

システムソフトウェア

2021年度

第12回 (11/15)

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

- 保護機構とセキュリティ

保護機構の目的

- データの正しい分離と共有
 - － 分離
 - 例：自分のデータを他人に勝手に読み書きされないようにしたい。
 - － 共有
 - 例：他人とデータの共有や通信をしたい。
- どちらか一方のみであれば簡単。分離しつつ一部は共有したいという要求をみたす必要がある。

外部デバイス（二次記憶装置等）

- 外部デバイスそのものの保護
 - － 正規のシステムコールを介さずに直接外部デバイスにアクセスしようとするプロセスからの保護
- ファイルシステムの保護
 - － ディスクの暗号化
 - － ファイルのアクセス制御

外部デバイスの保護(1)

- CPUの外部デバイスへのアクセス方式
 - － I/O命令
 - 例：x86の in, out 命令
 - － 外部デバイスに割り当てられたアドレスの読み書き (メモリマップドI/O)
- 保護方式
 - － I/O命令は特権命令
 - － メモリマップドI/Oの場合，デバイスコントローラが割り当てられたメモリ領域を特権モードでのみアクセス可能としておく．

外部デバイスの保護(2)

- ユーザプロセスが勝手に外部デバイスにアクセスすることはできない.
 - I/OマップドI/O の場合：I/O命令は特権命令なので、そもそも実行できない（実行しようとするするとトラップがかかる）.
 - メモリマップドI/O の場合：デバイスコントローラに割り当てられたメモリ領域は一般にユーザプロセスのメモリ空間にわりあてられることはない.

ファイルシステムの保護

- ファイルシステムのアクセス制御
 - どのような場合に、 ファイルをアクセスするシステムコールが成功するかを決める.
- アクセス制御を行うシステムコール
 - ファイルの読み書き
 - open, read, write, etc.
 - プログラムの実行
 - execv, etc.

アクセス制御 (access control)

- 誰が何に対してどのようなことができるかを定め、検査すること.
 - － 誰：主体(subject)
 - － 何：対象(object)
 - － どのようなこと：操作(method)
- 例: ファイルシステムの場合
 - － 主体：プロセス (ユーザ)
 - － 対象：ファイル
 - － 操作：読み出し, 書き込み, 実行, アクセス権変更, 所有者変更, etc.

保護ドメイン(Protection Domain)

- アクセス権(access right)の集合で、その中で動作するプロセスができることを規定する.
- アクセス権はオブジェクトとそれに対して許可される操作の集合の組である.
 - $\langle \text{オブジェクト名}, \{ \text{操作}, \dots \} \rangle$
- 例
 - $D1 = \{ \langle F1, \{r,w\} \rangle, \langle F2, \{r\} \rangle, \langle F3, \{r,x\} \rangle \}$
 - $D2 = \{ \langle F2, \{r,w\} \rangle, \langle F3, \{r,w,x\} \rangle \}$

保護ドメインの例

- ユーザ/グループ
 - Unixでは、ユーザやグループがドメインを規定するものの代表である
 - SELinux等では異なるドメイン規定方式も採用されている
 - setuidビットなどの機構により、一時的にドメインを切り替えることも可能である。
- プロセス

アクセス制御行列

- 各ドメインで、どのオブジェクトにどの操作ができるかを記述した表（行列）

	File ₁	File ₂	Service ₁	Service ₂
Alice	rd, wrt, del, exe, chmod, chown	rd, app		use, start, stop
Bob	rd, exe	rd, wr, app, del, chmod, chown	use, start, stop, log	use, start, stop, log
Process ₁	rd, exe		use	use

アクセス制御行列の実現

- 大域テーブル
 - － アクセス行列を表す大域的な表を作り，各プロセスはその表を参照する.
 - － 表の規模が大きくなり，効率的な実装が困難.
- アクセス制御リスト (ACL)
- ケーパビリティ

アクセス制御リスト (ACL)

- オブジェクト毎に定められたアクセス権の列
- アクセス制御行列の列を取り出したもの
 - － 例：ACL(File2) =
{ (Alice, {rd, app}), (Bob, {rd, wr, app, del, chmod, chown}), (Chris, {}) }
- macOS, Windows, Linux (カーネル2.6以降)では実装されている.
 - － API: POSIX ACL

ケイパビリティ (Capability)

- ドメイン毎の(オブジェクト,アクセス権)の列
- アクセス制御行列の行を取り出したものとみなすことができる.
 - 例: Capability(Alice) =
{(File₁, {rd, wr, del, exe, chmod, chown}), (File₂, {rd, app}), (Service₁, {}), (Service₂, {use, start, stop})}
- 可能な操作をインターフェースとする抽象データとして実現され, そのデータ(への参照)を持っていることが, そのケイパビリティをもつことになる.

Unixにおける保護ドメイン

- 伝統的なUnixにおける保護ドメインは以下の値によって定まる
 - － ユーザid
 - － グループid

ユーザid (uid)

- ユーザ毎に関連づけられた値
 - 正整数値(uid_t型)
 - ユーザ名(文字列)とは別
 - 各ファイルには, 所有者(owner)としてユーザidが一つ関連づけられている.
 - 各プロセスには, 実ユーザid(real uid)と実効ユーザid(effective uid)が関連づけられている.
 - uid_t getuid(void);
 - uid_t geteuid(void);

グループid (gid)

- 1人以上のユーザから構成されるグループに関連づけられた値
 - 各ユーザは一つ以上のグループに所属する
 - ユーザ名と同様にグループも名前（文字列）をもつ
 - グループidも正整数値(gid_t)
 - 各ファイルは所有者と同様に所属グループのgidが一つ関連づけられている
 - 各プロセスにも同様に， 実グループidと実行グループid(egid)が関連づけられている
 - `gid_t getgid(void);, gid_t getegid(void);`

ファイルのアクセス制御

- 各ファイルは、所有者、所属グループ、それ以外について、それぞれどのような操作ができるかが定められている。
 - － 操作：read, write, exec
 - － ls -l でアクセス権をみることができる.
 - － chown, chgrp, chmod コマンドで変更できる
- ファイルのアクセス権を変更できるのはファイルの所有者
 - － 所有者：ファイルに関連づけられたuidのユーザ
 - － 特権ユーザ(root)は所有者を変更できる.

chown, chgrp

- ファイルに関連づけられたuidやgidを変更
- 例
 - `chown takuo foo.c`
 - `foo.c`の所有者をtakuoにする
 - `chgrp staff project`
 - `project`のグループをstaffにする
- `-R` オプションでディレクトリのその下のファイルのuid/gidをまとめて変更できる

chmod

- chmod 644 file
 - 644は8進数で表したファイルのアクセス権限
 - rwx rwx rwx (r : 読み出し, w : 書き込み, x : 実行)
 - 上3ビットはファイルの所有者であるユーザの権限
 - 中3ビットはファイルに関連づけられたグループに所属するユーザの権限
 - 下3ビットは所有者, グループメンバー以外のユーザの権限
 - $644(8) = 110100100(2)$
 - 110 : 所有者は読み出しと書き込みができるが実行はできない
 - 100 : グループおよびその他のユーザは読み出しのみ
- chmod go-wx
 - グループ(g)とその他(o)から読み出し(w)と実行(x)権限を取り去る

- `chmod 640 foo`
 - 所有者：読み出し，書き込み
 - グループ：書き込みのみ
 - それ以外：アクセス不可
- `chmod 755 foo`
 - 所有者：読み出し，書き込み，実行
 - それ以外：読み出し，実行
- `chmod ugo+x foo`
 - 所有者，グループ，およびそれ以外の全ユーザに実行可能権限を付加

プロセスのuid/gid (1/2)

- 「ユーザAはファイルFを変更できる」という場合、実際にFを変更するのはAの代理となるプロセスである。
 - プロセスのuid：代理となっているユーザのuid
- 実uid
- 実効uid (euidとも呼ばれる)
 - アクセス権の制御はこちらに対して行われる。

プロセスのuid/gid (2/2)

- forkで生成された子プロセスは親プロセスと同じ uid, gid
- setuid, setgid等のシステムコールで、プロセスは自分の uid, gid を変えることができる.
- execXX が実行されると、プロセスは実行前と同じ uid, gid で実行される.
 - ただし setuid bit が指定された実行可能ファイルについてはその uid が euid になる.

eid/egidの変更

- `chmod 4755 foo`
 - `foo` を実行可能(755)にし, かつ実効uidをこのファイルの所有者とする
 - `foo`を実行するプロセスの実uidは実行したユーザのものであるが, 実効uidはこのファイルの所有者のものとなる
- `chmod 2755 foo`
 - `foo` を実行可能(755)にし, かつ実効gidをこのファイルのグループとする
- `chmod 6755 foo`
 - 上記2つを同時に指定

セキュリティ

- 3つの要素 (CIA)
 - － 秘密性 (Confidentiality)
 - 対象に対して権限のない主体がアクセスできないようにすること.
 - － 一貫性 (Integrity)
 - 対象の内容やメタデータが改竄されていないこと.
 - － 可用性 (Availability)
 - ある対象にアクセス権限がある場合はいつでもアクセスできること.

脅威と攻撃

- 脅威 (threats)
 - － セキュリティを脅かす可能性
 - － 例：セキュリティホール
- 攻撃 (attack)
 - － セキュリティを脅かす行為
 - － 例：セキュリティホールを利用した不正アクセス

攻撃の種類

- 秘密性を脅かす攻撃
 - － データの不正アクセス(読み出し)
 - － 盗聴(wiretapping)
- 一貫性を脅かす攻撃
 - － データやプログラムの改竄
 - － 成りすまし(masquerading)
- 可用性を脅かす攻撃
 - － DoS(Denial of Service)攻撃
 - － 分散DoS攻撃

セキュリティポリシーとメカニズム

- セキュリティポリシー
 - あるシステム(OS, ネットワーク, etc.)において, どの主体がどの対象に対して何ができて何ができないかを定義したもの.
- セキュリティメカニズム
 - システム内のあらゆるアクションに対し, それがセキュリティポリシーをみたすことを保証するための具体的な仕組み.
 - ポリシーの強制(enforcement)

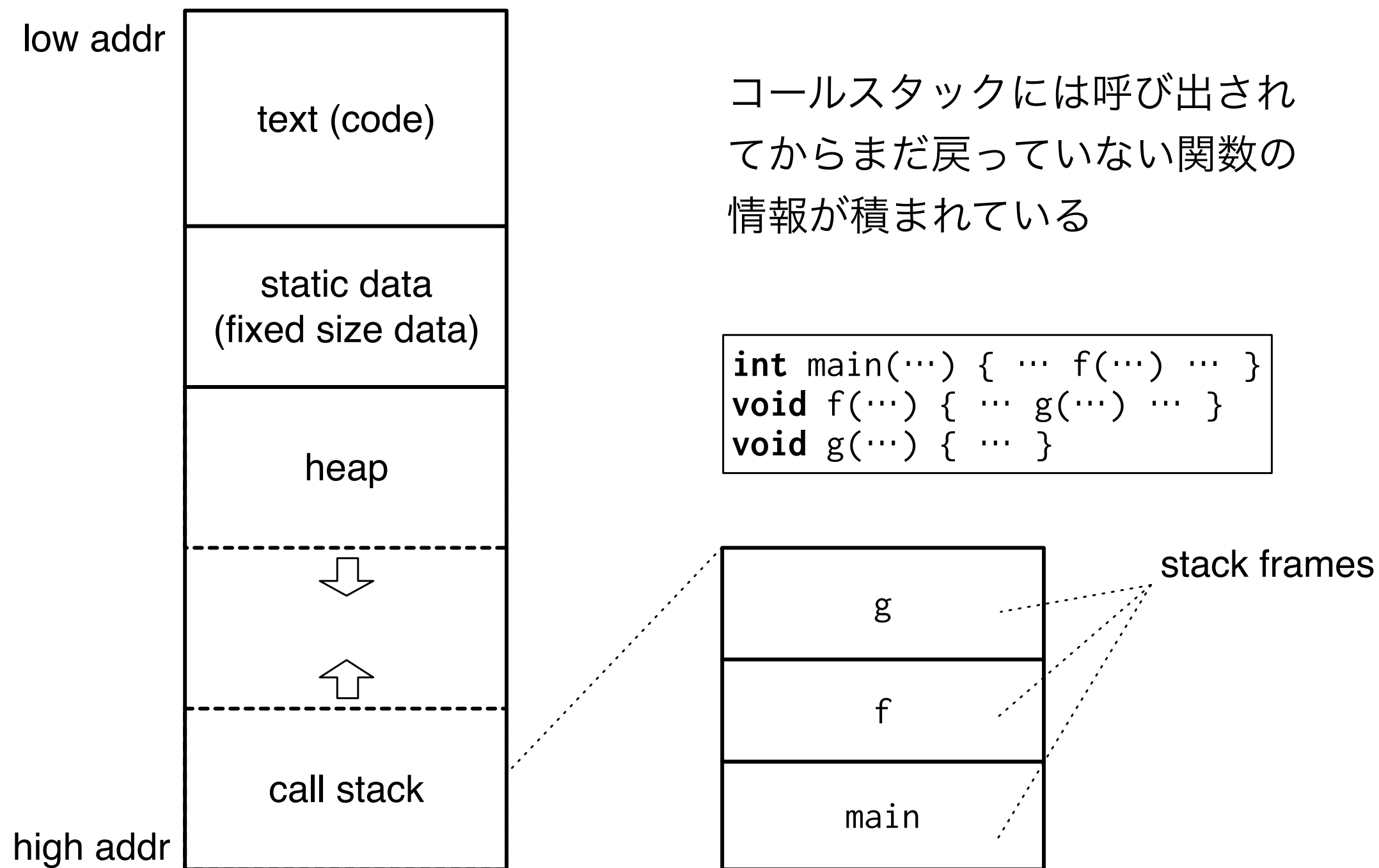
セキュリティの目的

- 検知 (Detection)
 - － セキュリティポリシーに違反した行為の検知
- 防止 (Prevention)
 - － セキュリティポリシーに違反した行為の防止
- 復帰 (Recovery)
 - － 攻撃を止め、被害状況を判断(assess)し、回復すること。

バッファオーバーフロー攻撃

- 配列等のメモリ領域の境界検査が正しく行われていないバグ
 - － 一般にC/C++では境界検査はプログラマの責任
- 攻撃手法の「古典」
 - － Robert Morrisのインターネットワーム(1988)
- 現在でも脅威のひとつ
 - － 例) CVE-2021-3156 (Jan. 2021)
 - Heap-Based Buffer Overflow in Sudo

プロセスのメモリレイアウト

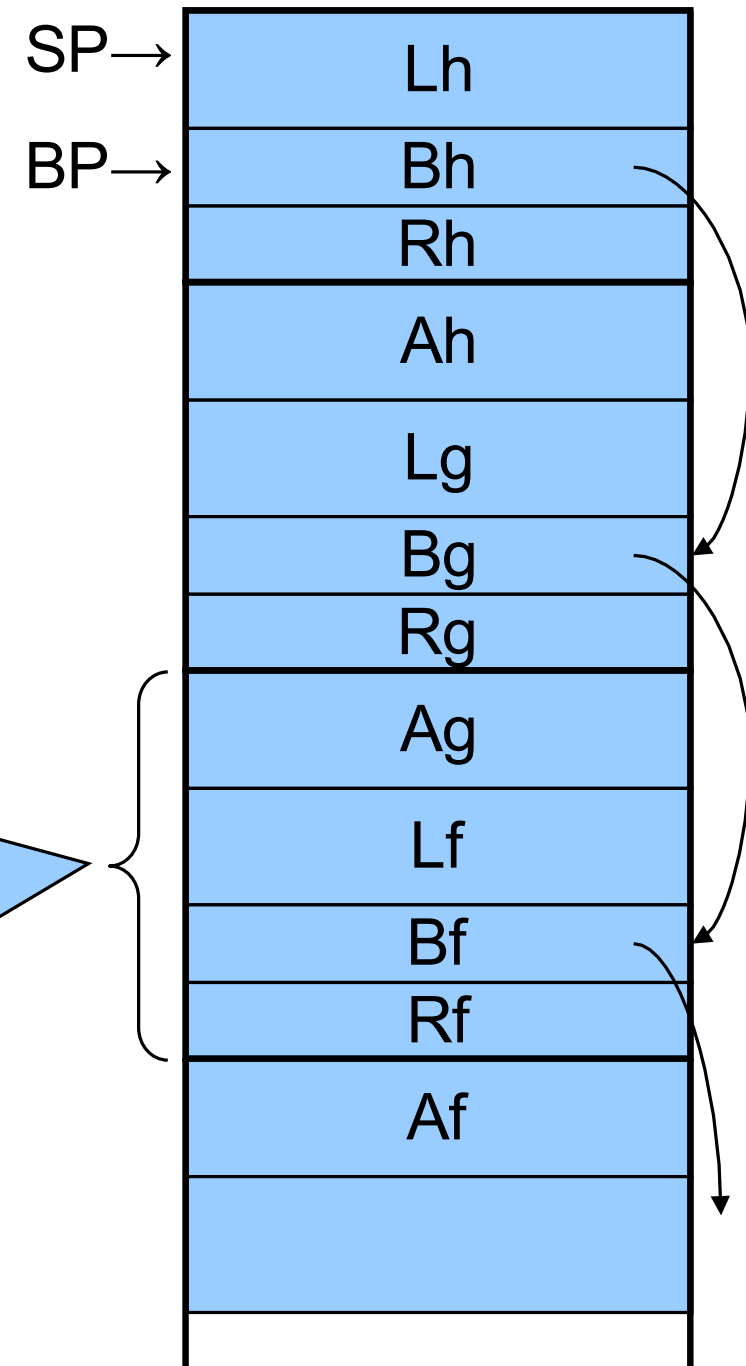


典型的なプロセスのメモリレイアウト

コールスタック

```
void f(...) { g(...); }  
void g(...) { h(...); }  
void h(...) { ... }
```

フレーム:
戻り番地(R)
ベースポインタ(B)
局所変数(L)
関数の実引数(A)



関数のコンパイル例

f:

```
pushl %ebp
movl  %esp, %ebp
subl  $8, %esp
```

```
movl  12(%ebp), %eax ; EAX = y
addl  8(%ebp), %eax  ; EAX = EAX + x
movl  %eax, -4(%ebp) ; z = EAX
movl  12(%ebp), %edx ; EDX = y
movl  8(%ebp), %eax  ; EAX = x
subl  %edx, %eax     ; EAX = EAX - EDX
movl  %eax, -8(%ebp) ; w = EAX
movl  -4(%ebp), %eax ; EAX = z
imull -8(%ebp), %eax ; EAX = EAX * w
```

```
movl  %ebp, %esp
pop   %ebp
ret
```

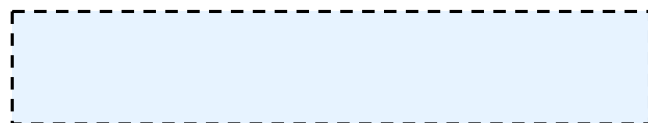
```
int f (int x, int y) {
    int z = x + y;
    int w = x - y;
    return z * w;
}
```

関数呼び出しの動作

f:

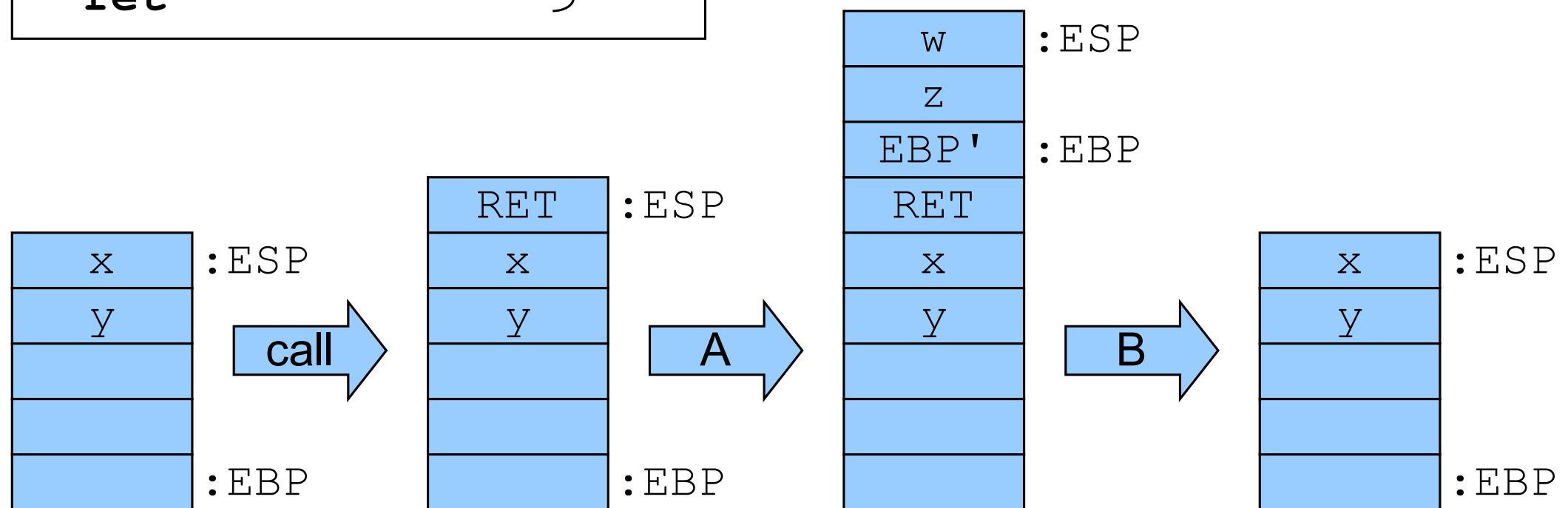
```
pushl %ebp  
movl  %esp, %ebp  
subl  $8, %esp
```

} A

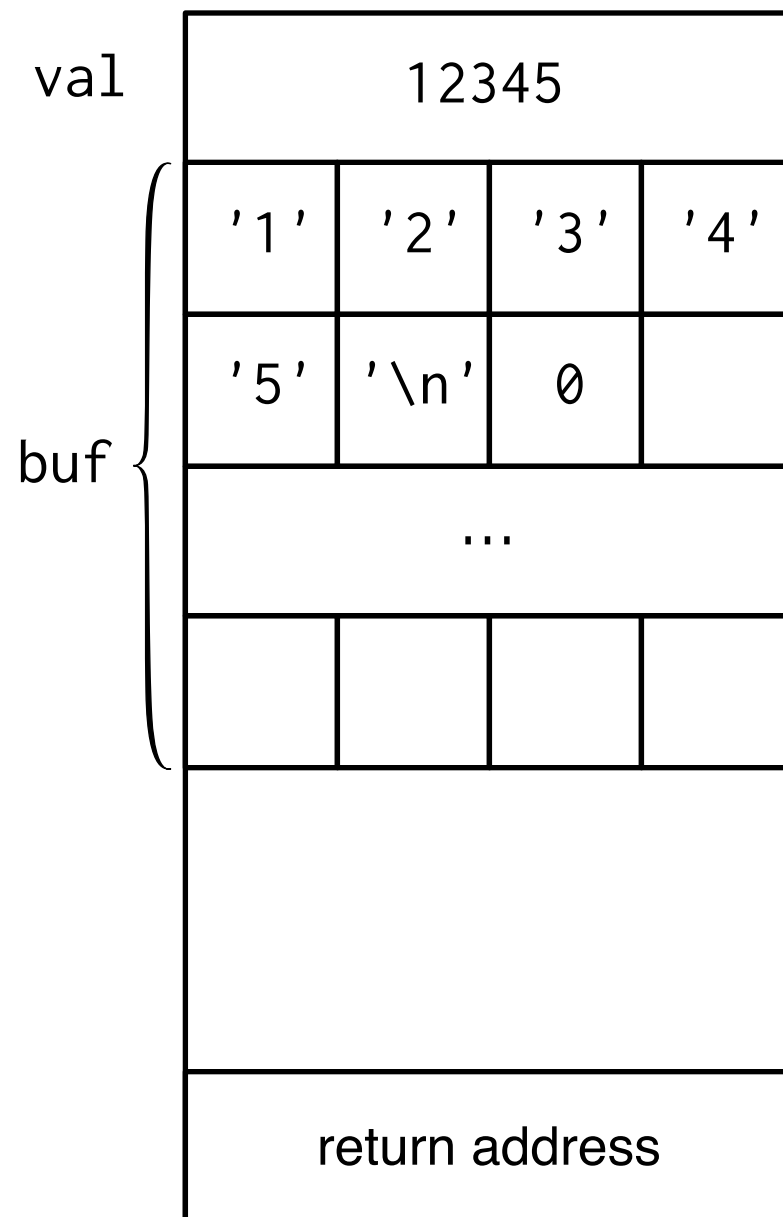


```
movl  %ebp, %esp  
pop   %ebp  
ret
```

} B



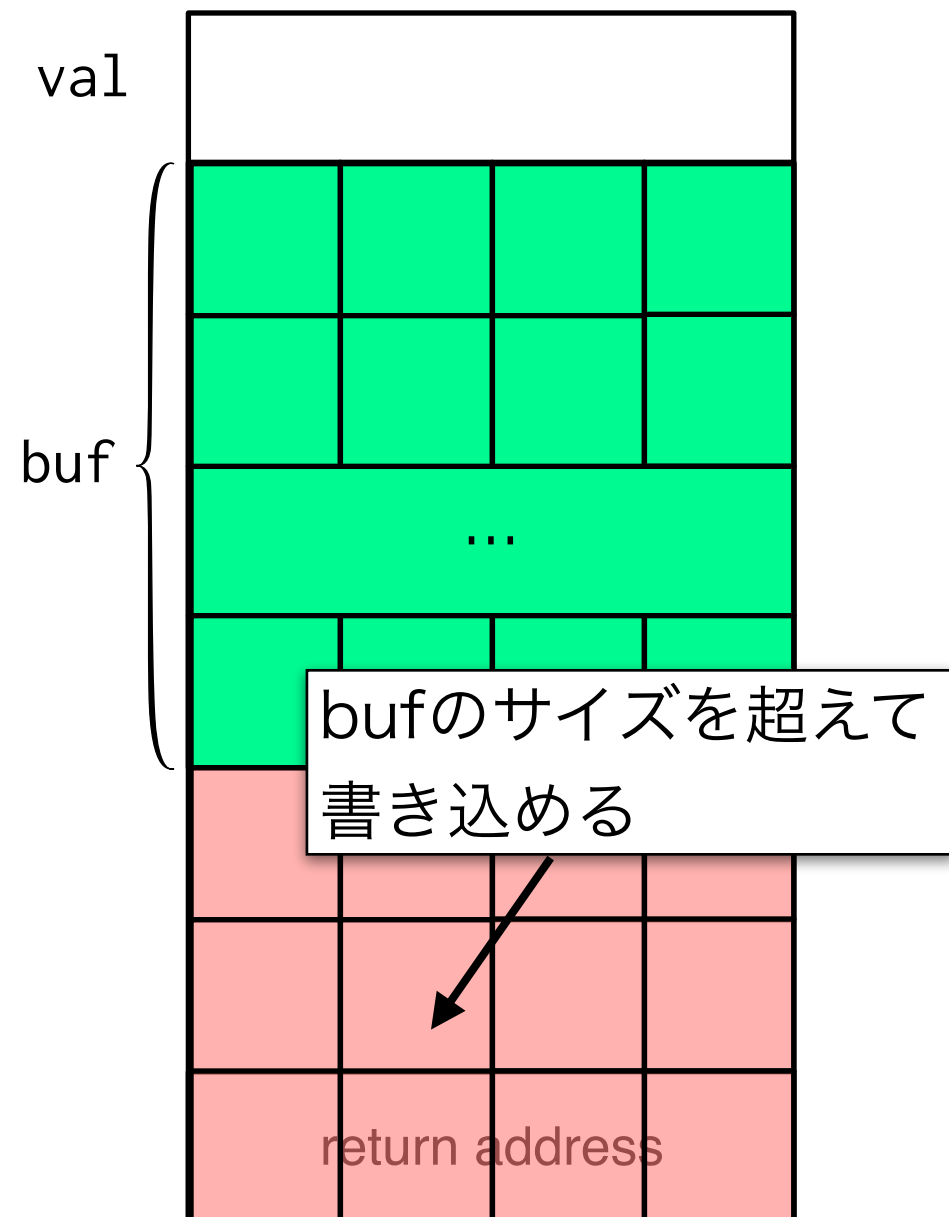
スタックフレーム



```
int f() {  
    int val;  
    char buf[64];  
    gets(buf);  
    val = atoi(buf);  
    return val * val;  
}
```

スタックフレームには、対応する関数の局所変数（局所配列）に加え、退避されたレジスタ値、戻りアドレス等が格納されている。

gets

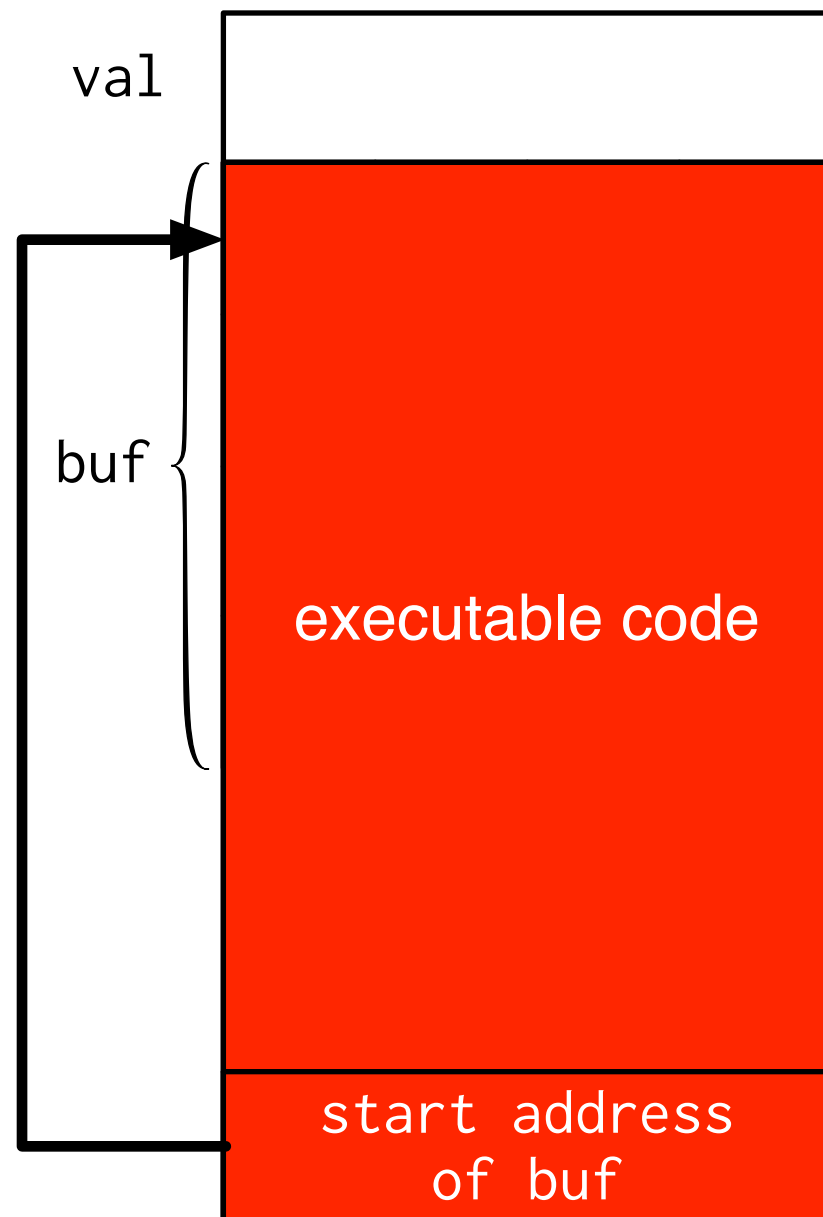


```
int f() {  
    int val;  
    char buf[64];  
    gets(buf);  
    val = atoi(buf);  
    return val * val;  
}
```

```
char *gets(char *buf) {  
    char *s = buf;  
    int c = getchar();  
    if (c == EOF) return NULL;  
    while (c != '\n' && c != EOF) {  
        *s++ = c;  
        c = getchar();  
    }  
    *s = '\0';  
    return buf;  
}
```

配列サイズのチェックは行われていない

スタック破壊攻撃



```
int f() {  
    int val;  
    char buf[64];  
    gets(buf);  
    val = atoi(buf);  
    return val * val;  
}
```

```
// an exploit that starts a shell  
shellcode: xorl    %eax, %eax  
            xorl    %edx, %edx  
            jmp     lblb  
lbla:       popl    %ebx  
            movb    %al, 7(%ebx)  
            movl    %ebx, 8(%ebx)  
            movl    %eax, 12(%ebx)  
            leal    8(%ebx), %ecx  
            movb    $11, %al  
            int     $0x80  
lblb:       call    lbla  
            .string "/bin/sh"
```

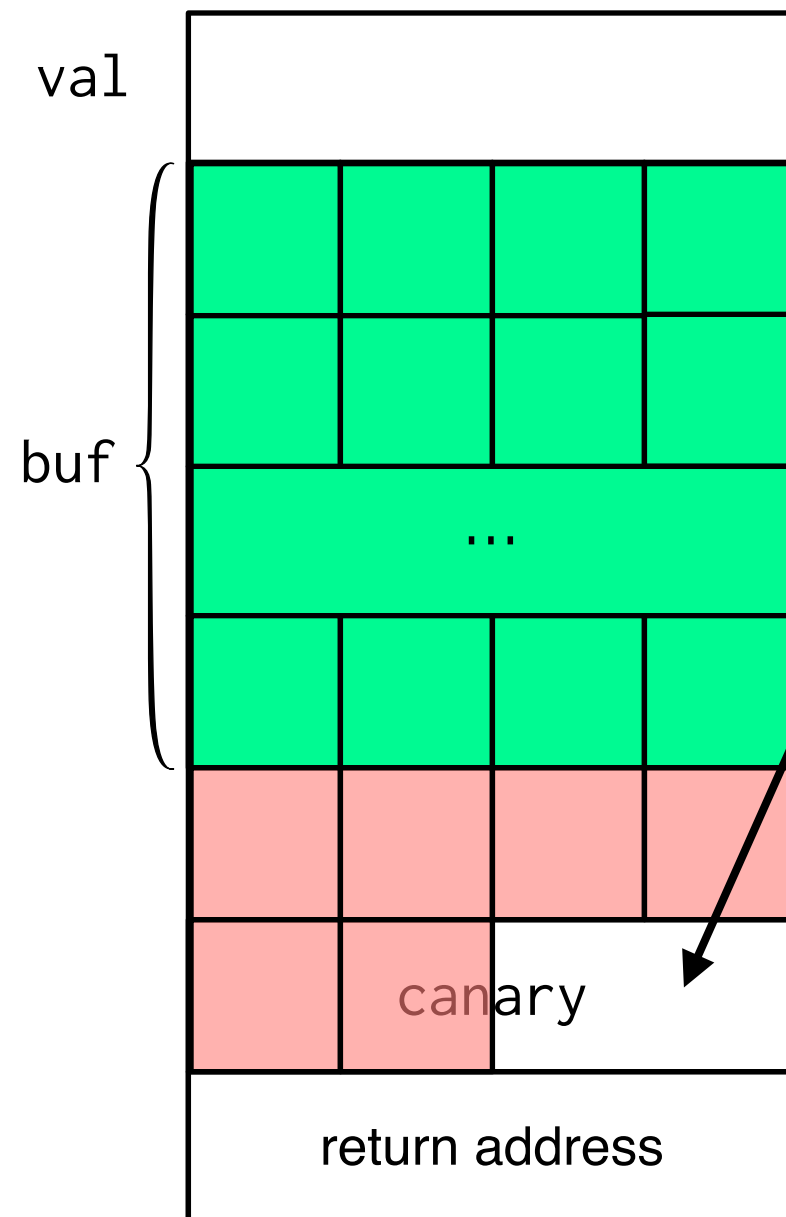
スタック破壊攻撃の危険性

- 攻撃者はスタックフレームに収まる範囲で任意のコードを実行できる
 - － シェルコードが実行できれば大きさは関係なくなる
 - － 犠牲となるプロセス（の実効ユーザ）の権限内で何でもできることになる
- 攻撃対象はローカルでもリモートでもよい
 - － ローカル：攻撃者が利用しているマシン自体を攻撃
 - － リモート：ネットワークを介して他のマシンを攻撃

スタック破壊の回避方法

- OSレベルでの対策
 - － プログラムに割りあてるメモリ番地のランダム化
 - － スタック上のコードの実行禁止
- 言語レベルでの対策
 - － スタック保護（カナリア等）
 - － 配列の境界検査コード
- プログラミングにおける対策
 - － 既知の脆弱性をもつ関数を使用しない
 - － C/C++を使用しない

スタック保護



戻りアドレスの上に、カナリアと呼ばれるデータを置いておく.

関数呼び出しから戻る際にカナリアの値を調べ、変更されていたらスタック破壊があったと判定する.

実例：gccの-fstack-protector

カナリアの値は攻撃者に予測されないようにする必要がある.

バッファオーバーフロー攻撃の進化

- フォーマット文字列のオーバーフロー
- ヒープ領域におけるバッファオーバーフロー
- Pointer Subterfuge
 - Function pointer clobbering
 - VPTR smashing
 - PLT/GOT modification
- Arc-injection
 - Return-to-libc

Function-Pointer Clobbering

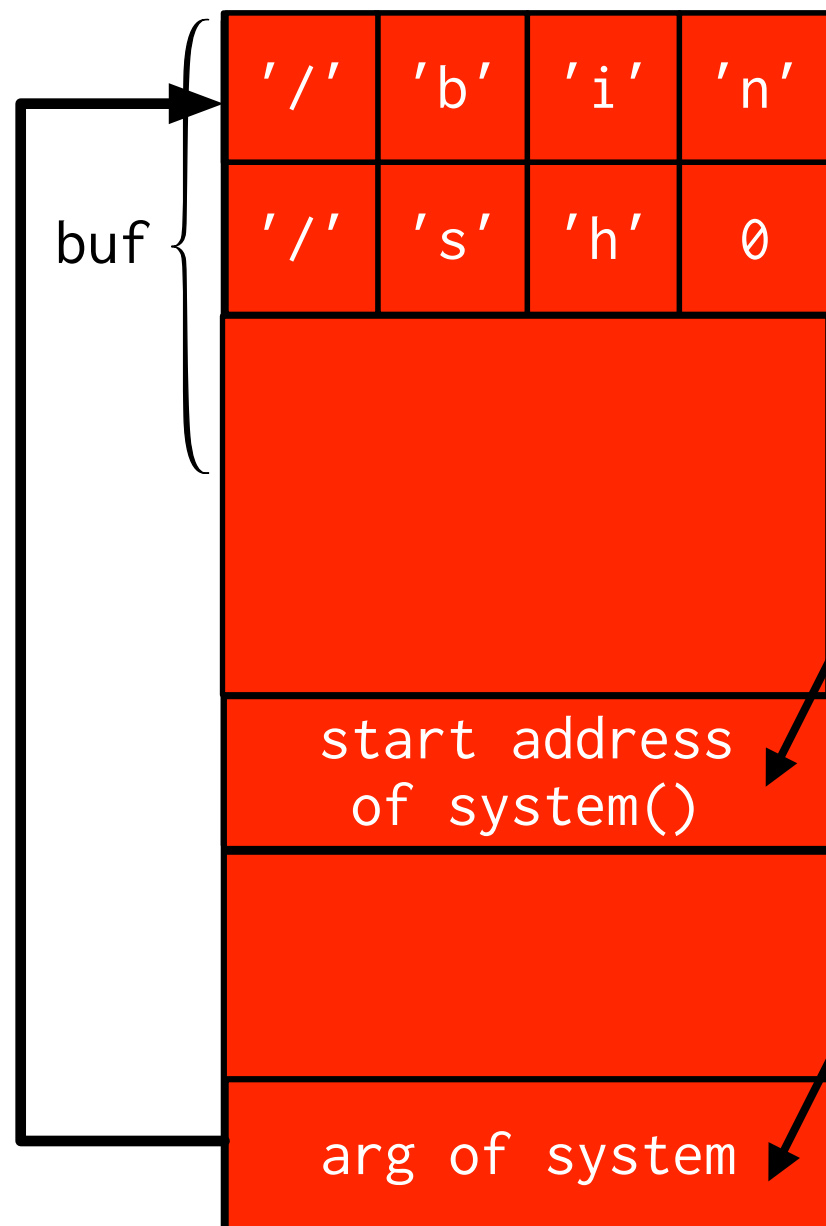
```
char buf[...];  
void (*fp)(int) = ...;  
...  
strcpy(buf, ...);  
...  
fp(...);
```

関数ポインタ型の変数がバッファの後に配置されていた場合、バッファオーバーフローによって書き換えられる可能性がある。

類似のテクニックに、C++のvtableをバッファオーバーフローによる書き換えがある。

カナリヤ等の方法では検知できない。また、ヒープ領域にも適用可能である。

Arc-Injection (Return-to-libc)



戻りアドレスを, 既存の関数 (例えば標準ライブラリ内の関数system) のアドレスで書き換える.

さらにその下に, systemの引数となるデータを書き込む. この場合は文字列 "/bin/sh".

これにより, 関数systemが引数 "/bin/sh" を与えられた形で呼び出されることになる.

スタック領域が実行不可能でもよい.

2015年度の小課題より

- 問題：カーネルでpanicを呼び出すことになるようなユーザプログラムを1つ以上作成せよ
 - void panic(char *s); (console.c)
 - 引数sで指定された文字列をコンソールに出力し、OSを停止する。致命的なエラーが発生してこれ以上処理を続けられ（そうに）ないときに呼び出される。
 - Unix系OSの「カーネルパニック」、Windowsでの「ブルースクリーン」に相当する状態にする
- 出題意図
 - カーネルの動作，特に資源保護機構等の理解

あるレポートより

- 用意した解答例は、pidが1のプロセス(init)をkillで停止させる等の5通り程度. 少なくとも1つ見つければよいことにしていた.
- ところが1人で19通りもの解答を提出した人がいた (2019年修士修了の長田晃太郎氏)
- 彼の解答の中に、execがELFファイルをロードする際にメモリチェックのバグを用いたものがあった.

ELFファイルのロード

- ELFファイルをメモリに読み込む部分(exec.c)
 - ph.vaddrとph.memszはELFヘッダに記述されたプログラムの先頭アドレスとサイズ

```
// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
        goto bad;
    if(ph.type != ELF_PROG_LOAD)
        continue;
    if(ph.memsz < ph.filesz)
        goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
        goto bad;
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
        goto bad;
}
```

ロード時のチェック (旧版)

- allocuvmは第3引数がKERNBASEより小さい値であることを確認して、必要なメモリを確保する. (vm.c)
 - xv6ではKERNBASEより大きな論理アドレスはカーネルが用いているため、ここ以上の場所にユーザプログラムはロードできない (x86版) .
- したがって、たとえph.vaddrがKERNBASE以上に設定されていたとしても、そのようなプログラムは実行できないようはずである.

- ところが、このチェックをすり抜けるコードを以下のようにして構成することができる
 - `ph.vaddr`として適当なシステムコールの先頭アドレス（`KERNBASE`以上の値）を持ち、かつ `ph.vaddr + ph.memsz` が32ビット演算のオーバーフローにより`KERNBASE`より小さい値になるようにする.
- そのファイルをロードすることで、`allocuvm`のチェックにかからず`ph.vaddr`で指定されたアドレスのシステムコールを上書きできる.
- 上書きされたシステムコールを実行させることで、カーネル内で任意のコードを実行できる.

ロード時のチェック (現版)

- 現バージョンでは以下のようにになっている

```
// Load program into memory.
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph))
        goto bad;
    if(ph.type != ELF_PROG_LOAD)
        continue;
    if(ph.memsz < ph.filesz)
        goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
        goto bad;
    uint64 sz1;
    if((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz)) == 0)
        goto bad;
    sz = sz1;
    if((ph.vaddr % PGSIZE) != 0)
        goto bad;
    if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0)
        goto bad;
}
```

安全なプログラムを作る指針

- 参考書

- Secure Coding in C and C++ (2nd ed)
 - Robert C. Seacord, Addison-Wesley, 2013.
 - 邦訳：C/C++セキュアコーディング, アスキー

- サイト

- CERT Cコーディングスタンダード
 - <https://www.jpccert.or.jp/sc-rules/>
- C/C++セキュアコーディングセミナー資料
 - <https://www.jpccert.or.jp/research/materials.html>

脆弱性の根絶に向けて

- これまでに紹介した回避法の多くは、実行時に攻撃を検出あるいは無効化するものである。つまりバッファオーバーフロー攻撃を許す脆弱性が残っていることを仮定している。
- これは現実的であるが、そもそもプログラムにそういった脆弱性がないことが理想である。
- このような「理想」を実現するにはどうすればよいか。

形式手法

- ソフトウェアの品質保証を目的として、科学的（主として数理論理的）手法にもとづいてソフトウェアを構成する手法
- 設計から実装段階までさまざまな手法がある
- 形式検証：プログラムが所定の性質をみたしていることを保証すること
 - － 演繹的検証
 - － モデル検査

演繹的な証明方法：ホーア論理

- $\{ P \} C \{ Q \}$
 - P : 事前条件(precondition)
 - Q : 事後条件(postcondition)
 - C : プログラムコード
- 部分正当性(partial correctness)
 - P が成立しているときに C を実行して停止すれば Q が成立する
- 全正当性(total correctness)
 - P が成立しているときに C を実行すれば, 必ず停止して Q が成立する

ホーア論理(1)

- 代入文の規則

$$\frac{}{\{Q[e/x]\} \ x = e \ \{Q\}}$$

- if文の規則

$$\frac{\{P \wedge B\} \ C \ \{Q\} \quad \{P \wedge \neg B\} \ D \ \{Q\}}{\{P\} \ \text{if } (B) \ C \ \text{else } D \ \{Q\}}$$

- while文

$$\frac{\{P \wedge B\} \ C \ \{P\}}{\{P\} \ \text{while } (B) \ C \ \{P \wedge \neg B\}}$$

- P はループ不変条件(loop invariant)と呼ばれる

ホーア論理(2)

- 複合文の規則

$$\frac{\{P\} C \{Q\} \quad \{Q\} D \{R\}}{\{P\} C; D \{R\}}$$

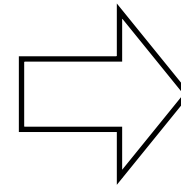
- 帰結の規則

$$\frac{P' \Rightarrow P \quad \{P\} C \{Q\} \quad Q \Rightarrow Q'}{\{P'\} C \{Q'\}}$$

記法

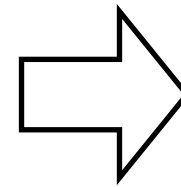
証明図の代わりにコメントを使って
以下のように書くことにする

$$\overline{\{Q[e/x]\} \ x = e \ \{Q\}}$$



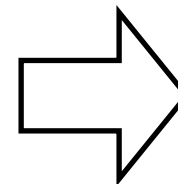
```
// Q[x/e]  
x = e;  
// Q
```

$$\frac{\{P\} \ C \ \{Q\} \quad \{Q\} \ D \ \{R\}}{\{P\} \ C; D \ \{R\}}$$



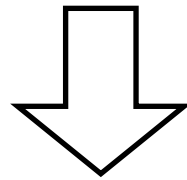
```
// P  
C  
// Q  
D  
// R
```

$$\frac{P' \Rightarrow P \quad \{P\} \ C \ \{Q\} \quad Q \Rightarrow Q'}{\{P'\} \ C \ \{Q'\}}$$



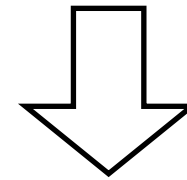
```
// P'  
// P  
C  
// Q  
// Q'
```


$$\frac{\{P \wedge B\} C \{Q\} \quad \{P \wedge \neg B\} D \{Q\}}{\{P\} \text{ if } (B) C \text{ else } D \{Q\}}$$



```
// P
if (B) {
    // P && B
    C
    // Q
}
else {
    // P && !B
    D
    // Q
}
// Q
```

$$\frac{\{P \wedge B\} C \{P\}}{\{P\} \text{ while } (B) C \{P \wedge \neg B\}}$$



```
// P
while (B) {
    // P && B
    C
    // P
}
// P && !B
```

例題： $\lfloor \sqrt{x} \rfloor$ の計算

```
int x, r;  
x = input(...);
```

```
// x >= 0
```

事前条件



```
r = 1;
```

```
while (r * r <= x) {  
    r = r + 1;  
}
```

```
r = r - 1;
```

```
// r >= 0 && r * r <= x && (r + 1) * (r + 1) > x
```

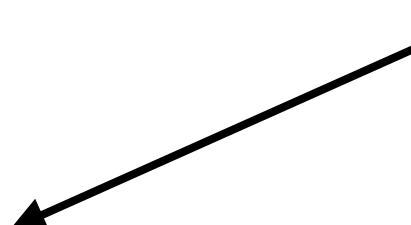
事後条件



```
output(r);
```

証明

```
// x >= 0
// 1 >= 1 && (1 - 1) * (1 - 1) <= x
r = 1;
// r >= 1 && (r - 1) * (r - 1) <= x
while (r * r <= x) {
    // r >= 1 && (r - 1) * (r - 1) <= x && r * r <= x
    // r + 1 >= 1 && ((r + 1) - 1) * ((r + 1) - 1) <= x
    r = r + 1;
    // r >= 1 && (r - 1) * (r - 1) <= x
}
// r >= 1 && (r - 1) * (r - 1) <= x && !(r * r <= x)
// r - 1 >= 0 && (r - 1) * (r - 1) <= x
//          && ((r - 1) + 1) * ((r - 1) + 1) > x
r = r - 1;
// r >= 0 && r * r <= x && (r + 1) * (r + 1) > x
```



ループ不変式

Cプログラムの検証

- 演繹的な方法に基づく検証ツール
 - Frama-C
 - 様々な検証器と組み合わせて使用する検証環境
 - Infer
 - ポインタを扱うことのできる分離論理(Separation Logic)をサポートする検証器
 - Java, Objective-Cにも対応

```

/*@ requires n > 0 && \valid_range(a, 0, n - 1);
    @ requires \forall integer i, j;
        @      l <= i <= j <= h ==> a[i] <= a[j];
    @ behavior success:
        @   assumes \exists integer i; 0 <= i < n && a[i] == v;
        @   ensures 0 <= \result < n && a[\result] == v;
    @ behavior failure:
        @   assumes \forall integer i; 0 <= i < n ==> a[i] != v;
        @   ensures \result == -1;
    @*/
int binsearch(int a[], int n, int v) {
    int lo = 0, hi = n - 1;
    /*@ loop invariant 0 <= lo <= hi + 1 && hi < n;
        @ loop invariant \forall integer k;
            @      0 <= k < n && a[k] == v ==> lo <= k <= hi;
        @ loop assigns hi, lo;
        @ loop variant hi - lo;
    @*/
    while (lo <= hi) {
        int m = (lo + hi) / 2;
        if (v == a[m])
            return m;
        else if (v < a[m])
            hi = m - 1;
        else
            lo = m + 1;
    }
    return -1;
}

```

例：Frama-C用の 2分探索の仕様記述

演繹的な方法のむずかしさ

- 多くの場合, 以下を与える必要がある
 - ループ不変式
 - 帰結の規則で用いる条件(検証条件: verification condition)
 - 検証に用いる各種補題
- これにより証明の完全自動化は難しい

例題：スレッドの相互排除

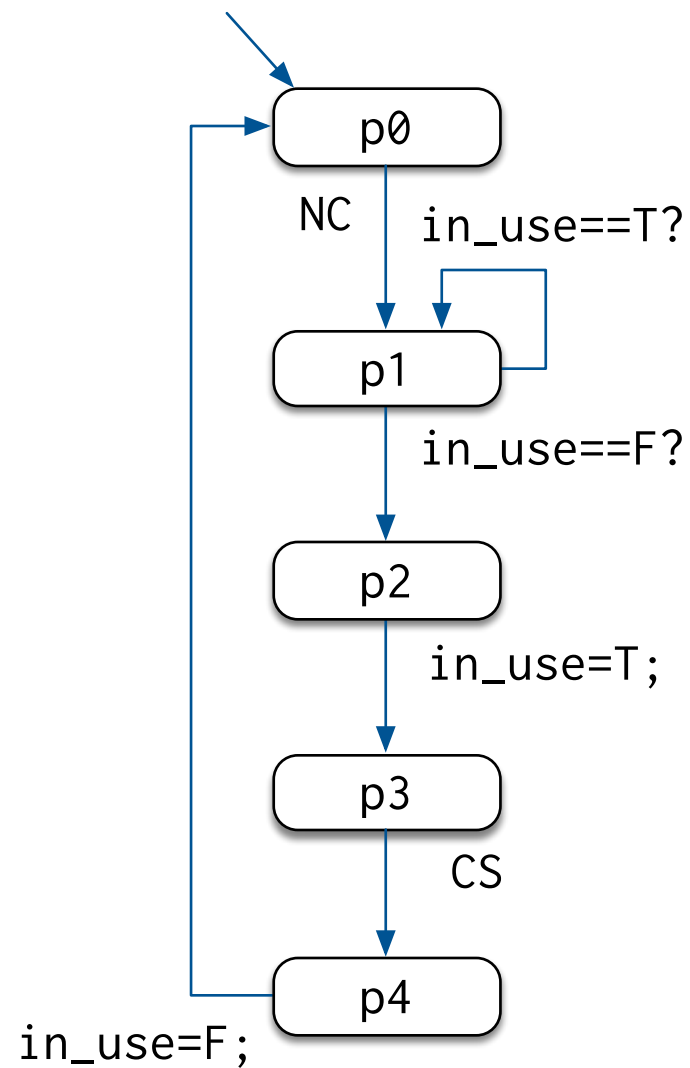
```
// 共有変数
boolean in_use = false;
```

```
// スレッドPのコード
void thread_P() {
    while (true) {
        p0: NC;
        p1: while (in_use);
        p2: in_use = true;
        p3: CS;
        p4: in_use = false;
    }
}
```

```
// スレッドQのコード
void thread_Q() {
    while (true) {
        q0: NC;
        q1: while (in_use);
        q2: in_use = true;
        q3: CS;
        q4: in_use = false;
    }
}
```

2つのスレッドPおよびQが、同時にクリティカルセクション(CS)を実行しないかどうかを確かめたい

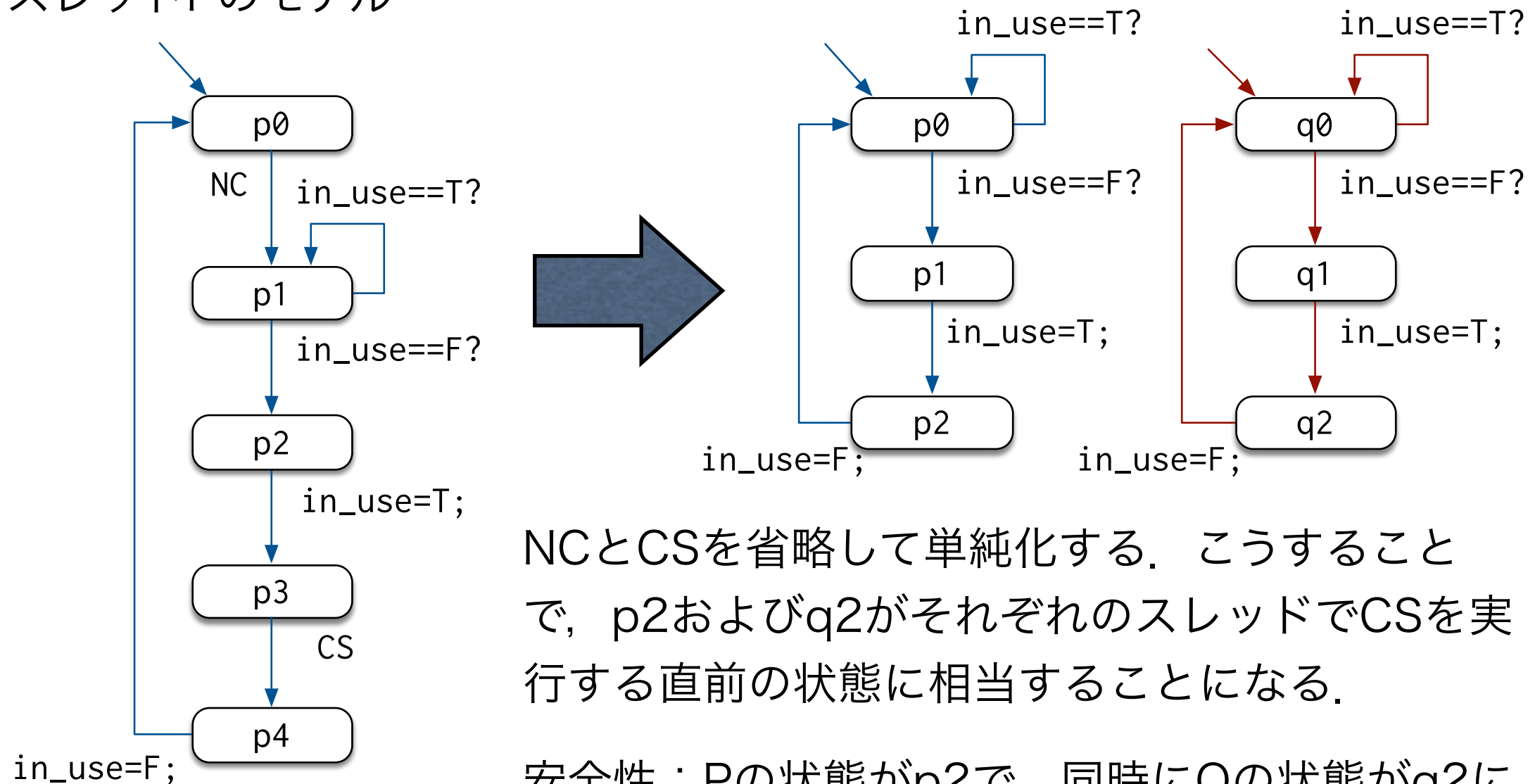
オートマトンによるモデル化(1)



```
// スレッドPのコード
void thread_P() {
    while (true) {
        p0: NC;
        p1: while (in_use);
        p2: in_use = true;
        p3: CS;
        p4: in_use = false;
    }
}
```


オートマトンによるモデル化(2)

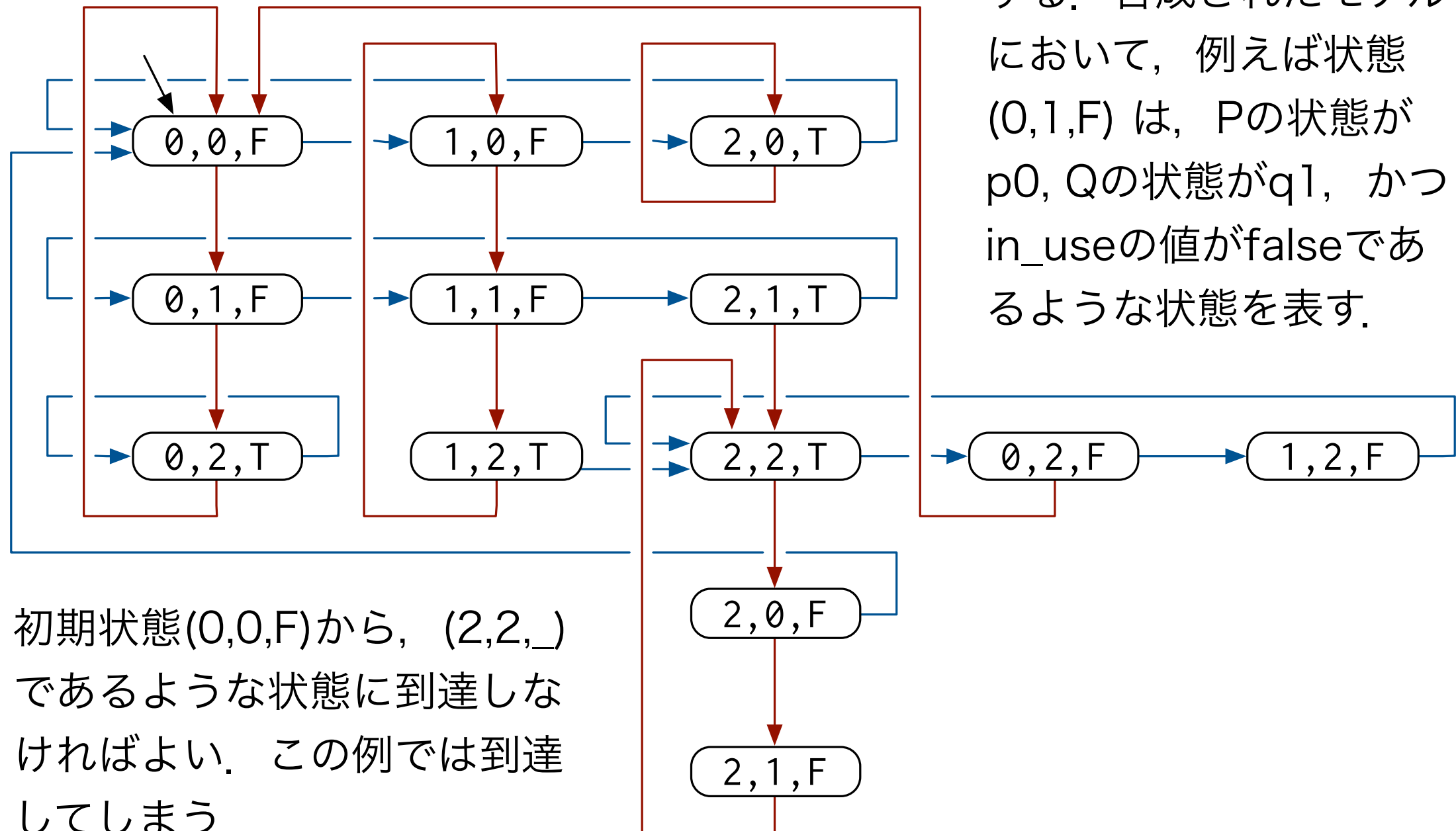
スレッドPのモデル



NCとCSを省略して単純化する。こうすることで、p2およびq2がそれぞれのスレッドでCSを実行する直前の状態に相当することになる。

安全性：Pの状態がp2で、同時にQの状態がq2になるようなことがない

網羅的探索

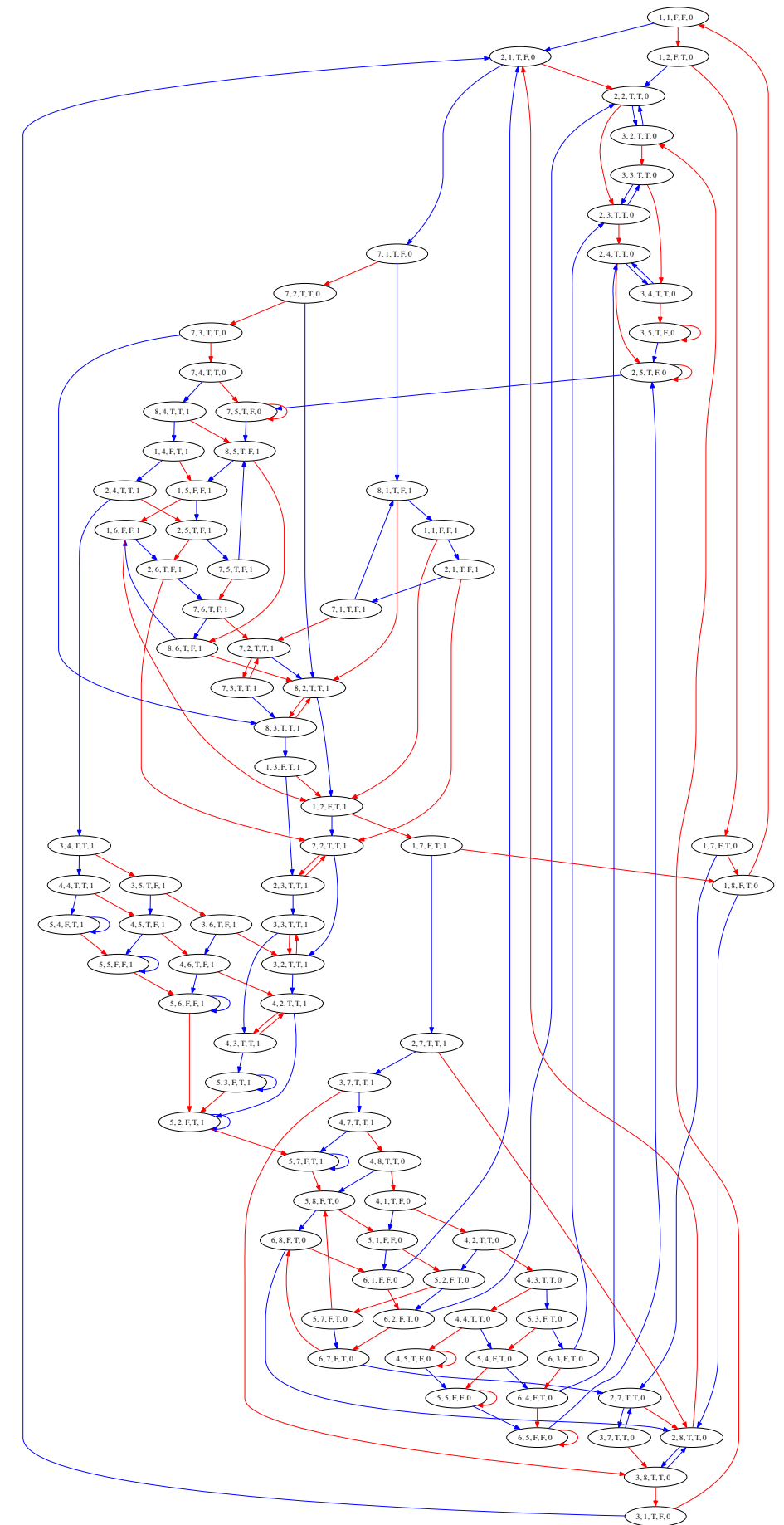


PとQの状態および共有変数の値から, 系全体の動作を表すモデルを合成する. 合成されたモデルにおいて, 例えば状態(0,1,F)は, Pの状態がp0, Qの状態がq1, かつin_useの値がfalseであるような状態を表す.

Dekkerのアルゴリズム

```
// shared variables
bool wantp = false, wantq = false;
int turn = 0; // 0 or 1
```

```
// thread p
while (true) {
  p0: NC
  p1: wantp = true;
  p2: while (wantq) {
  p3:   if (turn == 1) {
  p4:     wantp = false;
  p5:     while (turn == 1);
  p6:     wantp = true;
        }
  }
  p7: CS
  p8: turn = 1;
  p9: wantp = false;
}
```



検証の自動化

- 以下は自動化できる
 - － スレッドP, Qのモデルから, 全体の動作を表すモデルを合成する
 - － 合成したモデルにおいて, 初期状態から安全性をみたさない状態に到達可能か否かを網羅的に探索する
- このような方法による, システムがある性質をみたすか否かの判定方法をモデル検査と呼ぶ

モデル検査

- 対象となるシステム（プログラム）の動作をオートマトンとしてモデル化する
- オートマトンとして表現された, 検証したい性質の否定を対象のモデルと合成する
 - － LTLなどの時相論理の式で記述された性質を等価なオートマトンに変換する
- 合成されたオートマトンの網羅的探索により（検証したい性質の）反例の発見を試みる
 - － 受理された実行系列があれば反例

Spin

```
mtype = { TurnP, TurnQ };
mtype turn = TurnP;
bool wantp = false, wantq = false;
int ncs = 0;

active proctype P () {
    do :: wantp = true;
        do :: wantq ->
            if :: (turn == TurnQ) ->
                wantp = false;
                turn == TurnP;
                wantp = true;
            :: else -> skip;
        fi
    :: else -> break;
    od;
progress:
    ncs++;
    assert(ncs == 1);
    ncs--;
    turn = TurnQ;
    wantp = false;
od
}
```

PromelaによるDekkerのアルゴリズムの記述例

- 1980年代から開発が続けられている最も有名なモデル検査ツール
 - <http://spinroot.com>
- 検証対象をモデリング言語Promelaで記述し，検証したい性質を時相論理(LTL)の式で記述する

モデリング言語の問題点

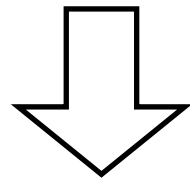
- Spin, nuSMV, UPPAALなどのモデル検査ツールでは、検証対象の振る舞いをモデリング言語と呼ばれる言語によって記述する
- プログラムを検証したい場合、その動作をモデリング言語で書き直す必要がある
- モデリング言語で記述されたモデルが、実際のプログラムの動作を正しく表現していることの保証はない

ソフトウェアモデル検査

- モデリング言語による記述ではなく，プログラミング言語のソースコードやバイトコード，機械語コードを検証対象とするモデル検査
 - － 例
 - BLAST, SLAM, CBMC
 - Java PathFinder, Bander
 - － CBMC
 - 有界モデル検査(Bounded Model Checking)にもとづくC用のモデル検査系
 - － <http://www.cprover.org/cbmc/>

有界モデル検査

```
while (P) C
```



```
if (P) {  
  C  
  if (P) {  
    C  
    if (P) {  
      C  
      :  
      if (P) {  
        C  
        assert(!P);  
      } ... }  
    }  
  }  
}
```

- 探索空間を適当な値 k で抑えることでモデルを有限サイズの命題論理式(CNF)に変換し, SATソルバによる検証を行う
- ループや再帰は有限回展開する (左図)
 - 反例のサイズが k 以下ならば正しく検証可能

例題(ex0.c)

```
int f() {
    int val;
    char buf[64];
    gets1(buf);
    val = atoi(buf);
    return val * val;
}

char *gets1(char *buf) { // gets
    char *s = buf;
    int c = getchar();
    if (c == EOF) return NULL;
    while (c != '\n' && c != EOF) {
        *s++ = c;
        c = getchar();
    }
    *s = '\0';
    return buf;
}
```

CBMCによる検証

- バッファの範囲外に代入する可能性があることが発見できた

```
$ cbmc ex0.c --bounds-check --pointer-check --unwind 65
....
Violated property:
  file ex.c line 19 function gets1
  dereference failure: object bounds in *tmp_post$1
  OBJECT_SIZE(tmp_post$1) >= 1 +
  POINTER_OFFSET(tmp_post$1) && POINTER_OFFSET(tmp_post$1)
  >= 0 || DYNAMIC_OBJECT(tmp_post$1)

VERIFICATION FAILED
```

改良版(1)

```
int f() {  
    int val;  
    char buf[64];  
    gets1(buf, 64);  
    val = atoi(buf);  
    return val * val;  
}
```

バッファサイズをチェックする
ようにしてみた

```
char *gets1(char *buf, int n) {  
    char *s = buf;  
    int c = getchar();  
    if (c == EOF) return NULL;  
    while (n > 0 && c != '\n' && c != EOF) {  
        *s++ = c;  
        c = getchar();  
        n--;  
    }  
    *s = '\0';  
    return buf;  
}
```

検証結果

- while文直後の '\0' の代入でバッファ外に書き込んでいる (off-by-oneエラー)

```
$ cbmc ex1.c --bounds-check --pointer-check --unwind 65
....
Violated property:
  file ex1.c line 23 function gets1
  dereference failure: object bounds in *s
  OBJECT_SIZE(s) >= 1 + POINTER_OFFSET(s) &&
  POINTER_OFFSET(s) >= 0 || DYNAMIC_OBJECT(s)

VERIFICATION FAILED
```

改良版(2)

```
int f() {  
    int val;  
    char buf[64];  
    gets1(buf, 64);  
    val = atoi(buf);  
    return val * val;  
}
```

```
char *gets1(char *buf, int n) {  
    char *s = buf;  
    int c = getchar();  
    if (c == EOF) return NULL;  
    while (n > 1 && c != '\n' && c != EOF) {  
        *s++ = c;  
        c = getchar();  
        n--;  
    }  
    *s = '\0';  
    return buf;  
}
```

行末の '\0' の余地を残す必要がある

検証結果

```
$ cbmc ex1.c --bounds-check --pointer-check --unwind 65
....
size of program expression: 7978 steps
simple slicing removed 6530 assignments
Generated 756 VCC(s), 6 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
40787 variables, 117236 clauses
SAT checker: instance is UNSATISFIABLE
Runtime decision procedure: 0.178s
VERIFICATION SUCCESSFUL
```

まとめ

- セキュリティ

期末試験

- 日時：2020年11月22日(月), 7-8限
(16:15-17:55)
- 試験形態：オンライン
 - － 問題をPDFで配布し, webフォームで解答
- 試験範囲：講義で扱った内容
 - － プロセス管理, 同期機構, メモリ管理, ファイルシステムおよびこれらのxv6 (RISC-V版)における実装
 - － 過去の問題は講義webサイトに掲載
- 配布資料やノート, ソースコードの参照可