第三章

多道程序的放置与加载

目的/计划解决的问题

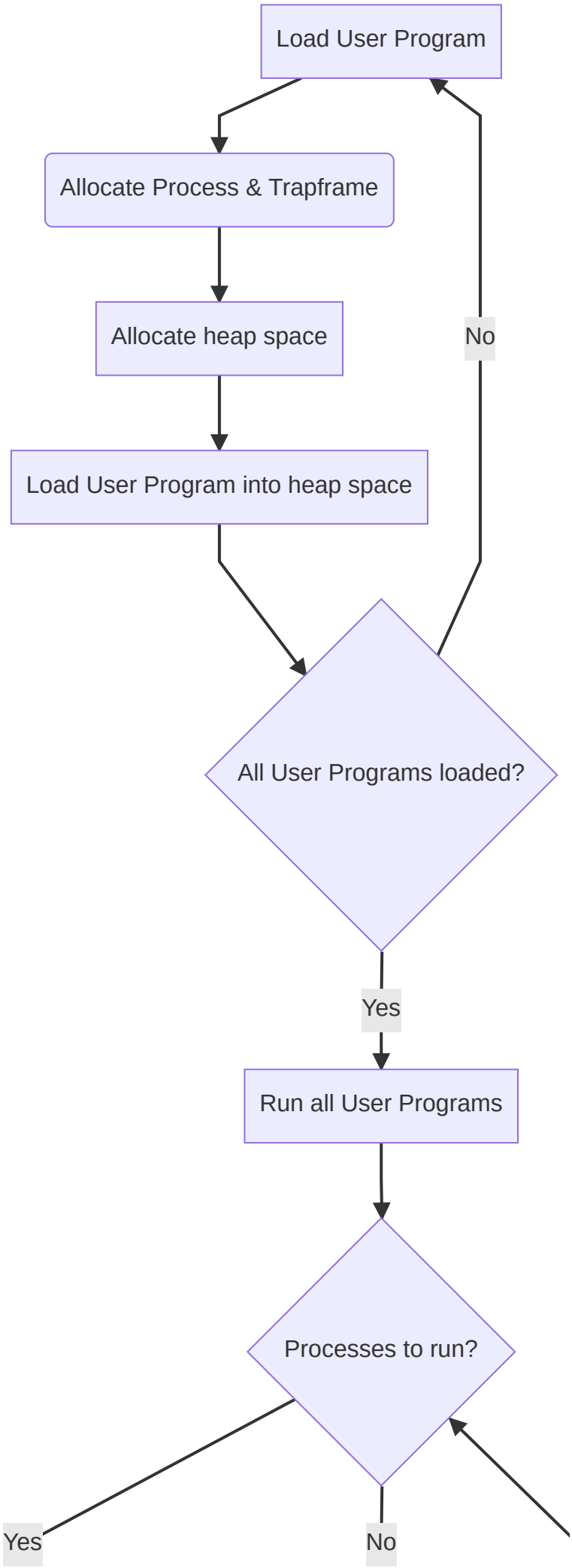
- 在系统应用程序运行时根据内存空间的动态空闲情况，将应用程序调整到合适的空闲内存空间。
- 可以使处理器响应临时的中断，并且避免无谓的外设等待来提高 CPU 利用率

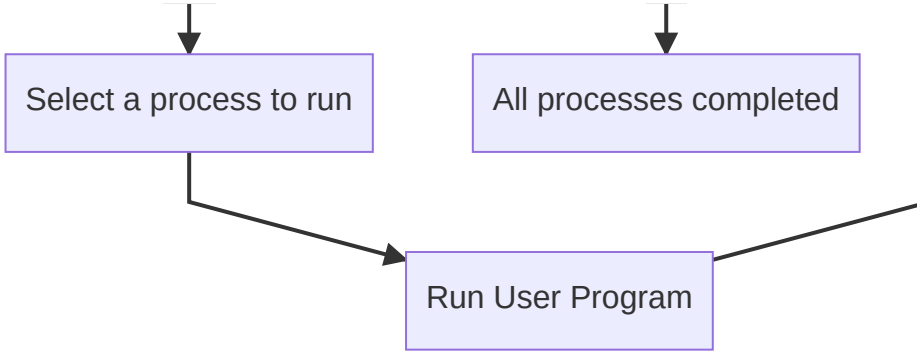
多道程序的加载

> 注：因为多道程序的放置与上一章中**问2**的链接放置在分配的内存空间的操作流程完全相同，因此这里忽略，重点关注多道程序的加载。

函数run_all_app一次性加载了所有程序，每个程序被放置在相应编号i对应的地址范围内。进程编号i用来计算相对于基地址BASE_ADDRESS的偏移量，进程使用的地址空间为 [0x80400000 + i*0x20000, 0x80400000 + (i+1)*0x20000)，每个进程的最大大小为0x20000。

```
//加载第 n 个用户应用程序
//用户程序加载的起始地址和终止地址：[BASE_ADDRESS + n * MAX_APP_SIZE, BASE_ADDRESS + (n+1) * MAX_APP_SIZE)
//每一个用户每个进程所使用的空间是 [0x80400000 + n*0x20000, 0x80400000 + (n+1)*0x20000)
int load_app(int n, uint64 *info)
{
    uint64 start = info[n], end = info[n + 1], length = end - start;
    memset((void *)BASE_ADDRESS + n * MAX_APP_SIZE, 0, MAX_APP_SIZE);
    memmove((void *)BASE_ADDRESS + n * MAX_APP_SIZE, (void *)start, length);
    return length;
}

//加载所有应用程序并初始化相应的 proc 结构。
//每一个用户每个进程所使用的空间是 [0x80400000 + i*0x20000, 0x80400000 + (i+1)*0x20000)
int run_all_app()
{
    //遍历每一个ieapp获取其放置位置
    for (int i = 0; i < app_num; ++i) {
        struct proc *p = allocproc();   //分配进程
        struct trapframe *trapframe = p->trapframe; //分配trapframe结构体
        load_app(i, app_info_ptr);  //根据是第几个用户程序，将其加载到对应位置
        uint64 entry = BASE_ADDRESS + i * MAX_APP_SIZE; //每一个应用程序的入口地址
        tracef("load app %d at %p", i, entry);
        trapframe->epc = entry; //设置程序入口地址
        trapframe->sp = (uint64)p->ustack + USER_STACK_SIZE;    //设置应用程序用户栈地址
        p->state = RUNNABLE;    //将进程设置为可运行状态
    }
    return 0;
}
```

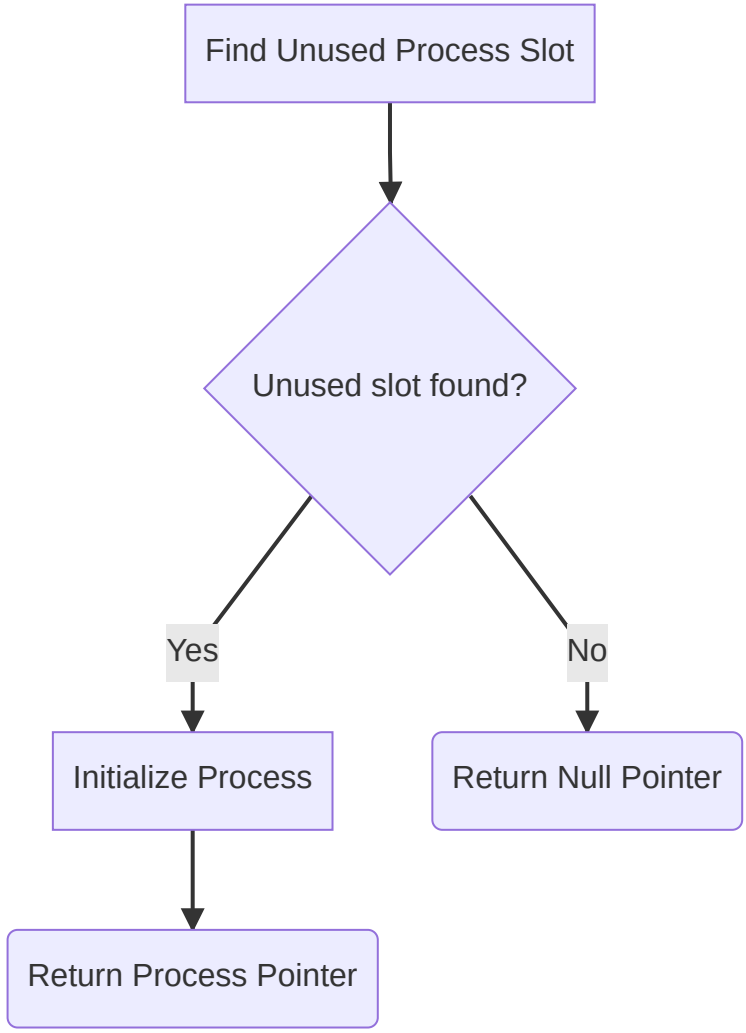

Load User Program
Allocate Process & Trapframe
Allocate heap space
Load User Program into heap space
All User Programs loaded?
No
Yes
Run all User Programs
Processes to run?
Yes
No



函数alloc_proc用于为每个用户进程分配一个名为proc的结构体。该函数的实现本质上是从进程池中选择一个未使用过（状态为UNUSED）的位置，并将proc结构体指针`p`指向该位置。具体代码如下:

```
/**
 *  @brief
 *  在进程表中查找一个UNUSED状态的进程
 *  如果在进程表中找到一个 UNUSED 进程槽，并且决定将其分配给一个新进程，
 *  那么需要对进程槽进行初始化，以便进程能够在内核态下运行。
 *  如果在进程表中没有找到可用的 UNUSED 进程槽，或者在分配内存空间时出现内存分配失败的情况，那么应该返回 0。
 *
 *
 */
proc *allocproc()
{
    proc *p;
    for (p = pool; p < &pool[NPROC]; p++) { // 同上 &pool[NPROC]
        if (p->state == UNUSED) {
            goto found;
        }
    }
    return 0;

found:
    // 初始化进程
    p->pid = allocpid();
    p->state = USED;
    p->ustack = 0;
    p->max_page = 0;
    p->parent = NULL;
    p->exit_code = 0; // 进程退出的状态码
    p->pagetable = uvmcreate((uint64)p->trapframe); // 这里需要看一下页表的部分，即ucore ch4的部分
    p->program_brk = 0; // 程序堆的顶部地址
        p->heap_bottom = 0; // 程序堆的底部地址
    // memset函数的作用是确保进程p的相关数据结构在创建或重新初始化时具有可预测的值，以避免未初始化内存中可能存在的随机数据或敏感信息的泄漏。
    memset(&p->context, 0, sizeof(p->context)); // 将进程p的context结构体全部置为0，该结构体用于保存进程上下文信息。
    memset((void *)p->kstack, 0, KSTACK_SIZE);  // 将进程p的内核栈中的内容全部清0。
```

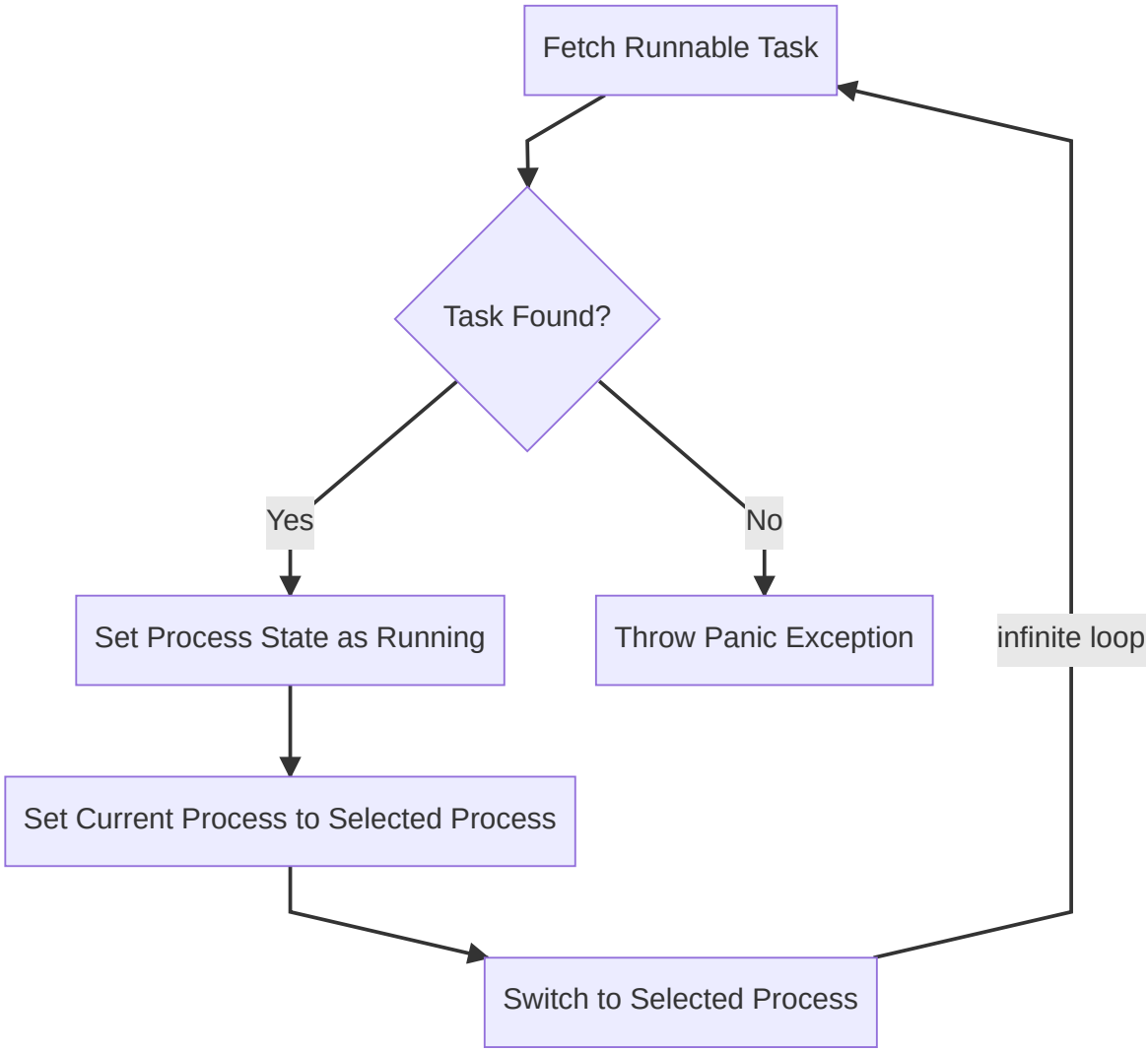
```
    memset((void *)p->trapframe, 0, TRAP_PAGE_SIZE); // 将进程p的trapframe数据结构中的内容全部置为0。
    p->context.ra = (uint64)usertrapret;
    p->context.sp = p->kstack + KSTACK_SIZE;
    return p;
}

```



开始调度

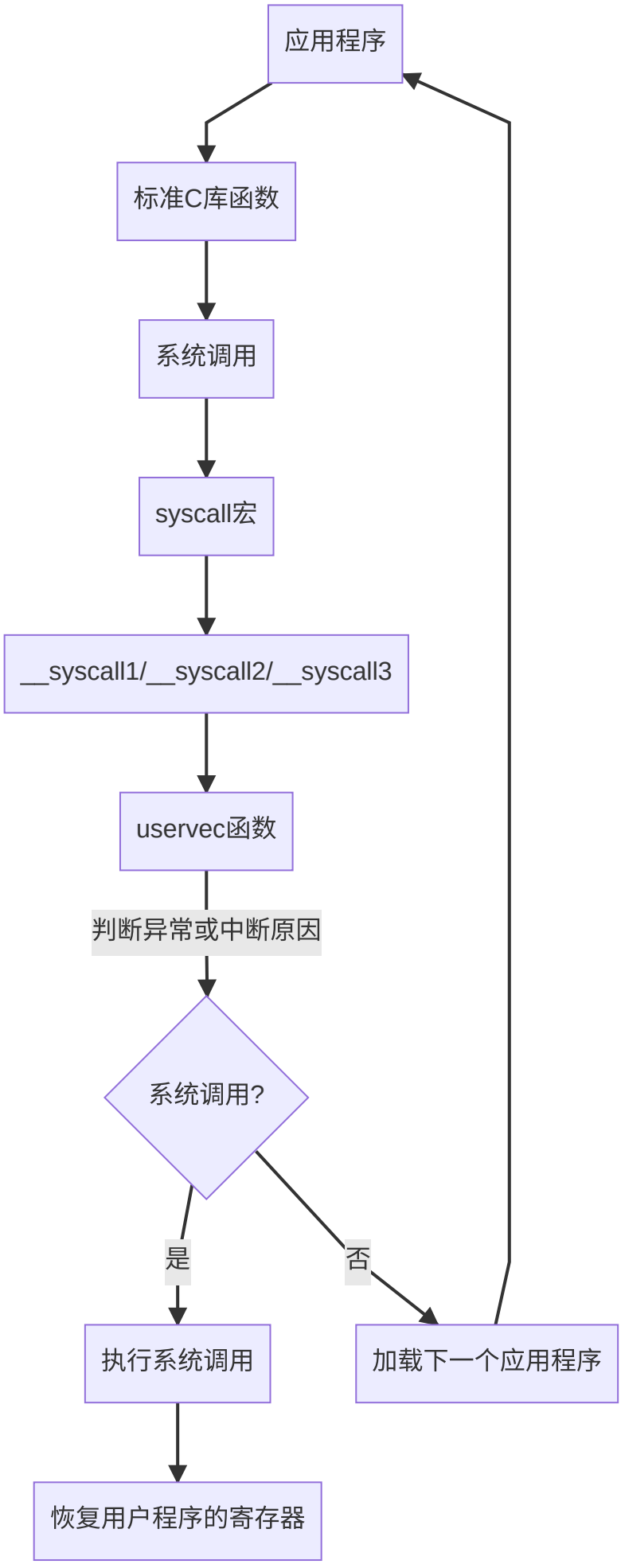
```
// os/main.c
/**
 *  void main(){
 *  ...
 *  run_all_app();
 *  scheduler();
 *  }
 */

// os/proc.c
/**
 *  @brief
 *  调度程序（Scheduler）永远不会返回。它会不断循环执行以下操作：
 *      * 选择一个进程来运行。
 *      * 使用 swtch 切换到开始运行该进程。
```

```
 *      * 最终，该进程通过 swtch 将控制权转移回调度程序。
 */
void scheduler()
{
    proc *p;
    for (;;) {
        p = fetch_task();
        if (p == NULL) {
            panic("all app are over!\n");
        }
        tracef("swtich to proc %d", p - pool);
        p->state = RUNNING;
        current_proc = p; // 将表示当前正在运行的指针指向当前进程。
        swtch(&idle.context, &p->context);
    }
}
```



> 其他有关进程的内容将在第5章里介绍

问：用户系统调用的流程是什么？


应用程序
标准C库函数
系统调用
syscall宏
__syscall1/__syscall2/__syscall3
uservec函数
判断异常或中断原因
系统调用?
是
否
执行系统调用
加载下一个应用程序
恢复用户程序的寄存器

当应用程序调用标准C库函数时，标准C库函数在user/lib中，如stdio.c、string.c等，stdio.c、string.c等文件中的标准C库调用ucore的系统调用，ucore的系统调用在user/lib/syscall.c中定义，user/lib/syscall.c中的系统调用函数会调用syscall，syscall是一个宏定义在user/lib/syscall.h中。user/lib/syscall.h中有几个宏，来获取系统调用名。如果在user/lib/syscall.c中调用syscall传递四个参数，最后就会转为调用__ syscall3。如果调用user/lib/syscall.c中调用syscall传递三个参数，最后就会转为调用__syscall2。

__ syscall1，__ syscall2，__ syscall3等函数定义在user/lib/arch/riscv/syscall_arch.h中，在__ syscall1、__ syscall2等函数中使用了ecall（异常的一种），会触发异常。然后执行异常处理函数（异常处理函数的地址在stvec中保存），即执行os/trampoline.S的uservec函数，该函数先将用户程序的各个寄存器保存，然后跳转到os/trap.c的usertrap函数，usertrap函数先读取sstatus寄存器，判断异常或中断原因，如果是系统调用就执行系统调用，然后恢复用户程序的寄存器，如果是其他异常，就直接去加载下一个应用程序，然后执行下一个应用程序。