Documentation for Phase3 Operating System Project - A.Y. 2023/2024

Introduction

The following document serves as a simple explanation of the main features of the third phase of the OS project for the *uRISCV* architecture.

Index

Virtual Memory Support

Pager

The pager is a module that is responsible for managing the virtual memory of the system. It is responsible for loading and unloading pages from the disk to the memory, using the swap pool as a support struct for handling pages.

```
void pager(void) {
  unsigned status;
  // get the support data of the current process
  support_t *support_data = getSupportData();

  state_t *exception_state = &(support_data->sup_exceptState[PGFAULTEXCEPT]);

  unsigned cause = exception_state->cause & 0x7FFFFFFF;

  // check if the exception is a TLB-Modification exception
  if (cause == TLBMOD) {
    // treat this exception as a program trap
    programTrapExceptionHandler(exception_state);
  }

  gainSwapMutex();

  /* enter the critical section */

  // it's not the actual vpn, but the page index in the backing store
  int vpn = ENTRYHI_GET_VPN(exception_state->entry_hi);
  if (vpn >= MAXPAGES) {
    vpn = MAXPAGES - 1;
```

```
}

// pick a frame from the swap pool
unsigned victim_frame = getFrameFromSwapPool();
memaddr victim_page_addr = SWAPPOOLADDR + (victim_frame * PAGESIZE);

// check if the frame is occupied
if (isSwapPoolFrameFree(victim_frame) == FALSE) {
  // we assume the frame is occupied by a dirty page

  // operations performed atomically
  OFFINTERRUPTS();

  // mark the page pointed by the swap pool as not valid
  swap_pool[victim_frame].sw_pte->pte_entryLO &= !VALIDON;

  // update the TLB if needed
  updateTLB(swap_pool[victim_frame].sw_pte);

  ONINTERRUPTS();

  // update the backing store
  status = writeBackingStore(victim_page_addr, swap_pool[victim_frame].sw_asid,
                             swap_pool[victim_frame].sw_pageNo);
  if (status != DEVRDY) {
    programTrapExceptionHandler(exception_state);
  }

}

// read the contents of the current process's backing store
status =
    readBackingStoreFromPage(victim_page_addr, support_data->sup_asid, vpn);

if (status != DEVRDY) // operation failed
{
  programTrapExceptionHandler(exception_state);
}

// update the swap pool table
swap_pool[victim_frame].sw_asid = support_data->sup_asid;
swap_pool[victim_frame].sw_pageNo = vpn;
swap_pool[victim_frame].sw_pte = &support_data->sup_privatePgTbl[vpn];

// #region atomic operations
OFFINTERRUPTS();
```

```
    // update the current process's page table
    support_data->sup_privatePgTbl[vpn].pte_entryLO |= DIRTYON;
    support_data->sup_privatePgTbl[vpn].pte_entryLO |= VALIDON;
    support_data->sup_privatePgTbl[vpn].pte_entryLO &= 0xfff;
    support_data->sup_privatePgTbl[vpn].pte_entryLO |= victim_page_addr & 0xfffff000;


    // update the TLB
    updateTLB(&support_data->sup_privatePgTbl[vpn]);

    ONINTERRUPTS();
    // #endregion atomic operations

    releaseSwapMutex();
    /* exit the critical section */

    // return control to the current process
    LDST(exception_state);
}
```

The pager is responsible for handling page faults. When a page fault occurs, the pager is invoked and it is responsible for loading the page from the backing store to the memory. The pager is also responsible for updating the TLB and the page table of the current process. It's important to ensure that the operations performed by the pager are atomic, so we disable the interrupts before performing the operations and re-enable them after the operations are completed (see the `OFFINTERRUPTS` and `ONINTERRUPTS` macros).

```
#define OFFINTERRUPTS() setSTATUS(getSTATUS() & (~MSTATUS_MIE_MASK))
#define ONINTERRUPTS() setSTATUS(getSTATUS() | MSTATUS_MIE_MASK)
```

We also need to ensure that the operations performed by the pager are thread-safe, so we use a mutex to ensure that only one pager can run at a time: that's accomplished by calling the `gainSwapMutex` and `releaseSwapMutex` functions, which acquire and release the mutex using message passing.

Swap Pool

The swap pool is a data structure that is used to store the pages that are swapped out of the memory. It is used by the pager to store the pages that are evicted from the memory when a page fault occurs. The swap pool is implemented as an array of swap pool entries, where each entry contains the address of the page in the backing store, the ASID of the process that owns the page, and the page number of the page in the backing store.

```
typedef struct swap_t
{
```

```
    int sw_asid;        /* ASID number          */
    int sw_pageNo;      /* page's virt page no. */
    pteEntry_t *sw_pte; /* page's PTE entry.    */
} swap_t;
```

Page Tables

Are part of the process control block (support struct) and are used to store the mapping between the virtual pages and the physical frames. The page table is implemented as an array of page table entries, where each entry contains the physical frame number, the ASID of the process that owns the page and the process register entriHI and entryLO.

```
typedef struct pteEntry_t
{
    unsigned int pte_entryHI;
    unsigned int pte_entryLO;
} pteEntry_t;
```

Page Replacement Algorithms

The pager uses a page replacement algorithm to decide which page to evict from the memory when a page fault occurs. The algorithm looks for a free frame in the swap pool and if there are no free frames, it selects one in a round-robin fashion. The algorithm is implemented in the `getFrameFromSwapPool` function.

```
unsigned getFrameFromSwapPool() {
  static unsigned frame = 0;
  // find a free frame in the swap pool
  for (unsigned i = 0; i < POOLSIZE; i++) {
    if (isSwapPoolFrameFree(i)) {
      frame = i;
      break;
    }
  }
  // otherwise implement the page replacement algorithm FIFO
  return frame++ % POOLSIZE;
}
```

Read/Write to Backing Store

We use the `readBackingStoreFromPage` and `writeBackingStore` functions to read and write to the backing store. The backing store is a disk that is used to store the pages that are swapped out of the memory. The backing store is implemented as a file that is stored on the disk. We use the ssi apis to read and write to the backing store using the mnemonics `DOIO` operation. Is important to note which how this operation is performed: 1. We set the page address in the flash device register 2. We set the command value in the flash device register 3.

We send a message to the ssi device to perform the operation 4. We receive the status of the operation

```
unsigned flashOperation(unsigned command, unsigned page_addr, unsigned asid,
                        unsigned page_number) {
  dtpreg_t *flash_dev_addr = (dtpreg_t *)DEV_REG_ADDR(IL_FLASH, asid - 1);
  flash_dev_addr->data0 = page_addr;

  unsigned value = (page_number << 8) | command;
  unsigned status = 0;
  ssi_do_io_t do_io = {
      .commandAddr = &(flash_dev_addr->command),
      .commandValue = value,
  };
  ssi_payload_t payload = {
      .service_code = DOIO,
      .arg = &do_io,
  };
  SYSCALL(SENDMESSAGE, (unsigned int)ssi_pcb, (unsigned int)(&payload), 0);
  SYSCALL(RECEIVEMESSAGE, (unsigned int)ssi_pcb, (unsigned int)(&status), 0);
  return status;
}
```

The function two functions: - `readBackingStoreFromPage` - `writeBackingStore`

act as a wrapper around the `flashOperation` function, they are used to read and write to the backing store.

TLB Update

This is one of the most important parts of the pager, the TLB is a cache that is used to store the mappings between the virtual pages and the physical frames. As the TLB is implemented as an array of TLB entries, we can access it using built in macros. The *TLB update* is by performing a check if the page is already in the TLB, if it is, we update the entry, otherwise we add a new entry to the TLB.

```
void updateTLB(pteEntry_t *page) {
  // place the new page in the Data0 register
  setENTRYHI(page->pte_entryHI);
  TLBP();
  // check if the page is already in the TLB
  unsigned is_present = getINDEX() & PRESENTFLAG;
  if (is_present == FALSE) {
    // the page is not in the TLB, so we need to insert it
    setENTRYHI(page->pte_entryHI);
    setENTRYLO(page->pte_entryLO);
    TLBWI();
```

```
  }
}
```

User Exception Handler

The user exception handler is a module that is responsible for handling exceptions that occur in user mode. It is responsible for handling exceptions such as system calls, it is implemented as a switch statement that checks the cause of the exception and calls the appropriate handler function.

```
...
switch (exception_code)
{
case SYSEXCEPTION:
  UsysCallHandler(exception_state, current_support->sup_asid);
  break;
default:
  programTrapExceptionHandler(exception_state);
  return;
  break;
}
...
```

User System Call Handler

The user system call handler act as a wrapper for communication between user process and kernel, in fact it 'wraps' both the `SENDMESSAGE`

```
...
case SENDMSG:
  /* This services cause the transmission of a message to a specified process.
    * The USYS1 service is essentially a user-mode "wrapper" for the
    * kernel-mode restricted SYS1 service. The USYS1 service is requested by
    * the calling process by placing the value 1 in a0, the destination process
    * PCB address or SST in a1, the payload of the message in a2 and then
    * executing the SYSCALL instruction. If a1 contains PARENT, then the
    * requesting process send the message to its SST [Section 6], that is its
    * parent.
    */

  dest_process =
      a1_reg == PARENT ? sst_pcb[asid-1] : (pcb_t *)a1_reg;

  SYSCALL(SENDMESSAGE, (unsigned)dest_process, a2_reg, 0);

    break;
...
```

and the RECEIVEMESSAGE system calls.

```
...
case RECEIVEMSG:
  /* This system call is used by a process to extract a message from its inbox
    * or, if this one is empty, to wait for a message. The USYS2 service is
    * essentially a user-mode "wrapper" for the kernel-mode restricted SYS2
    * service. The USYS2 service is requested by the calling process by placing
    * the value 2 in a0, the sender process PCB address or ANYMESSAGE in a1, a
    * pointer to an area where the nucleus will store the payload of the
    * message in a2 (NULL if the payload should be ignored) and then executing
    * the SYSCALL instruction. If a1 contains a ANYMESSAGE pointer, then the
    * requesting process is looking for the first message in its inbox, without
    * any restriction about the sender. In this case it will be frozen only if
    * the queue is empty, and the first message sent to it will wake up it and
    * put it in the Ready Queue.
    */
  receive_process = a1_reg == PARENT ? sst_pcb[asid-1] : (pcb_t *)a1_reg;
  SYSCALL(RECEIVEMESSAGE, (unsigned)receive_process, a2_reg, 0);

  break;
...
```

System Service Thread

The System Service Thread (SST) is a per-process thread that provide is child process useful services. Each SST child process can send a message to its SST to request a service (that can then be asked to the SSI if needed). The SST also initialize its child and Each share the same ID (ASID) and support struct of its child U-proc. Like the SSI the structure of SST works as a server: get the request, satisfy request and send back resoult.

```
void sstEntry() {
  // init the child
  support_t *sst_support = getSupportData();
  state_t *u_proc_prole = &u_proc_state[sst_support->sup_asid - 1];

  child_pcb[sst_support->sup_asid - 1] =
      initUProc(u_proc_prole, sst_support);
  // get the message from someone - user process
  // handle
  // reply
  while (TRUE) {
    ssi_payload_PTR process_request_payload;
    pcb_PTR process_request_ptr = (pcb_PTR)SYSCALL(
        RECEIVEMESSAGE, ANYMESSAGE, (unsigned)(&process_request_payload), 0);
    sstRequestHandler(process_request_ptr,
```

```
                        process_request_payload->service_code,
                        process_request_payload->arg);
  }
}
```

The SST services are: - `GetTOD`: this service should allow the sender to get back the number of microseconds since the system was last booted/reset.

```
cpu_t getTOD() {
  cpu_t tod_time;
  STCK(tod_time);
  return tod_time;
}
```

- `Terminate`: this service causes the sender U-proc and its SST (its parent) to cease to exist. It is essentially a SST "wrapper" for the SSI service TerminateProcess. It also marks all of the frames in the swap poll from *occupied* to **unoccupied**: this is accomplished by atomically setting the swap pool entry asid to `NOPROC`(-1).

```
void killSST(pcb_PTR sender) {
  notify(test_process);

  // invalidate the page table
  invalidateUProcPageTable(sst_pcb[asid]->p_supportStruct);

  // kill the sst and its child
  terminateProcess(SELF);
}
```

- `WritePrinter`: his service causes the print of a string of characters to the printer with the same number as the sender ASID.
- `WriteTerminal`: his service causes the print of a string of characters to the terminal with the same number as the sender ASID.

```
void write(char *msg, int lenght, devreg_t *devAddrBase, enum writet write_to, int asid) {
  int i = 0;
  unsigned status;
  // check if it's a terminal or a printer
  unsigned *command = write_to == TERMINAL ? &(devAddrBase->term.transm_command)
                                           : &(devAddrBase->dtp.command);

  while (TRUE) {
    if ((*msg == EOS) || (i >= lenght)) {
      break;
    }

    unsigned int value;
```

```
    if (write_to == TERMINAL) {
      value = PRINTCHR | (((unsigned int)*msg) << 8);
    } else {
      value = PRINTCHR;
      devAddrBase->dtp.data0 = *msg;
    }

    ssi_do_io_t do_io = {
        .commandAddr = command,
        .commandValue = value,
    };
    ssi_payload_t payload = {
        .service_code = DOIO,
        .arg = &do_io,
    };

    SYSCALL(SENDMESSAGE, (unsigned int)ssi_pcb, (unsigned int)(&payload), 0);
    SYSCALL(RECEIVEMESSAGE, (unsigned int)ssi_pcb, (unsigned int)(&status), 0);

    // device not ready -> error!
    if (write_to == TERMINAL && status != OKCHARTRANS) {
      programTrapExceptionHandler(&(sst_pcb[asid]->p_supportStruct->sup_exceptState[GENERALE
    } else if (write_to == PRINTER && status != DEVRDY) {
      programTrapExceptionHandler(&(sst_pcb[asid]->p_supportStruct->sup_exceptState[GENERALE
    }

    msg++;
    i++;
  }
}
```

Stdlib

this file serves as the main container of useful functions that are used through phase 3 of the project such as: - `Initializations functions`: *initUprocPageTable(), initFreeStackTop(), initUProc() and defaultSupportData()*; - `Utility functions`: *getSupportData(), getCurrentFreeStackTop(), createChild(), terminateProcess(), updateTLB(), invalidateUProcPageTable() and isOneOfSSTPids()*; - `Notification service & Mutual esclusion handling`: *notify(), gainSwapMutex() and releaseSwapMutex()*.

P3test

This file serves as the main test for the phase 3, it follow a simple schema: initialization, execution of the process's request and termination (after being notified).

```
void test3() {
  test_process = current_process;
  initFreeStackTop();
  /*While test was the name/external reference to a function that exercised the Level 3/Phas
   * in Level 4/Phase 3 it will be used as the instantiator process (InstantiatorProcess).3
   * The InstantiatorProcess will perform the following tasks:
   *  • Initialize the Level 4/Phase 3 data structures. These are:
   *     - The Swap Pool table and a Swap Mutex process that must provide mutex access to th
   *       swap table using message passing [Section 4.1].
   *     - Each (potentially) sharable peripheral I/O device can have a process for it. Thes
   *       will be used to receive complex requests (i.e. to write a string to a terminal)
   *       the correct DoIO service to the SSI (this feature is optional and can be delegat
   *       to the SST processes to simplify the project).
   *  • Initialize and launch eight SST, one for each U-procs.
   *  • Terminate after all of its U-proc "children" processes conclude. This will drive Pro
   *     to one, triggering the Nucleus to invoke HALT. Wait for 8 messages, that should be
   *     each SST is terminated.
   */

  // Init array of support struct (so each will be used for every u-proc init. in initSSTs)
  initSupportArray();

  // alloc swap mutex process
  swap_mutex = allocSwapMutex();

  // Init. sharable peripheral (done in initSSTs)
  /* Technical Point: A careful reading of the Level 4/Phase 3 specification reveals that th
   * actually no purposefully shared peripheral devices. Each of the [1..8] U-procs has its
   * (backing store), printer, and terminal device(s). Hence, one does not actually need a p
   * protect access to device registers. However, for purposes of correctness (or more appro
   * protect against erroneous behavior) and future phase compatibility, it is possible to h
   * each device that waits for messages and requests the single DoIO to the SSI.
   */

  //Init 8 SST
  initSSTs();

  //Terminate after the 8 sst die
  waitTermination(sst_pcb);

  // terminate the test process
  terminateProcess(SELF);
}
```

It also contains some utility functions such as: *initSupportArray(), allocSwapMutex() and waitTermination().*