CS230/330: Project 2 Supplement
Program Loading

Handed out: Friday, January 25, 2002
Due: Friday, February 1, 2002, 11:59 p.m.

1 Introduction

In section 1.4 of the Project 2 handout, you have to modify your kernel to load the `init` process. To do this, you need to locate `init` on the disk device, load it into memory, and start it as a process. This stage of the process is particularly tricky. Please follow these directions carefully and start early.

2 Processes

You should have already defined a process control block (PCB) structure and used it to set up a kernel idle process. Your PCB should have information for storing the machine context, page tables, and a process state. If you haven't implemented a PCB, go back to the Project 2 handout and re-read the instructions.

3 Your First User Program

- Write a yalnix user program such as the following.

```
/* init.c */
#include "yalnix.h"
int main(int argc, char *argv[]) {

   while (1) {
     ConsoleWrite("init: Idling...\n", 16);
     Pause();
   }
}
```

- Add 'init' to the list of user programs in the Yalnix makefile.
- Compile the user program by typing 'make user'. This should produce some output like this:

```
$ make user
*** ydisk: Reformatting user file system
Done.
Building init
gcc -c -g -I. -I/yalnix/include init.c
gcc -g -I. -I/yalnix/include -static -Wl,-M/yalnix/lib/LdMapFile.User -o init init.o -L/yalnix/lib -luser
start     size name
----- -------- -------------------------
16340    36360 init
$
```

4 Loading the disk directory

Before the disk can be used for anything, you have to load the disk directory into kernel memory so that files can be located. The disk directory is located in the last 8 blocks of the disk and is stored as an array of `yalnix_dirent` structures as defined in the header file `yalnixdir.h`.

- Write a function to start loading the disk directory into memory. This function will have to allocate some kernel memory to store the directory and will need to issue a command to the disk to start loading the first directory block.
- Modify your kernel `main()` to call the function in the previous step so that your kernel will load the directory when it starts.
- Modify the `TRAP_DISK` handler function to finish loading the disk directory by loading the remainder of the blocks. To help debugging, put some print statements in the `TRAP_DISK` handler to watch each block as it is loaded.
- When the entire directory has been loaded, print out the directory contents so that you can see if it worked.
- Recompile your kernel and try running. You should get some output similar to this:

```
HW: Initializing memory... 4194304 bytes
BootROM: Loading master boot record.
BootROM: Starting execution at 1000000.
Booting Dave's yalnix
Got kernel header.
About to bootstrap.
Welcome to Dave's yalnix
Initializing interrupt vector
Initializing memory
   Physical memory  = 4194304
```

```
   &_end            = 0x0100d460
   &_etext          = 0x01004c88
   textpn           = 3
   datapn           = 7
   Num free pages   = 504
   Kernel page table = 0x01006050
Enabling virtual memory
It's ALIVE!!!
Allocating disk directory
Directory block = 1
Directory block = 2
TrapClock
Directory block = 3
Directory block = 4
Directory block = 5
Directory block = 6
Directory block = 7
Directory block = 8
Disk ready
0 init  16340  36360
TrapClock
TrapClock
TrapClock
...
```

5 Loading Init

Once the yalnix disk directory has been loaded, your kernel needs to load `init`. This is even more complicated than loading the directory, but it's a similar idea.

- Start writing a function `LoadProgram(char *filename, char *args[], PCB *p)` that takes a filename, an array of command line options, and a process control block structure. This function will start the procedure of loading a program into a process.
- The first step of `LoadProgram()` should search the disk directory to find a filename match. If no match is found, it means the file doesn't exist. Your kernel should print some kind of error message and `LoadProgram()` should return.
- If the program exists, modify `LoadProgram()` so that the kernel makes a copy of the command line options. Use `pmalloc()` (or other memory management functions) to allocate memory and make a copy. Note: this step is important.
- Next, have `LoadProgram()` initiate a disk transfer by first loading the YOFF header of the program. The YOFF header is always the first block of the file. To find the

first block, you will need to look at the `start_block` entry of the corresponding disk directory structure. Since loading the header will take some time, your kernel will simply issue a command to the disk device and return (to the idle process) to wait for a `TRAP_DISK` interrupt.

- When the YOFF header has been loaded, a `TRAP_DISK` interrupt will occur. Have the `TRAP_DISK` handler examine the header to find out how much memory is needed to load the process. To do this, you will need to look at the following fields of the header:

```
header->text_base;    /* Base address of text segment */
header->text_size;    /* Size of text segment */
header->data_base;    /* Base address of data segment */
header->data_size;    /* Size of data segment */
header->bss_base;     /* Base address of BSS segment */
header->bss_size;     /* Size of bss segment */
```

  Modify your kernel to print out the above fields when the header is loaded. For example:

```
Loaded YOFF header
    Magic = YOFF
    text_base   = 0x02000000
    text_size   = 32088
    data_base   = 0x02008070
    data_size   = 2408
    bss_base    = 0x02008a08
    bss_size    = 2820
```

- Calculate the size of the process stack. To do this, you will need to compute the size of the command line options by adding together the following quantities:

  - The length of all strings in `argv[]` passed to `LoadProgram()`. Recall that a string is always terminated by a zero and the zero is included in the length.
  - The size of the `argv[]` array itself. This will be the number of arguments multipled by `sizeof(char *)`.
  - The size of the `argc` and `argv` parameters passed to `main()`.
  - An initial stack padding `INITIAL_STACK_FRAME_SIZE`.
  - Padding bytes to make sure the stack pointer is aligned on an 8-byte boundary.

  The following C code illustrates how to compute this size:

```
   char *argv[];   /* The list of arguments (passed to load program) */
   int  argc = 0;
   int  stack_size = 0;
```

```
/* Size of argument strings */
while (argv[argc]) {
   stack_size += strlen(argv[argc])+1;
   argc++;
}
/* Array of argument pointers */
stack_size += (argc+1)*sizeof(char *);

/* int, char *[] arguments to main() */
stack_size += sizeof(int) + sizeof(char **);

/* Padding for alignment */
stack_size = (stack_size + 8) & ~7;

/* Initial stack frame size */
stack_size += INITIAL_STACK_FRAME_SIZE;
```

- Using the above header fields and the stack size, allocate memory for the process. To do this, you will need to start filling in page table entries for the process. Use the `GetFreePage()` function to get pages of memory and assign them to the process. Set pages in the text segment with `PROT_READ` and `PROT_EXEC` protection. Set pages in the data and bss sections with `PROT_READ` and `PROT_WRITE` permission. Set pages in the stack region with `PROT_READ`, `PROT_WRITE` and `PROT_EXEC` permission. Note: since memory can only be allocated in page-sized chunks, you will need to round up the next page if the sizes only include part of a page.

  Note: to compute the stack size, you need to use the saved argument information that was passed to `LoadProgram()`. Since the allocation of memory occurs from within `TrapDisk()` after the YOFF header is loaded, you need to use the saved copy of the arguments—this is because `LoadProgram()` is not executing at this point.

- After memory has been allocated, start issuing commands to the disk to start bringing the program into memory one block at a time. This will be similar to the code that loaded the kernel (in the boot loader) and the code that loaded the directory. You will need to keep track of how much data has been loaded and how many blocks are remaining. Keep in mind that this will generate a long sequence of `TRAP_DISK` interrupts.

- After the program has loaded, context switch to the newly loaded program. Make sure the process page tables are active.

- Initialize the BSS section by zeroing all of the data starting at `bss_base` and continuing for `bss_size` bytes.

- Copy the `argv[]` options to the stack. To do this, you will need to use some code like this:

```
/* argsaved[] = Saved copy of arguments in kernel space */
/* argsize = The size of the strings in the argv[] array */
/* argc    = The number of arguments in argv[] */

int i;
char *cp = (char *) (VMEM_1_LIMIT - argsize);
char **cpp = (char **) (((int) cp - ((argc+3)*sizeof(void *))) & ~7);
char *cp2 = (char *) cpp - INITIAL_STACK_FRAME_SIZE;

frame->sp = (void *) cp2;   /* Set the process stack pointer */
*cpp++ = (char *) argc;
for (i = 0; i < argc; i++) {
    *cpp++ = cp;
    strcpy(cp,argsaved[i]);
    cp += strlen(cp) + 1;
}
```

- Set the process program counter to the `text_entry` field of the YOFF header. For example:

```
frame->pc = (void *) header->text_entry;
```

- Clean up any state in the disk handler used for loading.
- Return from TRAP_DISK. If successful, the newly loaded process should be running.

6 Sample output

If you get this part of the project to work, the output of your kernel might look something like this:

```
HW: Initializing memory... 4194304 bytes
BootROM: Loading master boot record.
BootROM: Starting execution at 1000000.
Booting Dave's yalnix
Got kernel header.
About to bootstrap.
Welcome to Dave's yalnix
Initializing interrupt vector
Initializing memory
   Physical memory  = 4194304
   &_end            = 0x0100d460
```

```
   &_etext          = 0x01004c88
   textpn           = 3
   datapn           = 7
   Num free pages   = 504
   Kernel page table = 0x01006050
Enabling virtual memory
It's ALIVE!!!
Allocating disk directory
   directory = 0x01ffa000
   pdirectory = 0x013fa000
Directory block = 1
Directory block = 2
TrapClock
Directory block = 3
Directory block = 4
Directory block = 5
Directory block = 6
Directory block = 7
Directory block = 8
Disk ready
0 init  16340  36360
Loaded YOFF header
    Magic = YOFF
    text_base   = 0x02000000
    text_size   = 32088
    text_entry  = 0x02000000
    data_base   = 0x02008070
    data_size   = 2408
    bss_base    = 0x02008a08
    bss_size    = 2820
Loading  = 16341
Loading  = 16342
Loading  = 16343
Loading  = 16344
Loading  = 16345
Loading  = 16346
Loading  = 16347
Loading  = 16348
Loading  = 16349
Loading  = 16350
Loading  = 16351
Loading  = 16352
Loading  = 16353
TrapClock
```

```
Loading  = 16354
Loading  = 16355
Loading  = 16356
Loading  = 16357
Loading  = 16358
Loading  = 16359
Loading  = 16360
Loading  = 16361
Loading  = 16362
Loading  = 16363
Loading  = 16364
Loading  = 16365
Loading  = 16366
Loading  = 16367
Loading  = 16368
Loading  = 16369
Loading  = 16370
Loading  = 16371
Loading  = 16372
Loading  = 16373
TrapClock
Loading  = 16374
Loading  = 16375
Done loading!
Running...
TrapKernel
init: idling...
TrapClock
TrapKernel
init: idling...
TrapKernel
init: idling...
TrapClock
TrapKernel
init: idling...
...
```

Note: Your kernel doesn't have to produce this exact output. However, you should see some output that is similar.

7 Suggestions

Almost everything can be done in the handler function for `TRAP_DISK` provided you think about the problem in terms of state machines.

- The disk device runs in two states, INIT and READY. In the INIT state, it is still loading the disk directory into memory and can not be bothered with program loading. In the READY state, the disk device can be used to load programs.
- A process being loaded is in one of three states, PRELOAD, YOFF, and LOADING. In the PRELOAD state, the process is ready to be loaded, but the disk device hasn't done anything yet (perhaps the disk is busy reading the directory). In the YOFF state, the YOFF header has been read, but not other part of the file has been loaded. In the LOADING state, data is being loaded from the disk into memory.
- To handle these states, `TRAP_DISK` should keep track of its current state. Then, just put an `if` or `switch` statement in the handler function to perform the appropriate operation. When loading a process, set the process state to one of the above three states and make the handler function behave differently depending on the current state.

Start early and ask us questions if you are unable to get any part of the loading procedure to work. Without loading, nothing else in your kernel will work!

When you have successfully loaded `init`, go back to the project 2 handout and continue with the instructions. You are almost done.