CS230/330: Project 3
Booting to a Shell

Handed out: Friday, April 28, 2000
Due: Friday, May 5, 2000, 11:59 p.m.

1 Assignment

Now that you've got your virtual memory system fully debugged, the purpose of this assignment is to implement the rest of process management. Although this assignment is going to sound like a lot of work, it should be extremely straightforward if your memory management code is working.

2 Instructions

1. If you are still working on memory management from Project 2, get that to work first.
2. Implement a queue data structure that you can use to create queues of PCBs.
3. Using your queue, implement the process dispatcher and a ready-queue. This code should implement a simple round-robin scheduling scheme that simply takes the first process off the ready queue and context switches to make it the active context. Modify the TRAP_CLOCK handler so that it invokes the dispatcher on each clock interrupt. Furthermore, your scheduler should only schedule the idle process if no other processes are available to run.
4. Modify your kernel so that it loads both init1 and init2 in KernelStart() and places them on the ready queue. Then try to run all of the tests from Project 2 to make sure your kernel still works.
5. Implement the GetPid() system call.
6. Implement the Delay() system call. To do this you will need to create a special "sleep" queue and modify your TRAP_CLOCK handler to check for processes that need to awaken.
7. Implement the Fork() system call.
8. Implement the Exec() system call. Hint: you've already implemented 90% of it in load.c.
9. Implement the Wait() system call.
10. Implement the Exit() system call. When a process exits, you should collect its exit code and free its pages. However, the process does not officially die until the exit code

is collected by the Wait() system call in the parent. Therefore, your kernel will need to maintain some kind of data structure in the PCB that keeps tracks all of the child processes that have called Exit().

11. Modify your kernel so that it loads a single process called 'init' at startup. Furthermore, you should modify your kernel so that it halts when there are no more processes to execute other than the idle process.

12. Congratulations, your Yalnix kernel should now be able to run a partial version of your Yash shell.

3 Specific Details and Suggestions

- To implement Fork() you will need to create a new user process and copy all of its memory (note: you should perform a simple copy–do not attempt to do anything fancy like page-sharing). However to make this copy, you will effectively need to have two different region 1 page-tables active (one for the parent process and one for the child process). Since it is impossible to map two page tables simultaneously, you will need to come up with some alternative scheme. One approach is to grab an unused page table entry in the parent process and use it temporarily to copy one page at a time to the child.

- To implement Delay(), you will probably need some kind of priority queue.

- To implement Wait() and Exit(), you may want to stick some kind of process queue in your PCB structure. This queue would be used to hold PCBs for child processes that have called Exit() but which have not been collected by Wait() (Zombies).

- If a process calls Exit() and its parent process is dead, the exit code should be discarded and the PCB destroyed.

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 `/export/home/groups/groupname`. Make sure your project name is "yalnix".

To test correctness, we will run your `yalnix` executable on an init process and shell that we will provide. This will then be used to run a number of test programs to exercise your process management system calls.

This project has a relatively short due-date. Do not wait until the last minute to start.