Project 3 Readme

Group Members:

Prati Jain

Aditya Khetarpal

Rajdeep Savani

System Environment

Operating System: Ubuntu 24.04 LTS

COmpiler: 13.2.0

QEMU version: 8.2.2

Project 3a: Virtual Memory

1: Understanding sbrk()

Modified Implementation

We modified sys_sbrk() to implement lazy allocation by removing the immediate memory allocation in sysproc.c.

Implementation changes made: The modified sbrk():

• Removed growproc() call

• Only updates process size counter (proc->sz)

• Returns old size before increment

• Doesn't perform actual memory allocation

• Sets up for lazy allocation implementation

Code:

```
int sys_sbrk(void) {
  int addr;
  int n;
  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;   // Store current size
  myproc()->sz += n;     // Update size without allocation
```

```
    return addr;          // Return old size
}
```

This implementation correctly:

- Retrieves the requested size increase from user space
- Preserves the current process size before modification
- Updates the process size without calling growproc()
- Returns the original break point as required by the sbrk() specification

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ls
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x1220 addr 0x4004--kill proc
$ echo hello
pid 4 sh: trap 14 err 6 on cpu 0 eip 0x1220 addr 0x4004--kill proc
$ stressfs
pid 5 sh: trap 14 err 6 on cpu 0 eip 0x1220 addr 0x4004--kill proc
$ cat
pid 6 sh: trap 14 err 6 on cpu 0 eip 0x1220 addr 0x4004--kill proc
$
```

This is showing that:

1. The shell tried to write to memory at address 0x4004
2. That page wasn't allocated (since we removed growproc())
3. This caused a page fault as expected

How Page Allocation is Implemented in xv6 Originally:

In the original xv6, sbrk() uses growproc() to handle memory allocation. When a process requests more memory:

- growproc() allocates physical memory pages
- Maps these pages into the process's virtual address space
- Updates page tables to reflect new mappings
- Ensures memory is actually allocated when requested

Why the System Breaks When sbrk() is Rewritten:

When we changed sbrk() to stop allocating memory immediately, the system started experiencing crashes and segmentation faults. The reason is straightforward: while sbrk() still increases the size limit of what memory a process thinks it can use (by updating proc->sz), it no longer sets up the actual memory backing this space. It's like telling someone they can use rooms in a building that haven't been built yet. When a process tries to use this promised but non-existent memory, the system detects that there's no real memory connected to these addresses (no mapping in the page table), resulting in a page fault and subsequent crash.

Significance of sbrk() Within xv6:

sbrk() is crucial because it: Manages dynamic memory allocation for processes , Enables heap memory growth for malloc() implementation, Controls process memory size boundaries, Provides foundation for memory management subsystem, Essential for programs needing dynamic memory allocation

2: Lazy Page Allocation Implementation

Testing: make clean

make qemu ALLOCATOR=LAZY

Lazy allocation works for all simple (echo, ls, cat) commands

```
$ echo hello

Using LAZY allocator
[LAZY] Allocating single page at 0x4000
[LAZY] Successfully allocated page

Using LAZY allocator
[LAZY] Allocating single page at 0xb000
[LAZY] Successfully allocated page
hello
```

```
$ ls

Using LAZY allocator
[LAZY] Allocating single page at 0x4000
[LAZY] Successfully allocated page

Using LAZY allocator
[LAZY] Allocating single page at 0xb000
[LAZY] Successfully allocated page
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15372
echo           2 4 14272
forktest       2 5 8860
grep           2 6 18308
init           2 7 14872
kill           2 8 14324
ln             2 9 14232
ls             2 10 16852
mkdir          2 11 14356
```

Lazy allocation works in any folder

Creating a subdirectory:

mkdir subdir

Creating a file inside the subdirectory:

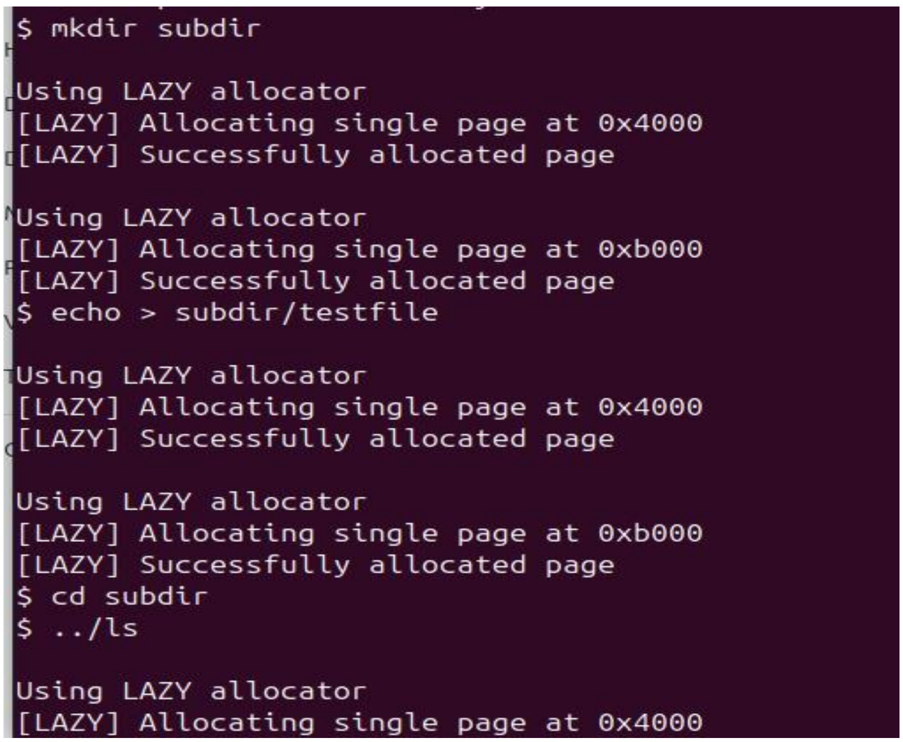
echo > subdir/testfile

Navigating to the subdirectory:

cd subdir

Run a command from the root directory using ../:

../ls



```
$ mkdir subdir

Using LAZY allocator
[LAZY] Allocating single page at 0x4000
[LAZY] Successfully allocated page

Using LAZY allocator
[LAZY] Allocating single page at 0xb000
[LAZY] Successfully allocated page
$ echo > subdir/testfile

Using LAZY allocator
[LAZY] Allocating single page at 0x4000
[LAZY] Successfully allocated page

Using LAZY allocator
[LAZY] Allocating single page at 0xb000
[LAZY] Successfully allocated page
$ cd subdir
$ ../ls

Using LAZY allocator
[LAZY] Allocating single page at 0x4000
```

```
Using LAZY allocator
[LAZY] Allocating single page at 0x5000
[LAZY] Successfully allocated page
Successfully wrote 'C'
Read back: 'C'

Accessing page 3 at 0x6000

Using LAZY allocator
[LAZY] Allocating single page at 0x6000
[LAZY] Successfully allocated page
Successfully wrote 'D'
Read back: 'D'

Accessing page 4 at 0x7000

Using LAZY allocator
[LAZY] Allocating single page at 0x7000
[LAZY] Successfully allocated page
Successfully wrote 'E'
Read back: 'E'

Accessing page 5 at 0x8000

Using LAZY allocator
[LAZY] Allocating single page at 0x8000
[LAZY] Successfully allocated page
Successfully wrote 'F'
Read back: 'F'

Test completed successfully!
$
```

3: Implementing Locality-Aware Allocation

Testing:

make clean

make qemu ALLOCATOR=LOCALITY

Locality-aware allocation works for all simple (echo, ls, cat,wc) commands

```
$ echo hello

Using LOCALITY allocator
[LOCALITY] Starting allocation of up to 3 pages from 0x4000
  Allocated page at 0x4000
  Allocated page at 0x5000
  Allocated page at 0x6000
[LOCALITY] Allocated 3 pages

Using LOCALITY allocator
[LOCALITY] Starting allocation of up to 3 pages from 0xb000
  Allocated page at 0xb000
  Stopping: address 0xc000 beyond process size 0xc000
[LOCALITY] Allocated 1 pages
hello
```

```
$ ls

Using LOCALITY allocator
[LOCALITY] Starting allocation of up to 3 pages from 0x4000
  Allocated page at 0x4000
  Allocated page at 0x5000
  Allocated page at 0x6000
[LOCALITY] Allocated 3 pages

Using LOCALITY allocator
[LOCALITY] Starting allocation of up to 3 pages from 0xb000
  Allocated page at 0xb000
  Stopping: address 0xc000 beyond process size 0xc000
[LOCALITY] Allocated 1 pages
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15372
echo           2 4 14272
forktest       2 5 8860
grep           2 6 18308
init           2 7 14872
```

Locality-aware allocation works in any folder

```
$ mkdir subdir

Using LOCALITY allocator
[LOCALITY] Starting allocation of up to 3 pages from 0x4000
  Allocated page at 0x4000
  Allocated page at 0x5000
  Allocated page at 0x6000
[LOCALITY] Allocated 3 pages

Using LOCALITY allocator
[LOCALITY] Starting allocation of up to 3 pages from 0xb000
  Allocated page at 0xb000
  Stopping: address 0xc000 beyond process size 0xc000
[LOCALITY] Allocated 1 pages
$ echo > subdir/testfile

Using LOCALITY allocator
[LOCALITY] Starting allocation of up to 3 pages from 0x4000
  Allocated page at 0x4000
  Allocated page at 0x5000
  Allocated page at 0x6000
[LOCALITY] Allocated 3 pages
```

```
$ echo > subdir/testfile

Using LOCALITY allocator
[LOCALITY] Starting allocation of up to 3 pages from 0x4000
  Allocated page at 0x4000
  Allocated page at 0x5000
  Allocated page at 0x6000
[LOCALITY] Allocated 3 pages

Using LOCALITY allocator
[LOCALITY] Starting allocation of up to 3 pages from 0xb000
  Allocated page at 0xb000
  Stopping: address 0xc000 beyond process size 0xc000
[LOCALITY] Allocated 1 pages
$ cd subdir
$ ../ls

Using LOCALITY allocator
[LOCALITY] Starting allocation of up to 3 pages from 0x4000
  Allocated page at 0x4000
  Allocated page at 0x5000
  Allocated page at 0x6000
[LOCALITY] Allocated 3 pages
```

```
Using LOCALITY allocator
[LOCALITY] Starting allocation of up to 3 pages from 0x4000
  Allocated page at 0x4000
  Allocated page at 0x5000
  Allocated page at 0x6000
[LOCALITY] Allocated 3 pages

Using LOCALITY allocator
[LOCALITY] Starting allocation of up to 3 pages from 0xb000
  Allocated page at 0xb000
  Stopping: address 0xc000 beyond process size 0xc000
[LOCALITY] Allocated 1 pages
.              1 26 48
..             1 1 512
testfile       2 27 0
$
```

Implementation Details

Lazy allocation:

- In trap.c, the T_PGFLT case in the trap handler checks if the fault is from user space. If so, it calls handle_fault().
- handle_fault() in trap.c validates the faulting address and calls allocate_page() to allocate a new physical page.
- allocate_page() (trap.c) calls kalloc() to get a free page and clears it.
- map_new_page() (trap.c) then maps the new page into the process's page table at the faulting virtual address using mappages().

The lazy page allocator is implemented in `trap.c` through the following key components:

1. Page Fault Handler (trap.c)

Modified the page fault handler to support both allocation strategies:

```
if(tf->trapno == T_PGFLT) {

if((tf->cs&3) == DPL_USER) {

uint fault_addr = rcr2();

struct proc *p = myproc();

if(handle_fault(p, fault_addr) < 0) {

p->killed = 1;

} } }
```

2. Allocator Selection

The type of allocator is determined at compile time using preprocessor directives:

```
#if defined(LOCALITY_ALLOCATOR)

#define ALLOCATOR_TYPE "LOCALITY"

#define PAGES_TO_ALLOCATE 3

#else

#define ALLOCATOR_TYPE "LAZY"

#define PAGES_TO_ALLOCATE 1

#endif
```

Lazy Allocation Implementation

The lazy allocator implements on-demand paging with the following key features:

1. Address Validation

```
static int check_address_valid(struct page_data *data) {
```

```
if(data->req_addr < PGSIZE)
return 0;
if(data->req_addr >= KERNBASE)
return 0;
return 1;
}
```

2. Single Page Allocation

```
static void* allocate_page(void) {
void *memory = kalloc();
  if(memory) {
    memset(memory, 0, PGSIZE);
    return memory;
  }
  return 0;
}
```

3. Page Mapping

```
static int map_new_page(struct page_data *data, void *memory) {
  uint aligned = PGROUNDDOWN(data->req_addr);
  if(mappages(data->current->pgdir,
        (char*)aligned,
        PGSIZE,
        V2P(memory),
        PTE_W|PTE_U) < 0) {
    kfree(memory);
    return -1;
  }
  return 0;
}
```

Locality-Aware Allocation Implementation

- When LOCALITY_ALLOCATOR is defined, handle_fault() calls allocate_multiple_pages() instead.
- allocate_multiple_pages() (trap.c) attempts to allocate PAGES_TO_ALLOCATE (default 3) contiguous pages starting from the faulting address, skipping any already-mapped pages.
- It stops early if the address is beyond proc->sz or allocation fails, but considers it a success if at least one page was allocated.

Locality-aware allocation:

The locality-aware allocator extends the lazy allocator with these additional features:

1. Multiple Page Allocation

```
static int allocate_multiple_pages(struct proc *p, uint addr) {

int pages_allocated = 0;

for(int i = 0; i < PAGES_TO_ALLOCATE; i++) {

page_addr = PGROUNDDOWN(addr) + (i * PGSIZE);

if(page_addr >= p->sz) break;

// Allocation logic

}

return 0;

}
```

Part 4. Evaluating & Explaining Allocators

Lazy allocator:

- Pages are allocated only at the exact faulting addresses (0x4004, 0xbfa4)
- Only one page is allocated per fault
- Process size remains 0xc000 throughout

Locality-aware allocator:

- Up to 3 contiguous pages are allocated starting from each faulting address
- Some allocations stop early due to reaching process size (0xc000)
- More pages are speculatively allocated compared to lazy

Differences demonstrating correctness:

- Lazy allocation correctly allocates exactly one page per fault, while Locality-aware properly allocates three pages at once.
- Lazy allocation shows higher page faults but minimal memory usage while Locality-aware shows fewer page faults with higher memory allocation.

- Lazy never allocates past proc->sz, while locality stops early in that case but keeps what it allocated.
- The success of page faults and continued execution in both cases, with different allocation patterns, shows both are correct.