COL331: Lab 1

Instructor: Smruti R. Sarangi

Due on February 25, 2019.

Rahul V
2016CS10370

1 Adding *syscalls* to xv6

1.1 *Syscalls* added

- `print_count`: prints the number of times each `syscall` was invoked since state changed to `TRACE_ON`.
- `toggle`: toggles the `trace_state` and resets the counter for each `syscall`.
- `add`: adds two integer arguments and returns their sum.
- `ps`: prints all current processes, i.e., all the processes which are not in `UNUSED` state.
- `send`: sends `msg` from process `sender_pid` to process `rec_pid`.
- `recv`: receives `msg` from mailbox. This is a blocking call.
- `send_multi`: sends `msg` from process `sender_pid` to list of processes `rec_pid[]`. An interrupt is sent to each of the `rec_pid[i]` to notify about new `msg`.

1.2 How to add a *syscall*

Adding a *syscall* requires changes made in multiple files, namely:

- `defs.h`: add a forward declaration for functions to be defined in `proc.c`
- `user.h`: declare the function that can be called through shell, i.e., name of the *syscall*.
- `usys.S`: Function call is then linked to the respective *syscall* using the macro in this file.
- `syscall.h`: define the position of the syscall vector, syscall number.
- `syscall.c`: `extern` declare the function to be defined in `sysproc.c` and add function to syscall vector.
- `sysproc.c`: add the real implementation of the *syscall*.
- `user_prog.c`: Handler function for the new *syscall*.

1.3 Example

```
// File: user.h
int add(int, int);

// File: usys.S
SYSCALL(add) // SYSCALL is the macro defined in the same file and mentioned above


// File: syscall.h
#define SYS_add 1 // ascending order is you will

// File: syscall.c
extern int sys_add(void); // arguments are taken from stack.
static int (*syscalls[])(void) = {
[SYS_add]   sys_add,
. . .
. . .
. . .
. . .
};

// File: sysproc.c
// argint is defined in syscall.c, reads int from tf->esp
int sys_add(void) {
    int a;
    if (argint(0, &a) < 0) return -1;

    int b;
    if (argint(1, &b) < 0) return -1;

    return a + b;
}

// File: add.c
// change atoi to convert negative numbers to int too.
int main(int argc, char **argv) {
    if (argc != 3) {
        printf(2, ``usage: add a b\n''); //error
    } else {
        printf(1, ``%d\n'', add(atoi(argv[1]), atoi(argv[2])));
    }
    exit();
}
```

1.4 Additional details

- A lock is necessary on `syscalls_count[]`. This is initialized in `proc.c` and called from `main.c`.
- *syscalls* like `ps` and `send` recquire access to ptable and so are defined in `proc.c` and declared in `defs.h`
- Without $*syscall*`.c`, you will not be able to invoke *syscall* from shell.
- `atoi` in `ulib.c` is changed to handle negative numbers.

2 Inter-process Communication

We use the *syscall*s `send` and `recv` for communication between the processes. Each process has a personal *message_queue*; functions like a mailbox.

2.1 Unicast

- `send`:
  1. `acquire`s the lock on `ptable` as it's a *write*.
  2. *enqueues* the `msg` in the `rec_pid`'s queue.
  3. On successful *enqueue*, wakes up the process `rec_pid`
  4. `release`s the acquired lock.
- `recv`:
  1. This is a blocking call.
  2. `acquire`s the lock on `ptable`.
  3. Tries to *dequeue* from it's message queue.
  4. If unsuccessful (empty message queue), it puts itself to sleep until there's a new message.
  5. On waking up, it again *dequeue*s and returns with the message.
  6. `release`s the lock on `ptable`.

3 Distributed Algorithm:

3.1 *Sum* using UNICAST

- The *array* is divided into `N_CHILD_PROCESSES` equal contiguous segments.
- Each child process computes *partial_sum* on it's assigned segment.
- Child then sends the result to it's parent using *syscall* `send`
- Parent collects all the *msg*s from it's child processes using *syscall* `recv`.
- Parent, then, consolidates these *partial_sum*s into *total_sum*

```
// File: assig1_8.c
// N_PROCESSES == 1 is handled seperately;
int get_sum(short* a, int size) {
    int parent_id = getpid();
    int length_of_segment = size/(N_PROCESSES - 1);

    for (int i = 0; i < N_PROCESSES - 1; ++i) {
        if (!fork()) {
            int start = i * length_of_segment;
            int end = start + length_of_segment;
            if (i == N_PROCESSES - 2) end = size;

            int sum = accumulate_range(a, start, end);
            send(getpid(), parent, itoa(sum));
            exit();
        }
    }

    int ret = consolidated_sum();
    return ret;
}

int consolidated_sum() {
    int ret = 0;
    char *sum = (char *) malloc(MSGSIZE);
    for (int i = 0; i < N_PROCESSES - 1; ++i) {
        recv(sum);
        ret += atoi(sum);
    }
    return ret;
}
```