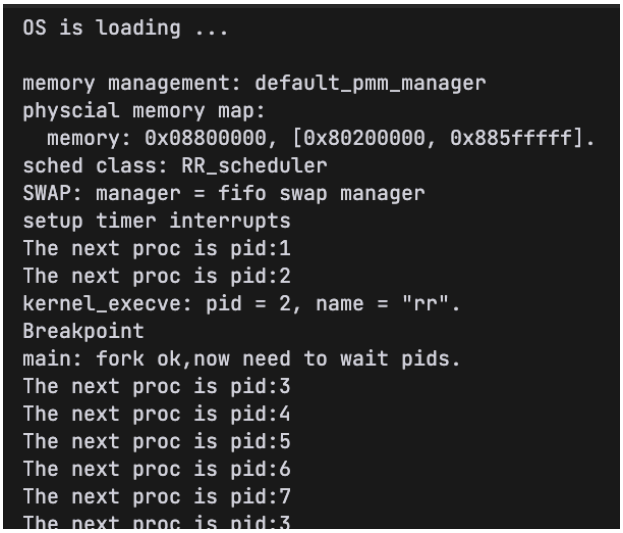
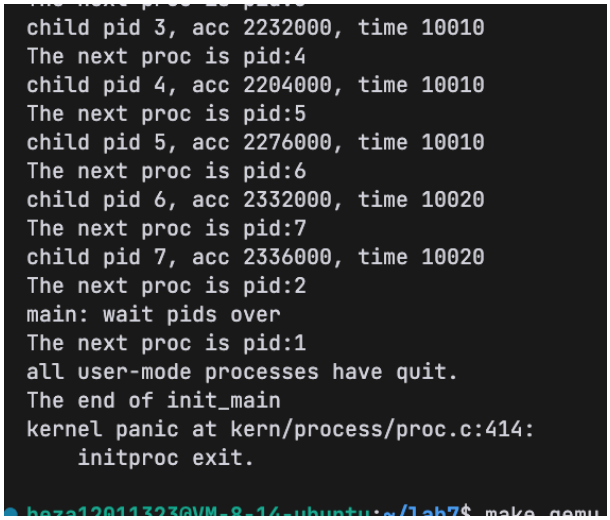
CS302 Lab7 Report

> 何泽安 12011323
>
> 2023.3.29

How the relevant processes are scheduled in `user\rr.c` (time slot Round-Robin):





1. *The order of execution of each process:*

   As shown above, after OS started up, the kernel process `pid=0` creates the idle process `pid=1`, then it runs the round-robin lab's main function, in `pid=2`. Then this process goes into the for loop and creates 5 child processes (note that for the uCore system, there is enough time for process `pid=2` to create 5 children, before it is scheduled).

```
int main(void) {
    int i, time;
    memset(pids, 0, sizeof(pids));

    for (i = 0; i < TOTAL; i++) {
        acc[i] = 0;
        if ((pids[i] = fork()) == 0) {  // <----
            // ...
```

After `pid=2` quickly created the children `pid=3,4,5,6,7`, it goes through the next for loop to wait for all of the children to finish and enter the ZOMBIE state. When waiting, the time slot assigned to this process is used up, and the round-robin starts executing the processes in the order of `3-4-5-6-7-2-3-...`. After the children exited, they will not be scheduled anymore, and finally, `pid=2` will also finish executing, and be removed from the scheduler.

2. *When and how does it enter the scheduler queue:*

   The `RR_enqueue` function which schedules a process to enqueue in boxed in the `default_sched_class` instance.

```
struct sched_class default_sched_class = {
    .name = "RR_scheduler",
    .init = RR_init,
    .enqueue = RR_enqueue,
    .dequeue = RR_dequeue,
    .pick_next = RR_pick_next,
    .proc_tick = RR_proc_tick,
};
```

```
// kern/schedule/sched.c
static inline void sched_class_enqueue(struct proc_struct *proc) {
    if (proc != idleproc) {
        sched_class->enqueue(rq, proc);
    }
}

void schedule(void) {
    bool intr_flag;
    struct proc_struct *next;
    local_intr_save(intr_flag);
    {
        current->need_resched = 0;
        if (current->state == PROC_RUNNABLE) {
            sched_class_enqueue(current);  // <----
        }
        if ((next = sched_class_pick_next()) != NULL) {
            sched_class_dequeue(next);
        }
```

```
        if (next == NULL) {
            next = idleproc;
        }
        next->runs ++;
        if (next != current) {
            cprintf("The next proc is pid:%d\n",next->pid);
            proc_run(next);
        }
    }
    local_intr_restore(intr_flag);
}
```

When a timer interrupt happens, the scheduler does the schedule, which will check if the current process is still runnable, and the current time slot is used up, it will set the next available time slot as the max_time_slot, and put the process (task) to the end of the scheduler queue.

3. *When it will be switched:*

   The scheduler performs when a timer interrupt happens, generally, it will run a tick. Other actions like enqueue will be then performed if specified conditions are satisfied, as previously mentioned.

```
static void RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        proc->time_slice --;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
}
```

4. *What happens at the end of execution:*

   The children are stuck in this for loop until MAX_TIME reaches.

```
if (acc[i] % 4000 == 0) {
  if ((time = gettime_msec()) > MAX_TIME) {
    cprintf("child pid %d, acc %d, time %d\n", getpid(), acc[i], time);
    exit(acc[i]);
  }
}
```

When finished, it prints the relevant information, and exits (mark itself as ZOMBIE and let its parent do the clean-up job).

As for `pid=2`, it exits after all the children are finished (after been blocked by `waitpid`).

```
cprintf("main: wait pids over\n");
return 0;
```