Lab 4 - Part B: Copy-on-Write Fork

Sriram V - CS11B058

Exercise 8

- Implemented `sys_env_set_pgfault_upcall()` in `kern/syscall.c`.
- User environments register a page fault handler entrypoint with the JOS kernel using this system call.
- The information is stored in `env_pgfault_upcall` fo the `Env` structure.
- The system call takes the Environment ID and the handler as its arguments.
- We assert that the handler `func` is not `NULL`, and access the relevant `Env` structure associated with `envid` using the `envid2env()` function defined in `kern/env.c`.
- `checkperm` is set to `1` above, as it is a 'dangerous' system call.
- If no such environment exists, we return `-E_BAD_ENV`
- Else, we set the `env_pgfault_upcall` variable of this environment as `func`.
- The function returns `0` on success.

Exercise 9

- Here, `page_fault_handler()` in `kern/trap.c` is modified to handle page faults from the user mode.
- Since the function is required to dispatch page faults to the user-mode handler, we check if `pgfault_upcall` exists for `curenv`. We also check if the environment's stack pointer (stored in `tf->tf_esp`) is within the stack limits. Else, we destroy the enironment.
- If the `esp` value is between the range `UXSTACKTOP-PGSIZE` and `UXSTACKTOP-1` inclusive, then the page fault handler itself has faulted. In such a case, a 32-bit empty word is pushed, and then a `struct UTrapframe`.
- Otherwise the `struct UTrapframe` is directly pushed onto the stack.
- The `UTrapframe` ovject's variables are all set to the current `tf`'s values.
- Then `tf`'s `eip` is set to `curenv`'s `env_pgfault_upcall`, so that the call to `env_run()` will start from that point.
- The `esp` value of `tf` is now updated to point to the beginning of the `UTrapframe` object, as it is the topmost on the stack.
- `env_run()` is then called on `curenv`.

Exercise 10

- The `_pgfault_upcall` routine in `lib/pfentry.S` is wgere the kernel redirects control whenever a page fault is caused in user space.
- The stack pointer `esp` currently points to the exception stack, which we temporarily save in `eax`.
- The `eip` value (where we have to return to) is loaded into `ebx`.

- We then switch to the trap-time stack (`utf_esp`), and push the return address (now in `ebx`) onto the top of this stack.
- The trap-time stack pointer is also updated in the exception stack, since a new value has been pushed onto it.
- We now switch back to the exception stack, skip the fault address and the error, and restore all the trap-time registers (`utf_regs`).
- The return address is skipped over, and the `eflags` are also restored.
- The top of the exception stack now points to the trap-time stack pointer. The value is popped into `esp` to switch to it.
- `ret` is called to return to the address on the top of the trap-time stack, which is the `eip` value we pushed at the start of this code segment.
- Thus, this code successfully restores the state at which the fault occured, and re-runs the code that caused the fault.

Exercise 11

- The `set_pgfault_handler()` method in `lib/pgfault.c` is how the system call to register the page fault handler with the JOS kernel is implemented.
- If `_pgfault_handler` is `0`, it is the first time a handler is being registered. In such a case, we need to allocate an exception stack of size `PGSIZE` at `UXSTACKTOP`, and also tell the kernel to call the `_pgfault_upcall` routine (implemented in the previous exercise) whenever a page fault occurs.
- This is done using the system calls `sys_page_alloc()` and `sys_env_set_pgfault_upcall()` respectively.
- The variable `_pgfault_handler` is then set to the `handler` that was passed, so that the assembly routine can call it when needed.

Exercise 12

1. **`fork()`:**

   - Until now, a `fork()` resulted in all the pages being copied from the parent to the child environment, but this causes a slowdown and also may not necessarily be used always, as an `exec()` may be called which replaces the child environment's address space.
   - Hence, the new implementation of `fork()` copies page mappings instead of pages.
   - A separate copy is created only when either environment modifies the pages – this is called copy-on-write.
   - First, `pgfault()` is set as the page fault handler for `fork()`.
   - Then a new environment is spawned using `sys_exofork()`.
   - The pages of the old environment are iterated over, and the pages are duplicated into the new environment by calling `duppage()`, apart from the last page.
   - The final page (just below `UXSTACKTOP`) is allocated for the exception stack, and the `pgfault_upcall` values are also copied.
   - Finally, the new environment is set to `ENV_RUNNABLE` and its ID is returned.

2. **`pgfault()`:**

   - This function is the page fault handler for the `fork()` process.
   - We check that the fault is a write (`FEC_WR`) and that the page is marked as `PTE_COW`.

- A new page is then allocated at `PFTEMP`, and the contents of the old faulting page at `addr` are copied into this page.
- Finally the new page is mapped with read/write permissions at the old address, in place of the old read-only mapping.

3. **`duppage()`**:

   - First, a check is performed to see if the page is writable or copy-on-write.
   - If it is, the page to be duplicated (present in the current environment `0`'s address space) is mapped to the same virtual address, but in `envid`'s address space, and with the User, Present and COW bits set.
   - Next, the permissions of duplicated page are also changed to copy-on-write.
   - To do this, the page at `va` in `0`'s address space is mapped to the same address in the same environment, but with the appropriate permission bits set.
   - Finally, on success of this entire procedure, `0` is returned.