Modern Operating Systems

Chapter 3 – Design & Implementation Issues for Paging

Zhang Yang

Spring 2022

Content of the Lecture

- 3.5 Design Issues for Paging Systems
- 3.6 Implementation Issues for Paging Systems

Design Issues for Paging System

- Resident Set Management
  - Replacement Scope
  - Resident Set Size
- Page Fault Frequency (PFF)
- Load Control
- Page Size
- Separate Instruction and Data Space
- Shared Pages
- Shared Libraries
- Cleaning Policy

Resident Set Management (1)

- The OS must decide how many frames to allocate to each process.
  - Allocating small number permits many processes in memory at once.
  - Too few frames results in high page fault rate.
  - Too many frames/process gives low multiprogramming level.
  - Beyond some minimum level, adding more frames has little effect on page fault rate.

Resident Set Management (2)

- Resident Set
  - Set of a process' pages which are in main memory.
- Resident Set Management
  - Replacement Scope
    - Local Replacement
    - Global Replacement
  - Resident Set Size
    - Fixed Allocation
    - Variable Allocation

Replacement Scope (1)

- The set of frames to be considered for replacement when a page fault occurs.
- Local Replacement Policy
  - Choose only among frames allocated to the process that faulted.
- Global Replacement Policy
  - Any unlocked frame in memory is candidate for replacement.
  - High priority process might be able to select frames among its own frames or those of lower priority processes

Replacement Scope (2)

- Example
  - Processes A, B, and C make up the set of runnable processes. Suppose A gets a page fault.

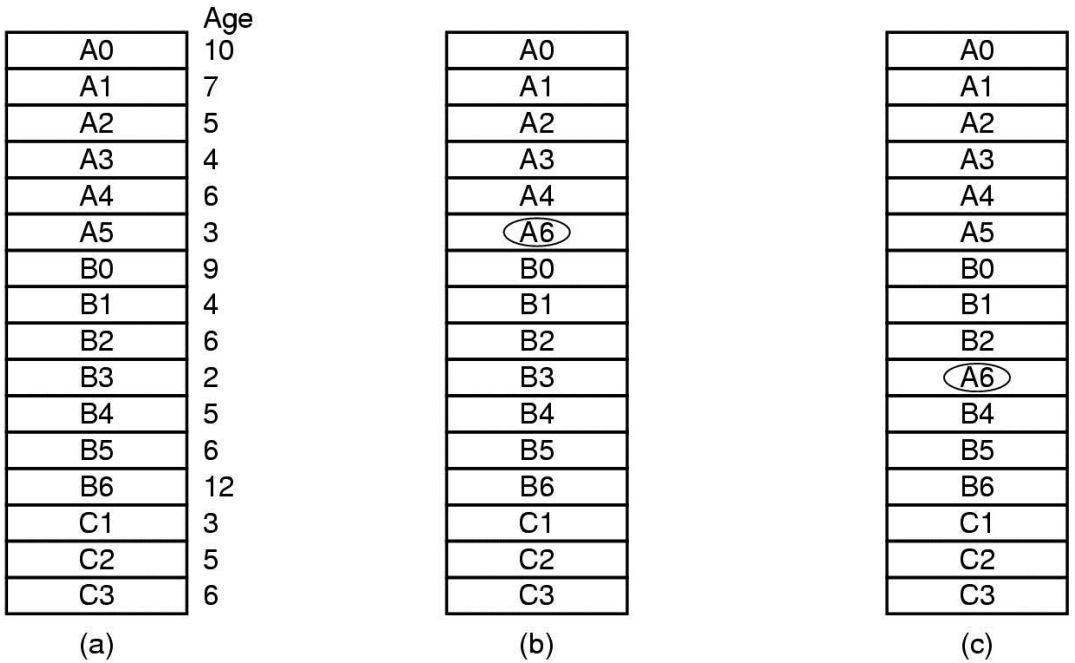


Figure 3-22 Local versus global page replacement
(a) Original configuration
(b) Local page replacement
(c) Global page replacement

Replacement Scope (3)

- Comparison
  - Local Replacement
    - Might slow a process unnecessarily as less used memory not available for replacement.
  - Global Replacement
    - Process can’t control own page fault rate – depends also on paging behavior of other processes (erratic execution times).
    - Generally results in greater throughput.
    - More common method.

Resident Set Size (1)

- Fixed Allocation Policy
  - Allocates a fixed number of frames to a process at load time by some criteria:
    - Equal allocation
      - E.g., with 12416 available page frames and 10 processes, each process gets 1241 frames, the remaining 6 go into a pool to be used when page faults occurs.
    - Proportional allocation
      - E.g. a 300-KB process gets 30 times the allotment of a 10-KB process.
    - Priority allocation
      - Use a proportional allocation scheme using process priorities rather than size.

Resident Set Size (2)

- Variable Allocation Policy
  - The number of frames for a process may vary over time.
    - Increase if page fault rate is high.
    - Decrease if page fault rate is very low.
  - Requires OS overhead to assess behavior of active processes.

Allocation vs Scope

	Local Replacement	**Global Replacement**
Fixed Allocation	• Number of frames per process is fixed • Replacement is chosen from process' frames	• Not Possible
Variable Allocation	• Number of frames per process will change to maintain the working set • Replacement is chosen from process' frames	• Replacement is chosen from all frames of all processes • This may cause the size of the resident set of a process (which hasn't caused the page fault) to vary

Fixed Allocation, Local Scope

- Frame number per process is decided beforehand and can't be changed.
  - Determined at load time and depends on application type.
- Problem: difficult to determine ahead of time a good number for the allocated frames.
  - Too Small: High page faults, poor performance.
  - Too Large: Small number of processes, high processor idle time and/or swapping.
- Example: P223 Figure 3-22 (b)

Variable Allocation, Global Scope

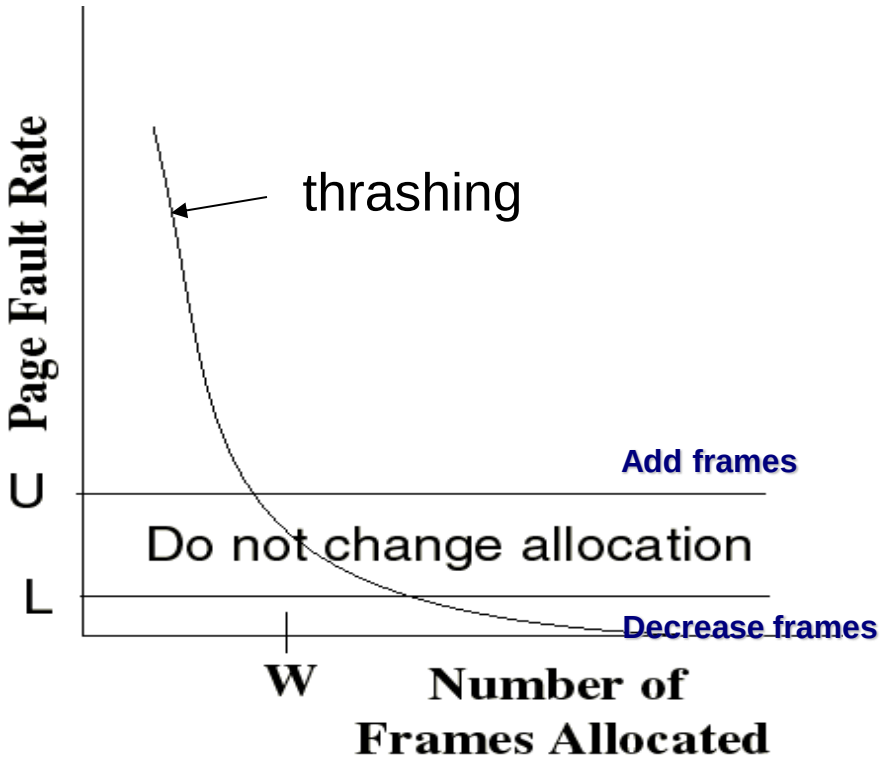
- Easiest to implement
  - Adopted by many OS (like Unix SVR4).
- When a page fault occurs, a free frame is added to the resident set of the process who experiences a page fault and the page is brought up in that frame.
- Harder to determine who should lose a page; the selection is made among all the frames in the memory (except the locked ones), there is no discipline to determine which process should lose a page from its resident set.
  - May remove a page from a process that needs it.
- Example: P223 Figure 3-22 (c)

Variable Allocation, Local Scope

- May be the best combination (used by Windows NT).
- Allocate at load time a certain number of frames to a new process based on application type.
  - Use either pre-paging or demand paging to fill up the allocation.
- When a page fault occurs, select the page to replace from the resident set of the process that suffers the fault.
- Example: working set model

Page Fault Frequency Algorithm (PFF)

- Define an upper bound U and lower bound L for page fault rates.
- Allocate more frames to a process if fault rate is higher than U.
- Allocate fewer frames if fault rate is < L.
- The resident set size should be close to the working set size W.
- Suspend a process if the PFF > U and no more free frames are available.



Load Control (1)

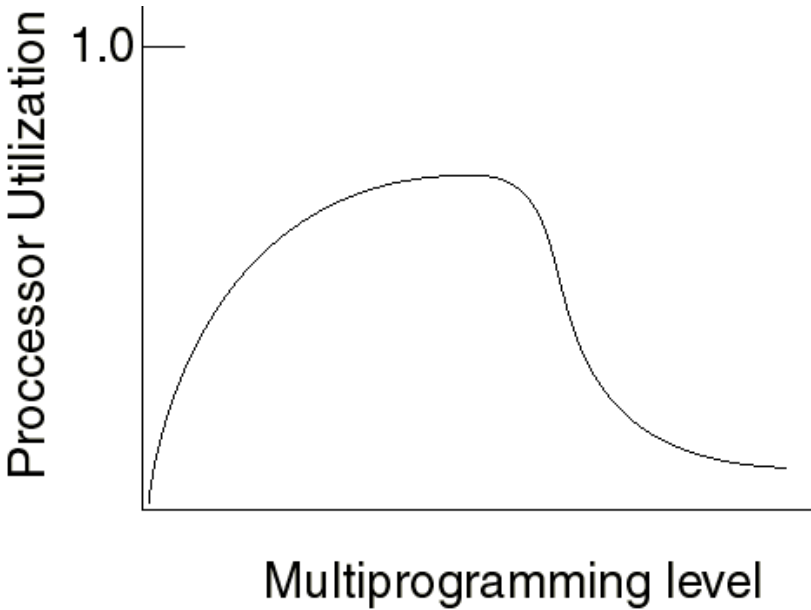
- Load Control
  - How many processes may be started to run and reside in main memory simultaneously.
- Despite good designs, system may still thrash.
  - Whenever the combined working sets of all processes exceed the capacity of memory, thrashing can be expected.
  - When PFF algorithm indicates
    - Some processes need more memory.
    - But no processes need less.

Load Control (2)

- Solution: Reduce number of processes competing for memory.
  - Swap one or more to disk, divide up pages they held.
  - Note
    - When deciding which process to swap out, consider not only process size and paging rate, but also its characteristics and what characteristics the remaining processes have.

Load Control (3)

- Reconsider degree of multiprogramming--Determines the number of processes that will be resident in main memory (ie: the multiprogramming level)
  - Too few processes: often all processes will be blocked and the processor will be idle.
  - Too many processes: the resident size of each process will be too small and flurries of page faults will result thrashing.



Page Size (1)

- How to select a page size?
  - Tradeoffs
    - Memory utilization-better with smaller pages because of internal fragmentation.
    - Capturing locality-better with smaller pages because of better resolution.
    - Size of the page table-to keep it relatively small, we need bigger pages
    - I/O time-latency/throughput characteristics call for bigger pages

Page Size (2)

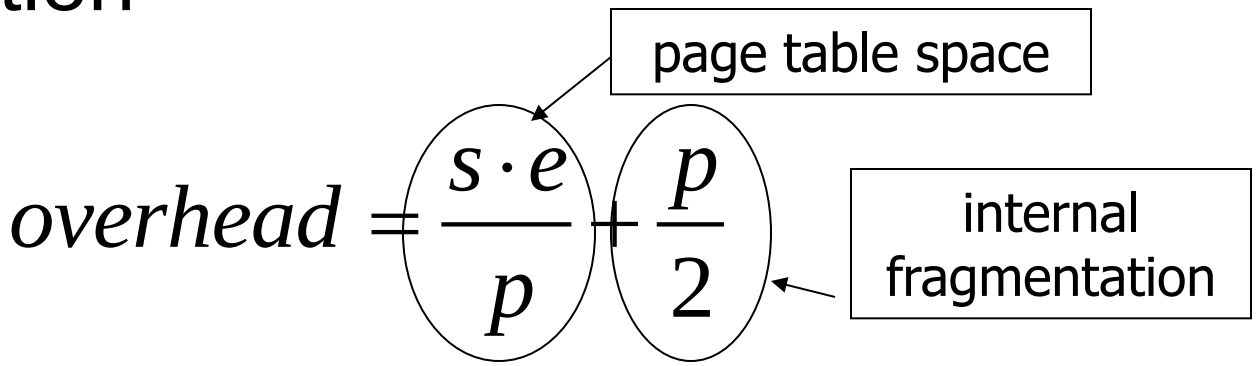
- Small Page Size
  - Advantages
    - Less internal fragmentation
    - Better with locality of reference
  - Disadvantages
    - Larger page table
    - More page faults
    - More overhead in reading/writing of pages

Page Size (3)

- Big Page Size
  - Advantages
    - Smaller page table
    - Less page faults
    - Less overhead in reading/writing of pages
  - Disadvantages
    - More internal fragmentation
    - Worse locality of reference

Page Size (4)

- Overhead due to page table and internal fragmentation



  - Where
    - s = average process size in bytes
    - p = page size in bytes
    - e = page entry size in bytes

Optimized when

$$p = \sqrt{2se}$$

s=1MB, e=8, p=4KB

Page Size (5)

- Example Page Size

Computer	Page Size
Atlas	512 48-bit words
Honeywell-Multics	1024 36-bit words
IBM 370/XA and 370/ESA	4 Kbytes
VAX family	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 Kbytes to 16 Mbytes
UltraSPARC	8 Kbytes to 4 Mbytes
Pentium	4 Kbytes or 4 Mbytes
IBM POWER	4 Kbytes
Itanium	4 Kbytes to 256 Mbytes

Separate Instruction and Data Spaces (1)

- Most computers have a single address space.
- What if it's too small?

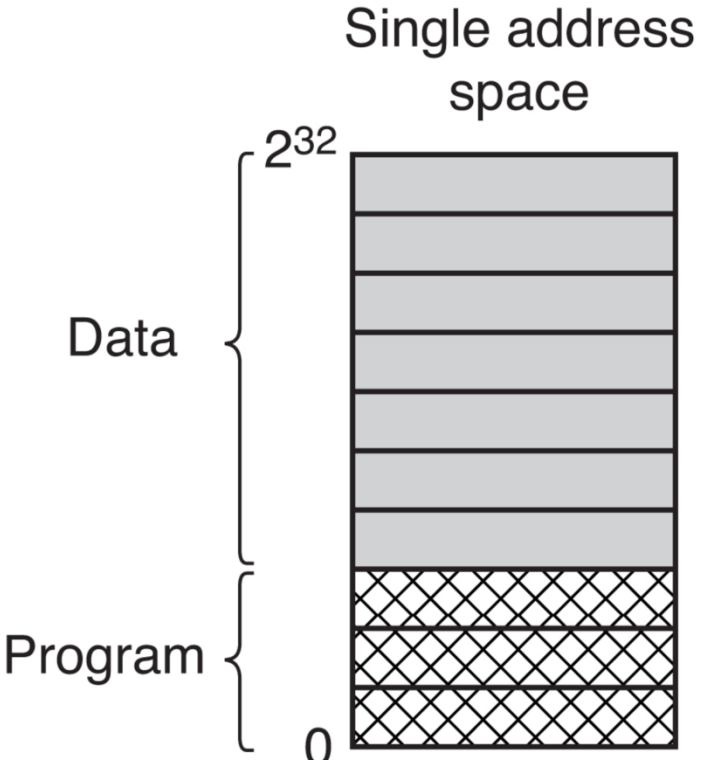


Figure 3-24 (a) One address space

Separate Instruction and Data Spaces (2

- Solution in 16-bit PDP-11 Machines
  - Separate data and program address spaces.
  - Two independent spaces, two paging systems.
  - The linker must know about the two address spaces.

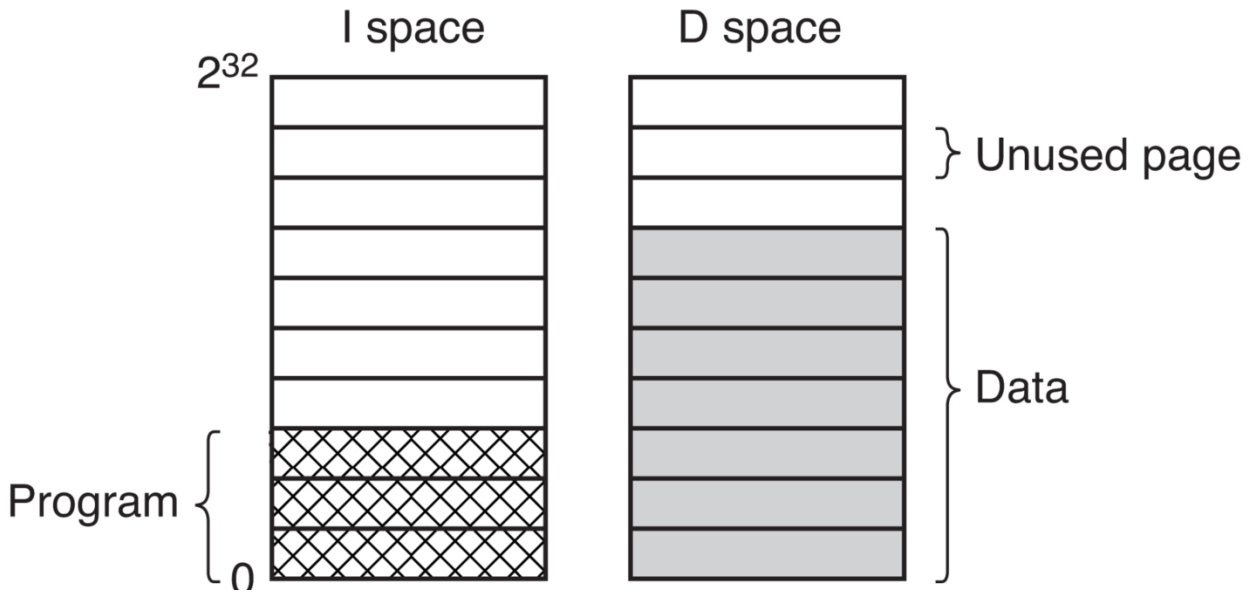


Figure 3-24 (b) Separate I and D spaces

Shared Pages (1)

- Sharing code: Pages that are read-only can be shared.
- Separate instruction and data space, easy to share code.
  - Example: Two processes share the same program by using the same I-space page table and different D-space page table.

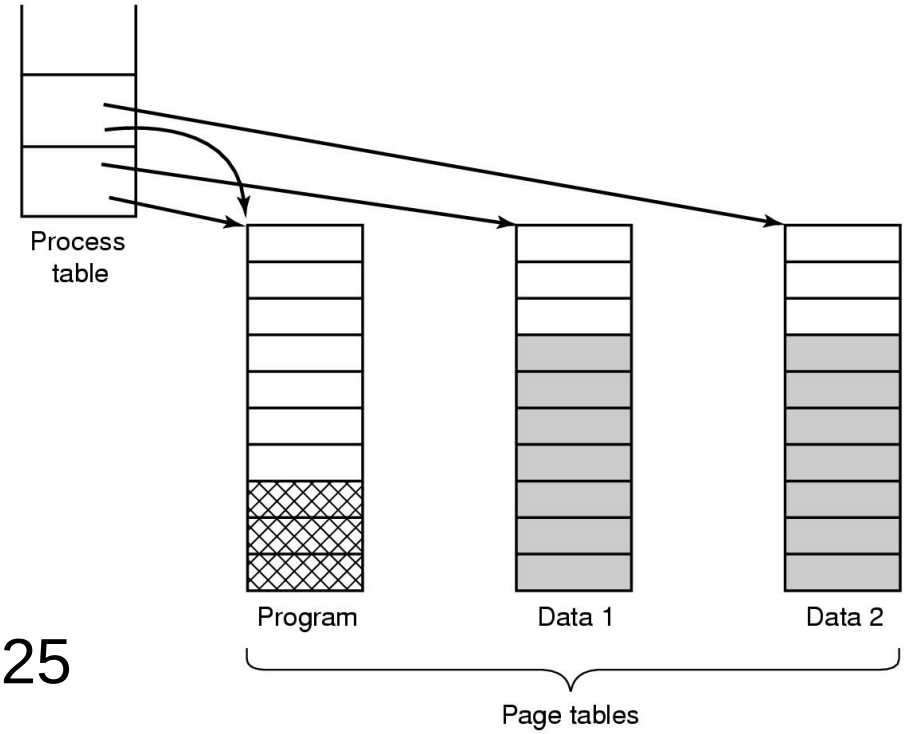
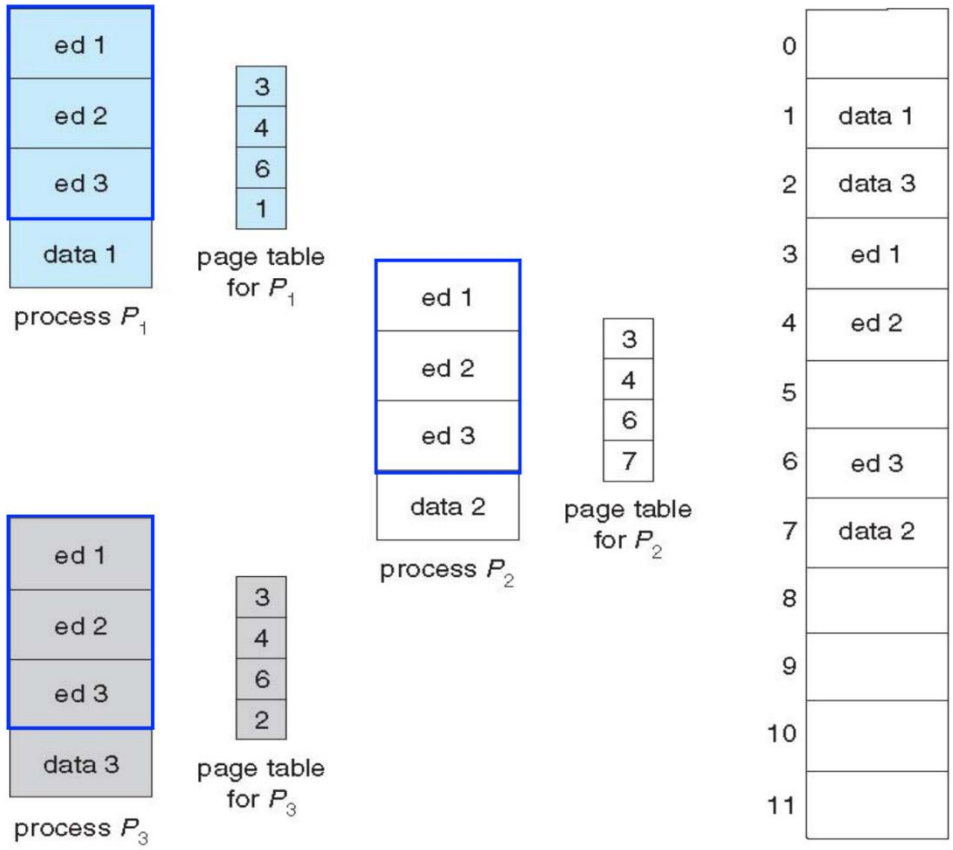


Figure 3-25

Shared Pages (2)

- Unite instruction and data space, complex to share code.
  - Example: a text editor
  - What if scheduler removes a process from memory?
  - What if a process terminates?



Shared Pages (3)

- Sharing Data
  - In UNIX, after a fork system call, the parent and child are required to share both program text and data.
  - No copying pages is done at *fork* time.
    - Give the parent and the child process its own page table.
    - Have both processes point to the same set of pages.
    - Shared data pages must be read only.
  - Question: What if either process updates a memory word?

Shared Pages (4)

- Sharing Data (ctd.)
  - Copy-on-Write (COW)
    - If a process writes to a frame
      - Trap to kernel.
      - Kernel allocates new frame.
      - Copy the old frame.
      - Restart the write instruction.

Shared Pages (5)

- Sharing Data (ctd.)
  - Copy-on-Write Example

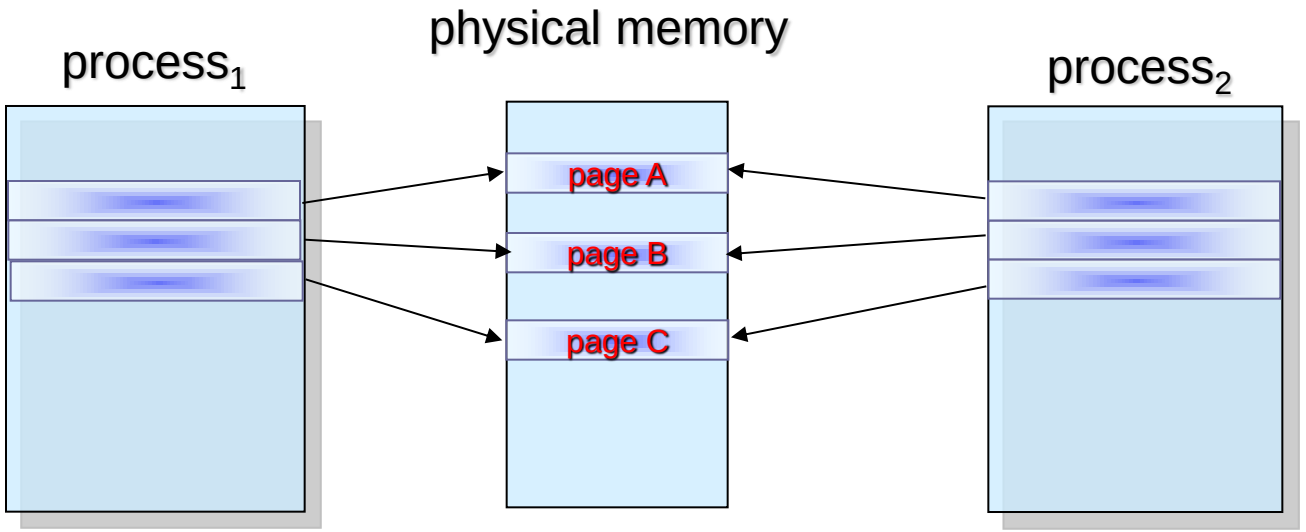


Fig. Before $process_1$ modifies page C.

Shared Pages (6)

- Sharing Data (ctd.)
  - Copy-on-Write Example (ctd.)

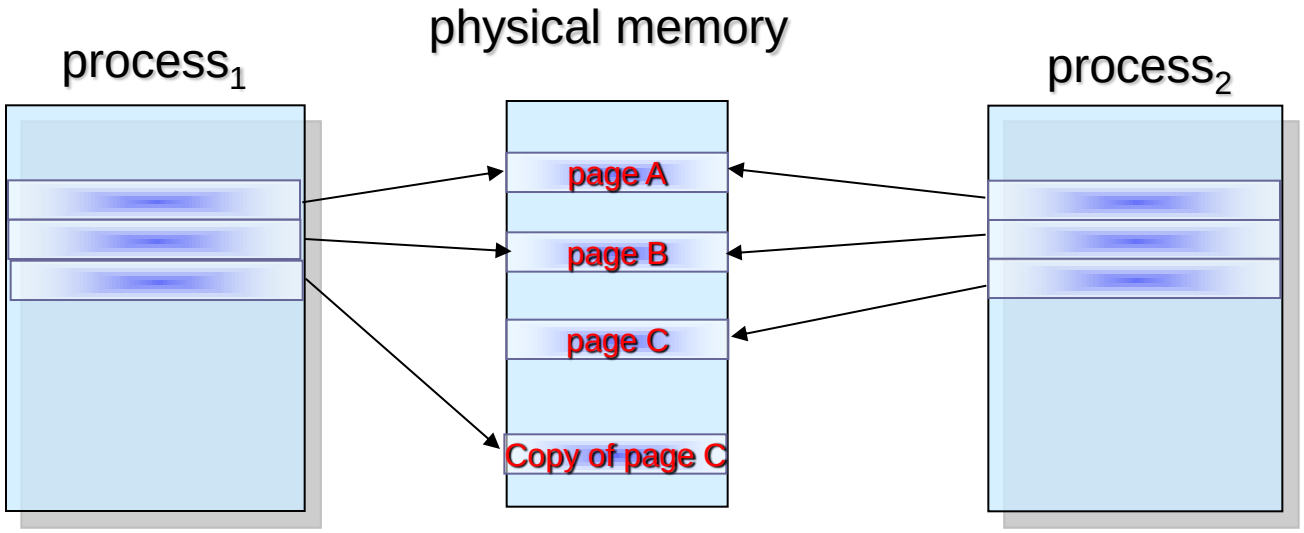


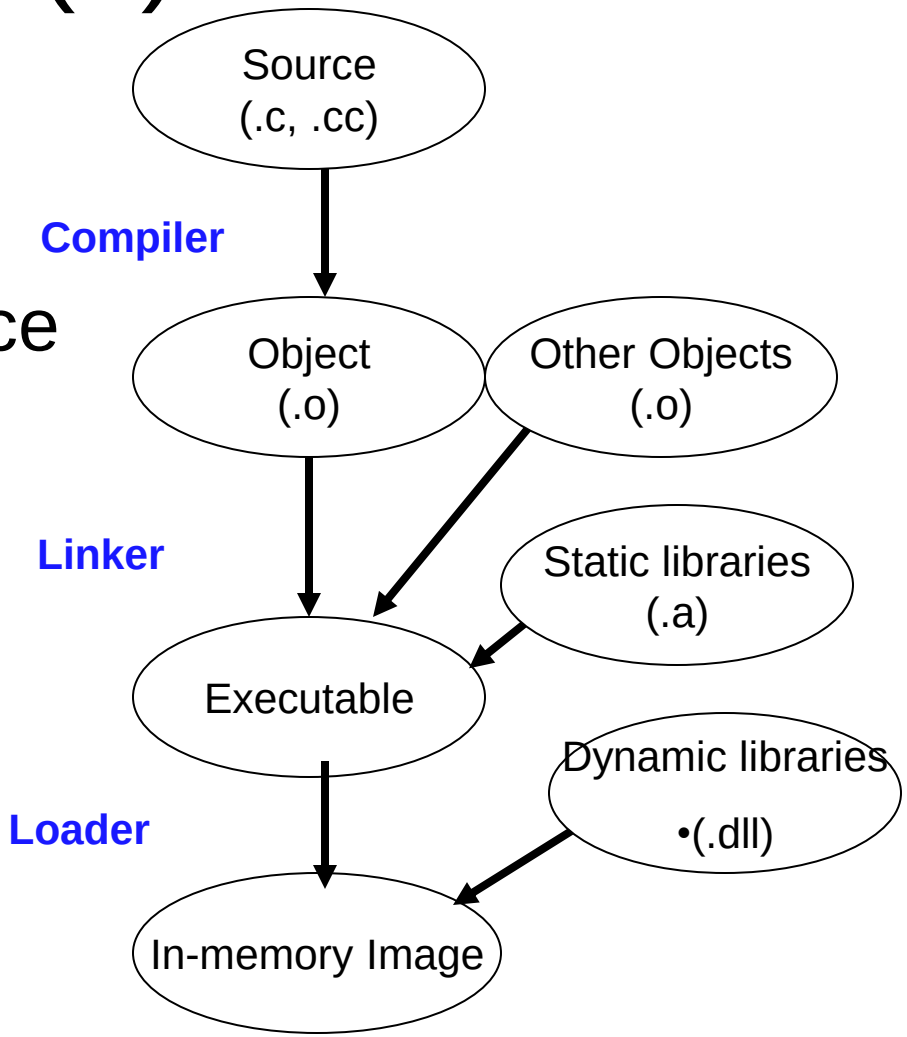
Fig. After $process_1$ modifies page C.

Shared Pages (7)

- Question
  - Can a page be in two working sets at the same time? Explain.
- Solution
  - If pages can be shared, yes. For example, if two users of a timesharing system are running the same editor at the same time, and the program text is shared rather than copied, some of those pages may be in each user's working set at the same time.

Shared Libraries (1)

- From Source to Execution
  - Most compilers produce relocatable object code.
  - The linker combines multiple object files and library modules into a single executable file.
  - The Loader reads the executable file.



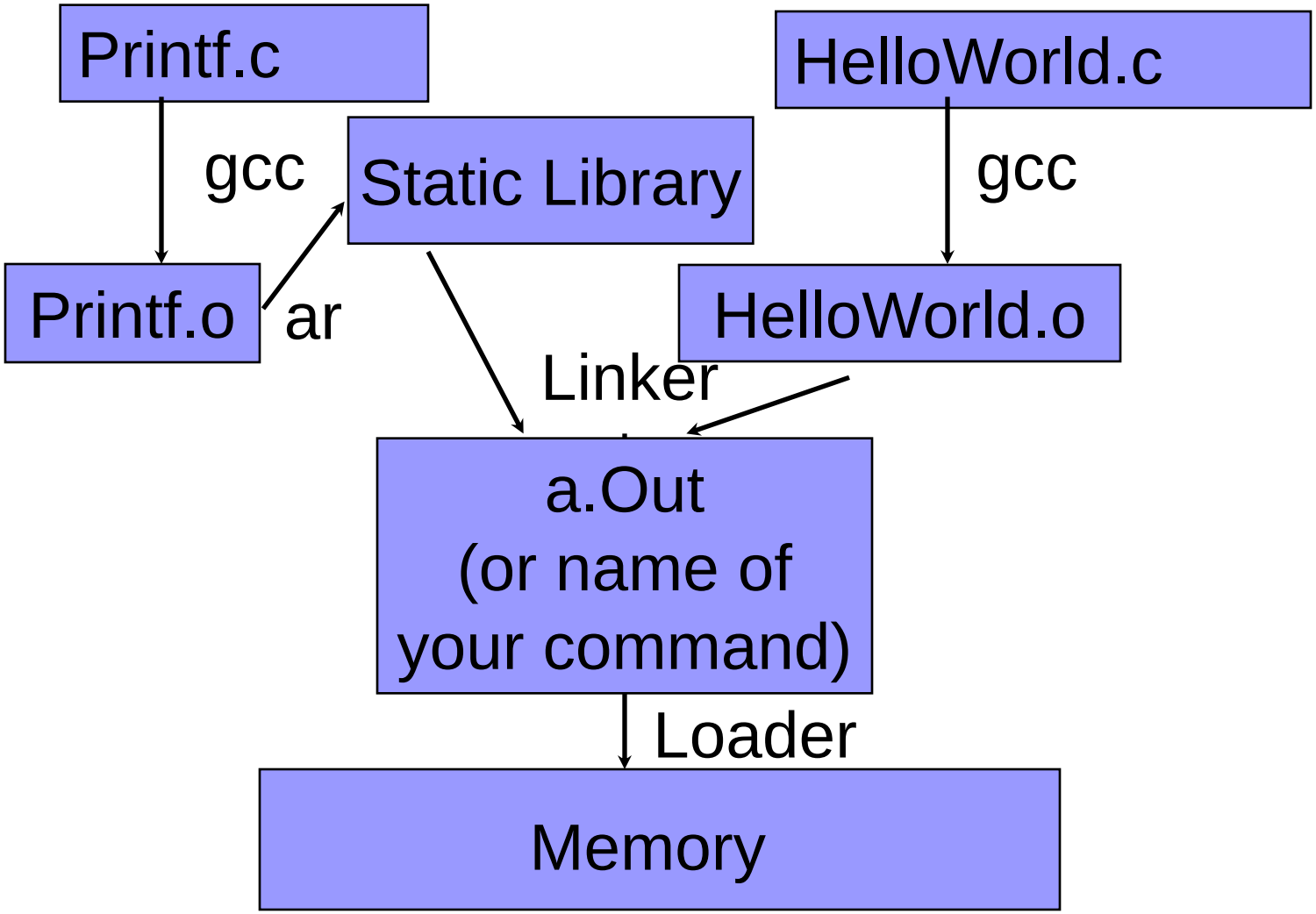
From Source to Execution

Shared Libraries (2)

- Static Linking
  - Static linking is the process of copying all relocatable object files and library modules used in the program into the final executable image.
  - Carried out only once to produce an executable file.
  - Static linking is performed by programs called linkers.
  - If static libraries are called, the linker will copy all the modules referenced by the program to the executable.

Shared Libraries (3)

- Static Linking (ctd.)



Shared Libraries (4)

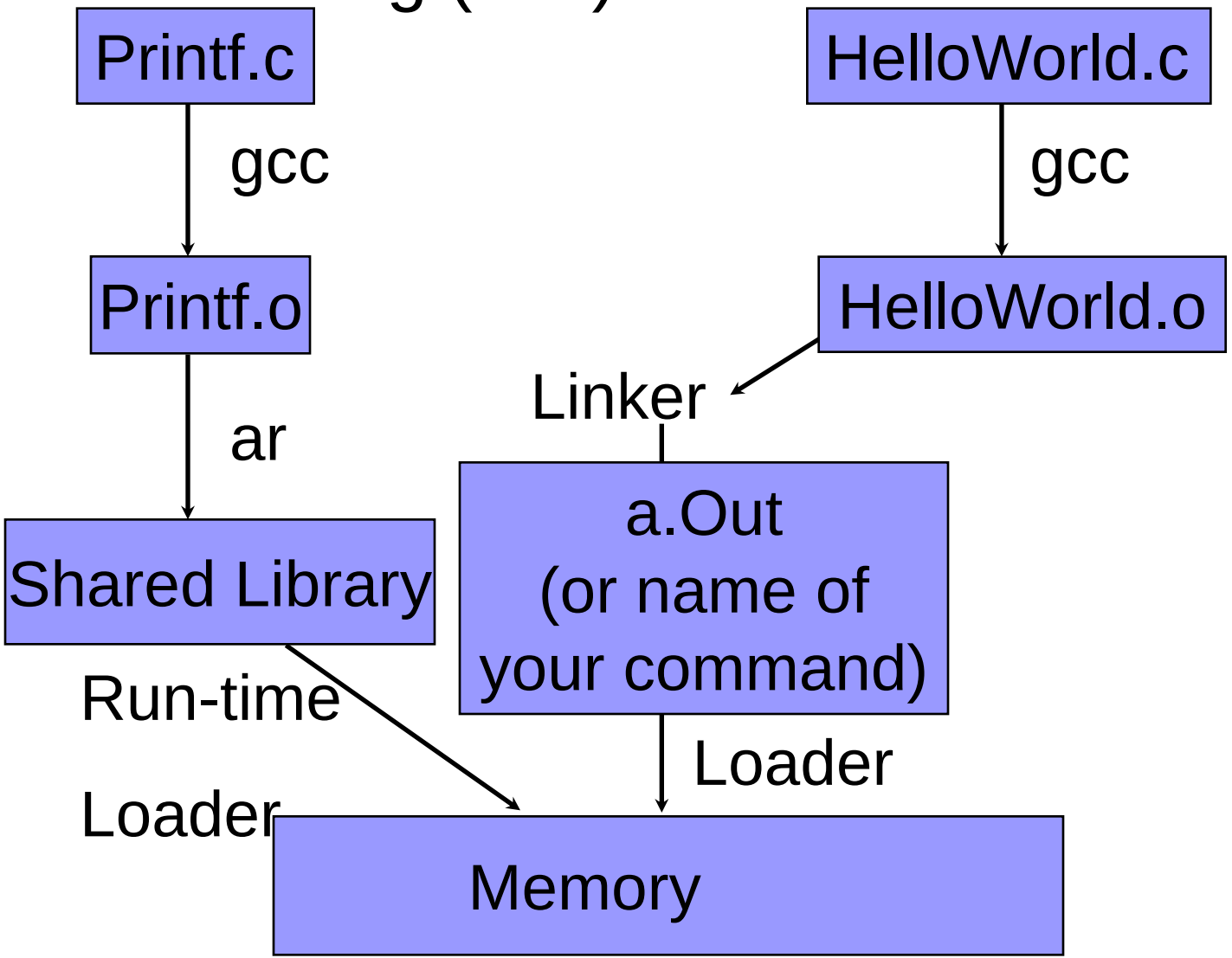
- Static Libraries
  - Consist of routines that are compiled and linked directly into your program.
  - Windows: .lib; linux: .a
  - Advantage
    - Easy to use; no dependency problem after compilation.
  - Disadvantages
    - The executable size will be larger.
    - Require re-linking when libraries changed.

Shared Libraries (5)

- Dynamic Linking
  - Complete linking postponed until running time.
  - Program only includes a stub for each library routine. When stub is first executed, the routine is loaded into memory, and the stub is replaced with the address of the routine.
  - If shared libraries are called
    - Only copy a little reference information when the executable file is created.
    - Completing the linking during loading time or running time.

Shared Libraries (6)

- Dynamic Linking (ctd.)



Shared Libraries (7)

- Shared Libraries
  - Consist of routines that are loaded into your application at run time.
  - Allow multiple processes to share the same code segment.
  - Windows: .dll; linux: .so
  - Advantages
    - Making executable files smaller and saving space in memory.
    - If a function in a shared library is updated to remove a bug, it is not necessary to recompile the programs that call it.

Shared Libraries (8)

- Shared Libraries (ctd.)
  - Disadvantages
    - Slow application at start time.
    - Dependent on the libraries when execution.

Shared Libraries (9)

- Problem in Shared Libraries
  - A 20KB shared library located at a different address in each process. Suppose that the first function in the library has to jump to address 16 in the library.

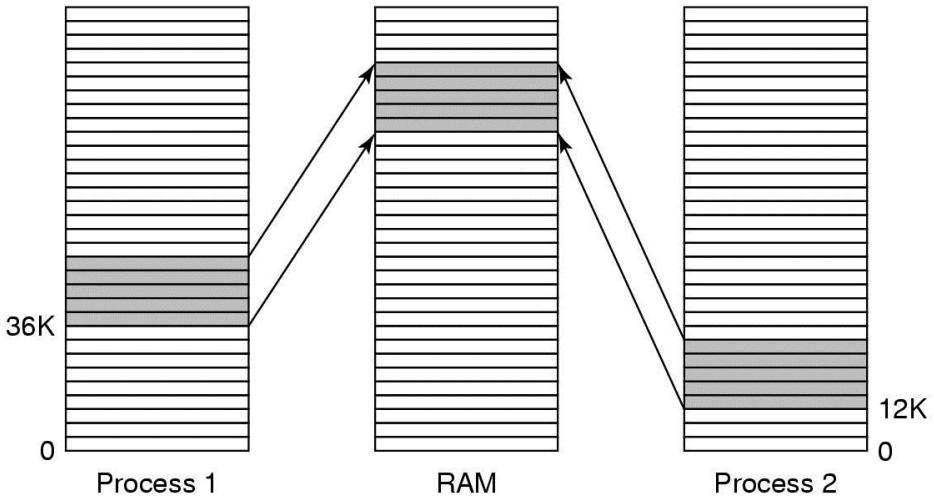


Figure 3-26. A shared library being used by two processes.

Shared Libraries (10)

- Problem in Shared Libraries (ctd.)
  - Solution
    - Copy-on-Write? Defeats the purpose of sharing the library.
    - Compile shared libraries with a special compiler flag telling the compiler not to produce any instructions that use absolute addresses.
      - Jump forward/backward by n bytes
      - Code must be position independent
      - At runtime, references are resolved as

      Library_relative_address + library_base_address

Cleaning Policy

- Cleaning Policy
  - Deciding when a modified page should be written out to secondary memory.
- Demand Cleaning
  - Write page out only when its frame has been selected for replacement (the process that page faulted has to wait for 2 page transfers).
- Precleaning
  - Modified pages are written in batches before their frames are needed.
    - But some of these pages get modified a second time before their frames are needed.

Implementation Issues for Paging

- Operating System Involvement with Paging
- Page Fault Handling
- Instruction Backup
- Locking Pages in Memory
- Backing Store

Operating System Involvement with Paging (1)

- Four times when OS is involved with paging
  - Process Creation Time
    - Determine program and data size
    - Create page table
    - Allocate memory space for page table & initialize
    - Allocate space in disk swap area & initialize
    - Record info about page table and disk swap area in process table
  - Process Execution Time
    - Reset the MMU for new process
    - Flush the TLB
    - Copy page table of new process

Operating System Involvement with Paging (2)

- Four times when OS is involved with paging (ctd.)
  - Page Fault Time
    - Determine virtual address causing fault, compute the needed page, locate the page on the disk.
    - Swap target page out, needed page in.
    - Back up PC, restart faulting instruction.
  - Process Termination Time
    - Release page table
    - Return pages to the free pool

Page Fault Handling (1)

- Sequence of Events on a Page Fault
  - 1. The hardware traps to the kernel, saving the PC on the stack. Save some info about the instruction in special CPU registers.
  - 2. Save the general registers and other volatile info.
  - 3. OS determines which virtual page needed
  - 4. OS checks validity of virtual address and the protection consistent with the access
    - No, kill or send a signal to the process
    - Yes, seek free page frame or run page rep algorithm

Page Fault Handling (2)

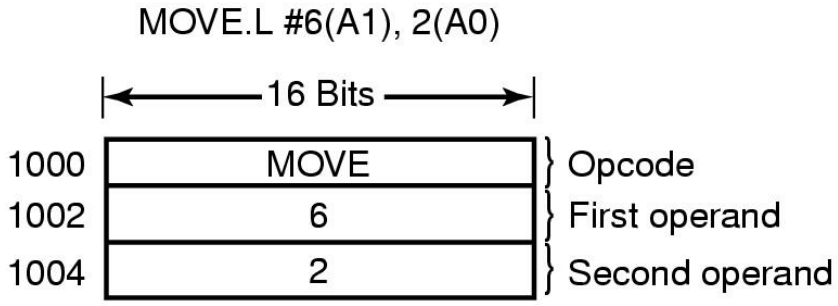
- Sequence of Events on a Page Fault (ctd.)
  - 5. If selected frame is dirty, write it to disk. Suspend the process, let another process run until the disk transfer completed.
  - 6. OS brings/schedules new page in from disk. Suspend the process and let another available process run.
  - 7. Transfer completed, disk interrupt. Page tables updated.
  - 8. Faulting instruction backed up to when it began.
  - 9. Faulting process scheduled.
  - 10. Registers restored, program continues.

Instruction Backup (1)

- Problem: page fault happens in the middle of instruction execution
  - Stopped part way through, traps to OS for fault handling, OS returns to instruction.
  - Must re-start instruction.
  - Easier said than done!

Instruction Backup (2)

- Example
  - Consider a CPU that has instructions with two addresses.
  - MOV.L #6(A1), 2(A0) (instruction length=6 bytes)
  - Page fault may occur in 1000, 1002, or 1004



  - Where to re-start the instruction? PC depends on which part of the instruction actually faulted. If it faults at 1002, how does OS know that instruction starts at 1000?.

Instruction Backup (3)

- Worse yet: Auto-incrementing loads registers either before or after instruction is executed. Do it before and needs to be un-done. Do it afterwards and must not do it.
- Hardware Solution to Instruction Backup
  - Copy the PC to a hidden internal register before each instruction is executed.
  - A second register telling which registers have already been incremented or decremented, and by how much.
- OS uses info provided by hardware to undo all the effects of the faulting instruction.

Locking Pages in Memory (1)

- Virtual memory and I/O occasionally interact.
- P1 issues call for read from device into buffer.
  - While it's waiting for I/O, P2 runs.
  - P2 has a page fault.
  - P1's I/O buffer might be chosen to be paged out.
  - This can create a problem because an I/O device is going to write to the buffer on P1's behalf.

Locking Pages in Memory (2)

- Solution1: Allow some pages to be locked into memory.
  - Locked pages are immune from being replaced.
  - Pages only stay locked for (relatively) short periods.
- Solution2: Do all I/O to kernel buffers, and then copy the data to user pages later.

Backing Store (1)

- Pages removed from memory are stored on disk.
- Where are they placed?
  - A swap area is designated where chunks of it are allocated to the processes when started.
    - The swap area can be initialized with an entire copy of the process image. Any pages can be brought in as needed. (Static partition)
    - Or load the entire process in memory and let it be pages out as needed. (Dynamic partition)

Backing Store (2)

- Static Partition: Allocate fixed partition to process when it starts up.
  - Manage as list of free chunks. Assign big enough chunk to hold process.
  - Starting address of partition kept in process table. Page offset in virtual address space corresponds to address on disk.
  - Can assign different areas for data, text , stack as data and stack can grow.

Backing Store (3)

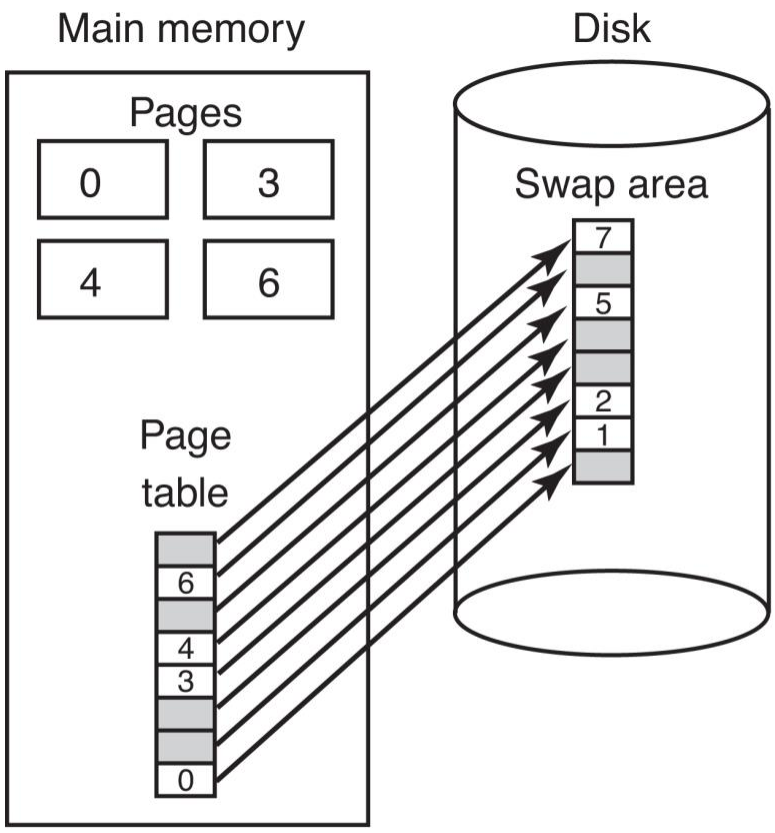


Figure 3-28 (a)Paging to a static swap area

Backing Store (4)

- Dynamic Partition
  - Don’t allocate disk space in advance. Swap pages in and out as needed.
  - Need disk map in memory.

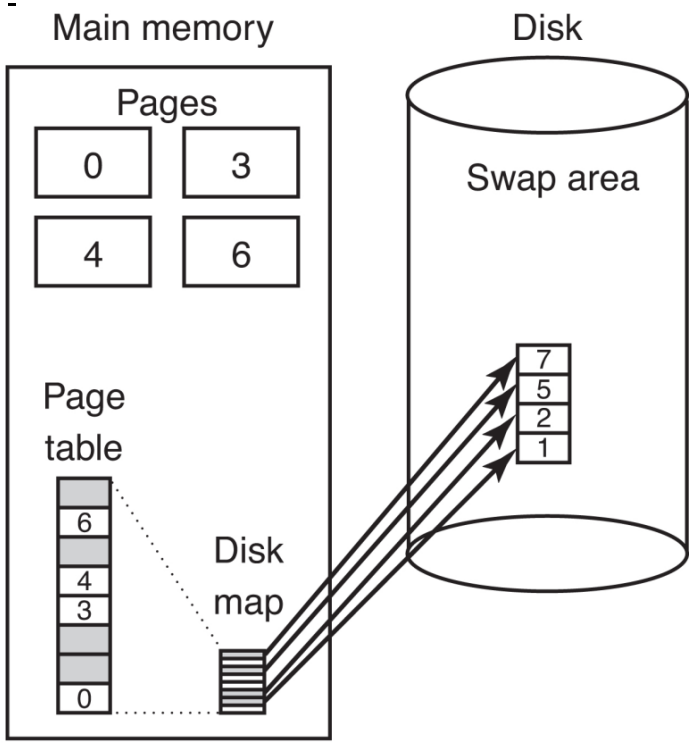


Figure 3-28 (b)Backing up pages dynamically (b)