Memory Management

Main memory is a very important component in a computer system because both the operating system and various user applications have to be loaded into main memory first before being able to be executed. The efficiency of memory management to much extent affects the efficiency of the whole system. This part first explores the requirements for memory management and then various approaches are introduced.

1 Memory management requirements

By examining the nature of the execution of programs, the following requirements are to be satisfied.

1.1 Address mapping and relocation

After a process is created and loaded into main memory, it is ready to run. For the operating system to dispatch it, the following information must be known for access, as Figure 1: the locations of process control information, the execution stack, and code entry.

However these are not enough. Within a program, there may be also memory references in various instructions, e.g. `jmp somewhere`, where `somewhere` is a reference to a memory location which contains the next instruction to be executed, or `move AX, [somewhere_else]`, where `somewhere_else` is the memory location in which the value is supposed to be moved into `AX`. Before the program is loaded, these references are usually the relative addresses to the entry point of the program, which is 0. These addresses are also called *logical addresses*, which make up a *logical address space*. After the loading of the program to a location in the main memory, some facility must be available to support transforming the logical addresses into *physical addresses*, which are the real addresses of main memory pointing to destination data or instructions and makes up the *physical address space*. Thus somehow the processor and the operating system must be able to translate logical addresses into physical ones, reflecting the current location of the program, as Figure 2.

The translation may be performed in different ways at different moments, as follows:

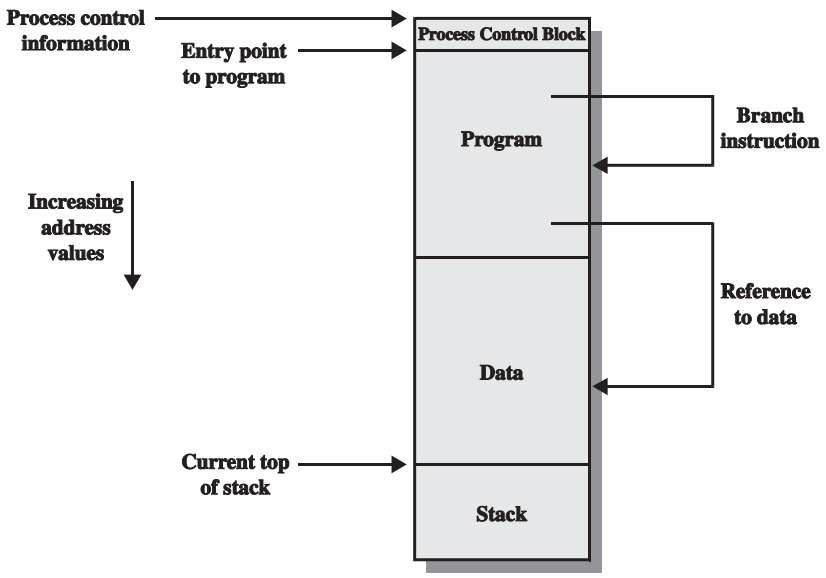

Figure 1: Addressing requirements for a process

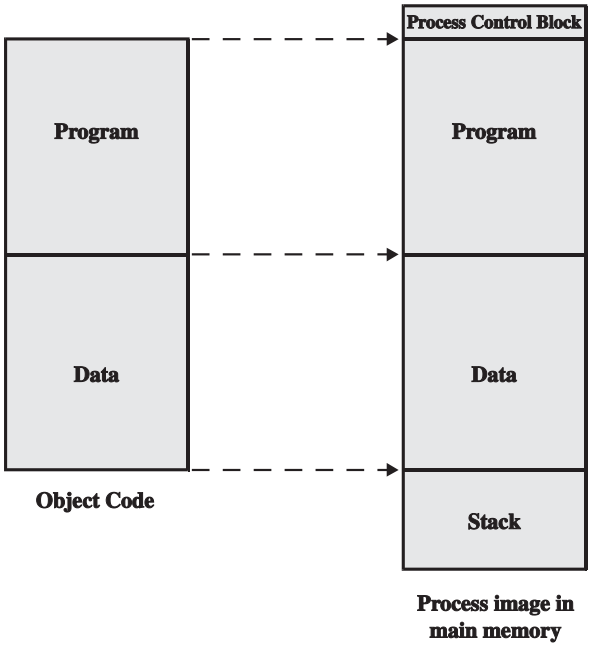

Figure 2: Address translation

- **Compiler time**: If it is known in advance that a program will reside at a specific location of main memory, then the compiler may be told to build the object code with absolute addresses right away. For example, the boot sect in a bootable disk may be compiled with the starting point of code set to 007C:0000.
- **Load time**: It is pretty rare that we know the location a program will be assigned ahead

of its execution. In most cases, the compiler must generate relocatable code with logical addresses. Thus the address translation may be performed on the code during load time. Figure 3 shows that a program is loaded at location x. If the whole program resides on a monolithic block, then every memory reference may be translated to be physical by added to x.

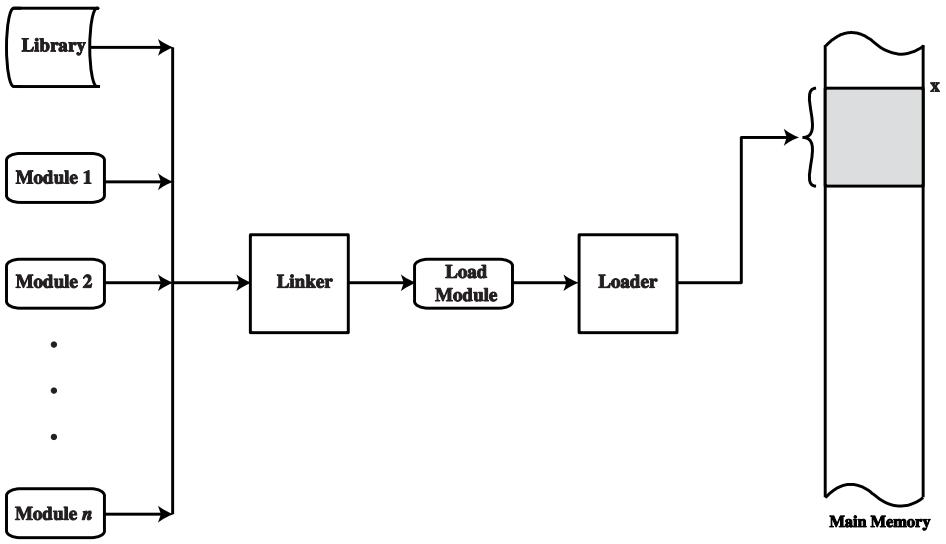

Figure 3: A loading scenario

- **Execution time**: The load-time translation can solve the problem in some sense, however in some cases, a process has to be moved from one location to another during execution. For example, a process may be swapped out to virtual memory area in the disk so that another process may be loaded. When it is swapped back into the main memory, it is painful to require that it resides at the previous location, since the location may still be occupied by another process. Thus the process needs to be *relocatable*. In this case, the translation is done dynamically during the execution time: Easy memory reference remains to be logical, and when it is to be involved in the execution, the translation facility works on it and generate the physical address. Note that the generated address does not replace the original logical address one. Most general-purpose operating systems use this method.

1.2 Protection and sharing

Memory references are valid if pointing to locations that belong to the current process itself, however usually prohibited if referring to locations in other processes, whether accidental or intentional. Note that the memory protection requirement needs help from the processor

rather than the operating system, since when a process occupies the processor, the operating system can hardly control it, say checking the validity of memory references.

Any protection mechanism must allow several processes to access the same portion of main memory. For example, multiple processes may use a same system library and it is natural to load one copy of the library in main memory and let it shared by those processes.

1.3 Application organization

As we know, main memory is organized as a linear, or one-dimensional, address space, while most programs are organized into modules. Whether regarding sharing or protection, it is natural for the users to express control in terms of modules, thus some supporting mechanism is needed. The segmentation method we will cover later on responds to this requirement.

1.4 Two-level Memory organization

We have discussed memory hierarchy which includes both fast expensive memories and slow but cheap storage devices. Typically two levels of storage are present, main memory and disks. The former is fast, but usually cannot provide enough space for concurrent processes. In this case, hard disks are used for the operating system to swap out inactive processes, which may later be swapped back into the main memory. Thus some facilities are required to swap processes. Issues relating to *virtual memory* will be addressed in the next chapter.

2 Memory partitioning

Allocating a single contiguous section of memory to each process is the most primitive method of memory management, usually called *partitioning*. It has been used in some now-obsolete operating systems. Each single section is called a *partition*.

2.1 Fixed partitioning

The simplest partitioning method is dividing memory into several fixed-sized partitions in advance, called *fixed partitioning*. Each partition may contain exactly one process. As Figure 4 (a) shows, a 64M memory is divided into 8 equal-size partitions with 8 megabytes each. Any process whose size is less than or equal to the partition size can be loaded into any partition available. And when no partition is available and a new process is to be loaded, a process already residing in main memory may be selected to be swapped out to free a partition. So

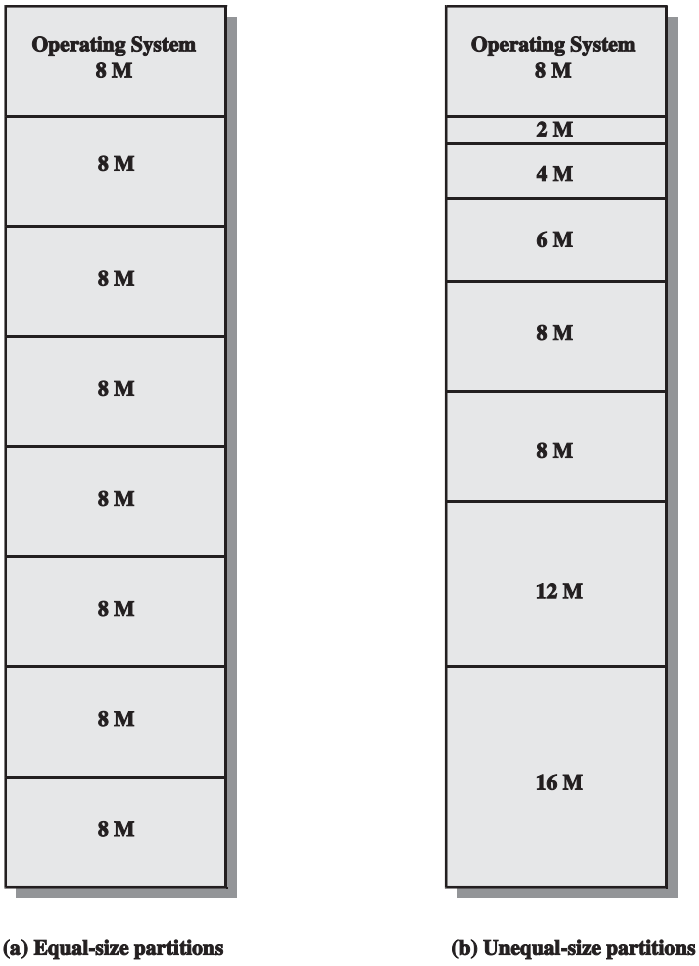

Figure 4: Example of fixed partitioning of a 64-Megabyte memory

the placement algorithm for this method is very simple. However two disadvantages are present with it:

- A program that is too big to be held in a partition needs some special design, called *overlay*, which brings heavy burden on programmers. With overlay, a process consists of several portions with each being mapped to the same location of the partition, and at any time, only one portion may reside in the partition. When another portion is referenced, the current portion will be switched out.
- A program may be much smaller than a partition, thus space left in the partition will be wasted, which is referred to as *internal fragmentation*.

As an improvement shown in Figure 4 (b), unequal-size partitions may be configured in main memory so that small programs will occupy small partitions and big programs are also likely to be able to fit into big partitions. Although this may solve the above problems with fixed

equal-size partitioning to some degree, the fundamental weakness still exists: The number of partitions are the maximum of the number of processes that could reside in main memory at the same time. When most processes are small, the system should be able to accommodate more of them but fails to do so due to the limitation. More flexibility is needed.

2.2 Dynamic partitioning

To overcome difficulties with fixed partitioning, partitioning may be done dynamically, called *dynamic partitioning*. With it, the main memory portion for user applications is initially a single contiguous block. When a new process is created, the exact amount of memory space is allocated to the process. Similarly when no enough space is available, a process may be swapped out temporarily to release space for a new process. The way how the dynamic partitioning works is illustrated in Figure 5.

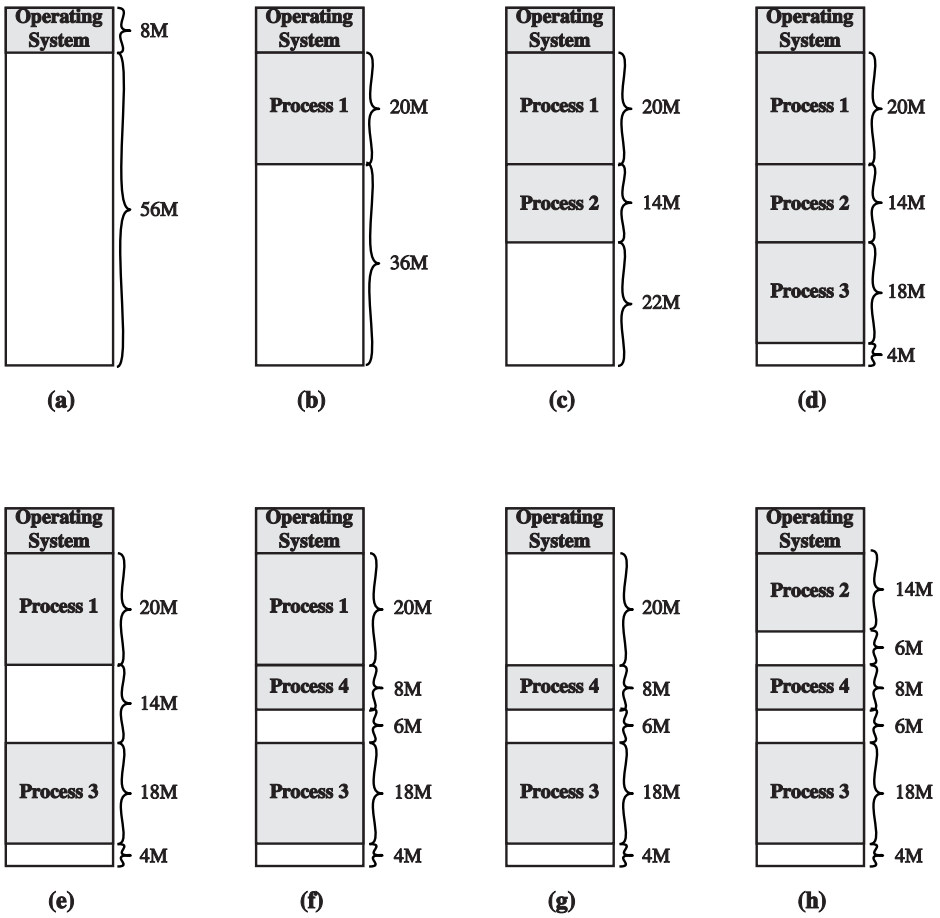

Figure 5: The effect of dynamic partitioning

As time goes on, there will appear many small holes in the main memory, which is referred to

as *external fragmentation*. Thus although much space is still available, it cannot be allocated to new processes. A method for overcoming external fragmentation is *compaction*. From time to time, the operating system moves the processes so that they occupy contiguous sections and all of the small holes are brought together to make a big block of space. The disadvantage of compaction is: The procedure is time-consuming and requires relocation capability.

Address translation

Figure 6 shows the address translation procedure with dynamic partitioning, where the processor provides hardware support for address translation, protection, and relocation.

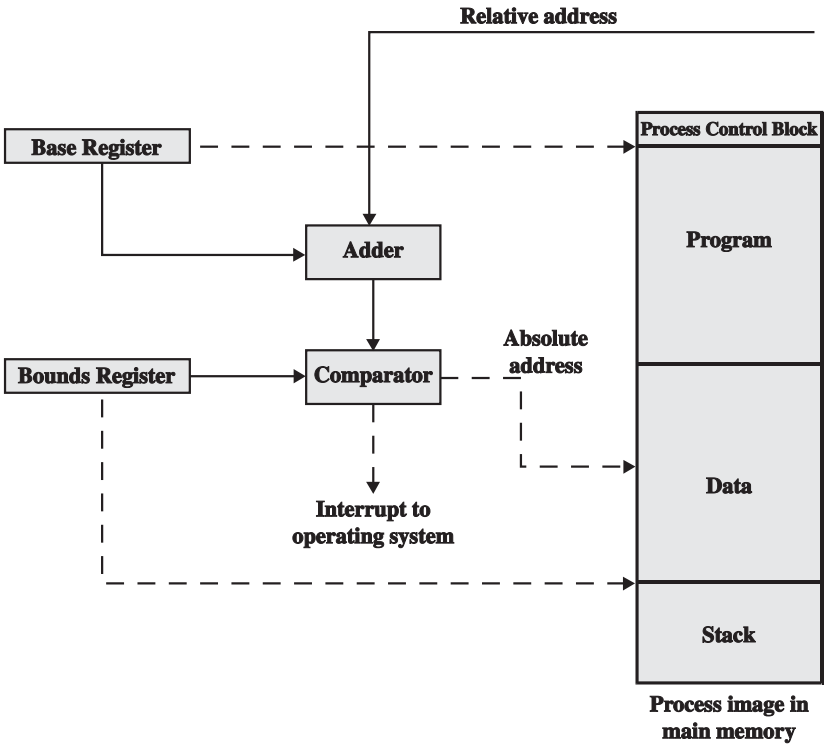

Figure 6: Address translation with dynamic partitioning

The *base register* holds the entry point of the program, and may be added to a relative address to generate an absolute address. The *bounds register* indicates the ending location of the program, which is used to compare with each physical address generated. If the later is within bounds, then the execution may proceed; otherwise, an interrupt is generated, indicating illegal access to memory.

The relocation can be easily supported with this mechanism with the new starting address and ending address assigned respectively to the base register and the bounds

register.

Placement algorithm

Different strategies may be taken as to how space is allocated to processes:

- **First fit**: Allocate the *first* hole that is big enough. Searching may start either at the beginning of the set of holes or where the previous first-fit search ended.
- **Best fit**: Allocate the *smallest* hole that is big enough. The entire list of holes must be searched unless it is sorted by size. This strategy produces the smallest leftover hole.
- **Worst fit**: Allocate the largest hole. In contrast, this strategy aims to produce the largest leftover hole, which may be big enough to hold another process.

Experiments have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization.

3 Paging

3.1 Pages and frames

Both fixed-sized and variable-sized partitions are inefficient due to the problem of either internal or external fragmentation. The fundamental reason of the inefficiency is that each program is allocated a monolithic section of memory. To permit the process image in the main memory to be noncontiguous, we break the main memory into fixed-sized blocks called *frames* and break the process image into blocks of the same size called *pages*. When a process is to be executed, its pages are loaded into any available frames, as illustrated in Figure 7. The frames allocated to a single process may be contiguous or not.

3.2 Page table and address translation

To transform the logical addresses into physical ones, a simple base address register will no longer suffice. Instead, a *page table* is needed. As Figure 8, each entry in a page table associates a page with a frame. The number of the frame holding page i is

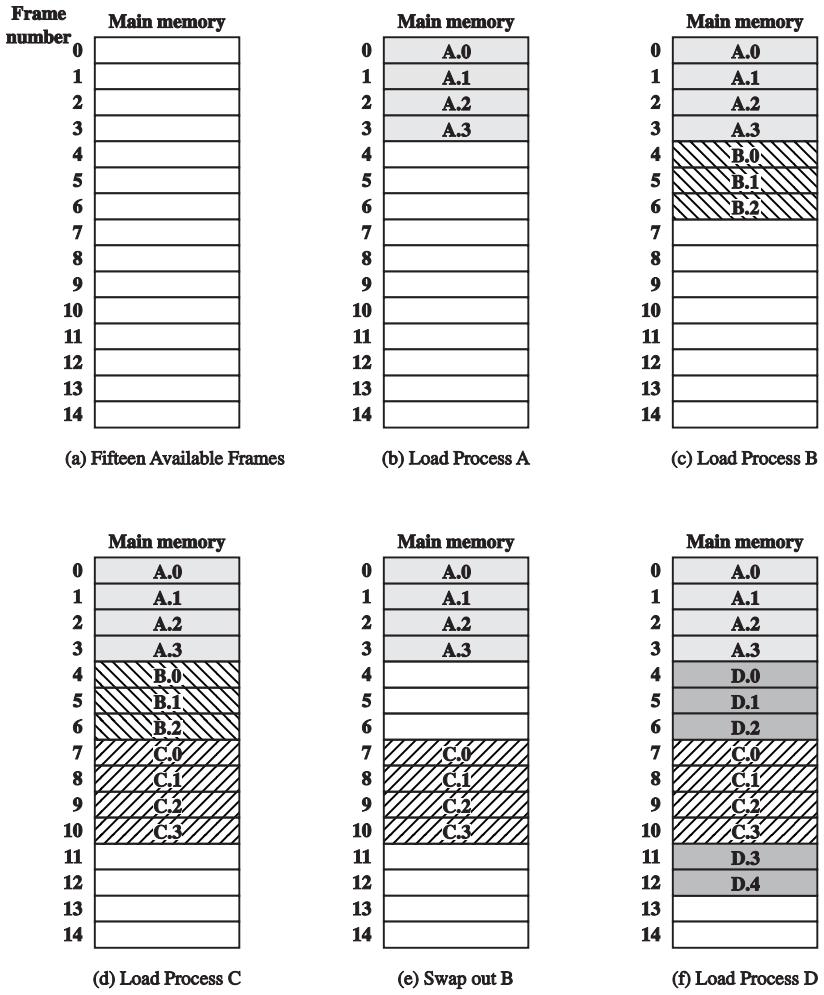

Figure 7: Assignment of process pages to free frames

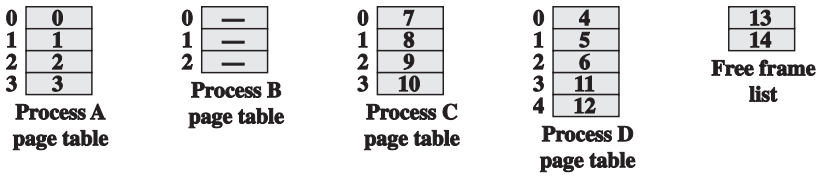

Figure 8: Page tables for processes in Figure 7 at Time (f)

available in item i of the page table. Accordingly, a list of free frames is maintained by the operating system.

To facilitate the address translation, the frame or page size is always a power of 2 so that a relative address in the logical address space can be easily split into two parts: the page number and the offset address. For example, suppose 16-bit addresses are used, and the page size is 1K = 1024 bytes. Thus 10 bits are needed as an offset

in a page, leaving 6 bits for the page number. As Figure 9 (b) shows, the relative address 1502, which in binary form is 0000010111011110, responds to an offset of 478 (0111011110) on page 1 (000001). Using a page size that is a power of 2 thus makes it easier not only for a compiler/linker to generate logical addresses prepared for this paging scheme, but also for the hardware to support the translation.

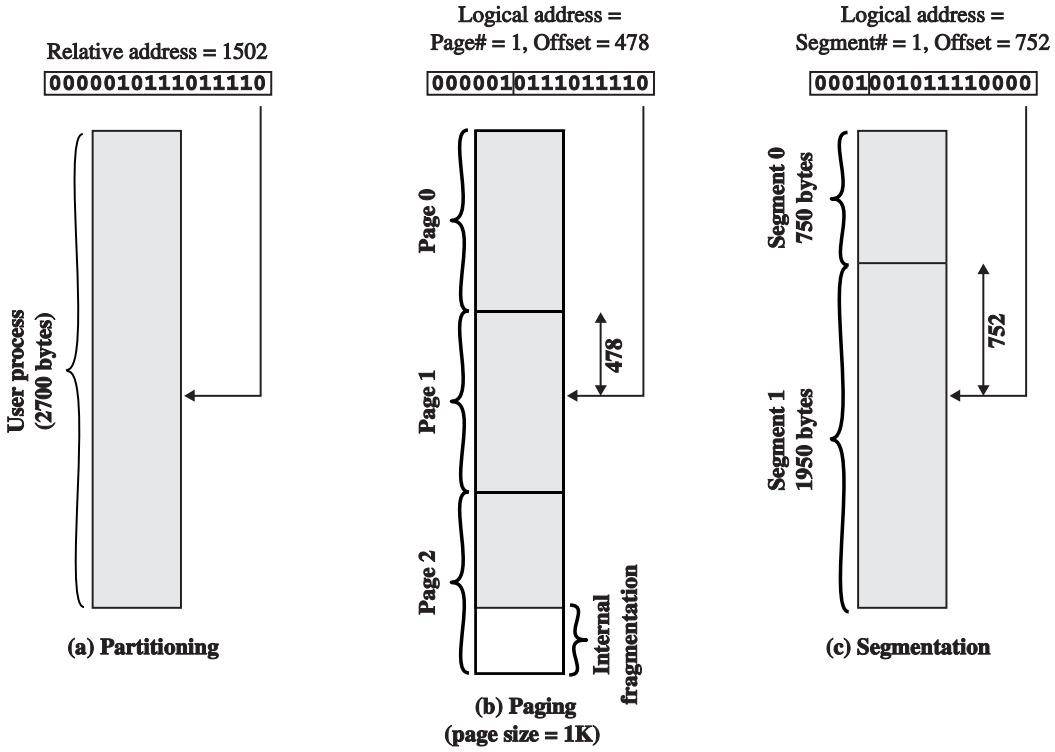

Figure 9: Logical addresses

Consider an address of $n+m$ bits, where the leftmost n bits specifies the page number and the rightmost m bits are the offset address, then as Figure 10 (a) illustrates, the following steps are needed for address translation:

1. Extract the leftmost n bits of the logical address as the page number.
2. Use the page number as an index into the process page table to find the corresponding frame number, k.
3. The starting physical address of the frame is $k \times 2^m$, and the physical address of the referenced byte is that number plus the offset, which may be easily constructed by appending the offset to the frame number.

Paging makes sharing easier by simply associating the frames holding the shared code with the pages in each process involved.

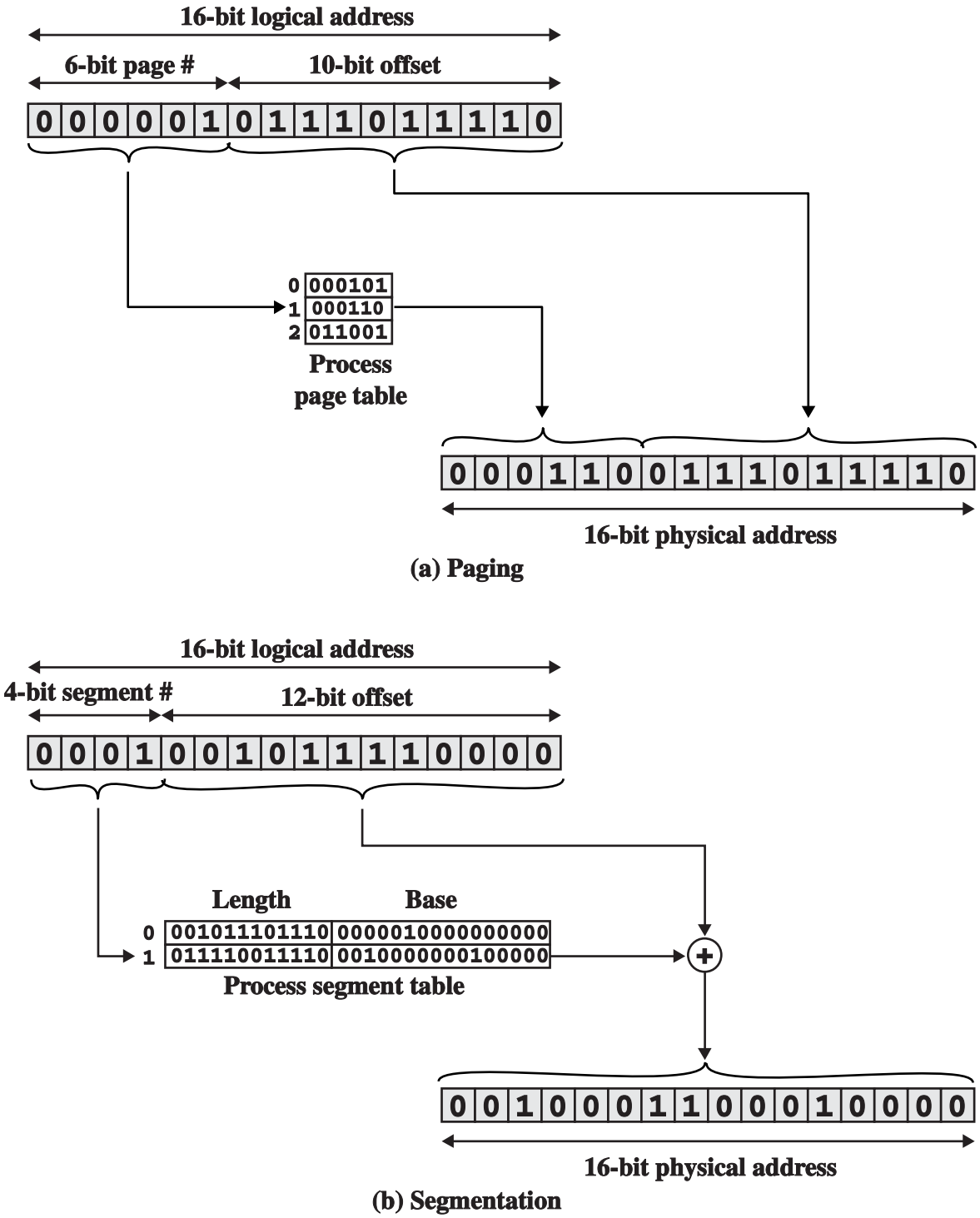

Figure 10: Address translation with paging and segmentation

3.3 Hardware support

Each computer system has its own methods for supporting paging, whose major part is how page tables are implemented. The page tables can be done in several ways:

- A set of registers may be dedicated to an extreme fast but small page table. For example, in the DEC PDP-11 system, the address consists of 16 bits, the page size is 8KB. The page table thus consists of only eight entries, which may feasibly stored in registers. The contents of all these registers are stored in the PCB when a process loses the processor so that they may be resumed again.
- To allow large page tables, a section of main memory may be used and a *page-table base register* points to it. Thus when the context is switched and a new process is dispatched, changing page tables requires changing only this one register.

  The problem with this scheme is the time required to access a location in the process image. For any access, two accesses to the main memory is needed, first the frame number in the page table and second the desired destination location. Since the access is so frequent that the delay like this is intolerable. A solution to this problem is to use a small but fast cache, called *translation look-aside buffer* (*TLB* for short). Each entry in the TLB consists of two parts: a key for the page number, and a value for the corresponding frame number. When an address needs to be translated, the page number may be compared simultaneously with all keys in the TLB. If the number is found, then the corresponding value field is returned so that the physical address is obtained.

4 Segmentation

Paging is in some sense similar to fixed partitioning since each page is of fixed size, but is different since multiple pages may be used instead of one single partition in the latter. Interestingly, another memory management approach, *segmentation*, is similar to dynamic partitioning. With segmentation, a program is divided into several *segments*, each similar to a partition of varied size.

Segmentation avoids internal fragmentation which are present in both fixed partitioning and paging, but like dynamic partitioning, it suffers from external fragmentation. However the problem is not that serious because a process may be broken into a number of smaller pieces and the resulting external holes will be much smaller.

Different from paging, which is invisible to the programmer and the compiler, segmentation, is usually visible, which is actually based on the programmers' logical view of programs. Typically, the programmers will assign programs and data to different segments, thus leading to a major advantage of segmentation that the protection and sharing may be easily supported.

With segmentation, the logical addresses and physical addresses do not have a simple relationship any more like with partitioning and paging. Each logical address is explicitly expressed by a two tuple: ¡segment-number, offset¿, as illustrated by Figure 9 (c). The operating system maintains a segment table for each process and a list of free blocks of main memory. As Figure 10 (b) shows, each segment table entry would have to give the starting address in main memory of the corresponding segment, as well as the length of the segment to assure that invalid addresses are not used.

Paging and segmentation nowadays have been combined to eliminate both external and internal fragmentation, and facilitate programmers's control on programs. An example for this is the architecture of the Intel 80x86.