CS230/330: Project 3
I/O Devices and Processes

Handed out: February 12, 2003
Due: Friday, February 21, 11:59 p.m.

1 Assignment

Now that you've got virtual memory and context switching fully debugged and working perfectly, it's time to start implementing a variety of system calls. This assignment should be extremely straightforward if your memory management and context switching code is working.

2 Instructions

1. Finish Project 2. If you are having trouble getting this to work, it is critical that you see us as soon as possible. Here is a short check-list:

   - Make sure the kernel has its own page table in region 0.
   - Make sure every process has its own PCB structure including the idle process.
   - Make sure every PCB has a pointer to a unique Region 1 page table.
   - Make sure your PCB has a CPU context frame structure. This structure must be at least MAX_CPU_CONTEXT_FRAME bytes in size.
   - When context switching, make sure you copy the entire CPU context frame. Just copying the PC and SP is not enough to do context switching. Please refer to the big yalnix handout.
   - Make sure you flush the TLB whenever you update a page table.

   DO NOT PROCEED BEYOND THIS STEP UNTIL YOU HAVE PROJECT 2 FULLY WORKING!.

2. Implement a queue structure and create the kernel ready queue. Put init1 and init2 on the ready queue and modify the TRAP_CLOCK handler to context switch by removing the current process and putting it onto the end of the ready queue. Then context-switch to the next process on the ready queue. If the ready queue is empty, context switch to the idle process.

3. Implement the DisplayWrite() system call.

4. Implement the PenRead() system call.

5. Implement the SerialWrite() system call.

6. Implement the SerialRead() system call.
7. Implement the Exit() system call. This should terminate a process by releasing its PCB structure, memory, and CPU context. The process will never run again.
8. Modify your kernel so that when there are no more user processes, it terminates by calling Halt(). The idle process is not a user process (so it doesn't count here).
9. Implement the Fork() system call.
10. Implement the Wait() system call.
11. Implement the Exec() system call.
12. Modify your kernel so that it loads a single user program called "init". Since Fork() and Exec() should now work, "init" can use those system calls to run more programs. If you still want to run the project 2 tests, use this program:

```
#include "yalnix.h"

int main() {
   char *args1[] = { "init1", "init1", 0 };
   char *args2[] = { "init2", "init2", 0 };
   int status;
   if (Fork() == 0) {
       if (Exec(args1[0],args1) < 0) {
          printf("Couldn't run init1\n");
       }
       Exit(1);
   }
   if (Fork() == 0) {
      if (Exec(args2[0],args2) < 0) {
          printf("Couldn't run init2\n");
      }
      Exit(2);
   }
   Wait(0, &status);
   Wait(0, &status);
   Exit(0);
}
```

3 Specific Details and Suggestions

- To implement SerialRead() and SerialWrite() you will need to have a variety of process queues and data structures for holding sleeping processes and received input. Please consult the big handout for some details of this.

- Try to keep your implementation as simple as possible.
- You will need to use queues in other parts of the kernel. Try to make your ready queue implementation generic so that you can reuse it in other parts of the kernel.
- Think about how you are going to test each part of the kernel. Make sure you test each component before proceeding to the next.
- To implement Fork() you need to make a copy of the calling process. This means you will need to allocate memory and use memcpy() to copy the data from all of the pages from the parent to the child. The only problem is that you can't map the parent and the child process in virtual memory at the same time. You'll need to devise a clever page table trick to do the copy.
- Each process should record the process ID of its parent process. Note: recording a pointer to the parent process is a bad idea—what happens if the parent dies before the child and you have a stale pointer?
- To implement Wait() and Exit(), each process will need to have a queue of dead children (i.e., zombies). When a process calls Exit(), it is placed on the zombie queue of its parent. If the parent is dead, the process is simply destroyed and the exit code ignored.
- To implement Exec() you will need to modify your program loading code. Pay careful attention to error handling and remember that Exec() always returns to the calling process with an error if the new program couldn't be loaded for some reason.

4 Logistics and Grading

This project must be handed in via CVS. Make sure that all of your source files, Makefile, and README files have been committed to the repository. If working in a group, the project should be committed to the repository located in `/u5/groups/groupname`. Make sure your project name is “yalnix”.

To test correctness, we will run your kernel on a variety of init programs. These tests will be provided to you in advance to help with your testing. However, in order to run these tests, your kernel must be able to run programs and context switch. Get project 2 working!

5 Preview

Project 4 implements the rest of the kernel. If you are bored, work ahead.