Memory Management

I233 Operating Systems

Memory Management (1)

- 3.1 NO MEMORY ABSTRACTION
- 3.2 A MEMORY ABSTRACTION: ADDRESS SPACES
  - 3.2.1 The Notion of an Address Space
  - 3.2.2 Swapping
  - 3.2.3 Managing Free Memory
- 3.3 VIRTUAL MEMORY
  - 3.3.1 Paging
  - 3.3.2 Page Tables
  - 3.3.3 Speeding Up Paging
  - 3.3.4 Page Tables for Large Memories
- 3.4 PAGE REPLACEMENT ALGORITHMS
  - 3.4.1 The Optimal Page Replacement Algorithm
  - 3.4.2 The Not Recently Used Page Replacement Algorithm
  - 3.4.3 The First-In, First-Out (FIFO) Page Replacement Algorithm
  - 3.4.4 The Second-Chance Page Replacement Algorithm
  - 3.4.5 The Clock Page Replacement Algorithm
  - 3.4.6 The Least Recently Used (LRU) Page Replacement Algorithm
  - 3.4.7 Simulating LRU in Software
  - 3.4.8 The Working Set Page Replacement Algorithm
  - 3.4.9 The WSClock Page Replacement Algorithm
  - 3.4.10 Summary of Page Replacement Algorithms

Memory Management (2)

- 3.5 DESIGN ISSUES FOR PAGING SYSTEMS
  - 3.5.1 Local versus Global Allocation Policies
  - 3.5.2 Load Control
  - 3.5.3 Page Size
  - 3.5.4 Separate Instruction and Data Spaces
  - 3.5.5 Shared Pages
  - 3.5.6 Shared Libraries
  - 3.5.7 Mapped Files
  - 3.5.8 Cleaning Policy
  - 3.5.9 Virtual Memory Interface
- 3.6 IMPLEMENTATION ISSUES
  - 3.6.1 Operating System Involvement with Paging
  - 3.6.2 Page Fault Handling
  - 3.6.3 Instruction Backup
  - 3.6.4 Locking Pages in Memory
  - 3.6.5 Backing Store
  - 3.6.6 Separation of Policy and Mechanism
- 3.7 SEGMENTATION
  - 3.7.1 Implementation of Pure Segmentation
  - 3.7.2 Segmentation with Paging: MULTICS
  - 3.7.3 Segmentation with Paging: The Intel x86

Basics on Memory Management

- Two categories of memory management by the OS.
  - Methods used in the swapping (migrating a running process between main memory and disk.) environment.
  - Methods used in the environment without swapping.
- Needs for memory management:
  - With memory becoming cheaper and cheaper, and the amount of memory installed in computers increasing, is there really a need to manage memory?
  - Memory is a limited resource (not disposable).
  - Growth in process size (software bloat).
  - The rise of small devices that run more sophisticated programs.

Comparison of Microsoft Windows minimum hardware requirements (for x86 versions)

Windows version	Processor	Memory	Hard disk
Windows 95[9]	25 MHz	4 MB	~50 MB
Windows 98[10]	66 MHz	16 MB	~200 MB
Windows 2000[11]	133 MHz	32 MB	650 MB
Windows XP[12] (2001)	233 MHz	64 MB	1.5 GB
Windows Vista[13] (2007)	800 MHz	512 MB	15 GB
Windows 7[14] (2009)	1 GHz	1 GB	16 GB
Windows 8[15] (2012)	1 GHz	1 GB	16 GB
Windows 10[16] (2015)	1 GHz	1 GB	16 GB
Windows 11[17] (2021)	1 GHz	4 GB	64 GB

Source: https://en.wikipedia.org/wiki/Software_bloat

3.1 No memory abstraction

- The simplest memory abstraction is to have no abstraction at all. (no swapping, no paging)
- Place and execute only one program at a time on memory.

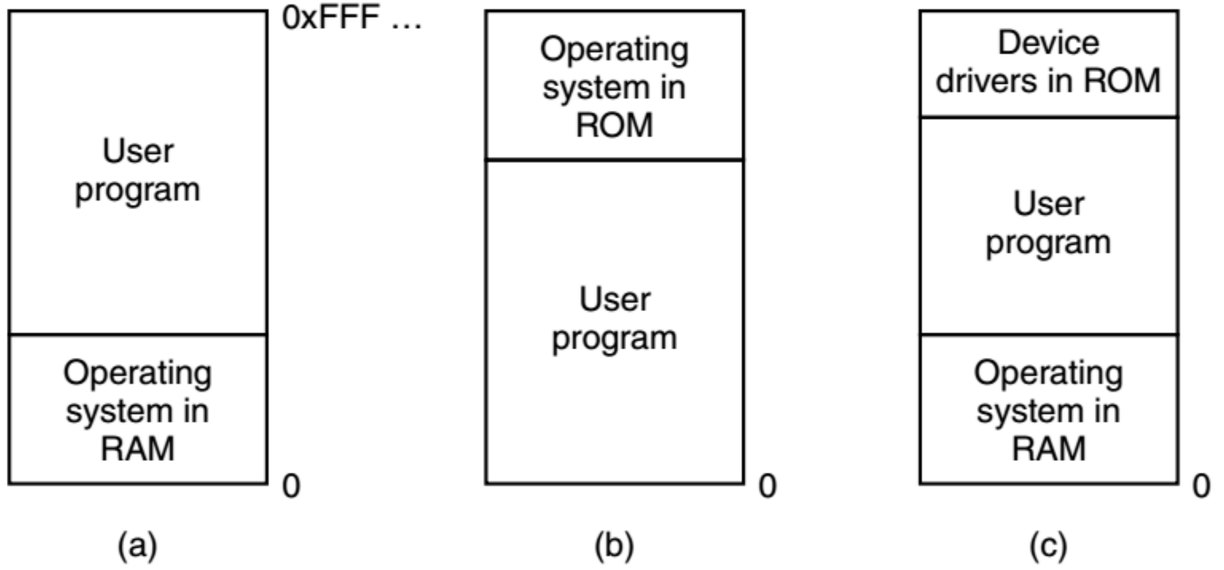


Figure 3-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.183.

Running Multiple Programs Without a Memory Abstraction

- Swapping
  - Saving the entire contents of memory to a disk file, then bring in and run the next program. As long as there is only one program at a time in memory, there are no conflicts.
- Problem on the two programs are placed consecutively in memory.

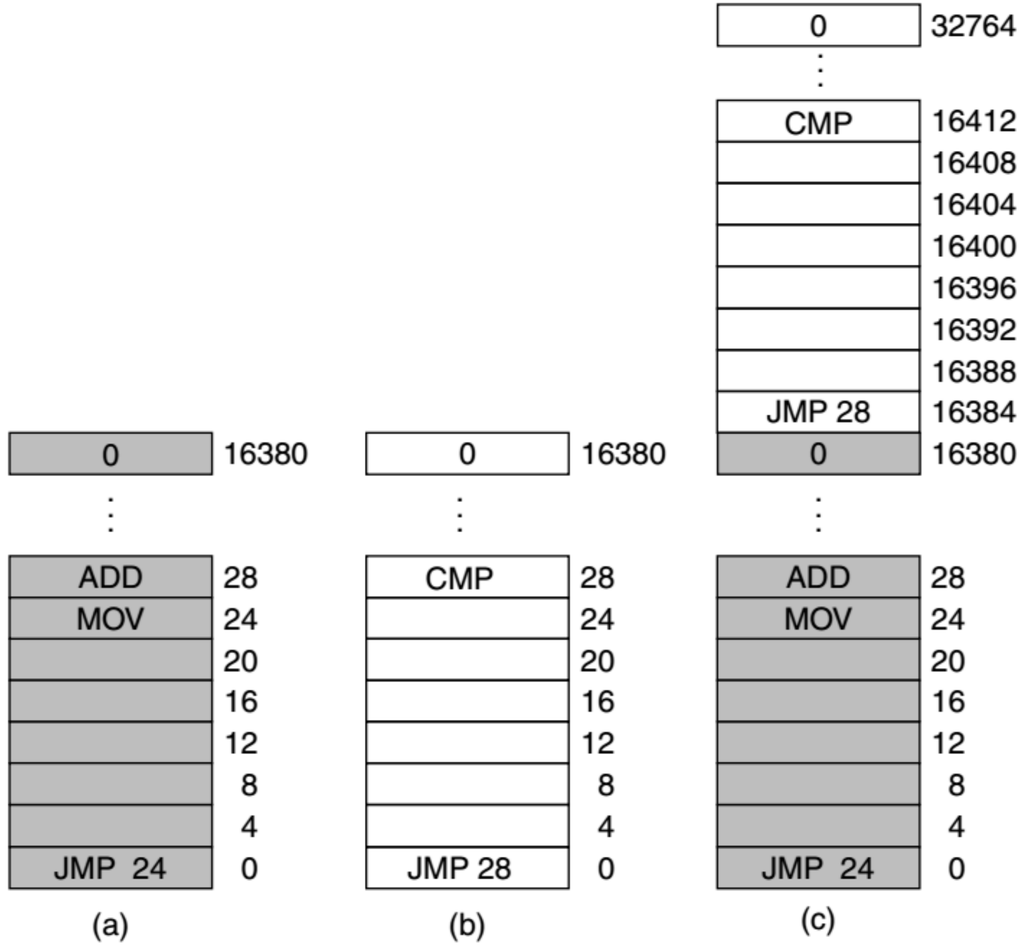


Figure 3-2. Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.184.

3.2.1 The Notion of an Address Space

- Two problems have to be solved to allow multiple applications to be in memory at the same time without interfering with each other: “protection” and “relocation”.
- Solution by “relocation”:
  - When loading a program into memory, rewrite the address notation using absolute addressing to make it fit the actual situation.
  - Solution by relocation hardware: Prepare a base register to hold the starting address of the program's memory area, and execute all memory accesses as relative addresses from the base register.
- Add "protection" function:
  - In addition to the base register, a limit register is provided to indicate the size of the memory area available to the process; memory accesses that exceed the limit register are recognized as abnormal behavior of the process.

3.2.2 Swapping

- A method of concurrently executing processes that do not entirely fit in memory.
  - Swapping
    - Migrate the entire memory of a process between memory and disk.
  - Virtual Memory
    - Placing only the part of the process's memory that is needed on memory.
    - The parts that are no longer needed will be pushed out to disk.
- Swapping procedure:
  - Determine which process to load into memory.
  - Load the memory of the process from disk to memory.
  - Execute the process.
  - (Migrate the memory of the process from memory to disk.)

Example on Swapping

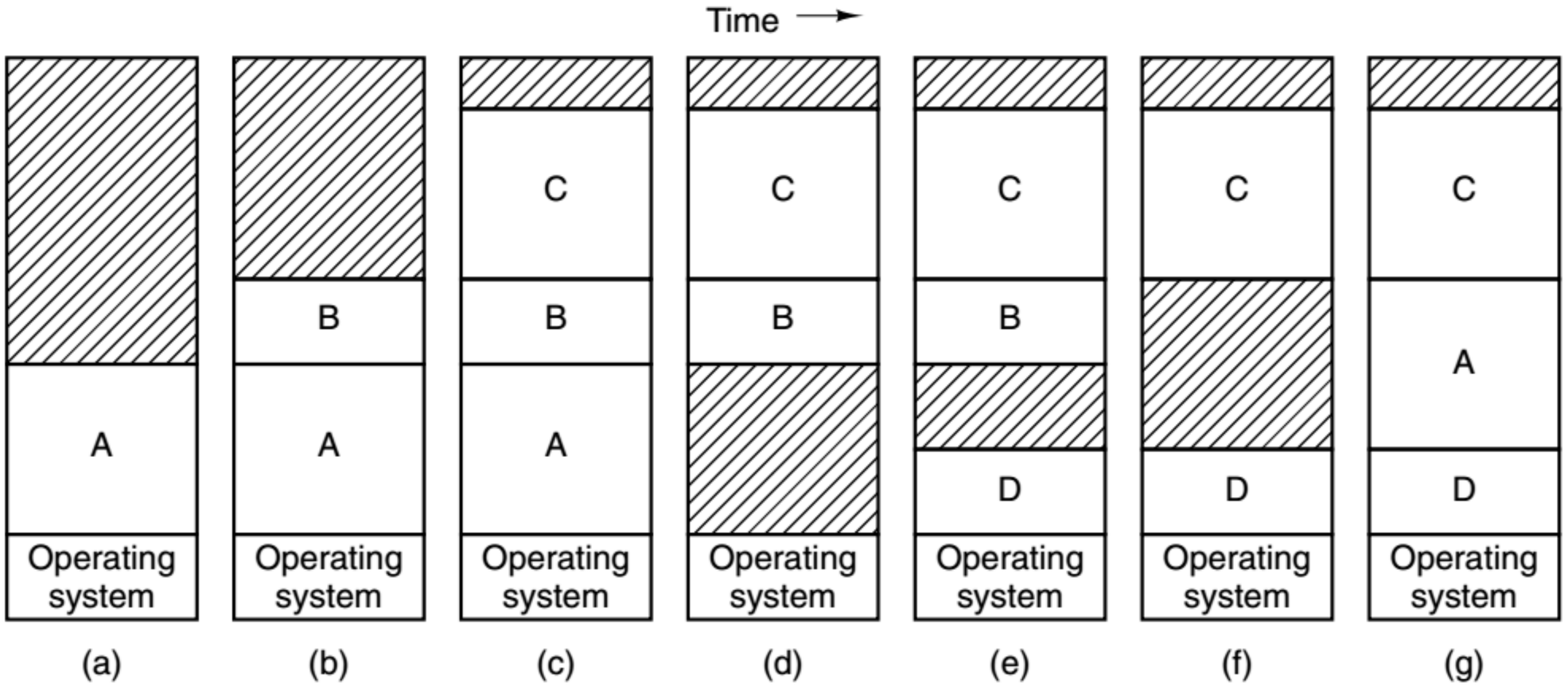


Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.189.

Consideration on Swapping

- When swapping creates multiple holes in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible.
  → “memory compaction” (but it require a lot of CPU time)
- How much memory should be allocated to each process?
- How to handle the case where a process uses multiple segments?
- How to manage processes and holes?

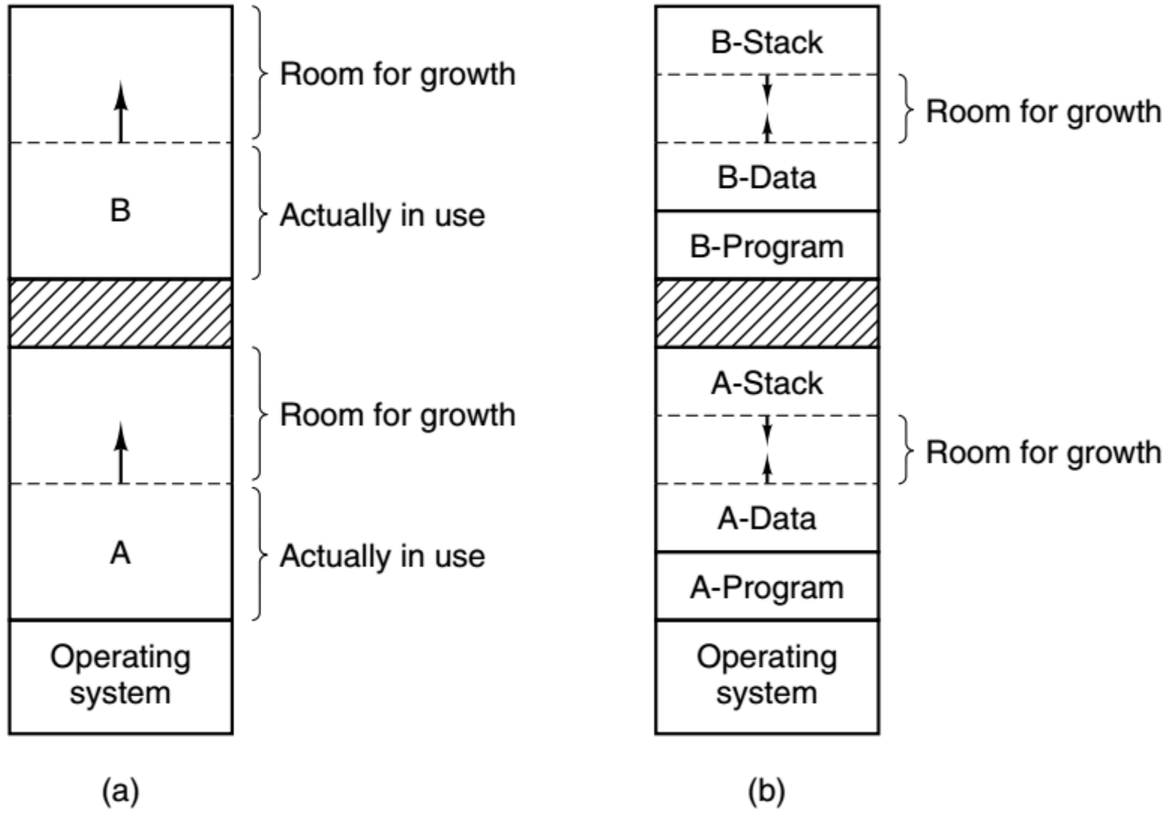


Figure 3-5. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.190.

Memory Management: bitmap v.s. linked-list

- Overhead due to administrative area v.s. Operating complexity

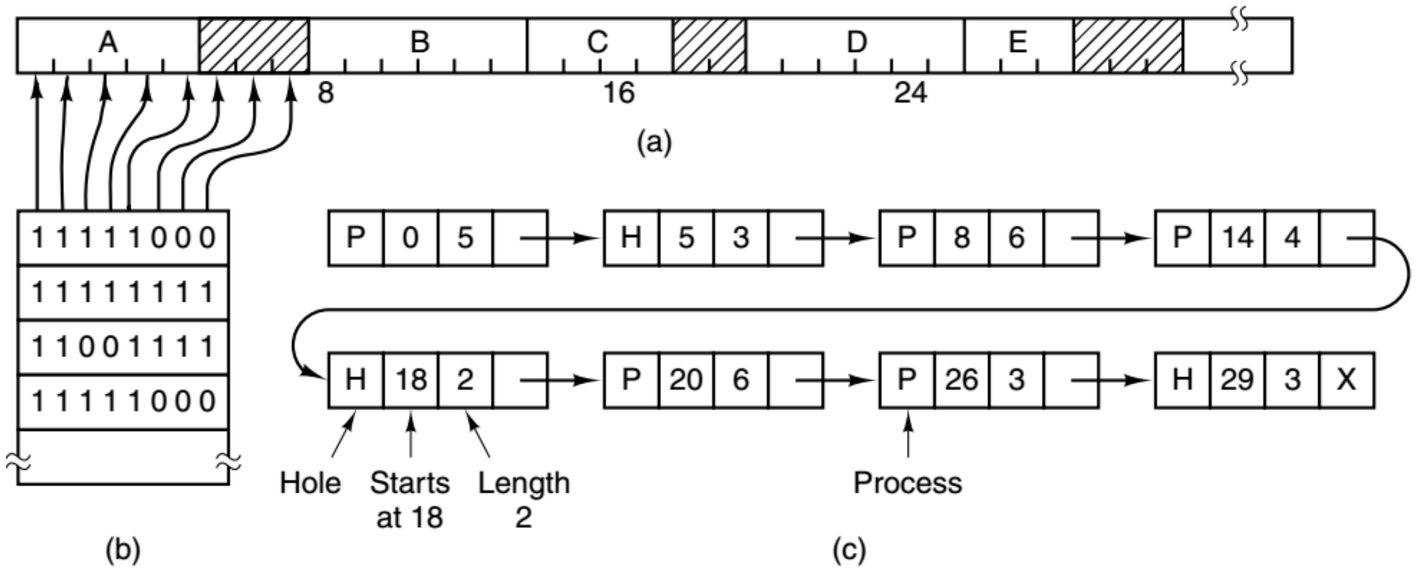


Figure 3-6. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

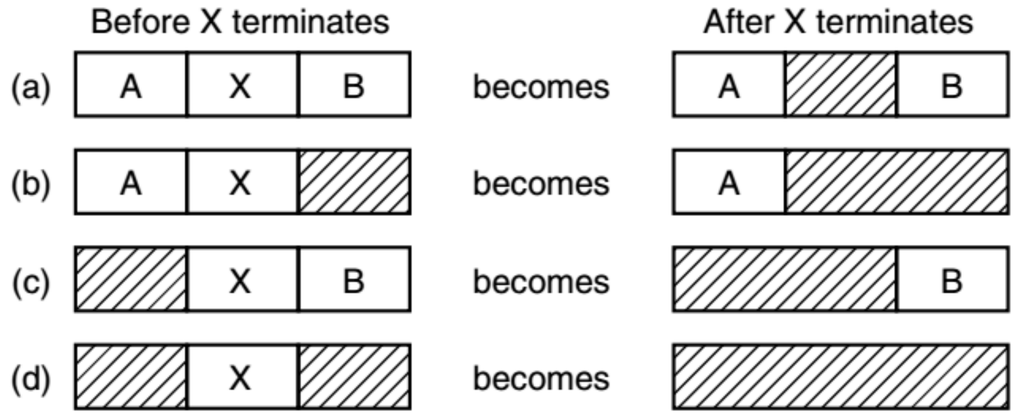


Figure 3-7. Four neighbor combinations for the terminating process, *X*.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.191, 192.

Algorithms to allocate memory for a process (1)

- The problem of which sized hole to choose when migrating a process onto memory.
- Performance index:
  - Overhead by management area
  - Time required for allocation

- First fit
  - The memory manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory.
  - A fast algorithm because it searches as little as possible.
- Next fit
  - A minor variation of first fit.
  - Searching the list from the place where it left off last time.
  - Simulations by Bays (1977) show that next fit gives slightly worse performance than first fit.

Algorithms to allocate memory for a process (2)

- Best fit
  - Best fit searches the entire list, from beginning to end, and takes the smallest hole that is adequate.
  - Best fit is slower than first fit because it must search the entire list every time it is called.
  - Somewhat surprisingly, it also results in more wasted memory than first fit or next fit because it tends to fill up memory with tiny, useless holes.
- Worst fit
  - To get around the problem of breaking up nearly exact matches into a process and a tiny hole.
  - Always take the largest available hole, so that the new hole will be big enough to be useful.
  - Simulation has shown that worst fit is not a very good idea either.

Algorithms to allocate memory for a process (3)

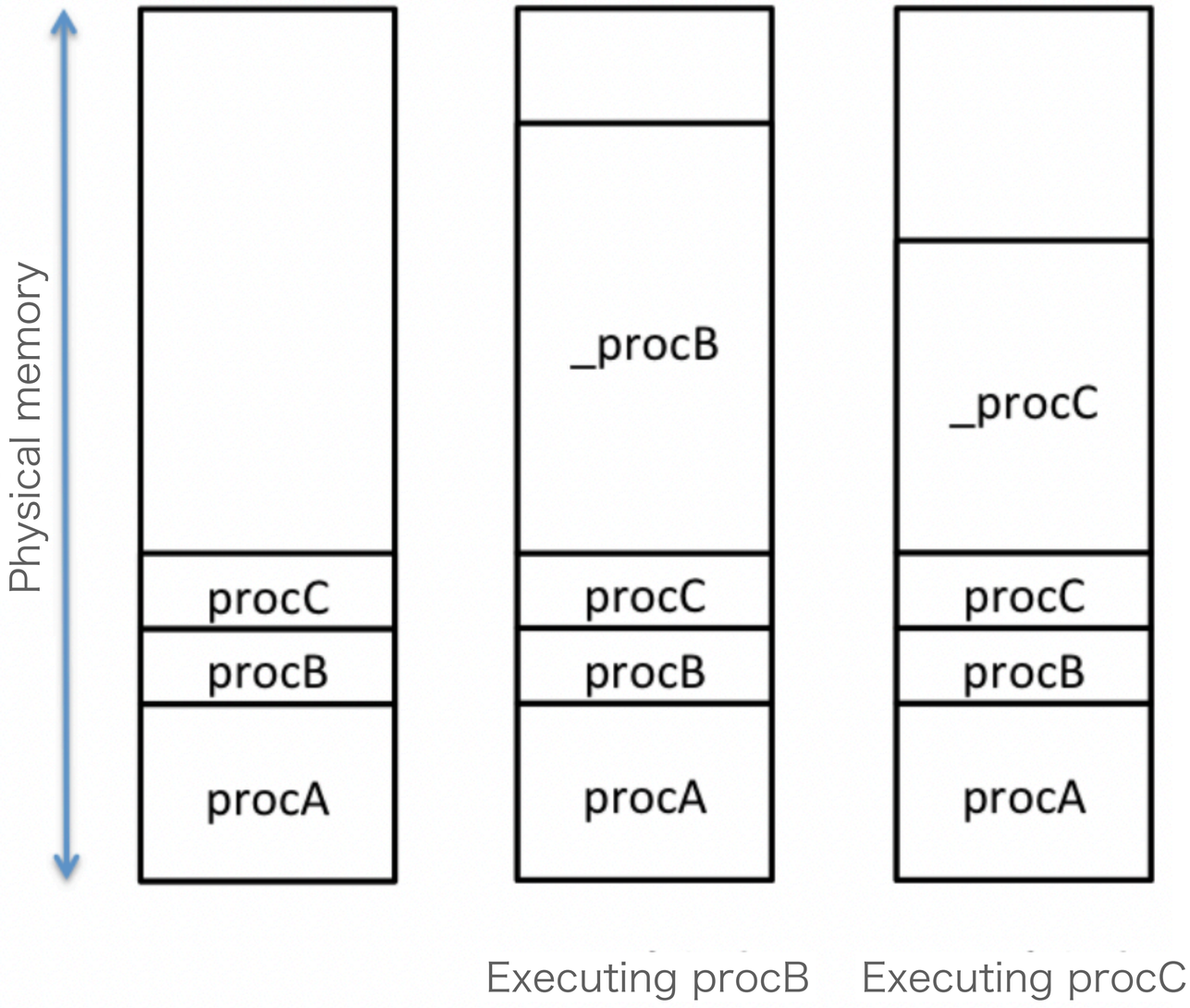
- Considerations
  - All four algorithms can be speeded up by maintaining separate lists for processes and holes.
  - The technique to maintains separate lists for some of the more common sizes requested can be considered. (quick fit)
  - In fact, If we consider only memory management algorithms in swapping, the speed of disk I/O is an order of magnitude slower than the management algorithm, so the speed of the algorithm does not matter much and we should focus on how to reduce the number of holes and keep many processes loaded.

3.3 Virtual Memory

- A way to execute large processes that cannot fit into physical memory.
  - Overlay
    - A method of dividing a program into several parts and loading the parts that are not executed at the same time into a shared memory for execution.
    - The control of the overlay is handled by the programmer or by the language processor.
  - Virtual Memory
    - The entire program is stored on disk.
    - It detects the accessed address (instruction/data/stack) and transfers the data of the corresponding address from the disk to the physical memory.
    - The detection is performed by hardware and the transfer is controlled by the OS.

Overlay

```
procA {
   call procB;
   call procC;
}
procB {
   load _procB;
   call _procB;
}
procC {
   load _procC;
   call _procC;
}
_procB {
   // The real entity of procB.
}
_procC {
   // The real entity of procC.
}
```

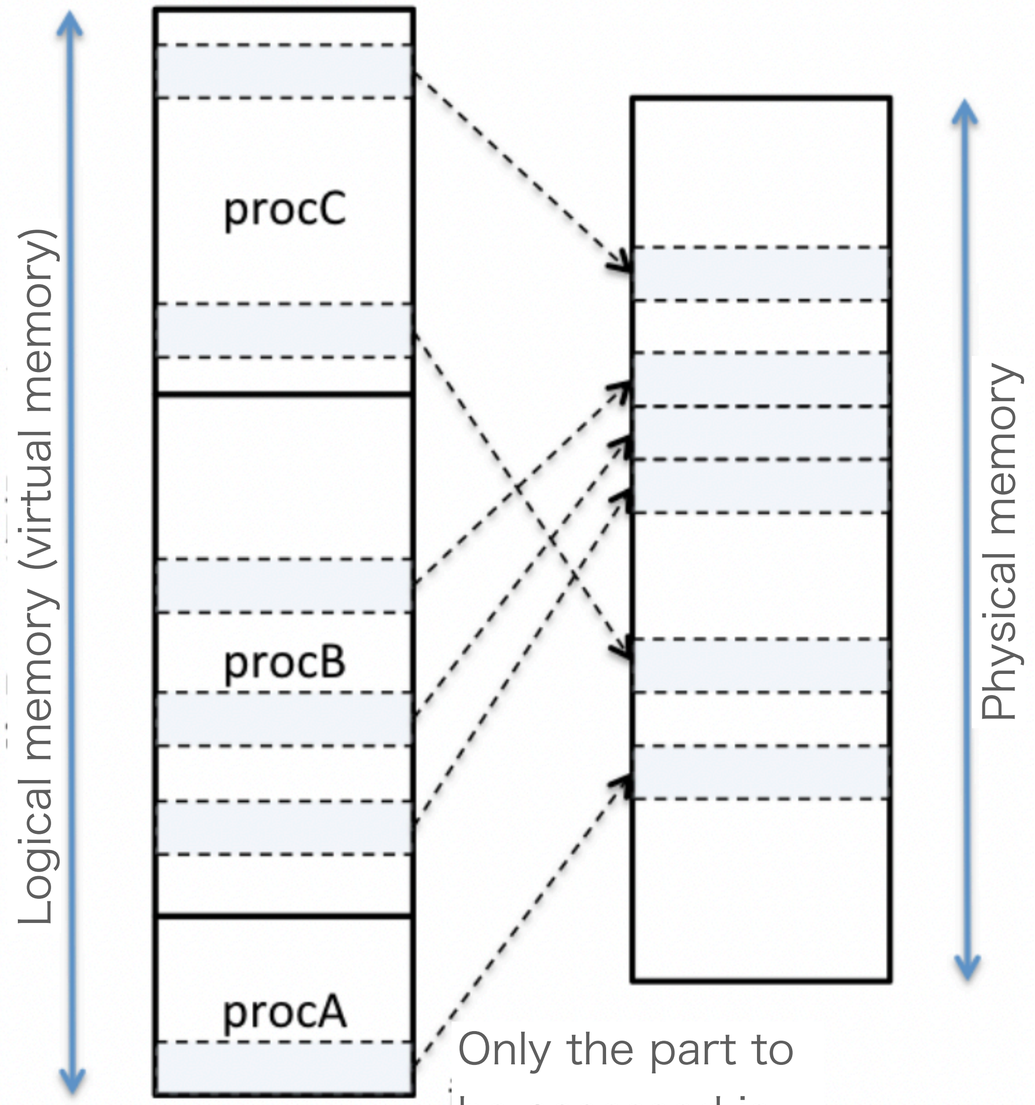


Virtual Memory

```
procA {
  call procB;
  call procC;
}

procB {
  // The real entity of procB
}

procC {
 // The real entity of procC
}
```



3.3.1 Paging

- MMU (Memory Management Unit)
  - Hardware essential for virtual memory

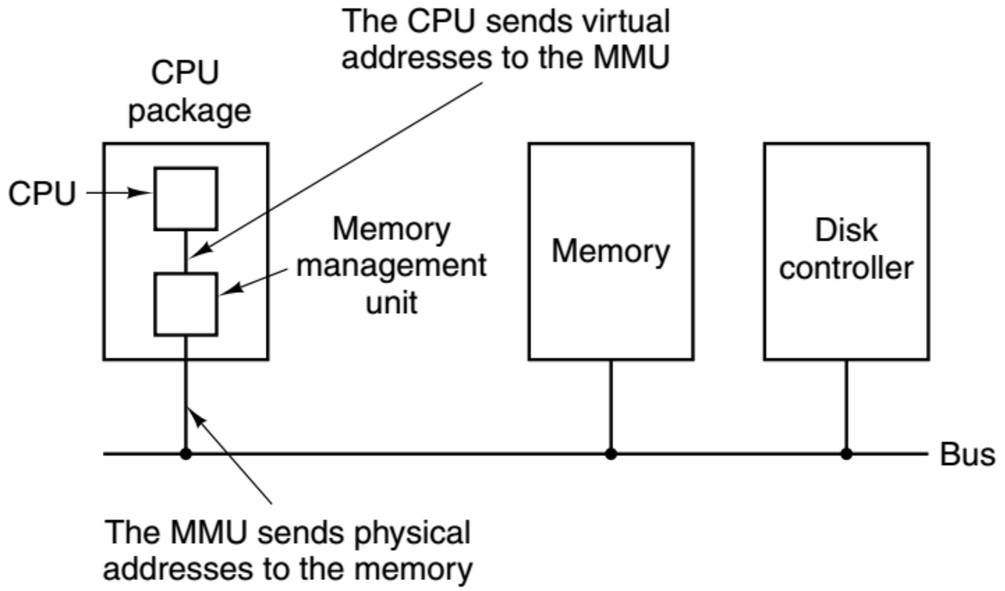


Figure 3-8. The position and function of the MMU. Here the MMU is shown as being a part of the CPU chip because it commonly is nowadays. However, logically it could be a separate chip and was years ago.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.195.

日本語資料

メモリ管理 (1)

- 3.1 メモリ抽象化なし
- 3.2 メモリ抽象化: アドレス空間
  - 3.2.1 アドレス空間という概念
  - 3.2.2 スワッピング
  - 3.2.3 空きメモリの管理
- 3.3 仮想メモリ
  - 3.3.1 ページング
  - 3.3.2 ページテーブル
  - 3.3.3 ページングの高速化
  - 3.3.4 広いメモリ空間のためのページテーブル
- 3.4 ページ置換アルゴリズム
  - 3.4.1 最適ページ置換アルゴリズム
  - 3.4.2 最前未使用 (NRU: Not Recently Used)
  - 3.4.3 先入れ先出し (FIFO: First-In First-Out)
  - 3.4.4 セカンドチャンス
  - 3.4.5 クロックページ
  - 3.4.6 LRU: Least Recently Used
  - 3.4.7 LRU のソフトウェアによるシミュレーション
  - 3.4.8 ワーキングセット
  - 3.4.9 WSClock
  - 3.4.10 ページ置換アルゴリズムのまとめ

メモリ管理 (2)

- 3.5 ページングシステム設計時の課題
  - 3.5.1 局所的割当 v.s. 大域的割当
  - 3.5.2 負荷制御
  - 3.5.3 ページサイズ
  - 3.5.4 命令空間とデータ空間の分離
  - 3.5.5 共有ページ
  - 3.5.6 共有ライブラリ
  - 3.5.7 メモリ写像ファイル
  - 3.5.8 クリーニング方策
  - 3.5.9 仮想メモリインタフェース
- 3.6 実装上の課題
  - 3.6.1 ページングへの OS の関与
  - 3.6.2 ページフォールト処理
  - 3.6.3 命令のバックアップ
  - 3.6.4 メモリ内のページのロック
  - 3.6.5 バッキングストア
  - 3.6.6 方策と機構の分離
- 3.7 セグメンテーション
  - 3.7.1 純粋なセグメンテーションの実現
  - 3.7.2 ページングを併用したセグメンテーション: MULTICS の例
  - 3.7.3 ページングを併用したセグメンテーション: Intel x86 の例

基本的メモリ管理

- OS におけるメモリ管理の 2 種類のカテゴリ
  - スワッピング (主メモリとディスクとの間で実行中のプロセスを行き来させる) 環境で用いられる手法
  - 非スワッピング環境で用いられる手法
- メモリ管理の必要性
  - メモリがどんどん安くなり、コンピュータに搭載されるメモリも増える一方の昨今、メモリを管理する必要などあるのだろうか?
  - メモリは有限のリソース (使い捨てできない)
  - プロセスの大型化
  - より高度なプログラムを実行する小型デバイスの台頭

Comparison of Microsoft Windows minimum hardware requirements (for x86 versions)

Windows version	Processor	Memory	Hard disk
Windows 95[9]	25 MHz	4 MB	~50 MB
Windows 98[10]	66 MHz	16 MB	~200 MB
Windows 2000[11]	133 MHz	32 MB	650 MB
Windows XP[12] (2001)	233 MHz	64 MB	1.5 GB
Windows Vista[13] (2007)	800 MHz	512 MB	15 GB
Windows 7[14] (2009)	1 GHz	1 GB	16 GB
Windows 8[15] (2012)	1 GHz	1 GB	16 GB
Windows 10[16] (2015)	1 GHz	1 GB	16 GB
Windows 11[17] (2021)	1 GHz	4 GB	64 GB

Source: https://en.wikipedia.org/wiki/Software_bloat

3.1 メモリ抽象化なし

- 最も単純なメモリ抽象化は、何も抽象化しないこと
  (スワッピングもページングも行わない)

- 一度に 1 つのプログラムをメモリに配置して実行する。

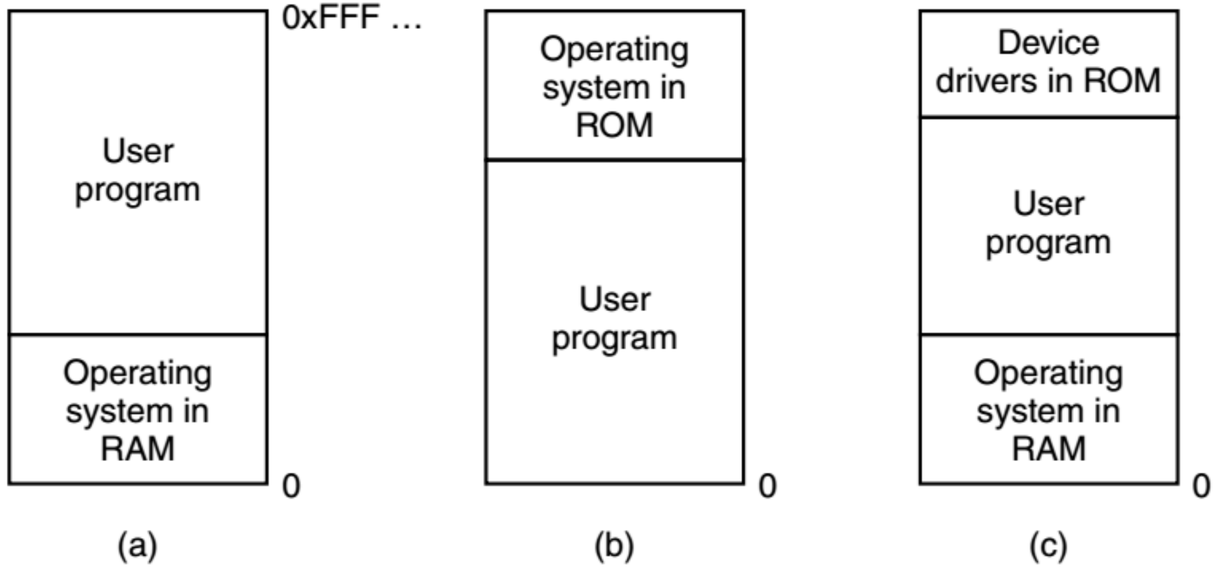


Figure 3-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.183.

メモリ抽象化なしに複数のプログラムを実行する

- スワッピング
  - プロセスのメモリ全体を、ディスクとの間で行き来させる。メモリ上にただ一つのプログラムしか存在しない限り競合は起こらない。
- メモリ上に 2 つのプログラムを連続して配置しようとした場合に問題が…

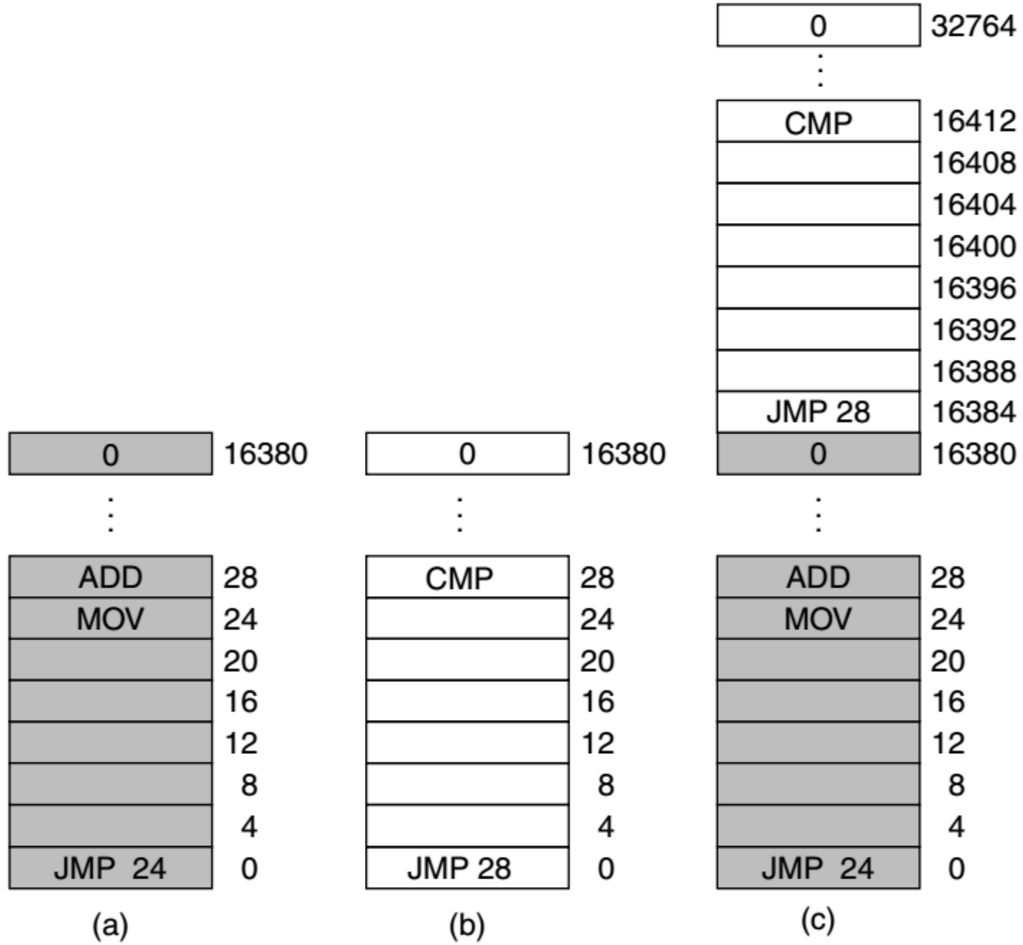


Figure 3-2. Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.184.

3.2.1 アドレス空間という概念

- プログラムが実行のたびに異なるメモリアドレスに格納されることによる問題: 命令には “絶対アドレッシング (メモリ上の特定のアドレスを直接指定してアクセスする)” を用いるものがある。
- 「再配置 (relocation)」による解決
  - メモリ上へのプログラムの展開時に、絶対アドレッシングのアドレス部分を書き換えて実情に合うようにする。
  - 再配置ハードウェアによる解決: プログラムのメモリ領域の開始アドレスを保持する base レジスタを用意し、全てのメモリアクセスを base レジスタからの相対アドレスとして実行する。
- 「保護 (protection)」機能の付加
  - base レジスタに加えて、そのプロセスが利用できるメモリ領域の大きさを示す limit レジスタを用意する。limit レジスタを越えて起きたメモリアクセスは、プロセスの異常な振る舞いとして認識される。

3.2.2 スワッピング

- 全体としてメモリに入りきらないプロセスを同時並行的に実行する手法
  - スワッピング
    - プロセスのメモリ全体を、ディスクとの間で行き来させる。
  - 仮想メモリ
    - プロセスのメモリのうち、必要とされている部分のみをメモリに置く。
    - 不要(!?)になった部分はディスクに追い出される。
- スワッピングの手順:
  - メモリにロードするプロセスを決定する。
  - プロセスのメモリをディスクからメモリに転送する。
  - プロセスを実行する。
  - (プロセスのメモリをメモリからディスクに転送する。)

スワップが行われている例

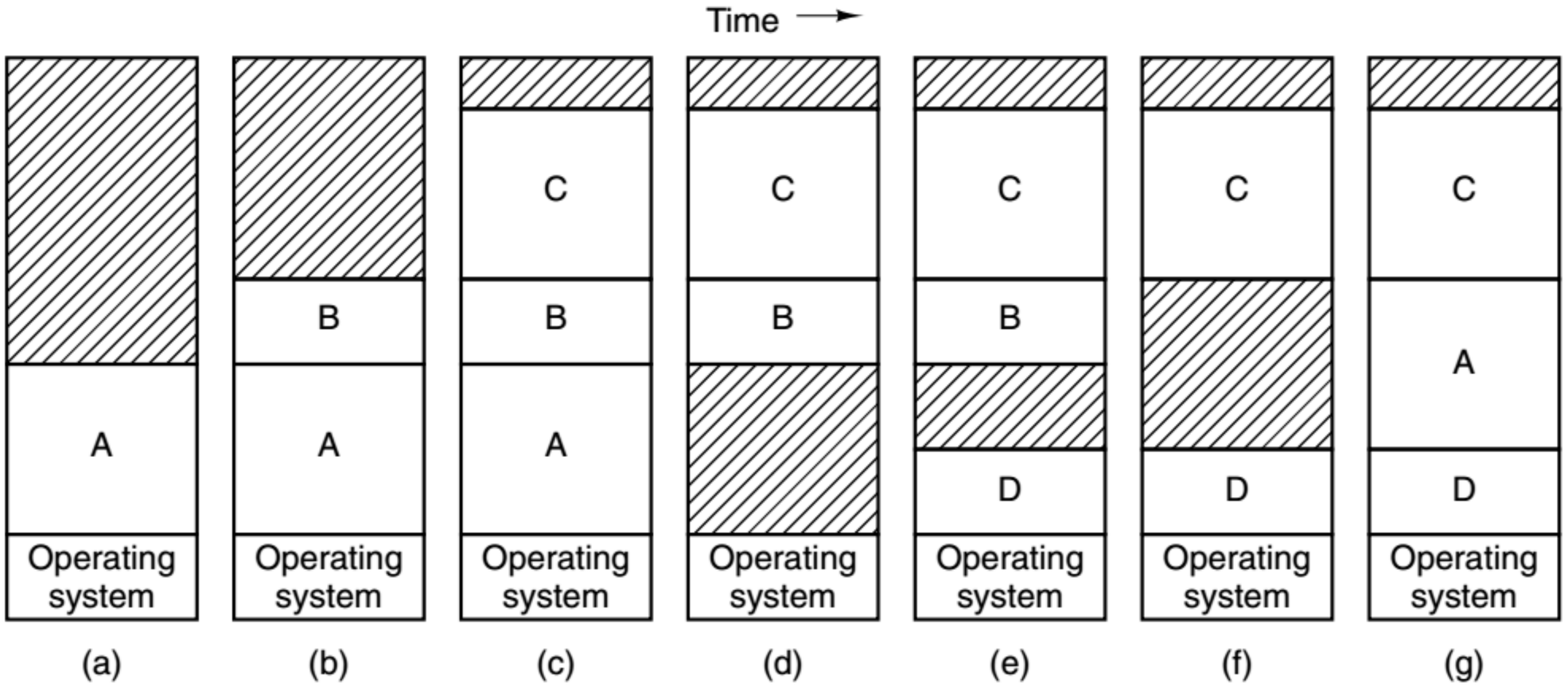


Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.189.

スワップにおける考察

- 固定パーティション v.s. 可変パーティション?
- メモリ内に未使用領域が発生した場合?
  → コンパクション (手間のかかる作業)
- プロセスへのメモリの割り当て量は?
- プロセスが複数のセグメントを利用する場合は?
- プロセスとホールの管理方法は?

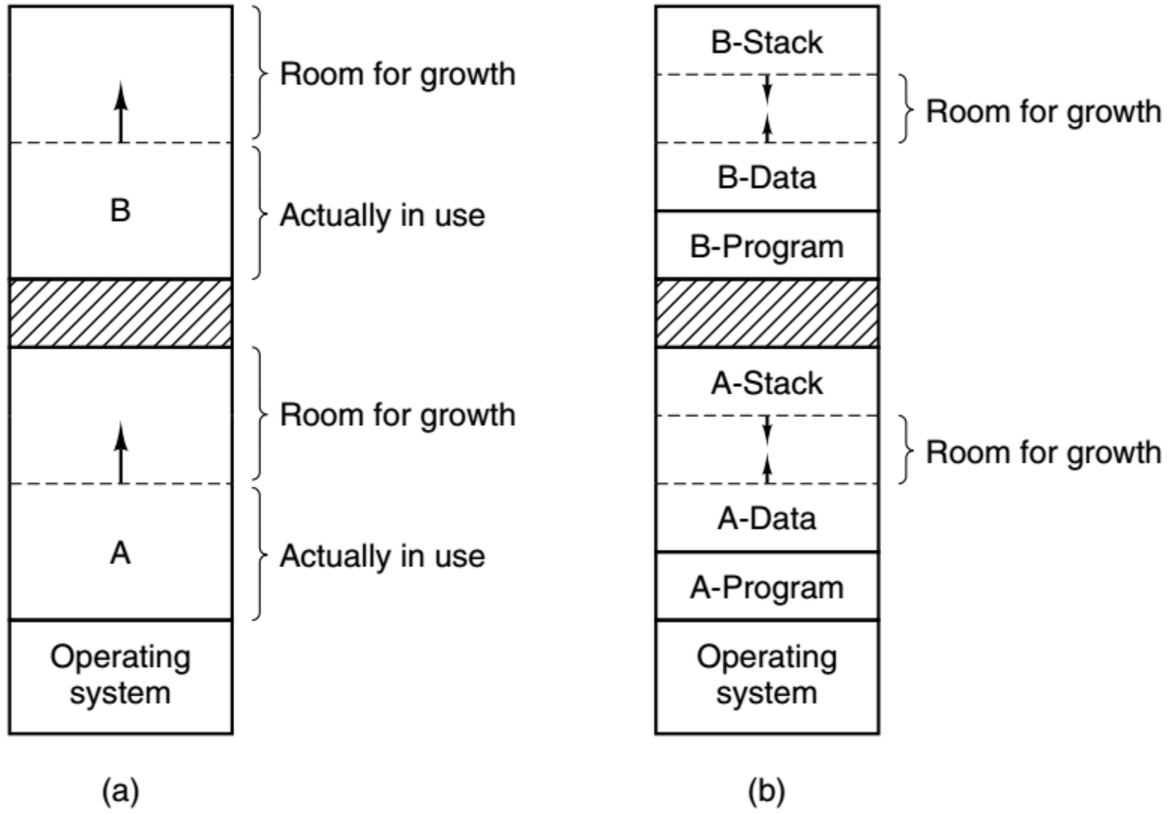


Figure 3-5. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.190.

メモリ管理: bitmap v.s. linked-list

- 管理領域によるオーバーヘッド v.s. 操作の手間

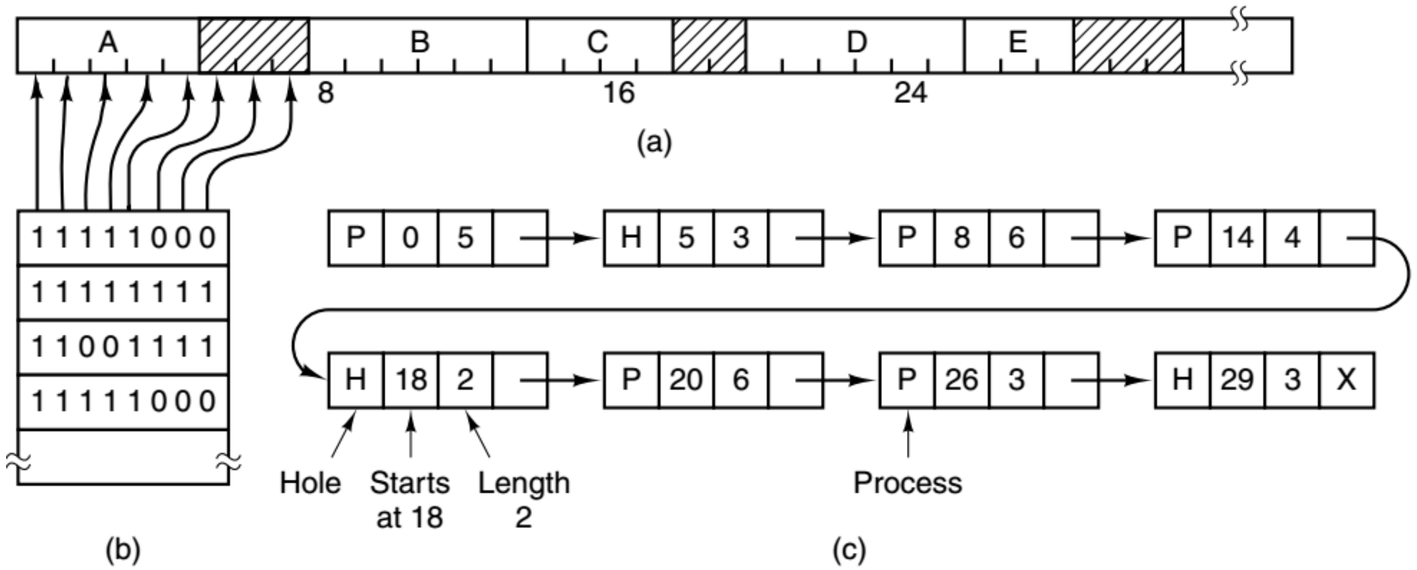


Figure 3-6. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

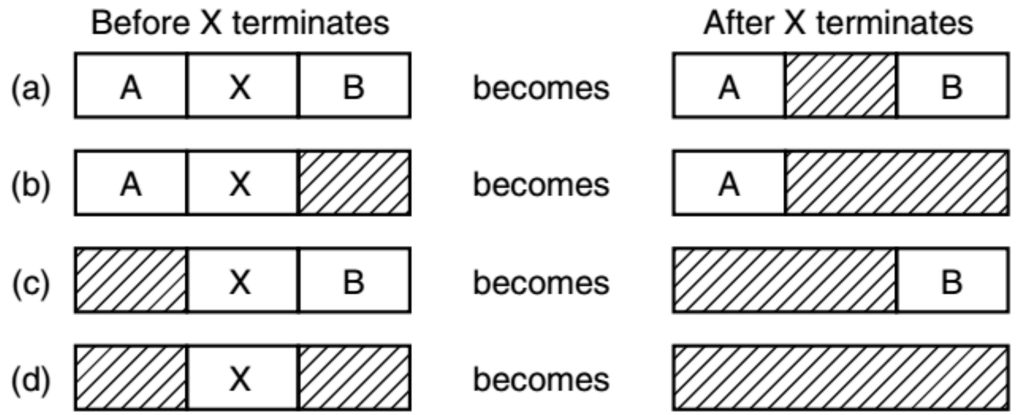


Figure 3-7. Four neighbor combinations for the terminating process, *X*.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.191, 192.

プロセスへのメモリの割り当てアルゴリズム (1)

- あるプロセスをメモリに持ってくる際に、どの大きさのホールを選ぶかという問題。
- 性能指標
  - 管理領域によるオーバーヘッド
  - 割り当てに要する時間

- ファーストフィット (First fit)
  - 空き領域(ホール)を探してスキャンし、最初に見つかった十分な大きさのホールを分割して割り当てる。
  - 余分な探索がないので高速。
- ネクストフィット (Next fit)
  - ファーストフィットの修正版。
  - 最後に割当が起きた位置を記憶しておき、新しい割り当ては記憶された位置から探索する。
  - シミュレーション(Bays:1977) ではファーストフィットよりも性能が悪い。

プロセスへのメモリの割り当てアルゴリズム (2)

- ベストフィット (Best fit)
  - プロセスを格納できる十分な大きさを持つ、最小のホールを探索。
  - 必ず全探索になるので性能は悪い。
  - さらに、Best fit は First & Next fit よりもメモリ効率が悪い。(使いものにならない小さなホールを産み出しやすい)
- ワーストフィット (Worst fit)
  - 小さなホールへの対応。
  - 最も大きなホールを探し、分割して割り当てる。
  - シミュレーションでは、あまり良い結果は出なかった。

プロセスへのメモリの割り当てアルゴリズム (3)

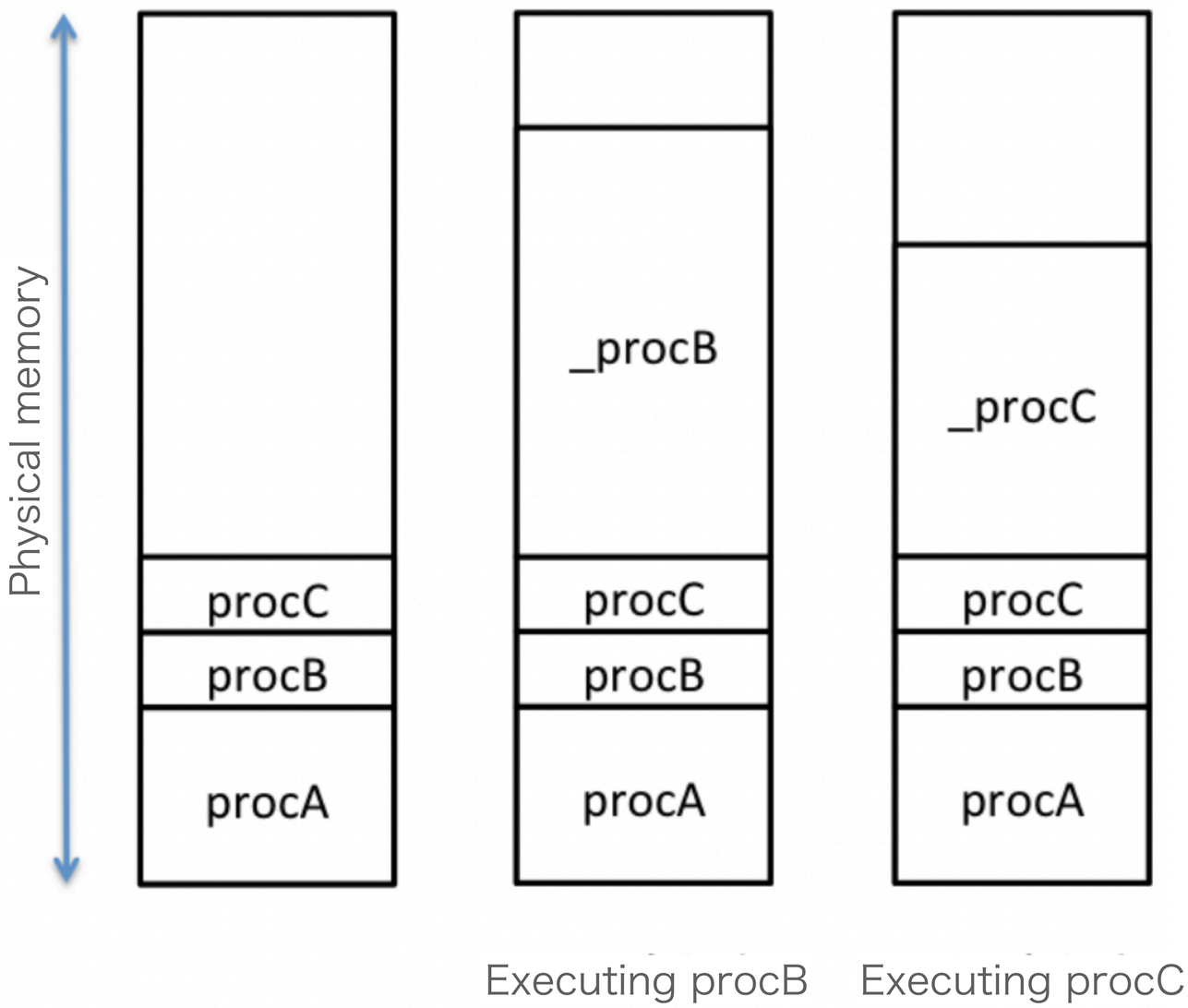
- 考察
  - ４ つのアルゴリズムはホールを探すので、ホールのリストとプロセスのリストに分けて管理することもできる。
  - よく要求されるサイズのものを分離して管理する方法も考えられる (クイックフィット)
  - よく考えると…
    スワップにおけるメモリ管理アルゴリズムに限定して言えば、ディスク I/O の方が管理アルゴリズムよりも桁違いに遅いので、アルゴリズムの速度はあまり問題にならず、いかにホールを減らし、多くのプロセスをロードしておくかに注力すべきである。

3.3 仮想メモリ

- 物理メモリに入りきらない大きなプロセスを実行する方法
  - オーバーレイ
    - プログラムをいくつかの部分に分割し、同時期に実行しない部分を共通のメモリにロードして実行する方法。
    - オーバーレイの制御はプログラマ、あるいは言語の処理系が処理する。
  - 仮想メモリ
    - プログラム全体がディスク上に格納されている。
    - アクセスされたアドレス (命令・データ・スタック) を検出し、該当するアドレスのデータをディスクから物理メモリに転送する。
    - 検出はハードウェア、転送は OS の制御によって行われる。

オーバーレイ

```
procA {
   call procB;
   call procC;
}
procB {
   load _procB;
   call _procB;
}
procC {
   load _procC;
   call _procC;
}
_procB {
   // The real entity of procB.
}
_procC {
   // The real entity of procC.
}
```

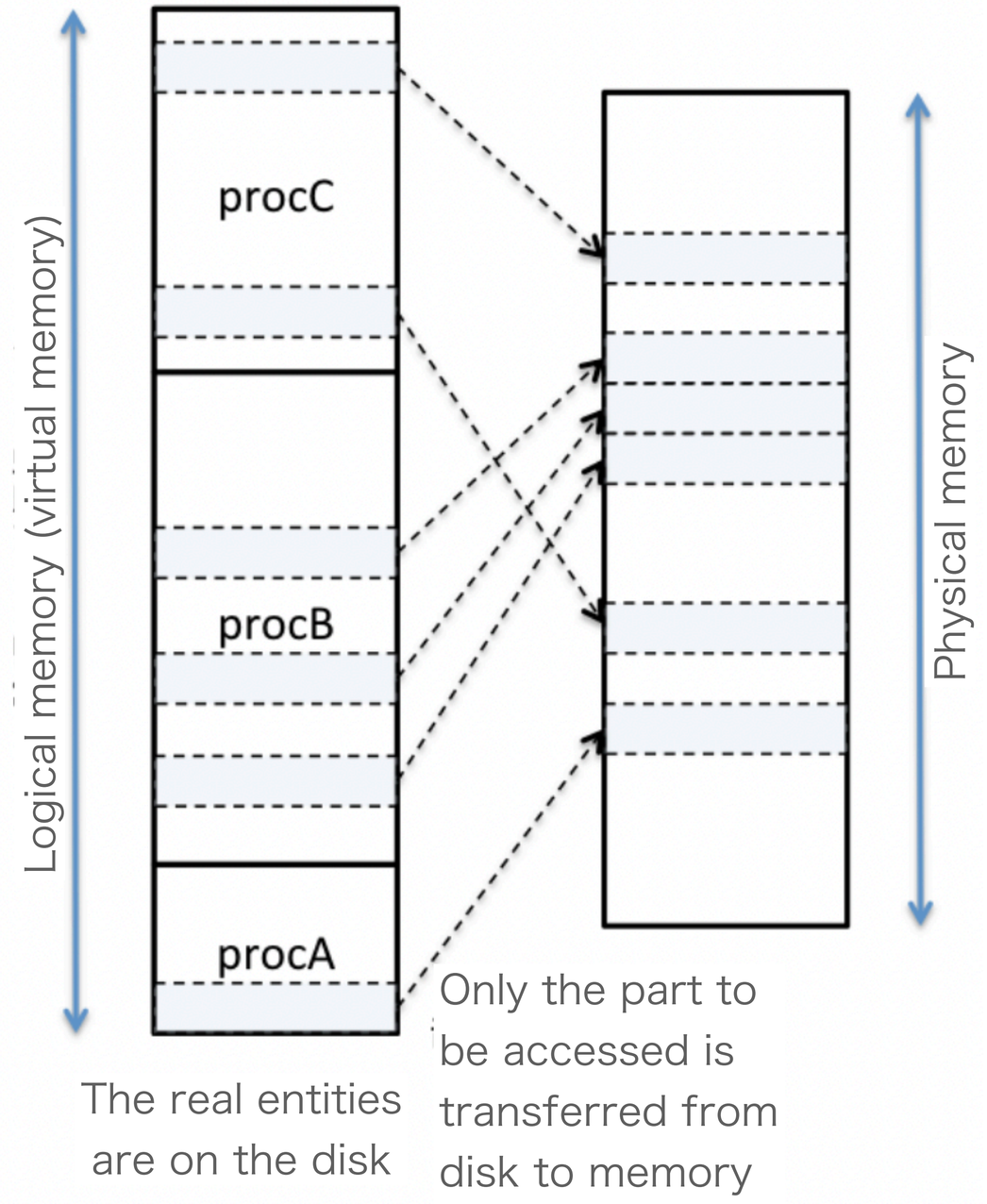


仮想メモリ

```
procA {
  call procB;
  call procC;
}

procB {
  // The real entity of procB
}

procC {
 // The real entity of procC
}
```



3.3.1 ページング

- MMU (Memory Management Unit)
  - 仮想メモリに不可欠なハードウェア

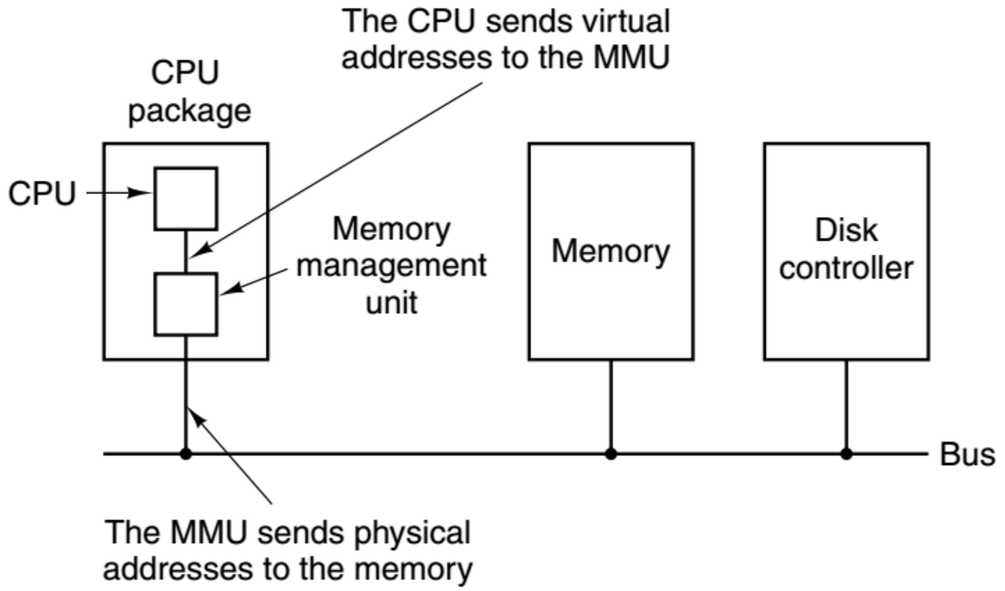


Figure 3-8. The position and function of the MMU. Here the MMU is shown as being a part of the CPU chip because it commonly is nowadays. However, logically it could be a separate chip and was years ago.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.195.

3.3.1 ページング (2)

- メモリを「ページ」と呼ばれる一定の大きさの連続領域に分割して仮想メモリを実現する方法。

“MOV REG, 1000” の処理

ページングなし	ページングあり
命令 (MOV REG, 1000) の読み込み	命令 (MOV REG, 1000) の読み込み
物理アドレスとして 1000 を生成	論理アドレスとして 1000 を生成
	メモリアドレス 1000 のページ番号を求める
	該当ページが、どの物理ページ(ページフレーム)に格納されているのかを調べる
	ページフレーム番号と論理アドレスを合成して物理アドレスを生成
物理アドレスの内容を REG に転送	物理アドレスの内容を REG に転送

3.3.2 ページテーブル

- “MOV REG, 0” は、物理アドレス 8192 をアクセスする

論理アドレス 0
↓
仮想ページ 0
↓
ページフレーム 2
↓
物理アドレス 8192

- “MOV REG, 8196” は、物理アドレス 24580 をアクセスする。

- “MOV REG, 33000” は、「ページフォールト」を引き起こす

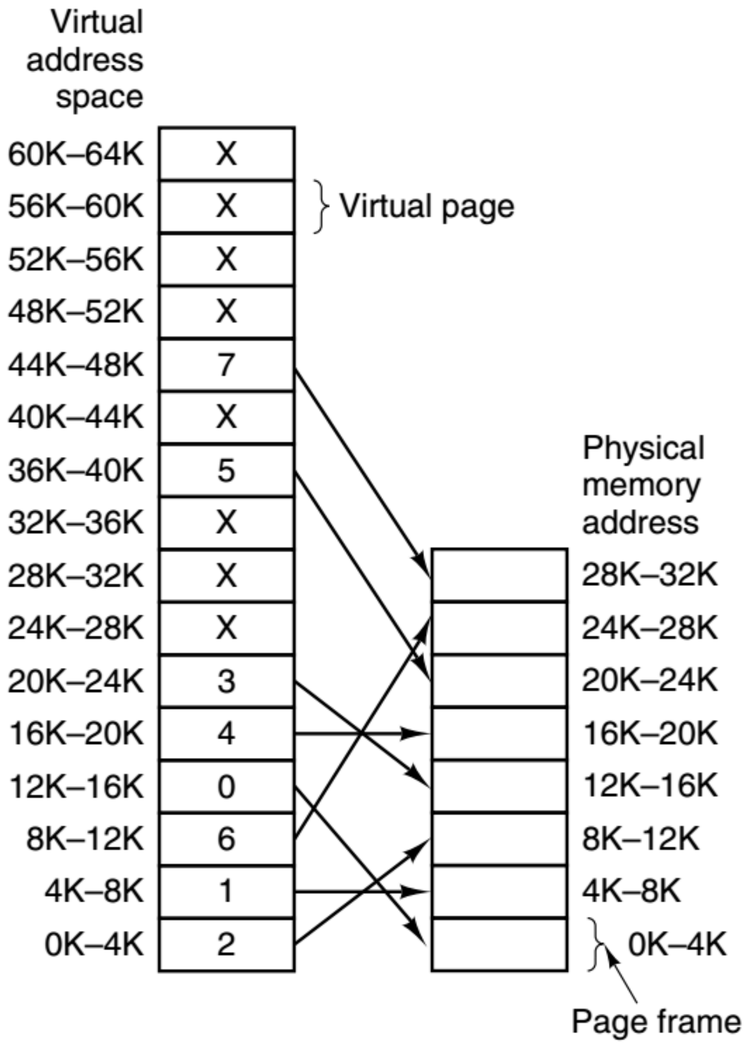


Figure 3-9. The relation between virtual addresses and physical memory addresses is given by the **page table**. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.197.

MMU の内部構造

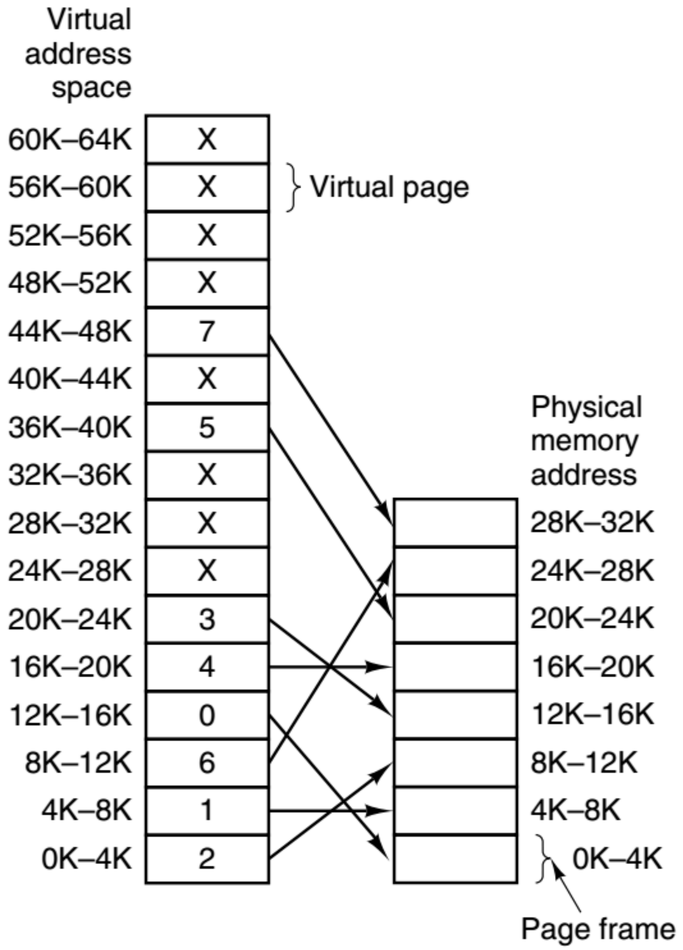


Figure 3-9. The relation between virtual addresses and physical memory addresses is given by the **page table**. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287.

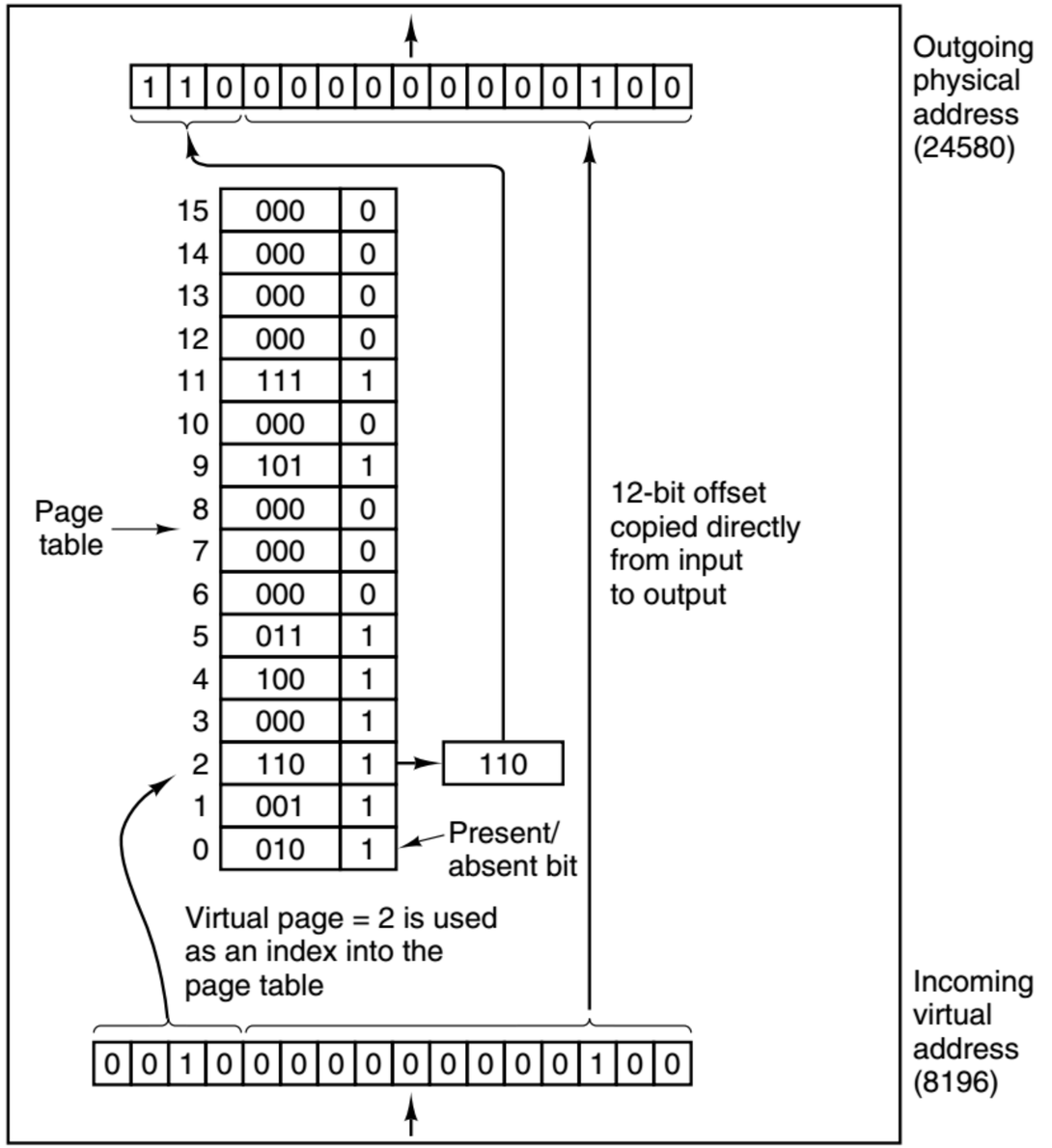


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.197, 199.

3.3.4 ページテーブルのサイズ

- ページテーブルは MMU 内の高速なメモリとして実現する必要があるが、大きな仮想空間では、ページテーブルのサイズも大きくなる。
- 試算: ページサイズ 4KB、アドレス空間 32 ビットの場合、ページテーブルは 1M 個 (2^20) のエントリが必要である。ページフレーム番号も 20 ビット必要になるため、ページテーブルの容量は少なくとも 21M ビット (2.75Mバイト) になる。
- ページテーブルは CPU の L1 キャッシュと同程度の速度で動作しなければならない (そうでなければ、ページテーブルを引く動作がメモリアクセス速度を律速してしまう) が、最新 CPU (e.g. Intel Core i7) の L1 キャッシュ容量が 64K バイトであることを考えると、2.75M バイト (約40倍) の容量は看過し難い大きさである。
- 2.75M バイトのページテーブルは、主メモリに配置するしかないが、そうすると、CPU による 1 回のメモリアクセスにつき、2 回のメモリアクセスが行われてしまう。

マルチレベル ページテーブル

- 「メモリ参照の局所性」を利用して、高速なメモリに格納するページテーブルを、その時点で頻繁にアクセスされているいくつかのページ群に限定する。
  - 第1レベルのページテーブル (1024エントリ)
    1エントリ11(10+1)ビットで1.4Kバイト
  - 第2レベルのページテーブル (1024エントリ)
    1エントリ11(10+1)ビットで1.4Kバイト
  - 32KBのページテーブル用メモリなら、第一レベルに加えて第2レベルを21個保持できる(可能性がある)

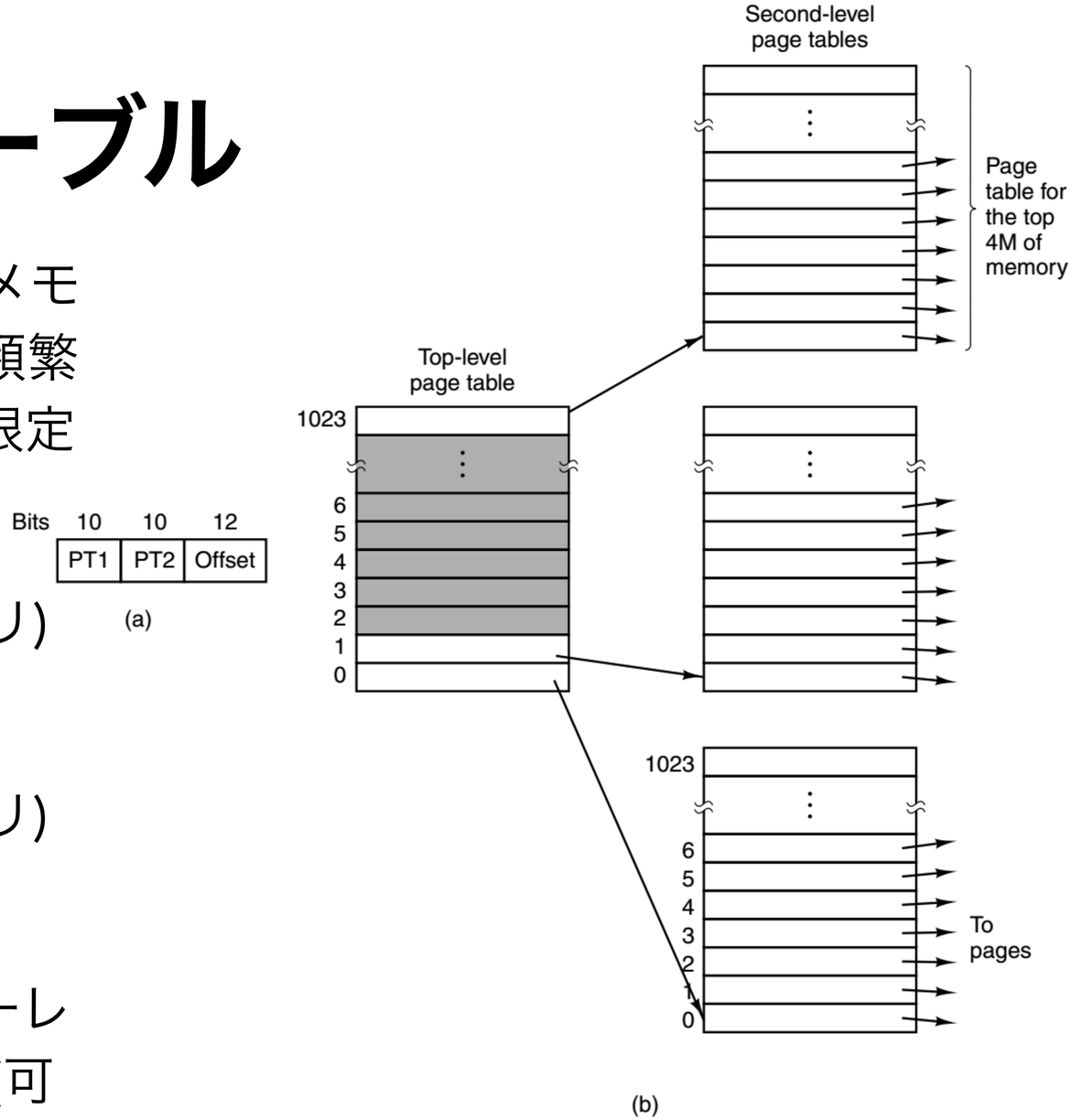


Figure 3-13. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.206.

ページテーブルエントリ(PTE)の構造

- 典型的なページテーブルエントリ
  - ページフレーム番号
  - 存在ビット
  - 保護ビット: アクセスの種類 (R, W, X) のそれぞれを許可するかどうか
  - 修正ビット: ページフレームへの書き込みアクセスが起きた場合に設定される
  - 参照ビット: ページフレームへの何らかのアクセスが起きた場合に設定される
  - キャッシング不可ビット: ページがデバイスレジスタ、またはマルチプロセッサで共有されているメモリの場合等に設定する。

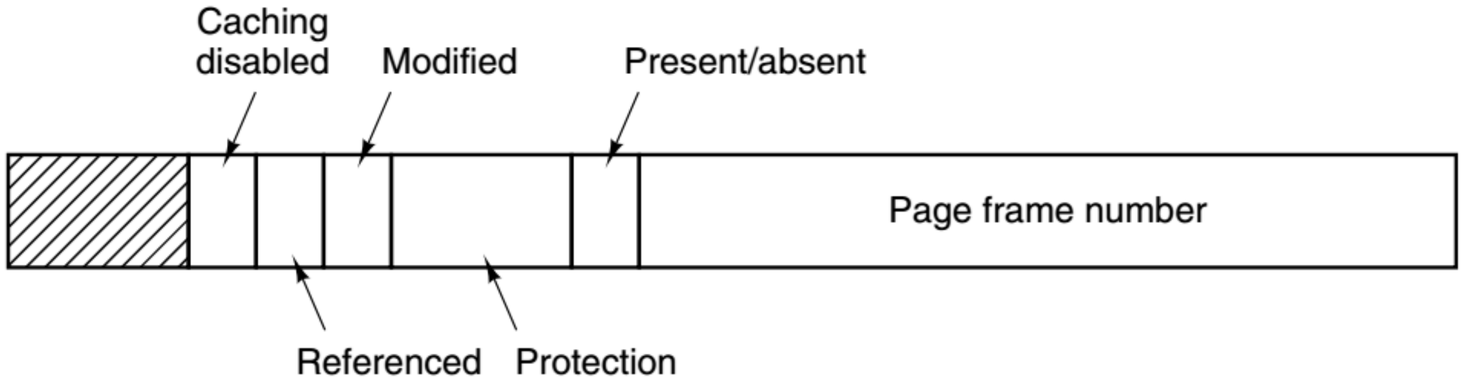
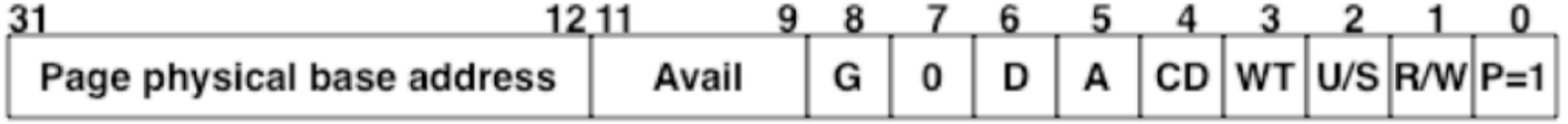


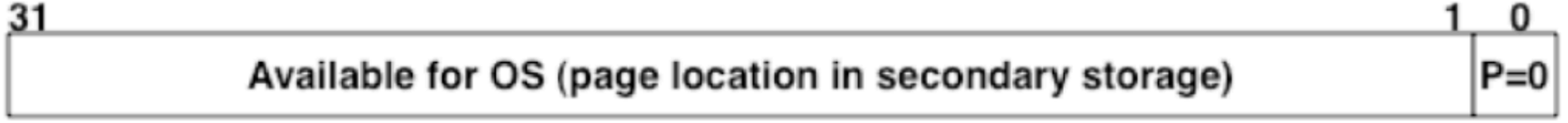
Figure 3-11. A typical page table entry.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.200.

Pentium III の PTE



Page base address: 20 most significant bits of physical page address (forces pages to be 4 KB aligned)
Avail: available for system programmers
G: global page (don't evict from TLB on task switch)
D: dirty (set by MMU on writes)
A: accessed (set by MMU on reads and writes)
CD: cache disabled or enabled
WT: write-through or write-back cache policy for this page
U/S: user/supervisor
R/W: read/write
P: page is present in physical memory (1) or not (0)



3.3.3 TLB

- マルチレベルページテーブルを採用しても、なおかつ現実には、ページテーブルは主メモリに置かれる。
- メモリ参照ごとに 2 回のメモリアクセスが起きる可能性がある。
- これを避けるために、PTE に対してある種のキャッシュ機構を設ける。これを TLB (Translation Look-aside Buffer) と呼ぶ。

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

Figure 3-12. A TLB to speed up paging.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.203.

ソフトウェアによる TLB 管理

- 古典的なアーキテクチャでは、TLB の管理 (主に TLB がミスした場合の主メモリ上のページテーブルの検索と TLB への導入) はハードウェアが行っていた。
- RISC アーキテクチャでは、TLB の管理をソフトウェアで行う場合が多い。
  - TLB のエントリ数が十分多く、ミスの頻度が少ない場合はソフトウェア処理によるオーバヘッドが許容できる。
  - MMU の回路規模を減らすことができ、空いたチップ領域を、例えば CPU のキャッシュに廻すことができる。
  - TLB ミスによるページテーブルの検索はメモリ参照を伴うが、そのメモリ参照がさらに TLB ミスを引き起こす場合がある。これを防ぐために、キャッシュメモリ中に TLB 専用の領域が設けられる場合もある。

3.3.4 逆引きページテーブル

- CPU アーキテクチャの 64 ビット化に伴い、ページテーブルのサイズは再度無視できない容量になってきた。
- そこで、1 つの仮想ページに 1 つのエントリではなく、1 つの物理ページフレームにつき 1 つのエントリを設ける方法が考案された。
- この方法では、テーブルの各エントリはプロセス識別子と、そのプロセス内の仮想アドレスを保持し、物理ページフレーム番号でインデックスされる。このことは、仮想アドレスの一部を使った配列的手法では、求めるエントリを検索することができないことを意味する。
- ナイーブな方法では全対策が必要になるが、仮想アドレスのハッシュ値をキーに持つハッシュテーブルを使用することで、検索が高速化できる。

逆引きページテーブルの例

64 ビット仮想アドレス空間、1GB 物理メモリ、4 KB/ページ の場合

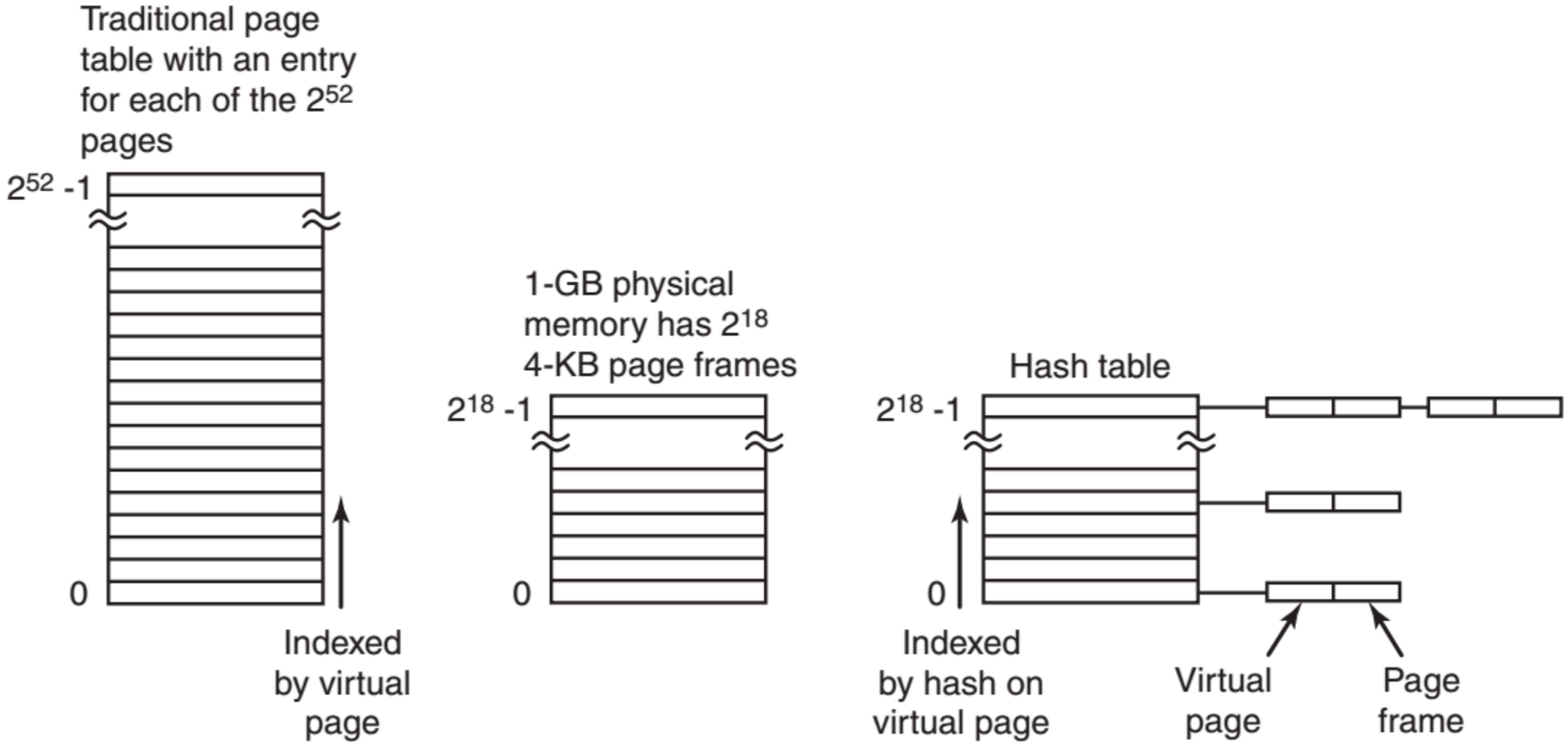


Figure 3-14. Comparison of a traditional page table with an inverted page table.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.208.

3.4 ページ置換アルゴリズム

- ページフォールトが発生すると、OS はアクセスされたページをディスクからメモリ上へ転送するために、「ページ置換アルゴリズム」に基づいて追い出すページを決定する。

- 考慮されるべき事柄

  - 書き込みがあったページはディスクに書き戻す必要がある。反対に、書き込みが無かったページは破棄するだけで良い。

3.4.1 最適置換アルゴリズム

- 各ページを、そのページが最初に参照されるまでに実行される命令数(≒時間)でラベル付けし、ラベル値の最も大きなページを置換する。

- 現実には実現不可能(プログラム全体の実行命令を予測しないといけない
  ― 完全なシミュレーション = 予備事項が必要)

- 実現可能なアルゴリズムの性能指標としては利用可能。

3.4.2 NRU (Not Recently Used)

- 「頻繁に使用されているページよりも、少なくとも最後の 1 クロック刻みの間参照されていなかったものを追い出す」(変更が無ければ、より良い選択となる)
- ページの R (参照) および M (変更) ビットの状態をもとに、ページを 4 つのクラスに分類する。
- プロセスの開始時に RM はクリアされ、以後 R ビットが定期的(クロック割り込み)にクリアされる。
- ページ置換時にはクラスの小さなものから追い出すページをランダムに選択する。
- オーバーヘッドが小さい。

クラス	R (参照)	M (変更)
0	0	0
1	0	1
2	1	0
3	1	1

3.4.3 先入れ先出し (FIFO)

- 最も長い間参照されていないページを追い出す。
- 全ページフレームを連結したリストを使用。
- 先頭は最も古くメモリに格納されたページ、最後尾は最も最近に格納されたページ。
- 先頭から追い出す。
- 新たにディスクから読み出された (ページインした) ページは、リストの最後尾に追加する。
- オーバーヘッドが小さい。
- ページ参照と独立したアルゴリズムのため、頻繁に参照されるページが追い出されることがある。そのため、純粋な FIFO が用いられることは現実的にはない。

3.4.4 セカンドチャンス

- 頻繁に参照されるページが追い出されないようにした FIFO の改良版。
- リストを先頭から調べ、R ビットが 0 のページを追い出す。
- R ビットが 1 のページは、R ビットをクリアし、リストの最後尾に移動する。(今メモリにロードされたばかりという状況をシミュレートする。)
- リストの維持にオーバーヘッド。

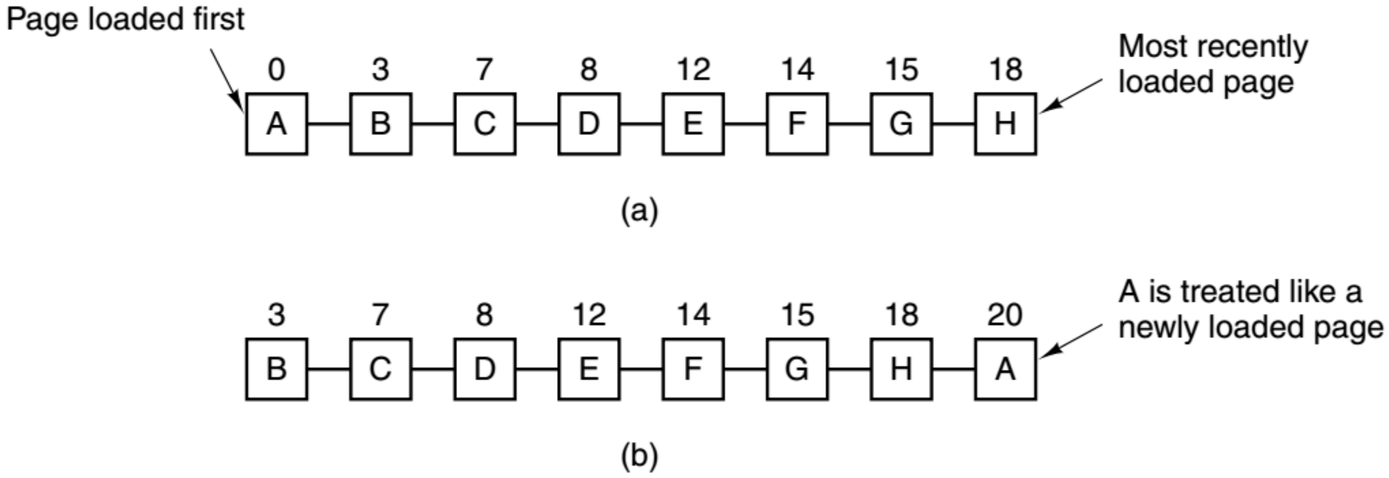
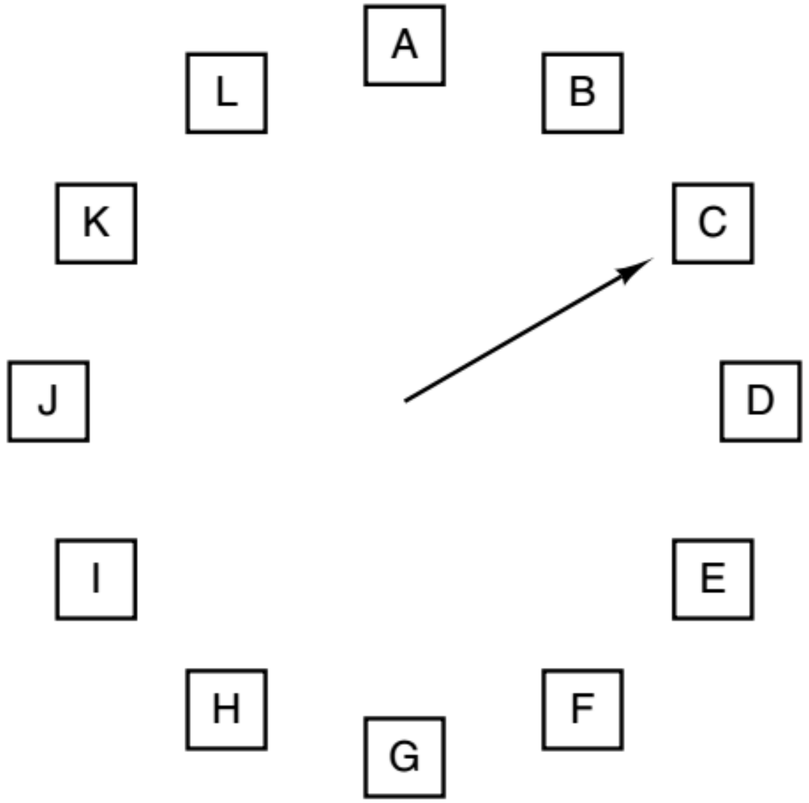


Figure 3-15. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set. The numbers above the pages are their load times.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.212.

3.4.5 クロックページ

- セカンドチャンスの効率化。



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

Figure 3-16. The clock page replacement algorithm.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.213.

3.4.6 LRU (Least Recently Used)

- 推測:「最後の数命令で頻繁に使用されたページは、その後の数命令でも頻繁に使用されるだろう」
- 最も長い間使用されていないページを追い出す。
- 実現可能であるが、コストは小さくない (完全な LRU を実現するには、全ページフレームのリストをメモリ参照ごとに更新しなければならない)。

LRU の実現: ハードウェア (1)

- 方法 1
  - 命令実行ごとにインクリメント (+1) されるカウンタ C を用意。
  - PTE には、カウンタ値を格納する場所を用意し、メモリ参照ごとに、対応する PTE にカウンタ C の値を格納。
  - PF 発生時には、カウンタ値の最小値を持つ PTE を探して置換。

LRU の実現: ハードウェア (2)

- 方法2
  - ページフレームが n 個の場合、n × n ビットの行列を用意 (初期値0)。
  - ページフレーム k が参照された場合、行 k をすべて 1 にセット、列 k をすべて 0 にクリア。
  - 行内のビット列を 2 進数と見なしたとき、値が最も小さいものが最も過去に参照されたページ。

3.4.7 LRU のソフトウェアシミュレーション (NFU)

- ページごとにソフトウェアで制御するカウンタを設ける。
- クロック割り込みごとにページをスキャンし、R ビットの内容 (0 or 1) をカウンタに加える(ページ参照の頻度を記録することになる)。
- PF 発生時には、最小のカウンタ値を持つページが置換対象となる。
- 問題点: 過去の履歴をいつまでも記憶してしまう。
  - 時間の経過に伴って移動する「頻繁に参照されるページ」を扱えない。
  - 新しくページインしたページの初期値はどうすれば良いのか(同じ)。
  - 時間の経過に伴って過去の履歴を消していけば良い (エージング)。

エージング付き NRU

- クロック割り込みのタイミングで…
  - ページカウンタを右へシフト (過去の履歴値を 2 で割ることに相当)
  - ページのカウンタの最上位ビットに、そのページの R ビットを転送

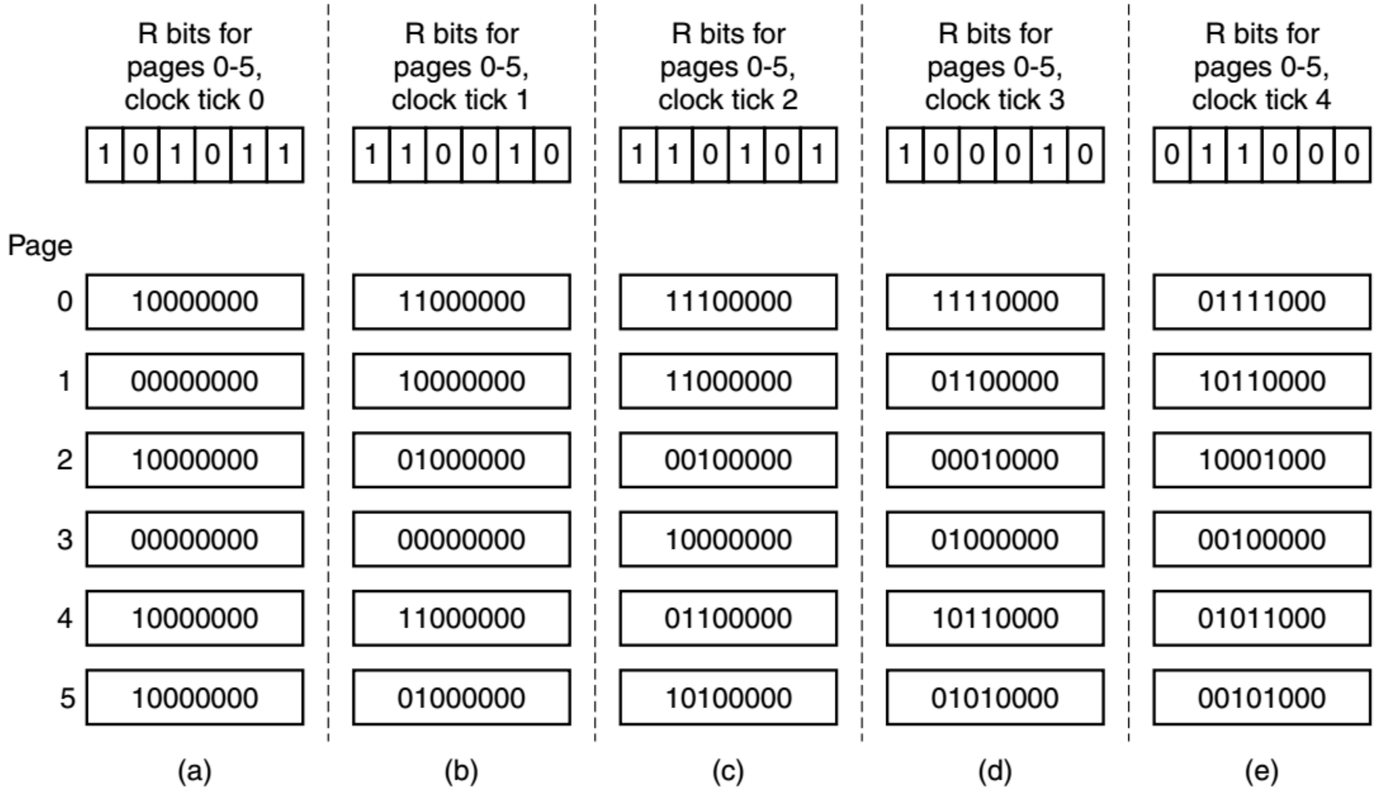


Figure 3-17. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.215.

3.4.8 ワーキングセット

- 一般的なプロセスには「参照の局所性」がある。
- ワーキングセット: プロセスがその時点で使用しているページの集合。
  - ワーキングセット全体がメモリ内にあれば、プロセスは次の実行フェーズに移行するまで PF 無しで実行できる。
  - スラッシング (slashing): ワーキングセット全体をメモリに置けない場合に、頻繁に PF が発生し、プロセスの実行 (そしてシステム全体の性能) が遅くなる現象。
  - ワーキングセットモデル: 各プロセスのワーキングセットを記録し、プロセス実行前にワーキングセットをメモリにロードする方法
    (= プリ・ページング v.s. デマンド・ページング)。

ワーキングセットとそのサイズ

- ワーキングセットの定義: 時刻 t において、最近の k 回のメモリアクセスで参照されたページの集合 w(k, t)
- w(k, t) は時刻 t と共に変化する。
- k を増加させると、w(k, t) は最初は急激に増加するが、次第に一定のサイズに近づく (参照の局所性)。
- この漸近的挙動により、ワーキングセットの内容は、ある程度以上の k の値に対して敏感ではない。

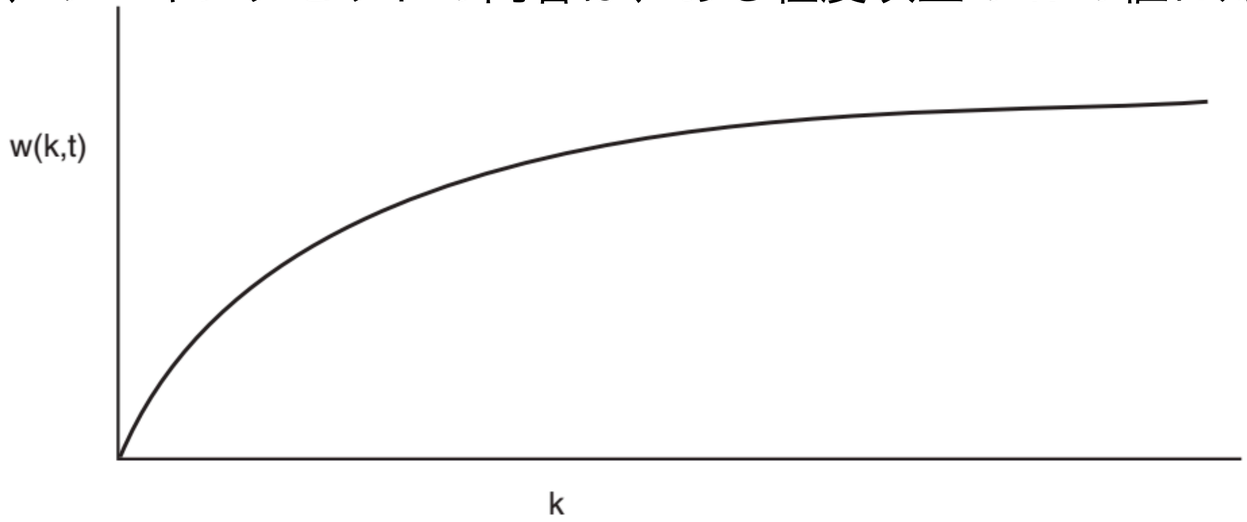


Figure 3-18. The working set is the set of pages used by the *k* most recent memory references. The function *w*(*k*, *t*) is the size of the working set at time *t*.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.217.

ワーキングセットに基づいたページ置換

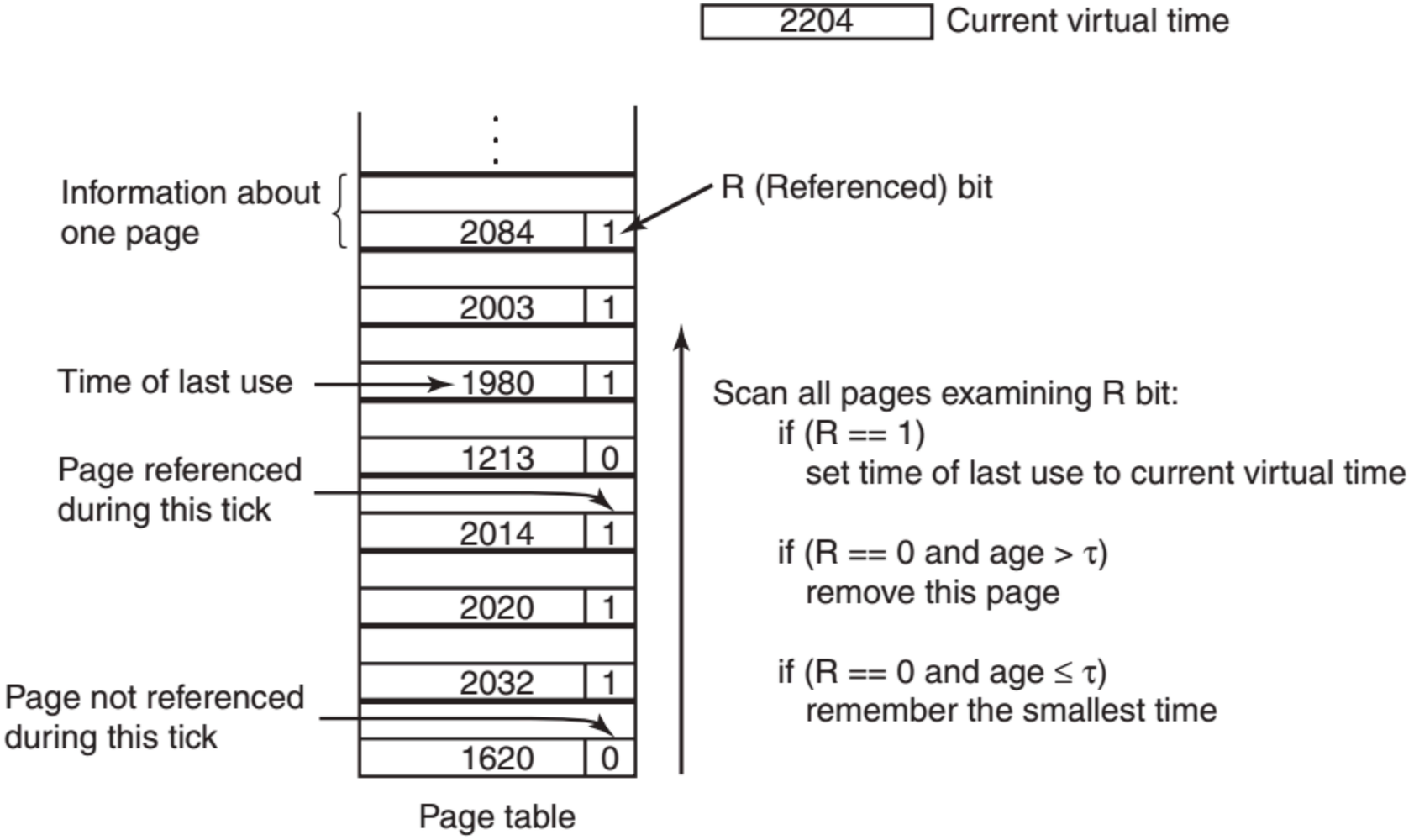


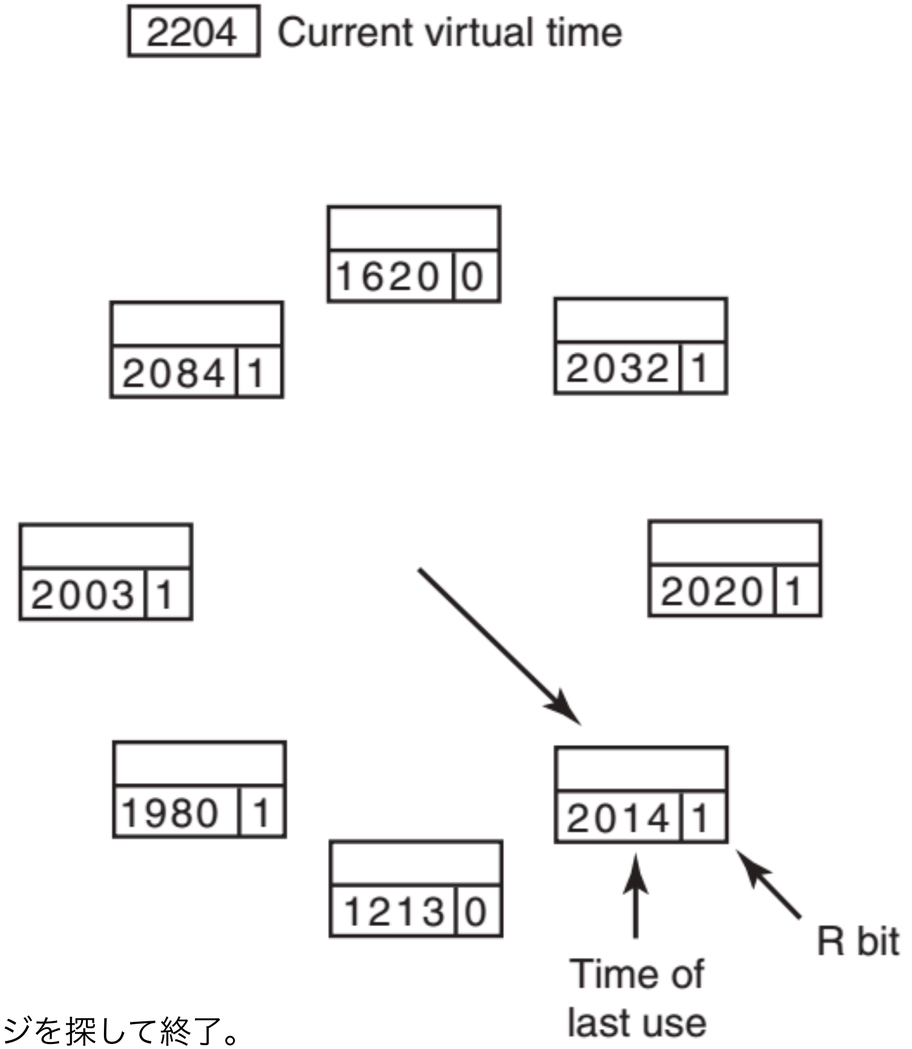
Figure 3-19. The working set algorithm.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.218.

3.4.9 WSClock

- ページテーブル全体のスキャンはオーバーヘッドが大きい。
- WSClock = ワーキングセット + クロックアルゴリズム
- 現実に広く利用されている。

- 2-pass のアルゴリズム
  - 1 回目の pass
    - R = 1 ならば R = 0, ts 更新
    - R = 0 かつ M = 0 かつ age > τ があれば、そのページを選んで終了
    - R = 0 かつ M = 0 かつ age < τ ならば、そのページを配置しておく。
    - R = 0 かつ M = 1 ならば、そのページの write back をスケジュール
  - 2 回目の pass
    - 1 回目のパスでスケジュールされた write back があるなら、write back が終了したページを探して終了。
    - write back がスケジュールされなかったときは、全ページが WS に含まれている。このときは情報不足なので、たとえば 1 回目の pass で記憶したクリーンなページから選択する。
    - クリーンなページが無い場合には、現在のページを選択する。



Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.220.

WSClock

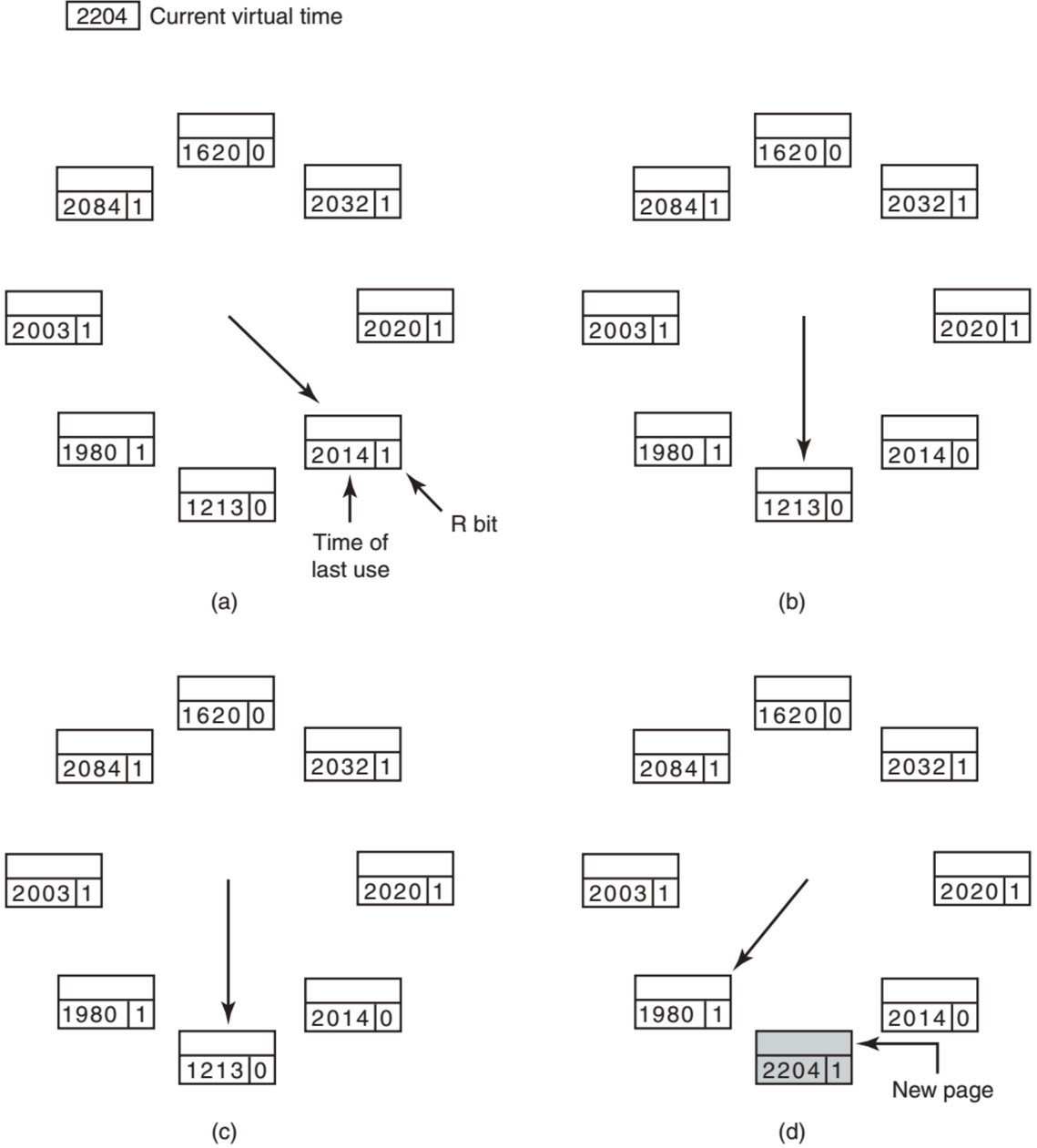


Figure 3-20. Operation of the WSClock algorithm. (a) and (b) give an example of what happens when $R = 1$. (c) and (d) give an example of $R = 0$.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.220.

3.4.10 ページ置換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

Figure 3-21. Page replacement algorithms discussed in the text.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.221.