C/C++ セキュアコーディング

File I/O part3:
ファイル入出力と競合状態

2010年3月23日
JPCERTコーディネーションセンター

本モジュールのゴール

- ✓ TOCTOU競合状態の脅威を理解する。
- ✓ 状況に合った対応策を理解する。

もくじ

競合状態とは

複数の処理を想定しない順序で並列実行することにより、望ましくない状況が発生すること。

- 共有オブジェクトの不整合な状態
- 誤った処理結果
- 共有オブジェクトの破壊、改ざん、漏えい

ソフトウェアの欠陥であり、多くの場合脆弱性に繋がる。

2つ以上の独立した処理を同時に実行することを
並列処理と呼ぶ [Dijkstra 65]

独立した実行フローの単位:

- スレッド
- プロセス
- タスク

JPCERT CC®

信頼できる制御フロー

同一プログラムの中の密接に関連するスレッドや親子関係を持つプロセスなど制御可能な実行フロー

信頼できない制御フロー

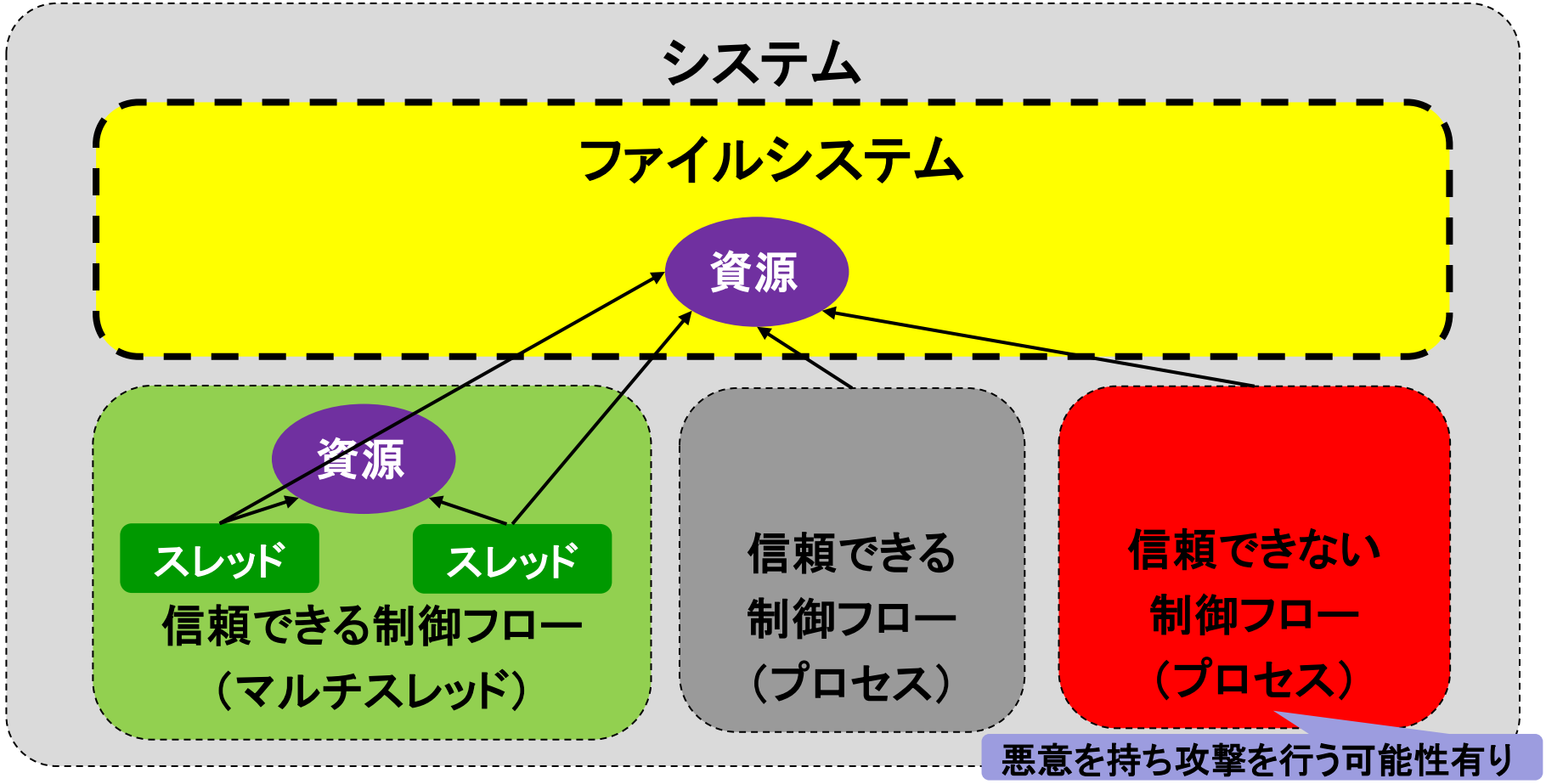
並行して実行される別のアプリケーションやプロセス(どこから実行されたか分からない)など制御できない実行フロー

競合状態はどちらのフローからでも発生しうる。

信頼できない制御フローによる競合状態

- 資源の共有を行うマルチタスクシステムには、信頼できない制御フローが原因で競合状態が発生しうる。
- 制御が難しい／不可能であるため、競合状態の発生に繋がり易い。
- システムで共有されるファイルシステムへの入出力の際には、TOCTOU競合状態が発生し易い。
- Time Of Check, Time Of Use：検査する時、使う時

信頼できる／できない制御フローと共有資源



競合状態を突いた攻撃

TOCTOU競合状態を突いたファイル／ディレクトリの乗っ取りによる、ファイルの破壊、改ざん、情報漏えいなどが可能に:

- symlink攻撃
- unlink()競合攻撃
- 一時ファイルへの攻撃

不適切なファイル入出力操作により、TOCTOU競合状態を突いた攻撃を受ける危険性が高い！

Linux init.d/xfs Race Condition Vulnerability

JPCERT CC®

http://cve.mitre.org/cgi-bin/cvename.cgi?name=2007-3103

CVE-2007-3103

xfs -- X Windows System のフォントサーバ

- ネットワーク経由では 7100/tcp で listen
- ローカルには UNIX domain ソケットで listen

  /etc/init.d/xfs -- xfs を起動するスクリプト

- 起動前に残っていた UNIX domain ソケット用のディレクトリを削除して、あらためて作成

```
rm -rf $FONT_UNIX_DIR
mkdir $FONT_UNIX_DIR
chown root:root $FONT_UNIX_DIR
chmod 1777 $FONT_UNIX_DIR
```

Linux init.d/xfs Race Condition Vulnerability

ディレクトリ削除と作成の間の時点でシンボリックリンクを作成することで，リンク先ファイルのパーミションを変更させられる．(mkdir はエラー終了し，その後が実行される.)

```
rm -rf $FONT_UNIX_DIR
** ここで先にシンボリックリンクを作成 **
mkdir $FONT_UNIX_DIR
chown root:root $FONT_UNIX_DIR
chmod 1777 $FONT_UNIX_DIR
```

(参考: http://www.milw0rm.com/exploits/5167)

対策: 不要な chmod をしない(Debianなどの例)

Gzip Race Condition Vulnerability

http://cve.mitre.org/cgi-bin/cvename.cgi?name=2005-0988

CVE-2005-0988

Gunzipは圧縮ファイル展開時に以下の順序で動作する:

1．圧縮ファイル展開
2．展開したファイルのパーミションのリストア

パーミション設定の前に，展開されたファイルを別ファイルへのハードリンクに置き換えることにより，Gunzipの権限でリンク先ファイルのパーミションを変更させることができる．

対策：パーミション設定には，ファイル名ではなくファイルディスクリプタを使う．

Gzip Race Condition Vulnerability

(FreeBSDにおける修正例)

```
/* === 修正前 === */
close(ofd);
/* Copy modes, times, ownership, and remove the input file */
chmod(ofname, ifstat->st_mode&07777);

/* === 修正後 === */
/* Copy modes, times, ownership, and remove the input file */
fchmod(ofd, ifstat->st_mode&07777);
close(ofd);
```

ファイル名を指定

ファイル記述子を指定

http://www.freebsd.org/cgi/cvsweb.cgi/src/gnu/usr.bin/gzip/Attic/gzip.c.diff?r1=1.11;r2=1.12;f=h

デッドロック状態により起こりうる被害

- 競合状態を回避するために、共有資源へのアクセスに対して排他制御を行うが、排他制御に欠陥があると、デッドロックを引き起こす。

- デッドロック状態を意図的に発生させてサービス停止を狙う攻撃にさらされてしまう。

- 問題への対応が新たな問題を作り込むケース。

デッドロック状態によるサービス停止

Apache HTTP Server 2.0.48 以前では、子プロセスが接続を受け付けるために利用する同期用のオブジェクトを確保したままデッドロック状態を発生させてしまった結果、新しい接続を受け付けることができなくなってしまった。
(VU#132110)

このように不適切な排他制御の実装の多くはデッドロック状態を発生させ、サービス停止や妨害などの被害に繋がってしまうため、必要な個所に適切な形で排他制御を実装する必要がある。

http://www.kb.cert.org/vuls/id/132110

もくじ

競合状態の必要条件

JPCERT CC®

1. 並列性

2つ以上の制御フローが同時に実行されている。

2. オブジェクトの共有

並列する複数のフローが同一のオブジェクトにアクセスする必要がある。

3. 状態の変更

1つ以上の制御フローが、オブジェクトの状態を変更する。

身近な競合状態：共有ファイルの編集

共有ファイルサーバ上に置かれたスプレッドシートを使って複数人が作業を行っている。

1．並列性

複数人が同時に何らかの一連の作業をしている。

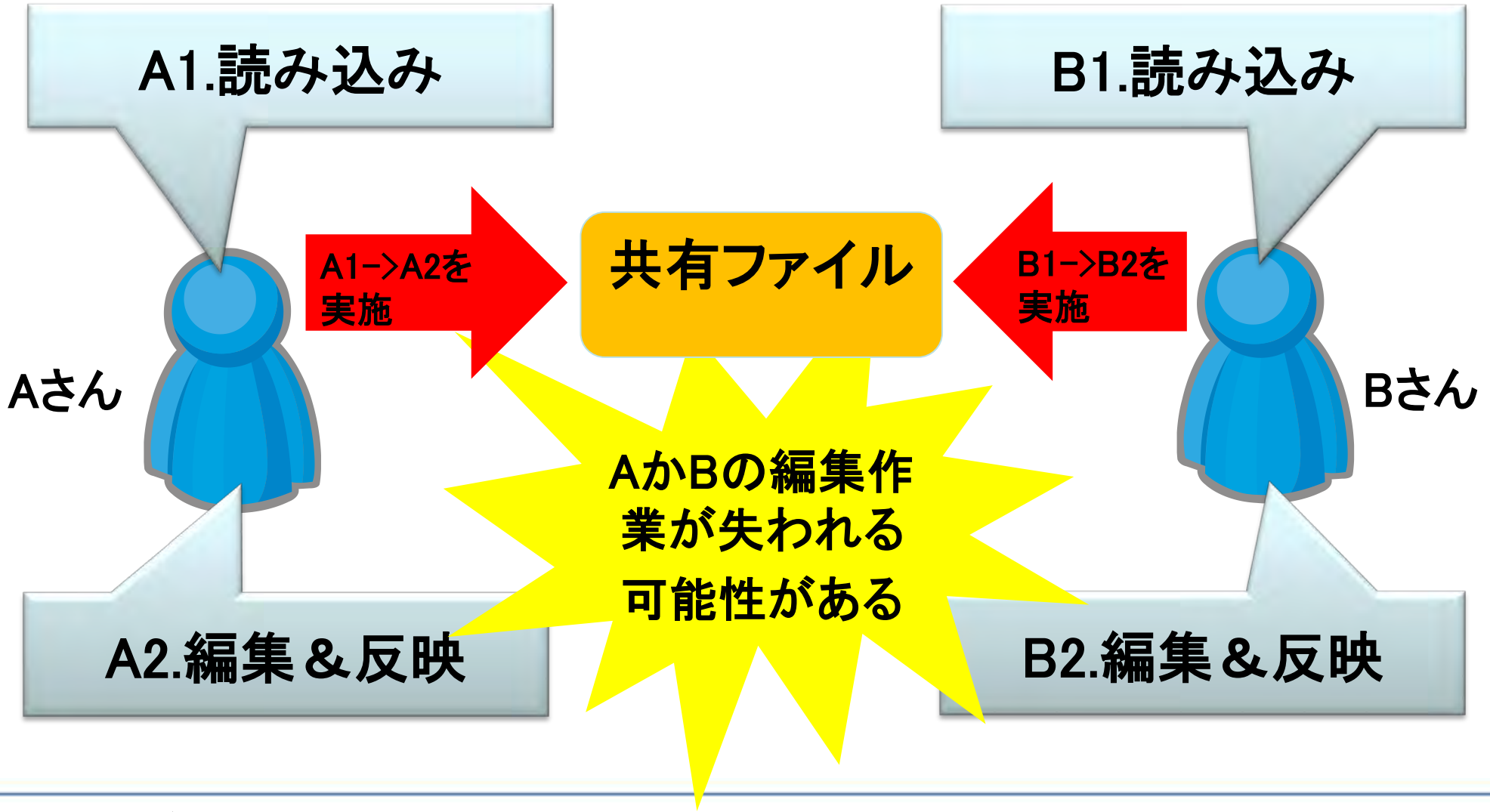
2．オブジェクトの共有

これら複数人が行う作業は、同一のスプレッドシートを利用する。

3．状態の変更

少なくとも　一人が、スプレッドシートを変更する。

身近な競合状態：共有ファイルの編集



身近な競合状態：共有ファイルの編集結果

- 二人以上ファイルへ変更を加えた場合、最後の変更以外の変更作業は失われる

- 変更前の内容を基に行った作業の成果物と不整合が発生（例えば、集計結果など）

競合ウィンドウ／クリティカルセクション

- 競合ウィンドウとは、他の制御フローと共有しているオブジェクトへアクセスする箇所のこと。

- 競合ウィンドウが同時に実行されないことを確保することが ”重要”（＝クリティカル）であることから、クリティカルセクションとも呼ばれる。

- 競合ウィンドウを無くすために、共有オブジェクトへのアクセスを排他制御するが、デッドロックを引き起こす可能性がある。

身近なデッドロック状態：直線を引く

もくじ

信頼できない制御フローとTOCTOU競合状態

JPCERT CC®

- TOCTOU　競合状態では、ある競合オブジェクトを検査し（Check）、その後でそのオブジェクト自体にアクセスする（Use）という一連の処理が競合ウィンドウを構成

- 不適切な検査とアクセス方法によっては、信頼できない制御フローによる操作で、強制的に検査時と使用時に異なるオブジェクトを参照させられてしまう危険性

TOCTOU競合状態の例1:access() → fopen()

JPCERT CC®

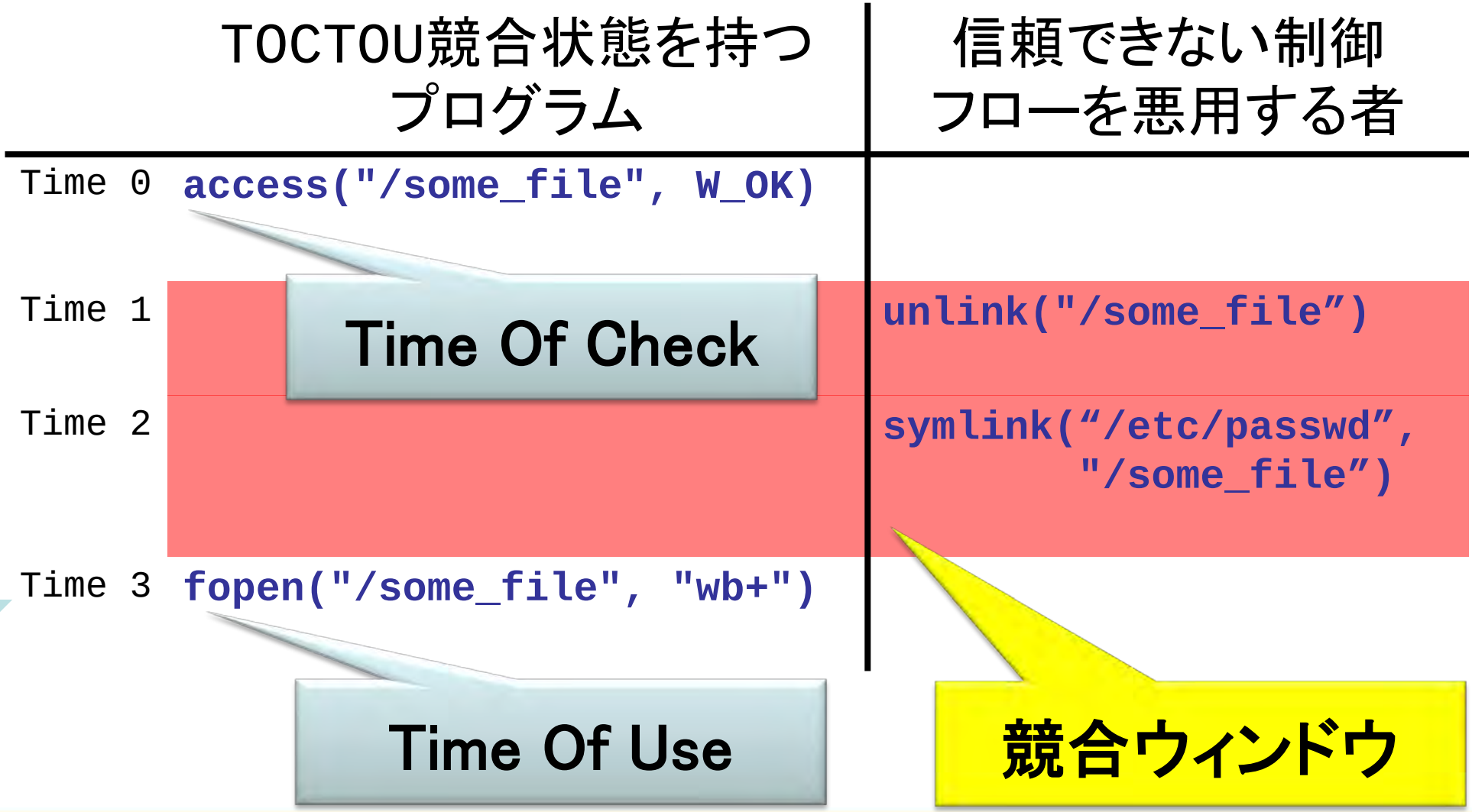
```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
  FILE *fp;
  if (access("/some_file", W_OK) == 0) {
    printf("access granted.¥n");
    fp = fopen("/some_file", "wb+");
    /*ファイルに書き出す */
    fclose(fp);
  }       . . .
  return 0;
}
```

access() 関数を呼び出しファイルの存在と書き込み権限があることをチェックしている。

競合ウィンドウ

書き込みのためにファイルをオープン。

例1に対する攻撃の概要



symlink を使ったTOCTOUに注意

- アクセス権限のないファイルにシンボリックリンクを作成できる。
- 攻撃者はシンボリックリンクを置くディレクトリへの書き込み権限さえあれば作成できる。
- ディレクトリへのシンボリックリンクを作成できると、たとえば `/dir` をまったく異なるディレクトリへのシンボリックリンクに置き換えられる。

unlink() 競合状態も攻撃の対象

ファイルをオープンし、その後にファイルを **unlink()** すると競合状態が生まれる。

オープンされたファイルを他のファイルやシンボリックリンクに置き換えられると、**unlink()** は想定外のファイルに対して操作を行ってしまう。

```
int unlink(const char *path);
```

ファイル名でファイルにアクセスする！

TOCTOU競合状態の例2:stat() → open()

JPCERT CC®

```
if (stat("/dir/some_file", &statbuf) == -1) {
    err(1, "stat");
}
if (statbuf.st_size >= MAX_FILE_SIZE) {
    err(2, "file size");
}
if ((fd=open("/dir/some_file", O_RDONLY)) == -1) {
    err(3, "open - /dir/some_file");
}
// ファイルを処理する
```

stat()を呼び出しているところがTOCTOU の検査部

/dir/some_file ファイルのstatusを調べ、一定のサイズに収まっていることを確認してからファイルを読み込む

open()を呼び出しているところがTOCTOU の使用部

例2に対する攻撃概要

攻撃は、競合ウィンドウの間に以下のコマンドを実行。

```
rm /dir/some_file

ln -s attacker_file /dir/some_file
```

`stat()` 関数に引数として渡されたファイルと、オープンされるファイルは異なる。

攻撃者は `/dir/some_file` を `attacker_file` へのリンクにすり替えて、`/dir/some_file` を乗っ取る。

一時ファイルを狙った攻撃コード

JPCERT CC®

```
……
cd /tmp
cat > sym.c << EOF
#include <unistd.h>
int main(){
   for(;;){if(symlink("/etc/passwd”,
              "/tmp/.font-unix")==0)
   {return 0;}}}EOF
……
```

出典：Xorg-x11-xfs Race Condition Vuln local root exploit (CVE-2007-3103)
http://www.milw0rm.com/exploits/5167

ファイルアクセスは狙われる

ファイルのオープン、リード、ライト、クローズ、いたるところで競合ウィンドウは発生する。

ファイルアクセスは狙われる。

- オープンされたファイルはスレッド間で共有される
- ファイルは他のプロセスにアクセスされうる

攻撃は以下の点を突いて行われる。

- ファイルのアクセス権限
- ファイルのネーミングルール
- ファイルシステムの特徴

見落としがちな共有オブジェクト

- システムによって提供される共有資源は、ソフトウェアの領域から一見離れているように思えるため、見落としやすい。
- たとえば、あるディレクトリにファイルを作成するプログラムは、それよりもルートに近いところにあるディレクトリにおける競合状態に影響を受ける可能性がある。
- また、Windows システムでは、重要な共有資源としてレジストリが加わる。

システム提供資源を不必要に使用しない

システムが提供する資源を不必要に使用しないことで、脆弱性の発生を最小限に押えることができる。

例:

- Windows の **ShellExecute()** はファイルに応じたアプリケーションを選び出すためにレジストリを参照
- レジストリに依存する構造よりは、**CreateProcess()** 関数を使ってファイルを処理するアプリケーションを明示的に指定する方が良い

もくじ

JPCERT CC®

1. 競合状態とその脅威
2. 競合状態の発生メカニズム
3. TOCTOU競合状態
4. 脅威の緩和方法
   - 脅威の緩和アプローチ
   - 実践テクニック
   - 解析ツール活用
5. まとめ
6. 補足資料：デッドロックについて

脅威の緩和方法

競合状態の3つの必要条件が揃わないようにする。

1．競合ウィンドウの並行性を無くす

- 排他制御する
- 競合ウィンドウを閉じる

2．オブジェクトの共有度を低くする

- 共有オブジェクトを無くす／減らす／分割する
- 共有範囲を限定する（信頼できない制御フローの関与排除）

3．オブジェクトの状態変更の機会を減らす

- 共有オブジェクトの変更処理を見直す
- 共有範囲を限定する（信頼できない制御フローの関与排除）

排他制御により競合ウィンドウの並行性を無くす

信頼できる制御フロー：

スレッドまたはプロセス間で利用する同期プリミティブ、ロックファイルを使う

⇒同じ同期メカニズム／ルールに従うことが前提

信頼できない制御フロー：

ファイルロック、あるいは、TOCTOU競合状態を極力無くす（競合ウィンドウを閉じる）

⇒同じ同期メカニズム／ルールに従うことは期待できない

競合ウィンドウを閉じて、並行性を無くす

- 不可分な実行単位の確保

  TOCTOU競合状態に繋がる複数ステップ（検査→使用）の処理を不可分な形（検査＆使用）に見直す

- ファイル記述子／ポインタの使用、属性の検査

  ファイル名だけに依存せずにファイルアクセスを行うことで、TOCTOU競合状態の発生を抑える

- スレッドセーフなデータ型、関数を利用

  新たな競合ウィンドウ、あるいは、デッドロック状態の要因となりえるネタを入れ込まない

オブジェクトの共有度を低くする

デザインアプローチ

- 共有オブジェクトの必要性、粒度を見直す。
  - 共有オブジェクトを無くする（必要性を見直す）
  - 共有オブジェクトを分割する（個々の共有度を下げる）

システム管理的アプローチ

- 信頼できない制御フローの関与をできる限り排除することで、想定される脅威を最小限にする。
  - 最小権限で動作させる
  - 共有ディレクトリをできるだけ使わない
  - セキュアディレクトリの利用

オブジェクトの状態変更の機会を減らす

JPCERT CC®

デザインアプローチ

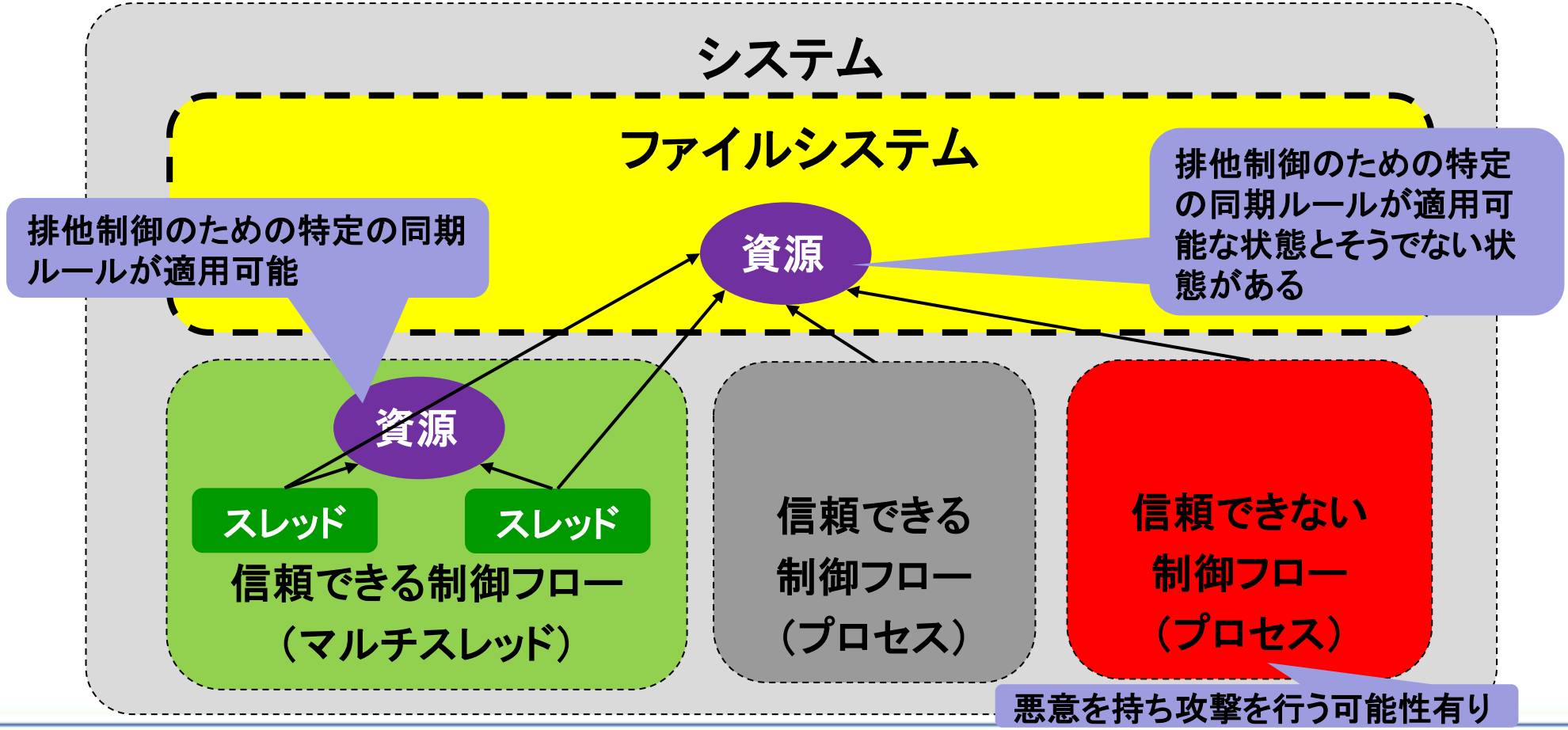
- 共有オブジェクトの変更処理を見直す。
  - 複数の更新処理の整理統合
  - 不必要な更新処理の排除

システム管理的アプローチ

- 信頼できない制御フローの関与をできる限り排除することで、想定される脅威を最小限にする。
  - 最小権限で動作させる
  - 共有ディレクトリをできるだけ使わない
  - セキュアディレクトリの利用

信頼できる／できない制御フローと共有資源

【資源を安全に共有するための条件（どちらか一方）】
- 共有資源へのアクセスが信頼できる制御フローからのみに限定されている
- 共有資源へのアクセスを強制的に制限するサービスが利用できる



制御フロー、対策、脅威の対応

	対策				脅威	
	同期プリミティブ	ロック用途ファイル	ファイルロック	システム管理的アプローチ	デッドロックライブロック	TOCTOU競合状態
信頼できる制御フロー間の競合	利用可能	利用可能	利用可能	利用可能	◎	○
信頼できない制御フローが関与する競合	利用不可	利用不可	利用可能（強制）	利用可能	◎	◎

攻撃の可能性

もくじ

JPCERT CC®

1. 競合状態とその脅威
2. 競合状態の発生メカニズム
3. TOCTOU競合状態
4. 脅威の緩和方法
   - 脅威の緩和アプローチ
   - 実践テクニック
   - 解析ツール活用
5. まとめ
6. 補足資料：デッドロックについて

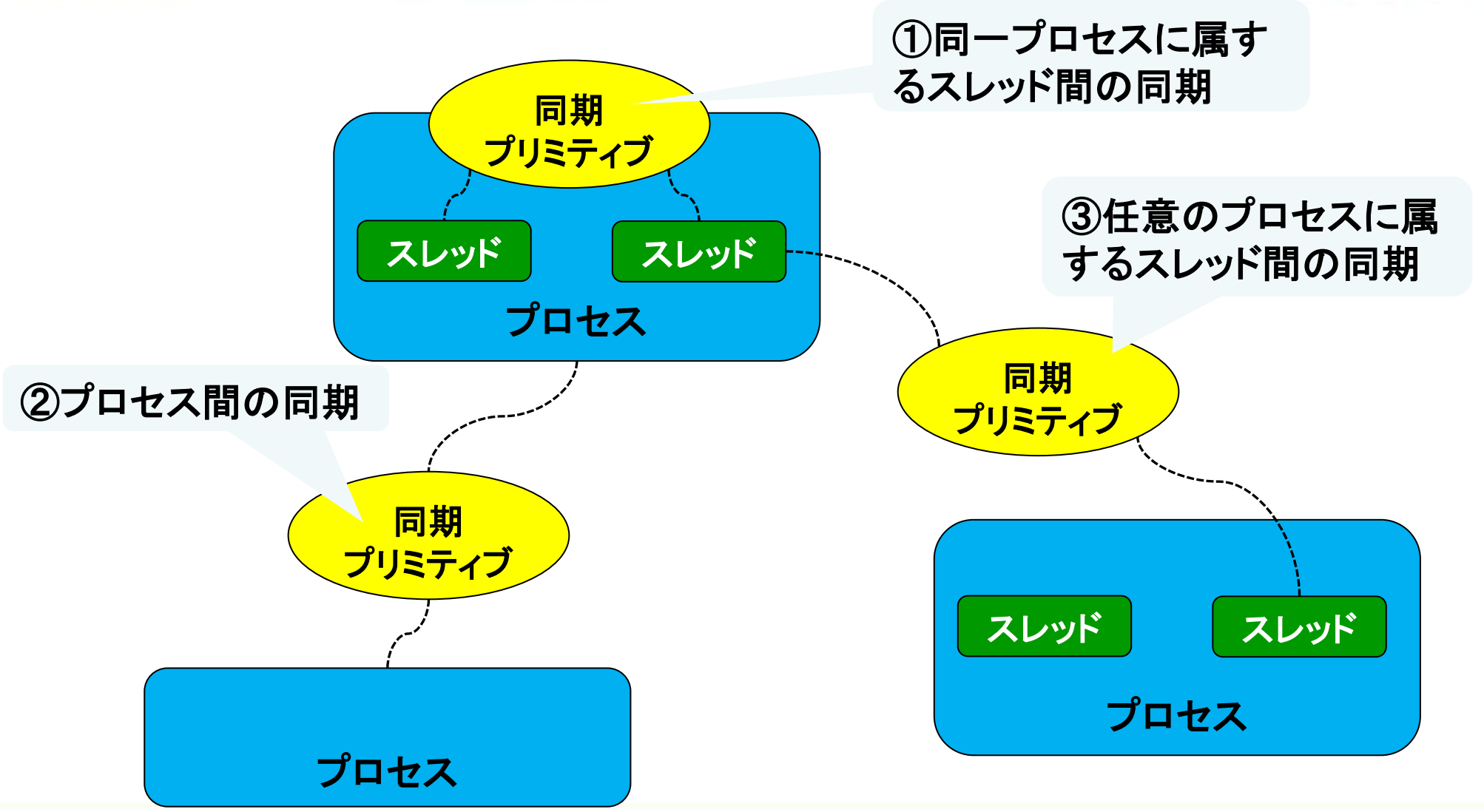
脅威を緩和する実践テクニック

1. 同期プリミティブを活用する
2. 不可分（アトミック）な操作を行う
3. ファイルを正しく特定し操作する
4. ファイルをロックして排他制御する
5. ロックファイルを使ってプロセスを同期させる
6. システム管理的アプローチを適用する

排他制御を実装する同期プリミティブ

- ミューテックス変数
  - リソースへの読み書き両方を排他するなどシンプルな制御向き
- セマフォ
  - カウンタが敷居値を超えたらアクセスを許可しないなど、許される並列性の制御を行うなど
  - バイナリセマフォはミューテックス変数と同様シンプルな制御
- 条件変数（状態変数、モニタ）
  - 複雑なシンクロ管理、状態を検知してシグナル
- ロック変数（リーダー／ライターロック）
  - ロックを持たない他のスレッドに読み込みを許可し、書き込みのみ排他制御を行うなど、状況に応じたアクセス制御が可能
  - ロック待ちを最小限にしてパフォーマンスを上げるなど

同期対象別の3つのタイプ



POSIX 同期プリミティブ

- ミューテックス変数
  - **pthread_mutex_lock()**
  - **スレッド間**
- セマフォ
  - **sem_wait(), sem_post()**
  - **スレッド間**
- 条件変数
  - **pthread_cond_wait()**
  - **スレッド間**
- ロック変数
  - **pthread_rwlock_wrlock(), pthread_rwlock_rdlock()**
  - スレッド間

Windows 同期プリミティブ

- ミューテックス変数
  名前付きミューテックス
  - CreateMutex(),WaitForSingleObject(),ReleaseMutex()
  - プロセススレッド間

  CRITICAL_SECTION オブジェクト
  - EnterCriticalSection(), LeaveCriticalSection()
  - スレッド間
- セマフォ
  - CreateSemaphore(), ReleaseSemaphore()
  - スレッド間
- 条件変数
  - WakeConditionVariable(), SleepConditionVariableCS()
  - スレッド間
- ロック変数
  - slim reader/writer ロック(SRWロック)
  - スレッド間

脅威を緩和する実践テクニック

JPCERT CC®

1. 同期プリミティブを活用する
2. 不可分(アトミック)な操作を行う
3. ファイルを正しく特定し操作する
4. ファイルをロックして排他制御する
5. ロックファイルを使ってプロセスを同期させる
6. システム管理的アプローチを適用する

不可分操作により競合ウィンドウを閉じる

典型的な処理フローである、ファイル検査後のファイルオープンは、TOCTOU競合状態となり易い。

処理ステップ:

1. ファイルの検査(存在、アクセス権限、その他)
2. ファイルのオープン

この二つのステップの間に競合ウィンドウを開かない(作らない)実装を行う。

TOCTOU競合状態の例:access() → fopen()

JPCERT CC®

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
  FILE *fp;
  if (access("/dir/some_file", W_OK) == 0){
    printf("access granted.¥n");
    fp = fopen("/dir/some_file", "wb+");
    /*ファイルに書き出す */
    fclose(fp);
  }      . . .
  return 0;
}
```

access() 関数を呼び出しファイルの存在と書き込み権限があることをチェックしている。(Time of Check)

競合ウィンドウ

書き込みのためにファイルをオープン。(Time of Use)

TOCTOU 攻撃と対策

JPCERT CC®

■ 攻撃

競合ウィンドウの間に次のシェルコマンドを実行する。

```
unlink /dir/some_file
ln -s /etc/passwd /dir/some_file
```

■ 対策

`access()` の呼び出しを修正し、競合ウィンドウを閉じる。

プロセスのUIDを実UIDに降格する。
`fopen()` でファイルをオープンする。
ファイルのオープンが成功したかどうかを確認する。

TOCTOU競合状態への対策例

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
  FILE *fp;
  if( setuid(getuid()) == 0 ){
    fp = fopen("/dir/some_file", "wb+");
    if(fp){
      printf("access granted.¥n");
      /*ファイルに書き出す */
      fclose(fp);
    }
  }      . . .
  return 0;
}
```

access() 関数と同等の権限確認を行うために、実ユーザIDを実効ユーザIDに設定

検査時と使用時の間の競合ウィンドウは閉じられている

C++ TOCTOUの例：存在確認後にオープン

JPCERT CC®

競合ウィンドウの間に
/tmp/some_file という名前
でシンボリックリンクを作ることで
攻撃に使われてしまう。

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
  ofstream outStrm;
  ifstream chkStrm;
  chkStrm.open("/tmp/some_file", ifstream::in);
  if (!chkStrm.fail()){
     /* 一時ファイルが既に存在するのでエラー処理 */
  }
  outStrm.open("/tmp/some_file", ofstream::out);
  ...
}
```

検査のタイミング

使用のタイミング

競合
ウィンドウ

C++ における TOCTOU

- 標準化前の C++ では、`<fstream.h>` の実装によっては `ios::nocreate` や `ios::noreplace` の各フラグを利用してファイルの作成を制御することができた。

- これらのフラグはあまりにプラットフォーム依存なので、`<fstream.h>` ヘッダを置き換えた標準の `<fstream>` ライブラリには採用されなかった。

- `ios::nocreate` フラグの使用も廃止されたので、 `fstream` には同等なアトミック操作を行うものが存在しない。

TOCTOU競合状態への対策例

```
int main() {
  int fd;
  FILE *fp;
  if ((fd = open("/tmp/some_file",
    O_EXCL|O_CREAT|O_TRUNC|O_RDWR, 0600)) == -1)
  {
      err(1, "/tmp/some_file");
  }
  fp = fdopen(fd, "w");
  ...
```

open() 関数の引数に O_EXCL を使う。

ストリームをオープンする際にファイル名ではなくファイル記述子を使う。

fopen() vs. open()

JPCERT CC®

fopen()

- C99
- FILE * I/O stream を返す
- 文字列でモードを指定 (ex.''w+'')
- 中で open() を呼んでいる
- fclose()でクローズする

open()

- POSIX (not C99)
- int (ファイルディスクリプタ)を返す
- bitmaskでモードを指定
- システムコールである close() でクローズする
- 第三引数でパーミッションを指定する(新規ファイル作成時)

O_CREAT と O_EXCL

O_CREAT と **O_EXCL** がセットされていると、ファイルが既に存在すれば **open()** は失敗する。

他のスレッドからもアトミックな動作。

O_EXCL と **O_CREAT** がセットされており、パスにシンボリックリンクが指定されている場合、シンボリックリンクの内容に関係なく、**open()** は失敗し、**errno** に [EEXIST] が設定される。

O_EXCL がセットされていて、**O_CREAT** がセットされていなければ、結果は未定義である。

脅威を緩和する実践テクニック

JPCERT CC®

1. 同期プリミティブを活用する
2. 不可分(アトミック)な操作を行う
3. ファイルを正しく特定し操作する
4. ファイルをロックして排他制御する
5. ロックファイルを使ってプロセスを同期させる
6. システム管理的アプローチを適用する

ファイル名を使ったファイル操作に注意

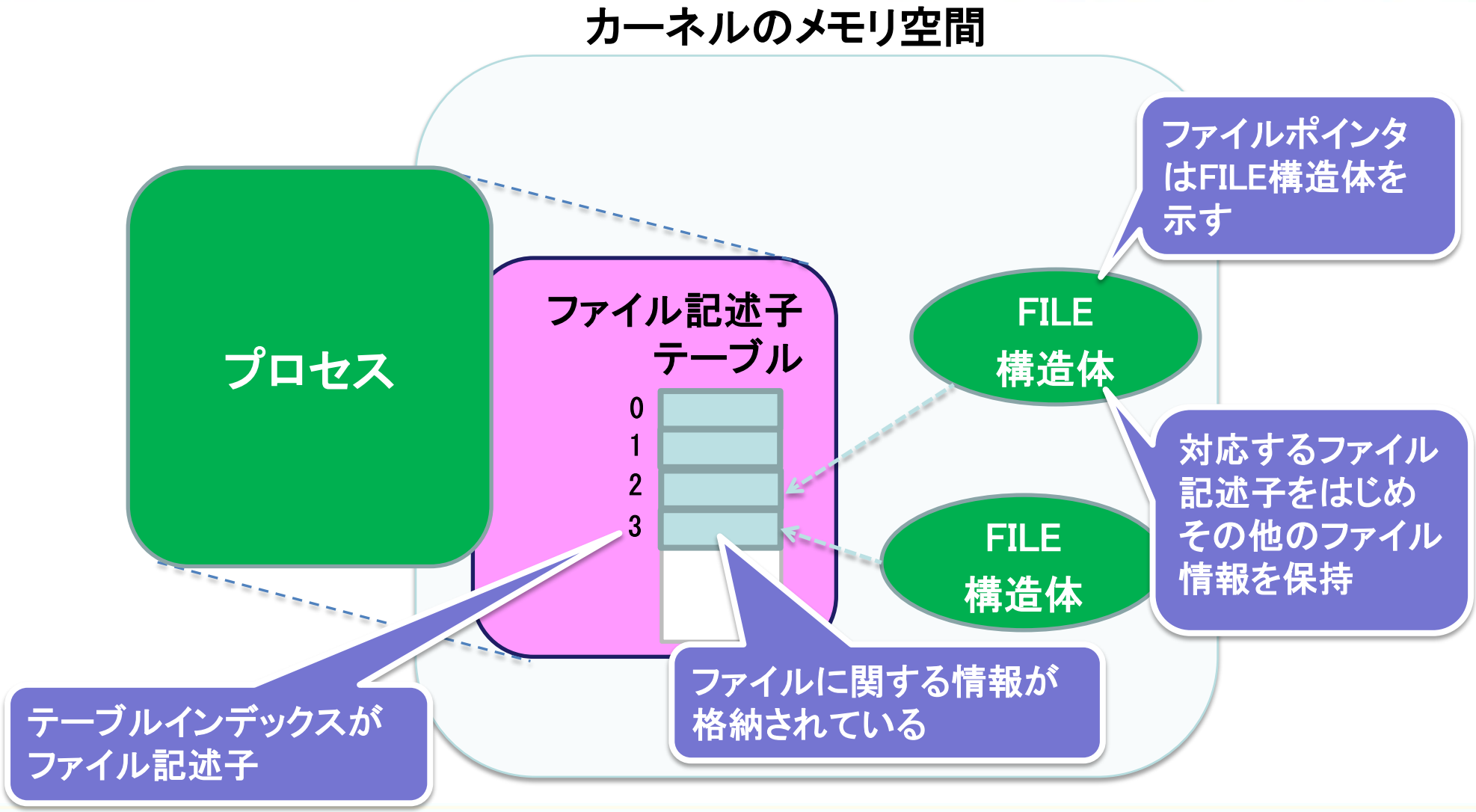
ファイル操作に起因する脆弱性の多くは、想定外のファイルオブジェクトへのアクセスが原因

- ファイル名は、実体であるファイルオブジェクトにゆるやかにひもづけられているに過ぎない
- ファイル名はファイルオブジェクト自体に関する情報をまったく提供しない
- ファイル操作でファイル名が使われる場合、その度にファイル名とファイルオブジェクトのバインディングが行われる

できるだけファイル名に頼らない

- ファイル記述子、ファイルポインタを使い意図したオブジェクトに正しくアクセスする。
- ファイルに関連して競合状態が発生する場合、競合オブジェクトはそのファイル自身ではなくファイルが置かれているディレクトリであることが多い。
- symlink 攻撃は、ディレクトリエントリの変更を行うことで成立する。
- ファイルはファイル記述子を使ってアクセスされている限り、symlink 攻撃をうけることはない。

ファイル記述子とファイルポインタ



危険なファイル再オープンの例

```
char *file_name = "good.txt";
FILE *fp = fopen(file_name, "w");
if (fp == NULL) { /* エラー処理 */ }
/*... ファイルに何か書き込む ...*/
fclose(fp);
fp = NULL;

/* … */

fp = fopen(file_name, "r");
if (fp == NULL) { /* エラー処理
/*... ファイルから何か読み取る ...*/
fclose(fp);
fp = NULL;
```

ファイルを書き込みのために
オープン＆クローズした後、同
じファイル名で読み取りのた
めにオープン＆クローズ

ファイル名だけに頼って
ファイルを識別

file_name が同じファイルを指
しているって言える？No!

修正例： ファイルを再オープンしない

ファイル書き込み時と、続く読み取り時、両方のタイミングで同じファイルオブジェクトが操作対象となる。

ファイルディスクリプタと FILE ポインタはOSによって結びつけられている

```
char *file_name = "good.txt";
FILE *fp = fopen(file_name, "w+");
if (fp == NULL) { /* エラー処理 */ }
/*... ファイルに何か書き込む ...*/

/* … */

fseek(fp, 0, SEEK_SET);
/*... ファイルから何か読み取る ...*/
fclose(fp);
fp = NULL;
```

このときのファイルは、書き込みのときのファイルと同じ である。

ファイルの再オープンが必要な場合

- ファイルはできるだけ再オープンしないほうがよい
- ただし、長期間起動し続けるプログラムは、ファイルディスクリプタを枯渇させないためにクローズせざるを得ないケースも

どうやってTOCTOU競合状態を解決したらよいのか？

複数のファイル属性情報でファイルを識別

- UNIX のファイルは、ファイル名に加え、ファイルのシリアル番号（iノード）やデバイスなどの属性によって識別できることが多い
- 自分で作成してクローズしたファイルに特定の属性情報を記録し、ファイルを再度オープンするときにその情報でファイルの同一性を確認
- ファイルの複数の属性情報を比較することで、適切なファイルを正しく識別する確率を高める

ファイルの同一性確認のため stat()を利用

POSIX **stat()**関数を使って、ファイルの属性情報を取得できる。

```
struct stat st;
char* file_name = "good.txt”
if (stat( file_name, &st) == -1) {
    /* Handle Error */
}
```

st 構造体には “**good.txt**”に関する情報が格納される。

stat 構造体

JPCERT CC®

```
struct stat {
    dev_t     st_dev;     /* ファイルがあるデバイスの ID */
    ino_t     st_ino;     /* inode 番号 */
    mode_t    st_mode;    /* アクセス保護 */
    nlink_t   st_nlink;   /* ハードリンクの数 */
    uid_t     st_uid;     /* 所有者のユーザ ID */
    gid_t     st_gid;     /* 所有者のグループ ID */
    dev_t     st_rdev;    /* デバイス ID (特殊ファイルの場合) */
    off_t     st_size;    /* 全体のサイズ (バイト単位) */
    blksize_t st_blksize; /* ファイルシステム I/O でのブロックサイズ */
    blkcnt_t  st_blocks;  /* 割り当てられた 512B のブロック数 */
    time_t    st_atime;   /* 最終アクセス時刻 */
    time_t    st_mtime;   /* 最終修正時刻 */
    time_t    st_ctime;   /* 最終状態変更時刻 */
};
```

デバイスID と i-node番号による特定

stat 構造体のメンバ **st_ino** と **st_dev** を使うことで、ファイルを一意に特定できる。

```
struct stat st1; //Time of Use その１時点の情報
struct stat st2; //Time of Use その２時点の情報

/* TOUその１時点で stat()で st1 に情報を格納 */
/* TOUその２時点で stat()で st2 に情報を格納 */

/* TOUその１とTOUその２の情報を比較 */
if ((st1.st_dev != st2.st_dev) ||
    (st1.st_ino != st2.st_ino)) {
  /* st1 と st2 は別々のファイルを参照している */
}
```

Check-Use-Check パターンの例

```
char *file_name = "good.txt";
struct stat orig_st;
if (stat(file_name, &orig_st) == -1) {
  /* エラー処理 */
}
FILE *fp = fopen(file_name, "w");
if (fp == NULL) { /* エラー処理 */ }
/*... ファイルに書き込む ...*/
fclose(fp);
/* 何らかの処理が行われる */
int fd;
fd = open(file_name, O_RDONLY);
if (fd == -1) {
  /* エラー処理 */
}
struct stat new_st;
if (fstat(fd, &new_st) == -1) {
  /* エラー処理 */
}
if ((orig_st.st_dev != new_st.st_dev) ||(orig_st.st_ino != new_st.st_ino)) {
  /* ファイルが改ざんされている! */
}
```

ファイルの情報を取得する(check)

ファイルを使う (use)

攻撃者はファイルが再オープンされるまえにファイルを改ざんできる

直前でオープンしたファイルの情報を取得する

このテストによってプログラムのコントロールフロー外でファイルが変更されたかどうかが分かる

Check-Use-Check パターンの例（修正）

```
char *file_name = "good.txt";
int fd;
fd = open(file_name, O_CREAT|O_WRONLY|O_TRUNC);
if (fd == -1) { /* エラー処理 */ }
struct stat orig_st;
if (fstat(fd, &orig_st) == -1) {
  /* エラー処理 */
}
/*... ファイルに書き込む ...*/
close(fd);
/* 何からの処理が行われる */
fd = open(file_name, O_RDONLY);
if (fd == -1) {
  /* エラー処理 */
}
struct stat new_st;
if (fstat(fd, &new_st) == -1) {
  /* エラー処理 */
}
if ((orig_st.st_dev != new_st.st_dev) ||(orig_st.st_ino != new_st.st_ino)) {
  /* ファイルが改ざんされている! */
}
```

fopen( )の代わりに、open( )でファイルディスクリプタを取得

stat() の代わりに、ファイルディスクリプタを引数に取る fstat() を利用

stat(), fstat(), lstat()利用上の注意点

stat(), fstat(), lstat() はファイルの状態に関する情報を取得するために使用可能

fstat()と stat()

- **stat()** 関数はファイル名を引数に取るが、**fstat()** 関数は ファイルディスクリプタを引数に取る
- 既にオープンしているファイルには **fstat()** が使用可能

lstat()と stat()

- **lstat()**関数は シンボリックリンクの場合、リンクファイル自体の情報を返す
- **stat()**はリンク先のファイルの情報を返す

Check-Use-Check を見直す必要があるケース

紹介した check-use-check パターンは、ファイルの同一性のみを確認しており、内容の改変を検知できない。

⇒より多くの属性情報を比較対象とすることで、検知の確率を上げることはできる。

ログローテーションし、ファイルの i-node 番号が変わるようなプログラムにおいて、ファイル属性に基づくチェックは意味をなさない。

⇒システム管理的な対策アプローチ含め他の対策を検討する必要がある。

ファイル名を引数とする関数に注意

C99 標準関数:

- `remove()` と `rename()`
- `fopen()` と `freopen()`

POSIX 関数:

- `link()` と `unlink()`
- `mkdir()` と `rmdir()`
- `mount()` と `unmount()`
- `lstat()`
- `mknod()`
- `symlink()`
- `utime()`

remove(), rename() 利用の際の注意点

JPCERT CC®

remove()

- 対象とするファイルがオープンされていた場合、動作は処理系依存となる。

rename()

- 指定した新しい名前のファイルが既に存在する場合、動作は処理系依存となる。

プロセス複製時に引き継ぐリソースに注意

JPCERT CC®

- 子プロセスは親プロセスのファイル記述子テーブルのコピーを持つ。
- 親の不必要なエントリにより、子プロセスのファイル記述子テーブルがあふれ、サービス不能状態に陥ることも考えられる。
- このため、**`stdin`**、**`stdout`**、**`stderr`** 以外のファイルはすべて、子プロセスを `fork` する前に閉じておくことが望ましい。

脅威を緩和する実践テクニック

JPCERT CC®

1. 同期プリミティブを活用する
2. 不可分(アトミック)な操作を行う
3. ファイルを正しく特定し操作する
4. ファイルをロックして排他制御する
5. ロックファイルを使ってプロセスを同期させる
6. システム管理的アプローチを適用する

ファイルをロックして排他制御

JPCERT CC®

複数のプロセスが同一のファイルにアクセスする競合状態を防ぐために、ファイルまたはその一部をロックする。

ファイルロックは2種類:

- アドバイザリロック(Advisory Lock)
- 強制ロック(Mandatory Lock)

強制ロックとアドバイザリロック

- 強制ロックは、ロックされたファイル領域にアクセスしようとする他の全てのプロセスが強制的に制限対象
- アドバイザリロックは、 強制できるものではなく、プロセス間での利用ルールに関する合意が必要

- Linux は、強制ロックとアドバイザリロックを実装
- Windows は、強制ロックのみ

Windows におけるファイルロック

ロック確保:**LockFile(),LockFileEx()**

ロック解除:**UnlockFile()**

二種類の強制ロックモード:

- **共有ロック:LockFileEx() で指定可**
  - 他のプロセスはファイルのロック指定領域への書き込み禁止
  - 他のプロセスはロック部分に読み取りアクセス可能
- **排他ロック:LockFile() のモード, LockFileEx() で指定可**
  - 他のプロセスはファイルのロック指定領域へ読み書き禁止

Linux におけるファイルロック

- flock():アドバイザリロック
- lockf():アドバイザリ／強制ロック
- fcntl():アドバイザリ／強制ロック

Windowsと同様に共有ロック、排他ロックの利用が可能。

Linux における強制ロックは注意して使う

次の点に十分注意した上で利用を検討:

- ローカルファイルシステム上のみで有効(ネットワークファイルシステムには未対応)
- ファイルシステムのマウント時に強制ロックが有効でなければならないが、デフォルトでは無効
- ロックは `setgid` ビットを立て、グループの実行ビットを無効にしなくてはならないが、他のプロセスがそれを無効化することでロックを破れる

場合により**ロックファイル**を使うことを検討

脅威を緩和する実践テクニック

JPCERT CC®

1. 同期プリミティブを活用する
2. 不可分(アトミック)な操作を行う
3. ファイルを正しく特定し操作する
4. ファイルをロックして排他制御する
5. ロックファイルを使ってプロセスを同期させる
6. システム管理的アプローチを適用する

ロックファイルを使用する目的・状況

- 信頼できる複数プロセスが共有するリソースへのアクセスを同期させ、競合状態の発生を防ぐ
- 強制／アドバイザリロックが利用できない
- リソースを共有する複数のプロセスは、ロックファイル名や配置場所について合意可能

ロックファイルの使用ルール

共有リソースに対するロックの状態は:

- ロックファイルが有る

  ⇒ロックが確保されている

- ロックファイルが無い

  ⇒ロックは解放されている

但し、無効なロックファイルが存在する可能性への配慮も必要

ロックファイル管理(スピンロック)の実装例

```
/* ロックを確保 */
int lock(char *fn) {
  int fd;
  int sleep_time = 100;
  while (((fd=open(fn,O_WRONLY|O_EXCL|O_CREAT,0)) == -1)
          && errno == EEXIST) {
    usleep(sleep_time);
    sleep_time *= 2;
    if (sleep_time > MAX_SLEEP) sleep_time = MAX_SLEEP;
  }
  return fd;
}

/* ロックを解放 */
void unlock(char *fn) {
  if (unlink(fn) == -1) err(1, "file unlock");
}
```

lock() と unlock() は、それぞれファイル名を引数にとり、それを共通のロックオブジェクトとして使用する。

ロックファイルが既に存在するとファイルオープンは失敗し続ける。

ロックファイルの有効性確認対策

- lock() 関数を修正し、ロックファイル中にPIDを記録
- 修正版 lock() 関数は、ロックファイルが存在する場合、記録された PID が実行中のプロセスであるかを確認
- 該当するPIDを持つプロセスが存在しなければ、ロックファイルは無効と判断し、ロックを確保した上で、新しい PID でロックファイルの内容を更新
- しかし、終了したプロセスの PID は、別のプロセスに再利用されている可能性は否定できない

Windows 名前付きミューテックス

Windows では、プロセス間同期のために名前付きミューテックス・オブジェクトが利用できる。

作成: **CreateMutex()**
確保: **WaitForSingleObject()**
解放: **ReleaseMutex()**

ファイルシステムに似た名前空間を持ち、ロックファイルの代用としてプロセス間同期に利用可能。

脅威を緩和する実践テクニック

JPCERT CC®

1. 同期プリミティブを活用する
2. 不可分(アトミック)な操作を行う
3. ファイルを正しく特定し操作する
4. ファイルをロックして排他制御する
5. ロックファイルを使ってプロセスを同期させる
6. システム管理的アプローチを適用する

最小権限の原則

最小権限の原則とは、システムの全てのプログラムおよびユーザはそのジョブを行うために必要な最低限の権限で操作を行うということ。
[Saltzer 74, Saltzer 75]

- 不必要に高い権限でプロセスを実行しない
- 高い権限での実行が必要なプロセスでは、共有資源にアクセスする時に権限を最小に

プロセスが競合ウィンドウを実行中、低い権限しか持たなければ、攻撃の影響範囲は限定される。

共有ディレクトリの利用は最小限に

- グループのユーザが同じディレクトリに対して書き込み権限を持っている状況は、ファイルのみが共有されている状況よりもはるかに危険性が高い。

- ハードリンクやシンボリックリンク攻撃への脅威を緩和するためにも、ディレクトリの共有は必要最小限にするべきである。

- できるだけ /tmp などの共有ディレクトリを利用せずによりセキュアなディレクトリを利用する。

もくじ

JPCERT CC®

1. 競合状態とその脅威
2. 競合状態の発生メカニズム
3. TOCTOU競合状態
4. 脅威の緩和方法
   - 脅威の緩和アプローチ
   - 実践テクニック
   - 解析ツール活用
5. まとめ
6. 補足資料：デッドロックについて

2種類の解析ツール

競合状態を発見・防止するツールには2種類ある

静的解析ツール

- ソースコードから競合状態が発生するかを分析

動的解析ツール

- 仮想環境などでプログラムを実行させ、競合状態の発生を検知

静的解析ツールの特徴

- 「静的」＝プログラムを実行しない
- ソースコードを分析する
  - 脆弱性に関する脅威や修正方法などを提供するツール
  - 型やスタイルチェック、バグ発見などが可能なツール
- 検知漏れの可能性
  - ツール分析が困難な場合、他の対策が必要となるケース
- 誤検知の可能性
  - 出力結果を吟味する必要性
- オープンソース / 商用製品いろいろ

http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

動的解析ツールの特徴

- 「動的」＝プログラムを実行する
- 実行時の挙動を分析する
- 検知漏れの可能性
  - 実行されない部分は検査できない
  - 実行分岐数の増加に伴い検知漏れの可能性は高く
- 実行時オーバヘッドが大きい
- オープンソース / 商用製品いろいろ

http://en.wikipedia.org/wiki/Dynamic_code_analysis

静的解析ツールの例

JPCERT CC®

WARLOCK

"WARLOCK: A Static Data Race Analysis Tool", in Proceedings of the USENIX Annual Technical Conference, 1993

ITS4

http://www.cigital.com/its4/

RacerX

"RacerX: Effective, Static Detection of Race Conditions and Deadlocks", In *Proceedings of the Symposium on Operating Systems Principles*, pages 237-253, October 2003

RATS

http://www.fortify.com/security-resources/rats.jsp

FlawFinder

http://www.dwheeler.com/flawfinder/

動的解析ツールの例

JPCERT CC®

Eraser

“A Dynamic Data Race Detector for Multi-Threaded Programs”, ACM Transactions on Computer Systems, Vol.15, No.4, 1997

MultiRace

“Efficient On-the-Fly Race Detection in Multithreaded C++ Programs”, in Proceedings of the 9th ACM SIGPALN Symposium on Principles and Practice of Parallel Programming, 2003

Intel Thread Checker

http://www.intel.com/software/products/threading/

Sun Thread Analyzer

http://docs.sun.com/app/docs/doc/820-0619

分析対象サンプルコード:choi.c

JPCERT CC®

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

int main(int argc, char *argv[]){
  FILE *fd;
  if (access("./some_file", W_OK) == 0){
    printf("access granted.¥n");
    fd = fopen("./some_file", "wb+");
    /* writing to the file ... */
    fclose(fd);
  }
  return 0;
}
```

RATSによる分析結果の例

JPCERT CC®

```
% rats choi.c
Entries in perl database: 33
Entries in python database: 62
Entries in c database: 336
Entries in php database: 55
Analyzing choi.c
choi.c:6: Medium: access
A potential TOCTOU (Time Of Check, Time Of Use) vulnerability exists.  This is
the first line where a check has occured.
The following line(s) contain uses that may match up with this check:
8 (fopen)

Total lines analyzed: 15
Total time 0.000328 seconds
45731 lines per second
%
```

FlawFinderによる分析結果の例

```
% flawfinder choi.c
Flawfinder version 1.26, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 158
Examining choi.c

choi.c:6:  [4] (race) access:
  This usually indicates a security flaw.  If an attacker can change anything along the
path between the call to access() and the file's actual use (e.g., by moving files), the
attacker can exploit the race condition. Set up the correct permissions (e.g., using
setuid()) and try to open the file directly.

choi.c:8:  [2] (misc) fopen:
  Check when opening files - can an attacker redirect it (via symlinks), force the
opening of  special file type (e.g., device files), move things around to create a race
condition, control its ancestors, or change its contents?.

Hits = 2
Lines analyzed = 14 in 0.53 seconds (542 lines/second)
Physical Source Lines of Code (SLOC) = 11
Hits@level = [0]   0 [1]   0 [2]   1 [3]   0 [4]   1 [5]   0
Hits@level+ = [0+]   2 [1+]   2 [2+]   2 [3+]   1 [4+]   1 [5+]   0
Hits/KSLOC@level+ = [0+] 181.818 [1+] 181.818 [2+] 181.818 [3+] 90.9091 [4+] 90.9091 [5+]   0
Minimum risk level = 1
```

Not every hit is necessarily a security vulnerability.

There may be other security vulnerabilities; review your code!

もくじ

ファイルアクセスは狙われる

ファイルのオープン、リード、ライト、クローズ、いたるところで競合ウィンドウは発生する。

ファイルアクセスは狙われる

- オープンされたファイルはスレッド間で共有される
- ファイルは他のプロセスにアクセスされうる

攻撃は以下の点を突いて行われる。

- ファイルのアクセス権限
- ファイルのネーミングルール
- ファイルシステムの特徴

信頼できる制御フローの競合対策

信頼できる制御フローにおいては、様々な同期プリミティブを利用することができる。

しかし、誤った同期プリミティブの使い方は、 デッドロック状態を引き起こす可能性があるので注意。

信頼できる制御フローにおいては、ファイルの存在をロックの保持とするような、特定のルールに従った排他制御メカニズム(ロックファイル)を利用することもできる。

信頼できない制御フローの競合対策

信頼できない制御フローに対しては、あるルールに従う、あるいは、特定の仕組みを利用することを前提とする排他制御メカニズムは無効である。

よって、OSあるいはファイルシステムレベルで強制的に適用される「ファイルロック」などの排他制御メカニズムが必要となる。

これら排他制御メカニズムを利用することができない場合は、

- 競合ウィンドウを無くす、または、並行性を下げる
- 競合オブジェクトの共有度合を下げる
- 競合オブジェクトの更新機会を減らす

TOCTOU競合状態を作らない

1. 不可分なファイル操作の適用
2. ファイル記述子・FILEポインタを使ってファイルの特定と操作
3. ファイル名でアクセスせざるをえない場合、複数のファイルの属性情報をチェックしてファイルの同一性を検査
4. 選択肢が無い場合、セキュアディレクトリなどシステム管理的な対策アプローチを取る

もくじ

デッドロック状態とは

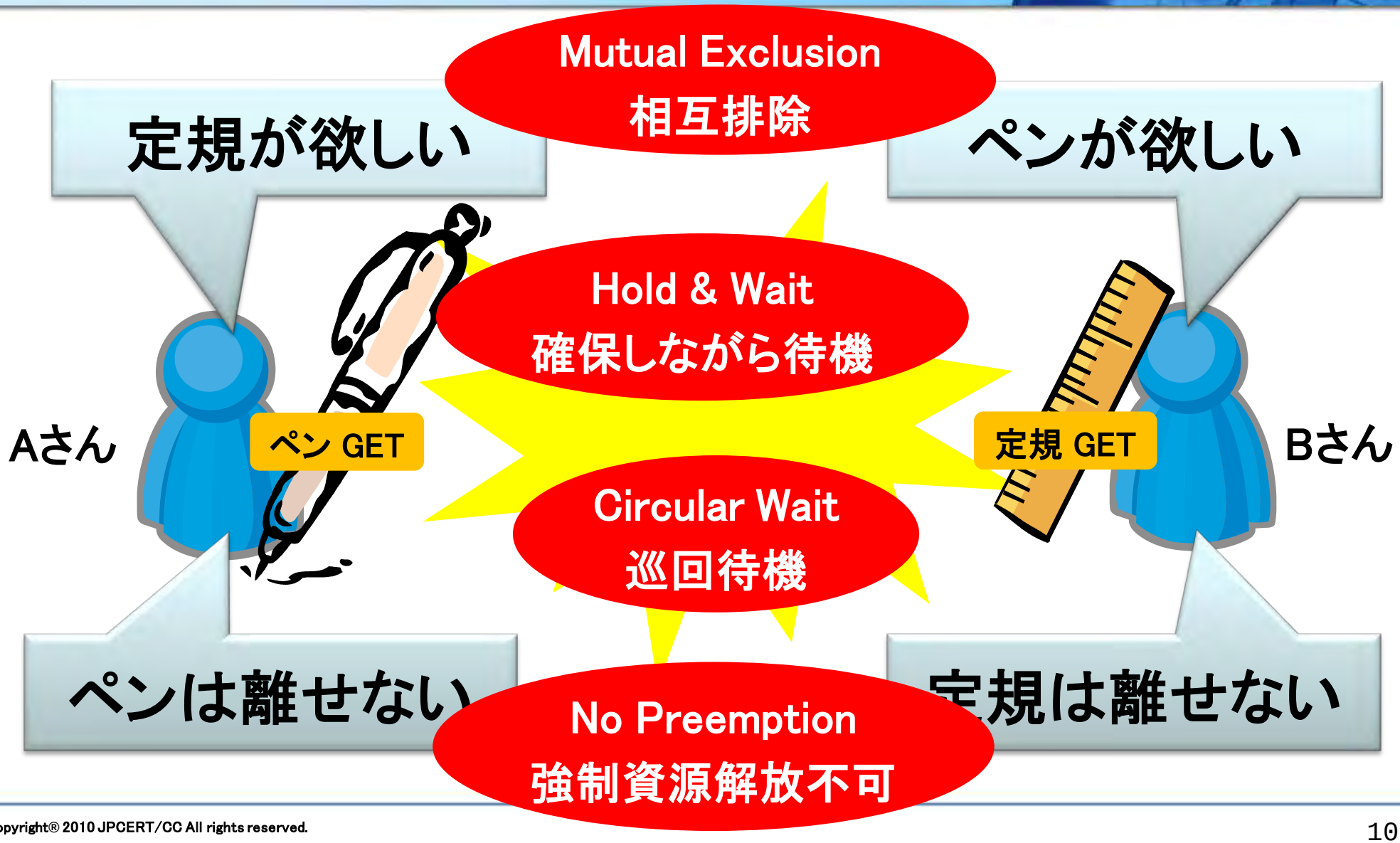
デッドロックとは、2つ以上の実行フローが互いに相手の処理フローをブロックしてしまい、すべての処理が継続できなくなってしまう状態のこと。

一連の並行処理を行う工程があった時に、あるフローが同期のために使用するオブジェクトを確保することで、同じ工程内で後に発生するフローの実行を妨げてしまうような場合に、デッドロックが発生しやすい。

デッドロックは、結果としてサービス運用妨害（DoS）攻撃を引き起こすことがある。

身近なデッドロック状態：直線を引く

デッドロックの必要条件



相互排除(Mutual Exclusion)

二つ以上の制御フローにおいて同時に使用することができない共有資源である。

資源であるペンと定規は、共に二人以上で同時に使用することはできない。

つまり、ペンと定規それぞれは、一度に一つの制御フローに割り当てられなければならない。

確保しながら待機(Hold & Wait)

少なくとも一つの資源を確保しながら、他の資源の確保を要求する。

例では、Aさん、Bさん、それぞれペンと定規を確保したままの状態で、他の共有資源(Aさんは定規、Bさんはペン)の確保を要求して、待機している。

巡回待機(Circular Wait)

資源の待機状態が巡回して、ループを形成している。

例では、

- Aさんは、資源「ペン」を持った状態で「定規」の確保を待機
- Bさんは、資源「定規」を持った状態で「ペン」の確保を待機

お互いが、相手の持つ資源の解放を待つ状態。
“リソースグラフのサイクル”とも呼ばれる。

強制的資源解放不可(No Preemption)

既に確保されている資源を強制的に開放させることができない。

例では、Aさん、Bさんそれぞれが確保している資源(ペンと定規)のどちらも強制的に開放させることができない。

デッドロック状態発生に影響が大きい要素

- プロセッサの速度
  - 速度ダウンで、競合ウィンドウの幅が広がる
- プロセスやスレッドのスケジューリングアルゴリズムの変更
  - 個々の制御フローにおける競合ウィンドウの実行タイミングに変化
- プログラム実行時に違うメモリの制約を適用する
  - スワップの多発によりスループットを低下、競合ウィンドウ幅の拡大と実行タイミングに変化
- プログラムの実行に割り込む可能性のある非同期イベント
  - 競合ウィンドウ中に別の競合ウィンドウが発生する可能性
- 同時に実行している他プロセスの状態
  - 他のプロセスがより多くの資源を確保している状態など

デッドロック状態を発生させる攻撃

典型的な攻撃は、様々な条件（前のスライドの例のこと）を自動的に変化させ続け、競合状態を明るみに出す。

計算機システムに対して異常な負荷をかけることで、競合ウィンドウの実行時間を必要とする長さを引き延ばすことができるかもしれない。

デッドロック状態が発生する可能性は、常にセキュリティ上の問題として捉えるべきである。

デッドロック状態の回避アプローチ

デッドロック回避の基本方針は、リソースを特定の順序に従い確保することで巡回待機を発生させない。

すべての共有資源に1からインクレメンタルに順序番号を付与した状態で、プロセスが共有資源 Rh (k < h) を確保している場合、共有資源 Rk を確保することを許さない。

つまり、プロセスが共有資源 Rh を確保している時、h よりも大きい資源番号をもった共有資源のみ確保できる。

ただし、マルチスレッドのプログラムでは、使用するデータ型やライブラリなどが、スレッドセーフでは無い場合、例えば、呼びだされたライブラリ関数が競合状態を生み出す可能性も否定できない。