```
#include <assert.h>
#include <string.h>
#include "sim.h"
#include "pagetable.h"

// The top-level page table (also known as the 'page directory')
pgdir_entry_t pgdir[PTRS_PER_PGDIR];

// Counters for various events.
// Your code must increment these when the related events occur.
int hit_count = 0;
int miss_count = 0;
int ref_count = 0;
int evict_clean_count = 0;
int evict_dirty_count = 0;

/*
 * Allocates a frame to be used for the virtual page represented by p.
 * If all frames are in use, calls the replacement algorithm's evict_fcn to
 * select a victim frame.  Writes victim to swap if needed, and updates
 * pagetable entry for victim to indicate that virtual page is no longer in
 * (simulated) physical memory.
 *
 * Counters for evictions should be updated appropriately in this function.
 */
int allocate_frame(pgtbl_entry_t *p) {
      int i;
      int frame = -1;
      for(i = 0; i < memsize; i++) {
            if(!coremap[i].in_use) {
                  frame = i;
                  break;
            }
      }
      if(frame == -1) { // Didn't find a free page.
            // Call replacement algorithm's evict function to select victim
            frame = evict_fcn();

            // All frames were in use, so victim frame must hold some page
            // Write victim page to swap, if needed, and update pagetable
            // IMPLEMENTATION NEEDED

        pgtbl_entry_t *victim = coremap[frame].pte;

        //If the page is not dirty.
        if (!(victim->frame & PG_DIRTY)) {
            evict_clean_count = evict_clean_count + 1;
        } else {
            victim->frame = (victim->frame | PG_ONSWAP);
            evict_dirty_count = evict_dirty_count + 1;
        }

        //Perform the swap.
        int swap_off_number = swap_pageout(frame, victim->swap_off);

        if (swap_off_number != -1) {
            victim->swap_off = swap_off_number;
        } else {
            perror("Swap Error for pageout.\n");
```

```
            exit(1);
        }

        victim->frame = (victim->frame & (~PG_VALID));
        victim->frame = (victim->frame & (~PG_DIRTY));

      }

      coremap[frame].in_use = 1;
      coremap[frame].pte = p;

      return frame;

}

/*
 * Initializes the top-level pagetable.
 * This function is called once at the start of the simulation.
 * For the simulation, there is a single "process" whose reference trace is
 * being simulated, so there is just one top-level page table (page directory).
 * To keep things simple, we use a global array of 'page directory entries'.
 *
 * In a real OS, each process would have its own page directory, which would
 * need to be allocated and initialized as part of process creation.
 */
void init_pagetable() {
      int i;
      // Set all entries in top-level pagetable to 0, which ensures valid
      // bits are all 0 initially.
      for (i=0; i < PTRS_PER_PGDIR; i++) {
            pgdir[i].pde = 0;
      }
}

// For simulation, we get second-level pagetables from ordinary memory
pgdir_entry_t init_second_level() {
      int i;
      pgdir_entry_t new_entry;
      pgtbl_entry_t *pgtbl;

      // Allocating aligned memory ensures the low bits in the pointer must
      // be zero, so we can use them to store our status bits, like PG_VALID
      if (posix_memalign((void **)&pgtbl, PAGE_SIZE,
                  PTRS_PER_PGTBL*sizeof(pgtbl_entry_t)) != 0) {
            perror("Failed to allocate aligned memory for page table");
            exit(1);
      }

      // Initialize all entries in second-level pagetable
      for (i=0; i < PTRS_PER_PGTBL; i++) {
            pgtbl[i].frame = 0; // sets all bits, including valid, to zero
            pgtbl[i].swap_off = INVALID_SWAP;
      }

      // Mark the new page directory entry as valid
      new_entry.pde = (uintptr_t)pgtbl | PG_VALID;

      return new_entry;
}
```

```
/*
 * Initializes the content of a (simulated) physical memory frame when it
 * is first allocated for some virtual address.  Just like in a real OS,
 * we fill the frame with zero's to prevent leaking information across
 * pages.
 *
 * In our simulation, we also store the the virtual address itself in the
 * page frame to help with error checking.
 *
 */
void init_frame(int frame, addr_t vaddr) {
      // Calculate pointer to start of frame in (simulated) physical memory
      char *mem_ptr = &physmem[frame*SIMPAGESIZE];
      // Calculate pointer to location in page where we keep the vaddr
        addr_t *vaddr_ptr = (addr_t *)(mem_ptr + sizeof(int));

      memset(mem_ptr, 0, SIMPAGESIZE); // zero-fill the frame
      *vaddr_ptr = vaddr;             // record the vaddr for error checking

      return;
}

/*
 * Locate the physical frame number for the given vaddr using the page table.
 *
 * If the entry is invalid and not on swap, then this is the first reference
 * to the page and a (simulated) physical frame should be allocated and
 * initialized (using init_frame).
 *
 * If the entry is invalid and on swap, then a (simulated) physical frame
 * should be allocated and filled by reading the page data from swap.
 *
 * Counters for hit, miss and reference events should be incremented in
 * this function.
 */
char *find_physpage(addr_t vaddr, char type) {
      pgtbl_entry_t *p=NULL; // pointer to the full page table entry for vaddr
      unsigned idx = PGDIR_INDEX(vaddr); // get index into page directory

      // IMPLEMENTATION NEEDED
      // Use top-level page directory to get pointer to 2nd-level page table

    uintptr_t entry = pgdir[idx].pde;

    if ((entry & PG_VALID) == 0) {
        pgdir[idx] = init_second_level();
    }


      // Use vaddr to get index into 2nd-level page table and initialize 'p'

    pgtbl_entry_t *pagetable = (pgtbl_entry_t *) (pgdir[idx].pde & PAGE_MASK);
    p = pagetable + PGTBL_INDEX(vaddr);

      // Check if p is valid or not, on swap or not, and handle appropriately

    if (p->frame & PG_VALID) {
        hit_count = hit_count + 1;
```

```
    } else {
        if (p->frame & PG_ONSWAP) {

            int frame = allocate_frame(p);
            int swap_in_page = swap_pagein(frame, p->swap_off);
            if (swap_in_page != 0) {
                perror("Swap Error for pagein.\n");
                exit(1);
            }

            p->frame = frame << PAGE_SHIFT;
            p->frame = p->frame | PG_ONSWAP;
            p->frame = p->frame & (~PG_DIRTY);

        } else {

            int frame = allocate_frame(p);
            init_frame(frame, vaddr);

            p->frame = frame << PAGE_SHIFT;
            p->frame = p->frame | PG_DIRTY;
            p->frame = p->frame | PG_ONSWAP;

        }

        miss_count = miss_count + 1;

    }



      // Make sure that p is marked valid and referenced. Also mark it
      // dirty if the access type indicates that the page will be written to.

    p->frame = p->frame | PG_VALID;
    p->frame = p->frame | PG_REF;

    ref_count = ref_count + 1;

    if (type == 'M' || type == 'S') {
        p->frame = p->frame | PG_DIRTY;
    }

      // Call replacement algorithm's ref_fcn for this page
      ref_fcn(p);

      // Return pointer into (simulated) physical memory at start of frame
      return  &physmem[(p->frame >> PAGE_SHIFT)*SIMPAGESIZE];

}

void print_pagetbl(pgtbl_entry_t *pgtbl) {
      int i;
      int first_invalid, last_invalid;
      first_invalid = last_invalid = -1;

      for (i=0; i < PTRS_PER_PGTBL; i++) {
            if (!(pgtbl[i].frame & PG_VALID) &&
                !(pgtbl[i].frame & PG_ONSWAP)) {
```

```
                if (first_invalid == -1) {
                        first_invalid = i;
                }
                last_invalid = i;
        } else {
                if (first_invalid != -1) {
                        printf("\t[%d] - [%d]: INVALID\n",
                                first_invalid, last_invalid);
                        first_invalid = last_invalid = -1;
                }
                printf("\t[%d]: ",i);
                if (pgtbl[i].frame & PG_VALID) {
                        printf("VALID, ");
                        if (pgtbl[i].frame & PG_DIRTY) {
                                printf("DIRTY, ");
                        }
                        printf("in frame %d\n",pgtbl[i].frame >> PAGE_SHIFT);
                } else {
                        assert(pgtbl[i].frame & PG_ONSWAP);
                        printf("ONSWAP, at offset %lu\n",pgtbl[i].swap_off);
                }
        }
    }
    if (first_invalid != -1) {
        printf("\t[%d] - [%d]: INVALID\n", first_invalid, last_invalid);
        first_invalid = last_invalid = -1;
    }
}

void print_pagedirectory() {
    int i; // index into pgdir
    int first_invalid,last_invalid;
    first_invalid = last_invalid = -1;

    pgtbl_entry_t *pgtbl;

    for (i=0; i < PTRS_PER_PGDIR; i++) {
        if (!(pgdir[i].pde & PG_VALID)) {
                if (first_invalid == -1) {
                        first_invalid = i;
                }
                last_invalid = i;
        } else {
                if (first_invalid != -1) {
                        printf("[%d]: INVALID\n  to\n[%d]: INVALID\n",
                                first_invalid, last_invalid);
                        first_invalid = last_invalid = -1;
                }
                pgtbl = (pgtbl_entry_t *)(pgdir[i].pde & PAGE_MASK);
                printf("[%d]: %p\n",i, pgtbl);
                print_pagetbl(pgtbl);
        }
    }
}
```