Memory Management

Week 07 – Lecture 2

Design Issues on Paging

Team

Instructors

Giancarlo Succi

Joseph Brown

Teaching Assistants

Vladimir Ivanov

Stanislav Litvinov

Alexey Reznik

Munir Makhmutov

Hamna Aslam

Sources

- These slides have been adapted from the original slides of the adopted book:
  - Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013
    Prentice-Hall, Inc.

  and customised for the needs of this course.
- Additional input for the slides are detailed later

Design Issues For Paging Systems

- Local versus Global Allocation Policies
- Load Control
- Page Size
- Separate Instruction and Data Spaces
- Shared Pages
- Shared Libraries
- Mapped Files
- Cleaning Policy
- Virtual Memory Interface

Local vs. Global Allocation Policies (1)

- Example: three processes, A, B, and C, make up the set of runnable processes
- Suppose A gets a page fault. The algorithm can try to find the least recently used page considering either only the pages currently allocated to A or all the pages in memory
- Local Page Replacement: it looks only at A's pages, the page with the lowest age value is A5 (Fig. 3-22b)
- Global Page Replacement: the page with the lowest age value is removed without regard to whose page it is; page B3 will be chosen (Fig. 3-22c)

Local vs. Global Allocation Policies (2)

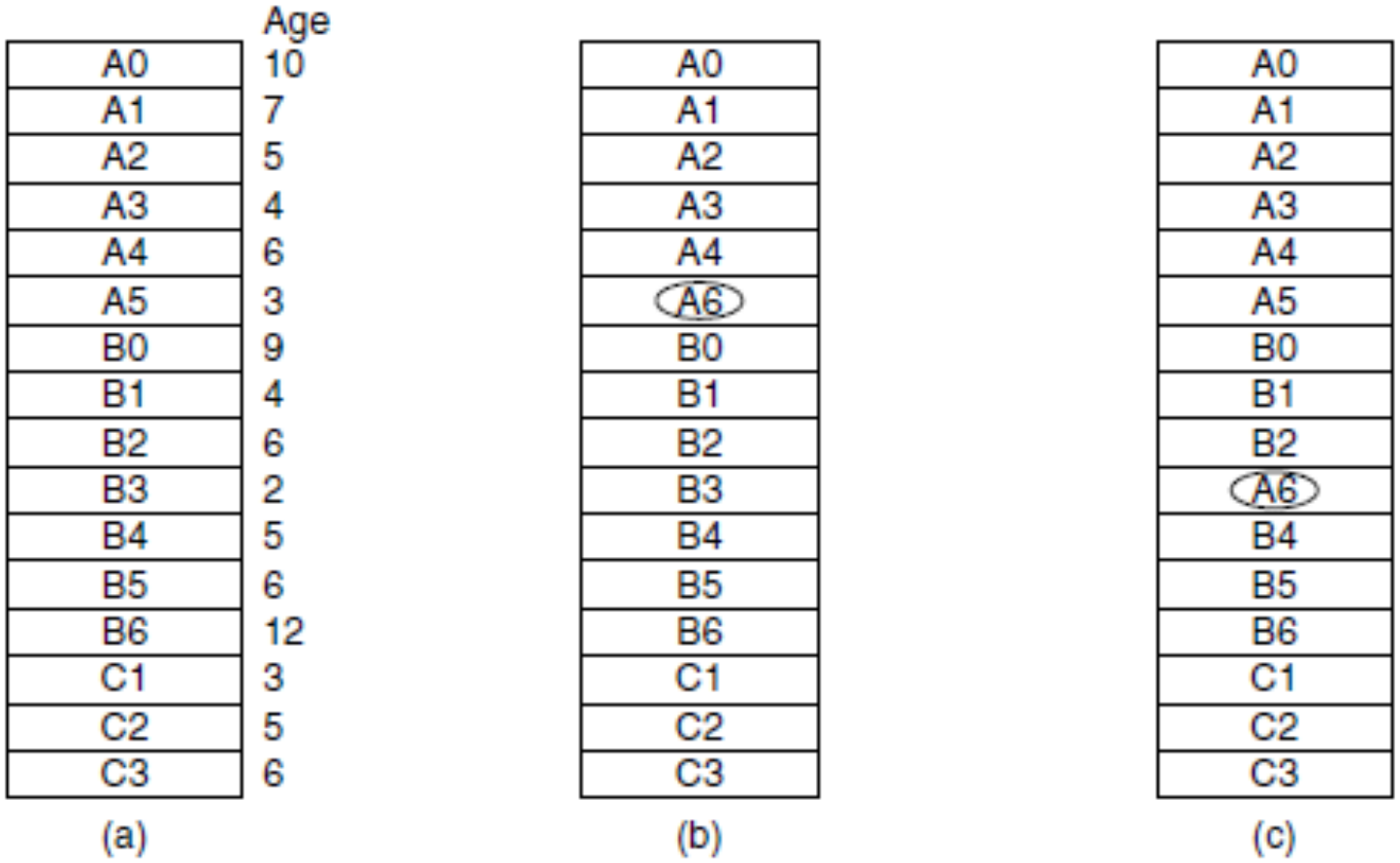

Figure 3-22. Local versus global page replacement.
(a) Original configuration. (b) Local page replacement.
(c) Global page replacement.

Local vs. Global Allocation Policies (3)

- If a local algorithm is used and the working set grows, thrashing will result. If the working set shrinks, local algorithms waste memory
- If a global algorithm is used, the system must continually decide how many page frames to assign to each process

Local vs. Global Allocation Policies (4)

- One way is to monitor the working set size as indicated by the aging bits, but this approach does not necessarily prevent thrashing
- Another approach is to have an algorithm for allocating page frames to processes:
  - to periodically determine the number of running processes and allocate each process an equal share, or
  - to allocate pages in proportion to each process' total size, with a 300-KB process getting 30 times the allotment of a 10-KB process

Local vs. Global Allocation Policies (5)

- If a global algorithm is used, the allocation has to be updated dynamically as the processes run
- One way to manage the allocation is to use the **PFF** (**Page Fault Frequency**) algorithm that tells when to increase or decrease a process' page allocation
- The assumption behind PFF is that the fault rate decreases as more pages are assigned (Fig. 3-23)

Local vs. Global Allocation Policies (6)

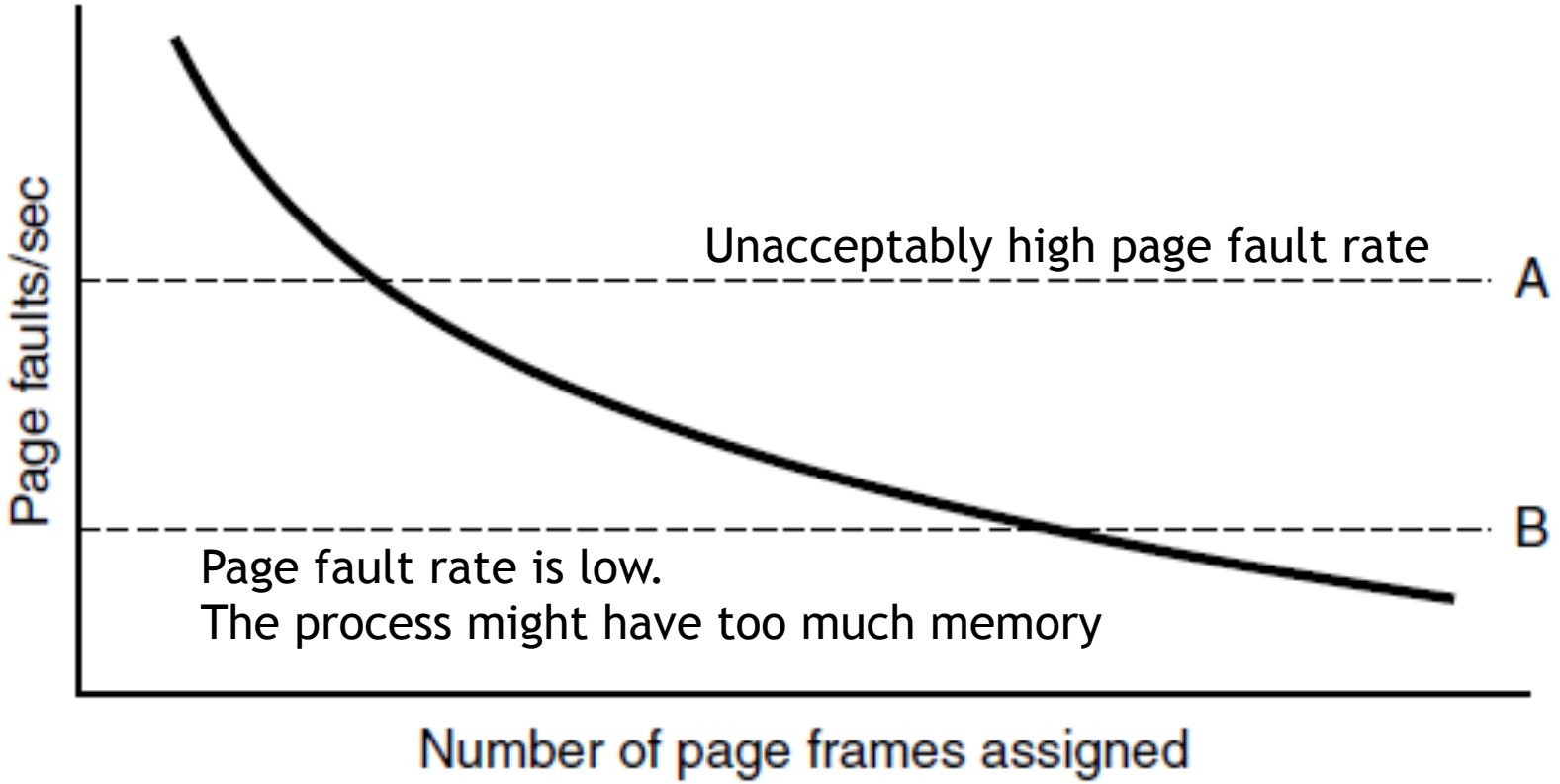

Figure 3-23. Page fault rate as a function of the number of page frames assigned.

Local vs. Global Allocation Policies (7)

- Some page replacement algorithms can work with either a local replacement policy or a global one
- For example, FIFO can replace the oldest page in all of memory or the oldest page owned by the current process
- For other page replacement algorithms, only a local strategy makes sense. In particular, **the working set** and **WSClock** algorithms refer to some specific process and must be applied in that context

Load Control (1)

- Even with the best page replacement algorithm and optimal global allocation of page frames to processes, the system might thrash
- One symptom of this situation is that the PFF algorithm indicates that some processes need more memory but no processes need less memory
- In this case, there is no way to give more memory to those processes needing it without hurting some other processes. The only real solution is to temporarily get rid of some processes

Load Control (2)

- Some of the processes may be swapped to the disk to free up all the pages they are holding
- When a process is swapped out, its page frames divided up among other processes that are thrashing
- If the thrashing does not stop, another process has to be swapped out, and so on, until the thrashing stops

Load Control (3)

- However, when the number of processes in main memory is too low, the CPU may be idle for substantial periods of time
- Thus, we need to consider not only process size and paging rate when deciding which process to swap out, but also its characteristics, such as whether it is CPU bound or I/O bound, and what characteristics the remaining processes have

Page Size (1)

- The page size is a parameter that can be chosen by the OS which, for example, can always regard a pair of pages as one page with a double size
- Determining the best page size requires balancing several competing factors and there is no overall optimum

Page Size (2)

- **Small page size. Pros:**
  - It reduces **internal fragmentation** which is a situation when text, data, or stack segment will not fill an integral number of pages, thus leaving a part of a page empty
  - It will help to allocate small programs consisting of several phases. For example, with a 32-KB page size, a program consisting of eight sequential phases of 4 KB each will take 32 KB all the time. The smaller the page size is, the less memory will be wasted

Page Size (3)

- **Small page size. Cons:**
  - Having many pages means a large page table
  - Transferring a small page takes almost as much time as transferring a large page
  - Small pages use up much valuable space in the TLB. A program that uses 1 MB of memory with a working set of 64 KB would occupy at least 16 entries in the TLB if a page size is 4 KB. With 2-MB pages, a single TLB entry would be sufficient

Page Size (4)

- **Optimal page size:**
  - Let the average process size be ***s*** bytes and the page size be ***p*** bytes
  - Assume that each page entry requires ***e*** bytes
  - The approximate number of pages needed per process is ***s/p***, occupying ***s*e/p*** bytes of page table space
  - The wasted memory due to internal fragmentation is ***p/2***

Page Size (5)

- **Optimal page size** (cont.):
  - overhead = $se/p + p/2$
  - The first term is large when the page size is small. The second term is large when the page size is large. By taking the first derivative with respect to $\boldsymbol{p}$ and equating it to zero, we get the equation
  - $-se/p^2 + 1/2 = 0$
  - $\boldsymbol{p = (2se)^{1/2}}$

Separate Instruction and Data Spaces (1)

- A single address space usually holds both programs and data (Fig. 3-24a)
- If the address space is too small, it forces programmers to invent new ways to fit everything
- One solution is to have separate address spaces for instructions (program text) and data, called **I-space** and **D-space** (Fig. 3-24b)
- Each one has its own page table, with its own mapping of virtual pages to physical page frames

Separate Instruction and Data Spaces (2)

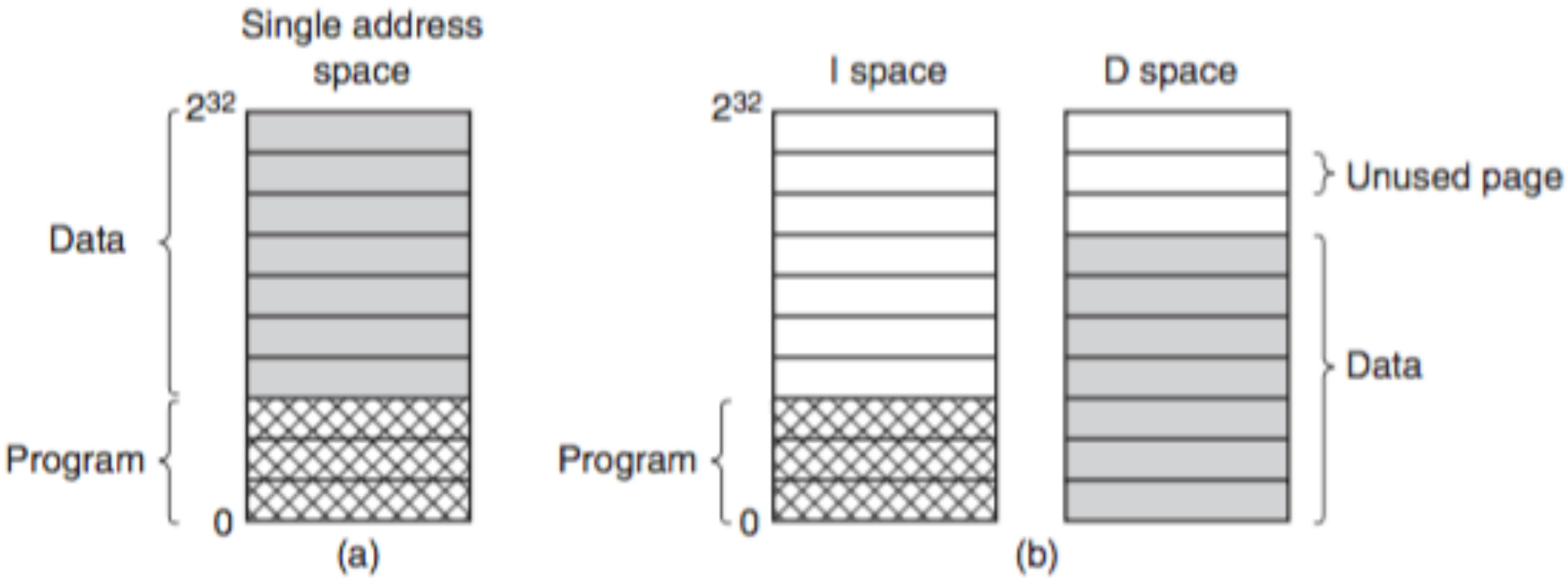

Figure 3-24. (a) One address space. (b) Separate I and D spaces.

Separate Instruction and Data Spaces (3)

- When the hardware wants to fetch an instruction, it knows that it must use I-space and the I-space page table
- Similarly, data must go through the D-space page table
- Having separate I- and D-spaces does not introduce any special complications for the OS and it does double the available address space.
- Rather than for the normal address spaces, separate I- and D-spaces are used nowadays to divide the L1 cache

Shared Pages (1)

- In a large multiprogramming system, it is common for several users to be running the same program at the same time or a single user may be running several programs that use the same library
- It is not efficient to have two copies of the same page in memory at the same time
- Not all the pages are shareable. Pages that are read-only, such as program text, can be shared, but for data pages sharing is more complicated

Shared Pages (2)

- If separate I- and D-spaces are supported, it is relatively straightforward **to share programs** by having two or more processes use the same page table for their I-space but different page tables for their D-spaces (Fig. 3-25)
- The problem: if the scheduler decides to remove process A from memory, evicting all its pages will cause process B to generate a large number of page faults to bring them back in again

Shared Pages (3)

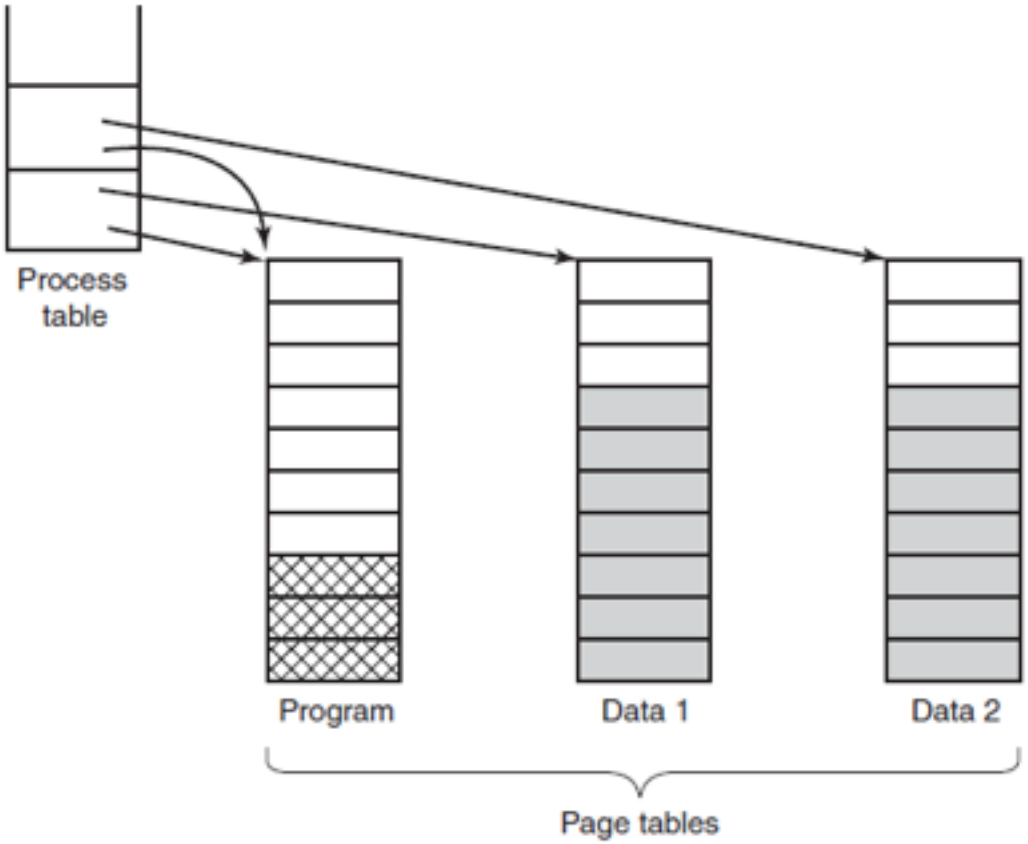

Figure 3-25. Two processes sharing the same program sharing its page table.

Shared Pages (4)

- When process A terminates, it is essential to be able to discover that the pages are still in use
- Searching all the page tables to see if a page is shared is usually too expensive, so special data structures are needed to keep track of shared pages

Shared Pages (5)

- Sharing data is trickier than sharing code, but it is not impossible:
  - In UNIX, after a *fork* system call, the parent and child are required to share both program text and data
  - No copying of pages is done at *fork* time
  - Each of the processes has **its own page table** and both of them point to the **same set of pages**

Shared Pages (6)

- Sharing data (cont.):
  - As soon as either process updates a memory word, the violation of the read-only protection causes a trap to the OS
  - A copy of the offending page is made, so that each process now has its own private copy
  - Both copies are now set to READ/WRITE, so subsequent writes to either copy proceed without trapping (**copy on write**)

Shared Libraries (1)

- In modern systems, there are many large libraries used by many processes, for example, multiple I/O and graphics libraries
- Consider traditional linking first:
  - When a program is linked, all the .o (object) files in the current directory are linked and libraries that are named in the command to the linker are scanned
  - Any functions called in the object files but not present there (e.g., printf) are called **undefined externals** and are sought in the libraries

Shared Libraries (2)

- Traditional linking (cont.):
  - Any functions called in the object files are included in the executable binary if they are found
  - Functions present in the libraries but not called are not included
  - When the linker is done, an executable binary file is written to the disk containing all the functions needed

Shared Libraries (3)

- Common programs use 20–50 MB worth of graphics and user interface functions
- Statically linking hundreds of programs with all these libraries would result in wasting a huge amount of space on the disk and space in RAM
- A common technique helping solve this problem is to use **shared libraries** (or **Dynamic Link Libraries** on Windows)

Shared Libraries (3)

- When a program is linked with shared libraries, the linker includes a **small stub routine** that binds to the called function at run time
- Shared libraries may be loaded either when the program is loaded or when functions in them are called for the first time
- The entire library is paged in into memory, page by page, as needed, so functions that are not called will not be brought into RAM

Shared Libraries (4)

- Another advantage is that it is possible to update a DLL without recompilation of the programs that call it
- However, there is one problem (Fig. 3-26):
  - When two different processes share the same library, it may be located at a different address in each process
  - Since the library is shared, relocation on the fly will not work

Shared Libraries (5)

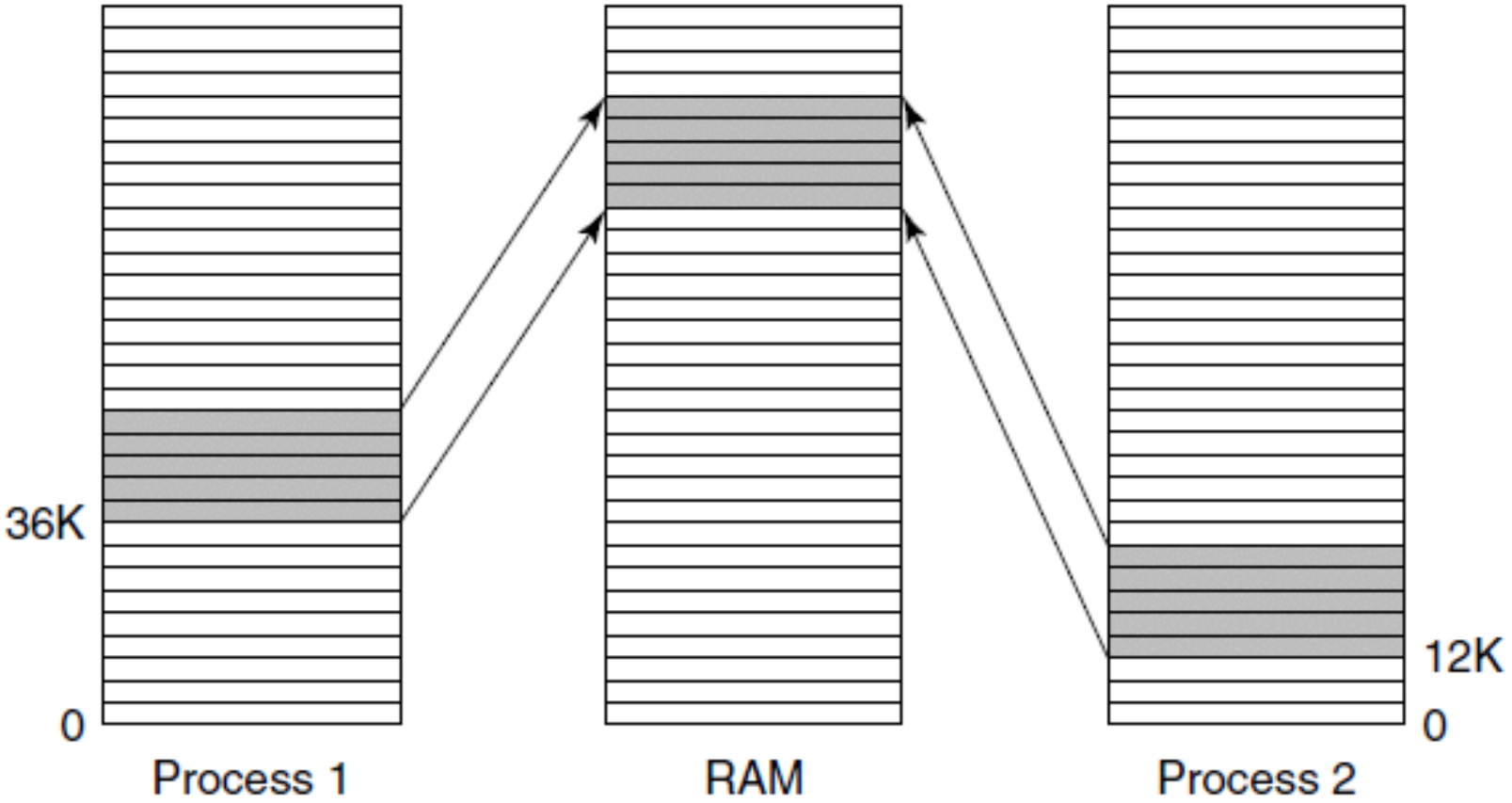

Figure 3-26. A shared library being used by two processes.

Shared Libraries (6)

- When the first function is called by process 2 (at address 12K), the jump instruction has to go to 12K+16, not 36K+16
- One way to solve the problem is to use copy on write and create new pages for each process sharing the library, but this scheme defeats the purpose of sharing the library
- A better solution is to compile shared libraries with a special compiler flag telling the compiler not to produce any instructions that use absolute addresses
- Code that uses only relative offsets is called **position-independent code**

Mapped Files (1)

- Shared libraries are a special case of a more general facility called **memory-mapped files**
- The idea is that a process can issue a system call to map a file onto a portion of its virtual address space
- As pages are touched, they are demand paged in one page at a time, using the disk file as the backing store
- When the process exits, or explicitly unmaps the file, all the modified pages are written back to the file on disk

Mapped Files (2)

- If two or more processes map onto the same file at the same time, they can communicate over shared memory
- Writes done by one process to the shared memory are immediately visible when the other one reads from the part of its virtual address spaced mapped onto the file
- If memory-mapped files are available, shared libraries can use this mechanism

Cleaning Policy (1)

- Paging works best when there is a large amount of free page frames available that can be claimed as page faults occur
- **Paging daemon** is a background process that sleeps most of the time, but is awakened periodically to inspect the state of memory
- If too few page frames are free, it begins selecting pages to evict using some page replacement algorithm

Cleaning Policy (2)

- The previous contents of the page are remembered
- In the event one of the evicted pages is needed again before its frame has been overwritten, it can be reclaimed by removing it from the pool of free page frames
- The paging daemon ensures that all the free frames are clean, so they need not be written to disk in a big hurry when they are required

Cleaning Policy (3)

- One way to implement this cleaning policy is with a two-handed clock:
  - The front hand is controlled by the paging daemon
  - When it points to a dirty page, that page is written back to disk and the front hand is advanced
  - When it points to a clean page, it is just advanced
  - The back hand is used for page replacement, as in the standard clock algorithm
  - Thus, the probability of the back hand hitting a clean page is increased due to the work of the paging daemon

Virtual Memory Interface (1)

- In some advanced systems, programmers have some control over the memory map and can use it in nontraditional ways to enhance program behavior
- If programmers can **name regions** of their memory, it may be possible for one process to give another process the name of a memory region so that process can also map it in

Virtual Memory Interface (2)

- Sharing of pages can also be used to implement a high-performance message-passing system:
  - Normally, when messages are passed, the data are copied from one address space to another
  - If processes can control their page map, a message can be passed by having the sending process unmap the page(s) containing the message
  - The receiving process will map them in
  - Only the page names have to be copied, instead of all the data

Virtual Memory Interface (3)

- Another advanced memory management technique is called **distributed shared memory:**
  - Multiple processes share a set of pages over a network, possibly, but not necessarily, as a single shared linear address space
  - When a process references a page that is not currently mapped in, it gets a page fault
  - The page fault handler locates the machine holding the page and sends it a message asking it to unmap the page and send it over the network
  - When the page arrives, it is mapped in and the faulting instruction is restarted

End

Week 07 – Lecture 2

References

- Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013 Prentice-Hall, Inc.