CS 202: Advanced Operating Systems
University of California, Riverside

Lab #1: System Call Implementation

In this assignment, you will add two new system calls that provide the information of the system and the target process, respectively.

Part 1: sysinfo

Add a new system call `int sysinfo(int param)` that takes as input one integer parameter of value 1, 2 or 3. Depending on the input value, it returns:

- If `param == 0`: the total number of active processes (ready, running, waiting, or zombie) in the system.
- If `param == 1`: the total number of system calls that has made so far since the system boot up. Do not include the current `sysinfo` syscall's attempt when returning the number.
  - Example: Suppose the system has made 5 syscalls so far. `sysinfo` should return 5 as output, instead of 6. If `sysinfo` is called again right after this, then it should return 6.
- If `param == 2`: the number of **free memory pages** in the system. If there is one free page left, the return value should be 1 (= 1 page).
- Otherwise: return error (-1)

Part 2: procinfo

Add a new system call `int procinfo(struct pinfo *in)` that provides information specific to the current process (= caller process of this syscall). It takes as input a pointer of `struct pinfo` and fills out the fields of this struct:

```
struct pinfo {
  int ppid;
  int syscall_count;
  int page_usage;
};
```

- `int ppid`: the PID of its **parent** process.
- `int syscall_count`: the total number of system calls that the **current process** has made so far. Do not include the current `sysinfo` syscall's attempt when returning the number.
- `int page_usage`: the current process's memory size in pages (e.g., 10000 bytes → 3 pages).

`procinfo()` returns 0 on success; -1 otherwise (e.g., NULL input pointer, fail to copy results back to the caller process, etc.)

Tips:

- To count the number of processes, see `kernel/proc.c` and `kernel/proc.h`
- To check the system call count, you will need to modify `kernel/syscall.c`
- Even if `sysinfo()` is the very first syscall you make in your user-level program, you will notice that a couple of other syscalls have been already issued. This is normal; recall how a process is created (e.g., fork, exec, etc).
- `printf` uses syscalls to display output. So you may see an unexpected increase in syscall count when printf is inserted in-between `sysinfo` or `procinfo`.
- To check the amount of free memory pages in the system, see `kernel/kalloc.c`. Xv6 manages free pages in a linked list (`kmem.freelist`).
- To retrieve a pointer type input in a syscall, use `argaddr()`. The kernel cannot directly write data to the userspace memory; you will need to use `copyout()` to do so. Search the kernel code to find how to use copyout (e.g., kernel/file.c or sysfile.c).
- The kernel function `myproc()` returns the PCB of the current process.
- To implement Part 2, you need to manage syscall count on a per-process basis. Initialize new per-process data fields in `allocproc()`.
- You can find the amount of memory used by each process from its PCB.
- To check if your syscall is reporting the correct number of free or used memory, allocate memory in your user-level program before making the system call.

How to test:

To test your implementation, you can create a user-level program using these syscalls under various scenarios. Below is just an example (`user/lab1_test.c`):

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

#define MAX_PROC 10
struct pinfo {
  int ppid;
  int syscall_count;
  int page_usage;
};
void print_sysinfo(void)
{
  int n_active_proc, n_syscalls, n_free_pages;
  n_active_proc = sysinfo(0);
  n_syscalls = sysinfo(1);
  n_free_pages = sysinfo(2);
  printf("[sysinfo] active proc: %d, syscalls: %d, free pages: %d\n",
    n_active_proc, n_syscalls, n_free_pages);
}
int main(int argc, char *argv[])
{
  int mem, n_proc, ret, proc_pid[MAX_PROC];
  if (argc < 3) {
    printf("Usage: %s [MEM] [N_PROC]\n", argv[0]);
    exit(-1);
  }
  mem = atoi(argv[1]);
  n_proc = atoi(argv[2]);
  if (n_proc > MAX_PROC) {
    printf("Cannot test with more than %d processes\n", MAX_PROC);
    exit(-1);
  }
  print_sysinfo();
  for (int i = 0; i < n_proc; i++) {
    sleep(1);
    ret = fork();
    if (ret == 0) { // child process
      struct pinfo param;
      malloc(mem); // this triggers a syscall
      for (int j = 0; j < 10; j++)
        procinfo(&param); // calls 10 times
      printf("[procinfo %d] ppid: %d, syscalls: %d, page usage: %d\n",
        getpid(), param.ppid, param.syscall_count, param.page_usage);
      while (1);
    }
    else { // parent
      proc_pid[i] = ret;
      continue;
    }
  }
  sleep(1);
  print_sysinfo();
  for (int i = 0; i < n_proc; i++) kill(proc_pid[i]);
  exit(0);
}
```

This user program calls `sysinfo()` and creates N child processes, each of which allocates a specified amount of memory and calls `procinfo()`. Once all child processes print `procinfo()`, the parent process calls `sysinfo()` again.

For example, if you want to test with 2 child processes with each allocating 65Kbytes of memory, type:

```
$ lab2_test 65536 2
```

(the first argument is the bytes of memory to be allocated by each child process, and the second argument is the number of child processes to be created).

Running example (you may need to press 'enter' or any key after each line is printed):

```
$ lab1_test 65536 2
[sysinfo] active proc: 3, syscalls: 49, free pages: 32564
[procinfo 4] ppid: 3, syscalls: 10, page usage: 21
[procinfo 5] ppid: 3, syscalls: 10, page usage: 21
[sysinfo] active proc: 5, syscalls: 241, free pages: 32510
$ $ $ $ lab1_test 65000 1
[sysinfo] active proc: 3, syscalls: 345, free pages: 32564
[procinfo 10] ppid: 9, syscalls: 10, page usage: 20
[sysinfo] active proc: 4, syscalls: 474, free pages: 32538
$ $ $
```

What to submit:

You need to submit the following:

(1) The **entire XV6 source code** with your modifications ('make clean' to reduce the size before submission)

(2) A **report** (must be in **PDF**; no other formats will be accepted) including:

a) The list of all files modified

b) A detailed explanation on what changes you have made and screenshots showing your work and results

c) A detailed description of XV6 source code (including your modifications) about how the info system call is processed, from the user-level program into the kernel code, and then back into the user-level program.

d) Link to the demo video on the very first page of the report.

e) A brief summary of the contributions of each member

(3) A **demo video** showing that all the functionalities you implemented works as expected, as if you were demonstrating your work in-person. For instructions about recording, refer to the *Demo Video Recording Guide*.

How to submit:

Upload the demo video to your own Google Drive (or YouTube) and create a shareable link. Add this link to the first page of the report. Pack the report and source code files into a **single zip file**, and upload it to eLearn/Canvas.

Only one person of your team needs to submit the file.

Grades breakdown:

- sysinfo() system call: 25 pts
- procinfo() system call: 25 pts
    - Correct implementation and functionalities of syscalls; correct return values
- Report: 10 pts
    - Clear and detailed explanation of the changes made
- Demo video: 10 pts

Total: 70 pts

Academic Integrity:

- Your submission should be **strictly** of your own.
- All submissions will be monitored for similarities with current and previous terms' submissions.
- Any similarities found will be escalated as per Academic Misconduct policy: http://conduct.ucr.edu/policies/academicintegrity.html