Project Report

13307130269

白帆

Contents

Overlook

It is a developing report and a manual for my course project of Operating System AY2016, a simple kernel with shell on MIPS32, MSIM.

In this project, I implemented a firmware(BOIS), a boot sector(by using existing codes), a kernel with virtual memory mechanism, time shared concurrent mechanism, a series of system call, a shell on the kernel, and some demo application. Before the trap handler, I just followed the tutorial, and afterwards was designed and implemented by myself.

Booting

Firmware

The firmware is what already in the ROM while the machine(simulator) powering up. And all CPU begin from here. So, what the firmware should do is dividing master CPU with others, loading and executing boot sector.

Diskread, diskwrite and kputs, kprintf was implemented in this period.

MBR

MBR is used to load kernel, but for compatibility, it should be location independent. So, the address of diskread was given by BIOS, and then it could be executed anywhere. I just put it in the stack of firmware.

Since the code of MBR was provided, nothing more to talk about.

Bootstrap

The kernel is worked on a fixed location, from 0x80300000. Since the kernel was mostly written in C, this bootstrap is only to do some primitive setting and jump into C. In my project, the TLB manager is also written in C.

Kernel

Device Initialize

Actually, device initialization is not necessary on MSIM, since virtual devices are always usable and never change. But I still followed the tutorial and did this.

SMP Spin Up

In the beginning, master CPU and slaves was divided. The master did all the booting jobs, and other just waiting for it by listening to themselves' mailbox.

Now, it is in the kernel, they should get up to work. The master have arranged stacks and starting location for slaves, and then send a mail consists by them into each one's mailbox. Slaves will jump into the starting location and use the given stack to work.

Trap Handler

We know, MIPS32 provides 0x80000000 and 0x80000180 for TLB miss and other generic exception. But 0x80 is too small, even not enough to saving and restoring registers. So, the code in these 2 location just jump out to the real handler.

In this project, clock time out, syscall, and TLB miss was implemented, the details are below. Other exception will be regarded as fatal error, and there is a reboot on fatal error mechanism mentioned below.

TLB Management

The page size is 4K, and the address space is 2G. In a multilevel page table model, 3 levels are necessary, which is too pesky. And if all the page tables are store in kernel as 1 level, 2G/4K*4B*256=512M kernel memory is required, which is too large.

As a result, I decided to use a larger page size, 2M here. I just maintain 2 4K*256=1M page table in kernel. While dealing with a page allocation, 2M memory, thus, 512 continuous physical page was allocated. While dealing with a TLB miss, the page table of current pid was checked first with the 10bits VPN in entryhi for the first 11bits(the 1st bit is always 0), then the entryhi was direct used for next 9bits, and 12bits remained was in a physical page.

PID	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	0	VPN										PPN_L									PO											

ASID	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	PPN_H											PPN_L									PO											

Process Model

PCB Entry

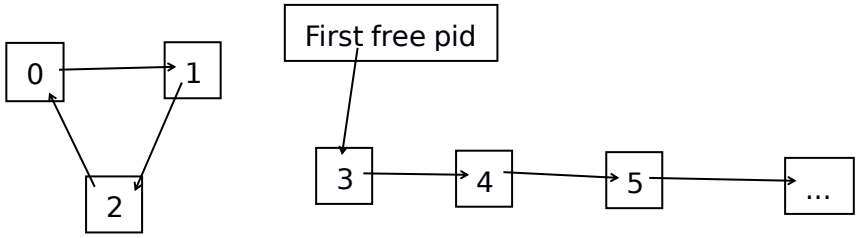
```
18 struct pcb_entry {
19     pid_t           ppid;
20     pid_t           wait;
21     unsigned long   compare;
22     unsigned long   pc;
23     int             status;
24     unsigned long   max_addr;
25     unsigned long   gpr[32];
26     pid_t           next;
27 };
```

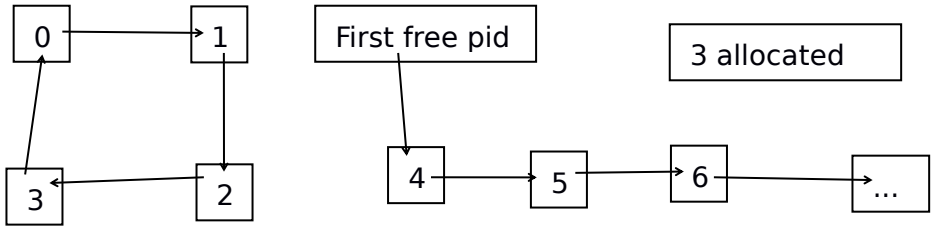
The ASID of TLB was used to avoid TLB clearing when process switch. So, max pid would be 255. And all the pcb entry was stored in a array of kernel space.

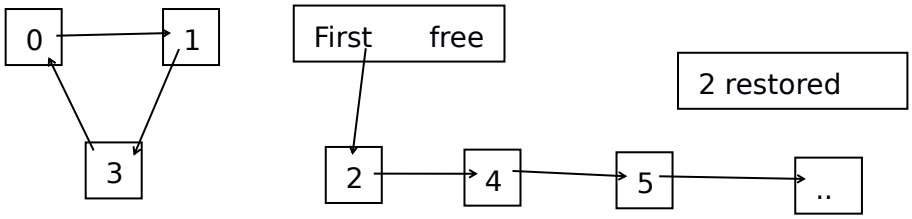
Name	Features
ppid	Parent pid
wait	The pid waiting for or 0 if not waiting
compare	The time up if sleeping, or 0
pc	Stored PC, when switched to, this value was load in EPC
status	Running, block, stoped or ready
max_addr	The max address already allocated
gpr[32]	Stored register
next	Next process in the lived queue if this pid is currently used, or next free pid if this pid is free. So the lived process is a circled list, and free pid is a stack.

PID Allocation/Restoration

The kernel it self has a fakes pid 0, and its next is 0 in the beginning. When dealing with a pid allocation, the first free pid(maintained) was used, and set the first free pid as its next. Then add it after its parent in the lived circle. And restoration is reversed.







Scheduling

Each time when clock time out, a process exited, waited, sleeped, or called other out-handler-syscall, the chance to use CPU will be transfer to next READY process, and current process is setting to READY(from RUNNING). Normally,

kernel(pid=0) in the queue will never run, because the status of kernel is always setting as RUNNING. But when dealing with a out-handler-syscall, it will temporally forced transfer to kernel. And kernel can only use __to_user syscall to transfer back to the next process(of the caller).

When switch, the old process' register and pc will be saved with the new one restored. And ASID was set to the new pid except out-handler-syscall, because the caller's page table is used but not kernel's.

Each time going into a user process, the clock is set, and disabled when going into kernel.

When travel in the lived circle, it's checked whether the next process is STOPED, if so, the pid will be restored. We cannot restore the pid immediately when exit, because the circle list is single direction. If a process with status=BLOCK met, the condition will be checked, if satisfied, it will be set to RUNNING directly.

System Call

List

#	Name	Arguments		Return	Action
0	pid	-		This process' pid	-
1	ppid	-		Parent's pid	-
2	fork	-		Child pid for parent, 0 for child	Create a new process, copy every thing except pid nor page table to child
3	exec	char*	fname	-1 when error or none	Clear all resources of current process and set up the specified application Do nothing if the application is not exist
4	sleep	unsigned int	time	0	Block current process with setting a time compare, wake up after time over
5	waitpid	pid_t	id	0	Block current process with setting wait, wake up after the waited process stoped
6	malloc	unsigned int	size	Address allocated	Just add max_addr or alloc new page

7	__to_kernel	-		0	Get into kernel mode
8	__to_user	-		0	Get into user mode, only used to return from out-trap-handler system call
9	exit	unsigned int	ex_num	-	Set current process as STOPED, restore allocated pages, but pid will restore later
A	puts	char*	s	0	Print s to screen, end with ‘\0’
B	gets	char*	s	Length of s, include ‘\n’ and ‘\0’	Get string from keyboard and store into s with a max length include ‘\0’
		unsigned int	Max_len		
F	none	-		-	Indicates no system call

Out-Trap-Handler System Call

When the EXL bit is set, any interrupt will not be accept except TLB missing. So, when doing syscall with using user space, like puts, exec etc., TLB missing may be occur. When it happens, the pc in handler will not be saved, it may cause some error. So these syscall should be dealt out-trap-handler.

When one of these(puts, gets, fork, exec) syscall coming, the handler just save the syscall number and arguments into kernel space, then make a forced switch to kernel. And kernel is in a forever loop for dealing with syscall.

So that the TLB missing can be dealt normally.

Fork

Fork is the key of multiprograming. It is a out-trap-handler syscall because of the coping action. It will allocate a new pid with parent is the caller, status is READY, and allocate the same size of its parent’s memory, copy memory, register and pc from parent. The difference is v0(return value), parent’s will be set to the child pid, and the child’s set to 0.

Fork is used by shell, and the pidspeaker is a demo of it.

Exec

To execute new program, exec will be used. It is a out-trap-handler syscall because of the coping action. It do not allocate new pid but copy the new program from file system(in kernel here) and set up stack, pc.

Exec is used by shell, and the pidspeaker is a demo of it.

Malloc

It was mentioned before that a virtual page is of 2M size. When allocate memory for a process, just add the max_addr directly, then check whether these page was allocated, if not, allocate enough page.

The free PPN is maintained like pid, using a stack. Space is enough in kernel because only the 10bits' PPN_H is needed.

It is a in-trap-handler syscall, because we just allocate memory but not access.

Isort is a demo of it, which is a insertion sort based on linked list.

Puts

The principle component of it was implemented early. The syscall is just a encapsulation. Since it needs to access the string in user space, it is out-trap.

And the puts was used, eh, almost everywhere.

Gets

Gets is more difficult than puts. Since it needs to access the string in user space, it is also out-trap.

It is different from gets in C std but like fgets because of the max length threshold to avoid overflow. Since the kernel is non-preemptive, when a gets is running, other processes can not run either.

The keyboard interrupt is not used but a polling. '\n' will be kept in result, and '\0' will be added. The puts was also used almost everywhere.

Reboot On Fatal Error

Some bug is still in this kernel, I think it may be some problem in TLB management or task scheduling. Sometimes strange exception or page access may occurs and crash the system.

To deal with that in as less as possible time, I designed a reboot on fatal error mechanism, which added an "else" to TLB manager and trap handler that set mode to kernel and eret to __reset and re-init the device and whole system.

```
Welcome to SWSH, type 'help' for help
SWSH: $ isort
Fatal error, unkown exception, rebooting...
--DEBUG-- Hello world from kernel!
--DEBUG-- Hello World from user space!
Welcome to SWSH, type 'help' for help
SWSH: $
```

User Programs

Init

It will be exec-ed by kernel, and stay alive forever. The init's job is only run(by fork, exec) a shell and keep it(by waitpid), that whenever the shell excited, init will run a new one.

User can also run new inits from shell, but so that these inits will all not exit and take resources.

```
sailwhite@sailwhite-CW35S:~/OSPJ$ msim -c run.conf
--DEBUG-- Hello world from BIOS in C!
--DEBUG-- Hello world from kernel!
--DEBUG-- Hello world from CPU #1!
--DEBUG-- Hello world from CPU #2!
--DEBUG-- Hello world from CPU #3!
--DEBUG-- Hello World from user space!
Welcome to SWSH, type 'help' for help
SWSH: $ exit
--DEBUG-- pid 2 exited with 1032184!
Welcome to SWSH, type 'help' for help
SWSH: $ init
--DEBUG-- Hello World from user space!
Welcome to SWSH, type 'help' for help
SWSH: $
```

Shell

I just reused the code of assignment5, with historic feature, readline, and back ground command removed. Each command refers to a application without any argument. The shell runs the application by fork, exec.

User can also run new shell with “swsh” command to create child shell, use “exit” in child shell to get back to parent shell.

```
Welcome to SWSH, type 'help' for help
SWSH: $ help
        Usage: appName
        appName in :
                init                    the init program, can also be excuted manually
                swsh                    this shell, you can run another as a child-shell
                aplusb                  input a,b, out put sum of them, a,b should both be pure digits in uint range
                isort                   a list based insertion sort, the claim of numbers are same as above
                pidspeaker              fork twice then print pid 10 times with 1 second interval
                help                    display this help
                exit                    exit this shell
        For details, please check the report
--DEBUG-- pid 3 exited with 0!
SWSH: $ swsh
Welcome to SWSH, type 'help' for help
SWSH: $ exit
--DEBUG-- pid 3 exited with 1032184!
SWSH: $
```

A Plus B

The classic a+b problem. Input a, input b, output a+b.

It is a demo of puts, gets using. Since sscanf is not provided, I have to transfer string to int by my self, only pure digits with in unsigned long range is supported.

```
Welcome to SWSH, type 'help' for help
SWSH: $ aplusb
a = 12345
b = 67890
a + b = 80235
--DEBUG-- pid 3 exited with 0!
SWSH: $
```

Insertion Sort

A insertion sort based on linked list.

It is a demo of dynamic memory allocating. The claim of numbers are same as above.

```
Welcome to SWSH, type 'help' for help
SWSH: $ isort
n = 5
a0 = 65
a1 = 49
a2 = 77
a3 = 58
a4 = 2
result: 2 49 58 65 77
--DEBUG-- pid 3 exited with 0!
SWSH: $
```

Pid Speaker

This application will fork twice at first, then print respective pid 10 times concurrently with 1 second interval.

It is a demo of fork and sleep.

```c
#include<stdio.h>
#include<sys/syscall.h>
int main() {
    fork();
    fork();
    int i;
    for(i=0;i<10;i++) {
        printf("My pid is %d\n",pid());
        sleep(1);
    }
    return 0;
}
```

```
My pid is 3
My pid is 6
My pid is 5
My pid is 4
My pid is 3
My pid is 6
My pid is 5
My pid is 4
My pid is 3
--DEBUG-- pid 6 exited with 0!
--DEBUG-- pid 5 exited with 0!
--DEBUG-- pid 4 exited with 0!
--DEBUG-- pid 3 exited with 0!
SWSH: $
```