作業系統工程

作業(二)

李奕承 611121212

练习 7-2

要求：參考 code/os/01-helloRVOS，在此基础上增加采用轮询方式读取控制台上输入的字符并 **回显** 在控制台上。另外用户按下回车后能够另起一行从头开始。

為了能夠讓作業系統從 UART 收到我們輸入的字，需要先知道要控制哪些 UART 控制暫存器。於是我開始查閱規格書。

RECEIVER HOLDING REGISTER (RHR)

The user can get the data received through the serial channel (pin *rxd*) reading this read-only location. Note that DLAB bit in LCR must be 0.

Depending on whether the FIFOs are implemented and enabled, this location will refer to a 1-byte register which receives the contents of the Receiver Shift Register once a character has been assembled, or to the top of a 16-word FIFO.

Before reading this register the user should check LSR for possible errors. The status shown in LSR corresponds to the character on top of the FIFO, which is the one ready to be read from RHR.

If a character less than 8 bits in width is received, the extra bits are read as '0'.

LINE STATUS REGISTER (LSR)

This register informs the user about the status of the transmitter and the receiver. In order to get information about a received character, LSR must be read before reading that received character from RHR.

Four interrupts are somewhat associated to the status reported by this register: the Received Data Ready, Reception Time-out, Receiver Line Status and THR Empty interrupts. Refer to the description of the ISR for details.

- bit 0: This is the Data Ready bit. It is set if one of more characters have been received and are waiting in the receiver's FIFO for the user to read them. It is zero if there is no available data in the receiver's FIFO. In case that the 16-character FIFO is not active the same description holds for the 1-character RHR register.

101	R	Line Status Register	LSR	FIFO data Error	Transmitter Empty	THR Empty	Break Interrupt	Framing Error	Parity Error	Overrun Error	Data Ready	60

從上面這些敘述中瞭解到，如果要存取 RHR 中收到的訊號，要先檢查 LSR 的第 0 個 bit 是否為 1

於是我開始寫用來收訊息的函數

首先來定義 RHR 的地址替代巨集。RHR 和 THR 在規格上共用同一個地址

```
#define RHR  ((volatile unsigned char *)(0x10000000L + 0))
```

接著造一個無窮迴圈來不斷檢查是否收到資料了，並判斷用戶輸入的是哪些字

```
int uart_r()
{
    char ch;
    //因為是使用輪詢的方式，待會會進入無窮迴圈。為了能夠在輸入exit時離開迴圈，用一個字串來儲存比對命令的字元
    char temp[255];
    int count = 0;
    while(1)            //進入無窮輪詢迴圈
    {
        if(*LSR & (1 << 0) == 1)        //檢查LSR暫存器的第一個bit是否為1，1代表接收字元完成
        {
            ch = *RHR;                  //讀取RHR中收到的字元到ch變數
            if(ch == '\r')              //如果ch中的字元是 Enter (也就是 \r )
            {
                uart_c('\n');           //則使用uart_c傳送一個換行符號 ( \n )

                //下面的if，比對用戶輸入 Enter 之前所輸入的字串是不是exit
                if(temp[0] == 'e' && temp[1] == 'x' && temp[2] == 'i' && temp[3] == 't')
                {
                    //用戶輸入是exit的話，用uart_puts()，傳送 quit 到用戶的螢幕上
                    uart_puts("quit\n");
                    //離開迴圈
                    break;
                }

                for(int i=1;i<count;i++)        //離開迴圈前，把temp[]裡面的字都清空
                {
                    temp[i] = ' ';
                }
                count = 0;                      //count重置為0，給輸入Enter後的下一行使用
            }
            else                        //如果不是 Enter 符號 ( \r )
            {
                uart_c(ch);             //就把收到的資料用 uart_c(ch) 傳回使用者的螢幕上
                temp[count] = ch;       //另外也把 ch 存到 temp[255] 中
                count++;                //count做為 temp[255] 的位置計數器，往前進一個，用來存下個字元
            }
        }
    }
}
```

接著將新寫的 uart_r( ) 加到 OS 中

```
extern void uart_r(void);
```

在 uart 初始化後使用它

```
void main(void)
{
    uart_init();                        //執行 uart_init 函數初始化 UART
    uart_puts("hello\n");               //使用 uart_puts 函數傳輸 hello 字串
    uart_puts("please input , and use Enter to newline , or use exit to quit\n");
    uart_puts("-------------------------------------------------------------\n");
    uart_r();
```

編譯執行看看，游標停在最下方等待我們輸入

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/github/Operating-System-Engineering/7-2作業 (main)
$ make qemu
qemu-system-riscv32 -nographic -smp 1 -machine virt -bios none -kernel a.out
hello
please input , and use Enter to newline , or use exit to quit
---------------------------------------------------------
```

隨便亂打字，都會出現在螢幕上，代表 UART 成功收到了字元並回傳給我們

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/github/Operating-System-Engineering/7-2作業 (main)
$ make qemu
qemu-system-riscv32 -nographic -smp 1 -machine virt -bios none -kernel a.out
hello
please input , and use Enter to newline , or use exit to quit
---------------------------------------------------------
a;lsdkfd;slfkdl;fkg
```

按下 Enter 看看，成功換行

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/github/Operating-System-Engineering/7-2作業 (main)
$ make qemu
qemu-system-riscv32 -nographic -smp 1 -machine virt -bios none -kernel a.out
hello
please input , and use Enter to newline , or use exit to quit
---------------------------------------------------------
a;lsk;dfk;sd
```

輸入 exit 指令看看

```
Lyciih@DESKTOP-CR5NUFU MINGW64 ~/Desktop/github/Operating-System-Engineering/7-2作業 (main)
$ make qemu
qemu-system-riscv32 -nographic -smp 1 -machine virt -bios none -kernel a.out
hello
please input , and use Enter to newline , or use exit to quit
---------------------------------------------------------
a;lsk;dfk;sd
exit
```

按下 Enter，成功跳出迴圈，進入到 OS 的盡頭

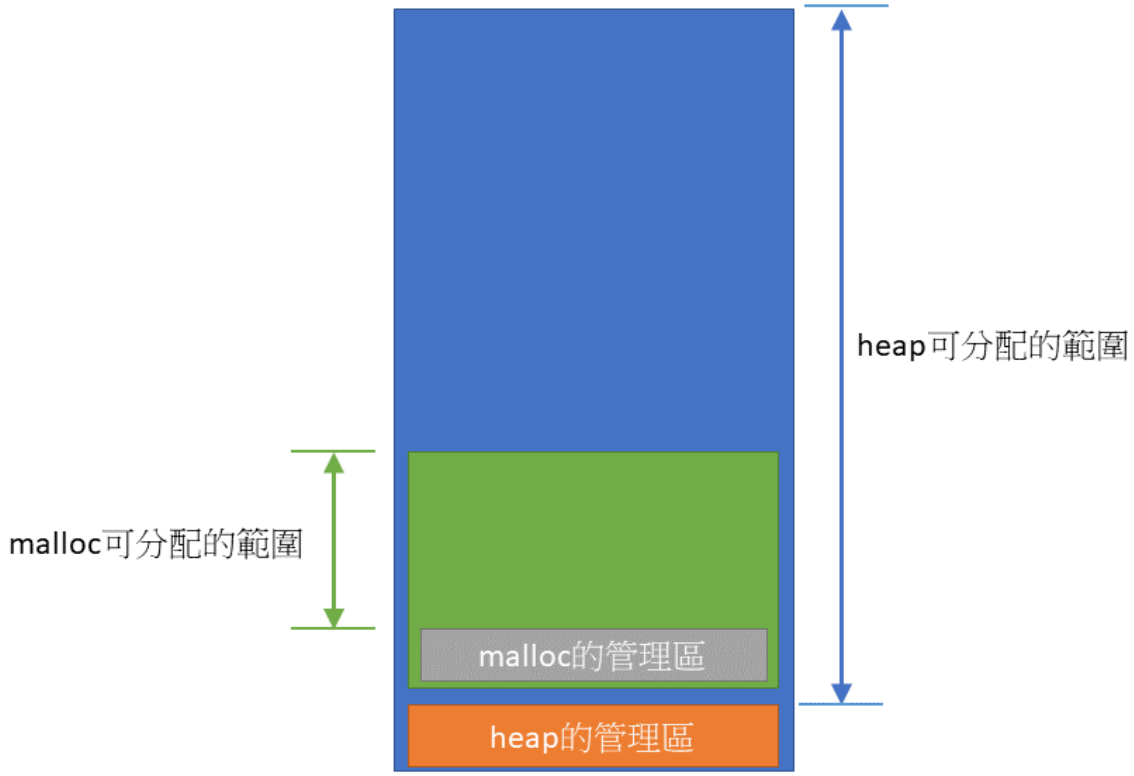
```
  don't go here
  don't go here
  don't go here
  don't go here
  don't go here
  don't go here
  don't go here
  don't go here
  don't go here
```

练习 8-1

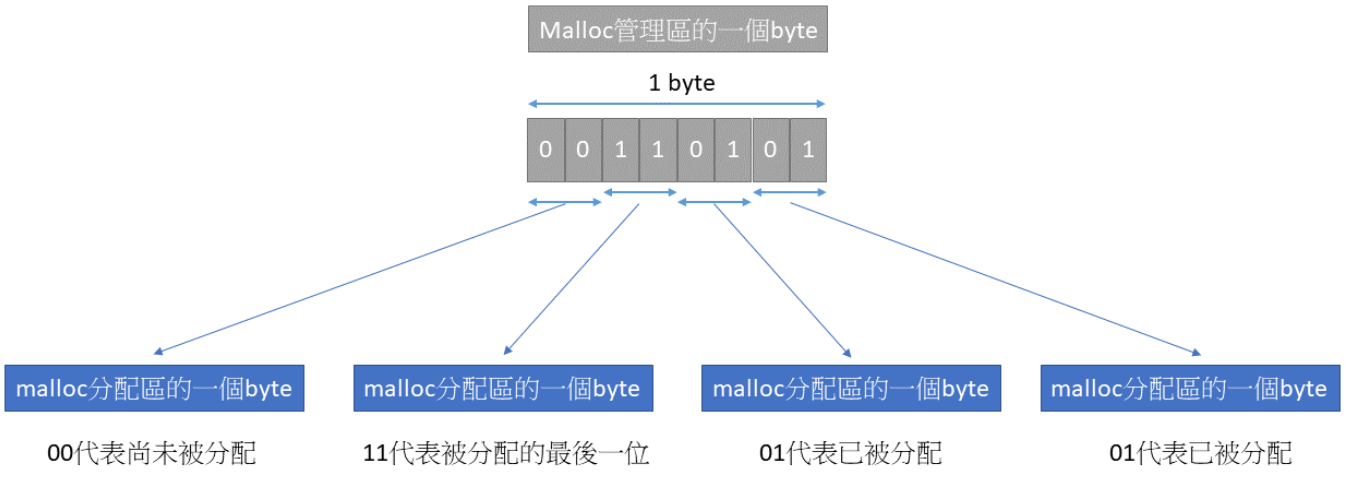
要求：参考 code/os/02-memanagement，在 page 分配的基础上实现更细颗粒度的，精确到字节为单位的内存管理。要求实现如下接口，具体描述参考 man(3) malloc：

```
void *malloc(size_t size);
void free(void *ptr);
```

我實作的 malloc 在初始化時會先用 page 申請幾個分頁做為 malloc 可分配的空間。另外我將 page 的管理模式也用在 malloc 的實踐上，所以初始化時所申請來的空間有一部份會用來管理 malloc 分配的空間。



原本的 page 管理是用一個 byte 來對映一個 page，只用掉 2 個 bit 來代表三個不同的狀態，但是這樣太浪費了，所以我在管理 malloc 時使用一個 byte 來管理 4 個 byte



使用 1 個 byte 管理 4 個 byte 的原理

宣告一個名為 offset 的 uint8_t 變數當遮罩，並將 offset 的值用來做為 while 是否繼續運行的依據

```
while(offset)
```

設定最後一個 byte 的方法

而 Free 也是利用一樣的原理

實際的程式碼太長不好排版，我將每一行都做了注釋，請老師直接看程式碼

在 OS 中使用 malloc_init 初始化並加上一段 malloc 跟 free 的測試，編譯並執行看看成果

```
printf("\n----- malloc init -----\n");
malloc_init(10);

printf("\n-----malloc test-----\n");
void *test1 = malloc(4);
void *test2 = malloc(1);
void *test3 = malloc(2);
void *test4 = malloc(3);
free(test3);
void *test5 = malloc(2);
free(test5);
void *test6 = malloc(7);
free(test4);
free(test6);
void *test7 = malloc(8);

free(test1);
free(test2);
free(test7);

test1 = malloc(4);
test2 = malloc(1);
test3 = malloc(2);
test4 = malloc(3);
test5 = malloc(2);
test6 = malloc(7);
test7 = malloc(8);
free(test1);
free(test2);
free(test3);
free(test4);
free(test5);
free(test6);
free(test7);

printf("\n----- malloc test success -----\n");
```

```
-----malloc test-----
malloc  8001b000 -> 8001b004
malloc  8001b004 -> 8001b005
malloc  8001b005 -> 8001b007
malloc  8001b007 -> 8001b00a
free  8001b005 -> 8001b007
malloc  8001b005 -> 8001b007
free  8001b005 -> 8001b007
malloc  8001b00a -> 8001b011
free  8001b007 -> 8001b00a
free  8001b00a -> 8001b011
malloc  8001b005 -> 8001b00d
free  8001b000 -> 8001b004
free  8001b004 -> 8001b005
free  8001b005 -> 8001b00d
malloc  8001b000 -> 8001b004
malloc  8001b004 -> 8001b005
malloc  8001b005 -> 8001b007
malloc  8001b007 -> 8001b00a
malloc  8001b00a -> 8001b00c
malloc  8001b00c -> 8001b013
malloc  8001b013 -> 8001b01b
free  8001b000 -> 8001b004
free  8001b004 -> 8001b005
free  8001b005 -> 8001b007
free  8001b007 -> 8001b00a
free  8001b00a -> 8001b00c
free  8001b00c -> 8001b013
free  8001b013 -> 8001b01b

----- malloc test success -----
```

測試成功，可以看到申請的位置是一路連續下去，並且在釋放後產生空位時，新的申請會優先放在第一個找到的可以放得下的區段。接著測試申請不是 5 的倍數的 page 會不會被阻擋，這裡申請 11 個 page

```
printf("\n----- malloc init -----\n");
malloc_init(11);
```

成功，並且初始化失敗後，malloc 跟 free 都被禁止

```
----- malloc init -----
page use for malloc must be a multiple of 5
malloc init fault

-----malloc test-----
malloc: malloc init fault or not yet
malloc: malloc init fault or not yet
malloc: malloc init fault or not yet
malloc: malloc init fault or not yet
free: malloc init fault or not yet
malloc: malloc init fault or not yet
```