

# Lab 4 - Part C: Preemptive Multitasking and Inter-Process Communication (IPC)

Sriram V - CS11B058

## Exercise 13

- Since I had implemented the Challenge question in lab 3, inserting IDT entries was trivial. I had to simply increase the number of iterations to 52 for the for loop in `trap_init()` in `kern/trap.c`.
- The corresponding `TRAPHANDLER_NOEC()` macros are also used in `kern/trapentry.s`.
- To ensure that user environments always run with interrupts enabled, we add the `FL_IF` flag to the environment's `EFLAGS`.
- This is done by using `e->env_tf.tf_eflags |= FL_IF;`
- Now if we run any program that runs for a non-trivial length of time (such as `spin`), we see that the kernel is able to preempt the running environment, but prints the trap frames for the hardware interrupts as it is not yet handling them.

## Exercise 14

- Here, the `trap_dispatch()` function in `kern/trap.c` is modified to handle the timer's hardware interrupt.
- We add in a case for the corresponding `trapno`, which would handle the case for the IDT entry corresponding to the timer IRQ.
- The mapping from IRQ number to IDT entry is not fixed. `pic_init` maps IRQs 0-15 to IDT entries `IRQ_OFFSET` to `IRQ_OFFSET+15`.
- Hence, the value we equate `trapno` to, to check for the timer hardware interrupt is `IRQ_OFFSET + IRQ_TIMER`, as this is the corresponding IDT entry.
- If the `trapno` value is indeed equal to this case, we call `lapi_eoi()` to acknowledge the interrupt, and then call the scheduler using `sched_yield()`.
- The `user/spin` test now works. This is because when the child environment running `spin` loops indefinitely, the kernel preempts this environment and allocates the system resources to the next environment in line.
- Thus we see that the parent environment forks off the child, `sys_yield()`s to it a couple of times, but in each case regains control of the CPU after one time slice, and finally kills the child environment and terminates gracefully.

## Exercise 15

### 1. `sys_ipc_recv()`:

- If `dstva` is less than `UTOP`, it means that the environment is willing to receive a page of data. In such a case, we check if `dstva` must be page-aligned, or else we throw an error.
- If it is page-aligned, `curenv's env_ipc_dstva` is set to `dstva`, `env_ipc_recving` is set to 1 and `env_status` is set to `ENV_NOT_RUNNABLE`, before returning a 0.
- This is done to indicate to other environments that this environment wants to receive data.

### 2. `sys_ipc_try_send()`:

- Here we first check if the environment associated with the passed `envid` exists using `envid2env()`, and return an error if no such environment exists.
- `checkperm` is set to 0 above, since we don't need to check permissions.

- `-E_IPC_NOT_RECV` error is thrown if the `envid` is not blocked is `sys_ipc_recv()`, or another environment has managed to send first.
- If `srcva` is less than `UTOP`, it means that the sender wants to send a page. Also, since the check above has also been passed, it means that `envid` is willing to receive.
- If it is not less than `UTOP`, a page won't be transferred and hence `env_ipc_perm` is set to 0.
- However, if it is lesser, we check that `srcva` is page aligned and that it has the relevant permissions of being present and a user page etc.
- The page is found by using `page_lookup()` at `srcva`, and if the page exists and both the page and its page table entry are writable, we proceed forward. Else, we throw an error.
- Since all checks have been passed, we finally insert the page at the relevant `dstva` address in the receiver's address space.
- `env_ipc_recving` is set to 0 since the message has been sent.
- Other variables such as `env_ipc_from` (environment that sent) and `env_ipc_value` (value that was sent) are also set.
- Finally the receiver is made runnable and the function returns 0 (success).

### 3. `ipc_recv()`:

- This is a library function implemented in `lib/ipc.c`, so that C programs can call this function to make the system call to receive a message.
- `from_env_store` stores the sender's `envid`, and `perm_store`, the permissions of the page sent.
- We set both these values in this function (copying `env_ipc_from` and `env_ipc_perm` into them) if the corresponding arguments are non-null.
- We also make use of the error returned by the system call `sys_ipc_recv()` here. If there is an error, the two `_store` variables are set to 0 before returning the error.
- If there are no errors, `ipc_recv()` returns the value it receives, which is stored in `env_ipc_value` of `thisenv`.

### 4. `ipc_send()`:

- This is a library function implemented in `lib/ipc.c`, so that C programs can call this function to make the system call to send a message.
- First we check if a page is to be sent, by checking if it is non-null. If it is `NULL`, it means that only the value is to be sent, so `pg` is set to `UTOP` to indicate to `ipc_recv()` that no page is being sent.
- Next, we try to send the passed `pg` (with permissions `perm`) and `value` to the environment (with `envid` equal to `to_env`).
- If the result is the error `-E_IPC_NOT_RECV`, it means that `to_env` does not wish to receive a message. Hence, the message isn't sent from `curenv` to `to_env`, and we do a `sys_yield()` to relinquish this environment's control over the CPU.
- Thus everytime control comes back to the current environment, the check is performed again and again till `to_env` is in a state where it is willing to receive, in which case the result is 0 (success), and we exit out of the while loop.
- In case of any other error, the `while` condition is false, but the result is non-zero, so the kernel panics.
- Both `user/pingpong` and `user/primes` work successfully now.
- `make grade` successfully gives 75/75.