OS - Labs

Utilities

Sleep

Task: we need to use the user function `sleep` that jumps from user code to the kernel to call the `sys_sleep` system call.

Notes:

- you need to pass `int argc` (count of CL arguments) as well as `char *argv[]` (actual arguments)
- you need to use `atoi` (ASCII to integer) in order to convert the integer passed in `argv[]` to an integer. (`argv[1]`)

Ping pong

Task: we need to design a program that uses a pipe to transfer a byte of data between two processes (a parent and a child).

The algorithm will be as follows:

1- make a file descriptor using an array of two entries `fd[2]` one for stdin (`fd[0]`) and one for stdout (`fd[1]`). Use `pipe(fd)` in order to convert this array into an actual pipe.

2- Make a `fork()` in order to create a child process that will send and receive data from its parent. `fork()` returns `0` in the context of the child process and a positive integer (the process PID) in the context of the parent.

3- In the context:

    1- Write a byte of data to the child.

    2- Wait for its response (reading and writing to the `fd`)

    3- Print a message

4- In the child context:

  1- Receive a byte of data from the `fd`

  2- Print a message

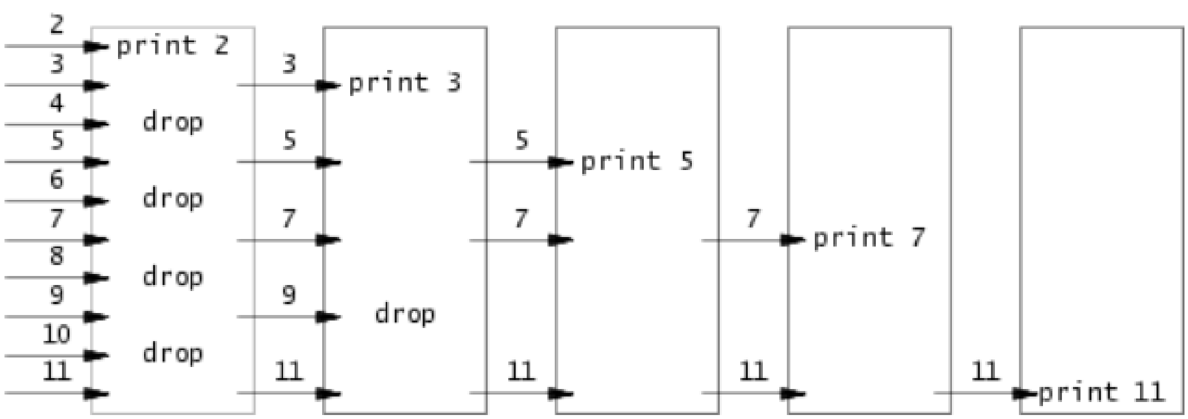
  3- Write a byte of data to the `fd`

Notes:

- Use `wait(0)` to wait for the child to read and write from the `fd` .
- After each read or write, don’t forget to `close` the file descriptor.

Primes

Task: write a concurrent version of primes sieve using pipes and forks.



The algorithm will be as follows:

1- Design an initial process and a fd to write all the numbers to its child.

2- Make a recursive call to your child (until reaching prime 31). Pass to the function your left file descriptor that the child will use to read the unfiltered integers.

3- In each recursive call do the following:

  1- Make a new fd to write the data to your right child.

2- Read the unfiltered integers from your left parent

3- Filter these integers and write them to your right child.

4- Make a fork to create the new left child.

5- In this child's context, call the recursive function and pass your file descriptor.

Notes

- Don't forget to use `wait(0)` in the initial parent to wait for the child to read and write from the `fd`.
- Don't forget to `close` your file descriptors (the left stdin and the right stdout) at each recursive call (to avoid running out of processes).

find

Task: write a user program to find the files in the current directory or its sub-directories that match the name `filename`.

Important background:

- `stat` structure is used to store information about a file (type - size - permissions). The most important piece of information we will be using is `type`:

```
#define T_DIR     1   // Directory
#define T_FILE    2   // File
#define T_DEVICE  3   // Device
```

We will be using the same functions as in the file `ls.c` . Also, we can pass the formatted file name (the name without its parent directories) to the recursive call of the function.

The algorithm will be as follows:

In the switch statement:

1- If the type is file, print its full path

2- If the file is a directory, do the following:

  1- As long as there is a sub-entry, read its contents

  2- If the current subentry's `inum` is equal to `0`, skip it (this indicates an empty subentry)

  3- If the current subentry is not the current directory "`.`" and not the parent directory "`..`", call the recursive function with the same `filename` but add the new name to your cumulative path.

xargs

Task: Implement a user program xargs that feeds the input arguments before the pipe "`|`" to the program after the pipe.

Example:

```
$ echo "1\n2" | xargs echo line
    line 1
    line 2
```

Important background:

The pipe will handle adding the additional input arguments to the `xargs` arguments.

The algorithm will be as follows:

1- Read all the arguments before the pipe (maybe `args1`)

2- Accumulate all the arguments after the pipe separately (maybe `args2`) (as we will be using them multiple times)

3- Iterate over the characters in `args1` and keep an arguments array `arg_arr`:

  If the current character is a new line "`\n`"

  - Create a child that will execute the function after `xargs` with `args2` along with the additional arguments currently appearing in `arg_arr`
  - Empty `arg_arr` to get the remaining arguments in `args1`

Else

- Add this character to `arg_arr`

System Calls

Before starting this lab, we need to have a standard cookbook to make a new system call in xv6:

1- add a prototype for the system call to `user/user.h`

Example:

```
int dummy_syscall(int)
```

2- add a new system call entry in `user/usys.pl`

```
entry("dummy_syscall");
```

3- add a syscall number to `kernel/syscall.h` (Not exceeding 31)

```
#define SYS_dummy  22
```

4- add the new syscall into `kernel/syscall.c` (Write an array of syscallnames as well)

```
extern uint64 sys_dummy(void);  // 1

static uint64 (*syscalls[])(void) = {
...
[SYS_dummy]   sys_dummy,          // 2
};
```

5- add `sys_dummy` (a function takes `void` as argument and returns `uint64`) in `kernel/sysproc.c`. This function will fetch the arguments about the system call and the return values.

6- implement the syscall somewhere in the kernel.

trace

Task: add a new system call to trace all of the system calls called by a program. Trace only the system calls appearing in the variable `mask` , that will be passed as an argument.

Important background:

- `argint` function is used to retrieve the arguments passed to a system call. There is also `argaddr` to retrieve the address of the argument passed to the system call. These two functions expect an integer as the first argument (the index of the argument needed).

- To check if the i^{th} bit is set in an integer mask, use the following statement:

```
if ((1 << i) & mask)
```

The algorithm will be as follows:

1- Do all of the cookbook steps

2- Modify two additional files:

1- In `kernel/proc.h` , add an attribute `tracemask` to the process in order to save the passed tracemask.

2- In `kernel/proc.c`, in the `fork()` part, copy the tracemask of the parent process to the child process.

3- In the `sys_trace` implementation, use `argint(0, &mask)` to get the integer arguments passed to your system call assigned to your `mask` variable.

4- Assign this `mask` to your process' `tracemask` .

5- In `kernel/syscall.c` , modify the `syscall` function to check if the current syscall's bit is set in the process' `tracemask`. If so, print the system call's details.

Sysinfo

Task: add a new system call that collects information about the running system.
The `freemem` field should be set to the number of bytes of free memory, and the `nproc` field should be set to the number of processes whose state is not `UNUSED`.

Important background:

- **`struct run`** is a data structure representing a free memory block in the system.

- The page size in xv6 is 4KB (4096 Bytes)

- The `kmem.freelist` represents the linked list of free memory blocks (pages)

- `kernel/proc.c` defines an array containing all of the system processes (called `proc[NPROC]`)

The algorithm will be as follows:

1- Do all of the cookbook steps

2- Implement the two helper functions `freemem` (return amount of free memory in bytes) and `nproc` (the number of processes that have any status other than `UNUSED` . Don't forget to define the prototypes of these functions as needed.

> `freemem` (In `kernel/kalloc.c`):
>
> 1- Acquire the lock of `kmem` . This is done to ensure that the operation of traversing the free memory list is atomic, preventing race conditions.
>
> 2- Make a new `struct run` (maybe called `r`)
>
> 3- Iterate over the `kmem.freelist` until the value of `r` is `NULL` and keep counting the memory blocks in a variable `count` .
>
> 4- Release the lock and return the value of `count * 4096` (Refer to the background section).

`nproc` (In `kernel/proc.c`):

1- Make a new `struct proc` (maybe called `p`)

2- Iterate over the `proc` array and keep incrementing `count` if this process is not `UNUSED` (acquire the lock of each process and release after checking its status).

3- Return `count`

3- Use `argaddr` to retrieve the `address` of the struct that you will be using to store the system info.

4- Make a new struct variable (maybe called `info`) to get the required info from `freemem` and `nproc`.

5- Use `copyout` to copy these info from the kernel space to the user space:

```
copyout(p->pagetable, address, (char *) &info, sizeof(info))
```

Notes:

- `copyout` is used to copy data from the kernel space to the user space. `address` is the destination address and `info` is the required information to copy.

Page Tables

Speed up system calls

Task: optimize the `getpid()` system call by sharing data between user and kernel spaces (using a page allocated for `usyscall`).

The algorithm will be as follows:

1- Add the `struct usyscall` attribute in `kernel/proc.h` .

2- Modify the `allocproc()` function in `kernel/proc.c` to allocate a page when creating the process:

1- allocate a new page using `kalloc()`

2- make a new `struct usyscall` (maybe called `u`) and assign to its pid the process pid (`p->pid`).

3- Modify the `freeproc()` function in `kernel/proc.c` to free the memory when destroying a process:

if the process has a `usyscall` page (not `NULL`), then free it using `kfree()`

4- Modify the `proc_pagetable()` in `kernel/proc.c` to map the initialized page table using:

```
mappages(pagetable, USYSCALL, PGSIZE, p->usyscall, PTE_R | PTE_U
```

Notes:

- If allocating the page in `kalloc()` failed (`kalloc() == 0`), call `freeproc(p)` to free the allocated process and release the process lock.
- `kfree()` takes the a single argument, which is a pointer to the beginning of the memory block that needs to be deallocated.
- The **`mappages`** function is used to map a range of virtual memory addresses to corresponding physical pages in the page table (**`pagetable`**).
- It maps the page **`USYSCALL`** (a single page) to the physical address stored in **`p->usyscall`**.
- `mappages` returns a value less than `0` to indicate an error
- Note that in this lab, we are not implementing a new syscall. Therefore, the cookbook is not needed.

Print a page table

Task: print the details of a pagetable (described by a given address `pagetable_t`).

Important background:

- XV6 uses a three-level page table as in the image:

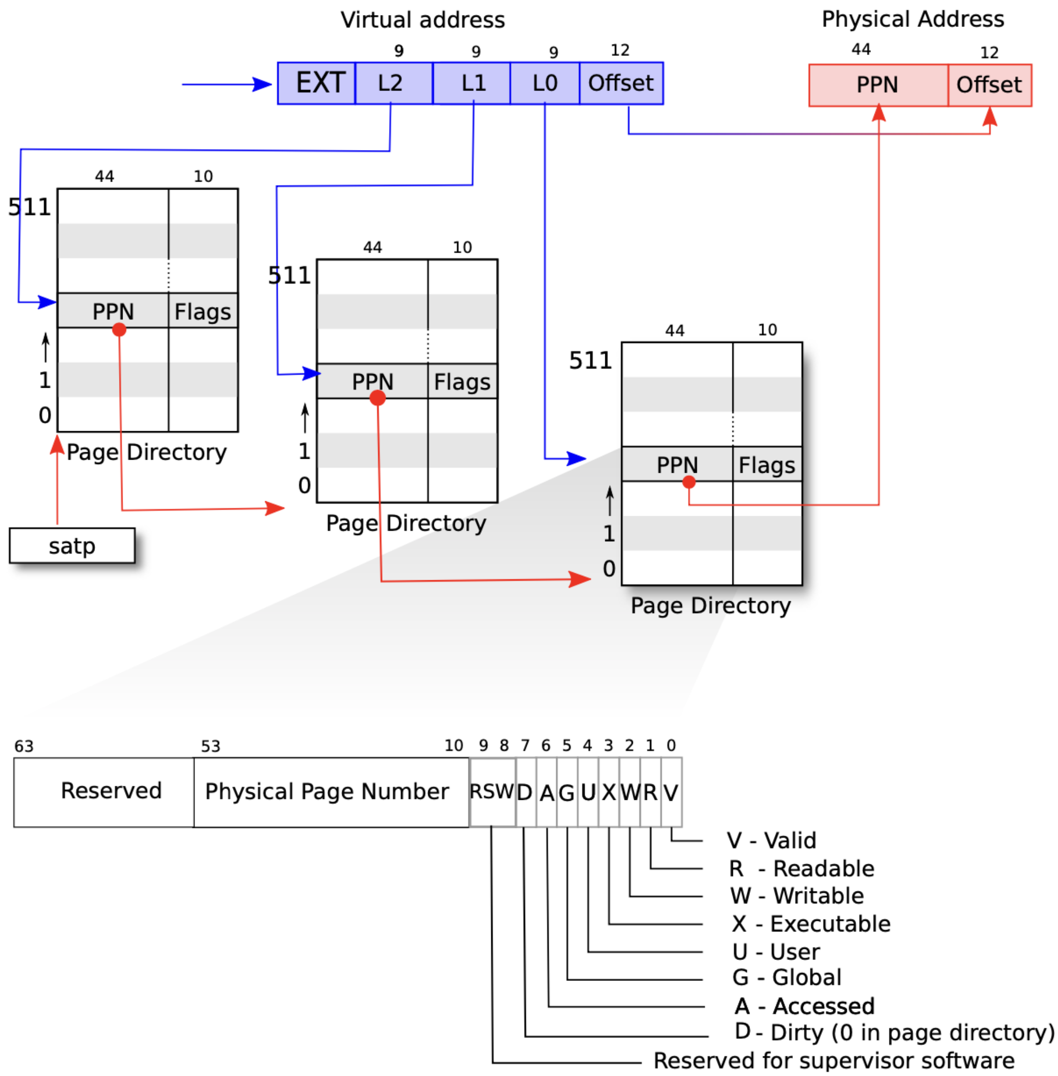


Figure 3.2: RISC-V address translation details.

The algorithm will be as follows:

We will be implementing a helper function (maybe called `page_print`) to walk over the allocated pages in the current level (which will eventually get us to the next levels as well). This function will be almost equal to `freewalk()` in `kernel/vm.c`

1- Iterate over the entries in the current level (a total of 512 entries), if the current page is valid (`PTE_V`) print the needed details.

2- If the page is not writable or readable (`PTE_W` and `PTE_R`), then it has a child (in the next level). Therefore, make another recursive call on its child.

Notes:

- Make use of the function `PTE2PA()` to print the physical address of your current entry.

Detecting which pages have been accessed

Task: implement a system call to detect pages that have been accessed (read or write) in a bitmask (this limits the number of pages to check to 32 - 64).

1- Apply the steps in the cookbook to add a new system call.

2- Parse the three system call arguments using `argint` and `argaddr` (Don't forget the argument index)

3- Add the accessed flag (`PTE_A`) to `kernel/riscv.h`

4- Iterate over the pagetable entries (not more than the number taken from `argint`).

    1- Get each entry's address using `walk()`

    2- If this entry is accessed:

        1- Set its bit in the return `mask`

        2- Clear the `PTE_A` flag

5- Copy the resultant `mask` to the user space using:

```
copyout(pagetable, address, (char *)&mask, sizeof(mask))
```

Notes:

- When using `walk()` start from the first virtual address given by `argaddr` added to the `number of pages so far * PGSIZE`

- Don’t forget to clear the `PTE_A` flag after checking the current entry. Otherwise, the entry will remain accessed forever.

Traps

Backtrace

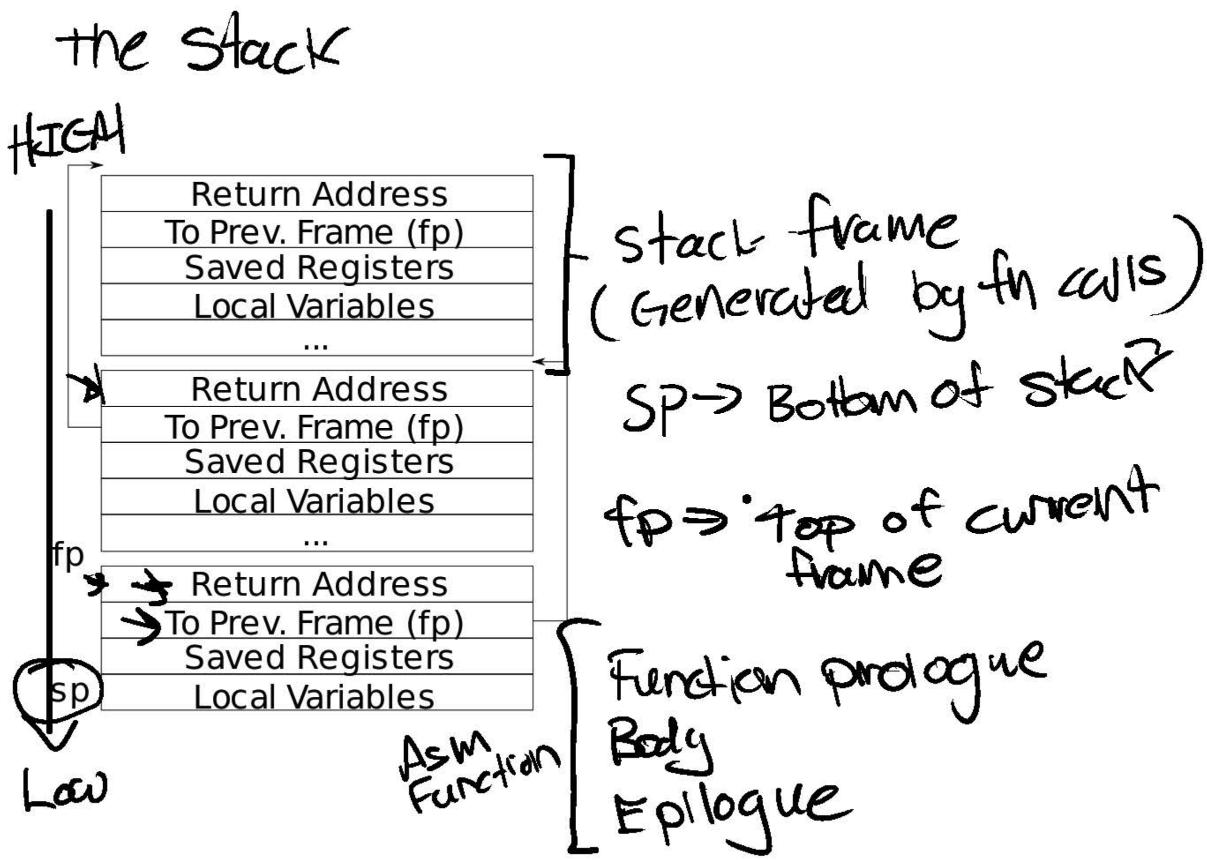
Task: Implement a tracing function to list function calls above the point at which an error occurred

---

Important background:

- The return address of a stack frame is at `fp - 8`

- To get the above call’s frame pointer, access `fp - 16`



The algorithm will be as follows:

We can check the frame pointer position using `PGROUNDDOWN(fp)` and `PGROUNDUP(fp)`

1- While the frame point `fp` is in the region of the stack page (`PGROUNDDOWN(fp) < PGROUNDUP(fp)`

    1- print the return address

    2- move the frame pointer to `fp - 16` (to get the above function call)

Notes:

- Don't forget to add the `r_fp()` function to get the frame pointer of the currently executing function.
- You can compute the top and bottom address of the stack page by using `PGROUNDDOWN(fp)` and `PGROUNDUP(fp)`

Alarm

Task: Add a system call to that alerts a process each `period` ticks. Every `period` , the running process should call the handler function (maybe called `fn`).

Both `period` and `fn` are given as arguments to the system call (integer and pointer).

The algorithm will be as follows:

1- Add the attributes `interval`, `ticks_passed`, `fn_handler`, and `trapframe` to `kernel/proc.h`

2- Add the alarm system call as follows:

    1- Get the `interval` and `fn_handler` using `argint` and `argaddr`. Assign them to the running process.

    2- Handle the clock interrupt in `usertrap()` in `kernel/trap.c` :

        1- if `which_dev` is `2` and the `interval` is not `0`:

            1- Increment the `ticks_passed`:

            2- If the `ticks_passed` is equal to `interval`:

                1- Save the current trapframe into the process' trapframe (using `memmove`) and set its `pc` to the `fn_handler` (in order to execute it).

        2- Call `yield()` to give up the process' CPU time to another process.

3- Add the return syscall (to return to your process):

    1- Just reverse the `memmove` we did in the previous system call. (Save the

Notes:

1- **`which_dev`** is a variable that represents the type of device interrupt being handled. In this context, **`which_dev == 2`** is checking if the interrupt is a clock interrupt.

Copy-on-Write Fork

Implement copy-on write

Task: Implement copy on write to copy the required pages only when a write happens when doing a `fork()`.

The algorithm will be as follows:

1- In `uvmcopy()`, clear the `PTE_W` flag in both the parent and the child.

2- Record that this page is COW mapping (maybe using a new flag `PTE_RSW`).

3- Modify the `kernel/trap.c` to handle the COW trap by allocating a new page with `kalloc()` and install the new page in the PTE with `PTE_W` set.

4- Implement two functions `inc_ref` and `dec_ref` to increase and decrease the page's `ref_count` (that is, the number of user page tables that refer to that page).

Notes:

1- Use `r_scause()` to check for the trap cause.

2- When `kalloc()` allocates a new COW page, set its `ref_count` to `1`

3- You can save the `ref_count` of each page in an array. You need to make its size as large as the largest physical address of any page placed on the freelist by `kinit()`. You can get a page index in this array using `(pa - KERNBASE) / PGSIZE`, where `KERNBASE` is the lowest possible physical address.