Report

Bruce Tan

In this Individual Study, I proposed to implement addtional components for my Operating System Project, namely, I planned to implement

1. Memory Allocation (sbrk() brk() malloc() free() calloc() realloc() kmalloc() kfree()),
2. A thread library
3. Handling Signals
4. A File System

Rewriting Memory Allocation Module

refer to kernel/proc.c kernel/sys_memory.c winix/bit_map.c winix/slab.c winix/mem_alloc.c winix/mem_map.c lib/stdlib.c

Memory Allocation is an imoprtant part of the modern operating system, without it, nothing could be done.

Before I started writing this project, the kernel's memory management is very trivial. So in my original implementation, there is a global variable called GLOBAL_MEM_START, indicating where the next free memory region begins. So whenever a new memory is needed, GLOBAL_MEM_START is incremented. However, this simple scheme has several downsides:

1. memories cannot be reused
2. memories are not page-aligned, so memory protection are not plausible
3. It's hard to keep track of used memory by each process

So instead of having a global variable to manage memories, I used mem_map instead. mem_map is a bitmap that indicates all the free pages inside the memory. 1 indicates it is being used, 0 indicates it is free. Whenver free pages are requested by either the kernle or the user process, bitmap_search is called to search continugous free pages, and set by bitmap_set. Bitmap can also be to free up pages using bitmap_reset to set bits to 0.

With the help of bitmap, fork() can be achieved using COPY On WRITE mechanism. In my OS, when a fork is called, a bit pattern of the original process is created. e.g. 111001, which means the process occupies the first three pages, and 6th pages. When a child process is to be created, a backtracking search is done to search the system memory bitmap to find the correct match. A backtracking search is more efficient than search contiguous free pages, because some used pages can overlap with the bit pattern. For instance, say we found a match place with the bitmap, 000110, then our original bitmap, 111001, can overlap with the bitmap. This help the system to save space, make the most use of memory.

So this individual study, I also implemented malloc, free, and kmalloc and kfree

In my operating system, each process has its own virtual space dynamically translated into physical address. And the virtual space is divided into several parts, one of which is the heap. Each process has its heap area that can be dynamically extended through sbrk() brk(). sbrk() and brk() both extends the heap segment and allows processes to manage memories through the heap

```
void *do_sbrk( proc_t *caller, size_t size) {

        unsigned long *ptable;
        int nstart,len;
        int *heap_break_bak;
        void *ptr_addr;
        if (size == 0)
                return get_virtual_addr(caller->heap_break,caller);

        heap_break_bak = caller->heap_break;

        if (is_addr_in_same_page( heap_break_bak, ( heap_break_bak + size)) ) {
                caller->heap_break = heap_break_bak + size;
                // kprintf("ptr %x ptr+size %x heap_break %x %x\n", heap_break_bak,heap_break_bak +
size, caller->heap_break,mem_map[0]);
                return get_virtual_addr(heap_break_bak,caller);
        }
        ptable = caller->protection_table;
        len = physical_len_to_page_len(size);
        nstart = bitmap_search(mem_map, MEM_MAP_LEN, len);
        if (nstart != -1) {
                //set mem_map and caller's ptable's corresponding bits to 1
                bitmap_set_nbits(mem_map, MEM_MAP_LEN, nstart, len);
                bitmap_set_nbits(ptable, PROTECTION_TABLE_LEN, nstart, len);
                ptr_addr = (void *)(nstart * 1024);

                if (is_addr_in_consecutive_page( heap_break_bak, ptr_addr)){

                        caller->heap_break = heap_break_bak + size ;
                        // kprintf("ptr %x pt %x obrk %x brk %x\n", heap_break_bak, mem_map[0],
heap_break_bak,caller->heap_break);
                        return get_virtual_addr(heap_break_bak,caller);
                }
                if((int *)ptr_addr > heap_break_bak)
                        caller->heap_break = (int *)ptr_addr + size;

                // kprintf("sbrk: optr %x ", ptr_addr);
                ptr_addr = get_virtual_addr(ptr_addr,caller);
                // kprintf("ptr %x pt %x obrk %x brk %x\n", ptr_addr, mem_map[0],  heap_break_bak,
caller->heap_break);
                return ptr_addr;
        }else{
                kprintf("System out of memory\n");
                for (nstart = 0; nstart < 32; ++nstart)
                {
                        kprintf(" %x |",mem_map[nstart]);
                }
                kprintf("\n");
        }
        return NULL;
}
```

In my OS, virtual memory is very simple, Virtual Address = Physical Address + rbase, where rbase is where the first line of the code of this process resides in the physical memory. when sbrk is called, the kernel checks if the current page can be extended, if the answer is yes, then the end of the heap will be extended to the new limit. On the other hand, if the new limit will be on the next page, and next page is free, then sbrk() will be successful, and heap will be extended to the next page. However, if the next page isn't free,

In the user space, heap memory are managed by malloc free. In each of the memory segments being allocated, a header is present that has the meta information about this memory.

```
typedef struct s_block {
        size_t size;
        struct s_block *next;
        struct s_block *prev;
        int free;
        void *ptr; //a pointer to the allocated block
        char data[1]; //the pointer where the real data is pointed at.
//b->data is what malloc returns
                                        //ptr equal the address of data
}block_t;
```

Internally both of malloc and free maintains an internal doubly linkedlist of all the memories being allocated. Upon calling malloc, first fit algorithms is used. Tt traverses through the double linked list, using prev and next pointer, to find one part of the memory that is free and is large enough to hold the request. If no memory holes that can be found, sbrk() is called to extend the heap. Upon returning the new address, the header struct is inserted before the memory hole.

when free is called, it looks at the header file, check if ptr points to the requested block. If yes, then this memory block is marked as free.

However, when requests get larger and larger, the heap segment may become scattered around, there could be the case where there are several consecutive small free memory holes. Memory fragmentation slows the search time. These holes can be potentially merged into a single one to improve performance.

For kmalloc free, they are pretty much the same with malloc except one tiny difference. When no free and large enough memory blocks can be found, it calls get_free_page to request new pages. If the request size is smaller than the size of a page, then the rest of the page will be initialised with a new free memory segment.

Fiber Library

refer to lib/ucontext.s lib/makecontext.c

In my fiber library, ucontext.h is used to implement the fiber. ucontext is one of the C library for context control, It allows more advanced control flow patterns in C like iterator, fiber, and coroutines.

Because I am working on a special Architecture called WRAMP, which is a MIPS like architecture. Since no ucontext library has been implemented yet, So I wrote the ucontext in this project.

```
typedef struct _ucontext_t{

        /* Process State */
        unsigned long regs[REGS_NR];    //Register values
        unsigned long *sp;
        void *ra;
        void (*pc)();

        // stack_t uc_stack;
        unsigned long  *ss_sp;     /* address of stack */
        int    ss_flags;  /* Flags */
        size_t ss_size;   /* Number of bytes in stack */

        struct _ucontext_t *uc_link;
}ucontext_t;
```

The process state is a machine dependent part that includes all the registers values, return address register, stack pointer. stack_t is the alternative stack that may be used for this ucontext. uc_link links to the next ucontext_t for context switching.

There are four library functions in ucontext

```
int  getcontext(ucontext_t *ucp);
int  setcontext(const ucontext_t *ucp);
void makecontext(ucontext_t *, void (* func)(), int argc, ...);
int  swapcontext(ucontext_t *oucp, const ucontext_t *ucp);
```

Getcontext saves the current context into the ucp.

Setcontext transfers the control to the context in ucp.

Make context sets up an alternative thread of control in ucp. Stack should be set up appropriately so that it does not interfere with the current context's stack. Execution will begin at the entry point of func, and arguments are passed onto the stack.

swapcontext() saves the current context into oucp, and transfers the control to ucp.

```
.global _ctx_start
_ctx_start:
        lw $1, 0($sp)
        addui $sp, $sp, 1
        jalr $1
        sw $13, 0($sp)
        j _ctx_end


void _ctx_end(ucontext_t *ucp){

        if(ucp->uc_link == NULL)
                sys_exit(0);
        setcontext(ucp->uc_link);

        //should never get here
        sys_exit(2);
}

void makecontext(ucontext_t *ucp, void (* func)(), int argc, ...){
        int *args = &argc + 1;
        uint32_t **spp;
        uint32_t *sp;
        if(ucp == NULL)
                return;

        ucp->pc = (void (*)())&_ctx_start;

        //allocate stack for the ucp context
        spp = (!ucp->ss_flags) ? &ucp->ss_sp : &ucp->sp;
        *spp -= (argc + 2);
        sp = *spp;

        /**
         * Arrange the stack as follows:
         *      func
         *      arg1
         *   ...
         *   argn
         *   ucp
         *
         *      The PC of the ucp will set to _ctx_start (refer to lib/ucontext.s)
         *      _ctx_start will pop the top of the stack, which is func. After that, the Stack
         *      will be arranged such that all args are left on the top of the  stack.
         *      Then func() is called.
         *      When func() returns, _ctx_start will pop ucp, then call         *       _ctx_end()and
pass
         *      the parameter to _ctx_end
        **/
        *sp++ = (int)func;
        memcpy(sp,args,argc);
        sp += argc;
        *sp = (uint32_t)ucp;

}
```

Make context sets up the alternative stack, and point the next control flow to _ctx_start(), _ctx_start will then call func(), and pass the parameters. On successful completion, _ctx_end() is called. _ctx_end checks if the next ucontext is present, if yes, it transfers the control to the next ucontext. otherwise exit() is called.

With the help of ucontext, a custom fiber can be built. refer to user/shell.c

Signal Handling

refer to kernel/signal.c

In my OS, signal handling is slightly complicated. Whenever a signal is to be delivered, a signal context is built on the user process's stack as follows

```
        signum
        sigreturn syscall parameters
        sigreturn syscall message_t
        sigreturn syscall assembly code
        old_context
```

signum is stored on top of the stack, so that when signal handler is called, signum can be popped.

When signal handler is called, PC is set to the entry point of the OS, and RA is set to the start of sigreturn syscall assembly code. So when signal handler is finished, PC will call the sigreturn assembly code.

```
   sigreturn syscall parameters
sigreturn syscall message_t
sigreturn syscall assembly code
```

All of the above are necessary to initialise a systemcall, including the parameters of syscall, the message_t passed to the kernel, and and assembly code to triger the system call.

Once the control is transferred to the kernel, kernel will restore the context by copying the old_context on the stack to the PCB, and then reschedule all the processes.

Since the signal delivery uses quite a lot of stack, a check is made to see if the stack is large enough to hold the signal delivery, before the signal is delivered.

Alarm and timer

refer to kernel/clock.c

Even though the architecture my OS is running on, has only one timer interrupt, multiple timer can be achieved by using a FIFO linkedlist of timers.

Whenever a timer interrupt is generated, it checks the front of the queue, see if there is any immediate timer to be triggered. If yes, that timer is dequeued and freeed, and a SIGALRM signal is delivered to the corresponding process.

FILE SYSTEM

refer to fs/

The file system is fairly complicated. Due to compatiblity reasons, the File System is not integrated as part of the kernel. But it can still can be compiled by gcc for testing purposes.

The file system is divided into several components, dev, filp, path, inode, open, read_write, close

dev is for the lower level drivers to read and write files.

filp is the data structure for file descriptors. When a file is opened, the one entry in the file table of the calling process will be referred to the filp entry in the kernel. It contains information about corresponding inode, file position, and so forth.

path responsible for parsing the path string. eat_path will return the last inode in the path string, or null if nothing can be found. last_dir, advance and get_name does the actual work of parsing the file name.

inode contains several methods including get_inode, put_inode, read_inode, alloc_inode. get_inode takes the inode number, and return the corresponding inode structure, or NULL if inode does not exist. put_inode saves the inode information and write it to the device driver. read_inode parses the raw data from the driver to the inode structure. alloc_inode allocate a new inode from the current file system.

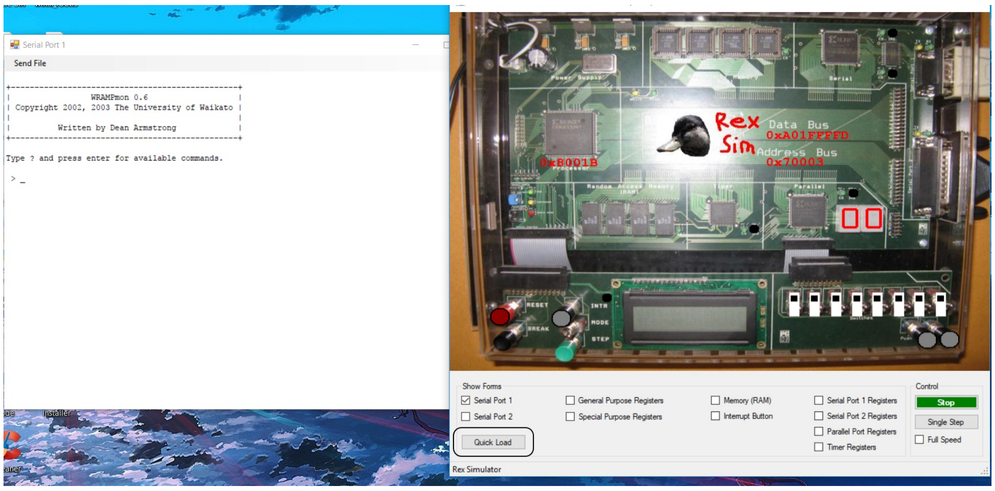
LRU is also used to speed up lookup. It would be a waste of time to read and write every time a file is written or read. Corresponding blocks can be saved in memory temporarily before it is written to the file. LRU stores the most recently used block cache in the front, and least recently used block cache in the rear. When new block is read, it is moved to the front of the queue. The rear of the queue is disposed. Before the rear is disposed, the file system checks if it is dirty, if it is, the content is written to the driver. Throughout the time, LRU will always contain the most recently used blocks at the front.

LRU also uses hashtable to speedup looking. When a block is requested, rather than looping through all the LRU blocks, a hashtable is checked to see if the block is already in the cache. if not, a new block is read from the driver and added to the front of the LRU.

open, read, write, and close all deals with the actual file. They all deal with file descriptor. When open is called, a file descriptor is created in the kernel fd table for the corresponding inode. Read and write all deals directly with this file descriptor. close deleted the entry of the corresponding file descriptor.

How to Test

Open RexSimulator



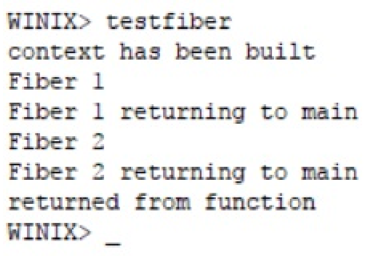
```
+-----------------------------------------+
|              WRAMPmon 0.6               |
| Copyright 2002, 2003 The University of Waikato |
|                                         |
|         Written by Dean Armstrong       |
+-----------------------------------------+

Type ? and press enter for available commands.

 > _
```

☑ Serial Port 1
☐ Serial Port 2
☐ General Purpose Registers
☐ Special Purpose Registers
☐ Memory (RAM)
☐ Interrupt Button
☐ Serial Port 1 Registers
☐ Serial Port 2 Registers
☐ Parallel Port Registers
☐ Timer Registers
☐ Full Speed

Click Quick Load Select winix.srec

test fiber

type testfiber



```
WINIX> testfiber
context has been built
Fiber 1
Fiber 1 returning to main
Fiber 2
Fiber 2 returning to main
returned from function
WINIX> _
```

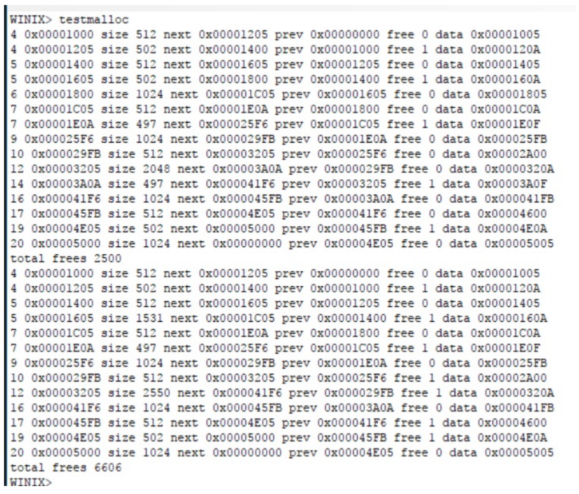
```
ucontext_t mcontext,fcontext,econtext;
int x = 0;

void func(int arg) {

  printf("Fiber %d\n",arg);
  x++;
  printf("Fiber %d returning to main\n",arg);
  setcontext(&mcontext);
}

int test_fiber(int argc, char **argv){
        int  value = 3;
        getcontext(&fcontext);
        if ((fcontext.ss_sp = (uint32_t *) malloc(1000)) != NULL) {
                fcontext.ss_sp += 1000;
                fcontext.ss_size = 1000;
                fcontext.ss_flags = 0;
                makecontext(&fcontext,func,1,1);
        }

        if ((econtext.ss_sp = (uint32_t *) malloc(1000)) != NULL) {
                econtext.ss_sp += 1000;
                econtext.ss_size = 1000;
                econtext.ss_flags = 0;
                makecontext(&econtext,func,1,2);
        }

        printf("context has been built\n");
        swapcontext(&mcontext,&fcontext);
        swapcontext(&mcontext,&econtext);
        if (!x) {
                printf("incorrect return from swapcontext");
        }
        else {
                printf("returned from function\n");
        }
        return 0;
}
```

test malloc

type testmalloc

```
          void *p0 = malloc(512);
          void *p1 = malloc(512);
          void *p2 = malloc(1024);
          void *p3 = malloc(512);
          void *p4 = malloc(1024);
          void *p5 = malloc(2048);
          void *p6 = malloc(512);
          void *p7 = malloc(1024);
          void *p8 = malloc(512);
          void *p9 = malloc(1024);
          block_overview();
          free(p5);
          free(p6);
          free(p2);
          free(p8);
          block_overview();
```

The result should be as follow

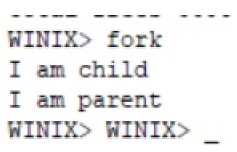


```
WINIX> testmalloc
4 0x00001000 size 512 next 0x00001205 prev 0x00000000 free 0 data 0x00001005
4 0x00001205 size 502 next 0x00001400 prev 0x00001000 free 1 data 0x0000120A
5 0x00001400 size 512 next 0x00001605 prev 0x00001205 free 0 data 0x00001405
5 0x00001605 size 502 next 0x00001800 prev 0x00001400 free 1 data 0x0000160A
6 0x00001800 size 1024 next 0x00001C05 prev 0x00001605 free 0 data 0x00001805
7 0x00001C05 size 512 next 0x00001E0A prev 0x00001800 free 0 data 0x00001C0A
7 0x00001E0A size 497 next 0x000025F6 prev 0x00001C05 free 1 data 0x00001E0F
9 0x000025F6 size 1024 next 0x000029FB prev 0x00001E0A free 0 data 0x000025FB
10 0x000029FB size 512 next 0x00003205 prev 0x000025F6 free 0 data 0x00002A00
12 0x00003205 size 2048 next 0x00003A0A prev 0x000029FB free 0 data 0x0000320A
14 0x00003A0A size 497 next 0x000041F6 prev 0x00003205 free 1 data 0x00003A0F
16 0x000041F6 size 1024 next 0x000045FB prev 0x00003A0A free 0 data 0x000041FB
17 0x000045FB size 512 next 0x00004E05 prev 0x000041F6 free 0 data 0x00004600
19 0x00004E05 size 502 next 0x00005000 prev 0x000045FB free 1 data 0x00004E0A
20 0x00005000 size 1024 next 0x00000000 prev 0x00004E05 free 0 data 0x00005005
total frees 2500
4 0x00001000 size 512 next 0x00001205 prev 0x00000000 free 0 data 0x00001005
4 0x00001205 size 502 next 0x00001400 prev 0x00001000 free 1 data 0x0000120A
5 0x00001400 size 512 next 0x00001605 prev 0x00001205 free 0 data 0x00001405
5 0x00001605 size 1531 next 0x00001C05 prev 0x00001400 free 1 data 0x0000160A
7 0x00001C05 size 512 next 0x00001E0A prev 0x00001800 free 0 data 0x00001C0A
7 0x00001E0A size 497 next 0x000025F6 prev 0x00001C05 free 1 data 0x00001E0F
9 0x000025F6 size 1024 next 0x000029FB prev 0x00001E0A free 0 data 0x000025FB
10 0x000029FB size 512 next 0x00003205 prev 0x000025F6 free 1 data 0x00002A00
12 0x00003205 size 2550 next 0x000041F6 prev 0x000029FB free 1 data 0x0000320A
16 0x000041F6 size 1024 next 0x000045FB prev 0x00003A0A free 0 data 0x000041FB
17 0x000045FB size 512 next 0x00004E05 prev 0x000041F6 free 1 data 0x00004600
19 0x00004E05 size 502 next 0x00005000 prev 0x000045FB free 1 data 0x00004E0A
20 0x00005000 size 1024 next 0x00000000 prev 0x00004E05 free 0 data 0x00005005
total frees 6606
WINIX>
```

test fork

type fork

The result should be as follow



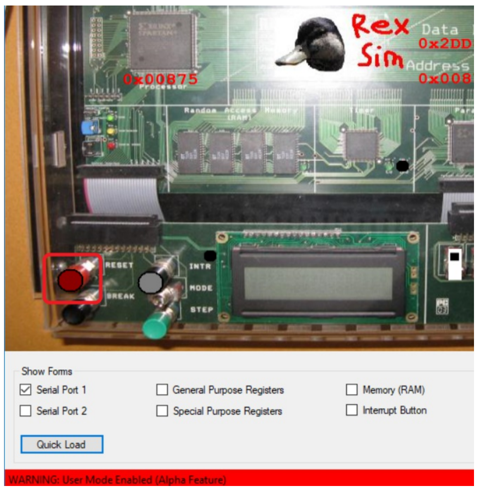
```
WINIX> fork
I am child
I am parent
WINIX> WINIX> _
```

```
int forkid = 0;
forkid = fork();
if (forkid != 0)
{
        printf("I am parent\n");
}else{
        printf("I am child\n");
}
return 0;
```

test signal

Since there are two shell programs running at the same time, so it would be impossible to continue testing

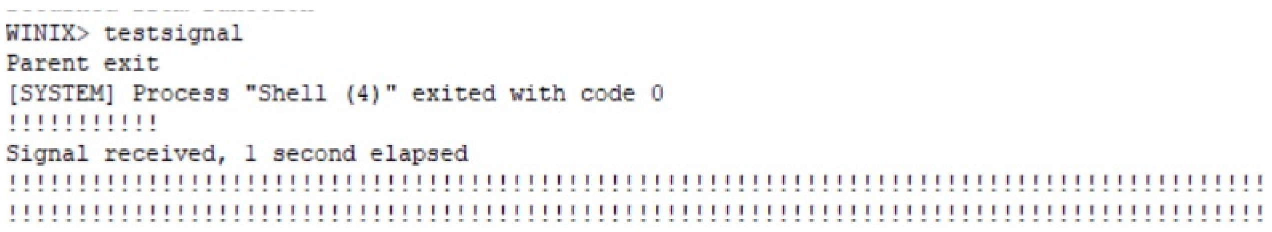
press the red button indicated below to stop the program



then Click Quick Load Select winix.srec

type testsignal

the signal should be delivered 1 second after, shown as below



```
WINIX> testsignal
Parent exit
[SYSTEM] Process "Shell (4)" exited with code 0
!!!!!!!!!!!
Signal received, 1 second elapsed
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

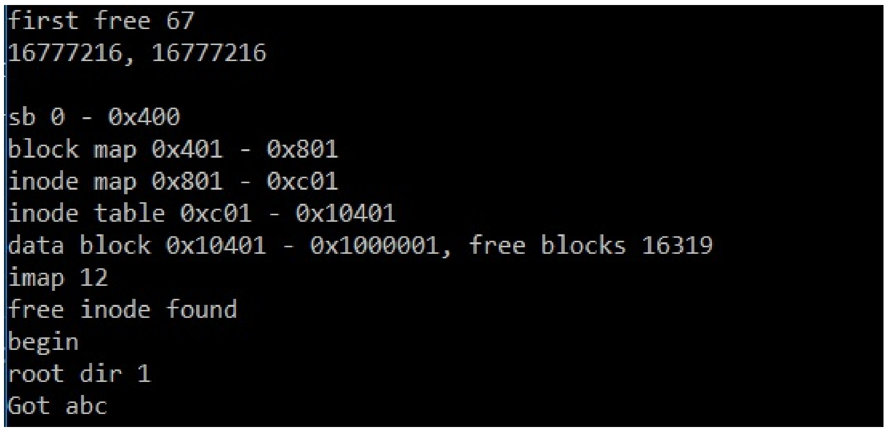
What happended is that the parent shell fork itself, then the parent shell exit. The child shell set up the signal and alarm, then wait for the signal to be delivered.

```
void sighandler(int signum){

        printf("\nSignal received, 1 second elapsed \n");
}

int test_signal(int argc, char **argv){
        int i;
        if(!fork()){
                signal(SIGALRM,sighandler);
                alarm(1);
                i = 10000;
                while(1){
                        while(i--){

                        }
                        putc('!');
                        i = 10000;
                }
        }else{
                printf("Parent exit");
                sys_exit(0);
        }
        return 0;
}
```

test file system

cd fs gcc *.c ./a.out

The result should be as follows



```
first free 67
16777216, 16777216

sb 0 - 0x400
block map 0x401 - 0x801
inode map 0x801 - 0xc01
inode table 0xc01 - 0x10401
data block 0x10401 - 0x1000001, free blocks 16319
imap 12
free inode found
begin
root dir 1
Got abc
```

```
	int fd = sys_open("/abc.txt",O_CREAT);
    sys_write(fd,"abc",4);
    sys_close(fd);

    char buf[4];
    fd = sys_open("/abc.txt",O_RDONLY);
    sys_read(fd,buf,4);
    sys_close(fd);

    printf("Got %s\n",buf);
```

side notes

You can also play around with the shell by typing the folloing commands

uptime

ps

exit