Linux Process Management

Contents

- **This module covers**
  - What is Process?
  - Process Descriptor
  - The current macro
  - The process list
  - Process switching
  - Process Creation and Termination
  - Process Scheduling
  - Threads
  - Threads Vs Process
  - Signals
  - Sending and Caching Signals

What is Process?

- A process is usually defined as an instance of a program in execution; thus, if 16 users are running vi at once, there are 16 separate processes (although they can share the same executable code)
- Processes are often called "tasks" in Linux source code
- Process include the program code in the text section , data section containing the global variables, a set of resources such as open files and pending signals and one or more threads of execution

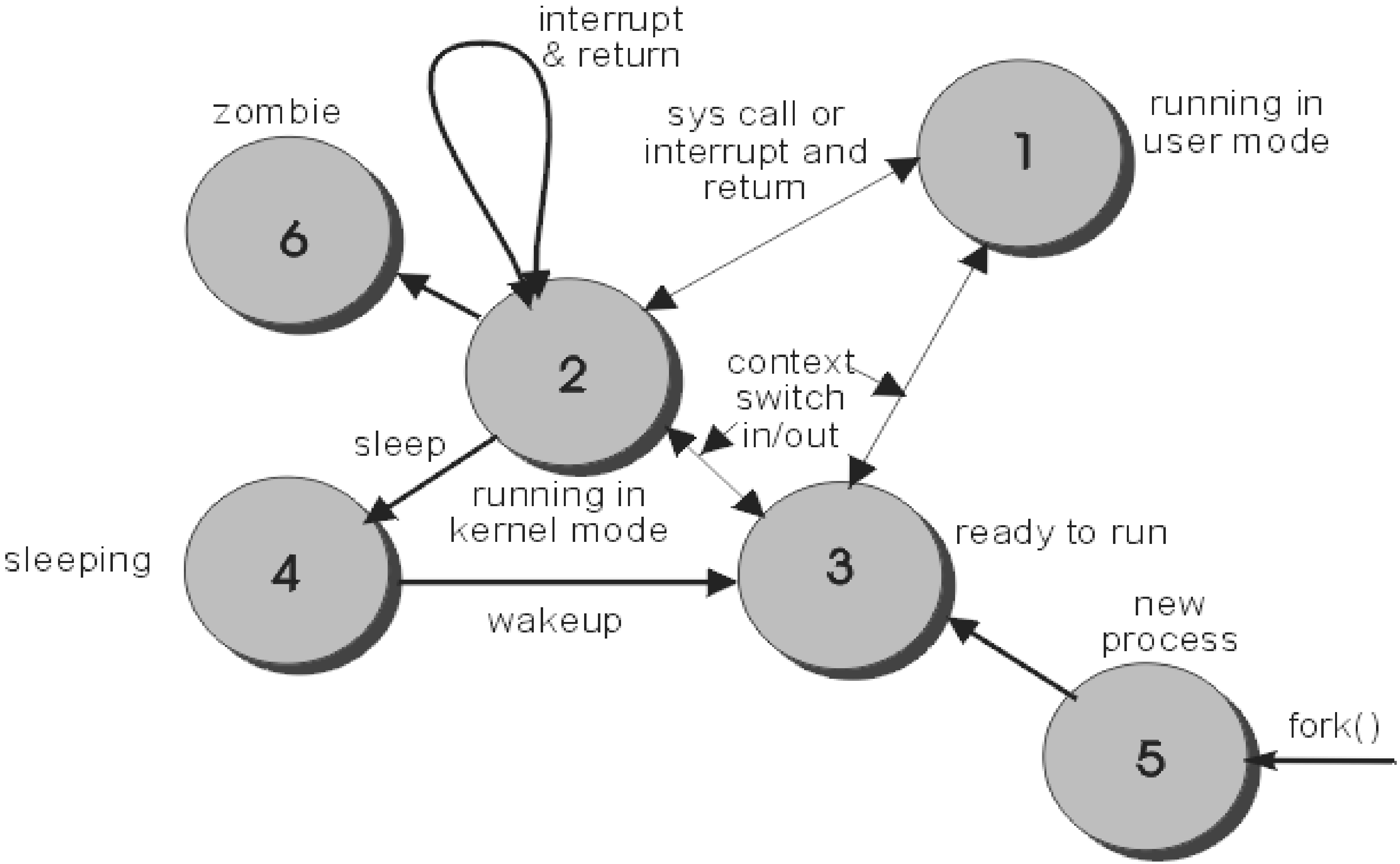
Process context

- **The context of a process is all of the characteristics, settings, values, etc., that a particular program uses as it runs, as well as those that it *needs* to run**
- **Even the internal state of the CPU and the contents of all its registers are part of the context of the process**
- **When a process has finished having its turn on the CPU and another process gets to run, the act of changing from one process to another is called a *context switch***

Process Table

- **It is a table containing information about all the processes on the system, whether that process is currently running or not**
- **The process' context is defined by two structures:**
  - its task structure and its *process table entry*
- **One piece of information that the process table entry (PTE) contains is the process' *Local Descriptor Table* (LDT)**
- **A descriptor is a data structure the process uses to gain access to different parts of the system. The descriptors are held in *descriptor tables***

The Life Cycle of Processes



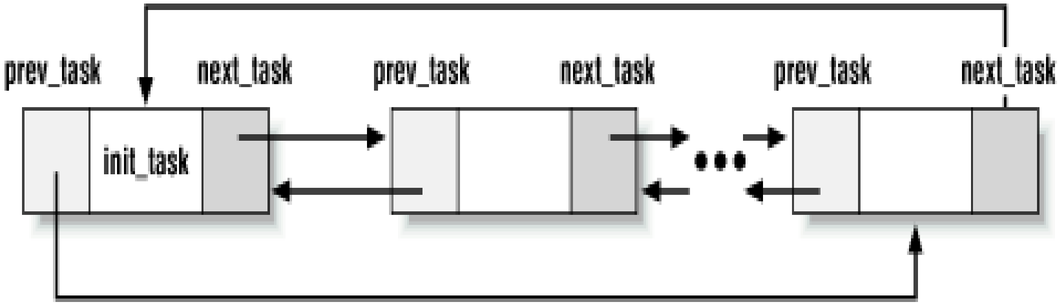
Identifying a Process

- **Every process has an entry in the process table**
- **The PID is a 32-bit unsigned integer stored in the pid field of the process descriptor**
- **The maximum PID number allowed on Linux is 32767**
- **When the kernel creates the 32768th process in the system, it must start recycling the lower unused PIDs**
- **Process ID:**

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
```

The process list

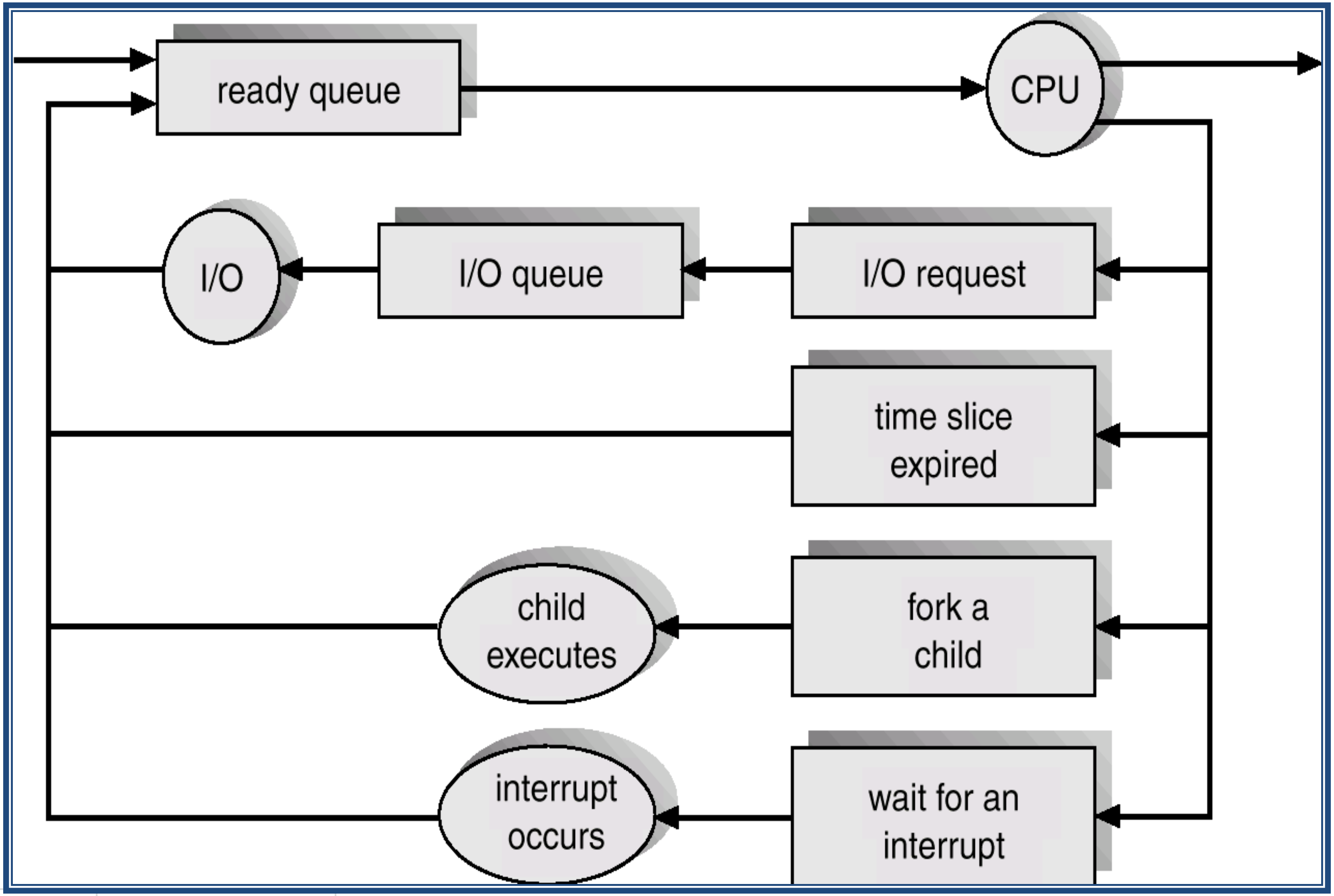
- **It is a circular doubly linked list links together all existing process descriptors;**
- **The prev_task and next_task fields of each process descriptor are used to implement the list.**
- **The head of the list is the init_task descriptor referenced by the first element of the task array: it is the ancestor of all processes, and it is called *process 0* or *swapper***



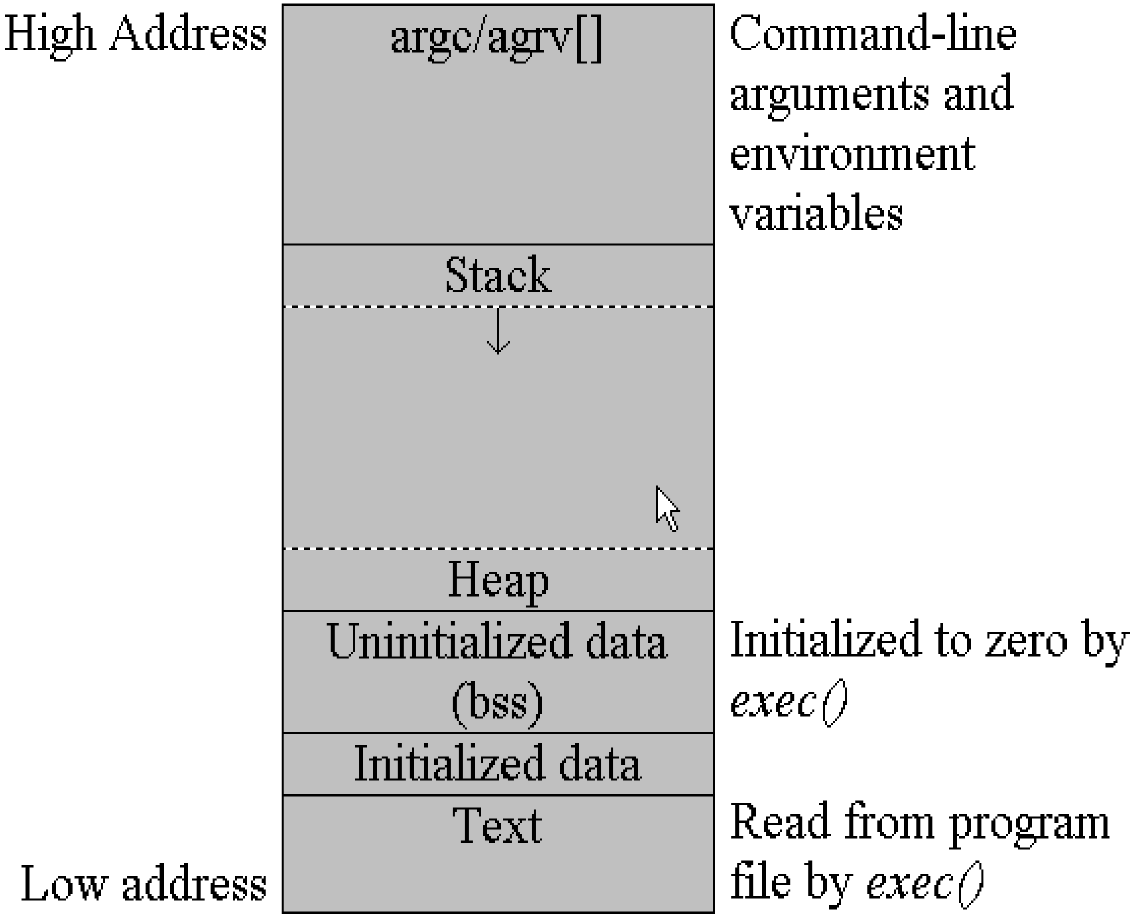
Process Scheduling

- **The set of rules used to determine when and how selecting a new process to run is called *scheduling policy***
- **Linux scheduling is based on the *time-sharing* technique**
  - Several processes are allowed to run "concurrently," which means that the CPU time is roughly divided into "slices," one for each runnable process.Of course, a single processor can run only one process at any given instant. If a currently running process is not terminated when its time slice or *quantum* expires, a process switch may take place.
- **Time-sharing relies on timer interrupts and is thus transparent to processes. No additional code needs to be inserted in the programs in order to ensure CPU time-sharing**

Queuing-Diagram of Process Scheduling



Typical Memory layout of process


High Address
argc/agrv[]
Command-line arguments and environment variables
Stack
Heap
Uninitialized data (bss)
Initialized to zero by *exec()*
Initialized data
Text
Read from program file by *exec()*
Low address

Typical logical memory layout of a process

Process Ids and init

- **Every process on the system has a parent, with the exception of pid 1: init**
  - the init process “hangs around”, it is responsible for the initialization and booting of the system, and for running any new programs, like the login program, and your shell
  - init executes /etc/rc* files during initialization, and is the ultimate parent of every subsequent process in the system
  - If init is killed, the system shuts down
- **Any process’s parent id (ppid) can be obtained with the pid_t getppid(void) call**

Orphans and Zombies

- **A *zombie* is a child process that that has exited *before it's parent's has called wait*() for the child's exit status**
- **A zombie holds nothing but the child's exit status (held in the program control block)**
- **Modern systems have init (pid == 1) *adopt* zombies after their parents die, so that zombies do not hang around forever as they used to, in case the parent never did get around to calling wait**
- **If a parent process dies before it's child, the child process becomes an orphan**
  - An orphan is a child process whose parent is no longer living
  - An *orphan* is immediately "adopted" by the init process (pid == 1), who will call wait() on behalf of the deceased parent when the child dies

vfork() and fork()

- **vfork has the same calling sequence and same return values as fork. But different semantics**
- **vfork is intended to create a new process when the purpose of the new process to exec a new program**
- **vfork creates the new process just like fork, without copying the address space of parent into the child since the child won’t reference that address space**
- **Instead, while the child is running, until it calls either exec or exit, the child runs in the address space of parent**
- **vfork guarantees that the child runs first, until the childs calls exec or exit. When the child calls one of these functions, the parents resumes**

Threads

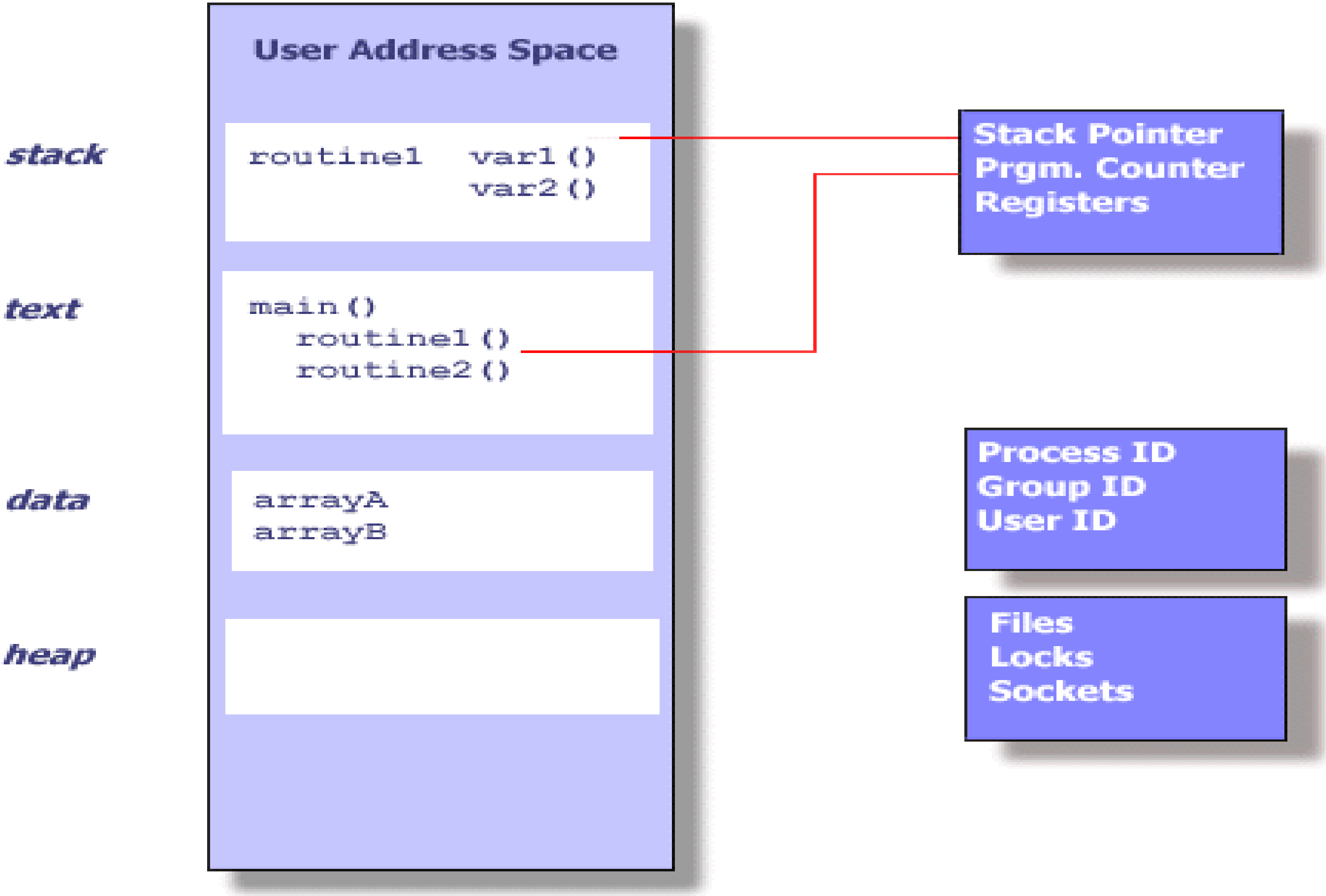
Table of Contents

- **What is a Threads?**
- **Designing Threaded Programs**
- **Synchronizing Threads**
- **Managing Threads**

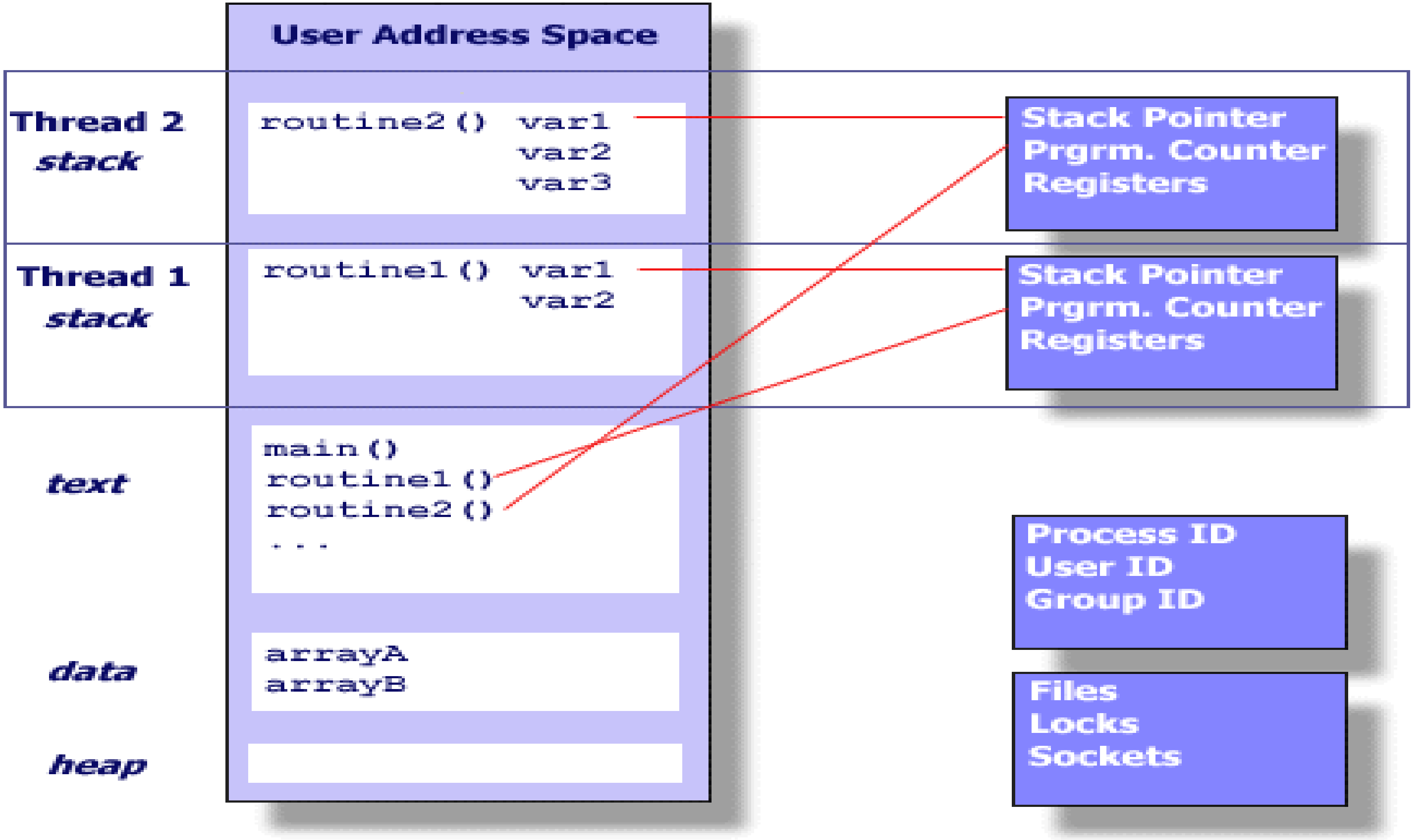
What is a Thread?

- **Relationship between a process and a thread.**
- **A process is created by the operating system. Processes contain information about program resources and program execution state, including:**
  - Process ID, process group ID, user ID, and group ID
  - Environment
  - Working directory.
  - Program instructions
  - Registers
  - Stack,Heap
  - File descriptors
  - Signal actions
  - Shared libraries
  - Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

A Process


User Address Space
stack
routine1 var1()
var2()
text
main()
routine1()
routine2()
data
arrayA
arrayB
heap
Stack Pointer
Prgm. Counter
Registers
Process ID
Group ID
User ID
Files
Locks
Sockets

Threads


User Address Space
Thread 2
stack
routine2() var1
var2
var3
Stack Pointer
Prgrm. Counter
Registers
Thread 1
stack
routine1() var1
var2
Stack Pointer
Prgrm. Counter
Registers
text
main()
routine1()
routine2()
. . .
data
arrayA
arrayB
heap
Process ID
User ID
Group ID
Files
Locks
Sockets

Linux: Processes or Threads?

- **Linux uses a neutral term: tasks**
- **Traditional view**
  - Threads exist "inside" processes
- **Linux view**
  - Threads: processes that share address space
  - Linux "threads" (tasks) are really "kernel threads"

What are Pthreads?

- **IEEE POSIX Section 1003.1c**
  - IEEE (Institute of Electric and Electronic Engineering)
  - POSIX (Portable Operating System Interface)
  - Pthread is a standardized model for dividing a program into subtasks whose execution can be interleaved or run in parallel
    - Specifics are different for each implementation
    - Mach Threads and NT Threads
- **Pthread of Linux is kernel level thread**
  - Implemented by clone() systemcall

The Pthreads API

- **The Pthreads API is defined in the ANSI/IEEE POSIX 1003.1 - 1995 standard.**
- **The subroutines which comprise the Pthreads API can be informally grouped into three major classes:**
  1. ***Thread management:*** The first class of functions work directly on threads - creating, detaching, joining, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)
  2. ***Mutexes:*** The second class of functions deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
  3. ***Condition variables:*** The third class of functions address communications between threads that share a mutex. They are based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

Creating Threads

- **Initially, your main() program comprises a single, default thread. All other threads must be explicitly created by the programmer.**
- **Routines:**

  pthread_create (thread,attr,start_routine,arg)

  - Creates a new thread. Once created, threads are peers, and may create other threads.
- **Returns:**
  - the new thread ID via the *thread* argument. This ID should be checked to ensure that the thread was successfully created.

Creating Threads

- ***attr* :**
  - is used to set thread attributes. You can specify a thread attributes object, or NULL for the default values. Thread attributes are discussed later.
- ***start_routine:***
  - is the C routine that the thread will execute once it is created.
- **arg :**
  - An argument may be passed to *start_routine* via *arg*. It must be passed by reference as a pointer cast of type void.
- **The maximum number of threads that may be created by a process is implementation dependent.**

Terminating Thread Execution

- **There are several ways in which a Pthread may be terminated:**
  - The thread returns from its starting function
  - The thread makes a call to the pthread_exit function
  - The thread is canceled by another thread via the pthread_cancel function
  - The entire process is terminated due to a call to either the exec or exit

- **Routines: pthread_exit (status)**

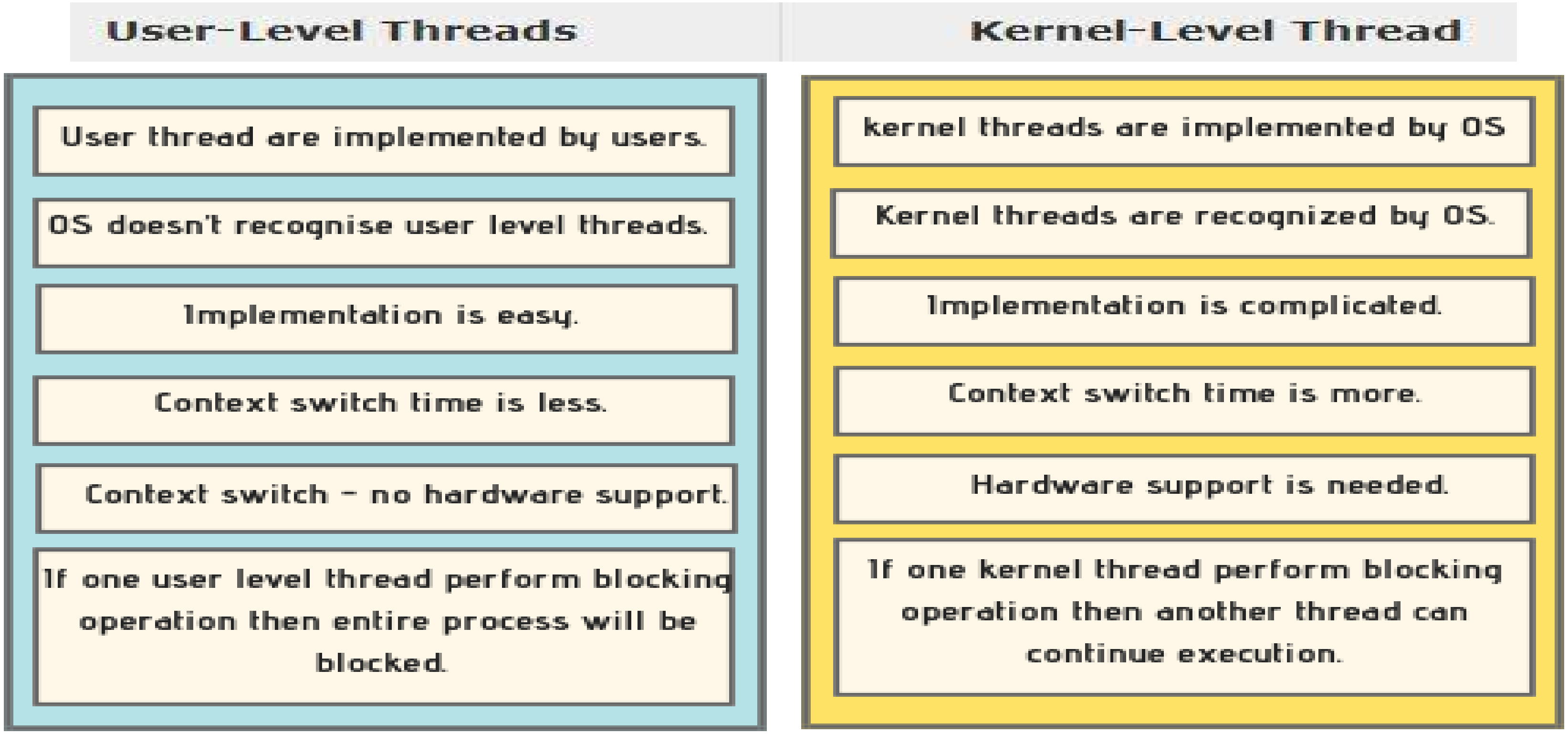
Pthread Creation and Termination

```
#include <pthread.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{      printf("\n%d: Hello World!\n", threadid);
       pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
pthread_t threads[NUM_THREADS];
int rc, t;
       for(t=0;t < NUM_THREADS;t++) {
                  printf("Creating thread %d\n", t);
                  rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
                  if (rc) {
                    printf("ERROR; return code from pthread_create() is %d\n", rc); exit(-1);
                  }
       }
       pthread_exit(NULL);
}
```

Passing Arguments to Threads

- **The pthread_create() routine permits the programmer to pass one argument to the thread start routine. For cases where multiple arguments must be passed, this limitation is easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the pthread_create() routine.**
- **All arguments must be passed by reference and cast to (void *).**

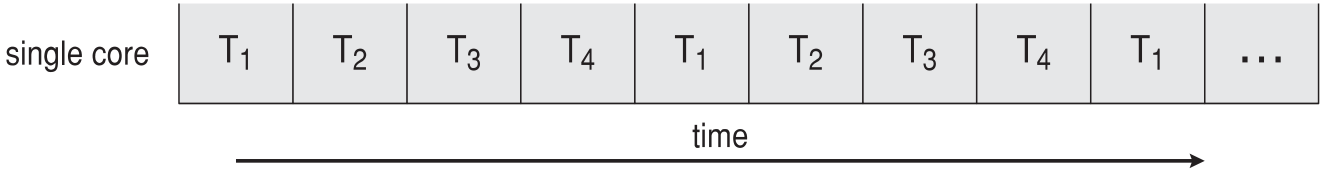
Types of Thread



User-Level Threads	Kernel-Level Thread
User thread are implemented by users.	kernel threads are implemented by OS
OS doesn't recognise user level threads.	Kernel threads are recognized by OS.
Implementation is easy.	Implementation is complicated.
Context switch time is less.	Context switch time is more.
Context switch - no hardware support.	Hardware support is needed.
If one user level thread perform blocking operation then entire process will be blocked.	If one kernel thread perform blocking operation then another thread can continue execution.

Concurrency vs. Parallelism

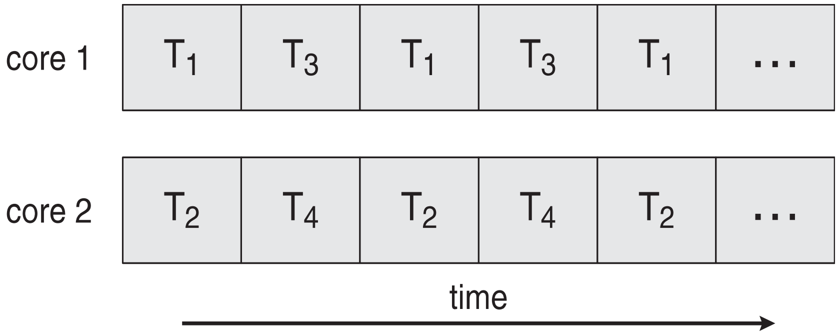
- **Concurrent execution on single-core system:**



single core T_1 T_2 T_3 T_4 T_1 T_2 T_3 T_4 T_1 . . .

time

- **Parallelism on a multi-core system:**

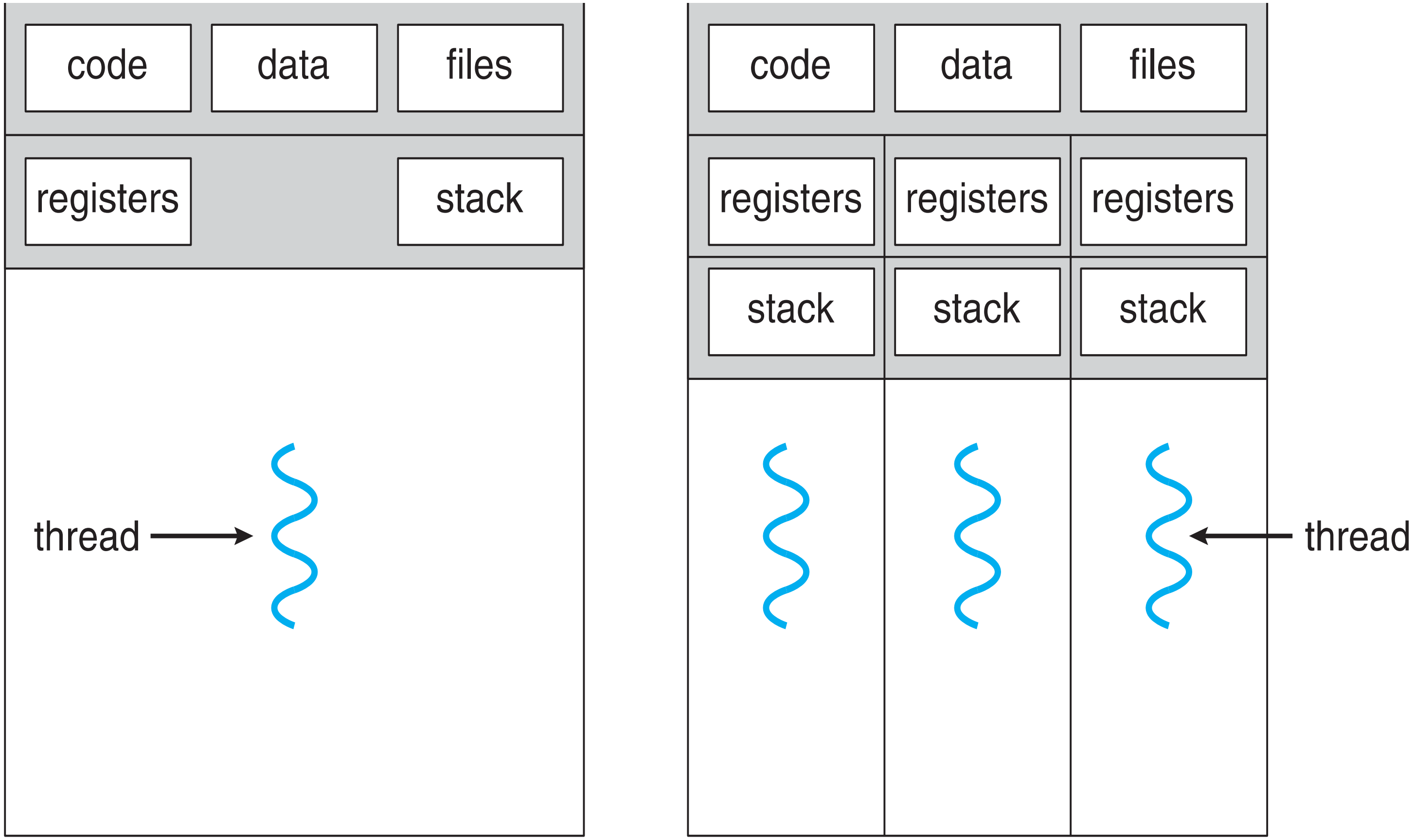


core 1 T_1 T_3 T_1 T_3 T_1 . . .

core 2 T_2 T_4 T_2 T_4 T_2 . . .

time

Single and Multithreaded Processes



single-threaded process

multithreaded process

Amdahl’s Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

- But does the law take into account contemporary multicore systems?

User Threads and Kernel Threads

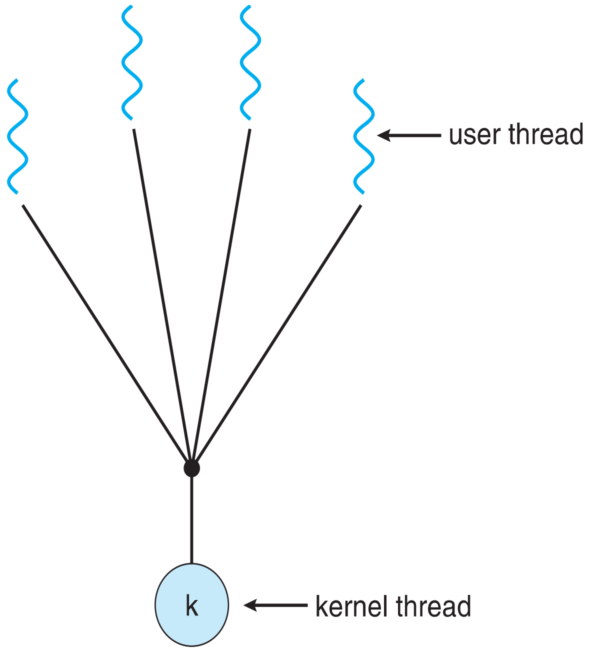
- **User threads - management done by user-level threads library**
- **Three primary thread libraries:**
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads - Supported by the Kernel**
- **Examples – virtually all general purpose operating systems, including:**
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

- **Many-to-One**
- **One-to-One**
- **Many-to-Many**

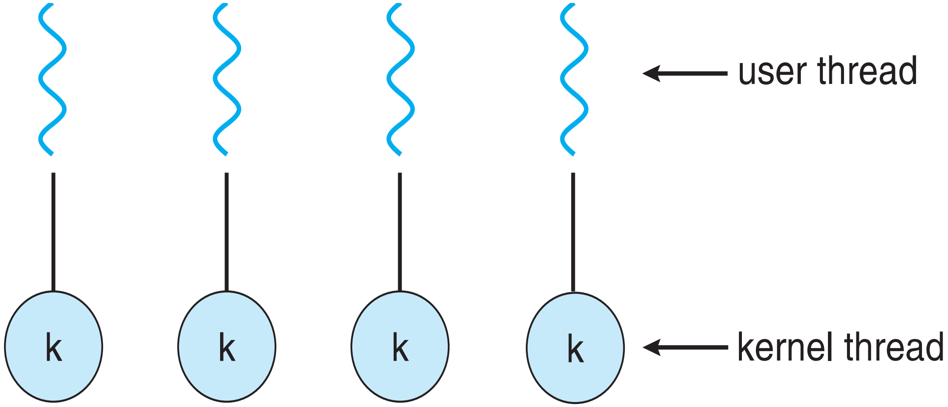
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



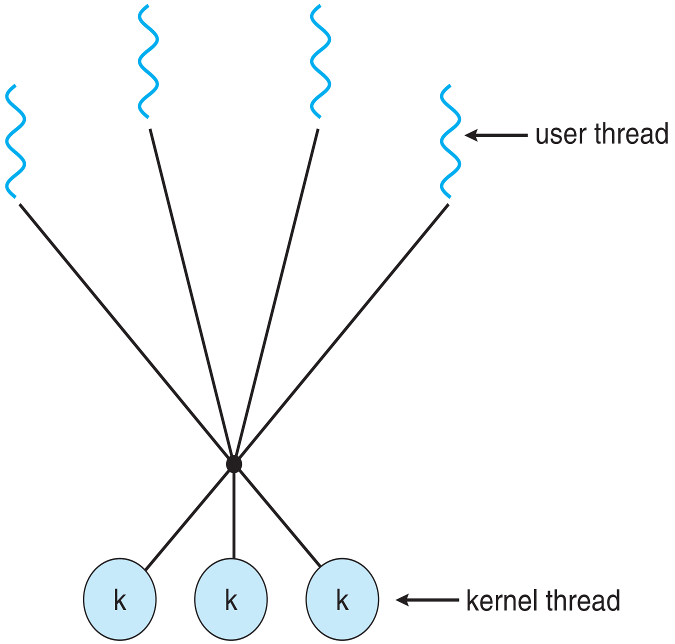
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



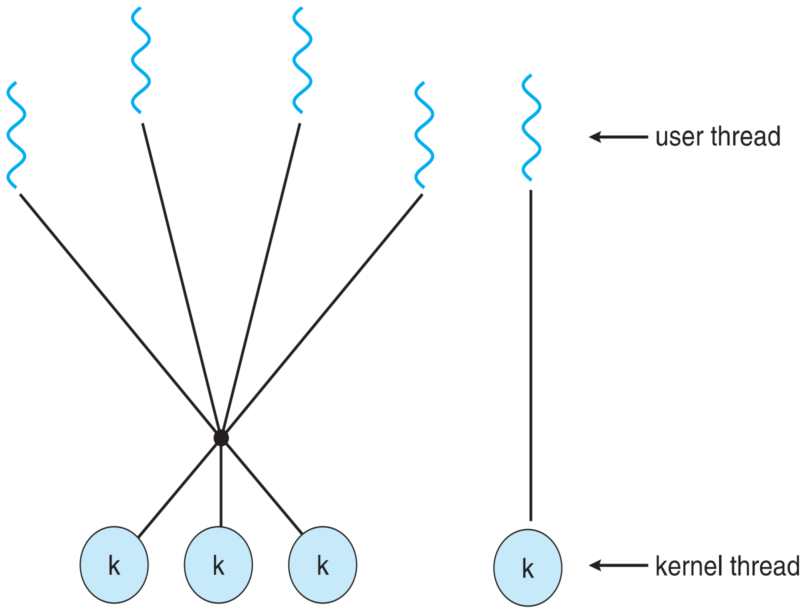
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



Two-level Model

- **Similar to M:M, except that it allows a user thread to be bound to kernel thread**
- **Examples**
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



Thread Libraries

- **Thread library provides programmer with API for creating and managing threads**
- **Two primary ways of implementing**
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- *Specification*, not *implementation*
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Semaphore

- Semaphore is a signaling mechanism.
- Semaphore is an integer variable.
- Semaphore allows multiple program threads to access a finite instance of resources.
- Semaphore value can be changed by any process acquiring or releasing the resource.
- Semaphore can be categorized into counting semaphore and binary semaphore. Mutex is not categorized further.
- Semaphore value is modified using wait() and signal() operation.
- If all resources are being used, the process requesting for resource performs wait() operation and block itself till semaphore count become greater than one.

Mutex

- Mutex is a locking mechanism.
- Mutex is an object.
- Mutex allow multiple program thread to access a single resource but not simultaneously.
- Mutex object lock is released only by the process that has acquired the lock on it.
- Mutex is not categorized further.
- Mutex object is locked or unlocked by the process requesting or releasing the resource.
- If a mutex object is already locked, the process requesting for resources waits and queued by the system till lock is released.

Mutex and semaphore

- **Mutex is object owned by thread.**
- **There is ownership in mutex**
- **Mutex allows only one thread to access**
- **Semaphore**
  - It is a mechanism to limit the access to n thread for the resource there is no ownership of the resource
  - If you want to give controlled access to multiple resources use semaphore
  - E.g trial rooms in malls on sundays
  - Useful for ordering multiple processes
  - For managing resources e.g 5 printers then set s=5
  - P(s) While(s<=0); s=s-1;
  - V(s) s=s+1

Semaphore

```
do{

p(s)
//critical section
v(s)
//remaining section
}
```

```
P(s){

While(s<=0);
s=s-1;
}
```

```
V(s){
S=s+1;


}
```

- **Demo**
  - Environment variable

Linux Memory Management

Contents

- **Module Coverage**
  - Virtual Memory
  - Memory Addressing
  - Paging in Hardware
  - Memory Area Management
  - Memory Management functions

Memory Management Basics

- **Bit**
  - Bit is a unit of memory
- **Word**
  - Smallest addressable unit of processor
- **MMU(Memory Management Unit)**
  - It is the hardware that manages memory and performs virtual to physical address translations
- **Page**
  - Basic unit of memory management

Background

- **Program must be brought into memory and placed within a process for it to be run.**
- **Input queue – collection of processes on the disk that are waiting to be brought into memory to run the program.**

Virtual Memory

- **Virtual Memory**
  - Technique that allows the execution of processes that may not be completely in memory.
  - Programs can be larger than physical memory
- **Goals**
  - Allow virtual address spaces that are larger than the physical address space
- **Methods**
    - Allow pages from the virtual address space to be stored in secondary memory, as well as primary memory
    - Move pages between secondary and primary memory so that they are in primary memory when they are needed

- Demand Paging
- Paging is a memory allocation strategy by transferring a fixed-sized unit of the virtual address space called virtual page whenever the page is needed to execute a program. As the size of frames and pages are the same, any logical page can be placed in any physical frame of memory.
- Every processes will be logical divided and allocate in the virtual address space. There is a page table in the virtual memory to allocate and keep tracking of the pages to map into the frames.
- Demand paging decreases the paging time and physical memory needed because only the needed pages will be paged and the reading time of the unused pages can be avoided.

- Page Fault Problem
- A page fault occurs when a program try to use a page that is not in the memory, due to demand paging will only paged the pages into the memory when it is needed. For example in figure 1.9, if the program try to use Page 1 for Process A in memory, the operating system will interrupt occurs as a result of trying access a missing page because Page 1 is not paged in the memory.

- Problem- No Free Frames
- When all the frames in the memory is been used, the other problem will occurs. This will cause the pages is unable to paged into the memory.

- Solution- Page Replacement Algorithms

- Solution for no free frames problem is to
- find a memory frame that is idle and free
- the frame using a page replacement algorithm.
- There are three common types of page replacement algorithm
- such as First in First out (FIFO)
- Optimal - the page which is farthest in future requests
- Least Recently Used (LRU).

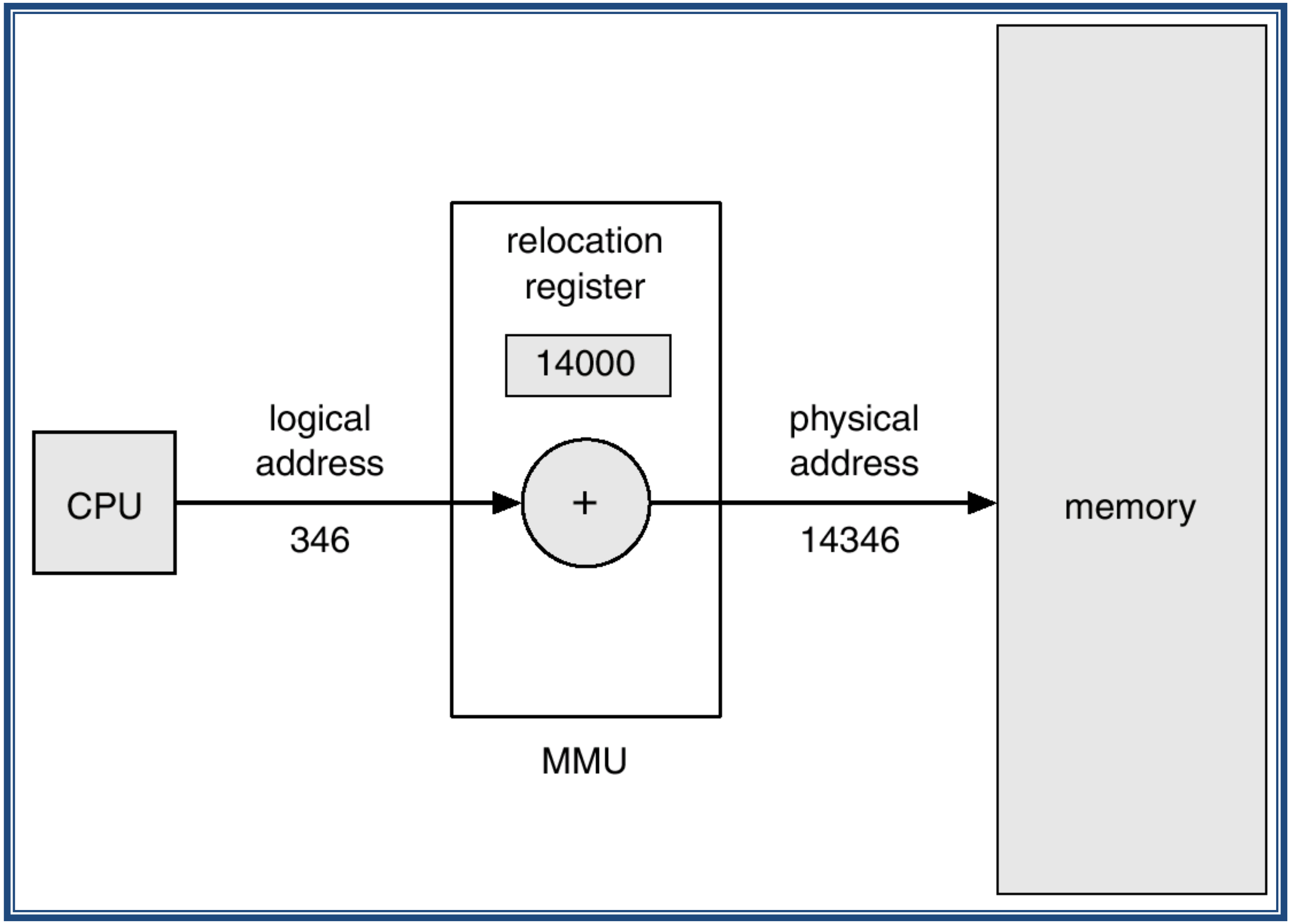
Logical vs. Physical Address Space

- **The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.**
  - *Logical address* – generated by the CPU; also referred to as *virtual address*.
  - *Physical address* – address seen by the memory unit.
- **Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.**

Memory-Management Unit (MMU)

- **Hardware device that maps virtual to physical address.**
- **In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.**
- **The user program deals with *logical* addresses; it never sees the *real* physical addresses.**

Dynamic relocation using a relocation register



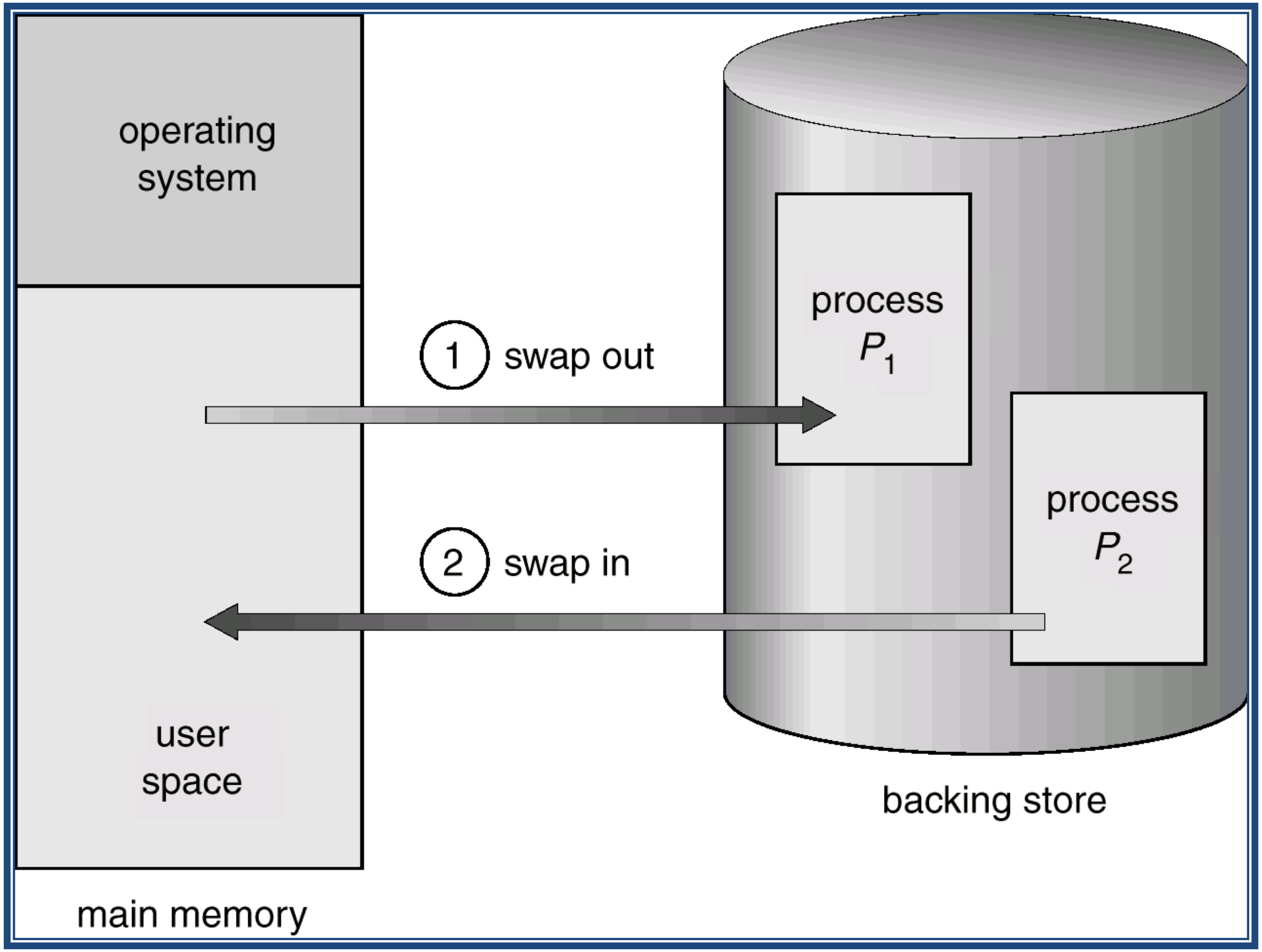
Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases.
- No special support from the operating system is required implemented through program design.

Swapping

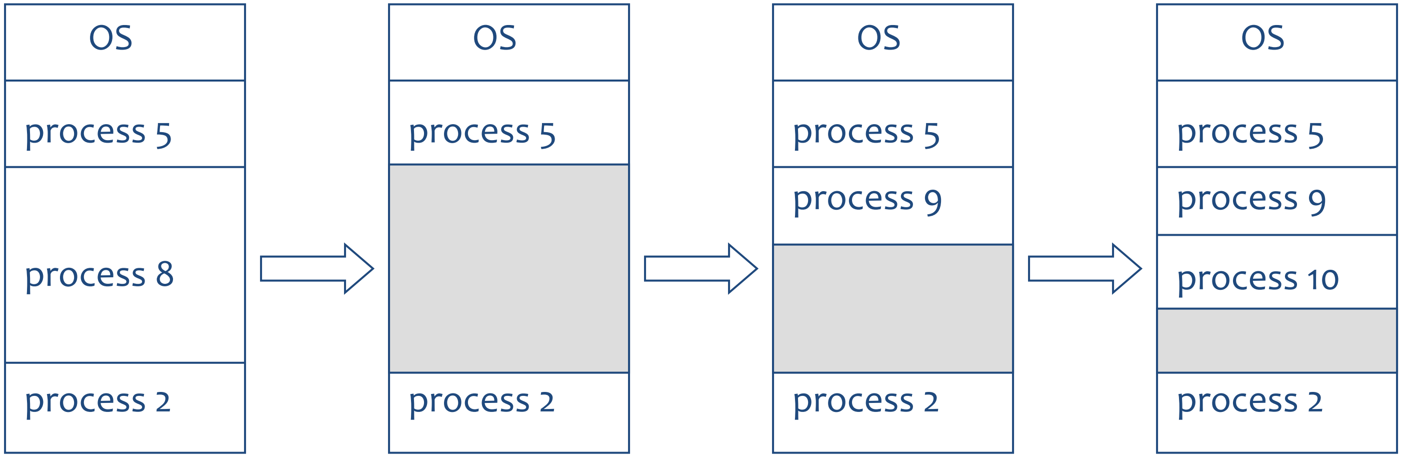
- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- Roll out, roll in – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.

Schematic View of Swapping



Contiguous Allocation

- **Multiple-partition allocation**
  - *Hole* – block of available memory; holes of various size are scattered throughout memory.
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it.
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

- How to satisfy a request of size n from a list of free holes.
- First-fit: Allocate the *first* hole that is big enough.
- Best-fit: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- Worst-fit: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization.

Fragmentation

- **External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous.**
- **Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.**
- **Reduce external fragmentation by compaction**
  - Shuffle memory contents to place all free memory together in one large block.
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time.

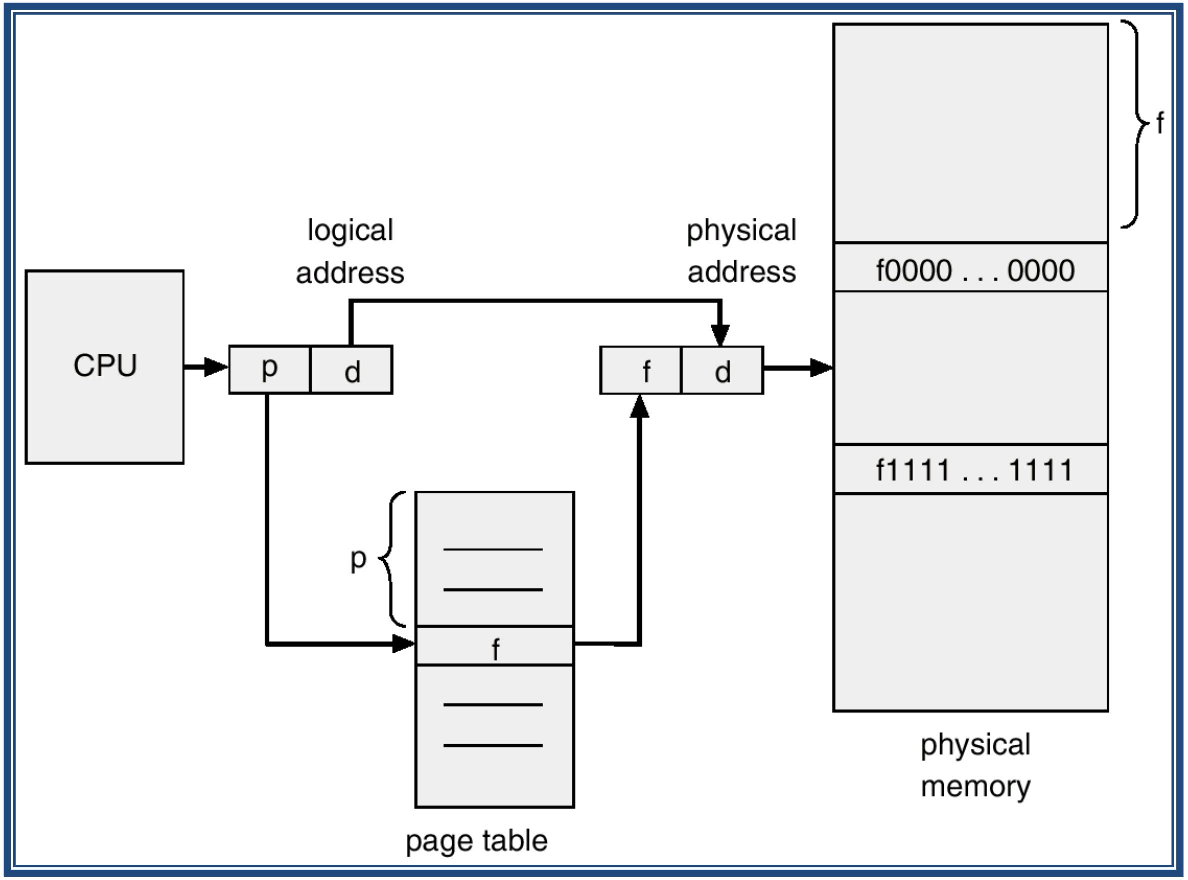
Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.
- Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 8192 bytes).
- Divide logical memory into blocks of same size called pages.
- Keep track of all free frames.
- To run a program of size *n* pages, need to find *n* free frames and load program.
- Set up a page table to translate logical to physical addresses.

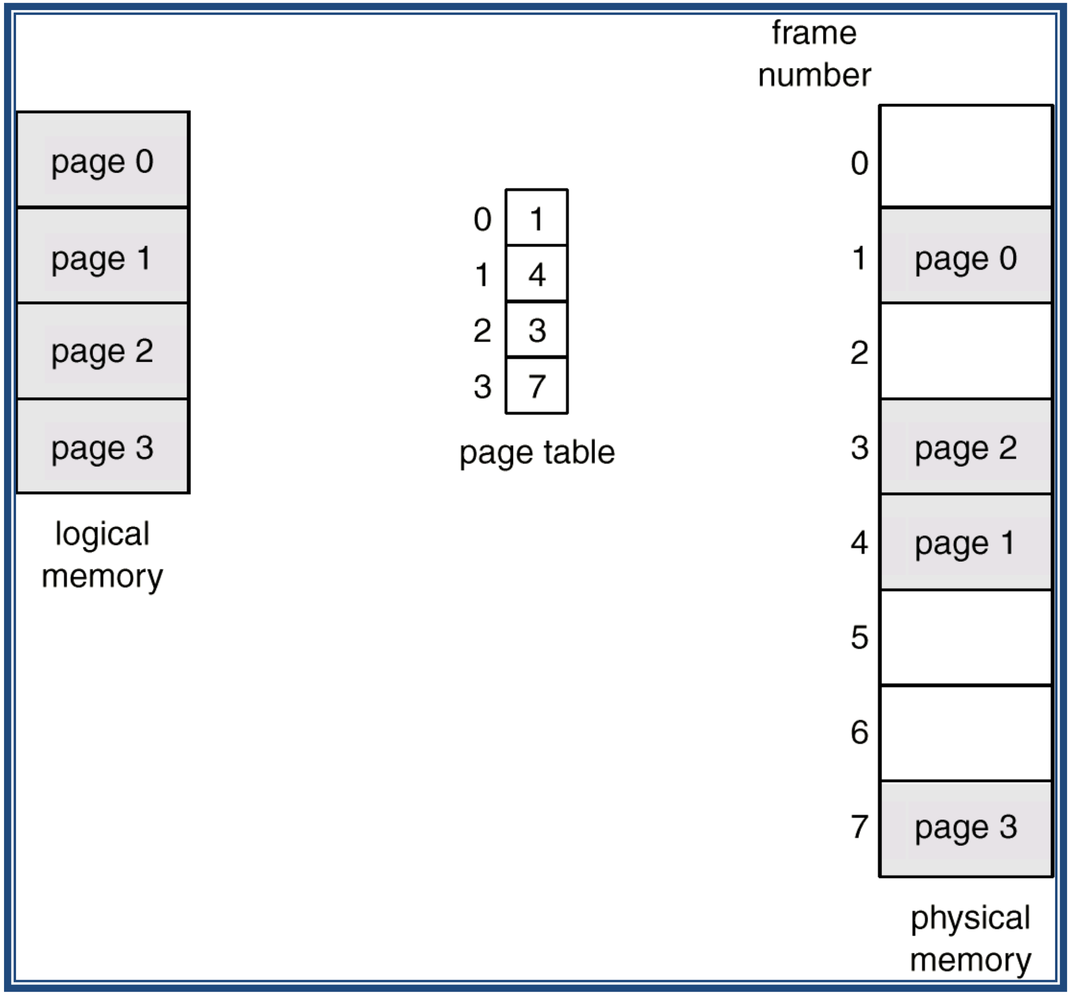
Address Translation Scheme

- **Address generated by CPU is divided into:**
  - *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory.
  - *Page offset (d)* – combined with base address to define the physical memory address that is sent to the memory unit.

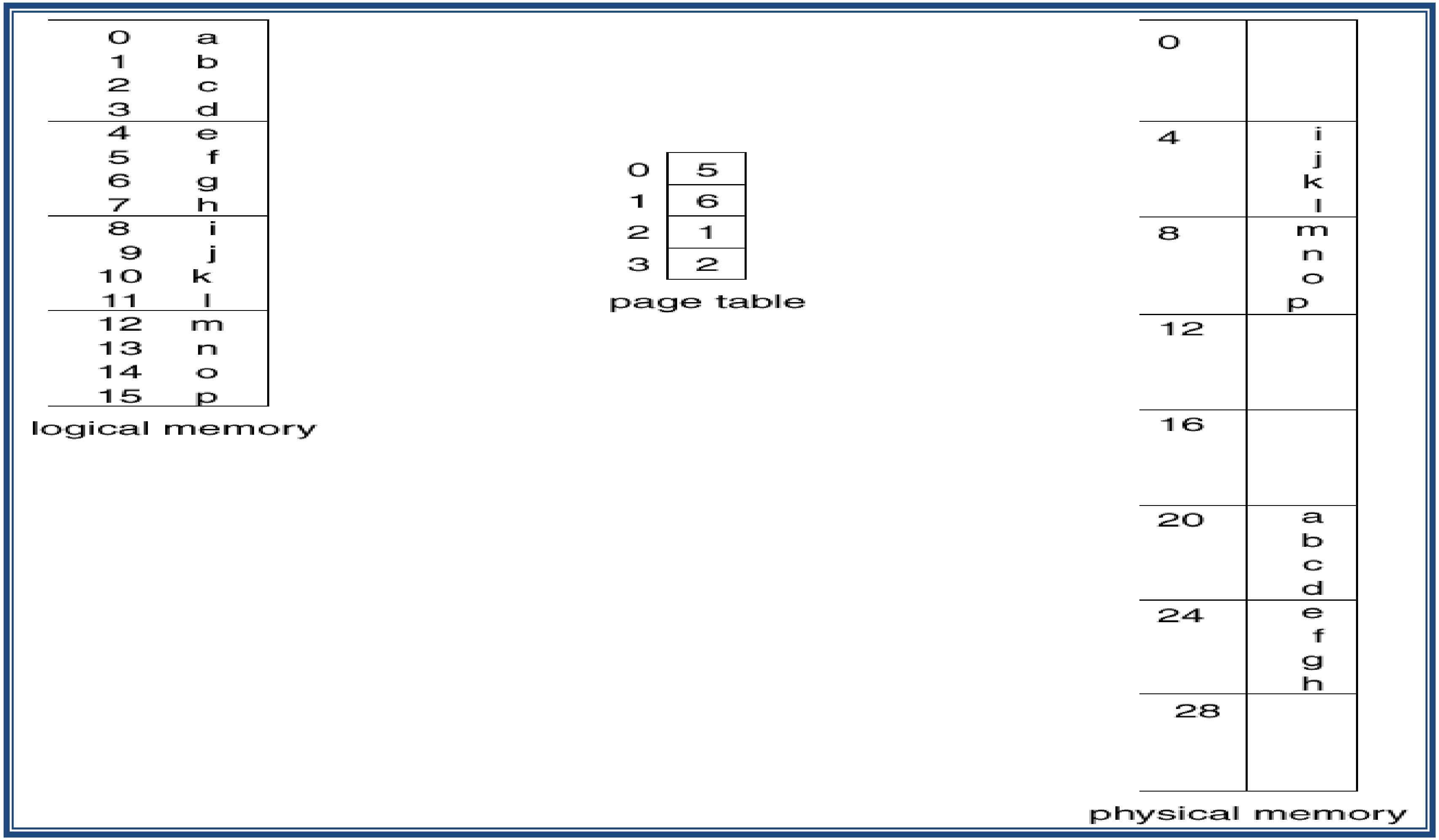
Address Translation Architecture



Paging Example



Paging Example



0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

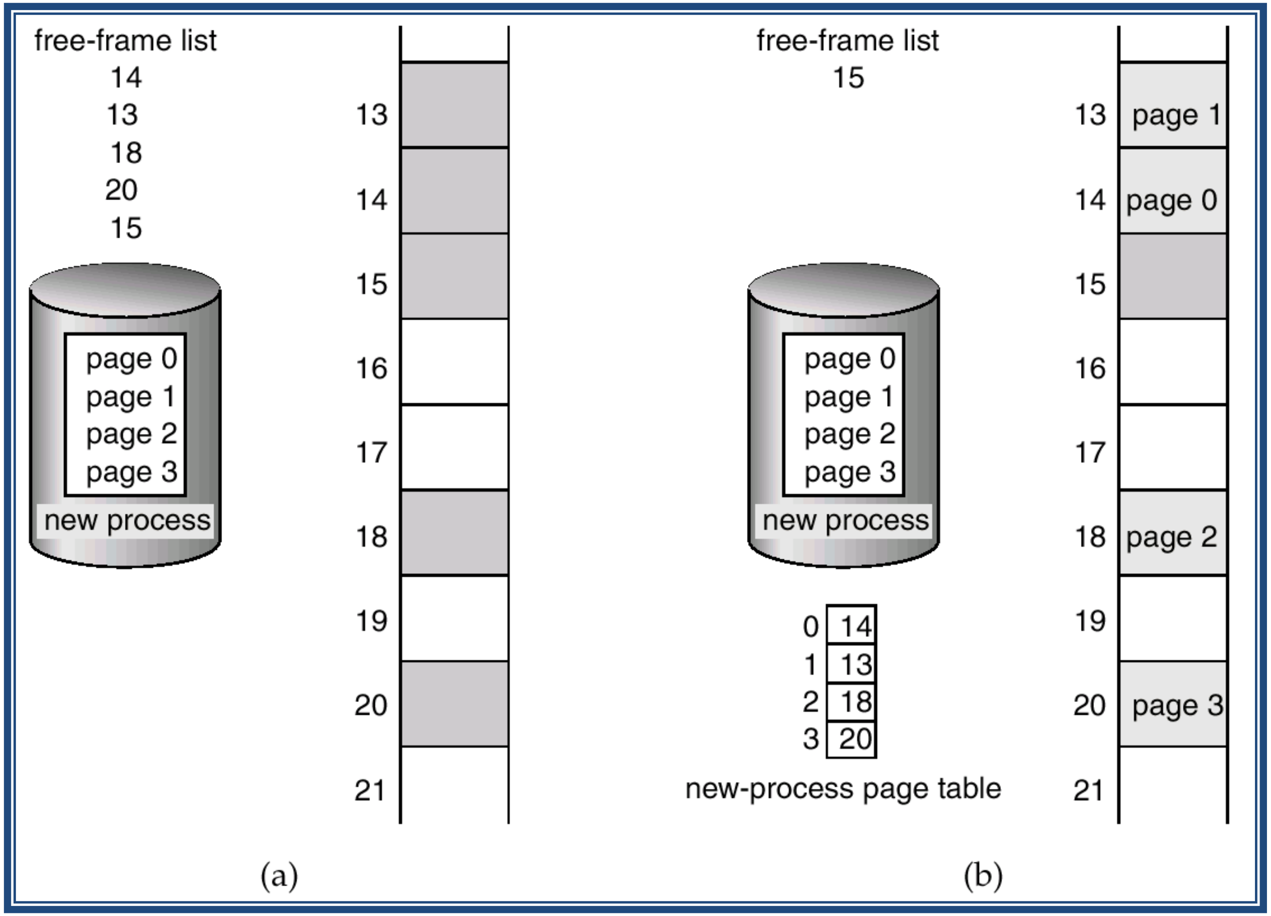
0	5
1	6
2	1
3	2

page table

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

physical memory

Free Frames



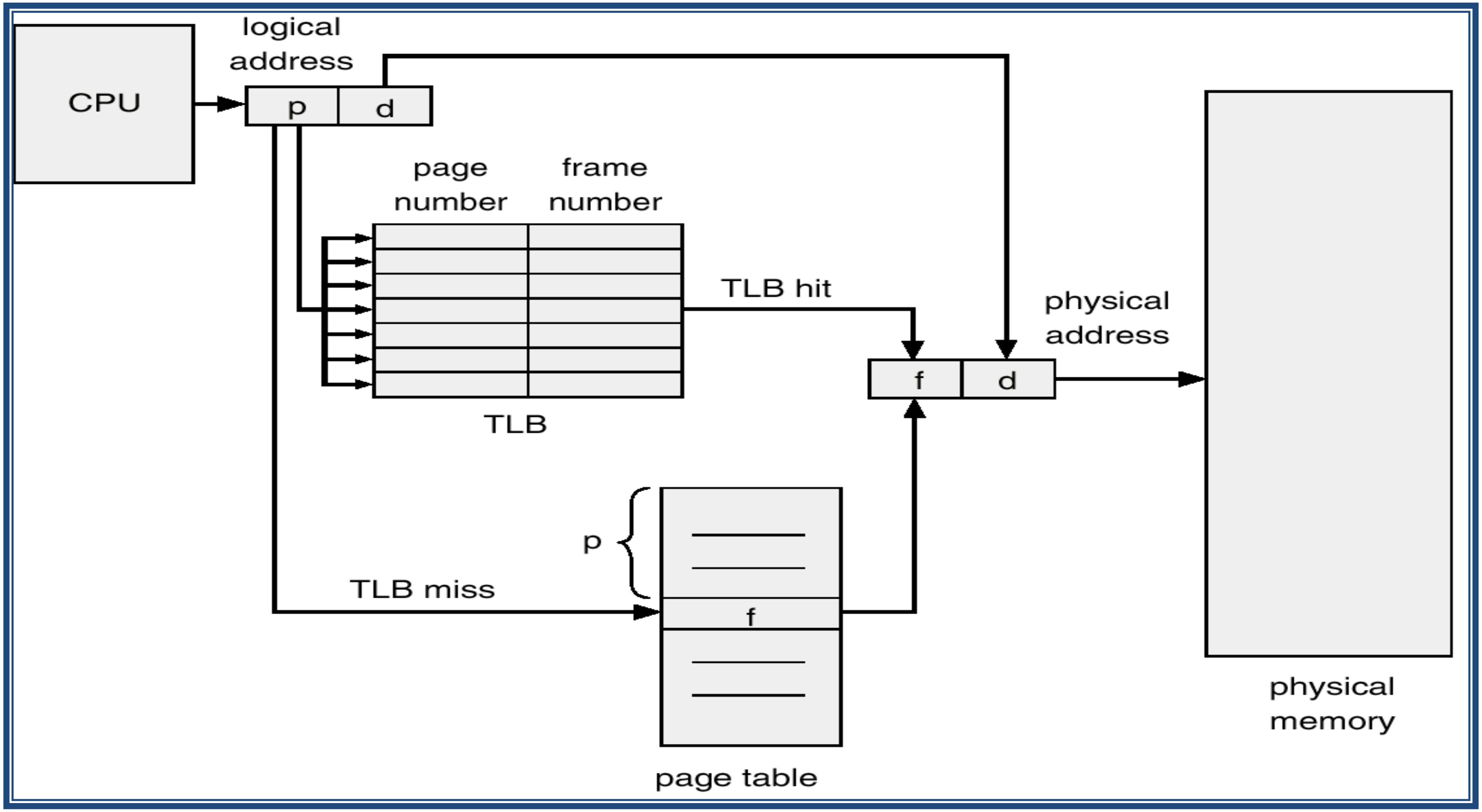
Before allocation

After allocation

Implementation of Page Table

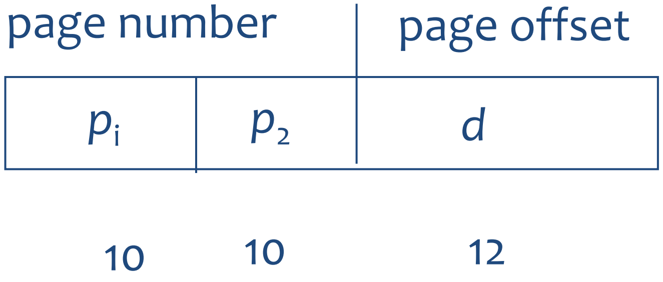
- Page table is kept in main memory.
- *Page-table base register* (PTBR) points to the page table.
- *Page-table length register* (PRLR) indicates size of the page table.
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called *associative memory* or *translation look-aside buffers (TLBs)*

Paging Hardware With TLB



Two-Level Paging Example

- **A logical address (on 32-bit machine with 4K page size) is divided into:**
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- **Since the page table is paged, the page number is further divided into:**
  - a 10-bit page number.
  - a 10-bit page offset.
- **Thus, a logical address is as follows:**



page number

page offset

p_i

p_2

d

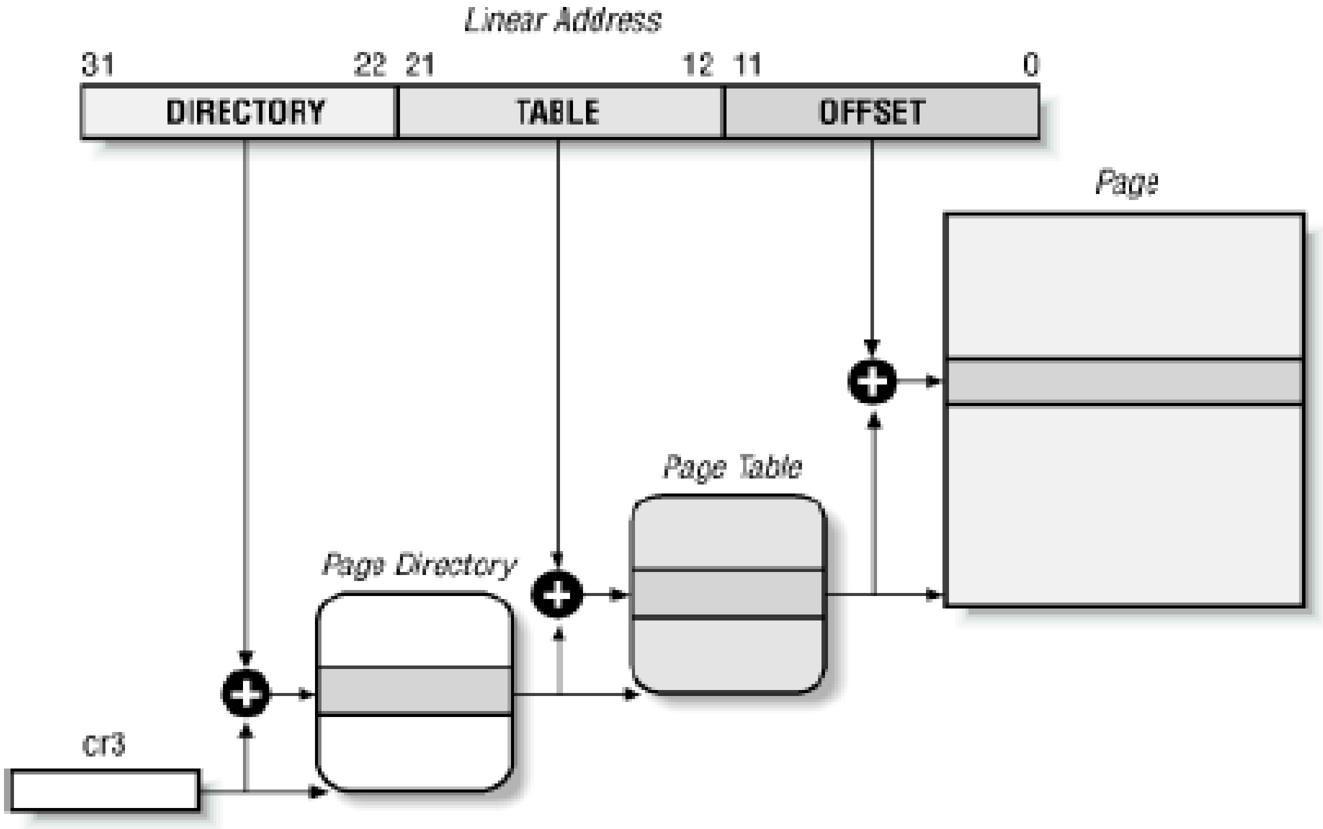
10

10

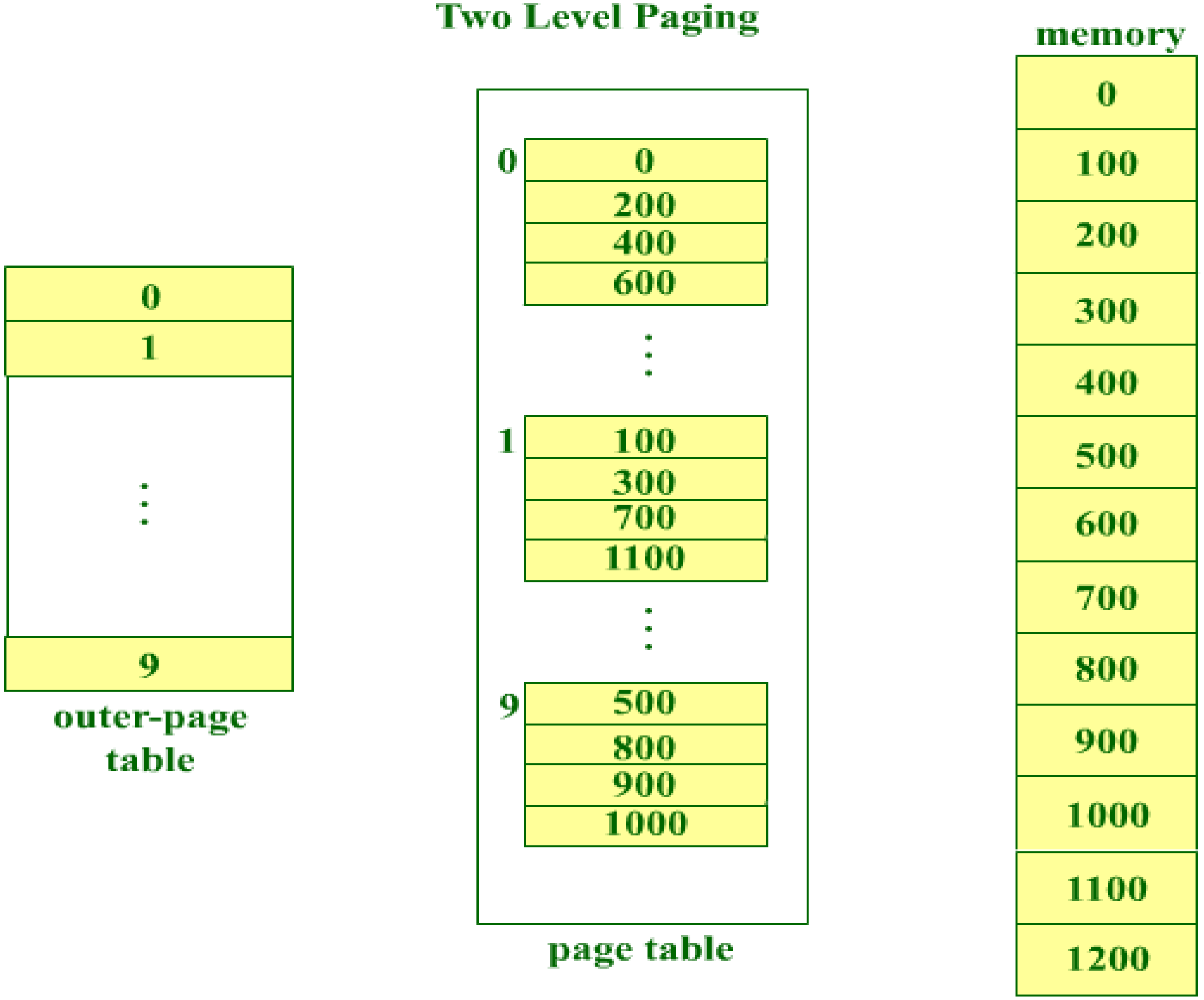
12

- **where pi is an index into the outer page table, and p2 is the displacement within the page of the outer page table.**

Two-Level Page-Table Scheme

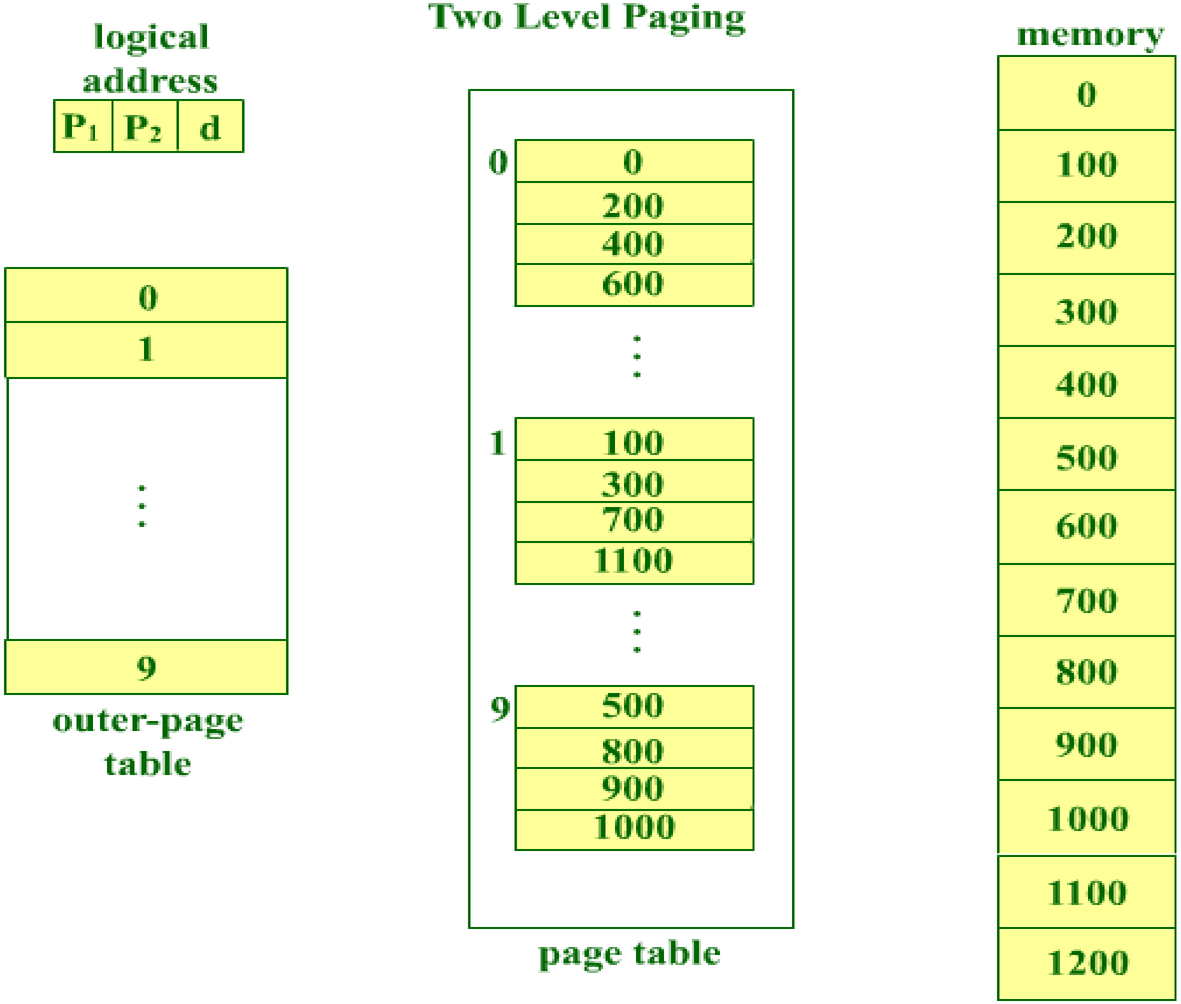


Two Level Paging



In a two level page table, the page table itself is divided into pages.

Two Level Paging



logical address

P1 | P2 | d

outer-page table

0
1
⋮
9

page table

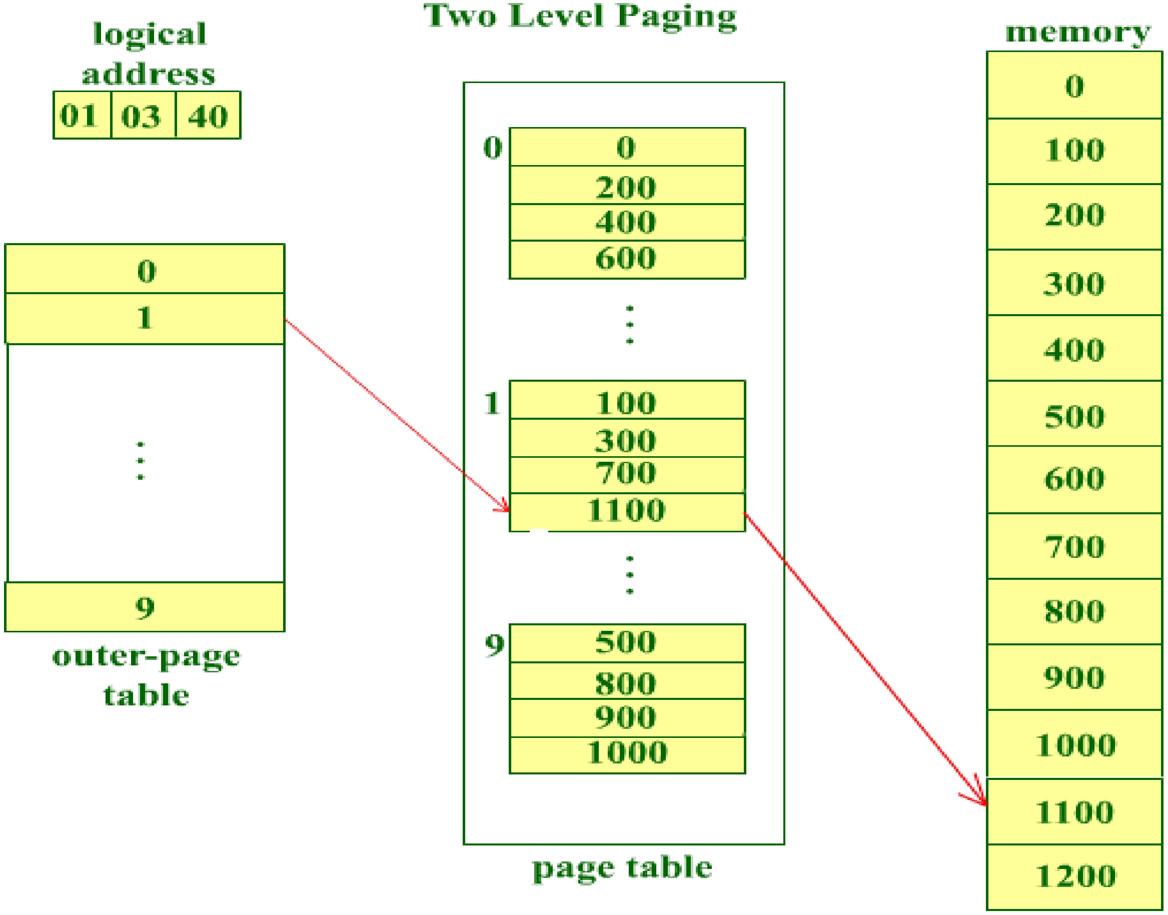
0: 0, 200, 400, 600
⋮
1: 100, 300, 700, 1100
⋮
9: 500, 800, 900, 1000

memory

0
100
200
300
400
500
600
700
800
900
1000
1100
1200

A logical address is divided into an outer page number (P_1), a page displacement within the page of the outer page (P_2), and a page offset (d).

Two Level Paging



logical address

01 | 03 | 40

0
1
⋮
9

outer-page table

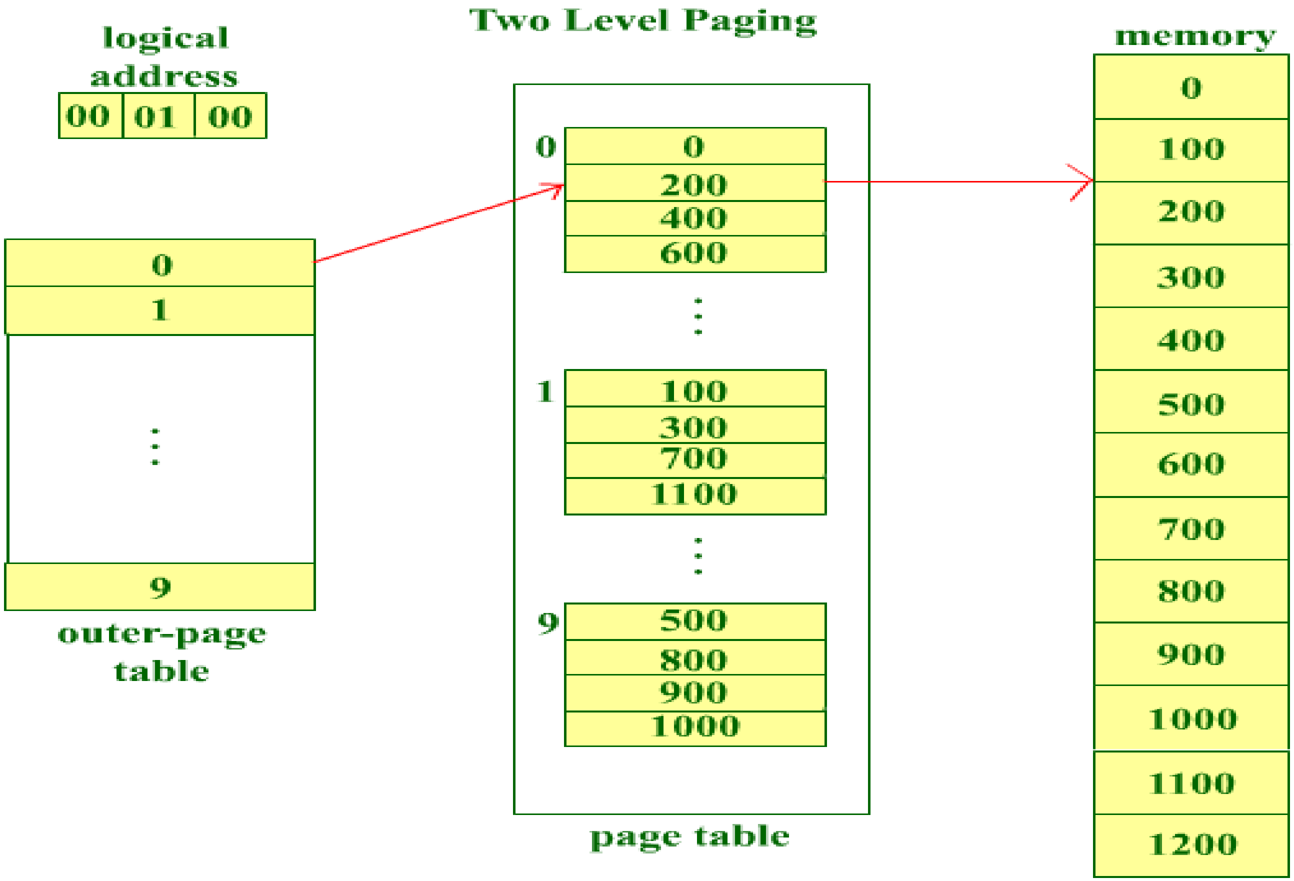
0: 0, 200, 400, 600
⋮
1: 100, 300, 700, 1100
⋮
9: 500, 800, 900, 1000

page table

memory

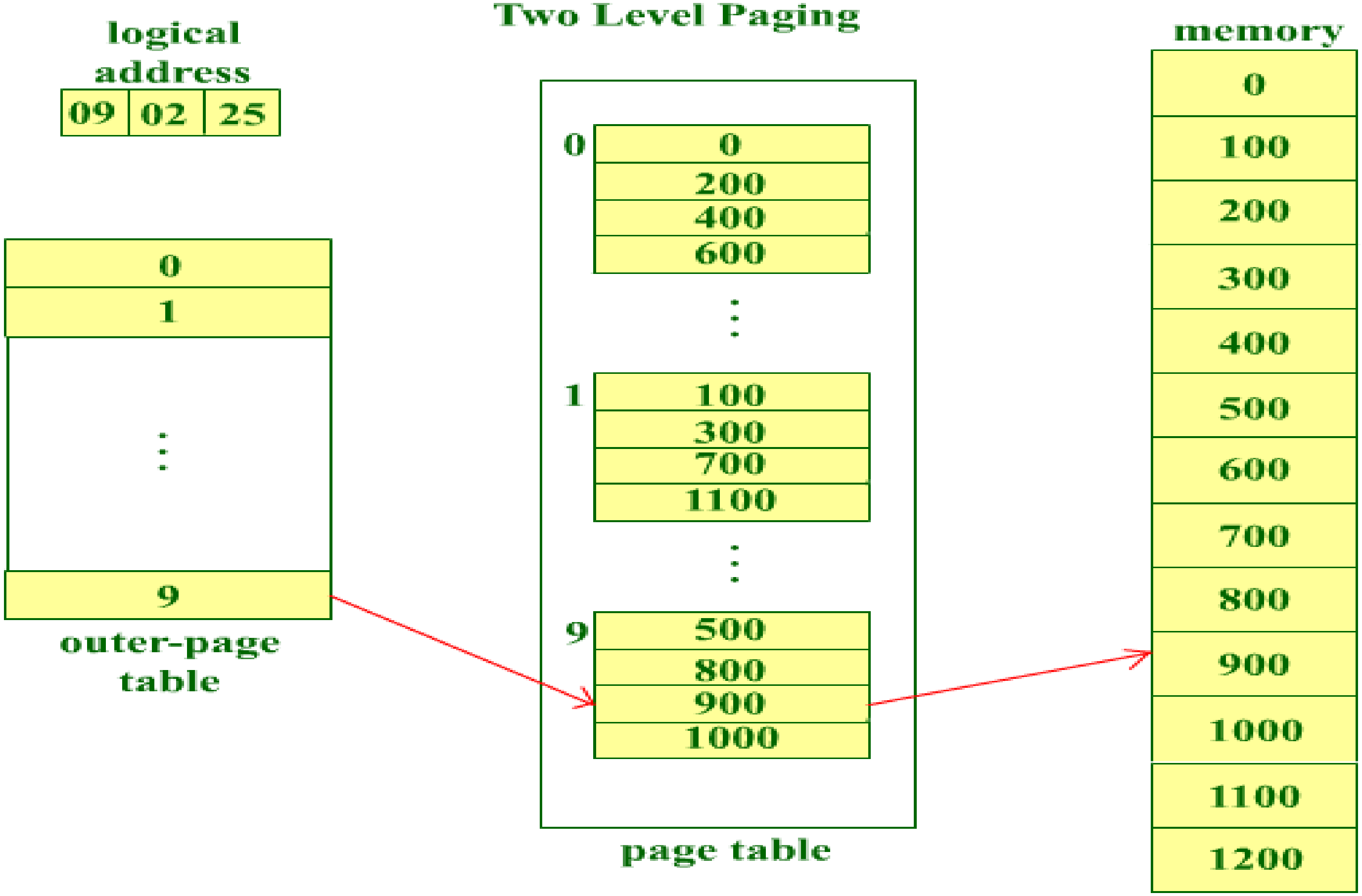
0
100
200
300
400
500
600
700
800
900
1000
1100
1200

'or example, if the CPU produces logical address 010340, it will reference uter page 01, which points to page 1 in the page table. Then we access offset 3 within the page to obtain frame number 1100. We then use the offset to roduce physical memory address 1140.



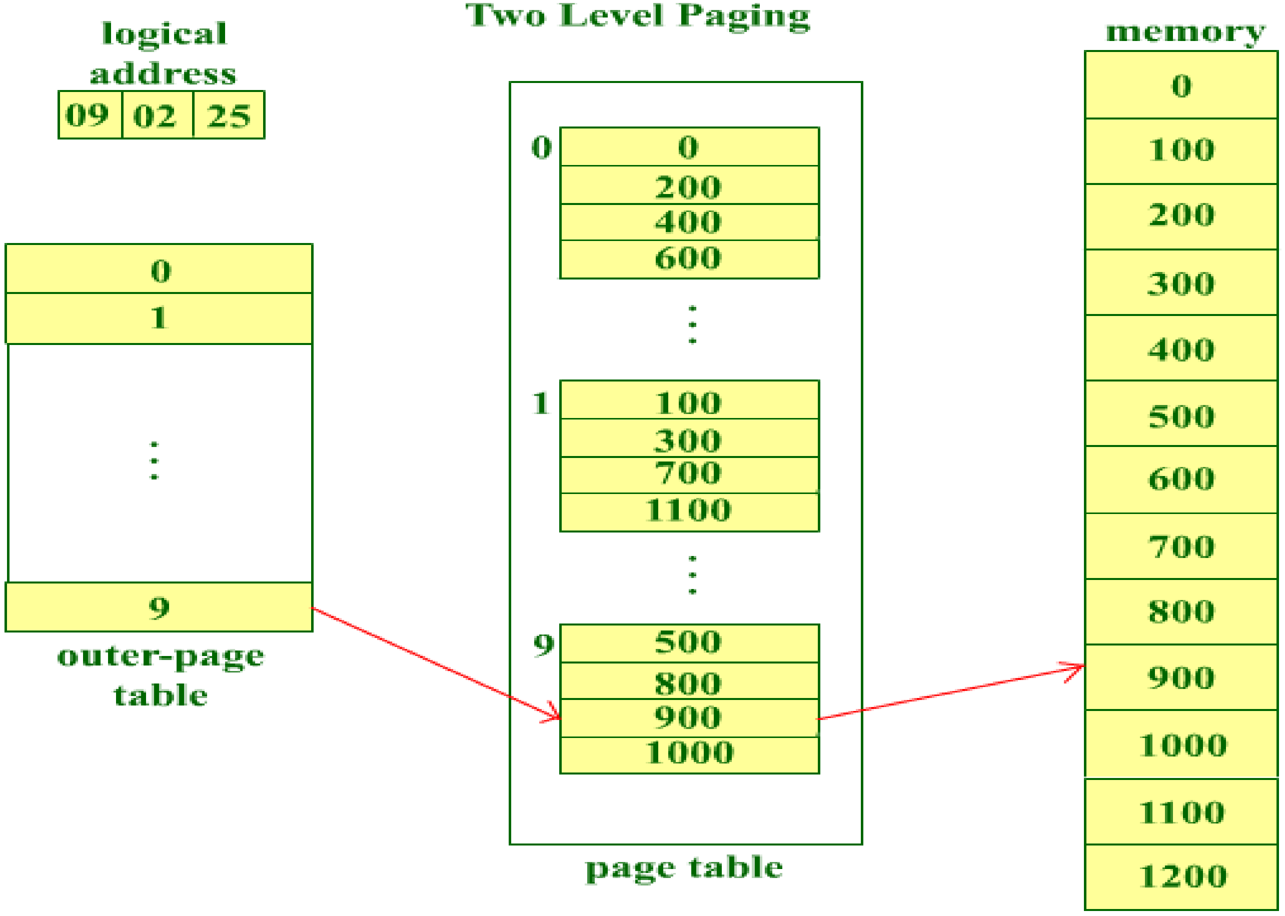
If the CPU produces logical address 000100, it will reference outer page 00, which points to page 0 in the page table. Then we access offset 01 within the page to obtain frame number 200. We then use the offset to produce physical memory address 200.

Two Level Paging



If the CPU produces logical address 090225, it will reference outer page 09, which points to page 9 in the page table. Then we access offset 02 within the page to obtain frame number 900. We then use the offset to produce physical memory address 925.

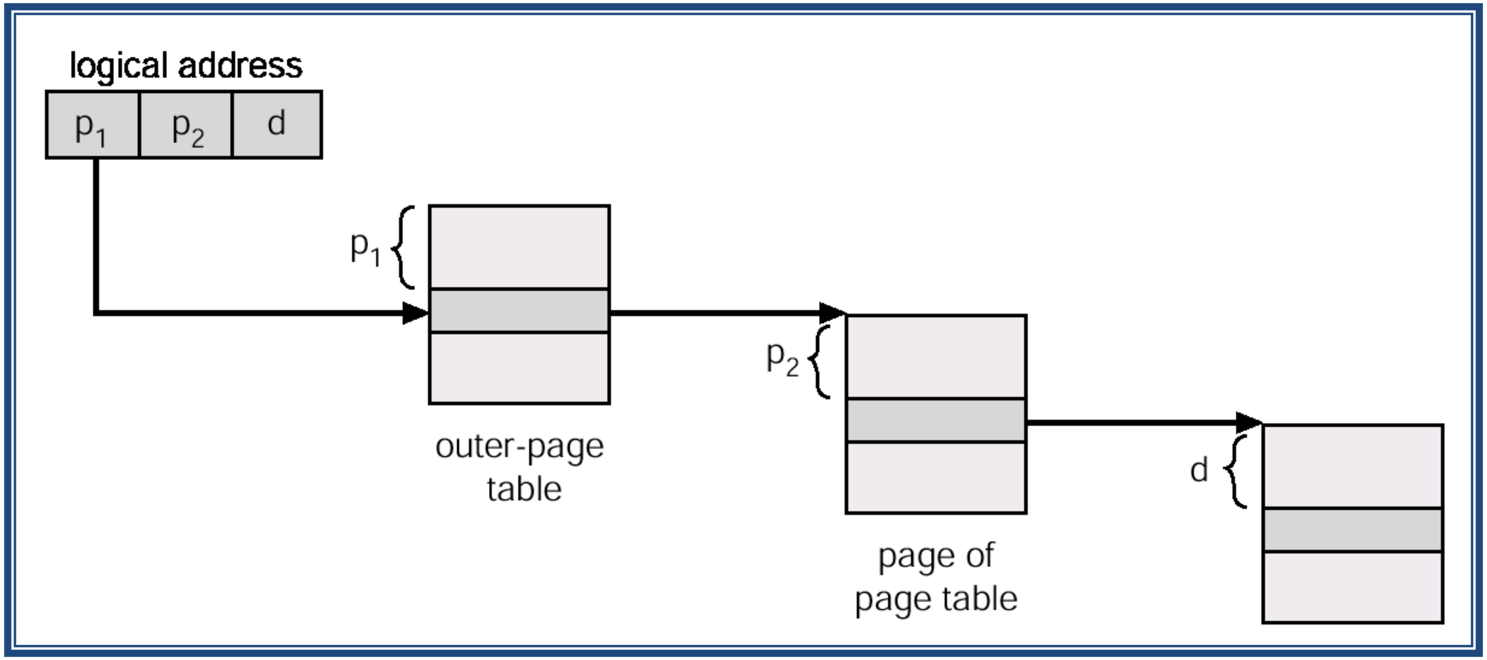
Two Level Paging



: CPU produces logical address 090225, it will reference outer page 09, 1 points to page 9 in the page table. Then we access offset 02 within the to obtain frame number 900. We then use the offset to produce physical

Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture



Page entries

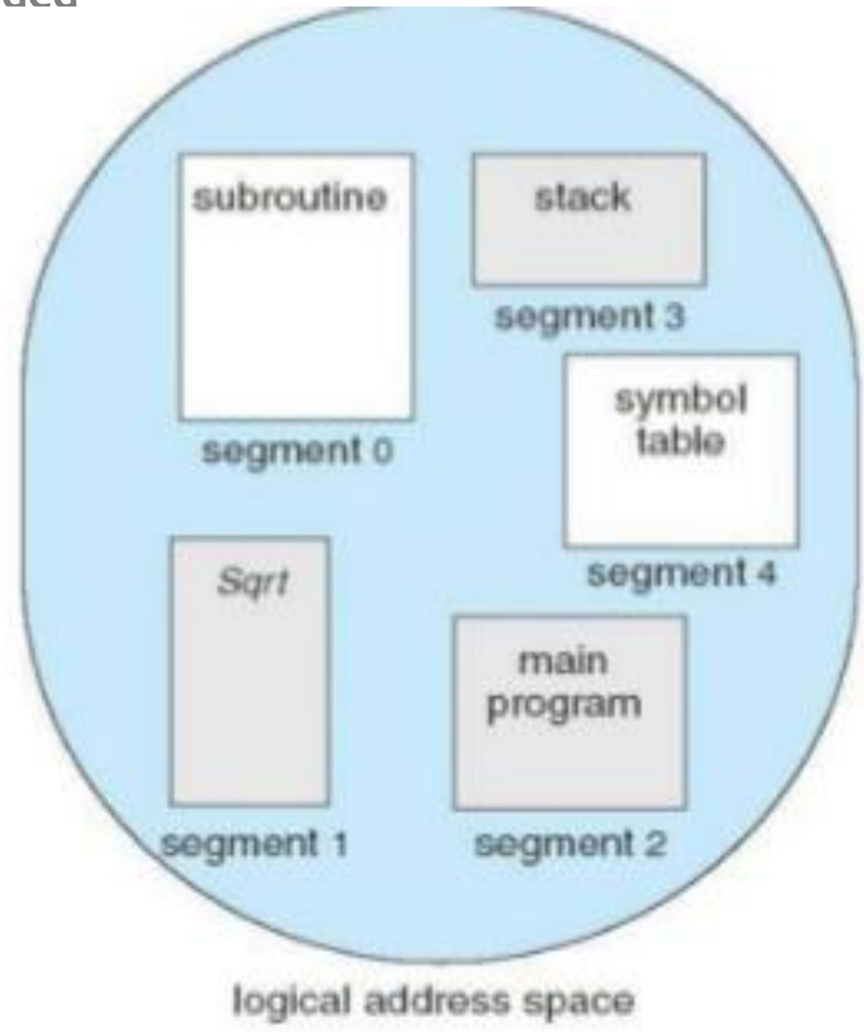
- **The entries of Page Directories and Page Tables have the same structure**
  - Present flag
  - Field containing the 20 most significant bits of a page frame physical address
  - Acessed flag
  - Dirty flag
  - Read/write flag
  - User/Superuser flag
  - Page size flag

Segmentation with Paging – Intel 386

- **As shown in the following diagram, the Intel 386 uses segmentation with paging for memory management with a two-level paging scheme.**

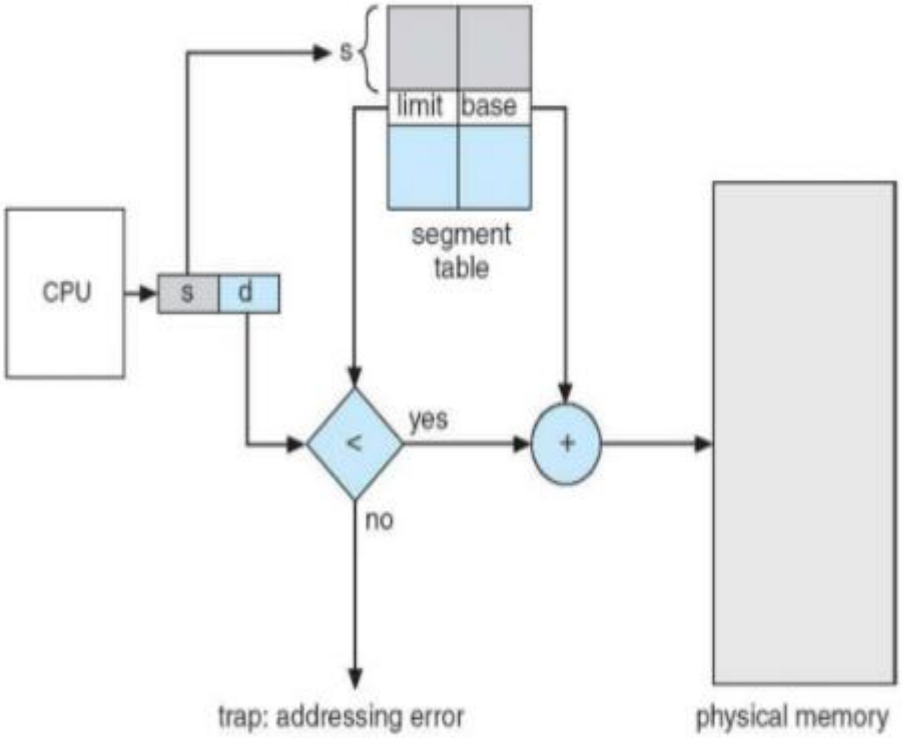
User's View

- **User's view – Program is divided into segments**
  - Subroutine
  - Stack
  - Main program
  - Symbol table



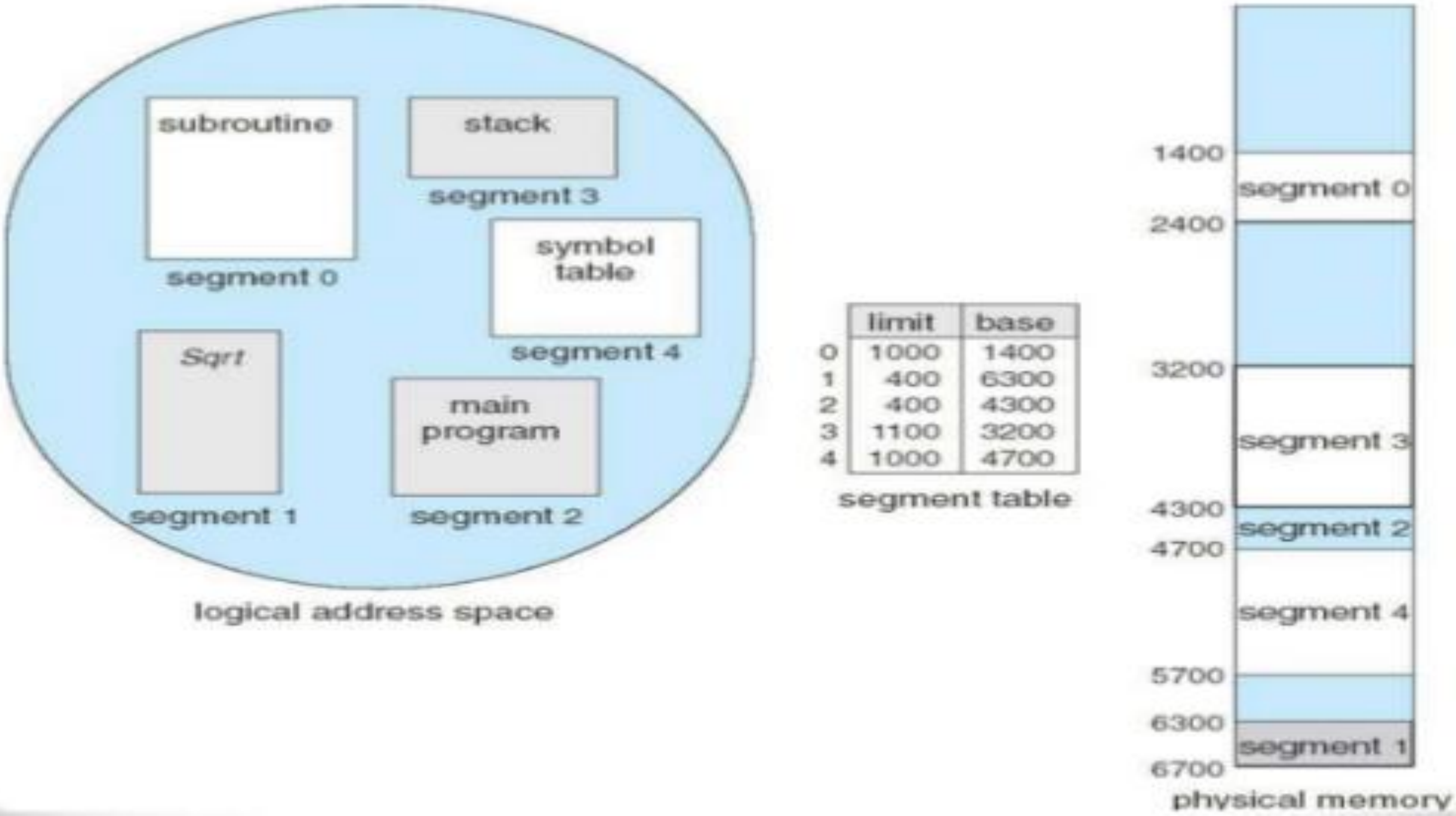
- Every address in user view contains 2 numbers Segment number and offset
- Segment table contains 1 entry for every segment
- Each entry contains limit and base
- Which helps to convert virtual address into physical address

SEGMENTATION HARDWARE



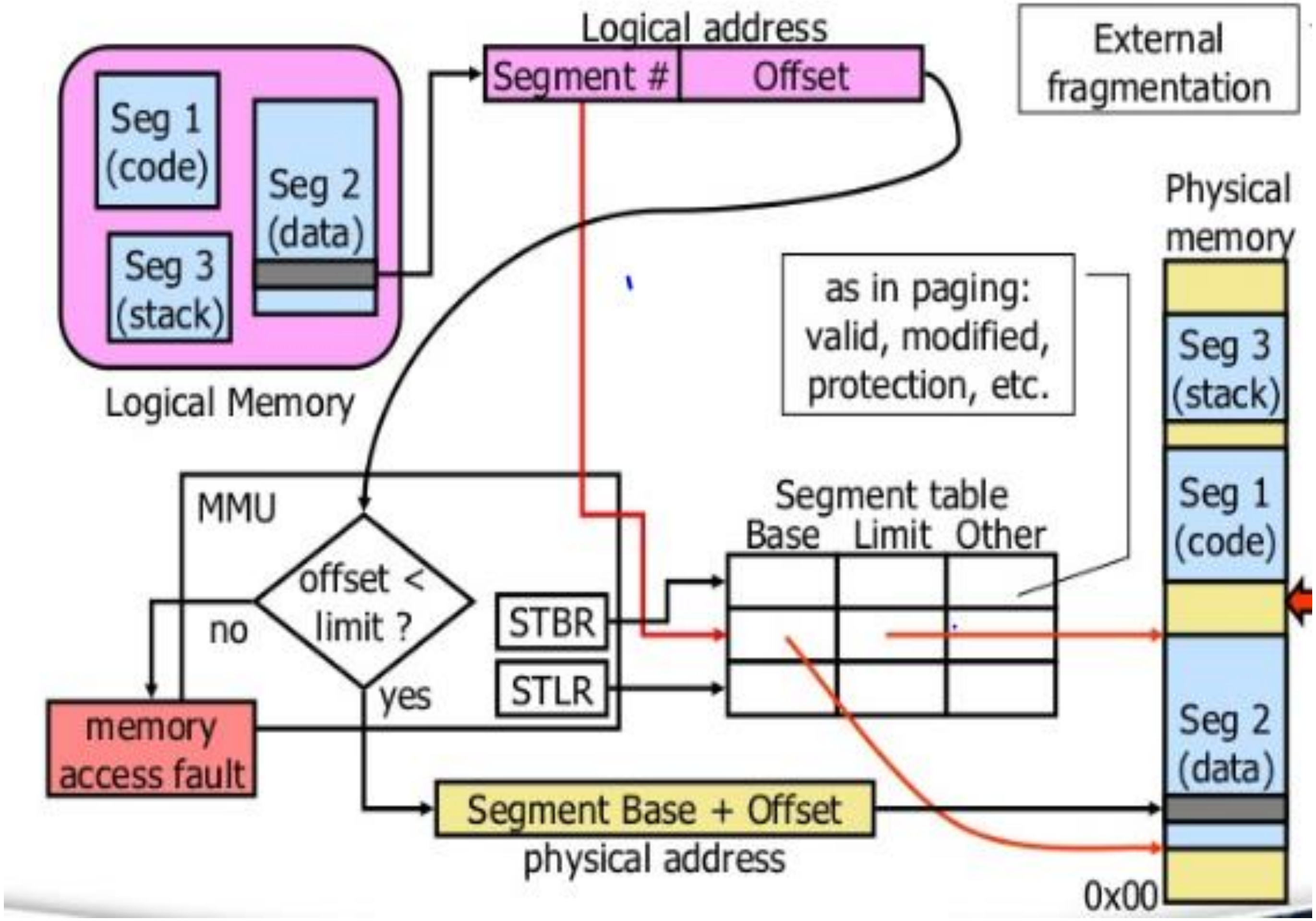
- Covert address segment 2 offset 30

EXAMPLE OF SEGMENTATION

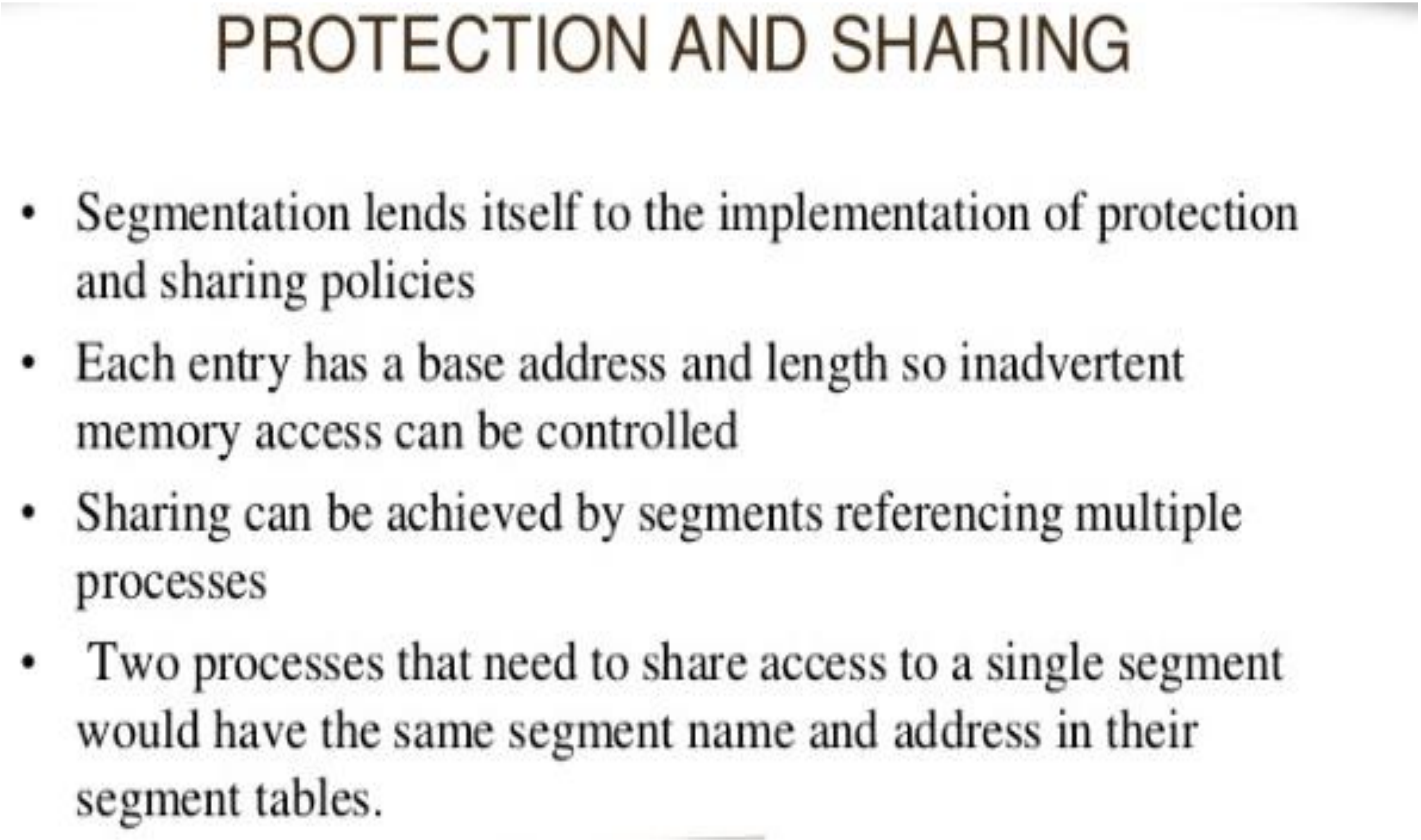


	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

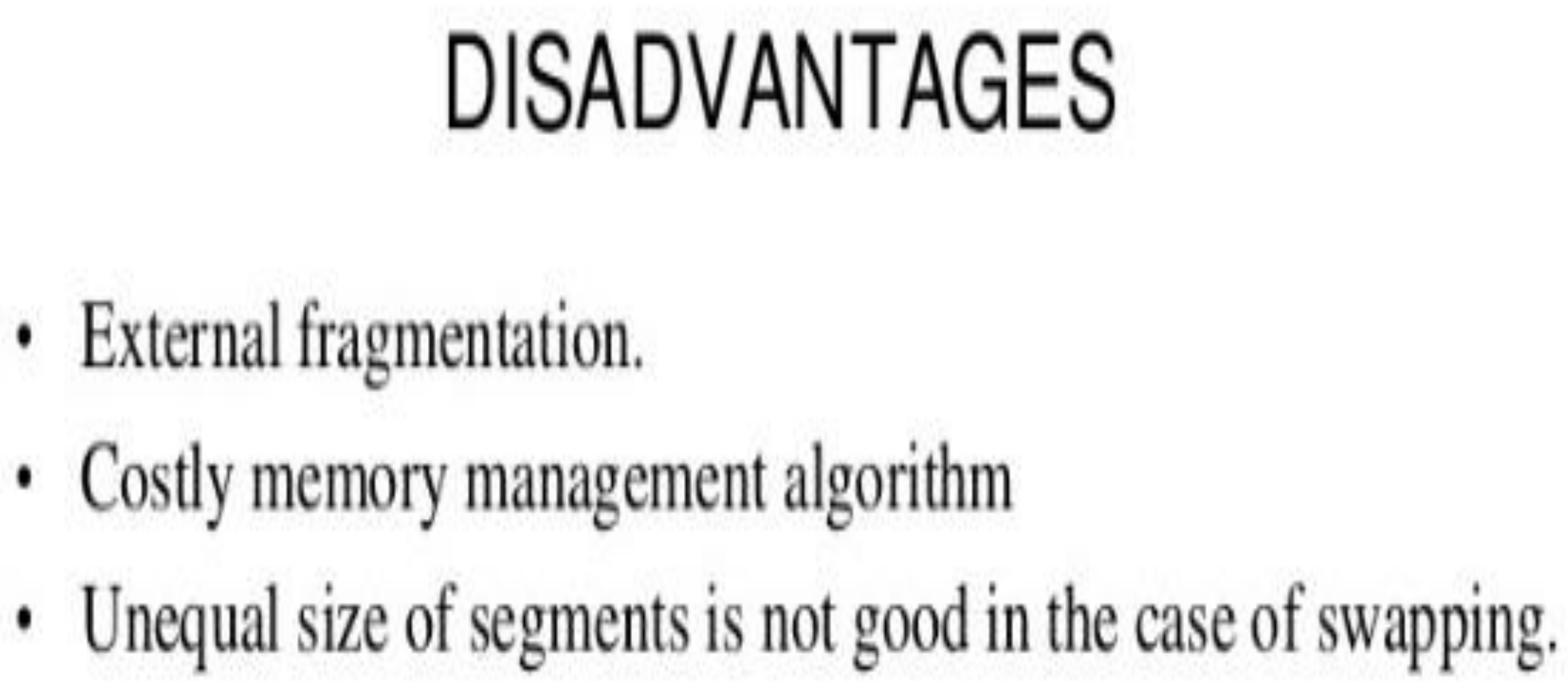


Logical address
Segment #
Offset
External fragmentation
Seg 1 (code)
Seg 2 (data)
Seg 3 (stack)
Logical Memory
Physical memory
as in paging: valid, modified, protection, etc.
Seg 3 (stack)
Seg 1 (code)
Seg 2 (data)
MMU
offset < limit ?
no
yes
STBR
STLR
Segment table
Base
Limit
Other
memory access fault
Segment Base + Offset
physical address
0x00



PROTECTION AND SHARING

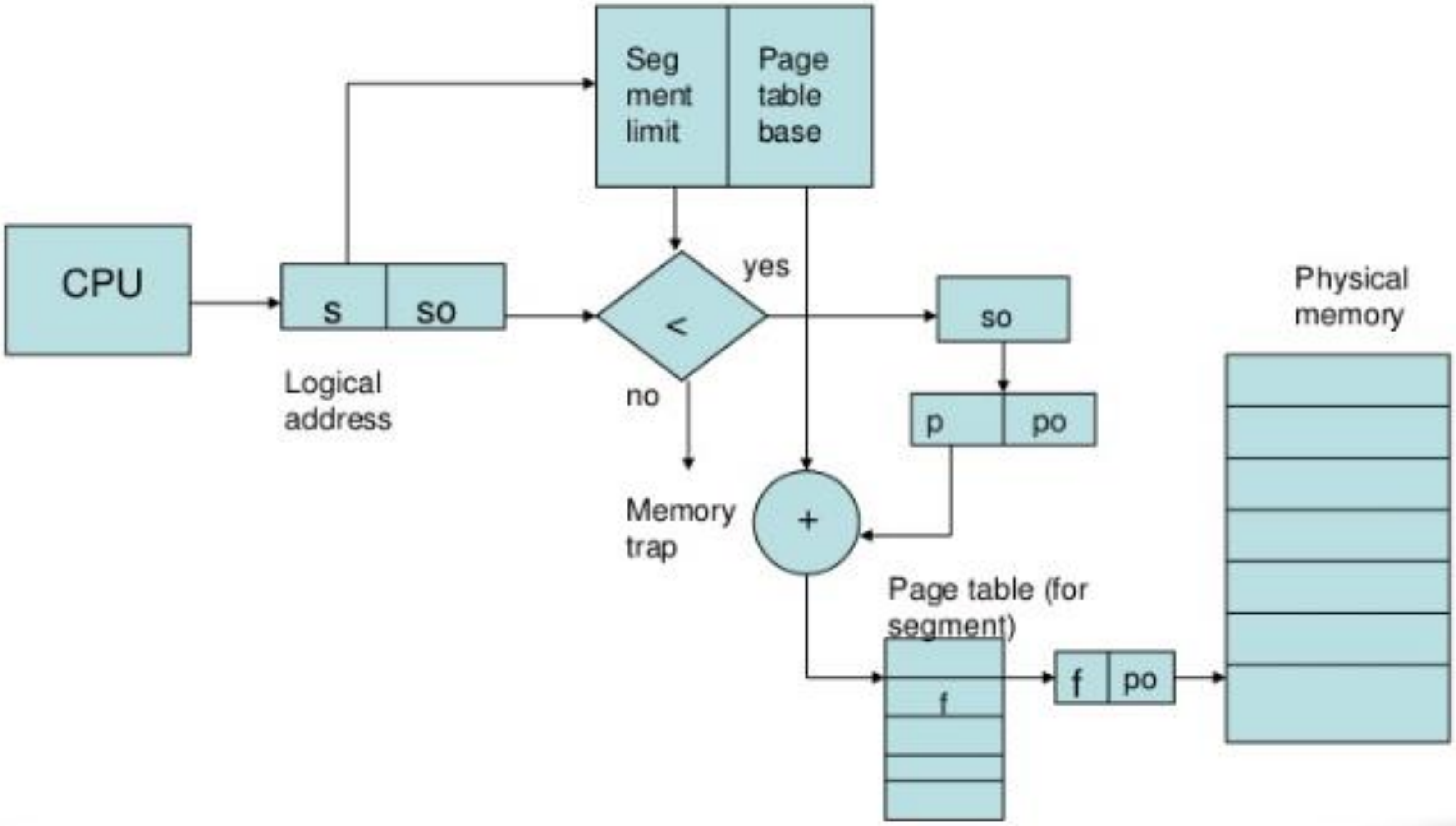
- Segmentation lends itself to the implementation of protection and sharing policies
- Each entry has a base address and length so inadvertent memory access can be controlled
- Sharing can be achieved by segments referencing multiple processes
- Two processes that need to share access to a single segment would have the same segment name and address in their segment tables.

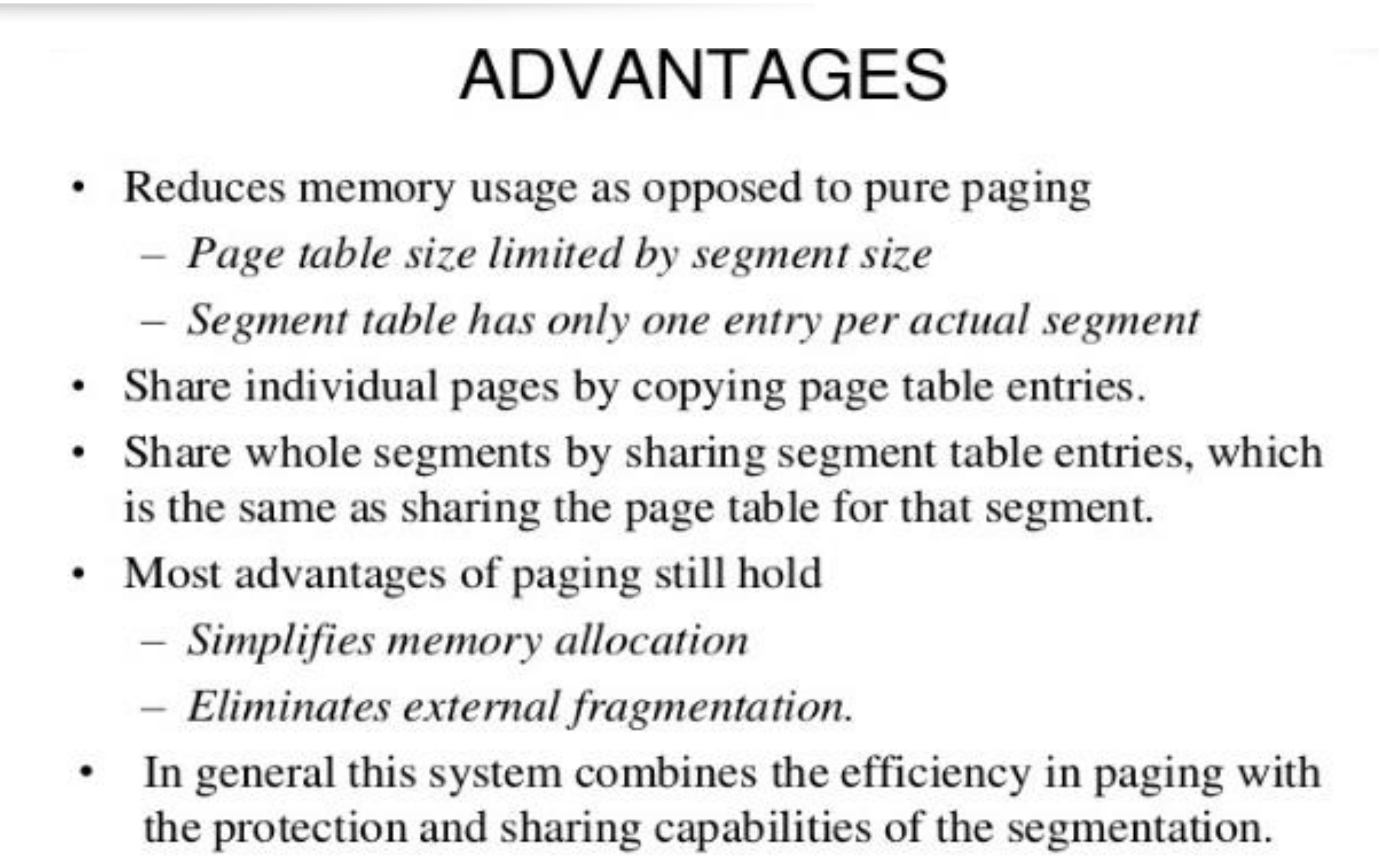


DISADVANTAGES

- External fragmentation.
- Costly memory management algorithm
- Unequal size of segments is not good in the case of swapping.

ADDRESS TRANSLATION





ADVANTAGES

- Reduces memory usage as opposed to pure paging
  - *Page table size limited by segment size*
  - *Segment table has only one entry per actual segment*
- Share individual pages by copying page table entries.
- Share whole segments by sharing segment table entries, which is the same as sharing the page table for that segment.
- Most advantages of paging still hold
  - *Simplifies memory allocation*
  - *Eliminates external fragmentation.*
- In general this system combines the efficiency in paging with the protection and sharing capabilities of the segmentation.

Trashing

Thrashing If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend, that process's execution. We should then page out its remaining pages, freeing all its allocated frames. This provision introduces a swap-in, swap-out level of intermediate CPU scheduling. In fact, look at any process that does not have "enough" frames. If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately. This high paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing.

Managing Heap

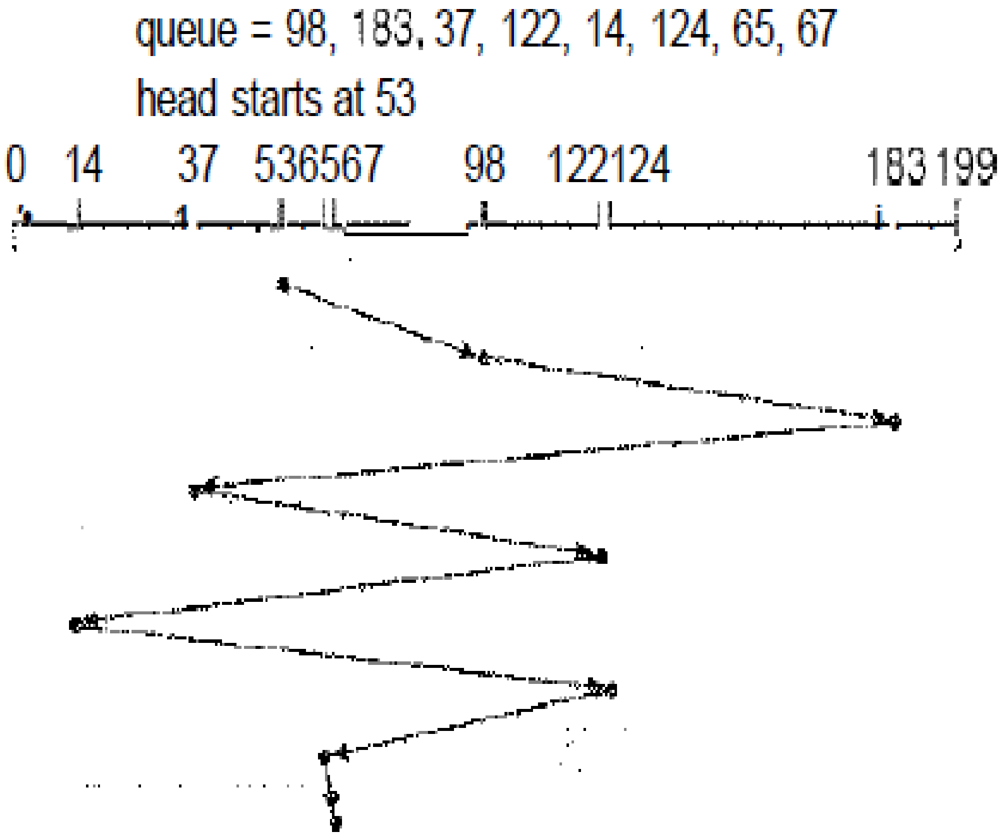
- **Each Unix process owns a specific memory region called *heap*, which is used to satisfy the process's dynamic memory requests**
- **The start_brk and brk fields of the memory descriptor delimit the starting and ending address, respectively, of that region**
- **The following C library functions can be used by the process to request and release dynamic memory:**
  - malloc(size) : Request size bytes of dynamic memory
  - calloc(n,size) : Request an array consisting of n elements of size size;
  - free(addr) : Release the memory region allocated by malloc( ) or calloc( ) having initial address addr.
  - brk(addr) : Modify the size of the heap directly

Disk scheduling

- FCFS(First come first serve)
- SSTF(Shortest seek time first
- Scan
- CScan
- Look Clook

- Common choices are SSTF and Look

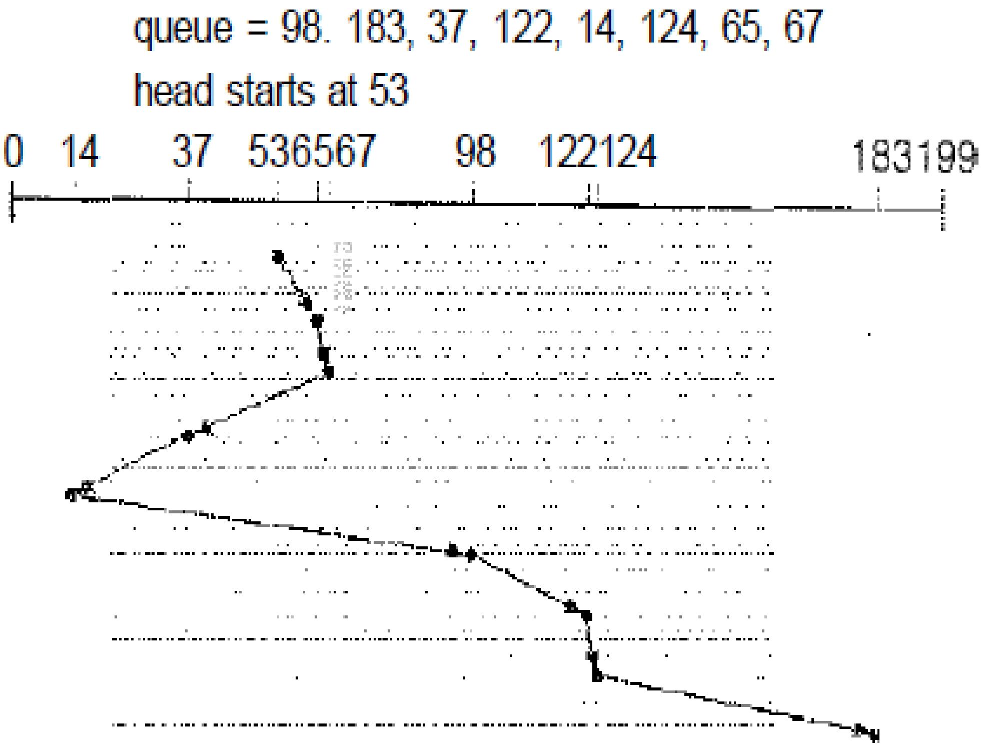
FCFS



queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

SSTF

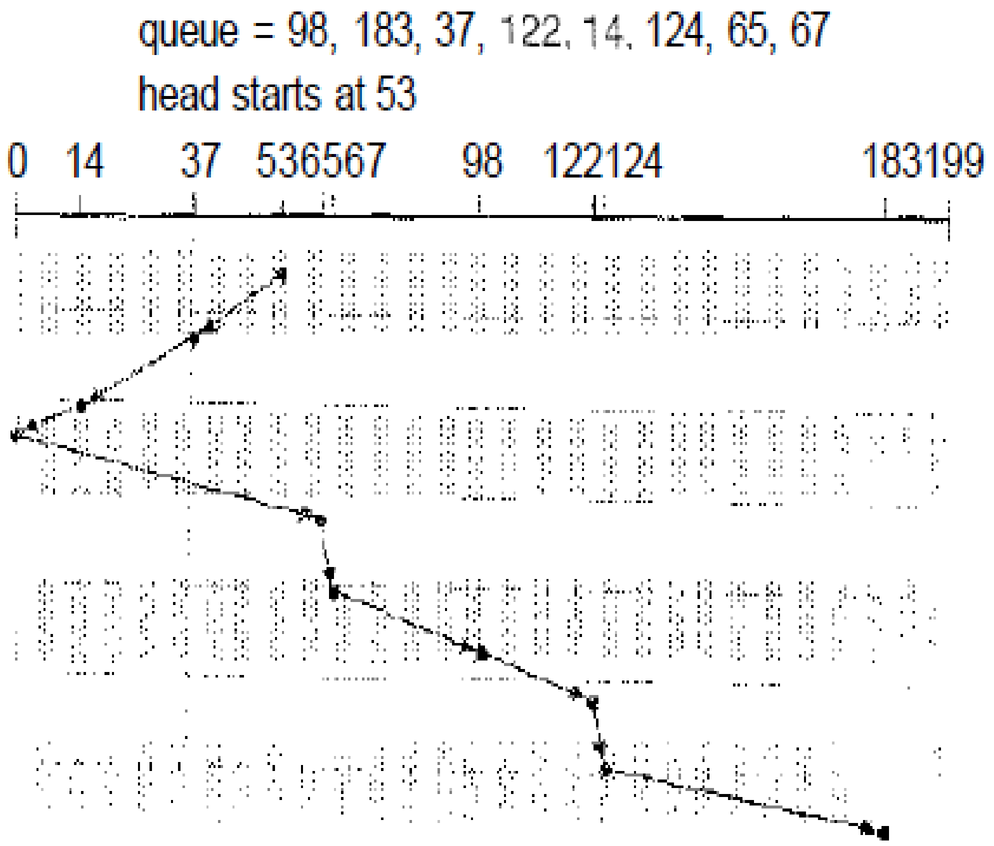


queue = 98. 183, 37, 122, 14, 124, 65, 67

head starts at 53

Total head movement reduce substantially - 208

Scan scheduling



queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

C-Scan(Circular scan)

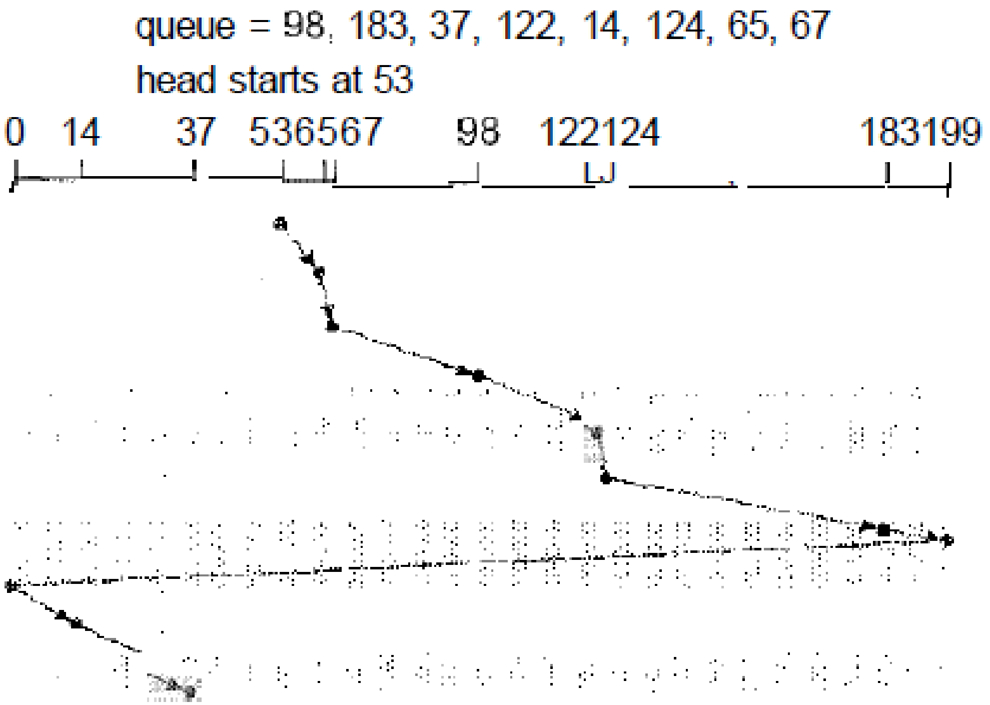
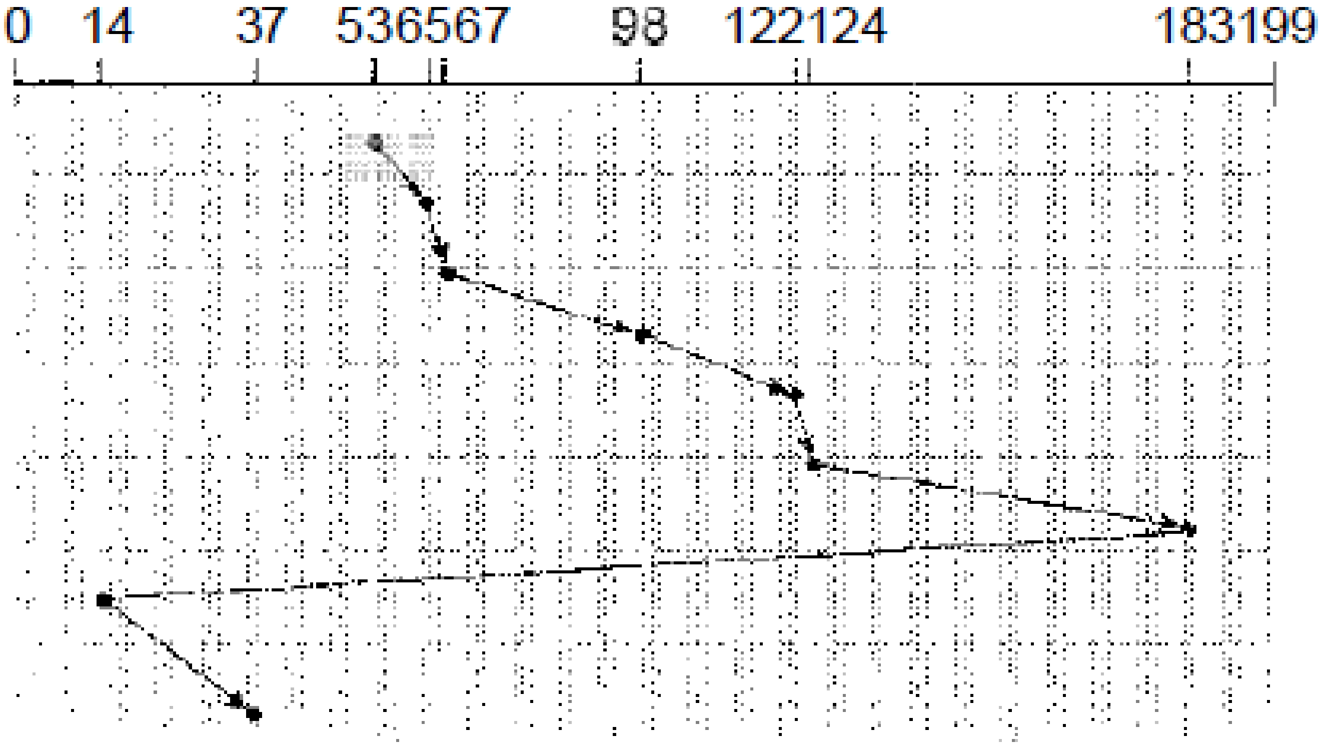


Figure 12.7 C-SCAN disk scheduling.

No request are processed in return trip

queue 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



File system

- VFS
- Ext2,etx3
- NTFS
- HDFS

Deadlocks

Necessary Conditions

- **A deadlock situation can arise if the following four conditions hold simultaneously in a system:**
- **1. Mutual exclusion.**
  - At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
- **2. Hold and wait.**
  - A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes

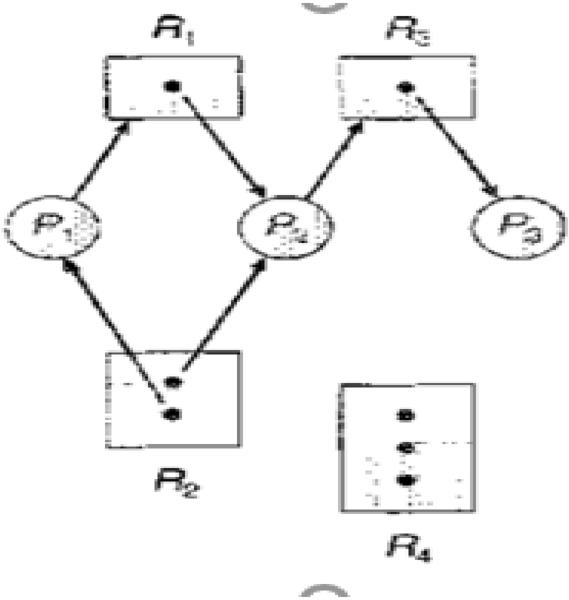
Deadlock necessary conditions

- **. 3. No preemption.**
  - Resources cannot be preempted.; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task. /.2 Deadlock Characterization 249

- **4. Circular wait.**
  - A set {P1, P2, …, Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P2 is waiting for a resource held by P3, • • •, Pn-1 is waiting for a resource held by Pn, and P1 is waiting for a resource held by Pn.

Resource-Allocation Graph

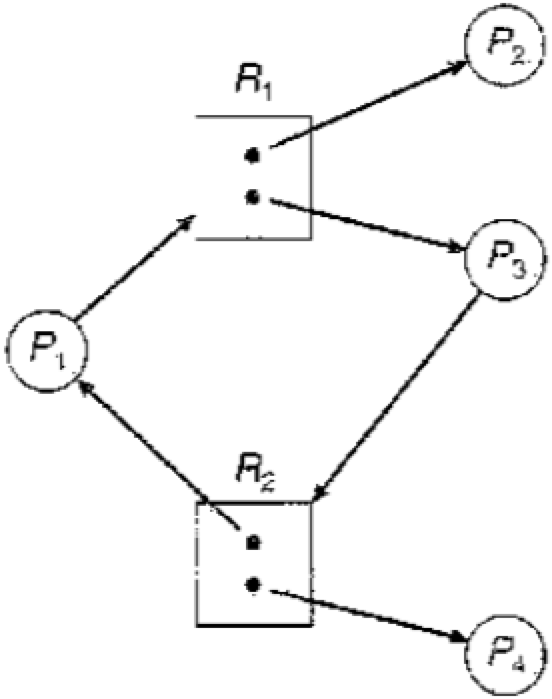
- **The sets P, R, and £:**
- **P={P1,P2,P3}**
- **R= {R1, R2,R3, R4}**
- **£ = {p1 -> R1, P2 _> R3 R1 -> p2, R2 -> P2, R2 -> p1, R3 -> P3 }**
- **Resource instances:**
  - One instance of resource type R1
  - Two instances of resource type R2
  - One instance of resource type R3
  - Three instances of R4



Details

- Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R|.
- Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
- Process P3 is holding an instance of R3.

No DeadLock



Observations

- **P4 may release its instance of resource type R2. That resource can then be allocated to P3,**
- **This will break the cycle. Hence no deadlock**
- **if a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state.**

Ways to handle deadlock

- Methods for Handling Deadlocks Generally speaking, we can deal with the deadlock problem in one of three ways:

1. use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
2. allow the system to enter a deadlock state, detect it, and recover.
3. Ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by linux

Deadlock Prevention

- **If any one of the four necessary condition becomes false then we can prevent deadlock**
- **Mutual Exclusion**
  - The mutual-exclusion condition must hold for nonsharable resources
  - we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable,

- **Hold and Wait**
  - To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
  - Protocol

1. Each process to request and be allocated all its resources before it begins execution.
2. alternative protocol allows a process to request resources only when it has none.

- A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

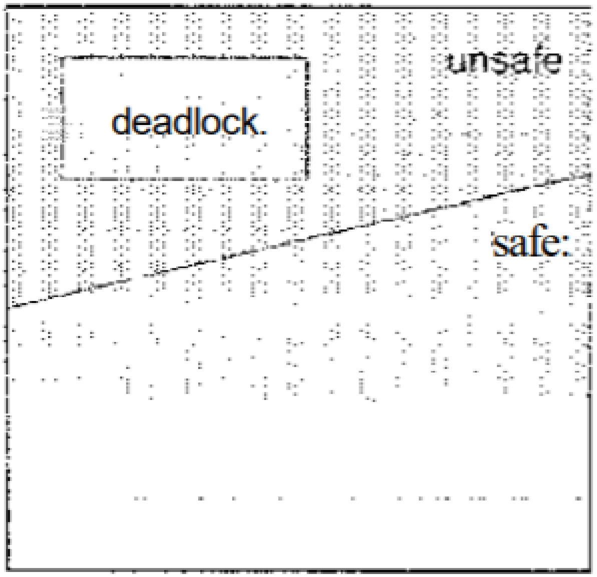
- **No Preemtion**
- **To nullify this condition**
  - If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted.

- **Circular Wait**
  - **each process requests resources in an increasing order of enumeration**

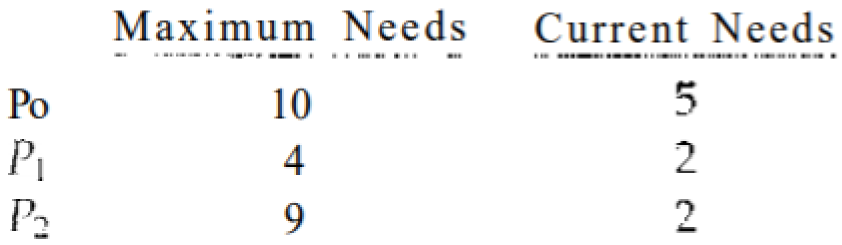
Deadlock Avoidance

- For avoiding deadlocks is to require additional information about how resources are to be requested.
- For example, in a system with one tape drive and one printer, the system might need to know that process P will request first the tape drive and then the printer before releasing both resources, whereas process Q will request first the printer and then the tape drive.
- With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.

- **Safe State A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence.**



- When deadlock occurs:
- The behavior of the processes controls unsafe states.
- Example
  - consider a system with 12 magnetic tape drives and three processes: P0/ P1, and P2. at time to current need is 5,2,2 respectively can be satisfied
- So system is in safe state <p1,p0,p2>
  -



	Maximum Needs	Current Needs
Po	10	5
P_1	4	2
P_2	9	2

Deadlock Avoidance Algorithm : Banker's Algorithm

- Data structures used by bankers algorithm
  - Let n be the number of processes in the system
  - m be the number of resource types.
  - We need the following data structures:

- Available. A vector of length m indicates the number of available resources of each type. If Available[j] equals k, there are k instances of resource type Rj available.

- Max. An n x m matrix defines the maximum demand of each process.

If Max[i][j] equals k, then process Pi may request at most k instances of resource type rj

- Allocation.

An n x m in matrix defines the number of resources of each type currently allocated to each process.

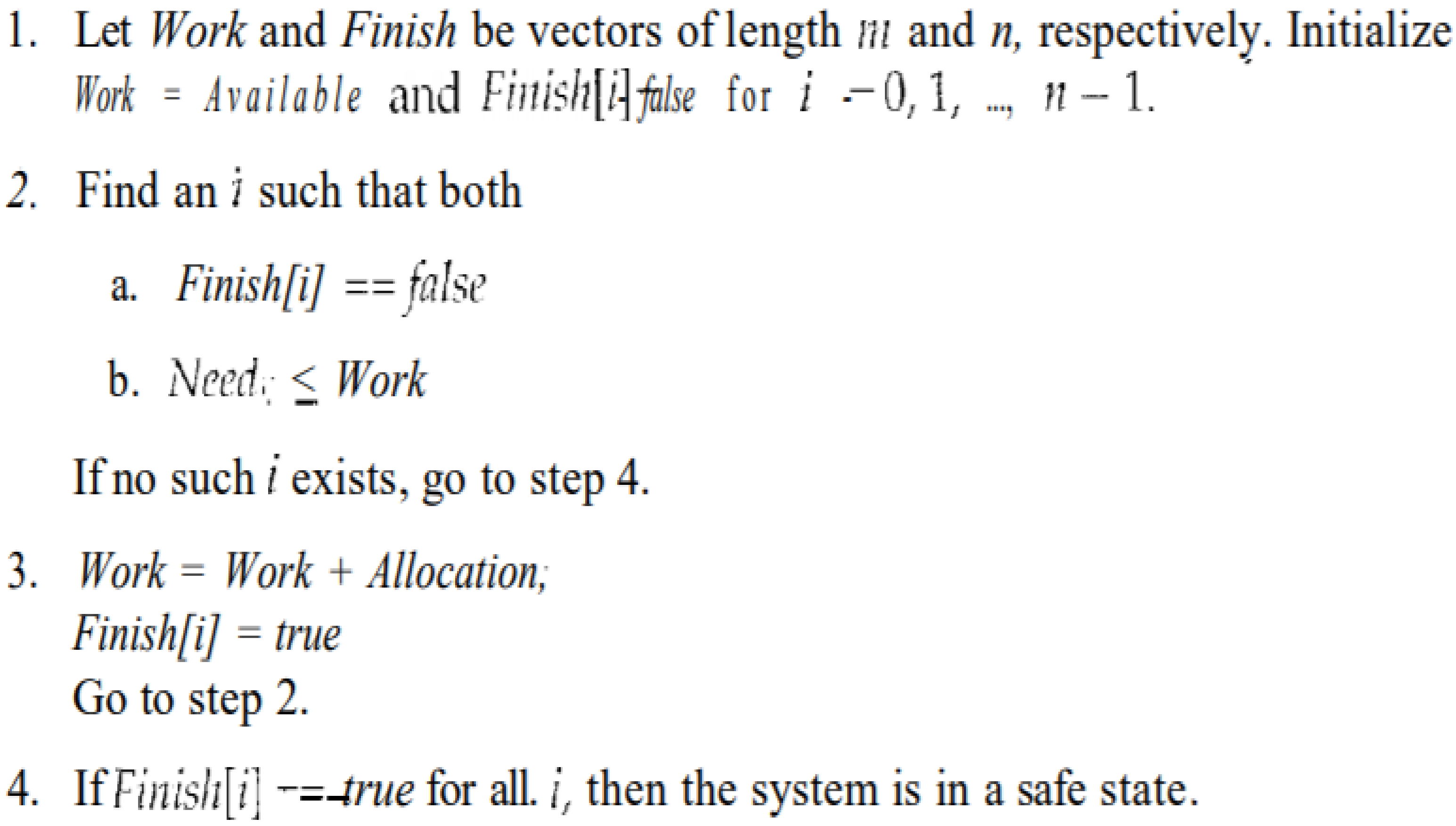
If Allocation[i][j] equals k, then process Pi is currently allocated k instances of resource type Rj.

- Need.

An n x m matrix indicates the remaining resource need of each process. If Need[i][j] equals k, then process Pi, may need k more instances of resource type Rj to complete its task.

➢ Note that Need[i][j] equals Max[i][j] - Allocntion[i][j].

Safety Algorithm



1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize *Work* = *Available* and *Finish*[*i*] = *false* for $i = 0, 1, \ldots, n-1$.

2. Find an *i* such that both

   a. *Finish[i]* == *false*

   b. $Need_i \leq Work$

   If no such *i* exists, go to step 4.

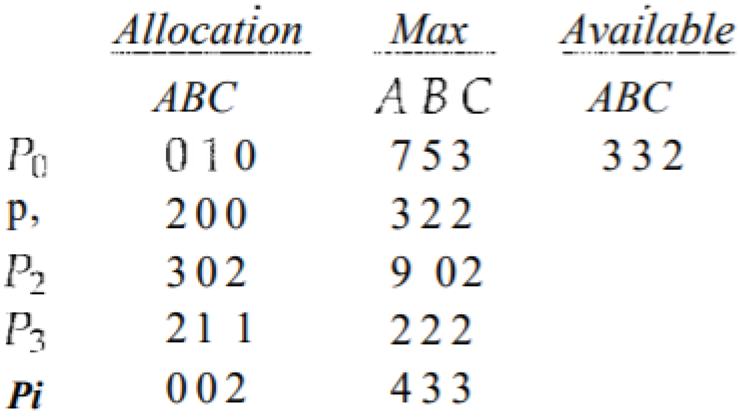
3. $Work = Work + Allocation_i$
   Finish[i] = *true*
   Go to step 2.

4. If *Finish*[*i*] == *true* for all *i*, then the system is in a safe state.

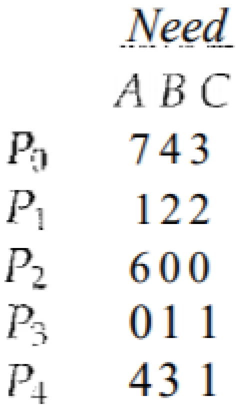
Resource-Request Algorithm

- **We now describe the algorithm which determines**
- **if requests can be safely granted.**
- **Let Request**
  - be the request vector for process P,. If Request [j] ==k, then process P, wants k instances of resource type Rj. When a request for resources is made by process P, then following actions are taken:

- **Finally, to illustrate the use of the banker's algorithm, consider a system with five processes PQ through P4 and three resource types A, B, and C. Resource type A has 10 instances, resource type B has 5 instances, and resource type C has 7 instances. Suppose that, at time To, the following snapshot of the system has been taken:**



	Allocation	Max	Available
	ABC	A B C	ABC
P_0	010	753	332
p,	200	322	
P_2	302	9 02	
P_3	21 1	222	
Pi	002	433	



	Need
	A B C
P_0	743
P_1	122
P_2	600
P_3	01 1
P_4	43 1

The content of the matrix Need is defined to be Max – Allocation

We claim that the system is currently in a safe state. Indeed,
(p1,p3,p4,p2,p0)
the sequence satisfies the safety criteria

Check whether p1 - (1,0,2) request can be granted
p3 - (3,3,0)
p0 - (0,2,0)

1. If Request<= Need:, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If Request i <= Available i, go to step 3. Otherwise, Pi must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P- by modifying the state as follows: Available = Available - Request;;

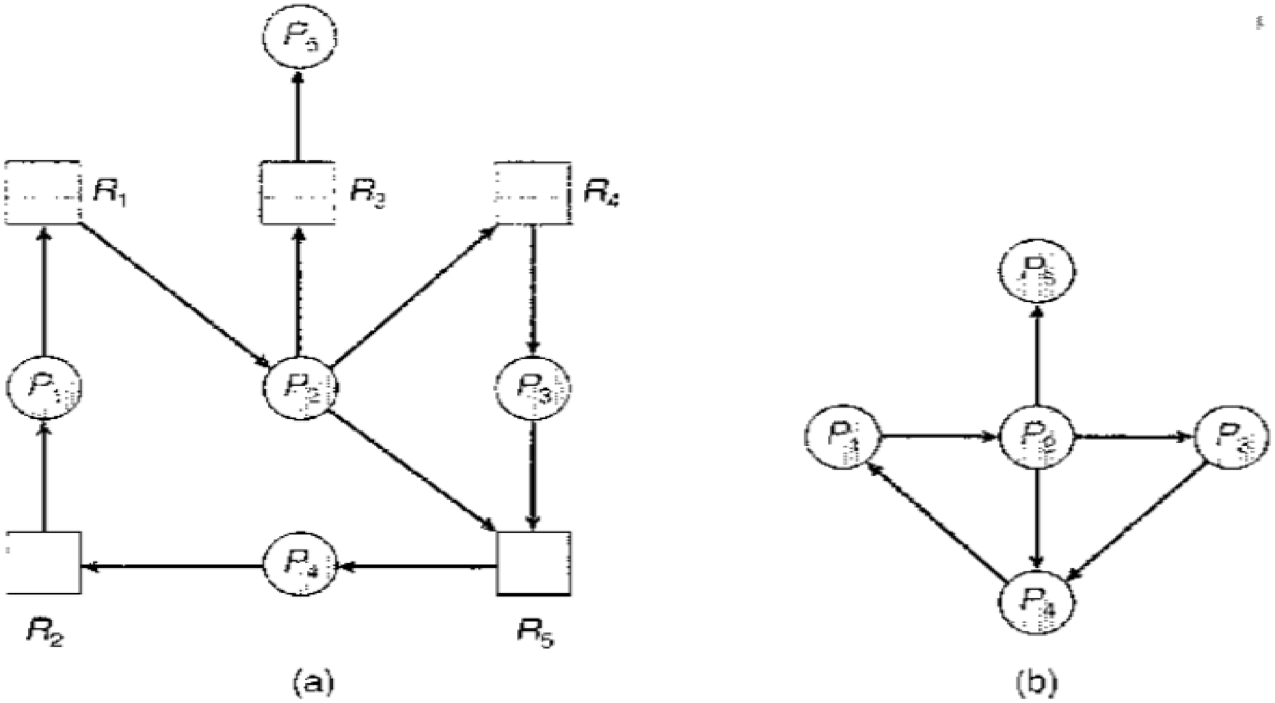
Allocation = Allocation + Request

Need; = Needj - Requestj

- **If the resulting resource-allocation state is safe, the transaction is completed, and process P; is allocated its resources. However, if the new state is unsafe, then P, must wait for Request;, and the old resource-allocation state is restored.**

Deadlock detection

- Single Instance of Each Resource Type



Several Instances of a Resource Type

- The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.
- We turn now to a deadlock detection algorithm
- that is applicable to such a system.
- The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm
- Available. A vector of length m indicates the number of available resources of each type.
- • Allocation. An n x m matrix defines the number of resources of each type currently allocated to each process.
- • Request. An n x in matrix indicates the current request of each process. If Request[i][j] equals k, then process P, is requesting k more instances of resource type Rj. Several Instances of a Resource Type

Recovery From Deadlock

- **Abort all deadlocked processes.**
  - This method clearly will break the deadlock cycle, but at great expense; the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated.**
  - This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

- solution is a total rollback: Abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes. 3. Starvation. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process? In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that must be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor. 7.8 Summary