジェクトの URL は https://github.com/rogerorr/NtTrace である．スタートメニューから Developers CommandPrompt を開き，保存したフォルダに行って，`namake NtTrace.mak` を実行することでビルドできる．`NtTrace.exe` ができた実行ファイルなので，それを用いて

shell

```
> NtTrace.exe hoge.exe
```

と実行すれば次のような行が表示されたはずである．

NtTrace.exe hoge.exe with Nt Build 19042 and Visual C++ 19.28.29334

```
NtCreateFile( FileHandle=0x7829cff968 [0x78], DesiredAccess=SYNCHRONIZE|
GENERIC_READ|0x80, ObjectAttributes=0x48:"hoge.c", IoStatusBlock=0x7829cf-
f970 [0/1], AllocationSize=null, FileAttributes=0x80, ShareAccess=3, CreateDis-
position=1, CreateOptions=0x60, EaBuffer=null, EaLength=0 ) => 0
NtReadFile( FileHandle=0x78, Event=0, ApcRoutine=null, ApcContext=null,
IoStatusBlock=0x7829cffbd0 [0/0xaa], Buffer=0x22969d69470, Length=0x1000,
```

```
ByteOffset=null, Key=null ) => 0
NtClose( Handle=0x78 ) => 0
```

`NtTrace` は呼ばれたシステムコールを表示してくれるプログラムである．`NtCreateFile`，`NtReadFile`，`NtClose` というシステムコールが呼ばれてるのが確認できる． それぞれは英語の意味の通りの動作をする関数であり，NT のシステムコールである．https://j00ru.vexillium.org/syscalls/nt/64/ を見ると，NT にある全てのシステムコールの一覧を見ることができる．Windows Driver Kit を使うと実際に自分で呼び出してみることができる．（または，Wine のヘッダファイルを流用するのも手である．）

1.2.5. 多様性のある OS の世界

さて，全ての OS が異なるシステムコールでファイルの読み書きをしていることがわかるだろうか．このように，OS は三者三様であるが，大きく 4 つの種別に分けることができる．

(1) モノリシックカーネル

モノリシックカーネルは，全ての OS の機能をカーネルが担う．この方法では次のような利点がある．

(1) 速い
(2) すぐに動くものが書ける
(3) 組み込み向けの機能が少ないマイコン上でも動作する
(4) 低機能なときはバグが少なく，容量も小さい

しかし，様々な機能を追加してコードが肥大化すると，次にあげるような欠点が生まれてしまう．

(1) 保守が困難
(2) コードの結合度が強く，分割しづらい
(3) 移植性が低く，カスタマイズしづらい
(4) 特定の機能だけをアップデートすることが困難

また，バグを作ってしまうと，そのバグがセキュリティ上の問題やフリーズ，リブート，ユーザのプログラムを壊したりしてしまう． これは， カーネルはカーネルランドで動作する， つまり特権モードで動くので，一つのバグが OS 全体に多大な影響を及ぼしてしまうからである．

モノリシックカーネルは，MS-DOS(Windows), Linux, FreeBSD, Solaris, NetBSD, OpenBSD, FreeRTOS などがある．あのでかいプログラムを保守している Linus Tordvalds 氏，マジやべぇ

(2) マイクロカーネル

一方，マイクロカーネルは様々な機能がたくさんのサーバに分割されている．マイクロカーネルはただ，プロセス管理，プロセス間通信，そしてハードウェアを操作する方法のみを提供する．マイクロカーネルは次の利点がある．

(1) 保守が容易

(2) セキュリティ的に有利

(3) 特定の機能だけをアップデートすることが可能

(4) 動作中に機能をアップデートすることができる

(5) 冗長化することでバグ耐性に強い

欠点は次の通りである．

(1) システムコールの呼び出し回数が多くなり，処理速度が低下する

(2) メモリ使用量が増大する

QNX や Mach, GNU Hurd がこれに当たる．

(3) ハイブリッドカーネル

また，マイクロカーネルとモノリシックカーネルの中間としてハイブリッドカーネルがある．処理を早くしたい機能をカーネルに含めることで速度低下を防ぐ．

NT(Windows), XNU(macOS, 自称ナノカーネル), NewOS(Haiku, 自称ナノカーネル), Fuchia, DragonflyBSD がこれに当たる．だから，Linux でプリンタするときにドライバにバグがあったらカーネルごと落ちるのに対し，Windows は落ちない．（グラフィクスドライバはブルースクリーンになるが，レジストリを変更して回避することができる．）

(4) Exokernel

ユーザのプログラムとカーネルの大部分 (Library OS) をくっつけて，その下で動くのが Exokernel である．Exokernel はハードウェアを保護する機能だけを提供し，ユーザランドで直接ハードウェアとやり取りするユーザのプログラムが動く．

L4 がこれである．また，Windows Linux Subsystem や DragonflyBSD の vkernel はこれと同じやり方で実現されている．