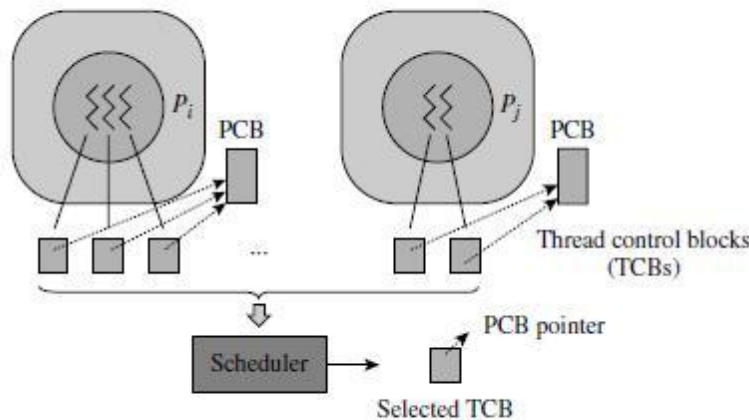
## $2^{nd}$ internals Solution

1. a) Explain with neat diagrams, i) User threads ii) Kernel level threads. (8M)

**Kernel-Level Threads**

A kernel-level thread is implemented by the kernel. Hence creation and termination of kernel-level threads, and checking of their status, is performed through system calls. Figure 3.14 shows a schematic of how the kernel handles kernel-level threads. When a process makes a *create_thread* system call, the kernel creates a thread, assigns an id to it, and allocates a thread control block (TCB). The TCB contains a pointer to the PCB of the parent process of the thread.




Scheduling of kernel-level threads.

TCB to check whether the selected thread belongs to a different process than the interrupted thread. If so, it saves the context of the process to which the interrupted thread belongs, and loads the context of the process to which the selected thread belongs. It then dispatches the selected thread. However, actions to save and load the process context are skipped if both threads belong to the same process. This feature reduces the switching overhead, hence switching between kernel-level threads of a process could be as much as an order of magnitude faster, i.e., 10 times faster, than switching between processes.

**Advantages and Disadvantages of Kernel-Level Threads**

A kernel-level thread is like a process except that it has a smaller amount of state information. This similarity is convenient for programmers—programming for threads is no different from programming for processes. In a multiprocessor system, kernel-level threads provide parallelism, as many threads belonging to a process can be scheduled simultaneously, which is not possible with the user-level threads described in the next section, so it provides better computation speedup than user-level threads.

**User-Level Threads**

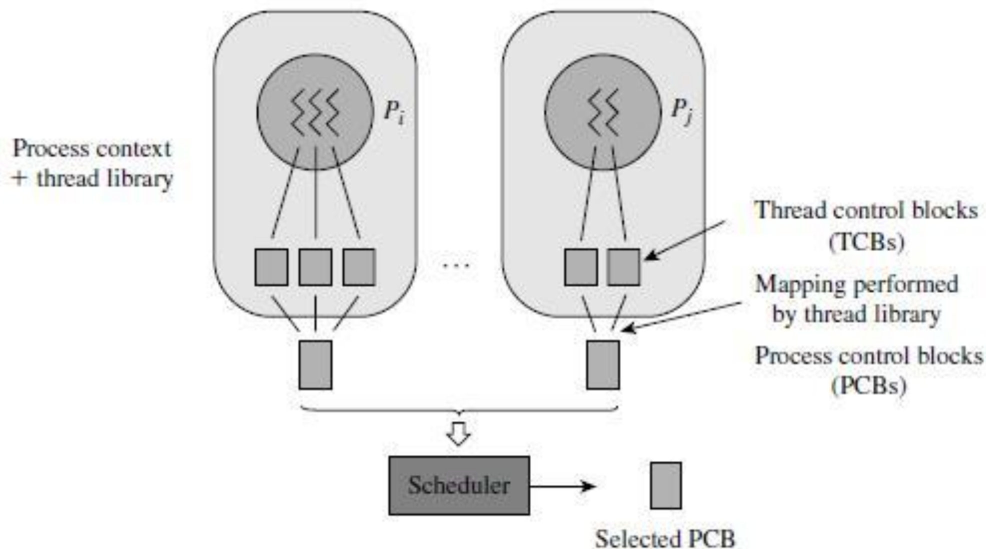
User-level threads are implemented by a *thread library*, which is linked to the code of

a process. The library sets up the thread implementation arrangement shown in Figure without involving the kernel, and itself interleaves operation of threads in the process. Thus, the kernel is not aware of presence of user-level threads in a process; it sees only the process. Most OSs implement the pthreads application program interface provided in the IEEE POSIX standard in this manner.

**Scheduling of User-Level Threads**

Figure below is a schematic diagram of scheduling of user-level threads. The thread library code is a part of each process. It performs ―scheduling‖ to select a thread, and organizes its execution. We view this operation as ―mapping‖ of the TCB of the selected thread into the
PCB of the process.

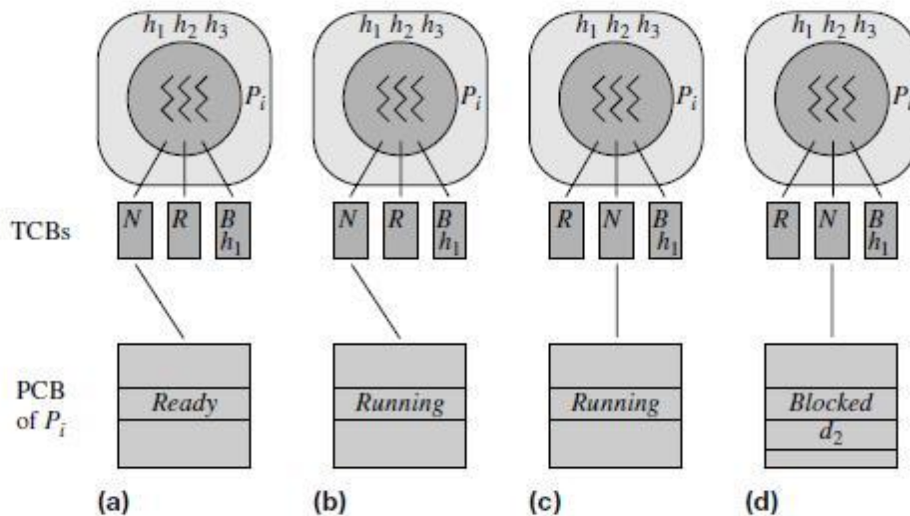
The thread library uses information in the TCBs to decide which thread should operate at any time. To ―dispatch‖ the thread, the CPU state of the thread should become the CPU state of the process, and the process stack pointer should point to the thread's stack. Since the thread library is a part of a process, the CPU is in the user mode. Hence a thread cannot be dispatched by loading new information into the PSW; the thread library has to use nonprivileged instructions to change PSW contents. Accordingly, it loads the address of the thread's stack into the stack address register, obtains the address contained in the *program counter* (PC) field of the thread's CPU state found in its TCB, and executes a branch instruction to transfer control to the instruction which has this address.




Scheduling of user-level threads.

**Advantages and Disadvantages of User-Level Threads**

Thread synchronization and scheduling is implemented by the thread library. This arrangement avoids the overhead of a system call for synchronization between threads, so the thread switching overhead could be as much as an order of magnitude smaller than in kernel-level threads.




Actions of the thread library (*N*,*R*,*B* indicate *running*, *ready*, and *blocked*).

This arrangement also enables each process to use a scheduling policy that best suits its nature. A process implementing a real-time application may use priority-based scheduling of its threads to meet its response requirements, whereas a process implementing a multithreaded server may perform round-robin scheduling of its threads. However, performance of an application would depend on whether scheduling of user-level threads performed by the thread library is compatible with scheduling of processes performed by the kernel.

For example, round-robin scheduling in the thread library would be compatible with either round-robin scheduling or priority-based scheduling in the kernel, whereas priority-based scheduling would be compatible only with priority-based scheduling in the kernel.

**b) Page tables are stored in a memory that has an access time of 100ns. The TLB can hold 64page table entries and has an access time of 10ns. During operation of a process, it is found that 85% of the time a required page table entry exists in the TLB and only 2% of the references lead to page faults. The average time for page replacement is 2ms. Compute the effective memory access time.** (2M)

E.M.A.T $= pr_2 * (t_{TLB} + t_{mem}) + (pr_1 - pr_2)(t_{TLB} + 2 * t_{mem}) + (1 - pr_1)(t_{TLB} + t_{mem} + t_{phf} + t_{TLB} + 2 * t_{mem})$

$= 0.85*(10+100)+(0.98-0.85)(10+2*100)+(1-0.98)(10+100+2000000+10+2*100)$ns

$= 40.01\mu s$

**3. a) Explain with a neat diagram, the different states and transitions of process in UNIX Operating system. (6M)**

A process in the *running* state is put in the *ready* state the moment its execution is interrupted. A system process then handles the event that caused the interrupt. If the running process had itself caused a software interrupt by executing an <*SI_instrn*>, its state may further change to *blocked* if its request cannot be granted immediately. In this model a user process executes only user code; it does not need any special privileges. A system process may have to use privileged instructions like I/O initiation and setting of memory protection information, so the

system process executes with the CPU in the kernel mode. Processes behave differently in the Unix model. When a process makes a system call, the process itself proceeds to execute the kernel code meant to handle the system call. To ensure that it has the necessary privileges, it needs to execute with the CPU in the kernel mode. A mode change is thus necessary every time a system call is made. The opposite mode change is necessary after processing a system call. Similar mode changes are needed when a process starts executing the interrupt servicing code in the kernel because of an interrupt, and when it returns after servicing an interrupt.

The Unix kernel code is made reentrant so that many processes can execute it concurrently. This feature takes care of the situation where a process gets blocked while executing kernel code, e.g., when it makes a system call to initiate an I/O operation, or makes a request that cannot be granted immediately. To ensure reentrancy of code, every process executing the kernel code must use its own kernel stack. This stack contains the history of function invocations since the time the process entered the kernel code. If another process also enters the kernel code, the history of its function invocations will be maintained on its own kernel stack. Thus, their operation would not interfere. In principle, the kernel stack of a process need not be distinct from its user stack; however, distinct stacks are used in practice because most computer architectures use different stacks when the CPU is in the kernel and user modes. Unix uses two distinct running states. These states are called user running and kernel running states. A user process executes user code while in the user running state, and kernel code while in the kernel running state. It makes the transition from user running to kernel *running* when it makes a system call, or when an interrupt occurs. It may get blocked while in the *kernel running* state because of an I/O operation or non availability of a resource.

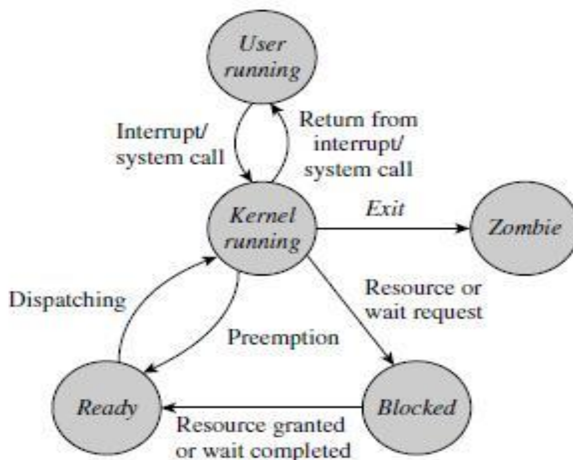



Figure 3.19 Process state transitions in Unix.

**b) List the different types of process interaction and explain them in brief. (4M)**

| Kind of interaction | Description |
|---|---|
| Data sharing | Shared data may become inconsistent if several processes modify the data at the same time. Hence processes must interact to decide when it is safe for a process to modify or use shared data. |
| Message passing | Processes exchange information by sending messages to one another. |
| Synchronization | To fulfill a common goal, processes must coordinate their activities and perform their actions in a desired order. |
| Signals | A signal is used to convey occurrence of an exceptional situation to a process. |

**4. Explain the important concepts in the operation of demand paging. (10M)**

**Demand Paging Preliminaries**

To implement demand paging, a copy of the entire logical address space of a process is maintained on a disk. The disk area used to store this copy is called the *swap space* of a process. While initiating a process, the virtual memory manager allocates the swap space for the process and copies its code and data into the swap space. During operation of the process, the virtual memory manager is alerted when the process wishes to use some data item or instruction that is located in a page that is not present in memory. It now loads the page from the swap space into memory. This operation is called a *page-in* operation. When the virtual memory manager decides to remove a page from memory, the page is copied back into the swap space of the process to which it belongs if the page was modified since the last time it was loaded in memory. This operation is called a *page-out* operation.

This way the swap space of a process contains an up-to-date copy of every page of the process that is not present in memory. A *page replacement* operation is one that loads a page into a page frame that previously contained another page.

It may involve a page-out operation if the previous page was modified while it occupied the page frame, and involves a page-in operation to load the new page.

**Page Table** The page table for a process facilitates implementation of address translation, demand loading, and page replacement operations. Figure 5.3 shows the format of a page table entry. The *valid bit* field contains a Boolean value to indicate whether the page exists in
memory. We use the convention that 1 indicates —resident in memory‖ and 0 indicates —not resident in memory.‖ The *page frame#* field, which was described earlier, facilitates address translation. The *misc info* field is divided into four subfields. Information in the *prot info* field is used for protecting contents of the page against interference. It indicates whether the process can read or write data in the page or execute instructions in it. *ref info* contains information concerning references made to the page while it is in memory.

The *modified* bit indicates whether the page has been modified, i.e., whether it is *dirty*. It is used to decide whether a page-out operation is needed while replacing the page. The *other info* field contains information such as the address of the disk block in the swap space where a copy of the page is maintained.

**Page Faults and Demand Loading of Pages**

Table below summarizes steps in address translation by the MMU. While performing address translation for a logical address *(pi , bi)*, theMMUchecks the valid bit of the page table entry of *pi*

| | | Misc info | | | |
|---|---|---|---|---|---|
| *Valid bit* | *Page frame #* | *Prot info* | *Ref info* | *Modi-fied* | *Other info* |
| | | | | | |
| | | | | | |
| | | | | | |

| Field | Description |
|---|---|
| Valid bit | Indicates whether the page described by the entry currently exists in memory. This bit is also called the *presence* bit. |
| Page frame # | Indicates which page frame of memory is occupied by the page. |
| Prot info | Indicates how the process may use contents of the page—whether read, write, or execute. |
| Ref info | Information concerning references made to the page while it is in memory. |
| Modified | Indicates whether the page has been modified while in memory, i.e., whether it is dirty. This field is a single bit called the *dirty* bit. |
| Other info | Other useful information concerning the page, e.g., its position in the swap space. |

| Step | Description |
|---|---|
| 1. Obtain page number and byte number in page | A logical address is viewed as a pair $(p_i, b_i)$, where $b_i$ consists of the lower order $n_b$ bits of the address, and $p_i$ consists of the higher order $n_p$ bits (see Section 11.8). |
| 2. Look up page table | $p_i$ is used to index the page table. A page fault is raised if the *valid bit* of the page table entry contains a 0, i.e., if the page in not present in memory. |
| 3. Form effective memory address | The *page frame* # field of the page table entry contains a frame number represented as an $n_f$-bit number. It is concatenated with $b_i$ to obtain the effective memory address of the byte. |

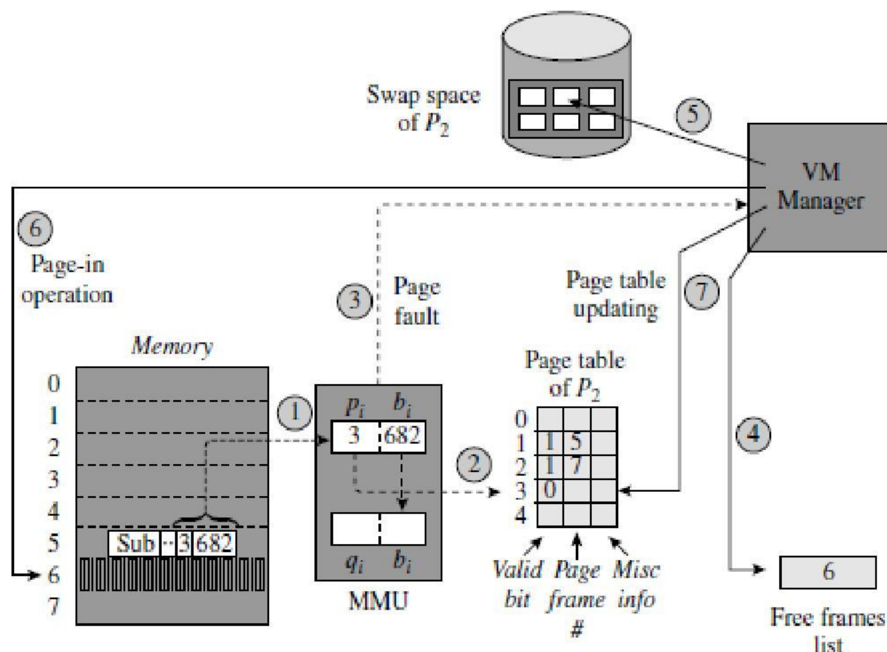



Figure 5.4 Demand loading of a page.

.

If the bit indicates that *pi* is not present in memory, the MMU raises an interrupt called a *missing page interrupt* or a *page fault*, which is a program interrupt (see Section 2.2.5). The interrupt servicing routine for program interrupts finds that the interrupt was caused by a page fault, so it invokes the virtual memory manager with the page number that caused the page fault, i.e., *pi* , as a parameter. The virtual memory manager now loads page *pi* in memory and updates its page table entry. Thus, the MMU and the virtual memory manager interact to decide *when* a page of a process should be loaded in memory.

Figure 5.4 is an overview of the virtual memory manager‘s actions in demand loading of a page. The broken arrows indicate actions of the MMU, whereas firm arrows indicate accesses to the data structures, memory, and the disk by the virtual memory manager when a page fault occurs. The numbers in circles indicate the steps in address translation, raising, and handling of the page fault—Steps 1–3 were described earlier in Table 12.2. Process *P*2 of Figure 5.2 is in operation. While translating the logical address (3, 682), the MMU raises a page fault because the valid bit of page 3‘s entry is 0.

**Page-in, Page-out, and Page Replacement Operations**

Figure 5.4 showed how a page-in operation is performed for a required page when a page fault occurs in a process and a free page frame is available in memory. If no page frame is free, the virtual memory manager performs a *page replacement operation* to replace one of the pages existing in memory with the page whose reference caused the page fault. It is performed as follows: The virtual memory manager uses a *page replacement algorithm* to select one of the pages currently in memory for replacement, accesses the page table entry of the selected page to mark it as ―not present‖ in memory, and initiates a page-out operation for it if the *modified* bit of its page table entry indicates that it is a *dirty* page.

In the next step, the virtual memory manager initiates a page-in operation to load the required page into the page frame that was occupied by the selected page. After the page-in operation completes, it updates the page table entry of the page to record the frame number of the page frame, marks the page as ―present,‖ and makes provision to resume operation of the process. The process now reexecutes its current instruction. This time, the address translation for the logical address in the current instruction completes without a page fault. The page-in and page-out operations required to implement demand paging constitute *page I/O*; we use the term*page traffic* to describe movement of pages in and out of memory. Note that page I/O is distinct from I/O operations performed by processes,whichwe will call *programI/O*. The state of a process that encounters a page fault is changed to *blocked* until the required page is loaded in memory, and so its performance suffers because of a page fault. The kernel can switch theCPU to another process to safeguard system performance.

**Effective Memory Access** Time The effective memory access time for a process in demand paging is the average memory access time experienced by the process.

It depends on two factors: time consumed by the MMU in performing address translation, and the average time consumed by the virtual memory manager in handling a page fault. We use the following notation to compute the effective memory access time:

$pr_1$ probability that a page exists in memory
$t_{mem}$ memory access time
$t_{pfh}$ time overhead of page fault handling

$pr_1$ is called the *memory hit ratio*. $t_{pfh}$ is a few orders of magnitude larger than $t_{mem}$ because it involves disk I/O—one disk I/O operation is required if only a page-in operation is sufficient, and two disk I/O operations are required if a page replacement is necessary.

A process‘s page table exists in memory when the process is in operation. Hence, accessing an operand with the logical address ($p_i$ , $b_i$ ) consumes two memory cycles if page $p_i$ exists in memory—one to access the page table entry of $p_i$ for address translation, and the other to access the operand in memory using the effective memory address of ($p_i$ , $b_i$ ). If the page is not present in memory, a page fault is raised after referencing the page table entry of $p_i$ , i.e., after one memory cycle.

Accordingly, the effective memory access time is as follows:

$$\text{Effective memory access time} = pr_1 \times 2 \times t_{mem} + (1 - pr_1) \times (t_{mem} + t_{pfh} + 2 \times t_{mem})$$

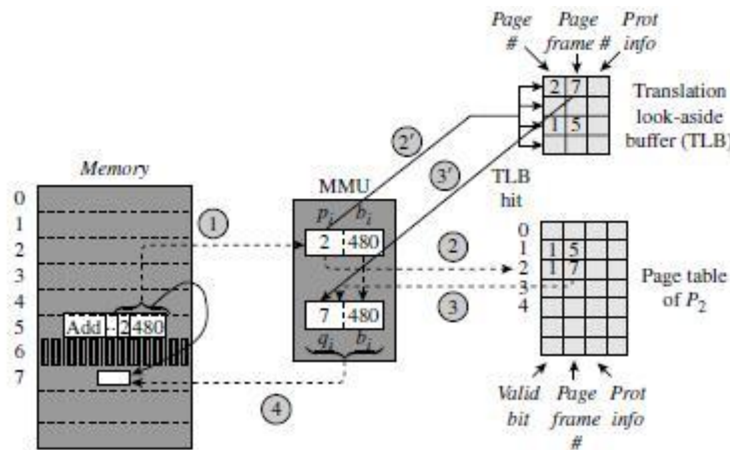
The effective memory access time can be improved by reducing the number of page faults.

### Address Translation and Page Fault Generation

The MMU follows the steps of Table 5.2 to perform address translation. For a logical address (*pi* , *bi* ), it accesses the page table entry of *pi* by using *pi* ×*lPT_entry* as an offset into the page table, where *lPT_entry* is the length of a page table entry.

*lPT_entry* is typically a power of 2, so *pi* ×*lPT_entry* can be computed efficiently by shifting the value of *pi* by a few bits.

**Address Translation Buffers** A reference to the page table during address translation consumes one memory cycle because the page table is stored in memory. The *translation look-aside buffer* (TLB) is a small and fast associative memory that is used to eliminate the reference to the page table, thus speeding up address translation. The TLB contains entries of the form (page #, page frame #, protection info) for a few recently accessed pages of a program that are in memory. During address translation of a logical address (*pi* , *bi* ), the TLB hardware searches for an entry of page *pi* . If an entry is found, the page frame # from the entry is used to complete address translation for the logical address (*pi* , *bi* ). Figure 5.8 illustrates operation of the TLB. The arrows marked 2_ and 3_ indicate TLB lookup. The TLB contains entries for pages 1 and 2 of process *P*2. If *pi* is either 1 or 2, the TLB lookup scores a hit, so the MMU takes the page frame number from the TLB and completes address translation. A TLB miss occurs if *pi* is some other page, hence theMMUaccesses the page table and completes the address translation if page *pi* is present in memory; otherwise, it generates a page fault, which activates the virtual memory manager to load *pi* in memory.




Address translation using the translation look-aside buffer and the page table.

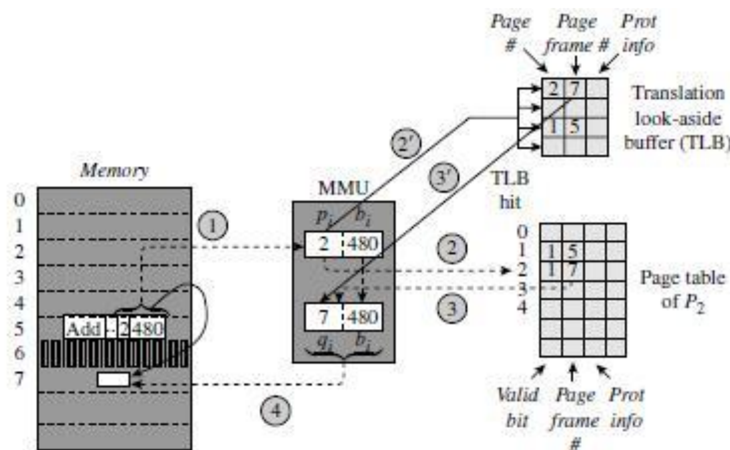
### 6. a) Explain how TLB is used to provide h/w support for paging. (5M)

The MMU follows the steps of Table 5.2 to perform address translation. For a logical address (*pi* , *bi* ), it accesses the page table entry of *pi* by using *pi* ×*lPT_entry* as an offset into the page table, where *lPT_entry* is the length of a page table entry.

*lPT_entry* is typically a power of 2, so *pi* ×*lPT_entry* can be computed efficiently by shifting the value of *pi* by a

few bits.

**Address Translation Buffers** A reference to the page table during address translation consumes one memory cycle because the page table is stored in memory. The *translation look-aside buffer* (TLB) is a small and fast associative memory that is used to eliminate the reference to the page table, thus speeding up address translation. The TLB contains entries of the form (page #, page frame #, protection info) for a few recently accessed pages of a program that are in memory. During address translation of a logical address (*pi* , *bi* ), the TLB hardware searches for an entry of page *pi* . If an entry is found, the page frame # from the entry is used to complete address translation for the logical address (*pi* , *bi* ). Figure 5.8 illustrates operation of the TLB. The arrows marked 2_ and 3_ indicate TLB lookup. The TLB contains entries for pages 1 and 2 of process *P*2. If *pi* is either 1 or 2, the TLB lookup scores a hit, so the MMU takes the page frame number from the TLB and completes address translation. A TLB miss occurs if *pi* is some other page, hence theMMUaccesses the page table and completes the address translation if page *pi* is present in memory; otherwise, it generates a page fault, which activates the virtual memory manager to load *pi* in memory.




**b) Explain inverted page tables. (5M)**

- A process with a large address space requires a large page table, which occupies too much memory
- Solutions:

– *Inverted page table*

- Describes contents of each page frame

– Size governed by size of memory

– Independent of number and sizes of processes

– Contains pairs of the form (program id, page #)

- Con: information about a page must be searched

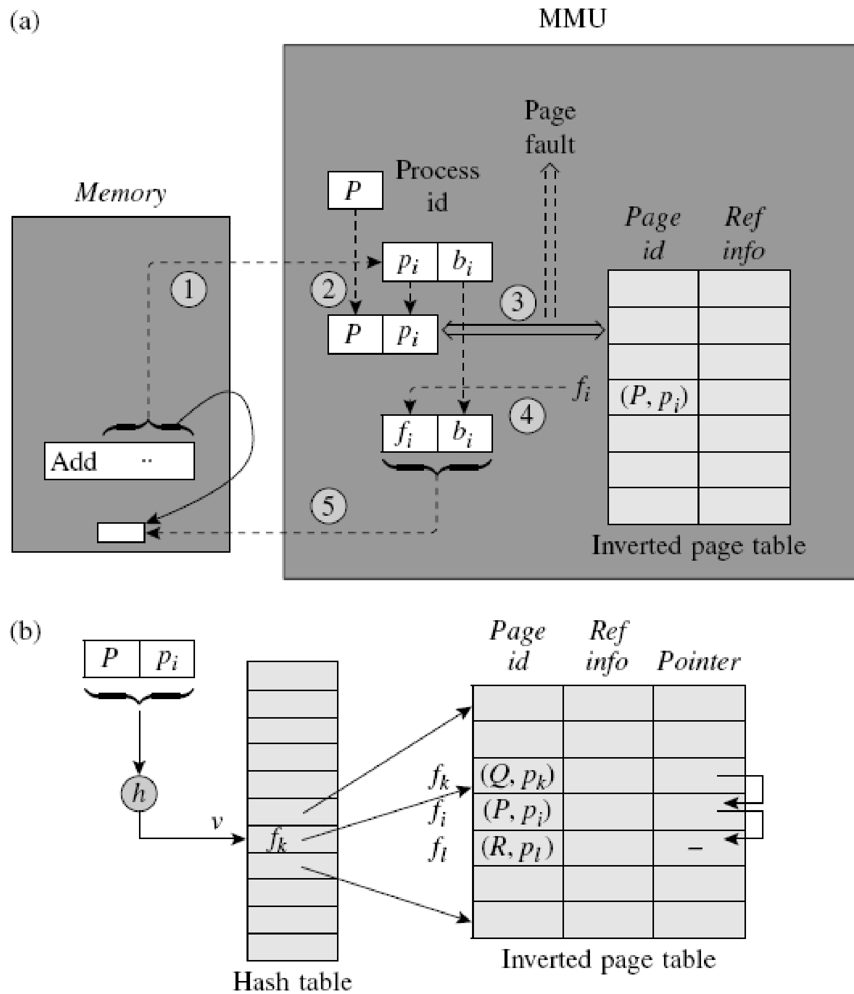



**Figure 12.10** Inverted page table: (a) concept; (b) implementation using a hash table.

7. a) Compare: i) Preemptive and non-preemptive scheduling ii) Long term and short term schedulers. (6M)

In ***preemptive scheduling***, the server can be switched to the processing of a newrequest before completing the current request. The preempted request is put back into the list of pending requests (see Figure 7.1). Its servicing is resumed when it is scheduled again. Thus, a request might have to be scheduled many times before it completed. This feature causes a larger scheduling overhead than when nonpreemptive scheduling is used.

The three preemptive scheduling policies are:

- Round-robin scheduling with time-slicing (RR)
- Least completed next (LCN) scheduling
- Shortest time to go (STG) scheduling

The RR scheduling policy shares the CPU among admitted requests by servicing them in turn. The other two policies take into account the CPU time required by a request or the CPU time consumed by it while making their scheduling decisions

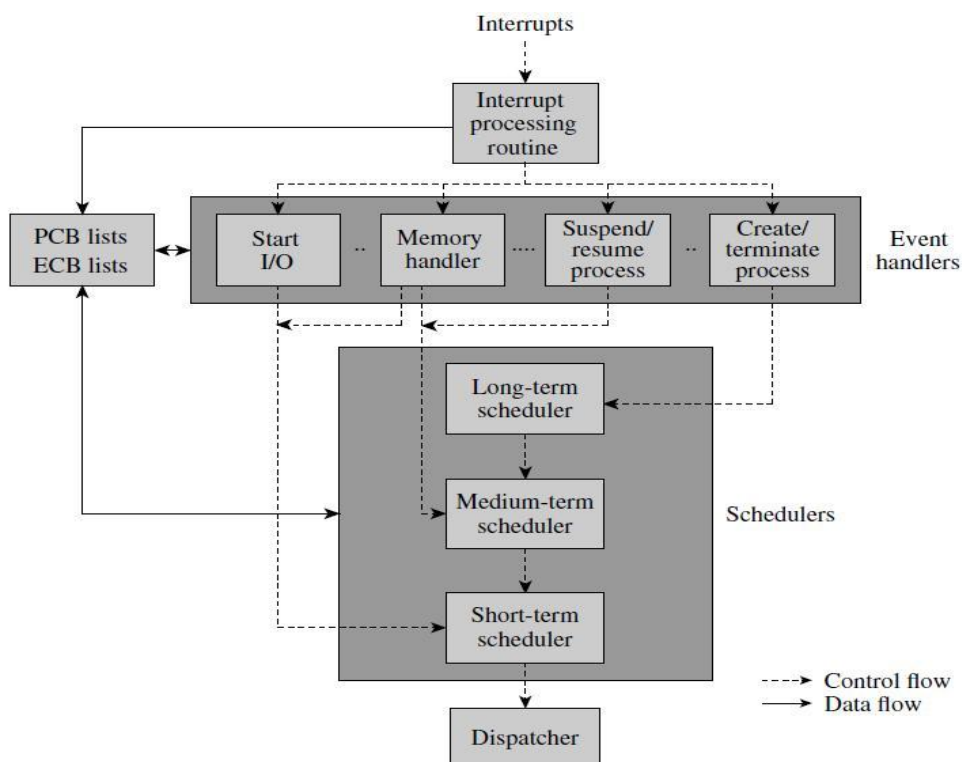
**In *nonpreemptive scheduling***, a server always services a scheduled request to completion. Thus, scheduling is performed only when servicing of a previously scheduled request is completed and so preemption of a request as shown in Figure 7.1 never occurs. Nonpreemptive scheduling is attractive because of its simplicity—the scheduler does not have to distinguish between an unserviced request and a partially serviced one. Since a request is never preempted, the scheduler's only function in improving user service or system performance is reordering of requests. The three nonpreemptive scheduling policies are:

- First-come, first-served (FCFS) scheduling
- Shortest request next (SRN) scheduling
- Highest response ratio next (HRN) scheduling

**Long-Term Scheduling** The long-term scheduler may defer admission of a request for two reasons: it may not be able to allocate sufficient resources like kernel data structures or I/O devices to a request when it arrives, or it may find that admission of a request would affect system performance in some way; e.g., if the system currently contained a large number of CPU-bound requests, the scheduler might defer admission of a new CPU-bound request, but it might admit a new I/O-bound request right away.

Long-term scheduling was used in the 1960s and 1970s for job scheduling because computer systems had limited resources, so a long-term scheduler was required to decide *whether* a process could be initiated at the present time. It continues to be important in operating systems where resources are limited. It is also used in systems where requests have deadlines, or a set of requests are repeated with a known periodicity, to decide *when* a process should be initiated to meet response requirements of applications. Long-term scheduling is not relevant in other operating systems.

**Short-Term Scheduling** Short-term scheduling is concerned with effective use of the CPU. It selects one process from a list of *ready* processes and hands it to the dispatching mechanism. It may also decide how long the process should be allowed to use the CPU and instruct the kernel to produce a timer interrupt accordingly.



b) Explain the process schedule with a neat schematic diagram. (4M)

Scheduling, very generally, is the activity of selecting the next request to be serviced by a *server*. Figure 7.1 is a schematic diagram of scheduling. The scheduler actively considers a list of pending requests for servicing and selects one of them. The server services the request selected by the scheduler. This request leaves the server either when it completes or when the scheduler preempts it and puts it back into the list of pending requests. In either situation, the scheduler selects the request that should be serviced next. From time to time, the scheduler admits one of the arrived requests for active consideration and enters it into the list of pending requests. Actions of the scheduler are shown by the dashed arrows in Figure 7.

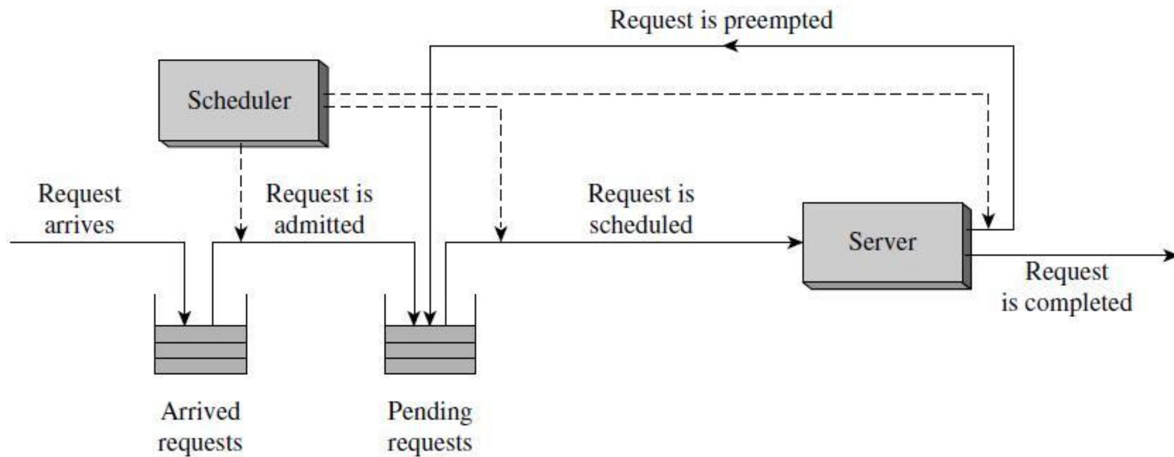



Figure 7.1 A schematic of scheduling.

Events related to a request are its *arrival, admission, scheduling, preemption*, and *completion*.