Temporal safety

In a programming language like C, memory management is the responsibility of the programmers. The programmers use malloc and free APIs to allocate and release memory on demand. Because the programmers can make mistakes in their program, it is possible that they accidentally freed a memory location that can be accessed (used) in the future. The temporal safety ensures the absence of use-after-free bugs. To illustrate why a use-after-free bug is dangerous, let us look at the following example:

```
struct list {
   int data;
   struct list *next;
};

struct point {
    int x;
    int y;
};

delete_node (struct list *l, int val) {
   while (l) {
      if (l->data == val)
         free (l);  // say l is 0x5000010
   }
}

add_cordinate (int x, int y) {
    struct point *p = malloc (sizeof(*p));   // say p is 0x5000010
    p->x = x;
    p->y = y;
}
```

In this example, the delete_node routine is trying to delete a node from a linked list. But as you can see this buggy code is simply freeing the node without updating the pointers in the list. Once an object is freed, malloc can give it to another allocation request. Let us say add_cordinate allocates an object which gets the same address as the freed linked list node in the delete_node routine. Due to this the linked list node and coordinates both pointing to the same memory location (0x5000010). The update to “y” field in coordinate eventually update the “next” field of list node. As a consequence, you may end up dereferencing a memory location that is actually an integer (the y coordinate). These bugs are hard to debug because the root cause of the problem might be at an unrelated piece of the code.

To prevent these bugs, one can modify malloc, never to reuse a virtual address. This solution makes sure that two allocated objects are not the alias of each other. The problem with this approach is that it requires a lot more memory than the existing malloc implementation. Another solution would be taking the physical page away as soon as an object is freed. If an application ever tries to access a freed page, it would cause a page fault. The OS can throw a dangling pointer exception on the page fault. A dangling pointer is a pointer which was allocated and then freed. So, it is not legal to access a dangling pointer. The problem with this solution is, we cannot take away the physical page as soon as the object is deleted because there might be other objects that are allocated on the same virtual page. For this scheme to work, we need to align all the allocations to page size and always return fresh virtual pages for each allocation.

Let us say the existing virtual pages are:
0x1000, 0x2000, 0x3000, 0x4000, 0x5000, 0x6000, 0x7000, 0x8000

We can support at most eight mallocs.

```
for (i = 0; i < 8; i++) {
    arr[i] = malloc (10);
}
```

Even though this code allocates only 80 bytes, if you use the above scheme you will end up allocating (4096 * 8) bytes. One sample output of the above code would be:

arr[0] == 0x1000
arr[1] == 0x2000
arr[2] == 0x3000
…
arr[7] == 0x8000

Notice that this scheme requires eight different virtual as well as physical pages. Can we do better than this? You might have noticed that even though we have to use different virtual pages for every allocation, the physical page can be shared. So, in the example above, all virtual pages in the range 0x1000 – 0x8000 can map to only one physical page. We just have to keep track of bytes which are allocated on each physical page.

arr[0] == 0x1000
arr[1] == 0x2008
arr[2] == 0x3010
…
arr[7] == 0x8048

If the workload does not have the fragmentation problem, then this scheme will consume a similar amount of physical memory that any other allocator will use. But, this scheme will use more virtual pages than needed. So, the TLB pressure might be a problem in this solution.