ICS-OS Lab 03: Environment Variables, Processes, and Threads

Objectives

At the end of this activity, you should be able to:

1. understand how environment variables are implemented and used in ICS-OS;
2. edit and compile C programs inside ICS-OS;
3. understand how processes and threads are implemented in ICS-OS; and
4. use process-related commands and utilities provided by ICS-OS.

1 Introduction

A process is a program in execution. In this lab, you will explore the process-related services provided by ICS-OS.

2 Prerequisites

IMPORTANT: This lab should be done in an Ubuntu VM. Keys F11 and F12 need to be captured by ICS-OS but this cannot be done in Codespaces.

To proceed with this lab, you should have completed `Lab 02`. Most of the commands that we will use in this lab will be run relative to the `$ICSOS_HOME/ics-os` directory. Update your local copy of the source code and create a new branch for this lab with the commands below.

```
$cd $ICSOS_HOME/ics-os
$git checkout master
$git pull
$git checkout -b lab03
$git branch    #to check the currrent branch
```

Take note that the *top-level directory* of the source tree contains the following.

```
extras  ics-os  ics-os.gif  labs  README.md  wiki
```

3 Deliverables and Credit

Perform the tasks below and capture screen shots while you do them. Answer all questions. Questions by default relate to ICS-OS. Submit a PDF file containing the screen shots with captions and answers to questions. Do not forget to put your name and laboratory section. You may need to review ICS-OS Lab 01 and ICS-OS Lab 02. Total is **26 points**.

4 Tasks

Task 1: Environment Variables (3 points)

Processes do not run in isolation. In fact, most processes are created from a *parent* process. In multi-tasking/timesharing systems, several processes are in memory and are scheduled for execution on the

CPU. Some processes even communicate with other processes. Thus, processes exist in an *environment*. Operating systems provide *environment variables* to allow processes to obtain information about the environment. An example of an environment variable is `PATH`. This environment variable has a value that specifies where in the filesystem executables can be found. A shell process can then use this environment variable so that users need not specify the absolute path to executables when running programs.

The implementation of environment variables in ICS-OS is in `kernel/process/environment.h` and `kernel/process/environment.c`. Examine these files and answer the following questions.

QUESTIONS(Provide screen shots to support your answer):

1. What data structure is used for the implementation of environment variables?
2. Are environment variables unique for each process or shared by all processes?
3. What are the functions used to set and get an environment variable?
4. Examine `kernel/console/console.c`. What console commands use the functions in question 3?

Task 2: Edit and compile programs within ICS-OS (3 points)

In the previous lab, you built the `chown.exe` system utility outside of the ICS-OS environment. In this task, you will build programs within the ICS-OS environment.

Build and boot ICS-OS. Run the following command line sequence and capture a screen shot.

```
[1]% cd tcc1
[2]% ls -l -oname
[3]% cc
```

The `cc` console command allows you to compile C programs inside ICS-OS. Check its implementation in `kernel/console/console.c`. The command requires that `PATH` be set to where the executables are and `SDK_HOME` to where the SDK is located. Accomplish this using the command line sequence below.

```
[4]% set
[5]% set SDK_HOME=/icsos/tcc1
[6]% set PATH=/icsos/apps
[7]% set
```

Your next task is to write and compile the `envtest.c` program given below. Use `/icsos/apps/ed.exe`, a simple text editor for ICS-OS. Use `CTRL+o,ENTER` to save and `CTRL+x` to exit. Take note that this editor has a very limited set of features. Unfortunately, there is no copy and paste option.

```
[8]% pwd
[9]% ed.exe envtest.c
[10]% cc envtest.exe envtest.c
[11]% ls -l -oname
[12]% envtest.exe
[13]% set
```

```c
/*envtest.c*/
int main(){
```



```
    char val[30];
    printf("Testing environment variables.\n");
    getenv("PATH",val);
    printf("PATH is %s\n",val);
    setenv("PERICO_HEART","liza s.");
    setenv("BETEL_HEART","enrique g.");
    return 0;
}
```

ICS-OS has its own implementation of the C standard library functions in `sdk/tccsdk.c`. However the implementation is not complete thus you cannot use some functions you normally use in your C programs. You can find the definition of `setenv()` and `getenv()` there along with other functions.

QUESTIONS(Provide screen shots to support your answer):

1. Observe that you were able to run `ed.exe` in command line [9] without specifying its absolute path despite the current directory being `/icsos/tcc1`. Why is this so?
2. Command line [12] will not work. Why? Show your fix to be able to run `envtest.exe`.
3. After successfully running `envtest.exe`. What is the output of command line [13]?Does this support your answer in Q2 from Task 1?

Task 3: Processes

Task 3.1: Process Control Block (4 points)

A process is represented by a data structure called the *Process Control Block (PCB)*. In ICS-OS the PCB is a C structure (see `kernel/process/process.h`).

```
typedef struct _PCB386{
    //contains fields related to a process
}PCB386;
```

QUESTIONS(Provide screen shots to support your answer):

1. What field in the PCB describes the security bits for a process?
2. What field in the PCB describes the time the process arrived in the system?
3. What field in the PCB describes the memory information used by a process?
4. What field in the PCB describes the execution context(hardware specific) of a process?

Task 3.2: Startup Processes (5 points)

After GRUB, the ICS-OS boot process proceeds with the execution of a sequence of functions from the kernel as shown below.

1. `startup:`, *Source:*`kernel/startup/startup.asm`, *Purpose:* enable 32-bit protected mode, set up memory address linear selectors, calls `main()`
2. `main()`, *Source:*`kernel/kernel32.c`, *Purpose:* enable IRQ lines, set up default interrupt handlers, initialize memory management, set the context switch rate, calls `dex3_startup()`
3. `dex32_startup()`, *Source:*`kernel/kernel32.c`, *Purpose:* initialize different managers(extension, device, memory, process), calls the `taskswitcher()`

4. `taskswitcher()`, *Source:*`kernel/process/process.c`, *Purpose:* select the next process to execute.

When the task switcher kicks in, the kernel is now in full control and begins selecting processes and threads for execution. The task switcher is essentially the *Process scheduler*.

During boot, several processes have already been created and are running in the background, including the `console` where you are typing commands.

You might be wondering how these startup processes/threads got created and started. There are two functions to look at in the source: `process_init()` from `kernel/process/process.c` and `dex_init()` from `kernel/kernel32.c`. Study these two functions and answer the questions. You will observe that the PCBs of the startup processes are *hard-coded*, which means that the values of the fields are set at *compile time of the kernel*.

Now, run the command below.

```
% ps
```

QUESTIONS(Provide screen shots to support your answer):

1. How many processes and kernel threads (those with `(t)` in the name) in total are running?
2. What is the name of the process with PID 0?
3. What is the PID of `console(0)`?What is its access level?
4. What function is used to create the running kernel threads?
5. To what function is the EIP register assigned to in the PCB of the very first process?

Task 3.3: Consoles (2 points)

Create a new console using the command below. You can move across consoles using `F11` and `F12`.

```
% newconsole
```

QUESTIONS(Provide screen shots to support your answer):

1. Using `ps`, what is the name and PID of the new console?What is the name and PID of its parent process?Is the new console a process or a thread?
2. Study the implementation of the `newconsole` command in the `kernel/console/console.c`. What function is used to create a new console?

Task 3.4: User Processes (2 points)

Edit and compile `count.c` within ICS-OS but DO NOT RUN yet. Copy the executable to `/icsos/apps` folder. Do not forget to set the needed environment variables described above.

```
% ed.exe count.c
% cc count.exe count.c
% copy count.exe /icsos/apps
```

```
1 /*count.c*/
2 int main(){
```



```
3     int i;
4     while(1){
5       printf("%d\n",(i=(i+1)%10));
6       sleep(10);
7     }
8   }
```

Create a new console then run `count.exe` on the new console. Try to press `CTRL+c`. You will observe that the process does not stop. Signals are not yet implemented.

```
% newconsole
% count.exe
```

Go back to the previous console using `F11` then run `ps`.

QUESTIONS(Provide screen shots to support your answer):

1. What is the PID of `count.exe` process?What is its access level?How much memory does it use?What is its parent process?

In order to terminate the `count.exe` process, use the `kill` console command with the process id or complete process name as argument.

```
% kill count.exe
```

Task 3.5: Process Creation (5 points)

How did the executable `count.exe` program eventually became a process?This is explained below.

When the user enters a command line, the console first checks if it is an internal command. If it is not, then the command line is treated as an executable, as in the case of `envtest.exe` and `count.exe`. If the executable is not found then a `"Command or executable not found."` message is displayed. If found, a process will be created and started.

Creating and running processes from user programs is implemented in the `user_execp()` function (see `kernel/console/console.c`). In this function, the contents of the executable (aka *program image*) is read. Then, the function `addmodule()` (see `kernel/process/pdispatch.c`) is called. This function creates a `createp_queue` node (see `kernel/process/pdispatch.h`). The new node type is set to `NEW_MODULE`, which means that the process that will be created will not be a copy of the parent process. The new node is then added to the global `pd_head` queue. The `process_dispatcher()` (see `kernel/process/pdispatch.c`) continuously inspects `pd_head` queue and when it finds a node with the `dispatched` field set to 0, it will create a new process using `createprocess()`. Note that the `process_dispatcher()` was called in the very first process(pid=0).

There are two low-level functions to create processes (see `kernel/process/process.c`): `forkprocess()` and `createprocess()`. The `forkprocess()` approach creates a duplicate of the PCB of the parent process while the `createprocess()` approach creates the PCB from scratch. In both approaches, after the PCB has been created, it is added to the ready queue using `ps_enqueue()`. At this point, a process has been *dispatched* and can now be scheduled by the CPU scheduler for execution.

As mentioned above, the instructions to be executed by the process will come from the program image, which can be in different formats. In ICS-OS, the following formats are supported: ELF32, PE,

COFF, B32. A *module loader* for each format is implemented and can be found in the `kernel/module/` folder. For example, the module to load linux executables is in `kernel/module/elf_module.c`. These loaders eventually call `createprocess()` after all the information needed to create the PCB have been obtained from the program image. The function `dex32_loader()` in `kernel/module/module.c` calls the appropriate loader for the program image.

For application programming, you can use the `execp()` function in the SDK which invokes `user_execp()` through a system call. It returns 0 if the call failed or the process id if successful.

```
int execp(char *fname, unsigned short mode, char *params);
```

The code below is a template for a basic shell. Edit, compile, and run `meshell.exe` inside ICS-OS. Show some screenshots of it in action.

```
/*meshell.c*/
int main() {
   char cmdline[255]="";
   char params[255];
   char *cmd;
   printf("MeShell v1.0\n");
   printf("Type 'exit' to end session.\n");
   while (strcmp(cmdline,"exit")!=0){
      printf("$");
      gets(cmdline);
      strcpy(params,cmdline);
      cmd=strtok(cmdline," ");
      if (cmd != 0){
         if(!execp(cmd, 0, params))
            printf("Executable not found.\n");
      }
   }
   return 0;
}

```

QUESTIONS(Provide screen shots to support your answer):

1. Use `hxdmp.exe` to determine the format of some of the executables in the `apps` folder. If the first few bytes has `MZ` then it is a windows executable, if `ELF` then it is a linux executable. What is the executable format of `count.exe`? `ed.exe`? `tcc.exe`? `nasm.exe`? `meshell.exe`?
2. Which line in the `forkprocess()` and `createprocess()` functions initializes the PCB of the new process. How do the functions differ?

The `forkprocess()` method of creating processes can be used through the `fork()` function in the SDK. This function invokes `user_fork()` through a system call. Compile `fork.c` below. Create a new console and run `meshell.exe` on the new console. Run `fork.exe` in meshell. Switch to the previous console and run `ps`. Observe the list of processes and capture some screenshots.

```
% cd /icsos/apps
% newconsole
```



```
% meshell.exe
$ fork.exe
```

```c
/*fork.c*/
int NITER=10000;
int main() {
   int retval,i;
   printf("Calling fork()..\n");
   retval = fork();
   if (retval == 0){
      printf("In child. mypid:%d\n",getpid());
      for (i=0;i<NITER;i++){
         printf("Child: %d\n",i);
         sleep(10);
      }
      exit(0);
   }
   else if (retval > 0){
      printf("In parent, mypid:%d, child pid: %d\n",getpid(), retval);
      for (i=0;i<NITER;i++){
         printf("Parent: %d\n",i);
         sleep(15);
      }
      exit(0);
   }
   return -1;
}
```

Task 3.6: Process Termination (2 points)

Terminating a process means freeing the resources allocated to it, such as memory and other data structures, as well as the PCB itself. As mentioned above, the `kill` console command (see `kernel/console/console.c`) is used to terminate a process. It calls the `dex32_killkthread_name()` function (see `kernel/process/process.c`). This function sets the variable `sigterm` to the process id of the process to kill. The `taskswitcher()`, when in control, then checks if `sigterm` is zero, if not, it calls `kill_process(sigterm)`. The actual work of freeing the resources and terminating the process is done in the `kill_process()` function (see `kernel/process/process.c`).

QUESTIONS(Provide screen shots to support your answer):
Study the `kill_process()` function.

1. What function is called to kill a kernel process/thread?
2. What function is called to kill a user thread?

5 Reflection

Write some realizations and questions that crossed your mind while doing this lab.

