Memory management

To increase the performance of the system, we must keep many processes in memory, that is, we must share the memory.

The memory-management method for a specific system depends on many factors, especially on the hardware design of the system.

The CPU fetches instructions from memory according to the value of the program counter.

The instruction is then decoded and may cause operands to be fetched from memory.

After the instruction has been executed on the operands, results may be stored back in memory.

The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data).

Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as logical address.

An address actually seen by the memory unit - the one loaded into the memory-address register of the memory - is known as physical address.

Binding addresses at either compile or load time generates identical logical and physical addresses.

However, the execution-time address-binding scheme results in differing logical and physical addresses.

In this case, we usually refer to the logical address as a virtual address.

The set of all logical addresses generated by a program is a logical address space.

The set of all physical addresses corresponding to these logical addresses is a physical address space.

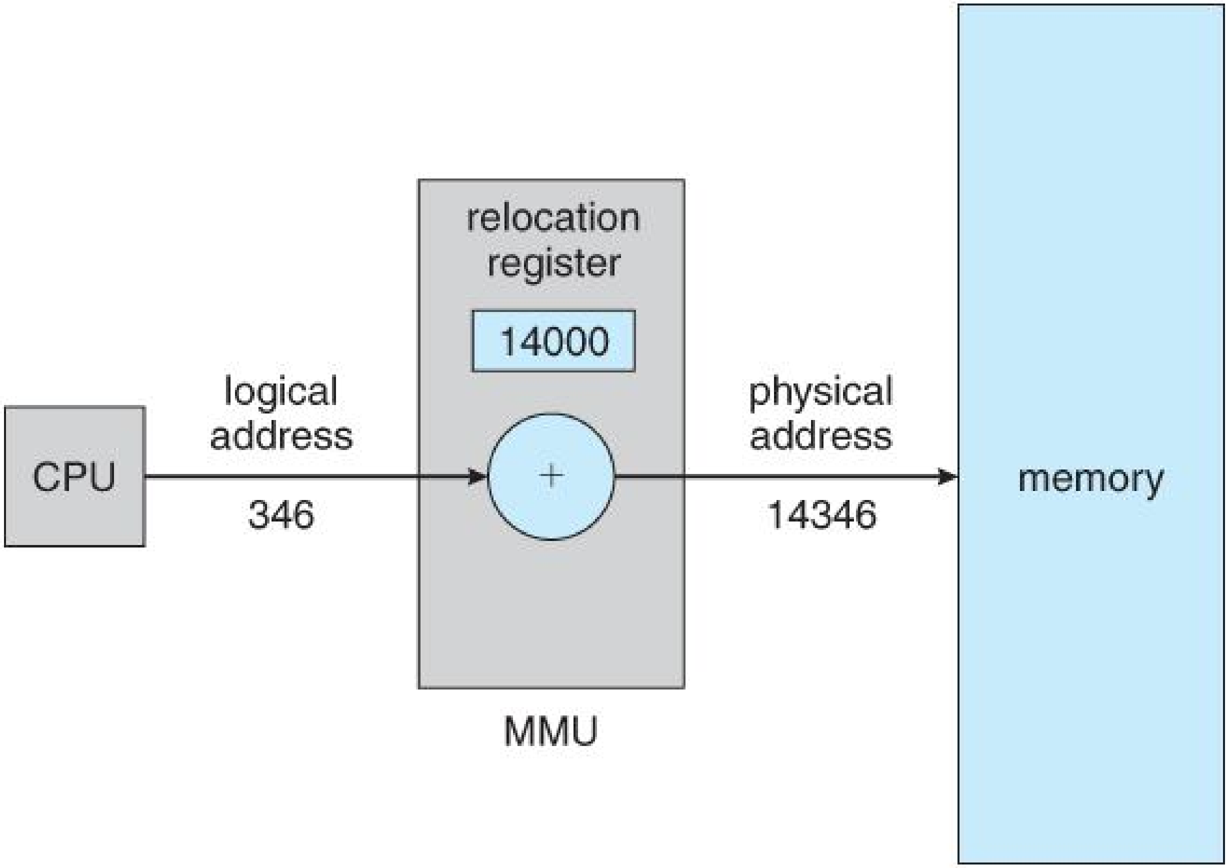


Fig 2: Dynamic relocation using a relocation register

Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.

The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU).

In a simple MMU scheme, the value of the relocation register is added to every address generated by a user process at the time the address is sent to memory.

The user program never accesses the real physical addresses and deals with logical address.

Now have two different types of addresses: logical addresses (in the range 0 to *max*) and physical addresses (in the range *R* + 0 to *R* + *max*) where *R* is the value of relocation register.

The user program generates only logical addresses and thinks that the process runs in memory locations from 0 to *max*.

Overlays

To enable a process to be larger than the amount of memory allocated to it, we can use overlays.

The idea of overlays is to keep in memory only those instructions and data that are needed at any given time.

When the instructions are needed, they are loaded into space occupied previously by instructions that are no longer needed.

Let consider the example of two pass assembler.

During pass 1, it constructs a symbol table; during pass 2, it generates machine language code.

The assembler can be partitioned into pass 1 code, pass 2 code, the symbol table, and the common routine used by both pass 1 and pass 2.

Assume that the sizes of these components are as follows:

Pass 1: 70 KB
Pass 2: 80 KB
Symbol table: 20 KB
Common routines: 30 KB

To load everything at once, we need 200KB of memory.

But if only 150 KB is available, then we cannot run our process.

But it can be observed that Pass 1 and Pass 2 do not need to be in memory at the same time.

To solve this problem, we define two overlays:

Overlay A is symbol table, common routines and Pass 1.

Overlay B is symbol table, common routines and Pass 2.

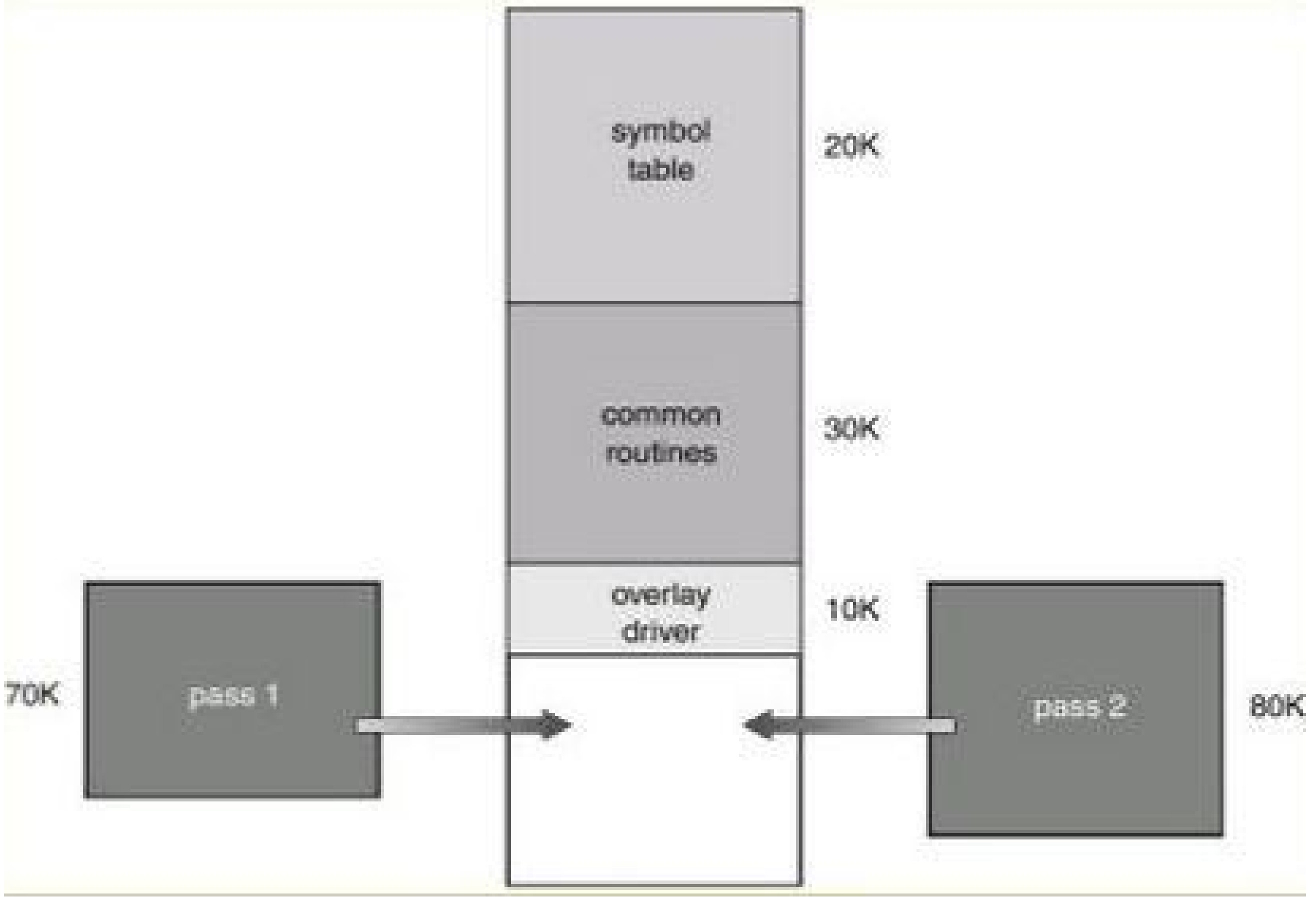


Fig 3: Overlays for two-pass assembler.

We add an overlay driver (size 10 KB) and start with overlay A in memory.

When we finish Pass 1, we jump to the overlay driver, which reads overlay B in memory, overwriting overlay A, and then transfers control to pass 2.

In this scheme overlay A needs 120 KB and overlay B needs 130 KB.

Therefore we can run the assembler in 150 KB memory.

However it will run slower due to the extra I/O to read the code for overlay B over the code for overlay A.

Overlays do not require any special support from the operating system.

They can be implemented completely by the user.

Operating system notices only that there is more I/O than usual.

Overlays are currently limited to microcomputer and other system that have limited amount of physical memory and that lack hardware support for more advanced techniques.

Swapping

A process needs to be in memory for execution.

However a process can be swapped temporary out of memory to a backing store, and then brought back into memory for continued execution.

For example, consider a multi programming environment with RR CPU scheduling algorithm.

When quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed.

In the meantime, the CPU scheduler will allocate a time slice to some other process in memory.

When each process finishes its quantum, it will be swapped with another process.

The quantum must also be sufficiently large that reasonable amounts of computing are done between swaps.

Swapping policy can also be applied with priority-based scheduling algorithms.

If higher priority process arrives and wants service, the memory manager can swap out the lower-priority process so that it can load and execute the higher-priority process.

When the higher-priority process finishes, the lower-priority process can be swapped back in and continued.

This type of swapping is also known as ***roll out***, ***roll in***.

Generally, a process that is swapped out will be swapped back into the same memory space that is occupied previously.

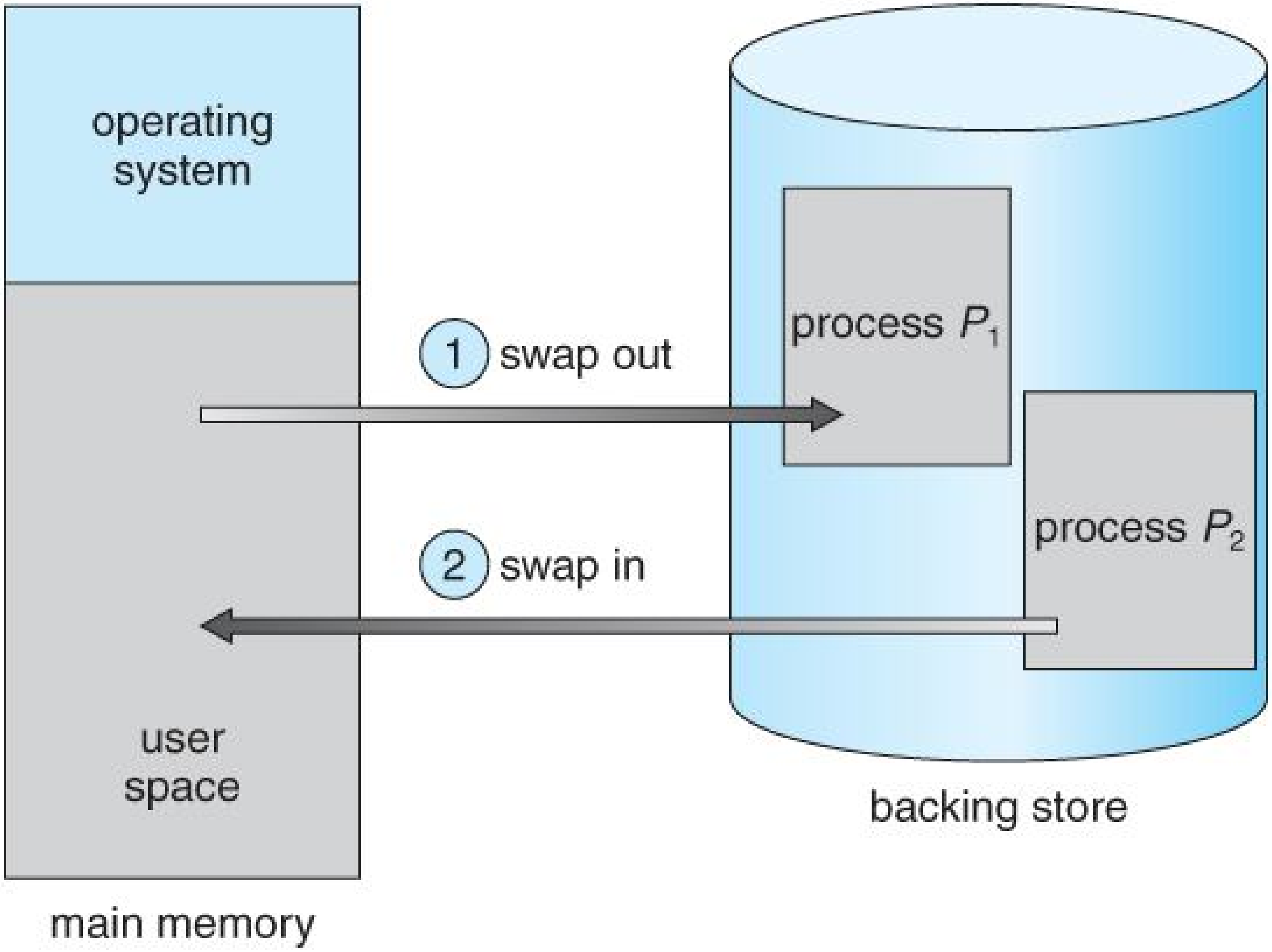


Fig 4: Swapping of two processes using a disk as a backing store.

This restriction is dictated by the method of address binding.

If binding is done at assembly or load time, then the process con not be moved to different locations.

If execution-time binding is being used, then a process can be swapped into a different memory space as the physical addresses are computed during execution time.

Swapping requires a backing store which must be large enough to accommodate copies of all memory images for all users.

The system should maintain a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run.

Whenever the CPU scheduler decides to execute a process, it calls the dispatcher.

The dispatcher checks to see whether the next process in the queue is in memory.

If not, and there is no free region, the dispatcher swaps out a process currently in memory and swap in the desired process.

The context switch time in such a swapping system is fairly high.

Let the size of the user process is 1 MB and the transfer rate of backing store is 5MB/sec.

So the actual transfer of the 1 MB process to or from memory takes 1/5 Sec= 200 milliseconds.

Assuming that the average latency is 8 milliseconds.
Therefore, the *swap in* as well as *swap out* takes 208 milliseconds. So the total swap time is (208×2) = 416 milliseconds.

Transfer time is the major part of swap time.

Swapping is constrained by other factors as well.

If we want to swap a process, we must be sure that it is completely idle.

If an I/O is asynchronously accessing the user memory for I/O buffers, then the process cannot be swapped.

Contiguous Memory Allocation

The memory is usually divided into two partitions: one for the operating system and one for the user processes.

Generally several user processes to reside in memory at the same time.

We therefore need to consider how to allocate available memory to the processes that are waiting to be brought into memory.

In contiguous memory allocation, each process is contained in a single section of memory that is contiguous.

Memory Protection

It is very necessary to protect operating system from user processes, and also protect user process from one another.

This protection can be provided by using a relocation register with a limit register.

The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600).

Each logical address must fall within the range specified by the limit register.

The MMU maps the logical address dynamically by adding the value in the relocation register.

This mapped address is sent to memory.

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch.

Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other user's programs and data from being modified by this running process.

The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically.

This flexibility is desirable in many situations.

For example, the operating system contains code and buffer space for device drivers.

If a device driver is not commonly in use, we do not want to keep the code and data in memory, and also we can use that space for other purpose.

This type of code is called transient operating system code; it comes and go as needed.

Therefore, this code changes the size of the operating system during program execution.

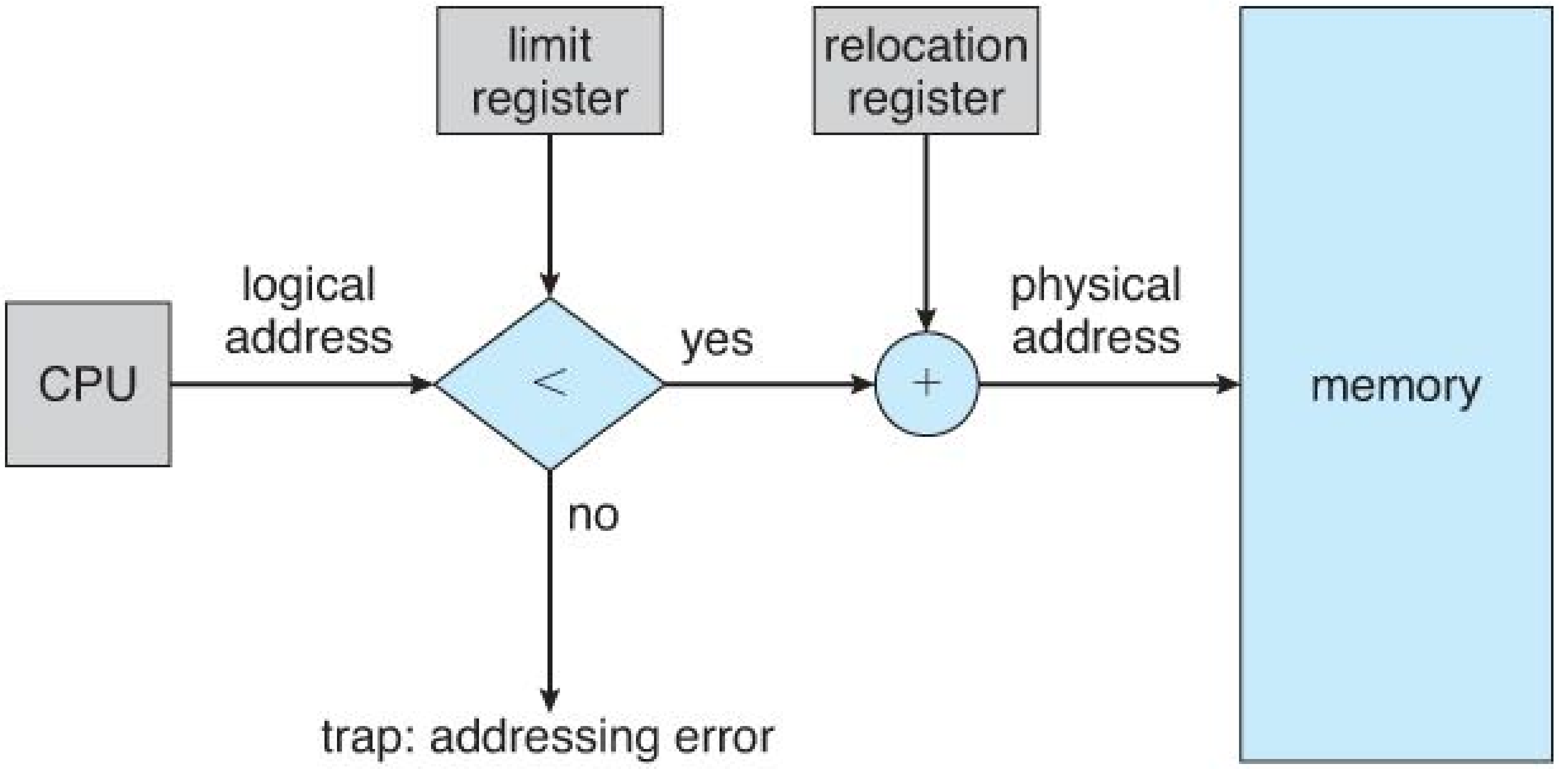


Fig 5: Hardware support for relocation and limit registers.

Memory Allocation

The simplest memory allocation method is to divide memory into several fixed-sized partitions, where each partition may contain exactly one process.

This method is known as ***multiple-partition*** method.

If there is a free partition, then a process is selected from the input queue and loaded into it.

When the process terminates, the partition is available for other process.

In this scheme the degree of multi-programming is bound by number of partitions.

But the drawback of this method is that, the size of a process may be greater than the size of a partition.

In this case, the method is unable to execute the process.

Size of all methods are not same, therefore there is a high chance of memory wastage.

This method is no longer used.

The operating system keeps a table indicating which parts of memory are available and which are occupied.

Initially, all memory is available for user processes and is considered one large block of available memory, a hole.

When a process arrives and needs memory, we search for a hole large enough for this process.

If we find one, we allocate only as much memory as needed, keeping the rest available to satisfy future requests.

At a given time, we have a list of available block sizes and input queue.

The operating system can order the input queue scheduling algorithm.

If there is no available block of memory (hole) is large enough to hold the process, it should wait.

The operating system can then skip down the input queue to see whether the smaller memory requirements of some other process can be meet.

Generally the memory blocks available comprise a set of holes of various sizes scattered throughout memory.

When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.

If the hole is too large, it is split into two parts.

One part is allocated to the arriving process; the other is returned to the set of holes.

When a process terminates, it releases its block of memory, which is then placed back in the set of holes.

If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

This procedure is a particular instance of the general dynamic storage allocation problem, which concerns how to satisfy a request of size *n* from a list of free holes.

There are many solutions to this problem. The first-fit, best-fit, and worst-fit strategies are the ones most commonly used to select a free hole from the set of available holes.

First fit:

Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended.

We can stop searching as soon as we find a free hole that is large enough.

Best fit:

Allocate the smallest hole that is big enough.

We must search the entire list, unless the list is ordered by size.

This strategy produces the smallest leftover hole.

Worst fit:

Allocate the largest hole.

Again, we must search the entire list, unless it is sorted by size.

This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

It can be observed that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization.

Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation.

As processes are loaded and removed from memory, the free memory space is broken into little pieces.

Fragmentation

Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation.

As processes are loaded and removed from memory, the free memory space is broken into little pieces.

External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes.

Statistical analysis of first fit, for instance, reveals that, even with some optimization, given *N* allocated blocks, another 0.5 *N* blocks will be lost to fragmentation.

That is, one-third of memory may be unusable!

This property is known as the **50-percent rule**.

Memory fragmentation can be internal as well as external.

We can also break the physical memory into fixed-sized blocks and allocate memory in units based on block size.

With this approach, the memory allocated to a process may be slightly larger than the requested memory.

The difference between these two numbers is internal fragmentation — unused memory that is internal to a partition.

One solution to the problem of external fragmentation is compaction.

The goal is to shuffle the memory contents so as to place all free memory together in one large block.

Compaction is not always possible; however, if relocation is static and is done at assembly or load time, compaction cannot be done.

It is possible only if relocation is dynamic and is done at execution time.

If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address.

When compaction is possible, we must determine its cost.

The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory.

This scheme can be expensive.

Paging

Paging is a memory management scheme that permits a process’s physical address space to be noncontiguous.

Paging avoids external fragmentation and the associated need for compaction, two problems that plague contiguous memory allocation.

Basic Method

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**.

When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store).

The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames.

Every address generated by the CPU is divided into two parts:

i) Page number (*p*)
ii) Page offset (*d*)

The page number is used as an index into a page table.

The page table contains the base address of each frame in physical memory, and the offset is the location in the frame being referenced.

Thus, the base address of the frame is combined with the page offset to define the physical memory address.

The following outlines the steps taken by the MMU to translate a logical address generated by the CPU to a physical address:

1. Extract the page number *p* and use it as an index into the page table.

2. Extract the corresponding frame number *f* from the page table.

3. Replace the page number *p* in the logical address with the frame number *f*.

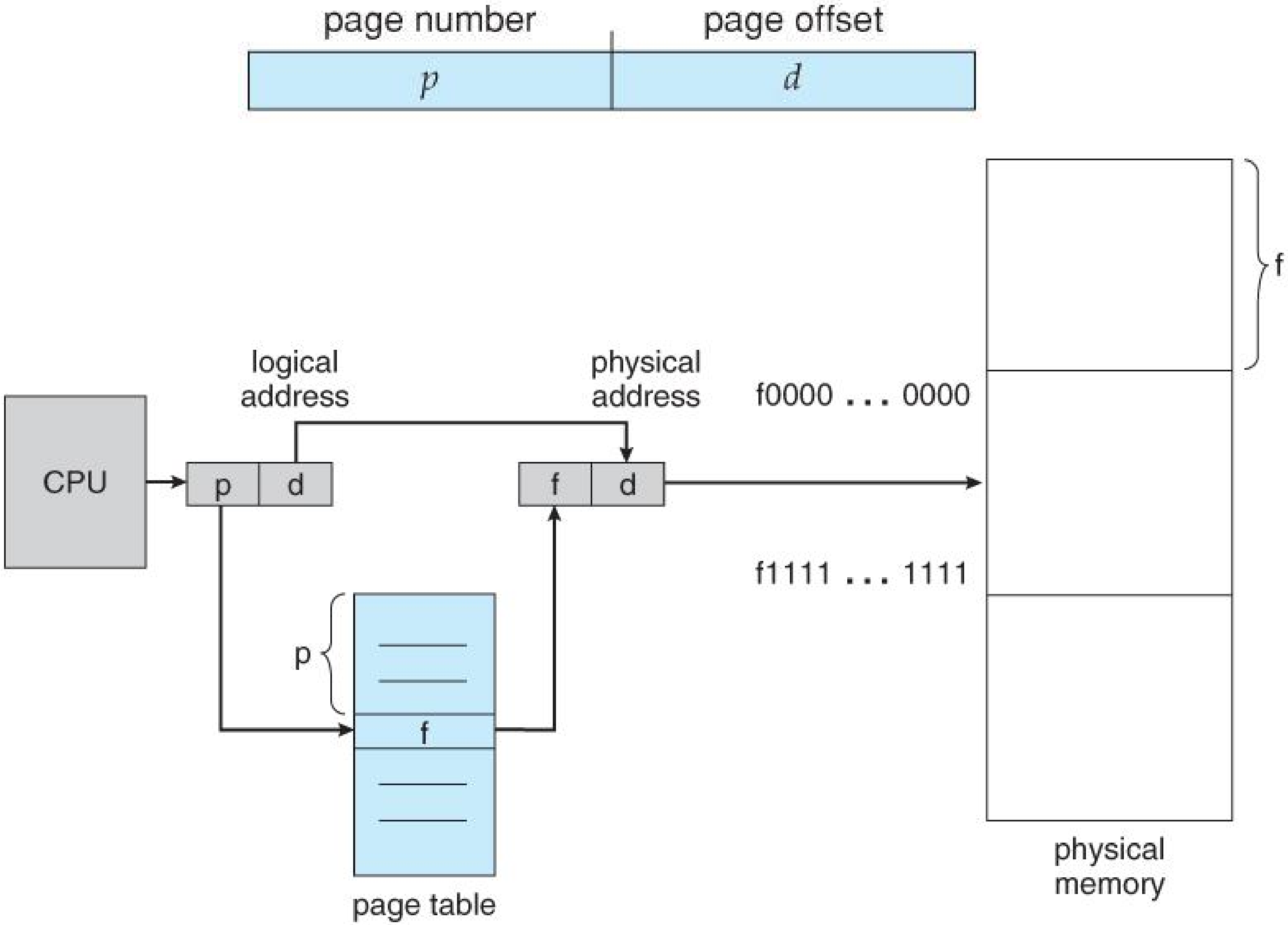


Fig 6: Paging hardware

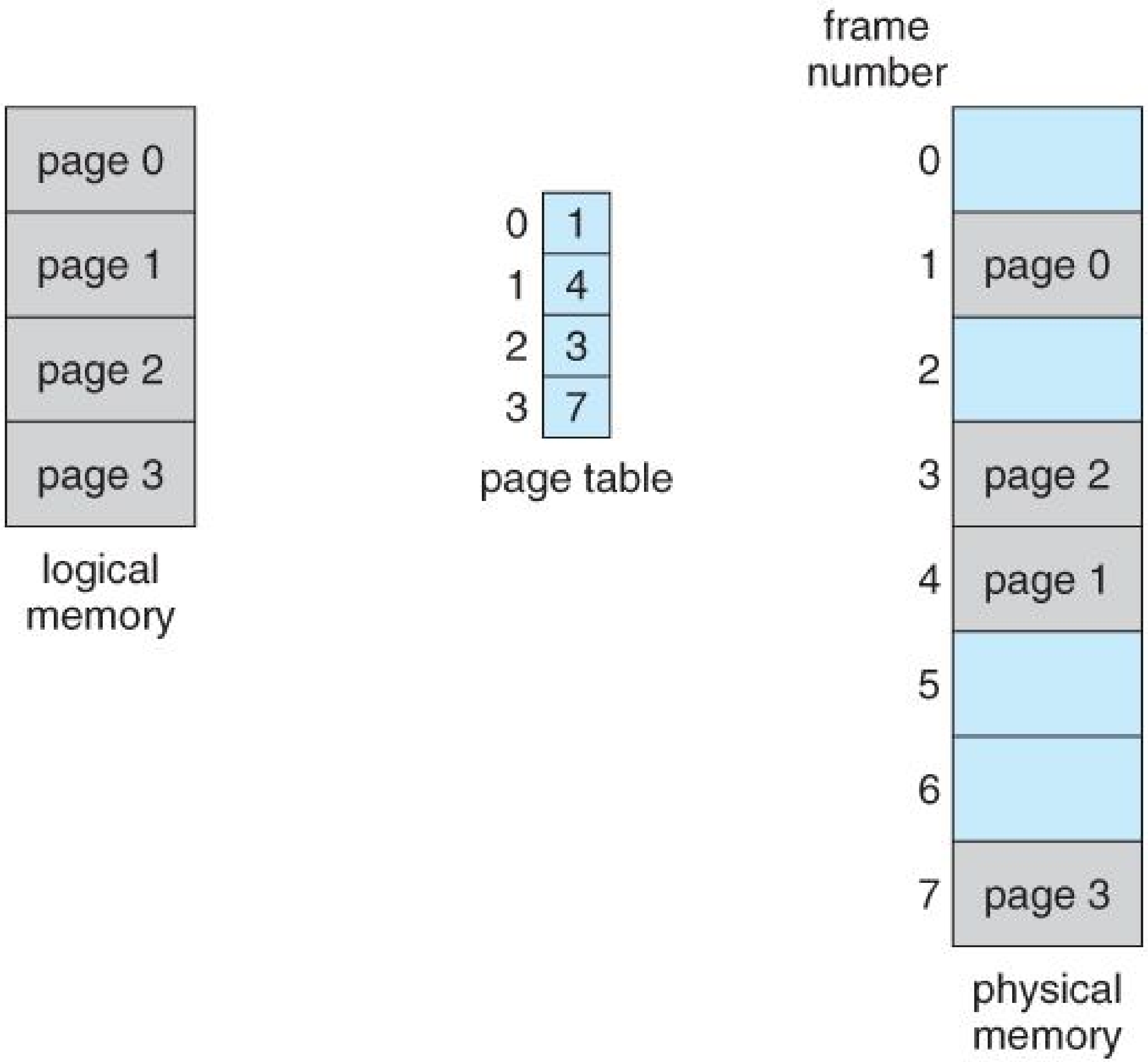


Fig 7: Paging model of logical and physical memory.

As the offset *d* does not change, it is not replaced, and the frame number and offset now comprise the physical address.

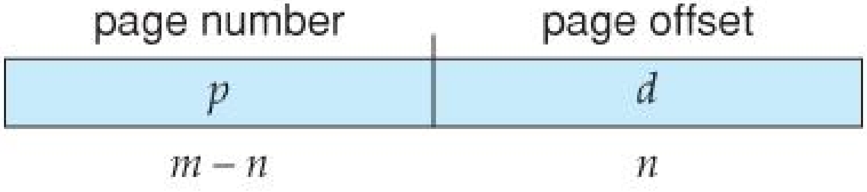
The page size (like the frame size) is defined by the hardware.

The size of a page is a power of 2, typically varying between 4 KB and 1 GB per page, depending on the computer architecture.

The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.

If the size of the logical address space is 2^m, and a page size is 2^n bytes, then the high-order *m-n* bits of a logical address designate the page number, and the *n* low-order bits designate the page offset.

Thus, the logical address is as follows:



where *p* is an index into the page table and *d* is the displacement within the page.

Let consider the paging example for a 32-byte memory with 4-byte pages shown in Fig 8.

Here, in the logical address, $n = 2$ and $m = 4$.

Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the programmer's view of memory can be mapped into physical memory.

Logical address 0 is page 0, offset 0.

Indexing into the page table, we find that page 0 is in frame 5.

Thus, logical address 0 maps to physical address [(5 × 4) +0]=20.

Logical address 3 (page 0, offset 3) maps to physical address [(5 × 4) +3]=23.

Similarly, the logical address 4 (page 1, offset 0) maps to physical address [(6× 4) + 0]=24.

Logical address 13 (page 3, offset 1) maps to physical address [(2× 4) + 1]=9.

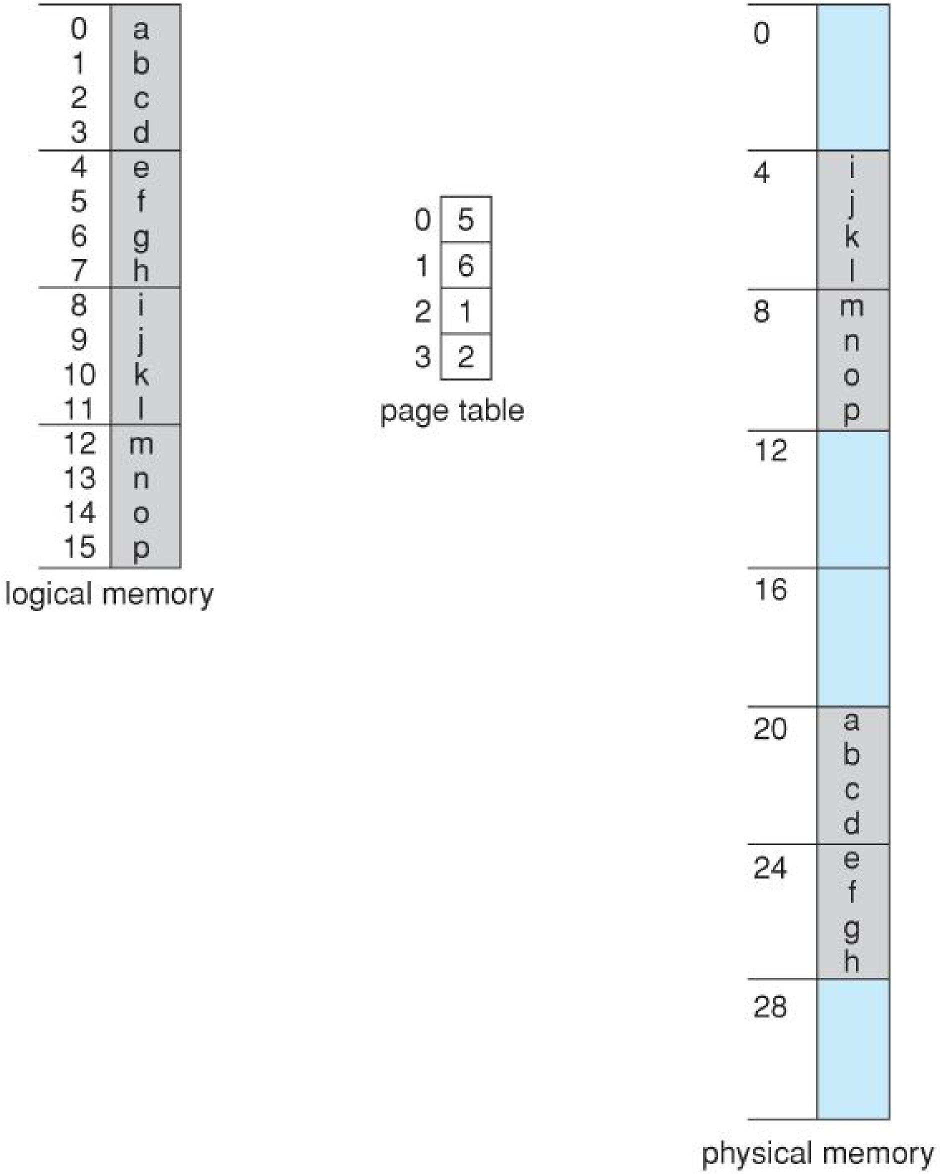


Fig 8: Paging example for a 32-byte memory with 4-byte pages.

In paging scheme, there is no external fragmentation; any free frame can be allocated to a process that needs it.

However, we may have some internal fragmentation.

It may be noted that frames are allocated as units.

If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full.

For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes.

It will be allocated 36 frames, resulting in internal fragmentation of (2,048 −1,086) = 962 bytes.

In the worst case, a process would need *n* pages plus 1 byte.

It would be allocated *n* + 1 frames, resulting in internal fragmentation of almost an entire frame.

If process size is independent of page size, we expect internal fragmentation to average one-half page per process.

This consideration suggests that small page sizes are desirable.

However, overhead is involved in each page table entry, and this overhead is reduced as the size of the pages increases.

Generally, page sizes have grown over time as processes, data sets, and main memory have become larger.

A 32-bit entry can point to one of 2^{32} physical page frames.

If the frame size is 4 KB (2^{12}), then a system with 4-byte (32 bit) entries can address 2^{44} bytes (or 16 TB) of physical memory.

When a process arrives in the system to be executed, its size, expressed in pages, is examined.

Each page of the process needs one frame.

Thus, if the process requires n pages, at least n frames must be available in memory.

If *n* frames are available, they are allocated to this arriving process.

The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process.

The next page is loaded into another frame, its frame number is put into the page table, and so on.

An important aspect of paging is the clear separation between the programmer’s view of memory and the actual physical memory.

The programmer views memory as one single space, containing only this one program.

In fact, the user program is scattered throughout physical memory, which also holds other programs.

The difference between the programmer’s view of memory and the actual physical memory is reconciled by the address-translation hardware.

The logical addresses are translated into physical addresses.

This mapping is hidden from the programmer and is controlled by the operating system.

Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory—which frames are allocated, which frames are available, how many total frames there are, and so on.

This information is generally kept in a single, system-wide data structure called a frame table.

The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process (or processes).

The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents.

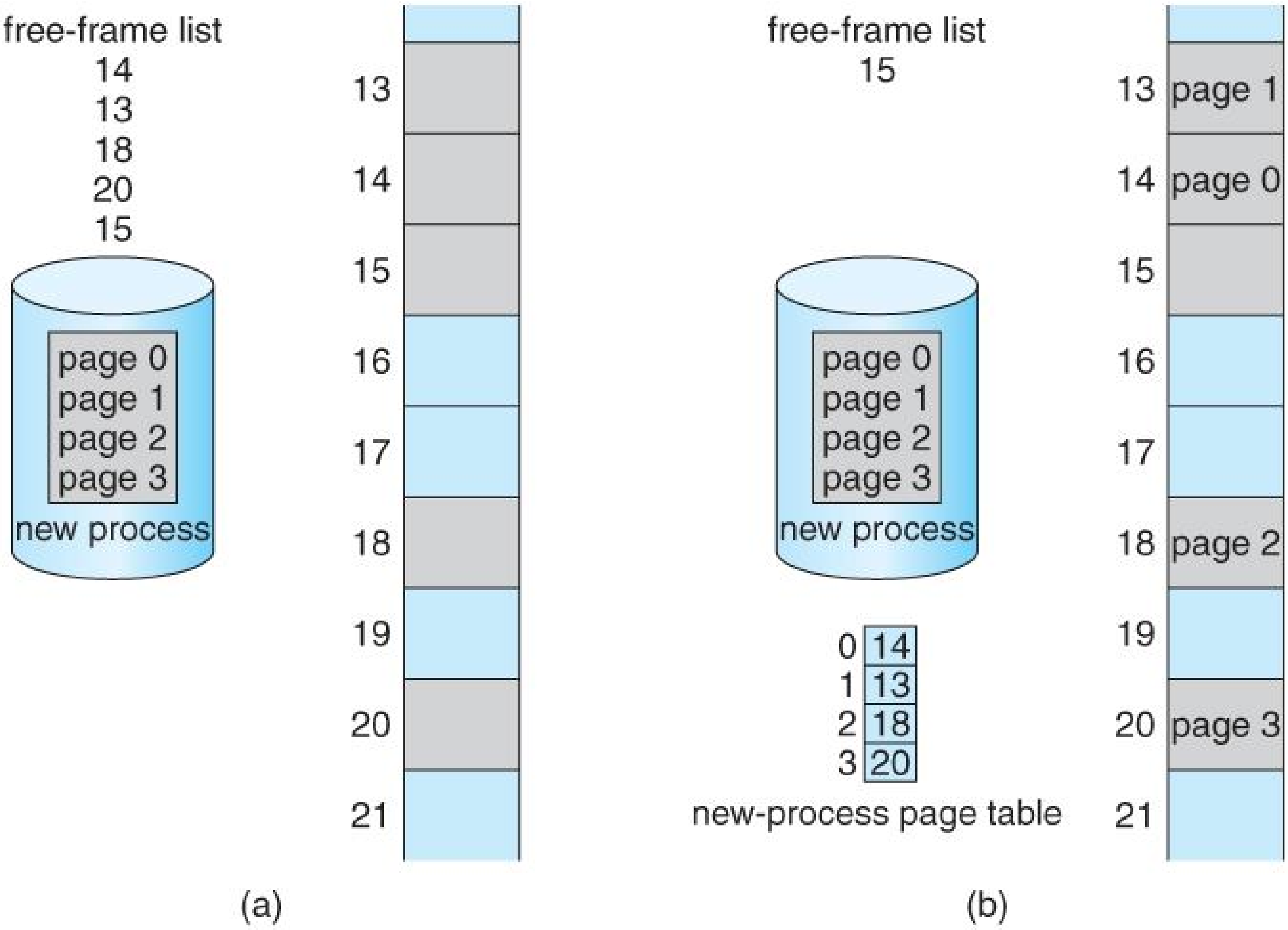


Fig 9: Free frames (a) before allocation and (b) after allocation.

This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually.

It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU.

Paging therefore increases the context-switch time.