5
MEMORY: *Virtual Memory Management*

5.1 Chapter Objective

The objective of project MEMORY is to teach students about virtual memory and other modern memory-management techniques and to provide them with a well-structured programming environment in which to implement these techniques. To this end, students will be asked to implement the five public classes of the MEMORY package. The main class, `MMU`, represents the memory-management unit, the piece of hardware that is responsible for memory access in a computer. The other classes are `FrameTableEntry`, `PageFaultHandler`, `PageTableEntry`, and `PageTable`. All of these classes are described in detail later on in the chapter, each in its own subsection. We begin with an overview of memory-management basics.

5.2 Overview of Memory Management

In a modern computer, the portion of the circuitry called the **memory management unit** (abbr. MMU) is responsible for providing access to main memory. In *OSP 2*, memory access is simulated by calling the method `refer()` of the class `MMU`, which is one of the key methods to be implemented in this project. It may seem strange at first that you are being asked to implement a piece of hardware as part of an operating systems programming project. However, the

MMU is the gateway to memory for executing threads, and it provides you with a golden opportunity to implement the memory-management technique your MEMORY assignment calls for.

Memory management and multiprogramming. Modern memory-management techniques are aimed at supporting multiprogramming and must therefore allow multiple processes or threads to be memory-resident simultaneously. In this way, when the currently executed thread becomes blocked, e.g. waiting for an I/O request to complete, control of the CPU can be easily switched to another memory-resident thread, albeit one that is in the ready state waiting to resume execution.

Partitioning memory. The basic memory-management technique in support of multiprogramming is to partition main memory into, shall we say, partitions or chunks of memory that different processes can occupy. Partitions can be either fixed-size or variable-size. The former results in internal fragmentation, which occurs when a process does not utilize the entirety of a partition. The latter results in external fragmentation, which occurs when a partition is too small to be of use to any process. We shall focus the remainder of our overview of memory-management basics on the fixed-sized partitions utilized by a technique known as paging. Segmentation is an alternative to paging that uses variable-size partitions.

Logical memory. The big advance in memory management came with the realization that the memory allocated to a process need not be contiguous! In the case of paging, a partition is called a page and a process's pages collectively make up its logical address space. Physical memory is divided into page frames, each the size of a page so the fit of a page in a page frame is exact. Thus, in theory, a page of a process can be placed in any available page frame. The point is that the pages of a process's logical address space may be dispersed noncontiguously among the page frames of main memory. Page sizes typically range from 512 bytes to 4K bytes but whatever the page size, it is fixed throughout system execution.

The primary mechanisms used for implementing logical memory are the **page table base register** (abbr. PTBR) and the **page table**. The key issue here is logical address translation, how to convert a logical address into a physical address, and this is the responsibility of the MMU. A logical address is just string of bits (e.g. 32 in the case of a 32-bit machine architecture) that can be logically viewed as consisting of two parts: a page number and a byte offset into the page. The number of bits taken up by the first part will depend

on the page size: 9 for a page size of 512 bytes, 10 for a page size of 1K bytes, etc. The remainder of the address bits are interpreted as the byte offset into the logical page being addressed.

Every process has a page table of its own and when a thread is dispatched on the CPU, the address of the page table of the process to which the thread belongs is placed in the PTBR. The MMU uses the PTBR to find the location of the page table and uses the page table to supply the mapping between the logical memory of processes and the main memory of the computer, represented by the physical page frames. The overall schema is depicted in Figure 5.1.

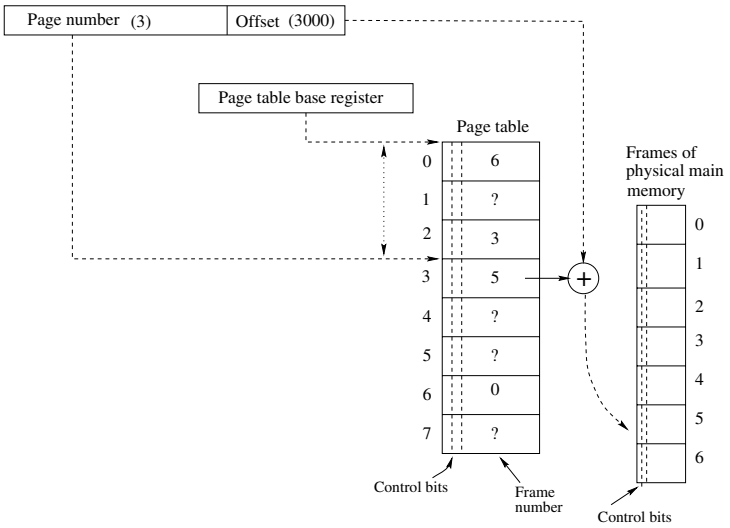

Figure 5.1 Logical/Virtual address translation using page tables.

Virtual memory. The simple memory-addressing mechanism just described works well as long as the frames corresponding to the pages of a process are all in main memory. However, as seen in Figure 5.1, some entries in a page table do not necessarily have to have frames assigned to them. The key insight behind virtual memory is that a page table can have more entries than the number of physical page frames, so a one-to-one assignment of frames to pages might not be possible. In other words, the size of virtual memory can and normally does exceed that of physical memory. Note that we use the term virtual memory now instead of logical memory to emphasize the fact that larger-than-physical-memory address spaces are supported by this scheme.

Pagefaults. The key mechanism for implementing virtual memory is that each page table entry has a **validity bit**, which indicates whether the page has a main-memory frame assigned to it. This bit is checked by the MMU hardware and whenever a running thread makes a reference to a page whose validity bit is zero, a **pagefault** occurs: a special kind of interrupt that is used to notify the operating system of references to frame-less pages. The intended response from the OS is to assign a suitable frame to the page. The module responsible for this action is called the **pagefault handler**. A page whose validity bit is one (i.e. has a page frame assigned to it) is said to be *valid*.

Before examing the steps involved in handling a pagefault, let us first look more carefully at frame-less pages and their relationship to other resources owned by processes. If no frame is assigned to a page, where is the program code or data that the running thread is supposedly referencing? The answer is that a copy of the entire process space is kept in secondary storage on a **swap device**. In high-performance systems, a swap device can be a separate disk, but typically it is just a partition occupying part of a physical disk. Nevertheless, the operating system assigns a **logical device** to each such partition and at that level the swap device can be viewed as a separate device with its own characteristics and device number. In particular, in *OSP 2*, a swap device is viewed as a real device with a special device number, `SwapDeviceID`. Thus, every process (i.e., OSP task) has a corresponding **swap file** on the swap device, which contains an exact image of the process memory.

When a pagefault on page P of task T occurs, the pagefault handler has to do several things:

1. Suspend the thread that caused the interrupt until the situation that gave rise to the pagefault is rectified. This is done by creating a *new* event, `pfEvent`, of type `SystemEvent` and then executing `suspend()` on the thread using `pfEvent` as a parameter. A new system event is created using the constructor `SystemEvent()` of class `SystemEvent`. This event must be kept around until the end of pagefault processing, as it is needed to resume the thread before returning from the pagefault handler.

2. Find a suitable frame to assign to page P.
   An obvious choice would be a free frame, i.e., a frame that is not assigned to any other page (of this or any other task). But there might not be such a frame at the moment (remember that there are fewer frames than pages!). In this case, **page replacement** must be performed, as described below. The result of a successful page-replacement action is that a free frame becomes available and is assigned to page P.

3. Perform a *swap-in*.
   Once a frame is assigned to the faulty page, you need to make sure that it

contains the exact image of the page, which is available in the task's swap file. To do this, the pagefault handler must initiate a *swap-in*: a file read operation that would input the requisite page from the swap device and store it in the frame.

4. Suspend the pagefault handler.
   An I/O operation takes time, so the pagefault handler must suspend itself until it is woken up by the disk interrupt coming from the swap device.[1] Suspension of the pagefault handler actually happens as part of the file read operation that swaps the page in—you do not need to do this explicitly.

5. Finish up.
   Once the image of the right page is copied into the frame, the pagefault handler should update the page table to make sure that the page entry is pointing to the right frame, and set the validity bit of the page appropriately. Next, the thread that caused the pagefault should be resumed and placed on the queue of the ready-to-run threads. This is done by executing the method `notifyThreads()` on the event `pfEvent`, which was created in Step 1. Finally, as with any other interrupt handler, the dispatcher should be called to give control of the CPU to some ready-to-run thread.

Page replacement. In describing the actions of the pagefault handler, we deliberately omitted a saga of its own: What should you do if, in Step 2, the pagefault handler cannot find a free frame? In this case, it becomes necessary to choose a frame F' occupied by some other page P' and use F' to satisfy the pagefault. The page P' is often called a **victim page** and it is said that the pagefault handler *evicts* this page from its frame.

The algorithm deployed by the pagefault handler for choosing such a frame is called the page-replacement algorithm, and the most well-known algorithm of this kind is LRU (Least Recently Used). LRU replaces the page in memory that has not been referenced for the longest time. Assuming that threads exhibit the principle of locality, meaning that they cluster memory references around a certain subset of their pages over a given window of time, then the LRU page should be the least likely page to be referenced in the near future and its replacement is a good bet.

The problem with the LRU policy is that it can be difficult to implement and for this reason other algorithms have been developed to approximate the performance of LRU while imposing little overhead. Many of these algorithms

[1] Handling disk interrupts is part of another project, module DEVICES. In the present project, one should assume that the disk interrupt handler functions according to the specifications given below.

are variants of a scheme known as the **clock policy**, which associates one or more **use bits** with each frame and organizes the frames as a circular buffer. Consider for simplicity the case of a single use bit added to each frame. A frame's use bit is set to 1 when a page is first loaded in the frame and whenever the page in the frame is referenced. When it comes time to replace a page, the clock algorithm scans the buffer looking for a frame with a use bit of zero; the page occupying the first such frame is chosen for replacement. Each time it encounters a frame with a use bit of 1, it resets that bit to zero and moves on to the next frame in the buffer. The use of multiple use bits per frame increases the algorithm's precision: the more use bits per frame deployed, the more closely the algorithm is able to approximate LRU.

From a performance perspective, a good page-replacement algorithm is characterized by a low pagefault rate. However, as far as the operating system is concerned, the only requirement of a page-replacement algorithm is that there should be no "undesirable side effects". One such side effect is due to the nature of the I/O subsystem. Suppose that a page-replacement algorithm chooses a frame F' that is involved in an active I/O operation. In some cases, a device that started an I/O cannot be stopped. So if you reuse the corresponding frame for some other purpose then the data in the frame may become corrupted (in case of a file-read operation) or, in case of a write operation, the data being written out might become corrupted if you change the content of the frame before the I/O is finished. Even if the device can be stopped immediately, it might still not be a good idea because stopping the device now might mean that the same I/O operation would have to be re-issued later.

Locking and unlocking of frames. How can an OS protect the frames associated with active I/O operations? A typical mechanism is to keep, for each frame, a count of the active or outstanding I/O operations that involve that frame. There are a variety of ways to maintain such a count. Here is an explanation of how it is done in *OSP 2*. When an I/O operation is to be performed on a certain I/O device, an I/O request block (abbr. IORB) is enqueued on the device's device queue. An IORB does not refer to frames directly. Instead, it references the *page* that is involved in the I/O. The thread that requested the I/O must perform a `lock()` operation (which is a method of class `PageTableEntry`) on the page involved. If no frame is assigned to the page, a pagefault occurs, and the IORB will not be enqueued on the device until the pagefault processing is over. The `lock()` operation increments the **lock count** of the frame associated with the page and the `unlock()` operation decrements it. A page is considered to be locked in a frame if the lock count of the associated frame is a positive number.

Thus, by the time the IORB makes it to the device queue, the page involved

is locked and has an associated frame. The page-replacement mechanism is prohibited from taking frames that have positive lock counts.

Note that a page involved in an I/O is locked into a frame when the corresponding IORB is *enqueued* on the device queue of the device in question (a device might be busy and have a queue of outstanding I/O's), not when the IORB is selected for processing by the device. The reason should be obvious: To perform an I/O, the page referenced by the IORB must be in some frame in main memory. If not, it would have to be swapped in. But this requires another I/O and takes time. So, the selected IORB cannot be processed and the device would remain idle. In contrast, if pages are locked just before the IORB is enqueued, the corresponding frames would remain protected for the entire period while the IORB remains on the device queue (and until the device finishes the corresponding I/O). If the page being locked is frame-less, a pagefault occurs and the page is brought in *before* the IORB is selected for processing.

Dirty frames. Locking is not the only constraint that a page-replacement mechanism must abide by. Another issue has to do with so-called **dirty frames**. A dirty frame is one whose contents has been changed since the last time a page was swapped into the frame. If such a frame is chosen for replacement, the current contents of the frame must be saved in the swap file of the task that owns the page that currently occupies the frame. Otherwise, all changes made to the page will be lost. Thus, each frame needs another bit, the **dirty bit**, which indicates whether the contents of the frame has been changed. The actions that change the contents of a frame are the memory operation (`MemoryWrite`) and the I/O operation `read()`, which transfers data from a file to main memory.

Thus, we see that finding a victim page and evicting it is no simple matter: It may require an extra I/O operation to *swap-out* the victim page and synchronize its contents with the contents of that page in the swap file.

Frame table. Information about physical, main-memory frames is kept in the **frame table**: an array that has one entry per frame. Each entry is an object of the class `FrameTableEntry`. In fact, an *OSP 2* frame table contains more information than that. For instance, each frame entry contains a back reference to the page that occupies that frame (or `null`). Every frame entry also has a so-called **reference bit**, which indicates whether the frame has been *referenced* (as the result of executing `refer()` or due to I/O into or out of the frame). The reference bit often plays a role (as a use bit) in page-replacement algorithms.

In real computers, the reference and dirty bits are set in hardware but they are unset by software using special instructions. In contrast the lock count and

the page reference in the frame table are manipulated entirely in software. In *OSP 2* you have to set and unset all of these items in software. In this sense, part of what you will do to implement the `refer()` method is really a simulation of various hardware functions. This includes setting the dirty and the reference bits, and also causing the pagefault interrupt itself. We describe these issues in more detail in Section 5.6.

Reserved frames. In *OSP 2*, frames have yet another bit, the **reserved bit**. Like the lock count, a reserved bit protects frames from the page-replacement mechanism, but it is used for a different reason. Suppose a thread Th causes a pagefault on page P and control is transferred to the pagefault handler after blocking Th. The pagefault handler may go through several distinct phases:

1. Finding a suitable frame F. Suppose F is dirty and is currently occupied by page P'.
2. Evicting P' by issuing an I/O operation that swaps P' out.
3. Waiting for the I/O to finish.
4. Initiating an I/O to swap page P into frame F.
5. Waiting for the I/O to finish.
6. Putting Th on the ready queue and quitting.

The problem is that while locking will prevent F from being grabbed by other threads *during* phases 3 and 5, nothing prevents it from being grabbed to satisfy other pagefaults between phases 1 and 2, between phases 3 and 5, and after phase 5. Thus, it might well happen that after trying so hard to assign a suitable frame to page P, the pagefault handler will find the frame stolen from under its nose before it gets a chance to assign F to P. To prevent this kind of unproductive behavior, the pagefault handler must *reserve* frame F in phase 1 and *un-reserve* it in phase 6.

Prepaging. Some pagefault handling algorithms perform **prepaging**, i.e., the swapping in of invalid pages that *did not* cause the pagefault. These algorithms are trying to guess which pages might be referenced by the thread in the near future and swap them in proactively. To implement prepaging, the pagefault handler can issue additional `read()` operations, which might require `write()` operations to swap some other pages out.

Prepaging a page is similar to bringing a page in as part of regular pagefault processing. However, selecting a frame for prepaging should be done with

caution. In particular, make sure that it is not the frame that was selected for the original faulty page. Otherwise, you will end up evicting the page that the pagefault handler was supposed to make valid!

Since prepaging involves I/O, it is possible that the thread that initiated the pagefault will be killed by the time prepaging is finished. When this happens, prepaging should stop. One special case arises when prepaging is done from within the pagefault handler. The question then is what should be the return code for `do_handlePageFault()`: `SUCCESS` or `FAILURE`? *OSP 2* expects `FAILURE` in this case. In particular, if the page that caused the pagefault became valid before the thread was killed, the page should be made invalid again prior to returning from the pagefault handler. However, you should realize that a more optimized operating system might make a different decision and keep such a page valid, because it might be used by other threads of the same task.

Proactive page cleaning. Some memory-management algorithms perform proactive page cleaning by periodically swapping them out on disk (but not invalidating them). The idea is to utilize the times when the swap device is idle and reduce the time needed to handle pagefaults by increasing the supply of clean pages.

Technically, this is done by setting up **daemons**: special system threads that are set to wake up periodically, perform the job assigned to them, and go back to sleep. We discussed the *OSP 2* support for daemons in Section 1.7.

In order to set up a cleaning daemon, one creates a class that implements `DaemonInterface` (see Section 1.7). The required method `unleash()` can then be made to execute the proactive cleaning algorithm. An essential part of this algorithm is a series of `write()` operations that write dirty frames out to the swap device (but keeps these pages valid). This daemon must be registered with the system at startup, as explained in Section 1.7.

Having surveyed the major issues involved in pagefault handling, we are now ready to discuss the actual *OSP 2* classes and methods that constitute the MEMORY module. The class diagrams of Figure 5.2 and Figure 5.3 place these classes in a larger context.

5.3 Class `FrameTableEntry`

This class implements the entries in the frame table, the main repository of information about the status of the main-memory frames. It is defined as follows:

◇ `public class FrameTableEntry extends IflFrameTableEntry`

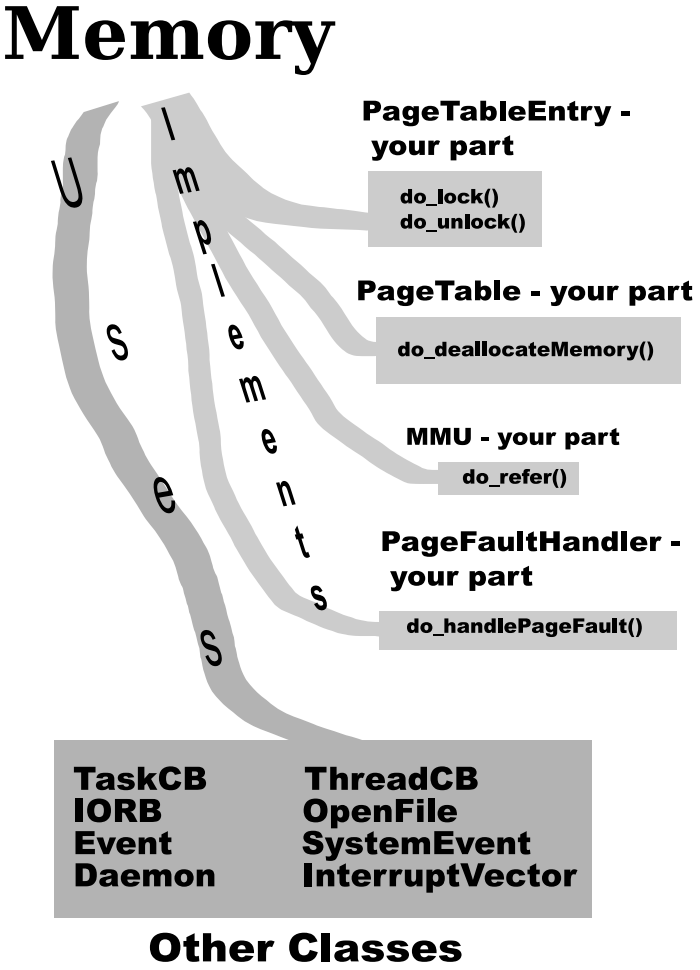

Figure 5.2 Overview of the package MEMORY, I.

The class constructor is the only method of this class that needs to be implemented as part of the project:

◇ `public FrameTableEntry(int frameID)`
Calls `super(frameID)` and might perform other initializations if the student implementation defines additional fields in this class.

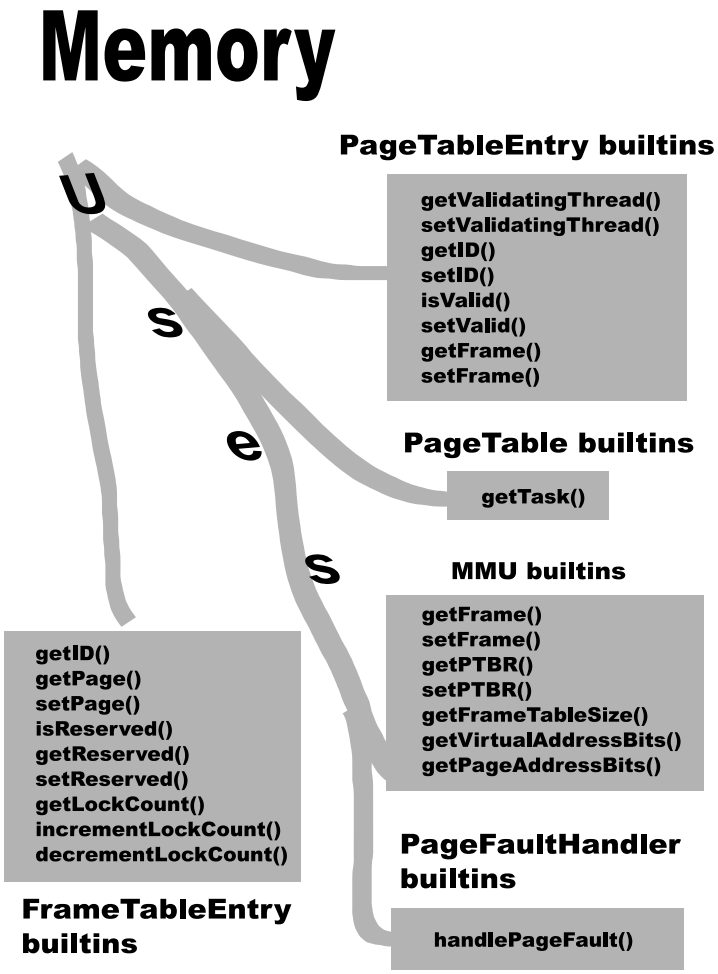

Figure 5.3 Overview of the package MEMORY, II.

However, this class inherits a number of methods from its superclasses, and these methods are used by other classes in this project:

◇ `public final int getLockCount()`
Returns the lock count of the frame.

◇ `final public void incrementLockCount()`
Increments the lock count of the frame by 1.

◇ `final public void decrementLockCount()`
Decrements the lock count of the frame by 1.

Summary of Class `FrameTableEntry`

The following table summarizes the attributes of the class `FrameTableEntry` and the methods for setting and querying them. These attributes and methods are all inherited from class `IflFrameTableEntry` and are described in more detail in Section 5.8.

Reserved flag: Indicates if a thread has reserved this frame. The corresponding methods are `isReserved()`, `getReserved()`, `setReserved()`, and `setUnreserved()`.

Dirty flag: The methods for manipulating the dirtiness of a frame are `isDirty()` and `setDirty()`.

Reference flag: Indicates if the frame has been referenced. The methods that handle this attribute are `getreferenced()` and `setreferenced()`.

Lock count: This attribute represents the number of times the frame has been locked minus the number of unlock operations performed on the frame, and is accessed using the methods `getLockCount()`, `incrementLockCount()`, and `decrementLockCount()`.

Identity: The identity of a frame is its sequence number in the system-wide array of all main-memory frames. It can be queried using the method `getID()`.

Page: This is the page that occupies the frame (null, if the frame is free). This attribute can be set using `setPage()` and retrieved using `getPage()`.

It should be noted that some information (such as page information and identity) in the *OSP 2* frame table entries is redundant and is not present in the frame table of a typical operating system. In fact, the *OSP 2* frame table can be seen as a cross between a normal frame table and what is known as an **inverted page table**.

5.4 Class `PageTableEntry`

This class implements the data structure that describes each entry in the page table. It is defined as follows:

◇ `public class PageTableEntry extends IflPageTableEntry`

For this project, the student must implement the following methods of this class:

◇ `public int do_lock(IORB iorb)`
The ultimate goal of this method is to increment the lock count of the frame associated with the page. However, the details are not as simple as one might think, because the page might be invalid at the time the lock operation is performed.

Thus, this method must first check if the page is in main memory by testing the validity bit of the page (using the method `isValid()`). If the page is invalid, a pagefault must be initiated.

To initiate a pagefault, the `do_lock()` method calls the static method `handlePageFault()` of class `PageFaultHandler`, i.e., it calls the pagefault handler directly, without initiating an interrupt. Note that page locking is performed as part of an I/O request, when the CPU is already in kernel mode, so there is no need to cause an interrupt.

We have already seen that page locking involves considerably more than simply incrementing the lock count. Yet, there is still much more to do. Consider the following situation. Suppose thread Th_1 of task T makes a reference to page P either via the `refer()` operation or through locking. If the page is invalid, a pagefault is initiated. Suppose now that thread Th_2 of the same task comes along and also wants to lock the same page P. Should this cause a pagefault as well? The answer, of course, is no. The pagefault handler must already have found a suitable frame for P and the corresponding I/O requests must already be in the pipeline. Another pagefault would only confuse the system.

To help identify the pages that are involved in a pagefault, *OSP 2* provides the method `getValidatingThread()`. When applied to a page, this method returns the thread that caused a pagefault on that page (or null, if the page is not involved in a pagefault). In our case, this method would return Th_1.

The proper action for Th_2 depends on whether $Th_2 = Th_1$. If $Th_2 = Th_1$, then the proper action is to return right after incrementing the lock count.[2]

[2] You might be wondering how a thread that caused a pagefault can come back and request a lock on the page. The answer is simple: The lock can be requested by the swap-in I/O operation that must be performed as part of pagefault handling. This swap-in operation is performed on behalf of the same thread that caused the pagefault, so the locking thread and the validating thread would be one and the same.

If $Th_2 \neq Th_1$, then the proper action is to wait until P becomes valid. This is easy to accomplish because the class `PageTableEntry` happens to be a subclass of `Event` (see Section 4.3 for the description of this class). Thus, we can execute the `suspend()` method on Th_2 and pass page P as a parameter.

When the page becomes valid (or if the pagefault handler fails to make the page valid, say, because the original thread, Th_1, that caused the pagefault was killed during the wait), the threads waiting on the page will be unblocked by the pagefault handler (which is another class in this project) and will be able to continue. When such threads become unblocked inside the `do_lock()` method, control falls through the call to `suspend()` and the `do_lock()` method must exit and return the appropriate value: `SUCCESS` if the page became valid as a result of the pagefault and `FAILURE` otherwise.

In general, `do_lock()` returns `SUCCESS` if the page was locked successfully (which does not necessarily involve a pagefault) or `FAILURE` if the page was not locked. The latter can happen if either the pagefault (which might occur due to locking) fails or if the thread that created `iorb` was killed while waiting for the lock operation to complete.

Finally, in the case of a successful return, you should remember to increment the lock count of the frame associated with the page, i.e., to do the actual locking. (Note that the focus of the previous discussion was on ensuring that the page is associated with a frame.) Incrementing the lock count of a frame is accomplished using method `incrementLockCount()` of class `FrameTableEntry`.

◇ `public void do_unlock()`
Unlocking is, fortunately, much simpler than locking. All that is needed is to decrement the lock count via a call to `decrementLockCount()` of class `FrameTableEntry`. Make sure that the lock count does not become negative, which is a sign of a problem.

Relevant methods defined in other classes. The following is a list of methods from other classes that can be useful in implementing the methods of class `PageTableEntry`:

◇ `public final int getLockCount()` `FrameTableEntry`
Returns the lock count of the frame.

◇ `final public void incrementLockCount()` `FrameTableEntry`
Increments the lock count of the frame by 1.

◇ `final public void decrementLockCount()` `FrameTableEntry`
Decrements the lock count of the frame by 1.

◇ `final public ThreadCB getValidatingThread()` `PageTableEntry`
Returns the validating thread of the page, i.e., the thread that caused the pagefault on this page. If the page is not in pagefault or its validating thread was killed before the page became valid, then this method returns `null`.
This method is inherited from a superclasses of `PageTableEntry`.

◇ `final public void suspend(Event event)` `ThreadCB`
Suspends the thread on which this method is called and puts the thread on the waiting queue of `event`.

◇ `final public int getStatus()` `TaskCB`
Returns the task's status.

◇ `final public int getStatus()` `ThreadCB`
Returns the status of this thread.

◇ `public final boolean isValid()` `PageTableEntry`
Tells if the page is valid by checking the validity bit.

◇ `public final boolean isReserved()` `FrameTableEntry`
Tests if the frame is reserved.

◇ `public final ThreadCB getThread()` `IORB`
Returns the thread that requested the I/O.

◇ `final public int getDeviceID()` `IORB`
Returns the device involved in the I/O operation.

◇ `final public int getIOType()` `IORB`
Returns the I/O type represented by the IORB. *OSP 2* supports two types: `FileRead` and `FileWrite`.

Summary of Class `PageTableEntry`

The following is a summary of the main attributes of class `PageTableEntry` and the methods for manipulating them. See Section 5.8 for a description of these methods.

Validity flag: The validity flag is handled by the methods `isValid()` and `setValid()`.

Frame: If the page is valid, there must be a frame associated with it, which is described by this attribute. The corresponding methods are `getFrame()` and `setFrame()`.

Identity: The identity of a page is its sequence number in the corresponding page table. It is set automatically by the system and can be queried using `getID()`.

Owner task: Points to the task that owns the page and is queried using method `getTask()` of `PageTableEntry`. This attribute is not really stored with the page; it is rather an attribute of the table to which the page belongs. Thus, this method simply queries the corresponding attribute of the page table.

Validating thread: If the page is currently in pagefault processing, this is the thread that caused the pagefault. This thread can be obtained using the method `getValidatingThread()` and is set using `setValidatingThread()`.

5.5 Class `PageTable`

The class `PageTable` represents page tables and is defined as follows:

◇ `public class PageTable extends IflPageTable`

The only mandatory method to be implemented here is the class constructor:

◇ `public PageTable(TaskCB ownerTask)`
This constructor gets as a parameter the task for which the table is to be created. It first calls `super(ownerTask)`, as all *OSP 2* constructors must do, and then constructs the page table. The page table is assumed to be an array of size equal to the maximum number of pages allowed, and is accessible through the variable `pages` inherited from the superclass `IflPageTable`. The maximal number of pages allowed is calculated using the method `MMU.getPageAddressBits()`, which represents the number of bits dedicated to representing a page number out of the total number of bits in an address.

After calling `super()`, the variable `pages` must be initialized to a new array of `PageTableEntry` whose size is determined as described above. Then each page must be initialized with a suitable `PageTableEntry` object using the constructor of that class. Make sure that you use correct page id numbers and the correct page table in the `PageTableEntry` constructor when creating these page objects.

◇ `public void do_deallocateMemory()`
This method is typically invoked by a terminating task on its page table object to unset the various flags for the frames allocated to the task. Specifically, it uses `setPage()` to nullify the `page` field that points to the page that occupies the frame (thereby freeing the frame), `setDirty()` to clean the page,

and `setReferenced()` to unset the reference bit. It also un-reserves each frame that was reserved by that task. To find out which task has reserved a given frame, use the method `getReserved()` of class `FrameTableEntry`.

Note that this method does not need to (and should not) set the frame attribute of the deallocated pages to null. It is possible that some of these pages are being used by ongoing I/O operations that pump data into or out of the frames that are currently allocated to the killed task. The disk interrupt handler (which will be called each time an I/O is finished) needs to know both the frame and the page objects involved in the finished operation, and it gets the former from the latter.

Note that if a page of a killed task is locked, it can be unlocked only by the device interrupt handler. Unlocking inside the memory-management module can lead to inconsistencies. Try to analyze what might happen in this case in order to understand why this is dangerous.

Summary of Class `PageTable`

Here is a summary of the attributes and methods of class `PageTable`. All of these attributes are provided by class `IflPageTable` and are inherited from there.

Page table: This is an array referenced by the variable `pages`. This array is created in the `PageTable()` constructor.

Owner task: Describes the task to which the page table belongs, which can be obtained via the method `getTask()`.

5.6 Class `MMU`

This class represents the memory-management unit of the simulated computer, and defines three methods: the initialization method that exist in every student module and `do_refer()`, which represents memory references made by the CPU while executing computer instructions. A detailed explanation is given below.

◇ `public static void init()`
This method is called once, at the beginning, to initialize the data structures. Typically, it is used to initialize the frame table.

Since the total number of frames is known (`MMU.getFrameTableSize()`), each frame in the frame table can be initialized in a `for`-loop. Initially, all

entries in the frame table are just `null`-objects and must be set to real frame table objects using the `FrameTableEntry()` constructor. To set a frame entry, use the method `setFrame()` in class `MMU`.

Another use of the `init()` method is for the initialization of private static variables defined in other classes of the MEMORY package. For example, one can define an `init()` method in class `PageFaultHandler` which would be able to access any variable defined in that class. Then `MMU.init()` can call `PageFaultHandler.init()`. Since `MMU.init()` is called at the very beginning of the simulation, `PageFaultHandler.init()` is also going to be called at the beginning of the simulation.

◇ `public PageTableEntry do_refer(int memoryAddress, int referenceType,ThreadCB thread)`

This method takes an address of a byte in the logical memory of the thread, a type of the memory reference (`MemoryRead`, `MemoryWrite`, or `MemoryLock`) and a thread that made the reference. The method then needs to determine the page of the thread's logical memory to which the reference was made. The methods `getVirtualAddressBits()` and `getPageAddressBits()`, both inherited from the superclass `IflMMU`, can be used to determine the number of bits allocated to represent the offset within the page. This number can then be used to compute the page size and then the page to which `memoryAddress` belongs.

Next, the method must check if the page is valid (the method `isValid()`). If so, you only need to appropriately set the referenced and the dirty bits of the page, and quit.

If the page is invalid, things are more interesting. There are two possibilities:

1. Some other thread of the same task has already caused a pagefault and the page is already on its way to main memory.
2. No other thread caused a pagefault on this invalid page.

As before, you can tell one case from the other with the help of the method `getValidatingThread()`.

In the first case, the thread (that was passed as a parameter to `do_refer()`) should simply suspend itself on the page and wait until the page becomes valid. When the page eventually becomes valid, the method should set the referenced and dirty bits appropriately and quit. A thread is suspended by an invocation of the method `suspend()` in class `ThreadCB`. When the page becomes valid, execution continues past the `suspend()` statement. Keep in mind that since a long time may pass between the initial pagefault and the time the faulty page becomes valid, the simulator might decide to destroy

the waiting thread. In this case, the dirty and referenced bit settings must not be changed. Thus, always use the `getStatus()` method to verify that the thread does not have status `ThreadKill`.

In the second case, the method must initiate a pagefault. Unlike in the `do_lock()` method, a pagefault interrupt must be caused. It is not enough to just invoke the method `handlePagefault()` because at the time when the thread executes `refer()`, the machine is in the user mode executing a user thread. In contrast, when pagefault is caused by the `lock()` operation, the machine must already be in kernel mode, since `lock()` is called by the operating system itself.

To cause an interrupt, one must suitably set the various static fields of the class `InterruptVector`. This is done using the static methods `setPage()`, `setReferenceType()`, and `setThread()`. Then one must call the `interrupt()` method of class `CPU` and pass it the the type of the interrupt (i.e., `PageFault`). Eventually, this will invoke the method `do_handlePageFault()` in class `PageFaultHandler`. Thus, when the `interrupt()` method returns, the page will be in the main memory and the thread will be in the ready queue.

Before exiting, `do_refer()` must set the reference and the dirty bits.

In both cases, it must be kept in mind that any thread might get killed while waiting for completion of I/O. Such is the wicked nature of the simulator. If a thread is killed, neither the dirty nor the reference bits should be changed. *OSP 2* is checking these conditions vigilantly. The method `getStatus()` should be used to determine the status of a thread.

On exit, `do_refer()` must return the referenced page.

Relevant methods defined in other classes. Here is a summary of the methods defined in other classes, which might be used in the implementation of the methods of class `MMU`:

◇ `final static public void setInterruptType(int inter)` `InterruptVector`

Sets the type of the interrupt in the interrupt vector. The valid values are `PageFault`, `DiskInterrupt`, and `TimerInterrupt`.

◇ `final static public int getInterruptType()` `InterruptVector`

Extracts the interrupt type from the interrupt vector.

◇ `final static public void setThread(ThreadCB thread)` `InterruptVector`

Sets the thread field in the interrupt vector so that pagefault handlers can find out who has caused the interrupt.

◇ `final static public ThreadCB getThread()` `InterruptVector`
Tells which thread has caused the interrupt.

◇ `final static public void setReferenceType(int ref)` `InterruptVector`
Sets the memory reference type in the interrupt vector. The valid types are `MemoryRead`, `MemoryWrite`, and `MemoryLock`. Applicable to pagefaults only.

◇ `final static public int getReferenceType()` `InterruptVector`
Tells what the reference type was that caused the interrupt. Applicable to pagefaults only.

◇ `final public void suspend(Event event)` `ThreadCB`
Suspends the thread on which this method is called and puts the thread on the waiting queue of `event`.

◇ `final static public FrameTableEntry getFrame(int frameNumber)` `MMU`
Returns the frame entry with the given frame number. This method is defined in the superclass of `MMU`, and is inherited.

◇ `final static public void setFrame(int index, FrameTableEntry entry)` `MMU`
Sets the frame with the given index to a non-null `FrameTableEntry`-object. This method is defined in the superclass of `MMU`, and is inherited.

◇ `final static public int getFrameTableSize()` `MMU`
Returns the number of frames in the simulated machine. This method is defined in the superclass of `MMU`, and is inherited.

◇ `final public ThreadCB getValidatingThread()` `PageTableEntry`
Returns the validating thread of the page.

◇ `final public void setValidatingThread(ThreadCB thread)` `PageTableEntry`
Sets the validating thread of the page.

Summary of Class `MMU`

The memory-management unit defines the hardware characteristics of the simulated computer. These characteristics and their access methods are described below.

Frame table: The table whose entries describe the individual main memory frames in *OSP 2*. The methods provided for accessing this table are: `getFrame()`, which returns a frame object at a given index in the frame

table; `setFrame()`, which sets a frame table entry with the given index; `getFrameTableSize()`, which returns the number of entries in the frame table (i.e., the number of main-memory frames in the system). These methods can be used to traverse the frame table in a `for`-loop.

Number of bits in a virtual address: The number of bits determines the maximum addressable space in the simulated computer. For instance, 16 bits yield 2^{16} bytes of addressable space (64Kb). The method to find out this value is `getVirtualAddressBits()`.

Number of bits used to represent the offset within pages: This property directly affects the size of the pages (and frames) in the computer. For instance, 10 bits lead to 1Kb pages, while 12 bits mean that the pages are 4Kb large. The method to find out this value is `getPageAddressBits()`.

Page table base register: This register points to the page table of the running task. It is available through the methods `getPTBR()` and `setPTBR()`.

5.7 Class `PageFaultHandler`

This class contains only one method that you are required to implement as part of the project. However, you might want to define additional methods to make the implementation more modular.

◇ `public static int do_handlePageFault(ThreadCB thread, int referenceType, PageTableEntry page)`

This is the actual pagefault handler. The `thread` and the `page` arguments are the thread and the page that caused the pagefault. The `referenceType` argument can be `MemoryRead`, `MemoryWrite`, or `MemoryLock`; it represents the type of memory reference that caused the pagefault. Knowing the type of memory reference is needed to set the dirty bit correctly. If the pagefault was caused by locking (in method `do_lock()` of `PageTableEntry`), the reference type must be `MemoryLock`. Note that locking does not modify the contents of a page, so the page should *not* be marked dirty due to this type of memory reference.

The implementation of this method follows the general outline of pagefault processing described earlier. However, it is also necessary to check several exception conditions. First, the pagefault handler might be called incorrectly by the other methods in this project. So, always check if the page that is passed as a parameter is valid (already has a page frame assigned to it) and return `FAILURE` if it is. Second, it is possible that all frames are either locked

or reserved and so it is not possible to find a victim page to evict and free up a frame. Return `NotEnoughMemory` if this is the case. Third, the thread that caused the pagefault can be killed by the simulator at any moment after the thread goes to sleep waiting for the swap-out or swap-in to complete. `FAILURE` should be returned in these cases.

The first two exceptional conditions must be checked at the beginning of pagefault processing, and the tests for destroyed threads must be done right after each swap-out and swap-in. In any case, before exiting, all threads that might be waiting on the page (see the explanations for `lock()` and `refer()`) must be notified using the `notifyThreads()` method of class `Event`. Finally, `dispatch()` must be called.

In case of an exception, you should always think of the appropriate ways to handle the various bits associated with pages and frames. For instance, if the thread that caused the pagefault was killed while waiting for a swap-out, we cannot be sure whether the frame has become clean or not. So, the dirty bit should not be changed in such a case.

The normal processing of a pagefault goes as follows. First, the thread must be suspended on a `SystemEvent` object created with the help of the `SystemEvent()` constructor. This event object must be saved in a variable, because when pagefault handling is finished you must resume the thread by executing `notifyThreads()` on that event.

Next, a suitable frame must be found *and reserved* to protect it from theft by other invocations of the pagefault handler (on behalf of other threads). If the frame is free, the page's frame attribute can be updated and a swap-in operation can be performed right away. If the frame contains a clean page, the frame should be freed (explained below) and then a swap-in operation should be performed. If the frame contains a dirty page, then swap-out must be performed, followed by freeing the frame, followed by a swap-in. If all is well and the thread was not killed while waiting for the two I/O operations, you update the page table (explained below) to indicate that `page` is now valid and the frame table to indicate that the newly freed frame is now occupied by `page`. Finally, the following actions must be performed:

– the frame used to satisfy the pagefault should be un-reserved

– the threads that might be waiting on `page` should be notified using `notifyThreads()`

– the thread that caused the pagefault must be resumed by executing `notifyThreads()` on the system event that you used to suspend the thread just after the entry into the pagefault handler

– `dispatch()` must be called

– `SUCCESS` should be returned.

Freeing frames: To free a frame, one should indicate that the frame does not hold any page (i.e., it holds the `null` page) using the `setPage()` method. The dirty and the reference bits should be set to `false`.

Updating a page table: To indicate that a page P is no longer valid, one must set its frame to `null` (using the `setFrame()` method) and the validity bit to `false` (using the `setValid()` method). To indicate that the page P has become valid and is now occupying a main memory frame F, you do the following:

– use `setFrame()` to set the frame of P to F

– use `setPage()` to set F's page to P

– set the P's validity flag correctly

– set the dirty and reference flags in F appropriately.

Performing a swap-in: This is done by issuing a read command on the swap file of the task that owns the page.

Performing a swap-out: This is done with the write command on the swap file of the task that owns the page.[3]

`read()` is a method of class `OpenFile` that is invoked on an `OpenFile`-object (which in our case is an open-file handle of a swap file) and takes three arguments: the *block number* in the file that is to be read, the *page* into which the file block is to be placed, and the *thread* that initiated the I/O. All these parameters can be obtained using the methods listed below. The only peculiarity is that a swap file contains an exact image of the task's memory, so there is a one-to-one correspondence between the pages and the blocks in the swap file. In other words, the block number should be equal to the page id.

`write()` is also a method in class `OpenFile` that is invoked on an open-file handle and takes the same arguments as `read()`.

Both `read()` and `write()` are blocking operations, i.e., they block the execution of the current thread until the I/O is finished.

Earlier we mentioned the method `getValidatingThread()`, which can be used to find out if a particular page is in the middle of a pagefault. It should

[3] Note: It must be the task of the page, not of the thread. Indeed, in case of a swap out, the thread and the page might belong to different tasks. Think why.

be emphasized, however, that it is the responsibility of the pagefault handler (i.e., your implementation) to maintain the validating threads correctly. In particular, when a pagefault occurs you must set the validating thread to be the thread that caused the pagefault and set it to `null` when the pagefault is over. All this is done with the help of the method `setValidatingThread()` of the class `PageTableEntry`. It should also be mentioned that *OSP 2* monitors the validating thread field in every page and issues error messages when it is incorrect. In particular, if a pagefault must occur and the validating thread of a page stays null, it might complain that your implementation missed the interrupt.

Relevant methods defined in other classes. In addition to the relevant methods listed earlier, the following methods are used in handling pagefaults:

◇ `public final boolean isReserved()` — `FrameTableEntry`
Tests if the frame is reserved.

◇ `public final boolean isDirty()` — `FrameTableEntry`
Tells if the frame is dirty by checking the "dirty" bit of the frame.

◇ `public final void isReferenced()` — `FrameTableEntry`
Checks the reference bit and tells if the frame has been referenced.

◇ `public final OpenFile getSwapFile()` — `TaskCB`
Returns the open swap file of the task. This swap file is then used in the `read()` and `write()` statements to perform the swap-in and swap-out operations. The swap file is represented by the `OpenFile` class, which is a handle that contains information about the disk blocks used by the file and some runtime information about the current status of the file. This operation blocks the current thread until the I/O operation is finished.

◇ `final public void read(int blockNumber, PageTableEntry memoryPage,ThreadCB thread)` — `OpenFile`
This method is invoked on an open-file handle (which is an instance of class `OpenFile`). It reads block `blockNumber` from the file (specified by an open-file handle) into page `memoryPage` on behalf of `thread`. The open-file handle mentioned above is an object of class `OpenFile`. In our concrete case, it would be a handle of a swap file. Since here `read()` is used for swapping pages into the memory, blocks in the swap file must directly correspond to pages in the main memory. Therefore `blockNumber` is determined by the ID of `memoryPage`. This operation blocks the current thread until the I/O operation is finished.

◇ `final public void write(int blockNumber, PageTableEntry memoryPage,ThreadCB thread)` — `OpenFile`

This method is invoked on an open-file handle (which is an instance of class `OpenFile`). It writes page `memoryPage` to block `blockNumber` of the file on behalf of `thread`. As in the case of `read()`, `blockNumber` is determined by the ID of `memoryPage`.

◇ `public void notifyThreads()` `Event`
Resumes all threads that might be waiting on the event. In pagefault handling, these are the threads that might be waiting on the page that has caused a pagefault and is being swapped in.

◇ `final public void suspend(Event event)` `ThreadCB`
Suspends the thread that calls this method, placing it on the waiting queue of `event`.

◇ `final static public void dispatch()` `ThreadCB`
Dispatches a thread.

◇ `final public ThreadCB getValidatingThread()` `PageTableEntry`
Returns the validating thread of the page.

◇ `final public void setValidatingThread(ThreadCB thread)` `PageTableEntry`
Sets the validating thread of the page. Note that you have to make sure that the validating thread of a page is set correctly by the pagefault handler. In other words, you must set the page's validating thread using `setValidatingThread()` when a pagefault happens and you must set it back to null when the pagefault is over.

◇ `final public static int handlePageFault (ThreadCB thread, int referenceType, PageTableEntry page)` `PageFaultHandler`
Invokes the pagefault handler. Returns `SUCCESS` if the pagefault has been handled successfully. Otherwise (for instance, it there is not enough memory) returns `FAILURE`.

◇ `public SystemEvent(String name)` `SystemEvent`
Constructor for system events. Used to create an event on which to suspend a thread at the beginning of pagefault processing. The argument, `name`, is a string that will appear in the system log and can help distinguish this event from other types of `SystemEvent`.

◇ `static public void create(String name, DaemonInterface work, int interval)` `Daemon`
Used to register a daemon with the system. See Section 1.7 for details.

In addition most of the methods in class `FrameTableEntry` (such as `getPage()`, `setReserved()`, etc.) are required for the implementation of the *OSP 2* pagefault handler.

Summary of Class `PageFaultHandler`

This class does not maintain important data structures of its own. However, it plays a central role in memory management by initiating the I/O operations that swap pages in and out of the system and by maintaining the page tables of the running processes and the frame table of the entire system.

5.8 Methods Exported by Package MEMORY

The following public methods are defined in the classes of the MEMORY package. They are useful for implementing other student modules and are also used to implement the methods that are part of the current project. To the right of each method we list the class of the objects to which the method applies.

◇ `static public PageTable getPTBR()` — `MMU`
Returns the page table base register of the MMU, which is supposed to point to the page table of the currently running thread; or it is `null` if no thread is running.

◇ `static public void setPTBR(PageTable table)` — `MMU`
This method changes the value of the page table base register.

◇ `static public int getVirtualAddressBits()` — `MMU`
Returns the number of bits used to represent an address. This method is defined in `IflMMU` and is inherited.

◇ `static public int getPageAddressBits()` — `MMU`
Returns the number of bits used to represent the page-number portion of an address. This method is defined in `IflMMU` and is inherited.

◇ `public final boolean isValid()` — `PageTableEntry`
Tells if the page is valid by checking the validity bit.

◇ `public final void setValid(boolean flag)` — `PageTableEntry`
Sets the validity bit of the page to `flag`.

Notice that there is a difference between setting the valid flag and setting the frame of a page (using `setFrame()`). The frame is set just before the swap-in operation so that the I/O subsystem will know which frame to load the page into. The method `setValid()` is used only after this operation is complete.

◇ `public final FrameTableEntry getFrame()` — `PageTableEntry`
Returns the frame of the page (or `null`).

◇ `public final void setFrame(FrameTableEntry frame)` — `PageTableEntry`

Sets the frame of the page to `frame`. If the page is being evicted, then `frame` is `null`.
`setFrame()` must be called before swapping in a page and after the page becomes invalid. In the former case, you need to set the frame of the page to tell the I/O subsystem where to put the page. The validity bit of the page should be set only after the page is loaded.

◇ `public final int getID()` — `PageTableEntry`
Returns the ID of the page.

◇ `public final TaskCB getTask()` — `PageTableEntry`
Returns the task that owns the page.

◇ `final public ThreadCB getValidatingThread()` — `PageTableEntry`
Returns the validating thread of the page.

◇ `final public void setValidatingThread(ThreadCB thread)` — `PageTableEntry`
Sets the validating thread of the page.

◇ `public final void isReferenced()` — `FrameTableEntry`
Checks the reference bit and tells if the frame has been referenced.

◇ `public final void setReferenced(boolean flag)` — `FrameTableEntry`
Sets the reference bit to the value of `flag`.

◇ `public final boolean isDirty()` — `FrameTableEntry`
Tells if the frame is dirty by checking the "dirty" bit of the frame.

◇ `public final void setReserved(TaskCB t)` — `FrameTableEntry`
Sets the frame as reserved by task t.

◇ `public final TaskCB getReserved()` — `FrameTableEntry`
Returns the task that has reserved this frame or `null`.

◇ `public final void setUnreserved(TaskCB t)` — `FrameTableEntry`
Un-reserves the frame previously reserved by task t; error, if the frame is not reserved by t.

◇ `public final void setDirty(boolean flag)` — `FrameTableEntry`
Sets the dirty bit to `flag`.

◇ `public PageTableEntry pages[]` — `PageTable`
This is the array that represents the page table. It must be initialized by the page table constructor described in Section 5.5.

◇ `public final TaskCB getTask()` — `PageTable`
Returns the owner task of the page table.