Project 3 Readme

Group members (same group as project 1 and 2)

- Molly Fawcett
- Shehzeen Syed
- Ethan Kocses

We ran xv6 through both WSL on Windows and directly on Ubuntu. xv6 was installed as detailed in project 0, except with a compiler downgrade.

Part 1: xv6 Page Allocation

Xv6 use of sbrk()

In xv6, memory allocation for a process is done using the sbrk(n) system call, which increases or decreases the process's memory size. In the original xv6, sys_sbrk() calls argint(0, &n) to get the requested size increase. It saves the old process size (addr = myproc()->sz). Then sbrk(n) calls growproc(n), which manages memory expansion by allocating or deallocating pages. Finally, it returns the old size (addr).

If n is positive, growproc(n) calls allocuvm() from vm.c. allocuvm allocates physical memory pages using kalloc(). It then maps these pages to the process's virtual address space by updating the page table using mappages(), ensuring that virtual memory points to the allocated physical memory.

If n is negative, growproc(n) calls deallocuvm(), which frees memory by unmapping pages and releasing them back to the system.

The sbrk(n) system call is crucial in xv6 because it manages dynamic memory allocation by allowing a process to expand its heap when needed. It enables programs to request memory at runtime rather than having all memory allocated at compile time. Standard libraries like malloc() depend on sbrk() to get memory from the kernel, which they then manage internally.

Sbrk() and page allocation in xv6

Sbrk() is called any time malloc() is called in xv6, meaning that when a user program requests virtual pages of memory, xv6 immediately also allocates those pages in physical memory. This means that any time a program requests a virtual page, a valid physical frame in memory should always already be entered into the page table for them.

What this also means for xv6 page allocation is that there shouldn't ever be a case in which a page fault happens in xv6 – any request for a virtual page should always have a physical page. We can see that there is also no handling for a page fault in xv6 in the trap.c file, meaning a page fault will kill the process. This makes sense, as a page fault would be completely unexpected and indicate some kind of error in xv6.

Why the system breaks when sbrk() is rewritten

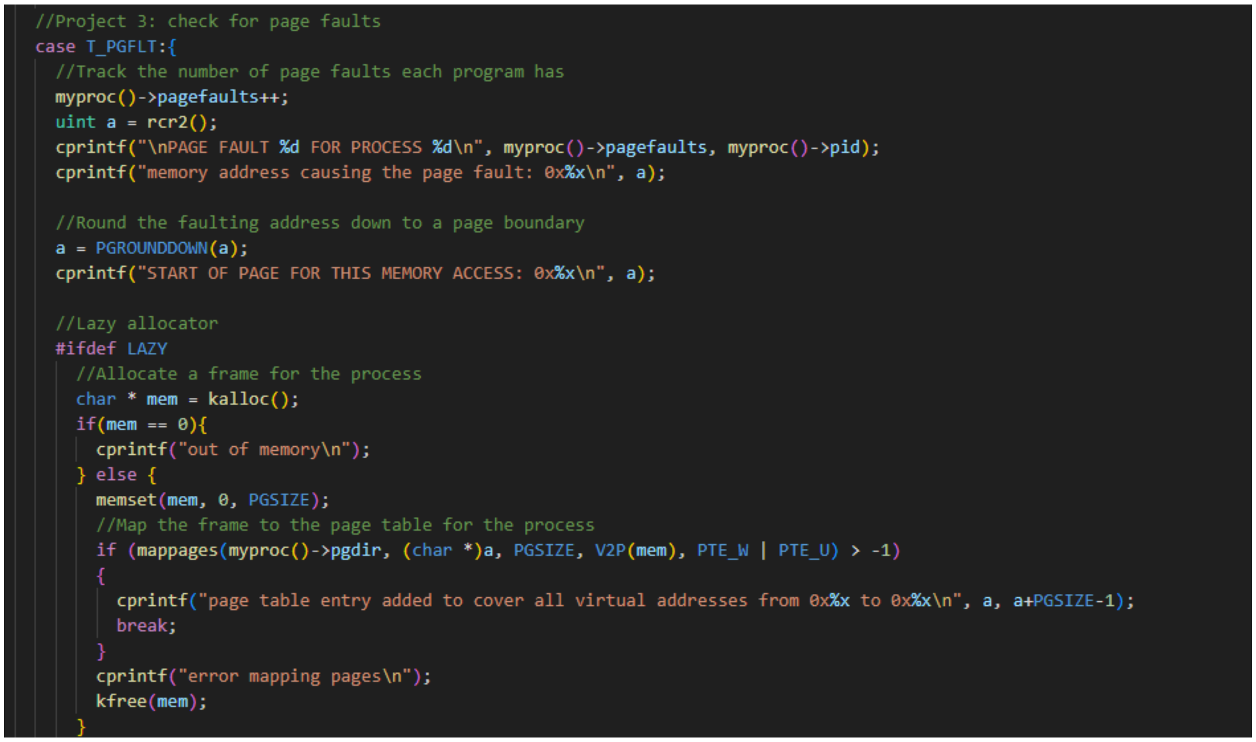
The system breaks when we modify sbrk(n) to remove growproc() because while sbrk(n) still increases the process's memory size (proc->sz), it no longer allocates physical memory or updates the page table. This creates a situation where a process believes it has access to more memory, but in reality, no actual memory is mapped to those addresses.

Implementing lazy page allocation means removing the part of malloc(), and in turn, sbrk(), in xv6 that allocates valid physical pages and putting them in the page table. This will cause page faults of course. In lazy allocation, we instead want to wait until a page fault happens, *and then* allocate those pages. This involves taking the logic sbrk() uses for actually allocating physical pages, and moving it to where a page fault happens.

Part 4: Lazy Page Allocation

Implementing Lazy Page Allocation

Adding the lazy page allocator only required adding a bit of new logic to the trap.c file. trap.c contains a main switch statement to catch certain fault types, but it does not natively have one for a page fault. So we added one:



```c
//Project 3: check for page faults
case T_PGFLT:{
  //Track the number of page faults each program has
  myproc()->pagefaults++;
  uint a = rcr2();
  cprintf("\nPAGE FAULT %d FOR PROCESS %d\n", myproc()->pagefaults, myproc()->pid);
  cprintf("memory address causing the page fault: 0x%x\n", a);

  //Round the faulting address down to a page boundary
  a = PGROUNDDOWN(a);
  cprintf("START OF PAGE FOR THIS MEMORY ACCESS: 0x%x\n", a);

  //Lazy allocator
  #ifdef LAZY
    //Allocate a frame for the process
    char * mem = kalloc();
    if(mem == 0){
      cprintf("out of memory\n");
    } else {
      memset(mem, 0, PGSIZE);
      //Map the frame to the page table for the process
      if (mappages(myproc()->pgdir, (char *)a, PGSIZE, V2P(mem), PTE_W | PTE_U) > -1)
      {
        cprintf("page table entry added to cover all virtual addresses from 0x%x to 0x%x\n", a, a+PGSIZE-1);
        break;
      }
      cprintf("error mapping pages\n");
      kfree(mem);
    }
```

The logic is very similar, and mostly copied from, the allocuvm() function in vm.c. allocuvm() is the function that requests new physical pages to be mapped to a process's virtual pages, using two functions.

Kalloc: allocates one 4096 byte page of physical memory for the kernel to use. This grabs any phyical page from the kernel's "free" page list.

Mappages: creates a page table entry to map a given virtual page to a physical page and puts it into the given page table.

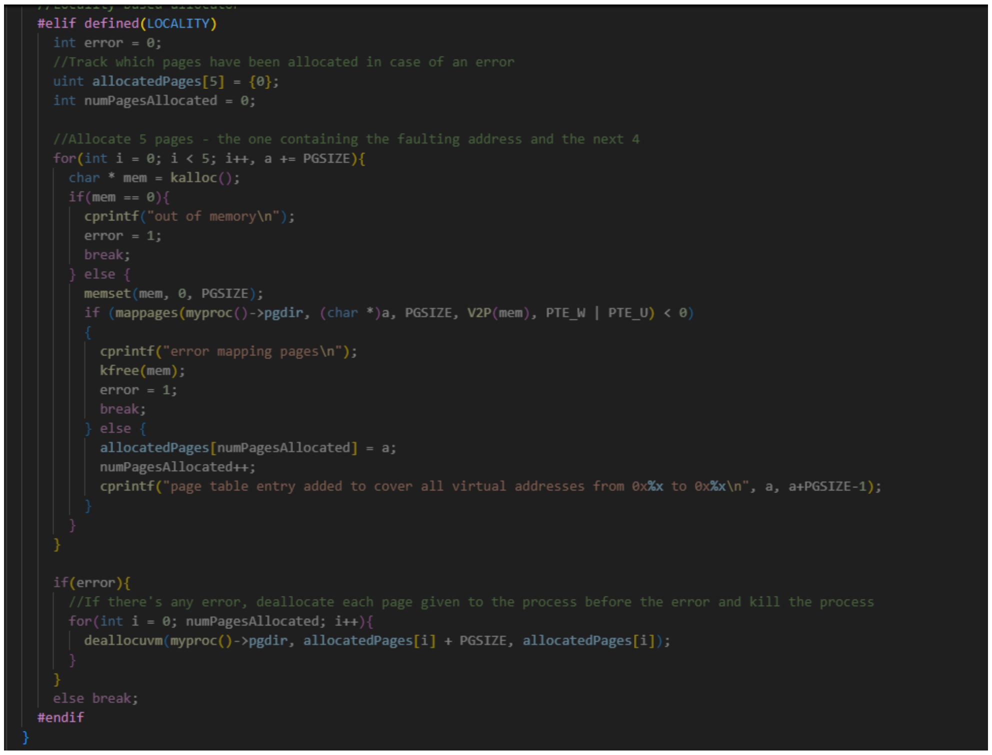
This logic allows us to do lazy allocation as follows:

1. Detect a page fault. The user program tried to reference a virtual memory address, but it wasnt mapped to any valid physical memory. **(line 116, in the switch statement)**
2. Round the faulting memory address down so we can find the starting bounds oof the virtual page. **(line 124, PGROUNDDOWN)**
3. Ask the kernel for a frame of physical memory to use. **(line 130, kalloc)**
4. Map the full page of virtual memory that our requested virtual memory address resides in, to a valid physical frame in our page table. **(line 136, mappages)**

From now on, any time we try to access a virtual memory address in the same virtual page, we will not get a page fault!!

Implementing Locality-Aware Page Allocation

Our lazy allocation made doing locality aware allocation extremely simple. All we had to add to the logic was a simple loop using the starting memory address for the virtual page we found that caused the fault.

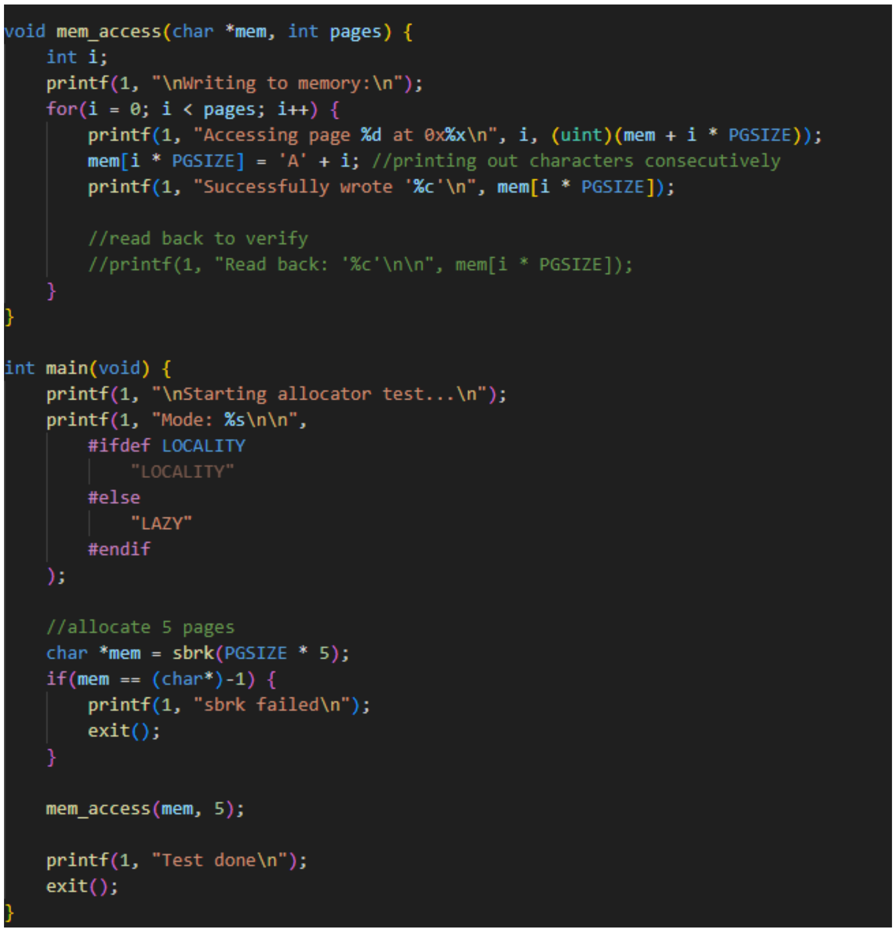


```
  #elif defined(LOCALITY)
    int error = 0;
    //Track which pages have been allocated in case of an error
    uint allocatedPages[5] = {0};
    int numPagesAllocated = 0;

    //Allocate 5 pages - the one containing the faulting address and the next 4
    for(int i = 0; i < 5; i++, a += PGSIZE){
      char * mem = kalloc();
      if(mem == 0){
        cprintf("out of memory\n");
        error = 1;
        break;
      } else {
        memset(mem, 0, PGSIZE);
        if (mappages(myproc()->pgdir, (char *)a, PGSIZE, V2P(mem), PTE_W | PTE_U) < 0)
        {
          cprintf("error mapping pages\n");
          kfree(mem);
          error = 1;
          break;
        } else {
          allocatedPages[numPagesAllocated] = a;
          numPagesAllocated++;
          cprintf("page table entry added to cover all virtual addresses from 0x%x to 0x%x\n", a, a+PGSIZE-1);
        }
      }
    }

    if(error){
      //If there's any error, deallocate each page given to the process before the error and kill the process
      for(int i = 0; numPagesAllocated; i++){
        deallocuvm(myproc()->pgdir, allocatedPages[i] + PGSIZE, allocatedPages[i]);
      }
    }
    else break;
  #endif
}
```

Starting at our new a address after running PGROUNDDOWN(a) **(line 124)**, iterate kalloc and mappages 4 more times, adding PGSIZE to the starting address each time. This allocates 5 pages, starting at address a, to the process, and enters them into the page table.

Demonstration of Differences

To demonstrate the differences between the two allocators, we wrote this test command to explicitly access memory locations in each of the first 5 virtual pages of a process.

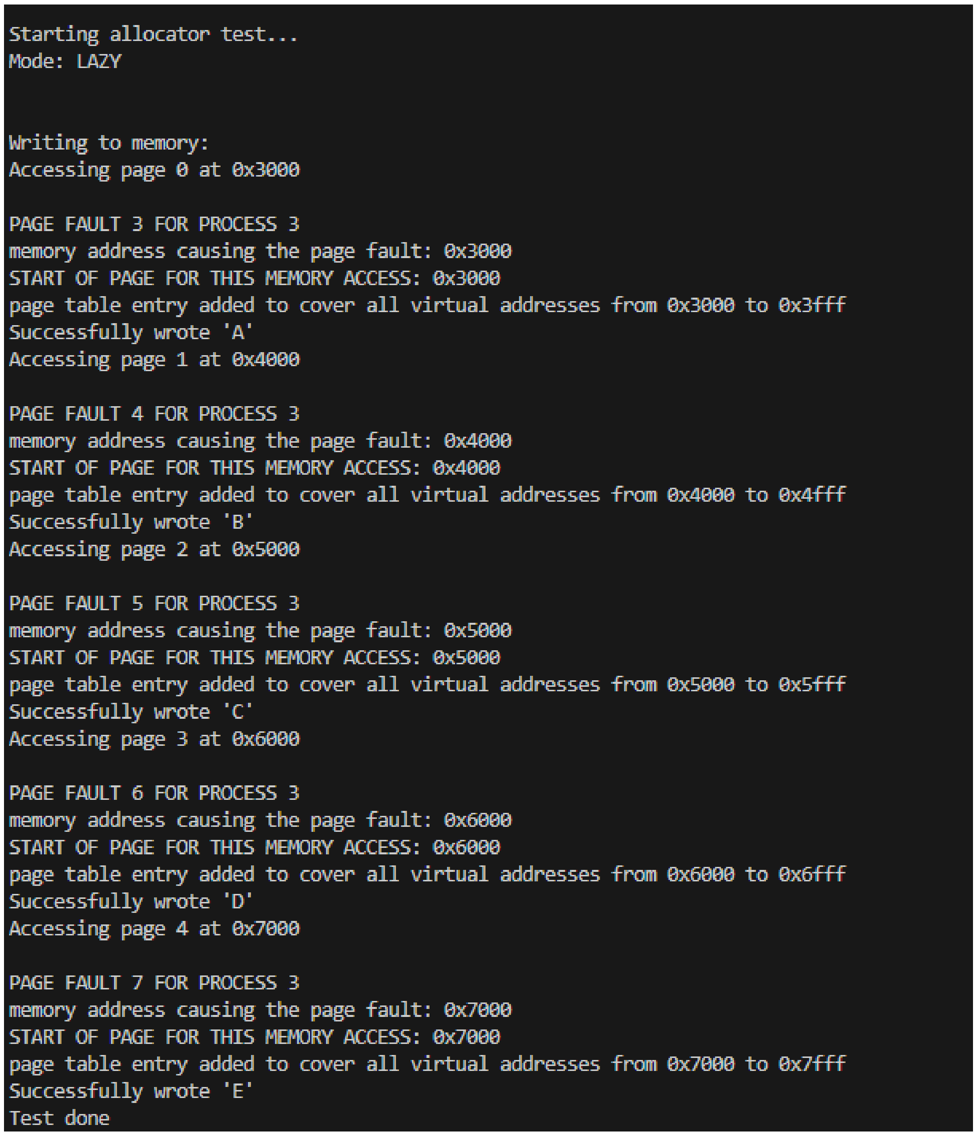


```c
void mem_access(char *mem, int pages) {
    int i;
    printf(1, "\nWriting to memory:\n");
    for(i = 0; i < pages; i++) {
        printf(1, "Accessing page %d at 0x%x\n", i, (uint)(mem + i * PGSIZE));
        mem[i * PGSIZE] = 'A' + i; //printing out characters consecutively
        printf(1, "Successfully wrote '%c'\n", mem[i * PGSIZE]);

        //read back to verify
        //printf(1, "Read back: '%c'\n\n", mem[i * PGSIZE]);
    }
}

int main(void) {
    printf(1, "\nStarting allocator test...\n");
    printf(1, "Mode: %s\n\n",
        #ifdef LOCALITY
            "LOCALITY"
        #else
            "LAZY"
        #endif
    );

    //allocate 5 pages
    char *mem = sbrk(PGSIZE * 5);
    if(mem == (char*)-1) {
        printf(1, "sbrk failed\n");
        exit();
    }

    mem_access(mem, 5);

    printf(1, "Test done\n");
    exit();
}
```

The process uses sbrk to create a character array that is 5 pages long, increasing the process size to 5 pages. Then, we call mem_access to test out accessing a memory address in each of those 5 pages. Our print statements that we added to our page fault logic will show how many of these accesses cause a page fault.

Here are the outputs of the two allocators:

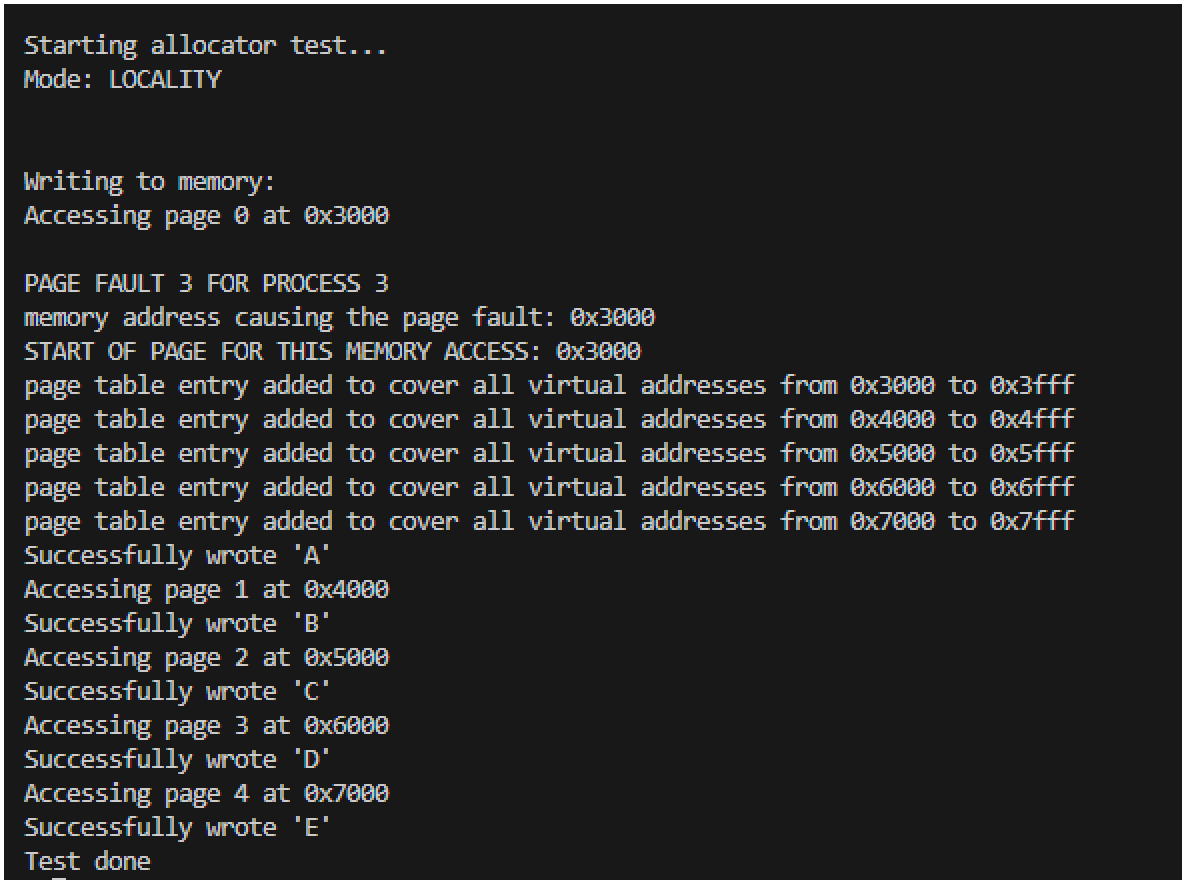
Lazy Allocator



```
Starting allocator test...
Mode: LAZY


Writing to memory:
Accessing page 0 at 0x3000

PAGE FAULT 3 FOR PROCESS 3
memory address causing the page fault: 0x3000
START OF PAGE FOR THIS MEMORY ACCESS: 0x3000
page table entry added to cover all virtual addresses from 0x3000 to 0x3fff
Successfully wrote 'A'
Accessing page 1 at 0x4000

PAGE FAULT 4 FOR PROCESS 3
memory address causing the page fault: 0x4000
START OF PAGE FOR THIS MEMORY ACCESS: 0x4000
page table entry added to cover all virtual addresses from 0x4000 to 0x4fff
Successfully wrote 'B'
Accessing page 2 at 0x5000

PAGE FAULT 5 FOR PROCESS 3
memory address causing the page fault: 0x5000
START OF PAGE FOR THIS MEMORY ACCESS: 0x5000
page table entry added to cover all virtual addresses from 0x5000 to 0x5fff
Successfully wrote 'C'
Accessing page 3 at 0x6000

PAGE FAULT 6 FOR PROCESS 3
memory address causing the page fault: 0x6000
START OF PAGE FOR THIS MEMORY ACCESS: 0x6000
page table entry added to cover all virtual addresses from 0x6000 to 0x6fff
Successfully wrote 'D'
Accessing page 4 at 0x7000

PAGE FAULT 7 FOR PROCESS 3
memory address causing the page fault: 0x7000
START OF PAGE FOR THIS MEMORY ACCESS: 0x7000
page table entry added to cover all virtual addresses from 0x7000 to 0x7fff
Successfully wrote 'E'
Test done
```

In running this program, you can observe that the basic lazy allocator page faults all 5 times that it accesses the memory addresses for the array, from 0x3000 to 0x7fff. Each time there is a page fault, we allocate a new physical page that maps to the virtual address we tried to access. We print out the memory addresses that each new allocated page will cover in the future, now that it's allocated. Because this program intentionally accesses a new page every time it does a memory lookup, we get a page fault every single time.

Locality Allocator



```
Starting allocator test...
Mode: LOCALITY


Writing to memory:
Accessing page 0 at 0x3000

PAGE FAULT 3 FOR PROCESS 3
memory address causing the page fault: 0x3000
START OF PAGE FOR THIS MEMORY ACCESS: 0x3000
page table entry added to cover all virtual addresses from 0x3000 to 0x3fff
page table entry added to cover all virtual addresses from 0x4000 to 0x4fff
page table entry added to cover all virtual addresses from 0x5000 to 0x5fff
page table entry added to cover all virtual addresses from 0x6000 to 0x6fff
page table entry added to cover all virtual addresses from 0x7000 to 0x7fff
Successfully wrote 'A'
Accessing page 1 at 0x4000
Successfully wrote 'B'
Accessing page 2 at 0x5000
Successfully wrote 'C'
Accessing page 3 at 0x6000
Successfully wrote 'D'
Accessing page 4 at 0x7000
Successfully wrote 'E'
Test done
```

With the locality aware allocator, we see the same print statements for page table entries being added for all memory addresses 0x3000 to 0x7fff. However, the locality aware allocator adds them all at the first page fault, which allows us to avoid those other 4 page fault interrupts. This demonstrates the benefits of a locality aware allocator as it can decrease the overhead from handling as many page faults.