Lab 5: Process Loading and Management

Xinyuan (Allen) Pan & Qinyu (Jack) Diao

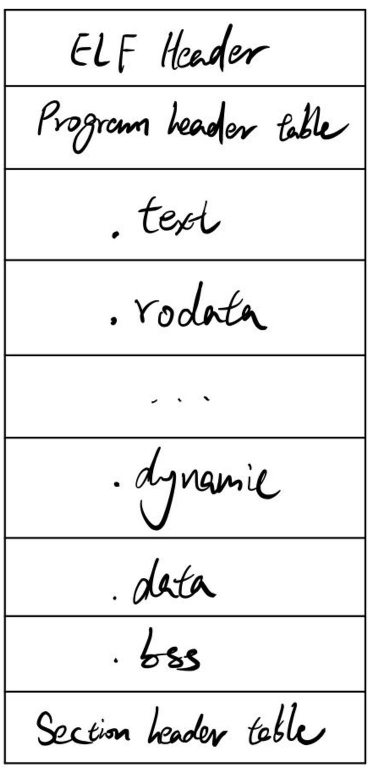
Objective

The goal of this lab is to add facilities to the OS to load a separately compiled user program from disk, dynamically create and launch an associated OS process, enable the user program to make calls to OS routines, and manage processes and system memory such that a process is removed from the system and all memory occupied by it is reclaimed when its last thread exits.

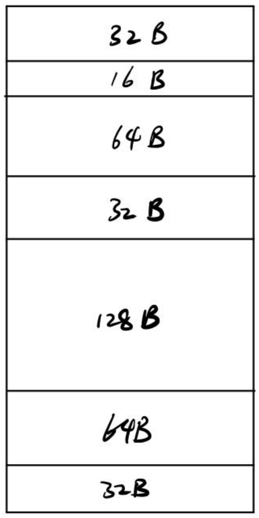
Software Design

1) Pictures illustrating the loader operation, showing:

- ELF file layout of compiled user program on disk



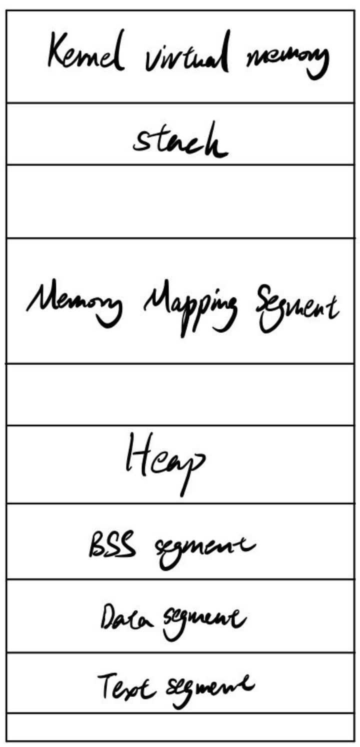
- Heap allocation scheme



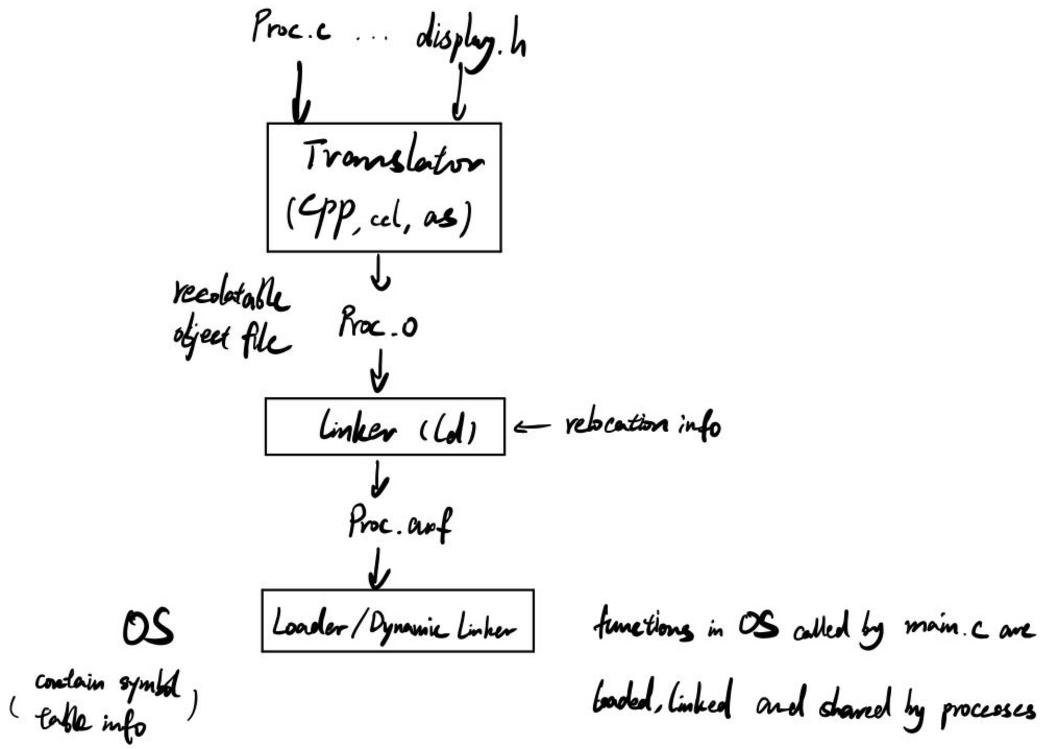
Free list segregated by size of 2^k

always allocate 2^k Bytes

- Memory layout of machine after loading the program



- Dynamic linking and relocation process



2) Operating system extensions (C and assembly), including SVC_Handler

```
SVC_Handler:    .func                       // Saves R0-R3,R12,LR,PC,PSR (top to bottom)
        PUSH    {R0,LR}
        BL              LED_RED_TOGGLE
        BL              LED_RED_TOGGLE
        POP             {R0,LR}
        LDR  R12,[SP,#24]                   // load PC to R12 (the instruction after SVC in user prog)
        LDRH R12,[R12,#-2]                  // load mem[PC-2] (the SVC instruction) to R12; SVC instruction is 2 byte
        BIC  R12,#0xFF00                    // &~ clear top bits (extract only the SVC ID)
        LDM  SP,{R0-R3}                     // load any parameters from stack into R0-R3 (auto increment SP?)

        // call corresponding OS_XXX
        PUSH    {R4,LR}                     // cannot use R0 here, cuz R0 used as return val
        CMP  R12, #0
        BEQ  JOS_Id
        CMP  R12, #1
        BEQ  JOS_Kill
        CMP  R12, #2
        BEQ  JOS_Sleep
        CMP  R12, #3
        BEQ  JOS_Time
        CMP  R12, #4
        BEQ  JOS_AddThread
```

```
JOS_Id:         BL      OS_Id
                        B   done
JOS_Kill:       BL  OS_Kill
                        B   done
JOS_Sleep:      BL  OS_Sleep
                        B   done
JOS_Time:       BL  OS_Time
                        B   done
JOS_AddThread:          BL  OS_AddThread
                                B   done
done:
        POP     {R4,LR}
        STR R0,[SP]                             // store return value
        BX LR                                   // restore R0-R3,R12,LR,PC,PSR

int OS_AddProcess(void(*entry)(void), void *text, void *data, unsigned long stackSize, unsigned long priority) {
        static uint32_t nextID = 0;
        unsigned long sr = StartCritical();
        pcbType *newPcb = Heap_Malloc(sizeof(pcbType));
        if (!newPcb) {
                EndCritical(sr);
                return 0;
        }

        newPcb->pid = nextID;
        if (text) newPcb->text = text;                  // for OS created processes that do not have text and data on
heap
        else newPcb->text = Heap_Malloc(4);
        if (data) newPcb->data = data;
        else newPcb->data = Heap_Malloc(4);

        newPcb->threadNum = 0;
        dataPt = newPcb->data;
        pcbType *temp = pcbPt;  // when adding new process, the thread needs to be added to the new process not
current running process
        pcbPt = newPcb;                         // therefore create a temporary pcb to store the current running process
        int in = OS_AddThread(entry, stackSize, priority);  // what's the priority here?
        if (!in) {
                killProcess(pcbPt);
                pcbPt = temp;
                EndCritical(sr);
                return 0;
        }

        pcbPt = temp;
        nextID++;
        EndCritical(sr);
        return 1;
}
```

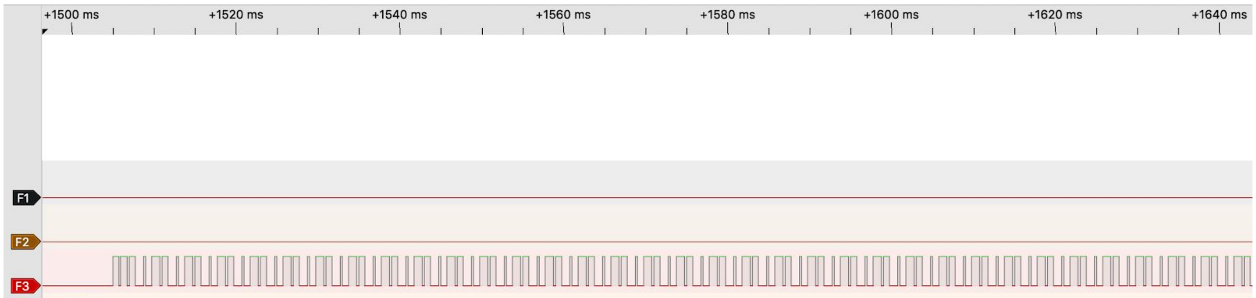
```c
// inside setInitialStack
  Stacks[i][STACKSIZE-11] = (int32_t) dataPt;  // R9

/*
 *          Thread Control Block structure
 */
typedef struct tcb {
        int32_t *sp;         // ** MUST be the first field ** saved stack pointer (not used by active thread)
        struct tcb *next;    // ** MUST be the second field
        struct tcb *prev;
        enum State state;
        int tid;
        uint32_t sleepTimeLeft;    // number of cycles left the thread needs to remain in sleep state
        Sema4Type *blocked;        // the semaphore it is blocked on
        int32_t priority;
        pcbType *pcb;
} tcbType;

/*
 * Process Control Block Structure
 */
struct pcb {
        int pid;
        void *text;
        void *data;
        int threadNum;
};
```

3) High level software system (the new interpreter commands)

```c
static void parse_load(char cmd[][20], int len) {
        if (len < 2) {
                Serial_printf("load: missing argument.\n\r");
                return;
        }
//              unsigned long startTime = OS_Time();
        uint32_t startTime = OS_MsTime();
        if (exec_elf(cmd[1], &env) != 1) {
                Serial_printf("load: exec_elf error.\n\r");
                return;
        }
//                unsigned long t = OS_TimeDifference(startTime, OS_Time());
        uint32_t t = OS_MsTime() - startTime;
        Serial_printf("load: process creation time is %u ms\r\n", t);
}
```

Measurement Data

1) Logic analyzer profile of idle task execution



2) Logic analyzer profile of user program execution
PF1: SVC traps
toggling of PF2 and PF3: LEDs by the user program's main and child threads and process completion. (The duration between the first PF2 toggle and the final PF1 toggle (OS_Kill) is the time for process completion.

PF1: process creation time

Analysis and Discussion

1) Briefly explain the dynamic memory allocation algorithm in your heap manager. Does this implementation have internal or external fragmentation?

The dynamic memory allocation in the heap manager is Knuth's Buddy allocation algorithm. It segregates heap memory into blocks of sizes of 2^k bytes. Whenever requested, it allocates the smallest 2^k number that's larger than the requested size. This way, there is only internal fragmentation and no external fragmentation.

2) How many simultaneously active processes can your system support? What factors limit this number, and how could it be increased?

Our OS can support up to 15 simultaneous active processes (one thread for each). The number of processes (or threads) is limited by the available memory, since stack has fixed size in our design. To increase the number, we can make stack growable.

We can also implement swapping mechanism, so that the data and text segments of some inactive processes can be stored in disk, and only be brought into memory whenever needed.