Exercise 8.

We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment. Remember the octal number should begin with '0'.

```
case 'o':
        putch('0', putdat);
        num = getuint(&ap, lflag);
        base = 8;
        goto number;
```

按照 case 'x'的写法，但是 case 'o'需要在最前面加'0'

Exercise 9.

You need also to add support for the "+" flag, which forces to precede the result with a plus or minus sign (+ or -) even for positive numbers.

```
int sign = 0;
case '+':
            sign = 1;
            goto reswitch;
```

```
if(sign == 1 && (long long) num > 0) putch('+', putdat);
```

设置一个默认为 0 的 sign 值，当读到'+'时，将其置为 1，然后在读数的时候确认。

在十进制输出（%+d）情况下，如果 sign 是 1 且读到的 num 是正数，那么在数字前加'+'（由于负数输出已经有'-'，所以不需要加）

Exercise 10.

Enhance the cprintf function to allow it print with the %n specifier, you can consult the %n specifier specification of the C99 printf function for your reference by typing "man 3 printf" on the console. In this lab, we will use the char * type argument instead of the C99 int * argument, that is, "the number of characters written so far is stored into thesigned char type integer indicated by the char * pointer argument. No argument is converted." You must deal with some special cases properly, because we are in kernel, such as when the argument is a NULL pointer, or when the char integer pointed by the argument has been overflowed. Find and fill in this code fragment.

```
int number = 0;
static void
putch(int ch, int *cnt)
{
	cputchar(ch);
	(*cnt)++;
	number++;
}
```

```
case 'n': {
const char *null_error = "\nerror! writing through NULL pointer!
(%n argument)\n";
const char *overflow_error = "\nwarning! The value %n
argument pointed to has been overflowed!\n";
char *a = va_arg(ap, char *);
if (a == NULL)
	printfmt(putch, putdat, "%s", null_error);
	else
	{
	*a = number;
	if(number > 127 || number < 0) printfmt(putch, putdat,
"%s", overflow_error);
	}
	break;
}
```

由于计算输出到屏幕上的字符数量，所以在 putch 时累加最为准确。

在 printf.c 中设置全局变量 number 并在每次调用 putch 时累加

在 printfmt.c 中引用 number，当%n 时，如果不是空指针，则把指针内容赋值为 number。

Exercise 11.

Modify the function printnum() in lib/printfmt.c to support "%-" when printing numbers. With the directives starting with "%-", the printed number should be left adjusted. (i.e., paddings are on the right side.)

```
static void
printnum(void (*putch)(int, void*), void *putdat, unsigned long long num, unsigned base, int width, int padc)
{
    if(padc == '-')
        putch("0123456789abcdef"[num % base], putdat);
        if (num >= base)   printnum(putch, putdat, num / base, base, width - 1, padc);
        else
        while (--width > 0)
            if(padc == '-') putch(' ', putdat);
            else putch(padc, putdat);

    if(padc != '-')
        putch("0123456789abcdef"[num % base], putdat);
}
```

当出现%-的情况时，左对齐，因为是递归输出数字，所以先递归地输出数字，再填充对齐符号，在%-的情况下，对齐符号为' '

Exercise 14.

Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused. When you think you have it working right, run make grade to see if its output conforms to what our grading script expects, and fix it if it doesn't. After you have handed in your Lab 1 code, you are welcome to change the output format of the backtrace function any way you like.

Exercise 15.

Modify your stack backtrace function to display, for each eip, the function name, source file name, and line number corresponding to that eip.

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    uint32_t *ebp_addr;
    cprintf("Stack backtrace:\n");
    ebp_addr = (uint32_t*)read_ebp();
    while(ebp_addr)
    {
        uint32_t old_ebp = ebp_addr[0];
        uint32_t ret_addr = ebp_addr[1];
        struct Eipdebuginfo info;
        debuginfo_eip(ret_addr, &info);
        cprintf("     eip   %08x     ebp   %08x
    args %08x %08x %08x %08x %08x\n        %s:%d: %.*s
    +%u\n" , ret_addr, ebp_addr, ebp_addr[2], ebp_addr[3],
    ebp_addr[4],  ebp_addr[5],  ebp_addr[6],  info.eip_file,
    info.eip_line,   info.eip_fn_namelen,   info.eip_fn_name,
    ret_addr - info.eip_fn_addr);
        ebp_addr = (uint32_t *)old_ebp;
    }
    overflow_me();
    cprintf("Backtrace success\n");
    return 0;
}
```

```
if (lfun <= rfun) {
    stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
    if (lline > rline) return -1;
} else {
    int index;
    for (index = lline; index <= rline; ++index)
        if (stabs[index].n_type == N_SLINE) {
            uintptr_t stab_addr = stabs[index].n_value;
            if (stab_addr == addr) {
                lline = index;
                break;
           } else if (stab_addr > addr) {
                lline = index - 1;
                break;
           }
        }
    if (index > rline)
        return -1;
}
info->eip_line = stabs[lline].n_desc;
```

backtrace：可以用 read_ebp 函数得到当前所在函数的栈顶 ebp 指针。由于栈的结构，我们可以得到 old ebp，ret addr，以及 args，当前文件信息（通过 debuginfo_eip）。
递归的看 old ebp，当 old ebp 不是 0 的时候，说明该函数是被调用函数，可以继续递归，寻找最初的函数。

寻找 line：如果当前文件的排列顺序正常，那么可以直接调用 stab_binsearch，STAB 给 line 定了特殊值 N_SLINE
如果当前文件排列混乱，那么只能用最基本的方式一行行寻找，确定是不是相同的内容。

Exercise 16.

Recall the buffer overflow attack in ICS Lab. Modify your start_overflow function to use a technique similar to the buffer overflow to invoke the do_overflow function. You must use the above cprintf function with the %n specifier you augmented in "Exercise 9" to do this job, or else you won't get the points of this exercise, and the do_overflow function should return normally.

```
char str[256] = {};
uint32_t *pret_addr;
uint32_t* ebp_addr = (uint32_t*)read_ebp();
uint32_t* esp_addr = (uint32_t*)read_esp();
uint32_t ebp = *ebp_addr;
uint32_t number = ((uint32_t)ebp_addr-(uint32_t)esp_addr-12);
int i;
for(i=0;i<number;i++) str[i] = 0;

str[number-1] = (ebp & 0xff000000) >> 24;
str[number-2] = (ebp & 0x00ff0000) >> 16;
str[number-3] = (ebp & 0x0000ff00) >> 8;
str[number-4] = ebp & 0x000000ff;
```

```
str[number] = 1<<4|14;
str[number+1] = 9;
str[number+2] = 1<<4;
str[number+3] = 15<<4;
cprintf("%s%n",str);
```

buffer overflow：将 ret_addr 改为 do_overflow 的位置（在 kernel.asm 中，跳过压栈步骤）且不改变 old_ebp 的值。

利用 str[256]数组，找到 ebp 与 esp 的地址（并取出 old_ebp 备份），相减再减去 cprintf 压栈参数所占大小（8），再减去 4，即为 ret_addr 位置，把该地址用 little_edian 方式改为 do_overflow 位置，其余位用 0 填充，且把 old_ebp 恢复。

Exercise 17.

In this exercise, you need to implement a rather easy "time" command. The output of the "time" is the running time (in clocks cycles) of the command. The usage of this command is like this: "time [command]".

```
int
mon_time(int argc, char **argv, struct Trapframe *tf)
{
	char* fn_name = argv[1];
	int start = 0, end = 0;
	if(strcmp(fn_name,"kerninfo") == 0)
	{
	   start = read_tsc();
	   mon_kerninfo(argc,argv,tf);
	   end = read_tsc();
	}
	else if(strcmp(fn_name,"help") == 0)
	{
	   start = read_tsc();
	   mon_help(argc,argv,tf);
	   end = read_tsc();
```

```
        }
        cprintf("kerninfo cycles: %d\n",end-start);
        return 0;
}
```

在 Command 中新加入 time，并链接 mon_time 函数。

当使用 time 时，argv[0]为 time，argv[1]即为所测试函数，依次判断是否为 Command 中函数，用 read_tsc()记录开始与结束时间，相减即为最终 cycle