*The original source of this document and project is Prof. David Chiu. They have been modified for our class.

Project 1: Ready Queue

Due: 24/02/2020 at 11:59pm
Instructor: Hicham Elmongui | Spring 2020.

Disclaimer

*The instructor is aware that students already have the solution for this project, and that they might have already gone through it. By continuing reading this page, students understand that, in order to **gain** the intended experience, they will cease looking at the solution provided in the stock Xinu codebase. Gaining the experience takes time, effort, and possibly frustration. That experience is crucial to be able to continue with the subsequent projects, which students would not have their solutions.*

Overview

Xinu is an operating system developed by Prof. Douglas Comer's group at Purdue University. Xinu is used in an impressive number of real computer systems (*e.g.*, embedded controllers and even an IBM mainframe computer, among others). The description of Xinu from its website:

> *"XINU stands for Xinu Is Not Unix -- although it shares concepts and even names with Unix, the internal design differs completely. Xinu is a small, elegant operating system that supports dynamic process creation, dynamic memory allocation, network communication, local and remote file systems, a shell, and device-independent I/O functions. The small size makes Xinu suitable for embedded environments."*

In this project, you will be implementing an essential data structure, which pervades most OS kernels including Xinu: a (dynamically allocated) queue of processes, known as the Ready Queue. It stores pointers to process control blocks (called "process entries" in Xinu), providing a set of processes for the CPU scheduler to choose from for execution.

Objectives

- To become familiar with the Xinu development and runtime environment
- To become familiar with the Xinu kernel codebase
- To provide more experience with pointers and dynamically-allocated structures.

Part 1: Installing and Running Xinu

In this section, we'll get Xinu up and running on a virtual machine.

1. Download the two appliances that constitutes the VirtualBox version of Xinu:
   - xinu-vbox-appliances.tar.gz (3.33 GB)
2. Extract from xinu-vbox-appliances.tar.gz, the two appliances:
   - The development system appliance: development-system.ova (3.38 GB)
   - The back-end appliance: backend.ova (8 KB)
3. Open up VirtualBox. Then click on the `File > Import Appliance...` menu. Browse and find the `development-system.ova` file you just extracted, then click `Next`. You may change RAM to be 1024MB. Do *not* change any other settings. , then click `Import`.
4. Highlight `development-system` in the left-hand menu, and click on Settings. From the Settings menu, navigate to Ports. Make sure that `Enable Serial Port` is **checked** and `Connect to Existing Pipe/Socket` is ***un*checked**. Under Path/adress:
   - If you're on a Linux or Mac: type in `/tmp/xinu_serial`
   - If you're on Windows: type in `\\.\pipe\xinu_com1`

   Click OK to exit Settings.

5. Open up VirtualBox. Then click on the `File > Import Appliance...` menu. Browse and find the `backend.ova` file you extracted, then click `Next`. You may change RAM to be 16MB. Do *not* change any other settings. , then click `Import`.

6. Highlight `backend` in the left-hand menu, and click on Settings. From the Settings menu, navigate to Ports. Make sure that `Enable Serial Port` and `Connect to Existing Pipe/Socket` are **both checked**. Under Path/adress:

   - If you're on a Linux or Mac: type in `/tmp/xinu_serial`
   - If you're on Windows: type in `\\.\pipe\xinu_com1`

   Click OK to exit Settings.

7. Double-click on `development-system` VM that you will be using to do your assignments. Use the following credentials:

   - login: xinu
   - password: xinurocks
   - Fix minicom (only once) to make it run with VirtualBox. Open the **Terminal**, and do the following:
     - Run the command `sudo minicom --color=on`
     - Type `ctrl+a`
     - Type `o`
     - Navigate to `'Serial Port Setup'`, and hit `Enter`
     - Type `f` to toggle the Hardware Flow Control off
     - hit `Enter`
     - Select `'Save setup as dfl'`, and hit `Enter`
     - Select `'Exit'`, and hit `Enter`
     - Power off the 'development-system' VM by running the command `sudo shutdown -P now`
     - Restart VirtualBox completely, and it should be fixed.

8. Open the `development-system` and open the **Terminal**. Let's download the first project from the course page:

```
wget http://eng.staff.alexu.edu.eg/~elmongui/course_materials/2020/spring/os2/projects/proj1.tar.gz
```

9. The `proj1.tar.gz` file is a zip archive. Now we need to expand it:

```
tar -zxvf proj1.tar.gz
```

   You should see a series of files being extracted, and when it's done, you will get a new directory named `proj1`, where all those files were extracted into.

10. Navigate into the `proj1` directory, and you'll see the following subdirectories:

   - `compile/` - contains the `Makefile` and scripts to upload the kernel to the back-end.
   - `config/` - contains device configurations (do not touch files in this directory).
   - `device/` - contains device files (do not touch files in this directory).
   - `include/` - contains header files, which define constants and function prototypes.
   - `lib/` - contains a small library of standard C functions. The UNIX system libraries are not available.
   - `system/` - contains the source code for the Xinu kernel.

   Most of your time in development will be spent in the `include/` and `system/` directories.

Part 2: Compiling and Running Xinu

1. Navigate into the `compile/` directory. You can type `make` to compile the kernel, but you'll soon be inundated with compile errors. This is because there are several important functions that you need to implement for this project. Let's clean up the mess by running `make clean`.

2. Normally, if the compilation was successful, it would create a binary file called `xinu` in this directory. You would then need to run `./upload.sh` to prepare it for upload onto the back-end VM.

3. Don't worry, you are provided with a precompiled solution called `xinuSol`, so let's run that for now so you can see what to expect for this assignment. Run `./uploadSol.sh` to upload the precompiled kernel to the back-end VM.

4. Now type `sudo minicom`. You'll be prompted for the password. This turns the terminal window into a serial console that is connected to the back-end VM, effectively emulating a terminal for the back-end VM. *Speaking of the back-end...*

5. At this point, start the back-end virtual machine from VirtualBox. It should take a few seconds for it to automatically retrieve the kernel binary from development-system and boot it. Because `minicom` turned the development-system VM into the screen that's "attached" to the back-end machine, you can see Xinu boot up and run there. If everything went smoothly, you should get this output:

```
Hello XINU WORLD!
This is process 2
This is process 2
This is process 2
This is process 2
This is process 2
This is process 2
This is process 2
This is process 2
This is process 2
This is process 2
Hello XINU WORLD!
This is process 3
This is process 3
This is process 3
This is process 3
This is process 3
This is process 3
This is process 3
This is process 3
This is process 3
This is process 3
Hello XINU WORLD!
This is process 4
This is process 4
This is process 4
This is process 4
This is process 4
This is process 4
This is process 4
This is process 4
This is process 4
This is process 4
Hello XINU WORLD!
This is process 5
This is process 5
This is process 5
This is process 5
This is process 5
This is process 5
This is process 5
This is process 5
This is process 5
This is process 5
1
2
3

Hello XINU WORLD!
This is process 6
This is process 6
This is process 6
This is process 6
This is process 6
This is process 6
This is process 6
This is process 6
This is process 6
This is process 6
10
20
30
40
50
60



All user processes have completed.
```

Afterwards, Xinu is still "running," over on the back-end but it's in an infinite loop called the null-process, and not accepting any other commands (there's no shell). We'll see what this output means later.

6. To exit `minicom`, press `ctrl-a` followed by pressing `q`. This brings the terminal back on the development-system.

7. Shutdown the back-end VM to terminate Xinu.

8. From here on, remember this workflow as you proceed with development:

    - Write your code on development-system
    - Navigate into the `compile/` subdirectory
    - `make clean`
    - `make` to compile the kernel
    - `./upload.sh` to upload the kernel
    - `sudo minicom`
    - Start up the back-end VM

Part 3: Quick Tour of Xinu Structures and Types

You need to spend some time exploring Xinu's codebase, specifically, the files in `include/` and `system/`. I don't expect you to understand everything, but it would be to your benefit to obtain even a high-level understanding of the kernel's structures. I will point out a few important items to spend some time on:

Types and Constants

- `include/xinu.h`: unifies the inclusion of all necessary header files. This makes it convenient for when we're developing; we only have to place a single line `#include <xinu.h>` at the top of our files to gain access to all constants and functions.

- `include/prototypes.h`: most system-call prototypes are declared here, but implemented elsewhere in `.c` files.

- `include/kernel.h`: contains definition of some important constants, `typedef`s, and function prototypes.

  Types: data types in Xinu are renamed to be more convenient and specific. Notably, you'll see these often:

Name in Xinu	Really just a...
`byte`	8-bit char
`bool8`	byte
`int32`	32-bit int
`uint32`	32-bit unsigned int
`pid32`	int32 (a process ID)
`sid32`	int32 (a semaphore ID)
`status`	int32 (return status of system call: `OK`, `SYSERR`, `TIMEOUT` see below)

  Constants: the list of typedefs are followed by constants. You should commit these to memory, but here are some important ones.

Name of Constant	Value (Description)
`NULL`	`0` (this is the `NULL` you worked with for pointers)
`OK`	`1` (used as normal return-status for a system call)
`SYSERR`	`-1` (used as error return-status for a system call)
`EOF`	`-2` (end of file)
`TIMEOUT`	`-3` (used as the timeout return-status for a system call)

For Debugging

- `system/kprintf.c`: defines `kprintf()`, which is a system call you can use to ask the the kernel to print something to the console. Used just like regular `printf()`.

Process Structures

Xinu processes have a unique identifier of type `pid32`, and are defined by a process control block (PCB) structure, called `struct procent` (process entry).

- `include/process.h`: defines constants (such as process states) and structures (like the PCB) relating to the process. Read through this file and be able to answer the questions below. **Know this:** Toward the end of the file, there are three important global variables (accessible from anywhere) that pertains to processes in Xinu:

`pid32 currpid`	The PID of the currently-running process.
`int32 prcount`	The number of processes in the system.
`struct procent proctab[]`	The process table. An array of PCBs, indexed by PID.

- `system/create.c`: defines the system call to create a new process.
- `system/kill.c`: defines the system call to kill a process.
- `system/ready.c`: defines the system call to put process on the ready queue.
- `system/resched.c`: pulls the next process off of the ready queue, and schedules it for CPU execution.
- `system/suspend.c` and `system/resume.c`: defines the system call to suspend and resume a process.
- `system/yield.c`: defines the system call to cause the currently running process to voluntarily yield the CPU to next process on the ready queue.
-

Answer these questions (not graded):

- What is the maximum number of processes accepted by Xinu? Where is it defined?
- What does Xinu define as a "illegal PID"? Find and check out the `isbadpid()` inline function. (Aside: What's an inline function in C?)
- Name all the states in which a Xinu process can be.
- What is the default stack size Xinu assigns each process? Where is it defined? (Recall from the previous assignment that this is called `RLIMIT_STACK` in Linux)

Part 4: How Does Xinu Run (and What Is the Null Process)?

When Xinu boots up, the `nulluser()` system call is invoked by the bootstrap, which in turn invokes `sysinit()`. To see what they do, open up this file: `system/initialize.c`.

- Inside this file, scroll down to take a look at the `sysinit()` function.
  - This function starts out with some code to initialize interrupt vectors and a free-list for memory allocation. About half-way down, it initializes the PCB table:

```
 1 static  void    sysinit(void)
 2 {
 3     //(code omitted)
 4
 5     /* Initialize process table entries free */
 6     for (i = 0; i < NPROC; i++) {
 7         prptr = &proctab[i];
 8         prptr->prstate = PR_FREE;
 9         prptr->prname[0] = NULLCH;
10         prptr->prstkbase = NULL;
11     }
12
13     //(code omitted)
14 }
```

  - **On Lines 6-11:** all entries in the PCB table are initialized. Initially, every slot is free (i.e., in the `PR_FREE` state), the name of each process, `prname` is an empty string, and the stack base pointer `prstkbase` refers to `NULL`.
  - The loop implies that there can only be `NPROC` processes existing in the table, and that the array index from 0 to `NPROC-1` serves as the process ID.

Now find the `nulluser()` function, which is the *first process* (known as Xinu's **null process**) that Xinu runs after it boots.

- The first thing it does is invoke `sysinit()`, which sets up important data structures (see above).
- The code will then print out address-space information (sizes of text, data, heap segments) and enable interrupts.
- Just after the interrupts are enabled, we see the creation of a process named `MAIN1`, running a function called `main()` with various arguments.

```
 1 void    nulluser(void)
 2 {
 3     sysinit();
 4 
 5     //(code omitted)
 6 
 7     //spawn a process running main() from main.c
 8     ready(create((void*) main, INITSTK, "MAIN1", 2, 0, NULL), FALSE);
 9 
10     //schedule the above process
11     while (TRUE)
12     {
13         if (nonempty(readyqueue))
14         {
15             //everytime resched() is called, it pulls the next process off the ready queue
16             resched();
17         }
18     }
19 }
```

- **On Line 8:** A call to `create()` will create a process in Xinu. Remember from your exploration earlier that this function takes as argument:

   1. A function pointer to the code that the process will run (`main()`)
   2. The stack size in words (`INITSTK`)
   3. A name for the process (`MAIN1`)
   4. The number of arguments given to the function referred to (`2` here because `main()` takes two arguments)
   5. List of arguments given to the function referred (`0` followed by `NULL`)

   The `create()` function returns the PID that was assigned to this new process, which is in turn input into `ready()`.

   The `ready()` function inputs two arguments. The first is the PID. It will place the given PID on the ready queue. The second argument, `FALSE`, tells Xinu that it should not run the scheduler another process for execution after this process was introduced to the ready queue.

- **On Line 11-18:** This infinite loop examines the ready queue, and schedules the next process as long as it is non-empty. Notice how, when there are no processes to execute, this portion of the code essentially turns into an infinite loop, waiting for the next process to enter the ready queue.

   This gives us a clue as to what this **null process** actually is: when there's no other processes for the CPU to execute, *something* has to run, to keep the kernel active.

Part 5: The main() Function

Okay, so the `nulluser()` function created five processes, all executing a function called `main()` with various arguments.

- Let's take a look at this function, which is defined in `system/main.c`:

```
#include <xinu.h>
#include <stdio.h>

void    printpid()
{
    int i;
    kprintf("Hello XINU WORLD!\r\n");

    for (i=0; i<10; i++)
    {
        kprintf("This is process %d\r\n", currpid);

        //uncomment the line below to see cooperative scheduling
        //resched();
    }
16 }
17
18 void    printargs(uint32 argc, uint32 *argv)
19 {
20     printpid();
21
22     int i;
23     if (argc > 0)
24     {
25         for (i=0; i<argc; i++)
26             kprintf("%d\n", argv[i]);
27         kprintf("\r\n");
28     }
29 }
30
31 int main(uint32 argc, uint32 *argv)
32 {
33     static uint32 main2args[] = {1, 2, 3};
34     static uint32 main3args[] = {10, 20, 30, 40, 50, 60};
35
36     // Create 5 processes
37     ready(create((void*) printpid, INITSTK, "MAIN1", 2, 0, NULL), FALSE);
38     ready(create((void*) printpid, INITSTK, "MAIN2", 2, 0, NULL), FALSE);
39     ready(create((void*) printpid, INITSTK, "MAIN3", 2, 0, NULL), FALSE);
40     ready(create((void*) printargs, INITSTK, "MAIN4", 2, 3, main2args), FALSE);
41     ready(create((void*) printargs, INITSTK, "MAIN5", 2, 6, main3args), FALSE);
42
43     return 0;
44 }
```

- **On Lines 4-16:** the function `printpid()` is defined. It loops and prints out the PID ten times. Remember that `currpid` is a global variable that *always* holds the currently running PID.
- **On Lines 18-29**: the function `printargs()` is defined. It takes two arguments as input. They willw be passed in from `main()`. This function loops through each argument and prints them out to the console.
- **On Line 31**: Notice that `main()` takes as input two arguments:
  - `uint32 argc`: the number of arguments in `argv`
  - `uint32 *argv`: a pointer to the input arguments

  These were passed in from `nulluser()` as `0` and `NULL`, respectively.
- **On Lines 33-34:** arguments are defined for the processes created to execute the `printArgs()` function on Line 40 and 41, respectively.
- **On Lines 37-39:** Creates three processes, each of which will run `printpid()`, and places them on the ready queue.
- **On Line 40:** A fourth process is created to run `main()`, but this time, we want to input an array of three integers as arguments to `main()`. We're passing two arguments to `printargs()`, but this time, `3` and an array of three ints `main2args`.
- **On Line 41:** A fifth process is created to run `main()`, but we're passing this time, `6` and an array of six ints `main3args`.
- When run, the **null process** first gain control, and runs `main()` as a process. Then `main()` creates and puts five processes on the ready queue. Because the ready queue is not a priority queue *(yet)*, we'd expect the processes to be run in FIFO scheduling order:

```
Hello XINU WORLD!
This is process 2
This is process 2
This is process 2
This is process 2
This is process 2
This is process 2
This is process 2
This is process 2
This is process 2
This is process 2
Hello XINU WORLD!
This is process 3
This is process 3
This is process 3
This is process 3
This is process 3
This is process 3
This is process 3
This is process 3
This is process 3
This is process 3
Hello XINU WORLD!
This is process 4
This is process 4
This is process 4
This is process 4
This is process 4
This is process 4
This is process 4
This is process 4
This is process 4
This is process 4
Hello XINU WORLD!
This is process 5
This is process 5
This is process 5
This is process 5
This is process 5
This is process 5
This is process 5
This is process 5
This is process 5
This is process 5
1
2
3

Hello XINU WORLD!
This is process 6
This is process 6
This is process 6
This is process 6
This is process 6
This is process 6
This is process 6
This is process 6
This is process 6
This is process 6
10
20
30
40
50
60



All user processes have completed.
```

- If we uncomment **Line 14**, then a process yields itself and calls the scheduler to run the next process on the ready queue for cooperative scheduling:

```
Hello XINU WORLD!
This is process 2
Hello XINU WORLD!
This is process 3
Hello XINU WORLD!
This is process 4
Hello XINU WORLD!
This is process 5
Hello XINU WORLD!
This is process 6
This is process 2
This is process 3
This is process 4
This is process 5
This is process 6
This is process 2
This is process 3
This is process 4
This is process 5
This is process 6
This is process 2
This is process 3
This is process 4
This is process 5
This is process 6
This is process 2
This is process 3
This is process 4
This is process 5
This is process 6
This is process 2
This is process 3
This is process 4
This is process 5
This is process 6
This is process 2
This is process 3
This is process 4
This is process 5
This is process 6
This is process 2
This is process 3
This is process 4
This is process 5
This is process 6
This is process 2
This is process 3
This is process 4
This is process 5
This is process 6
This is process 2
This is process 3
This is process 4
This is process 5
This is process 6
1
2
3

10
20
30
40
50
60



All user processes have completed.
```

Part 6: The Queue (Graded)

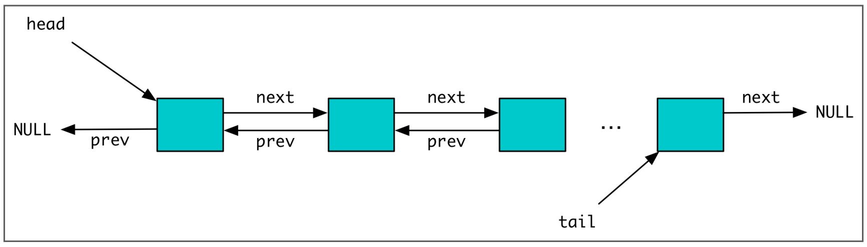
Queues are essential data structures for operating systems, and Xinu is no exception. The ready queue is used to order processes for CPU scheduling, the sleep queue holds a number of processes that are sleeping and waiting on the timer, other wait queues hold processes that are waiting for I/O to return, and each semaphore has a queue of processes waiting on it.

The codebase you have did not come with queues. Your objective is to implement it.

Specific Requirements

You need to make changes to the following files. I marked everywhere that you need to make changes with `//TODO` comments. Open the following files, find those comments, and add your code.

1. In `include/queue.h`:
   - Find `struct queue` and finish declaring its data members. Examples might be a pointer to the head, pointer to tail, and the queue's size.
   - Find `struct qentry` and finish declaring its data members. Each entry *must* hold a `pid` of the process (recall that pids are of type `pid32`), and pointers to the previous and next queue entries.



2. In the following .c files, I've provided you with the skeletons. All you need to do is complete the implementation guided by the `//TODO` comments. The queue must be dynamically allocated (and freed) on the heap. Make the changes in the following files:

   - `system/queue.c` - this file contains queue access and manipulation functions.
   - `system/newqueue.c` - this file contains the `newqueue()` function, which is used to create and initialize a new queue structure.

   In Xinu, the dynamic allocation functions are as follows:

   - `void* malloc(uint32 nbytes)` -- this is just like what you've used before. Inputs the number of bytes to allocate on the heap, and returns a pointer to the first byte.
   - `void free(void *memblk, uint32 nbytes)` -- this is a tiny bit different from the `free()` you're used to seeing, in that you must supply it with the number of bytes you'd like to free up.

   It goes without saying that your queue must be free of memory leaks.

3. After you've completed the implementation of the queue, we need to add some code to get it working with the kernel's scheduler. Again, open up and find the `//TODO` comments in the following files:

   - `system/ready.c` - this file contains the `ready()` function, which is used to put an already-created process on the ready queue.
   - `system/resched.c` - this file contains the `resched()` method, which schedules the next process on the ready queue for execution, while placing the process currently running at the tail of the ready queue.

4. Remember that you can use `kprintf()` to print messages to the console (useful when debugging).

5. On success, your output should match those shown to you earlier in my precompiled solution. If you uncomment `resched()` in `main.c`, you should get the cooperative version of the output, also shown earlier.

Grading

This assignment will be graded out of 100 points:

- [5pt] `isfull()`, `isempty()`, `nonempty()` are properly implemented.
- [5pt] `printqueue()` is properly implemented. The output format should be `[(pid=<p1>), (pid=<p2>), ...]`.
- [50pt] `newqueue()`, `enqueue()`, `dequeue()`, `getbypid()`, and `remove()` are all properly implemented and are free from memory leaks
- [35pt] Interaction with the ready queue have been properly implemented for `resched()` and `ready()`
- [5pt] Your program observes good style and commenting.

Submission

You can submit the assignment with the following steps:

- Log into the development-system VM
- Run `make clean` in the `compile/` directory
- Navigate to the directory containing your project directory, `proj1/`
- Create an archive: `tar -czvf <id>_proj1.tar.gz proj1/` where `<id>` is your student id
- Submit it on Moodle via the browser on the development-system VM.

Credits

Written by David Chiu in 2015.
Edited by Hicham Elmongui in 2020.