Paging without COW

Livepage: Consider the pages that are non-executable (PTE_X bit is 0) as livepage

In xv6, pages are created in **uvmalloc** and **uvmcopy** function. **Uvmalloc** is called by **exec** and **sbrk**, **uvmcopy** is called by **fork**.

In **uvmalloc**, when a new page is created, we need to add it to livepage list.
In **uvmcopy**, at first it copies a parent process's memory to "mem" and then maps the page to child process's va(virtual address). After that we need to add that page as live page if (PTE_X bit is off). Note: you don't need to think about the PTE_X bit anywhere else as it won't be in livepage list and no chance of getting swapped out to disk and cause any trouble.

When a page is going to be added to livepage list, first check if the list is full(crosses the MAXPHYPAGE), if not, we are good to add it to a live page and we're done. If the list is full, then first we need to swap a live page to disk, save the swap record, free the "pa" and then add the page of out interest to livepage list.

When a page is swapped to disk, make PTE_V = 0 and PTE_SWAP = 1

If a panic occurs due to PTE_V==0 checking, update the condition as (PTE_V == 0 && PTE_SWAP == 0)

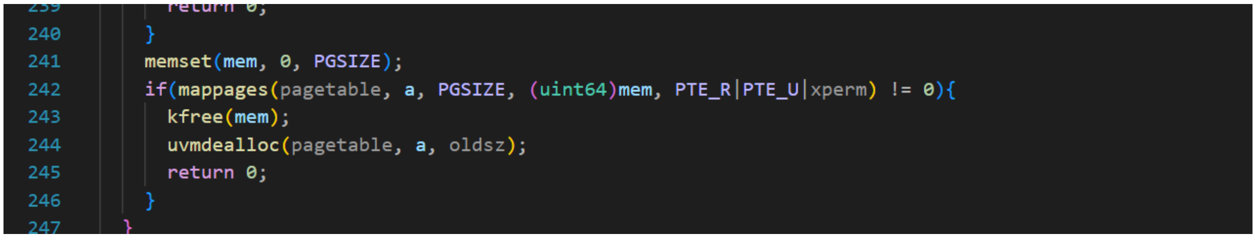
When a page fault occurs, check if it is a swap page (PTE_SWAP == 1), then find the swap record from the swaplist and then swap in the page from disk and then add it to livepage and make PTE_V = 1 and PTE_SWAP = 0. (if the livepage list is full, you need to swap out a page to disk as before).

Code flow:

Step 1:
Create page structure for livepage list
Page structure should have **pid** and **va** to uniquely identify a page.

Create a structure for swap record list
This structure should have swap struct(provided by sir), pid and va.

Step 2:
Modify uvmalloc. When a page is allocated, add this page to livepage list.

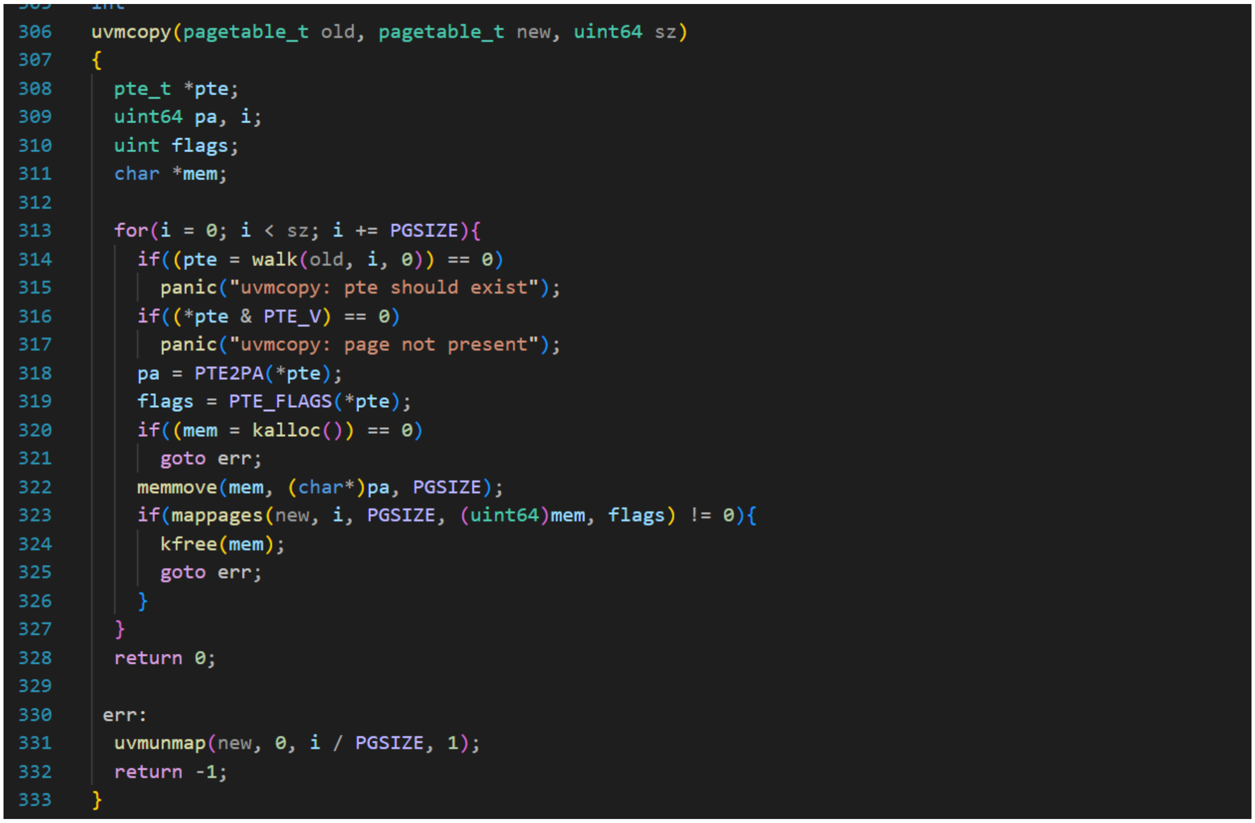


```
239        return 0;
240      }
241      memset(mem, 0, PGSIZE);
242      if(mappages(pagetable, a, PGSIZE, (uint64)mem, PTE_R|PTE_U|xperm) != 0){
243        kfree(mem);
244        uvmdealloc(pagetable, a, oldsz);
245        return 0;
246      }
247    }
```

You may do the modifications after line 246.

Modify uvmcopy:

First check if the page is in disk, or if it is a live page. If it is a livepage, then no problem. If it is a swapped page, then at first you need to swap in that page and then copy it for child process. (This is the easier way according to me)



```
305  int
306  uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
307  {
308    pte_t *pte;
309    uint64 pa, i;
310    uint flags;
311    char *mem;
312
313    for(i = 0; i < sz; i += PGSIZE){
314      if((pte = walk(old, i, 0)) == 0)
315        panic("uvmcopy: pte should exist");
316      if((*pte & PTE_V) == 0)
317        panic("uvmcopy: page not present");
318      pa = PTE2PA(*pte);
319      flags = PTE_FLAGS(*pte);
320      if((mem = kalloc()) == 0)
321        goto err;
322      memmove(mem, (char*)pa, PGSIZE);
323      if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
324        kfree(mem);
325        goto err;
326      }
327    }
328    return 0;
329
330   err:
331    uvmunmap(new, 0, i / PGSIZE, 1);
332    return -1;
333  }
```

Edit line 316 as (*pte & PTE_V) == 0 && (*pte & PTE_SWAP)==0

After line 317, check if it is a live page or swapped page. If it is a live page, then after **line 326**, add the new page to livepage

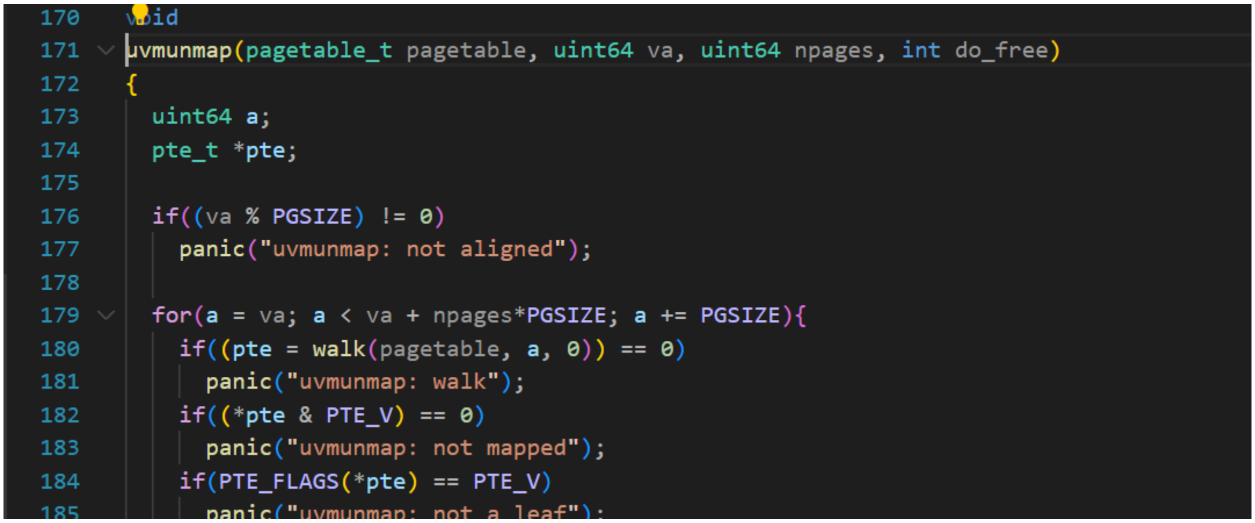
If you find that it is a swapped page **after line 317**, then first swap in the page from disk, then memmove(mem, (char*)swapped_pa, PGSIZE) and then rest of the work is as before.

Note: update necessary flag bits carefully.

Step 3:

When a process is killed, you need to remove it’s pages from livepages and swap records if it has any. (No need to swap in it’s pages from disk and then free them, I think. As when you free swap record, the block numbers are added to free block pool and they can be reused).

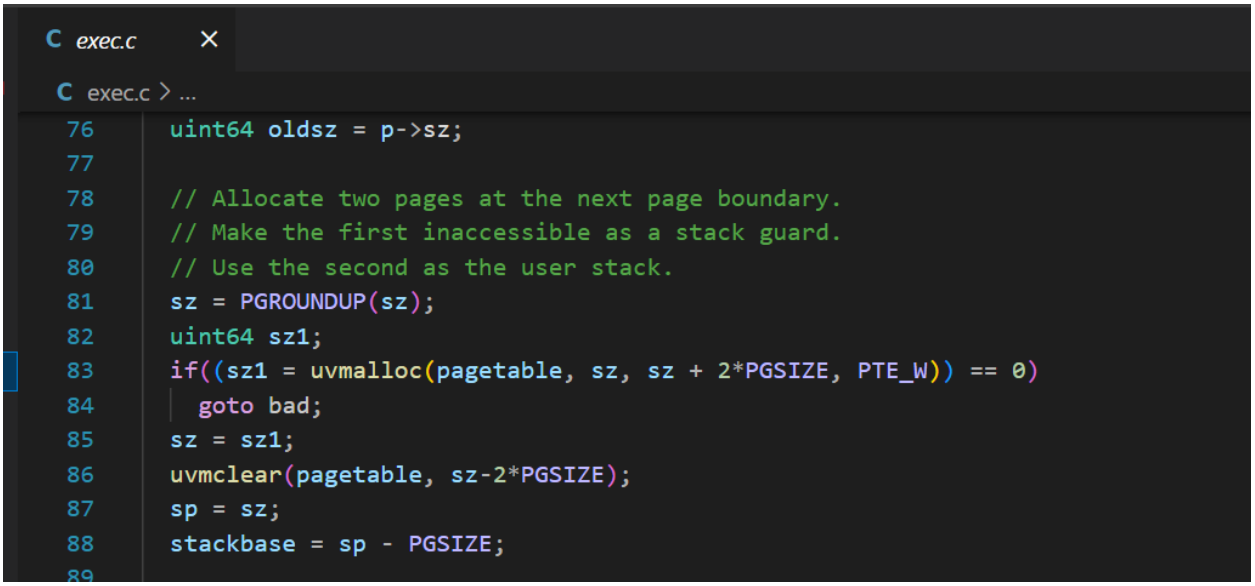
Whenever a page is freed, **uvmunmap** is called. So we can handle the works of removing livepage and swap records in uvmunmap.



```
170  void
171  uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
172  {
173    uint64 a;
174    pte_t *pte;
175
176    if((va % PGSIZE) != 0)
177      panic("uvmunmap: not aligned");
178
179    for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
180      if((pte = walk(pagetable, a, 0)) == 0)
181        panic("uvmunmap: walk");
182      if((*pte & PTE_V) == 0)
183        panic("uvmunmap: not mapped");
184      if(PTE_FLAGS(*pte) == PTE_V)
185        panic("uvmunmap: not a leaf");
```

Don’t forget to edit line 182 with (*pte & PTE_V) == 0 && (*pte & PTE_SWAP)==0

Step 4:



```
uint64 oldsz = p->sz;

// Allocate two pages at the next page boundary.
// Make the first inaccessible as a stack guard.
// Use the second as the user stack.
sz = PGROUNDUP(sz);
uint64 sz1;
if((sz1 = uvmalloc(pagetable, sz, sz + 2*PGSIZE, PTE_W)) == 0)
  goto bad;
sz = sz1;
uvmclear(pagetable, sz-2*PGSIZE);
sp = sz;
stackbase = sp - PGSIZE;
```

In exec, this 2 extra page is created for stack and stack guard. Do not add these as live page or swap them in disk. Keep a flag bit for them so that you can detect them in uvmalloc and avoid manipulating them.

Step 5:

Trap.c:
When a page fault occurs–
i) detect if it is a swap page
ii) allocate a physical memory: uint64 pa = (uint64) kalloc();
iii) find the swap record
iv) swap in the page from disk in pa of step ii
v) map the process's va with pa by calling mappages.
vi) add the page to livepage and you're done

Step 6:
To perform the steps explained so far, write necessary functions to operate the livepage list and swap records.

Step 7:
Write a user test code to test swap out, swap in
Say we have MAXPHYSIZE = 10
In user test code, allocate more than 10 pages using sbrk (say 15 pages)

Then access those allocated pages. Write something. Doing this will cause all the pages to swap in and out from disk (due to FIFO page replacement. Don’t ask how, do it and try to understand how it is happening)

Then deallocate the pages.

If this test code is passed, then you’re done with this offline (Which sir has told today in the class)

If the test code fails, then you’ll have a kernel trap or some panic. Debug, cry, debug :3