Ipek Uzun
30837

CS307 PA4 Report

This assignment is about building a simple virtual memory system for a basic operating system. It supports creating processes, giving memory to them, reading and writing to virtual addresses, switching between processes, and changing the heap size during execution. Each process gets its own page table and memory space, so they don’t interfere with each other. Memory is managed using bitmaps, and each memory page has rules like "can read" or "can write."

Functions

1. initOS()

Initializes the memory and OS-specific status variables. Sets the PID registers, memory allocation bitmaps, and clears any process-related state.

The memory region starting from mem[0] to mem[4] is reserved for system metadata: mem[0]: current running process (initially 0xFFFF), mem[1]: total number of created processes mem[3] and mem[4]: bitmaps for page frame allocation (first 16 and last 16 bits)

2. createProc(char *fname, char *hname)

This function is responsible for creating a new user process within the simulated OS. It sets up the PCB, allocates memory pages for code and heap segments, and loads the program binaries into memory.

Before allocating any resources, the function checks if the OS is currently marked as full by checking mem[2].Then the function retrieves the next available process ID from mem[1], which serves as a counter for the number of processes created so far. This PID is then used to compute the memory address of the PCB for the new process. Then it initializes PCB by process ID, program counter and page table base register.

It uses the allocMem() function to search for free physical frames and update the page table entries. If any of the allocations fail, the function performs cleanup by freeing previously allocated pages and returns with an error.

After successful memory allocation, the function calculates the physical offsets for each allocated frame. These offsets are derived by shifting the allocated page frame numbers to the left by 11 bits. The binary image files specified by fname (for code) and hname (for heap) are loaded into the respective memory regions using the ld_img() function.

If all previous steps succeed, the function increments mem[1], which tracks the number of processes, and returns success.

Ipek Uzun
30837

3. loadProc()

Restores context of the process by setting reg[RPC] and reg[PTBR] from PCB. Updates mem[0] to reflect the current running process.

4. allocMem()

The allocMem() function is responsible for allocating a free physical page frame to a virtual page number (VPN) in a process's address space. It first checks whether the requested VPN is already mapped by inspecting the valid bit in the page table entry (PTE). If not, it searches the system's physical memory bitmap stored in mem[3] and mem[4] for the first available page frame, skipping OS reserved frames. Once a free frame is found, the corresponding bit is cleared to mark it as used, and a new PTE is constructed by combining the physical frame number with the read/write bits and setting the valid bit. This PTE is then written into the page table of the process at PTBR + VPN. The function returns the allocated physical frame number if successful, or -1 if no free frame is found.

5. freeMem()

The freeMem() function deallocates a previously allocated virtual page by clearing its corresponding page table entry (PTE) and marking the associated physical frame as free in the system bitmap. It first checks whether the page is currently valid. If not, the function returns immediately. If the page is valid, it extracts the physical frame number (PFN) from the PTE. Depending on whether the PFN is in the lower or upper half of the memory space, the function updates the appropriate bitmap word (mem[3] for PFNs 0–15, mem[4] for 16–31) by setting the corresponding bit to 1, indicating that the frame is now available. Finally, it clears the valid bit in the PTE to remove the virtual-to-physical mapping. This design ensures safe and efficient memory deallocation with minimal overhead, and it prevents accidental reuse of active frames.

6. tbrk()

The tbrk() function implements dynamic heap memory management by interpreting a request encoded in the reg[R0] register and either allocating or freeing a virtual page accordingly. The upper 5 bits of the register specify the virtual page number (VPN), while the lowest 3 bits indicate the operation type (allocation or free) and the desired read/write permissions. If the request is for allocation, the function first prints a message indicating a heap increase, then checks if the page is already allocated, if so, it prints an error. If not, it attempts to allocate a physical frame using allocMem(), and prints an error if the allocation fails due to lack of space. If the request is for deallocation, the function prints a heap decrease message and verifies whether the page is currently valid; if it is not, it reports an error. Otherwise, it calls freeMem() to release the page.

Ipek Uzun
30837

7. tyld()

The tyld() function performs a context switch by saving the state of the currently running process and transferring control to the next available process in a round-robin manner. It begins by checking whether there is more than one active process; if not, no switch is performed. If a switch is necessary, it searches sequentially through the process table to find the next valid, non-terminated process. Before switching, it saves the current process's state, specifically, its program counter (RPC) and page table base register (PTBR) into its PCB. Then, it loads the next process's RPC and PTBR from its PCB and updates mem[0] to reflect the new active process ID. If the switch results in a change of process, a message is printed indicating the transition.

8. thalt()

The thalt() function is used to gracefully terminate the currently running process by releasing all of its allocated resources and switching to the next available process if one exists. It begins by iterating over the current process's page table entries and freeing any pages marked as valid using freeMem(). Once memory deallocation is complete, the process's PCB entry is marked as terminated by setting its PID field to 0xFFFF. The function then searches for another active process using a circular scan. If none are found, it sets the global running flag to false, effectively halting the system. If a valid process is found, it restores that process's RPC and PTBR from its PCB and updates mem[0] to reflect the switch. If the switch is to a different process, a context-switch message is printed. This function ensures proper cleanup and safe transition of control, simulating operating-system-level process termination and scheduling behavior.

9. mr()

The mr() function reads a 16-bit word from a virtual memory address, translating it into a physical address using the process's page table. It begins by extracting the virtual page number (VPN) and offset from the given address. If the VPN is zero, (reserved for OS use) or if the corresponding page table entry (PTE) is invalid, a segmentation fault is reported and the program exits. Additionally, if the page lacks read permission (bit 1 unset), an error is printed and execution halts. Once all checks pass, the function retrieves the physical frame number (PFN) from the PTE, computes the physical address using the PFN and offset, and returns the value stored at that location in memory.

10. mw()

The mw() function writes a 16-bit value to a virtual memory address, translating it similarly to mr(). It extracts the VPN and offset, then checks for invalid access. If the VPN is 0, or the page is invalid, or if the page does not have write permissions (bit 2 not set), appropriate errors are reported and the program terminates. If the checks pass, the physical address is calculated by combining the PFN from the page table entry with the offset, and the value is written into that physical location.

Ipek Uzun
30837

Helper Function

I have only one helper function which is getfilesize(const char *fname). This function returns the number of 16-bit words in a binary file by opening it in binary mode, seeking to the end to measure its byte size, and dividing by two to convert from bytes to words.