Processes & Multi-thread

(Please go through all the links in this passage)

Process/Thread Concepts:

https://en.wikipedia.org/wiki/Thread_(computing)
http://linux.linti.unlp.edu.ar/images/5/55/Ulk3-cap3.pdf
https://www.tutorialspoint.com/operating_system/os_multi_threading.htm

Related system calls in Linux:

Fork:
https://en.wikipedia.org/wiki/Fork_(system_call)
fork()
http://man7.org/linux/man-pages/man2/fork.2.html
clone()
http://man7.org/linux/man-pages/man2/clone.2.html
wait()
http://man7.org/linux/man-pages/man2/waitid.2.html
kill()
http://man7.org/linux/man-pages/man2/kill.2.html
SIGNALs
http://man7.org/linux/man-pages/man7/signal.7.html
  Term   Default action is to terminate the process.
  Stop   Default action is to stop the process.
  Cont   Default action is to continue the process if it is currently stopped.
getpid()
http://man7.org/linux/man-pages/man2/getpid.2.html
http://stackoverflow.com/questions/9305992/linux-threads-and-process
exit()

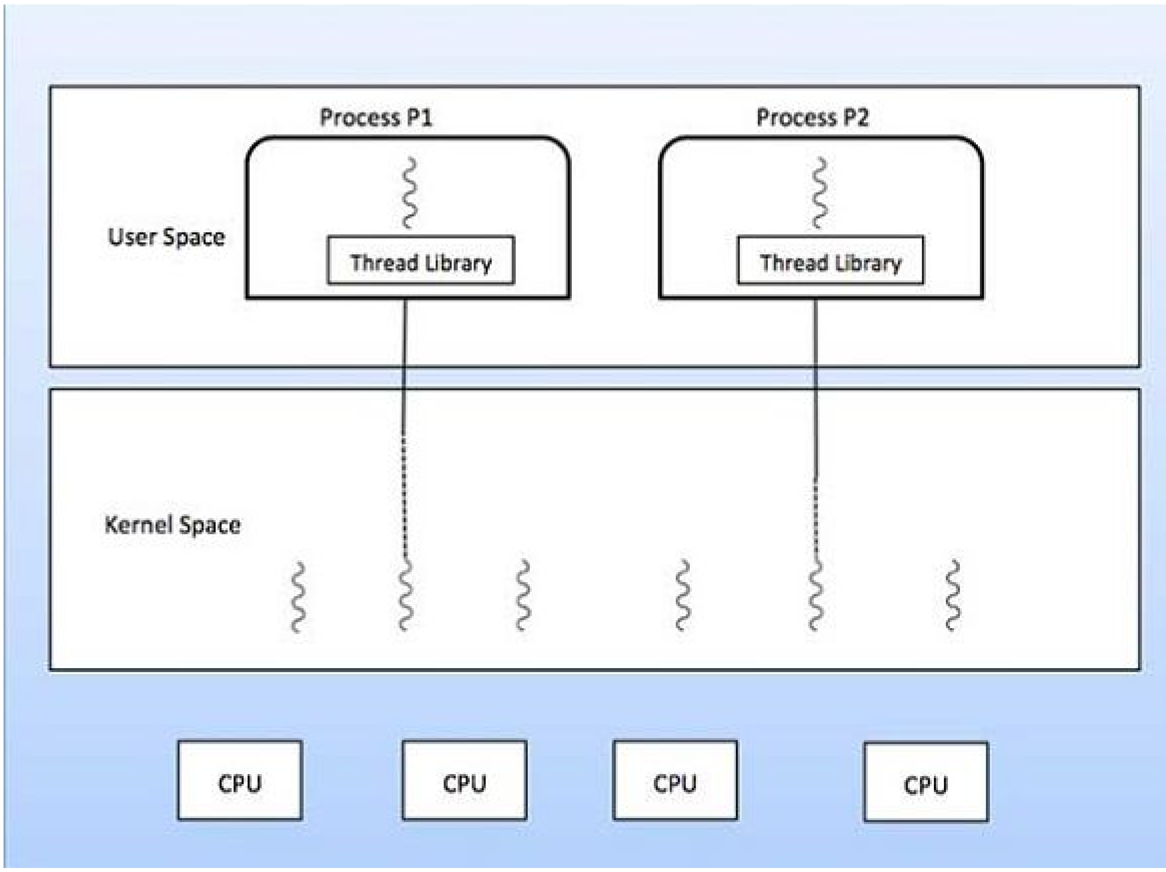
Data structure:

Doubly linked list:
https://0xax.gitbooks.io/linux-insides/content/DataStructures/dlist.html
http://isis.poly.edu/kulesh/stuff/src/klist/list.h
http://isis.poly.edu/kulesh/stuff/src/klist/
https://github.com/torvalds/linux/blob/master/include/linux/list.h

Registers in ARM32
https://en.wikibooks.org/wiki/Embedded_Systems/ARM_Microprocessors
https://www.cl.cam.ac.uk/~fms27/teaching/2001-02/arm-project/02-sort/apcs.txt

Thread Model in pi-OS

One to one thread model



There're four CPU cores on Raspberry Pi 3B. In Pi-OS, we're currently using only one CPU processor(CPU Core 0). Both the kernel and user tasks are running on Core 0. We're not going to simulate true threads on different cores, since it can be complex to synchronize thread resources. We're going to do something similar to Linux multi-thread/multi-process concepts.

Implementation idea in pi-OS is similar to http://linux.linti.unlp.edu.ar/images/5/55/Ulk3-cap3.pdf.

Missions:

Read carefully: (don't need to care about X86 part)
http://linux.linti.unlp.edu.ar/images/5/55/Ulk3-cap3.pdf

Fork() system call

When a process call fork(), the result is to create a new process/thread sharing the same code but have a separate task_struct and stack. There're many advantages for multi-threads or multi-processes, and the main purpose is to increase parallel computing capability. The process/thread that fork() created is a new process managed by kernel, but the user may just feel they're different threads in one process. So they're called light-weight process. It may not be conceptually accurate here, but this doesn't affect our experiment.

Implementation: then a user task fork(), it need to store current CPU context in its task_struct.Regs, just like sleep() does. When the kernel receives the request, kernel would create a new child task.

It first copies the task_struct from parent, then modifies task_struct. Child process needs to allocate a new stack, to replace stack from parent in the task_struct. Child process needs to modify the sp(r13) and fp(r11) registers in task_struct.Regs, since these two registers record displacement from stack head. **Sp** and **fp** need to be adjusted to new child stack. For parent process and child process, they both restart/start from just after fork(), so **pc(r15)** and **lr(r14)** will be the same.

The return value of fork() is different: return value is 0 for child process and child_pid for parent process. return value is -1 on any types of error. After all these preparation, kernel would add the new process into run queue. So now child process and parent process are two different processes in the task queue. When the system call handling finishes, it'll return control to scheduler. The scheduler then schedules normally from the run queue.

Wait() system call

Functionality: before exiting, the parent process wants to ensure that child process resources have been released. So the parent process will end after the exit of all child processes. We can implement this feature through *active_children_count* in *task_struct*.

When the parent wait() for all child processes, if the *active_children_count* is not zero, its state will become TASK_INTERRUPTIBLE; if *active_children_count* equals zero, *wait()* just do nothing (we can check this in pi_libc.c, and no need to make real system call).

When a child process exit(), it will notify the parent, and *active_children_count* in parent *task_struct* will be decremented.

Wait()
returns 1 for process that need to wait,
returns 0 for process not need to wait.

You need to consider all situations here to make sure parent process state can be handled properly.

Process relationship management

If a process is created by kernel/scheduler, its parent is itself; otherwise parenthood-ship need to be carefully maintained. Child process has the same priority as parent process.

Implementation explanation

There're some changes concerning process creation and management, find related changes in main.cpp, kernel.cpp and sched.cpp.

```
    InitializeScheduler();
    sysCall * pSystemCall = getSysCallPointer();
    pSystemCall->print = & kernelPrint;
    pSystemCall->printV = & kernelPrintV;
    pSystemCall->addKernelTimer = & addKernelTimer;

    task_struct * pSchedulerTask = Kernel.createTaskStruct("scheduler", 10);
    Kernel.initTask(pSchedulerTask,&TaskEntry2, 0);
    setSchedulerTask(pSchedulerTask);

    task_struct * pTask4 = Kernel.createTaskStruct("task4", 30);
    task_struct * pTask5 = Kernel.createTaskStruct("task5", 30);
    task_struct * pTask6 = Kernel.createTaskStruct("task6", 30);
    Kernel.initTask(pTask4,&TaskEntry2, &task4_run);
    Kernel.initTask(pTask5,&TaskEntry2, &task5_run);
    Kernel.initTask(pTask6,&TaskEntry2, &task6_run);
    Kernel.schedNewTask(pTask4);
    Kernel.schedNewTask(pTask5);
    Kernel.schedNewTask(pTask6);
```

Task_struct is defined in task.h:

```c
typedef struct Ttask_struct{
    int state;
    int prio;
    int policy;
    struct Ttask_struct* parent;
    struct list_head tasks;
    int pid;
    void (*RunFunc)();
    void (*taks_entry)(struct Ttask_struct * pTask);
    unsigned    WakeTicks;
    TTaskRegisters      Regs;
    u8          *pStack;
    unsigned        StackSize;
    int active_child_count;
    int isWaiting;
}task_struct;
```

List_head is defined in list.h. It’s used to manage the task doubly linked list.

- *pid* is short for Process ID;
- *ppid* is short for Parent Process ID;
- *state* is one value from TaskState:

```c
typedef enum
{
    TaskStateReady,
    TASKRUNNING,
    TASKINTERRUPTIBLE,
    TASKSTOPPED,
    TASKZOMIE,
    TASKDEAD,
    TaskStateUnknown
}TTaskState;
```

You need to implement codes here:

```
int CKernel::fork(){
//TODO

    return 0;
}

int CKernel::forkInitChildTask(task_struct * pTask){
//TODO

}

int CKernel::wait(){
//TODO
    return 0;

}

int CKernel::threadExit(){
//TODO

}
```

forkInitChildTask() implementation detail:

Please refer to **Fork() system call** above for implementation detail.

threadExit() implementation detail:

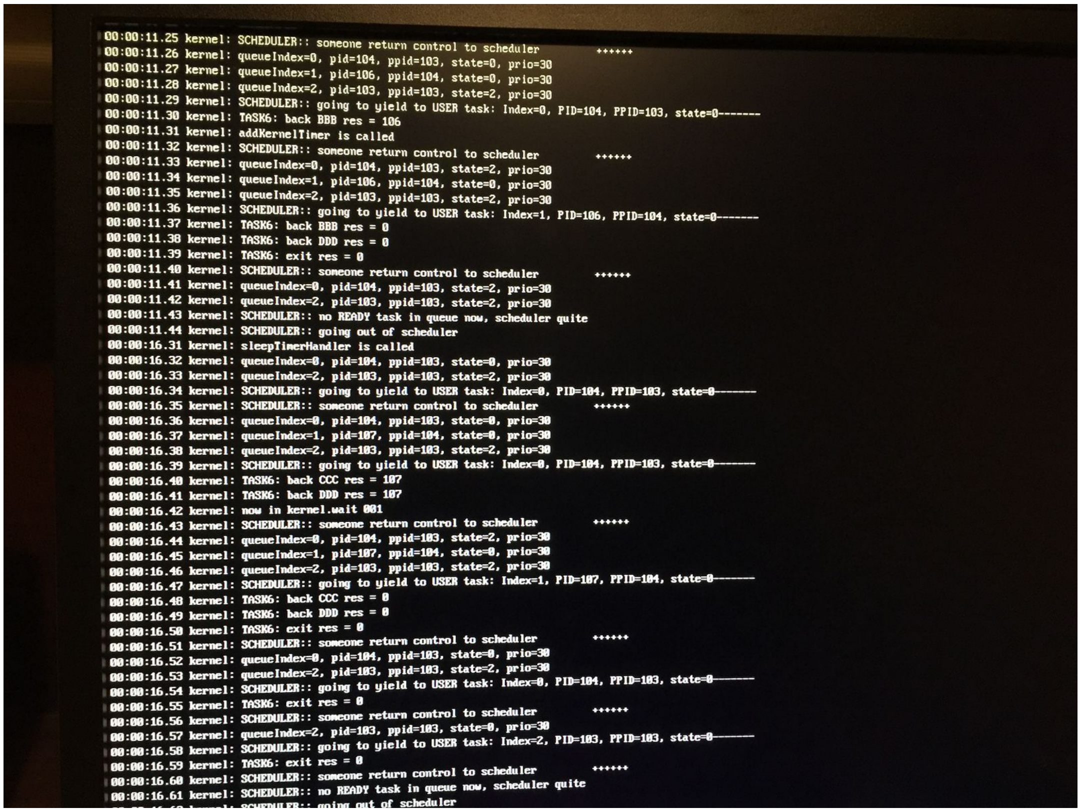
When a process exit(), if it is a process having a parent process which is different from itself, it should notify its parent process and update parent process's *state* and *active_child_count* accordingly. A process's state after exit could be in either TASKZOMBIE or TASKDEAD. If parent has active child process when it is trying to exit, then its state is TASKZOMBIE; otherwise TASKDEAD.

The last step of threadExit() should properly call yieldPrepare() to remove itself from run queue;

Task 6 is already defined:

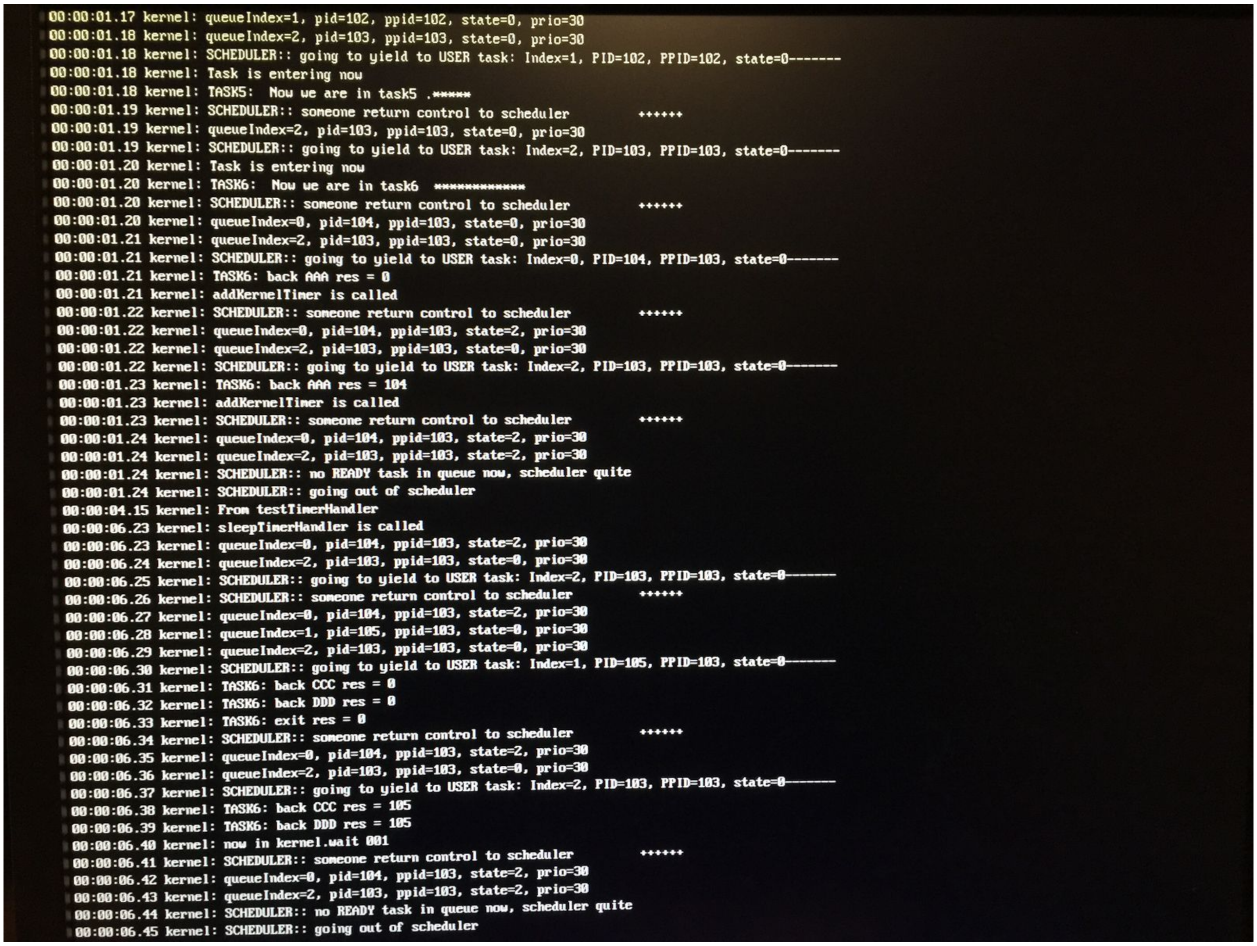
```c
void task6_run()
{
    printf("TASK6:  Now we are in task6  ************");
    int res = fork(2);
    printf("TASK6: back AAA res = %d ", res);

    if(res == 0) {
        sleep(10);
        res = fork(2);
        printf("TASK6: back BBB res = %d", res);
    }

    if(res != 0) {
        sleep(5);
        res = fork(2);
        printf("TASK6: back CCC res = %d", res);
    }

    printf("TASK6: back DDD res = %d ", res);
    wait(2);
    printf("TASK6: exit res = %d ", res);
    exit(1);
}
```

Expected output:



```
00:00:11.25 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:11.26 kernel: queueIndex=0, pid=104, ppid=103, state=0, prio=30
00:00:11.27 kernel: queueIndex=1, pid=106, ppid=104, state=0, prio=30
00:00:11.28 kernel: queueIndex=2, pid=103, ppid=103, state=2, prio=30
00:00:11.29 kernel: SCHEDULER:: going to yield to USER task: Index=0, PID=104, PPID=103, state=0-------
00:00:11.30 kernel: TASK6: back BBB res = 106
00:00:11.31 kernel: addKernelTimer is called
00:00:11.32 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:11.33 kernel: queueIndex=0, pid=104, ppid=103, state=2, prio=30
00:00:11.34 kernel: queueIndex=1, pid=106, ppid=104, state=0, prio=30
00:00:11.35 kernel: queueIndex=2, pid=103, ppid=103, state=2, prio=30
00:00:11.36 kernel: SCHEDULER:: going to yield to USER task: Index=1, PID=106, PPID=104, state=0-------
00:00:11.37 kernel: TASK6: back BBB res = 0
00:00:11.38 kernel: TASK6: back DDD res = 0
00:00:11.39 kernel: TASK6: exit res = 0
00:00:11.40 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:11.41 kernel: queueIndex=0, pid=104, ppid=103, state=2, prio=30
00:00:11.42 kernel: queueIndex=2, pid=103, ppid=103, state=2, prio=30
00:00:11.43 kernel: SCHEDULER:: no READY task in queue now, scheduler quite
00:00:11.44 kernel: SCHEDULER:: going out of scheduler
00:00:16.31 kernel: sleepTimerHandler is called
00:00:16.32 kernel: queueIndex=0, pid=104, ppid=103, state=0, prio=30
00:00:16.33 kernel: queueIndex=2, pid=103, ppid=103, state=2, prio=30
00:00:16.34 kernel: SCHEDULER:: going to yield to USER task: Index=0, PID=104, PPID=103, state=0-------
00:00:16.35 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:16.36 kernel: queueIndex=0, pid=104, ppid=103, state=0, prio=30
00:00:16.37 kernel: queueIndex=1, pid=107, ppid=104, state=0, prio=30
00:00:16.38 kernel: queueIndex=2, pid=103, ppid=103, state=2, prio=30
00:00:16.39 kernel: SCHEDULER:: going to yield to USER task: Index=0, PID=104, PPID=103, state=0-------
00:00:16.40 kernel: TASK6: back CCC res = 107
00:00:16.41 kernel: TASK6: back DDD res = 107
00:00:16.42 kernel: now in kernel.wait 001
00:00:16.43 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:16.44 kernel: queueIndex=0, pid=104, ppid=103, state=2, prio=30
00:00:16.45 kernel: queueIndex=1, pid=107, ppid=104, state=0, prio=30
00:00:16.46 kernel: queueIndex=2, pid=103, ppid=103, state=2, prio=30
00:00:16.47 kernel: SCHEDULER:: going to yield to USER task: Index=1, PID=107, PPID=104, state=0-------
00:00:16.48 kernel: TASK6: back CCC res = 0
00:00:16.49 kernel: TASK6: back DDD res = 0
00:00:16.50 kernel: TASK6: exit res = 0
00:00:16.51 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:16.52 kernel: queueIndex=0, pid=104, ppid=103, state=0, prio=30
00:00:16.53 kernel: queueIndex=2, pid=103, ppid=103, state=2, prio=30
00:00:16.54 kernel: SCHEDULER:: going to yield to USER task: Index=0, PID=104, PPID=103, state=0-------
00:00:16.55 kernel: TASK6: exit res = 0
00:00:16.56 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:16.57 kernel: queueIndex=2, pid=103, ppid=103, state=0, prio=30
00:00:16.58 kernel: SCHEDULER:: going to yield to USER task: Index=2, PID=103, PPID=103, state=0-------
00:00:16.59 kernel: TASK6: exit res = 0
00:00:16.60 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:16.61 kernel: SCHEDULER:: no READY task in queue now, scheduler quite
```

```
00:00:01.24 kernel: queueIndex=2, pid=103, ppid=103, state=2, prio=30
00:00:01.24 kernel: SCHEDULER:: no READY task in queue now, scheduler quite
00:00:01.24 kernel: SCHEDULER:: going out of scheduler
00:00:04.15 kernel: From testTimerHandler
00:00:06.23 kernel: sleepTimerHandler is called
00:00:06.23 kernel: queueIndex=0, pid=104, ppid=103, state=2, prio=30
00:00:06.24 kernel: queueIndex=2, pid=103, ppid=103, state=0, prio=30
00:00:06.25 kernel: SCHEDULER:: going to yield to USER task: Index=2, PID=103, PPID=103, state=0-------
00:00:06.26 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:06.27 kernel: queueIndex=0, pid=104, ppid=103, state=2, prio=30
00:00:06.28 kernel: queueIndex=1, pid=105, ppid=103, state=0, prio=30
00:00:06.29 kernel: queueIndex=2, pid=103, ppid=103, state=0, prio=30
00:00:06.30 kernel: SCHEDULER:: going to yield to USER task: Index=1, PID=105, PPID=103, state=0-------
00:00:06.31 kernel: TASK6: back CCC res = 0
00:00:06.32 kernel: TASK6: back DDD res = 0
00:00:06.33 kernel: TASK6: exit res = 0
00:00:06.34 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:06.35 kernel: queueIndex=0, pid=104, ppid=103, state=2, prio=30
00:00:06.36 kernel: queueIndex=2, pid=103, ppid=103, state=0, prio=30
00:00:06.37 kernel: SCHEDULER:: going to yield to USER task: Index=2, PID=103, PPID=103, state=0-------
00:00:06.38 kernel: TASK6: back CCC res = 105
00:00:06.39 kernel: TASK6: back DDD res = 105
00:00:06.40 kernel: now in kernel.wait 001
00:00:06.41 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:06.42 kernel: queueIndex=0, pid=104, ppid=103, state=2, prio=30
00:00:06.43 kernel: queueIndex=2, pid=103, ppid=103, state=2, prio=30
00:00:06.44 kernel: SCHEDULER:: no READY task in queue now, scheduler quite
00:00:06.45 kernel: SCHEDULER:: going out of scheduler
00:00:11.21 kernel: sleepTimerHandler is called
00:00:11.22 kernel: queueIndex=0, pid=104, ppid=103, state=0, prio=30
00:00:11.23 kernel: queueIndex=2, pid=103, ppid=103, state=2, prio=30
00:00:11.24 kernel: SCHEDULER:: going to yield to USER task: Index=0, PID=104, PPID=103, state=0-------
00:00:11.25 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:11.26 kernel: queueIndex=0, pid=104, ppid=103, state=0, prio=30
00:00:11.27 kernel: queueIndex=1, pid=106, ppid=104, state=0, prio=30
00:00:11.28 kernel: queueIndex=2, pid=103, ppid=103, state=2, prio=30
00:00:11.29 kernel: SCHEDULER:: going to yield to USER task: Index=0, PID=104, PPID=103, state=0-------
00:00:11.30 kernel: TASK6: back BBB res = 106
00:00:11.31 kernel: addKernelTimer is called
00:00:11.32 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:11.33 kernel: queueIndex=0, pid=104, ppid=103, state=2, prio=30
00:00:11.34 kernel: queueIndex=1, pid=106, ppid=104, state=0, prio=30
00:00:11.35 kernel: queueIndex=2, pid=103, ppid=103, state=2, prio=30
00:00:11.36 kernel: SCHEDULER:: going to yield to USER task: Index=1, PID=106, PPID=104, state=0-------
00:00:11.37 kernel: TASK6: back BBB res = 0
00:00:11.38 kernel: TASK6: back DDD res = 0
00:00:11.39 kernel: TASK6: exit res = 0
00:00:11.40 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:11.41 kernel: queueIndex=0, pid=104, ppid=103, state=2, prio=30
00:00:11.42 kernel: queueIndex=2, pid=103, ppid=103, state=2, prio=30
00:00:11.43 kernel: SCHEDULER:: no READY task in queue now, scheduler quite
00:00:11.44 kernel: SCHEDULER:: going out of scheduler
```



```
00:00:01.17 kernel: queueIndex=1, pid=102, ppid=102, state=0, prio=30
00:00:01.18 kernel: queueIndex=2, pid=103, ppid=103, state=0, prio=30
00:00:01.18 kernel: SCHEDULER:: going to yield to USER task: Index=1, PID=102, PPID=102, state=0-------
00:00:01.18 kernel: Task is entering now
00:00:01.18 kernel: TASK5:  Now we are in task5 .*****
00:00:01.19 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:01.19 kernel: queueIndex=2, pid=103, ppid=103, state=0, prio=30
00:00:01.19 kernel: SCHEDULER:: going to yield to USER task: Index=2, PID=103, PPID=103, state=0-------
00:00:01.20 kernel: Task is entering now
00:00:01.20 kernel: TASK6:  Now we are in task6  ************
00:00:01.20 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:01.20 kernel: queueIndex=0, pid=104, ppid=103, state=0, prio=30
00:00:01.21 kernel: queueIndex=2, pid=103, ppid=103, state=0, prio=30
00:00:01.21 kernel: SCHEDULER:: going to yield to USER task: Index=0, PID=104, PPID=103, state=0-------
00:00:01.21 kernel: TASK6: back AAA res = 0
00:00:01.21 kernel: addKernelTimer is called
00:00:01.22 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:01.22 kernel: queueIndex=0, pid=104, ppid=103, state=2, prio=30
00:00:01.22 kernel: queueIndex=2, pid=103, ppid=103, state=0, prio=30
00:00:01.22 kernel: SCHEDULER:: going to yield to USER task: Index=2, PID=103, PPID=103, state=0-------
00:00:01.23 kernel: TASK6: back AAA res = 104
00:00:01.23 kernel: addKernelTimer is called
00:00:01.23 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:01.24 kernel: queueIndex=0, pid=104, ppid=103, state=2, prio=30
00:00:01.24 kernel: queueIndex=2, pid=103, ppid=103, state=2, prio=30
00:00:01.24 kernel: SCHEDULER:: no READY task in queue now, scheduler quite
00:00:01.24 kernel: SCHEDULER:: going out of scheduler
00:00:04.15 kernel: From testTimerHandler
00:00:06.23 kernel: sleepTimerHandler is called
00:00:06.23 kernel: queueIndex=0, pid=104, ppid=103, state=2, prio=30
00:00:06.24 kernel: queueIndex=2, pid=103, ppid=103, state=0, prio=30
00:00:06.25 kernel: SCHEDULER:: going to yield to USER task: Index=2, PID=103, PPID=103, state=0-------
00:00:06.26 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:06.27 kernel: queueIndex=0, pid=104, ppid=103, state=2, prio=30
00:00:06.28 kernel: queueIndex=1, pid=105, ppid=103, state=0, prio=30
00:00:06.29 kernel: queueIndex=2, pid=103, ppid=103, state=0, prio=30
00:00:06.30 kernel: SCHEDULER:: going to yield to USER task: Index=1, PID=105, PPID=103, state=0-------
00:00:06.31 kernel: TASK6: back CCC res = 0
00:00:06.32 kernel: TASK6: back DDD res = 0
00:00:06.33 kernel: TASK6: exit res = 0
00:00:06.34 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:06.35 kernel: queueIndex=0, pid=104, ppid=103, state=2, prio=30
00:00:06.36 kernel: queueIndex=2, pid=103, ppid=103, state=0, prio=30
00:00:06.37 kernel: SCHEDULER:: going to yield to USER task: Index=2, PID=103, PPID=103, state=0-------
00:00:06.38 kernel: TASK6: back CCC res = 105
00:00:06.39 kernel: TASK6: back DDD res = 105
00:00:06.40 kernel: now in kernel.wait 001
00:00:06.41 kernel: SCHEDULER:: someone return control to scheduler        ++++++
00:00:06.42 kernel: queueIndex=0, pid=104, ppid=103, state=2, prio=30
00:00:06.43 kernel: queueIndex=2, pid=103, ppid=103, state=2, prio=30
00:00:06.44 kernel: SCHEDULER:: no READY task in queue now, scheduler quite
00:00:06.45 kernel: SCHEDULER:: going out of scheduler
```

Goal:

1. Understand process and thread concepts
2. Learn how to create process in Linux through, fork(), wait() and other related system calls; learn what happens when OS handles fork(), wait()
3. Implement fork(), wait() in pi-OS; manage process states and relationships.