Project3a-README

Group Members:
Asha Tummuru(U38611026)
Vani Pailla(U95251377)

PART 1: Understanding `sbrk()`

Changed Implementation of `sbrk()`

The `sbrk()` system call was modified to increment the process size (`proc->sz`) by `n` bytes without allocating any physical memory. The call to `growproc()` responsible for immediate memory allocation was removed. The modified `sys_sbrk()` function now only updates `proc->sz` and returns the old size.

Code Snippet for Modified `sys_sbrk()`:

```
int
sys_sbrk(void)
{
    int addr;
    int n;

    if (argint(0, &n) < 0)
        return -1;
    addr = myproc()->sz;
    myproc()->sz += n; // Increment size without calling growproc()
    return addr;
}
```

How Page Allocation is Implemented in xv6 Originally

In the original xv6 implementation:

1. When a process calls `sbrk(n)`, it requests to grow its memory by `n` bytes.
2. The kernel adjusts the process size (`proc->sz`) and then calls `growproc(n)` to allocate physical memory for the newly requested space.
3. The `growproc()` function updates the page table to map the new virtual address range to physical pages.
4. This ensures that the memory is ready to use immediately after the `sbrk()` call.

This immediate allocation simplifies memory management for user programs, ensuring that accessing the allocated memory does not cause errors or page faults.

Why the System Breaks When `sbrk()` is Modified

When `sbrk()` is rewritten to exclude physical memory allocation:

1. Processes assume the memory is backed by physical pages after the `sbrk()` call, but in the modified version, it is not.
2. Accessing the "allocated" memory results in a **page fault** because there are no physical pages mapped to the virtual addresses in the process's page table.
3. Since the default xv6 page fault handler does not allocate memory on fault, the process is terminated with an error.

Example Error Output:

```
$ pid 3 sh: trap 14 err 6 on cpu 0 eip 0x1250 addr 0x4004--kill proc
```

This error indicates that the process tried to access memory at `0x4004`, but no physical page was allocated, leading to a segmentation fault.

Significance of `sbrk()` in xv6

The `sbrk()` system call plays a critical role in dynamic memory allocation for user processes:

1. It allows processes to request additional memory from the kernel during runtime, commonly used for heap management by dynamic memory allocators like `malloc()`.
2. By allocating physical memory immediately, the original implementation ensures that user programs can safely use the memory without additional kernel intervention.
3. A correctly implemented `sbrk()` reduces the complexity of user-space memory management by providing a seamless abstraction of memory allocation.

When `sbrk()` is rewritten to assume lazy allocation:

- The kernel defers physical memory allocation until the memory is accessed, requiring robust page fault handling.
- This highlights the dependency of user-space programs on the kernel's memory allocation mechanisms and the importance of properly handling deferred allocations.

Conclusion

In this part, the implementation of `sbrk()` was modified to assume lazy allocation, breaking the immediate allocation behavior of xv6. This led to unhandled page faults, demonstrating the critical role of `sbrk()` in xv6's memory management. Screenshots of the results are included in the `screenshots/part1/` folder for reference.

PART 2: Implementing Lazy Page Allocation

Implementation Steps

1. **sysproc.c**:
   - Modified `sys_sbrk()` to only adjust the process size (`proc->sz`) without allocating physical memory. This causes page faults when the process tries to access the memory.
2. **trap.c**:
   - Added logic in the page fault handler (`T_PGFLT` case) to dynamically allocate a page of physical memory when a fault occurs.
3. **Makefile**: Flags are defined for both ALLOCATOR = LAZY or LOCALITY

Code Snippet for Lazy Page Allocation:

```
case T_PGFLT: {
    uint faulting_address = rcr2(); // Get the faulting address
    struct proc *curproc = myproc();

    if (faulting_address >= KERNBASE || faulting_address >= curproc->sz) {
        cprintf("pid %d %s: page fault at 0x%x (error code %d)\n", curproc->pid,
curproc->name, faulting_address, tf->err);
        curproc->killed = 1;
        break;
    }

    int num_pages = 1; // Default to lazy allocation
    #ifdef LOCALITY
    num_pages = 3; // locality-aware allocation if defined
    #endif

    for (int i = 0; i < num_pages; ++i) {
        uint current_address = PGROUNDDOWN(faulting_address) + (i * PGSIZE);
        if (current_address >= curproc->sz) {
            break; // Stop if beyond process size
        }

        char *mem = kalloc();
        if (!mem) {
            cprintf("pid %d %s: out of memory\n", curproc->pid, curproc->name);
            curproc->killed = 1;
            break;
        }
        memset(mem, 0, PGSIZE);

        if (mappages(curproc->pgdir, (void*)current_address, PGSIZE, V2P(mem),
PTE_W|PTE_U) < 0) {
            kfree(mem);
            curproc->killed = 1;
            break;
        }
    }
    #ifdef ALLOCATOR_LOCALITY
```

```
      cprintf("[LOCALITY ALLOCATION] Allocated %d page(s) for faulting_address:
0x%x\n", num_pages, faulting_address);
    #else
      cprintf("[LAZY ALLOCATION] Allocated %d page(s) for faulting_address:
0x%x\n", num_pages, faulting_address);
    #endif

    if(curproc->killed) break;
      break; // Successfully handled the page fault
  }
```

1. Helper Functions:
    - `walkpgdir()`: Ensures the page table entry for the faulting address is created if it does not exist.
    - `mappages()`: Maps the newly allocated physical memory to the virtual address.
    - These functions can be found in vm.c as well.

Testing

Test Commands

1. **Basic Commands**:
    - `echo hello`
    - `ls`, `cat`, `wc`, and `mkdir`
    - All these commands executed successfully, indicating correct lazy allocation behavior.
2. **Custom Test Program**:
    - A custom user program (`allocation_test`) was created to allocate a large array and sequentially access it, triggering page faults.

Custom Test Code Snippet (from `allocation_test.c`):

```
#define ARRAY_SIZE (1024*10) // Enough to cause page faults -> size of 10 pages
int *array = malloc(ARRAY_SIZE * sizeof(int));
for (int i = 0; i < ARRAY_SIZE; i++) {
    array[i] = i; // Writing to memory to trigger page faults
}
```

1. Results:
    - Each page fault produced the following message:

      ```
      [LAZY ALLOCATION] Allocated 1 page(s) for faulting_address: 0x...
      ```

    - All commands and the test program completed without errors.

Analysis

1. **Lazy Allocation Works in One Scenario**:
   - The `allocation_test` program successfully triggered multiple page faults, each handled by the lazy allocation logic.
   - Memory was allocated dynamically only when accessed, confirming correctness.
2. **Lazy Allocation Works for All Basic Commands**:
   - Commands like `echo`, `ls`, and `mkdir` executed without errors, demonstrating that the lazy allocation implementation handles typical user commands seamlessly.
3. **Lazy Allocation Works in Any Folder**:
   - Directory-related commands, such as `mkdir subdir`, were tested and executed successfully using both relative and absolute paths.

Key Observations

1. **Efficiency**:
   - Memory is only allocated when accessed, optimizing memory usage.
2. **Correctness**:
   - The system handled page faults as expected, ensuring no errors or crashes occurred.
3. **Scalability**:
   - The implementation is sufficient for handling basic operations and more complex programs requiring dynamic memory allocation.

Conclusion

Lazy page allocation was successfully implemented and validated through testing. The system dynamically allocates memory when accessed, ensuring efficient memory usage while maintaining system stability. This implementation prepares the foundation for advanced memory allocation strategies, such as locality-aware allocation.

Screenshots of the results are included in the `screenshots/part2/` folder for reference.

PART 3: Implementing Locality-Aware Allocation

Implementation Steps

1. trap.c:
   - Modified the page fault handler (T_PGFLT case) to allocate

     three pages:
     - One for the faulting address.
     - Two additional pages for subsequent memory accesses.
   - This anticipates future memory access patterns and reduces the number of page faults for workloads with strong locality.

2. Makefile:
    - Modified the `Makefile` to enable switching between `LAZY` and `LOCALITY` allocation strategies at compile time.
    - Command to build with locality-aware allocation:

```
make clean && make qemu-nox ALLOCATOR=LOCALITY
```

Testing and Results

Test Commands

1. Basic Commands:
    - Tested commands like `echo`, `ls`, `cat`, `wc`, and `mkdir`. All executed successfully, confirming that locality-aware allocation functions correctly.
2. Custom Test Program (`allocation_test`):
    - Sequentially accessed a large array to simulate workloads with spatial locality.

Custom Test Code Snippet:

```
#define ARRAY_SIZE (1024*10) // Large enough to cause multiple faults
int *array = malloc(ARRAY_SIZE * sizeof(int));
for (int i = 0; i < ARRAY_SIZE; i++) {
    array[i] = i; // Trigger faults during sequential access
}
```

Results

Locality-Aware Allocation Output:

Console output shows that multiple pages are allocated per fault:

```
[LOCALITY ALLOCATION] Allocated 3 page(s) for faulting_address: 0x6000
[LOCALITY ALLOCATION] Allocated 3 page(s) for faulting_address: 0x9000
[LOCALITY ALLOCATION] Allocated 3 page(s) for faulting_address: 0xc000
```

Comparison with Lazy Allocation:

1. **Lazy Allocation**:
    - Allocates one page per fault, causing a higher number of page faults for sequential access patterns.
    - Example output:

```
[LAZY ALLOCATION] Allocated 1 page(s) for faulting_address: 0x4000
[LAZY ALLOCATION] Allocated 1 page(s) for faulting_address: 0x5000
[LAZY ALLOCATION] Allocated 1 page(s) for faulting_address: 0x6000
...
[LAZY ALLOCATION] Allocated 1 page(s) for faulting_address: 0xc000
[LAZY ALLOCATION] Allocated 1 page(s) for faulting_address: 0xd000
```

2. **Locality-Aware Allocation**:
    - Allocates three pages per fault, reducing the total number of page faults for sequential memory accesses.
    - Example output:

```
[LOCALITY ALLOCATION] Allocated 3 page(s) for faulting_address: 0x6000
[LOCALITY ALLOCATION] Allocated 3 page(s) for faulting_address: 0x9000
[LOCALITY ALLOCATION] Allocated 3 page(s) for faulting_address: 0xc000
```

Key Observations

1. Performance:
    - Locality-aware allocation significantly reduces the overhead of page fault handling for workloads with strong locality, such as sequential access patterns.
2. Efficiency:
    - While more memory is allocated upfront, this strategy minimizes interruptions caused by frequent faults.
3. Correctness:
    - Commands and custom programs execute successfully, confirming the implementation is robust.

Conclusion

Locality-aware allocation was successfully implemented and validated. By preemptively allocating multiple pages on a single fault, the strategy reduces page fault overhead for sequential workloads. This demonstrates the trade-off between memory usage and fault handling efficiency.

Screenshots of the results are included in the `screenshots/part3/` folder for reference.

PART 4: Evaluating and Explaining Allocators

Evaluation

A. Print Statements and Screenshots

Print statements were added in the page fault handler (`trap.c`) to track allocations:

1. **Lazy Allocation**:
    - Logs the allocation of a single page per fault:

```
[LAZY ALLOCATION] Allocated 1 page(s) for faulting_address: 0x%x
```

2. **Locality-Aware Allocation**:
    - Logs the allocation of three pages per fault:

```
[LOCALITY ALLOCATION] Allocated 3 page(s) for faulting_address: 0x%x
```

Screenshots:

- Screenshots of the program outputs for both allocators are included in the

  ```
  screenshots/
  ```

  folder:
  - Lazy allocation: `screenshots/part2/`
  - Locality-aware allocation: `screenshots/part3/`

B. Implementation Explanation

1. **Lazy Allocation**:
   - Implementation:
     - When a page fault occurs, the faulting address is aligned to the nearest page boundary using `PGROUNDDOWN()`.
     - One physical page is allocated using `kalloc()` and mapped to the faulting virtual address using `mappages()`.
   - Behavior:
     - Allocates memory only when accessed, optimizing memory usage.
     - Causes frequent page faults for workloads with sequential or clustered memory access.
2. **Locality-Aware Allocation**:
   - Implementation:
     - Upon a page fault, three physical pages are allocated:
       - One for the faulting address.
       - Two additional pages for subsequent memory accesses.
     - These pages are preemptively mapped to reduce future faults.
   - Behavior:
     - Anticipates future memory access patterns and minimizes the number of page faults.
     - Trades off increased memory usage for reduced fault handling overhead.

C. Analyzing the Differences

1. **Page Fault Handling**:
   - Lazy Allocation:
     - Handles each page fault independently, leading to one allocation per fault.
   - Locality-Aware Allocation:
     - Bundles multiple allocations into a single fault, reducing overall fault count.

2. **Performance Impact**:
    - Lazy Allocation:
        - Higher page fault rate, especially for sequential memory access, resulting in more interruptions.
    - Locality-Aware Allocation:
        - Reduces fault handling overhead for workloads with spatial locality, improving performance.
3. **Memory Usage**:
    - Lazy Allocation:
        - Allocates only the memory needed, optimizing memory usage.
    - Locality-Aware Allocation:
        - Allocates additional memory preemptively, which may result in wasted memory if unused.

Custom User Program Test

A custom test program (`allocation_test`) was used to demonstrate the behavior of both allocators:

- **Lazy Allocation**:
    - Faults occur for each page accessed, as shown in the output:

```
[LAZY ALLOCATION] Allocated 1 page(s) for faulting_address: 0x4000
[LAZY ALLOCATION] Allocated 1 page(s) for faulting_address: 0x5000
[LAZY ALLOCATION] Allocated 1 page(s) for faulting_address: 0x6000
...
[LAZY ALLOCATION] Allocated 1 page(s) for faulting_address: 0xc000
[LAZY ALLOCATION] Allocated 1 page(s) for faulting_address: 0xd000
```

- **Locality-Aware Allocation**:
    - Fewer faults occur because multiple pages are allocated per fault:

```
[LOCALITY ALLOCATION] Allocated 3 page(s) for faulting_address: 0x6000
[LOCALITY ALLOCATION] Allocated 3 page(s) for faulting_address: 0x9000
[LOCALITY ALLOCATION] Allocated 3 page(s) for faulting_address: 0xc000
```

Explanation of Results:

- Lazy allocation is efficient for workloads with sparse access patterns but incurs higher overhead for sequential access.
- Locality-aware allocation reduces faults for sequential workloads, improving performance at the cost of increased memory usage.

Conclusion

The evaluations confirm that both allocators are correctly implemented:

1. Lazy Allocation:
    - Optimizes memory usage by allocating on demand.
    - Suitable for workloads with sparse memory access.
2. Locality-Aware Allocation:
    - Improves performance for workloads with spatial locality by reducing fault handling overhead.
    - Best suited for sequential or clustered access patterns.

Screenshots and test logs are included in the `screenshots/` folder to demonstrate the correctness and behavior of both strategies.