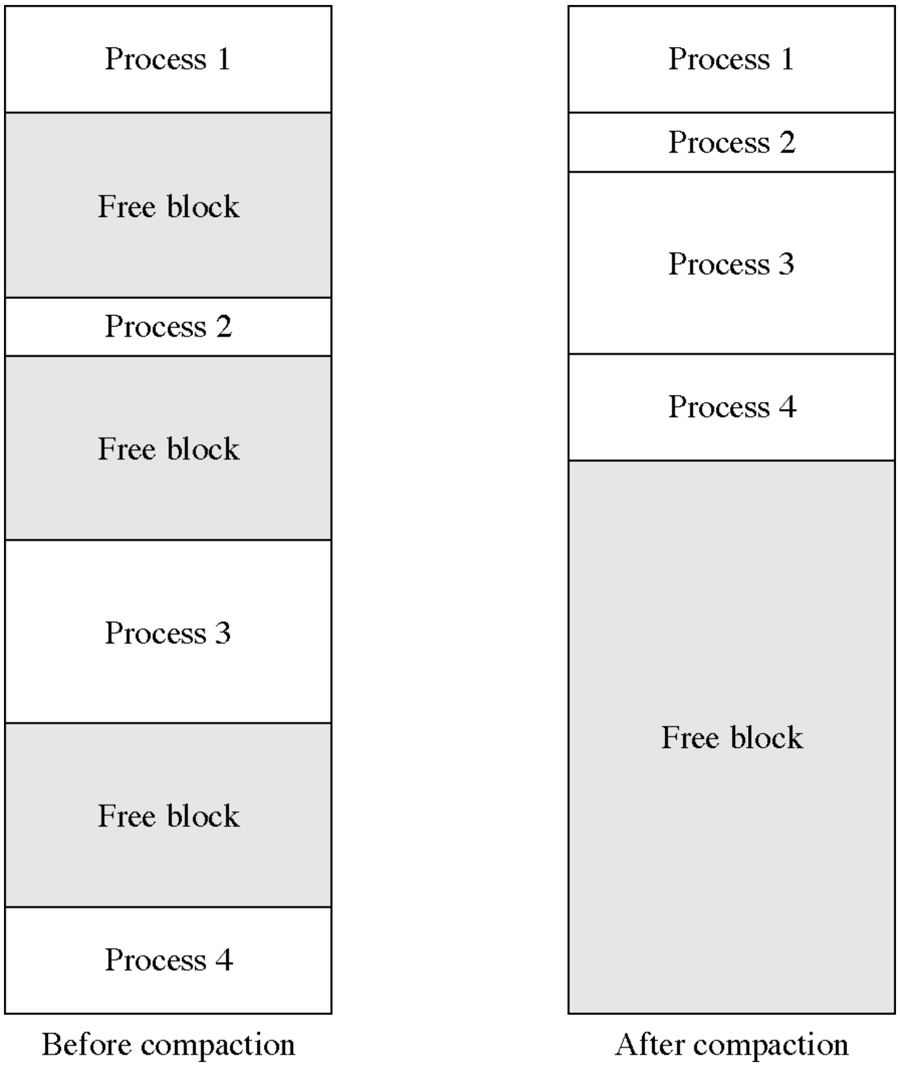
Virtual Memory

Chapter 11

Key concepts in chapter 11

- Fragmentation
- Virtual memory
- Paging
- File mapping

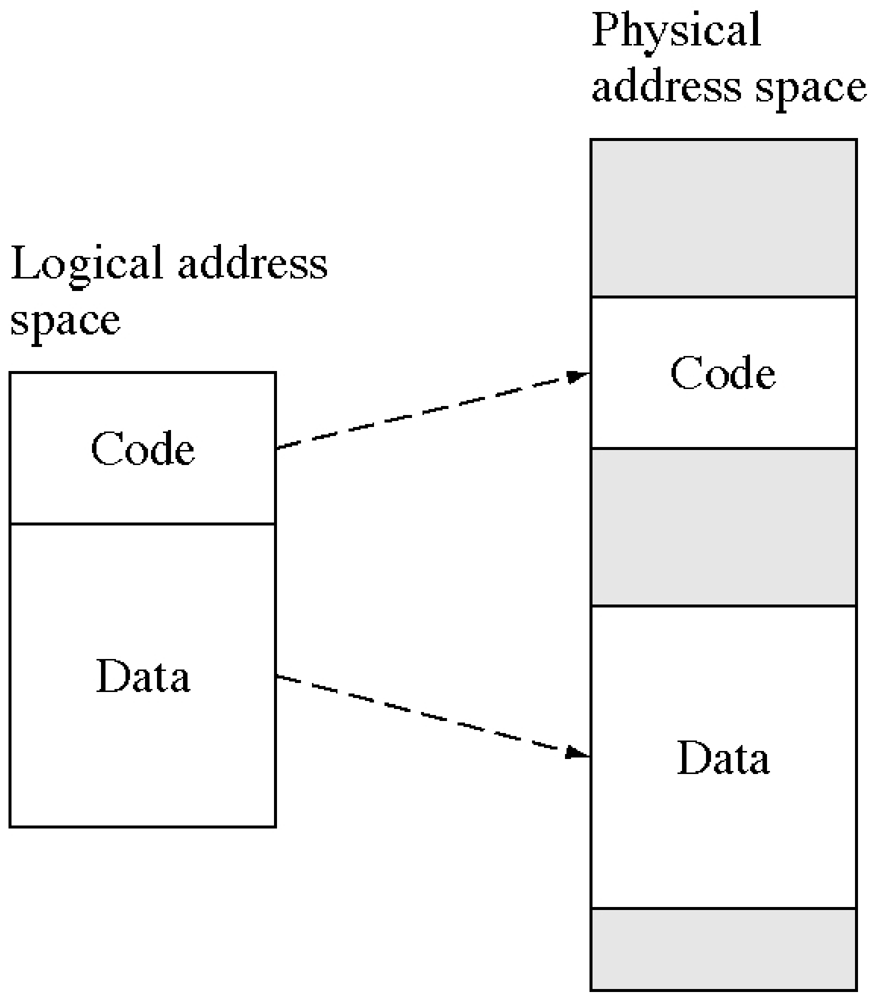
Compacting memory



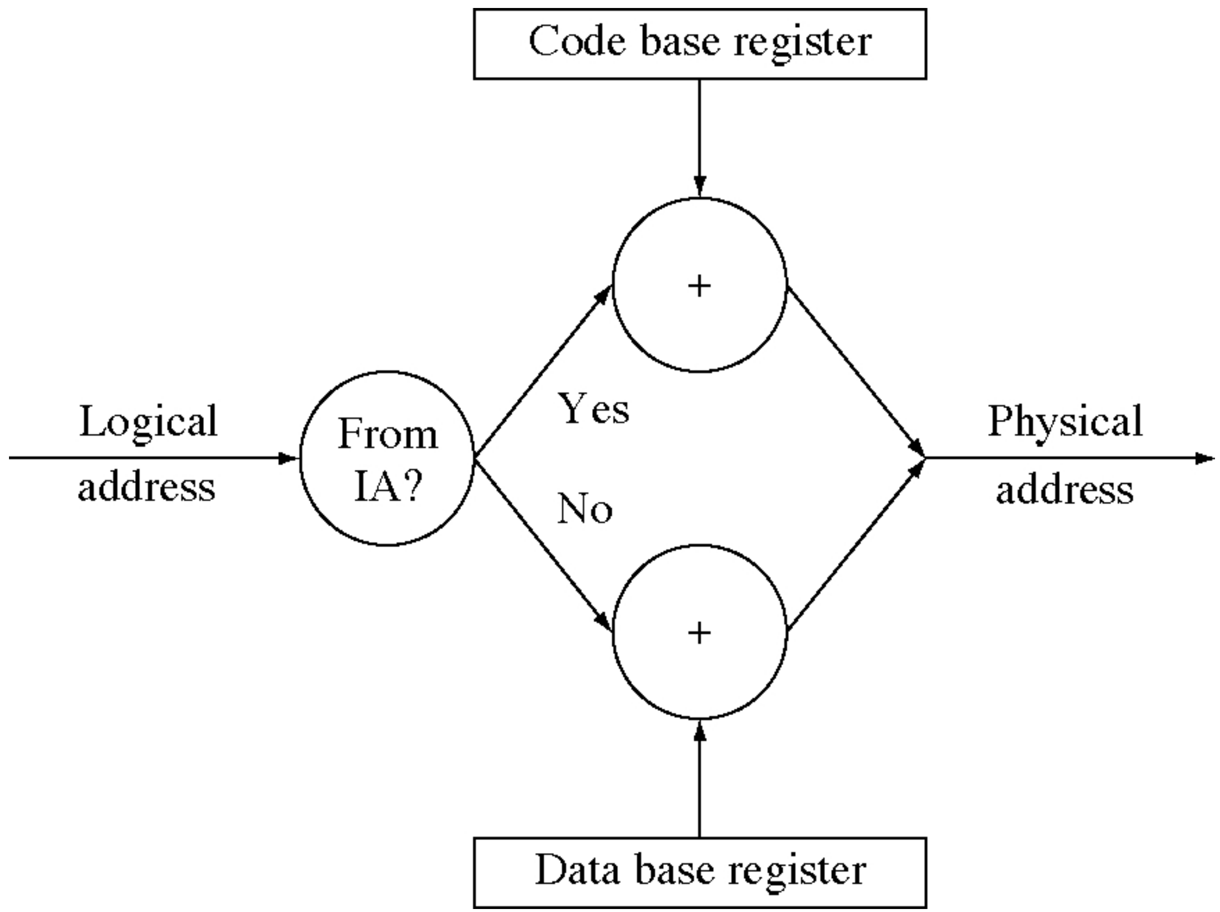
Fragmentation

- Without memory mapping, programs require physically continuous memory
- Large blocks mean large fragments
  - and wasted memory
- We need hardware memory mapping to address this problem
  - segments
  - pages
- We will look at a series of potential solutions

Separate code and data spaces



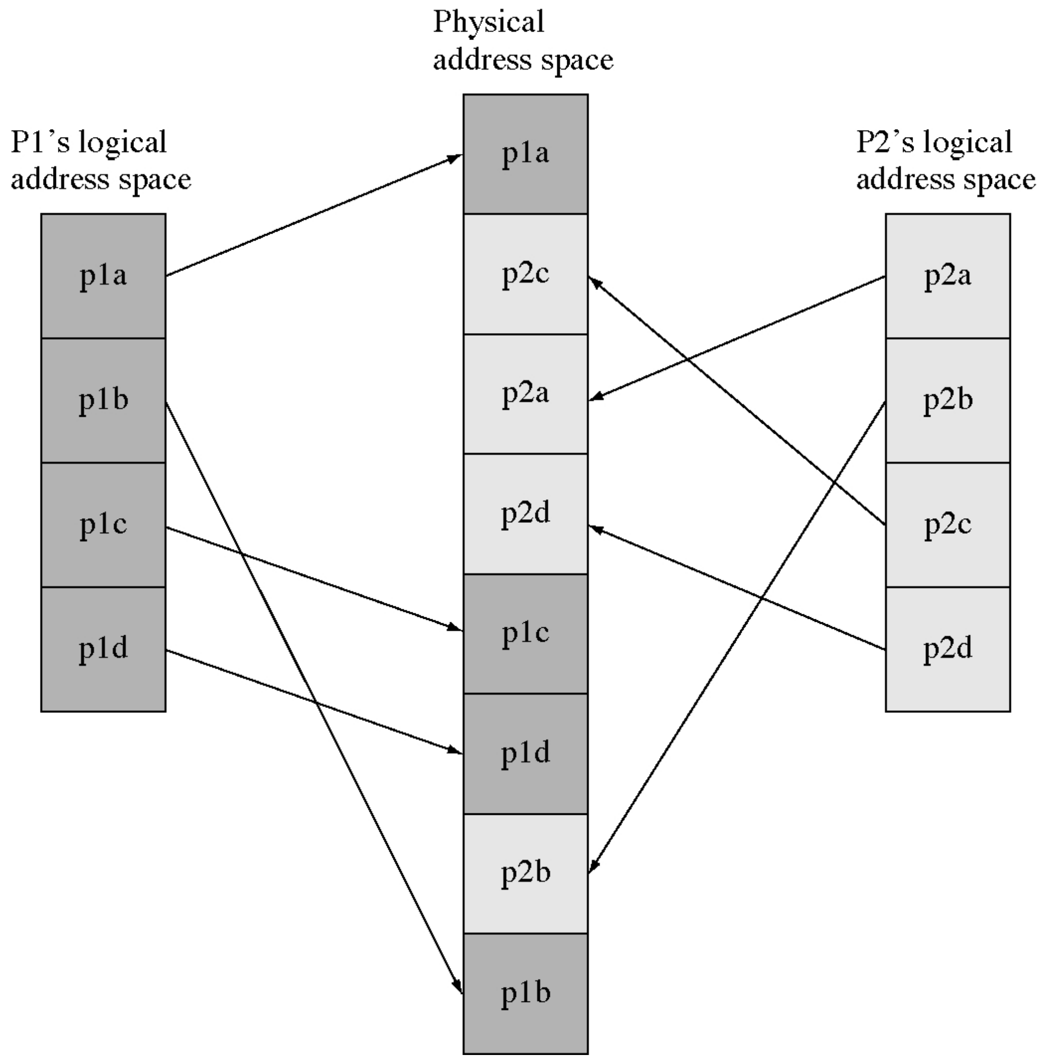
Code/data memory relocation



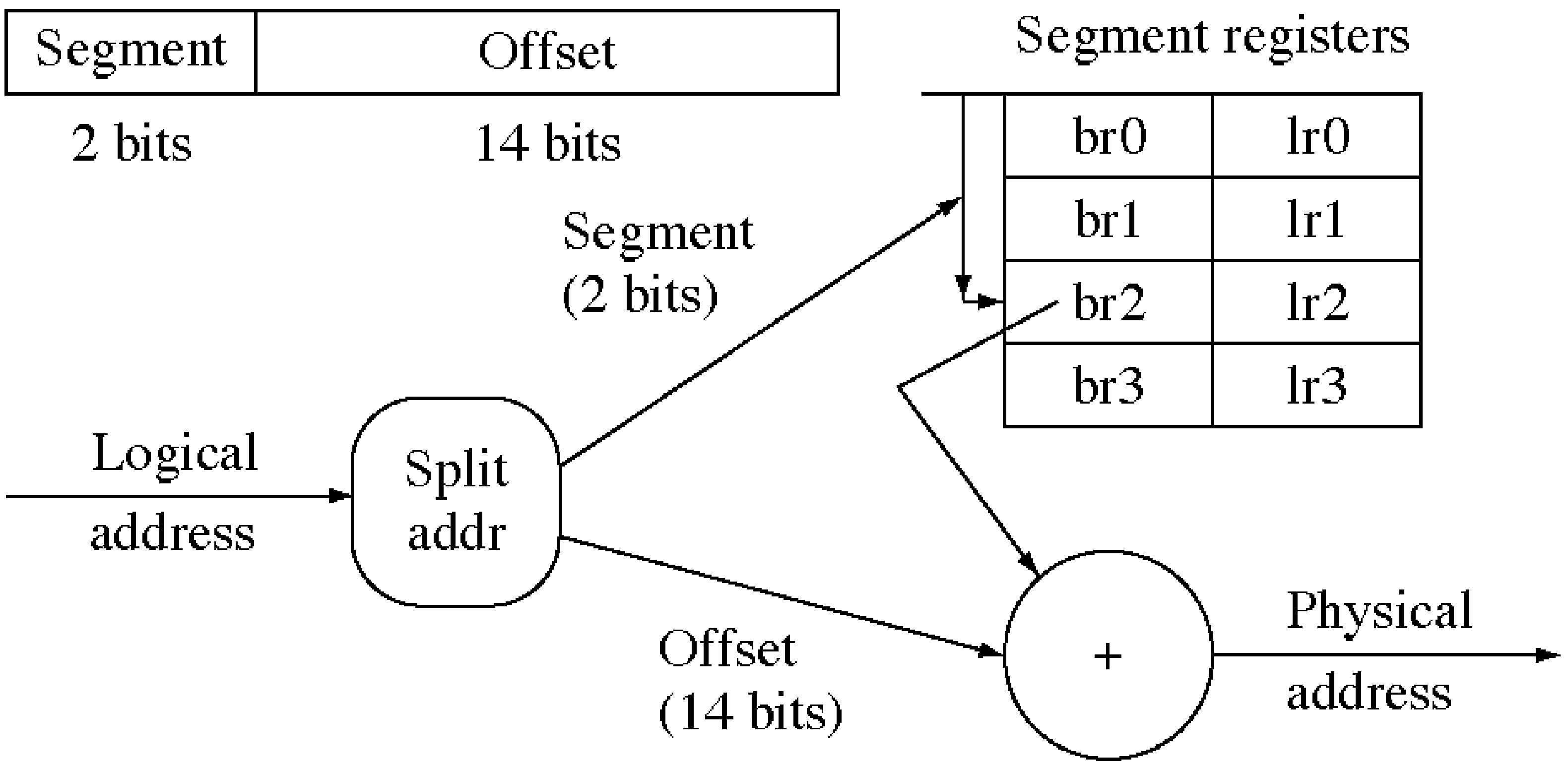
Segmentation

- Divide the logical address space into segments (variable-sized chunks of memory)
- Each segment has a base and bound register
  – and so segments do not need to be contiguous in the physical address space
  – but the logical address space is still contiguous
- DEC PDP11
  – eight segments
  – up to 8K bytes per segment

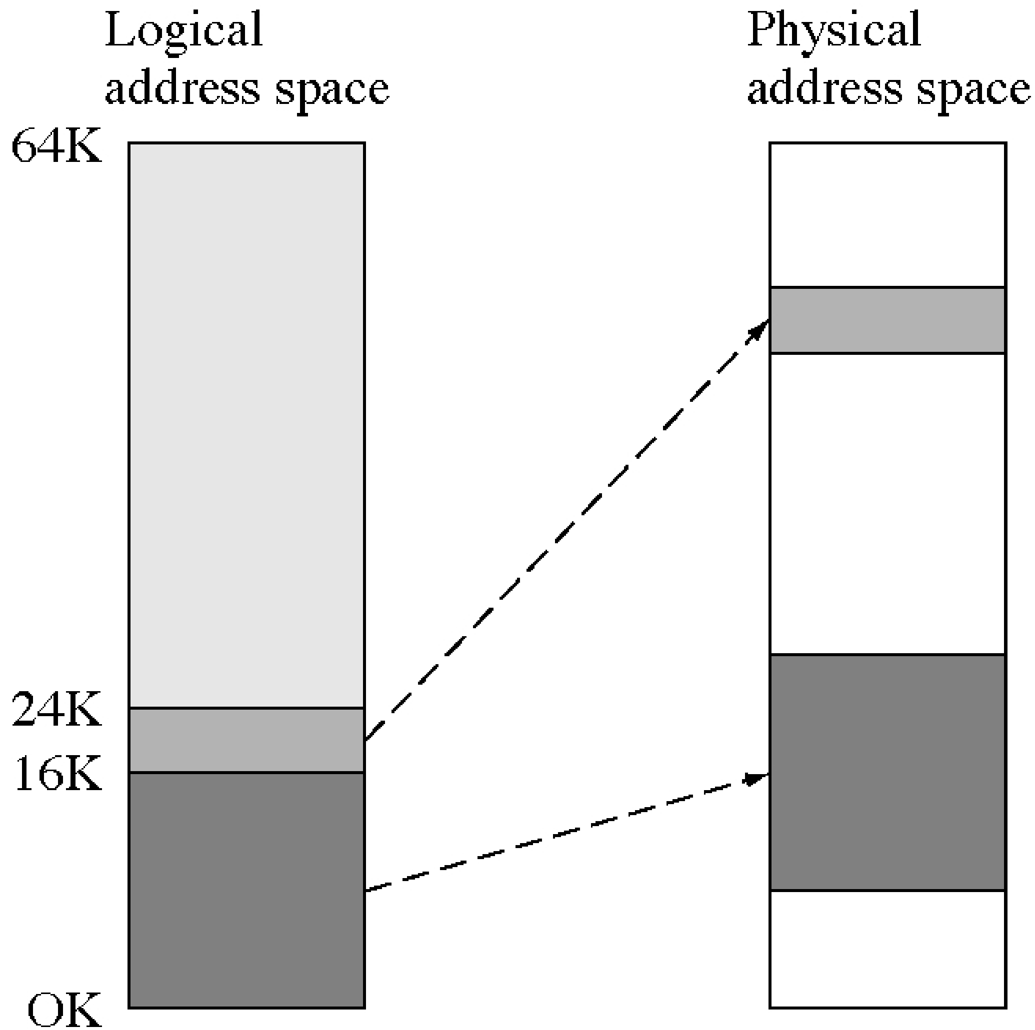
Two segmented address spaces



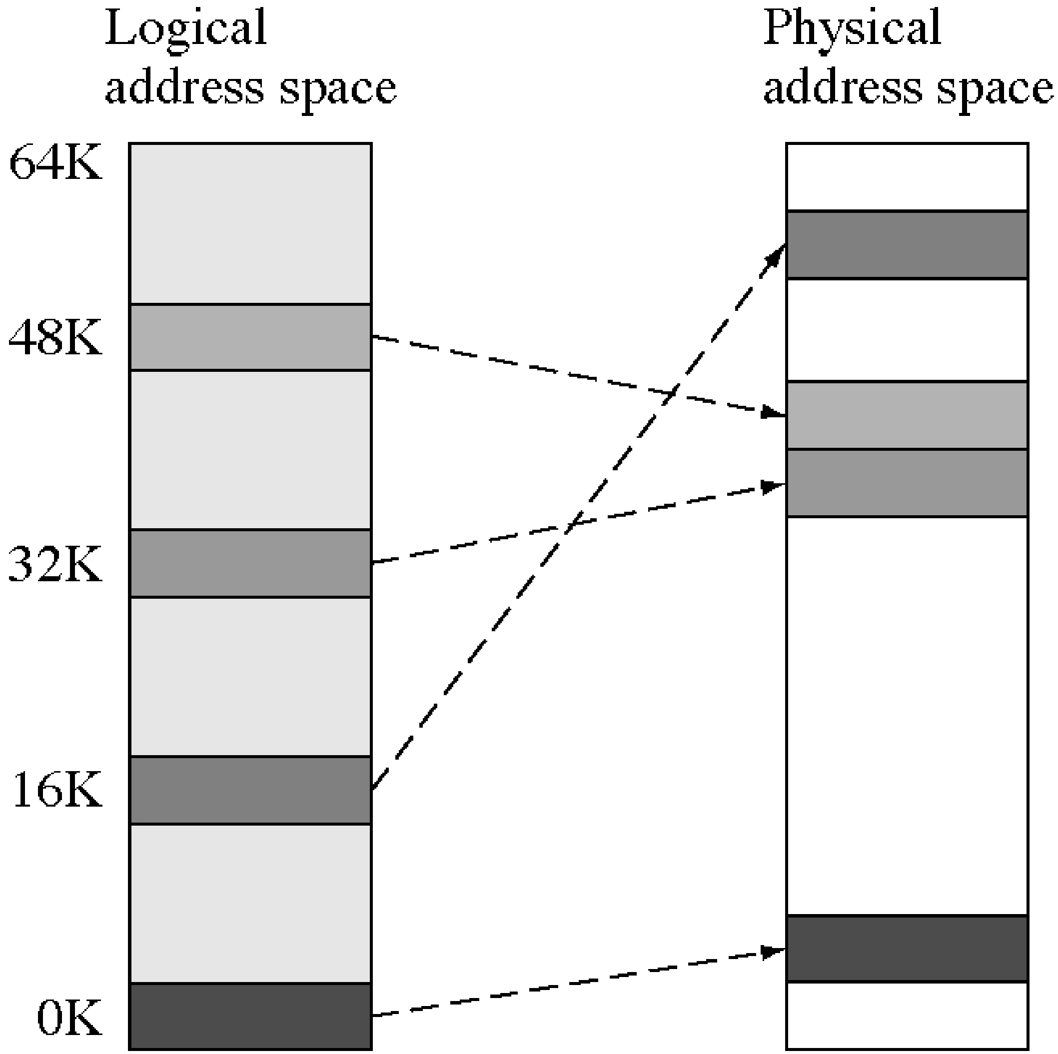
Segmentation memory mapping



Contiguous 24K logical address space



Noncontiguous 24K logical address space



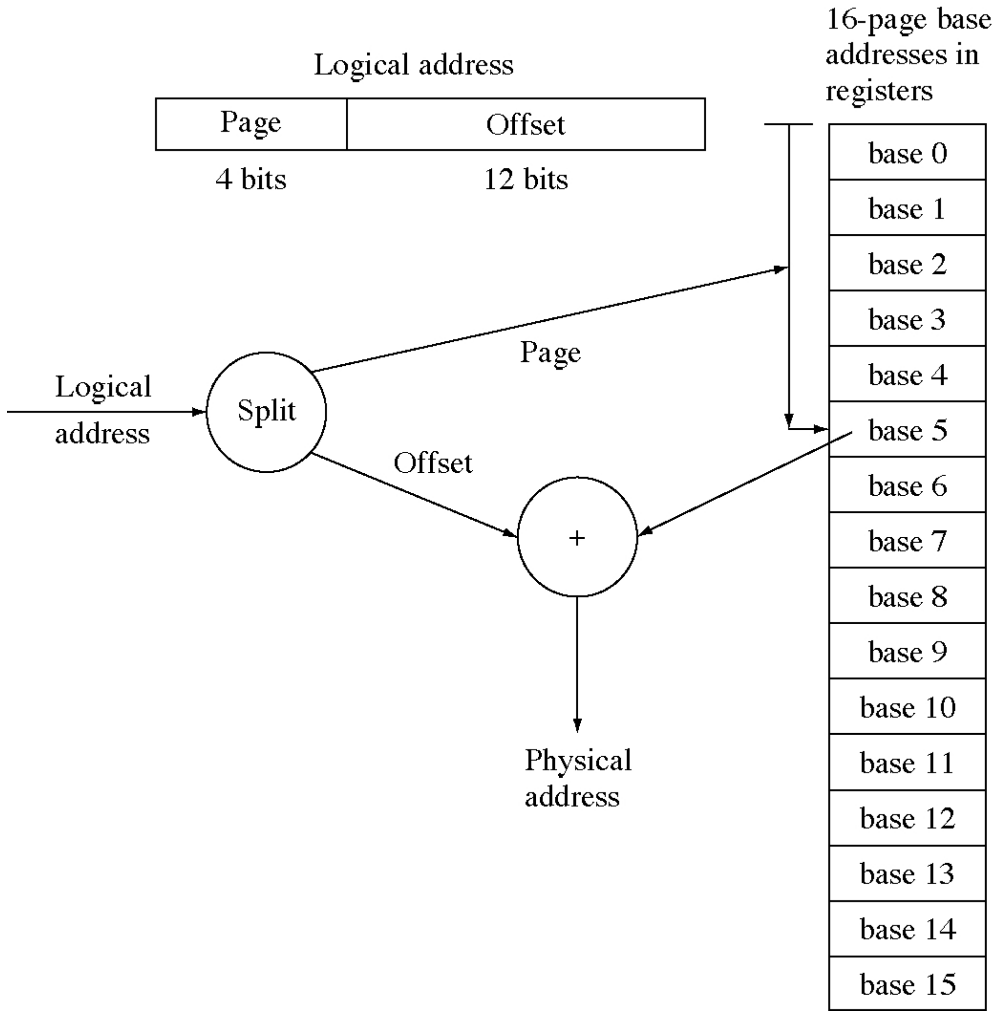
Noncontiguous logical address spaces

- This is possible with segmentation hardware
  - but is not usually a good idea
- Example segmentation map
- Segment-Base - Limit - Logical address
  - 0 100K 6K 0K-6K
  - 1 194K 6K 16K-22K
  - 2 132K 6K 32K-38K
  - 3 240K 6K 48K-54K

Segments to pages

- Large segments do not help the fragmentation problem
  - so we need small segments
- Small segments are usually full
  - so we don't need a length register
  - just make them all the same length
- Identical length segments are called *pages*
- We use page tables instead of segment tables
  - base register but no limit register

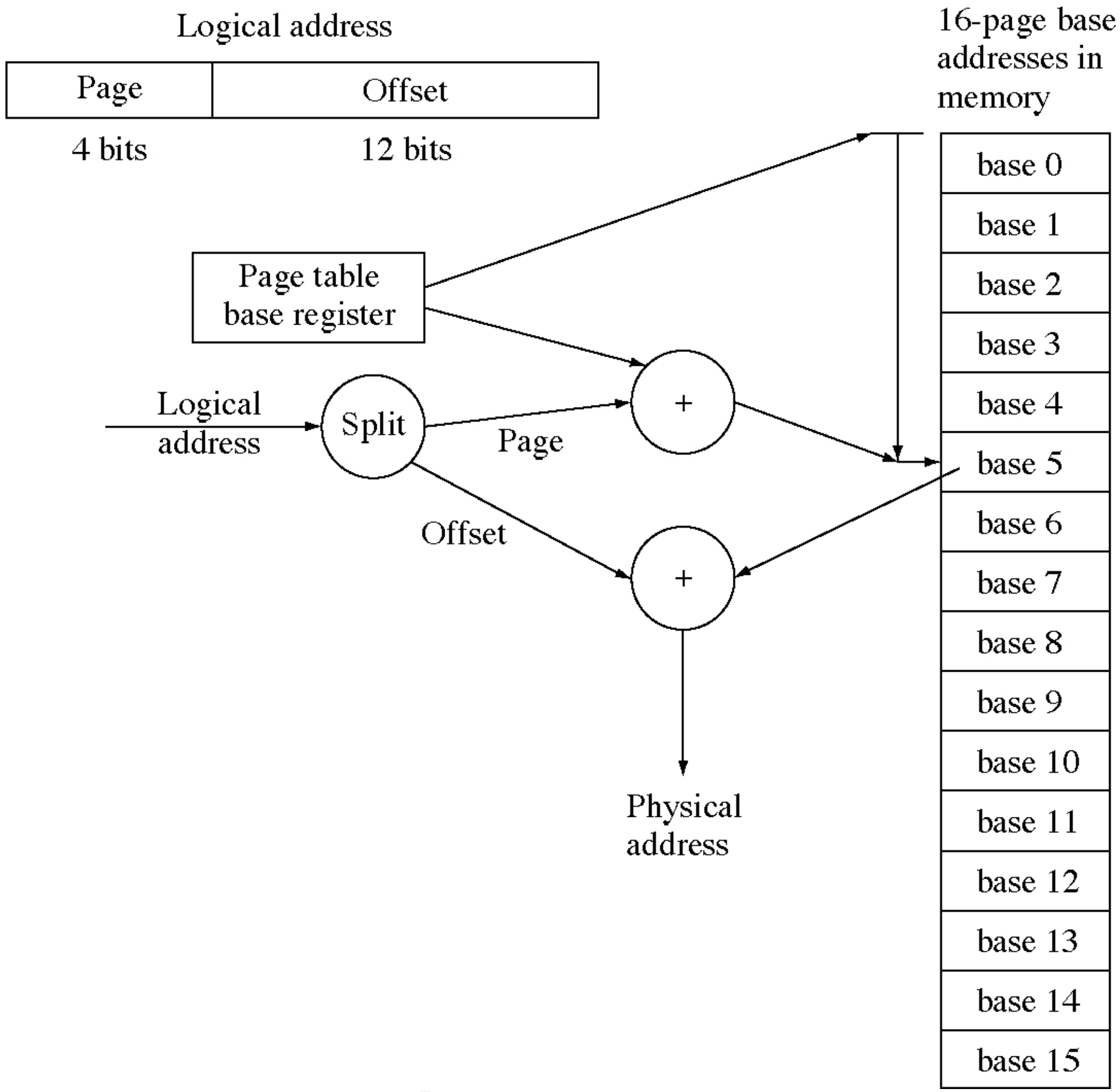
Hardware register page table



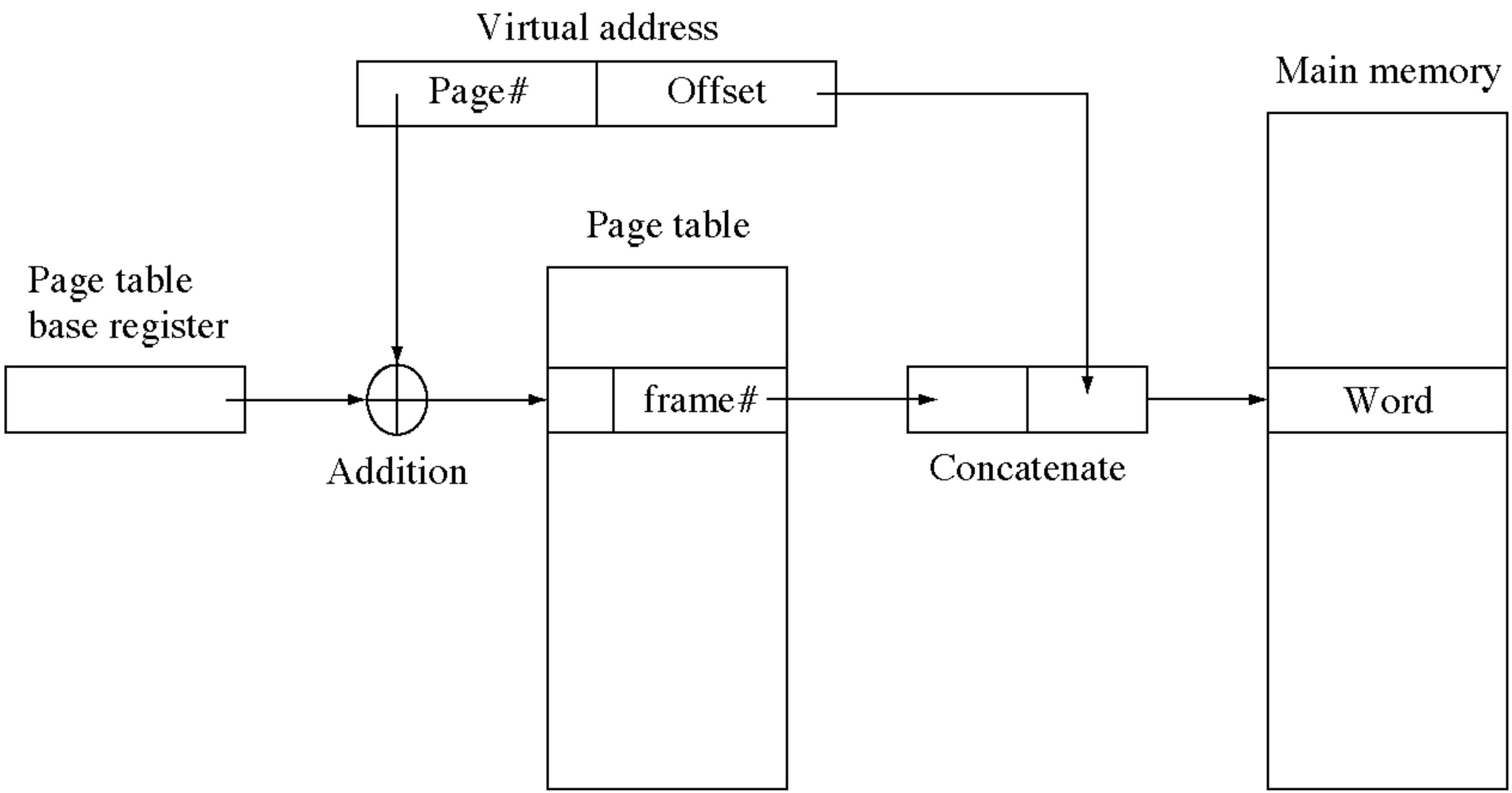
Problems with page tables in registers

- Practical limit on the number of pages
- Time to save and load page registers on context switches
- Cost of hardware registers
- *Solution*: put the page table in memory and have a single register that points to it

Page tables in memory



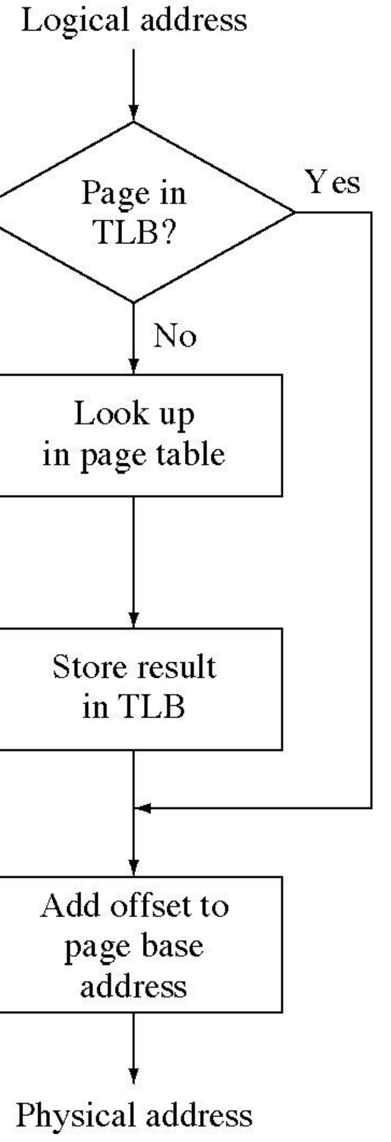
Page table mapping



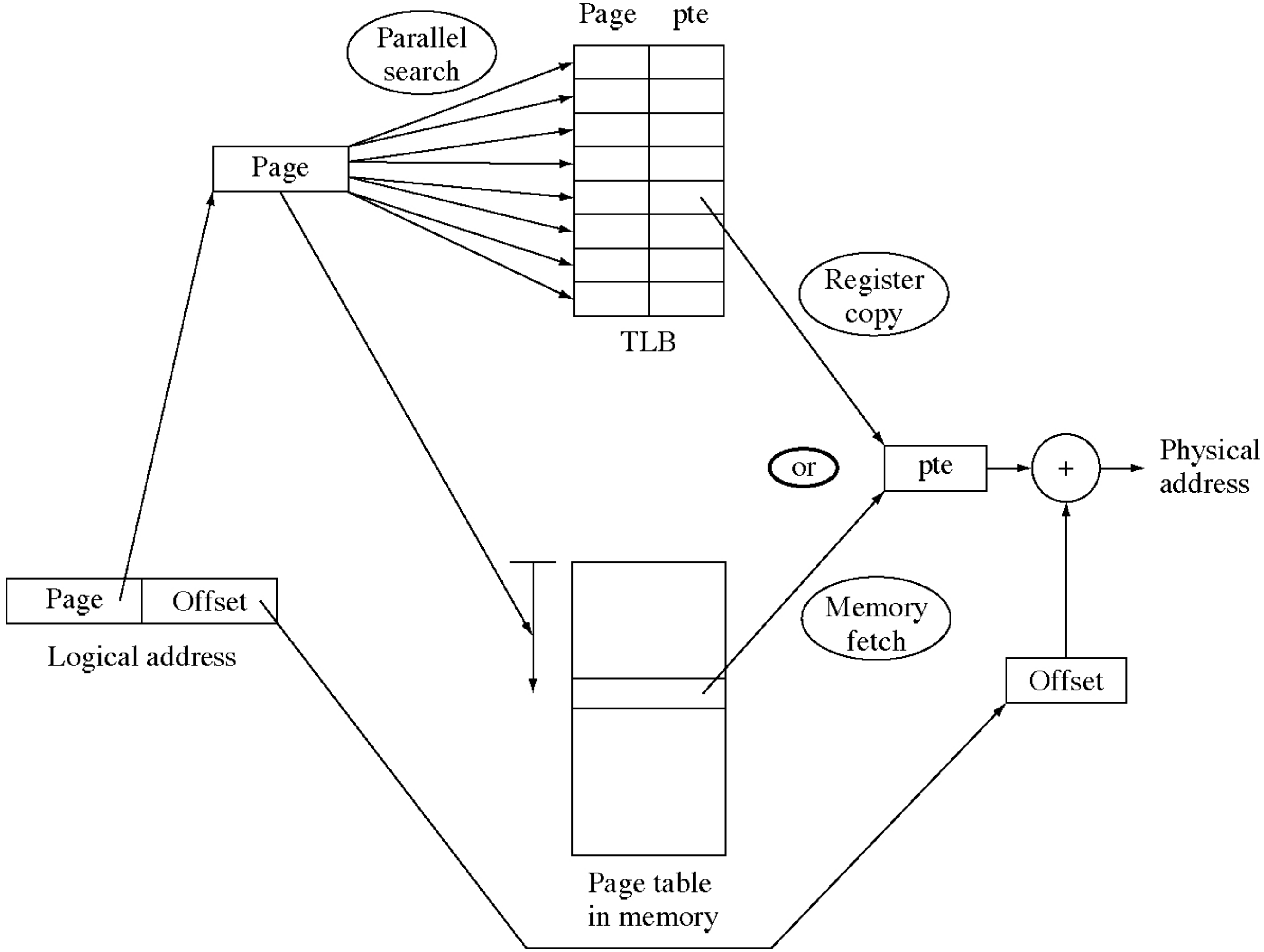
Problems with page tables in memory

- Every data memory access requires a corresponding page table memory access
  - the memory usage has doubled
  - and program speed is cut in half
- *Solution*: caching page table entries
  - called a *translation lookaside buffer*
  - or TLB

TLB flow chart



TLB lookup


Page
pte
Parallel search
Page
TLB
Register copy
or
pte
+
Physical address
Page
Offset
Logical address
Memory fetch
Offset
Page table in memory

Caching the page table (1 of 2)

- ```
  const int PageTableCacheSize = 8;
  const int pageSizeShift = 12;
  const int pageSizeMask = 0xFFF;
  struct CacheEntry {
    int logicalPageAddress;
    int pageBaseAddress;
  } PageTableCache[PageTableCacheSize];
  extern int pageTableBaseRegister;
  int LeastRecentlyUsedCacheSlot( void );
  ```
  ```

Caching the page table (2 of 2)

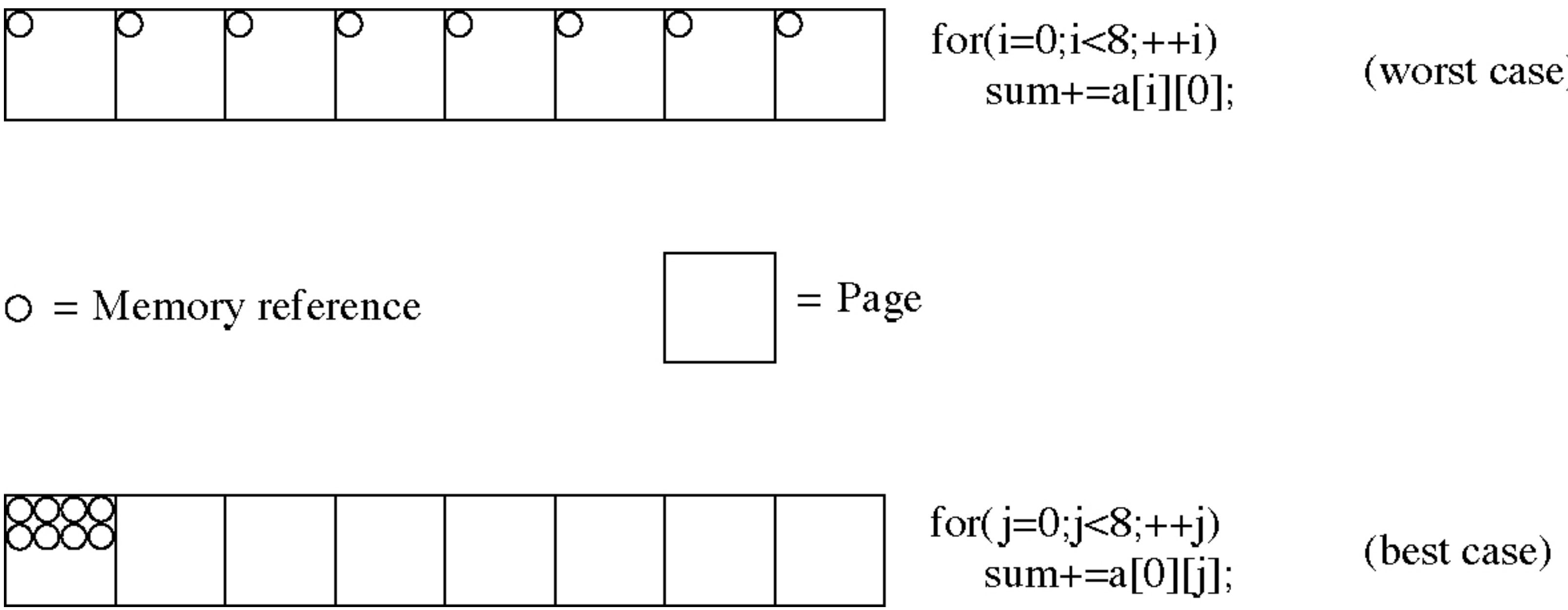
- ```
  int LogicalToPhysical( int logicalAddress ) {
    int logicalPageAddress = logicalAddress & ~pageSizeMask;
    for( int i = 0; i < PageTableCacheSize; ++i ) {
      // the hardware lookup is done in parallel
      if( PageTableCache[i].logicalPageAddress
          == logicalPageAddress )
        return PageTableCache[i].pageBaseAddress;
    }
    int pteAddress = pageTableBaseRegister
                   + (logicalAddress >> pageSizeShift);
    int pageBaseAddress = MemoryFetch( pteAddress );
    // now update the cache by replacing the entry that has
    // not been used in the longest time (the least recently

    // used one) with this new entry
    i = LeastRecentlyUsedCacheSlot();
    PageTableCache[i].logicalPageAddres=logicalPageAddress;
    PageTableCache[i].pageBaseAddress = pageBaseAddress;
    return pageBaseAddress;
  }
  ```
  ```

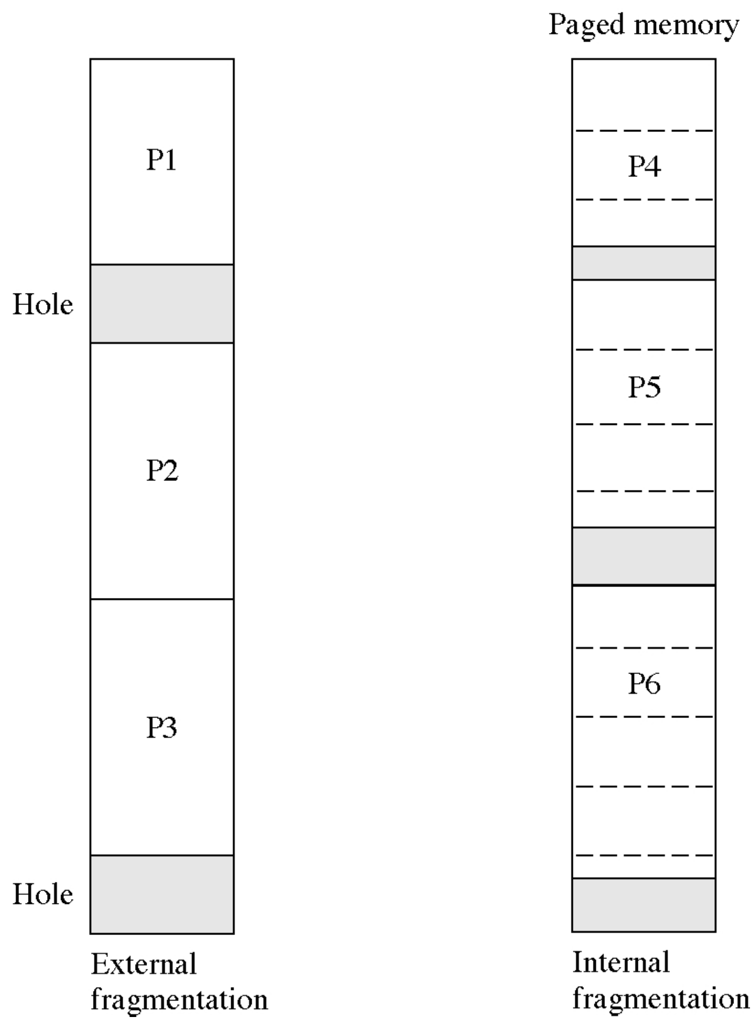
Why TLBs work

- Memory access is not random, that is, not all locations in the address space are equally likely to be referenced
- References are localized because
  - sequential code execution
  - loops in code
  - groups of data accessed together
  - data is accessed many times
- This property is called *locality*
- TLB hit rates are 90+% .

Good and bad cases for paging



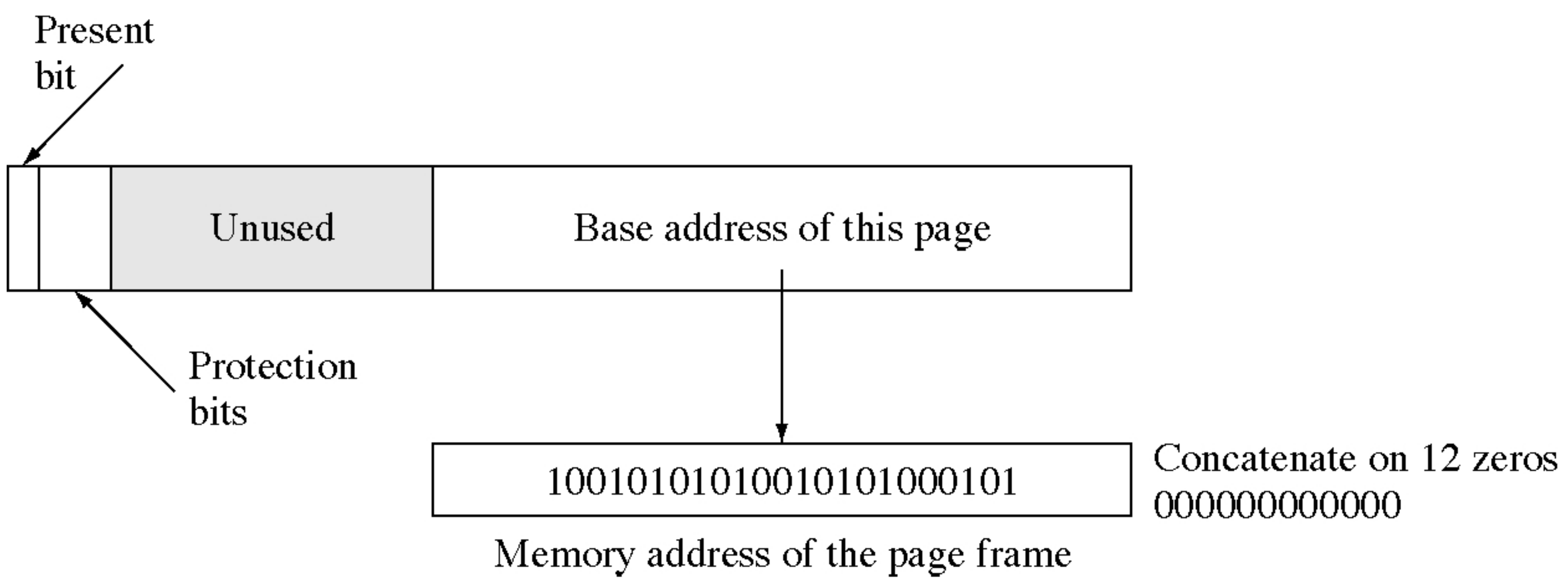
Internal and external fragmentation



Page and page frame

- Page
  - the information in the page frame
  - can be stored in memory (in a page frame)
  - can be stored on disk
  - multiple copies are possible
- Page frame
  - the physical memory that holds a page
  - a resource to be allocated

Page table entry



Page table protection

- Three bits control: read, write, execute
- Possible protection modes:
  - 000: page cannot be accessed at all
  - 001: page is read only
  - 010: page is write only
  - 100: page is execute only
  - 011: page can be read or written
  - 101: page can be read as data or executed
  - 110: write or execute, unlikely to be used
  - 111: any access is allowed

Paged memory allocator (1 of 4)

- ```
  const int PageSize = 4096;
  struct MemoryRequest {
    int npages;
    // size of the request in pages
    Semaphore satisfied; // signal when memory allocated
    int * pageTableArray;   // store page numbers here
    MemoryRequest *next, *prev; // doubly linked list
  };
  // The memory request list
  // keep a front and back pointer for queue discipline
  MemoryRequest * RequestListFront, *RequestListBack;
  // The structure for the free page list
  struct FreePage {
    int pageNumber;
    FreePage *next;
  };
  // The free page list
  FreePage * FreePageList;
  int NumberOfFreePages;
  ```
  ```

Paged memory allocator (2 of 4)

```
• void Initialize( int npages ) {
    RequestListFront = 0; FreePageList = 0;
    NumberOfFreePages = npages;
    for( int i = 0; i < NumberOfFreePages; ++i ) {
      FreePageList = new FreePage( i, FreePageList );
    }
  }
  // request procedure: request a piece to be allocated
  void RequestBlock( int npages, Semaphore * satisfied,
      int * pageTableArray ) {
    MemoryRequest * n = new MemoryRequest( npages,
      satisfied, pageTableArray, 0 , 0);
    if( RequestListFront == 0 ) { // list was empty
      RequestListFront = RequestListBack = n;
    } else {
      RequestListBack->next = n;
      RequestListBack = n;
    }
    TryAllocating();
  }
```

Paged memory allocator (3 of 4)

- ```
  void TryAllocating( void ) {
    MemoryRequest * request = RequestListFront;
    while( request != 0 ) {
      if( CanAllocate( request ) {
        if( RequestListFront == RequestListBack ) {
          RequestListFront = 0;
          request = 0; // drop out of loop
        } else {
          request->prev->next = request->next;
          request->next->prev = request->prev;
          MemoryRequest * oldreq = request;
          // save the address
          request = request->next;
          delete oldreq;
        }
      } else
        request = request->next;
    }
  }
  ```
  ```

Paged memory allocator (4 of 4)

```
• void FreePages( int npages, int pageTable[] ) {
    for( int i = 0; i < npages; ++i )
      FreePageList
        = new FreePage( pageTable[i], FreePageList );
  }
  int CanAllocate( MemoryRequest * request ) {
    if( request->npages >= NumberOfFreePages ) {
      NumberOfFreePages -= request->npages;
      int * p = request->pageTableArray;
      for( int i = 0; i <  request->npages; ++i ) {
        *p++ = FreePageList->pageNumber;
        FreePage * fpl = FreePageList;
        FreePageList = FreePageList->next;
        delete fpl;
      }
      return True;
    }
    return False;
  }
```

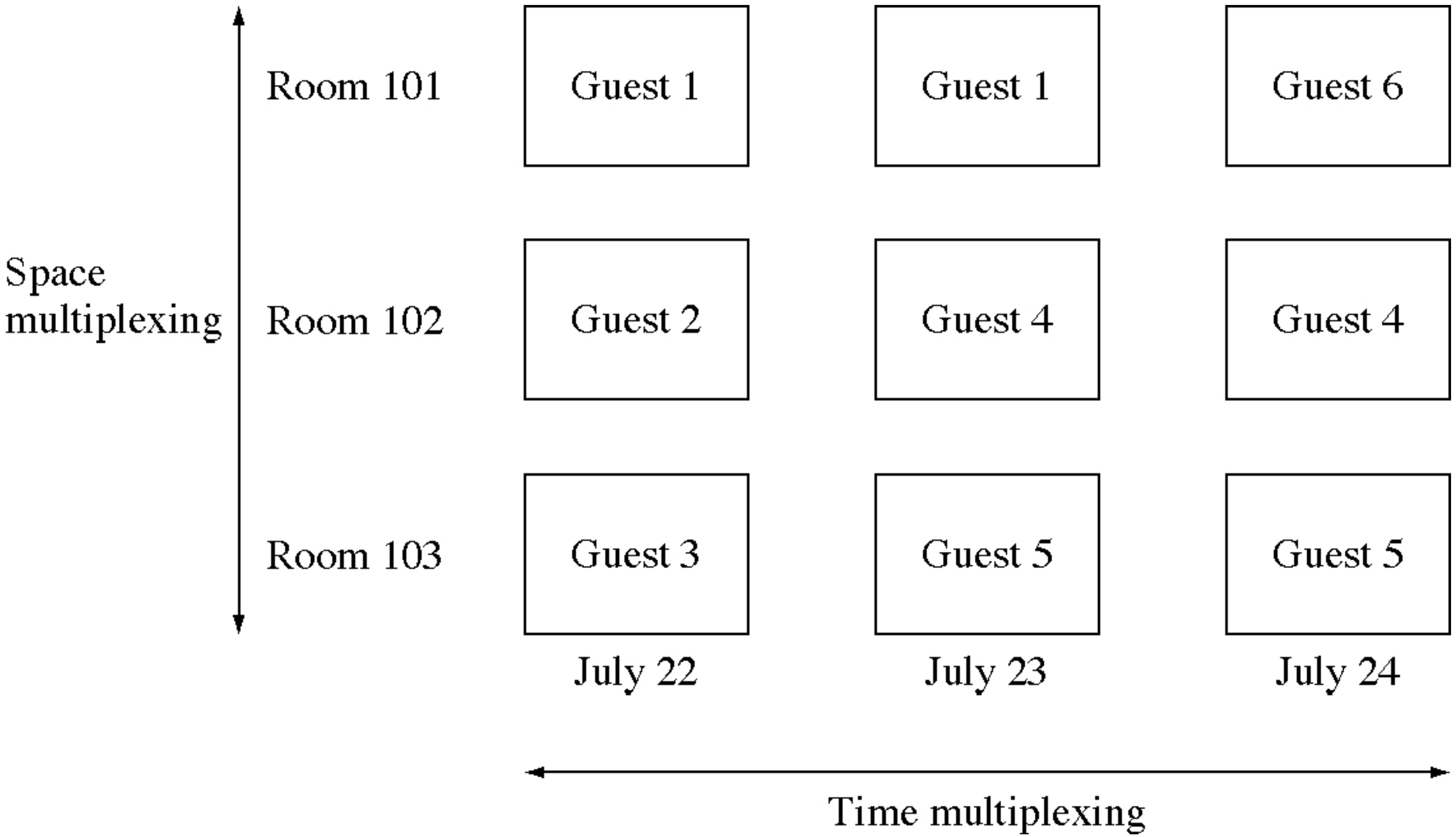
The design process

- Evolution of solutions to the memory problem is a good example of the design process in action
  - at each stage the current solution had a problem
  - we modified the design to fix the problem
  - this created a new problem
  - we continued until the solution was good enough
- Sometimes we reused previously discarded ideas

Time and space multiplexing

- The processor is a time resource
  - It can only be time multiplexed
- Memory is a space resource
  - We have looked at space multiplexing of memory
  - Now we will look at time multiplexing of memory

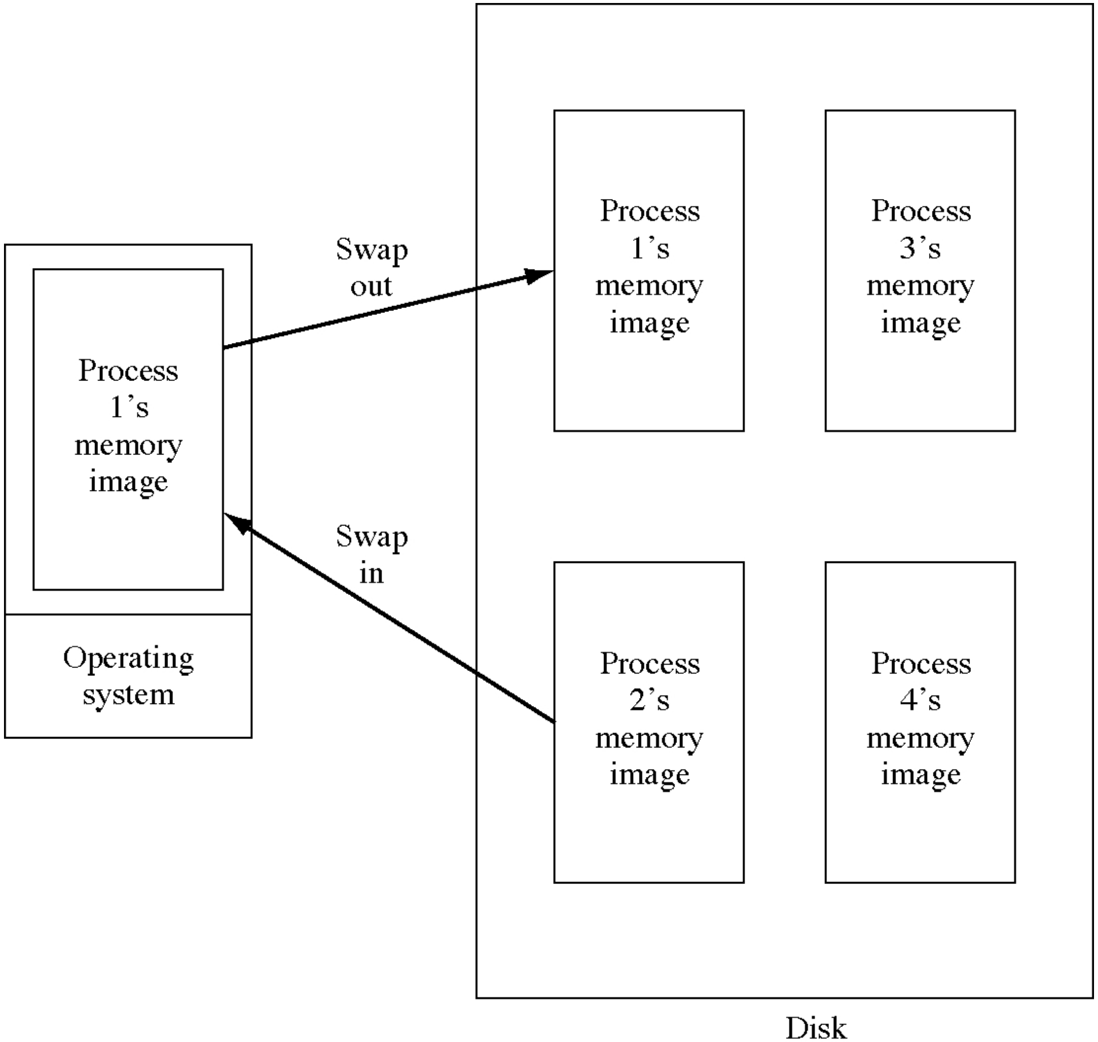
Time and space multiplexing of hotel rooms



Time multiplexing memory

- Swapping: move whole programs in and out of memory (to disk or tape)
  - allowed time-sharing in early Oss
- Overlays: move parts of program in and out of memory (to disk or tape)
  - allowed the running of programs that were larger than the physical memory available
  - widely used in early PC systems

Swapping



Swap out
Swap in
Process 1's memory image
Operating system
Process 1's memory image
Process 3's memory image
Process 2's memory image
Process 4's memory image
Disk

Design technique: Persistent objects

- A process was a dynamic entity in the system
  - we wanted to write it out to disk
  - and read it back in again later
  - kind of freeze and unfreeze it
- The ability to write an object to disk is called *persistence*, and it very useful
- It allows objects to live beyond the execution of the program that creates them

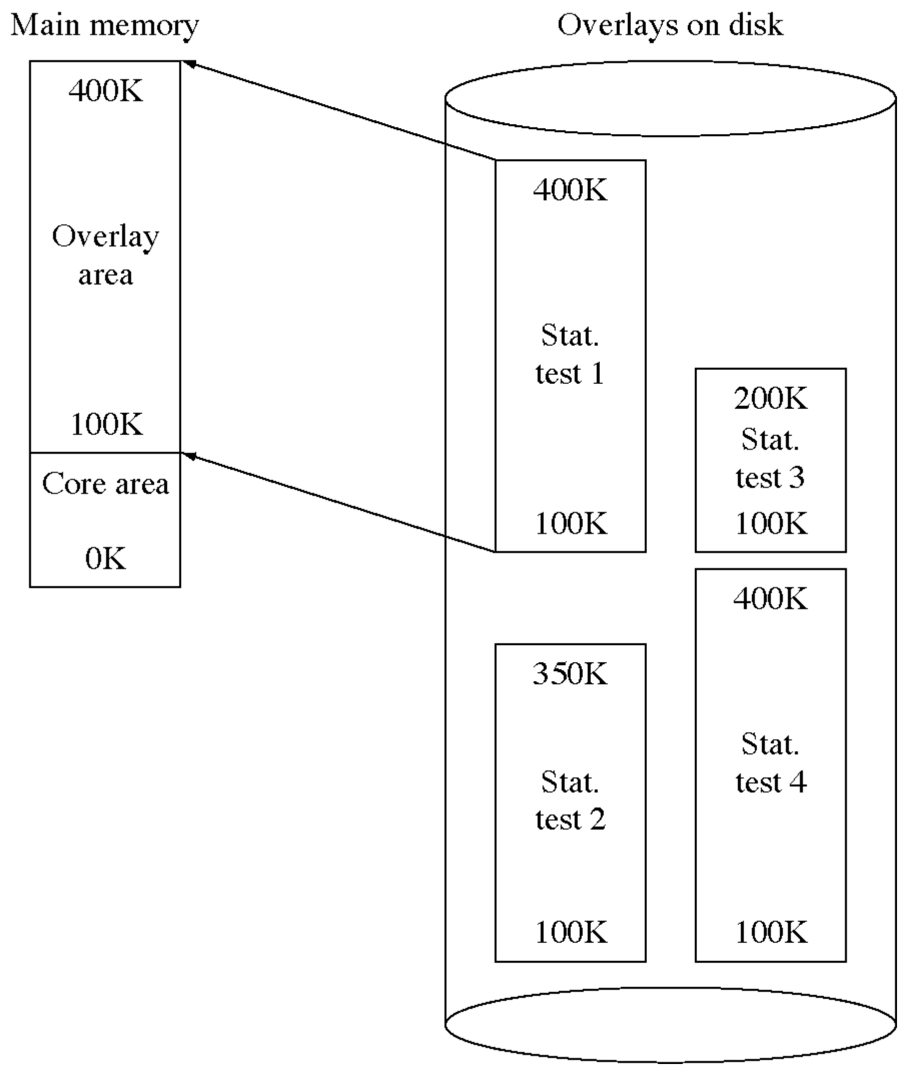
How to create persistence

- Basically simple
  - write out a representation of the objects
- Problems
  - pointers and references: we must encode these in some way in order to write them out
  - following references: we also have to write out everything the object refers to.

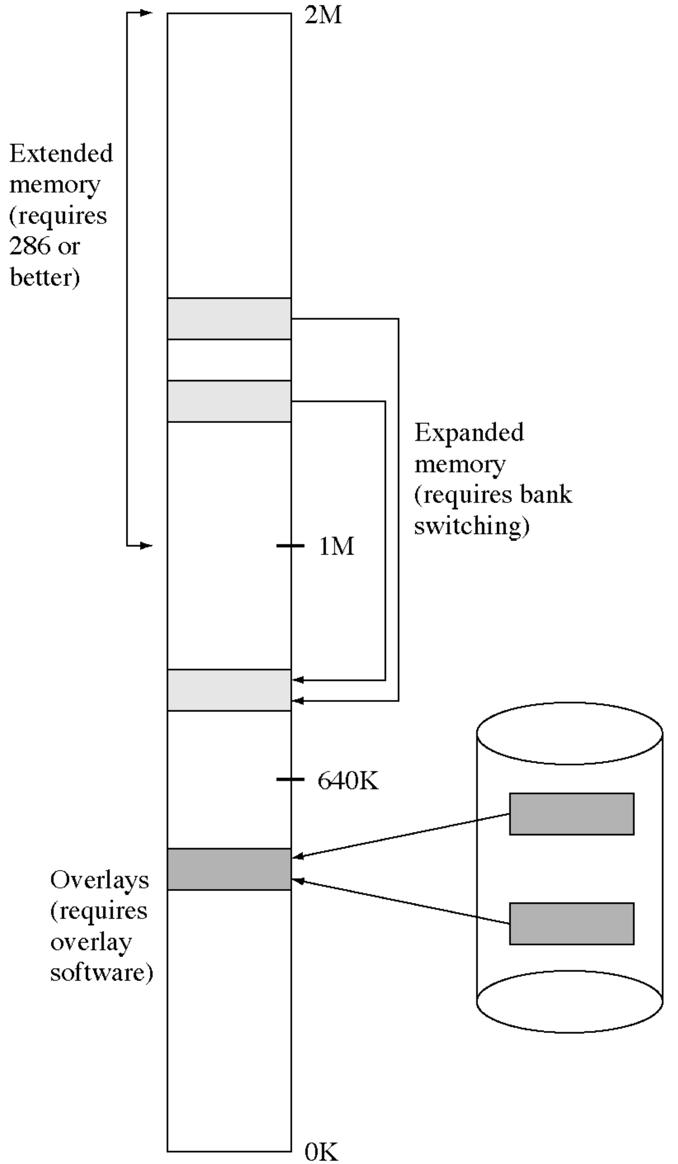
Design stages: Design and implementation

- First we decide what we want
- Then we decide how to implement it
- Each stage requires design
- Feedback is often required in order to avoid inefficient designs
- In operating systems
  - we often useful services to the processes
  - and try to implement them efficiently

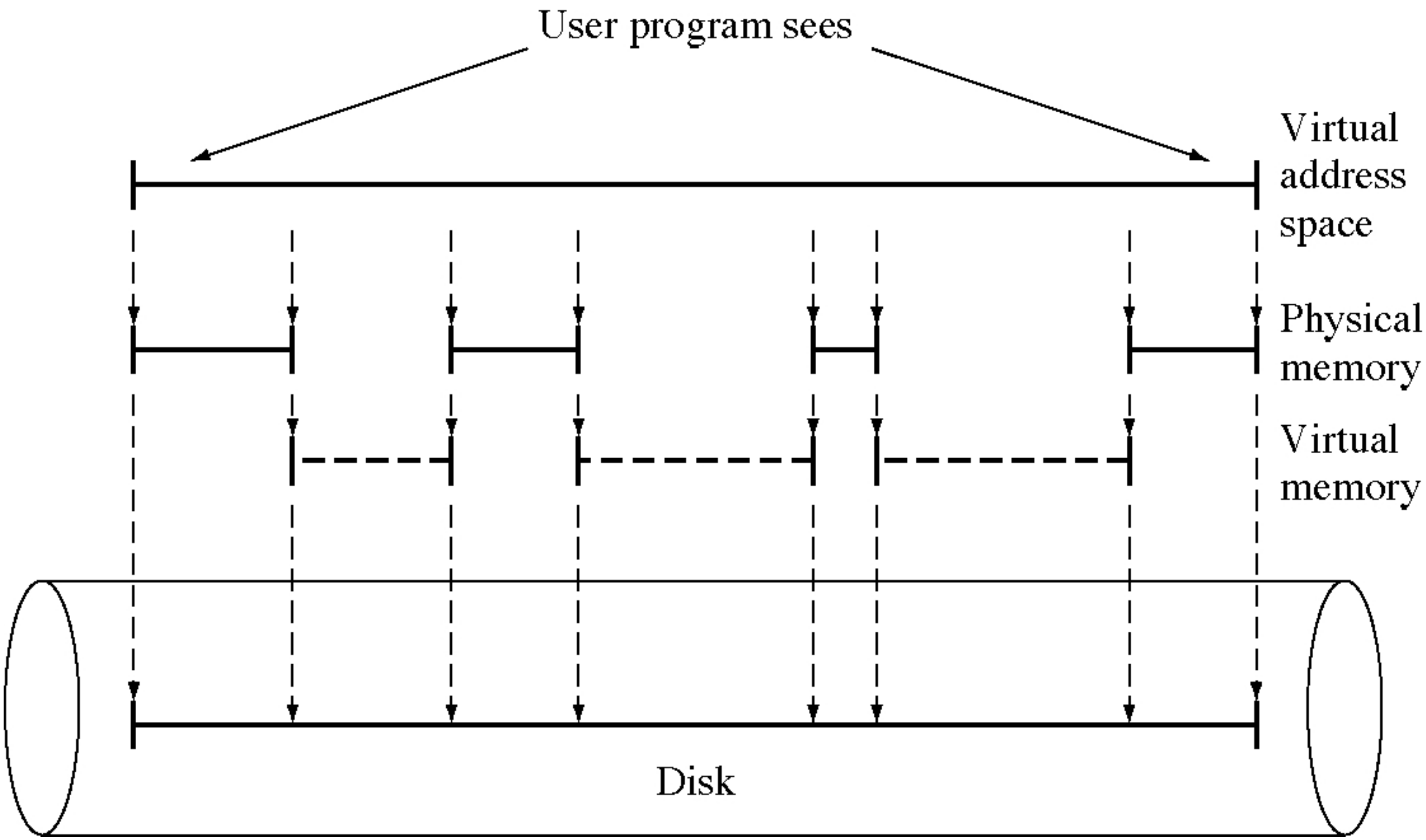
Overlays


Main memory
Overlays on disk
400K
Overlay area
100K
Core area
0K
400K
Stat. test 1
100K
200K
Stat. test 3
100K
400K
Stat. test 4
100K
350K
Stat. test 2
100K

Overlays in PCs



Virtual memory



Implementation of virtual memory

- Virtual memory allows
  - time multiplexing of memory
  - users to see a larger (virtual) address space than the physical address space
  - the operating system to split up a process in physical memory
- Implementation requires extensive hardware assistance and a lot of OS code and time
  - but it is worth it

Virtual memory algorithm (1 of 2)

- ```
  const int LogicalPages  = 1024;
  const int ByterPerPage  = 4096;
  const int OffsetShift   = 12;
  const int OffsetMask    = 0xFFF;
  const int PhysicalPages =  512;
  enum AccessType { invalid = 0, read = 1, write = 2,
  execute = 3 };
  struct PageTableEntry {
    int pageBase   : 9;
    int present    : 1;
    AccessType protection : 2;
    int fill       : 4; // fill to 16 bits
  };
  PageTableEntry UserPageTable[LogicalPages];
  ```
  ```

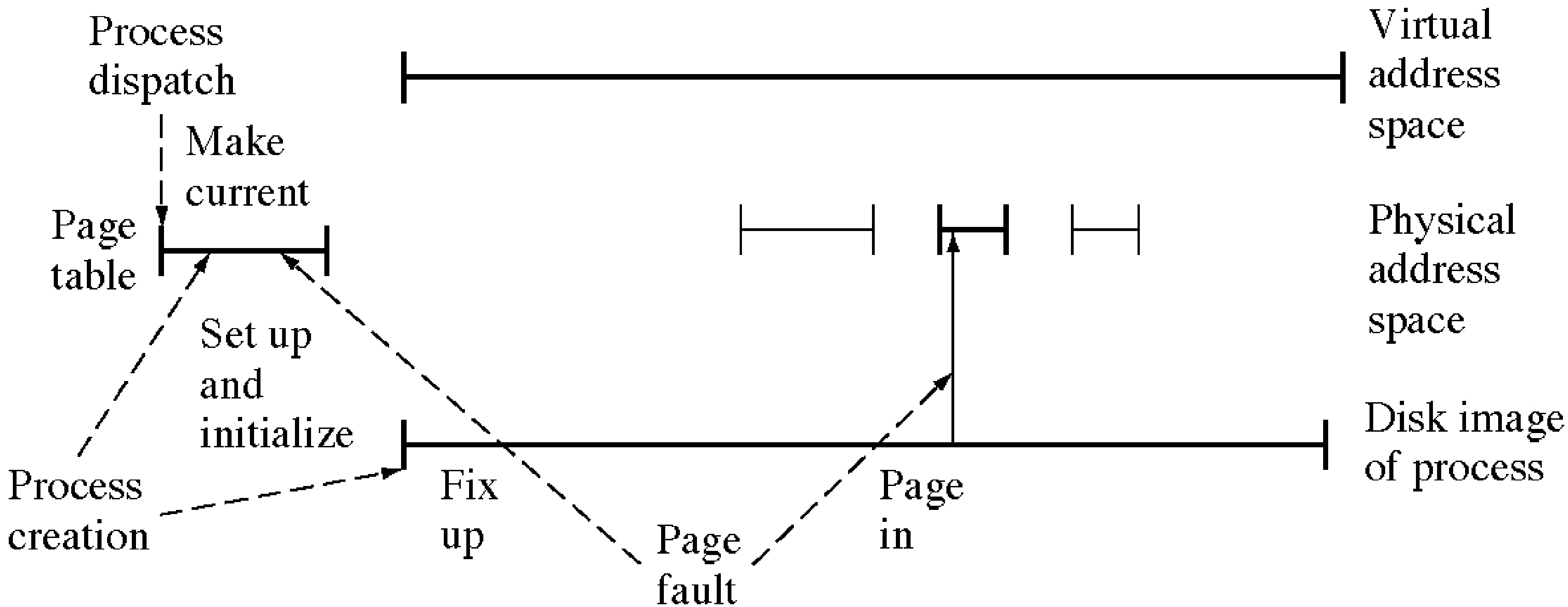
Virtual memory algorithm (2 of 2)

- ```
  int MemoryAccess( int logicalAddress,
      AccessType how, int dataToWrite = 0 ) {
    int page = logicalAddress >> OffsetShift;
    int offset = logicalAddress & OffsetMask;
    PageTableEntry pte = UserPageTable[page];
    if( how != pte.protection )
      if( !(how = read && ptr.protection = write) ) {
        CauseInterrupt( ProtectionViolation );
        return 0;}
    if( pte.present == 0 ) {
      GenerateInterrupt( PageFault, page );
      return 0; }
    int physicalAddress
      = (pte.pageBase << OffsetShift) + offset;
    switch( how ) {
      case read: case execute:
        return PhysicalMemoryFetch(physicalAddress);
      case write:
        PhysicalMemoryStore(
          physicalAddress, dataToWrite );
        return 0;
    }
  }
  ```
  ```

Virtual memory software

- The virtual memory (a.k.a. paging) system in the OS must respond to four events
  - process creation
  - process exit
  - process dispatch
  - page fault

Virtual memory events



Process creation actions

- 1. Compute program size (say N pages)
- 2. Allocate N page frames of swap space
- 3. Allocate a page table (in the OSs memory) for N page table entries.
- 4. Initialize the swap area
- 5. Initialize the page table: all pages are marked as not present.
- 6. Record the location in the swap area and of the page table in the process descriptor

Process exit actions

- 1. Free the memory used by the page table
- 2. Free the disk space in the swap area
- 3. Free the page frames in process was using

Process dispatch actions

- 1. Invalidate the TLB (since we are changing address spaces)
- 2. Load the hardware page table base register with the address of the page table for this process

Page fault actions

- 1. Find the faulting page (say page K)
- 2. Find an empty page frame. This will involve replacing a page.
- 3. Read in page K to this page frame
- 4. Fix up the page table entry for page K. Mark it present and set the base address.
- 5. Restart the process with the instruction that caused the page fault

Locality

- Programs do not access their address space uniformly
  - they access the same location over and over
- *Spatial locality*: processes tend to access location near to location they just accessed
  - because of sequential program execution
  - because data for a function is grouped together
- *Temporal locality*: processes tend to access data over and over again
  - because of program loops
  - because data is processed over and over again

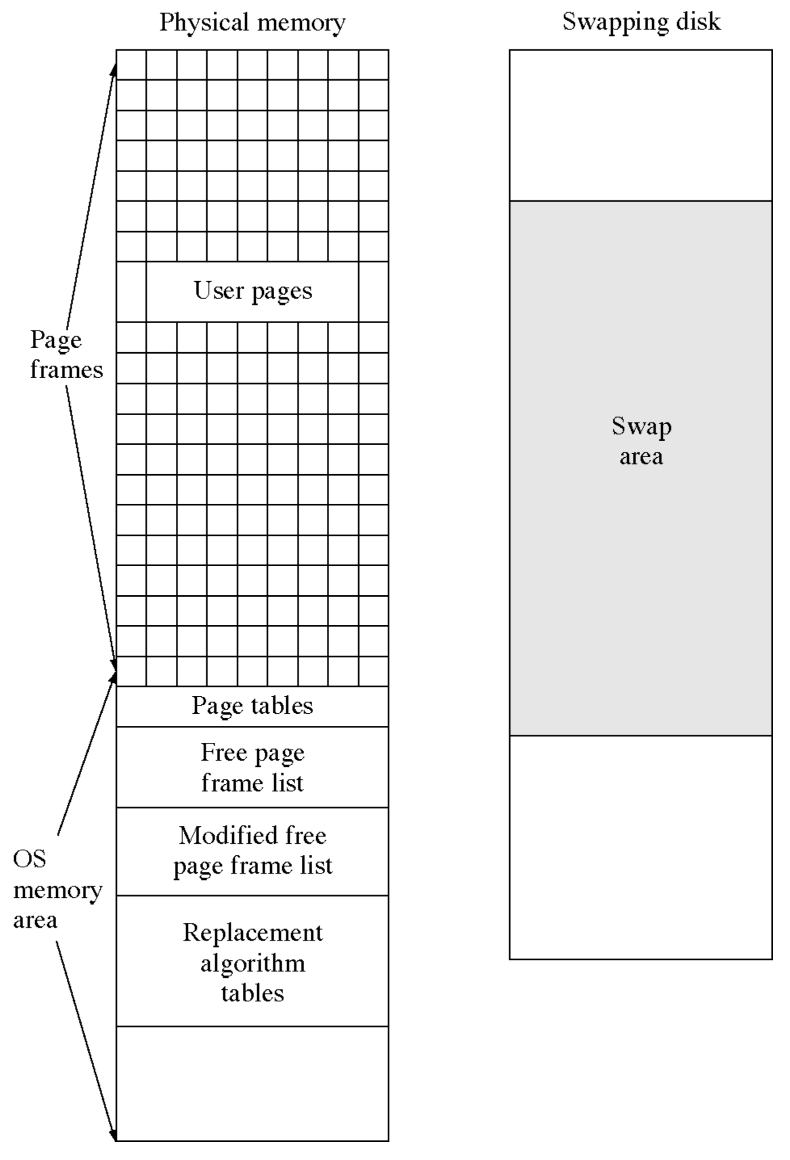
Design technique: Locality

- Locality is almost always present
  - and often we can optimize a design by taking advantage of it.
- Caching is a common name for systems that take advantage of locality to optimize operations

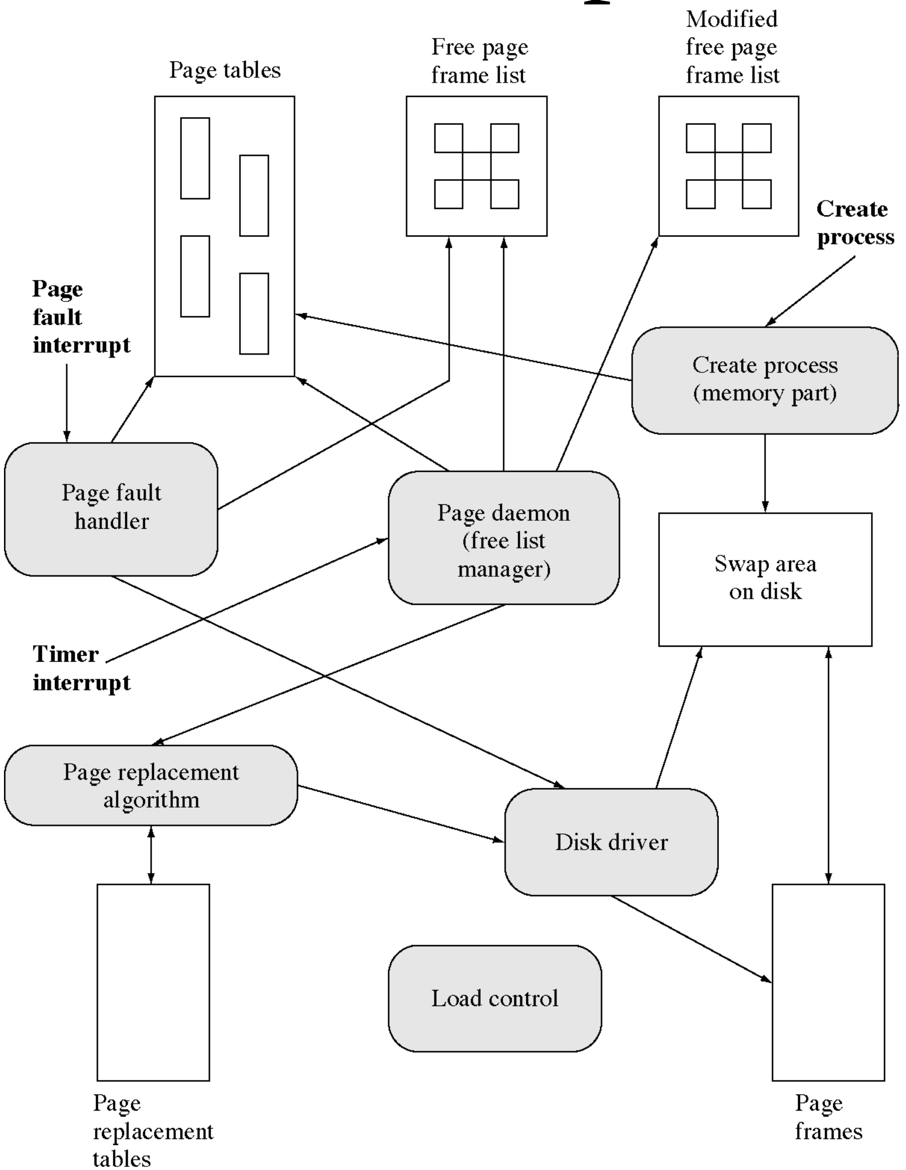
Practicality of paging

- Paging only works because of locality
  - at any one point in time programs don't need most of their pages
- Page fault rates must be very, very low for paging to be practical
  - like one page fault per 100,000 or more memory references

VM data structures



VM events and procedures



Daemons and events

- OS usually respond to events
  - but sometime they need to be proactive
- An OS daemon is a process that wakes up every so often and looks to see if it has any work to do
- It is useful to keep a pool of free pages
  - so page faults can be handled immediately
- A paging daemon wakes up every so often and keeps the free page pool large enough

Design technique: System models and daemons

- Operating systems are essentially reactive
  - they react to interrupts
- But if we include timer interrupts
  - then operating systems are sort of doing things on their own, that is, being proactive
- This is what we call a daemon
  - a daemon wakes up and checks to see if something needs doing

Reactive and proactive user interfaces

- Graphical user interfaces are generally reactive, they wait for user actions
  - this is a good model and puts the user in control which is good psychologically
- Agents are a new user interface concept
  - agents are proactive
  - they go out and look for useful things to do

Design technique: Polling, software interrupts and hooks

- How can a process know when an event occurs? There are two approaches
- *polling*: it can check periodically
- *interrupts*: it can ask another process to interrupt it when the event occurs
  - the other process is usually the one that causes or handles the event
  - so it is not much trouble for it to inform the waiting process that the event has ocurred.

Polling versus interrupts

- Daemons use polling to discover events
  - and then react to the event
- Polling is easier to set up but is less efficient than being interrupted
  - but interrupts require a process to do the interrupting
  - there might be no such process
  - or it might not be set up to provide interrupts

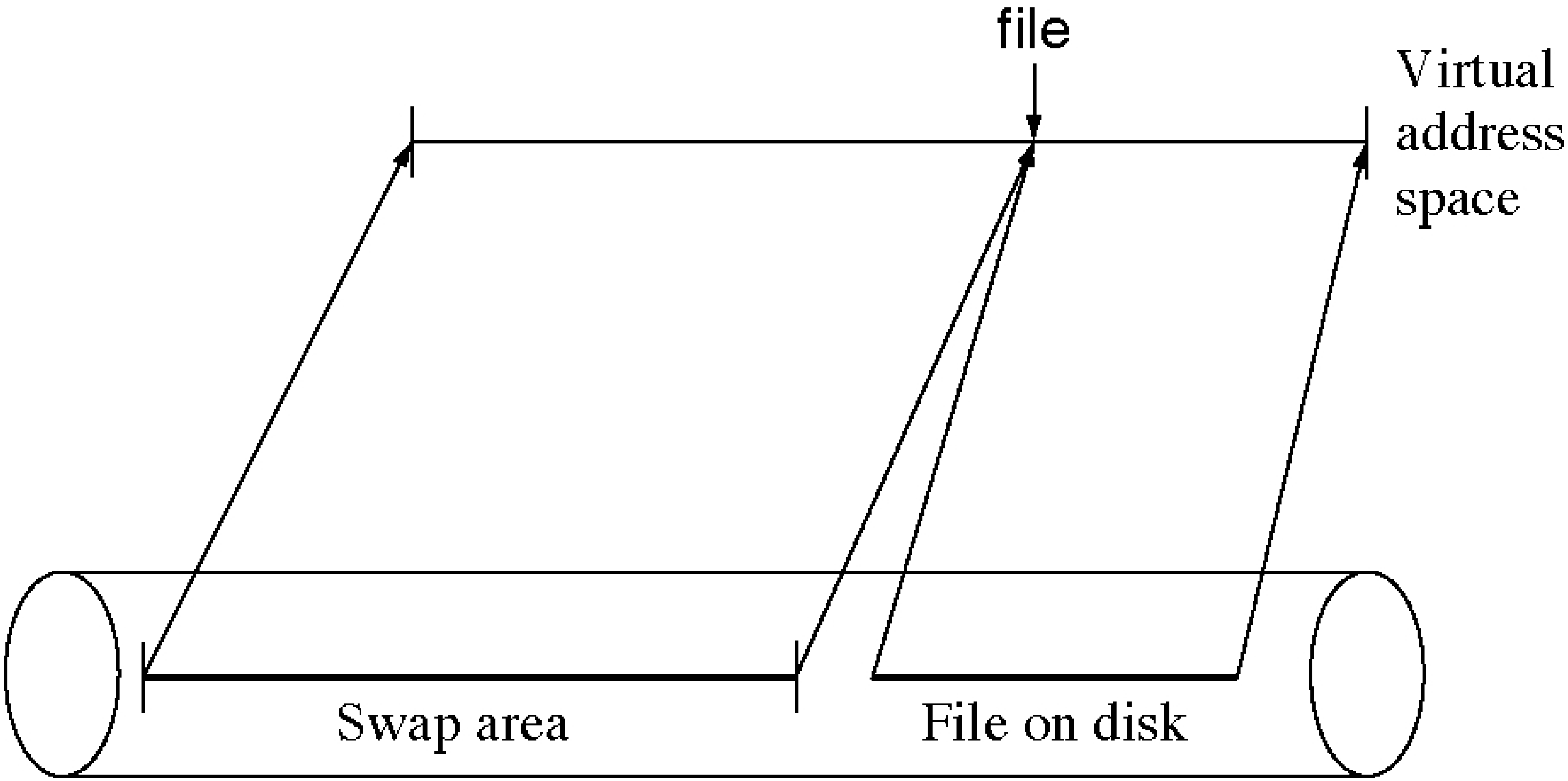
Hooks

- A hook is the ability to register a procedure to be called when an event occurs
- Systems that provide hooks are easy to modify
  - emacs provides hooks for many editing events
  - widget callback functions are hooks
  - a hook is basically a software interrupt

File mapping

- *File mapping* is the mapping of a file on disk into the virtual address space of a process.
- File I/O then consists of reading and writing words in the virtual address space
  - no system calls are required for read and write
- This is also called a *memory-mapped file*.
- The I/O system and the paging system both move data between disk and memory
  - so it makes sense to combine them

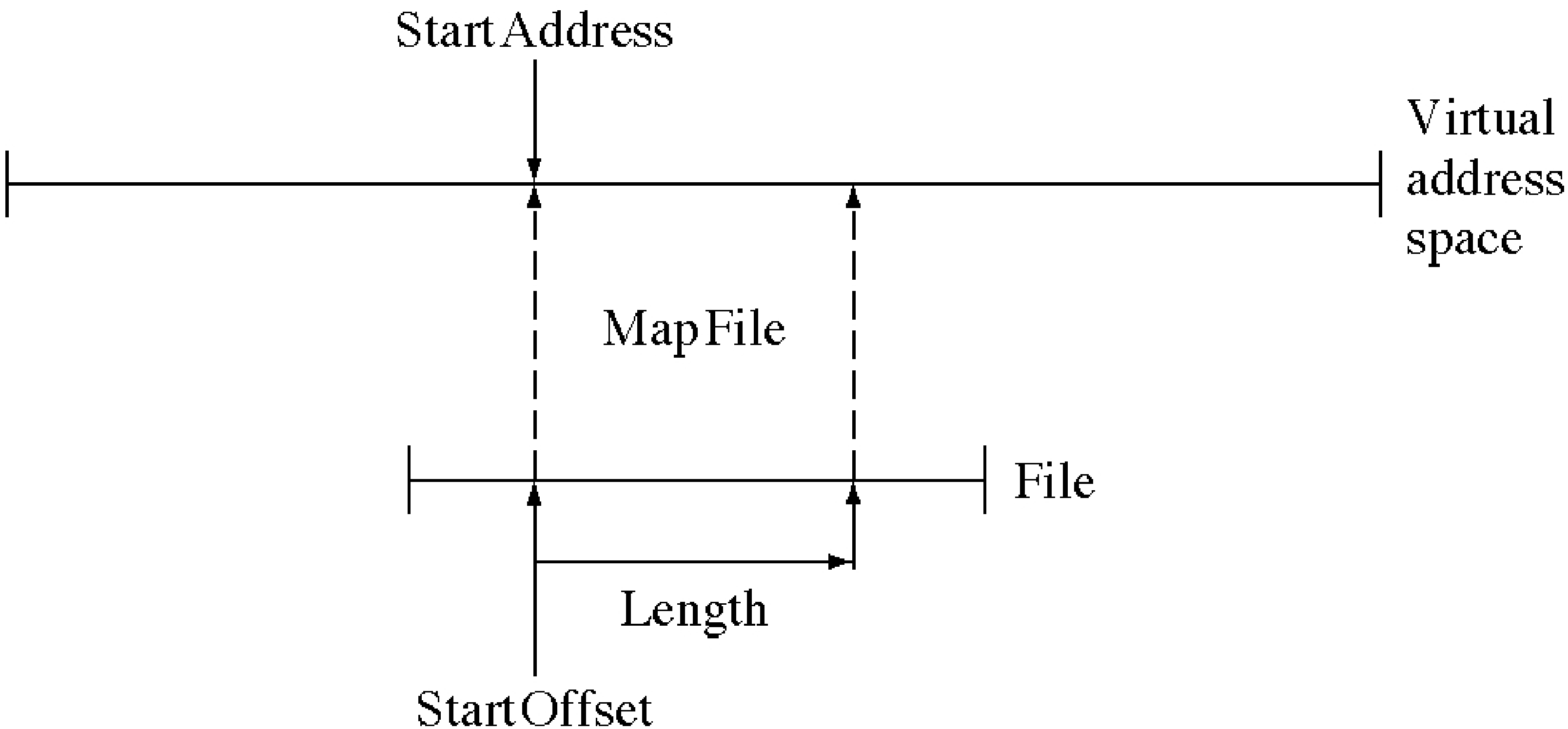
File mapping



File mapping system calls

- A system call is required to map the file into the address space (and one to remove it).
- char * MapFile(
  int openFileId, // file already open
  char * startAddress = 0, // 0: OS does it
  int startOffset = 0, // into the file
  int length = 0); // 0: whole file
- This call allows you to map a file in pieces
  – to save virtual address space

The MapFile system call



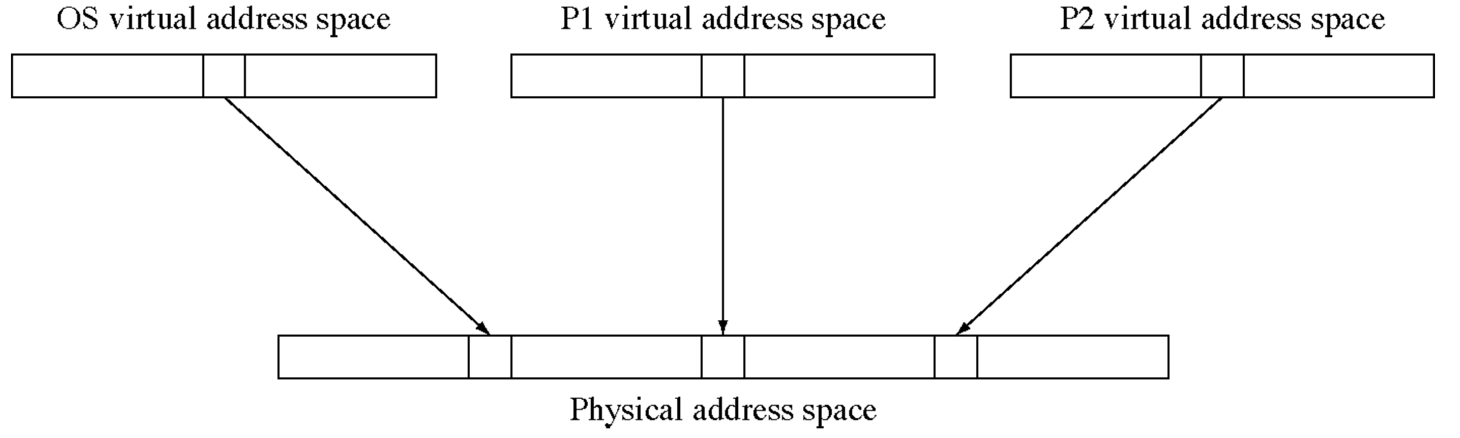
File mapping example code

- ```
  int CountLetter(
      char *fileName, char letter) {
    int fid = open(fileName, Reading);
    char * fileArea = MapFile(fid);
    int fileLength = GetFileLength(fid);
    int letterCount = 0;
    for( int i=0; i < fileLength; ++i) {
      if( fileArea[i] == letter )
        ++letterCount;
    }
    UnMapFile(fid);
    return letterCount;
  }
  ```
  ```

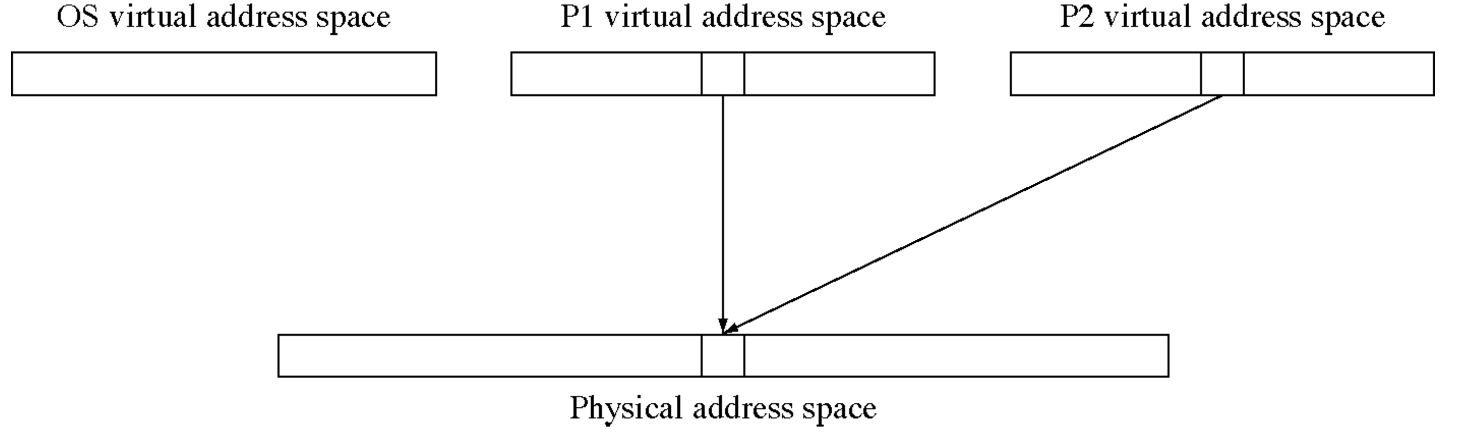
Advantages of file mapping

- Simpler OS interface: no explicit I/O
- More efficient: system calls are not required for reading and writing files
- Reduces the number of copies of file data in memory
- Almost all modern operating system use it

Multiple memory copies of file data



(a) Each process has a physical copy of the data



(b) The processes share a physical copy of the file data

Virtual memory in the IBM 801