# Project 1B
# COM S 352
# Spring 2022

## 1. Introduction

For this project iteration, you will prepare system calls and utilities that will be used to implement and evaluate a new scheduler.

## 2. Nice system call (15 points)

Implement a system call that sets the nice value for the calling process. A process should be able to call the following function to initiate the system call.

```
int nice(int nicevalue);
```

> Sets the nice value for the process that called the function.
> `nicevalue` – an integer between -20 and 19 inclusive
> `returns` – 0 if successful, -1 on error

The system call is similar to the one implemented in Project 1A. The following questions and answers provide additional details.

**Where to store the nice value?**
The information for each process is stored in the array `proc[]`. That makes it a good place to store the nice value for each process. You should modify the struct of this array to store the nice value. The struct is defined in the file `kernel/proc.h`.

**Where to add the system call function on the kernel side?**
A logical location is `kernel/sysproc.c`, it contains the system calls used for controlling processes.

**How can an argument (e.g., `nicevalue`) be passed in a system call?**
A system call is called indirectly through an interrupt handler, for this reason the function that implements a system call cannot have parameters. For example, notice that all of the system call functions in `kernel/sysproc.c` have void parameters. To receive arguments from the caller, the system call must manually inspect the call stack. For example, helper function `argint()` exists to get the value of an integer. Look at `sys_sleep()` to learn how `argint()` is used. The `sleep()` system call takes a single integer as an argument.

**How does the system call know which process called it?**
The function `myproc()` returns the proc struct (from the array `proc[]`) for the calling process.

**What should be the default nice value when a process is created?**
The default nice value is 10. Find an appropriate place (e.g., where a new process is allocated) in `kernel/proc.c` to set the default nice value.

**What to do if the user provides a nicevalue that is not between -20 and 19 inclusive?**
Do not update the value of nice for the process and return -1.

## 3. Nice command (15 points)

Implement a nice utility program in a file named `user/nice.c`.

From the command line, a user should be able to start a program and set its nice value like this:
`$ nice 19 wc README`

In the example, the command "`nice 19 wc README`" executes the `nice` utility program (the one you will implement), which in turn sets the nice value to 19 and then executes the command "`wc README`". The command "`wc README`" executes the `wc` (word count) utility program passing it the argument `README`.

The specification for the `nice` command is the following.

**`nice`** *`N PROG [ARG]...`*

> `N` – the nice value
> `PROG` – the program to execute
> `[ARG]…` – zero to many arguments that are passed to the program to execute

**How to get and parse the parts of the nice command?**
Command line arguments are available in the char array `argv[]` passed into `main(int argc, char *argv[])`. For example, the command:

```
$ nice 19 wc README
```

results in main's `argv[]` looking like this:

```
argv[0] = "nice"
argv[1] = "19"
argv[2] = "wc"
argv[3] = "README"
argv[4] = 0 // the array ends with a null string
```

To convert a string into an integer use the function `atoi()`. The best way to learn how to write systems code is by looking at examples. In this case, two good examples are `user/wc.c` and `user/kill.c`.

**How can `nice` start another program?**
A process can change the program it is executing by calling `exec()`. The xv6 implement of `exec()` has the following definition.

```
int exec(char* pathname, char* argv[]);
```

The first parameter is the name of the command to execute, for example "`wc`". The second parameter takes an array of strings that must end with the last string being only a null character. Notice the similarity between the parameters of `exec()` and the arguments passed to `main(char* pathname, char* argv[])`. Remember that you can get the address of something with the symbol `&` and pointers can be treated as arrays. Using the example from the previous question `char **arr = (char **)argv[2]` produces:

```
arr[0] = "wc" // the first element should be the program name
arr[1] = "README"
arr[2] = 0 // the array ends with a null
```

## 3. Process info system call (10 points)

Implement a system call to collected detailed information about processes on the system. A process should be able to call the following function to initiate the system call.

```
int getpstat(struct pstat*);
```

> The `pstat` object passed in the function is modified by the kernel to contain information about processes on the system.
> `pstat` – a pointer to a pstat object which will be updated by the kernel
> `returns` – 0 on success, -1 on error

The `pstat` struct is defined as follows (put this in a file `kernel/pstat.h`).

```
/**
 * Adapted from https://github.com/remzi-arpacidusseau/ostep-
projects/tree/master/scheduling-xv6-lottery
 **/
#ifndef _PSTAT_H_
#define _PSTAT_H_

#include "param.h"

struct pstat {
  int inuse[NPROC]; // whether this slot of the process table is in use (1 or 0)
  int pid[NPROC];   // the PID of each process
  int nice[NPROC]; // the nice value of the process
  // More will be added in Project 1C...
};

#endif // _PSTAT_H_
```

**Where to add the system call function on the kernel side?**
A logical location is `kernel/sysproc.c`, it contains the system calls used for controlling processes.

**How to get access to the array `proc[]` from `kernel/sysproc.c`?**
To get access to the process table from any file in the kernel, add the following as the very last line of `kernel/proc.h`:

```
extern struct proc proc[NPROC];
```

**How to take `struct *pstat` as an argument to a system call on the kernel side?**
For the `nice()` systems call described in Section 2, we saw that taking an int as an argument can be done with `argint()`. A pointer to a struct is a little more difficult because the process memory may not be contiguous. The best strategy is to create an identical struct in kernel memory and then use the helper function `copyout()` to copy the kernel struct to the user struct. You are provided with this solution below, only the TODO needs to be completed.

```
uint64 sys_getpstat(void) {
    uint64 result = 0;
    struct proc *p = myproc();
    uint64 upstat; // the virtual (user) address of the passed argument struct pstat
    struct pstat kpstat; // a struct pstat in kernel memory

    // get the system call argument passed by the user program
    if (argaddr(0, &upstat) < 0)
        return -1;


    // TODO: fill the arrays in kpstat (see the definition of struct pstat above).
    // The data to fill in the arrays comes from the process table array proc[].


    // copy pstat from kernel memory to user memory
    if (copyout(p->pagetable, upstat, (char *)&kpstat, sizeof(kpstat)) < 0)
        return -1;

    return result;
}
```

**How to test `getpstat()`?**
In a user program create a `struct pstat` and pass it by pointer when calling `getpstat()`.

```
struct pstat stats;
getpstat(&stats);
// print the arrays in stats
```

## 4. Documentation (10 points)

Documentation is required for the project. Every location that you add/modify code in the kernel must have a comment explain the purpose of the change.

The user programs you write must also have brief comments on their purpose and usage.

Include a `README` file with your names, a brief description, and a list of all files added to the project.

## 5. Submission

Make sure the code compiles, focus on working code over having every feature complete. We will look at the code for partial credit. Document anything that is incomplete in the `README` file.

Submit a zip file of the xv6-riscv directory. On pyrite the zip file can be created using:

```
$ zip -r project-1b-xv6-riscv.zip xv6-riscv
```

Submit `project-1b-xv6-riscv.zip`.