

I233

オペレーティングシステム特論

田中 清史 (Kiyofumi Tanaka)
kiyofumi@jaist.ac.jp

日程

□ 講義日程

- 7月 9日(土) 13:50~17:20 (第1, 2回)
- 7月16日(土) 13:50~17:20 (第3, 4回)
- 7月23日(土) 13:50~17:20 (第5, 6回)
- 7月30日(土) 13:50~17:20 (第7, 8回)
- 8月 6日(土) 13:50~17:20 (第9, 10回)
- 9月 3日(土) 13:50~17:20 (第11, 12回)
- 9月10日(土) 13:50~17:20 (第13, 14回)

□ 試験

- 9月17日(土) 13:50~15:30

教科書

1. Andrew S. Tanenbaum, Herbert Bos:
“Modern Operating Systems” (4th ed.), Prentice Hall, 2014.

和訳書

Andrew S. Tanenbaum著, 水野忠則訳.
“モダンオペレーティングシステム(第2版)”
ピアソン・エデュケーション・ジャパン. 2004.

参考) Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating System Concepts (9th ed.), John Wiley & Sons, Inc., 2012.

講義計画 (Schedule)

1. イントロダクション／概要
2. プロセス1(プロセス, スレッド)
3. プロセス2(プロセス間通信, 同期)
4. プロセス3(スケジューリング)
5. デッドロック1(リソース, デッドロック)
6. デッドロック2(デッドロックの検出, 回復, 回避, 防止)
7. メモリ管理1(仮想記憶, ページング, ページテーブル)
8. メモリ管理2(ページ置き換えアルゴリズム)
9. メモリ管理3(設計時の課題, セグメンテーション)
10. 入出力1(I/Oハードウェア)
11. 入出力2(I/Oソフトウェア)
12. ファイルシステム1(ファイル, ディレクトリ)
13. ファイルシステム2(ファイルシステムの実装)
14. ファイルシステム3(ファイルシステムの例)
15. 試験

評価基準

- レポート問題2回(40点)
- 期末試験(60点)

オペレーティングシステムの概要



オペレーティングシステムの目的

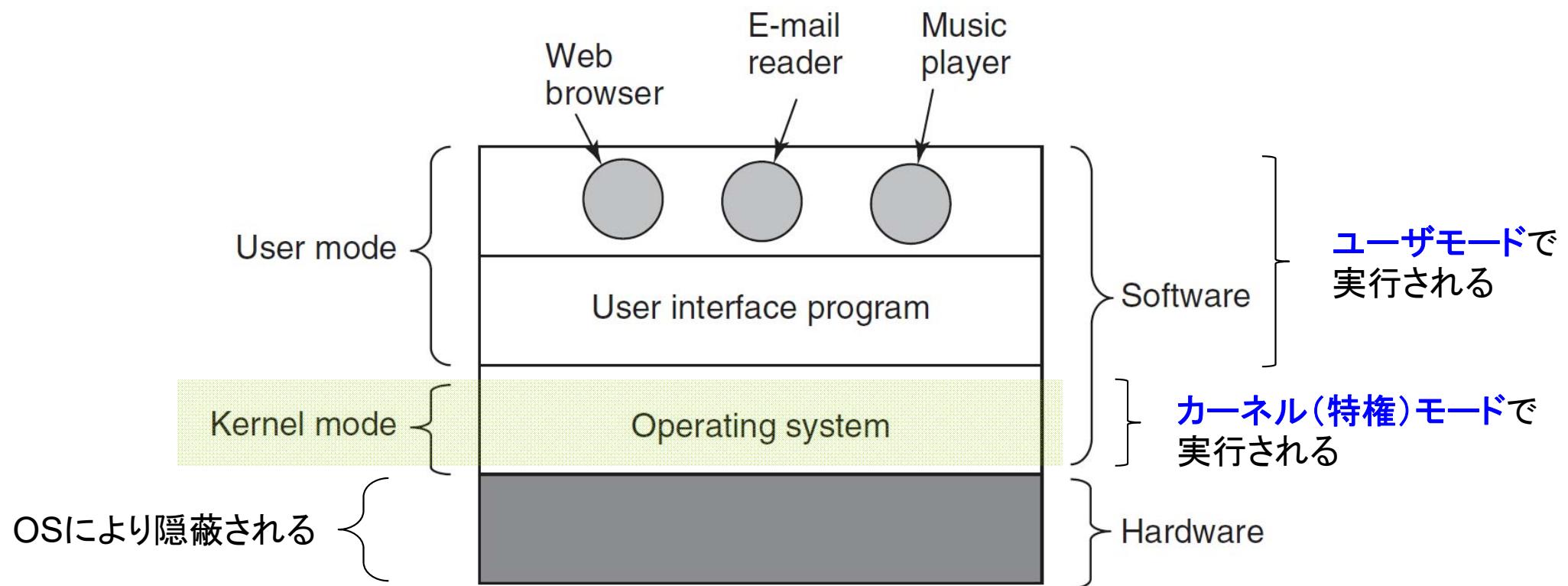
□ Silberschatz, et al. の書籍より

- 第一に, OSは計算機ハードウェアを管理するソフトウェアである
- 第二に, OSはユーザが**便利で効率よく**プログラムを実行できるよ
うな環境を提供するべきである
- (曖昧さ, 考え方の違いから)したがって, 内部的には, OSによって
構造が大きく異なる

概要

□ コンピュータシステムは以下から構成される

- ハードウェア
- システムプログラム
- アプリケーションプログラム



オペレーティングシステムとは何か

□ 仮想マシンとしてのOS

- ユーザからの要求を受け付けて、ハードウェアに対して処理を行う
- 様々なハードウェアの制御の(乱雑な)詳細を隠蔽
 - I/Oに関しては、ハードウェア制御をデバイスドライバによって隠す
- ユーザに仮想マシン(virtual machine)を提供し、より使いやすくする
 - システムコールによってユーザ(プログラマ)にサービスを提供

□ 資源マネジャーとしてのOS

- ハードウェアを多重化して、複数のプロセス間で共有可能にする
- 資源を多重化(共有化)する2つの方法
 - 各プロセスは資源を使用する時間を割り当てられる
 - CPU, プリンタなど
 - 各プロセスは空間的に分割された資源を割り当てられる
 - メモリ, ディスクなど

オペレーティングシステムの歴史(1)

□ 第一世代(1945 – 1955)

- 真空管, リレー
- プログラミング用「言語」は無し
 - 配線板(plugboard)あるいはパンチカードでプログラミング

□ 第二世代(1955 – 1965) – 初期のメインフレーム(mainframess)

- トランジスタ
- 一部プログラミング言語(Fortran等)を使用するジョブも存在
- バッチシステム(OSの元祖)(次スライド)

□ 第三世代(1965 – 1980)

- IC
- マルチプログラミング

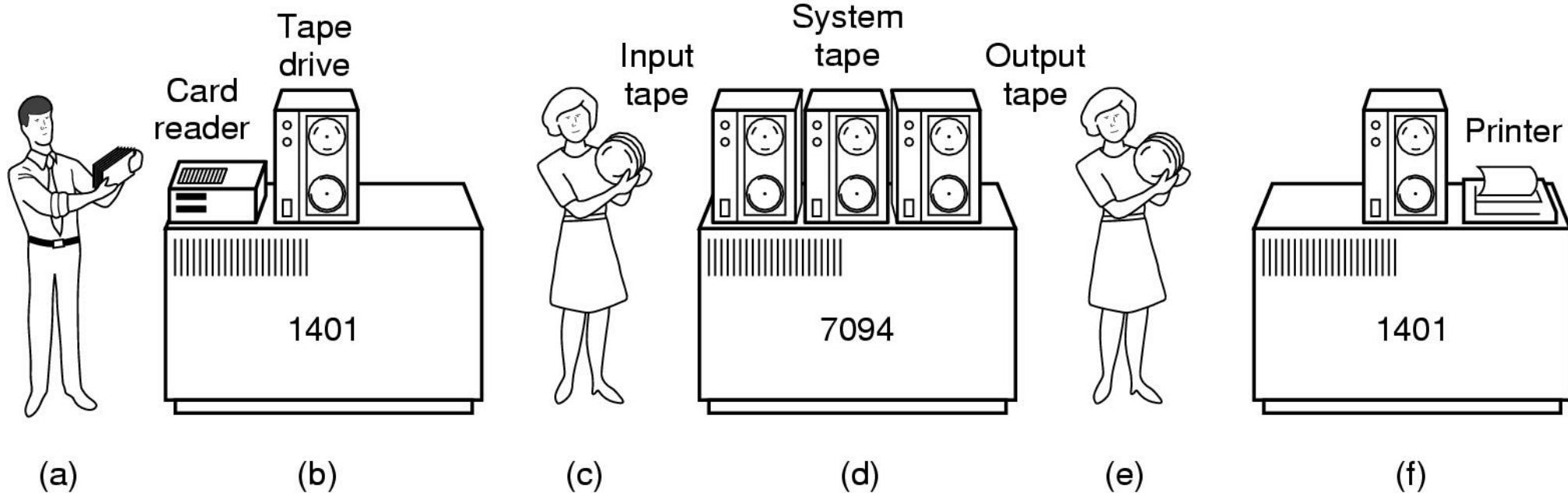
□ 第四世代(1980 – present)

- (V)LSI
- パーソナルコンピュータ

□ 第五世代(1990 – present) – モバイルコンピュータ

オペレーティングシステムの歴史(2)

□ 第二世代におけるバッチシステム

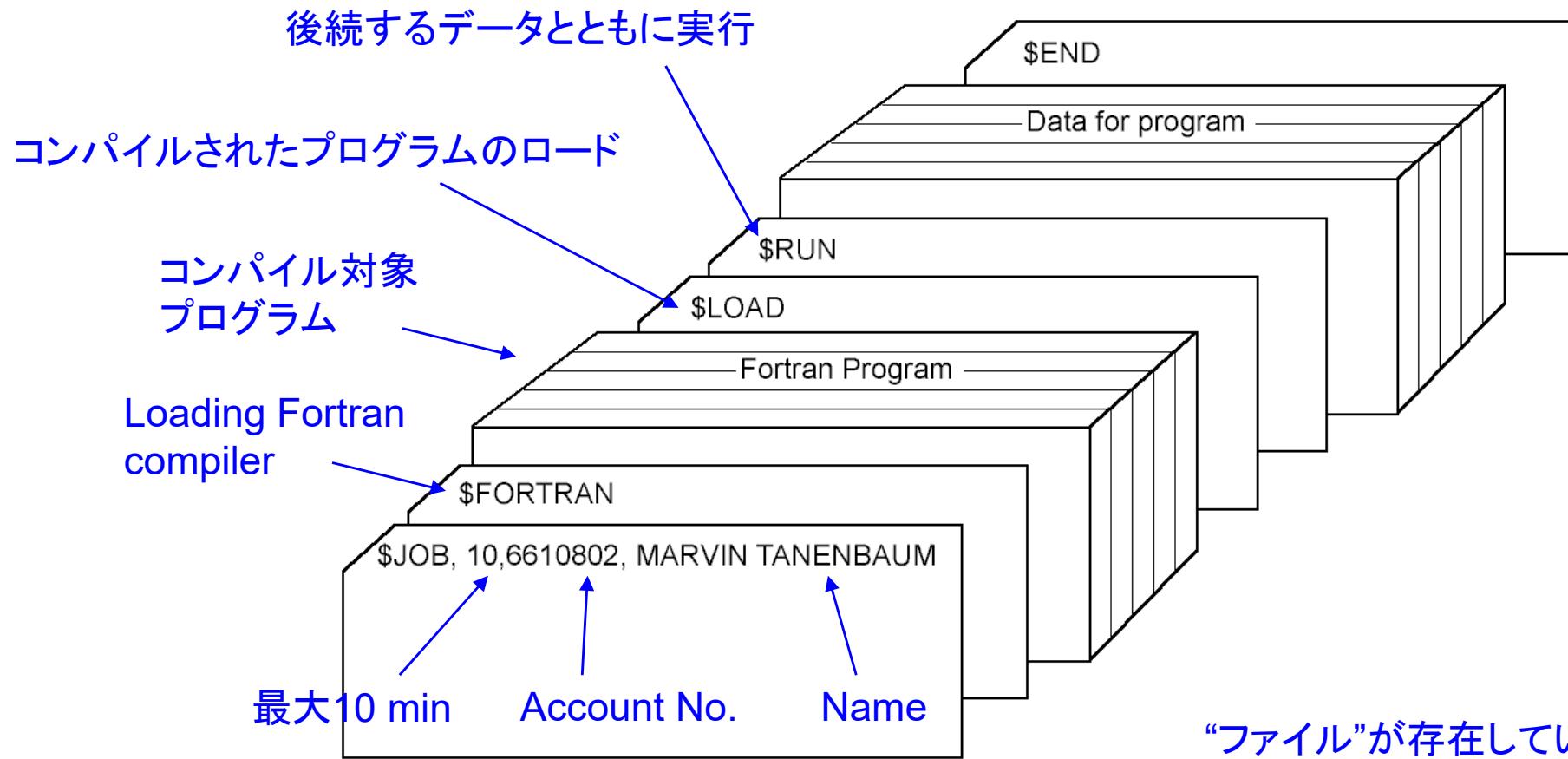


Early batch system

- (a) プログラマはカードを1401の所へ持っていく
- (b) 1401はカードを読み込み、ジョブのバッチを作成してテープへ書き込む
- (c) オペレータはテープを7094の所へ持っていく
- (d) 7094は計算を行う
- (e) オペレータは出力テープを1401の所へ持っていく
- (f) 1401は出力を印刷する

オペレーティングシステムの歴史(3)

□ 第二世代における典型的なFMS (Fortran Monitor System) ジョブの構成



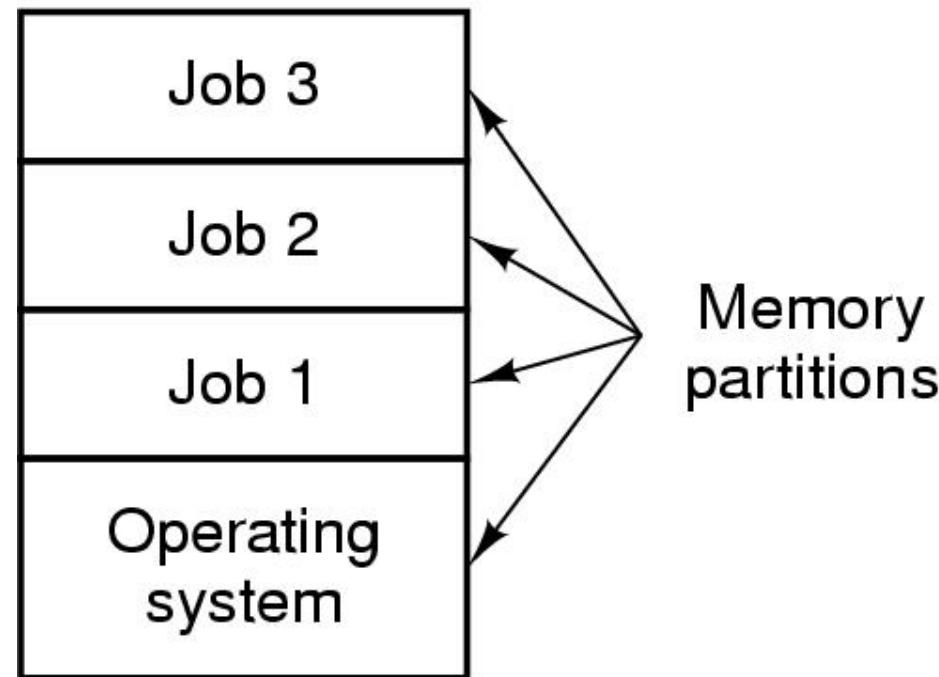
“ファイル”が存在していないため、このような煩雑な入力を用意する必要

オペレーティングシステムの歴史(4)

□ 第三世代の特徴

■ マルチプログラミングシステム

- I/Oを待つ間、ジョブを切り替える
- 保護が必要(ジョブ間で盗み見、改ざんをしないように)



この方法が、高速応答のための
“時分割方式”につながっていく

オペレーティングシステムの歴史(5)

□ 第三世代の例: IBM System/360 family

- ソフトウェア互換マシンのシリーズで、小さなものから大きく高性能なものまで存在
 - 個々のトランジスタではなく、集積回路(ICs)を使用
- OS/360がファミリの全てのモデル上で動作
 - 時分割(Timesharing)システムを採用

□ その他の時分割(Timesharing)システム

- MULTICS (MULtiplexed Information Computing Service)
 - 1960年代に開発されたタイムシェアリングOS。後のUNIXに影響を与えた
- Minicomputers:
 - 例) DEC PDP-1 ~ PDP-11(incompatible family)
- UNIX OS and its derivatives
 - System V, BSD, MINIX, 後にLinuxも
 - POSIX (Portable Operating System Interface)を採用

オペレーティングシステムの歴史(6)

□ 第四世代の特徴

- パーソナルコンピュータ用のOS
 - CP/M, DOS, Windows, Macintosh (macOS), FreeBSD, Linux
 - ディスク操作からGUIまで
- ネットワークオペレーティングシステム
 - シングルプロセッサ用OSにあるいくつかを付加したもの
 - ネットワークインターフェースコントローラ
 - コントローラを制御する低レベルソフトウェア
 - 遠隔ログイン, 遠隔ファイルアクセスを実現するプログラム
- 分散オペレーティングシステム
 - ユーザは自分のプログラムがどこで実行されているか, 自分のファイルがどこにあるのかを知らないよ; すなわち, 全てはOSによって自動的かつ効率良く管理されている
 - より複雑なスケジューリングアルゴリズムが必要

オペレーティングシステムの歴史(7)

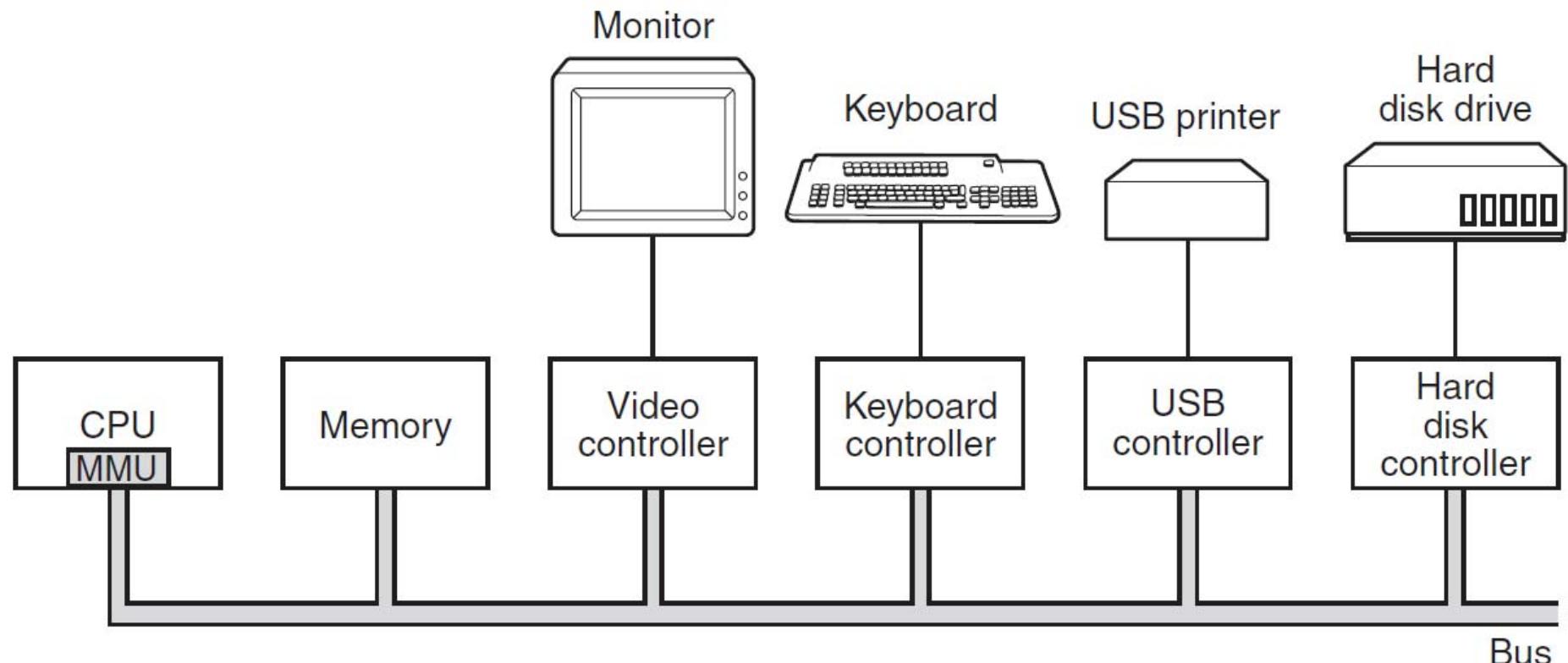
□ 第五世代(1990 – present) – モバイルコンピュータ

- 最初のモバイル電話は1946年, 40kg
- 携帯(handheld)電話は1970年代に登場, 1kg
 - brick(レンガ)と呼ばれていた
- 電話とコンピュータの統合, 1990年代 by Nokia
- スマートフォン用OS
 - Symbian OS
 - 2010年頃まで, スマートフォン用OSとしてはシェア1位
 - BlackBerry OS
 - BlackBerry端末のOS(2013年まで. 現在はAndroidに置き換わった)
 - iOS for iPhone
 - Android by Google (Linux-based)

コンピュータハードウェアの概要(1)

□ 単純なパーソナルコンピュータの構成要素

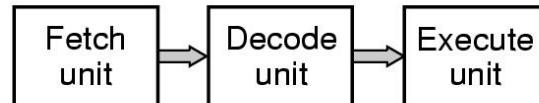
- CPU, メモリ, I/Oデバイスは全てシステムバスによって接続
 - 最近のPCは、階層化されたバスなど、より複雑な構成となっている



コンピュータハードウェアの概要(2)

□ プロセッサ (CPU)

パイプライン



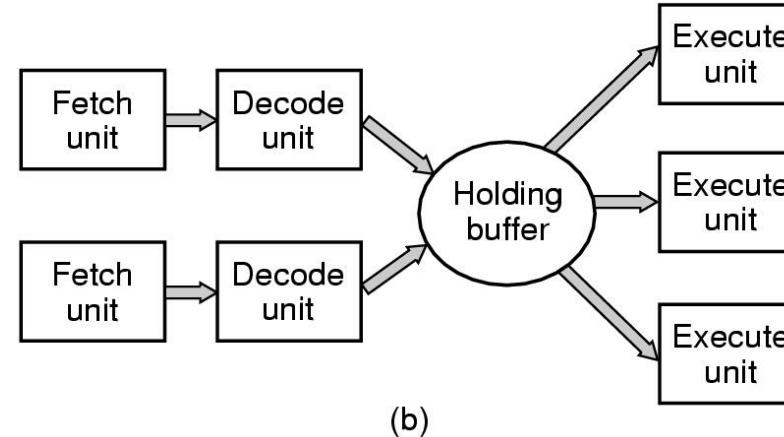
プロセッサコンテキスト (a)

- 汎用レジスタ
- プログラムカウンタ
- スタックポインタ(汎用レジスタの一つを使用することもある)
- プロセッサ状態ワード／レジスタ(PSW/PSR)

□ 条件コードビット, 割込み許可／不許可, 実行モード(ユーザ／カーネル), などの情報を持つ

- ## □ OSはレジスタ集合(=コンテキスト)を管理する必要がある
- 時分割処理において, コンテキストはスワップされる

スーパースカラ／アウトオブオーダ実行



コンピュータハードウェアの概要(3)

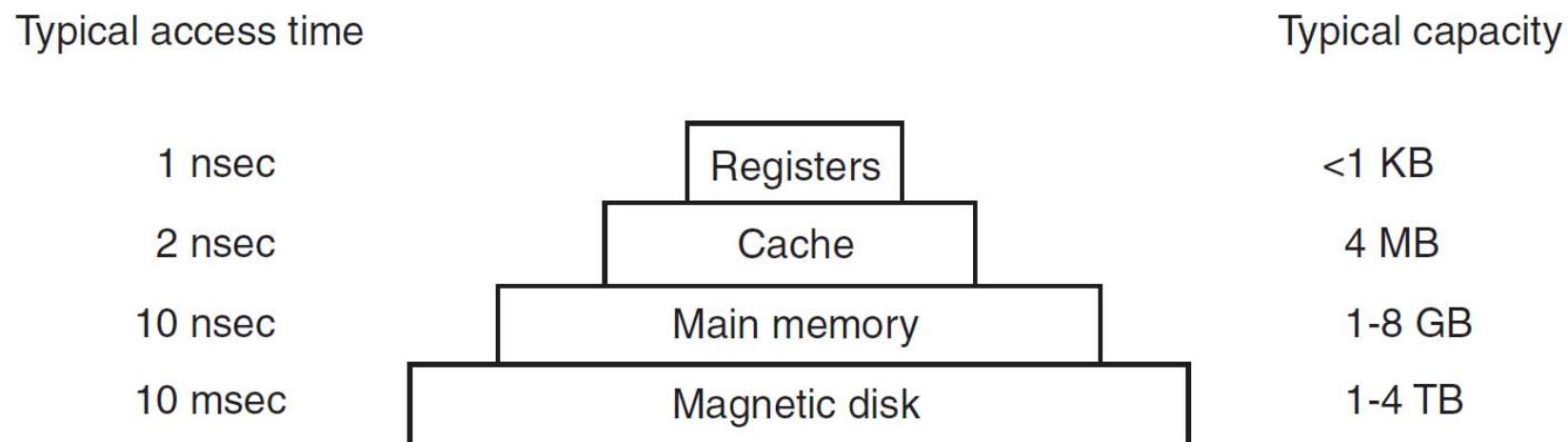
□ プロセッサの実行モード

- カーネル(スーパーバイザ／特権)モードでは、全ての命令およびハードウェアがプログラムから使用可能
 - OSルーチン(デバイスドライバ等)の実行時のモード
- ユーザモードでは、一部の命令およびハードウェアは使用できない
 - ユーザプログラムの実行時のモード
 - 特権命令を実行しようとすると、CPUハードウェアによる例外が発生する
- 例) システムコール
 - OSのサービスを受けるために、ユーザプログラムはシステムコールを実行する必要がある
 - システムコール関数内でトラップ命令(意図的に例外を発生させる命令)を実行することにより、プロセッサはユーザモードからカーネルモードに切り替わり、OSルーチンにジャンプする。

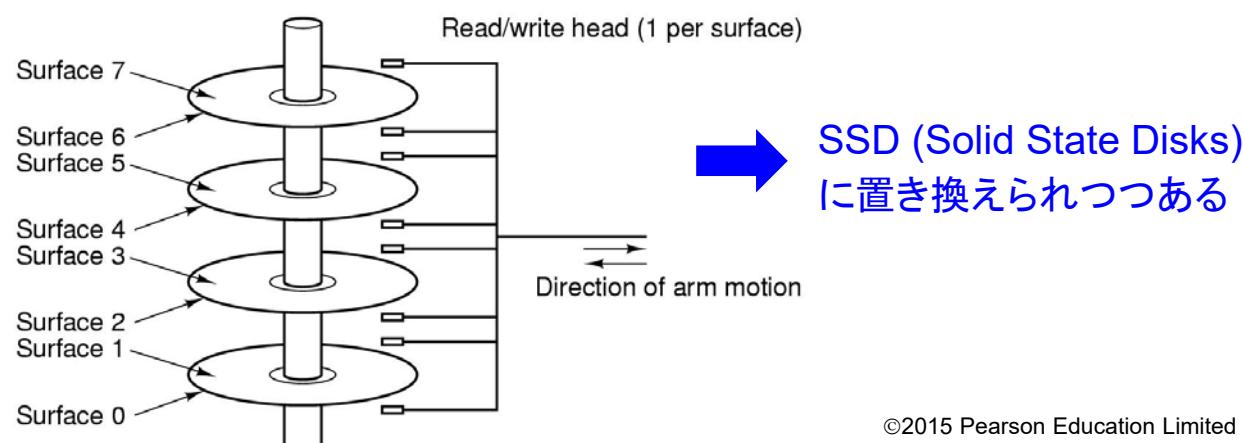
コンピュータハードウェアの概要(4)

□ メモリ

- 高速で大容量のメモリ空間を提供するために、階層化した構成



- ディスクドライブの構造
 - シリンダ、トラック、セクタ



コンピュータハードウェアの概要(5)

□ メモリを共有する際の2つの問題

- プログラム同士、およびOSとユーザ間でどのように保護を行うか
- リロケーション(再配置)をどのように行うか

□ 簡単な(古い)解決の例

- 専用レジスタ

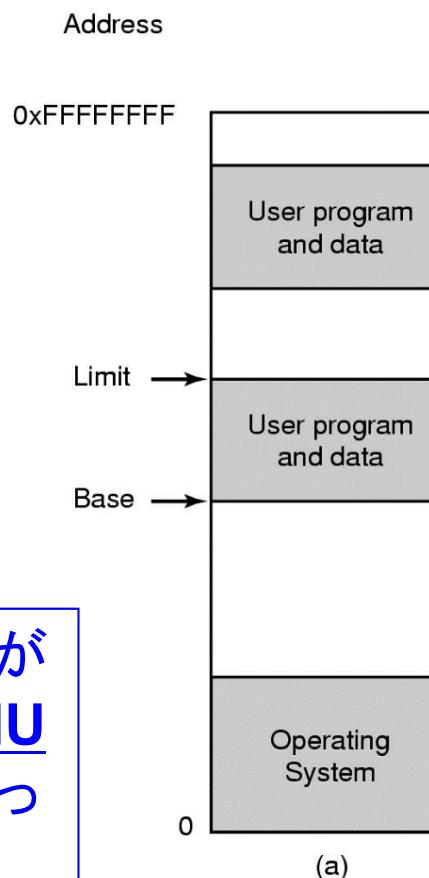
- Base
 - Limit

1つのbase-limitペア (a)

VS.

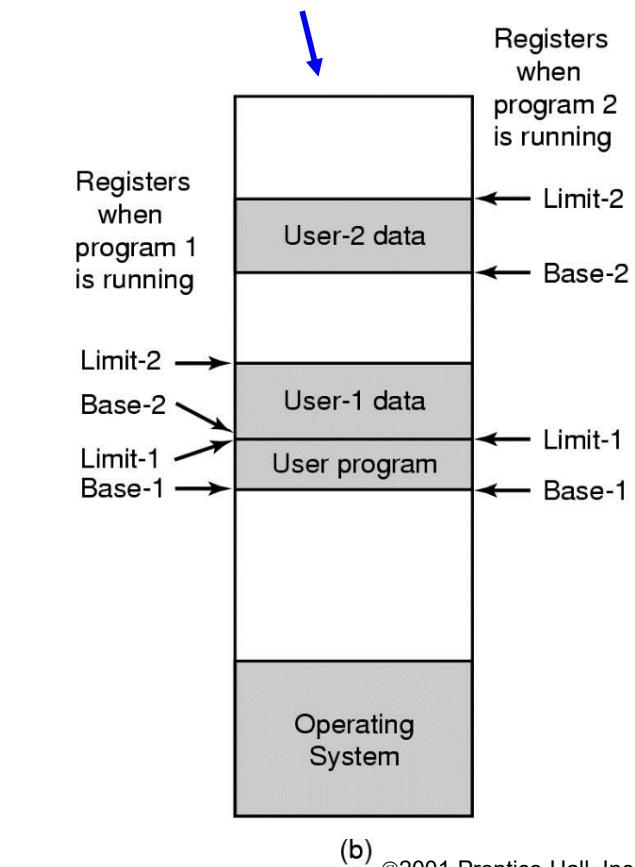
2つのbase-limitペア (b)

実際は、より複雑な”仮想記憶”方式が使用されており、プロセッサ内のMMU(memory management unit)によってアドレス変換が行われる



(a)

プログラム間でプログラム
テキストを共有可能



(b) ©2001 Prentice-Hall, Inc.

コンピュータハードウェアの概要(6)

□ I/Oデバイス

- デバイス自体

- 複雑なインターフェースを持つ

- コントローラ

- OS(デバイスドライバ)に対するより単純なインターフェースを持つ
 - デバイスドライバからのコマンド受付, 結果通知などのために, いくつか専用レジスタを持つ

□ デバイスドライバ

- コントローラ(のレジスタ)にコマンドを送り, 返答を受け取る
- コントローラの製造者は各OS用のドライバを提供する必要がある

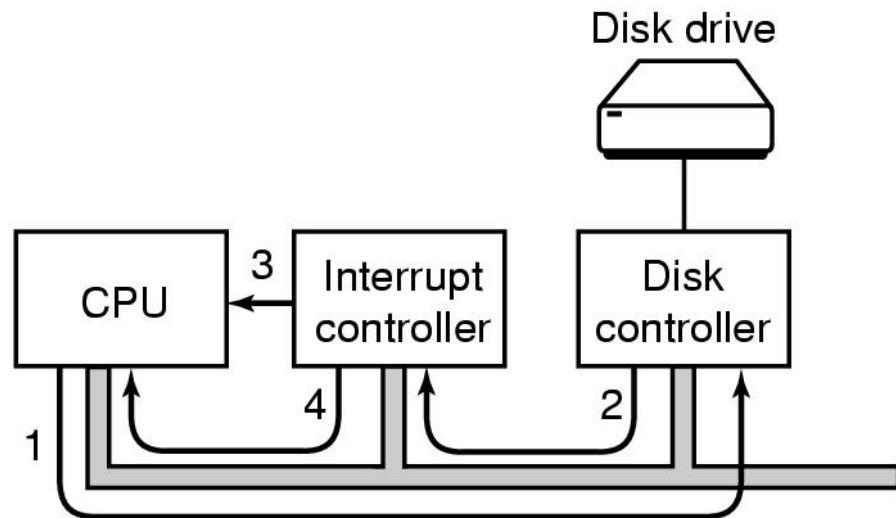
□ I/Oは3つの方法により行われる

- ビジーウェイト(ポーリング)
- 割込み
- DMA(Direct Memory Access)(割込みを併用)

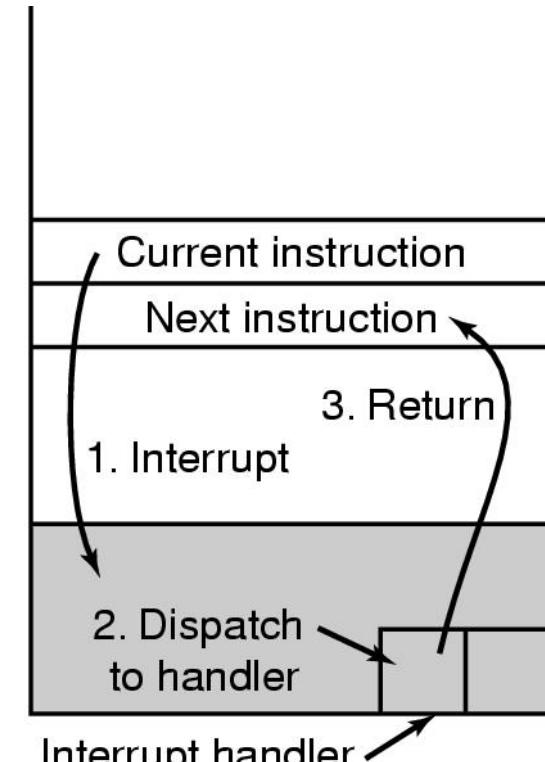
コンピュータハードウェアの概要(7)

□ 割込みの手順

- (a) I/Oを開始し、割込みを受けるまでのステップ
- (b) 割込みを受け、ハンドラを実行し、ユーザプログラムに復帰するまで



(a)

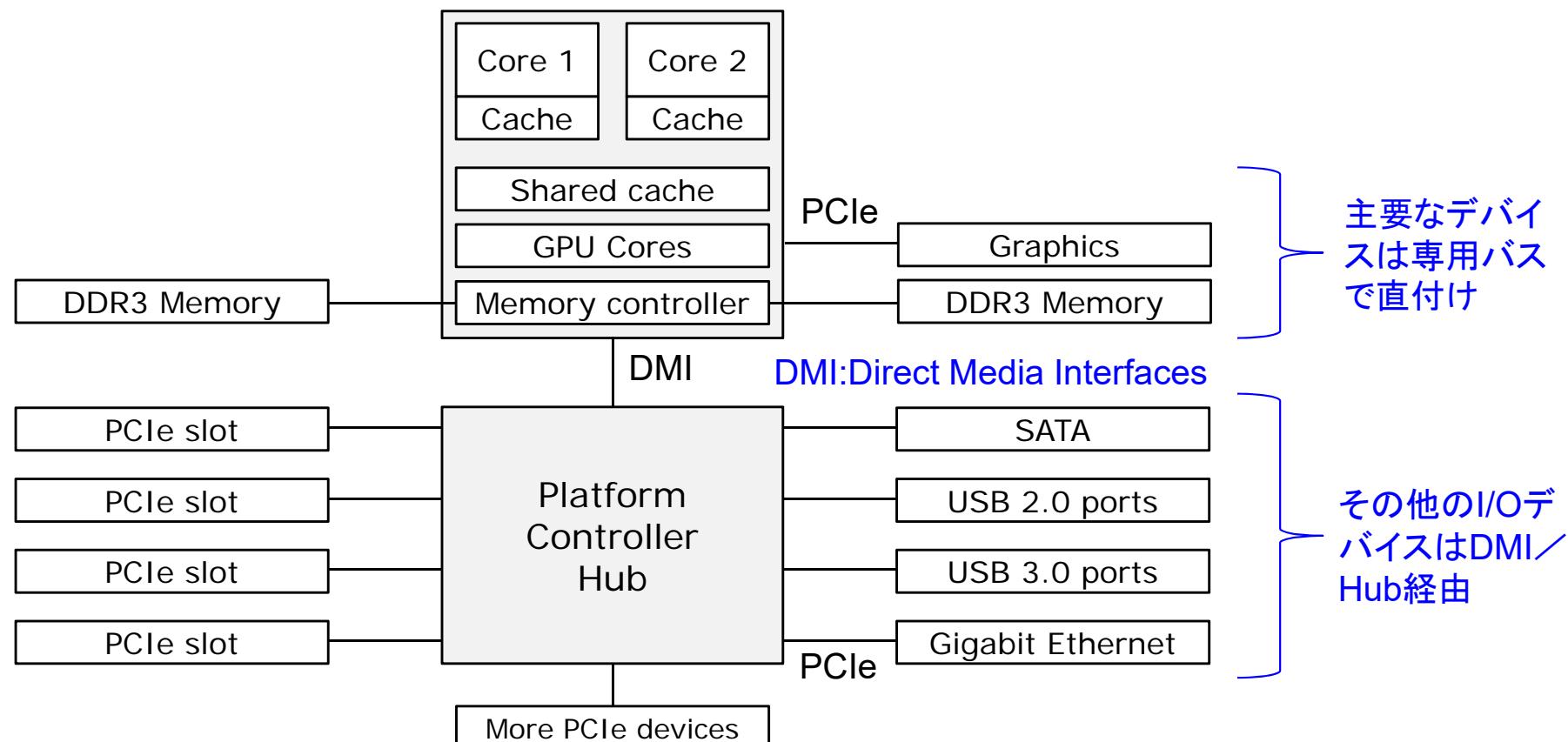


(b)

コンピュータハードウェアの概要(8)

□ バス

- CPUとメモリの高速化により、システムのバス構成が変化してきた



様々なオペレーティングシステムの世界(1)

□ メインフレーム用OS

- 多数のジョブ & I/O要求に対応
- OS/360 → OS/390 → z/OS (by IBM)
- 最近はLinuxなどに置き換えられつつあるが、信頼性の面からなお現役

□ サーバ用OS

- ネットワークを介した複数ユーザからの要求に対応
- UNIX(Solaris, FreeBSD, Linux), Windows Server 201x/2022

□ マルチプロセッサ用OS

- サーバ用OSにプロセッサ間通信、一貫性保証機能を付加したもの
- 最近のPCはマルチコア化が進み、カテゴリ間の区別がなくなってきた

□ パーソナルコンピュータ用OS

- PC(個人用)のOSs
- Windows *, Apple's OS X (macOS), FreeBSD, Linux

様々なオペレーティングシステムの世界(2)

□ 携帯端末(Handheld Computer)用 OS

- PDA, タブレット, スマートフォン
- Google's Android, Apple's iOS, その他

□ 組込み(Embedded)OS

- 組込み機器(電子レンジ, TV, 自動車など)で使用されるOS
- 携帯端末用OSと異なり, ユーザによるアプリケーションの追加は想定していない

□ センサーノードOS

- センサーネットワーク
 - 互いに通信し合う超小型コンピュータ(センサー付)群
(ベースステーションとも通信する)
 - 極端な電力&メモリ資源制約が課せられる
 - 国境監視, 森林火災監視, 気温&降水量観測, 各種軍事目的など
- TinyOS

様々なオペレーティングシステムの世界(3)

□ リアルタイム(Real-time, RT)OS

- パラメータとして“時間”を持つもの
 - ジョブには締切時刻(デッドライン)がある
- VxWorks, eCos, OSEK/AUTOSAR, ITRON, など
- 前出の組込みOS とリアルタイムOSは多くの共通点を持つ(同一視できる)

□ スマートカード用OS

- CPUチップを含む小さなカードに搭載されるOS
- Java Virtual Machine(JVM)が使われることがある

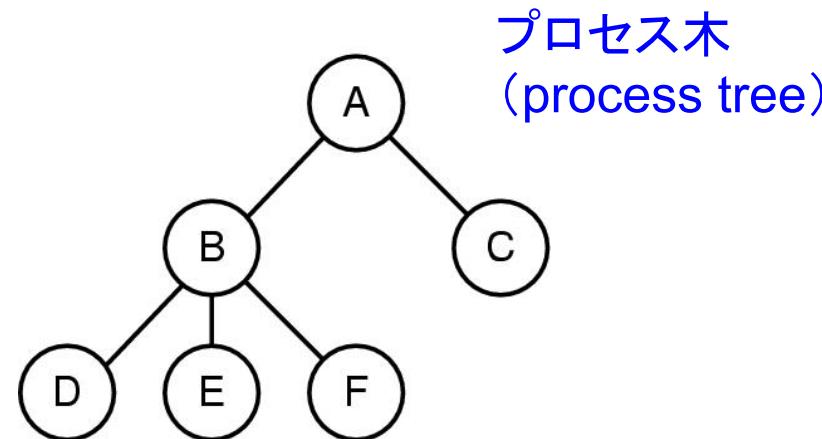
オペレーティングシステムの概念(1)

- 多くのOSが共通の概念／抽象化を持つ
 - 「プロセス」, 「アドレス空間」, 「ファイル」など
- プロセス(process)
 - 実行中のプログラム
 - 独自のアドレス空間(address space)(0番地～)を持つ
 - アドレス空間内にプログラムコード, データ, スタックを持つ
 - 使用リソースが関連付けられる
 - プログラムカウンタ, スタックポインタ, および他のハードウェアレジスタ
 - オープンしたファイル群
- 時分割方式において
 - プロセスが中断される場合は, 後に再開される必要がある. このことは, そのプロセスについての全ての情報をどこかに保存しておく必要があることを意味する.
 - 全ての情報はプロセス表(process table)を利用して保存される

オペレーティングシステムの概念(2)

□ プロセスの生成と終了

- プロセスはシステムコールによって、他のプロセスを生成したり、自分自身を終了させたりする
 - プロセス群は親子関係で表現される



- 関係するプロセス同士はしばしばお互いに通信しあったり、同期をとりあつたりする必要がある
 - プロセス間通信

オペレーティングシステムの概念(3)

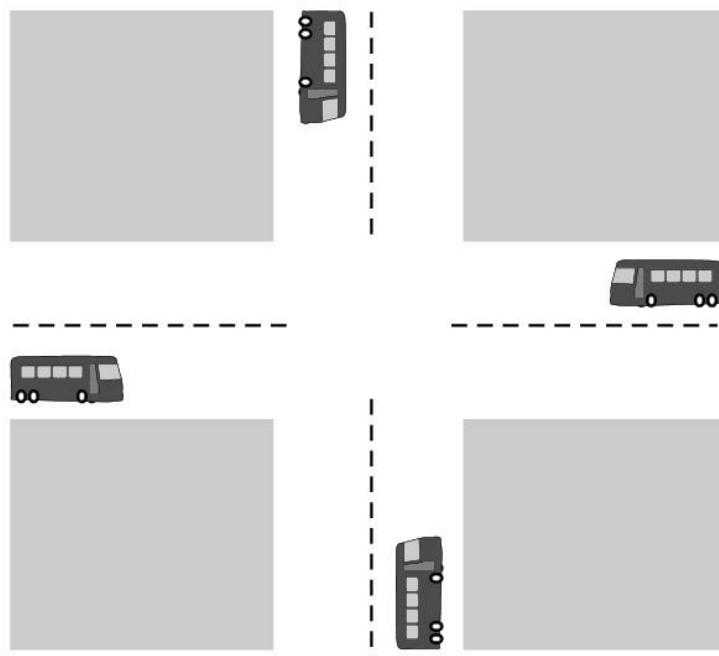
□ アドレス空間とメモリ管理

- マルチプログラミング／時分割管理において、複数のプログラムが同時にメモリ内に存在
- 各プロセスは、物理メモリのアドレスとは異なる「仮想／論理アドレス」を見ている
- 実際のメモリ参照の際は、(仮想記憶([virtual memory](#))の技術により)仮想アドレス → 物理アドレスの変換を行ってから参照
- これにより、異なるプロセスは異なる物理アドレス部分に配置されるため、保護が実現される
- また、各プロセスは、(仮想記憶([virtual memory](#))の技術により)物理メモリよりも大きな仮想アドレス空間を持つことが可能

オペレーティングシステムの概念(4)

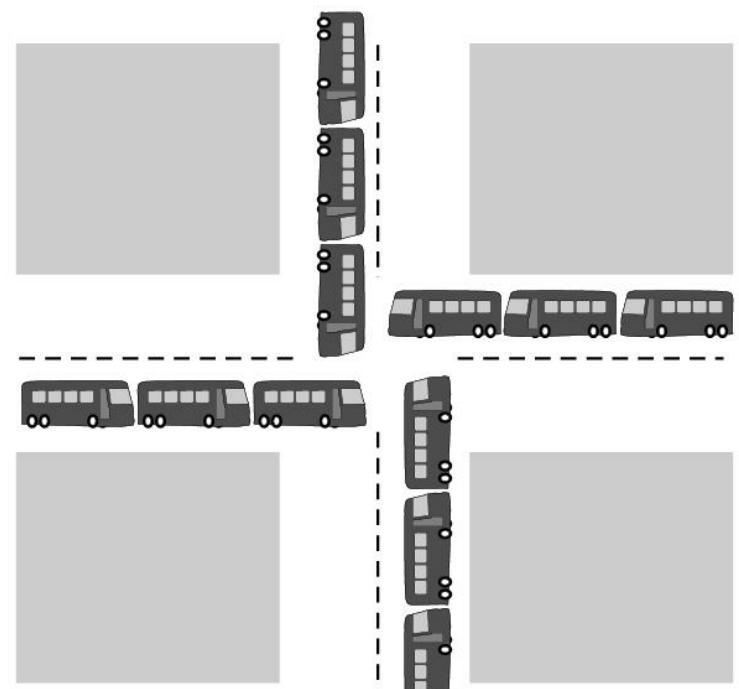
□ デッドロック

- 2つ以上のプロセスが相互作用する場合、行き詰って先に進めない状態に陥ることがある



(a)

デッドロックの可能性



(b)

デッドロックの発生

オペレーティングシステムの概念(5)

□ Input/Output

- OSは様々な入出力デバイスを以下によって管理

- デバイス非依存ソフトウェア

- デバイスの種類に限らず共通した仕事を行う

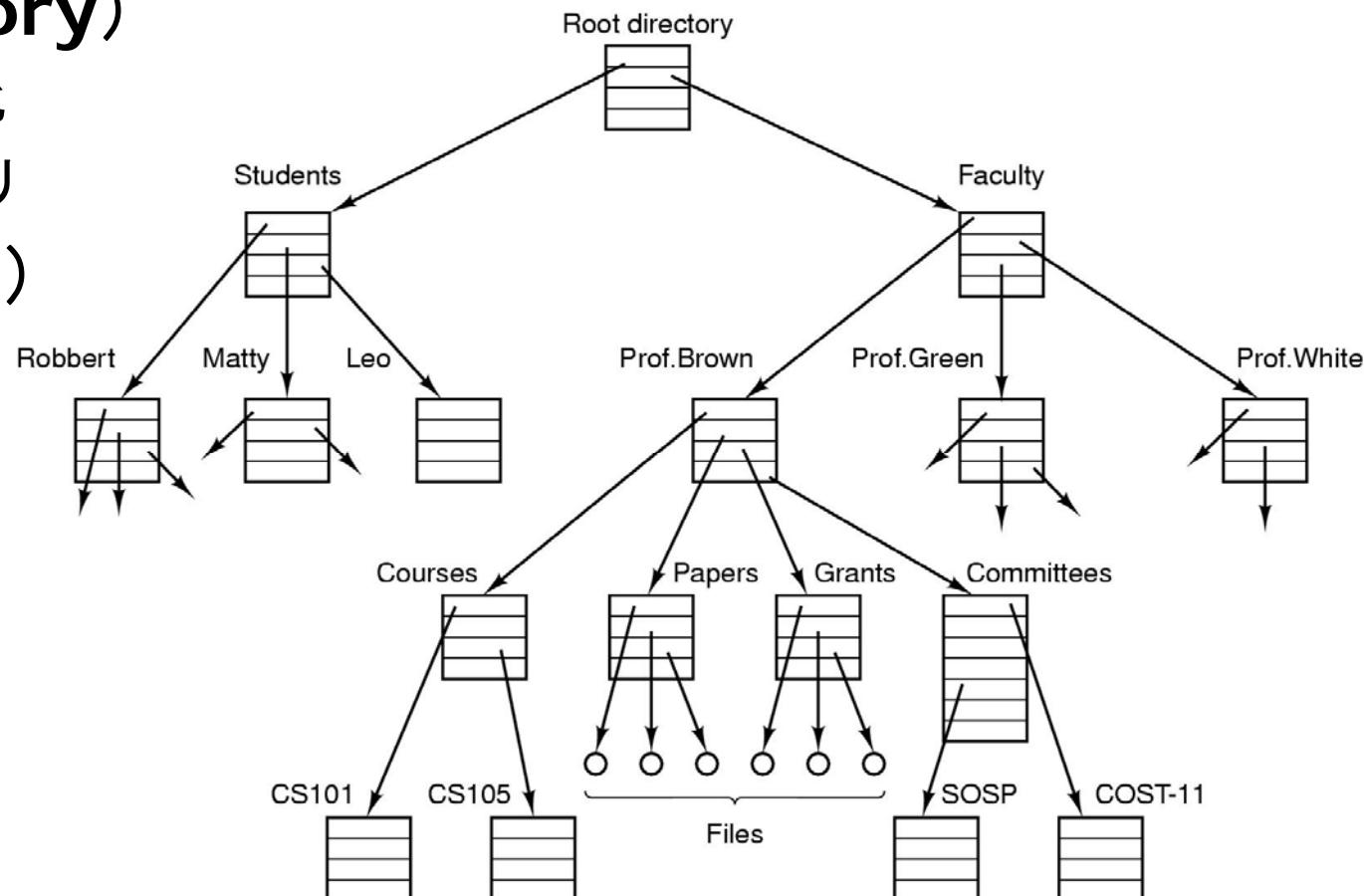
- デバイスドライバ

- デバイス依存であり、特定のI/Oデバイスに特化された仕事を行う

オペレーティングシステムの概念(6)

□ ファイル(ファイルシステム)

- ファイルを生成, オープン, 読み出し／書き込み, 閉じる, 削除するシステムコールが提供されている
- ディレクトリ(directory)
 - ファイルのグループ化
 - ワーキングディレクトリ
- パス(path)(パス名)
 - 階層の中でファイルを特定
- ファイル記述子(file descriptor)
 - ファイル操作で使用

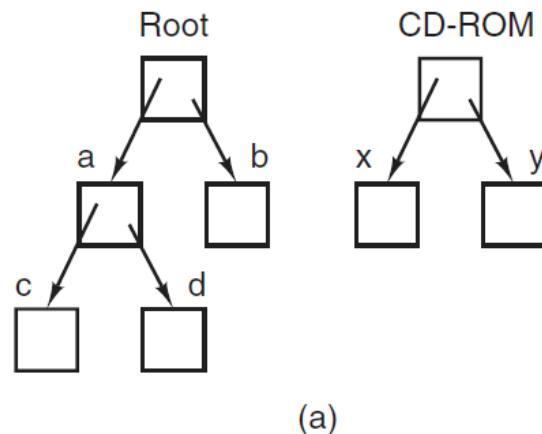


デバイス非依存な「ファイル」という概念で扱えるようにするのが目的

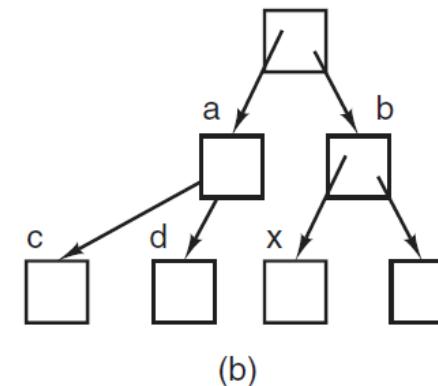
オペレーティングシステムの概念(7)

■ マウント(mount) (UNIXにおけるシステムコール)

- あるファイルシステムをディスク上のルートファイルシステムに結合



(a)



(b)

■ 特殊ファイル(Special file) (in UNIX)

- ブロック型特殊ファイル

- ディスクのような、ランダムアクセス可能な(アドレスを持つ)ブロックの集合からなるデバイスに使用される

- キャラクタ型特殊ファイル

- ラインプリンタやモ뎀のような、(アドレスを持たない)キャラクタストリームを入力、あるいは出力するデバイスに使用される

システムコール(1)

□ システムコール

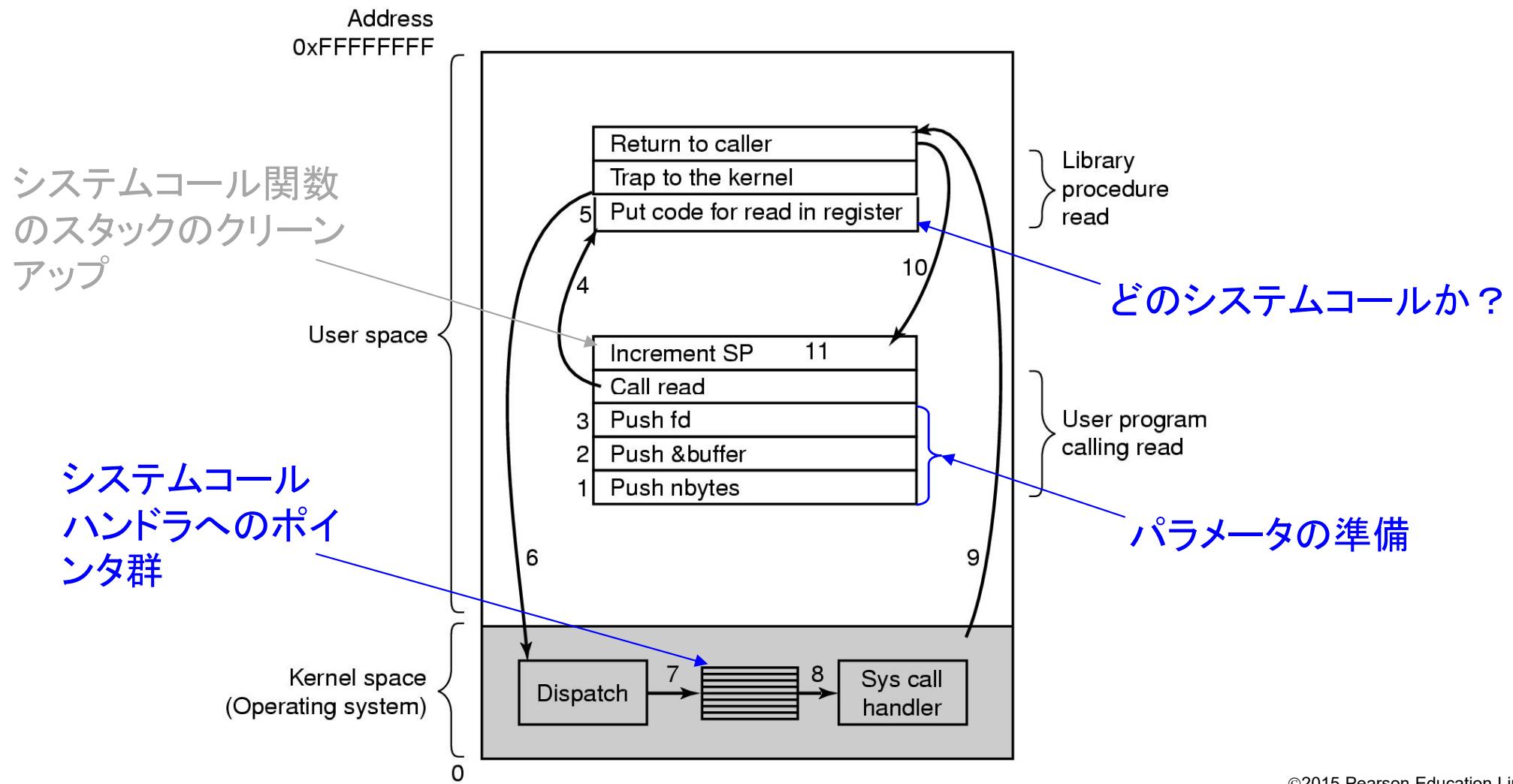
- OSが提供するサービスを利用する手段
- インタフェースが定義されている

□ 制御の流れ

- (1) ユーザモードで実行中のプロセスがシステムのサービスが必要となった場合、システムコール関数を呼び出す。その関数の中で、OSに制御を移すためにトラップ命令(システムコール命令)を実行し、OSのルーチンに移動
- (2) OSは呼び出したプロセスが何を望んでいるのかを、パラメータ(レジスタの値)を調べることによって判断する
- (3) OSは要求された処理(各種サービス)を行う
- (4) その後、システムコール関数内のトラップ命令の次の命令に制御を返す。その後、システムコール関数からユーザプログラムにリターンすることになる

システムコール(2)

- システムコールにおける11のステップ
 - read (fd, buffer, nbytes)



システムコール(3)

□ 主なPOSIXシステムコール

POSIX: Portable Operating System Interface, UNIX

異なる(UNIX系)OSに対して共通のAPIを定めたもの。アプリケーションの移植性の確保が目的。

Process management

Call	Description
pid = fork()	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

システムコール(4)

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

システムコール(5)

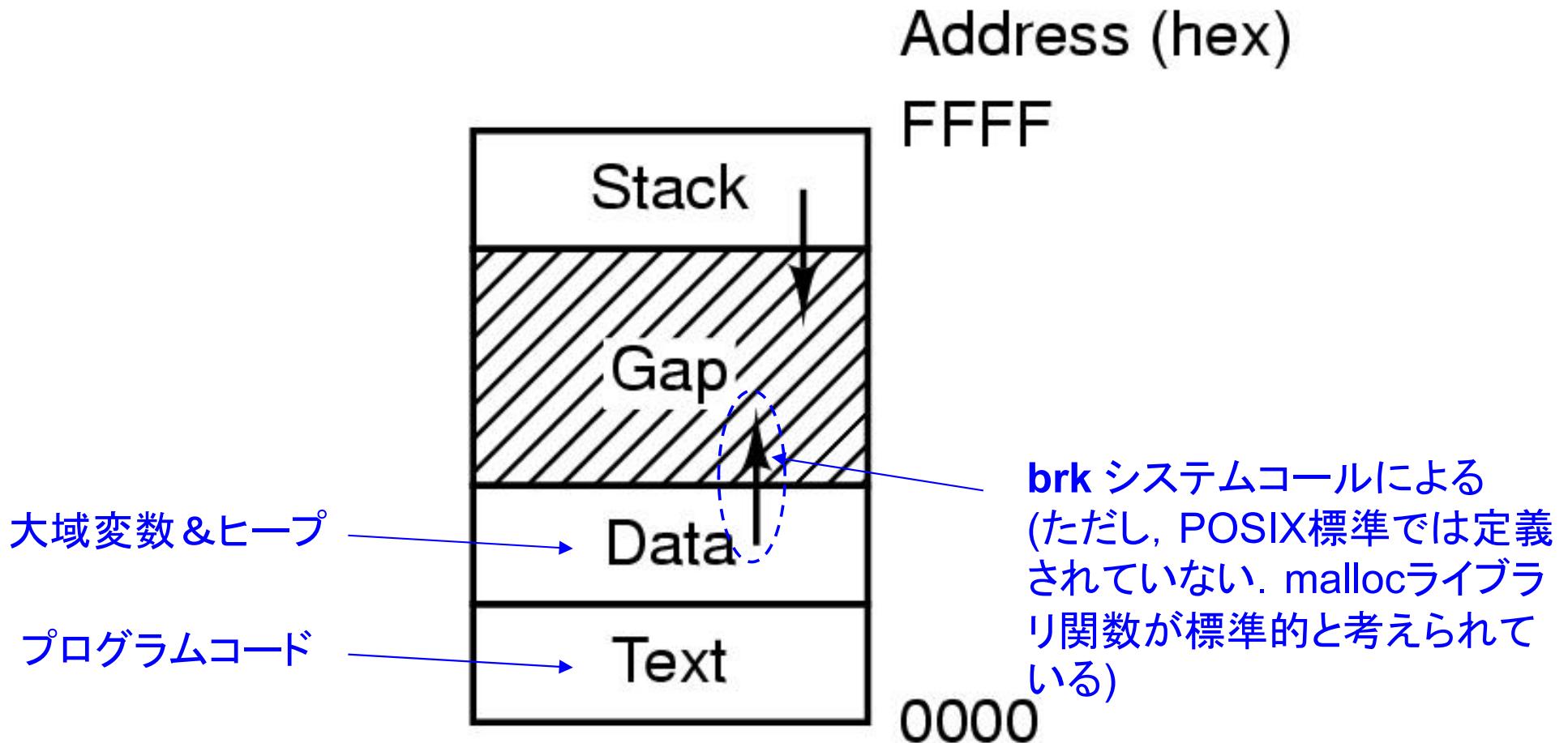
□ プロセス管理のためのシステムコール ■ 簡単なシェルの例

```
while (TRUE) {  
    type_prompt( );  
    read_command (command, parameters)          /* repeat forever */  
                                                /* display prompt */  
                                                /* read input from terminal */  
  
    if (fork() != 0) {  
        /* Parent code */  
        waitpid(-1, &status, 0);                  /* fork off child process */  
    } else {  
        /* Child code */  
        execve (command, parameters, 0);           /* wait for child to exit */  
    }  
}
```

Waiting any child

システムコール(6)

- プロセスは3つのセグメントを持つ: text, data, stack



システムコール(7)

□ ファイル管理のためのシステムコール

- open, close, read, write, lseek, stat
- ファイル内の現在の位置は位置ポインタによって指されており、これはreadやwrite, lseekによって変化する
- テキストファイルの中身を表示する例

```
int    fd, n;
char  buf[BUFSIZ];

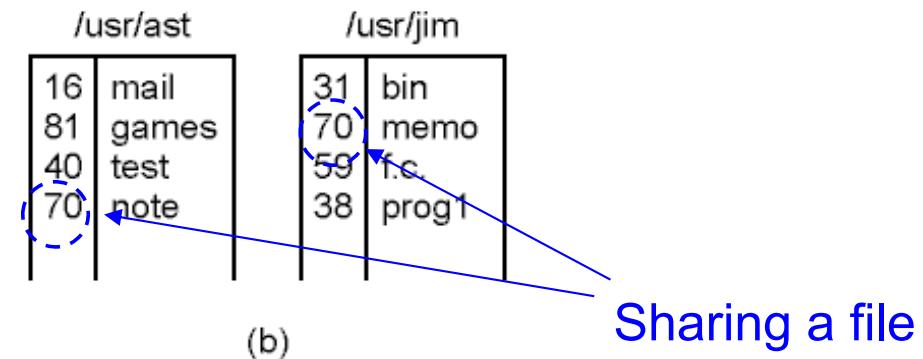
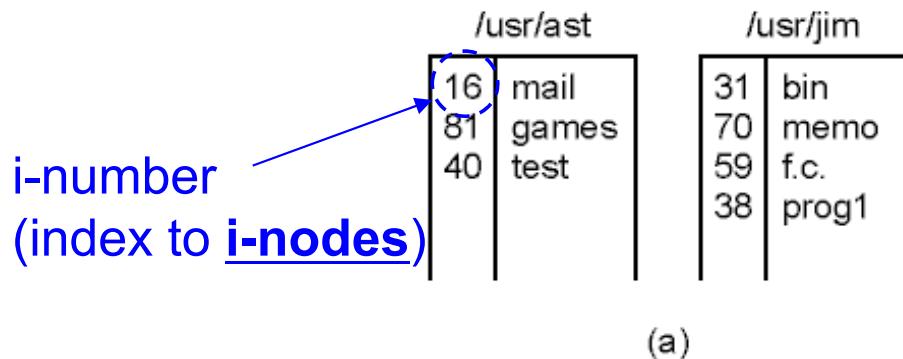
fd = open ( “./foo.txt”, O_RDONLY );           /* ファイルをオープン */
if ( fd == -1 ) { /* エラー発生 */
    perror ( “open failed” );
    exit ( 1 );
}
while ( ( n = read ( fd, buf, 256 ) ) > 0 )      /* ファイルから読み出し、bufへコピー */
    write ( 1, buf, n ); /* 標準出力に書き込む(画面表示) */

close ( fd );
```

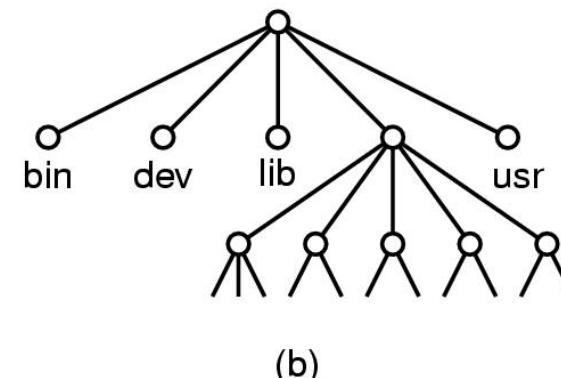
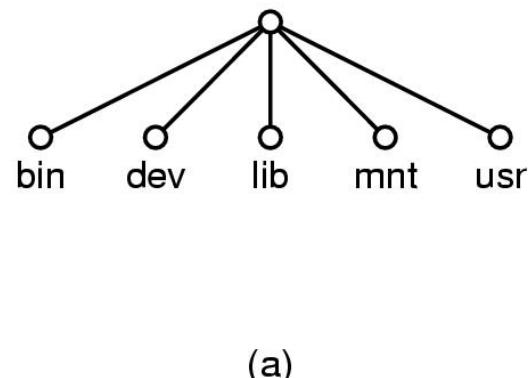
システムコール(8)

□ ディレクトリ管理のためのシステムコール

- mkdir, rmdir: ディレクトリの生成, 削除
- link, unlink: 同一のファイルを異なる名前で共有



- mount, umount



システムコール(9)

□ その他のシステムコール

- chdir, chmod, kill, time
- chdir: 実行プロセスのワーキングディレクトリが変更される
- chmod: ファイルの保護情報(読み／書き／実行権)が変更される
- kill: プロセスにシグナルを送り, シグナルハンドラが実行される
 - シグナルハンドラが登録されていなければ, プロセスを終了させる
- time: 1970年1月1日からの経過時間(秒数)を得る
 - 32ビット UNIXシステムは2106年まで有効(136年間)
 - (関連)2000年問題
 - コンピュータシステムの内部で, 日付に関して西暦の下2桁のみを取り扱い, 上位2桁を省略していたことが原因で発生するとされていた問題

Windows Win32 API(1)

- UNIX系では、基本的にシステムコールと、ライブラリ関数が1対1で対応している。
 - 例)readシステムコールはライブラリ関数read()によって起動
- Windowsでは、システムコールとWin32/64 APIの関数が対応していない。
 - 各Win32/64ライブラリ関数は、適宜必要なシステムコールを呼び出す
 - Windowsでは、バージョンによってシステムコールが大幅に変更される傾向がある。したがって、compatibilityのために、アプリケーションはシステムコールを直接使用せず、Win32/64 APIでプログラミングすることが推奨されている
- 現在の64ビット版Windowsでは、Win32 APIで作成されたアプリケーションはWOW64(Windows 32bit emulation on Windows 64bit)によってエミュレーション実行される

Windows Win32 API(2)

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

プロセスとスレッド



プロセス(1)

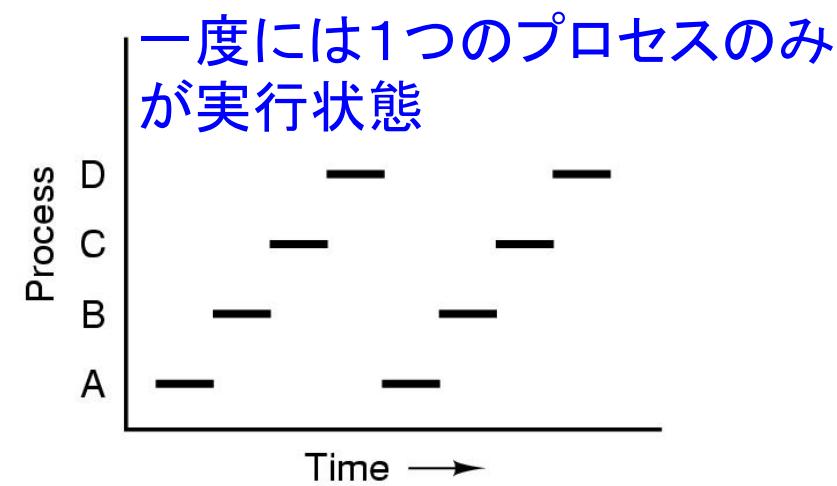
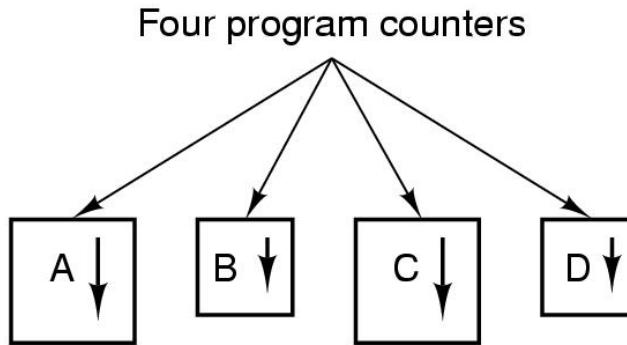
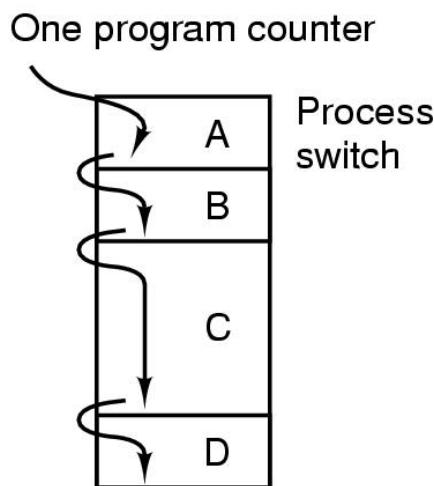
□ プロセスモデル

■ 実行中の(起動された)プログラム

- プログラム自身、入力、出力、状態、コンテキスト(プログラムカウンタ(PC)値、レジスタ値、変数値)からなる

マルチプログラミング

- (a) 物理的にはプログラムカウンタ(PC)は1つ
- (b) 論理的にはPCはプロセス毎に独立



(a)

(b)

(c)

4つの独立な、(逐次)プロセス

プロセス(2)

- 汎用システムでは、稼働中にプロセスを生成し、終了させる
 - 一方、多くの組込みシステムでは、全てのプロセス(タスク)を常に存在させる
- プロセス生成
 - プロセス生成を引き起こす4つの主要ケース
 - 1) システムの初期化
 - 2) 実行中プロセスによるプロセス生成のシステムコールの実行
 - 3) ユーザ(コマンド入力、またはダブルクリック)による新たなプロセス生成の要求(システムコールの実行)
 - 4) バッチジョブの開始時(同じく、システムコールの実行)
 - 新しいプロセスは、プロセスによるシステムコールの実行によって生成される
 - fork (UNIX), CreateProcess (Windows)
(fork+execve ≈ CreateProcess)
 - UNIXは、子プロセスによるプログラムスタート前に入出力のリダイレクトを操作可能とするために、プロセス生成(fork)とプログラム開始(execve)を分離している

プロセス(3)

□ プロセス生成(つづき)

- プロセスの生成後、親プロセスと子プロセスは異なるアドレス空間を持つ
 - fork実行時、子プロセスのアドレス空間内は、親プロセスのデータのコピーでスタート
 - 実際は、コピー処理のオーバヘッドを避けるために、子プロセスのページテーブルから親プロセスのページをREAD-ONLYでマップし、書き込み要求があった場合にページコピーを生成してマップを変更する(copy on write)
 - 一方、生成前にオープンしたファイルは両者で共有する(両者ともオープンした状態のまま)
 - exec実行時、古いアドレス空間は破棄され、新しいアドレス空間が与えられる(ページテーブルを一新する)

```
pid = fork( );
if (pid < 0) {
    handle_error( );
} else if (pid > 0) {
    /* parent code goes here. */
} else {
    /* child code goes here. */
}
```

[親プロセスは子プロセスのプロセスIDを取得
子プロセスはゼロを取得]

/* if the fork succeeds, pid > 0 in the parent */

/* fork failed (e.g., memory or some table is full) */

/* child code goes here. */

プロセス(4)

□ プロセスの終了

- プロセス終了を引き起こす4つの条件
 - 1) 正常終了(自発的)
 - 2) エラーで終了(自発的)
 - 3) 致命的エラー(非自発的, OSにより強制終了)
 - 4) 他プロセスによる強制終了(非自発的, シグナルの利用による)
- 正常あるいはエラーで終了する場合は以下のシステムコールによる
 - exit (UNIX), ExitProcess (Windows)
- 他のプロセスを強制終了させるシステムコール
 - kill (UNIX), TerminateProcess (Windows)
 - (ただし, kill()はプロセスにシグナルを送ることを目的としており, シグナルハンドラを登録することにより, 強制終了以外の実行が可能)

プロセス(5)

□ プロセスの階層

- プロセスが他のプロセスを生成する
 - 親子の関係
 - 子が更にプロセスを生成可能
 - UNIXでは、親プロセスが終了すると“init”プロセスが子プロセスの親となる
- 階層構造を形成
 - UNIXでは、“プロセスグループ”と呼ばれ、ルートのプロセスがグループリーダとなる
 - 基本的には“init”プロセスがプロセスの階層木のルートとなる
 - システムコール “setpgid()” によって変更可能
 - Windowsでは、プロセス階層の概念は無い
 - 全てのプロセスが平等

プロセス(6)

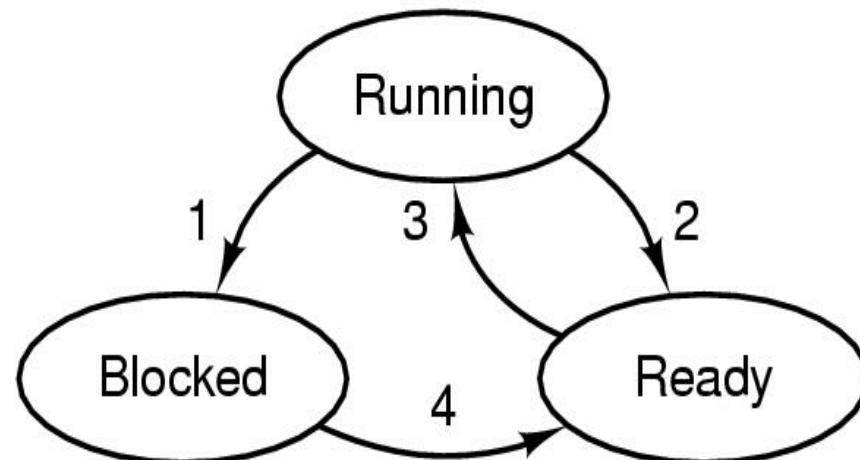
□ プロセスの状態

- 実行中状態(running)
 - 実際にCPUを使用している状態

- 実行可能状態(ready)
 - 実行可能な状態。生成された直後の状態、あるいは他のプロセスが実行されているために一時的に中断した状態

- 待ち状態(blocked)
 - ある外部事象が発生するまで実行不可能な状態(例えば入力待ちなど)

状態遷移



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

プロセス(7)

□ プロセスの情報を管理するデータ構造

■ プロセステーブル中のエントリ=プロセス制御ブロック(PCB)

- プロセスが切り替わるときには、実行していたプロセスについての様々な情報を格納する。これによって、後で再開可能となる

PCBのフィールド例

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

Linuxにおいては “task_struct”

プロセス(8)

- 割込みが発生したときに、OSの最下位レベルで行われる処理の概要

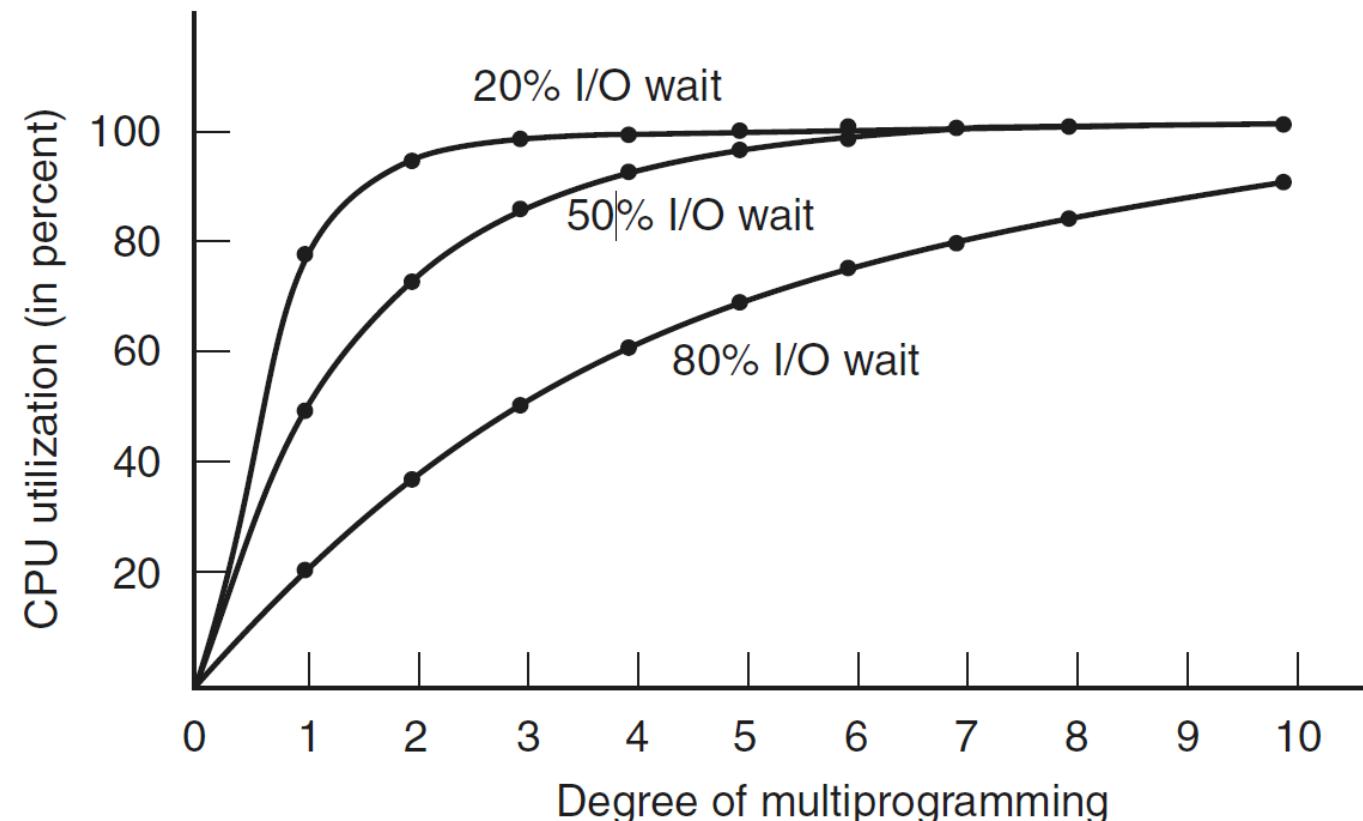
1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

プロセス(9)

□ マルチプログラミングとCPU使用率の関係

p : プロセスあたりのI/O待ち時間の割合 : p
 n : マルチプログラミングのプロセス数

$$\text{CPU利用率} = 1 - p^n$$

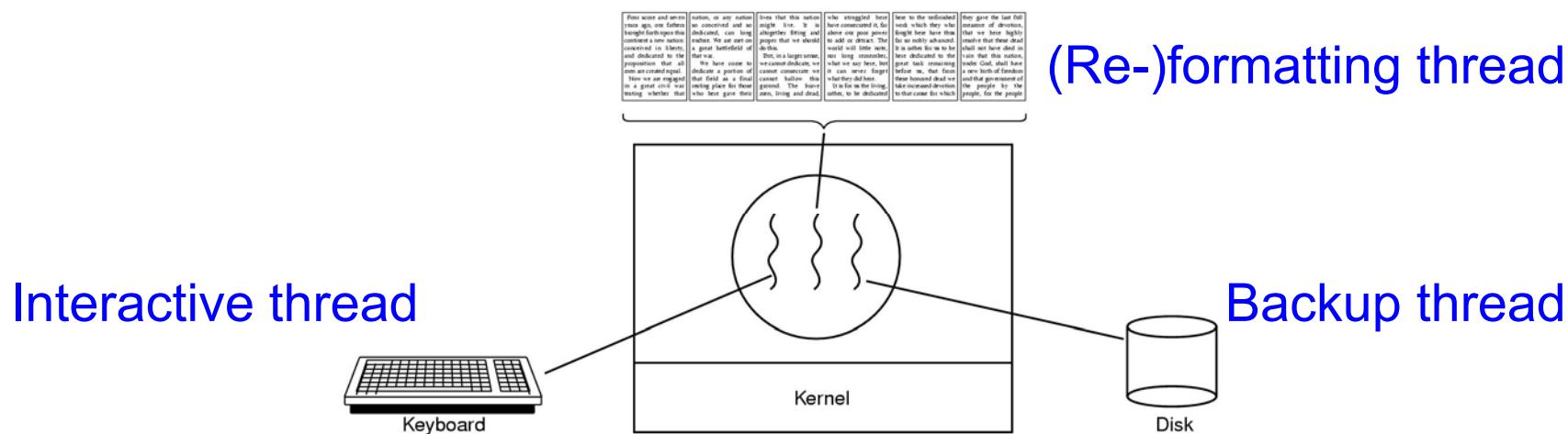


スレッド(1)

□ スレッドの利用

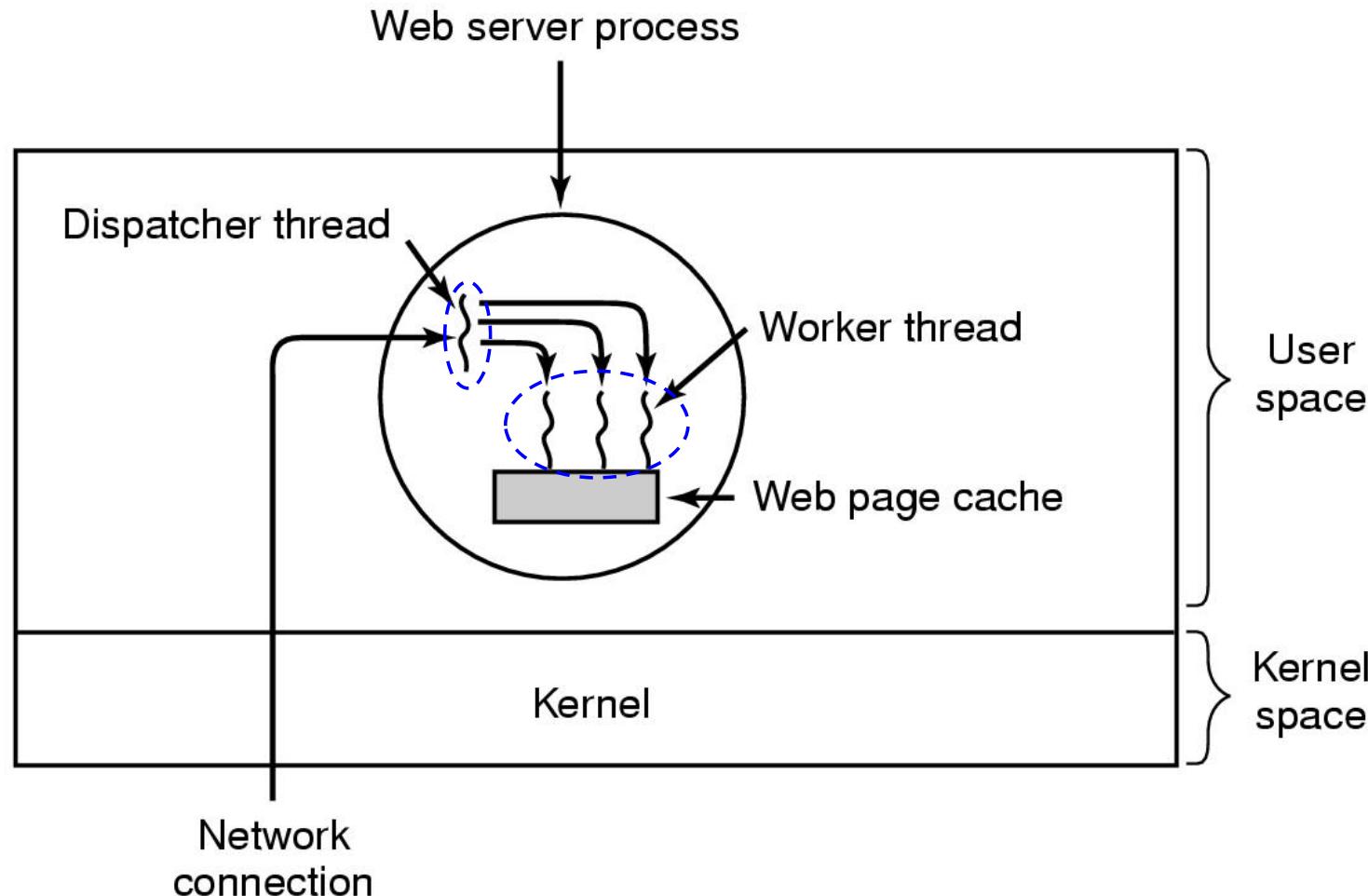
- 多くのアプリケーションにおいて、複数の仕事を同時にに行いたいことがある
 - 各仕事は独立しており、それぞれオンデマンド／イベントドリブンで仕事をする
 - 次のイベントを待つため、各仕事は頻繁に待ち状態になる傾向がある
 - 各仕事は、データ／資源を共有する → アドレス空間を共有したい
- アプリケーションを複数のスレッドに分割する(すなわち、各仕事をスレッドに割り当てる)ことにより、プログラミングモデルはより単純になり、実行も効率的になる
 - スレッド生成はプロセス生成と比較し、ずっと高速に行える(10~100倍の速度)
 - 並列マシン／マルチコアを使うと、マルチスレッドはずっと高速になる
- ワープロの例

A file is shared by all threads



スレッド(2)

■ WEBサーバの例

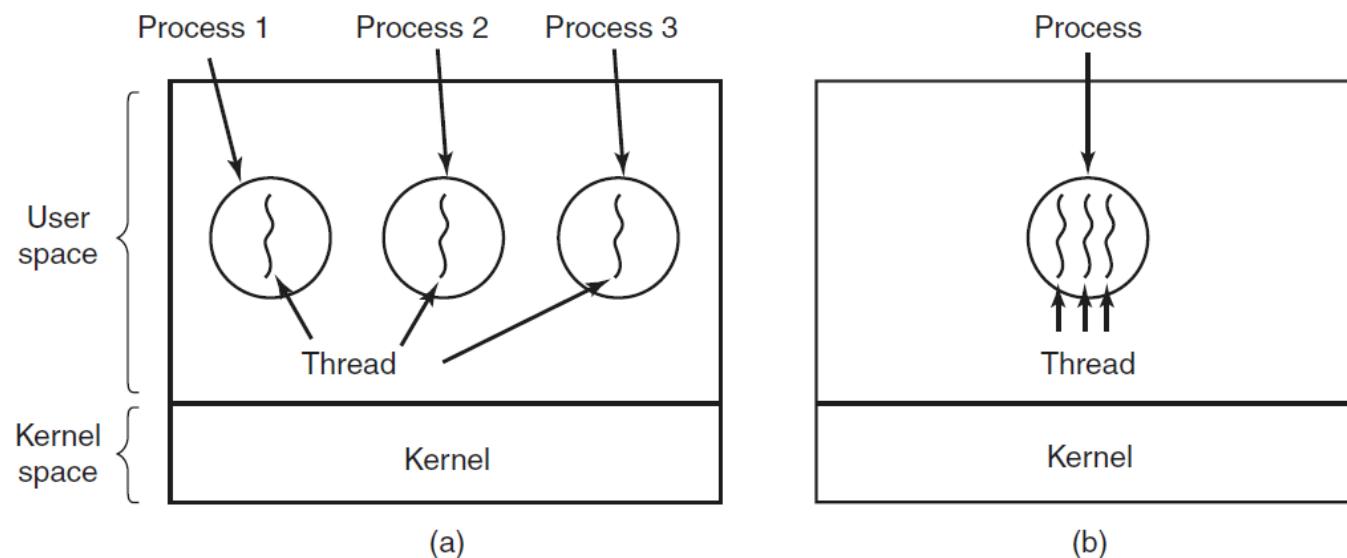


ネットワークからリクエストが来るたびに、workerスレッドに担当させる。
Cacheでページが見つかればすぐに返答。
Cacheに無ければディスクアクセスを依頼し、そのスレッドはブロックする。

スレッド(3)

□ スレッドモデル

- 固有のPC, レジスタ値, スタック, 状態を持つ
- 1つのプロセス内の複数のスレッドはアドレス空間とその他の資源(オープンしたファイルなど)を共有



- (a) Three processes each with one thread
(b) One process with three threads

スレッド(4)

Per process items

Address space
Global variables
Open files
Child processes
Pending alarms
Signals and signal handlers
Accounting information

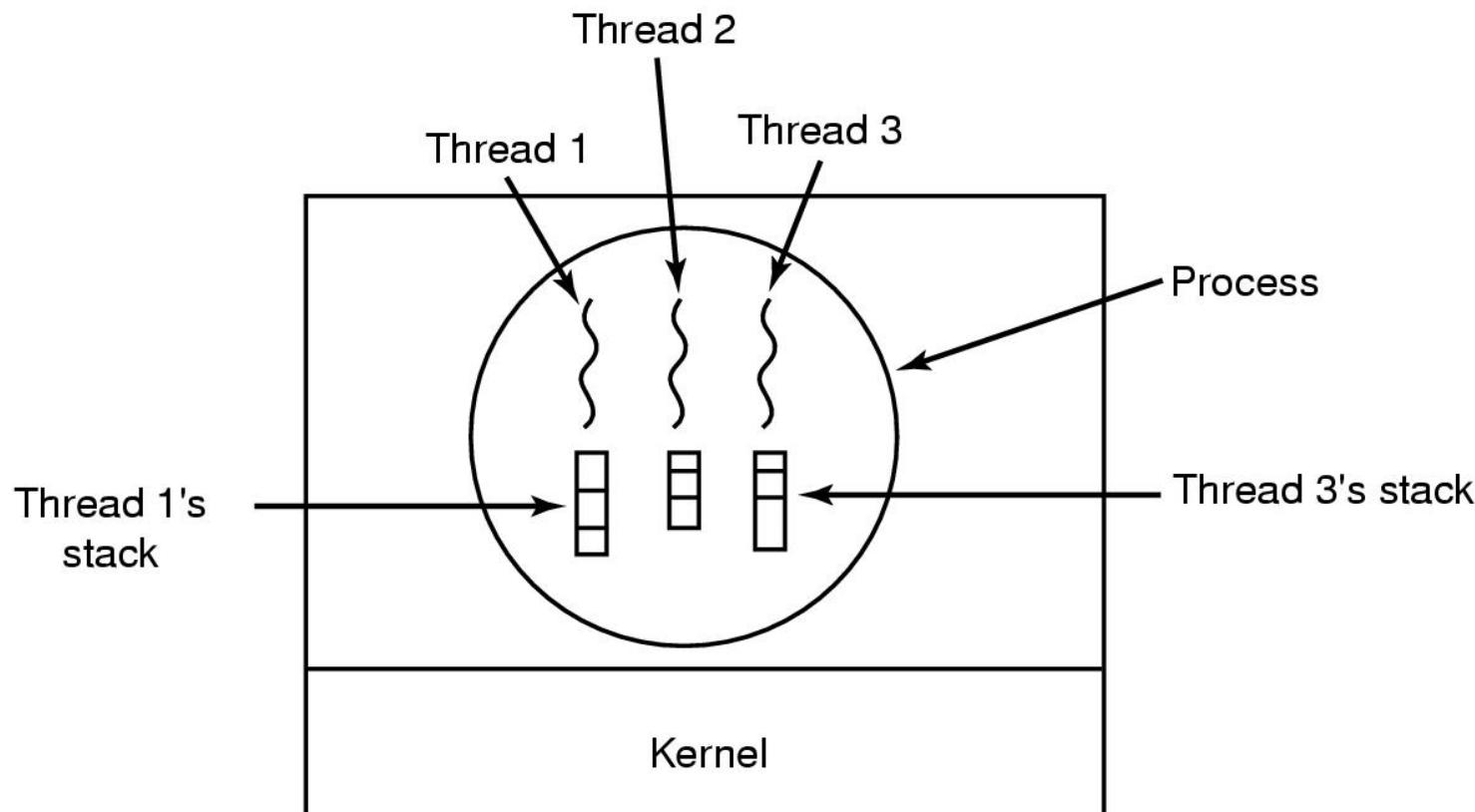
Per thread items

Program counter
Registers
Stack
State {
 Running
 Ready
 Blocked

- 左欄: プロセス内の全てのスレッドで共有されるもの
- 右欄: 各スレッド毎に持つもの

スレッド(5)

- 各スレッドは独自のプログラムカウンタとスタックを持つ
 - それぞれのスレッドは独自の手続き／関数コールの履歴を持つため
 - スタックには局所変数、戻りアドレスなどが格納されている



スレッド(6)

□ マルチスレッド

- 同一プロセス内で複数のスレッドを実行する状況
- プロセスは一つのスレッドで開始し、新しいスレッドを生成(create)しながらマルチスレッドとなっていく

- スレッドを制御する手続き例
 - `thread_create`: パラメータとして生成されたスレッドが実行する手続き名を指定
 - `thread_exit`: スレッドを終了させる
 - `thread_join(wait)`: パラメータで指定したスレッドの終了を待つ
 - `thread_yield`: 自発的にCPUを放棄し他スレッドに切り替える

スレッド(7)

□ POSIXスレッド

- “Pthreads” defined as IEEE standard 1003.1c
- 60以上の関数から成る

pthreadsの主な関数

Thread call	Description
pthread_create	Create a new thread
pthread_exit	Terminate the calling thread
pthread_join	Wait for a specific thread to exit
pthread_yield	Release the CPU to let another thread run
pthread_attr_init	Create and initialize a thread's attribute structure
pthread_attr_destroy	Remove a thread's attribute structure

スレッド毎に属性(スタックサイズ, スケジューリングパラメータ, 優先度など)を持つことが可能

スレッド(8)

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

#define NUMBER_OF_THREADS 10

void *print_hello_world(void *tid)
{
    /* This function prints the thread's identifier and then exits. */
    printf("Hello World. Greetings from thread %d\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    /* The main program creates 10 threads and then exits. */
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

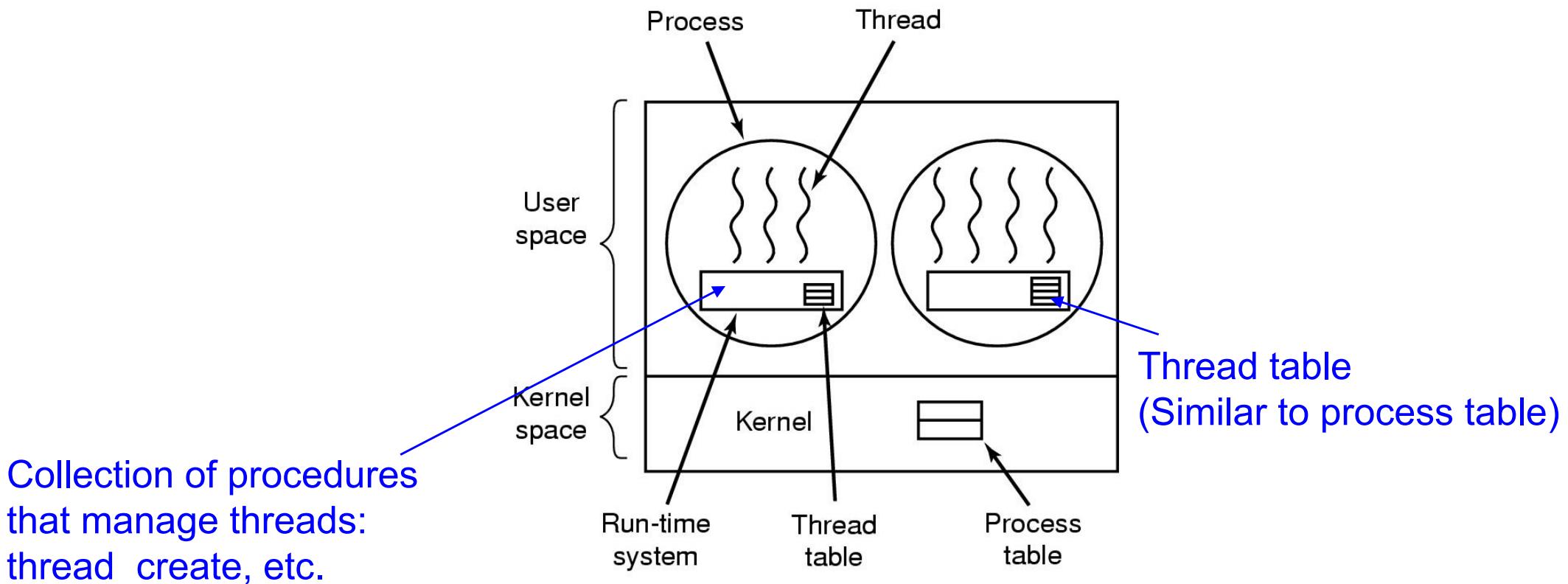
    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Main here. Creating thread %d\n", i);
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);
                                            デフォルト属性
        if (status != 0) {
            printf("Oops. pthread_create returned error code %d\n", status);
            exit(-1);
        }
    }
    exit(NULL);
}
```

pthread_t: スレッド識別子
(Linuxでは整数型)

スレッド(9)

□ ユーザ空間でのスレッドの実現

- スレッドのパッケージを完全にユーザ空間に置く
 - カーネル空間で実現する(トラップを伴うことになる)よりも、ずっと高速になる
 - スケジューリングを目的にあわせてカスタマイズすることが可能
- 各プロセスには自分自身のスレッドテーブルが必要となる



スレッド(10)

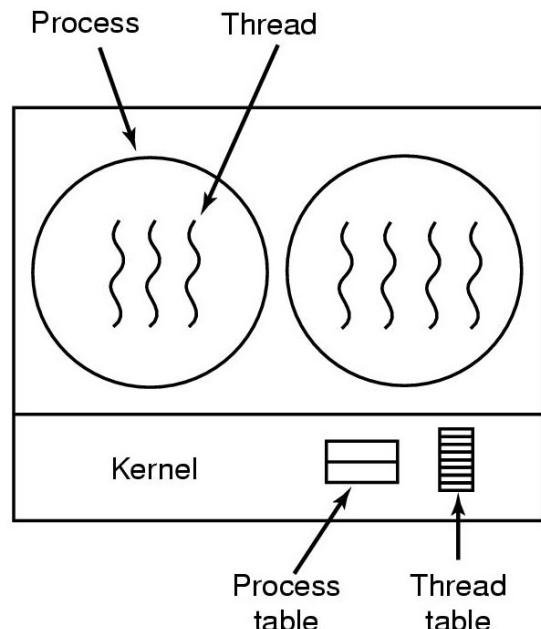
□ ユーザ空間での実現における欠点

- スレッドがシステムコールを実行し、その途中でブロックされたとき、プロセス全体がブロックされる
 - I/O待ちのブロックなど
 - 同プロセス内の他のスレッドに切り替えて実行することが不可能
 - ラッパ(Wrapper), またはジャケット(jacket)によるブロッキング問題の解決
 - あるUNIXシステムにおける“select”: 将来の“read”がブロックされるかどうかチェックできるnon-blockingシステムコール
 - “select”がその“read”がブロックされるであろうと伝えた場合は、ライブラリはスレッド切り替えを決断する。それ以外の場合は“read”を実行する
 - 上記の方法を使用しても、ページフォルトによるブロッキングは回避不可能
- スレッド実行がスレッドのランタイムシステムに突入しない限り、スケジューリングは実行されない
 - ラウンドロビン方式のスケジューリングは実現不可能

スレッド(11)

□ カーネル内のスレッドの実現

- カーネルがスレッドテーブルを持ち、システム内の全てのスレッドを管理する（カーネルが各スレッドの存在を認識している）
- スレッド管理の要求はカーネル呼び出し（システムコール）によってなされる
- プロセス全体のブロッキング問題は生じないが、スレッド管理のオーバヘッドが大きくなる（低速である）



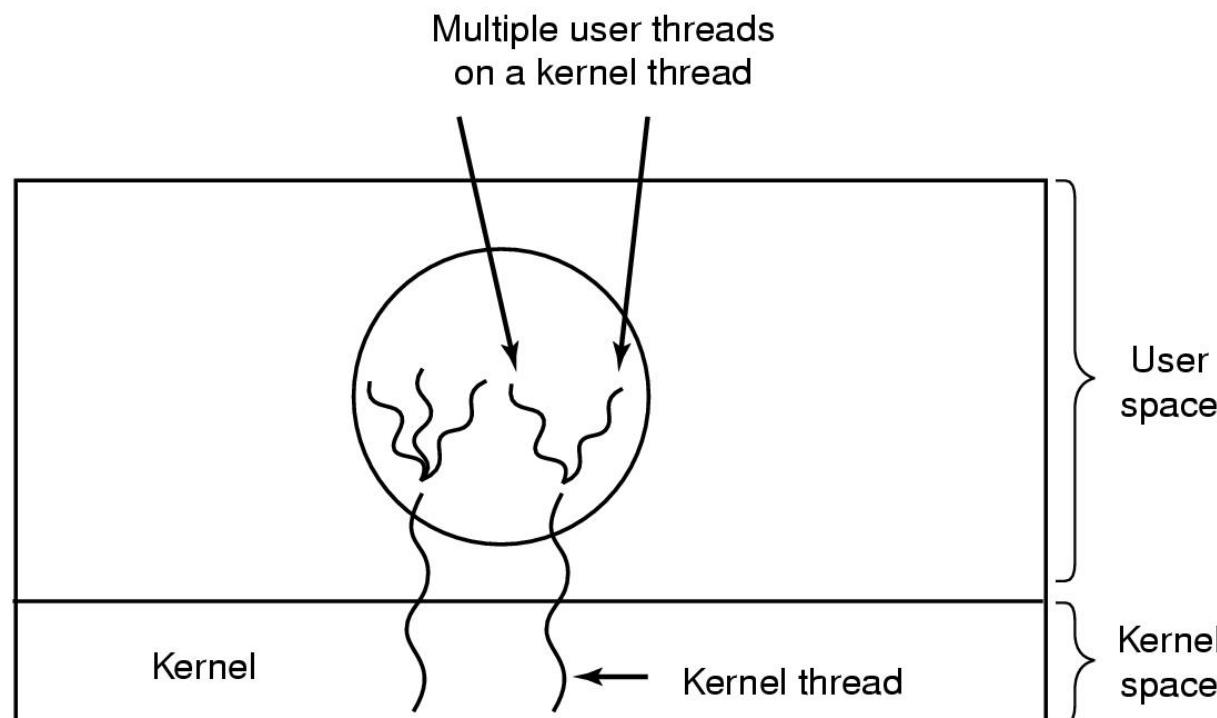
- × Heavyweight switching
- × Cannot use optimized scheduling algorithm dedicated to the process
- ◎ No blocking problem
 - (Can perform switching when a thread is blocked on, for example, a page fault)

スレッド(12)

□ ハイブリッドな実現

- ユーザレベルスレッドとカーネルレベルスレッドの利点を組み合わせる

- カーネルレベルスレッド内で、複数のユーザレベルスレッドを小さいオーバヘッドで生成、消滅させることを可能とする
- ブロッキング時には他のカーネルレベルスレッドに切り替える



シグナル(1)

□ プロセス／スレッドにイベントを通知する機能

- 通常は、(暴走した)プロセスにシグナルを送信して強制終了するために使用
 - [Ctrl + C], など
 - kill(pid, sig), sigsend(idtype, id, sig), pthread_kill(tid, sig) などにより明示的にプロセス/スレッドにシグナルを送信可能
- または、ユーザ定義のシグナルハンドラを登録することで、シグナル受信時に特定の処理を行うことが可能

```
struct sigaction act;  
memset ( &act, 0, sizeof act );
```

```
act.sa_handler = my_handler;  
sigaction ( SIGINT, &act, NULL ); /* ハンドラの登録 */  
/* [Ctrl+C]発生時にmy_handler()が実行される */
```

シグナル(2)

□ POSIXにおけるシグナル

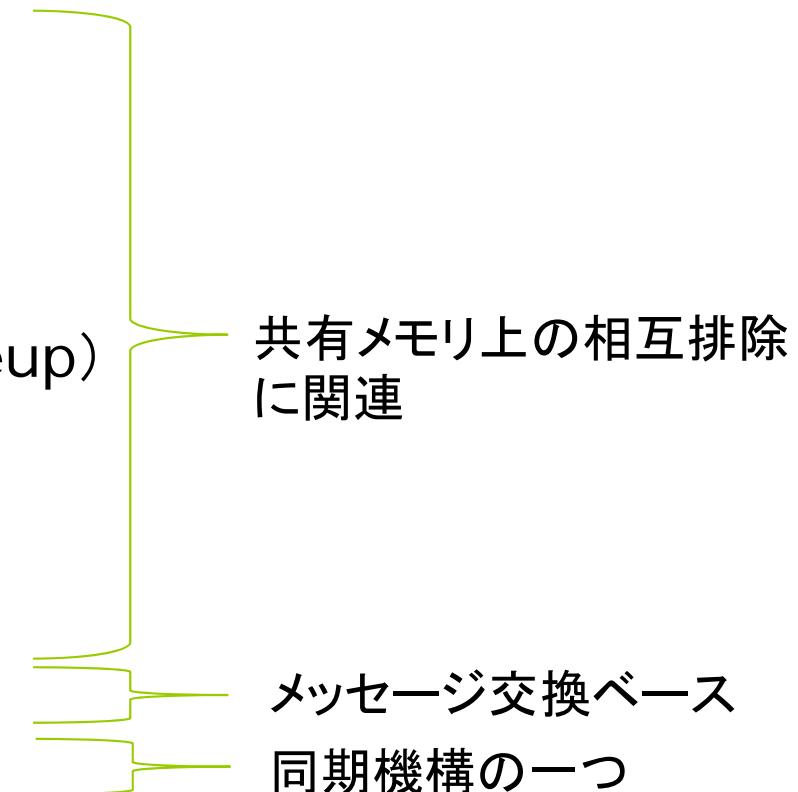
Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

プロセス間通信(1)

□ プロセス間通信(Interprocess Communication, IPC)

- プロセスはしばしば、互いに同期および通信が必要
- IPCのトピック

- 競合状態(Race Condition)
- クリティカルリージョン(Critical Region)
- ビジー等待による相互排除
(Mutual Exclusion with Busy Waiting)
- スリープとウェイクアップ(Sleep and Wakeup)
- セマフォ(Semaphores)
- ミューテックス(Mutexes)
- モニタ(Monitors)
- メッセージ交換(Message passing)
- バリア(Barriers)

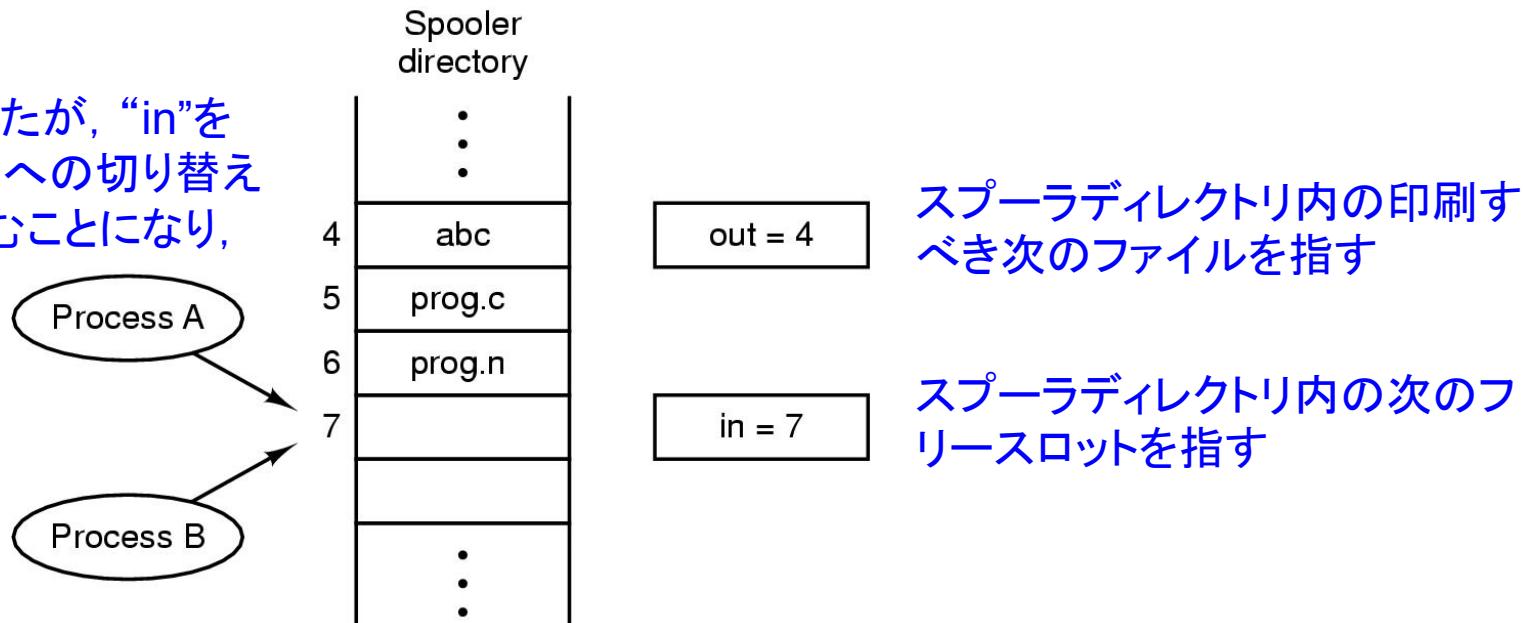


プロセス間通信(2)

□ 競合状態(Race Conditions)

- 協調して仕事を行うプロセス同士は、それぞれが読み書き可能な共通のストレージ(メモリあるいはファイル)を共有しうる
- プロセスは共有データを読み書きする。最終結果はどのプロセスがいつ実行されたかに依存する
- 例) プリンタスプール
 - 仮定：印刷されるファイルはスプーラディレクトリに挿入される

プロセスAが“in” = 7の値を得たが、“in”を更新する前にプロセスAからBへの切り替えが起こると、Bも“in” = 7を読むことになり、間違った実行となる



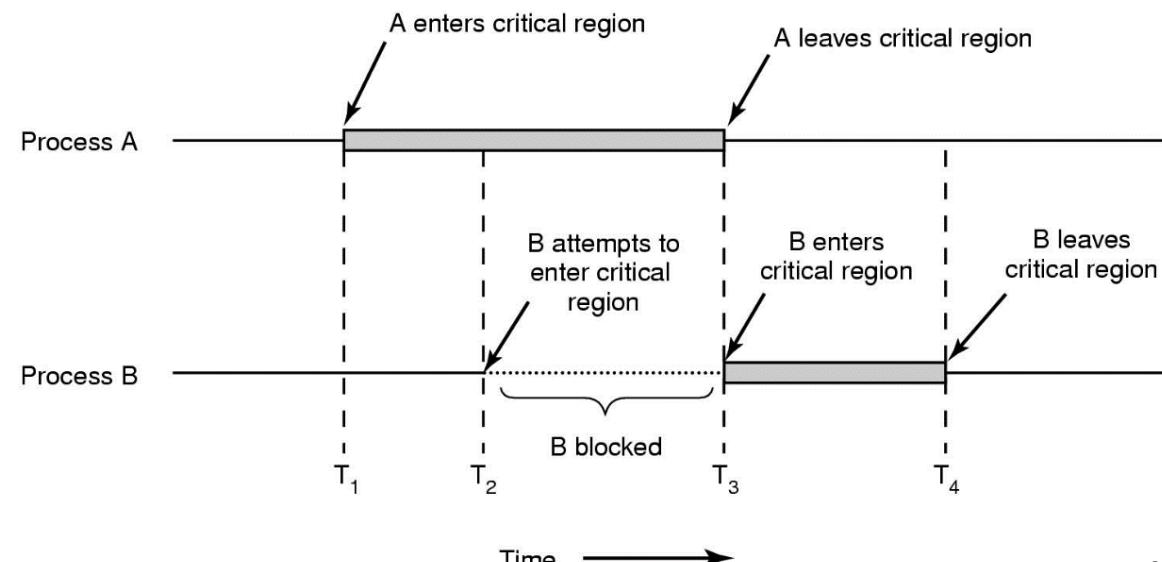
プロセス間通信(3)

- 競合状態を解決する鍵は、2つ以上のプロセスが共有データを同時に読み書きできないようにすることである — 相互排除 (Mutual exclusion)

□ クリティカルリージョン(Critical Regions／sections)

- プログラム中で(メモリなどの)共有ストレージが参照される部分
- 2つ以上のプロセスが同時にクリティカルリージョンに入るのを禁止することにより、競合状態を回避可能

クリティカルリージョンを使用した相互排除



プロセス間通信(4)

□ ビジー・ウェイトによる相互排除(Mutual Exclusion with Busy Waiting)

■ 相互排除／クリティカルリージョンのための方法

- 割込みの不許可



不完全, あるいは非効率

- ロック変数



完全だが, やや非効率

- 完全な交互実行



完全, かつ効率的

- Petersonの解決法

- TSL命令

プロセス間通信(5)

■ 割込みの不許可(バグの危険性＆マルチ環境で不完全)

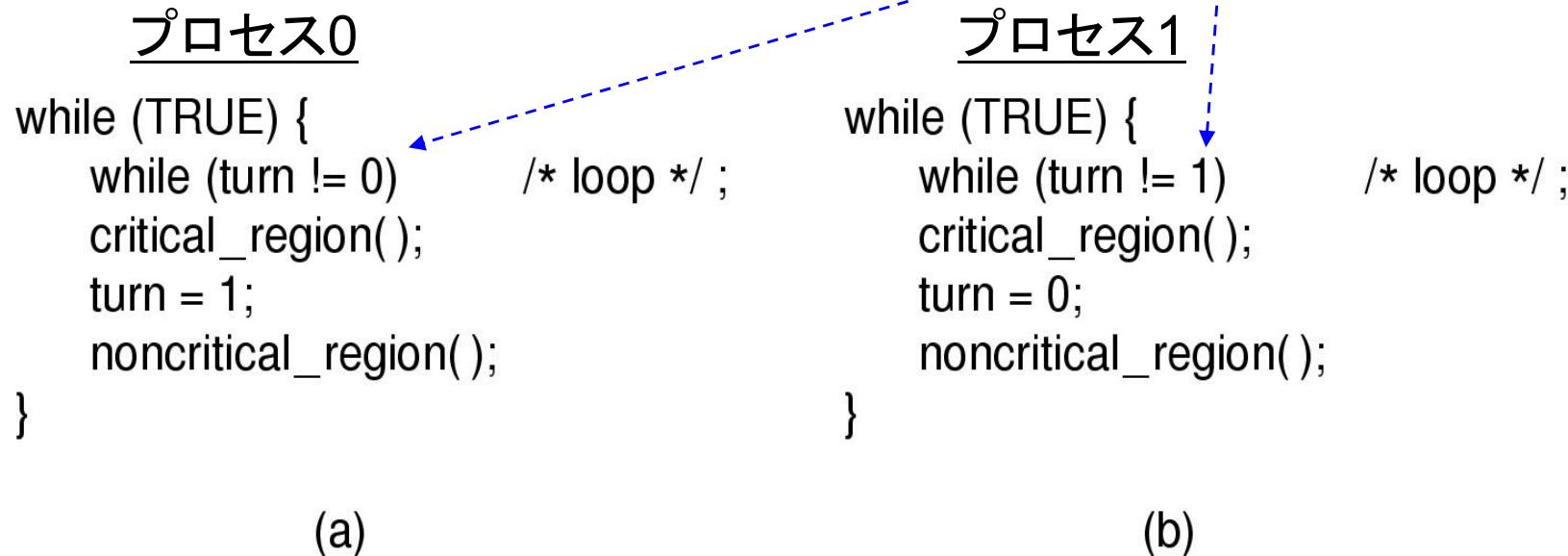
- 最も単純な解決法は、各プロセスがクリティカルリージョンに入った直後、全ての割込みを不許可にし、そのクリティカルリージョンを出るときに再び許可にすること
 - 割込み不許可状態では、プロセスが切り替わるチャンスは無い
 - したがって、共有データを参照する際、他のプロセスが介入することを心配する必要がない
- ユーザプロセスに割込み許可・不許可を制御する権限を与える必要があるため、この方法は有力候補とはなりえない(不注意により、割込み不許可のままになる恐れ)
 - OS自体(カーネルデータ用)にはこの方法がよく使用される
- マルチコア／マルチプロセッサ環境では、一つのコア／プロセッサを割込み不許可にしても、他のコア／プロセッサが介入する可能性あり

■ ロック変数(不完全)

- 共有(ロック)変数(初期値は0)を用意
- プロセスがクリティカルリージョンに入ろうとするとき、最初にロック変数を調べる
 - ロック変数の値が0ならば、そのプロセスはロックを1にセットし、リージョンに入る
 - 0でなければ、0になるまで単に待つ
- この方法は不完全
 - プロセスがロック値として0を読んだ直後、他のプロセスがスケジュールされ実行権を得て同じくロック値として0を読んだ場合

プロセス間通信(6)

■ 完全な交互実行(正しいが、非効率)



- 完全に交互の関係で仕事を行うプロセス同士にのみ適用可能
- プロセスの1つが他よりも遅い場合、これは良い方法ではない
 - 他方がnoncritical_region処理を行っている間、待たされることになる

プロセス間通信(7)

■ Petersonの解決法(正しいが、やや非効率)

- ソフトウェアによる(前述の方法と比較し)より良い方法

```
#define FALSE 0
#define TRUE 1
#define N      2           /* number of processes */

int turn;                  /* whose turn is it? */
int interested[N];         /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;               /* number of the other process */

    other = 1 - process;    /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;          /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

プロセス間通信(8)

■ TSL命令(ハードウェアサポートにより効率的)

- 多くのCPUは test & set lock (またはそれに類似の)命令を持っている

TSL Rx, lock

- この命令はメモリ変数“lock”的内容をレジスタ“Rx”に読み込み, 非ゼロ値を“lock”に書き込む
- 読む操作と書く操作は不可分となることがハードウェアにより保証されている
 - 命令実行(読む then 書く)が完了するまで, 他のプロセッサはそのメモリワードをアクセスできない → マルチコア/プロセッサで排他制御が可能

クリティカルリ → enter_region:

ーションの入口
で実行

```
TSL REGISTER,LOCK  
CMP REGISTER,#0  
JNE enter_region  
RET
```

```
| copy lock to register and set lock to 1  
| was lock zero?  
| if it was not zero, lock was set, so loop  
| return to caller; critical region entered
```

クリティカルリ → leave_region:

ーションの出口
で実行

```
MOVE LOCK,#0  
RET
```

```
| store a 0 in lock  
| return to caller
```

プロセス間通信(9)

- Petersonの解決法とTSLによる解決法は共に正しく動作するが、ビジー・ウェイトが必要であるという欠点を持つ
 - CPU時間の浪費
 - スリープとウェイクアップ(**Sleep and Wakeup**)
 - 2つのシステムコール
 - “sleep()”により呼び出しプロセスはブロックし、他のプロセスによって起床させられるまでスリープ状態となる
 - “wakeup(proc)”は“proc”番プロセスを起床させる
 - 例) ここからは、生産者・消費者問題の一つを扱う
 - 生産者は情報をバッファへ入れる
 - バッファが一杯のときはスリープする
 - 消費者はそれを取り出す
 - バッファが空のときはスリープする
- ⇒ Next slide

プロセス間通信(10)

```
#define N 100                                /* number of slots in the buffer */
int count = 0;                                /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                            /* repeat forever */
        item = produce_item();                /* generate next item */
        if (count == N) sleep();              /* if buffer is full, go to sleep */
        insert_item(item);                  /* put item in buffer */
        count = count + 1;                  /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);    /* was buffer empty? */
    }
}
```

不完全なコード！ “count”がクリティカルリージョンで守られていない

```
void consumer(void)
{
    int item;

    while (TRUE) {
        if (count == 0) sleep();            /* if buffer is empty, got to sleep */
        item = remove_item();             /* take item out of buffer */
        count = count - 1;               /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item);
    }
}
```

“count” (= 0)を読み出した直後、値をテストする
前にプロセス切り替えが起こった場合

wakeupが無視される

ゆくゆくは、両方がス
リープする

セマフォが有効

プロセス間通信(11)

□ セマフォ(Semaphores)

- (将来の)ウェイクアップの回数をカウントする(溜め込む)変数
(一般的には“資源数”を意味する変数)
- 以下の2つの操作により変数操作およびプロセス動作を制御
 - “down” (“sleep” / “P”)
 - セマフォ変数が0より大きいかどうかチェックする。大きければ、1を減じて継続。そうでなければ、プロセスはスリープする
 - 値をチェックし、更新し、スリープするまでの全てが1つの不可分動作として実行される
 - “up” (“wakeup” / “V”)
 - 1つ以上のプロセスがそのセマフォでスリープ状態となっている場合は、その中の1つを選び起床させる。スリープ状態のプロセスが無い場合はセマフォ変数の値を1増加させる。
 - これも不可分動作

プロセス間通信(12)

セマフォによる生産者・消費者問題の解法

3つのセマフォを使用:

mutex:

相互排除に使用

empty, full:

バッファ管理に使用

going to sleep

when not satisfied
(vs. busy-waiting)

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;           /* controls access to critical region */
semaphore empty = N;          /* counts empty buffer slots */
semaphore full = 0;           /* counts full buffer slots */
```

```
void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```
void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

```
/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */
```

初期値 = 1 : “binary semaphore”と呼ばれ,
排他制御のロックとして利用される

```
/* TRUE is the constant 1 */
/* generate something to put in buffer */
/* decrement empty count */
/* enter critical region */
/* put new item in buffer */
/* leave critical region */
/* increment count of full slots */
```

```
/* infinite loop */
/* decrement full count */
/* enter critical region */
/* take item from buffer */
/* leave critical region */
/* increment count of empty slots */
/* do something with the item */
```

プロセス間通信(13)

□ POSIXにおけるセマフォ

Thread call	Description
sem_init	Initialize a semaphore
sem_destroy	Destroy an existing semaphore
sem_wait	Decrement semaphore or block
sem_trywait	Decrement semaphore or fail
sem_post	Increment semaphore

sem_trywait: セマフォ値が0の場合は、ブロックせずに、減算失敗のエラーコードでリターンする。減算ができなくても他の仕事がある場合に有効

```
#include <semaphore.h>
sem_t    sem;
int      ret;
ret = sem_init ( &sem, 0, 1 );
...
ret = sem_wait ( &sem );
...
ret = sem_post ( &sem );
```

プロセス間通信(14)

□ ミューテックス(Mutexes)

- セマフォを簡単化・限定したもの：共有資源への相互排除を管理

- “mutex_lock”

- ミューテックス変数がロックされているかどうかチェックし、ロックされていなければ、呼出し元がクリティカルリージョンに入ることを許す。ロックされていれば、呼出し元をブロックする(スリープさせる)

- “mutex_unlock”

- ブロックされているスレッドを1つ選び、ロックを獲得させる。無ければロックを開放。

mutex_lock:

```
TSL REGISTER,MUTEX  
CMP REGISTER,#0  
JZE ok  
CALL thread_yield  
JMP mutex_lock  
ok: RET
```

```
| copy mutex to register and set mutex to 1  
| was mutex zero?  
| if it was zero, mutex was unlocked, so return  
| mutex is busy; schedule another thread  
| try again  
| return to caller; critical region entered
```

ここでの方法はプロセスレベルではなく、
スレッドライブリレベルで実現されている

mutex_unlock:

```
MOVE MUTEX,#0  
RET
```

```
| store a 0 in mutex  
| return to caller
```

プロセス間通信(15)

□ Pthreadsにおけるミューテックス

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Pthread_mutex_trylock: 既にロックされていた場合は、ブロックせずに、獲得失敗のエラーコードでリターンする。スレッドにとって、ロックの獲得ができないても他の仕事がある場合に有効

プロセス間通信(16)

- セマフォは注意深く使用しなければならない。さもなければ、簡単にデッドロックに陥る
 - 例) 生産者側の2つの“down”的順序を入れ替えた場合
("down(&empty)") \leftrightarrow "down(&mutex)"

Skip

□ モニタ(Monitors)

- mutexやwait/signalなどによって実現される高位レベルプリミティブ
 - コンパイラが低位プリミティブを利用して排他実行になるコードを生成することを仮定している
- 手続き、変数およびデータ構造の集合(特別なモジュール／パッケージ)
- プロセスはモニタ内の手続きを呼び出すことができるが、モニタ内のデータを直接参照することはできない
- いかなる瞬間も、モニタ内でアクティブになれるプロセスは1つのみ

```
monitor example
    integer i;
    condition c; 条件変数

    procedure producer();
        .
        .
    end;

    procedure consumer();
        .
        .
    end;
end monitor;
```

- モニタ単位で排他アクセス／実行は保証されるが、（バッファがFull,などの）先に進めない状況で、プロセス自身をブロックして、CPUを他に明け渡す仕組みが更に必要
- 条件変数とwait/signal
 - モニタの手続き内で、継続不可能であると判断した場合、ある条件変数に対して“**wait**”を実行
 - 呼出し元プロセスをブロック
 - 他のプロセスがモニタに入ることを許す
 - その他のプロセスがスリープ中のプロセスを起床させる際、その条件変数に対して“**signal**”を実行
 - 複数のプロセスがウェイトしている条件変数に対して“**signal**”を実行した場合、そのうち1つのみがシステムスケジューラによって選ばれ、実行される
 - モニタ内で2つのプロセスが同時にアクティブになるのを回避するために、“**signal**”は手続きの最後に実行される必要がある

□ 生産者・消費者問題のモニタによる解法の概要

■ Buffer has N slots

```
monitor ProducerConsumer
  condition full, empty;
  integer count;
  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;
  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;
  count := 0;
end monitor;
```

```
procedure producer;
begin
  while true do
    begin
      item = produce_item;
      ProducerConsumer.insert(item)
    end
  end;
procedure consumer;
begin
  while true do
    begin
      item = ProducerConsumer.remove;
      consume_item(item)
    end
  end;
```

プロセス間通信(19)

□ メッセージ交換(Message Passing)

- 2つのプリミティブ: **send** と **receive**
 - **send(destination, &message);**
 - 指定されたデスティネーションへメッセージを送る
 - **receive(source, &message);**
 - 指定されたソース(あるいはANY)からメッセージを受取る
 - メッセージが到着していない場合は、到着するまでブロックするか、エラーコードで即座にリターンする
- MPI/MPI-2/MPI-3 (Message Passing Interface)
 - メッセージパッシングのためのライブラリの規格であり、MPIに従う様々な実装がある(MPICH, OpenMPI, など)

プロセス間通信(20)

■ メッセージ交換による生産者・消費者問題の例

```
#define N 100                                     /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                                     /* message buffer */

    while (TRUE) {
        item = produce_item();                     /* generate something to put in buffer */
        receive(consumer, &m);                    /* wait for an empty to arrive */
        build_message(&m, item);                  /* construct a message to send */
        send(consumer, &m);                      /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

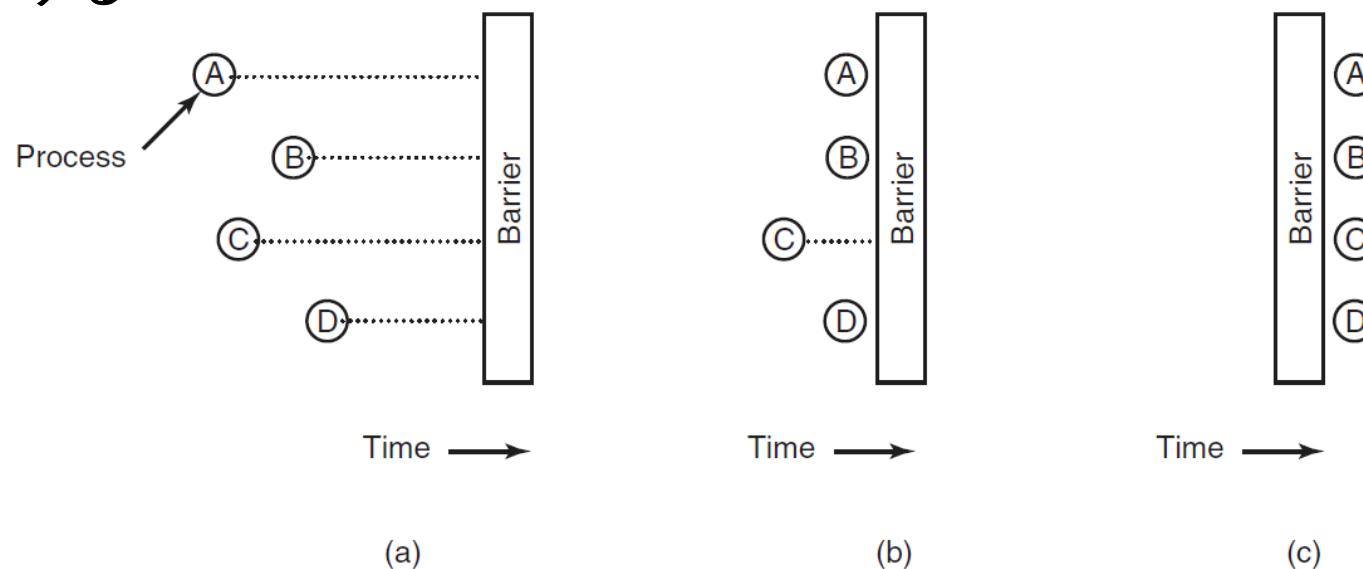
    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);                  /* get message containing item */
        item = extract_item(&m);                /* extract item from message */
        send(producer, &m);                    /* send back empty reply */
        consume_item(item);                   /* do something with the item */
    }
}
```

Emulation of
N-entry buffer,
(equivalent to
previous examples)

プロセス間通信(21)

□ バリア(Barriers)

- プロセス群(2つのプロセスによる生産者・消費者を越えて)が対象
 - アプリケーションの実行が複数フェーズに分割され、全てのプロセスが次のフェーズに進む準備ができるまで待つ必要がある場合などに有用
- 各フェーズの最後にバリア(barrier)を配置
 - プロセスはバリアに到達したとき、全プロセスがそのバリアに到達するまでブロックする



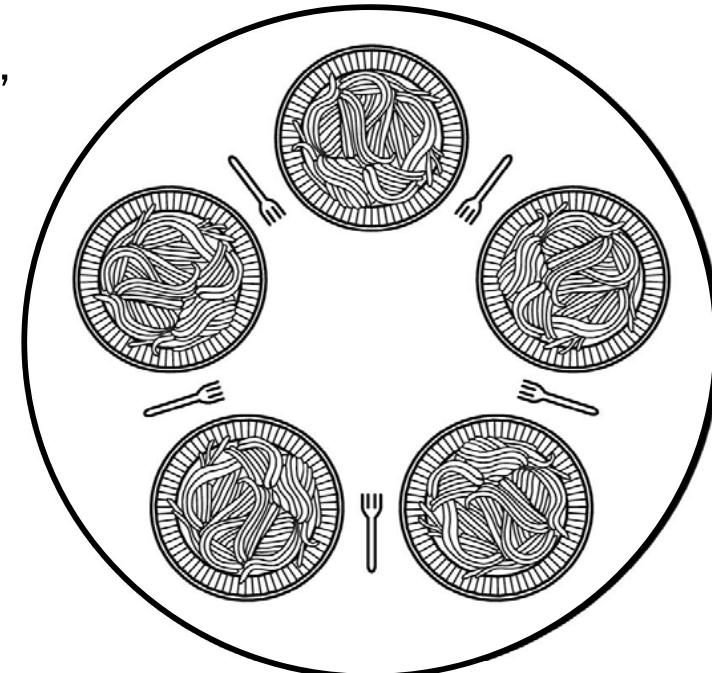
- Pthreadsにおけるpthread_barrier_init/wait/destroy

古典的プロセス間通信問題(1)

□ 哲学者の食事問題

5人の哲学者が円卓に座っている。各哲学者にはスパゲッティの皿がある。スパゲッティはつるつるしているため、彼らには食べるためには2つのフォークが必要であるが、皿の間に1つのフォークしかない。

哲学者の生活は食事と思考を交互に行うことである。哲学者が空腹になると、左右のフォークを1つずつ(順不同)に手に取る。2つのフォークを得ることができたら、しばらくの間食事をし、フォークを置き、思考を続ける。



以上の行動を、行き詰る(デッドロックする)ことなく行う各哲学者のプログラムをどのように記述できるだろうか？

古典的プロセス間通信問題(2)

■ 自明な解法の例

- ▣ “take_fork(i)”: フォーク”i”が利用可能となるまで待ってからフォークを得る
- ▣ この解は間違っている
 - 哲学者5人全てが左のフォークを同時に手に取った場合に行き詰まる
 - この状況を デッドロック(deadlock) と呼ぶ

```
#define N 5                                     /* number of philosophers */  
  
void philosopher(int i)                        /* i: philosopher number, from 0 to 4 */  
{  
    while (TRUE) {  
        think();  
        take_fork(i);  
        take_fork((i+1) % N);  
        eat();  
        put_fork(i);  
        put_fork((i+1) % N);  
    }  
}
```

/* philosopher is thinking */
/* take left fork */
/* take right fork; % is modulo operator */
/* yum-yum, spaghetti */
/* put left fork back on the table */
/* put right fork back on the table */

古典的プロセス間通信問題(3)

■ 若干の修正

- 左のフォークを手に取った後、右のフォークが使用可能かどうかチェックする。もし使用不可能であれば、哲学者は左のフォークを手放し、しばらく待つ。そして最初から繰り返す。

- この解も失敗しうる

- 全ての哲学者がこのアルゴリズムを同時に開始した場合、全員が左のフォークを取り、右のフォークが使用できないことを確認し、左のフォークを手放し、しばらく待ち、全員が同時に左のフォークを再度取り…というように、永遠に続く可能性
- このような状況を ライブロック(livelock) と呼ぶ
 - 全てのプログラムが永遠に走り続けるが、決して前に進めない状況

■ 次スライドの解法はデッドロックもライブロックも起こらず、最大の並列性を持つ

- 配列 *state* は哲学者が食事中、思考中、空腹中のどれであるかを示す
- 配列 *s* は哲学者ごとのセマフォ

古典的プロセス間通信問題(4)

```
#define N          5           /* number of philosophers */  
#define LEFT       (i+N-1)%N  /* number of i's left neighbor */  
#define RIGHT      (i+1)%N   /* number of i's right neighbor */  
#define THINKING   0           /* philosopher is thinking */  
#define HUNGRY     1           /* philosopher is trying to get forks */  
#define EATING     2           /* philosopher is eating */  
  
typedef int semaphore;  
int state[N];  
semaphore mutex = 1;  
semaphore s[N];  
  
void philosopher(int i)  
{  
    while (TRUE) {  
        think();  
        take_forks(i);  
        eat();  
        put_forks(i);  
    }  
}  
  
/* semaphores are a special kind of int */  
/* array to keep track of everyone's state */  
/* mutual exclusion for critical regions */  
/* one semaphore per philosopher */  
  
/* i: philosopher number, from 0 to N-1 */  
  
/* repeat forever */  
/* philosopher is thinking */  
/* acquire two forks or block */  
/* yum-yum, spaghetti */  
/* put both forks back on table */
```

古典的プロセス間通信問題(5)

```
void take_forks(int i)                                /* i: philosopher number, from 0 to N-1 */  
{  
    down(&mutex);  
    state[i] = HUNGRY;  
    test(i);  
    up(&mutex);  
    down(&s[i]);  
}  
  
void put_forks(i)                                     /* i: philosopher number, from 0 to N-1 */  
{  
    down(&mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT);  
    up(&mutex);  
}  
  
void test(i) /* i: philosopher number, from 0 to N-1 */  
{  
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {  
        state[i] = EATING;  
        up(&s[i]);  
    }  
}
```

/* enter critical region */
/* record fact that philosopher i is hungry */
/* try to acquire 2 forks */
/* exit critical region */
/* block if forks were not acquired */

/* enter critical region */
/* philosopher has finished eating */
/* see if left neighbor can now eat */
/* see if right neighbor can now eat */
/* exit critical region */

Skillful use of semaphore !!
(Initially, zero)

古典的プロセス間通信問題(6)

□ リーダ・ライタ問題

複数のプロセスが同時にデータベースを読むのは許されるが、何らかのプロセスによるデータベースの更新中は、他のプロセスは、たとえ読むだけでも、データベースにアクセスしてはならない

【問題点】

すくなくとも1つのreaderがアクティブの間は、後続するreaderは許可される。readerが存在しなくなるまで、writerは停止させられる。したがって、新しいreaderが絶えず到着する状況では、writerは許可されることはない
⇒ 解決法が望まれる

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

/* use your imagination */
/* controls access to 'rc' */
/* controls access to the database */
/* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}

/* repeat forever */
/* get exclusive access to 'rc' */
/* one reader more now */
/* if this is the first reader ... */
/* release exclusive access to 'rc' */
/* access the data */
/* get exclusive access to 'rc' */
/* one reader fewer now */
/* if this is the last reader ... */
/* release exclusive access to 'rc' */
/* noncritical region */

void writer(void)
{
    while (TRUE) {
        think_up_data();
        down(&db);
        write_data_base();
        up(&db);
    }
}

/* repeat forever */
/* noncritical region */
/* get exclusive access */
/* update the data */
/* release exclusive access */
```

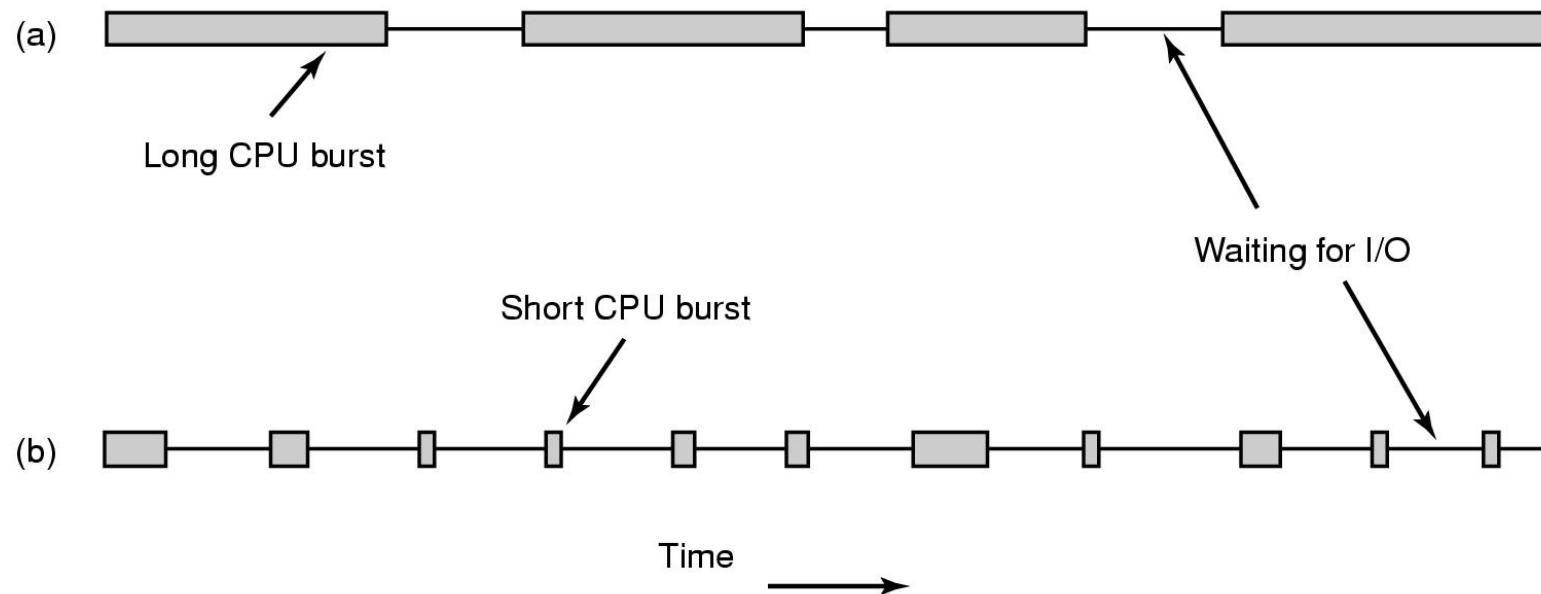
スケジューリング(1)

- マルチプログラミングでは、複数のプロセスが同時に1つのCPUを獲得しようとして競合する状況が頻繁に起こる
 - この状況は、2つ以上のプロセスが同時に実行可能(ready)状態にあるときはいつでも起こる
 - 使用可能なCPU／コアが1つしかない場合は、次に実行すべきプロセスを1つ選択する必要がある
 - 使用可能なCPU／コアがN個のときは、実行すべきN個のプロセスを選択する
 - OSの内部でこの選択を行う部分のことをスケジューラ(scheduler)と呼ぶ
 - 選択のアルゴリズムをスケジューリングアルゴリズム(scheduling algorithm)と呼ぶ
 - スケジューラが実行プロセスを切り替える際に、プロセス／コンテキスト切り替え(process/context switching)を行う

スケジューリング(2)

- プロセスは通常、計算処理とI/O要求(待ち)を繰り返す
 - 計算部分が長いプロセスや、短い(頻繁なI/O要求)プロセスが存在する

- (a) Compute-bound (CPU-bound) vs. (b) I/O-bound



- I/O待ちになる場合、プロセスを切り替えることでCPUを有効利用することが重要

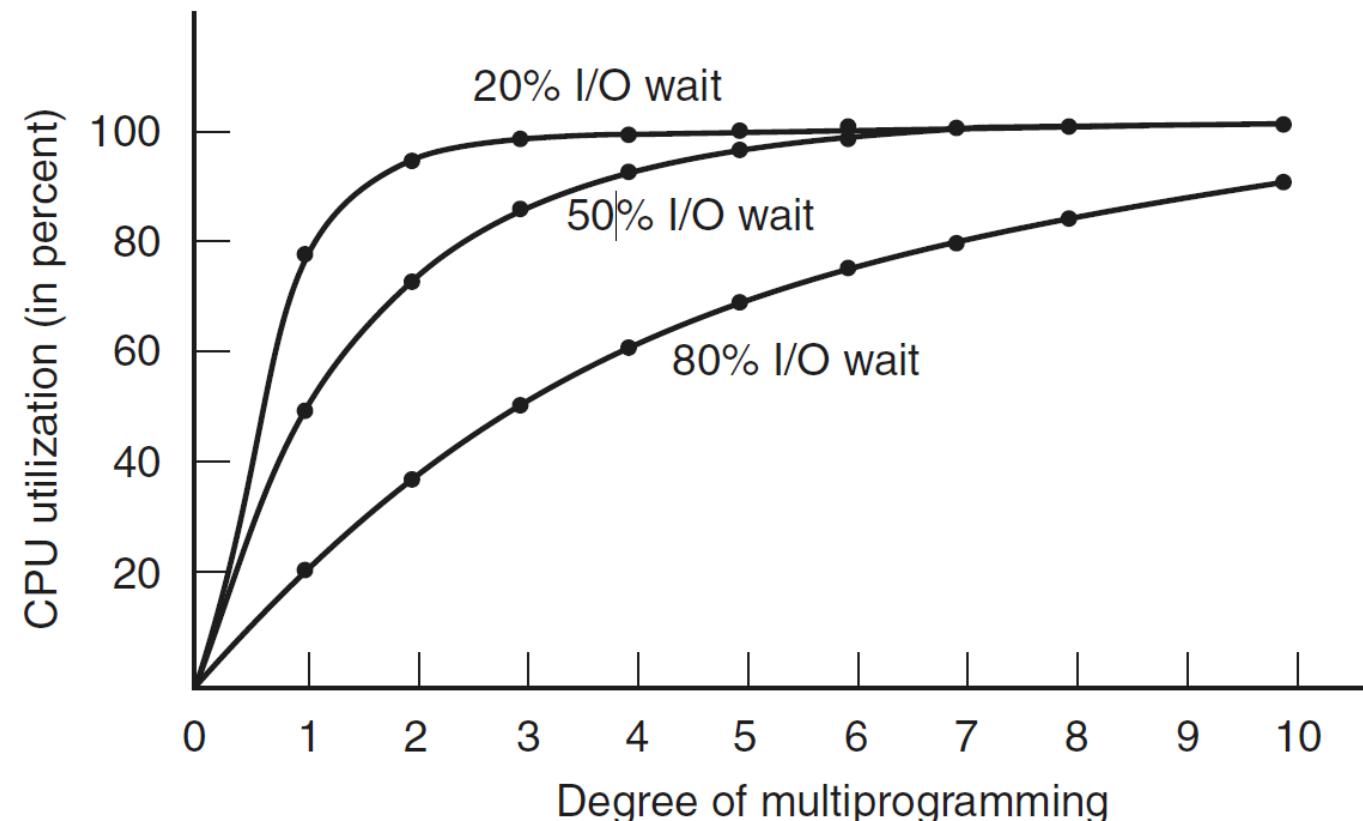
スケジューリング(3)

Revisiting “process (9)” in Lec2

□ マルチプログラミングとCPU使用率の関係

p : プロセスあたりのI/O待ち時間の割合 : p
 n : マルチプログラミングのプロセス数

$$\text{CPU利用率} = 1 - p^n$$



スケジューリング(4)

□ スケジューリング処理のタイミング

1. 新たなプロセスの生成時
 - 親プロセスと子プロセスのうち、どちらかを選択
2. 実行プロセスの終了時
 - 他の実行可能(ready)プロセス群から一つを選択
3. 実行プロセスがI/O待ちになるとき
 - 他の実行可能(ready)プロセス群から一つを選択

■ Nonpreemptive vs. preemptive

- Nonpreemptive
 - プロセスは開始後、終了するかI/O待ちになるまで実行を継続するシステム
- Preemptive
 - 割込み発生時にプロセス切替を許すシステム

スケジューリング(5)

□ スケジューリング処理のタイミング(つづき)

4. 割込み発生時(preemptiveの場合)

- 割込み処理が終了したとき、スケジューリングを行うことが可能
- 割込み前の実行プロセスと実行可能プロセス群の中から一つを選択
- 割込み前の(割りこまれた)実行プロセスか、割込みによって(I/O処理完了などで)リスタートできるプロセスが有力候補

4b. 周期(ティック)割込み発生時(preemptiveの場合)

- 一定時間(ティック)毎にプロセスを切り替える選択(ラウンドロビン)が一般的
- 実行可能プロセス群から一つを選択

スケジューリング(6)

□ 異なる状況下でのスケジューリングアルゴリズムの目標

All systems

Fairness - giving each process a fair share of the CPU

Policy enforcement - seeing that stated policy is carried out

Balance - keeping all parts of the system busy

例) 安全制御タスクは給料支払いタスクよりも重要, など

Batch systems

Throughput - maximize jobs per hour

Turnaround time - minimize time between submission and termination

CPU utilization - keep the CPU busy all the time

Interactive systems

Response time - respond to requests quickly

Proportionality - meet users' expectations →
釣り合い

比例性: ユーザの予測・期待

例) 大サイズ映像ファイルをクラウドにアップロードするタスクは、クラウドとの接続を切るタスクよりも時間がかかる, など

Real-time systems

Meeting deadlines - avoid losing data

Predictability - avoid quality degradation in multimedia systems

スケジューリング(7)

□ バッチシステムにおけるスケジューリング

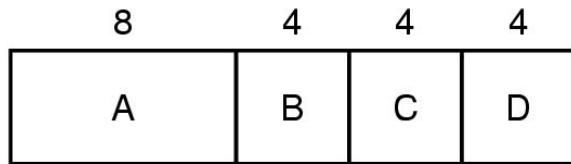
■ 到着順サービス(First-Come First-Served)

- 最も簡単なスケジューリングアルゴリズム
- 新しい到着タスクはレディキューの最後に挿入される

■ 最短ジョブ優先(Shortest Job First)

- 全てのタスクの実行要求が同時に発生し、事前に実行時間がわかっているものと仮定する

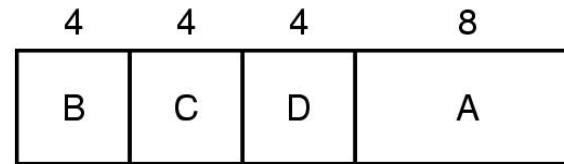
A→B→C→Dの場合



(a)

Average turnaround time = 14

Shortest Job First: B→C→D→A



(b)

Average turnaround time = 11

■ 最小残り時間優先(Shortest Remaining Time Next)

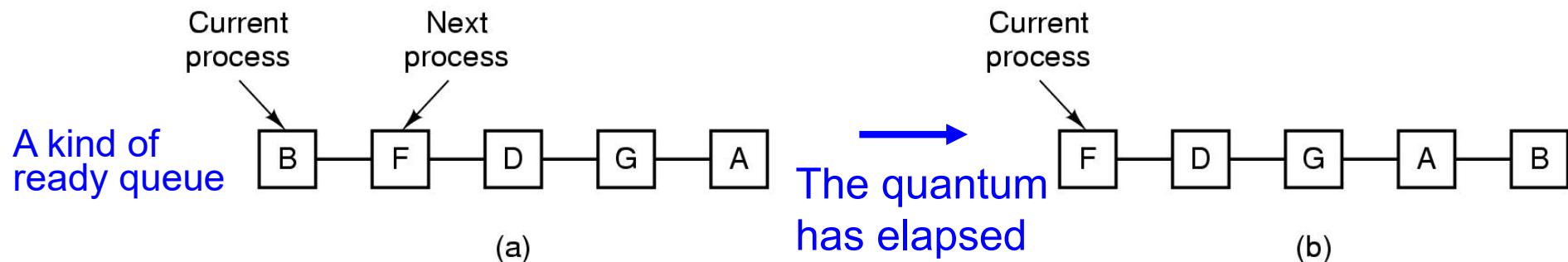
- 最短ジョブ優先に類似しているが、全タスクの同時発生の仮定がない
- プリエンプティブ方式に属する

スケジューリング(8)

□ 会話型システムにおけるスケジューリング

■ ラウンドロビンスケジューリング(Round-Robin Scheduling)

- (最も古く,) 最も単純で、最も公平で、最も広く使用されている
- 各プロセス実行に一定時間(クオントム, quantum)が与えられる
 - 実行プロセスはクオントムの終わりにお実行中の場合は、CPUが奪われ、他のプロセスに渡される(プリエンプション)

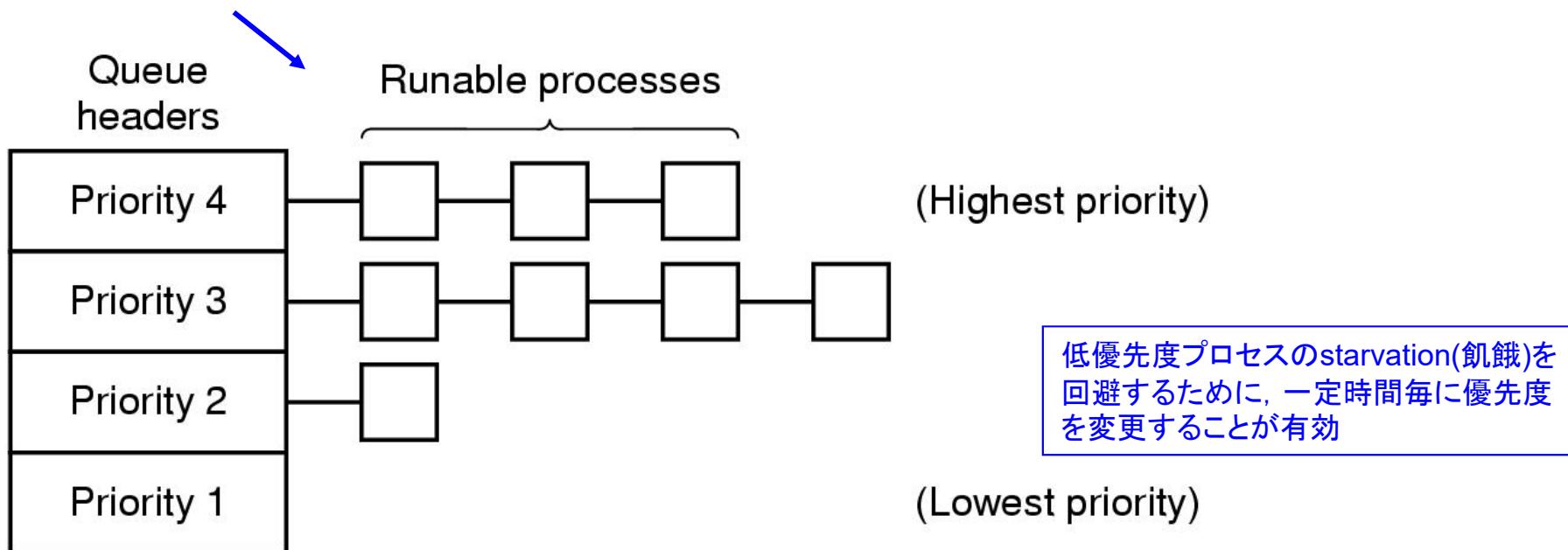


- クオントムを過度に短くすると、プロセス切り替えの頻度が高くなりすぎ、CPU使用率が下がる
 - プリエンプション(プロセス切替)には切替オーバヘッドが存在する
- 長くしそぎると、(実行時間が短い)会話型プロセスの応答が遅くなる
- クオントムとして、20-50 ミリ秒くらいが設定されることが多い

スケジューリング(9)

■ 優先度スケジューリング(Priority Scheduling)

- 各タスクに(静的あるいは動的に)優先度が割り当てられる
- 最も高い優先度を持つ実行可能プロセスが実行を許可される
- ラウンドロビンと組合せ可能(同一優先度プロセス群内)



- 会話型システム用の他のアルゴリズムがいくつか存在する。教科書参照

スケジューリング(10)

□ リアルタイムシステムにおけるスケジューリング

- リアルタイムシステムでは、正しい計算結果でも、遅すぎると無意味になることがある

- ハードリアルタイム：守らなければならない絶対的な締切時刻を持つ
 - ソフトリアルタイム：稀に締切り時刻を越えることは許容される

- 2種類のタスク起動タイミング

- 周期的タスク：一定の時間間隔で起動する
 - 非周期的タスク：不定期に発生する

- 周期的タスクは、少なくとも、以下を満たす必要がある

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1 \quad \begin{cases} C_i: \text{exec. time for task } i \\ P_i: \text{period of task } i \end{cases}$$

- スケジューラは、全てのタスクの締切時刻(デッドライン)が満たされるようにタスク群をスケジュールすることが求められる

- 詳しくは、I440「高機能オペレーティングシステム」またはI470F「統合アーキテクチャ」

スケジューリング(11)

□ スレッドスケジューリング

■ ユーザレベルスレッド

□ 利点

- 軽いスレッド切り替え
- アプリケーションに特化したスケジューリングアルゴリズムが使用可能

□ 欠点

- スレッドがI/O要求でブロックした場合、プロセス全体が中断

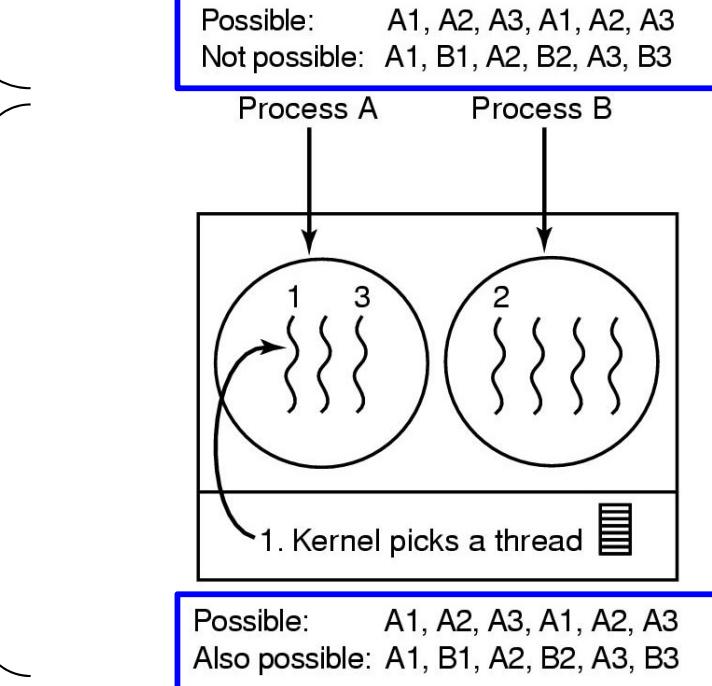
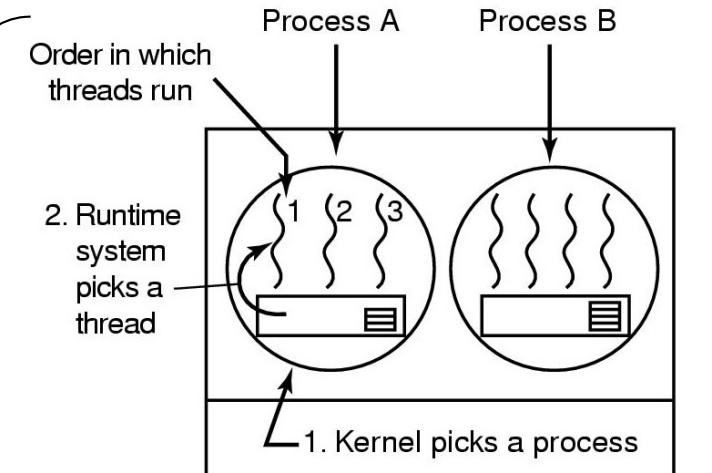
■ カーネルレベルスレッド

□ 利点

- I/O要求でブロックした場合でも、プロセス全体を中断させることはない

□ 欠点

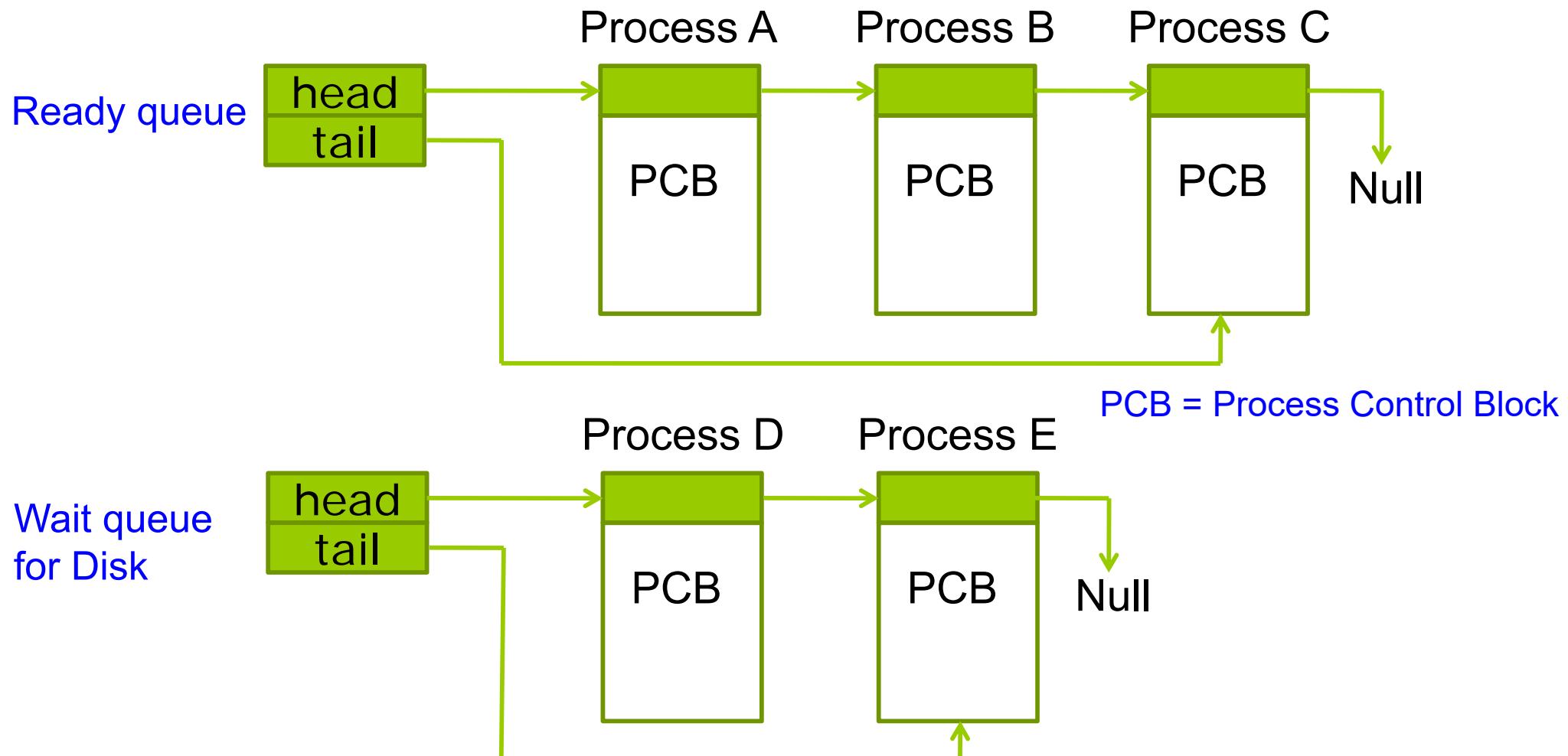
- スレッド切り替えが低速
- スケジューリングポリシーがプロセスのスケジューリングに基づいている(カーネルに委ねられる)



スケジューリング(12)

- スケジューリングおよびタスクの管理にはタスクのキューが使用される

- レディキュー, 待ちキュー(デバイスキュー)



スケジューリング(13)

□ 復習(もう一つの整理方法): CPUスケジューリングが行われるのは以下の5つの状態遷移状況が発生した場合:

(1) プロセスが実行状態から待ち状態に変わったとき

□ 例) I/O処理を要求したとき

(2) プロセスが実行状態から実行可能(レディ)状態に変わったとき

□ 例) ティック割り込み発生時(ラウンドロビン)

(3) プロセスが待ち状態から実行可能状態に変わったとき

□ 例) I/O処理が完了したとき(完了通知割込み発生時)

(4) プロセスが終了したとき(実行状態 → Non-existence)

(5) プロセスが生成されたとき(Non-existence → 実行可能状態)

プロセス切替が(1)と(4)の状況のみ発生する場合, ノンプリエンプティブと呼ばれる

(2)と(3), および(5)も起こりうる場合はプリエンプティブ

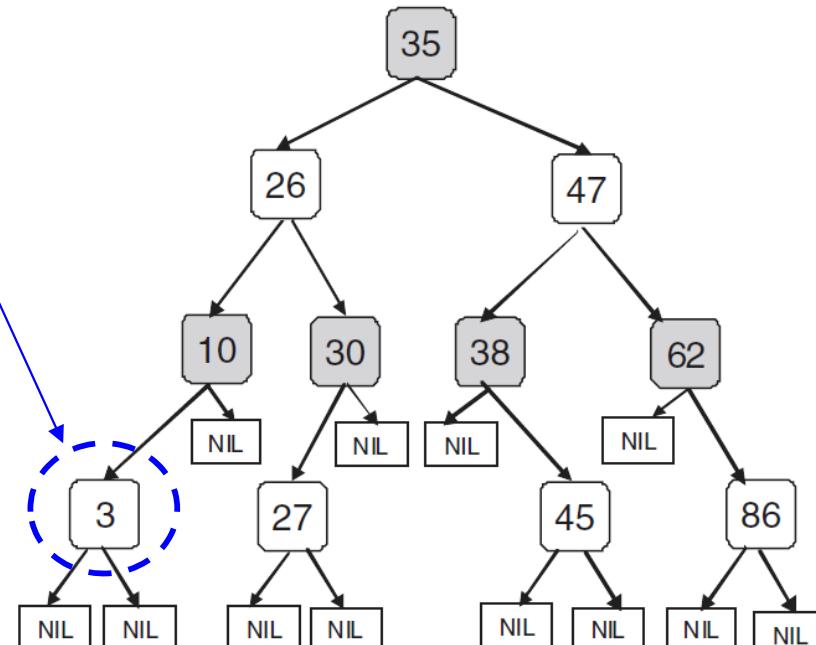
Linuxにおけるスケジューリング(1)

- Linuxではスケジューリングの対象はカーネルスレッド
- スレッド群を3つのクラスに分類
 - 1. リアルタイム FIFO
 - 高い優先度(0~99)を持つスレッドをノンプリエンプティブで実行
 - 2. リアルタイムラウンドロビン
 - 高い優先度(0~99)を持つスレッドをラウンドロビン(プリエンプティブ)で実行
 - 3. 時分割(timesharing)
 - 低い優先度(100~139)を持つスレッドをラウンドロビンで実行
- 上記3は, “runqueue”: 実行可能スレッドのキュー(= ready queue)を木構造で構築してスケジュールする(次スライド)

デッドラインは
考慮されない

Linuxにおけるスケジューリング(2)

- Completely Fair Scheduler (CFS) (release 2.6.23以降)
 - red-black tree (平衡二分木の一種)
 - CFSはタスクの実行時間を管理する
 - タスクのこれまでの実行時間(vruntime)に基づいて、タスク集合を木構造の中で順序付けする
 - 左の子ノードはより短いvruntimeを持つタスク
 - 右の 子ノードはより長いvruntimeを持つタスク
 - スケジューラは、木の左下に位置するタスク(最小vruntime)を実行タスクとして選択
 - 優先度を反映するために、低優先度タスクは実行時間がより早く経過するようにする (= their vruntime increases more rapidly)



- 実行タスクの選択は $O(1)$
- タスクの挿入は $O(\log N)$

デッドロック



リソース(1)

□ リソース(資源)

- プロセスによって使用されるもの
 - CD／DVDドライブなどのハードウェアデバイス
 - メモリ内データ, ファイル, データベースのレコードのような情報, など
- 横取り可能なリソース(Preemptable resources)
 - 所有しているプロセスに悪影響を与えることなく横取りできる
 - 例) メモリ内のデータ格納場所
 - 使用部分はディスクに書き出すことにより, 安全に横取りできる
- 横取り不可能なリソース(Nonpreemptable resources)
 - 横取りすると失敗する
 - 例) CD／DVDレコーダー
- 一般的には, デッドロックは横取り不可能なリソースに関して発生する

リソース(2)

■ リソースに対するイベント列

1. リソースへの要求
2. リソースの使用
3. リソースの開放

■ 要求が却下された場合

- 要求を出したプロセスはブロックされる, あるいは,
- 要求がエラーで終了する
 - 失敗でリターン → 非同期処理が可能(callerプロセスが少し待ってリトライ, 等)

■ ユーザがリソース管理を行う方法として, 各リソースにセマフォを関連付ける方法がある ⇒ 次スライド

リソース(3)

```
typedef int semaphore;  
semaphore resource_1 = 1;  
semaphore resource_2 = 1;
```

プロセス1とプロセス2が
リソース1とリソース2の
両方を占有して計算する

```
void process_A ( void )  
{  
    down ( &resource_1 );  
    down ( &resource_2 );  
    use_both_resources ();  
    up ( &resource_2 );  
    up ( &resource_1 );  
}
```

```
void process_B ( void )  
{  
    down ( &resource_1 );  
    down ( &resource_2 );  
    use_both_resources ();  
    up ( &resource_2 );  
    up ( &resource_1 );  
}
```

Deadlock-free code

Code with a potential
deadlock

```
typedef int semaphore;  
semaphore resource_1 = 1;  
semaphore resource_2 = 1;
```

```
void process_A ( void )  
{  
    down ( &resource_1 );  
    down ( &resource_2 );  
    use_both_resources ();  
    up ( &resource_2 );  
    up ( &resource_1 );  
}
```

```
void process_B ( void )  
{  
    down ( &resource_2 );  
    down ( &resource_1 );  
    use_both_resources ();  
    up ( &resource_1 );  
    up ( &resource_2 );  
}
```

デッドロック序論(1)

□ 定義

- プロセス集合がデッドロック状態 = 集合内の各プロセスが、集合内の他のプロセスのみが引き起こしうる事象を待っている状態

□ 通常、その事象とは現在所有しているリソースの開放

□ デッドロック状態では、いかなるプロセスも

- 実行を進めることができない
- リソースを開放できない
- 起床させられない

デッドロック序論(2)

- デッドロックは以下の4つの条件が同時に成立している場合に起こる

1. 相互排除条件(Mutual exclusion condition)(前提)

- 各リソースは、1つのプロセスが保有しているか、もしくは利用可能な状態となっている

2. 保有と待ち条件(Hold-and-wait condition)(前提)

- リソースを保有しているプロセスは新たにリソースの要求ができる

3. 横取り無し条件(No-preemption condition)(前提)

- 一度保有を許可されたリソースを強制的にプロセスから横取りすることはできない

- リソースは保有しているプロセスが明示的に開放しなければならない

4. 循環待ち条件(Circular wait condition)

- 2つ以上のプロセスの循環鎖(circular chain)が存在し、循環鎖内の各プロセスは、隣のプロセスが保有しているリソースを待っている

デッドロック序論(3)

- デッドロックのモデリング
 - 有向グラフによるモデル

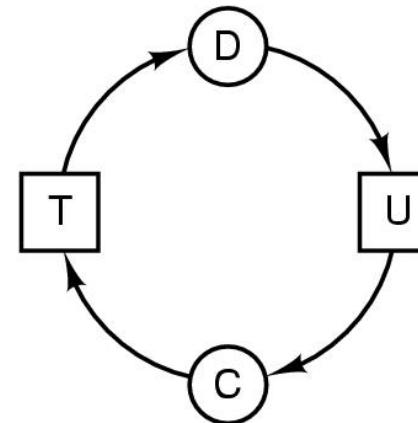
○ プロセス
□ リソース



(a)



(b)

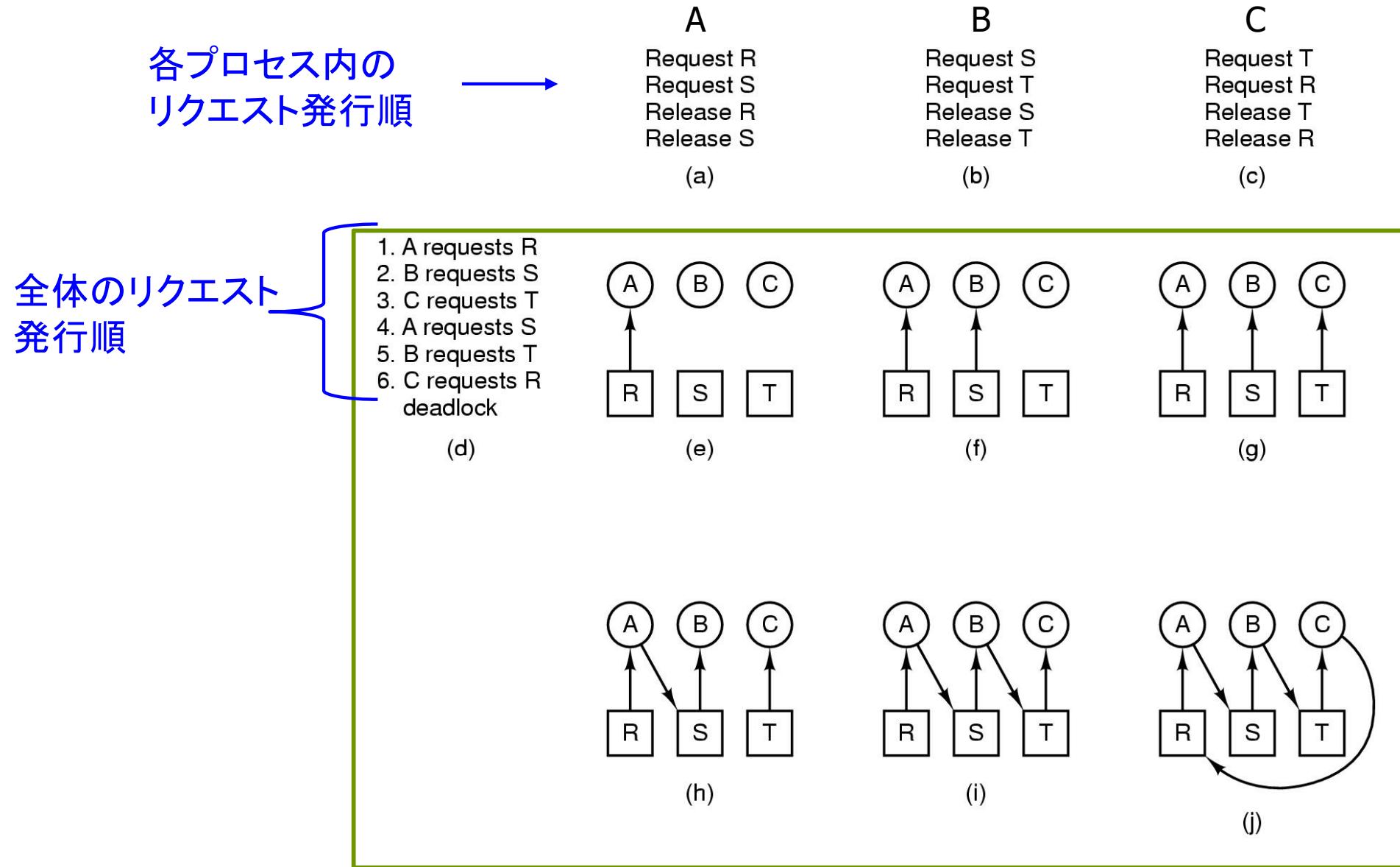


(c)

- (a) リソースRはプロセスAに割り当てられている(=プロセスAはリソースRを保有している)
- (b) プロセスBはリソースSに要求を出している(=割り当てを待っている)
- (c) プロセスCとDはリソースTとUに関してデッドロック状態となっている

デッドロック序論(4)

■ どのように推移してデッドロックが発生するか

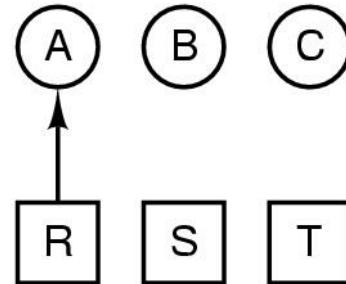


デッドロック序論(5)

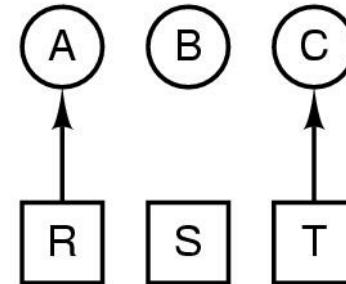
■ どのようにしてデッドロックが回避できるか

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
- no deadlock

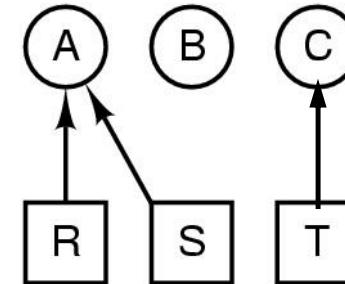
(k)



(l)



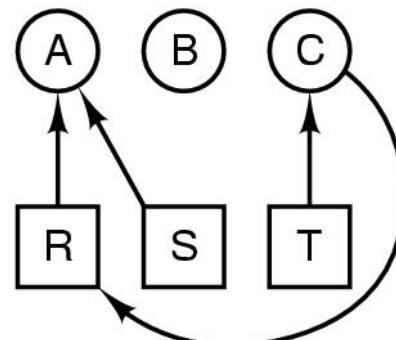
(m)



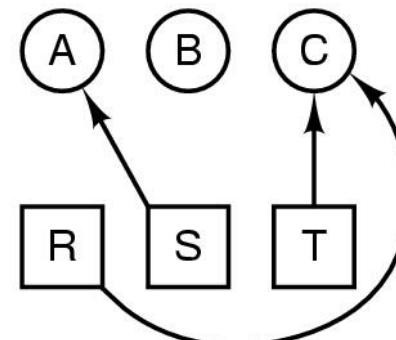
(n)

例えばプロセスBの実行を遅らせる

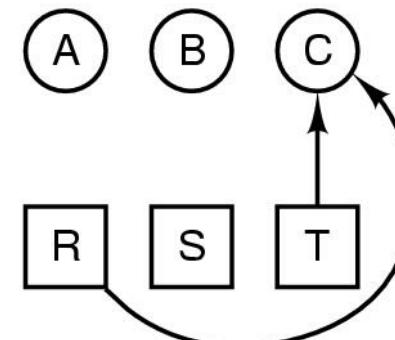
↓
この判断が可能だろうか？



(o)



(p)



(q)

デッドロック序論(6)

□ デッドロックに対する方策

1. 完全に問題を無視する — 現実逃避(ostrich)アルゴリズム

- ▣ 現状のシステムはデッドロックの可能性を無視している

2. 検出と回復(detection and recovery)

- ▣ デッドロックの発生後、それを検出して回復を試みる

3. 動的な回避(dynamic avoidance)

- ▣ デッドラインが発生しないように注意深くリソース割当てを行う

4. 防止(prevention)

- ▣ 4つの必要条件の一つを無効にする

現実逃避(Ostrich)アルゴリズム

□ 現実逃避(Ostrich)アルゴリズム

- 問題がないようなふりをする
- 以下の2つが成り立つ場合には合理的
 1. デッドロックがめったに発生しない
 2. 防止するためのコストが大きい
- UNIXやWindowsはOSレベルでこのアプローチを探る
- 理由:以下のトレードオフが重要
 - 正しさ
 - プロセスが使用(保有)可能なリソースを1つに制限すると、デッドロックが発生しないことを保証できるが…
 - 便利さ
 - プロセスは同時に複数のファイルをオープンしたり、たくさんの子プロセスをforkしたい、など

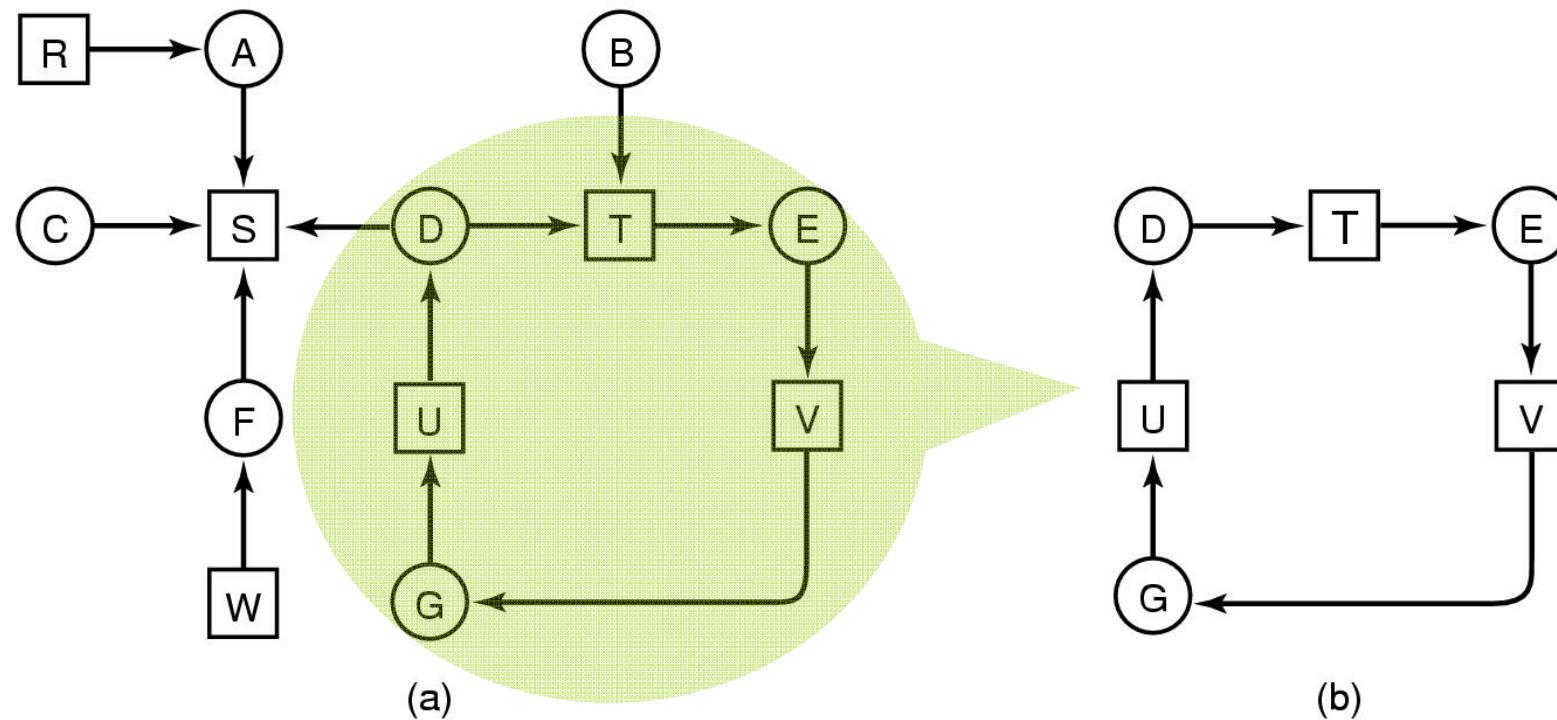
デッドロックの検出と回復(1)

- 検出と回復
 - システムがデッドロックの発生を防ぐことはしない
 - デッドロックが発生したときに、それを検出し、回復するために何らかの行動をとる
- 各タイプに1つのリソースのみ存在するときのデッドロックの検出
 - 各タイプに1つのみのリソースが存在する場合
 - 1つのスキャナ、1つのDVDレコーダー、1つのテープドライブ、など

例) 7つのプロセス(A ~ G)と6つのリソース(R~W):
⇒ 次スライド

デッドロックの検出と回復(2)

- リソースグラフ内で循環鎖が見つかった場合、デッドロックが存在



- より複雑なグラフでは、循環鎖を検出する何らかのアルゴリズムが必要となる ⇒ 次スライド

デッドロックの検出と回復(3)

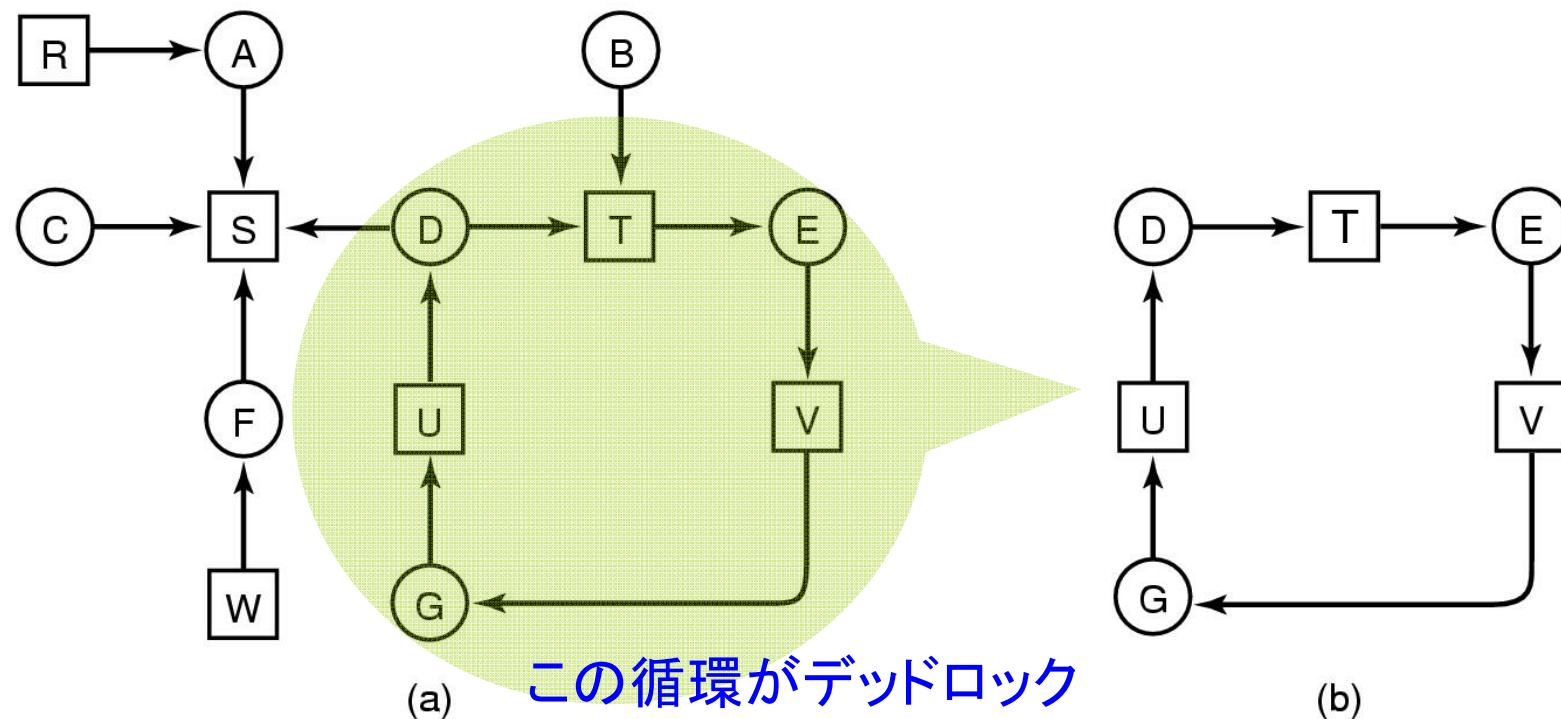
- ノードのリストを示すデータ構造 L
 - アルゴリズムの実行中, 一度調べた辺(arc)はマークされる
1. グラフ内の各ノード N について, N を開始ノードとして, 以下の5つのステップ(2 ~6)を実行する.
 2. L を空リストに初期化する. かつ全ての辺のマークを外す
 3. 現在のノードを L の最後に加え, そのノードが L 内に2回現れるかどうか調べる. もしそうならば, グラフは循環鎖を含むことになり, アルゴリズムは終了する
 4. そのノードから, マークの付加されていない出力辺があるかどうか調べる. もしあればステップ5に進み, なければステップ6に進む
 5. マークされていない辺をランダムに1つ選択し, マークを付け, その辺をたどって新たな次のノードに進み, ステップ3に戻る
 6. もしこのノードが開始のノードならば, (N を開始とする)グラフは循環鎖を含まないことになり, (N を開始とする)アルゴリズムは終了する. 開始ノードでない場合, このノードを(L から)取り除き, 1つ前のノードに戻り, ステップ4に戻る.

⇒ このアルゴリズムは深さ優先探索

デッドロックの検出と回復(4)

■ アルゴリズムを適用した例

- Rで開始 $\cdots \rightarrow L = \{R, A, S\}$ の段階で行き止まりに達する： Aに戻り、 Rに戻って終了
- ...
- Bで開始、 $\cdots \rightarrow L = \{B, T, E, V, G, U, D, T\}$ の段階で循環鎖に達する



デッドロックの検出と回復(5)

□ 各タイプに複数のリソースが存在するときのデッドロックの検出

■ 行列に基づくアルゴリズム

- n 個のプロセス ($P_1 \sim P_n$) と m 個のリソースクラス (クラス i に E_i 個のリソースが存在, $1 \leq i \leq m$) が存在する場合のデッドロックを検出

- E : 存在リソースベクトル

- 要素 E_i はクラス i の存在するリソース数

- A : 利用可能リソースベクトル

- 要素 A_i はクラス i の(現在)利用可能なリソース数

- C : 現在の割当て行列

- 要素 C_{ij} はプロセス P_i によって(現在)保有されているリソース j の数

- R : 要求行列

- 要素 R_{ij} はプロセス P_i によるリソース j への(現在の)要求数

デッドロックの検出と回復(6)

存在リソースベクトル

Resources in existence

($E_1, E_2, E_3, \dots, E_m$)

現在の割当て行列

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation
to process n

利用可能リソースベクトル

Resources available

($A_1, A_2, A_3, \dots, A_m$)

要求行列

Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

- 全てのリソースは、保有されているか、利用可能かどちらか

$$\sum_{i=1}^n C_{ij} + A_j = E_j$$

- デッドロック検出のアルゴリズムはベクトルの比較に基づく
⇒ 次スライド

デッドロックの検出と回復(7)

- ベクトル A と B の関係 $A \leq B$ は, $A_i \leq B_i (1 \leq i \leq m)$ を意味する
- 各プロセスは最初, マークされていない. アルゴリズムの過程でプロセスがマークを付加された場合, そのプロセスが終了可能であることを意味する(そのプロセスを含むデッドロックは発生しない)
- アルゴリズムが終了したとき, マークされていないプロセスはデッドロック状態になっていると認識される

1. マークの付いていないプロセスのうち, R の i 番目の行が A よりも大きくないプロセス P_i を見つける
2. そのようなプロセスが見つかった場合, C の i 番目の行を A に加え, プロセスにマークを付加し, ステップ1に戻る
 - ✿ このことは, このプロセスが終了することが可能であり, 終了時に保有しているリソースを返却することを意味する
3. そのようなプロセスが存在しない場合, アルゴリズムは終了する

デッドロックの検出と回復(8)

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Tape drives
Plotters
Scanners
Blu-rays

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

Tape drives
Plotters
Scanners
Blu-rays

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

- ステップ1で、3番目のプロセスのみが条件を満たす
- ステップ2で、3番目のプロセスはリソースを返却し、 $A = (2\ 2\ 2\ 0)$ となる
- 再びステップ1で、2番目のプロセスが条件を満たし、続いてステップ2で、 $A = (4\ 2\ 2\ 1)$ となる
- 最後に、1番目のプロセスが条件を満たし、デッドロックが発生していないことがわかる

デッドロックの検出と回復(9)

□ デッドロックからの回復

- デッドロックを検出した後、回復させてシステムを正常化する方法が必要となる
 - 横取り(preemption)による回復
 - リソースを現在の保有プロセスから一時的に横取りし、他のプロセスに与えることが可能な場合がある
 - プロセスからリソースを横取りし、他のプロセスに使用させた後、元のプロセスに戻すことが可能かどうかは、そのリソースの性質に深く依存する
 - プリンタで横取りが可能だろうか？
 - このような回復方法は、リソースの性質によって、難しいか、不可能であることが多い
- 有望な方法ではない

デッドロックの検出と回復(10)

■ ロールバックによる回復

- 定期的にプロセスのチェックポイントを取る(メモリイメージやリソース状態などをファイルに書き出す)
- デッドロックが検出された場合、必要なリソースを保有するプロセスをリソースを保有する前の過去のポイントにロールバックさせる
- 当該リソースをデッドロック状態にある他のプロセスの1つに割り当てる
→ この方法も、リソースの性質に依存する

■ プロセスの強制終了による回復

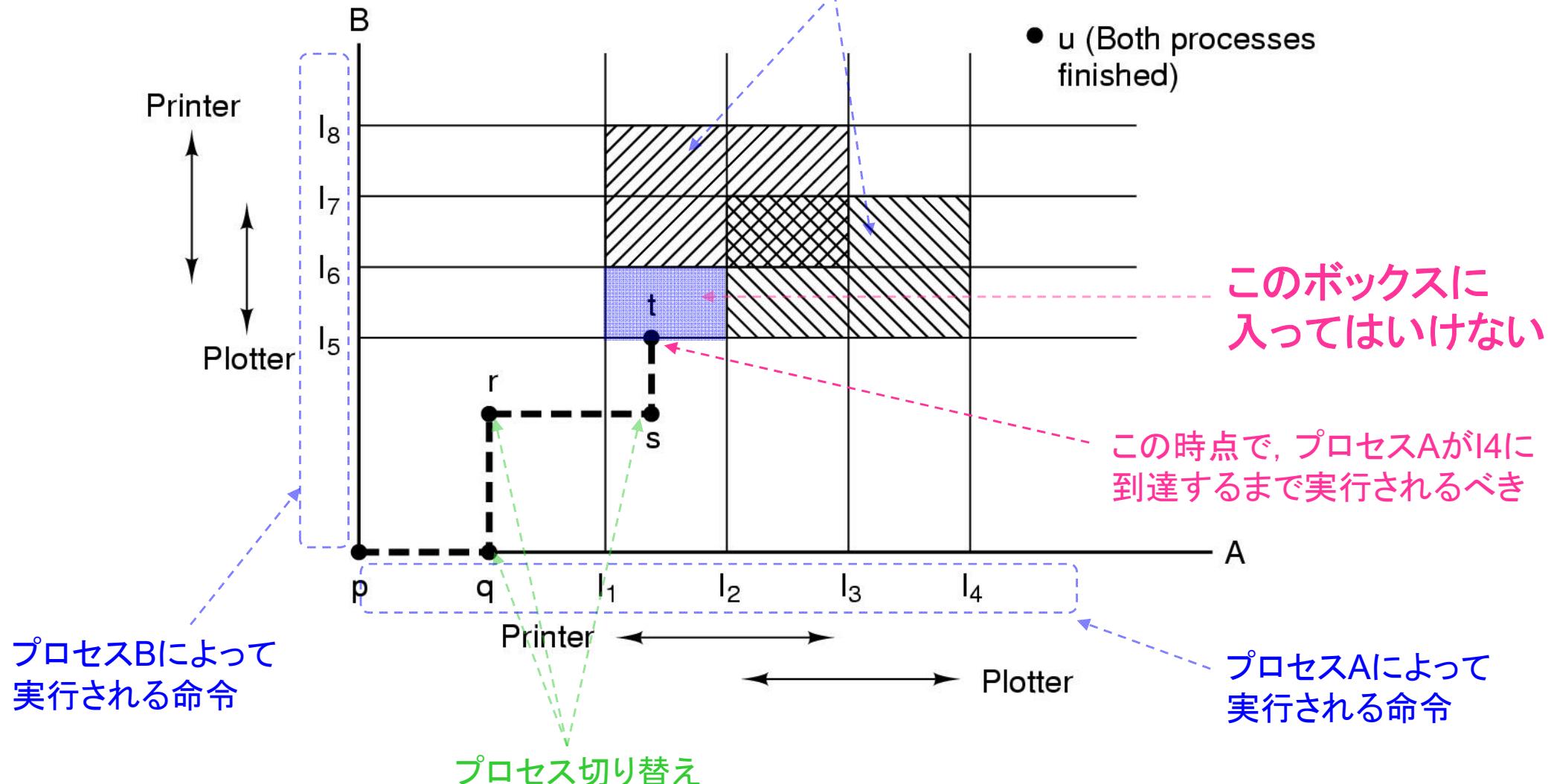
- デッドロック状態を解消する最も簡単な方法は、リソースを保有する1つ以上のプロセスを強制的に終了させること
- 終了させられたプロセスは再び最初から実行される
- しかし、プロセスを強制終了させ、やり直しをさせることは安全とは限らない。例えば、データベースを更新するプロセスは安全にやり直しができるとは限らない

⇒ デッドロックからの回復が困難であるため、次スライドからデッドロックを回避する方法を扱う

デッドロックの回避(1)

□ 準備：リソースの獲得軌道

両プロセスが同一リソースを保有することになるため、実行不可能な領域

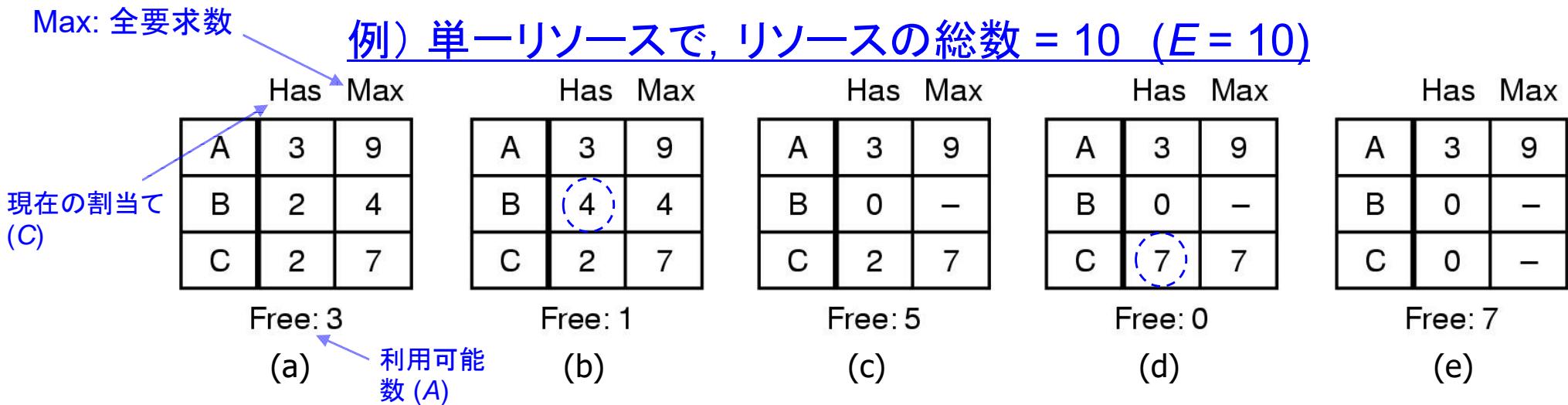


どのようにしてこのような回避が可能となるか？

デッドロックの回避(2)

□ 安全な状態と非安全な状態

- デッドロック回避のアルゴリズムは E, A, C, R を使用する
- 状態が安全(safe)とは、デッドロックが発生しておらず、かつ全てのプロセスが、たとえ突然最大数のリソース要求を行ったとしても終了できる何らかのスケジューリング順序が存在する場合である



(現在の)最大要求数
 $R = \text{Max} - \text{Has}$

(a)の状態は安全(Safe): $B \rightarrow C \rightarrow A$ の実行順で終了可能

デッドロックの回避(3)

■ 以下の例は安全ではない

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(b)

“A” がもう1つリソースを
要求して保有



この時点で、安全 → 非安全



振り返ってみると、Aの要求は満たされるべきではなかった

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4

(d)

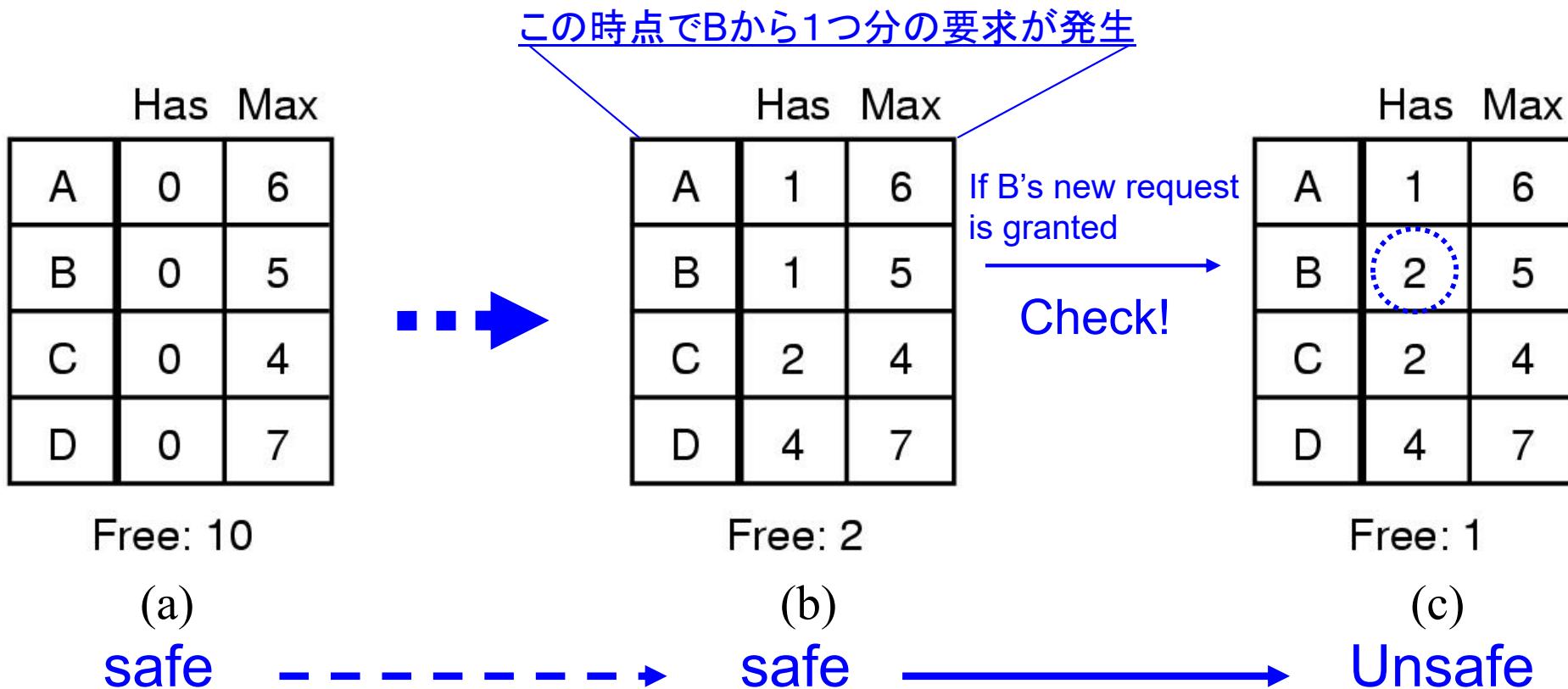
非安全な状態はデッドロック状態ではない
安全な状態と非安全な状態の違いは、安全な状態からは全てのプロセスを終了させることをシステムが保証できる一方、非安全な状態からは、そのような保証はできないことがある

デッドロックの回避(4)

□ 単一リソースの場合のBankerのアルゴリズム

- 各要求に対して、要求がなされたときに熟考する。すなわち、それを認めた場合、安全な状態になるかどうかをチェックする

- そうであれば、要求を満たす
- そうでなければ、要求を拒否する(後回しにする)



デッドロックの回避(5)

□ 複数リソースの場合のBankerのアルゴリズム

	Process	Tape drives	Plotters	Printers	Blu-rays
A	3	0	1	1	
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	0	

Resources assigned
Matrix C

	Process	Tape drives	Plotters	Printers	Blu-rays
A	1	1	0	0	
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	

Resources still assigned
Matrix R

$E = (6342)$ Existing
 $P = (5322)$ Possessed
 $A = (1020)$ Available

この例は安全な状態

⇒ アルゴリズムは次スライド

デッドロックの回避(6)

1. R の中で、必要とするリソース数が全て A 以下の行を探す。そのような行が存在しなければ、システムは非安全であり結果的にデッドロック状態になりうる
 2. 選ばれた行のプロセスが残りのリソースを全て要求して、終了すると仮定する。そのプロセスに終了の印を付け、その保有リソースを全て A ベクトルに加える
 3. ステップ1と2を、全てのプロセスに終了印が付くか、終了できるプロセスが見つからなくなるまで繰り返す。前者が、初期状態が安全な状態であった場合で、後者が非安全だった場合である
-
- この例(前のスライド)では、
 - B のスキャナに対する新たな要求を満たすと安全な状態になるだろうか？
 - その後、 E のスキャナに対する新たな要求を満たすと安全な状態になるだろうか？

デッドロックの回避(7)

- 残念ながら、このアルゴリズムは現実的ではない
 - プロセスが自身の必要とするリソースの最大数を事前に知っていることはほとんどの場合ない
 - (プロセス数は変動する)
 - (利用可能とみなされていたリソースが突然消滅しうる(故障など))
- したがって、デッドロックを回避することは非常に困難

⇒ デッドロックの回避が困難であるため、次スライドからデッドロックを防止する方法を扱う

デッドロックの防止(1)

□ デッドロック発生のための4つの条件は以下であった

1. 相互排除条件

- ・ 各リソースは、1つのプロセスが保有しているか、もしくは利用可能な状態となっている

2. 保有と待ち条件

- ・ リソースを保有しているプロセスは新たにリソースの要求ができる

3. 横取り無し条件

- ・ 一度保有を許可されたリソースを強制的にプロセスから横取りすることはできない
- ・ リソースは保有しているプロセスが明示的に開放しなければならない

4. 循環待ち条件

- ・ 2つ以上のプロセスの循環鎖(circular chain)が存在し、循環鎖内の各プロセスは、隣のプロセスが保有しているリソースを待っている

□ これらの条件のうち、少なくとも1つが満たされないようにできれば、デッドロックは構造上起こりえなくなる

デッドロックの防止(2)

□ 相互排除条件の撤廃

- もしリソースが1つのプロセスのみに割り当てられるわけでなければ、デッドロックは発生しない
- しかし、例えば2つのプロセスに同時にプリンタに出力させれば、混乱に陥ることは明らかである
- プリンタにおけるデッドロックの解決策:スプーリング
 - プリンタ出力をメモリ内にスプールすることにより、複数のプロセスが同時に出力を生成可能
 - このモデルでは、物理プリンタに実際に要求を出せる唯一のプロセスはプリンターモンである
 - デーモンは他のリソースに対して要求を出すことはないので、プリンタに関するデッドロックを無くすことができる …… これは本当だろうか？
⇒ スプーリングのための空間がデッドロックの原因となりうる。（2つのプロセスがそれぞれ巨大なファイルをスプールする場合など）
- 残念ながら、全てのデバイスがスプール可能なわけではなく、またスプーリングがデッドロックを発生させることもある

デッドロックの防止(3)

□ 保有と待ち条件の撤廃

- リソースを保有しているプロセスがさらにリソースを要求して待つことを禁止できれば、デッドロックは発生しない
- これを実現する1つの方法は、全てのプロセスに実行開始前に必要なリソースを全て要求させることである
 - 全てが利用可能な場合は、プロセスにはその全てが割り当てられ、実行して終了できる
 - 1つ以上のリソースが利用不可能な場合は、どのリソースも割り当てられず、プロセスは単に待つことになる
- このアプローチは良くない
 - 多くのプロセスは、実行するまで自身がどのくらいリソースを必要とするのかはわからない
 - もしわかっているとすれば、Bankerのアルゴリズムが使用可能
 - リソースの使用が最適とはならなくなる
 - 要求したリソースは長時間(使用されていなくとも)割り当てられた状態になる

デッドロックの防止(4)

□ 横取り無し条件の撤廃

- 有望な方法ではない
- 例) プリンタやデータベースのレコードが強制的に横取りされたら... !

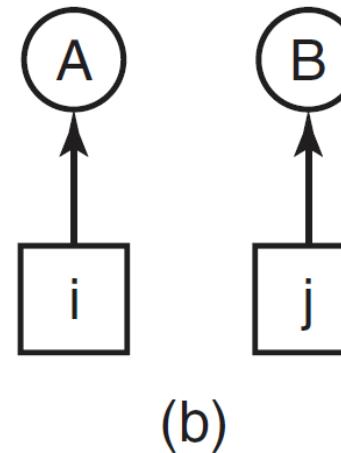
□ 循環待ち条件の撤廃

- 全てのリソースに番号付けをする方法
- 規則: プロセスは必要なときにいつでもリソースを要求できるが、全てのリクエストは番号的に順番になされなければならない

Rule: Can request
only in increasing
order

1. Imagesetter
2. Printer
3. Plotter
4. Tape drive
5. Blu-ray drive

(a)



A requests j
B request i
↓
If $i > j$, violation by A
If $i < j$, violation by B
 $i == j$, N.A.

しかしながら、たくさんのデバイス／資源を順序付けるのは極めて困難！
(特にメモリ内のデータ／オブジェクト)

ライブロック(Livelock)

- 実行は続いているが、前進(progress)できない状況

```
void process_A( void ) {
    acquire_lock( &resource_1 );
    while ( try_lock( &resource_2 ) == FAIL ) {
        release_lock( &resource_1 );
        wait_fixed_time();
        acquire_lock( &resource_1 );
    }
    use_both_resources();
    release_lock( &resource_2 );
    release_lock( &resource_1 );
}
```

```
void process_B( void ) {
    acquire_lock( &resource_2 );
    while ( try_lock( &resource_1 ) == FAIL ) {
        release_lock( &resource_2 );
        wait_fixed_time();
        acquire_lock( &resource_2 );
    }
    use_both_resources();
    release_lock( &resource_1 );
    release_lock( &resource_2 );
}
```

可能性は低いが、両プロセスが同時に同じ動作を行えば、前進できない。

飢餓状態(Starvation)

- 飢餓状態はデッドロックに似ているが、デッドロックではない
- プロセスは、要求が永遠に応えられないと餓死する
- プロセスが餓死するかどうかは、使用される応答ポリシーに依存する
 - 例えば、プリンタにおいて、最小サイズのファイルの要求から先に処理するポリシーでは、小さいサイズの要求が絶えず発生している状況では、巨大サイズを要求するプロセスは餓死するかもしれない
- 解法の1つ：
 - 到着順サービス(First-come, first-serve)によるリソース割当てポリシー

メモリ管理



概要

□ メモリ階層(memory hierarchy)

- 数十メガバイトの小容量, 高速, 高価な揮発性キャッシュメモリ
 - 数十ギガバイトの中速, 並価格な揮発性主記憶(RAM)
 - 数テラバイトの低速, 安価な不揮発性ディスクストレージ
- } OSによる
} 管理

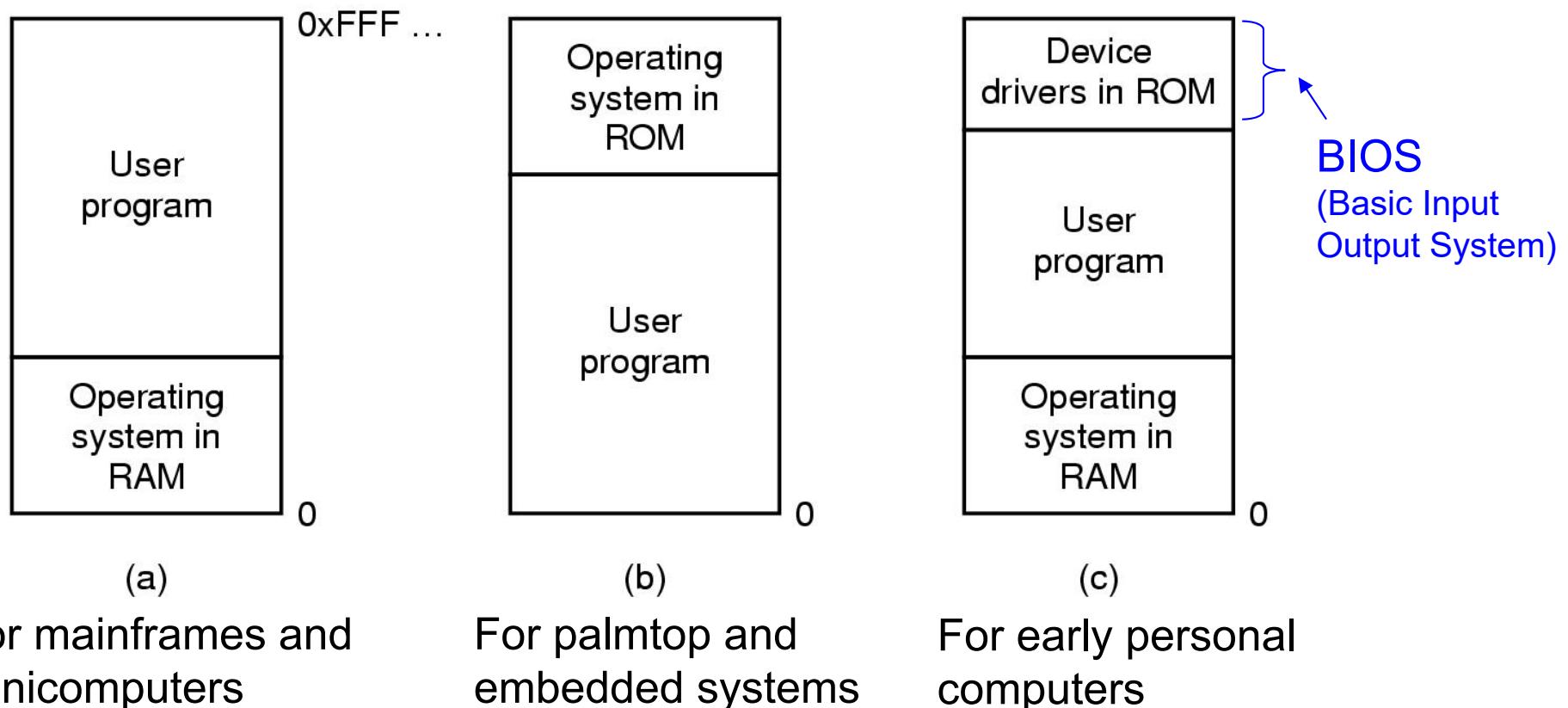
□ これらのメモリをどのように協調して使用させるかはプロセッサハードウェアとOSの仕事

- プロセスが必要とした場合にメモリを割当て, 使用後は開放する.
- これを実現するために, メモリのどの部分が使用中で, どの部分が使用されていないのかを記録する.
- 全てのプロセスを格納するために主記憶が足りない場合は, 主記憶とディスク間のスワッピングを管理する

基本的なメモリ管理(1)

■ スワッピングやページングを行わない単一プログラミング

- 一度に1つのプログラムのみを実行し、そのプログラムとOSがメモリを共有する
- プロセスが終了したとき、OSは新たなプログラムをメモリにロードする

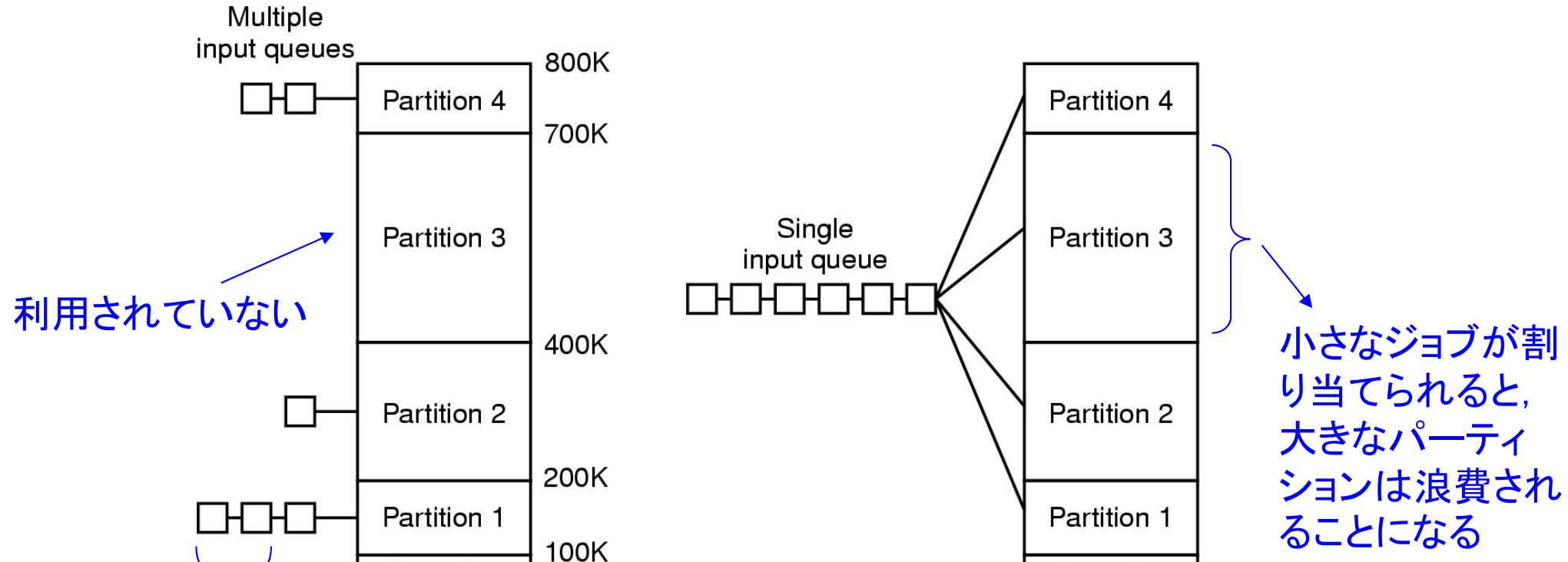


基本的なメモリ管理(2)

□ 固定パーティションを用いたマルチプログラミング

- 最近のほとんどのシステムでは、複数のプロセスが（見かけ上同時に）実行可能である
 - プロセスがI/O処理待ちでブロックした場合、他のプロセスがCPUを使用できる → CPU使用率が向上
- マルチプログラミングを実現するための最も簡単な方法は、単にメモリを n 個のパーティションに分割すること
 - ジョブが到着したとき、
 - そのジョブを保持できる最小のパーティションの入力キューに入れるか、
 - 全パーティションで共有される入力キューに入れる
→ 次スライドの図を参照
- このようなシステムはIBMの大型メインフレーム上の OS/360 で長年使用された
 - OS/MFT (Multiprogramming with a Fixed number of Tasks) と呼ばれた

基本的なメモリ管理(3)

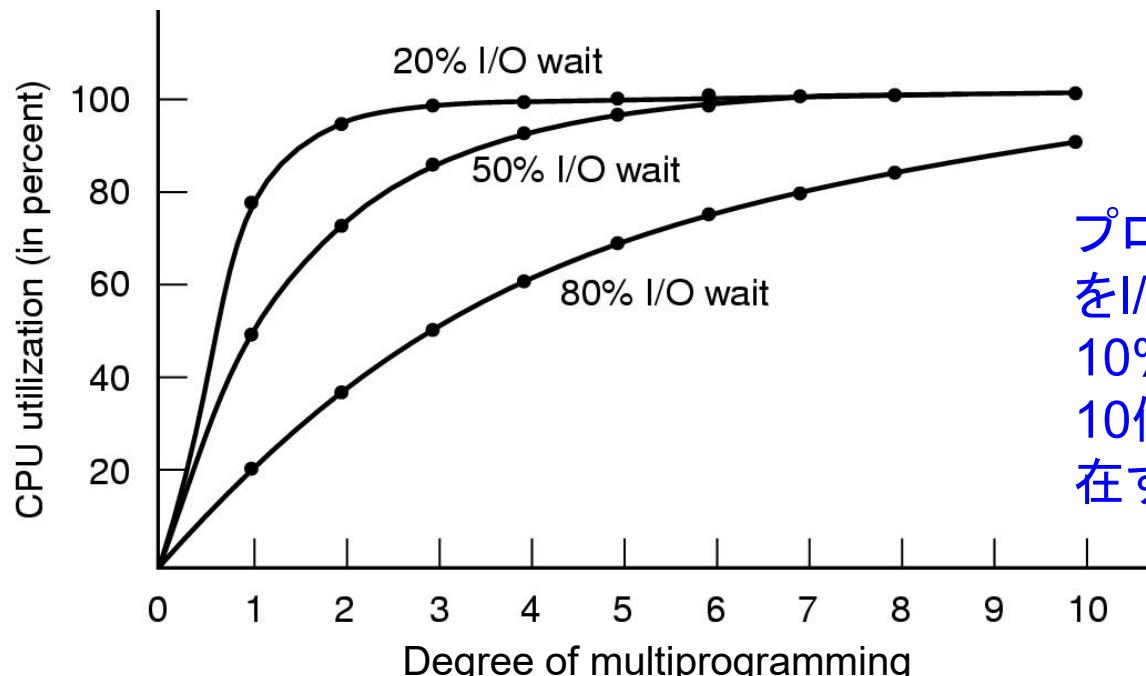


パーティション3が空であるにもかかわらず、待つ必要がある

基本的なメモリ管理(4)

□ マルチプログラミングのモデル化

- プロセスがI/Oの完了を待つのに費やす時間の割合を p とする
- n 個のプロセスがメモリ内に同時に存在する場合, n 個全てのプロセスがI/O待ちとなる確率は p^n となる
- CPU使用率は $1 - p^n$



プロセスが平均的に実行時間の 80% をI/O待ちに費やすとき, CPUの浪費を 10%以下に抑えるためには, 少なくとも 10個のプロセスが同時にメモリ内に存在する必要がある

基本的なメモリ管理(5)

□ 再配置(Relocation)

- 別々のプロセスが別々の(物理)アドレス上で実行される
 - コード(命令列)と変数群の物理アドレスは静的には不明
 - 命令読み出しのアドレス(Program Counterの値)
 - 変数へのload/store時のアドレス
 - 手続き／関数の呼び出し時の(ジャンプ先)アドレス
 - 可能な解決法の1つ
 - プログラムがメモリにロードされるときに命令を更新する
 - バイナリコード内の全てのアドレスにオフセット値を加算
 - 例) OS/MFT
- ⇒ 現在は仮想記憶技術により再配置問題を解決

基本的なメモリ管理(6)

□ 保護(Protection)

- 悪意のあるプログラムは、他のプログラムに割り当てられたメモリ内のデータを読み書きするかもしれない
- IBM 360 では、
 - メモリはサイズが2KBのブロックに分割される
 - 4ビットの保護コードが各ブロックに割り当てられる
 - PSW(プログラム状態ワード)レジスタは4ビットのキーを含む
 - → 現在実行中のプロセスのキーを意味する
 - 実行プログラムが、保護コードがPSWのキーと異なるブロックにアクセスしようとしたとき、ハードウェアは例外を発生させる
 - 保護コードとキーを変更できるのはOSのみ
- 別の解決法
 - 次スライド参照

基本的なメモリ管理(7)

■ ベース(base)レジスタ とリミット(limit)レジスタ

- プロセスがスケジュールされるとき、ベースレジスタとリミットレジスタにそれぞれ、パーティションの先頭アドレスとパーティションの長さがロードされる
- 全てのメモリアドレスはハードウェアによって自動的に生成される
 - メモリに送られる前のコード内の(相対)アドレスに、ベースレジスタの値が足される
 $\text{call } 100 \rightarrow \text{jump to } 100 + \underline{100K}$
 ↑
 ベースレジスタの値
 - 生成されたアドレスは、リミットレジスタにより、現在のパーティションの外部をアクセスしようとしているかチェックされる

$$100 + 100K < 100K + \text{limit} ?$$

- ハードウェアはユーザプログラムがこれら2つのレジスタを変更できないようにしている(特権モードでの実行のみが変更できる)

□ 例)

- CDC 6600 (the 1st supercomputer, 1964)
- Intel 8088 (without the limit register, used in the original IBM PC)

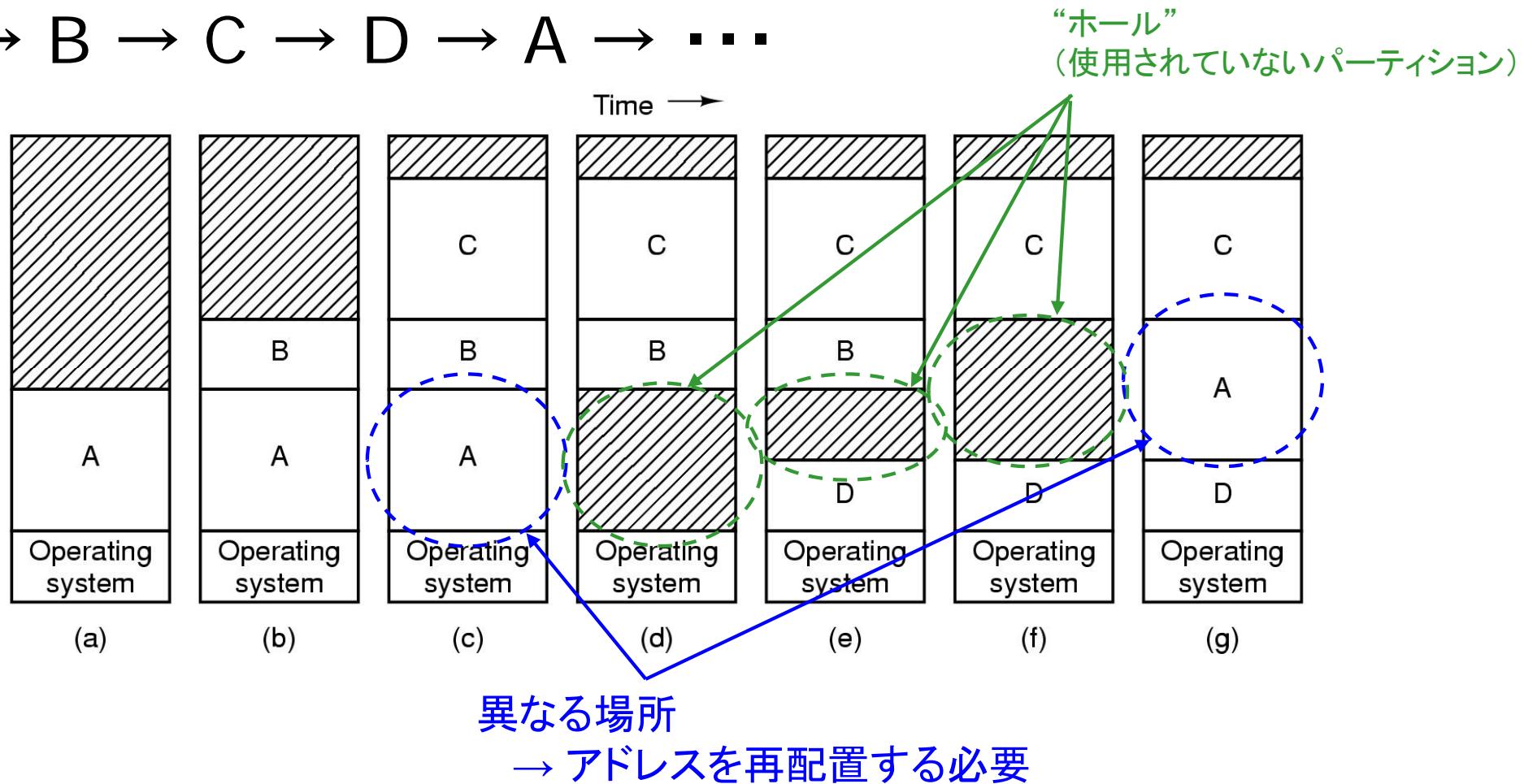
⇒ 現在は仮想記憶技術により保護問題を解決

スワッピング(1)

- 時分割システムでは、現在のアクティブなプロセスを全て保持できるだけのサイズの主記憶がない場合がほとんど
- 溢れたプロセスはディスク上に保持し、実行するときに動的にメモリにロードする必要がある
- 2つの一般的なアプローチ
 - スワッピング(Swapping)
 - 各プロセスにつき、その全体をメモリにロードし、しばらく実行し、その後ディスクに戻す
 - 仮想記憶(Virtual memory)
 - プログラムが部分的にしかメモリにロードされなくとも実行できるようにする仕組み。（同時に再配置と保護を実現する）

スワッピング(2)

- $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A \rightarrow \dots$



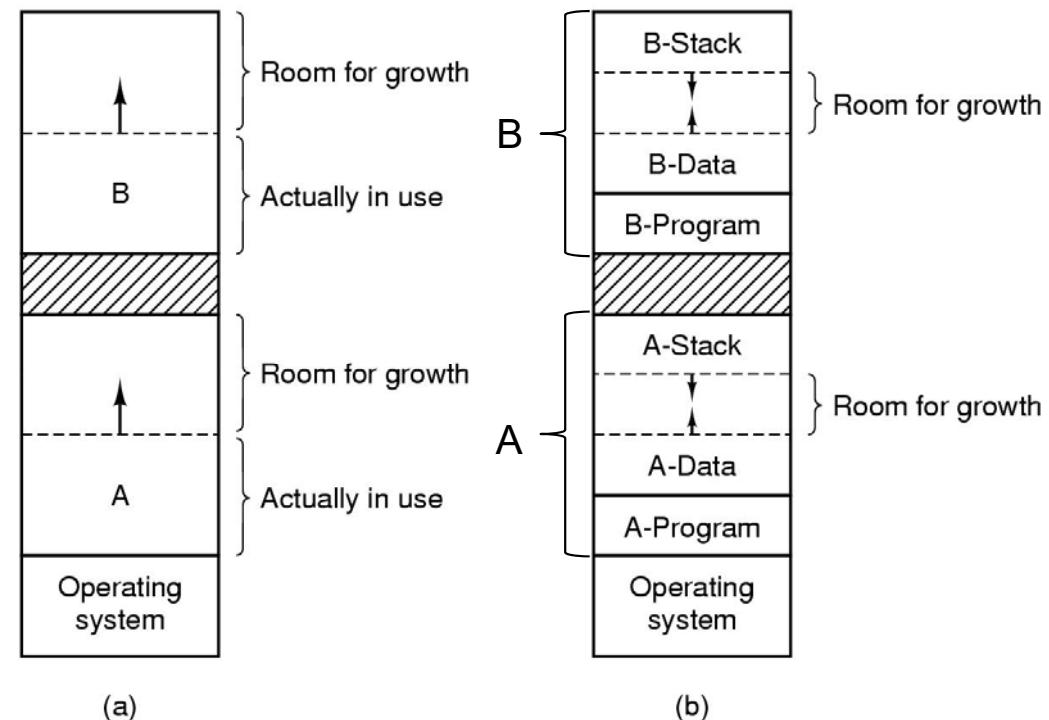
- プロセスの到着／終了にしたがい、パーティションの数、場所、サイズは動的に変化する

スワッピング(3)

- プロセスのデータ(およびスタック)セグメントが実行時に大きくなると、問題が発生する
 - 例) ヒープからの動的メモリ確保・割当て
 - ホールがプロセスに隣接している場合は割当て可能
 - プロセス同士が隣接している場合は、大きくなろうとしているプロセスは十分な大きなホールへ移動する必要がある

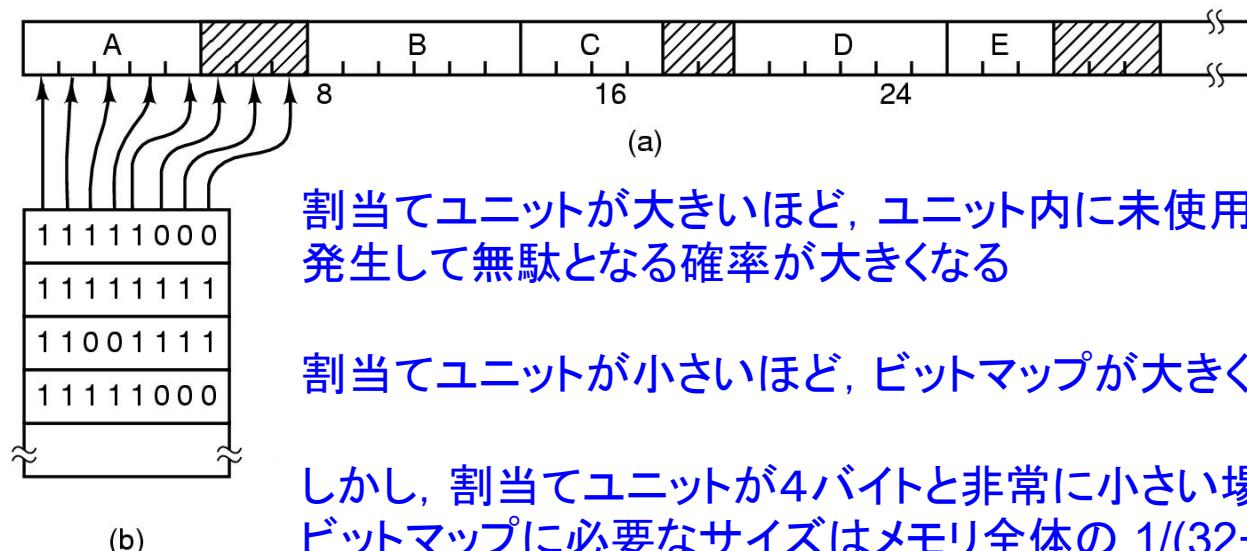
- 一方、事前に余地(Room)を割り当てておくことも可能

⇒ 現在は仮想記憶技術により動的メモリ確保の問題を解決



スワッピング(4)

- スワッピングを実現するためには、OSがメモリ内の使用領域／未使用領域を管理する必要がある
- ビットマップによるメモリ管理
 - メモリは割当てユニットに分割される
 - ビットマップ内には、各割当てユニットに対応する1ビットがある(フリーの場合は0, 使用中の場合は1)



割当てユニットが大きいほど、ユニット内に未使用領域が発生して無駄となる確率が大きくなる

割当てユニットが小さいほど、ビットマップが大きくなる

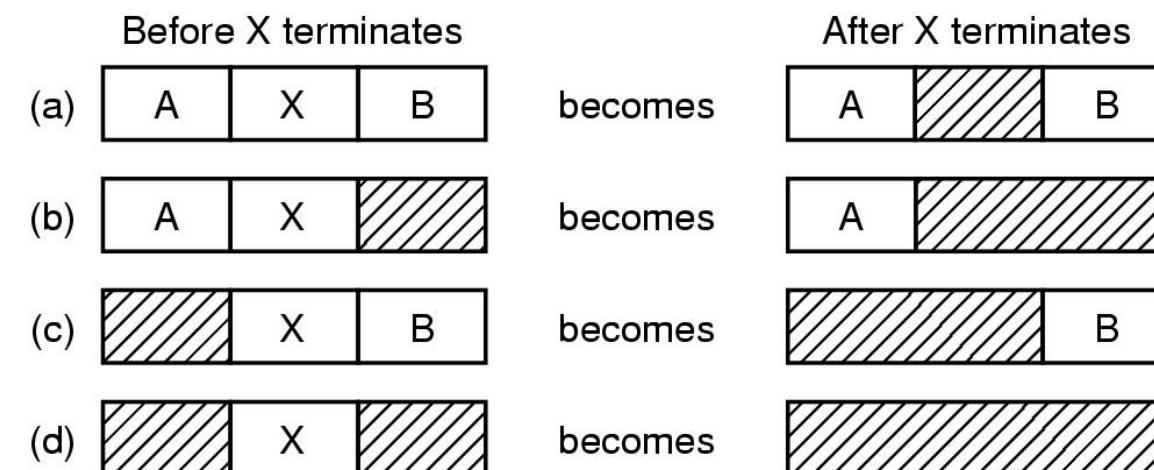
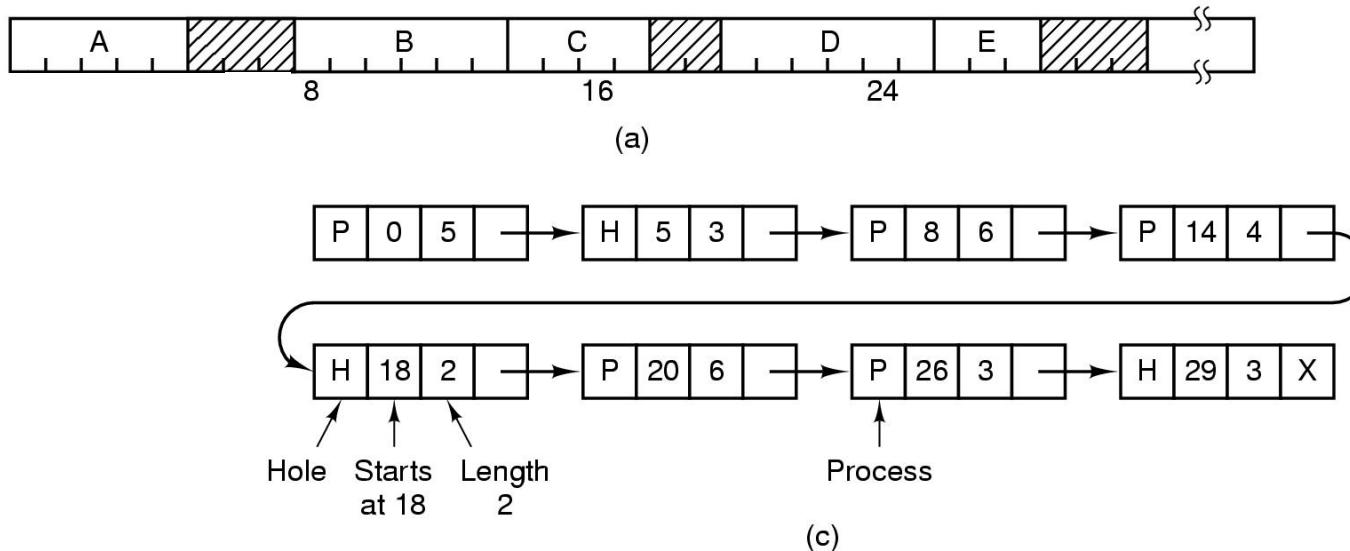
しかし、割当てユニットが4バイトと非常に小さい場合でも、ビットマップに必要なサイズはメモリ全体の $1/(32+1) = 3\%$ に過ぎない

スワッピング(5)

- ビットマップ方式の主な問題点は、 k 個の割当てユニットを使用するプロセスをメモリにロードする際、メモリマネージャーはビットマップ内の連續する k 個の “0” を見つける必要があるということである
- 必要な長さの連續する 0 をビットマップ内で探すことは、低速な処理となる
- このことが、ビットマップ方式採用への反対意見となっている
→ 連結リストによるメモリ管理(次スライド)

スワッピング(6)

□ 連結リスト(linked lists)によるメモリ管理



プロセス X が終了する際の、4つの隣接の組合せ

スワッピング(7)

□ 連結リストを使用するメモリ割当てのいくつかのアルゴリズムが存在

■ First fit

- リストにそって要求サイズ以上の大きさのホールを見つける
- その後、ホールを2つに分割する(サイズが要求したものとぴったりの場合を除く)

■ Next fit

- First fit と同様であるが、ホールを見つけたときは必ず、その場所の情報を残す
- 次にホールを探すとき、前回見つけた場所から探索を開始する

■ Best fit

- リスト全体を探索し、要求サイズを収容可能な最小のホールを見つける
- First fit/Next fit よりも低速
- シミュレーションでは、First/Next fit よりもメモリを浪費する結果となる。なぜなら、ホールを埋めるときに分割して残されるホールがその後使用するには小さすぎる傾向があるため

スワッピング(8)

■ Worst fit

- 最も大きなホールを選び、分割して残されるホールが十分使用可能なサイズとなるようにする
- 低速
- シミュレーション結果から、Worst fit も良いアイデアではないことがわかる

■ Quick fit

- あらかじめ要求サイズのリストを別々に管理する
- 例えば、
 - n エントリのテーブルを用意する。最初のエントリは 4KBのホールからなるリストの先頭へのポインタ、2番目のエントリは 8KBのホールからなるリストへのポインタ、などとする
 - プロセスの要求サイズに近いエントリを探索する
- ホールを見つけるのは高速だが、プロセス終了時に連続するホールの結合が困難

仮想記憶(Virtual Memory)(1)

□ 基本的アイデア

- 複数のプロセス全てを、限られたサイズの物理メモリ内に常駐させるのは困難な場合が多い
- また、単一のプログラム(命令コード、データ、スタックの合計サイズ)が物理メモリのサイズを超える場合もある
- OSは、プログラムの現在使用中の部分のみをメモリ内に置き、残りの部分を二次記憶(ディスク)に置く
- プログラムのそれぞれの部分はディスクとメモリとの間で必要に応じてスワップされる

□ 仮想記憶はマルチプログラミングシステムにおいてうまく働く

- スワップの際、プログラムの一部分がI/Oを通して読み込まれるのを待つ間、CPUを他のプロセスの実行に充てることが可能

仮想記憶(Virtual Memory)(2)

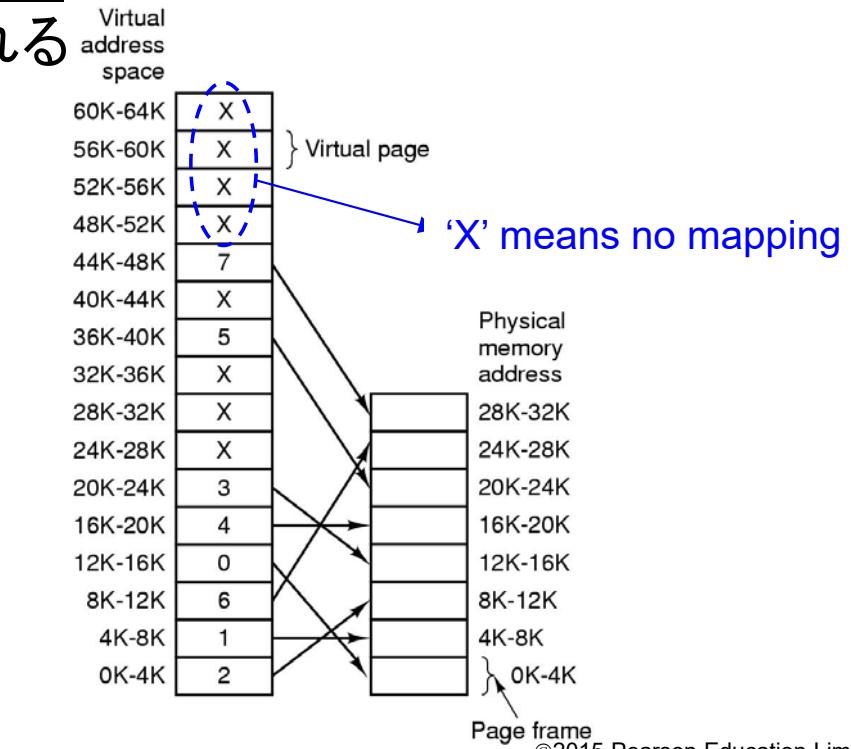
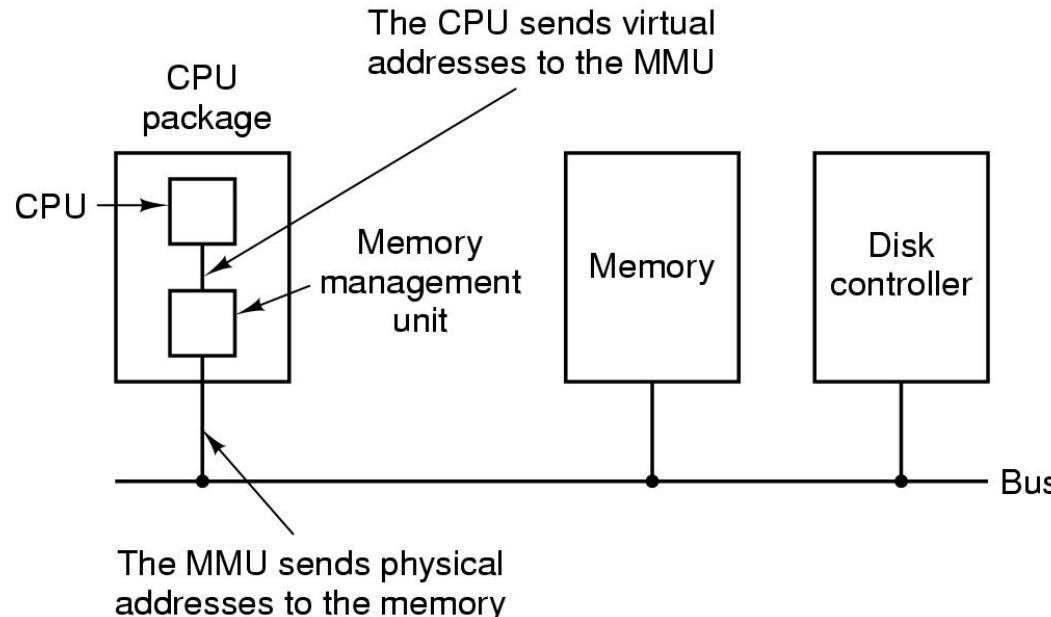
□ 仮想記憶におけるページング(Paging)

■ 仮想アドレス(Virtual addresses)

- プログラムが生成するアドレス(仮想アドレス空間内)
- 仮想アドレスは物理メモリに直接は渡されない

■ 物理アドレス(Physical addresses)

- MMU(Memory Management Unit)が仮想アドレスを物理アドレスにマップ／変換する → 物理メモリに渡される



仮想記憶(Virtual Memory)(3)

■ ページ(仮想ページ)

- 仮想アドレス空間は(仮想)ページと呼ばれる単位に分割される

■ ページフレーム(物理ページ)

- 仮想ページに対応する物理メモリ内の単位
- 仮想ページと同じサイズ

■ 物理メモリと二次記憶(ディスク)間の転送は常にページ単位で行われる

■ 例) ページサイズ:4KB, 前スライドのマッピングに従うと,

- 仮想アドレス $20,500$ ($4,096 \times 5 + 20$ = 仮想ページ5番に含まれる) は物理アドレス $4,096 \times 3 + 20 = 12,308$ にマップされる
- 仮想アドレス $32,780$ ($4,096 \times 8 + 12$ = 仮想ページ8番に含まれる) は物理アドレスにマップできず, 参照するとページFAULT(page fault)例外が発生

- CPUで例外が発生するとOSに制御が移る
- OSは空いているか, あるいはあまり使用されていないページフレームを選び, 後者の場合はその内容をディスクに書き出し, そのページフレームに今参照されたページを読み込み, マッピングを更新し, トランプを発生させた命令に制御を戻す

仮想記憶(Virtual Memory)(4)

- MMUの処理の例(16ビットの仮想アドレス, 15ビットの物理アドレス, 4KBページの場合)

- 仮想アドレス 8,196

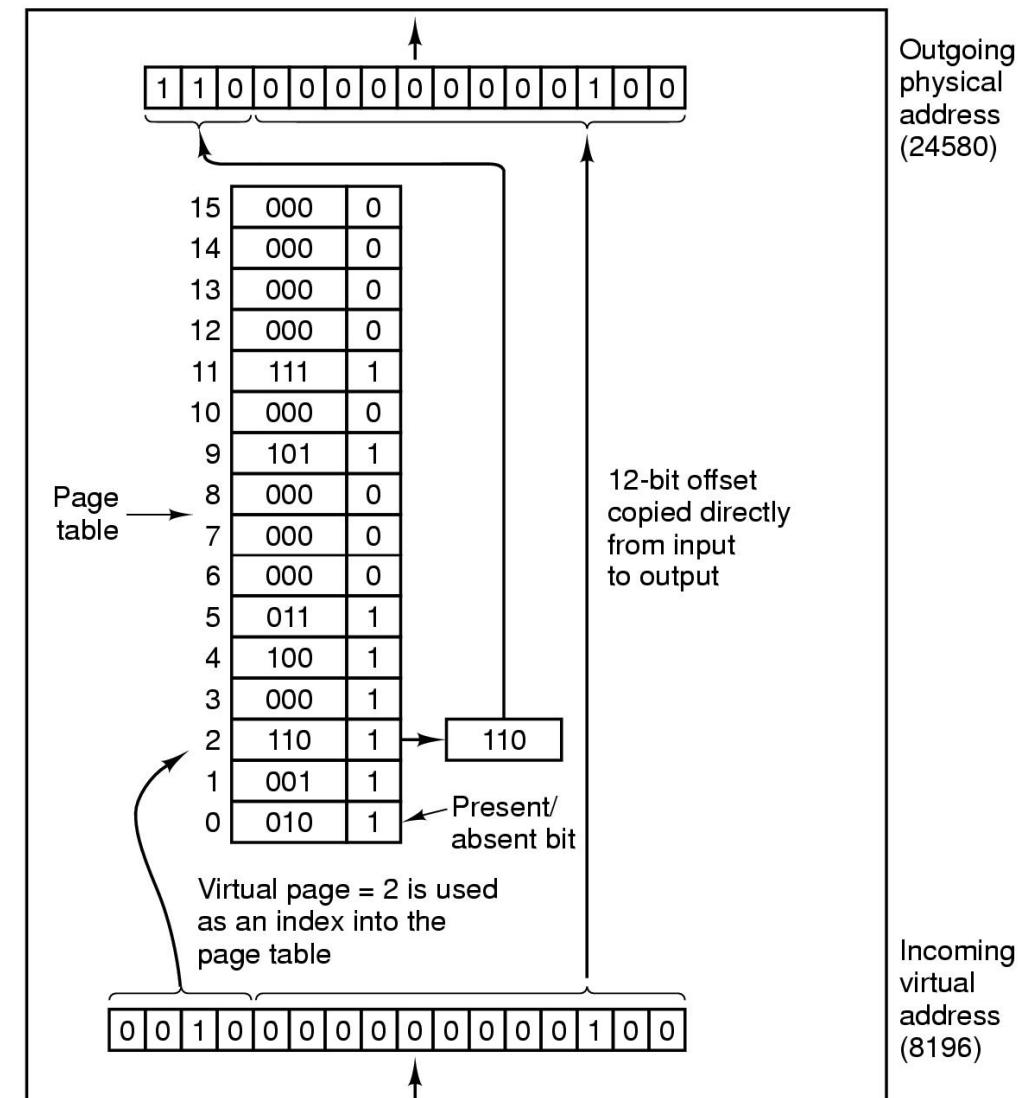
(0010 000000000100)

- 4ビットのページ番号
 - 12ビットのページ内オフセット

- ページ番号はページテーブル(page table)へのインデックスとして使用される

- 参照されたページテーブルエントリは、その仮想ページに対応するページフレームの番号を保持

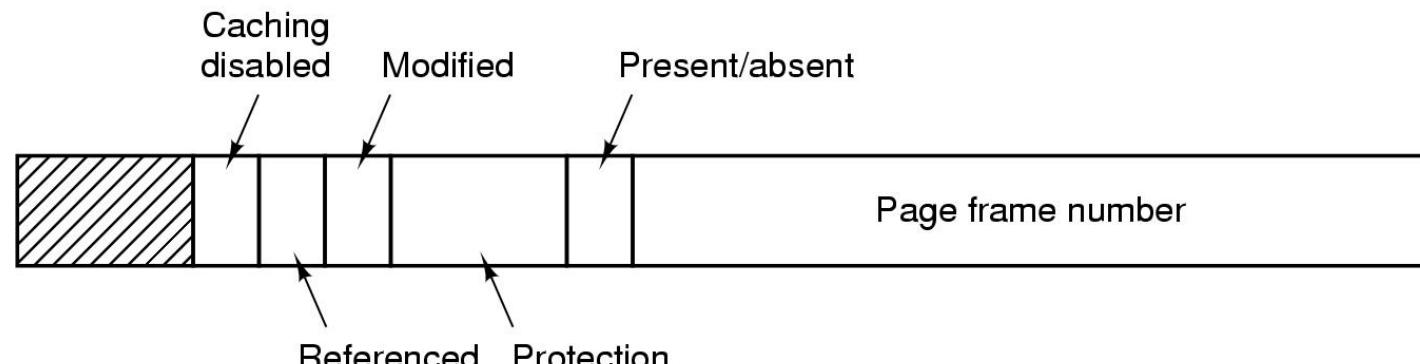
- 存在(present/absent)ビットが0の場合、ページFAULT例外が発生



仮想記憶(Virtual Memory)(5)

□ ページテーブル(Page tables)

- 仮想ページからページフレームへのマッピングを行うためのもの
- ページテーブルエントリの構成
 - エントリの正確な情報はシステム／アーキテクチャ依存
 - しかし、存在する情報の種類は異なるシステム間でもおおよそ同じ
 - ページフレーム番号(Page frame number) (物理ページ番号)
 - 存在ビット(Present/absent bit)
 - 保護(Protection)
 - 更新(Modified(dirty))
 - 参照(Referenced)
 - キャッシュ不可(Caching disabled)



仮想記憶(Virtual Memory)(6)

□ 保護(Protection)

- 許可されるアクセスの種類を示す
- 1~3ビット: 例) read/write or read only, executable or not, user or supervisor, etc.

□ 更新(Modified / dirty)

- ページが書き込まれたら、ハードウェア(あるいはOSハンドラ)がセットする
- ページ置き換えとして選ばれたページフレームは、もしこのビットが1ならば内容を二次記憶(ディスク)に書き戻す必要がある。0ならば単に上書きすればよい

□ 参照(Referenced)

- ページが参照されたときこのビットは1になる
- OSによって定期的にクリアされる
- ページFAULTが発生した場合、この値を使用してOSは追い出すページを選択
 - 最近使用されていないページが追い出し候補としてはふさわしい

□ キャッシュ不可(Caching disabled)

- このビットにより当該ページはキャッシュ不可にできる
- 例) デバイスレジスタアクセスのためのページ → キャッシュにコピーを生成しても無意味
- メモリアドレス空間とは独立したI/O空間を持つマシンにはこのビットは必要ない

仮想記憶(Virtual Memory)(7)

□ ページテーブルの2つの問題点

問題点1. ページテーブルは極端に大きくなり得る

- 最近のコンピュータは少なくとも32ビットの仮想アドレスを使用
- 4KBページを使用した場合
 - 32ビットアドレス空間では100万個のページが存在
 - 64ビットアドレス空間では非常に多くのページが存在
- ページテーブルはページ数と同数のエントリを持つ必要がある
- 各プロセスは自身の(独立した)ページテーブルを必要とする。なぜなら、プロセス各々が自身の仮想アドレス空間を持つため、したがってプロセス数分のページテーブルが存在する必要がある

問題点2. アドレス変換は高速である必要がある

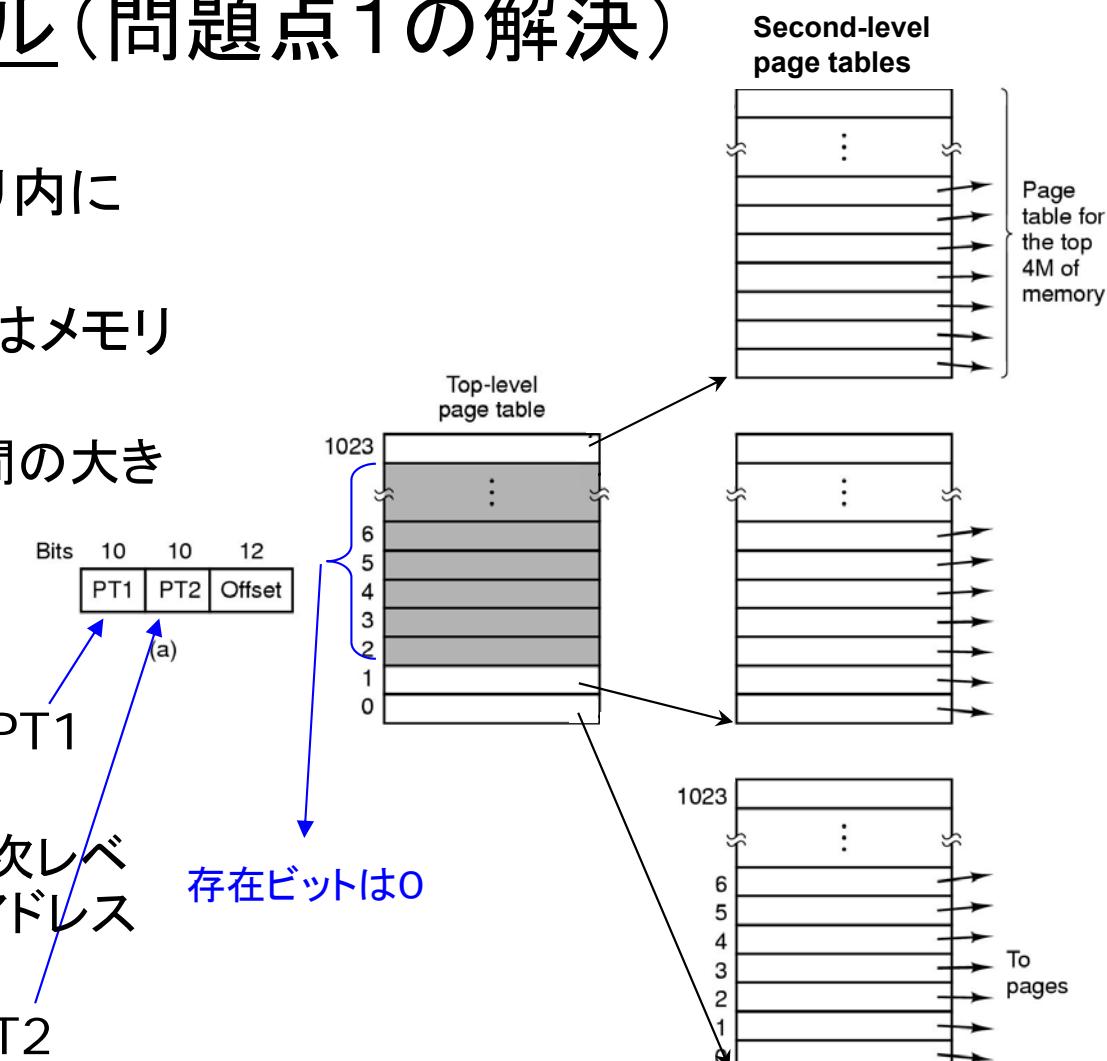
- 仮想→物理マッピングは全てのメモリ参照で行う必要がある
 - 命令ワードがメモリから読み込まれ、命令の内容にしたがって実行される
 - 命令はしばしばメモリオペランド(メモリ内データ)を使用するため、メモリの読み出し／書き込みが行われる
- 結果的に、命令毎に1回以上のページテーブル参照を行う

仮想記憶(Virtual Memory)(8)

□ マルチレベル・ページテーブル(問題点1の解決)

- 多くのコンピュータで採用
- ページテーブル全体を同時にメモリ内に置くことを避ける
- 必要としていないページのエントリはメモリ内に置いておくべきではない
 - 例) データ領域とスタック領域の大きなホールのためのエントリ群

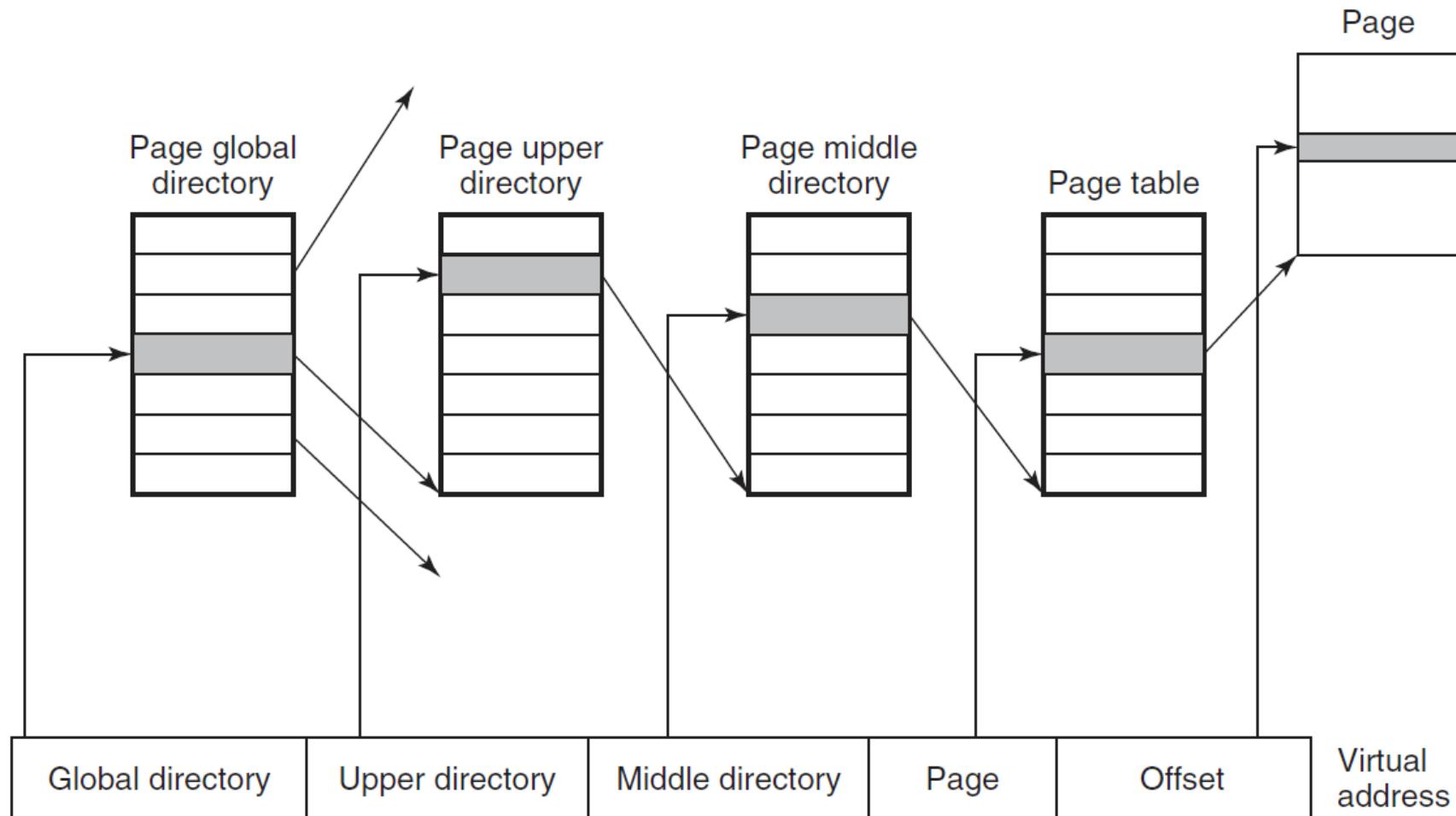
- 例) 2レベルページテーブル
 - トップレベルのページテーブルは PT1 フィールドでインデックスされる
 - インデックスされたエントリは次レベルのページテーブルの先頭アドレスを持つ
 - 第2レベルのページテーブルは PT2 フィールドでインデックスされる
 - インデックスされたエントリは対応するページフレーム番号を持つ



同様に、3レベル以上のページテーブルも可能(次スライド)

仮想記憶(Virtual Memory) (9)

■ Linuxにおける4レベルページテーブル(Version 2.6.11以降)



仮想記憶(Virtual Memory)(10)

□ TLB – Translation Lookaside Buffers

(問題点2の解決)

- ページテーブルはサイズが大きく、通常物理メモリ内に置かれる
 - ページテーブル参照のためにメモリアクセスを行うため、性能が低下
- 所見
 - 大抵のプログラムは限られた数ページを何度も参照する傾向がある
 - したがって、ページテーブルのごく限られたエントリが頻繁に読まれる；残りのエントリはまったく使用されない
- TLB: ページテーブル情報専用のキャッシュハードウェア
 - 物理メモリ内のページテーブルを参照することなしに、高速なTLBを参照することで、仮想アドレスから物理アドレスへの変換と保護のチェックを行う
 - 通常、MMU(Memory Management Unit)に含まれ、少数の(多くはない)エントリから成る(256エントリ程度まで)

仮想記憶(Virtual Memory)(11)

- TLB内の各エントリは1つのページのための情報を持つ
 - 仮想ページ番号および、その他ページテーブルエントリと同じ情報を持つ。ただし“有効(valid)”ビットと“存在(Present/absent)”ビットは意味が異なる
 - TLBエントリの有効ビットはそのTLBエントリが有効かどうかを示す
 - 一方、ページテーブルエントリの存在ビットは仮想ページがページフレームにマップされているかどうかを意味する
 - (マップされていないページのエントリはTLBには存在させないため、TLBエントリに存在ビットは必要ない。)

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

この図は参照ビットと
キャッシュ不可ビットは
省略されている

仮想記憶(Virtual Memory)(12)

■ TLBはどのように機能するか？

- 仮想アドレスがMMUに渡されると、MMUハードウェアはその仮想ページ番号のエントリがTLB内に存在するかをどうかを、全てのエントリと(一般的には同時並行に)比較することによりチェックする
 - マッチするエントリが存在し、そのアクセスが保護コードに違反しない場合は、ページフレーム番号がTLBから直接得られる
 - マッチするエントリが存在するが、保護違反が発見されると、保護違反例外を発生させる
 - TLB内にマッチする仮想アドレスが見つからなかった場合、TLBミスとなり、物理メモリに存在するページテーブルを参照する
 - この場合、TLB内のエントリを1つ選び、リプレースすることになる。追い出されるエントリの更新ビットおよび参照ビットは対応するメモリ内のページテーブルエントリにコピー・バックされる
 - そのTLBエントリに、ページテーブルから読み出したページテーブルエントリ情報を入れる

仮想記憶(Virtual Memory)(13)

- TLBミス処理はハードウェアによって行われるか、あるいはOSのソフトウェア(TLBミスハンドラ)によって行われる
 - ハードウェアによる管理の場合は、例外(OSへのトラップ)はメモリ内にページが存在しないとき(ページフォルト)、または保護違反のときのみ発生する
- OSソフトウェア(例外ハンドラ)によるTLB管理
 - 多くのRISC プロセッサ(SPARC, MIPS, Alpha, HP PA)では、TLB管理のほとんどがソフトウェアで行われる
 - TLBミスが発生したらTLBミス例外を発生させ、処理をOSにゆだねる
 - OSはTLBから(なるべく使用されていない)エントリ情報を1つ探し出し、そのエントリに参照されたページのページテーブルエントリ情報を入れ、TLBミスを発生させた命令に制御を返す
 - ハードウェアによる管理と比較すると低速
- TLBが適度に大きい(例えば64エントリ)場合は、TLBミス率は低下し、ソフトウェアによる管理でも十分効率的となる
 - MMUハードウェアを単純化することができる
 - リプレースのための複雑なアルゴリズムが使用可能となる

仮想記憶(Virtual Memory)(14)

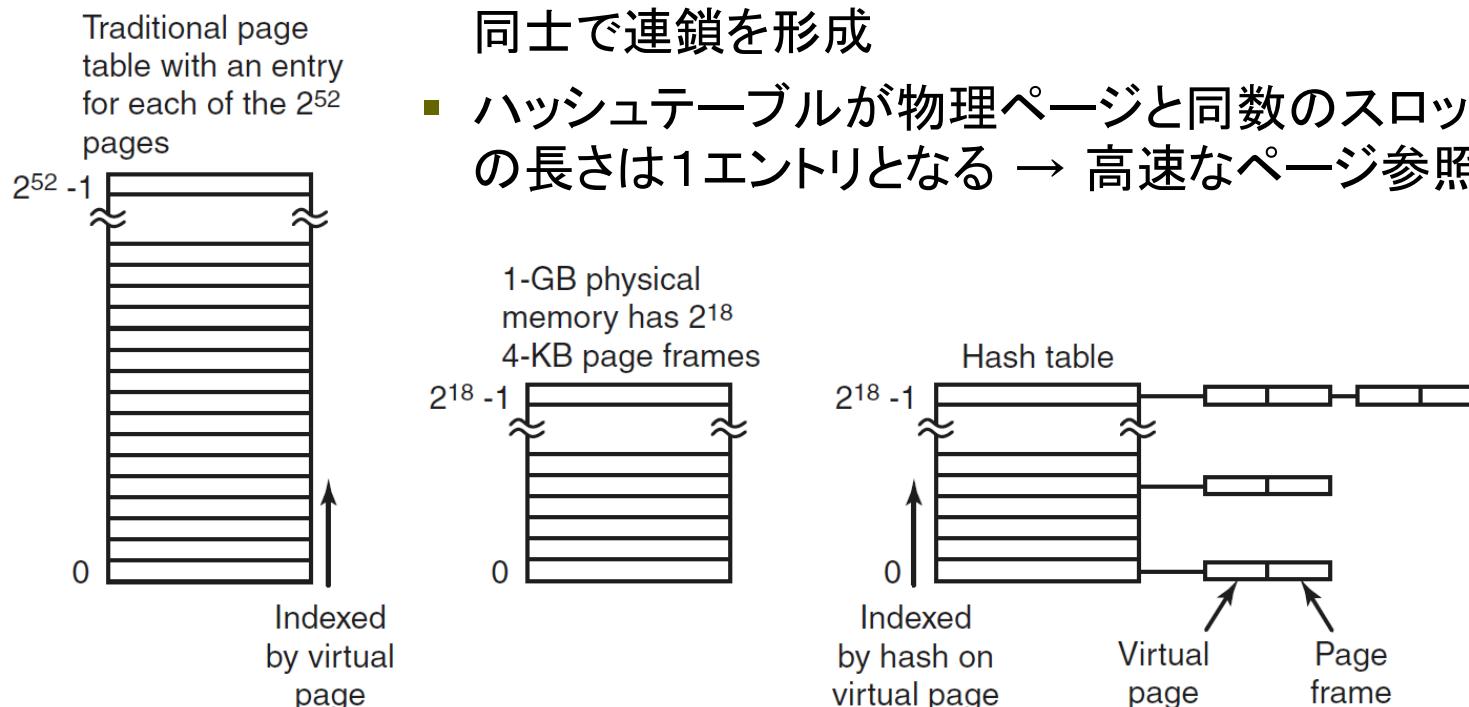
- 64ビットの計算機が主流になりつつある
 - 仮想アドレス空間が 2^{64} バイトで、ページサイズが4KBのとき、 2^{52} 個のエントリのページテーブルが必要となる
- 逆引き(Inverted)ページテーブル
 - メモリ内にページフレーム毎に1つのページテーブルエントリを存在させる(仮想ページ毎に1つエントリではない)
 - 各エントリには、どのプロセスのどの仮想ページがそのページフレームに置かれているかを記録する
 - 例) 64ビット仮想空間、ページサイズが4KB、4GBの物理メモリのとき、逆引きページテーブルに必要なエントリ数は 1,048,576 だけとなる
 - PowerPC, Ultra SPARC, Itaniumなどでこの方式が使われた

仮想記憶(Virtual Memory)(15)

- 逆引きページテーブルを使用すると、仮想アドレスから物理アドレスへの変換が著しく困難(低速)になる

- TLBでヒットする限りは問題とならない
- TLBミスの場合は、逆引きページテーブルを検索しなければならない
- この検索を行う現実的な方法として、仮想アドレスでハッシュされたハッシュテーブルを使用することが挙げられる

- 現在メモリ内に存在する全ての仮想ページは、同じハッシュ値を持つもの同士で連鎖を形成
- ハッシュテーブルが物理ページと同数のスロットを持つ場合は、鎖の平均の長さは1エントリとなる → 高速なページ参照が可能



ページ置き換えアルゴリズム(1)

- ページフォルトの発生時に、置き換えるページをランダムに選ぶ選択肢もあるが、できれば頻繁には使用されていないページを選ぶほうがシステム性能が良い
 - もし頻繁に使用されているページを削除してしまったら、そのページはまたすぐに読み戻されることになり、余分なオーバヘッドとなる
- 置き換えアルゴリズムは重要
 - “ページ置き換え”と同様の問題はコンピュータ設計の他の領域でも起こる
 - キャッシュメモリ(メモリブロックが対象)
 - ウェブサーバ(ウェブページが対象)(通常はクリーン(clean)のみであるが)

ページ置き換えアルゴリズム(2)

□ 最適ページ置き換えアルゴリズム

- ページフォルトが発生した瞬間、メモリ内にはあるページ集合が存在
- 集合内の各ページは、そのページがその後最初に参照されるまでに実行される命令数によってラベル付けできる(実現可能性は無視)
 - あるページが次の命令で参照されるとき、そのページのラベルは“1”
 - 200命令後初めて参照されるとき、そのページのラベルは“200”
- ページフォルト時、最も大きいラベルを持つページを削除すべき
 - ページフォルトの発生をできるだけ遠い未来にする
- 説明は簡単だが、実現不可能
 - 例えばシミュレーションによってプログラムを事前に実行して、ページ参照のトレースを残しておくようなことをしない限り、各ページがいつ参照されるかを知る手段は無い
 - しかし、実際のシステムでは入力パターンは様々であり、実行パスは入力によって変動するため、事前の情報は役に立たないことが多い
- 実現は不可能だが、実際の(実現可能な)ページ置き換えアルゴリズムを評価する際の比較対象として有益

ページ置き換えアルゴリズム(3)

□ NRU(Not Recently Used) ページ置き換えアルゴリズム

- ページの使用に関する有益な統計情報を収集するために、ページテーブルエントリ内に2つの状態ビットがある
 - R ：参照(referenced) (by read or write)
 - このビットはOSによって定期的にクリアされ、最近参照されたものとされていないものを区別する
 - M ：更新(modified) (by write)
- ページフォルトが発生したとき、OSは各ページを R と M にしたがって以下の4つに分類して、追い出しへページを決定する
 - クラス0：参照も更新もされていない
 - クラス1：参照されていないが、更新されている
 - クラス2：参照されたが、更新されていない
 - クラス3：参照かつ更新されている
- NRUアルゴリズムは低いクラスからランダムにページを選択して追い出す

ページ置き換えアルゴリズム(4)

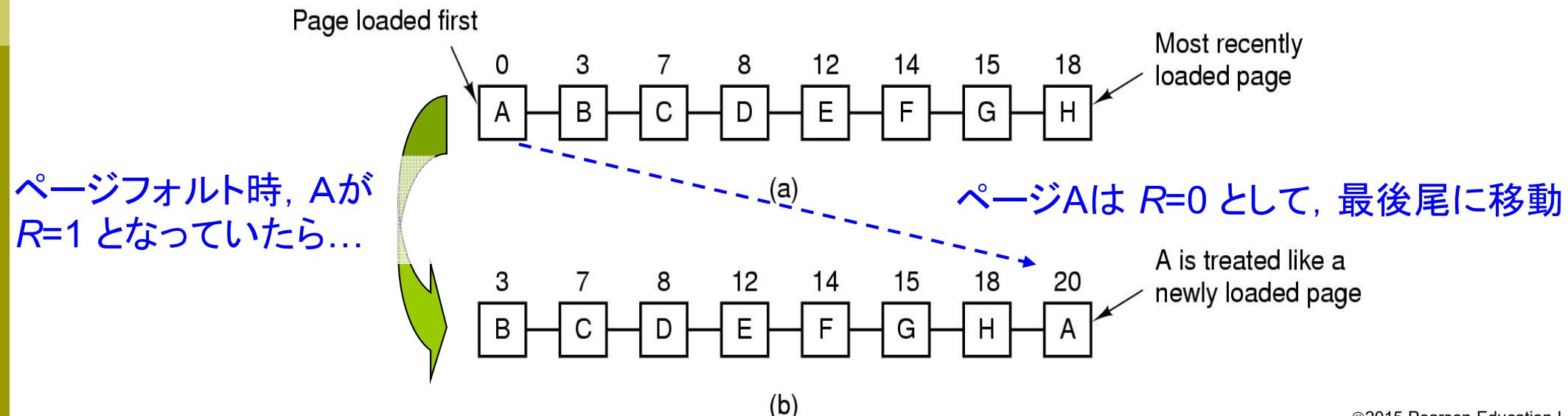
□ FIFO(First-In, First-Out)ページ置き換えアルゴリズム

- OSは現在メモリ内にある全てのページのリストを保持
 - リストの先頭にあるページが最も古くメモリに入れられたページであり、最後尾にあるページが最も最近にメモリに入れられたページ
- ページフォルトの際、先頭のページを削除し、新たなページをリストの最後尾に追加する
- 頻繁に何度も参照されるページが削除されることもあるため、このアルゴリズムをそのまま使用することはほとんどない
→ 次スライドの改良バージョンがより良い

ページ置き換えアルゴリズム(5)

□ セカンドチャンスページ置き換えアルゴリズム

- FIFO方式の簡単な改良
- もし、先頭ページのRビットが0ならば、そのページは古く、かつ参照されていないため、すぐに置き換え、FIFOを再構成する
- もし、Rビットが1ならば、ビットをクリアし、そのページをリストの最後尾に移動する。（すなわち、たった今メモリ内に入れられたかのように時間を変更したことになる。）その後、リストの次のページに対して検索を続ける

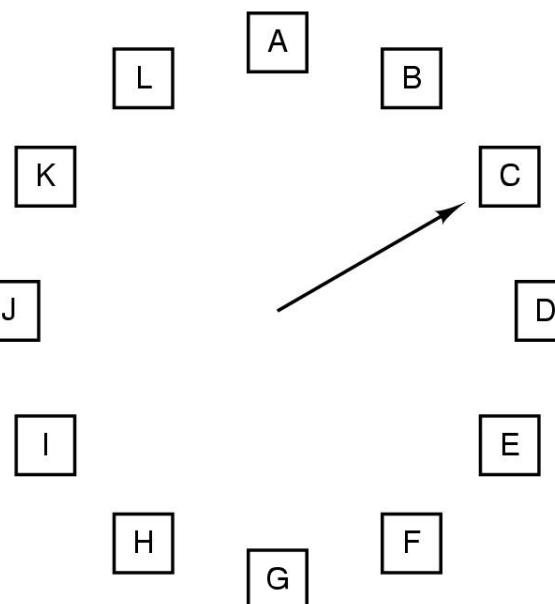


ページ置き換えアルゴリズム(6)

- セカンドチャンス方式は理にかなっているが、絶えずリスト内でページを移動させるため効率が良くない

□ クロックページ置き換えアルゴリズム

- 全てのページフレームを環状リストで管理
- ページフォルト発生時には、針(hand)で指されているページを調べる
- 操作はセカンドチャンス方式とほとんど同じ
 - 実装方法の違いでしかない



When a page fault occurs,
the page the hand is
pointing to is inspected.
The action taken depends
on the R bit:

R = 0: Evict the page
R = 1: Clear R and advance hand
+ continuing the inspection
+ advance hand to point to head

ページ置き換えアルゴリズム(7)

□ LRU(Least Recently Used) ページ置き換えアルゴリズム

■ 所見

- しばらく使用されなかったページは、おそらく今後もしばらく使用されないままであろう

■ ページフォルト発生時、最も長時間使用されていないページを置き換え対象とする

■ 実現可能だが容易・低コストではない

- 完全な実現はメモリ内の全ページのリンクリスト(例えば、最も最近使用されたページを先頭とし、最も長く使用されていないページを最後尾とする)を管理する必要がある

- メモリ参照毎にリストを更新する必要がある

- 参照されたページを見つけ、リストから削除し、先頭に移動させる

■ 簡単な実現 → 次スライド

ページ置き換えアルゴリズム(8)

■ LRUの最も単純な実現方法

- 命令実行毎に自動的にカウントアップする(例えば64ビットの)ハードウェアカウンタ(C)を用意する
- 各ページテーブルエントリも、カウンタ値を入れるフィールドを持つ
- 各メモリ参照後に、 C の現在の値を対応するページテーブルエントリに格納する
- ページフォルト発生時、OSはページテーブル内の全てのカウンタ値を調べて、最も古いものを見つける

- ページフォルトが発生する度に、最も古いページを見つけるために全てのページテーブルエントリを走査するオーバヘッドが存在する
- また、比較的大きいカウンタが必要であり、その値を格納するフィールドのメモリオーバヘッドも存在するため、現実的ではない

ページ置き換えアルゴリズム(9)

□ ソフトウェアによるLRUのシミュレーション

■ NFU(Not Frequently Used)アルゴリズム

- 各ページにソフトウェアによる(適当なサイズの)カウンタを関連付ける(初期値は0)
- 周期(ティック)割込み毎に, OSは全ページをスキャンする
- 各ページについて, Rビットの値(0か1)をカウンタに加算する
 - カウンタは各ページが参照された頻度を記録しようとしている
- ページフォルト発生時, 最小のカウンタ値を持つページを置き換え対象とする

■ NFUの問題点は何も忘れないということ

- ずっと昔に頻繁に参照されていたページは, 不必要になった後でも生き残る可能性がある
- 簡単な改良でこの問題点を解決可能 ⇒ 次スライド

ページ置き換えアルゴリズム(10)

■ 簡単な改良でLRUをシミュレート可能: + エージング(Aging)

□ Rビット値を加算する際

- まず、カウンタ値を右に1ビットシフトする
- それから、Rの値を最左ビットに加える

⇒ エージング(Aging)

	R bits for pages 0-5, clock tick 0	R bits for pages 0-5, clock tick 1	R bits for pages 0-5, clock tick 2	R bits for pages 0-5, clock tick 3	R bits for pages 0-5, clock tick 4	
Page	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0	厳密には、これはLRUと同じではない
0	10000000	11000000	11100000	11110000	01111000	↓ ティック毎に粗い記録 (1ビット)
1	00000000	10000000	11000000	01100000	10110000	例) (d)のページ0と4では、どちらへの参照がより最近かわからない
2	10000000	01000000	00100000	00100000	10001000	
3	00000000	00000000	10000000	01000000	00100000	
4	10000000	11000000	01100000	10110000	01011000	
5	10000000	01000000	10100000	01010000	00101000	また、カウンタのサイズは有限であり、全てを記憶してはいない

(a) (b) (c) (d) (e)

ページ置き換えアルゴリズム(11)

□ ワーキングセットページ置き換えアルゴリズム

■ 用語

□ 参照の局所性(**Locality of reference**)

- いかなる実行フェーズのときでも、プロセスは比較的限られたページのみ参照している

□ ワーキングセット(**Working set**)

- プロセスが現在使用中のページの集合

Skip □ デマンドページング(**Demand paging**)

- ページは要求されたときに始めてロードされる。事前にロードされることはない



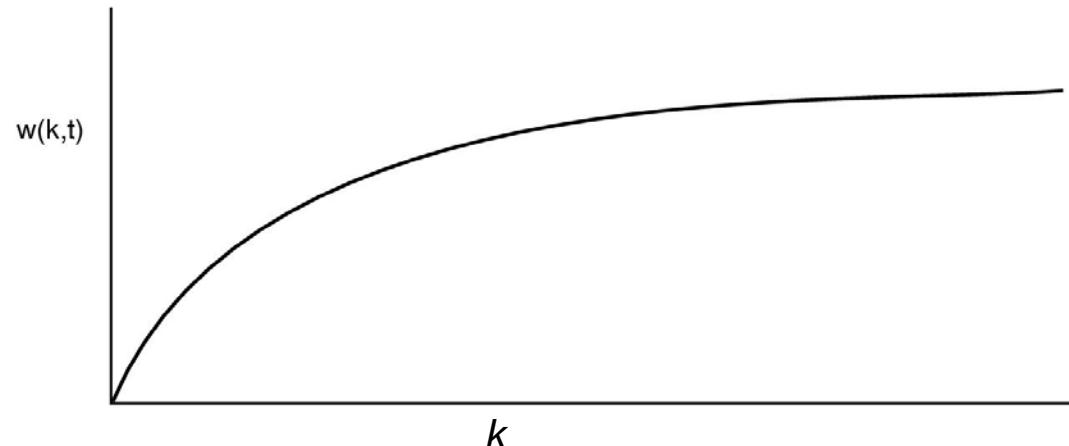
□ プリページング(**Prepaging**)

- マルチプログラミング環境下の多くのページングシステムにおいて、各プロセスのワーキングセットを記録し、中断したプロセスを再び実行するときにメモリにそのワーキングセットを用意しようとする → ページフォルト率の低減

ページ置き換えアルゴリズム(12)

■ ワーキングセット: $w(k, t)$

- いかなる時刻 t でも、最近の k 回の参照で使用されたページの集合が存在する
- $w(k, t)$ は k に関して単調非減少関数
- $w(k, t)$ は有限(最大でも、全アドレス空間)



- ワーキングセットは時の経過とともにゆっくりと変化し、その内容は k に左右されにくい(ただし k が小さいときは変化が大きい)
- プリページングでは、プログラムが最後に停止させられたときのワーキングセットをロードする

ページ置き換えアルゴリズム(13)

- ワーキングセットページ置き換えアルゴリズム
 - ページFAULT発生時, ワーキングセットに含まれないページを置き換え対象とする
- ワーキングセットモデルを実現するためには, どのページがワーキングセット内にあるかをOSが記録しておく必要がある
 - 正確なワーキングセットの記録を残すことは困難
- 近似的なワーキングセットの情報をどのように得ることができるか?
 - ワーキングセットの再定義: 例えば, 過去100ミリ秒($= \tau$)間の実行で使用されたページの集合
 - ページテーブル内の参照(R)ビットが使用可能
 - 加えて, 各ページテーブルエントリに, 最後に使用されたおよそその時刻を含める

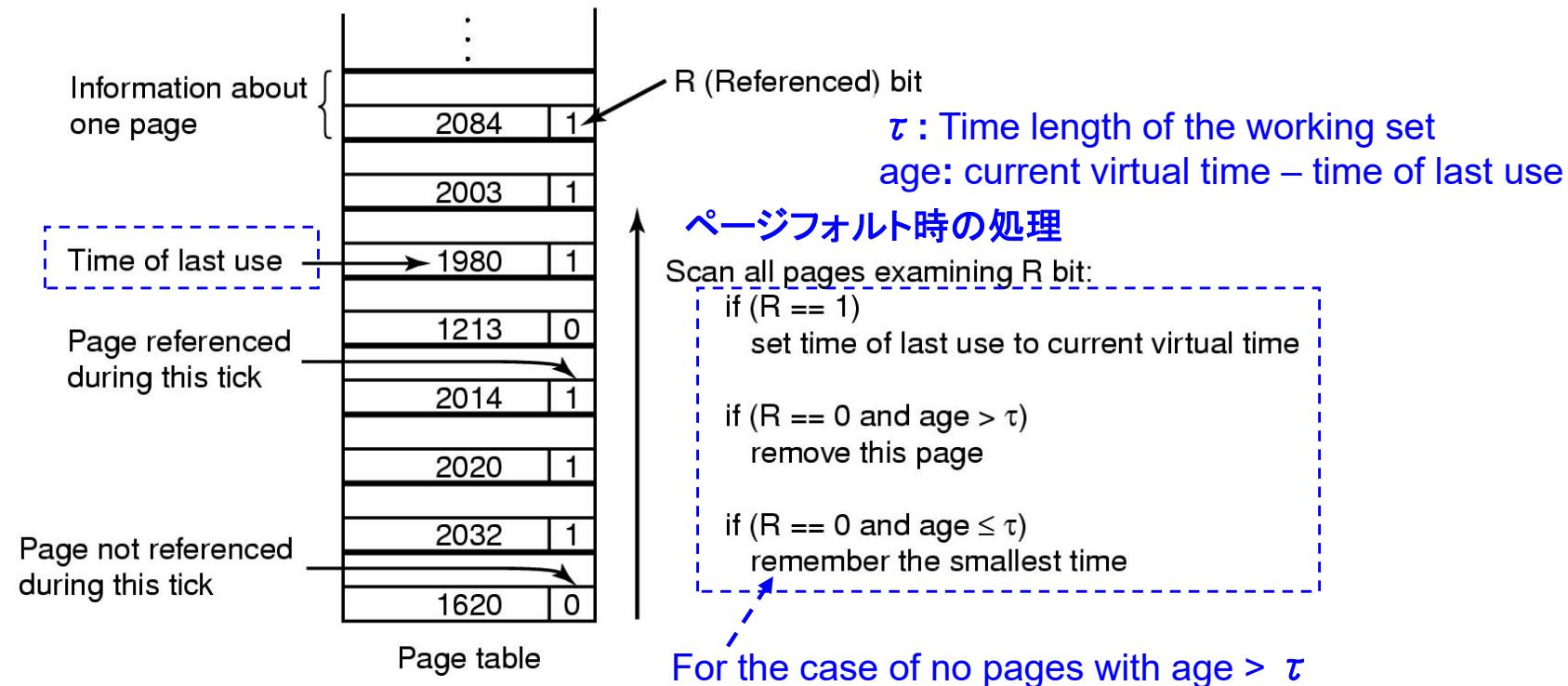
ページ置き換えアルゴリズム(14)

■ アルゴリズム

- ハードウェアによって R および M ビットがセットされる(前述のとおり)
- 周期(ティック)割込みで起動されるソフトウェア(周期ハンドラ)が、 $R=1$ のページは時刻を更新し、 R ビットをクリアする。
- ページフォルトの発生毎に、ページテーブルをスキャンし、置き換えにふさわしいページを探す(以下の図を参照)

2204

Current virtual time



ページ置き換えアルゴリズム(15)

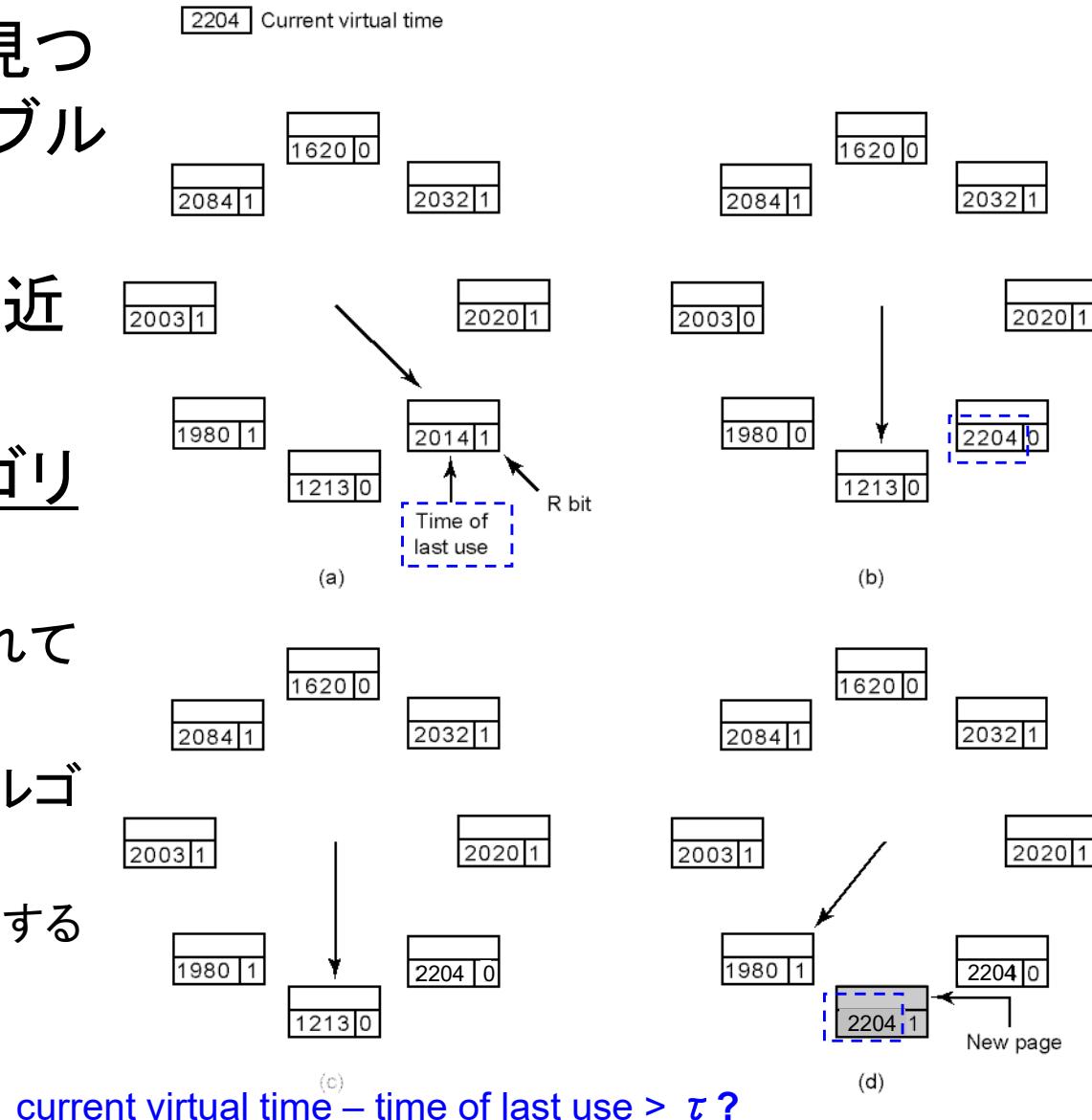
□ ワーキングセットアルゴリズムは、ワーキングセット外のページが見つからなかった場合にページテーブル全体をスキャンする必要がある

□ リプレースを繰り返すと、先頭に近いページ群は新しい傾向がある

□ WSClockページ置き換えアルゴリズム

- 実現が容易で、実際に広く使用されている改良版アルゴリズム
- クロック方式でワーキングセットアルゴリズムを実装
 - ページフォルト毎に、先頭からスキャンするのは非効率であるため

→ より詳しくは教科書参照



ページ置き換えアルゴリズム(16)

□ ページ置き換えアルゴリズムのまとめ

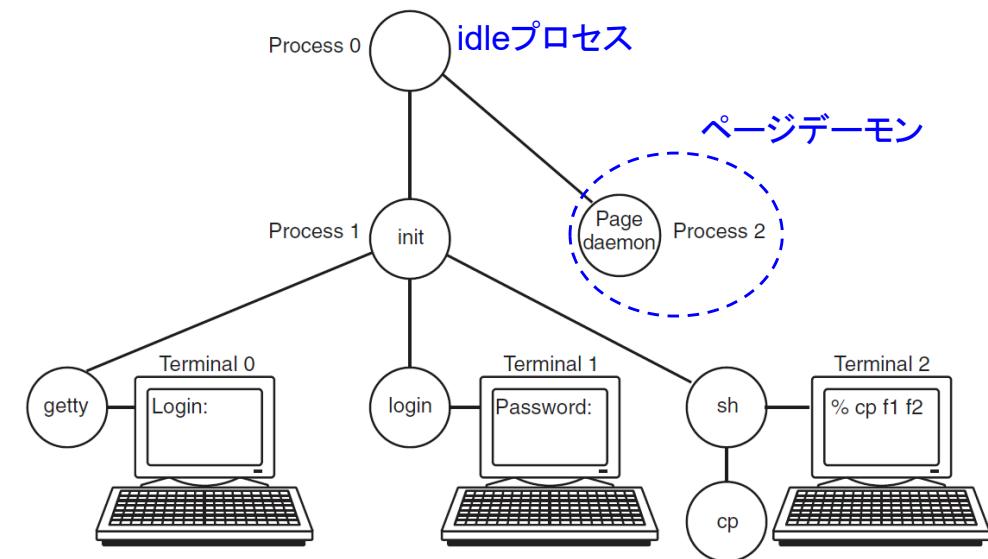
Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

- ## □ 最良の2つのアルゴリズムはエージング(aging)とWSClock
- それぞれ、LRUとワーキングセットの考え方に基づいている
 - 両アルゴリズムとも、高性能なページングが可能で、効率良く実現できる

Linuxにおけるページ置き換えの概要(1)

□ Page daemon (kswapd) による、周期的なページフレーム回収

- フリーのページフレームの数が少ないときには、使用されているページフレームを回収してフリーリストに入れる（一度に最大32個のページフレームを回収）
- これにより、ページフォルト時にページ置き換えの必要がなくなる



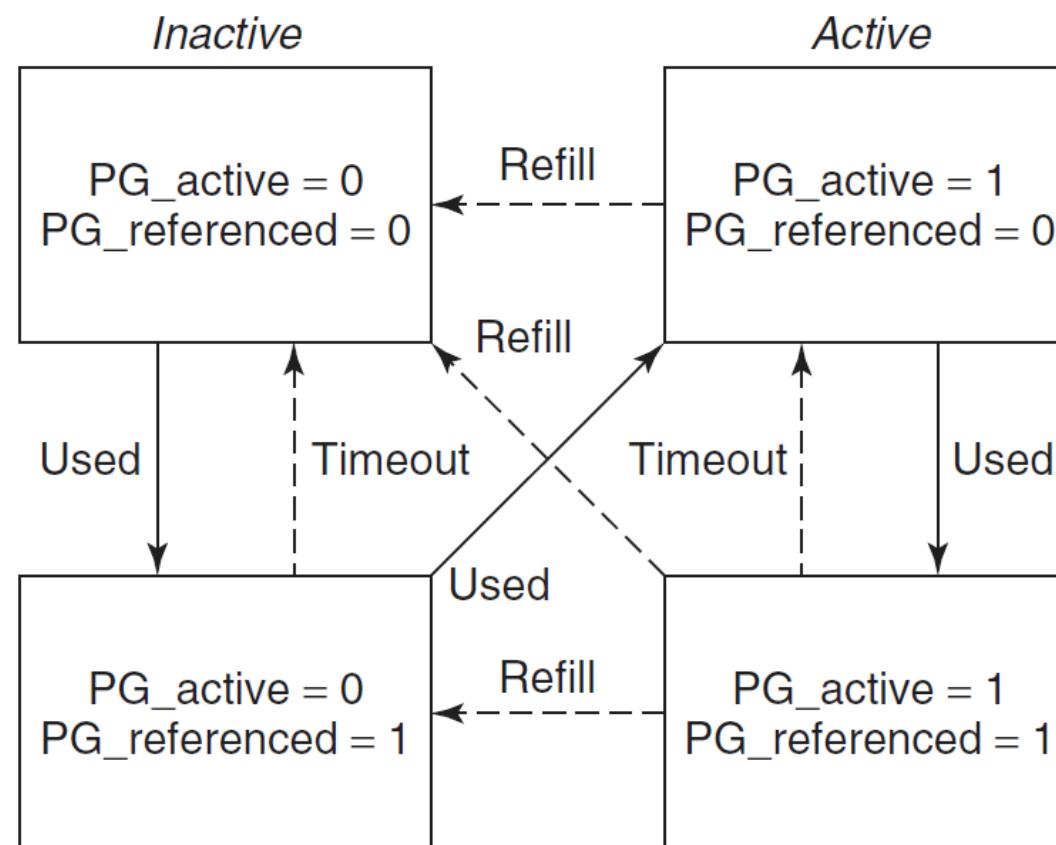
■ PFRA (Page Frame Reclaiming Algorithm) で管理

- 回収の優先順位
 - メモリ内のディスクキャッシュ。（ページテーブルを更新する必要がないため）
 - ユーザプロセスの共有されていないページ
 - 共有されているページ（複数のページテーブルを更新する必要あり）
- Clock式の走査
- NRUとLRUの中間的なアルゴリズム（次スライド）

Linuxにおけるページ置き換えの概要(2)

- PG_activeとPG_referencedの2段階のフラグでLRU情報を管理

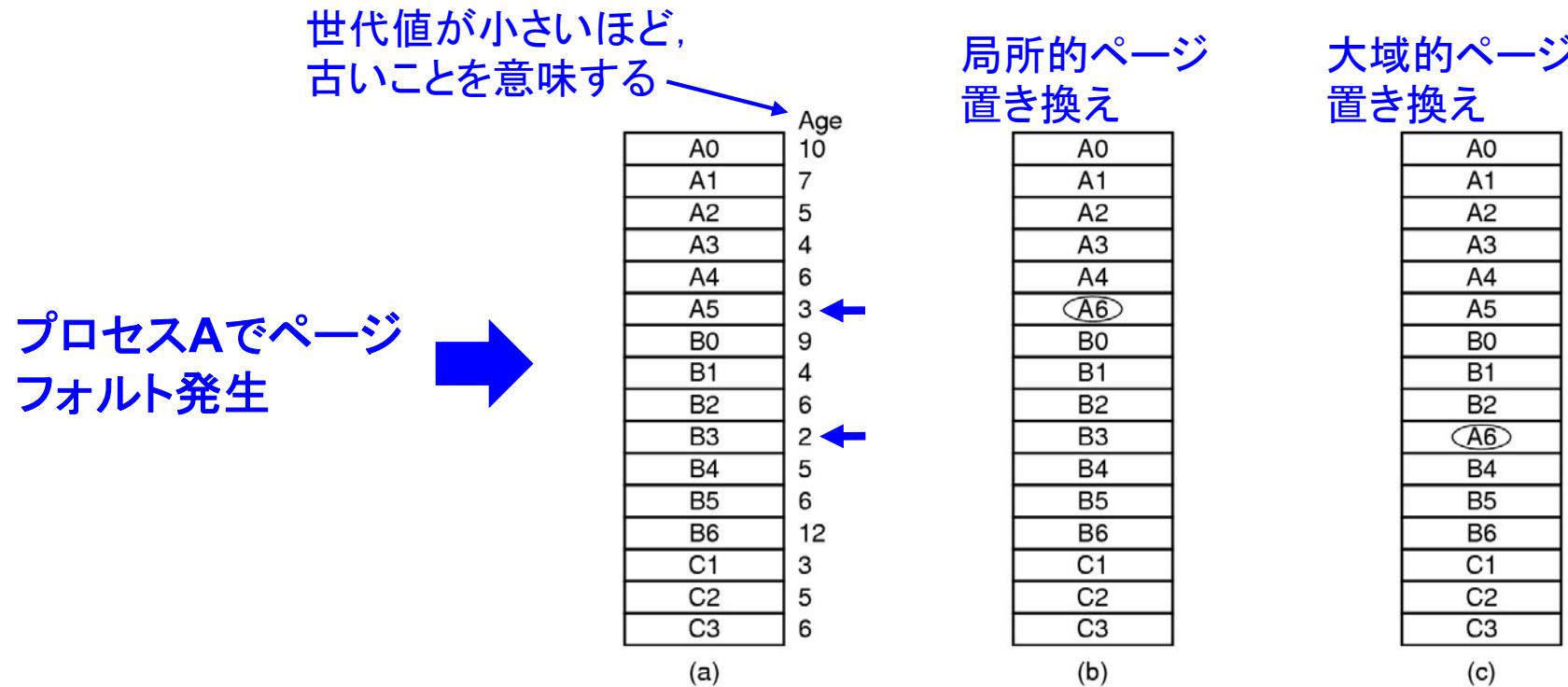
- フレーム回収は基本的にはInactiveなページフレームが対象
- 連続する2回の走査でreferencedが確認されると, Inactive → Activeに昇格



Refill: 回収候補を増やすために
意図的にactiveページを
減らす手続き

ページングシステムの設計時の課題(1)

- 複数のプロセスに対してページフレームをどのように割り当てるべきか
 - LRUなどの"置き換えアルゴリズム"とは直交する問題
- 局所的 vs. 大域的割当てポリシー



- 一般的には、大域的アルゴリズムがよりメモリ使用効率が良い
 - 特に、プロセスの実行中にワーキングセットサイズが変動するとき
 - 局所的アルゴリズムでは、ワーキングセットが大きくなるとスラッシングが起こり、ワーキングセットが小さくなるとメモリが浪費される

ページングシステムの設計時の課題(2)

□ 局所的割当てにおいて、ワーキングセットサイズの変化に対処する方法

■ PFF(Page Fault Frequency)アルゴリズム

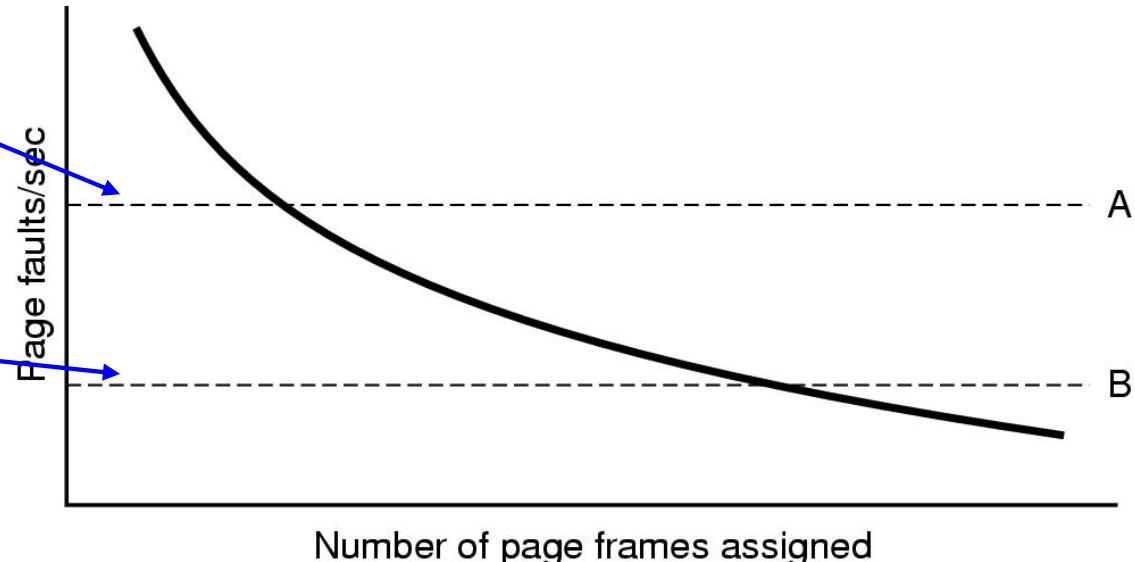
- 動的にプロセスへのページ割当てサイズを管理する1つの方法
 - プロセスのページ割当てをいつ増やすのか、減らすのかを指示

容認できない高いページ fault 率

→ 当該プロセスはもっとたくさんのページフレームを与えられるべき

低すぎるページ fault 率

→ 当該プロセスからページフレームを取り上げる



- PFFは各プロセスのページ fault 率を容認できる範囲に抑えようとする

ページングシステムの設計時の課題(3)

□ 負荷制御(Load Control)

- 最良のページ置き換え、およびPFFに基づく割当てを行っても、システムがスラッシングを起こすことがある
 - 例) 全プロセスのワーキングセットの総サイズがメモリ容量を超えるとき
- したがって、ページ単位の仮想化を行っている場合でも、プロセス単位の二次記憶(ディスク)への追い出し(=マルチプログラミング度の低下)はなお必要となる
 - あるプロセスをディスクに追い出し、それに割り当てられていたページフレームをスラッシングを起こしているプロセス間に分配する
 - 追い出されたプロセスはしばらくはスケジューリングの対象としない
 - → マルチプログラミング度の低下
 - → I/O boundプロセスの割合によってはCPU使用率が低下するため、要注意
 - スラッシングが無くなるまで、プロセスの追い出しを続ける

ページングシステムの設計時の課題(4)

□ ページサイズ

- 最適なページサイズを決定するためには、いくつかの相反する要因間のバランスをとることが必要となる
 - アプリケーションに依存するため、最適解は存在しない
- 大きなページサイズの問題
 - 内部フラグメンテーション(内部断片化)
 - テキスト、データ、STACKセグメントの大きさが、ページサイズのちょうど整数倍になることは(滅多に)ない
 - 平均的には、最後のページの半分が空となる
 - そのページの余分な空間は浪費されることになる
- 小さなページサイズの問題
 - プロセスが多数のページを必要とすることになり、大きなページテーブルが必要となる(≒プロセスが多数のTLBエントリを使用)
 - 小さなページをディスクとの間で転送することは、大きなページの場合と比較し、ほとんど同じくらいの時間がかかる → 効率が悪い

ページングシステムの設計時の課題(5)

■ ページサイズの数学的解析

p : page size

s : average process size

e : size of page table entry

$$overhead = se/p + p/2 \quad \leftarrow \text{ページテーブルのサイズと内部フラグメンテーション}$$

p に関して一階微分し、0との方程式を形成することにより

$$-se/p^2 + 1/2 = 0 \quad \rightarrow \quad p = \sqrt{2se}$$

- 例) $s = 1\text{MB}$, $e = 8\text{B}$ に対して, $p = 4\text{KB}$ となる
- 現在はプロセスサイズが増大しているが、歴史的慣習とメモリの低価格化により、商用システムでは典型的な値として 4KB や 8KB が使用されている

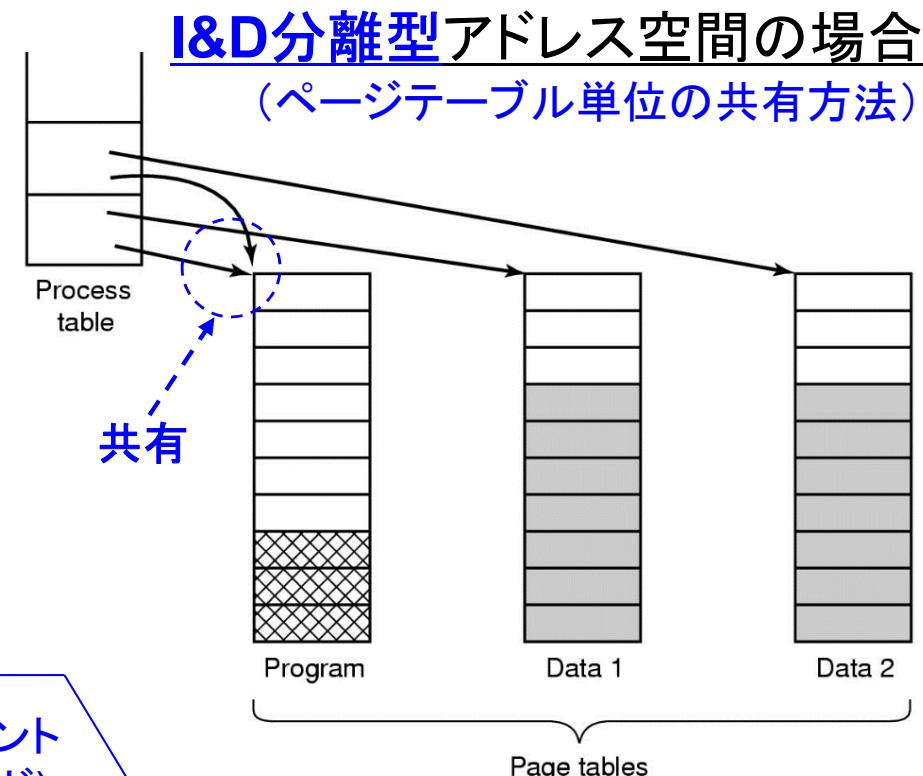
ページングシステムの設計時の課題(6)

□ 共有ページ

- 複数のユーザが同時に同じプログラムを実行することがよくある
- ページを共有することにより、同じ内容(プログラムコード等)のページをメモリ内で2つ用意するのを避けることが効率的なのは明らか

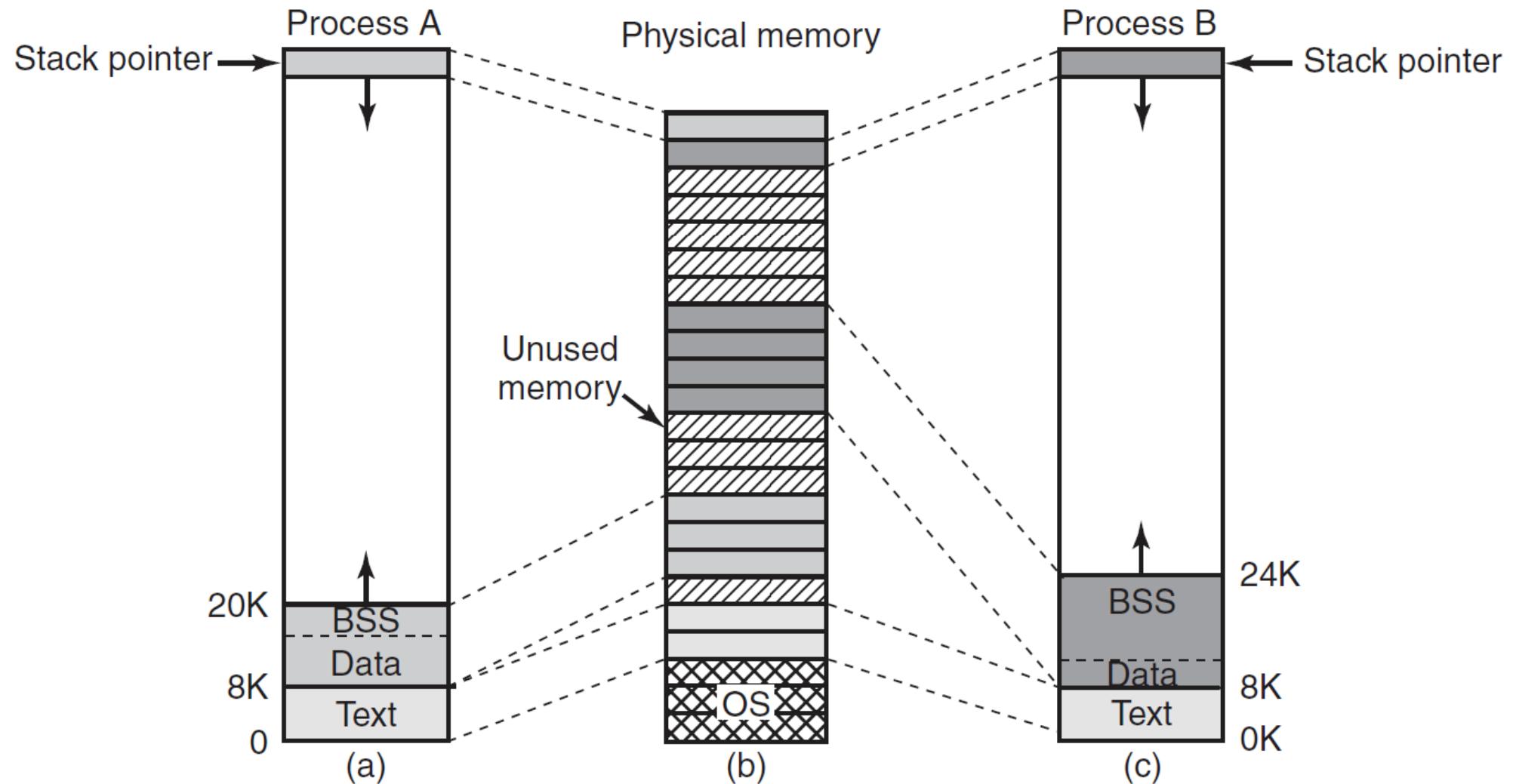
- 全てのページが共有可能というわけではない
 - プログラムテキスト(命令列)のような読み出し専用ページが共有可能
 - データは基本的には(アプリケーションの性質上)共有できない

他の(通常の)方法として、異なるプロセスのページテーブルエントリが同じページフレームを指示することで共有が可能(次スライド)



ページングシステムの設計時の課題(7)

□ 共有ページ(続き)



Text領域については、両プロセスの(仮想)ページ群が
同じ(物理)ページフレーム群にマップされている

ページングシステムの設計時の課題(8)

- データ(READ ONLY)の共有は不可能なわけではない
- UNIXでは、**fork**システムコールの実行後、親と子はプログラムテキストとデータの両方を共有するようになっている
 - 子プロセスに自身のページテーブルを与え、親と同じページフレーム集合を指すようにしている
 - したがって、fork時にはページのコピー(複製)は行われていない
 - しかし、全てのデータのページは **READ ONLY** としてマップされる
 - どちらかのプロセスがデータを更新しようとしたとき、アクセス違反例外によりOSへのトラップが生ずる
 - その場合、コピーが生成され、今度は各プロセスが自身のコピーを持つようになる
- このやり方が意味するところは、(プログラムテキストページを含めて)更新されることのないページはコピーを生成する必要がないということ
 - 実際に書き込まれるデータのページのみコピーを生成する
 - このアプローチは、**copy on write** と呼ばれ、性能を向上させる

ページングシステムの設計時の課題(9)

□ 共有ライブラリ(Shared Libraries)

- I/Oやグラフィックスのためのライブラリは多くのアプリケーションが利用するが、各アプリケーションバイナリファイルに(static linkによって)含めると、ディスクおよび実行時のメモリを圧迫する
- Shared libraries / DLL(Dynamic Link Libraries)
 - アプリケーションバイナリ内のライブラリ関数呼び出し個所には、実行時に関数をバインドするための小さなstubルーチンのみを含めておく
 - 関数呼び出し時にstubルーチンの実行によりOSへのトラップが発生し、対象ライブラリ関数がメモリにロードされる
 - もし他のアプリケーションが既に同じライブラリを使用している場合は、新たにコードはせず、メモリ内の同じ実体へ(ページテーブルによって)紐づけるのみ
 - ライブラリをアップデートすると、次のアプリケーション実行から利用可能
 - アプリケーションの再コンパイルは必要無い
 - 例)Windows Updateなどによる標準DLLのアップデート

実装上の課題(1)

□ ページFAULT処理(の一例)

1. ハードウェアによって例外(カーネルへのトラップ & 実行モードの変更)が発生し、直前のプログラムカウンタ値が自動的に特別なレジスタに保存され、(プログラムカウンタが書き換わることによって)例外ハンドラの最初のルーチンにジャンプする
2. (アセンブリコードで作られた)ルーチンがスタートし、汎用レジスタや他の(プログラムカウンタ値、プロセッサ状態ワード等の)揮発性情報をセーブし、それらがこれから始まるOS実行により壊されないようにする。このルーチンからの手続き呼び出しによってOSの例外メイン処理に移る
3. OSの例外メイン処理はページFAULTが発生したことを発見し、どの仮想ページが必要とされているのかを調べる。多くのアーキテクチャで、特別なハードウェアレジスタの1つがこの情報を含んでいる。(そうでなければ、OSは(格納された)プログラムカウンタ値を読み出し、その命令を読み込み、ソフトウェアで解析し、FAULTが起ったときにその命令は何をしていたのか(仮想アドレス)を判断する。)

実装上の課題(2)

4. フォルトを起こした仮想アドレスが判明すると, OSはそのアドレスに対してアクセス保護違反が無いかをチェックする. 違反ならば強制終了させる. 違反が無ければ, OSはフリーのページフレームがあるかをチェックする. もし無ければ, ページ置き換えアルゴリズムにしたがって置き換えページフレームを選択する
5. 選択したページフレームが modified(dirty) の場合, そのページをディスクに書き出すためのディスク操作をディスク転送スケジュールに入れて, フォルトを発生させたプロセスを中断し, プロセス切り替えを行い, ディスク転送が完了するまで他のプロセスを実行する. (当該ページフレームには, 切り替え後のプロセスによって使用されないようにマークを付しておく.)

実装上の課題(3)

6. ページフレームが元々cleanの場合, あるいはディスク書き出し後に cleanになつたらすぐに, OSは必要とされているページが存在するディスクアドレスを調べ, そのページを読み込むためのディスク操作をディスク転送スケジュールに入れる. ページを読み込んでいる間は, フォルトを起こしたプロセスはやはり中断させ(あるいは既に中断中), 他のユーザプロセスを実行する. (当該ページフレームにマーク付加.)
7. ディスクからの割込みによってページが到着したことがわかると, ページテーブルを更新する. (対応するエントリのページフレーム番号がその位置を指すようにする.)
8. フォルトを起こしたプロセスをスケジュールし, OSは自身を呼び出したセンブリ言語ルーチンに制御を戻す
9. このルーチンはレジスタ値や状態情報を再ロードし, フォルトを起こした命令にリターンする. (かつ, ユーザモードに戻す.) まるでフォルトが起こらなかつたかのように実行が再開・継続される

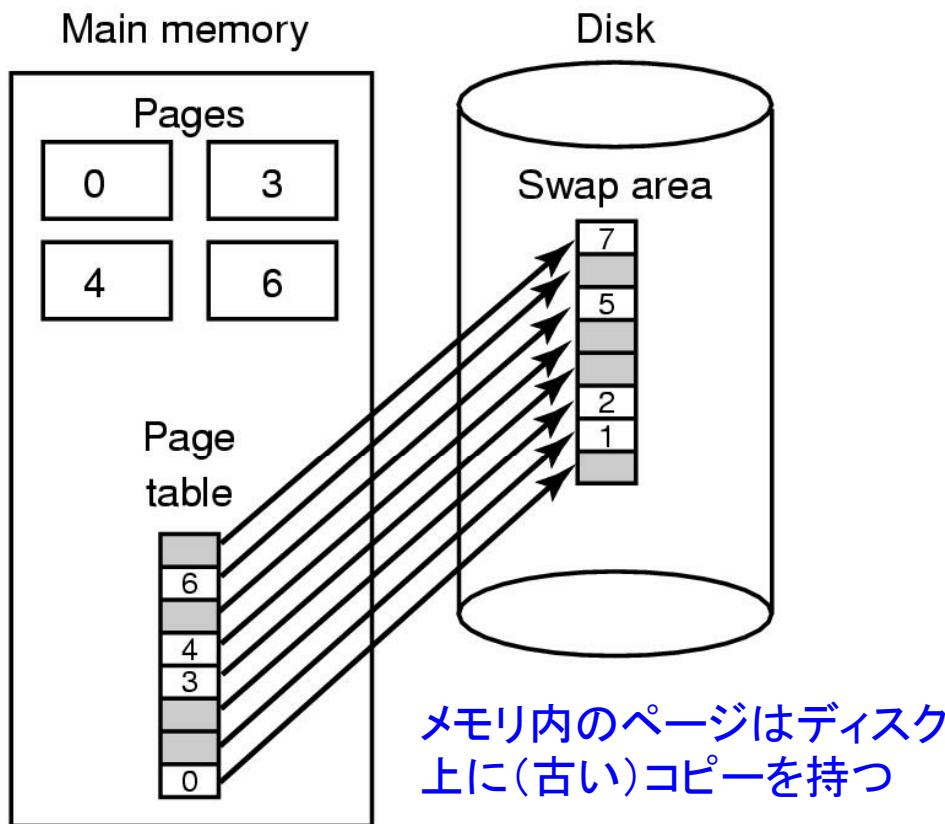
□ メモリ内でのページのロック(追い出しの対象外とすること)

- ファイルやデバイスから、アドレス空間内のバッファへ読み込みを行うシステムコールを発行したプロセスを考える
 - (DMA転送による)I/Oが完了するのを待つ間、プロセスは中断し、他のプロセスが実行を許可される
 - この「他のプロセス」がページフォルトを起こす可能性がある
 - そのとき、DMA転送はその新しいフォルトを気にしていない
 - もしけれどもページ置き換えアルゴリズム(大域的割当てポリシー)がI/Oバッファを含むページを追い出し対象に選んでしまったら、何が起こるだろうか？
- この問題への1つの解決策としてI/Oに携わるページをメモリ内にロックしておき、追い出されないようにする方法がある
→ ピニング／ピンダウン(pinning / pin down)
POSIXでは `mlock()`/`munlock()`システムコールが提供されている
- 他の方法として、I/O処理はカーネル空間内のバッファを使用して行い、その後データをユーザページにコピーする方法がある

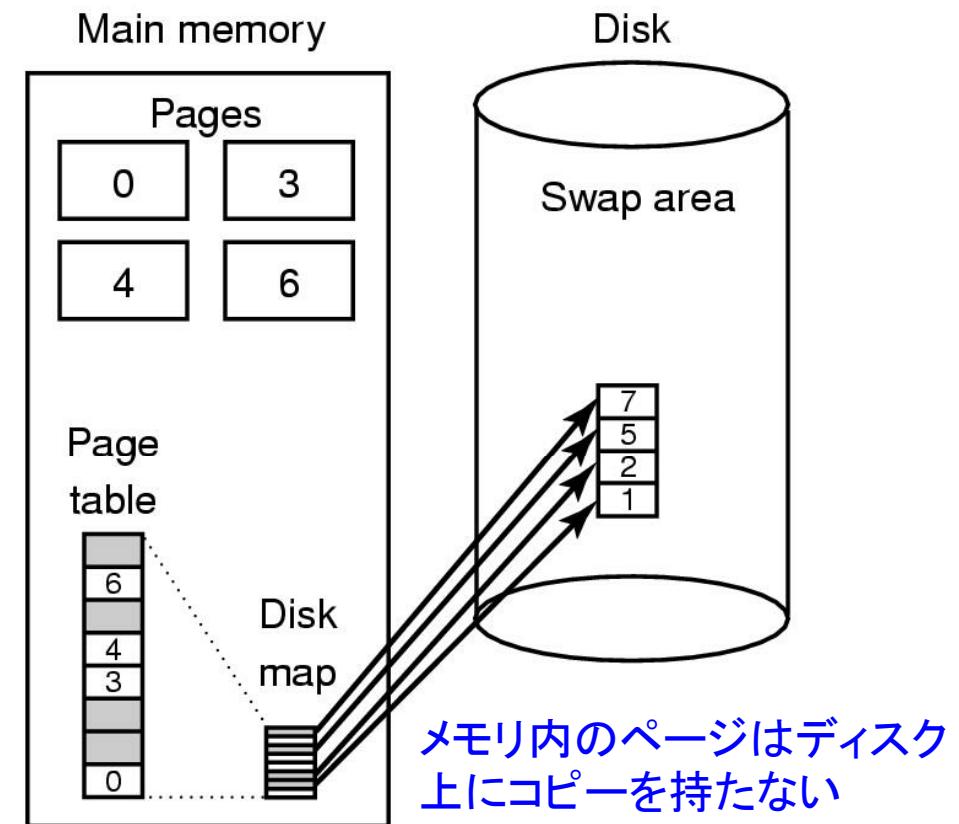
実装上の課題(5)

□ バックストア(Backing Store)

- ページがページアウトされるときに、ディスク上のスワップ領域のどこに置かれるか



(a) 静的スワップ領域へのページング
ディスク内の使用サイズが大きい



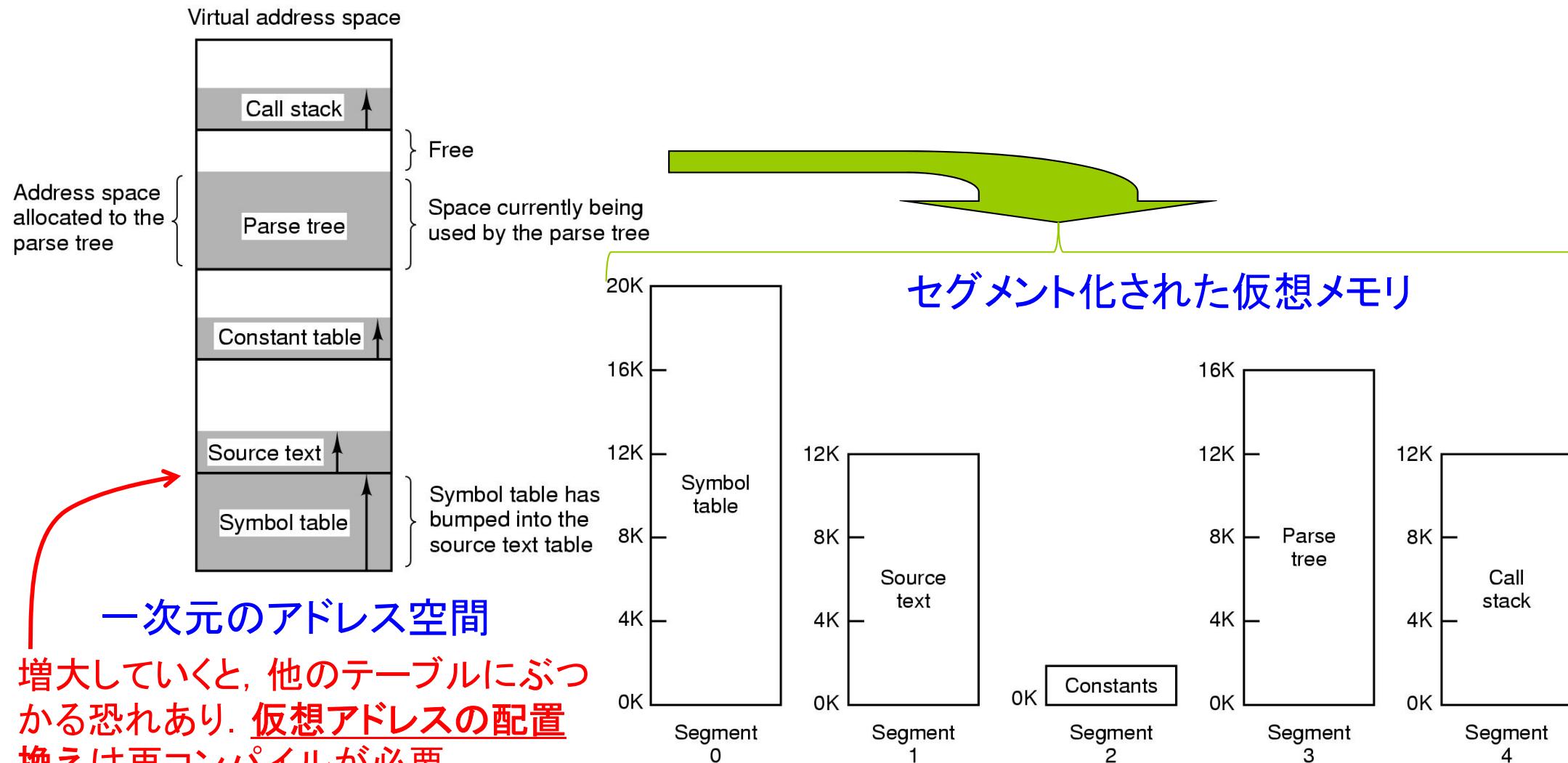
(b) ページの動的バックディングアップ
専用ディスクマップ情報が必要

セグメンテーション(1)

- セグメンテーション: 仮想記憶のもう一つの実現方法
- セグメント
 - 1つのプロセスに対する、複数の互いに独立したアドレス空間
 - 各セグメントはアドレスの線形列からなる
 - セグメントが異なれば、長さも異なりうる
 - セグメント長は実行中も変化しうる
例) スタックセグメント
 - 異なるセグメント同士はお互いに影響を与えることなく、独立して伸び縮みできる
 - プログラムはメモリ参照の際、2つのアドレス（セグメント番号とセグメント内アドレス）をメモリシステムへ供給する
 - プログラマ／コンパイラは論理的な実体としてセグメントを認識する
 - 1つのセグメントは同じ種類の実体を含む
例) 手続きセグメント、配列セグメント、スタックセグメント、など

セグメンテーション(2)

□ 例：コンパイラプログラムが使用(生成)するデータ構造/テーブル



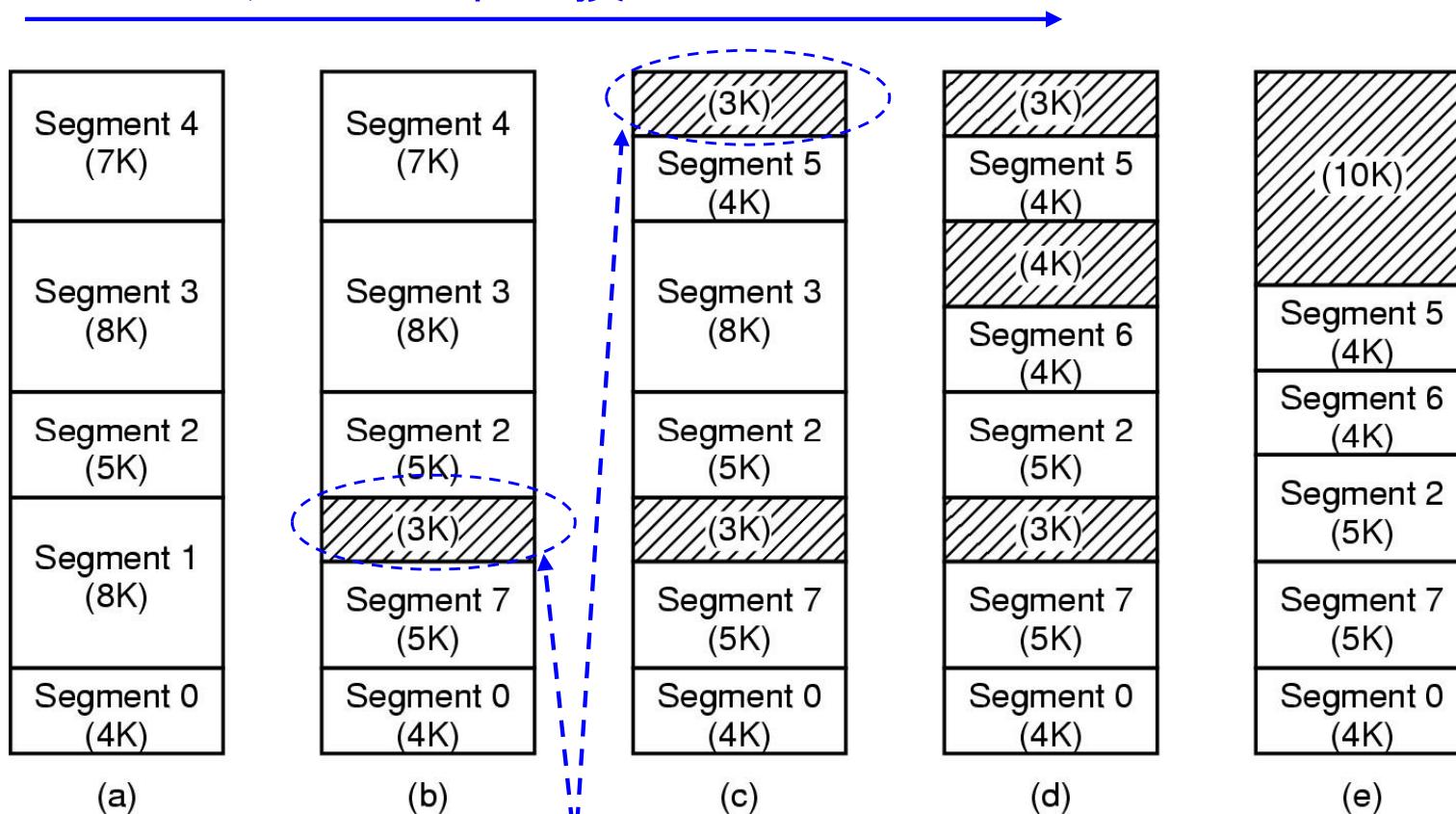
セグメンテーション(3)

- 各セグメントはプログラマが認識する論理的な実体(例えば、手続き, 配列, スタック)を形成し, 異なるセグメントは異なる種類の保護をかけられる
 - 手続きセグメントは実行専用(execute-only)
 - 浮動少数配列は読み書き(read/write)だが, 実行不可
- 一方, ページングの場合, ページの内容は, ある意味では, 意図しないもの
 - プログラマはページングが行われている事実すら気が付いていない
 - コンパイラやOS(そしてMMU)が協調してページ毎の保護を提供しており, セグメント化メモリよりも複雑である

セグメンテーション(4)

□ 純粹なセグメンテーションの実現

セグメントの置き換え



ホールは浪費(無駄)

External fragmentation
(外部断片化)

要するに、セグメント単位でのスワッピング。
サイズが膨張したら、メモリ内での場所を変更する。

Removal of external
fragmentation by compaction

セグメンテーション(5)

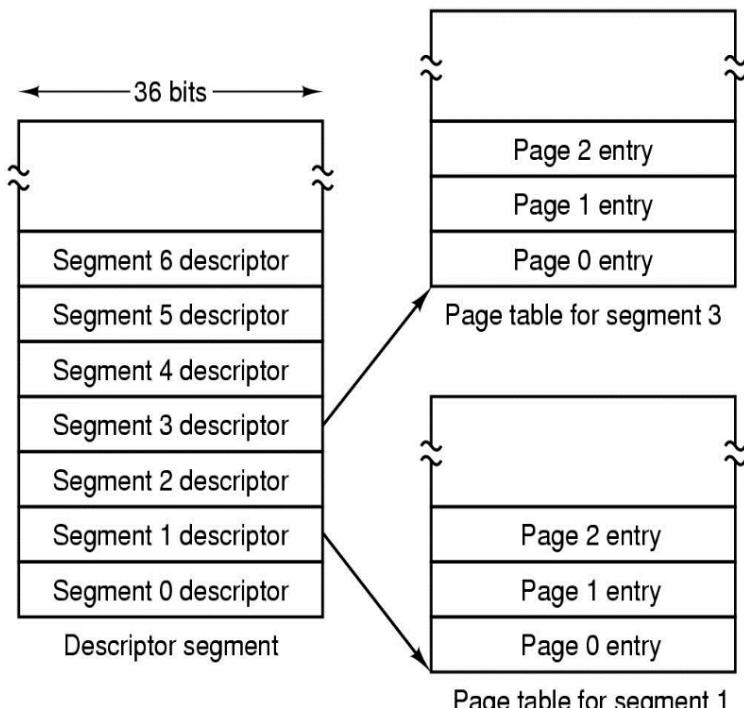
- セグメントのサイズが大きいと、全体を主記憶に置いておくのは不可能な場合がある
- ページングの考え方を利用 → Paged segments
 - セグメントをページに分割し、実際に実行に必要な部分のページのみをメモリに置いておくことができる
- ページングを併用したセグメンテーション：MULTICS
 - 各プログラムに 2^{18} 個(250,000以上)までのセグメントの仮想記憶を提供し、それぞれのセグメントは 65,536個の36ビットワードまでのサイズ
 - 各セグメントにページングを適用する
 - ページングの利点(セグメント全体をメモリ内に置いておく必要がなく、外部断片化も発生しない)と、セグメンテーションの利点(各データ領域／テーブルの伸長や保護が容易)を組み合わせる

セグメンテーション(6)

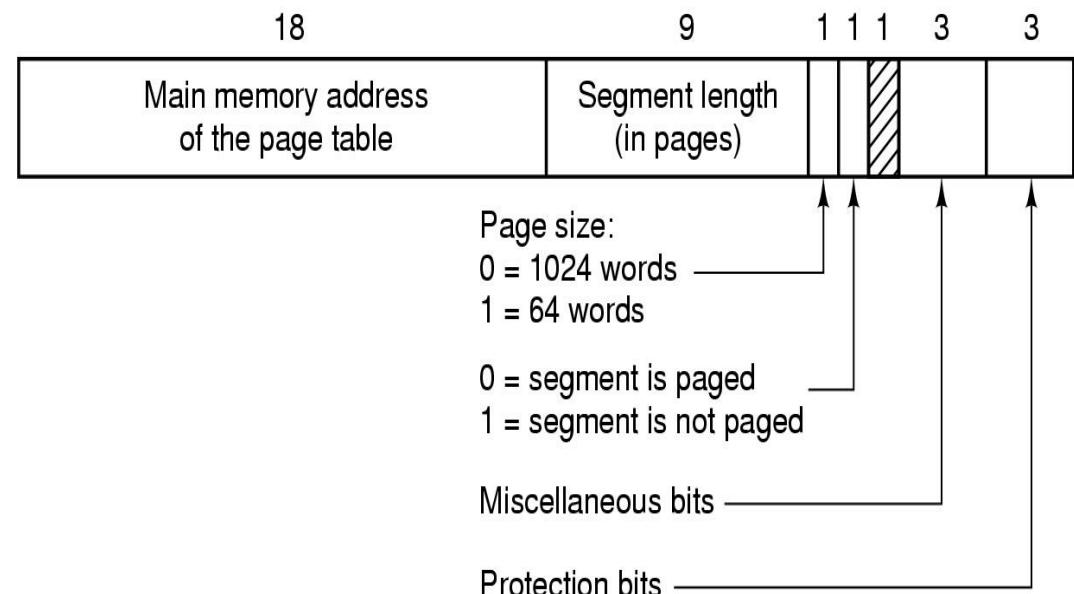
- MULTICS上の各プロセスはセグメントテーブルを持つ。セグメントテーブルのエントリは、該当するセグメントのための記述子(segment descriptor)である
- セグメント記述子はそのセグメントが主記憶内にあるかどうかの情報を含む
 - セグメントの一部がメモリ内にある場合、そのセグメントはメモリ内にあると見なされ、そのページテーブルはメモリに存在することになる
 - セグメントがメモリ内に存在する場合、記述子はページテーブルへのポインタを含む(次スライドの左図)
 - 記述子は、セグメントサイズ、保護情報、その他のいくつかの項目を含む(次スライドの右上図)
- MULTICSにおけるアドレスは2つの部分からなる：セグメント番号とセグメント内アドレス
 - セグメント内アドレスは更に、ページ番号とページ内オフセットに分けられる(次スライドの右下図)

セグメンテーション(7)

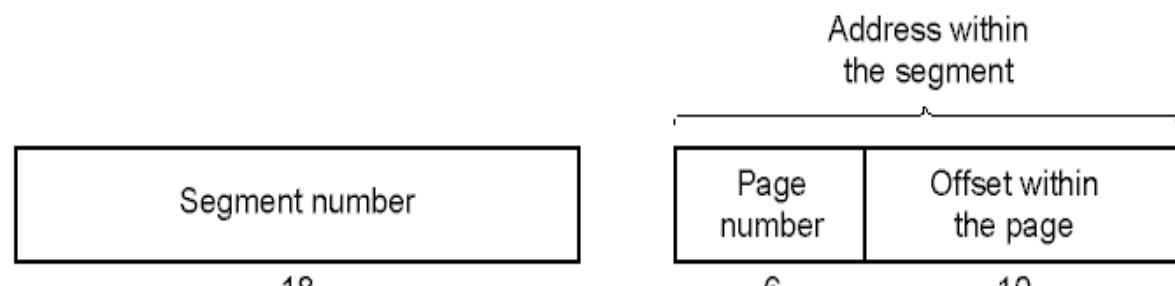
セグメント記述子はページテーブルを指す



セグメント記述子



34ビットのMULTICS 仮想アドレス



ページサイズ=1024ワードの場合

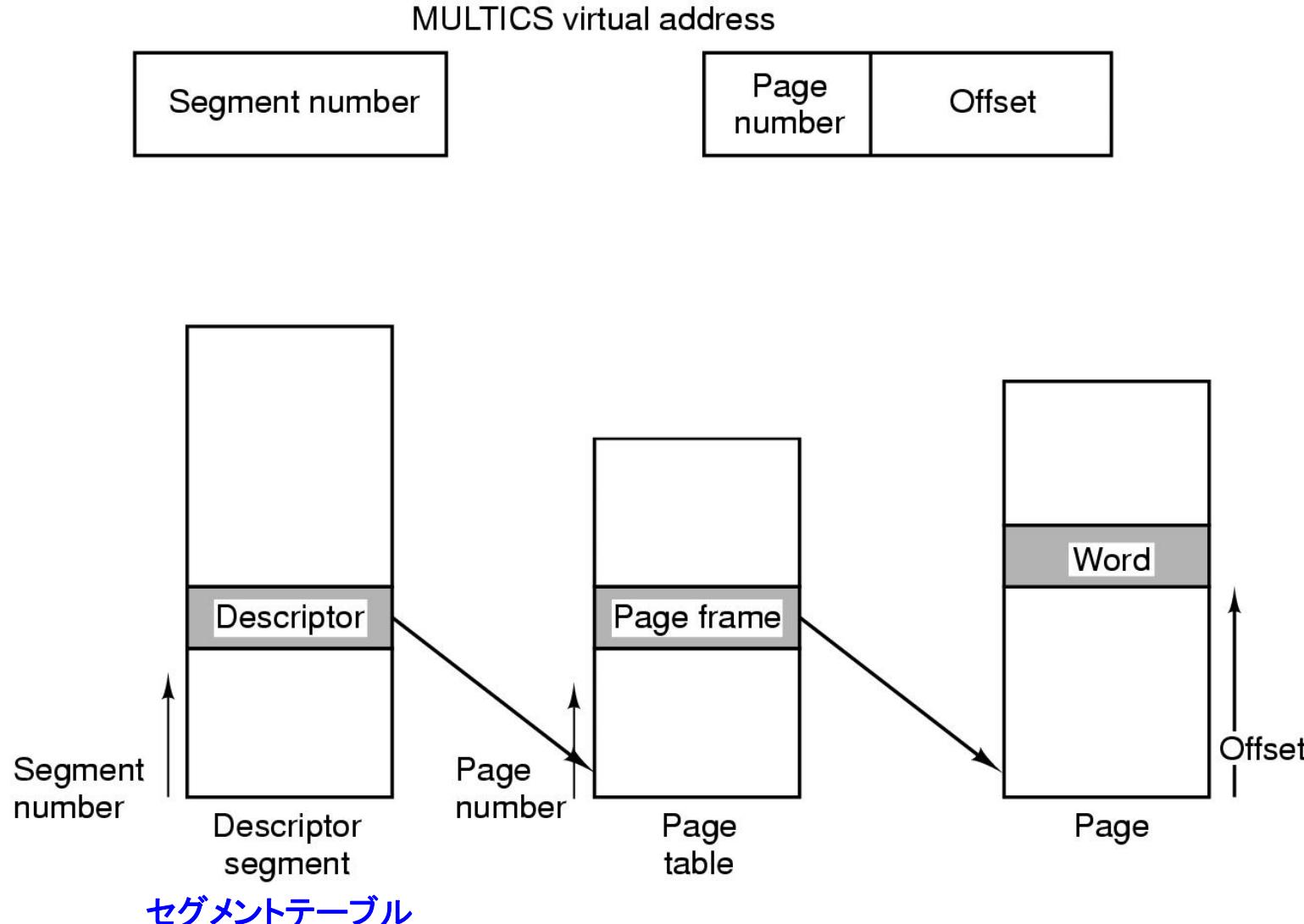
セグメンテーション(8)

■ メモリ参照を行うとき

1. セグメント番号を使用し、セグメント記述子を見つける
2. セグメントのページテーブルがメモリ内にあるかどうかチェックする。メモリ内に存在する場合は、位置を特定する。無い場合は、セグメントフォルトが発生する。保護違反の場合も、トラップが発生する
3. 要求された仮想ページのページテーブルエントリを調べる。ページがメモリ内に無い場合は、ページフォルトが発生する。メモリ内に存在する場合は、ページテーブルエントリから、ページの先頭の主記憶アドレスが得られる
4. その先頭アドレスにオフセットを加算し、ワードの主記憶アドレスを生成する
5. 最終的に、リードあるいはライトを行う

一連の流れは次スライドの図

セグメンテーション(9)



セグメンテーション(10)

- 以上のようなメモリ参照の手続きにしたがうと、2段階のテーブル参照オーバヘッドのためにプログラムを高速に実行できない
- MULTICSのハードウェアは16エントリのTLBを持つ

Comparison field					Is this entry used?
Segment number	Virtual page	Page frame	Protection	Age	
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

Skip

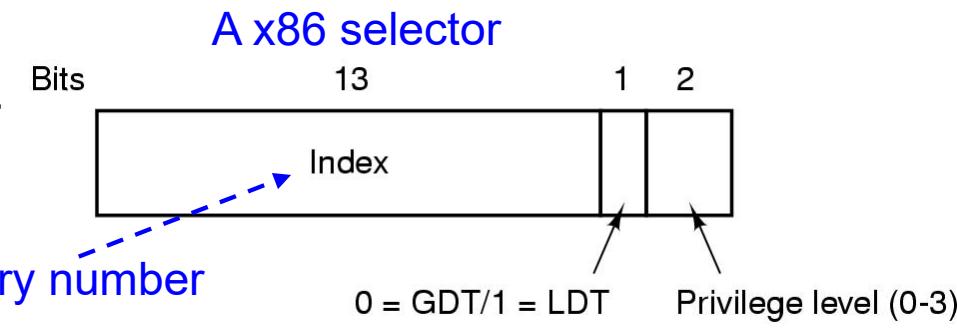
セグメンテーション(11)

□ ページングを併用したセグメンテーション: The Intel X86

- MULTICSよりもセグメント数は少ない(16K個まで)が、セグメントサイズが大きい(10億個の32ビットワード分)
- LDT (Local Descriptor Table)
 - 各プロセスは自身の LDT を持つ
 - コードやデータ、スタックなど、プロセスローカルなセグメントを記述
- GDT (Global Descriptor Table)
 - 全てのプロセスで共有される
 - OSを含めた、システム全体用のセグメントを記述
- セグメントレジスタ(トータル6つ存在)とセレクタ
 - CS: コードセグメントのセレクタを保持
 - DS: データセグメントのセレクタを保持
 -

注)x86-64では、セグメンテーションはobsolete扱い(互換性のみのため残されている)

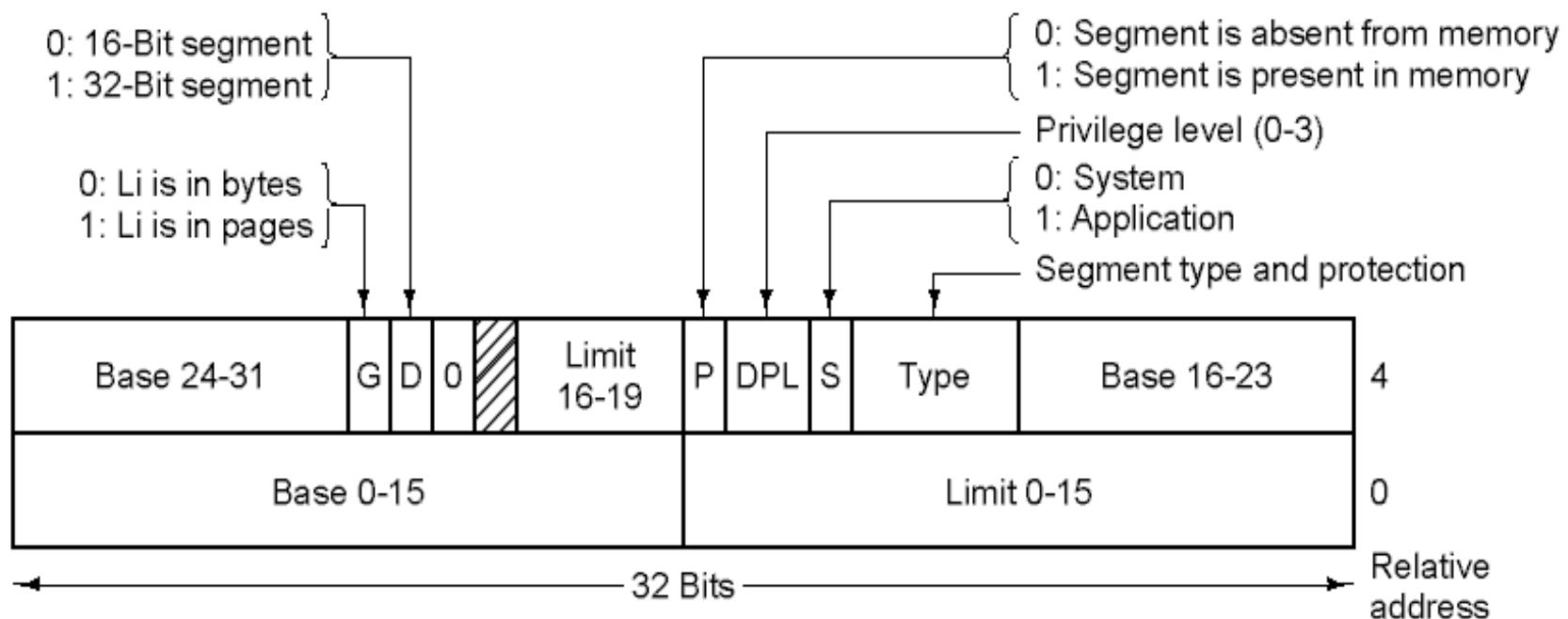
LDT/GDT entry number



セグメンテーション(12)

- セグメントレジスタにセレクタがロードされたとき、対応する(参照するセグメントの)セグメント記述子がLDTかGDTから読み出され、マイクロプログラムのレジスタへ格納される。これにより、その記述子に高速にアクセスすることが可能となっている

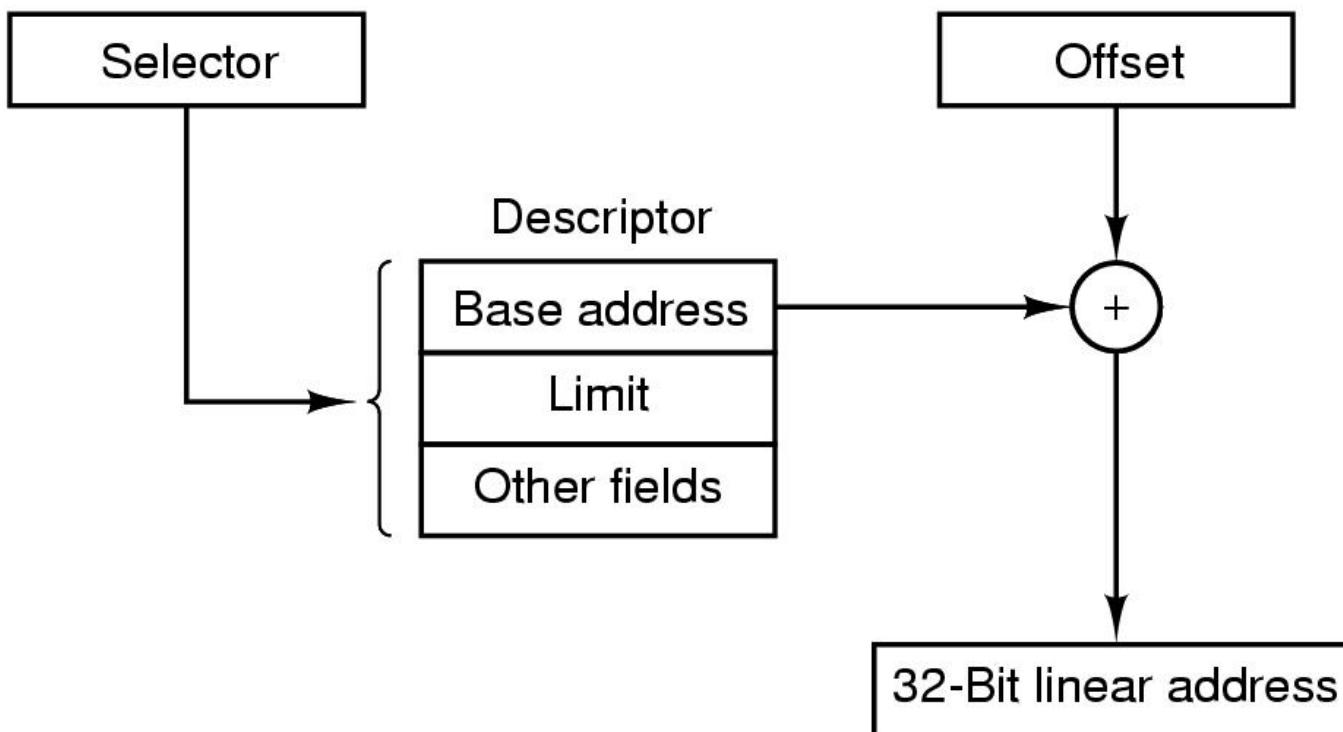
x86 code segment descriptor. (Data segments differ slightly.)



セグメンテーション(13)

- セグメントがメモリ内に存在し、オフセットが範囲内(Limit)の場合、記述子内の32ビットのベース(Base)フィールドをオフセットに加算し、線形アドレス(linear address)を形成する

Conversion of a (selector, offset) pair to a linear address



Skip

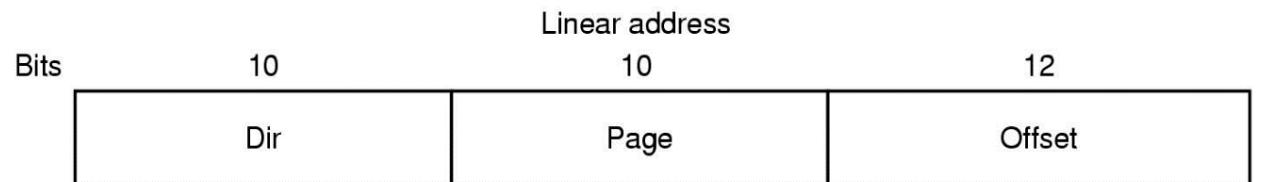
セグメンテーション(14)

- ページングが使用不可(disabled)（グローバル制御レジスタ内のビットによる）となっている場合は、前スライドの線形アドレスが物理アドレスとして使用される
- ページングが使用可能(enabled)となっている場合は、この線形アドレスは仮想アドレスとして解釈され、ページテーブルを使用して物理アドレスにマップされる

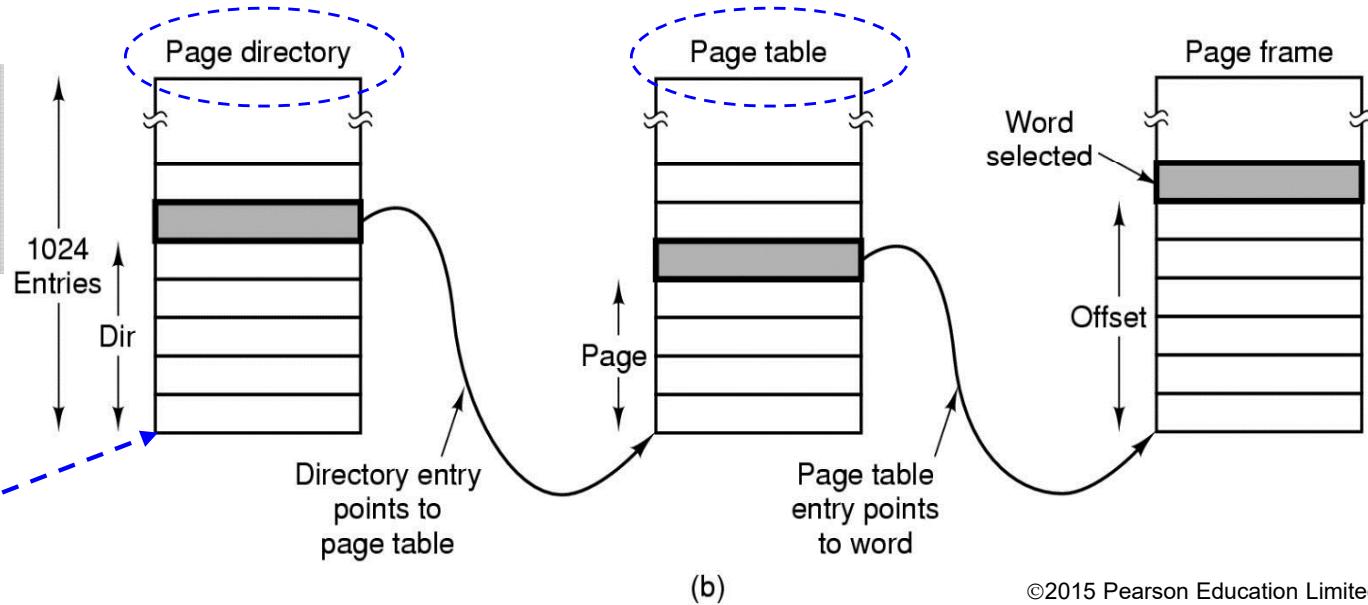
Mapping of a linear address
onto a physical address
(2-level mapping)

TLBを持つため、このテーブル
ウォークはTLBミスの場合のみ行
われる

Pointed to by a global reg.



(a)



セグメンテーション(15)

- セグメント化はせず、ページ化されたメモリのみも使用可能
 - 全てのセグメントレジスタは同一のセレクタでセットされ、記述子のベース(*Base*)を0、リミット(*Limit*)を最大値にセットする
 - プログラムが発行するアドレスは線形アドレスとして扱われ、1つのアドレス空間のみが使用されることになる → 事実上、普通のページングとなる
- 実際は、x86(-32)用のほとんど全てのOSはこの方法で動作している
 - OS/2 が Intelの MMUアーキテクチャをフル活用した唯一のOSであった
- UNIX系、Windowsはこのセグメント機構を使用しなかったため、Intelはx86-64からは提供しないことに決定した
 - セグメンテーション実現のためのハードウェアを削除可能

入出力



I/Oハードウェアの原理(1)

□ I/Oデバイス

- 大雑把には、以下の2種類に分類される

□ ブロックデバイス(block devices)

- 情報を固定サイズ(512~65,536バイト)のブロックで格納する。ブロックはアドレス付けされる
- 各ブロックを独立して(不連続に)読み書き可能
- 例) ハードディスク, Blu-rayディスク, USBメモリ

□ キャラクタデバイス(character devices)

- (キャラクタの)ストリームを送り出したり、受け取ったりする
- ストリーム内のデータをアドレス付けできない
- 例) プリンタ、ネットワークインターフェース、マウス

I/Oハードウェアの原理(2)

■ よく使用されるデバイスのデータ転送率

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner at 300 dpi	1 MB/sec
Digital camcorder	3.5 MB/sec
4x Blu-ray disc	18 MB/sec
802.11n Wireless	37.5 MB/sec
USB 2.0	60 MB/sec
FireWire 800 (IEEE1394)	100 MB/sec
Gigabit Ethernet	125 MB/sec
SATA 3 disk drive	600 MB/sec
USB 3.0	625 MB/sec
SCSI Ultra 5 bus	640 MB/sec
Single-lane PCIe 3.0 bus	985 MB/sec
Thunderbolt 2 bus (FireWireの後継)	2.5 GB/sec
SONET OC-768 network (2000年代初頭)	5 GB/sec

I/Oハードウェアの原理(3)

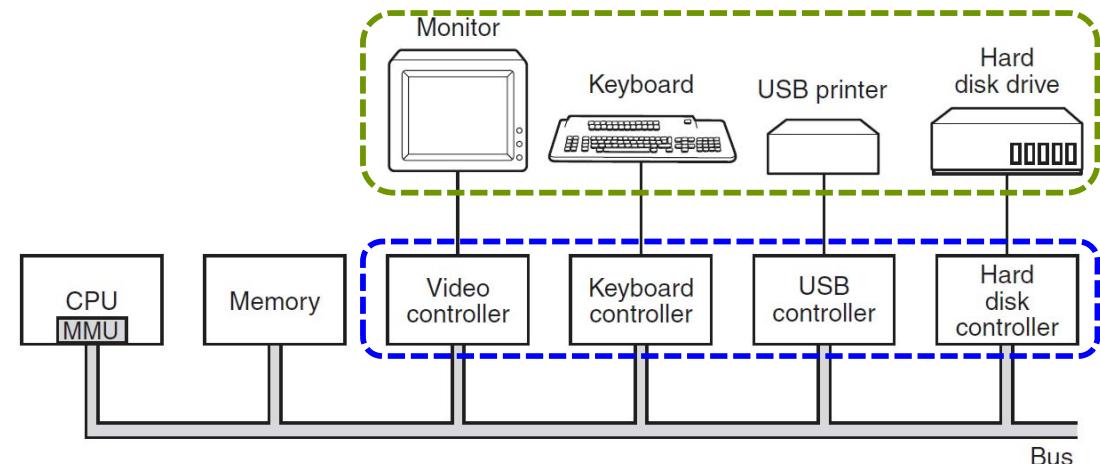
- I/Oユニットは通常、機械的動作部分と電子的動作部分から構成される

- 機械的動作部分

- デバイス自身

- 電子的動作部分

- デバイスコントローラ(Device controller/device adapter)



I/Oハードウェアの原理(4)

- 各デバイスコントローラは、OS／CPUと通信するためのいくつかの制御レジスタを持つ
 - コマンドレジスタ
 - CPU上のOSはコマンドレジスタに書き込むことにより、デバイスに対してデータを送るよう、あるいはデータを受け取るように命令できる。（スイッチのオン・オフなどの命令を行うことも可能）
 - 状態レジスタ
 - CPU上のOSは状態レジスタから読み出すことにより、デバイスの状態や、次の命令の受け取り準備ができているかどうかなどを知ることが可能
- 制御（コマンド、状態）レジスタに加えて、多くのデバイスでOSが読み書き可能なデータバッファが存在する
 - 例）スクリーン上のピクセル群に対して表示をする際、プログラムやOSが書き込むことができるデータバッファ（ビデオRAM）を使用するのが一般的な方法

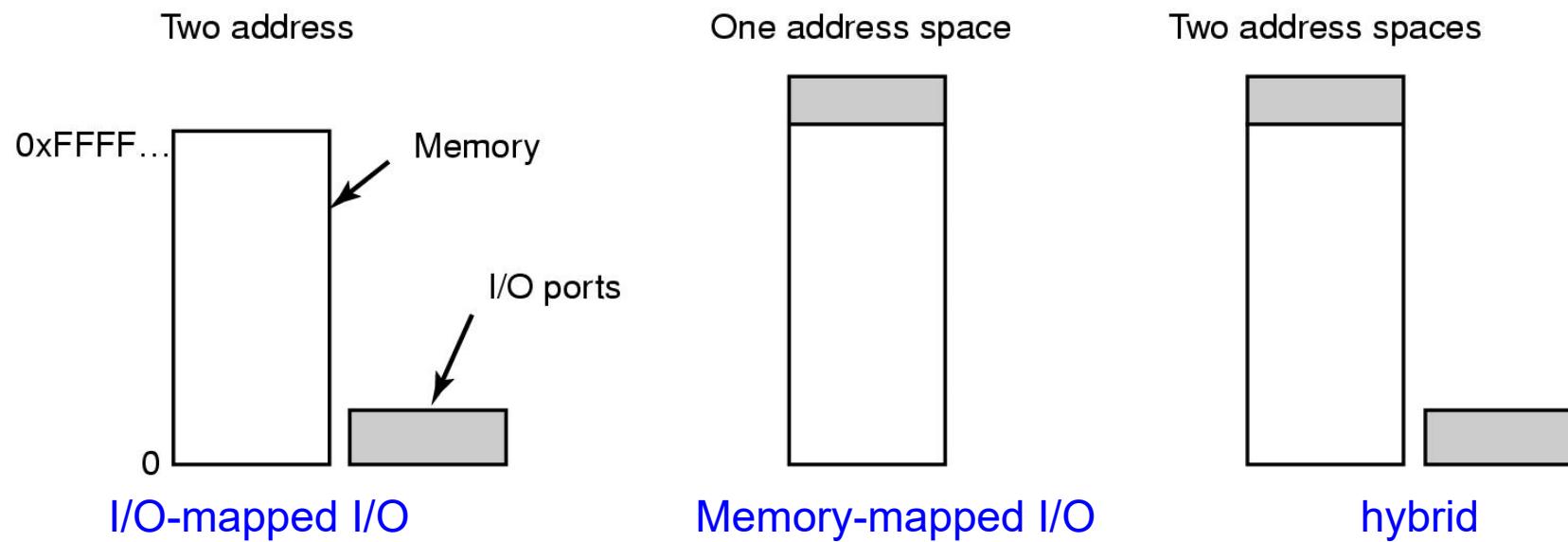
I/Oハードウェアの原理(5)

- CPUはどのようにして制御(コマンド, 状態)レジスタやデータバッファと通信を行うか?
- **I/O-(or port-)mapped I/O**
 - 各制御レジスタは8~16ビット整数のI/Oポート番号が割り当てられる
 - 以下のような専用I/O命令を使用してCPUは制御レジスタ(PORT)を読み出し, 結果をCPUのレジスタ(REG)に格納する
 - **IN REG, PORT**
 - 同様に, 以下のようにCPUレジスタ(REG)の値を制御レジスタ(PORT)に書き込む
 - **OUT PORT, REG**
 - 例) IBM 360とその後継機

I/Oハードウェアの原理(6)

□ Memory-mapped I/O

- 全ての制御レジスタをメモリ空間にマップする(例:PDP-11)
 - 各制御レジスタは専用のメモリアドレスが割り当てられる(そのアドレスに対応するメモリは存在しない). 通常のメモリ参照命令で読み書きする.
- I/O-mapped I/O と組合せ可能(ハイブリッド方式)
 - データバッファとは memory-mapped I/O で通信し, 制御レジスタとは I/Oポートで通信する(例:x86)



I/Oハードウェアの原理(7)

□ Memory-mapped I/Oの利点

■ I/O用の専用命令は必要ない

- アセンブリでプログラミングしなくても、例えばC言語でデバイスドライバを記述可能
- 一方、I/O-mapped I/Oでは専用命令(IN, OUT)を使用するためにアセンブリでの記述が強いられる

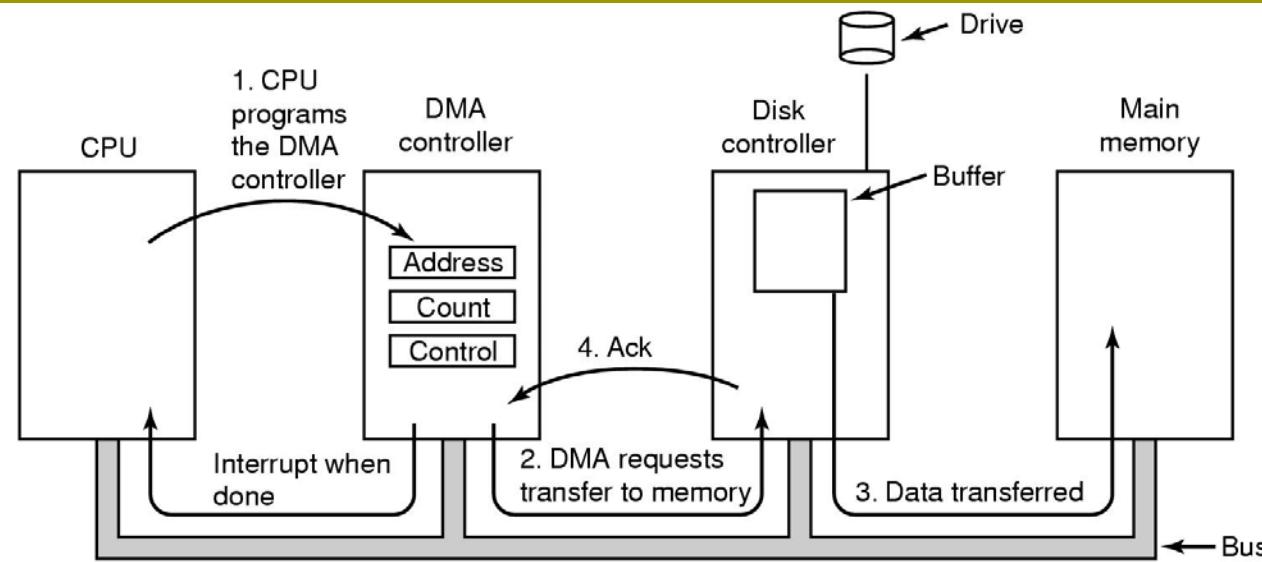
■ 特別な保護機構は必要ない

- ページング/仮想記憶の機構が通常のやり方で、I/Oレジスタやバッファに保護(ユーザが自由にアクセスできないように)をかけることが可能
- 一方、I/O-mapped I/Oでは専用命令(IN, OUT)が特権命令として用意されており、ユーザモードで実行すると例外が発生する

I/Oハードウェアの原理(8)

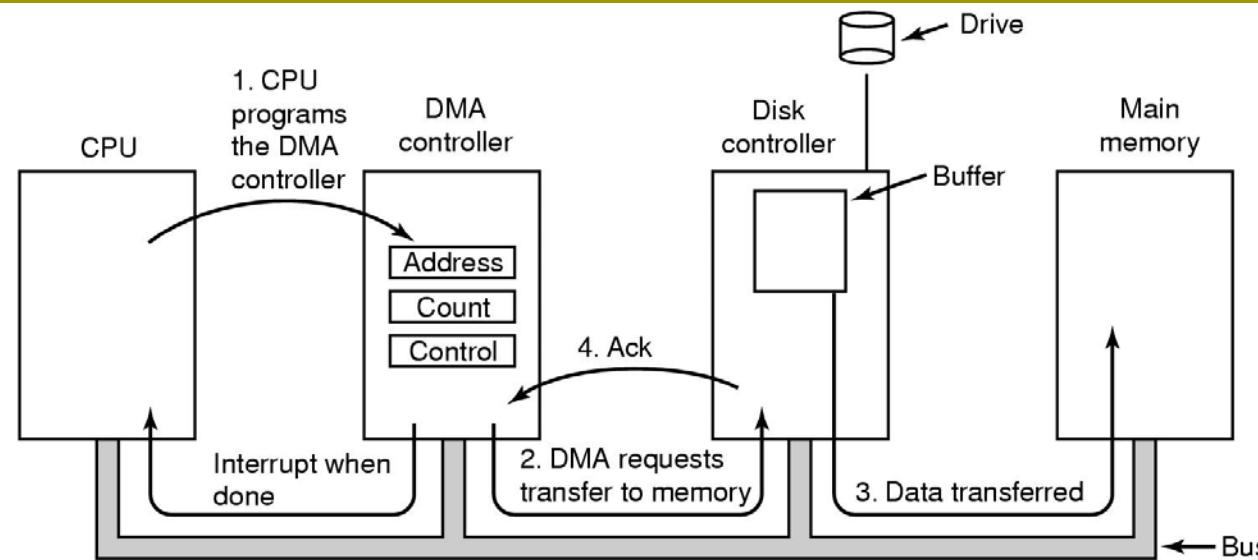
- CPUとI/Oコントローラ間でのデータ転送を1バイトあるいは1ワード単位で行うと、CPU時間を浪費することになる
- **Direct Memory Access (DMA)**
 - CPUの代わりにデータ転送を行う機構
 - DMAコントローラはいくつかのレジスタを持つ
 - それらのレジスタを使用して、CPUからDMAコントローラに命令や情報を送り、データ転送を起動する
 - 転送が完了したら、DMAコントローラはCPUに割込みをかける
 - 転送はCPUの処理のバックグラウンドで行われる
 - 例) ディスク読み出しの際のDMA(次スライド)

I/Oハードウェアの原理(9)



- ステップ1：DMAコントローラの各レジスタに値をセットすることにより、CPUはDMAに対して、何を何処へ、どのくらいのサイズを転送するのかを指示する
- ステップ2：DMAコントローラはバスを介してディスクコントローラに（基本サイズ分の）読み出し要求を発行することにより、転送を開始する

I/Oハードウェアの原理(10)

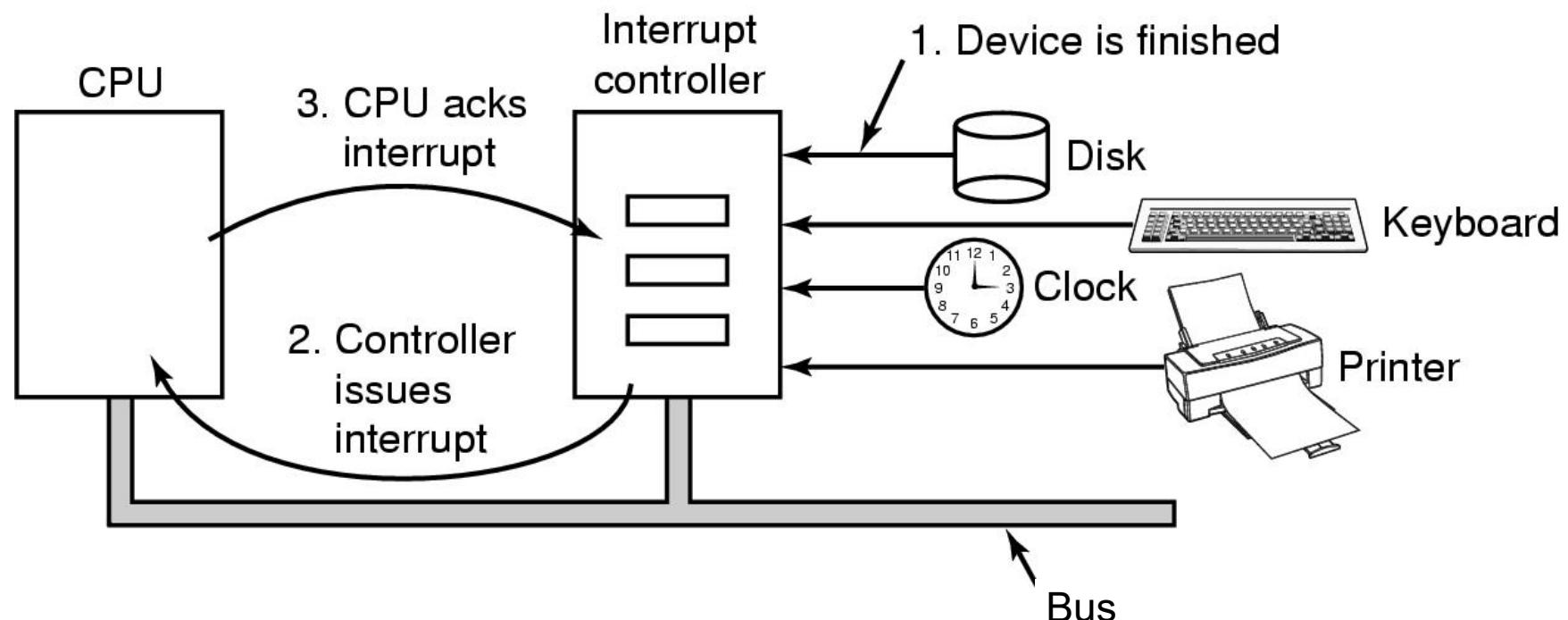


- ステップ3：メモリへの書き込みを行う
 - バス上でバースト転送を使用する
- ステップ4：書き込みが完了すると、ディスクコントローラはDMAコントローラに応答信号(acknowledgement signal)を送る
 - その後、DMAコントローラはメモリアドレス("Address")をインクリメントし、転送バイトカウント("Count")をデクリメントする
 - 転送バイトカウントがなお0より大きいときは、ステップ2から4を繰り返す。0に達したときは、CPUに割込みをかける

I/Oハードウェアの原理(11)

□ 割込みの確認

1. I/Oデバイスが与えられた仕事を完了したとき、割り当てられた信号線上に信号を出すことにより、割込み要求を発生させる



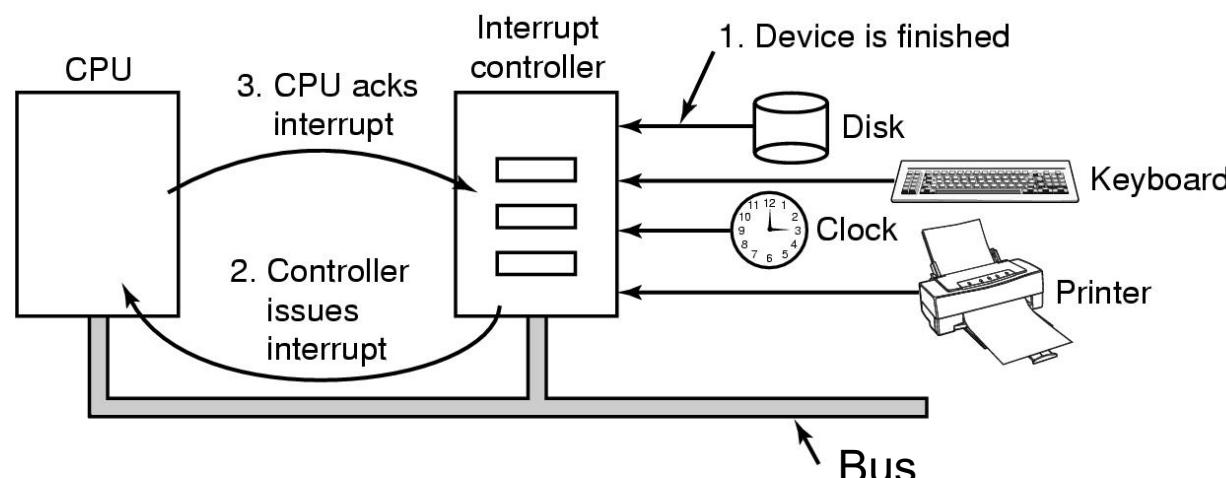
I/Oハードウェアの原理(12)

□ 割込みの確認(続き)

2. 他の未解決割込みが無ければ、割込みコントローラはCPUにその割込み要求をただちに伝え、CPU割込みが発生する

- 他の割込みの処理中、あるいは他のデバイスから同時により高優先度の割込みが発生していれば、そのデバイスは当面は無視される。その場合、割込み要求信号を出し続ける

割込み発生時のジャンプ先割込みハンドラの場所はハードウェアで固定されているか、あるいは専用レジスタ(OSが値をセット)で示される

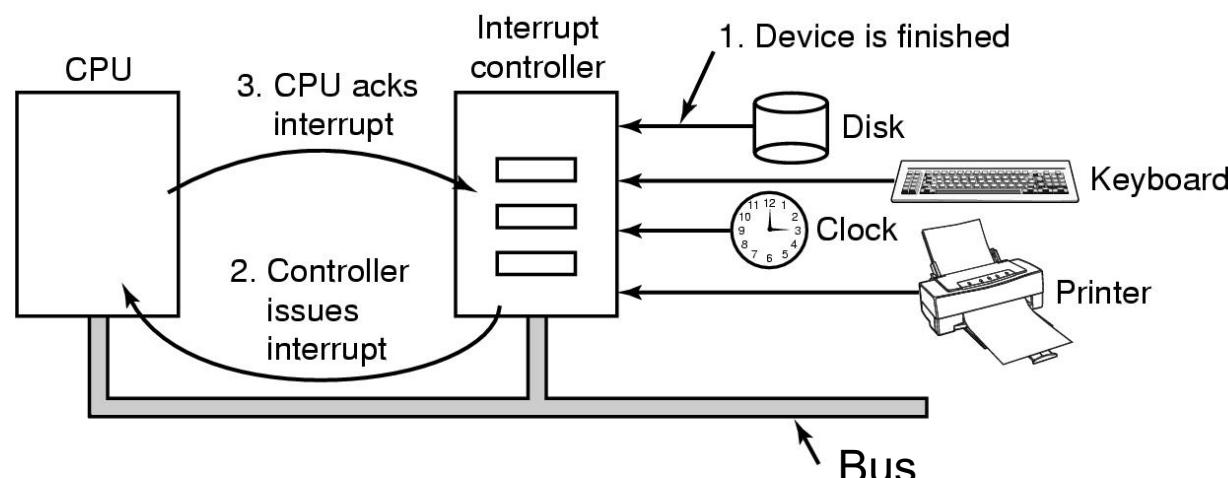


I/Oハードウェアの原理(13)

□ 割込みの確認(続き)

3. 割込みハンドラの実行がスタートした直後、割込みコントローラのレジスタの1つに値を書き込むことにより、割込み受理の応答をする
 - ・ 割込みコントローラはCPUへの割込み要求をオフにする

割込みサービスを開始する前に、後で割り込まれたプロセスに戻るのを可能とするために、割込みハンドラの開始時にコントキスト(PC, 他のレジスタ値など)を保存する必要がある



I/Oソフトウェアの原理(1)

□ I/Oソフトウェアの目標

■ デバイス独立(**Device independence**)

- 事前にデバイスを特定しなくても、いかなる I/Oデバイスにもアクセスできるプログラムを記述可能であることが必要
- 例) ファイルを入力とするプログラムは、ハードディスクあるいはDVD-ROM上のファイルを、それぞれのデバイス用にプログラムを変更することなく読むことができるべき
 - `sort < in.txt > out.txt`

■ 一様な名前付け(**Uniform naming**)

- ファイルやデバイスの名前は単に文字列や数字であるべきで、デバイスに依存すべきではない
- 全てのファイルやデバイスは、(マウントを適用した後に)同様なパス名によって場所を特定できるべき

I/Oソフトウェアの原理(2)

□ I/Oソフトウェアの目標(続き)

■ エラー処理(Error handling)

- ハードウェアが発生したエラーの処理を自身でできない場合は、デバイスドライバが処理・解決すべき
- 低位レイヤーでこの問題を扱えない場合に限って上位レイヤーに伝えるべき

■ 同期(Synchronous/blocking)転送 vs. 非同期(asynchronous/interrupt-driven)転送

- 同期転送： CPUはI/O処理が完了するまでブロックする
- 非同期転送： CPUは転送を開始した後、割込みが発生するまで他の仕事（同一プログラム内の他の仕事、あるいは他のプロセス）を実行する
- 非同期転送のほうがCPUの使用効率が良いが、I/O処理が“論理的に”ブロックしたほうがユーザプログラムが記述しやすい
 - リードシステムコールの後、データがバッファに到着するまでプログラムは自動的に中断し、他のプロセスに切り替えるべき
- ユーザプログラムに対して、実際は非同期(割込み駆動)である動作をブロッキングのように見せるのはOSの役目である

I/Oソフトウェアの原理(3)

□ I/Oソフトウェアの目標(続き)

■ バッファリング(Buffering)

- デバイスからのデータを最終的な場所に直接格納できないことがよくある
 - 例) ネットワークからパケットが到着したとき, OSはそのパケットをどこかに一時的に格納して中身を調べるまでは、何処に置くべきか(どのプロセスに渡すか)は不明である
 - バッファリングは多くのコピーを必要とし、I/Oの性能に大きな影響を与えることがよくある

■ 共有(Sharable)デバイス vs. 専用(dedicated)デバイス

- (ディスクのような)いくつかのI/Oデバイスは、同時にたくさんのユーザ／プロセスが使用しうる → 共有デバイス
- その他の(プリンタのような)デバイスでは、処理が終わるまで1人のユーザ(1つのプロセス)に占有されるべきである → 専用デバイス
 - 専用デバイスはデッドロックの問題を引き起こすこともある
- OSは共有デバイスと専用デバイスの両方を、問題が起こらないように操作できなければならぬ

I/Oソフトウェアの原理(4)

□ I/O処理には3つの異なる方法がある

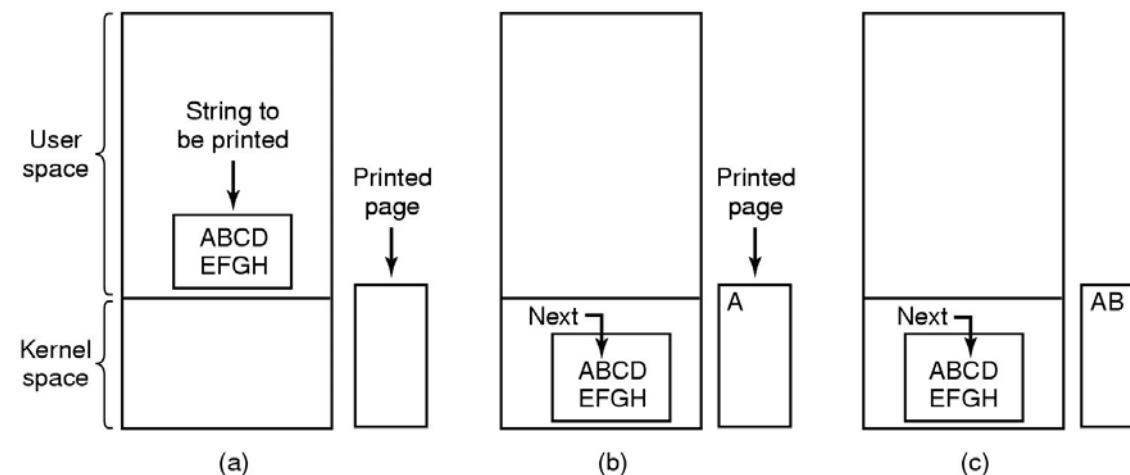
- (1) プログラムド(Programmed) I/O, (2) 割込み駆動(Interrupt-driven) I/O, (3) DMAを用いたI/O

□ プログラムドI/O(Programmed I/O)

- I/Oの最も単純な形であり, CPUが全ての仕事を行う

□ 例) 文字列 “ABCDEFGH” の印刷

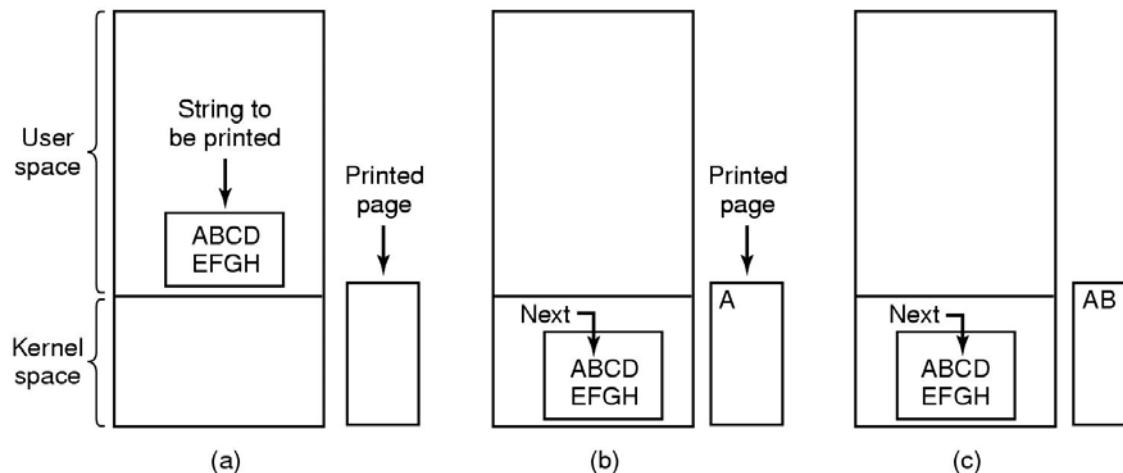
1. 最初に, ユーザ空間内のバッファで文字列を組み立てる
2. 続いて, openシステムコールによって印刷用プリンタを得る
3. OSに対して文字列を印刷するシステムコールを実行する



I/Oソフトウェアの原理(5)

- 例) 文字列 “ABCDEFGH” の印刷(続き)

4. OSは文字列が存在するバッファをカーネル空間の配列にコピーする
5. OSはプリンタが現在使用可能か(状態レジスタ)をチェックする
使用可能でなければ、使用可能となるまで待つ(ポーリング, polling)
6. OSは次に印刷すべき一文字をプリンタのデータレジスタに書き込む
7. 5 ~ 6を繰り返し、全ての文字を印刷する



プログラムI/Oは単純であるが、CPUをずっと拘束するという欠点がある

printシステムコールが呼び出されたときに実行されるコード

```
copy_from_user(buffer, p, count);
for (i = 0; i < count; i++) {
    while (*printer_status_reg != READY) ;
    *printer_data_register = p[i];
}
return_to_user();
```

/* p is the kernel buffer */
/* loop on every character */
/* loop until ready */
/* output one character */

I/Oソフトウェアの原理(6)

□ 割込み駆動I/O(Interrupt-Driven I/O)

- 例) プリンタは毎秒100文字印刷できる(各文字に10ミリ秒かかる)と仮定
 - プログラムドI/Oでは、次の文字が出力できるまで、CPUは10ミリ秒アイドルループを回って待つことになる
 - CPUはコンテキスト切り替えをして他のプロセスを実行したほうが良い

printシステムコールが呼び出さ
れたときに実行されるコード

```
copy_from_user(buffer, p, count);
enable_interrupts();
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler(); プロセスをブロックし、プロセス切替え
```

プリンタが文字を印刷し、次の文字を受け取る準備ができ
たとき、割り込みを発生させる

(a)

割込みサービスルーチン

```
if (count == 0) {
    unblock_user();
} else {
    *printer_data_register = p[i];
    count = count - 1;
    i = i + 1;
}
acknowledge_interrupt();
return_from_interrupt();
```

(b)

I/Oソフトウェアの原理(7)

- 割込み駆動I/Oの欠点は、全ての文字で割込みが発生すること

□ DMAを用いたI/O

- プログラムドI/Oの一種であるが、メインのCPUの代わりに、DMAコントローラが（ほとんど）全ての仕事を行う
- 割込み発生回数を減らせる（1文字毎の割込み発生から、バッファ全体の印刷後の一回の割込みに減らせる）

printシステムコールが呼び出されたときに実行されるコード

```
copy_from_user(buffer, p, count);
set_up_DMA_controller();
scheduler();
```

(a)

割込みサービスルーチン

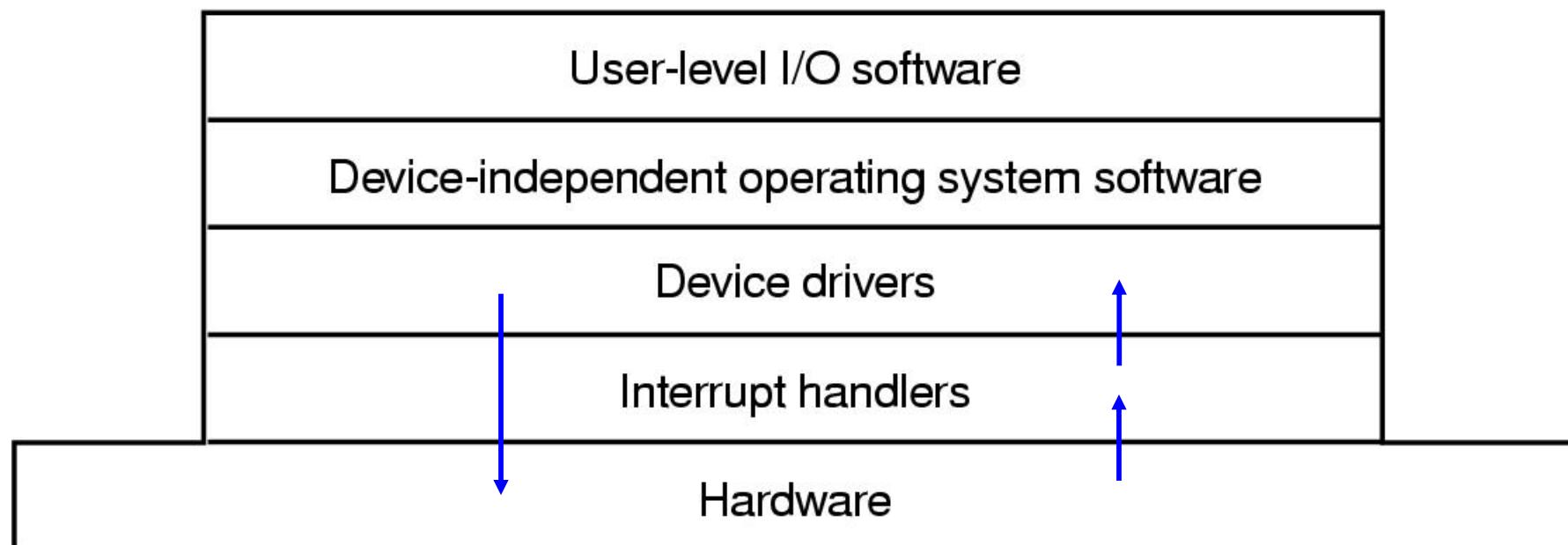
```
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

(b)

I/Oソフトウェアの階層(1)

□ I/Oソフトウェアシステムの階層

- 各レイヤーは、明確に定義された機能と、隣接するレイヤーとの間の明確に定義されたインターフェースを持つ
- 機能とインターフェースはシステムによって異なる



I/Oソフトウェアの階層(2)

□ 割込みハンドラ(Interrupt Handlers)

- I/O処理を開始したデバイスドライバは、そのI/Oが完了して割込みが発生するまで(そのプロセスを)ブロックする。(例えばセマフォを利用)



- 割込みが発生した際、割込みハンドラはその割込みを処理するためにすべきことをする

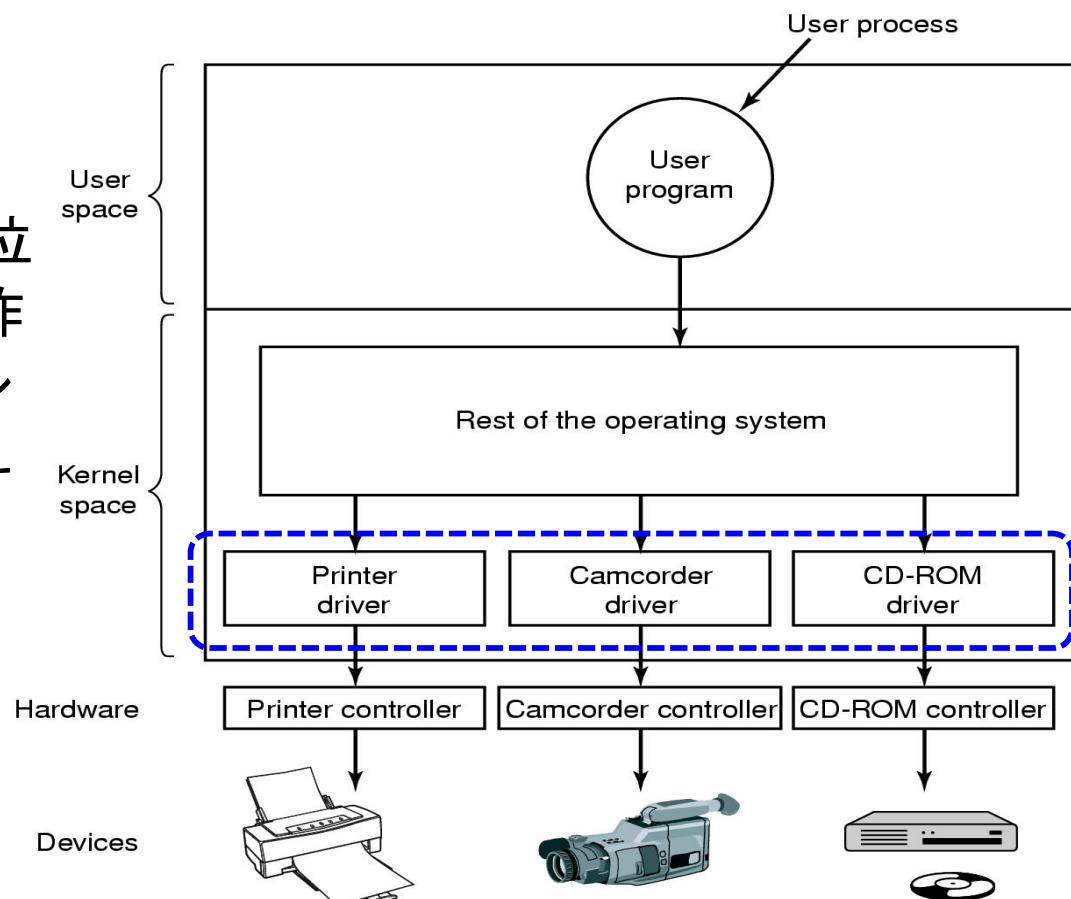


- その後、そのI/Oを開始したデバイスドライバ(プロセス)のブロックを解除できる
 - ブロックおよびその解除は、セマフォ(down, up)やモニタ(wait, signal), メッセージ交換(receive, send)などで可能

I/Oソフトウェアの階層(3)

□ デバイスドライバ(Device Drivers)

- I/Oデバイスを制御するための、デバイスに特化したコード
- (一般的に)デバイスの開発者によって記述されたデバイスドライバが、OSにインストールされる
- インストールに必要なもの
 - ドライバが何を行うか、OSの(上位の)残りの部分とどのように相互作用するかを明確に定義したモデル
 - そのようなインストールを可能とするアーキテクチャ



I/Oソフトウェアの階層(4)

□ デバイスドライバ(Device Drivers)(続き)

- OSは通常、ドライバを2つのカテゴリに分類している
 - ブロックデバイス用
 - アドレス可能な複数のデータブロックからなる。(ディスクなど)
 - キャラクタデバイス用
 - キャラクタのストリームを生成あるいは受け付ける。(キーボード、プリンタなど)
- OSは、全てのブロックデバイスと全てのキャラクタデバイスのドライバがサポートすべき標準インターフェースをそれぞれ定義している
 - これらのインターフェースとして多くの手続きが存在する。
 - 典型的な手続きとして、(ブロックデバイスの)ブロックを読むものや、(キャラクタデバイスに)文字列を書き込むものがある
 - OSの残りの部分(上位レイヤー)は、ドライバに仕事をさせるためにそれらの手続きを呼び出す

I/Oソフトウェアの階層(5)

□ デバイスドライバ(Device Drivers)(続き)

- デバイスドライバの最も明白な機能は、上位レイヤーであるデバイス独立なソフトウェアからの抽象的な読み書きの要求を受け取り、(それをデバイスコントローラレジスタへの読み書き／コマンド列に変換することにより)実行することである
 - 他の機能として、デバイスの初期化や電力管理、イベントログなどがある
- デバイスを制御するということは、デバイスへコマンド列を発行することを意味する
 - ドライバは、なされるべきことにしたがって、コマンド列が定義されている場所である
 - コマンド列は順番にコントローラのレジスタに書かれる

I/Oソフトウェアの階層(6)

□ デバイスドライバ(Device Drivers)(続き)

- コマンドが発行された後,
 - デバイスドライバは自身をブロックし, 割込みが到着したら解除される(非同期)か, あるいは,
 - I/O処理がナノ秒オーダーで終了する場合は, I/O処理の完了を待つ(同期)
- 処理が完了した後, ドライバはエラーのチェックをする
- 全てが正常(エラー無し)のときは, ドライバは自身を呼び出したデバイス独立ソフトウェアにデータを(あれば)渡す
- デバイスドライバは再入可能(reentrant)でなければならない。
(例えば, 実行中のドライバが終了する前に, 同じドライバが別のプロセスのために再度呼び出されることを想定していることを意味する)

I/Oソフトウェアの階層(7)

□ デバイス独立(Device-Independent)I/Oソフトウェア

- 低レイヤーのI/Oソフトウェア(デバイスドライバ, 割込みハンドラ)はデバイス特有であるが, 上位レイヤーのソフトウェアはデバイス独立である
- デバイス独立I/Oソフトウェアの役割は, 全てのデバイスに共通なI/O機能を実行(= ユーザレベルソフトウェアに統一したインターフェースを提供)することである

デバイス独立I/Oソフトウェアの機能

デバイスドライバ統一インターフェース
バッファリング
エラーレポート
専用デバイスの割当てと開放
デバイス独立なブロックサイズの提供

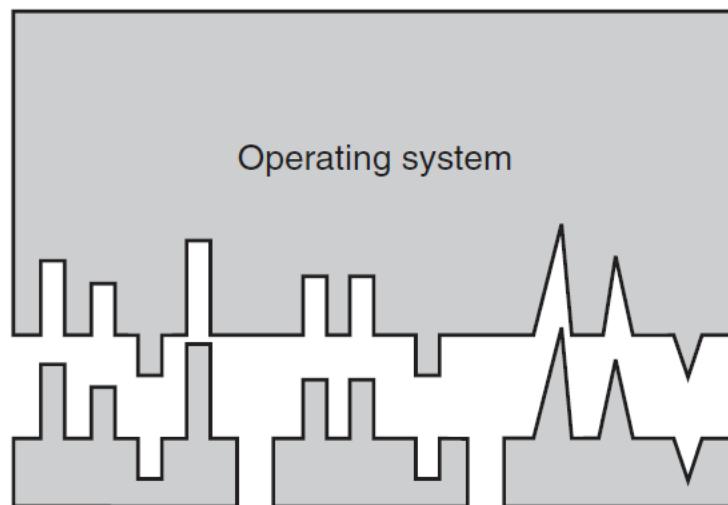
I/Oソフトウェアの階層(8)

■ デバイスドライバ統一インターフェース

- もしデバイスドライバが上位レイヤーに対して異なるインターフェースであれば、新しいデバイスを加えるときはいつも、そのデバイスのためにOSを手直し（再コンパイル、再構築）しなければならない
- したがって、全てのI/Oデバイスやドライバが、上位レイヤーから同じように見えるようにしたい

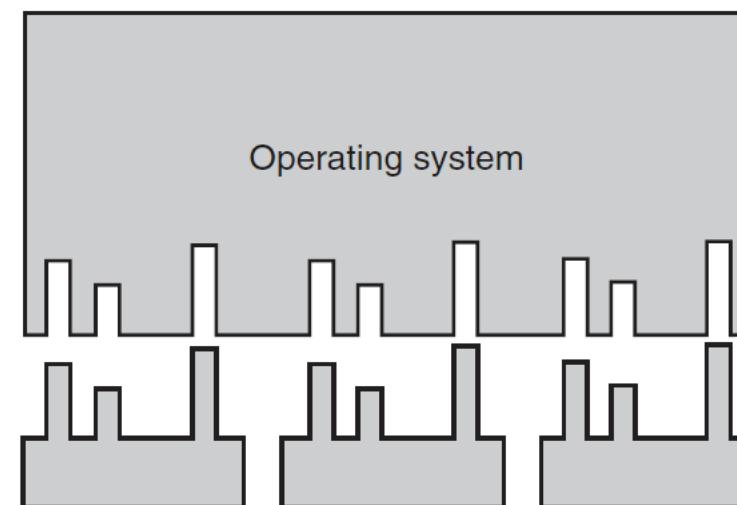
⇒ 全てのI/Oデバイスドライバを同じインターフェースにする

標準のドライバインターフェースが無い場合



SATA disk driver USB disk driver SCSI disk driver

標準のドライバインターフェースがある場合



SATA disk driver USB disk driver SCSI disk driver

I/Oソフトウェアの階層(9)

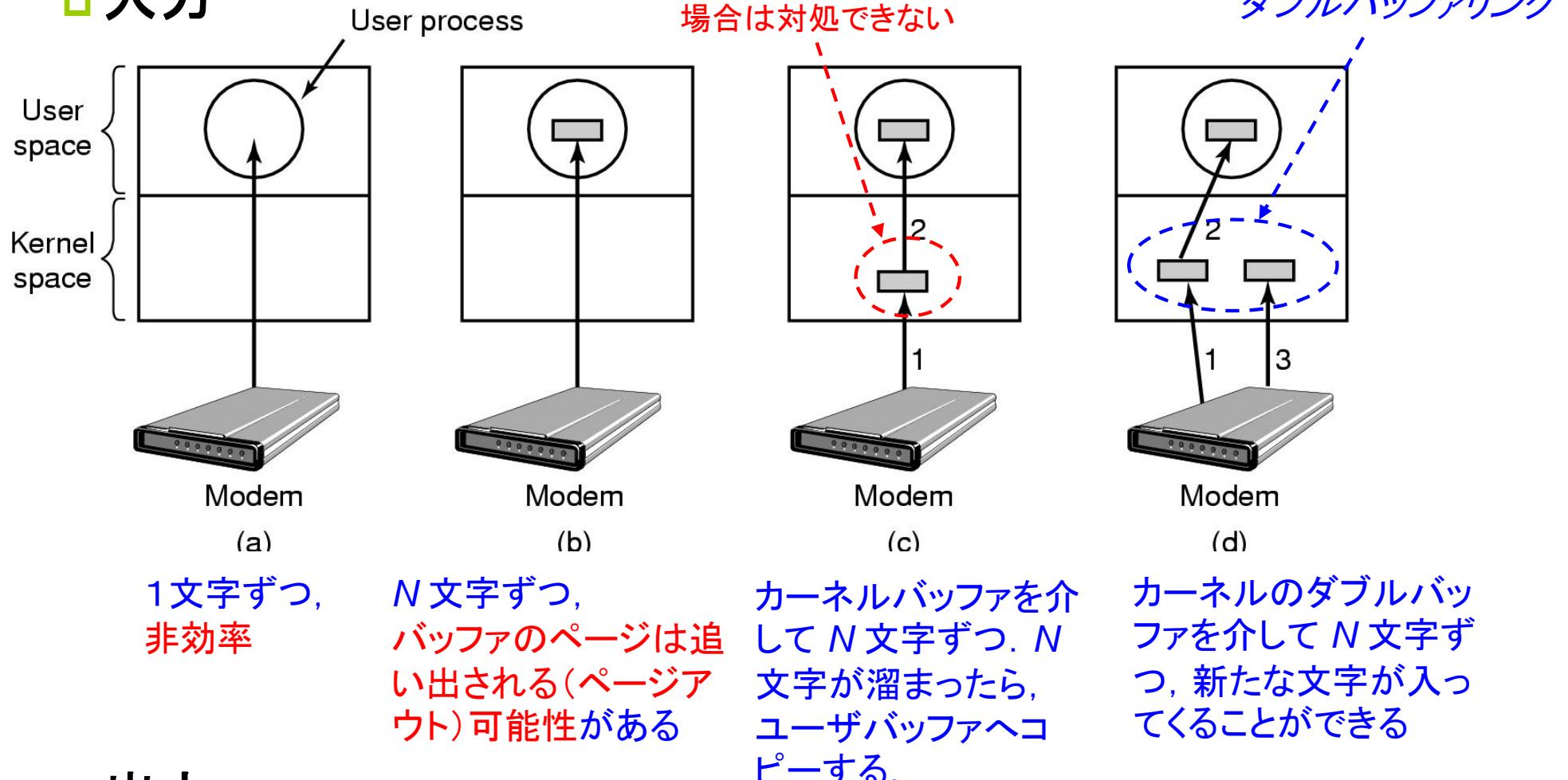
■ デバイスドライバ統一インターフェース(続き)

- デバイスのタイプごとに、ドライバの必須機能インターフェースを定義
 - 例えばディスクタイプならば、read, write, power on/off, formatting, etc.
- 上位レイヤー(デバイス独立ソフトウェア)が記号からなるデバイス名を実際のドライバにマッピングする方法
 - 例) UNIXでは、/dev/disk0 のようなデバイスファイル名は特殊ファイルの i-nodeをユニークに特定しており、この i-node には対応するドライバを選択するための情報(メジャー／マイナーデバイス番号)が記録されている
- 保護(Protection)
 - UNIXとWindowsの両方において、デバイスは名前が付けられたファイル(/dev/disk0 など)としてファイルシステム内に存在する。これにより、ファイルのための通常の保護規則がI/Oデバイスにも同様に適用される。すなわち、適切な設定により、ユーザが直接I/Oデバイスをアクセスできないようにしている

I/Oソフトウェアの階層(10)

■ バッファリング

□ 入力



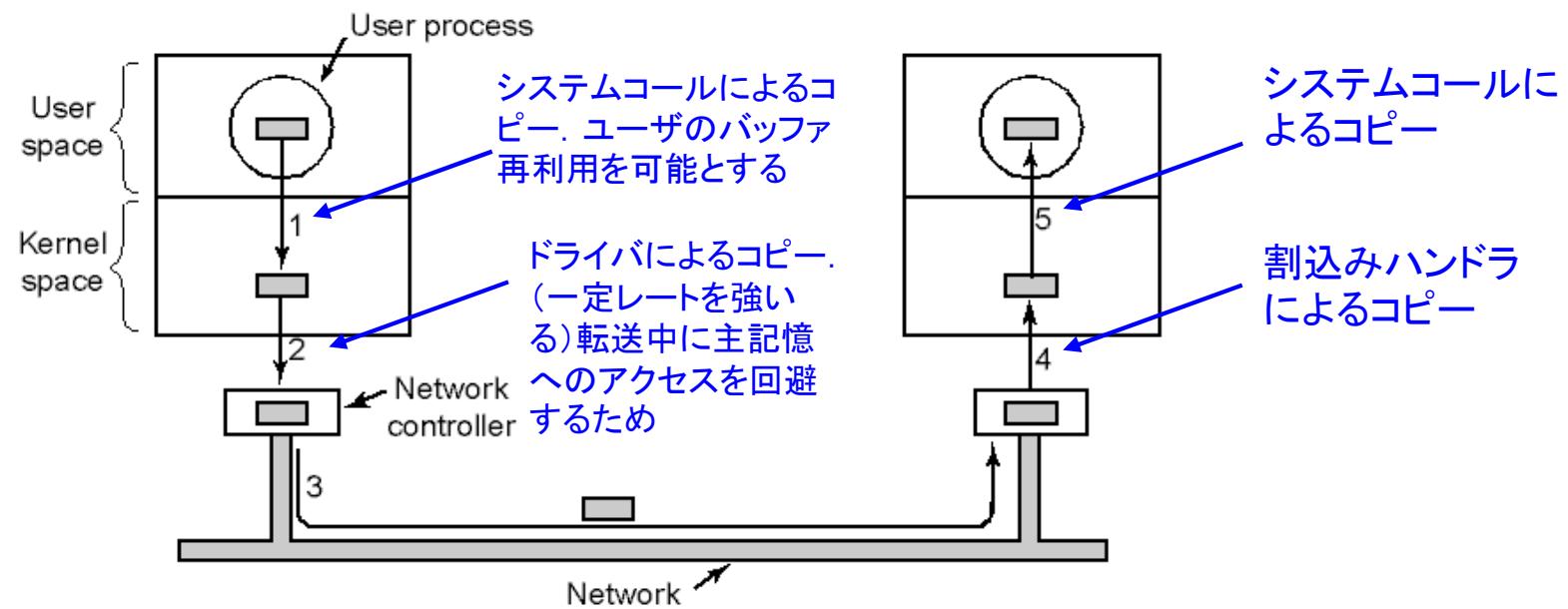
□ 出力

- プロセスがI/Oが終了するのを待つことなしに、ユーザバッファを再利用するた
めには、カーネルバッファが不可欠

I/Oソフトウェアの階層(11)

■ バッファリング(続き)

□ 例) ネットワーク送信



- これらのコピー処理全てが、逐次的に行われる必要があるため、転送レートを相当落とすことになる

I/Oソフトウェアの階層(12)

■ エラーレポート

- 多くのエラーはデバイス固有のものであり、ドライバで処理しなければならないが、デバイス独立なエラーもある
- I/Oエラーのクラス
 - プログラミングエラー(Programming errors)
 - 例) 入力デバイスへの書き込み、出力デバイスからの読み出しなど
 - 例) 無効なバッファアドレスやパラメータを渡したとき
 - 例) 無効な(存在しない)デバイスを指定したとき
 - これらのエラーに対する動作としては、単にエラーコードを報告するのみ
 - 実際のI/Oエラー(Actual I/O errors)
 - 例) 故障しているディスクのブロックに書き込もうとしたとき
 - 例) 電源が入っていないビデオカメラから読み出そうとしたとき
 - 何をすべきかの決定はドライバにまかせられる
 - ドライバが何をしてよいのかわからない場合は、問題をデバイス独立ソフトウェアに渡すことも可能な場合がある → 大抵はシステムコールをエラーで終了させる

I/Oソフトウェアの階層(13)

■ 専用デバイスの割当てと開放

- プリンタ、DVD-Rレコーダなどのいくつかのデバイスは、いかなるときも1つのプロセスからのみ使用可能
- デバイス使用の要求を調べて、許可したり拒否するのはOSの役目である
- 簡単な方法として、プロセスに直接デバイスの特殊ファイルをオープン(open)させる
 - デバイスが使用不可の場合は、そのオープンを失敗させてエラーでリターンするか、あるいはプロセスをロックさせる

■ デバイス独立なブロックサイズ

- 異なるディスクは異なるセクタサイズかもしれない
- キャラクタデバイスタイプでも、マウスのように1バイトずつ転送するものや、Ethernetインターフェースのように大きな単位で転送するものがある
- 実際の物理的な転送サイズが異なるという事実を隠し、高位レイヤーに統一した(論理的な)ブロックサイズを見せることは、デバイス独立ソフトウェアの役目である(実際は、デバイスドライバと協調して物理サイズを隠蔽)

I/Oソフトウェアの階層(14)

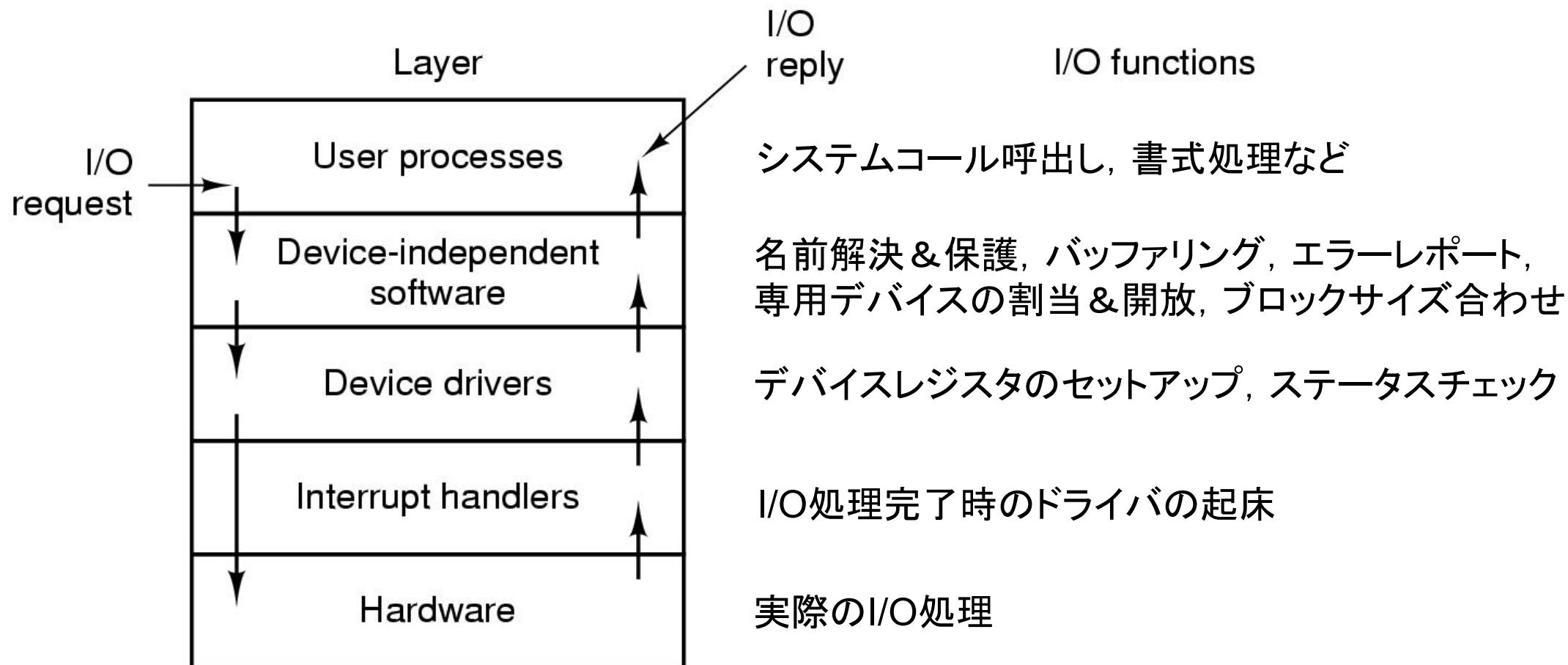
□ ユーザ空間(User-Space) I/Oソフトウェア

- ほとんどのI/OソフトウェアはOSの内部にあるが、一部はカーネル外で実行されるライブラリ(libraries)からなる
- 通常、ライブラリ手続きによってシステムコールが呼ばれる
 - 手続きは通常、システムコールのための適切なパラメータの準備以上のこと はほとんど行わない
 - しかし、実際に仕事を行うI/O手続きもある
 - 例) printf や scanf における書式処理

```
printf ( "Val=%d", 7 ); → write ( 1, "Val=7", 5 );
```

I/Oソフトウェアの階層(15)

□ I/Oシステムのレイヤーと、各レイヤーの主な機能



ファイルシステム



ストレージの3つの本質的な要件

- コンピュータが扱う情報に対する要求
 1. きわめて大量の情報を格納可能であること
 2. 情報は使用するプロセスが終了した後でも残り続けること
 3. 複数のプロセスが同時に情報にアクセスできること
- これら全ての問題への通常の解決法は、情報をファイル(files)という単位でディスクや他の外部メディアに格納することである
- OSのうち、ファイルを扱う部分がファイルシステム(file system)である

ファイル(1)

□ ファイルは抽象的な存在であることが重要

- OSは情報をディスク等に格納し、後で読み出せる方法を提供する
- 情報がどのようにしてどこへ格納されているか、実際にどのようにディスクが動いているのかの詳細をユーザに対して隠すべき

□ ファイルの名前付け(File Naming)

- プロセスがファイルを生成するとき、ファイルに名前を与える
- プロセスが終了した後、ファイルは存在しつづけ、その名前を使用して他のプロセスがアクセスできる
- 多くのファイルシステムで、255文字(アルファベット、数字、特殊記号)までの名前がサポートされている
 - UNIXでは大文字と小文字が区別されるが、MS-DOSでは区別されない

ファイル(2)

□ ファイルの名前付け(File Naming)(続き)

- 多くのOSで, “prog.c”のようにピリオドで分割される2つの部分からなるファイル名をサポートしている
 - ファイル拡張子(**File extension**)：ピリオドに続く部分で, ファイルについての情報を示す
 - 例) MS-DOSでは, ファイル名は1～8文字に, 1～3文字の拡張子を付加できる. UNIXでは, 拡張子のサイズはユーザにまかせられ, ファイルは2つ以上の拡張子を持つこともできる(prog.c.Z)
- あるシステム(例, UNIX)では, ファイル拡張子は慣習にすぎず, OSが強いるものではない
- 対照的に, Windowsは拡張子を認識し, 意味を割り当てている
 - ユーザはOSに拡張子を登録し, 各拡張子にどのプログラムがその拡張子を「所有」しているかを指定する

ファイル(3)

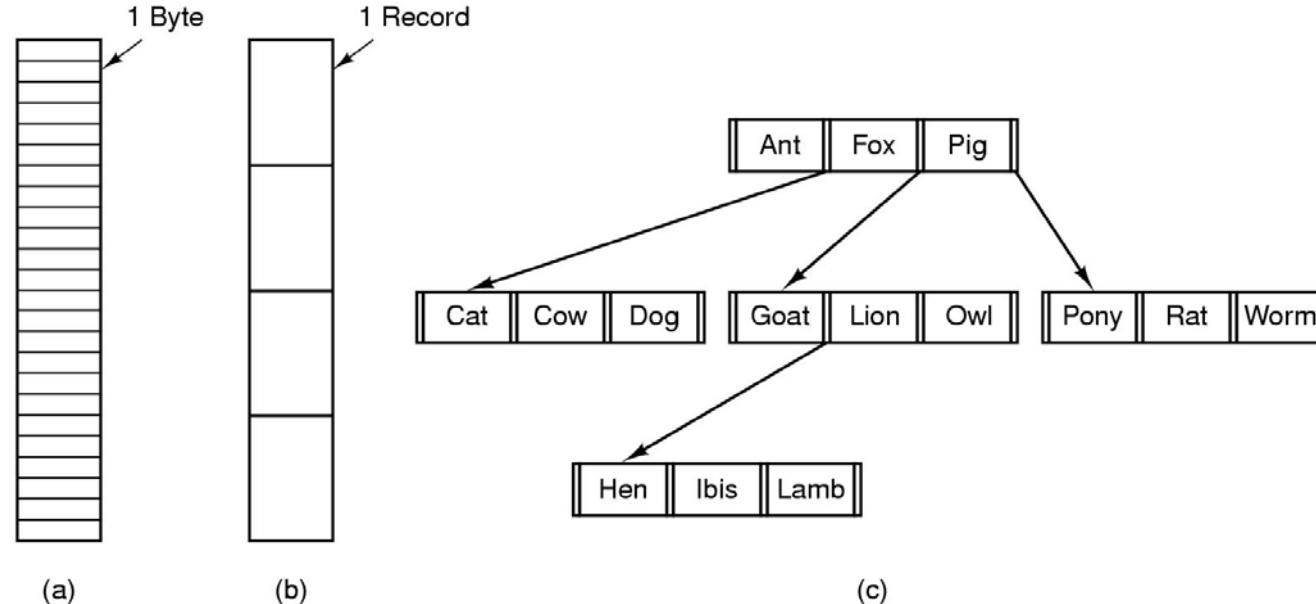
■ 典型的なファイル拡張子

Extension	Meaning
.bak	Backup file
.c	C source program
.gif	Compuserve Graphical Interchange Format image
.hlp	Help file
.html	World Wide Web HyperText Markup Language document
.jpg	Still picture encoded with the JPEG standard
.mp3	Music encoded in MPEG layer 3 audio format
.mpg	Movie encoded with the MPEG standard
.o	Object file (compiler output, not yet linked)
.pdf	Portable Document Format file
.ps	PostScript file
.tex	Input for the TEX formatting program
.txt	General text file
.zip	Compressed archive

ファイル(4)

□ ファイル構造

■ ファイル構造の一般的な3つの例



- (a) バイト列: OSはファイル内に何があるか知らないか、気にしない。全ては単にバイトデータである。ユーザプログラムによって意味が与えられる(UNIX, Windows)
- (b) レコード列: ファイルは固定長レコードの列であり、各レコードは何らかの内部構造を持っている。(歴史的には、パンチカードやラインプリンタの行単位に基づく)
- (c) 木: ファイルはレコードの木からなり、レコードは必ずしも同じ長さではなく、各レコードは決まった場所にキーのフィールドを持つ。木はキーのフィールドでソートされ、速やかな検索を可能とする(メインフレーム計算機)

ファイル(5)

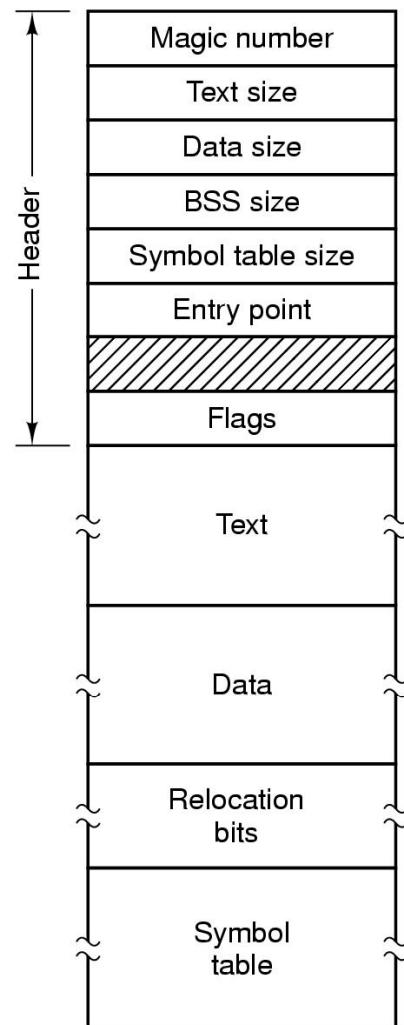
□ ファイルの種類

- レギュラーファイル(Regular files)
 - ユーザ情報／アプリケーションデータを含むファイル
 - 一般的に、ASCIIファイルかバイナリファイル
- ディレクトリ(Directories)
 - ファイルシステムの構造を維持するためのシステムファイル
- キャラクタ型特殊ファイル(Character special files)
 - 入出力に関連するファイルであり、ターミナルやプリンタ、ネットワークなどのシリアル入出力デバイスをモデル化するために使用される
- ブロック型特殊ファイル(Block special files)
 - ディスクをモデル化するために使用される
- 本章では、主にレギュラーファイルとディレクトリに注目する

ファイル(6)

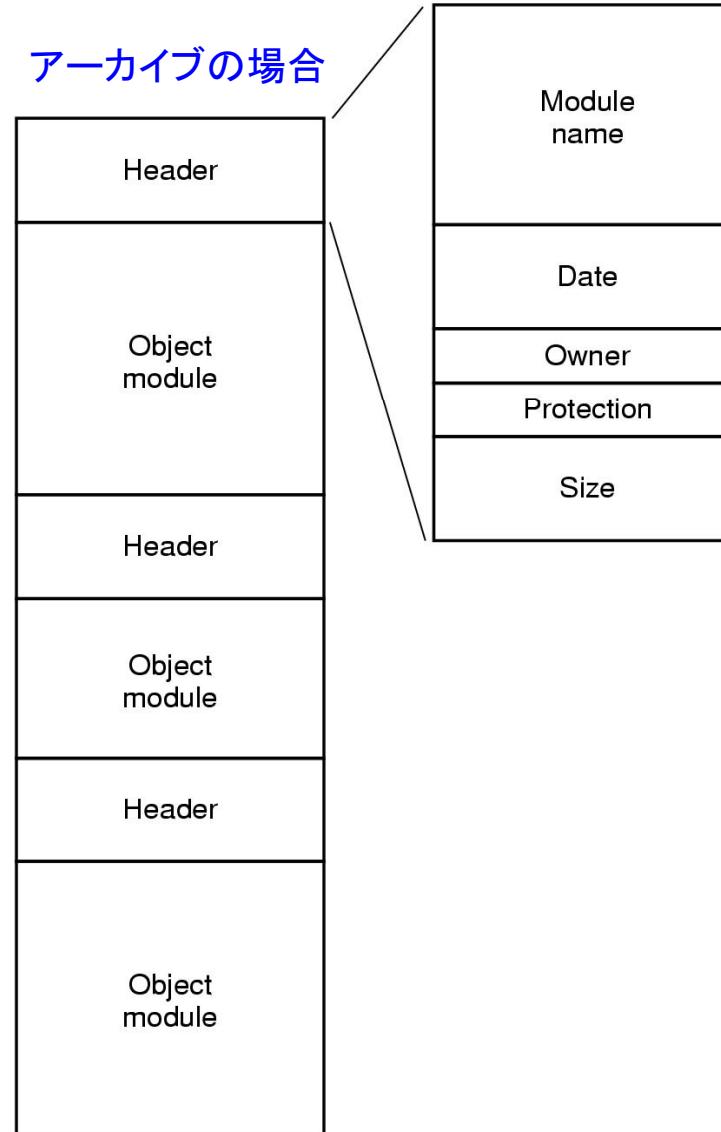
■ レギュラーファイル(バイナリファイル)の例

オブジェクト／実行ファイルの場合



(a)

アーカイブの場合



(b)

ファイル(7)

□ ファイルアクセス

■ 順アクセス(**Sequential access**)

- プロセスはファイル内のバイト, あるいはレコードを, 最初から順番に読むことができるが, 途中を飛ばしたり, 順不同に読むことはできない — ディスクではなく, 磁気テープ等が該当する

■ ランダムアクセスファイル(**Random access files**)

- プロセスはファイル内のバイト, あるいはレコードを, 順不同に読み書き可能
- データベースシステムなど, 多くのアプリケーションで必要とされる
- どこから読み始めるかを指定するために, 2つの方法が使われる
 1. **read** 操作毎に, ファイル内の読み始める位置を指定する
 2. **seek** 操作で現在の位置を指定する. seek後に, ファイル内のその現在の位置から順番に読み出すことができる

■ 現行のOSはランダムアクセスファイル(with seek操作)を使用している

ファイル(8)

□ ファイル属性

- 全てのファイルは名前とデータを持つ
- 加えて、全てのOSで、各ファイルに他の情報を関連付けている →
ファイルの属性(**attributes/metadata**)
 - 例) ファイルが生成された日にちと時刻、ファイルのサイズ、保護情報、所有者情報、書き込み不可(read-only)情報、など
- どのような属性があるかはシステムによって大きく異なる
- 次のスライドはその一例であり、他の属性も存在する

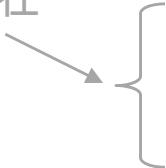
ファイル(9)

□ ファイル属性(続き)

これら全てを持つシステムは存在しない

キーを使ってレコードが参照されるようなファイルにのみ存在

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to



ファイル(10)

□ ファイル操作

■ ファイルに関連した最も一般的なシステムコール

□ create

- データ無しでファイルを生成し、属性のいくつかを設定する

□ delete

- ファイルを削除し、ディスクスペースを開放する

□ open

- ファイルを使用する前に、プロセスはファイルをオープンしなければならない。これにより、システムはファイルの属性とディスクアドレスをメモリに読み込み、後の操作で高速なアクセスが可能となる

□ close

- ファイルをクローズして、メモリ内に読み込んでいた属性 & ディスクアドレスを開放する

ファイル(11)

□ ファイル操作(続き)

□ read

- ファイルからデータが読み出される。通常は、現在位置からバイトが読み出される。呼出し元は必要なデータサイズを指定し、読み込むバッファを与える

□ write

- データを、ファイルに(現在の位置に)書き込む。現在の位置がファイルの終端である場合は、ファイルのサイズが増大することになる。現在の位置がファイルの途中である場合は、古いデータは上書きされ、永遠に消えることになる

□ append

- ファイルの終端へのデータ書き込みを行う(制限されたwrite)

□ seek

- ファイル内の現在位置を指すポインタの値を変更する。この呼出しが完了した後、新しい位置からのデータ読み出し、新しい位置へのデータ書き込みが行える

ファイル(12)

□ ファイル操作(続き)

□ get attributes

- ファイルの属性を読み出す
- 例) make が実行されたとき、全てのソースとオブジェクトファイルの属性の中の更新時刻が検査され、全てを最新にするのに必要な最小のコンパイルが行われる。

□ set attributes

- 属性のいくつかはユーザが設定したり変更することが可能であり、この呼び出しにより行われる

□ rename

- このシステムコールにより、既に存在するファイルの名前を変更できる

ファイル(13)

□ ファイルに関するシステムコールを使用したプログラムの例

■ あるファイルから別のファイルへのコピーを行うプログラム

▣ > copyfile abc xyz (Copying the file "abc" to a new file "xyz")

```
int main (int argc, char *argv[])
{
    int  in_fd, out_fd, rd_count, wt_count; /* file descriptors & read/write counts */
    char buffer[BUF_SIZE] /* #define BUF_SIZE 4096 */

    if (argc != 3) /* syntax error if argc is not 3 */
        exit (1); /* any status code other than 0 means error occurrence */
    /* open the input file and create the output file */

    if ( ( in_fd = open ( argv[1], O_RDONLY ) ) < 0 ) /* open the source file */
        exit (2);

    if ( ( out_fd = creat ( argv[2], S_IWRITE ) ) < 0 ) /* create the destination file */
        exit (3);
    /* → continued to next slide */
```

ファイル(14)

□ ファイルに関するシステムコールを使用したプログラムの例 (続き)

```
/* continued */
/* copy loop */
while ( TRUE ) {

    if ( ( rd_count = read ( in_fd, buffer, BUF_SIZE ) ) <= 0 ) /* read a block of data */
        break;

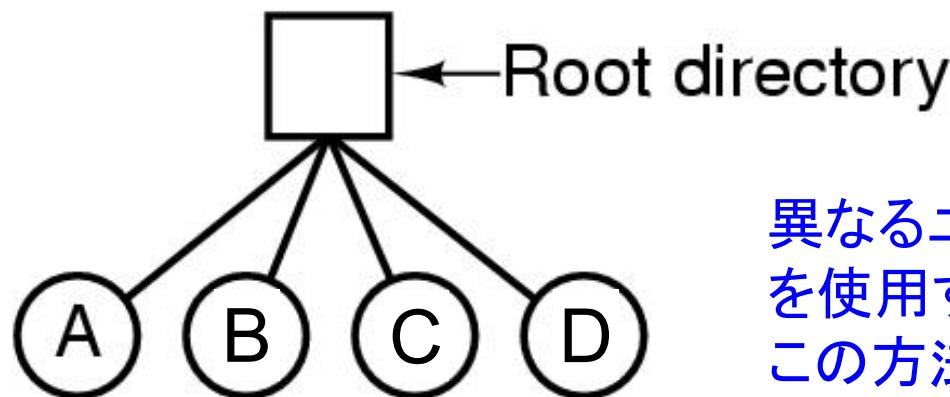
    if ( ( wt_count = write ( out_fd, buffer, rd_count ) ) <= 0 ) /* write data */
        exit (4);

}

/* close the files */
close ( in_fd );
close ( out_fd );
if ( rd_count == 0 ) /* no error on last read */
    exit ( 0 );
else
    exit ( 5 );
}
```

ディレクトリ(1)

- ファイルを管理するために、ファイルシステムは通常、ディレクトリ(フォルダ)を持つ。多くのシステムにおいて、ディレクトリはそれ自身ファイルである
- 単層型ディレクトリシステム
 - 1つのディレクトリが全てのファイルを含む
 - ルートディレクトリ(**root directory**)と呼ばれることがある
 - 単純な設計

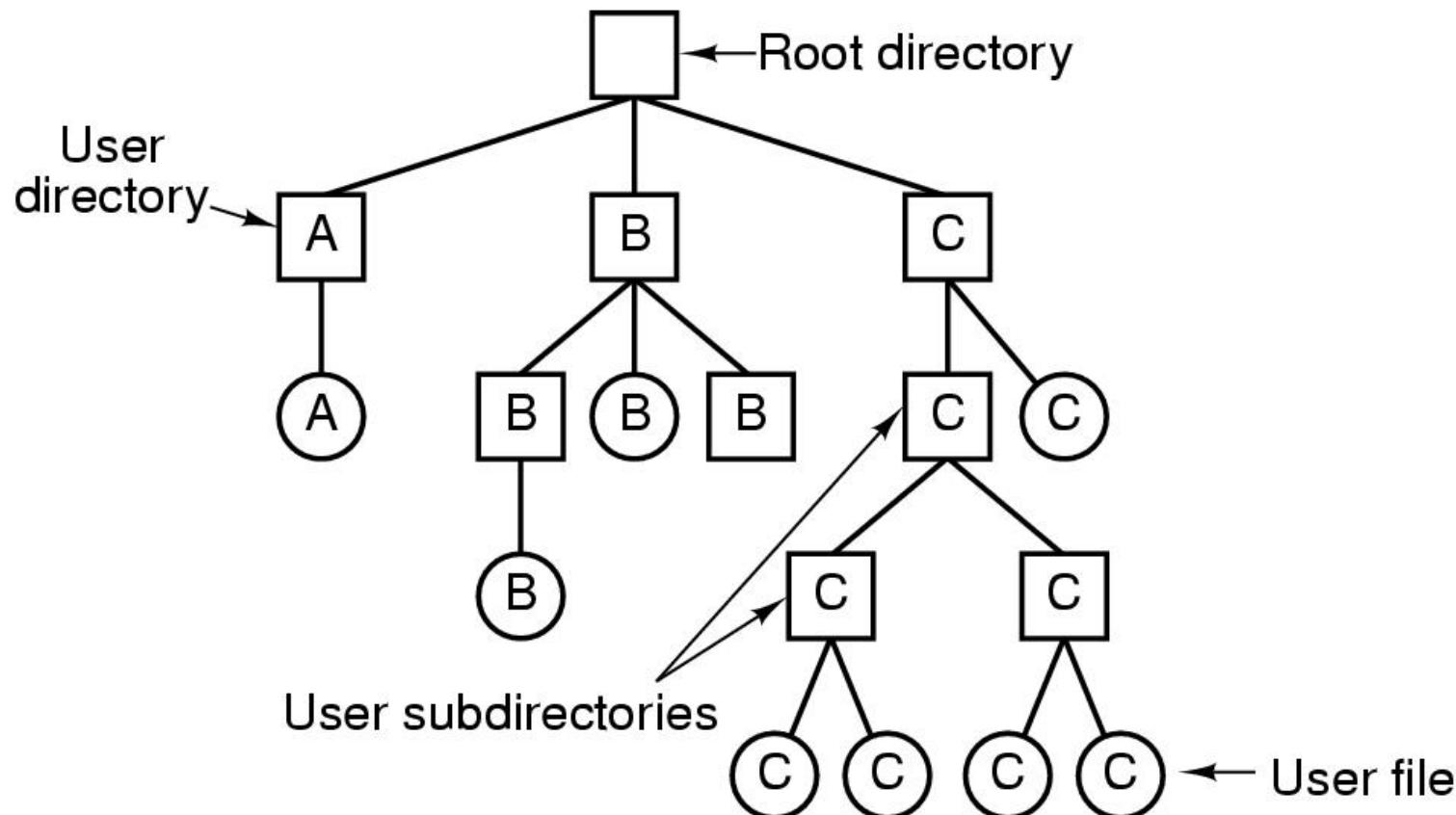


異なるユーザがたまたま同じファイル名を使用するかもしれない。したがって、この方法はマルチユーザのシステムでは不便

ディレクトリ(2)

□ 階層型ディレクトリシステム

- ユーザがファイルをグループ分けしたくなるのは極めて一般的
- ユーザは任意数のサブディレクトリを生成可能
- 現在のほとんど全てのファイルシステムがこの方法で構成される



ディレクトリ(3)

□ パス名(Path Names)

- ファイルシステムをディレクトリ木で構成する場合、ファイルを指定する何らかの方法が必要となる
- 一般的に使用される2つの方法
 - 絶対パス名(**Absolute path name**)
 - ルートディレクトリからファイルまでのパスで構成される
 - 例) /usr/ast/mailbox (UNIX), ¥usr¥ast¥mailbox (Windows)
 - 相対パス名(**Relative path name**)
 - ワーキング/カレント(working directory/current directory)ディレクトリという概念と共に使用される
 - プロセスは現在のワーキングディレクトリを一つ持つ。この方法では、ルートディレクトリではなく、ワーキングディレクトリから相対的に見たパス名が指定される
 - 例) 現在のワーキングディレクトリが /usr/ast であるとすると、単に mailbox と書くだけで、/usr/ast/mailbox を指定したことになる

ディレクトリ(4)

□ パス名(Path Names)(続き)

- 全てのディレクトリに特別なエントリがある

- “.” (dot) : ワーキング/カレントディレクトリ
- “..” (dotdot): その親ディレクトリ

- 例) /usr/ast がカレント

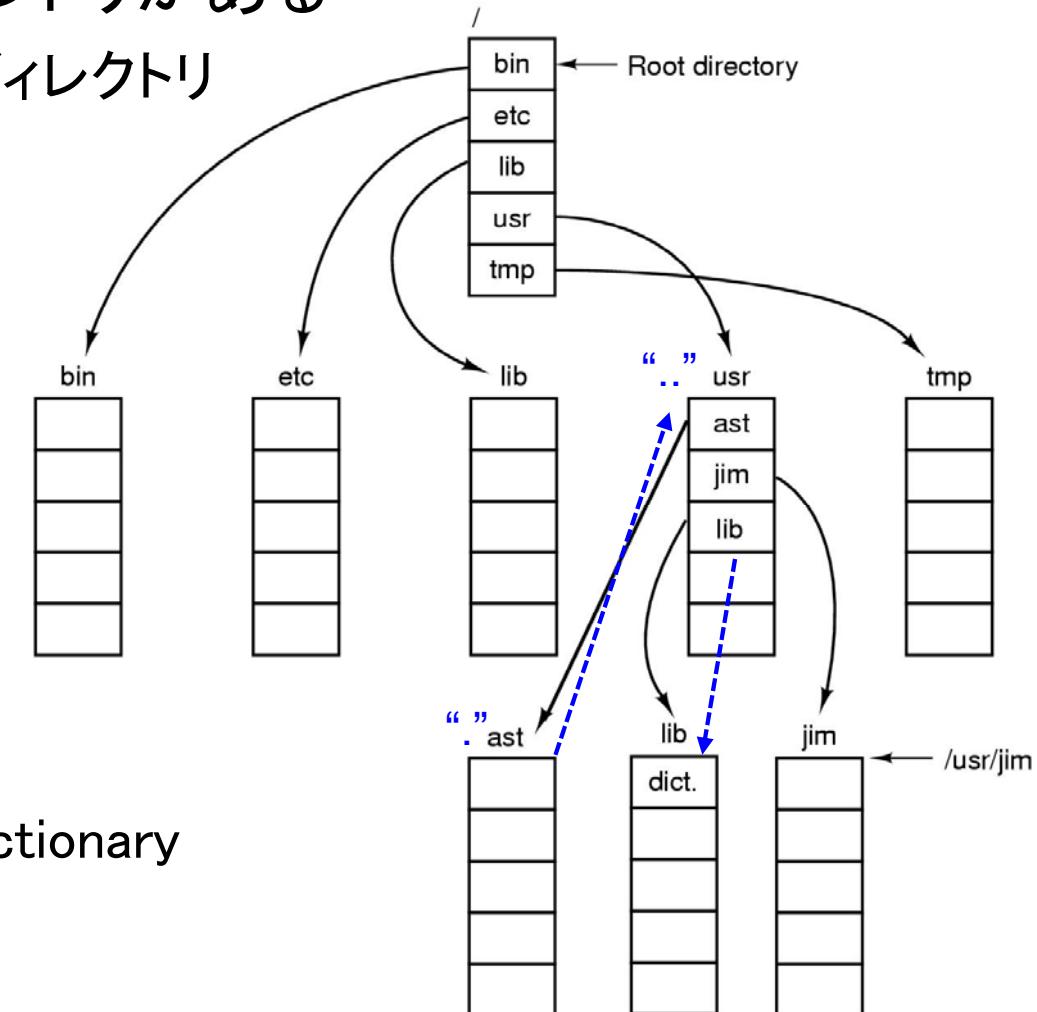
```
cp ..../lib/dictionary .
```

||

```
cp /usr/lib/dictionary dictionary
```

||

```
cp /usr/lib/dictionary /usr/ast/dictionary
```



ディレクトリ(5)

□ ディレクトリ操作

■ UNIXにおける例

□ create

- ディレクトリを生成する。".." と "...." 以外は空となる

□ delete

- ディレクトリを削除する。空ディレクトリ(".." と "...." のみ存在)のみ削除可能

□ opendir

- ディレクトリをオープンすることにより、ディレクトリ内のエントリが読み出せるようになる。例えば、ディレクトリ内の全ファイル名を列挙するプログラムは、ディレクトリをオープンし、含まれる全ファイルの名前を読み出す

□ closedir

- ディレクトリをクローズし、内部テーブルのスペースを開放する

□ readdir

- この呼出しは、オープンしたディレクトリ内の(次の)エントリを(inode番号やファイル名を含む標準フォーマットで)返す

ディレクトリ(6)

□ ディレクトリ操作(続き)

□ rename

- ディレクトリは、ファイルと同じように、名前を変更することができる

□ link

- この呼出しでは、存在するファイルと、別のパス名を指定し、指定したパス名からそのファイルへのリンクを生成する
- この方法で、同一ファイルが複数のディレクトリから見えるようになる
- この種のリンクは、ファイルの i-node 内のカウンタをインクリメントするもので、**ハードリンク(hard link)** と呼ばれる

□ unlink

- リンク(=ディレクトリエントリ)を削除する。もし unlinkされるファイルが1つのディレクトリのみに存在するならば、ファイルシステムから削除する。複数のディレクトリに存在する場合は、指定されたパス名のみが削除され、他は残る。UNIXでは、(先ほど挙げた)ファイルの削除のためのシステムコールは、実は unlink である

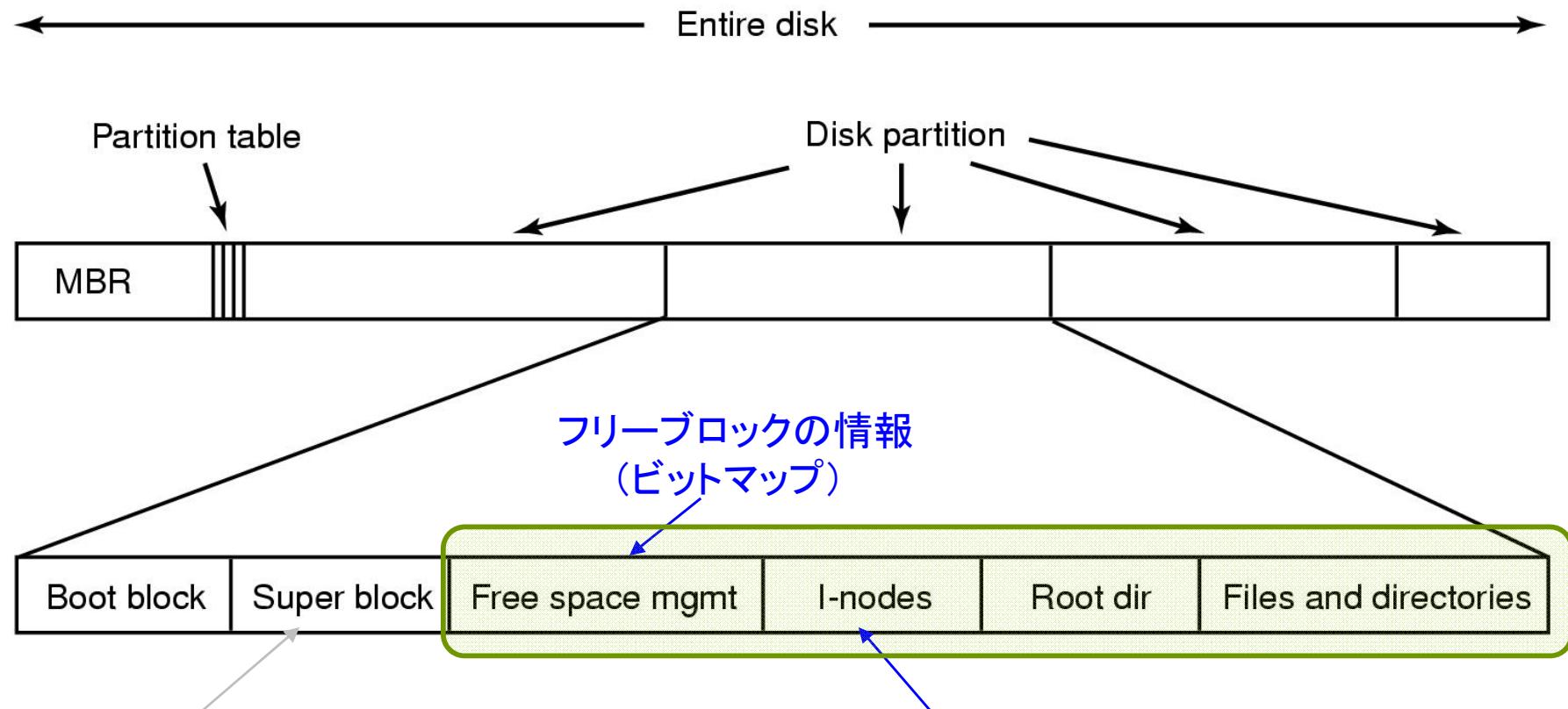
ファイルシステムの実装・例(1)

□ ファイルシステムの配置(File System Layout)

- ファイルシステムはディスク上に格納される
- 大抵のディスクは、1つ以上のパーティションに分割され、各パーティションは独立したファイルシステムを持つことができる
- ディスクのセクタ 0 は MBR (Master Boot Record) と呼ばれ、計算機をブートするのに使用される
 - MBR の終端にパーティションテーブルがあり、ここで各パーティションの開始アドレスと終了アドレスが与えられる
 - テーブル内のパーティションの1つがアクティブ(active)と印付けされている
 - 計算機がブートするとき、BIOS は MBRを読み出して実行する
 - MBRプログラムで最初に行なうことは、アクティブパーティションを特定し、その最初のブロック(ブートブロック/boot blockと呼ばれる)を読み込み、それを実行することである
 - ブートブロック内のプログラムはそのパーティションに含まれるOSをロードする

ファイルシステムの実装・例(2)

□ ファイルシステムの配置(File System Layout)(続き)



全ての重要なパラメータ情報を含む
ファイルシステムの型を示すマジック番号
ファイルシステム内のブロック数
その他の管理情報

ファイル毎のデータ構造
の配列

ファイルシステムの実装・例(3)

□ ファイルの実装

- どのディスクブロックがどのファイルに割り当てられているかを管理する
- 連続割当て(Contiguous Allocation)
 - 一つのファイルを連続するディスクブロックに格納する
 - 利点

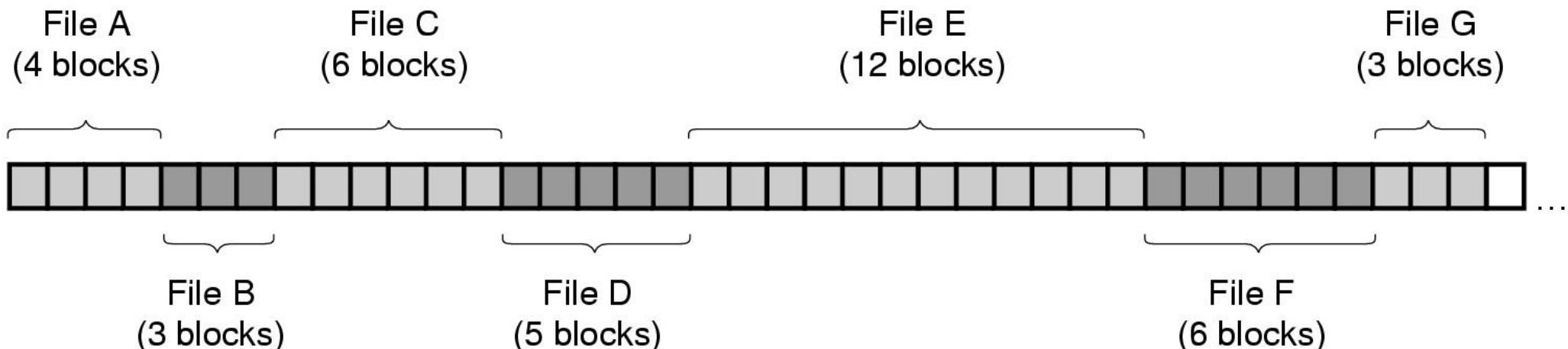
- 実装が簡単. 各ファイルにつき, 対応するディレクトリエントリに最初のブロックの番号(ディスクアドレス)とブロック数で管理可能
- 読み出し性能が高い. これは単一操作でファイルをディスクから読み出せるため. (最初のブロックへの一回の seekのみが必要)

□ 欠点

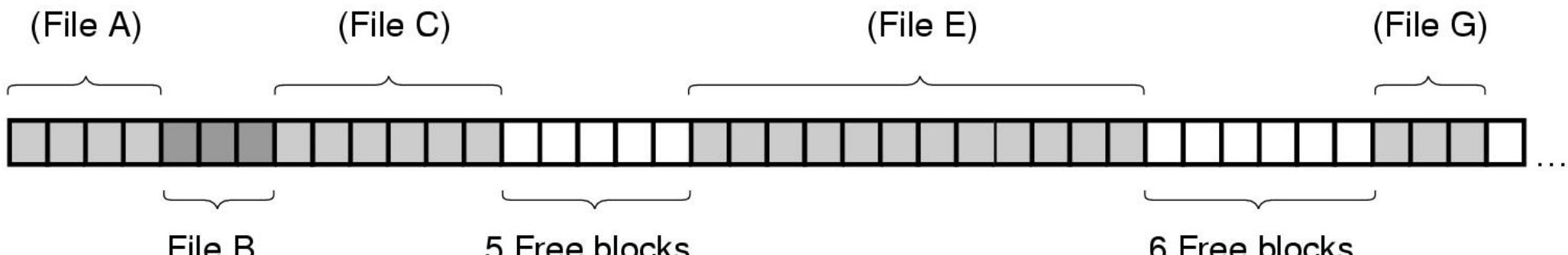
- ファイルの削除を行うと, 簡単に断片化が生じる. (次スライドの例)
- 削除後の空間(ホール/holes)を再利用するためには, ホールのリストを管理する必要がある. これは可能であるが, 新しいファイルを生成してホールに配置するとき, 最終的なサイズを知っている必要がある. (既存ファイルも伸長が困難)
- したがって, この割当て方法は CD-ROM/DVD 等の一回のみ書き込み可能な光学メディアでのみ使用される. このような用途では, 全てのファイルサイズは既知であり, 後で使用する際もサイズが変わることはないため.

ファイルシステムの実装・例(4)

■ 連続割当て(Contiguous Allocation)(続き)



(a) 7つのファイルのディスクへの連続割当て



(b) ファイルDとFが削除された後のディスク状態

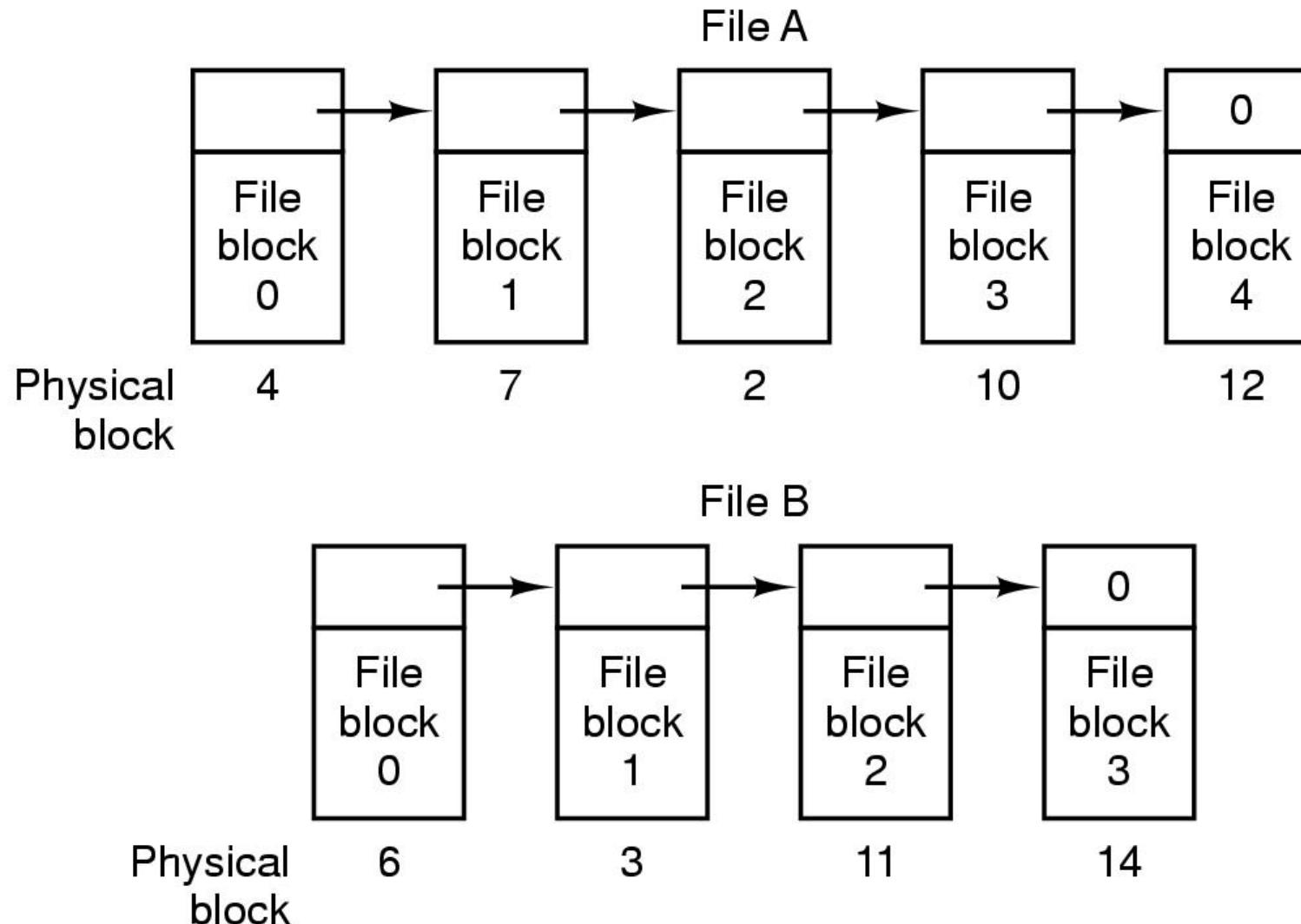
ファイルシステムの実装・例(5)

■ 連結リスト割当て(Linked List Allocation)

- 各ファイルを、ディスクブロックの連結リストとして管理する。各ブロックの最初のワードは次のブロックへのポインタとして使用される
- 利点
 - ファイルサイズ伸長の問題がない(散在するディスクブロックが利用できる)(最後のブロックの内部断片化を除いては、断片化は起こらない)
 - 各ファイルにつき、対応するディレクトリエントリに最初のブロック番号(ディスクアドレス)を格納しておけば十分である。残りはリストをたどれば見つかる
- 欠点
 - ランダムアクセスが極端に低速である。(n 番目のブロックを得るために、OSはその前に最初のブロックから読み始めて $n-1$ 個のブロックを読む必要がある)
 - 多くのプログラムでは、2のべき乗のサイズのデータ集合が読み書きされる。したがって、ポインタがブロック内の最初の数バイト占めているため、例えばディスクブロックと同サイズのデータを読むためには、2つのディスクブロックを読んで繋ぐ作業が必要となる。

ファイルシステムの実装・例(6)

■ 連結リスト割当て(Linked List Allocation)(続き)



ファイルシステムの実装・例(7)

■ メモリ上のテーブルを用いた連結リスト割当て

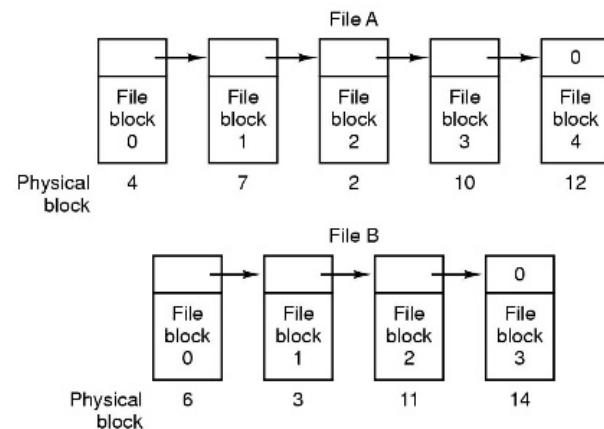
- 連結リスト割当ての2つの欠点は、ポインタを各ディスクブロック内ではなく、ディスクブロックから分離してメモリ内のテーブルに置くことにより解決できる
- このテーブルを **FAT (File Allocation Table)** と呼ぶ
- 利点
 - ブロック全体がデータに使用できる
 - リストはメモリ内に存在するため、ランダムアクセスが簡単・高速
 - ディレクトリエントリは開始ブロック番号を保持するのみでよい
 - FATがフリーブロックの管理情報(後述)を兼ねることが可能
- 欠点
 - テーブル全体が常にメモリ内になければならない。例えば、1TBのディスクおよび1KBのブロックサイズで、10億エントリが必要となり、テーブルに4GB程度のメモリを消費する
 - → パーティションのサイズを制限(FAT16は2GBまで, FAT32は2TBまで)
 - テーブルを仮想メモリ内に配置することは可能である。

ファイルシステムの実装・例(8)

■ メモリ上のテーブルを用いた連結リスト割当て(続き)

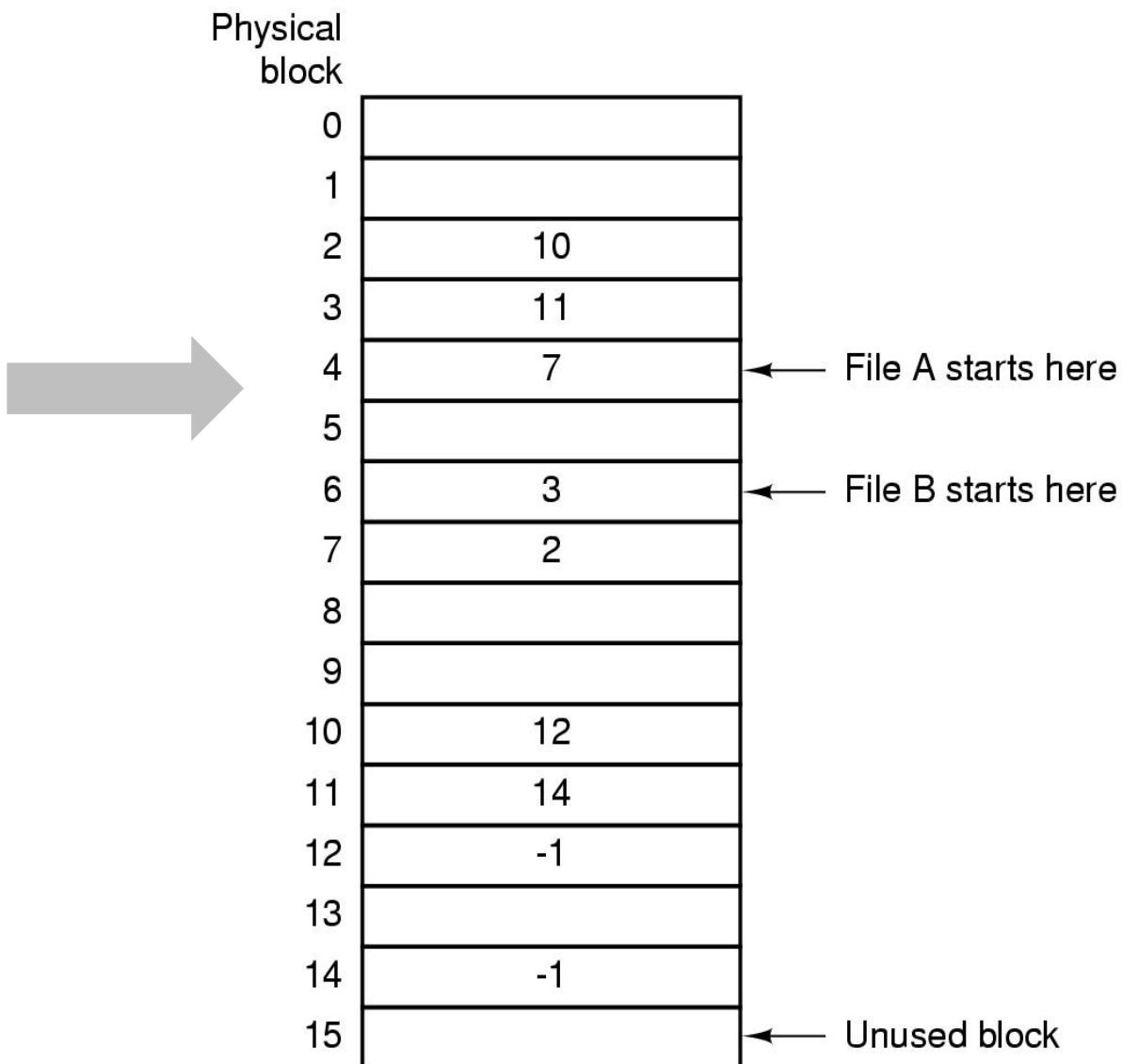
□ 例

- 前の図と同じリストの表現



File A: 4→7→2→10→12

File B: 6→3→11→14



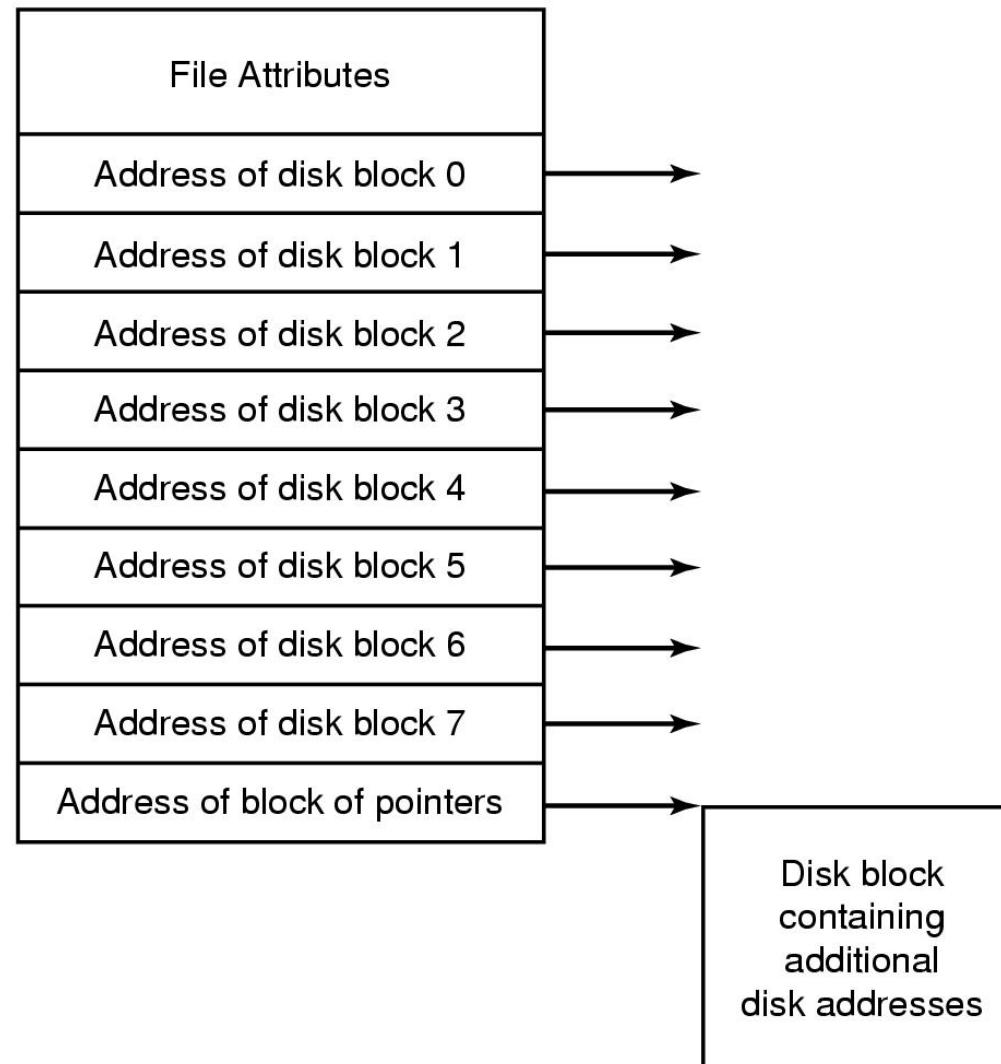
ファイルシステムの実装・例(9)

■ I-nodes

- 各ファイルに **i-node (index-node)** と呼ばれるデータ構造を関連付ける
 - ファイルの属性とブロック番号(ディスクアドレス)のリスト
 - ディレクトリエントリは対応するファイルのi-node番号を保持する
- 利点
 - i-node は、対応するファイルがオープンされているときのみ、メモリ内にあればよい
 - オープンされたファイルのみの i-node の配列が管理される。この配列は FATで使用するサイズよりもずっと小さい
- 問題が1つ
 - i-node は固定サイズであり、その中で固定数のディスクアドレスのスペースしかない
 - 1つの解決法として、最後のディスクアドレスの部分はデータブロックではなく、残りのディスクブロックのアドレスを保持するブロックのアドレスを保持させる。(次スライドの図)

ファイルシステムの実装・例(10)

■ I-nodes (続き)



ファイルシステムの実装・例(11)

□ ディレクトリの実装

- ファイルをオープンするとき, OSはユーザから与えられたパス名を使用してディレクトリエントリを特定する
- 各ディレクトリエントリには, 対応するファイルを構成するディスクブロック列を得るための情報がある
 - 連續割当てでは先頭のブロックアドレス, 連結リストでは先頭のブロックアドレス(FATの場合), または i-node番号
- したがって, ディレクトリシステムの主な機能は, ファイルのASCII名を, ファイル内データの特定に必要な情報(ブロック番号／ディスクアドレス)にマップすることである
- その他として, ファイル属性情報をどこに格納すべきか?
 - FATでは, ディレクトリエントリに直接格納する(次スライド左図)
 - i-node方式では, ディレクトリエントリが指す i-node に格納する(右図)

ファイルシステムの実装・例(12)

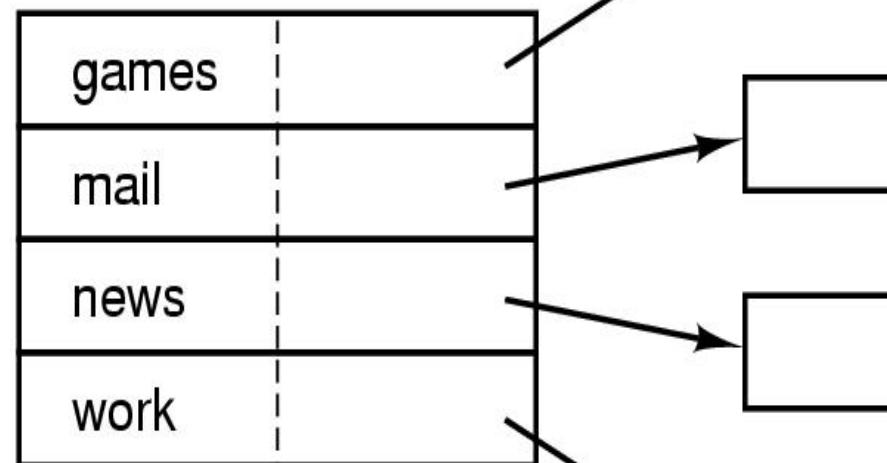
□ ディレクトリの実装(続き)

games	attributes
mail	attributes
news	attributes
work	attributes

(a)

FAT:
(In MS-DOS/Windows)

先頭ブロック番号も含む



(b)

Referring to i-nodes:
(In UNIX)

i-node

Data structure
containing the
attributes

ファイルシステムの実装・例(13)

□ i-node方式における、/usr/ast/mboxへのアクセス

ルートディレクトリのみ、
特別に“..”は自分自身と
なる

1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

Looking up
usr yields
i-node 6

I-node 6
is for /usr

attributes
Mode
size
times

132

Block 132
is /usr
directory

6	•
1	..
19	dick
30	erik
51	jim
26	ast
45	bal

I-node 6
says that
/usr is in
block 132

I-node 26
is for
/usr/ast

Mode
size
times

406

/usr/ast
is i-node
26

Block 406
is /usr/ast
directory

26	•
6	..
64	grants
92	books
60	mbox
81	minix
17	src

I-node 26
says that
/usr/ast is in
block 406

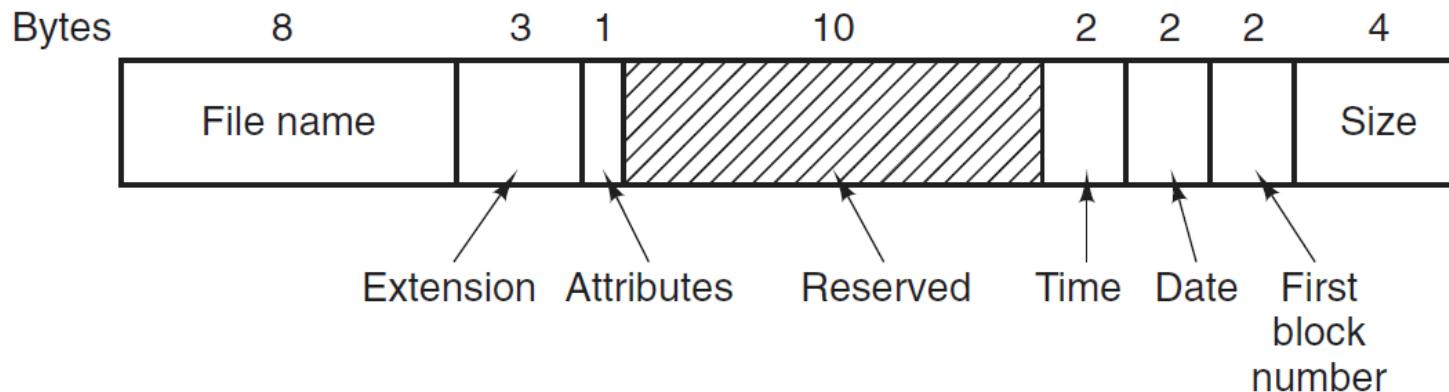
/usr/ast/mbox
is i-node
60

ファイルシステムの実装・例(14)

□ ディレクトリエントリ

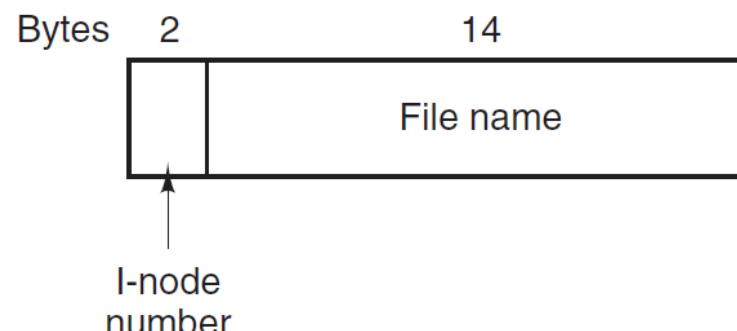
■ MS-DOS File System (FAT-12,16,32)

- 32-byte directory entry
- 名前は8バイト+3バイト(拡張子)まで



■ UNIX V7 File System (in UNIX for PDP-11)

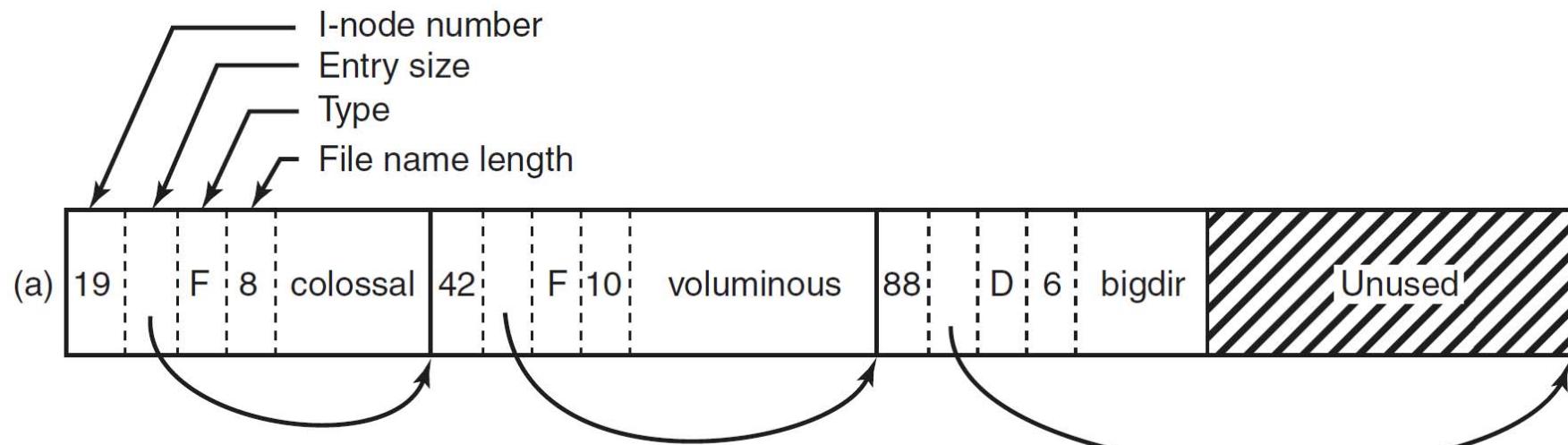
- 名前は14バイトまで



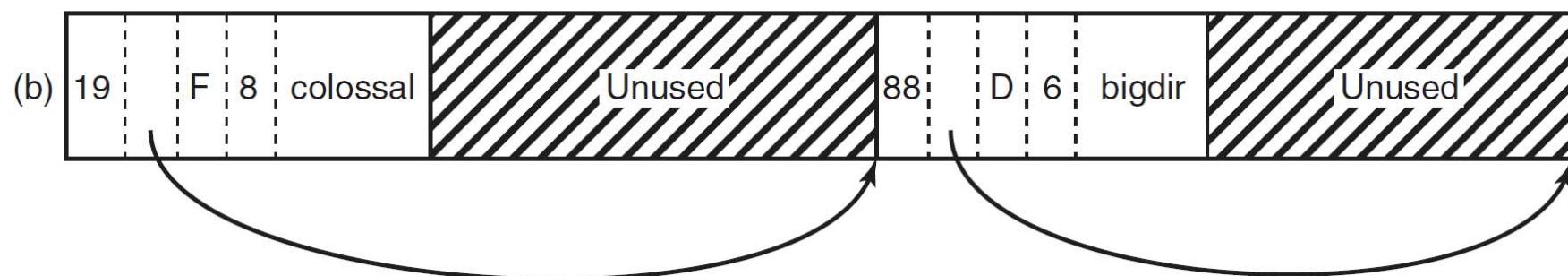
ファイルシステムの実装・例(15)

■ Ext2 File System in Linux

- 可変長サイズディレクトリエントリ(ファイル／ディレクトリ名が可変長)



ファイル“voluminous”的削除後



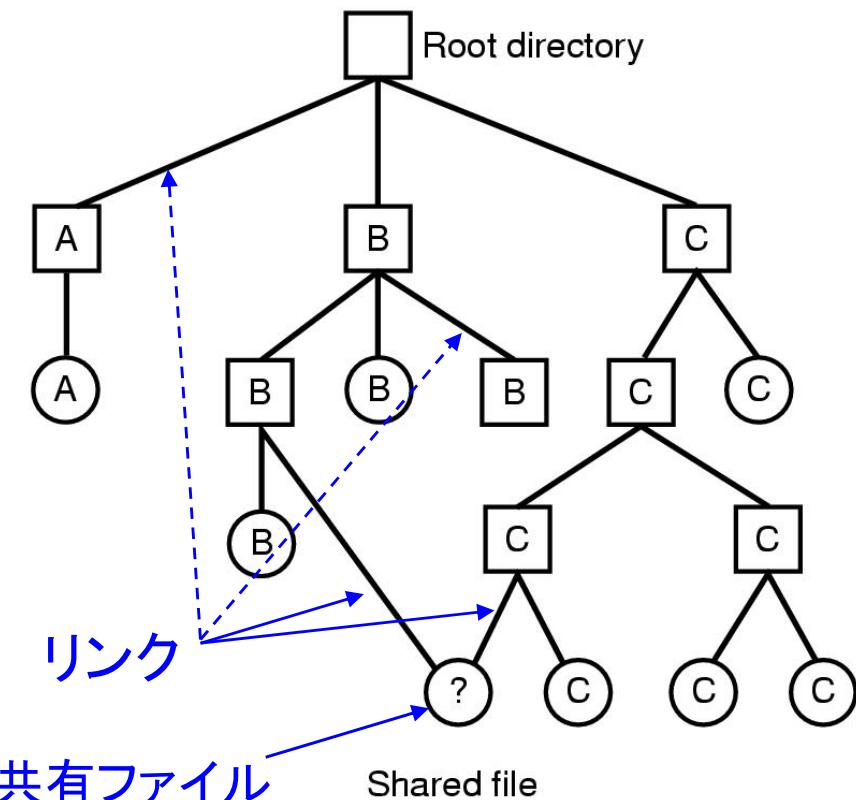
ファイルシステムの実装・例(16)

□ 共有ファイル(Shared Files)

- 複数のユーザがプロジェクトに関わっている場合、しばしばファイルを共有する必要がでてくる
- 異なるユーザのそれぞれのディレクトリに、共有(同一の)ファイルが同時に存在できたら便利である

■ リンク(Link)

- ディレクトリとファイルとの間のつながり
- 共有を可能とすることで、ファイルシステム自体が木構造ではなく、有向非循環グラフ(Directed Acyclic Graph, DAG)となる

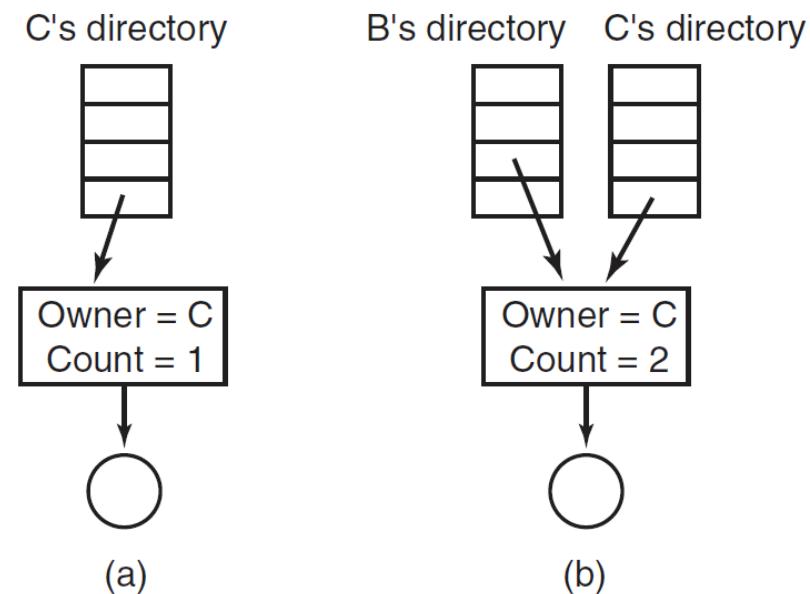


ファイルシステムの実装・例(17)

□ 共有ファイル(続き)

■ ハードリンク(Hard link)

- 対応する i-node を指す



■ シンボリックリンク(Symbolic link)

- リンクを行うとき、システムは LINK タイプの新たなファイルを生成し、ディレクトリに置く
 - その生成したファイルには、リンクするファイルへのパス名だけが書かれる
 - シンボリックリンクを削除しても、ファイルの実体には全く影響を与えない
 - 問題はオーバヘッド
 - 真のファイルを読み出すためには、その前にシンボリックリンクファイルを読み出す必要がある

ファイルシステムの実装・例(18)

□ ディスク空間管理

■ ブロックサイズ

- 割当て単位として、セクタやトラック、シリンドがわかりやすい候補
- 大きな割当て単位(シリンド)になると、どんなファイルでも(たとえ1バイトのファイルでも)シリンド全体を占めることになる(空間効率が悪い)
- 小さな単位になると、各ファイルはたくさんのブロックから構成されることになる。各ブロックを読み出すために、毎回、シークと回転遅延という遅い要因が伴う
 - 例) トラックあたり 1MB ($=2^{20}$) バイト、回転速度が 8.33 msec、平均シーク時間が 5 msec のディスクがある。k バイトのブロックを読み出すための時間は
 - $5 + 8.33/2 + (k/1,048,576) * 8.33$
 - 小さなブロックに対しては、シーク時間と回転遅延が支配的要因となる
- OSは、トラック上の連続するセクタからなる"クラスタ"をブロックとして扱う

ファイルシステムの実装・例(19)

□ ディスク空間管理(続き)

■ データ転送率＆空間効率(全てのファイルが4KBの場合の実験)

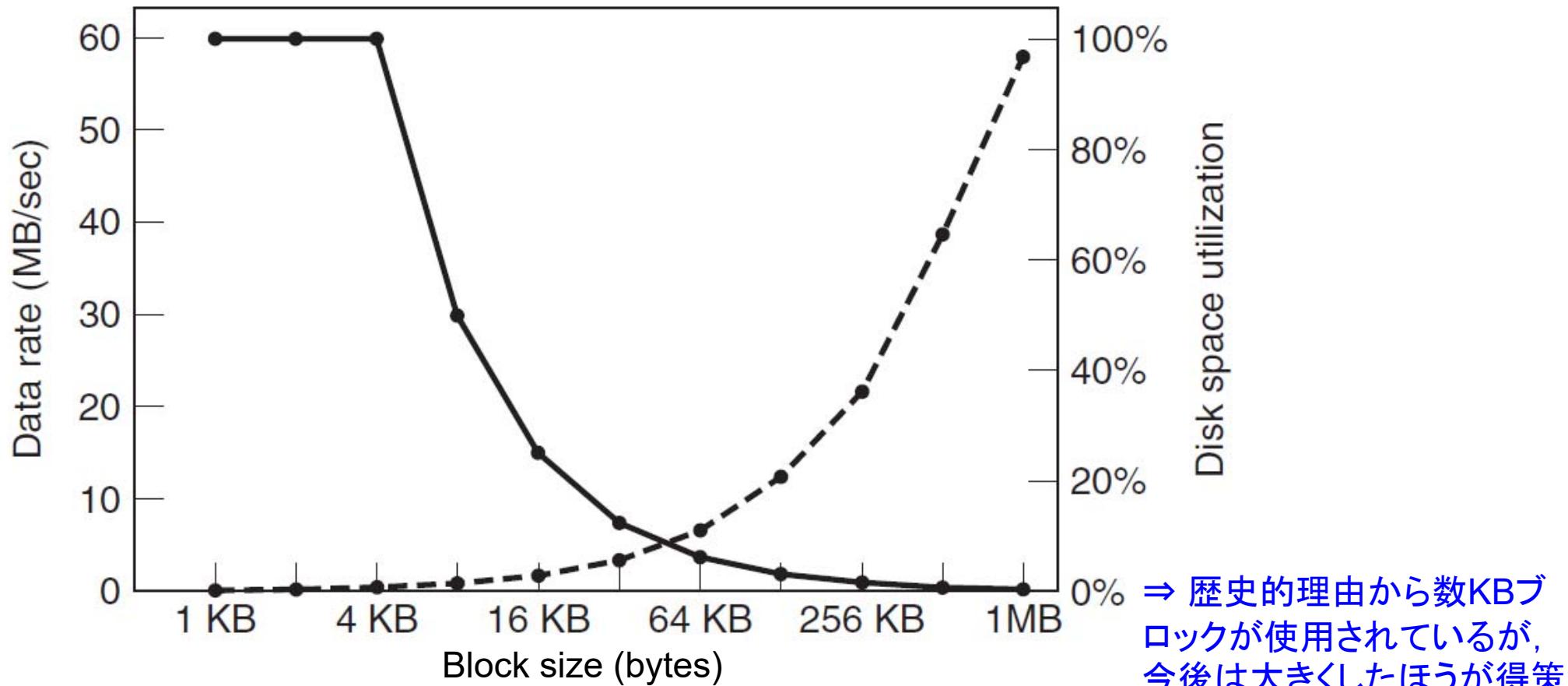


Figure 4-21. The dashed curve (left-hand scale) gives the data rate of a disk. The solid curve (right-hand scale) gives the disk-space efficiency. All files are 4 KB.

ファイルシステムの実装・例(20)

□ ディスク空間管理(続き)

■ フリーブロックの管理

□ 連結リストによるフリーリスト(Free list on a linked list)(次スライド左図)

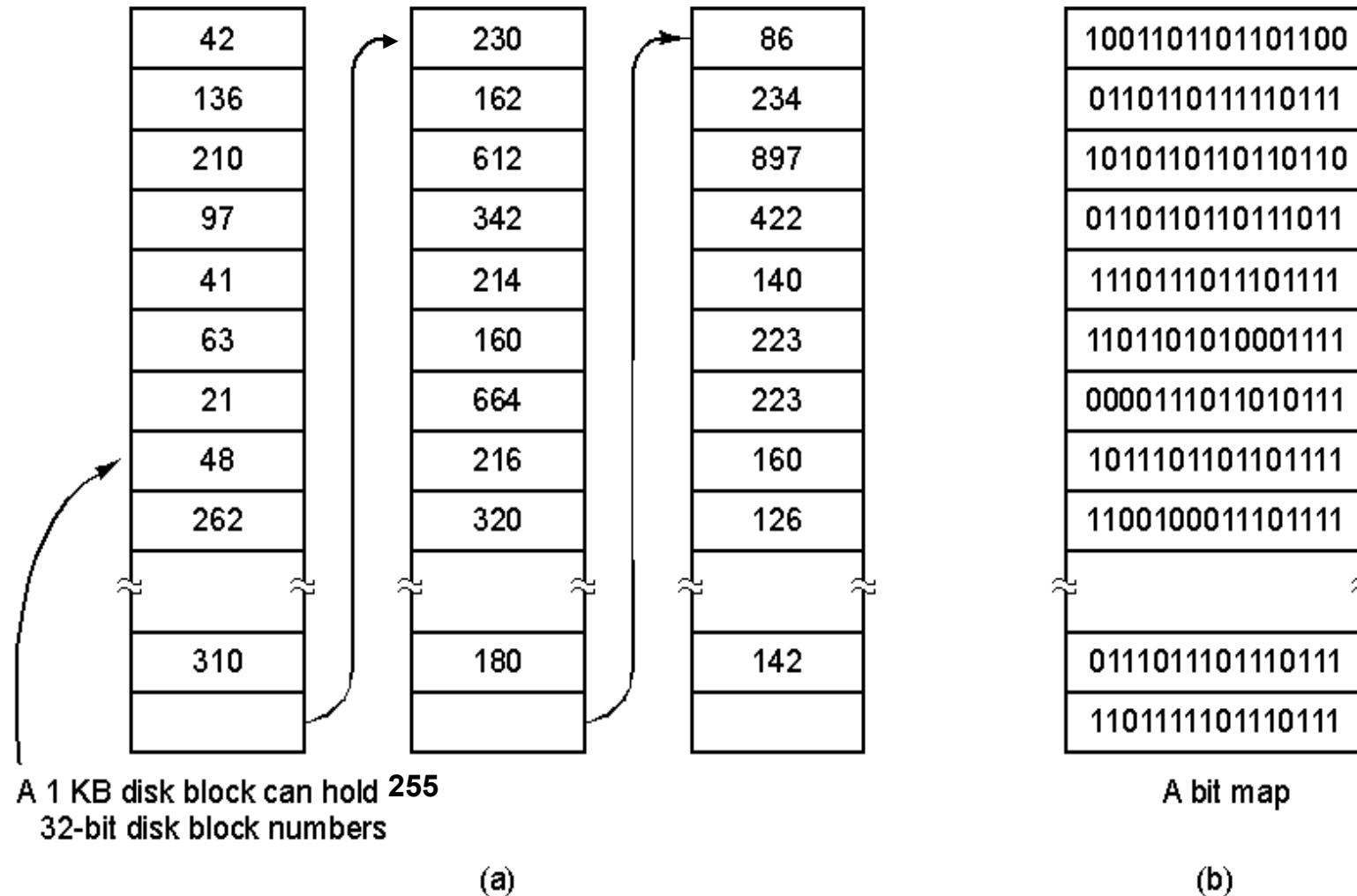
- ディスクブロックの連結リストを使用する。各ブロックは入りうるだけのフリーディスクブロック番号を保持している
- フリーリストのサイズは、フリーブロックの数に応じて変化する
 - フリーリストを保持するためにフリーブロックが使用されるため, サイズは問題にならない。また、一般的には、リスト内の1つのブロックがメモリ内にキャッシュされる

□ ビットマップ(Bitmap)(次スライド右図)

- n ブロックからなるディスクは n ビットのビットマップが必要である
- フリーブロックはビットマップ内で 1 で表現され、割当て済みのブロックは 0 で表現される(または、その逆)
- ビットマップのサイズは固定
 - ビットマップは複数のディスクブロックを利用して格納され、一般的には、そのうちの1つのブロックがメモリ内にキャッシュされる

ファイルシステムの実装・例(21)

□ ディスク空間管理(続き)



注)FAT方式の場合はテーブル内でフリーブロックを識別できるため、フリーリストやビットマップは必要ない。

ファイルシステムの実装・例(22)

□ ファイルシステムの性能(File System Performance)

■ キヤッシング(Caching)

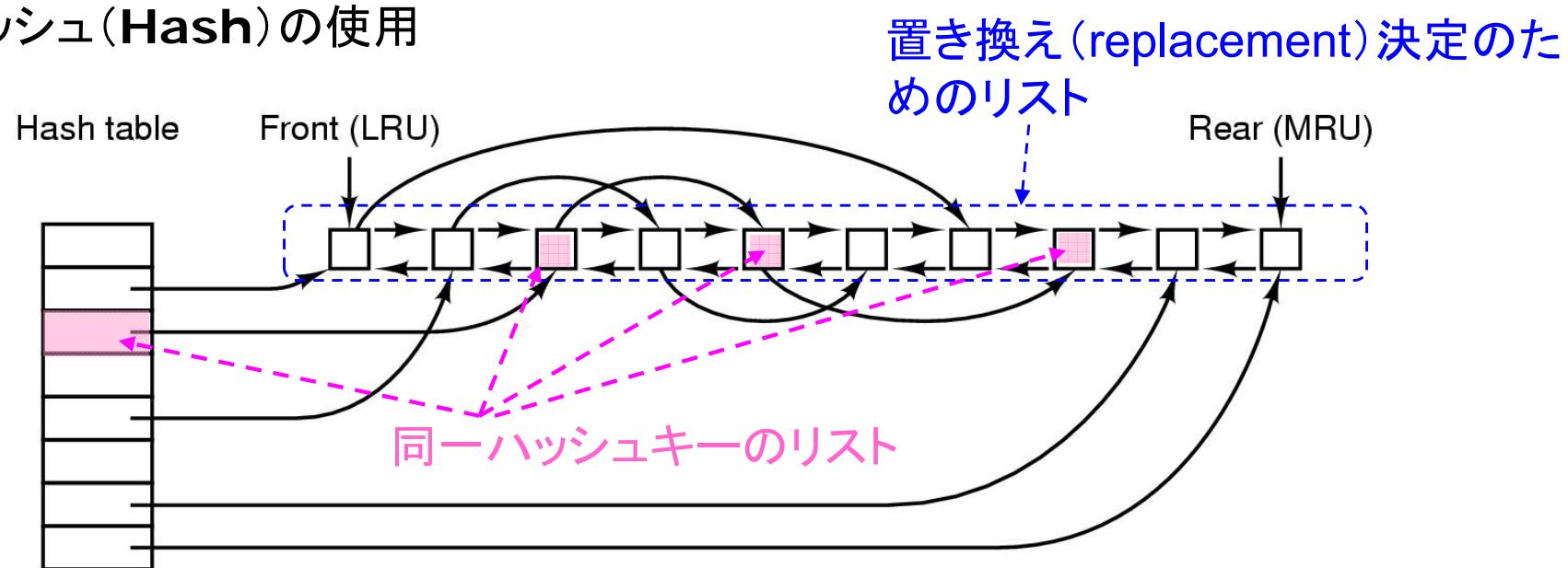
□ ディスクアクセスを減らすための最も一般的な方法は、バッファキャッシュ(buffer cache) (or ブロックキャッシュ(block cache))である

- 性能向上のためにメモリに置かれるブロックの集合
- 全ての読み出し or 書込み要求で、要求されたブロックがキャッシュ内に存在するかどうかチェックする
 - もし存在すれば、その要求はディスクアクセス無しで満たされる
 - もし存在しなければ、ディスクアクセスを行い、キャッシュ内に入れる
 - 同一ブロックへの後続要求はキャッシュ内のブロックに対して行うことが可能となる

ファイルシステムの実装・例(23)

■ キャッシング(続き)

- キャッシュ内にはたくさんのブロックが存在するため、要求されたブロックが存在するかどうかをすばやくチェックできる方法が必要である
 - ハッシュ(Hash)の使用



- クラッシュに備えて、更新ブロックは遅くならないうちに書き出されるべき
 - 例) UNIXでは、“sync”システムコールが明示的な書き出しを実行する。“update”デーモンが(例えば)30秒毎にsyncを実行
 - 例) 最近のWindowsでも，“FlushFileBuffers”システムコールをサポート
 - 以前は、全てのブロックキャッシュ書き込みで、ディスク書き込みを伴っていた(ライトスルーキャッシュ)

ファイルシステムの実装・例(24)

□ ファイルシステムの性能(続き)

■ ブロック先読み(Block Read Ahead)

- ブロックを、要求される前にキャッシュに入れておく
- 多くのファイルは逐次的に読まれる傾向がある
 - ファイルシステムに対して、ファイル内の k 番ブロックへの読み出し要求があった場合、それを実行し、その後 block $k+1$ 番ブロックが既にキャッシュ内に存在するかどうかチェックする。もしなければ、 $k+1$ 番ブロックへの読み出し要求を(事前に)スケジュールする
- 逐次アクセスとならないファイルに対して、この方法は使用すべきではない
 - ディスクのバンド幅とメモリ内のキャッシュが浪費される
- ファイルに対して逐次アクセスかランダムアクセスのどちらがなされるかの決定は重要であり、動的な制御が必要とされる
 - アプリケーションによるファイルアクセスのパターンを監視しながら決定

ファイルシステムの実装・例(25)

□ ファイルシステムの性能(続き)

■ ディスクアーム動作の削減(Reducing Disk Arm Motion)

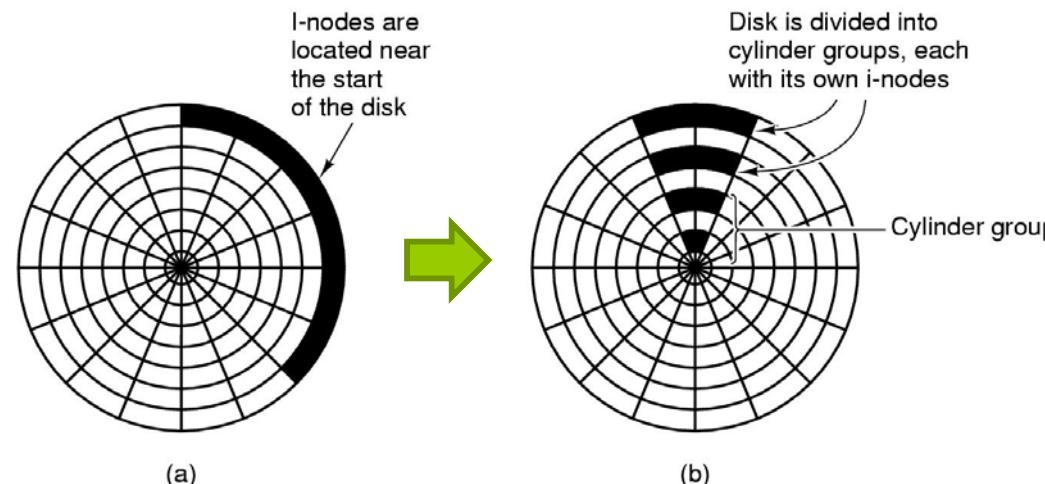
- 順番にアクセスされそうなブロックをお互いに近くに配置する

- シーク時間を回避/削減できる

- 例) ファイルをオープンして内容を読み出すときに, i-node とデータブロックに
対して, 少なくとも2回, ディスクがアクセスされる

- シーク時間を削減するために, i-nodeのブロックとデータのブロックをお互いに近く
に配置すべきである(右下図)

i-nodeとデータブロック
との距離は平均すると
半径の1/2



i-nodeとデータブロック
を同じグループに配置
すると, シーク距離が
短くなる

- ### ■ solid-state disks (SSD)の出現で, シーク/回転時間に対する対 処は不要になりつつある

ファイルシステムの実装・例(26)

□ NFS (Network File System)

- Originally by Sun Microsystems
- ファイルシステムを異なる計算機間で共有する機能を提供

- Client-Server間プロトコルを定義

- Clientからの依頼(lookup message)に対して, Serverは exportされるディレクトリやファイルの “file handle”を提供

- “file handle” はファイルシステムタイプ, ディスク情報, ディレクトリのi-node番号, セキュリティ情報を含む

- Clientは, この“file handle”を指定してread/write要求を出す

