Virtual memory

The size of virtual memory is much greater than the size of real memory banks attached to the MMU. Consequently, some memory requests originating from the CPU cannot be satisfied by the MMU and result in **memory faults** (Page faults or Segmentation faults) which are interrupts triggered by the MMU when it has no direct access to a requested memory location.

Virtual memory makes sense only if the logical address space is fragmented into small chunks that can be moved around independently; in practice this implies **paging** or/and segmentation. Paging is particularly well suited for virtual memory because the size of a page is fixed and can be made compatible with the size of a disk sector (making I/O transfer as fast as possible).

Hence, we will focus on VM implemented using paging.

Virtual memory is not imaginary

In a virtual memory system, a page belonging to the logical address space of a process can be stored:

In main memory: if it is there, it is immediately accessible.

On a swap device (hard drive): if it is on a drive, it has to be brought into main memory before it can be accessed.

Nowhere: it does not exist. It can be accessed immediately, if accessing it is legal at all; otherwise accessing it results in an error.

If a page is in main memory, it is quite possible that another copy of it exists on disk. This may lead to an increase in efficiency.

The basic idea behind virtual memory is to extend the use of memory protection offered in paging. Page tables have special entries informing that a page is not accessible or does not exist.

Virtual memory simply replaces the term **does not exist** with **exists elsewhere or nowhere at all**.

Page table

Frame	V	W
	1	1
	1	1
	1	1
	1	1
	0	0
	0	0
	0	0
	0	0
	0	0
	0	0
	0	0
	0	0
	0	0
	1	1
	1	1
	0	0
	1	1
	0	0
	1	1
	1	1
	1	1
	1	1
	1	0
	1	0
	1	0
	1	0
	0	1
	0	1

V: **Valid** bit (also called Present). If this bit is 0, the MMU will trigger an interrupt when it attempts to access this page table entry.

W: if this bit is a 1, the page can be modified; otherwise, the MMU will trigger an interrupt when the CPU attempts to execute a store instruction targeting this page.

C/W: (Copy–on–Write) will be discussed in due time (it is useful after a **fork()**).

other bits: they exist.

A reference to an entry with the Valid bit off results in a page fault. This allows the OS to implement the most popular form virtual memory management: **demand paging**.

Demand paging

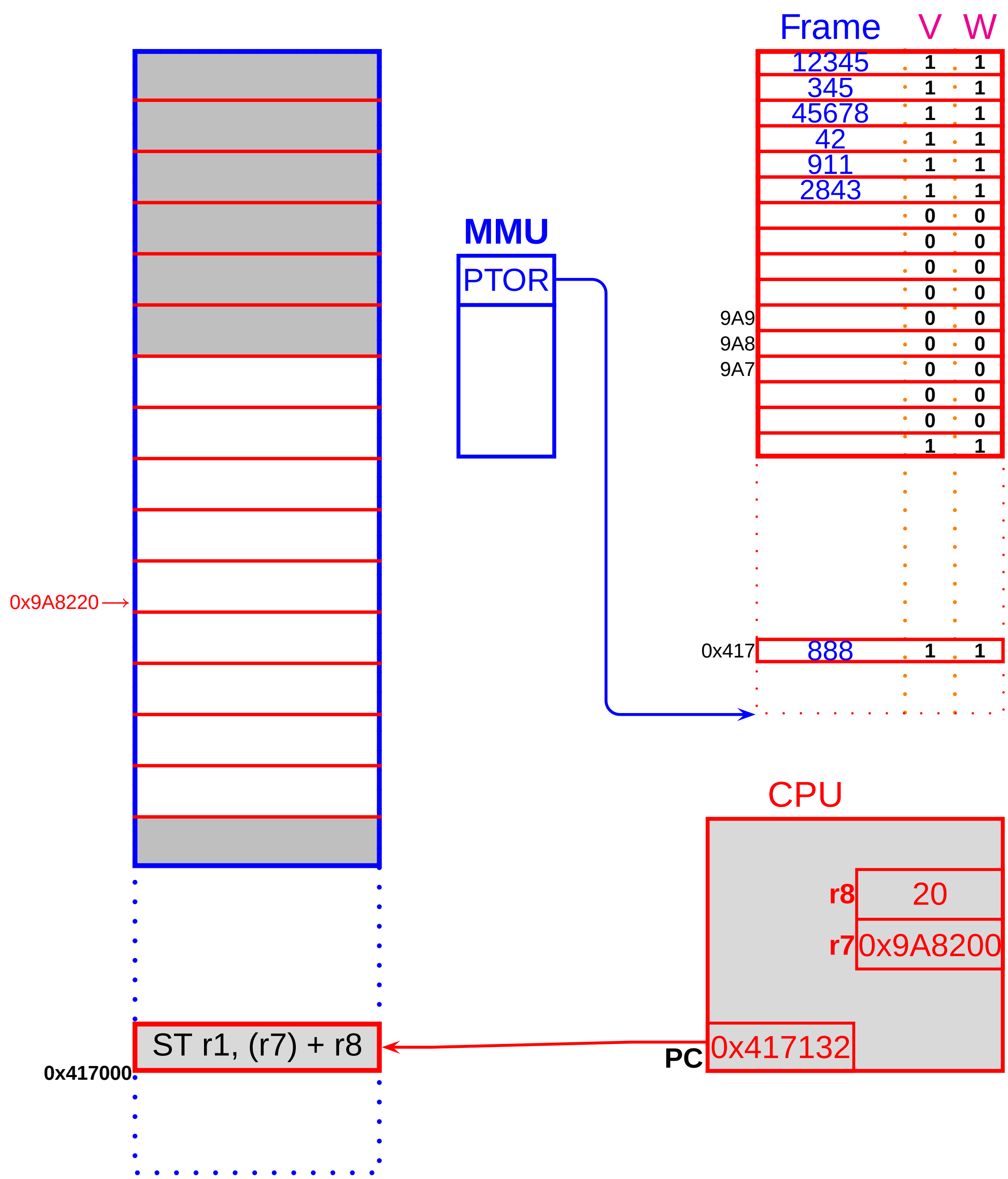
A system that brings (or creates) pages only when they are immediately needed is said to use demand paging.

Demand paging operates as follows:

- A process starts with no pages residing in main memory.
- The first memory reference made by a process causes a page fault. This fault causes the OS to either bring the needed page from the swap device (if it is there) or to create a zero–filled page. The first fault definitely will bring a page of code from the swap device (Why?).
- Subsequently, if a page fault is triggered in a legal context, the OS will either bring the page from the swap device or create a zero–filled page (as the circumstances dictate).
- If enough pages are brought into main memory, the supply of frames will be exhausted. When the OS needs a frame to accommodate a page fault and there are no free frames, a non–empty frame is picked and emptied (done by copying it to the swap device).

A reference to a vacuous place

The CPU is going to execute a machine instruction.

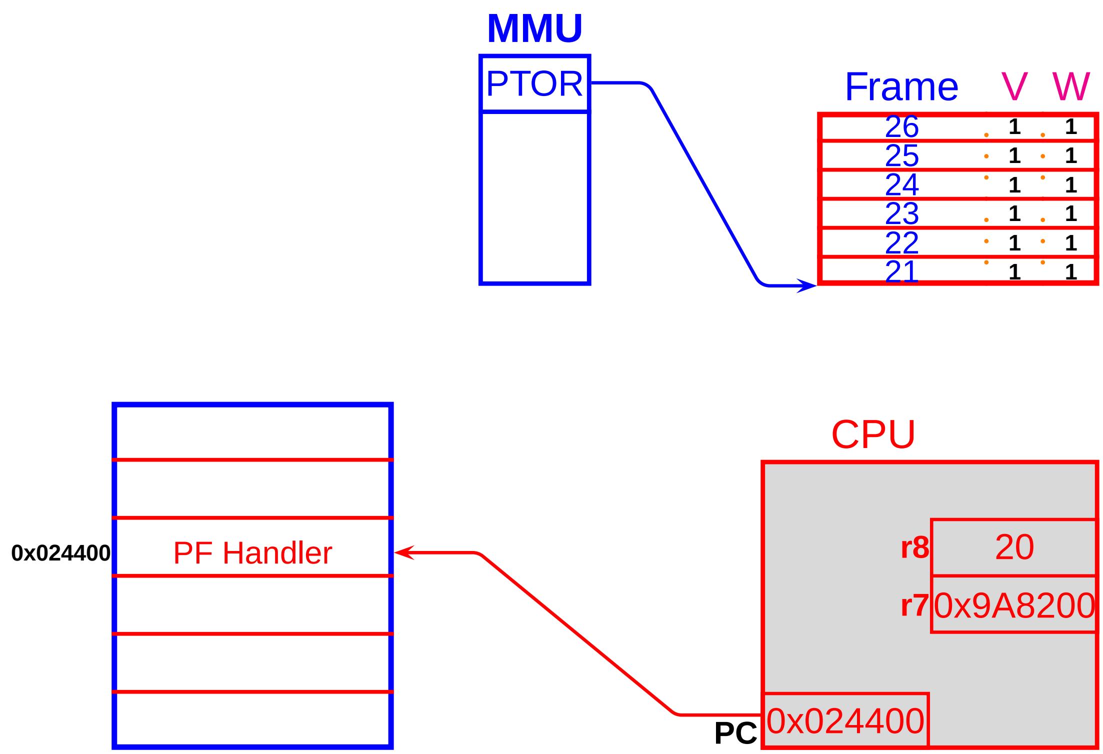


the road to a page fault

The following sequence of steps takes place:

1. The CPU attempts to fetch the machine instruction pointed to by the **PC** by handing the address **0x417132** to the MMU.
2. Page 0x417 is present, so the MMU fetches the contents of location 0x888132 from memory. The instruction is fetched without problems.
3. The CPU interprets the instruction and prepares arguments. (r7)+r8 is computed (equals 0x9A8220) and the execution of the instruction starts: the contents of r1 are placed in the Data Register of the MMU and 0x9A8220 is placed in the Address Register of the MMU.
4. The MMU fetches location PTOR + 0x9A8 and finds the page table entry invalid. It interrupts the CPU.
5. The CPU suspends the current instruction (page fault), save the PC (etc.), and jumps to the **Page Fault Handler**.

A context switch is unavoidable (Is it really?), so the running process is pushed out of the CPU.



The cause and context are determined. Two outcomes:

- it is a legitimate fault: a reference to a legal page that happens not to exist in main memory. The Page Fault Handler continues.
- It is a memory protection violation of some sort. An signal is prepared; the process is marked **ready** and the Scheduler is invoked.

The Page Fault Handler in action

1. An empty frame is located. If there are no empty frames, the **Page Replacement** algorithm is invoked; when it is done, there is a free frame. Next, the frame will be filled.

2. The current location of the desired page is found. Two possibilities:

   - It is on the swap device (disk). An **asynchronous** disk i/o transfer (disk⟶memory) is initiated.
   - It never existed. The frame is filled with zeroes. the faulting process is marked **ready**.

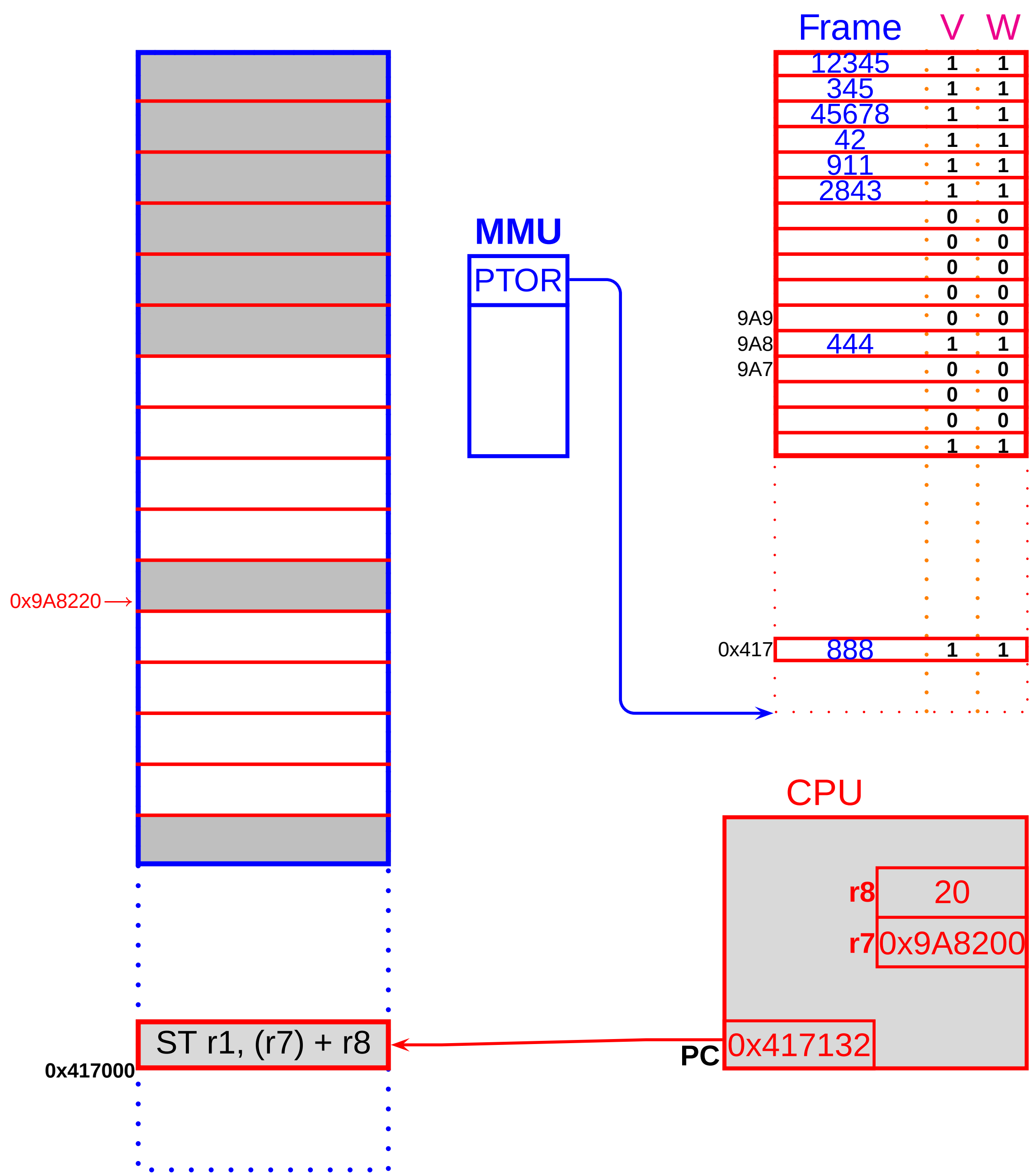
3. The page table of the faulting process is updated. The process is marked **blocked** (for i/o).

4. The handler jumps to the **Scheduler** which performs a context switch, giving the CPU to a ready process.

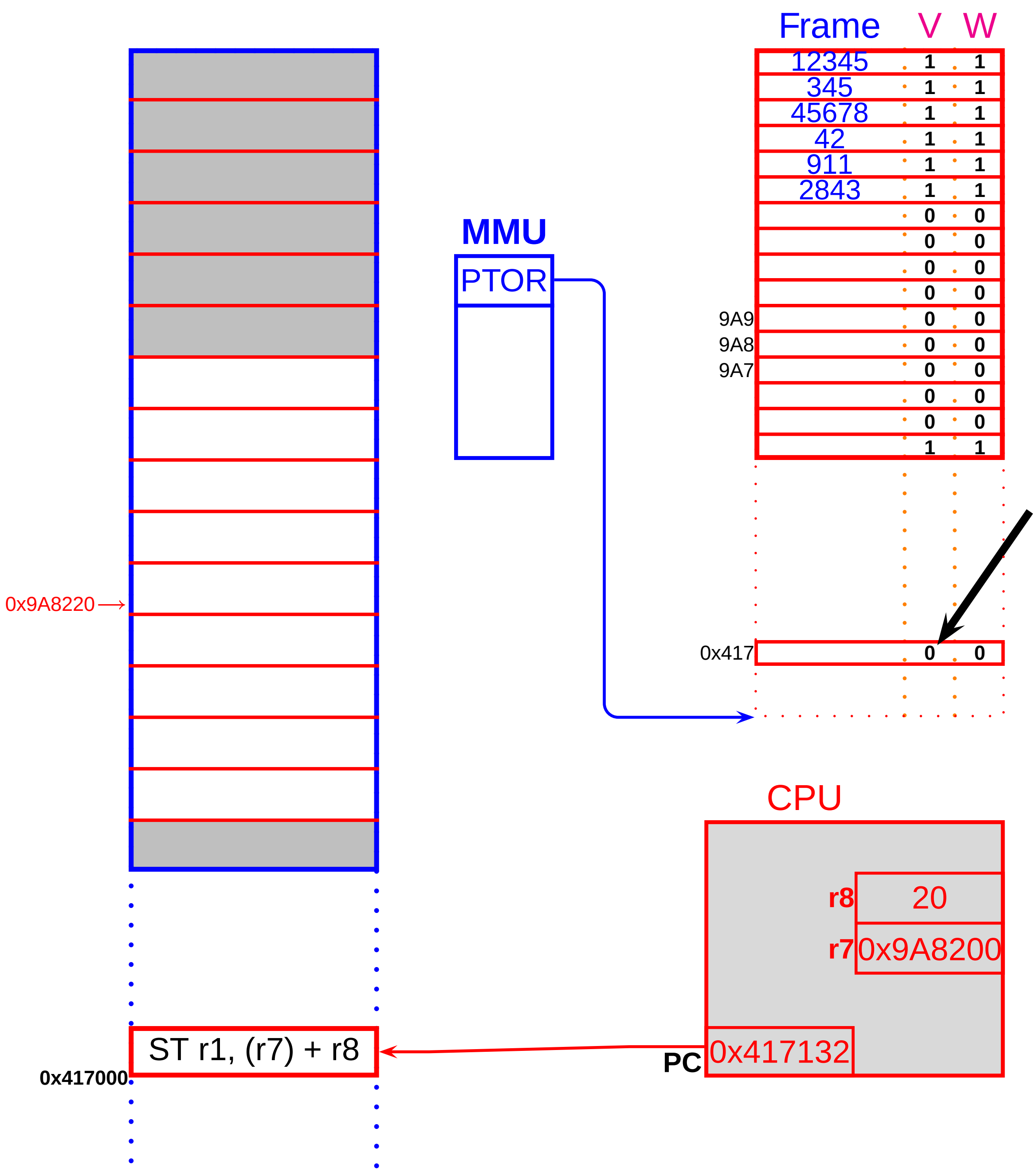
If a paging i/o operation is initiated, it will eventually terminate. If it terminates unsuccessfully, suitable steps are taken. Otherwise, the process is marked ready.

Some time later

Sooner or later, the Scheduler will give the CPU to the process. The interrupted instruction is continued:

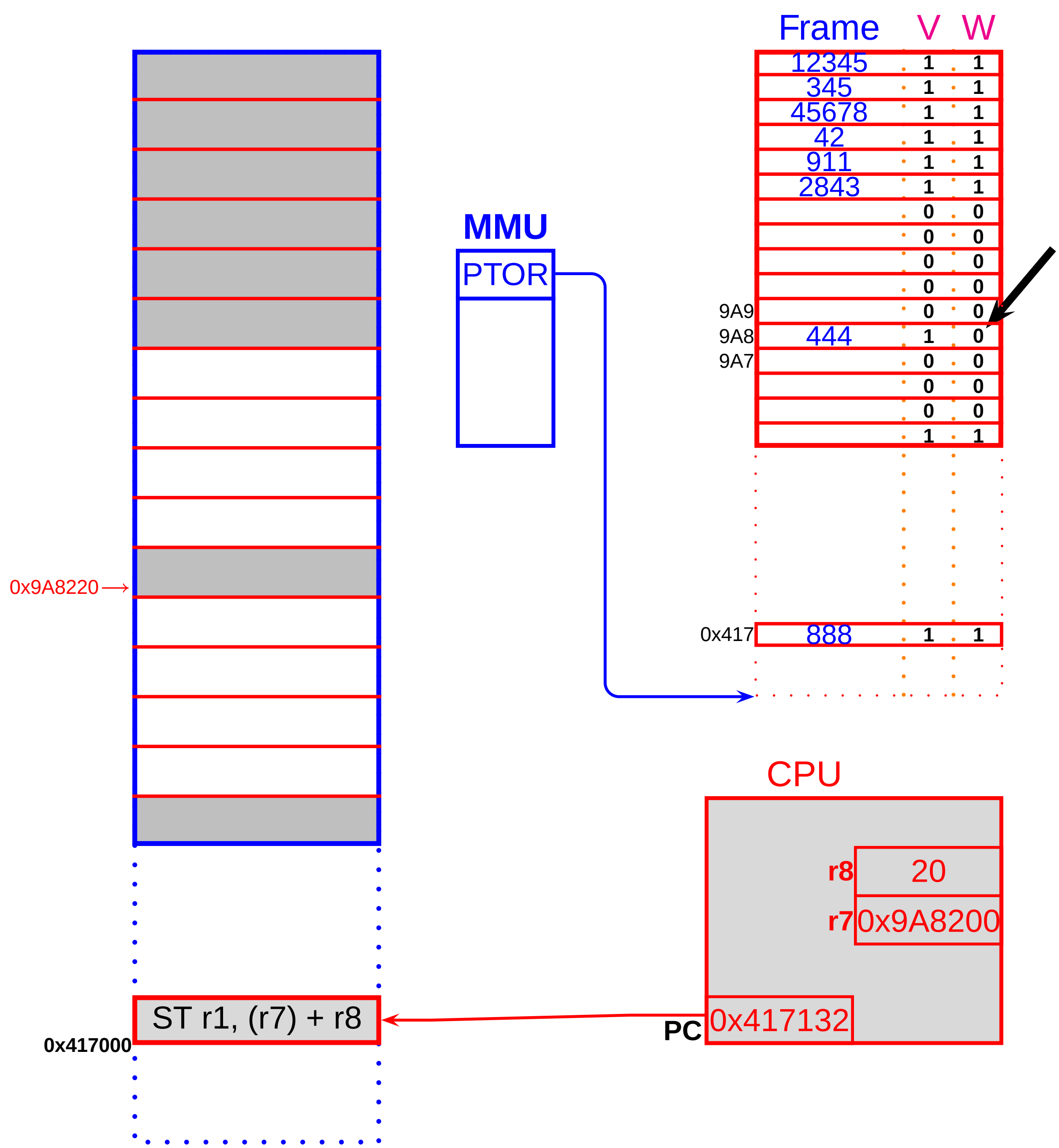


A variation



Frame
V
W
12345
345
45678
42
911
2843
1
0
9A9
9A8
9A7
0x417
MMU
PTOR
0x9A8220
0x417000
ST r1, (r7) + r8
CPU
r8
20
r7
0x9A8200
PC
0x417132

Another variation



Page replacement

When a **free frame** is needed as a result of a page fault, the kernel will look for one:

- It starts by trying to find a free frame. A frame may be free because a process just terminated (and all its pages became irrelevant, releasing the frames holding them) or because the system was booted recently and some frame have not been used so far. The first free frame found is used (they are all identical, so the first is as good as the second, etc.).
- If there are no free frames, the kernel calls in the Page Replacement algorithm; this algorithm will create a free frame by confiscating it. The **PR** inspects the occupied frames and picks one of them. It removes from it the page occupying it (called the victim) thus making it free. The **PR** always succeeds, even if the price is high (see **thrashing**).

We say that **PR** is **local** if it checks only frames belonging to the faulting process; otherwise, it is **global**. Global **PR** is more efficient, but is also more dangerous, because of **thrashing** and because it creates a security threat (I know how to steal all your frames).

When a process starts, it is given a number of frames. In global replacement this number usually is 1 (containing the entry point to the program) but does not matter because the number will change in time. In local replacement the number of frames given to a process is of crucial importance but is hard to guess.

Most systems use local replacement in the short term, adjusting the number of frames allocated to each process from time to time (based on page fault rates).

Page replacement

Several **PR** algorithms are used:

Not–Used–Recently:

Second Chance (Clock):

Least–Recently–Used:

Aging (pseudo LRU:

Belady's anomaly: this is not an algorithm but a strange behaviour that might be exhibited by a process.

Not–Used–Recently

The **NUR** algorithm has not been used recently because it is not very efficient. It is, however, cheap to implement without any special–purpose hardware.

The fate of pages in main memory is decided in **rounds**. A page may be swapped out if it had not been used during the latest round.

A soft timer is used to indicate to the kernel the beginning of a new round. When it ticks, all the entries in the current page table are marked invalid but are left otherwise intact (in particular, the frame number is kept, with 0 meaning “not there”).

The duration of a round is a mysterious parameter usually set to $\frac{1}{60}$s out of respect for the good old days.

During a round, the NUR algorithm behaves as follows:

- When a page fault occurs, NUR checks whether there is a frame number in the entry (i.e. not 0):
  - If so, the V bit is set and the faulting instruction is restarted (this is called a **minor fault** because it takes little time to handle).
  - If not so, this is a major page fault and NUR picks a victim: any entry in the page table with the V bit off and a legal frame number shows a suitable victim. The victim page is booted out and the faulting page gets the vacated frame.
- When a memory reference does not trigger a fault, we happily proceed on.

Note that NUR can only be **local** (Why?).

Second chance (Clock)

This algorithm is better than NUR but it requires a bit of extra hardware: an extra bit in each page table entry (the **R**eference bit). There are no rounds.

Whenever a page table entry is referenced (used for translation), the **R** bit is set.

A pointer (**hand**) points to the page table entry that will be the first pick in the search for a victim. the hand moves (in a circular motion) through the whole page table, so **SC** is fair.

When a page fault occurs, **SC** performs the following step in a loop:

```
while( PTE[hand].R ) {
    PTE[hand].R = 0 ;
    hand = (hand+1) % SIZE ;
}
```

This loop is bound to end; when it ends, **hand** points to the PTE of the victim page.

Least Recently Used

LRU is considered the “best” practical **PR** algorithm; it is too expensive to be implemented in full in normal computers, but cheaper variants of it are predominant.

LRU calls for victimising the page that was the least recently used, meaning that it has remained unused the longest (among the pages residing in frames).

One way to implement **LRU** is to have a **timestamp** field as part of every PTE. whenever the PTE is referenced, the **timestamp** is updated. When a victim is needed, the page with the oldest **timestamp** is it.

Slightly cheaper ways exist but none is cheap enough, so **LRU** remains an academic algorithm.

Aging

An 8–bit shift register is added to every **PTE**. Time is divided into rounds (as in NUR), with a soft timer ticking every, say, $\frac{1}{60}$s ($\frac{1}{50}$s in Europe).

These rules apply:

- Every time a **PTE** is referenced, the leftmost bit of the shift register is set to 1.
- Every time the timer ticks, all shift registers are shifted 1 bit to the right.
- When a page fault occurs (all page faults are major in LRU), the **PTE** with the smallest value in its shift register is the victim (the registers are treated as unsigned integers).

  If there is a tie, any tiebreaker can be used.

Belady's anomaly

Some **PR** algorithms (very rarely) exhibit a peculiar behaviour: they generate **more page faults** if they give **more frames** to a process.

Example: a process is made of 5 pages which it accesses in this order:

$$1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$$

If one uses a local FIFO page replacement algorithm, there will be 10 faults if the process is given 4 frames and 9 if the process is given 3.

Additional concepts

A few (of many) issues related to virtual memory:

I/O interlock: **Frames** involved in a paging i/0 transfer cannot be touched.

Thrashing: In a worst–possible scenario, every page reference may cause a page fault. If anything similar happens, the system will be practically idle with the exception of a frantically working hard drive busy moving pages back and from the swap device.

Copy–on–Write: When a **writable** page is duplicated, it will eventually need to be physically copied. The **C/W** bit in a page table allows to delay this action until it is necessary.

Dirty bit: A page that exists on the swap device does not have to be written out unless it was modified.

I/O interlock

Frames involved in i/o operations cannot be chosen for replacement because the device controller understands only physical addresses and is not aware of paging: it deals with a frame not a page.

Moreover, once an i/o transfer request reaches a device controller, it is impossible to modify (change the memory address) and to stop (usually).

Moral: frames involved in i/o operations are **locked**.

A note on segmentation: i/o interlock offers a beautiful mechanism for a seamless implementation of asynchronous i/o: the segment is simply locked until the transfer is done.

Consider this situation:

1. Page 0x234 resides in frame 0x5678.
2. The owning process executes a system call:

   read(fd , 0x234080 , 0x100) ;

   (The address is part of page **0x234**).
3. The kernel converts the logical address to a physical address which happens to be 0x5678080.
4. The **read** operation is enqueued (the disk controller is busy with another transfer) as a transfer to location 0x5678080.
5. A page fault causes the frame 0x5678 to be freed (page 0x234 is swapped out) and reallocated to page 0x888.
6. The i/o request if performed now, putting data into page 0x888 (sitting in frame 0x5678).

This situation must be prevented.

Thrashing

Consider the program:

```
main() {
    int *p = (int *)malloc( 4 * 102400 ) ;
    for( int i = 0 ; i < 1024 ; i++ )
    for( j = i ; j < 102400 ; j = j + 1024 )
      p[j] = i ;
    }
```

When the execution of this program starts, it will have 2 pages: a **text** page (with the code) and a **stack** page, first empty, then containing **p, i, j**.

After the **malloc** call, the **heap** will contain 100 pages,assuming a 4096B page.

The way the code is written, it will access a different page every time it performs p[j] = i (still assuming a page size of 4096B).

If this process is given 3 frames, the double loop will trigger 102400 page faults. If a page fault requires 2 ms (a fast disk i/o) and one loop iteration of the inner loop 20 ns (excluding the necessary context switch), we will observe a CPU utilisation of 0.001%. If the time of a context switch is included (say, 2000 ns), the appearance will be slightly better (0.1%) even though most of the time the CPU will be doing useless things. The utilisation of the swap device will approach 100%.

When one process starts to thrash because of shortage of frames, things get bad.

They get really bad if **global page replacement** is used. Then the thrashing process will **steal** frames from other processes; if those processes become short of frames, they will start to thrash, too.

The situation gets even worse because the **i/o interlock** rule will take the thrashing frames out of consideration for replacement.

Copy–on–Write

A process executes a fork(). Just before the fork, its page table looks like this:

Frame	V	W	C/W
225	1	1	
196	1	1	
169	1	1	
	0	0	
	0	0	
	0	0	
	0	0	
	0	0	
	0	0	
	0	0	
361	1	1	
324	1	1	
	0	0	
289	1	1	
	0	0	
2843	1	1	
911	1	0	
42	1	0	
45678	1	0	
345	1	0	
12345	0	1	

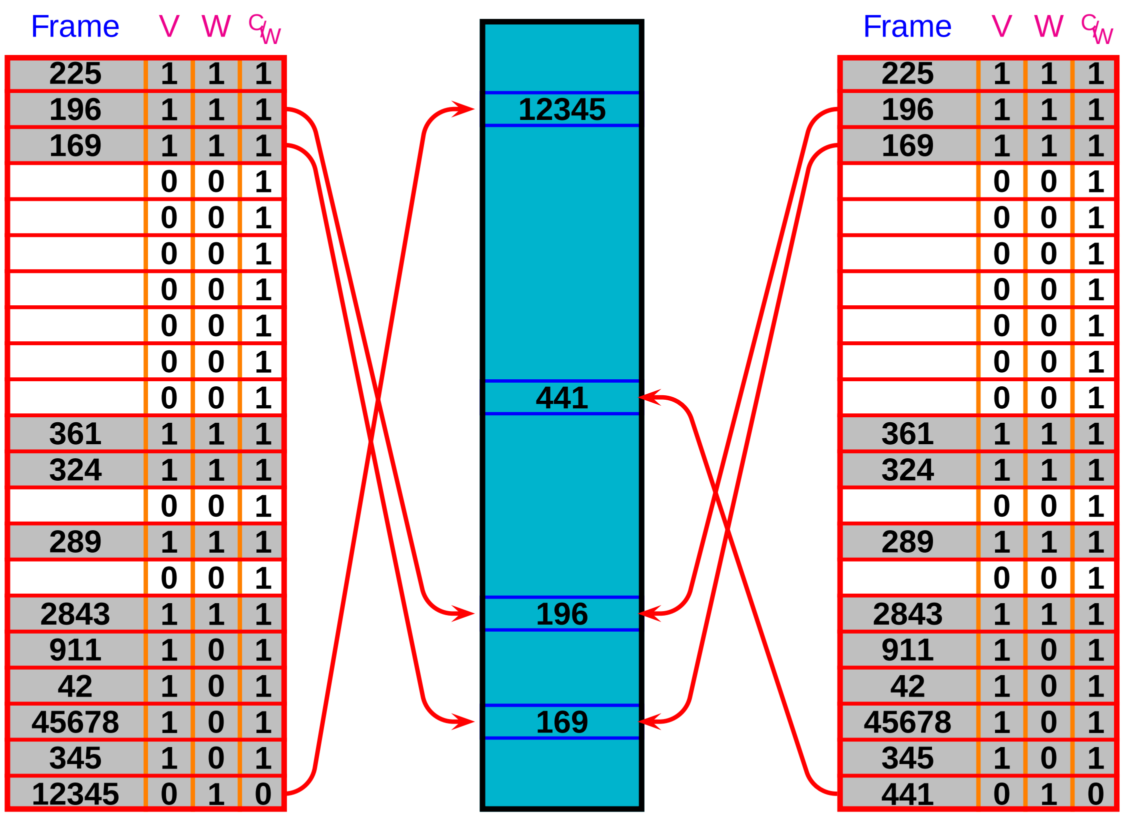
One can see the:

- u area
- text
- data
- stack
- and the rest known as the heap.

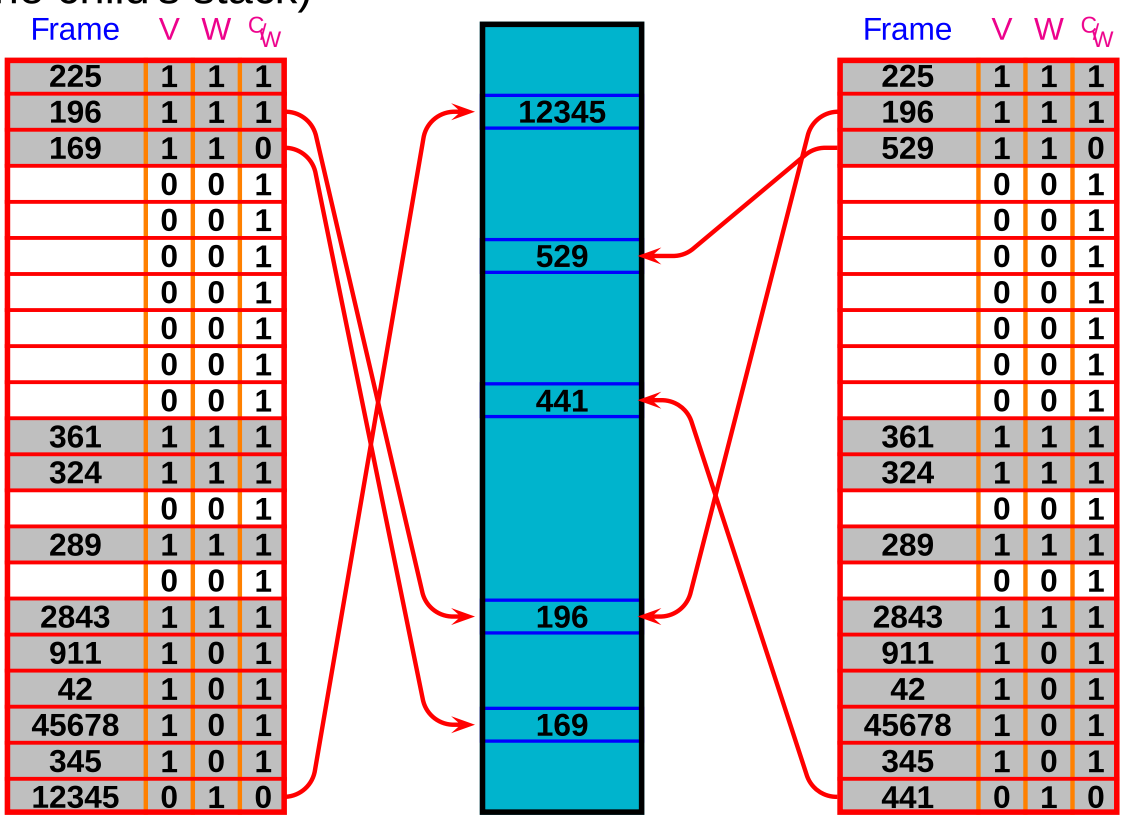
The C/W bits are turned **off**.

fork results in the following:

- A new process is created.
- As part of it, a new logical address space is created by making a copy of the page table of the parent. All the entries in both tables are marked **Copy–on–Write**.
- The u area of the child is updated before the child is ready. That will result in duplicating the pages forming the u area and (so that each occupies two frames) and modifying one copy (the child's).



The child process (eventually) starts executing. Its first instruction starts the action of returning from the fork. As part of it, the returned value (0) is pushed on the stack, causing a **page fault** (the **C/W** bit is on. The faulting page is duplicated and the child's copy updated (the 0 is pushed on the child's stack)



Modified page

Another trick used in page replacement is the addition of the Modified bit to each page table entry. Where it exists, the **M** bit is set every time the page table entry is used for a store operation.

When the page is being victimised, the **M** bit indicates whether a copy on the swap device is up–to–date or not.

Shared pages

Pages can be shared by existing in several page tables. This requires special care: when a shared paged is swapped out, all the page tables must be updated. The same is happens when it is subsequently swapped in.

To achieve this, the kernel keeps a list of all the shared structures. This list is updated during every context switch (the modified bit).