P4: xv6 Memory Encryption

Due Jul 25 by 11:59am **Points** 120 **Available** Jul 6 at 12am - Jul 26 at 11:59pm 21 days

This assignment was locked Jul 26 at 11:59pm.

Project 4: Kernel Memory Encryption

You can work on this project with one other partner. P4 is due Friday, July 23.

Objectives

- To learn about the virtual memory system in xv6
- To understand page table entries in detail
- To modify page table entries to be able to detect the current state of a page
- To modify the trap handler to be able to handle the page fault
- To implement the clock algorithm to keep track of the most referenced page

Background

The page tables, traps, and interrupts of xv6 are described in Chapters 2 and 3 of the **xv6 book. (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)**

For this project, we are providing you with a modified version of xv6 that implements a few features you may find helpful completing this assignment. A significant portion of the background here is dedicated to describing what's in the modified version.

You can find a copy of the modified xv6 source code in `/p/course/cs537-yuvraj/public/xv6-p4.tar.gz`. Copy it into your private directory and untar it.

```
prompt> cp /p/course/cs537-yuvraj/public/xv6-p4.tar.gz /path/to/your/private/dir
prompt> tar -xvzf xv6-p4.tar.gz
```

In your Makefile, make sure you set CPUS = 1 and **change the compilation flag from O2 to O0 (not Og).**

Main Changes in xv6-p4:

As you know, the abstraction of an **address space** per process is a great way to provide **isolation** (i.e., protection) across processes. However, even with this abstraction, attackers can still try to modify or access portions of memory that they do not have permission for. For example, in a Rowhammer attack, a malicious process can repeatedly write to certain addresses in order to cause bit flips in DRAM in the nearby (physical) addresses that the process does not have permission to write directly. If pages are left in clear text in DRAM, it may be possible for a clever malicious process to read those pages (e.g., by using a Rowhammer attack to modify the page tables or by leveraging physical access to the hardware). Therefore, for extra security, a running process may wish for its pages to be stored in an encrypted format in memory -- as long as those pages are not being accessed frequently.

To simplify this project, we approximate the benefits of encryption by **flipping all bits in a page** (i.e., xor every bit with 1 or just use ~); actually performing encryption would require more computation and recording a corresponding key, but those are not relevant to this project.

User-level Memory Encryption

1. **Page Encryption**
   Any user process is able to encrypt a range of virtual pages with a new system call:

   ### int mencrypt(char *virtual_addr, int len)

   The virtual page associated with the parameter **virtual_addr** will be the starting virtual page. The parameter **len** specifies how many pages will be encrypted. The system call will not assume virtual_addr is always page-aligned. A successful call to **mencrypt** will encrypt the virtual addresses ranging from [PGROUNDDOWN(virtual_addr), PGROUNDDOWN(virtual_addr) + len * PGSIZE) and returns 0. For instance, suppose the page size is 4KB, a successful call to mencrypt(0x3000, 2) will encrypt the virtual addresses in the range [0x3000, 0x5000). A call to mencrypt(0x3050, 2) will do the same.
   - The implementation of **mencrypt(char *virtual_addr, int len)** will execute successfully in the following cases and return 0:
   1. When the parameter **len** equals **0,** the implementation does nothing. This happens before doing any error checking.
   2. When part of or all the pages in the range have already been encrypted: **Encrypted pages and their corresponding page table entries remain unchanged.** All the unencrypted pages are encrypted and the function returns 0.
   - The implementation will return -1 in the following cases:
   1. The calling process does not have permission or privilege to access or modify some pages in the range. The implementation will

return -1 without encrypting any page in this case. To be specific, either all the pages in the range are successfully encrypted or none of them is encrypted.
2. The parameter **virtual_addr** points to an invalid address (e.g., out-of-range value).
3. The parameter **len** is a negative value or a very large value that will let the page range exceed the upper bound of the user virtual space.

Additional tips:
- In order to encrypt the page residing in the physical memory, the system call will access and modify the physical memory from the kernel. You might want to understand the layout of the virtual address space of a process. Specifically, xv6 places the kernel in the virtual address space of each process from KERNBASE to KERNBASE + PHYSTOP; these addresses are mapped in physical memory from 0 to PHYSTOP. In other words, virtual address KERNBASE + **pa** is mapped to physical address **pa**. You might find the macro and constants defined in the **memlayout.h** can help you to do the translation between virtual and physical address. A good reference for the xv6 memory layout is Figure 2-2 (Page 31) from the xv6 reference **book (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)** .
- There are macros defined in **mmu.h** and **memlayout.h,** including PGROUNDDOWN and PGROUNDUP that round down and up the virtual address to the nearest page-aligned address, respectively. Note that you calculate a particular virtual address's page number by rounding down.
- Understanding the functions defined in **vm.c** (e.g., **walkpgdir(), uva2ka(), and copyout()**) might be helpful because you may need to either use or modify those routines, or implement similar functionality. **uva2ka()** can help with error checking. You can pass a virtual address to this function and it will return a null pointer or pointer to 0 if there is an error. Just remember to modify uva2ka to handle the PTE_P bit being clear for encrypted pages.
-The TLB caches virtual address translations. If you edit the page table in memory, the CPU has no way of knowing that the page table in memory has changed. Unless you flush the TLB, it's possible that the CPU will not see those changes immediately. So make sure to **flush the TLB after modifying the page table.** One way to do this is to overwrite the `CR3` register (page table base register) even with the same value by calling switchuvm() to the same process, and thus flush the TLB

2. **Page Decryption**
- An encrypted page must be decrypted whenever it is accessed, so the kernel must catch a user's access to any encrypted page. To do this, xv6-p4 modifies the kernel to use one bit (called PTE_E) in page table entries (PTEs) to record whether or not the corresponding page is currently encrypted (fortunately, there are plenty of unused bits in the current PTEs). It sets this bit to 1 in the

PTEs when the corresponding pages are encrypted.
- Secondly, recall xv6 is running on emulated x86 hardware, so every time a user process tries to access some virtual address, the hardware walks the page tables to find the PTE and grab the corresponding physical address translation. The OS usually isn't involved when doing the address translation -- except when there is a page fault (i.e., PTE_P bit is not set). So, the trick is, **xv6-p4 will clear the PTE_P bit when it sets the PTE_E bit**. Then, when a user process tries to access this page, a page fault will be triggered, and it looks at the faulting address; if the faulting address occurred for a page where PTE_P was cleared and PTE_E was set, it knows to decrypt the page, reset the appropriate bits in the PTE, and return from the trap.
- Note that after a page has been decrypted, it will stay decrypted until being encrypted again. Repeated accesses to that (previously encrypted) page will not cause additional page faults into xv6; after a page has been decrypted, the hardware will be able to walk the page tables without involving the OS. In addition, when a child process is created, its initial memory state (including whether a page is encrypted or not), matches that of its parent.
- Keep in mind that page faults and memory errors, in general, are still possible - not all traps that are marked as page faults are necessarily decryption requests. In these cases, xv6-p4 will just silently exit() from trap.c.

Additional Tips:
- You might want to look through the information in **mmu.h** in detail. For instance, you will see the format for 32-bit virtual addresses (defined by the x86 architecture): 10 bits for the page directory index, 10 bits for the inner page table index, and then 12 bits for the offset within a page. Next in **mmu.h,** you will see the format of a **PTE** itself (again defined by the x86 architecture). From the macro PTE_ADDR, you can see that the upper 20 bits designate the address (the physical page) stored in the PTE; from PTE_FLAGS, you can see that the lower 12 bits designate the flags in the PTE. From PTE_P, PTE_W, and PTE_U you can see that the 3 least-significant bits record whether or not the corresponding page is **present** (which we would have called "valid"), **writable** , and part of **user** address space. Which previously-unused bit amongst the flags does xv6-p4 use for the PTE_E bit? You might find Figure 2-1 (Page 30) in the xv6 reference **book (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)** is helpful for you to identify which bit(s) was previously unused.
- You might also want to learn how to set and clear certain bits in the PTE by reading through the code of other functions (e.g., **mappages() or clearpteu()**) defined in **vm.c**.

Other Parts of the Kernel

Remember that trick where we clear the PTE_P bit on an encrypted page table entry to trigger a page fault when it's accessed? That PTE_P bit is used by other parts of xv6 for various reasons. Originally, PTE_P equals 0 meant that this page is not present or is otherwise

invalid. But now if PTE_P equals 0, that page could be an encrypted page, which is a valid, in-use page. Therefore, xv6-p4 changes some of the original code that does the following kind of check:

```
if (*pte & PTE_P) // Check whether this pde is valid or in-used.
```

Statistics

In order to gather statistics about your memory system and test your implementation, you can use the following two syscalls:

1. int getpgtable(struct pt_entry* entries, int num)

This will allow the user to gather information about the state of the page table. The parameter **entries** is an array of pt_entries with **num** elements that should be allocated by the user application and are filled up by the kernel.
The header file which defines **struct pt_entry** is `ptentry.h` Do not edit `ptentry.h`.

```
struct pt_entry {
    uint pdx : 10; // page directory index of the virtual page (see PDX macro defined in mmu.h)
    uint ptx : 10; // page table index of the virtual page (see PTX macro defined in mmu.h)
    uint ppage : 20; // physical page number
    uint present : 1; // Set to 1 if PTE_P == 1, otherwise 0. THIS IS A SIMPLE DUMP OF THE PTE_P BIT!
    uint writable : 1; // Set to 1 if PTE_W == 1, otherwise 0
    uint encrypted : 1; // Set to 1 if this page is currently encrypted, otherwise 0
};
```

Note that struct pt_entry uses bitfields to conserve space. Those fields that have a ': 1' next to them have a size of 1 bit. Attempting to set a value greater than 1 will cause an **overflow error**.
The kernel will fill up the entries array using the information from the page table of the currently running process. Only valid (allocated) virtual pages that belong to the user will be considered. When the actual number of valid virtual pages is greater than the **num**, filling up the array starts from the allocated virtual page with the highest page numbers and returns **num** in this case. You might find **sz** field in the proc structure of each process is useful to identify the most top user page. For instance, if one process has allocated 10 virtual pages with page numbers ranging from 0x0 - 0x9 and page 0x9 is encrypted, then page 0x9 - 0x7 is used to fill up the array when **num** is 3. The array would look as follows (ppage might be different):

```
0: pdx: 0x0, ptx: 0x9, ppage: 0xC3, present: 0, writable: 1, encrypted: 1
1: pdx: 0x0, ptx: 0x8, ppage: 0xC2, present: 1, writable: 1, encrypted: 0
2: pdx: 0x0, ptx: 0x7, ppage: 0xC1, present: 1, writable: 1, encrypted: 0
```

When the actual number of valid virtual pages is less than or equals to the **num,** the system call will only fill up the array using those valid virtual pages. It returns the actual number of elements that are filled in **entries**. The only error defined for this function is if **entries** is a null pointer, in which case it returns -1. The system call will also return -1 if it encounters any other error.

2. int dump_rawphymem(uint physical_addr, char * buffer)

This call allows the user to dump the raw content of one physical page where **physical_addr** resides (This is very dangerous! We're implementing this syscall only for testing purposes.). The kernel will fill up the buffer with the current content of the page where **physical_addr** resides -- it does not affect any of the page table entries that might point to this physical page (i.e., it won't modify PTE_P or PTE_E) and it doesnt't do any decryption or encryption. Note that **physical_addr** may not be the first address of a page (i.e., may not be page aligned). **buffer** will be allocated by the user and have the size of **PGSIZE**. Note that **argptr()** will do a boundary check, which would cause an error for the pointer **physical_addr.** Therefore, when you grab the value of **physical_addr** from the stack, use **argint()** instead of **argptr().**
dump_rawphymem will return 0 on success and -1 on any error.

Finally, we've covered the changes in xv6-p4, and are ready to find out what's required in P4 itself!

Main Idea: Kernel Memory Encryption Pager

In this project, we will explore the idea of letting the kernel manage page encryption and decryption. Encrypting pages with the mencrypt interface as it stands would require modifying each application, and require each application to know which pages are worth encrypting. You will modify the system to keep a fixed number of recently-accessed pages for each process stored in cleartext (decrypted). The idea is to minimize the number of page decryptions (and re-encryptions) by keeping each process's working set in cleartext.

Let the constant N be the number of recently accessed pages that should be tracked for each process (and kept decrypted). Add the following line to `param.h` which defines N:

```
#define CLOCKSIZE 8   // CLOCKSIZE represents N above
```

When an encrypted page is accessed by the user, a page fault should be triggered. If the number of currently decrypted pages of the calling process is smaller than N, then this page should be decrypted and pushed to the tail of a queue (see details in the next paragraph). If the calling process already has N decrypted pages, we need to find a victim page to replace.

In order to decide the victim page, you will implement a clock (also called FIFO-with-second-chance) algorithm. To implement this algorithm, you need to (statically) allocate a clock queue for each process. The clock queue is a queue-like structure storing all the virtual pages that are currently decrypted. The other essential part of a clock algorithm is a reference bit that gets set to 1 every time a page is accessed. Luckily for us, x86 hardware sets the **sixth** bit (or fifth if you start from 0. 0x020, to be precise) of the corresponding page table entry to 1 every time a page is accessed. Let's call this bit **PTE_A**. See Figure 2-1 (Page 30) in the xv6 reference **book (https://pdos.csail.mit.edu/6.828/2014/xv6/book-rev8.pdf)** for more details. The hardware-managed access bit should be cleared by the kernel (in software) at the appropriate time and have the hardware automatically set the bit when that page is accessed.

To select a victim, examine the page at the head of the queue. If the head page has been accessed since it was last enqueued (PTE_A is one), then clear the reference bit and move the node to the tail of the queue; the victim selection should proceed to the next page in the queue. Repeat this procedure until you find a head page that has not been accessed since it was last enqueued (reference bit is zero); this page should be evicted. When a page is "evicted", it should be encrypted and the appropriate bits in the PTE should be reset. When a virtual page is decrypted, it should be placed at the tail of the clock queue. The hardware will subsequently set PTE_A to 1, but there's no harm in manually setting it.

Make sure that pages that are in the working set are in cleartext (PTE_E is 0) and do not generate page faults (PTE_P is 1).

Note that

- In this part of the project, new user pages (including the program text, data, and stack pages) start as encrypted and are ONLY decrypted when accessed.
- When a child process is created, its initial memory state (including whether or not a page is encrypted) and clock queue, should match that of its parent.
- If a decrypted page is deallocated by the user, it should be removed from the clock queue.
- If a process calls exec(), it starts with fresh memory, and thus the working set should be emptied. All user-level pages should be encrypted.

Example 1:

Suppose we have one running process A, N = 2, and the state of the clock queue (leftmost is the head) is as follows:

Virtual page number: 0x3 Reference bit: 1	Virtual page number: 0x4 Reference bit: 1

When A accesses virtual page 0x5, a victim page should be selected. In the first iteration, the reference bit of both virtual pages is cleared and the order is not changed. On the second iteration, virtual page 0x3 will be selected as the victim. The resulting state of the clock queue will be the following:

Virtual page number: 0x4 Reference bit: 0	Virtual page number: 0x5 Reference bit: 1

Example 2:

Suppose we have one running process A, N = 4, and the state of the clock queue (leftmost is the head) is as follows:

Virtual page number: 0x3 Reference bit: 1	Virtual page number: 0x4 Reference bit: 0	Virtual page number: 0x5 Reference bit: 0	Virtual page number: 0x6 Reference bit: 1

When A accesses virtual page 0x4, the reference bit of virtual page 0x4 should be set to 1 again,

Virtual page number: 0x3 Reference bit: 1	Virtual page number: 0x4 Reference bit: 1	Virtual page number: 0x5 Reference bit: 0	Virtual page number: 0x6 Reference bit: 1

Then A access an encrypted virtual page 0x7. Virtual page 0x5 will be chosen as the victim and the resulting state would be

Virtual page number: 0x6 Reference bit: 1	Virtual page number: 0x3 Reference bit: 0	Virtual page number: 0x4 Reference bit: 0	Virtual page number: 0x7 Reference bit: 1

Statistics

In order to gather statistics about your memory system and test your implementation, you will create modified implementations of the two syscalls described previously. **BOTH** functions are slightly different from before.

1. int getpgtable(struct pt_entry* entries, int num, int wsetOnly)

This will still allow the user to gather information about the state of the page table. The parameter **entries** is an array of pt_entries with **num** elements that should be filled up by the kernel. If **wsetOnly** is 1, you should filter the results and only output the page table entries for the pages in your working set. If **wsetOnly** is 0, then this function does not filter by the working set. Return an error if **wsetOnly** is any other value.

```
struct pt_entry {
    uint pdx : 10; // page directory index of the virtual page
    uint ptx : 10; // page table index of the virtual page
    uint ppage : 20; // physical page number
    uint present : 1; // 1 if page is present
    uint writable : 1; // 1 if page is writable
    uint user : 1;  // 1 if page belongs to user
    uint encrypted : 1; // 1 if page is currently encrypted
    uint ref : 1; // 1 if reference bit is set
};
```

The kernel should fill up the entries array using the information from the page table of the currently running process. Only valid virtual pages should be considered. In addition, filling up the array starts from the valid virtual page with the highest page numbers. For instance, if one process has allocated 10 virtual pages with page numbers ranging from 0x0 - 0x9, then page 0x9 - 0x7 should be used to fill up the array when **num** is 3. Assume all these three pages are in the clock queue, then the array should look as follows (ppage might be different):

```
0: pdx: 0x0, ptx: 0x9, ppage: 0xC3, present: 1, writable: 1, user: 1, encrypted: 0, ref: 1
1: pdx: 0x0, ptx: 0x8, ppage: 0xC2, present: 1, writable: 1, user: 1, encrypted: 0, ref: 1
2: pdx: 0x0, ptx: 0x7, ppage: 0xC1, present: 1, writable: 1, user: 1, encrypted: 0, ref: 1
```

Do not edit `ptentry.h`.

2. int dump_rawphymem(uint physical_addr, char * buffer)

This will allow the user to dump the raw content of one physical page where **physical_addr** resides (This is very dangerous! We're implementing this syscall only for testing purposes.). The kernel should fill up the buffer with the content of the page where **physical_addr** resides. **buffer**will be allocated by the user and have the size of **PGSIZE**. You are not required to do any error handling here.

> **Note**: Simply using copyout will not be sufficient. The buffer might be encrypted, in which case you should decrypt that page. Otherwise, when the user program tries to read buffer, it'll read flipped values. It's as simple as either using the buffer's uva for memmove (which copyout does not do) or touching the buffer using `*buffer = *buffer` before copyout.

Hints

Before you start with the coding part, try to understand the layout of the virtual memory and how to index and manipulate the page table in xv6. This includes:

1. How to grab certain entry from the page table

2. How to change a certain bit in the page entry

3. How to access the physical memory from the kernel

Reading through the existing codebase is a good start for you to figure out the above question.

Before you start, it may be helpful to understand the behavior of the new system calls in xv6-P4. We recommend you first write a few small user programs to test these new calls. A simple user application could look like the following:

```
char *ptr = sbrk(PGSIZE); // Allocate one page
mencrypt(ptr, 1); // Encrypt the pages
struct pt_entry pt_entry;
getpgtable(&pt_entry, 1); // Get the page table information for newly allocated page
```

Next we would suggest you break down this project into the following steps:

1. Make sure the system will use the decryption mechanism when an encrypted page is accessed, and init all the user pages as the encrypted state. You should do this whenever a process grows or shrinks (check out growproc()) or when a new program is executed (check out exec() in exec.c).

All the user pages are allocated through the function `allocuvm()` in `vm.c`, but directly encrypting all the pages in `allocuvm()` might not work out as you might expect. The reason is that another system call `exec` will call `allocuvm` to init those text, data, and stack pages and copy program content (e.g. program text) into it. If you do the encryption inside the `allocuvm()`, then you probably need to modify other functions like `loaduvm()` and `copyout()` to make sure that those pages are decrypted before the content is copied into it. Considering the difficulty you will encounter, we encourage you to do the initial memory encryption in two parts:

A. Encrypt all the newly-allocated heap pages in `growproc()`. These pages are allocated by the user through syscall `sbrk()` which will call `growproc()`.
B. Encrypt all those pages set up by the exec function at the end of the exec function. These pages include program text, data, and stack pages. These pages are not allocated through `growproc()` and thus not handle by the first case.

2. Add the clock queue mechanism. Make sure you encrypt pages when they get kicked from the queue and add pages to the queue when you decrypt them.

In this step, you will mainly extend your memory decryption implementation. Before you do that, you will need to determine what's kind of data structure you wish to use as a queue. This data structure should be able to append elements to the tail, check the head element, and remove elements in the middle of the queue. One of the choices is to use the linked list implementation similar to the one in P3. Another choice is to use a ring buffer as a queue. A ring buffer can be implemented using an array that representing the queue and an index pointer pointing to the head of the queue. Each method has its pros and cons that you need to deal with. So choose whatever way make you feel comfortable to implement. See TA's discussion material for more information.

Remembering that we should remove the pages from the queue if it's deallocated. `deallocuvm` might be one of the candidate places to remove the pages from the queue. But you should be aware that you can't directly call `myproc()` to get the clock queue in `deallocuvm` since this function could be called by the parent process to deallocate pages for the child process.

3. Modify the corresponding code to handle the `fork()` behavior and deallocation of pages. This is mainly just queue management.

As we mentioned above, the child process should inherit the page table entry flags and clock queue from the parent process. You probably want to check out the `fork()` function in `proc.c` and `copyuvm()` in `vm.c`. The modification mainly involves copying extra flags and clock queue. If you are using a linked list as the queue, you should do a deep copy instead of just copying the pointer.

Make sure you fully test your code after each step.

Code Delivery

Handing in Your Code

EACH project partner should turn in their joint code to each of their turnin directories.

So that we can tell who worked together on this project, each person should place a file named **partners.txt** in their p4 directory. The format of **partners.txt** should be exactly as follows:

```
cslogin1 wisclogin1 Lastname1 Firstname1
cslogin2 wisclogin2 Lastname2 Firstname2
```

It does not matter who is 1 and who is 2. If you worked alone, your **partners.txt** file should have only one line. There should be no spaces within your first or last name; just use spaces to separate fields.

The location for partners.txt will be **outside** the **src** directory, i.e.:

```
/p/course/cs537-yuvraj/turnin/login/p4/partners.txt
```

Within your p4 directory, make the following directories and place your xv6 code in them as follows:

```
/p/course/cs537-yuvraj/turnin/login/p4/ontime/src/<xv6 files>
```

If you wish to use slip days in this project, then you should submit your code to the corresponding slip directory: **slip1**, **slip2**, or **slip3. slip1** indicates that you wish to use one slip day in this project. We will use the latest submission for grading. This is saying that if you submit both at slip3 and slip1, then we will use the version submitted at slip3 to grade.

Suppose you are submitting to slip1, the final directory tree should look like:

```
/p/course/cs537-yuvraj/turnin/login/p4/partners.txt
/p/course/cs537-yuvraj/turnin/login/p4/slip_days -- should contain exactly a number 1
/p/course/cs537-yuvraj/turnin/login/p4/slip1/src/Makefile
/p/course/cs537-yuvraj/turnin/login/p4/slip1/src/proc.c
/p/course/cs537-yuvraj/turnin/login/p4/slip1/src/...
```

Testing

We strongly recommend you first write a few small user programs to test various aspects of this project. A simple user application could look as following

```
char *ptr = sbrk(PGSIZE); // Allocate one page
struct pt_entry pt_entry;
// Get the page table information for newly allocated page
// and the page should be encrypted at this point
getpgtable(&pt_entry, 1, 0);
ptr[0] = 0x0;
// The page should now be decrypted and put into the clock queue
getpgtable(&pt_entry, 1, 1);
```

Ensure correct behavior from these tests before moving on to our tester. The tester is at `/p/course/cs537-yuvraj/tests/p4/run-tests.sh` If you want to run just test n, you can run `/p/course/cs537-yuvraj/tests/p4/run-tests.sh -t n` On any CSL machine, use `cat /p/course/cs537-yuvraj/tests/p4/README.md` to read more details about how to list the tests and how to run the tests in batch. Note that there will be a small number of hidden test cases (25%).

NOTE: In your Makefile, make sure you set CPUS = 1 and change the compilation flag from O2 to O0 (not Og). Otherwise, the test suite wouldn't work because the compiler would optimize out some of the unnecessary access used in the tests.

Slip Day Policy

A maximum of 3 slip days can be used for this project no matter you are working with a partner or not. Additional 2 slip days for each one have been added as described in this **post (https://piazza.com/class/kolxzgun59y5rs?cid=87)** .

If you are working with a partner, then

- Each of you will only need to contribute 1/2 of any slip days you use; for example, if you use 1 slip day, each of you is charged 1/2 of a day.
- If only one of you runs out of slip days, the needed slip days will be taken from the partner who still has them.
- A 1/2 slip day can't be used (unless you are combining with a 1/2 slip day from your partner). We will assume you are aware of your partner's slip days and the implications.