《操作系统》

主讲教师：**翟高寿**

联系电话：**010-51684177** (办)

电子邮件：**gszhai@bjtu.edu.cn**

制作人：**翟高寿**

制作单位：**北京交通大学计算机学院**

第二章 进程管理

2.1 进程的基本概念

2.2 进程控制

2.3 进程同步

2.4 经典进程同步问题

2.5 管程

2.6 进程通信

2.7 线程

2.1 进程的基本概念

2.1.1 前趋图

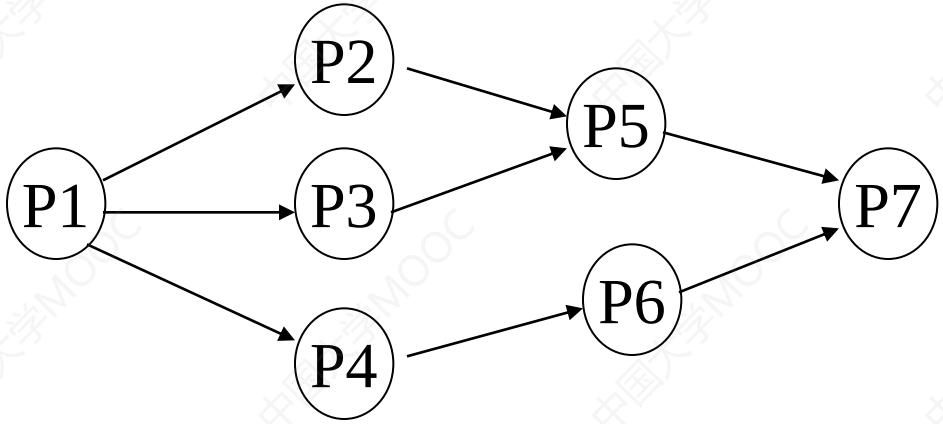
2.1.2 程序顺序执行及特征

2.1.3 程序并发执行及特征

2.1.4 进程的特征和定义

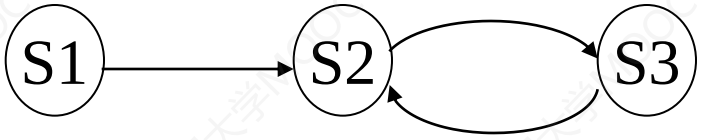
2.1.5 进程状态及状态转换图

前趋图概念及举例说明



- →={(P1,P2),(P1,P3),(P1,P4),(P2,P5),(P3,P5),(P4,P6),(P5,P7),(P6,P7)}
- Pi→Pj
  - 称Pi是Pj的直接前趋，Pj是Pi的直接后继

前趋图中必须不存在循环



- 图例中存在前趋关系S2→S3和S3→S2，显然，这种前趋关系是无法满足的。

2.1 进程的基本概念

2.1.1 前趋图

2.1.2 程序顺序执行及特征

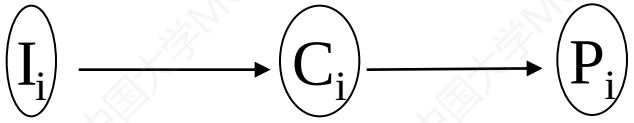
2.1.3 程序并发执行及特征

2.1.4 进程的特征和定义

2.1.5 进程状态及状态转换图

程序顺序执行

- 程序执行过程中通常存在顺序执行问题
  - 构成程序的若干个程序段之间
  - 组成程序段的多条语句之间



S1: a:=x+y;

S2: b:=a-5;

S3: c:=b+1;

程序顺序执行时的特征

单道程序系统

- 顺序性
  - 处理机的操作，严格按照规定顺序执行
- 封闭性
  - 封闭环境下运行，程序独占全机资源
  - 只有当前运行程序才能改变资源状态
  - 程序执行结果不受外界因素的影响
- 可再现性
  - 只要程序执行时的环境和初始条件相同，程序重复执行结果相同

程序顺序执行时的特性，将为程序员检测和校正程序的错误，带来极大的方便

2.1 进程的基本概念

2.1.1 前趋图

2.1.2 程序顺序执行及特征

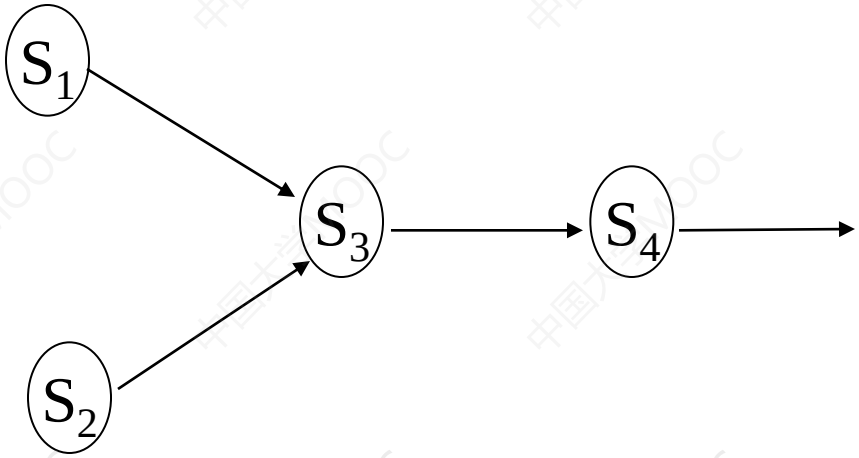
2.1.3 程序并发执行及特征

2.1.4 进程的特征和定义

2.1.5 进程状态及状态转换图

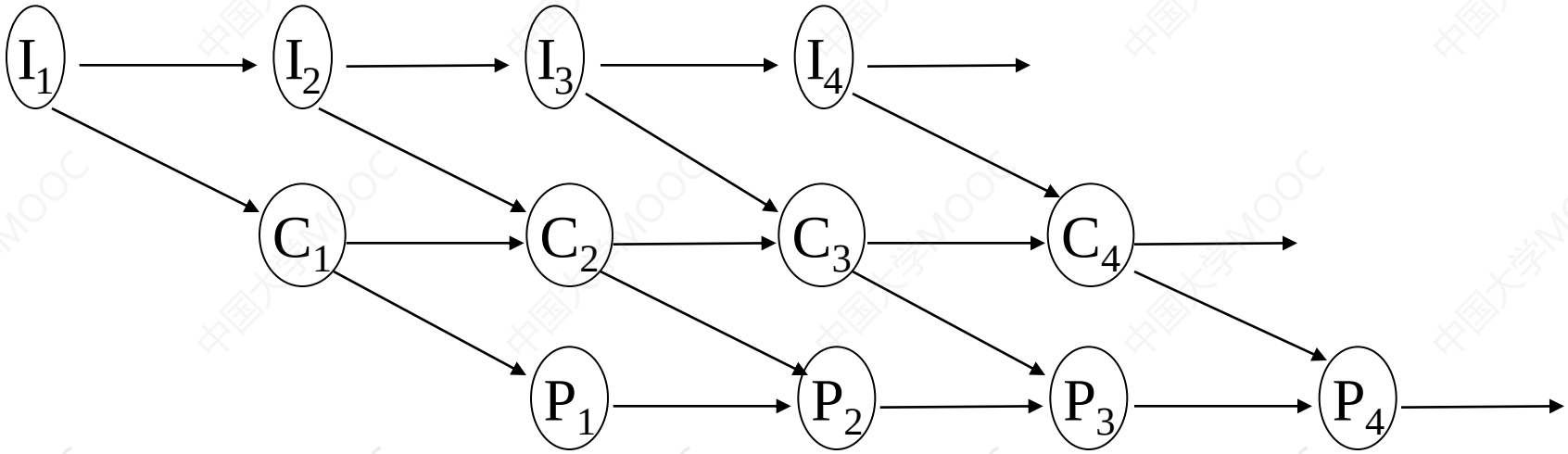
程序并发执行例1

- 程序段语句间并发执行
  - S1: a:=x+2　　S2: b:=y+4
  - S3: c:=a+b　　S4: d:=c+6



程序并发执行例2

- 一批程序$I_i \rightarrow C_i \rightarrow P_i$并发执行
  - $I_i \rightarrow I_{i+1}$　　$C_i \rightarrow C_{i+1}$　　$P_i \rightarrow P_{i+1}$



程序并发执行时的特征

多道程序系统

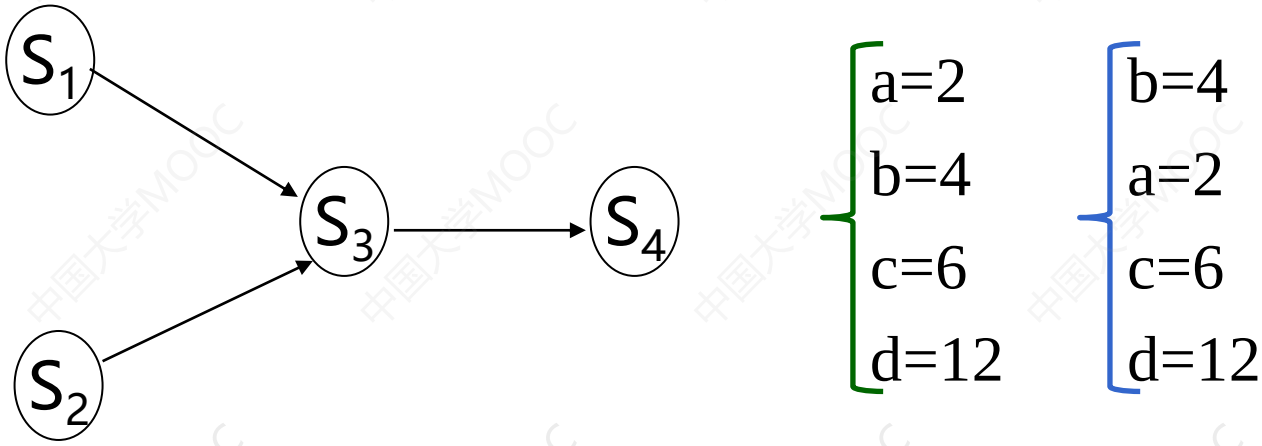
- 间断性
  - “执行—暂停执行—执行”的活动规律
- 失去封闭性
  - 系统资源共享及资源状态改变的多样性，致使程序运行失去封闭性，程序运行必然会受到其它程序的影响
- 不可再现性
  - 并发执行的程序，计算结果与其执行速度及时间有关

程序并发执行不可再现性举例

- 共享初值为0的**变量N**的两程序段A、B
  - A: N:=N+1
  - B: Print(N); N:=0
- 执行结果分析
  - 先A: 1, **1**, **0**
  - 中A: **0**, 1, **0**
  - 后A: **0**, 0, **1**

程序并发执行可再现性分析

- 程序段语句间并发执行(各变量初值均为0)
  - S1: a:=x+2　　S2: b:=y+4
  - S3: c:=a+b　　S4: d:=c+6



2.1 进程的基本概念

2.1.1 前趋图

2.1.2 程序顺序执行及特征

2.1.3 程序并发执行及特征

2.1.4 进程的特征和定义

2.1.5 进程状态及状态转换图

进程的引入

- 并发、共享及多道程序环境
- 基于程序的概念已不能完整、有效地描述并发程序在内存中的运行状态
- 必须建立并发程序的新的描述和控制机制
- 基于程序段、数据段和**进程控制块**而引入进程的概念以对应程序的运行过程
- 进程控制块存放了进程标识符、进程运行的当前状态、程序和数据的地址以及关于该程序运行时的CPU环境信息

进程的定义

- 进程是可并发执行的程序在一个数据集合上的运行过程，亦即进程实体的运行过程
  - 进程实体由程序段、数据段及进程控制块三部分构成
- 进程是系统进行资源分配和调度的一个独立单位

进程的特征——与程序的区别与联系

- 结构特征
  - 程序段、数据段及进程控制块
- 动态性
  - 生命周期及“执行” 本质
- 并发性
  - 共存于内存、宏观同时运行
- 独立性
  - 调度、资源分配、运行
- 异步性
  - 推进相互独立、速度不可预知

2.1 进程的基本概念

2.1.1 前趋图

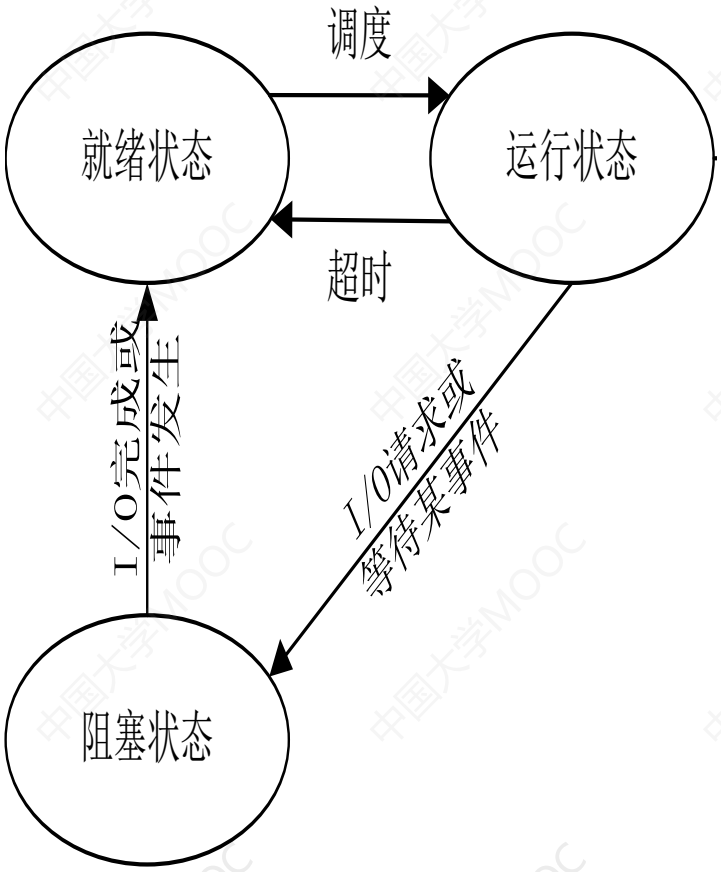
2.1.2 程序顺序执行及特征

2.1.3 程序并发执行及特征

2.1.4 进程的特征和定义

2.1.5 进程状态及状态转换图

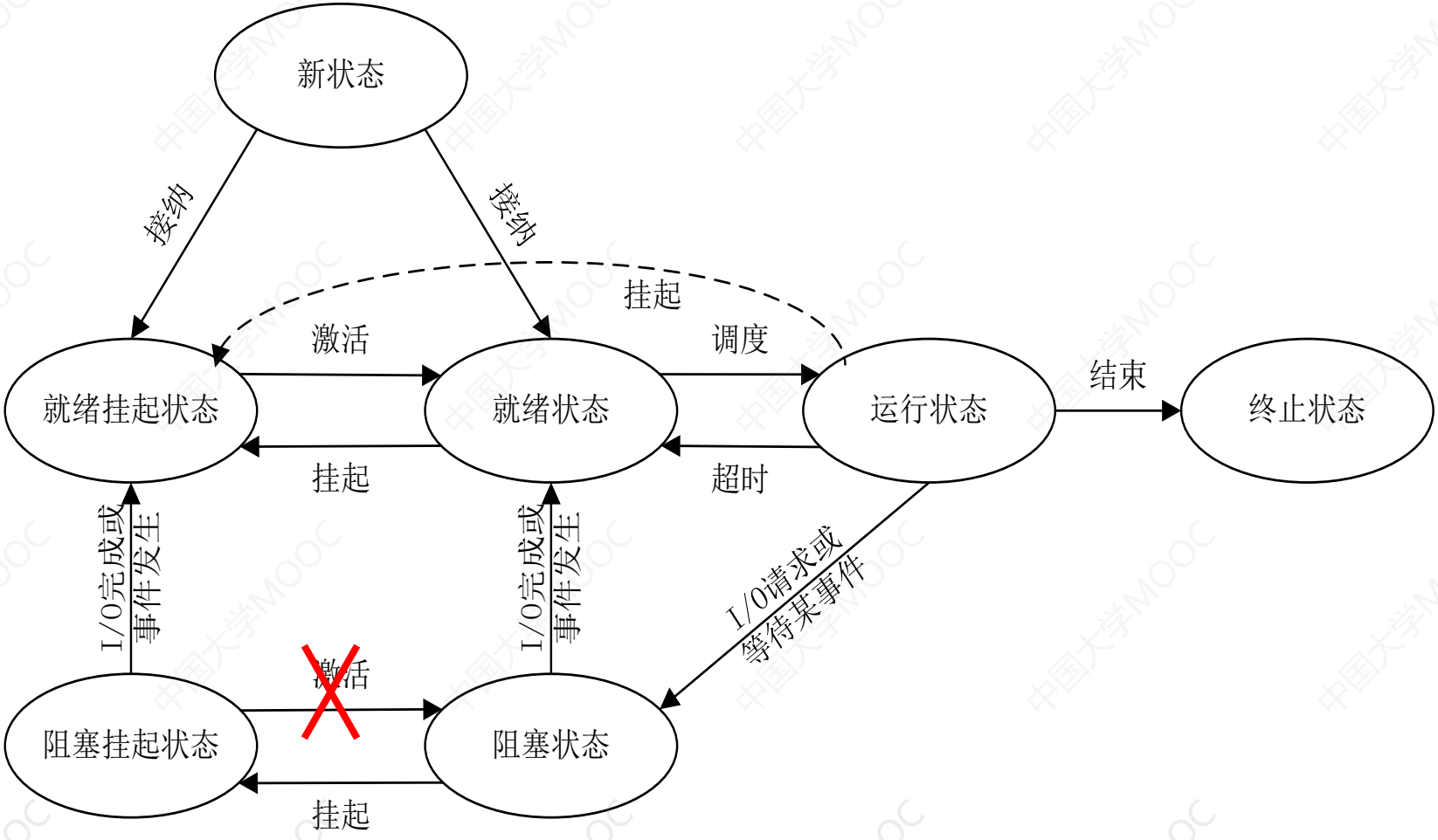
进程的基本状态及状态转换



引入挂起状态的可能原因

- 终端用户的请求
  - 程序运行期间发现可疑问题暂停进程
- 父进程的请求
  - 考察、修改或协调子进程
- 操作系统的需要
  - 运行中资源使用情况的检查和记账
- 负载调节的需要
  - 负荷调节和保证实时系统正常运行

具有挂起状态的进程状态图



2.1 进程的基本概念

2.1.1 前趋图

2.1.2 程序顺序执行及特征

2.1.3 程序并发执行及特征

2.1.4 进程的特征和定义

2.1.5 进程状态及状态转换图

作业题

- **2.1** 比较程序的顺序执行和并发执行。
- **2.2** 比较程序和进程。
- **2.3** 试对进程的状态及状态转换进行总结，注意状态转换的物理含义及转化条件。
- **2.4** 试举例说明引起进程创建、撤消、阻塞或被唤醒的主要事件分别有哪些？

第二章 进程管理

2.1 进程的基本概念

2.2 进程控制

2.3 进程同步

2.4 经典进程同步问题

2.5 管程

2.6 进程通信

2.7 线程

2.2 进程控制

2.2.1 进程控制块

2.2.2 进程图

2.2.3 进程的创建与终止

2.2.4 进程的阻塞与唤醒

2.2.5 进程的挂起与激活

2.2.6 UNIX进程描述与控制

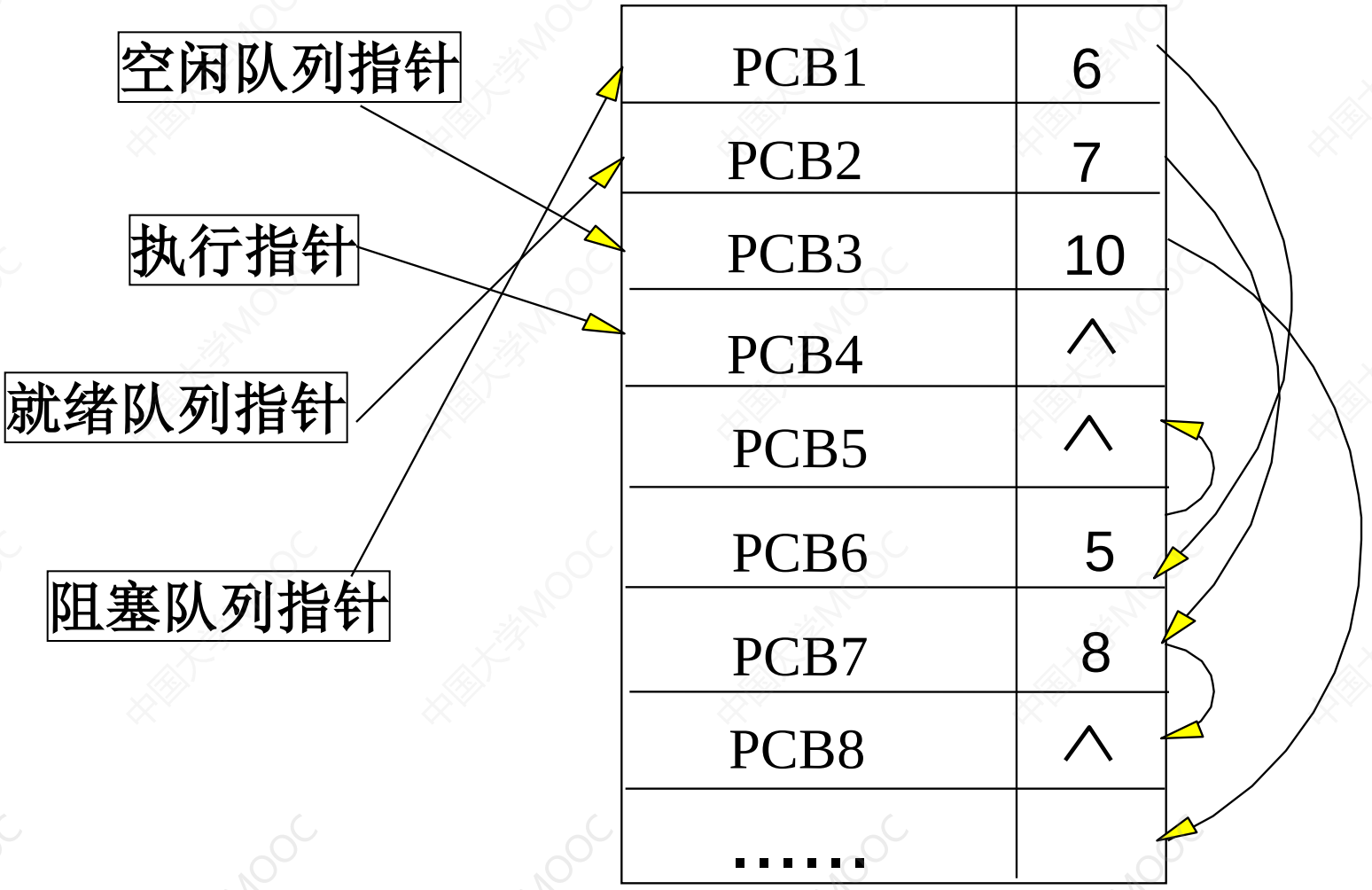
进程控制块

- 进程实体的一部分，拥有描述进程情况及控制进程运行所需的全部信息的记录性数据结构
- 使一个在多道程序环境下不能独立运行的程序，成为一个能独立运行的基本单位，一个能与其它进程并发执行的进程
- 操作系统控制和管理并发执行进程的依据
- 进程存在的惟一标志
- 常驻内存并存放于操作系统专门开辟的PCB区

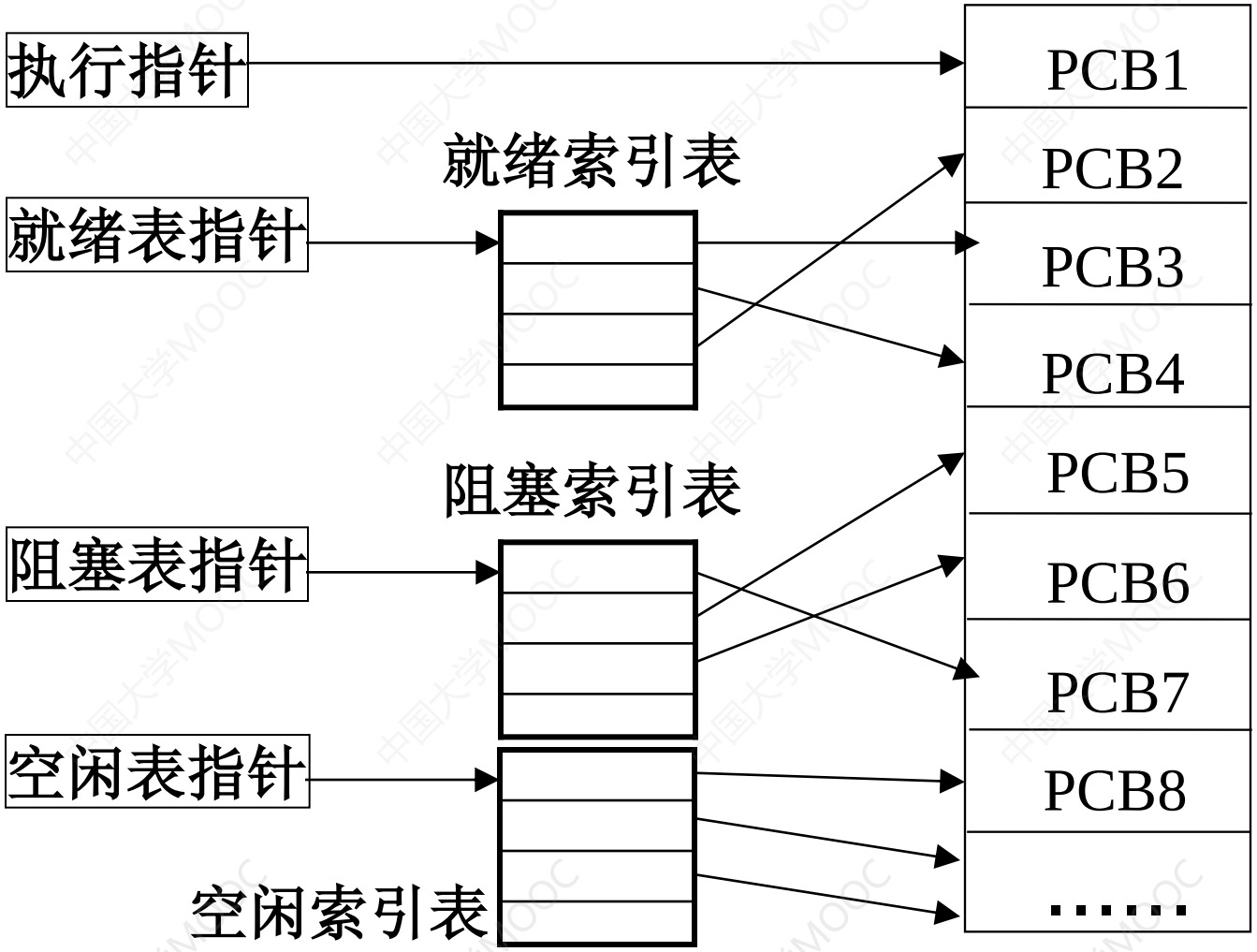
进程控制块中的信息

- 进程标识符
  - 内/外部、父/子进程、用户标识符
- 处理器状态信息
  - 通用、PC、PSW、用户栈指针寄存器
- 进程调度信息
  - 进程状态、进程优先级、事件及其它
- 进程控制信息
  - 程序和数据地址、进程同步通信机制
  - 资源清单、链接指针

进程控制块的组织方式1——链接方式



进程控制块的组织方式2——索引方式



Linux-3.2.72内核struct task_struct 1/15

```
struct task_struct {
        volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
        void *stack;
        atomic_t usage;
        unsigned int flags;     /* per process flags, defined below */
        unsigned int ptrace;

#ifdef CONFIG_SMP
        struct llist_node wake_entry;
        int on_cpu;
#endif
        int on_rq;

        int prio, static_prio, normal_prio;
        unsigned int rt_priority;
        const struct sched_class *sched_class;
        struct sched_entity se;
        struct sched_rt_entity rt;
#ifdef CONFIG_CGROUP_SCHED
        struct task_group *sched_task_group;
#endif
```

/linux-3.2.72/include/linux/sched.h

Linux-3.2.72内核struct task_struct 2/15

```
#ifdef CONFIG_PREEMPT_NOTIFIERS
        /* list of struct preempt_notifier: */
        struct hlist_head preempt_notifiers;
#endif

        /*
         * fpu_counter contains the number of consecutive context switches
         * that the FPU is used. If this is over a threshold, the lazy fpu
         * saving becomes unlazy to save the trap. This is an unsigned char
         * so that after 256 times the counter wraps and the behavior turns
         * lazy again; this to deal with bursty apps that only use FPU for
         * a short time
         */
        unsigned char fpu_counter;
#ifdef CONFIG_BLK_DEV_IO_TRACE
        unsigned int btrace_seq;
#endif

        unsigned int policy;
        cpumask_t cpus_allowed;

#ifdef CONFIG_PREEMPT_RCU
        int rcu_read_lock_nesting;
        char rcu_read_unlock_special;
        struct list_head rcu_node_entry;
#endif /* #ifdef CONFIG_PREEMPT_RCU */
```

/linux-3.2.72/include/linux/sched.h

Linux-3.2.72内核struct task_struct 3/15

```
#ifdef CONFIG_TREE_PREEMPT_RCU
        struct rcu_node *rcu_blocked_node;
#endif /* #ifdef CONFIG_TREE_PREEMPT_RCU */
#ifdef CONFIG_RCU_BOOST
        struct rt_mutex *rcu_boost_mutex;
#endif /* #ifdef CONFIG_RCU_BOOST */

#if defined(CONFIG_SCHEDSTATS) || defined(CONFIG_TASK_DELAY_ACCT)
        struct sched_info sched_info;
#endif

        struct list_head tasks;
#ifdef CONFIG_SMP
        struct plist_node pushable_tasks;
#endif

        struct mm_struct *mm, *active_mm;
#ifdef CONFIG_COMPAT_BRK
        unsigned brk_randomized:1;
#endif
#if defined(SPLIT_RSS_COUNTING)
        struct task_rss_stat    rss_stat;
#endif
```

/linux-3.2.72/include/linux/sched.h

Linux-3.2.72内核struct task_struct 4/15

```
/* task state */
        int exit_state;
        int exit_code, exit_signal;
        int pdeath_signal;  /*  The signal sent when the parent dies  */
        unsigned int jobctl;    /* JOBCTL_*, siglock protected */
        /* ??? */
        unsigned int personality;
        unsigned did_exec:1;
        unsigned in_execve:1;   /* Tell the LSMs that the process is doing
an
                                 * execve */
        unsigned in_iowait:1;


        /* Revert to default priority/policy when forking */
        unsigned sched_reset_on_fork:1;
        unsigned sched_contributes_to_load:1;

        pid_t pid;
        pid_t tgid;

#ifdef CONFIG_CC_STACKPROTECTOR
        /* Canary value for the -fstack-protector gcc feature */
        unsigned long stack_canary;
#endif
```

/linux-3.2.72/include/linux/sched.h

Linux-3.2.72内核struct task_struct 5/15

```
        /*
         * pointers to (original) parent process, youngest child, younger
sibling,
         * older sibling, respectively.  (p->father can be replaced with
         * p->real_parent->pid)
         */
        struct task_struct *real_parent; /* real parent process */
        struct task_struct *parent; /* recipient of SIGCHLD, wait4()
reports */
        /*
         * children/sibling forms the list of my natural children
         */
        struct list_head children;      /* list of my children */
        struct list_head sibling;       /* linkage in my parent's children
list */
        struct task_struct *group_leader;       /* threadgroup leader */

        /*
         * ptraced is the list of tasks this task is using ptrace on.
         * This includes both natural children and PTRACE_ATTACH targets.
         * p->ptrace_entry is p's link on the p->parent->ptraced list.
         */
        struct list_head ptraced;
```

/linux-3.2.72/include/linux/sched.h

Linux-3.2.72内核struct task_struct 6/15

```
        struct list_head ptrace_entry;

        /* PID/PID hash table linkage. */
        struct pid_link pids[PIDTYPE_MAX];
        struct list_head thread_group;

        struct completion *vfork_done;          /* for vfork() */
        int __user *set_child_tid;              /* CLONE_CHILD_SETTID */
        int __user *clear_child_tid;            /* CLONE_CHILD_CLEARTID */

        cputime_t utime, stime, utimescaled, stimescaled;
        cputime_t gtime;
#ifndef CONFIG_VIRT_CPU_ACCOUNTING
        cputime_t prev_utime, prev_stime;
#endif
        unsigned long nvcsw, nivcsw; /* context switch counts */
        struct timespec start_time;             /* monotonic time */
        struct timespec real_start_time;        /* boot based time */
/* mm fault and swap info: this can arguably be seen as either mm-specific
or thread-specific */
        unsigned long min_flt, maj_flt;

        struct task_cputime cputime_expires;
        struct list_head cpu_timers[3];
```

/linux-3.2.72/include/linux/sched.h

Linux-3.2.72内核struct task_struct 7/15

```
/* process credentials */
        const struct cred __rcu *real_cred; /* objective and real
subjective task
                                            * credentials (COW) */
        const struct cred __rcu *cred;  /* effective (overridable)
subjective task
                                            * credentials (COW) */
        struct cred *replacement_session_keyring; /* for
KEYCTL_SESSION_TO_PARENT */

        char comm[TASK_COMM_LEN]; /* executable name excluding path
                                     - access with [gs]et_task_comm (which
lock
                                       it with task_lock())
                                     - initialized normally by
setup_new_exec */
/* file system info */
        int link_count, total_link_count;
#ifdef CONFIG_SYSVIPC
/* ipc stuff */
        struct sysv_sem sysvsem;
#endif
#ifdef CONFIG_DETECT_HUNG_TASK
/* hung task detection */
        unsigned long last_switch_count;
#endif
```

/linux-3.2.72/include/linux/sched.h

Linux-3.2.72内核struct task_struct 8/15

/linux-3.2.72/include/linux/sched.h

```
/* CPU-specific state of this task */
        struct thread_struct thread;
/* filesystem information */
        struct fs_struct *fs;
/* open file information */
        struct files_struct *files;
/* namespaces */
        struct nsproxy *nsproxy;
/* signal handlers */
        struct signal_struct *signal;
        struct sighand_struct *sighand;

        sigset_t blocked, real_blocked;
        sigset_t saved_sigmask; /* restored if set_restore_sigmask() was
used */
        struct sigpending pending;

        unsigned long sas_ss_sp;
        size_t sas_ss_size;
        int (*notifier)(void *priv);
        void *notifier_data;
        sigset_t *notifier_mask;
        struct audit_context *audit_context;
```

Linux-3.2.72内核struct task_struct 9/15

```
#ifdef CONFIG_AUDITSYSCALL
        uid_t loginuid;
        unsigned int sessionid;
#endif
        seccomp_t seccomp;

/* Thread group tracking */
        u32 parent_exec_id;
        u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty, keyrings,
mems_allowed,
 * mempolicy */
        spinlock_t alloc_lock;

#ifdef CONFIG_GENERIC_HARDIRQS
        /* IRQ handler threads */
        struct irqaction *irqaction;
#endif

        /* Protection of the PI data structures: */
        raw_spinlock_t pi_lock;
```

/linux-3.2.72/include/linux/sched.h

Linux-3.2.72内核struct task_struct 10/15

```
#ifdef CONFIG_RT_MUTEXES
        /* PI waiters blocked on a rt_mutex held by this task */
        struct plist_head pi_waiters;
        /* Deadlock detection and priority inheritance handling */
        struct rt_mutex_waiter *pi_blocked_on;
#endif

#ifdef CONFIG_DEBUG_MUTEXES
        /* mutex deadlock detection */
        struct mutex_waiter *blocked_on;
#endif
#ifdef CONFIG_TRACE_IRQFLAGS
        unsigned int irq_events;
        unsigned long hardirq_enable_ip;
        unsigned long hardirq_disable_ip;
        unsigned int hardirq_enable_event;
        unsigned int hardirq_disable_event;
        int hardirqs_enabled;
        int hardirq_context;
        unsigned long softirq_disable_ip;
        unsigned long softirq_enable_ip;
        unsigned int softirq_disable_event;
        unsigned int softirq_enable_event;
        int softirqs_enabled;
        int softirq_context;
#endif
```

/linux-3.2.72/include/linux/sched.h

Linux-3.2.72内核struct task_struct 11/15

/linux-3.2.72/include/linux/sched.h

```
#ifdef CONFIG_LOCKDEP
# define MAX_LOCK_DEPTH 48UL
        u64 curr_chain_key;
        int lockdep_depth;
        unsigned int lockdep_recursion;
        struct held_lock held_locks[MAX_LOCK_DEPTH];
        gfp_t lockdep_reclaim_gfp;
#endif

/* journalling filesystem info */
        void *journal_info;

/* stacked block device info */
        struct bio_list *bio_list;

#ifdef CONFIG_BLOCK
/* stack plugging */
        struct blk_plug *plug;
#endif

/* VM state */
        struct reclaim_state *reclaim_state;

        struct backing_dev_info *backing_dev_info;

        struct io_context *io_context;

        unsigned long ptrace_message;
        siginfo_t *last_siginfo; /* For ptrace use.  */
        struct task_io_accounting ioac;
```

Linux-3.2.72内核struct task_struct 12/15

```
#if defined(CONFIG_TASK_XACCT)
        u64 acct_rss_mem1;      /* accumulated rss usage */
        u64 acct_vm_mem1;       /* accumulated virtual memory usage */
        cputime_t acct_timexpd; /* stime + utime since last update */
#endif
#ifdef CONFIG_CPUSETS
        nodemask_t mems_allowed;        /* Protected by alloc_lock */
        seqcount_t mems_allowed_seq;    /* Seqence no to catch updates */
        int cpuset_mem_spread_rotor;
        int cpuset_slab_spread_rotor;
#endif
#ifdef CONFIG_CGROUPS
        /* Control Group info protected by css_set_lock */
        struct css_set __rcu *cgroups;
        /* cg_list protected by css_set_lock and tsk->alloc_lock */
        struct list_head cg_list;
#endif
#ifdef CONFIG_FUTEX
        struct robust_list_head __user *robust_list;
#ifdef CONFIG_COMPAT
        struct compat_robust_list_head __user *compat_robust_list;
#endif
        struct list_head pi_state_list;
        struct futex_pi_state *pi_state_cache;
#endif
```

/linux-3.2.72/include/linux/sched.h

Linux-3.2.72内核struct task_struct 13/15

```
#ifdef CONFIG_PERF_EVENTS
        struct perf_event_context *perf_event_ctxp[perf_nr_task_contexts];
        struct mutex perf_event_mutex;
        struct list_head perf_event_list;
#endif
#ifdef CONFIG_NUMA
        struct mempolicy *mempolicy;    /* Protected by alloc_lock */
        short il_next;
        short pref_node_fork;
#endif
        struct rcu_head rcu;

        /*
         * cache last used pipe for splice
         */
        struct pipe_inode_info *splice_pipe;
#ifdef  CONFIG_TASK_DELAY_ACCT
        struct task_delay_info *delays;
#endif
#ifdef CONFIG_FAULT_INJECTION
        int make_it_fail;
#endif
        /*
         * when (nr_dirtied >= nr_dirtied_pause), it's time to call
         * balance_dirty_pages() for some dirty throttling pause
         */
        int nr_dirtied;
        int nr_dirtied_pause;
```

/linux-3.2.72/include/linux/sched.h

Linux-3.2.72内核struct task_struct 14/15

```
#ifdef CONFIG_LATENCYTOP
        int latency_record_count;
        struct latency_record latency_record[LT_SAVECOUNT];
#endif
        /*
         * time slack values; these are used to round up poll() and
         * select() etc timeout values. These are in nanoseconds.
         */
        unsigned long timer_slack_ns;
        unsigned long default_timer_slack_ns;

        struct list_head        *scm_work_list;
#ifdef CONFIG_FUNCTION_GRAPH_TRACER
        /* Index of current stored address in ret_stack */
        int curr_ret_stack;
        /* Stack of return addresses for return function tracing */
        struct ftrace_ret_stack *ret_stack;
        /* time stamp for last schedule */
        unsigned long long ftrace_timestamp;
        /*
         * Number of functions that haven't been traced
         * because of depth overrun.
         */
        atomic_t trace_overrun;
        /* Pause for the tracing */
        atomic_t tracing_graph_pause;
#endif
```

/linux-3.2.72/include/linux/sched.h

Linux-3.2.72内核struct task_struct 15/15

```
#ifdef CONFIG_TRACING
        /* state flags for use by tracers */
        unsigned long trace;
        /* bitmask and counter of trace recursion */
        unsigned long trace_recursion;
#endif /* CONFIG_TRACING */
#ifdef CONFIG_CGROUP_MEM_RES_CTLR /* memcg uses this to do batch job */
        struct memcg_batch_info {
                int do_batch;   /* incremented when batch uncharge started */
                struct mem_cgroup *memcg; /* target memcg of uncharge */
                unsigned long nr_pages; /* uncharged usage */
                unsigned long memsw_nr_pages; /* uncharged mem+swap usage */
        } memcg_batch;
#endif
#ifdef CONFIG_HAVE_HW_BREAKPOINT
        atomic_t ptrace_bp_refcnt;
#endif
};
```

/linux-3.2.72/include/linux/sched.h

2.2 进程控制

2.2.1 进程控制块

2.2.2 进程图

2.2.3 进程的创建与终止

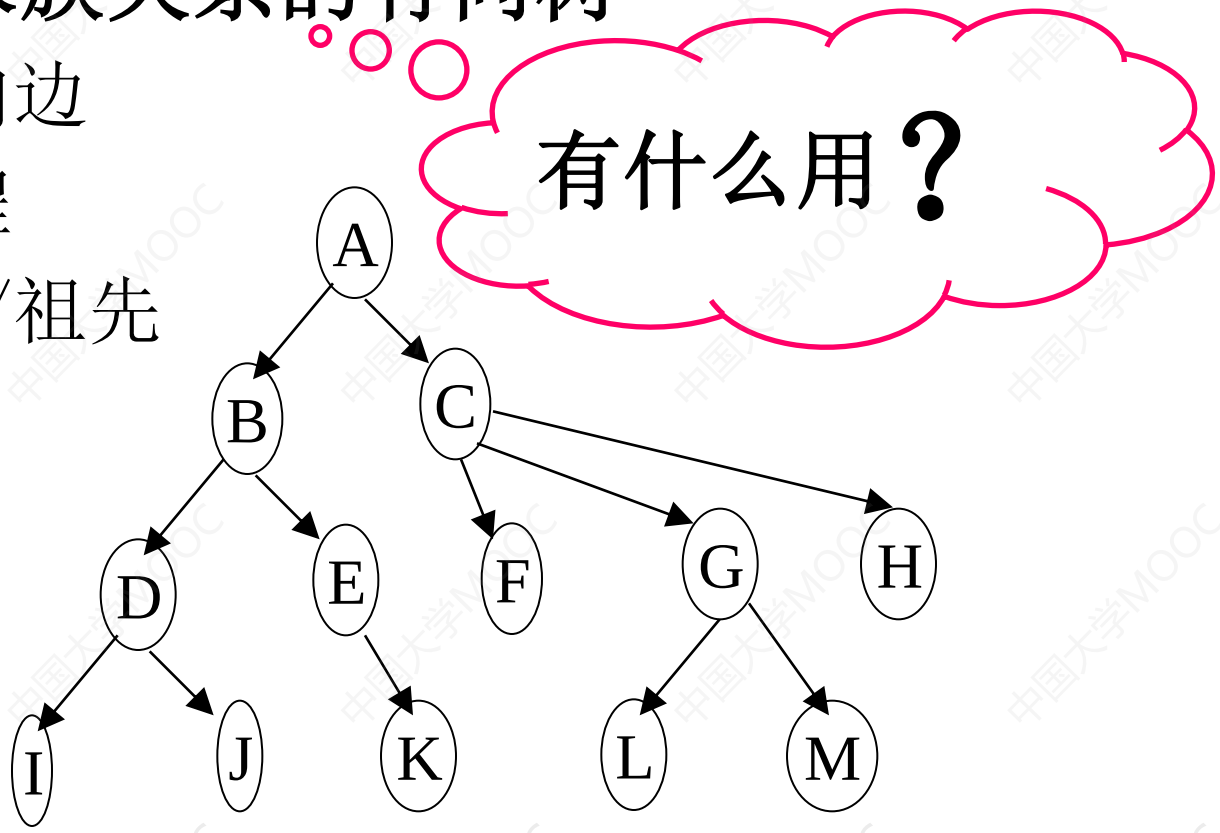
2.2.4 进程的阻塞与唤醒

2.2.5 进程的挂起与激活

2.2.6 UNIX进程描述与控制

进程图（进程树）

- 描述进程家族关系的有向树
  - 结点/有向边
  - 父/子进程
  - 祖父进程/祖先



2.2 进程控制

2.2.1 进程控制块

2.2.2 进程图

2.2.3 进程的创建与终止

2.2.4 进程的阻塞与唤醒

2.2.5 进程的挂起与激活

2.2.6 UNIX进程描述与控制

引起创建/终止进程的事件

- 用户登录
  - 分时系统中，验证为合法的终端用户登录
- 作业调度
  - 批处理系统中作业调度程序调度到某作业
- 提供服务
  - 运行中的用户程序提出某种请求
- 应用请求
  - 基于应用进程的需要由其自身创建新进程

- 正常结束
  - 批处理系统中Halt，分时系统中LogsOff
- 异常结束
  - 越界错误、保护错
  - 特权指令错
  - 非法指令错
  - 运行超时、等待超时
  - 算术运算错、I/O故障
- 外界干预
  - 操作员或操作系统干预
  - 父进程请求/终止

进程创建/终止过程

Create()原语

1、分配标识符，并申请空白进程控制块

2、为新进程的程序和数据及用户栈分配必要的内存空间

- 所需内存大小问题

3、初始化进程控制块

- 自身/父进程标识符
- 处理机状态/调度信息

4、将新进程插入到就绪进程队列

Terminate()原语

1、检索被终止进程PCB，读取进程状态

2、若其正处于执行状态，应立即中止执行并设置调度标志为真，以指示调度新进程

3、终止子孙进程

4、资源归还

5、移出被终止进程PCB，等待其它程序查询利用

2.2 进程控制

2.2.1 进程控制块

2.2.2 进程图

2.2.3 进程的创建与终止

2.2.4 进程的阻塞与唤醒

2.2.5 进程的挂起与激活

2.2.6 UNIX进程描述与控制

引起进程阻塞/唤醒的事件

- 请求系统服务
  - 但不能立即满足
- 启动某种操作
  - 且必须在该操作完成之后才能继续执行
- 新数据尚未到达
  - 相互合作进程的一方需首先获得另一进程数据才能继续
- 无新工作可做
  - 特定功能系统进程当完成任务且暂无任务

- 系统服务满足
- 操作完成
- 数据到达
- 新任务出现

进程阻塞/唤醒过程

原语配对！

Block()原语

1、先立即停止执行，把进程控制块中的现行状态由“执行”改为阻塞，并将它插入到对应的阻塞队列中

2、转调度程序进行重新调度，将处理机分配给另一就绪进程，并进行切换

？

Wakeup()原语

首先把被阻塞进程从等待该事件的阻塞进程队列中移出，将其PCB中的现行状态由阻塞改为就绪，然后再将该进程插入到就绪队列中

2.2 进程控制

2.2.1 进程控制块

2.2.2 进程图

2.2.3 进程的创建与终止

2.2.4 进程的阻塞与唤醒

2.2.5 进程的挂起与激活

2.2.6 UNIX进程描述与控制

进程挂起/激活过程

Suspend()原语

1、检查被挂进程现行状态并修改和插队

2、复制PCB到指定区域

3、若被挂进程正在执行则转向调度程序重新调度

Activate()原语

1、检查进程现行状态并修改和插队

2、若有新进程进入就绪队列且采用了抢占式调度策略，则检查和决定是否重新调度

2.2 进程控制

2.2.1 进程控制块

2.2.2 进程图

2.2.3 进程的创建与终止

2.2.4 进程的阻塞与唤醒

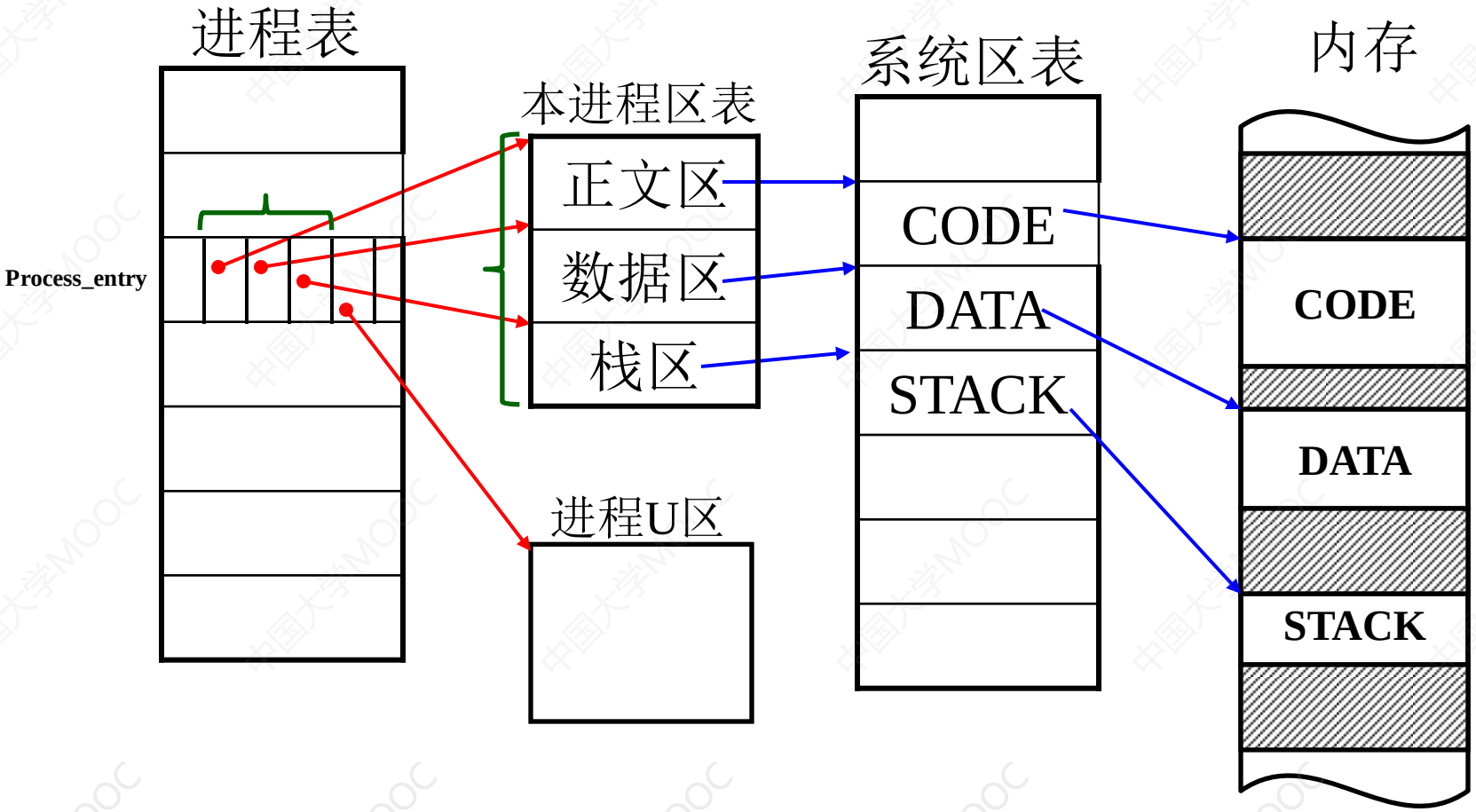
2.2.5 进程的挂起与激活

2.2.6 UNIX进程描述与控制

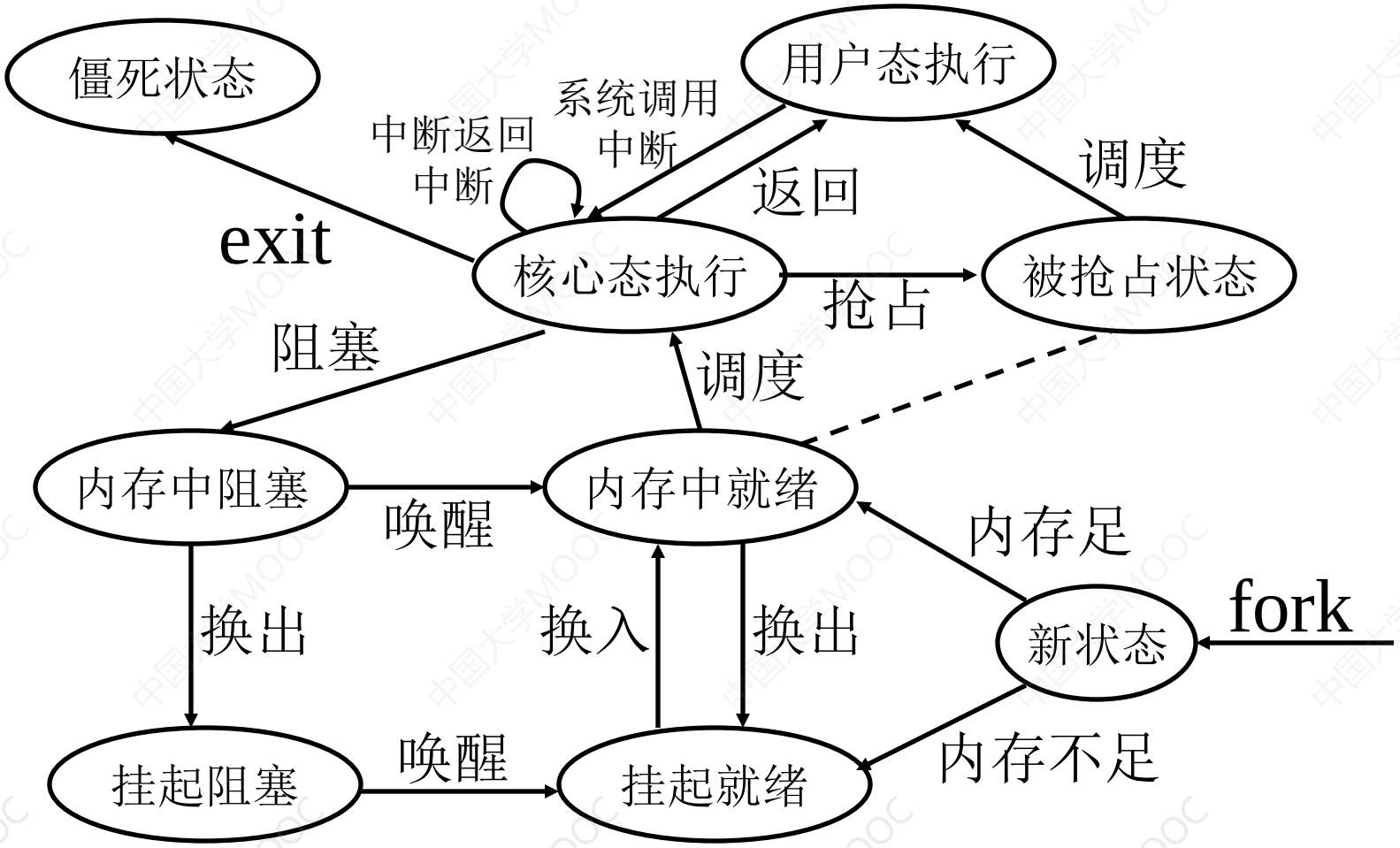
进程映像

- 进程是进程映像的执行过程，进程映像则是正在运行进程的实体
- 用户级上下文
  - 用户程序（正文区、数据区）、用户栈区、共享存储区
- 寄存器上下文
  - PC、PSW、栈指针、通用寄存器
- 系统级上下文
  - 进程表项、U区、本进程区表、系统区表项、页表
  - 核心栈、若干层寄存器上下文

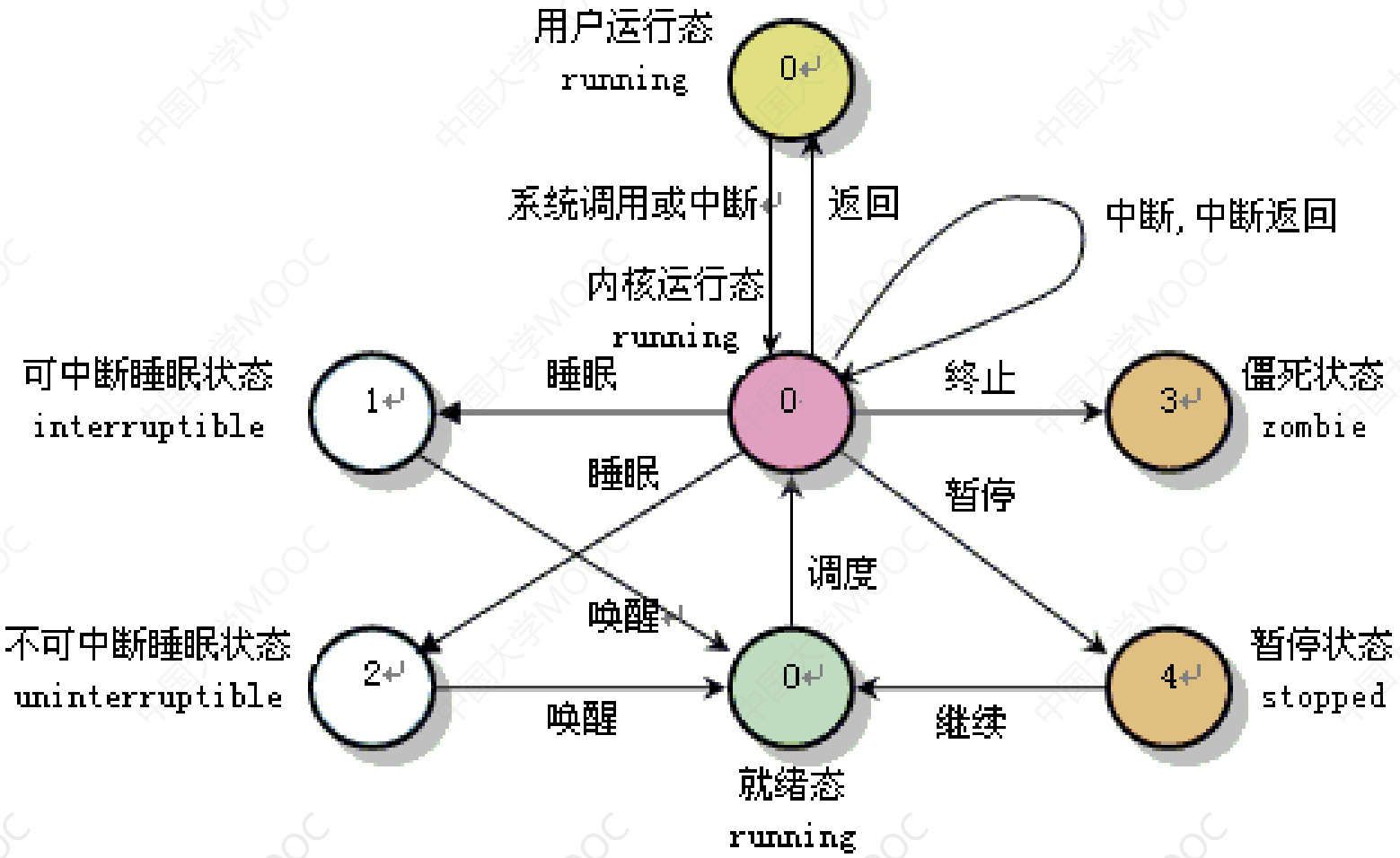
UNIX进程控制用数据结构



UNIX进程状态转换图



Linux进程状态转换图



进程控制

- fork系统调用
  - 创建新进程
  - 0号（对换）进程 => 1号（始祖）进程
- exit系统调用
  - 实现进程自我终止
- exec系统调用
  - 改变进程原有代码（更新用户级上下文）
- wait系统调用
  - 阻塞主调进程并等待子进程终止

进程调度与切换

- 引起进程调度的原因
  - 时钟中断、核心态执行返回、放弃处理机
- 调用算法
  - 动态优先数轮转调度算法
- 进程优先级分类
  - 内核优先级（可/不可中断）、用户优先级
- 优先数计算
  - 基本用户优先数、进程本次CPU使用时间
- 进程切换

2.2 进程控制

2.2.1 进程控制块

2.2.2 进程图

2.2.3 进程的创建与终止

2.2.4 进程的阻塞与唤醒

2.2.5 进程的挂起与激活

2.2.6 UNIX进程描述与控制

作业题

- **2.5** 试根据你自己的理解，采用类C语言设计和描述操作系统关于进程控制块的数据结构、组织方式及管理机制。在此基础上，给出进程的创建、终止、阻塞、唤醒、挂起与激活等函数原型及函数代码。注意，对于过于复杂的功能或你无法解决的细节可采用指定功能的函数模块如处理机调度scheduler()来替代。

第二章 进程管理

2.1 进程的基本概念

2.2 进程控制

2.3 进程同步

2.4 经典进程同步问题

2.5 管程

2.6 进程通信

2.7 线程

2.3 进程同步

2.3.1 并发进程间制约关系

2.3.2 临界资源与临界区

2.3.3 进程同步机制准则

2.3.4 信号量机制

2.3.5 信号量机制应用基础

并发进程间制约关系

- 资源共享关系—间接制约
  - 多个进程彼此无关，完全不知道或只能间接感知其它进程的存在
  - 系统须保证诸进程能互斥地访问临界资源
  - 系统资源应统一分配，而不允许用户进程直接使用
- 相互合作关系—直接制约
  - 系统应保证相互合作的诸进程在执行次序上的协调和防止与时间有关的差错

2.3 进程同步

2.3.1 并发进程间制约关系

2.3.2 临界资源与临界区

2.3.3 进程同步机制准则

2.3.4 信号量机制

2.3.5 信号量机制应用基础

临界资源

- 一段时间内只允许一个进程访问的资源
  - 许多物理设备、变量及表格
- 举例
  - 两个进程A、B共享变量**n**(初值为0)

A[循环]: 生产数据

```
if (n<MAX)
 放数据到Buf[n];
n:=n+1;
```

B[循环]: 生产数据

```
if (n<MAX)
 放数据到Buf[n];
n:=n+1;
```

  - 共享变量n不互斥访问的后果：(1) AB两个进程把数据放到同一缓冲数组元素，冲掉前一个数据；(2) n值和数据量不一致；(3) 缓冲满时强放数据

临界区

- 每个进程中访问临界资源的那段代码称为临界区
- 保证诸进程互斥地进入自己的临界区是实现它们对临界资源的互斥访问的充要条件

访问临界资源的循环进程描述

主程序描述框架

```
begin
   parbegin
      Pi;
      Pj;
   parend
end
```

进程P_i

```
begin
      repeat
         ……
         进入区
         临界区
         退出区
         ……
      until false;
end
```

2.3 进程同步

2.3.1 并发进程间制约关系

2.3.2 临界资源与临界区

2.3.3 进程同步机制准则

2.3.4 信号量机制

2.3.5 信号量机制应用基础

进程同步机制基本准则

- 空闲让进
  - 当无进程处于某临界区时，可允许一个请求进入“对等”临界区的进程立即进入自己的临界区
- 忙则等待
  - 当已有进程进入自己的临界区时，所有企图进入“对等”临界区的进程必须等待
- 有限等待
  - 对要求访问临界资源的进程，应保证该进程能在有限时间内进入自己的临界区
- 让权等待
  - 当进程不能进入自己的临界区时，应释放处理机

解决进程互斥的软硬件方法

进程互斥问题描述

- 由两个进程P_i、P_j，共享某临界资源R
- 主程序

```
begin
    parbegin
        Pi;
        Pj;
    parend
end
```

进程互斥算法1——设置访问编号

Var turn: integer :=i; [全局变量]

P_i	P_j
repeat	
……	
while turn<>i do	**while turn<>j do**
临界区	临界区
turn:=j;	**turn:=i;**
……	……
until false;	

强制交替访问临界资源，未考虑进程实际需求；有违……准则?

进程互斥算法2——设置访问标志

Var $flag_i$, $flag_j$: boolean :=false, false; [全局变量]

P_i

repeat

……

while fl

$flag_i$:=

临界区

$flag_i$:= false;

……

until false;

P_j

lo

;

$flag_j$:= false;

……

访问临界资源前检查访问标志；但可能违背……准则?

进程互斥算法3——设置欲访问标志

Var $flag_i$, $flag_j$: boolean :=false, f e; [全局变量]

P_j

repeat

……

fl

w

临

$flag_i$

……

until false;

$flag_j$:= false;

……

访问临界资源前先公告自己访问意愿再检查其它进程有无访问意愿；但可能违背……准则?

进程互斥

(Pe

……

P_j

1 两者都有意愿进去时，$flag_i$、$flag_j$均为真，但turn取值确定，肯定会有一个进程进入临界区（不会都挡在外边），所以空闲让进。

2 一个进程进入临界区，说明另一进程或者对应flag为假、或者后执行的turn赋值。这两种情况前者flag均为真且turn指向前者保持不变直到前者退出临界区，故而后者不会同时进入，即忙则等待。

unt

P_i

利用硬件方法解决进程互斥问题

- 完全利用软件方法解决诸进程互斥进入临界区的问题，有一定难度，且有很大局限性，因而现代很少采用此方法。针对这一点，现在许多计算机已提供了一些特殊的硬件指令，相关指令允许对一个字中的内容进行检测和修正或交换两个字的内容，故可用于解决临界区问题
  - Test-and-Set指令
  - Swap指令

Test-and-Set指令

```
function TS(var lock:boolean):boolean;
begin
   TS:=lock;
   lock:=true;
end
```

利用Test-and-Set指令实现互斥

Var **lock**: boolean:=false; **[全局变量]**

```
repeat
    ……
    while TS(lock)  do skip;
    临界区
    lock:= false;
    ……
until false;
```

Swap指令

```
procedure Swap(var a,b: boolean);
Var temp: boolean;
begin
   temp:=a;
   a:=b;
   b:=temp;
end
```

利用Swap指令实现互斥

Var **lock**: boolean:=false; **[全局变量]**

```
Var key: boolean;
repeat
   key:=true;
   repeat
      Swap(lock, key);
   until key=false;
   临界区
   lock:= false;
until false;
```

硬件方法解决进程同步的局限性

- 利用硬件指令能有效地实现进程互斥，但它却不能满足“让权等待”的准则，造成了处理机时间的浪费，而且也很难将它用于解决较复杂的进程同步问题
- 信号量机制是一种卓有成效的进程同步工具，其已被广泛应用于单处理机和多处理机系统，以及计算机网络中

2.3 进程同步

2.3.1 并发进程间制约关系

2.3.2 临界资源与临界区

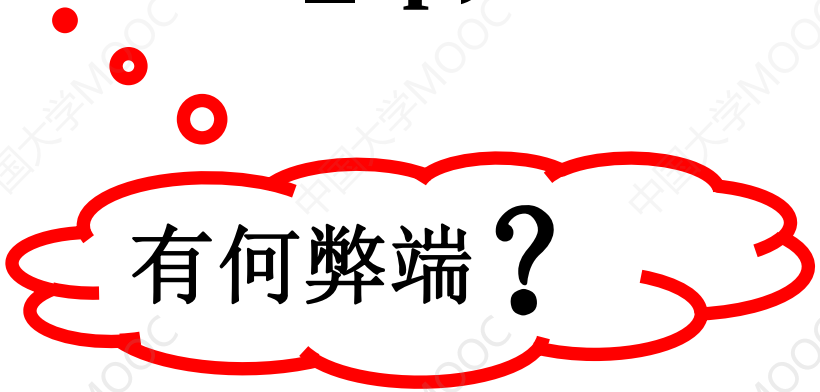
2.3.3 进程同步机制准则

2.3.4 信号量机制

2.3.5 信号量机制应用基础

整型信号量

- 整型信号量s除初始化外，仅能被两个标准的原子操作wait(s)和signal(s)亦即P/V操作来访问。
- wait(s): **while s≤0 do no_op;**
  s:=s-1;
- signal(s): **s:=s+1;**



记录型信号量机制

——信号量类型声明

```
type semphore=record
   value: integer;
   L: list of process;
end
```

记录型信号量机制

——wait(s)操作描述

```
procedure wait(s)
Var s: semphore;
begin
   s.value:=s.value - 1;
   if s.value<0 then block(s.L);
end
```

记录型信号量机制

—signal(s)操作描述

```
procedure signal(s)
Var s: semphore;
begin
   s.value:=s.value + 1;
   if s.value≤0 then wakeup(s.L);
end
```

AND型信号量集机制的引入

- 对于多个进程要共享两个以上的资源的情况，上述机制则可能导致发生死锁
- 例两个进程A、B要求同时访问共享数据C、D

process A:

```
wait(Dmutex);
wait(Cmutex);
……
```

process B:

```
wait(Cmutex);
wait(Dmutex);
……
```

- 对策：

“若干个临界资源”的分配采取原子操作方式

Swait(s_1, s_2, … , s_n)操作

```
procedure Swait(s1, s2,
Var s1, s2, … , sn : sem
begin
  if si.value ≥1 and …
  then for i:=1 to n do
            si.value:=si.va
  else blockProcessAr
end
```

```
Var zSbG: boolean:=false;
for i:=1 to n do
{ if (si.value<1)
  { zSbG:=true; break; }
}
if (zSbG)
{ ResetPC();
  block(si.L); }
else
{ for i:=1 to n do
     si.value :=si.value －1;  }
```

Ssignal(s_1, s_2, … , s_n)操作

```
procedure Ssignal(s1, s2, … , sn)
Var s1, s2, … , sn : semphore;
begin
   for i:=1 to n do
      si.value:=si.value + 1;
      wakeupAllProcesses(si);
   endfor;
end
```

一般信号量集机制的引入

- 记录型信号量集机制中，wait(s)和signal(s)操作仅能对信号量施以增1和减1的操作，即每次只能获得或释放一个单位的临界资源。当一次需d个某类临界资源时，便需要进行d次wait(s)操作，这显然是低效的。
- 此外，在有些情况下，要求当资源数量低于某一下限值时，便不予分配。故在每次分配之前，都必须测试该资源的数量是否不小于测试值t。
- 基于以上两点可以对AND信号量机制进行扩充，形成一般化的“信号量集”机制。

Swait(s_1, t_1, d_1, … , s_n , t_n, d_n)操作

```
procedure Swait(s1, t1
Var s1, s2, … , sn : sem
      t1, t2, … , tn, d1, d
begin
  if s1.value ≥ t1 and … a
  then for i:=1 to n do
            si.value:=si.v
  else blockProcessA
end
```

```
Var zSbG: boolean:=false;
for i:=1 to n do
{ if (si.value<ti)
  { zSbG:=true; break; }
}
if (zSbG)
{ ResetPC();
  block(si.L); }
else
{ for i:=1 to n do
    si.value :=si.value − di; }
```

Ssignal(s_1, d_1,… ,s_n, d_n)操作

```
procedure Ssignal(s1, d1, … , sn , dn)
Var s1, s2, … , sn : semphore;
      d1, d2, … , dn : integer;
begin
   for i:=1 to n do
      si.value:=si.value + di;
      wakeupAllProcesses(si);
   endfor;
end
```

一般信号量集的几种特殊情况

- Swait(s, d, d)
  - 信号量集中只有一个信号量，但它允许每次申请d个资源；当现有资源少于d个时，便不予分配
- Swait(s, 1, 1)
  - 此时的信号量集已退化为一般的记录型信号量
- Swait(s, 1, 0)
  - 一种特殊且很有用的信号量，相当于可控开关
  - 当s≥1时，允许多个进程进入某特定区；当s变为0后，将阻止任何进程进入该特定区

2.3 进程同步

2.3.1 并发进程间制约关系

2.3.2 临界资源与临界区

2.3.3 进程同步机制准则

2.3.4 信号量机制

2.3.5 信号量机制应用基础

利用信号量实现互斥——主程序

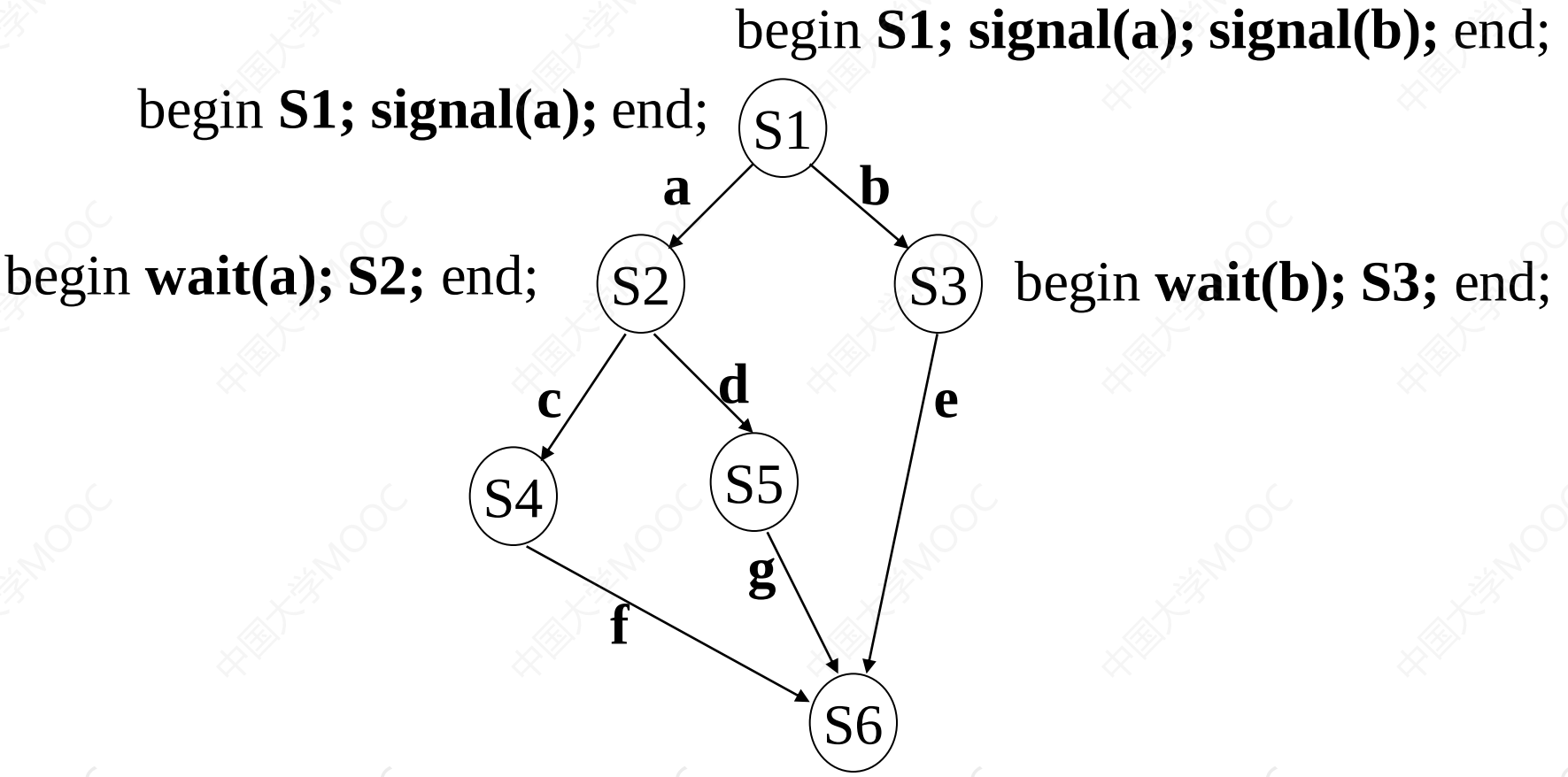
```
Var mutex: semphore:=1;
begin
   parbegin
      process1;
      process2;
   parend
end
```

利用信号量实现互斥——子程序

```
process1:
begin
   repeat
      ……
      wait(mutex);
      临界区
      signal(mutex);
      ……
   until false;
end
```

```
process2:
begin
   repeat
      ……
      wait(mutex);
      临界区
      signal(mutex);
      ……
   until false;
end
```

信号量要描述的前趋关系示例



利用信号量来描述前趋关系

```
Var a,b,c,d,e,f,g: semphore:=0,0,0,0,0,0,0;
begin
   parbegin
      begin S1; signal(a); signal(b); end;
      begin wait(a); S2; signal(c); signal(d); end;
      begin wait(b); S3; signal(e); end;
      begin wait(c); S4; signal(f);  end;
      begin wait(d); S5; signal(g); end;
      begin wait(e); wait(f); wait(g); S6; end;
   parend
end
```

2.3 进程同步

2.3.1 并发进程间制约关系

2.3.2 临界资源与临界区

2.3.3 进程同步机制准则

2.3.4 信号量机制

2.3.5 信号量机制应用基础

作业题

- **2.6** 什么是临界资源和临界区？试举例说明。并谈谈你对进程同步机制准则的理解。
- **2.7** 试阐述你对整型信号量机制与记录型信号量机制的完整理解以及AND型信号量机制与一般信号量集机制的基本思想。

实验课题

实验课题5

同步机制及应用编程实现与比较

第二章 进程管理

2.1 进程的基本概念

2.2 进程控制

2.3 进程同步

2.4 经典进程同步问题

2.5 管程

2.6 进程通信

2.7 线程

2.4 经典进程同步问题

2.4.1 生产者—消费者问题

2.4.2 哲学家进餐问题

2.4.3 读者—写者问题

生产者—消费者问题背景

- 生产者—消费者问题是相互合作进程关系的一种抽象
  - 输入时的输入进程与计算进程
  - 输出时的计算进程与输出进程
- 生产者—消费者问题具有很大的代表性和实用价值
  - 计算机系统⇔IPO系统

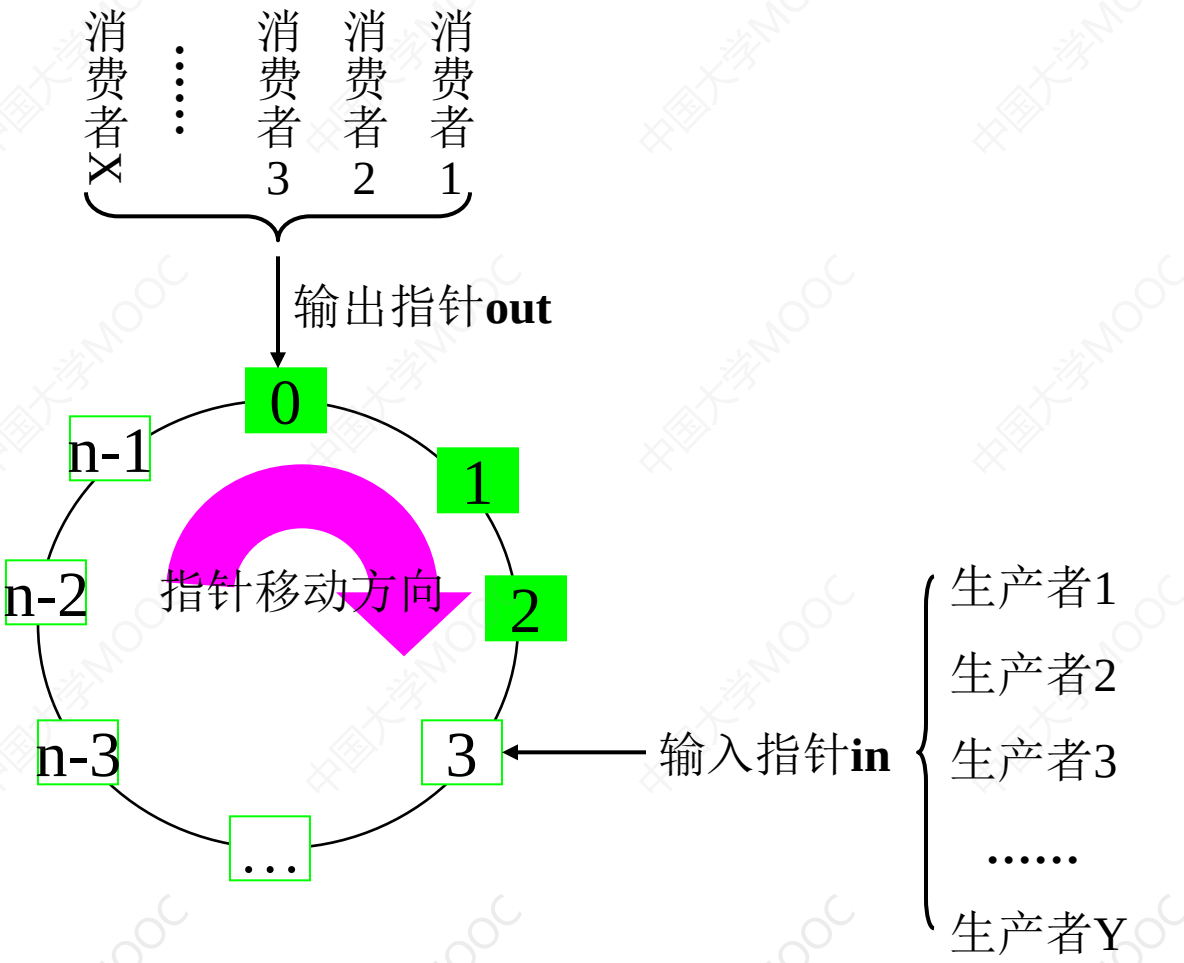
生产者—消费者问题描述

- 一群生产者进程在生产数据，并将此数据提供给一群消费者进程去消费处理
- 为使二者可以并发执行，在它们之间设置了一个具有n个缓冲区的循环缓冲，生产者进程可以将它所生产的数据放入一个缓冲区中，消费者进程可以从一个缓冲区中取得一个数据消费
- 异步运行方式及**彼此必须保持同步**

生产者－消费者问题剖析

- 空缓冲区与满缓冲区
  - 空缓冲区是指未投放数据或虽曾投放数据但对应数据已被取走的缓冲区
  - 满缓冲区则指已投放数据且对应数据尚未被取走的缓冲区
- 进程同步
  - 当生产者进程要把所生产的数据送入循环缓冲时，首先应检查是否有空缓冲区存在，若有，则可向对应空缓冲区中投放数据，同时通知消费者进程；否则只有等待。
  - 当消费者进程要从循环缓冲中提取数据时，首先应检查是否有满缓冲区存在，若有，则从对应满缓冲区中提取数据，并通知生产者进程，否则只有等待。
- 进程互斥
  - 缓冲区及其“指针”是临界资源：多个生产者/消费者进程

生产者—消费者问题临界资源剖析



生产者—消费者程序变量设计

- 循环缓冲表示机制
  - 一维数组buffer: array [0..n-1] of item;
- 输入指针in
  - 指示下一个可以投放数据的缓冲区
  - 初始值为0；变化方式：in := (in+1) mod n
- 输出指针out
  - 指示下一个可以获取数据的缓冲区
  - 初始值为0；变化方式：out:=(out+1) mod n
- nextp/nextc暂存每次要生产或消费的数据

生产者—消费者程序信号量设计

- 循环缓冲（缓冲区及其指针）的互斥使用
  - 互斥信号量mutex，初始值为1
- 资源信号量
  - empty 表示循环缓冲中的空缓冲区的数量，其初始值为n
  - full 表示循环缓冲中的满缓冲区的数量，其初始值为0

生产者－消费者主程序设计

```
Var buffer: array [0..n-1] of item;
      in, out: integer := 0, 0;
      mutex, empty, full : semphore := 1, n, 0 ;
begin
   parbegin
      producer1; …; produceri; …; producerY;
      consumer1; …; consumerj; …; consumerX;
   parend
end
```

生产者子程序设计

```
producer_i:
Var nextp: item;
  begin
    repeat
      produce an item in nextp;
      wait(empty);
      wait(mutex);
      buffer[in] := nextp;   in := (in+1) mod n;
      signal(mutex);
      signal(full);
    until false;
  end
```

消费者子程序设计

```
consumer_j:
Var nextc: item;
  begin
    repeat
      wait(full);
      wait(mutex);
      nextc:=buffer[out];   out := (out+1) mod n;
      signal(mutex);
      signal(empty);
      consume the item in nextc;
    until false;
  end
```

生产者-消费者问题解决方案

```
producer_i:
Var nextp: item;
  begin
    repeat
      produce an item in nextp;
      wait(empty);
      wait(mutex);
      buffer[in] := nextp;
      in := (in+1) mod n;
      signal(mutex);
      signal(full);
    until false;
  end
```

```
consumer_j:
Var nextc: item;
  begin
    repeat
      wait(full);
      wait(mutex);
      nextc:=buffer[out];
      out := (out+1) mod n;
      signal(mutex);
      signal(empty);
      consume the item in nextc;
    until false;
  end
```

同步问题程序设计要领

- 每个并发子程序关于互斥信号量的wait与signal操作必须在同一子程序中成对出现
- 关于资源信号量的wait与signal操作同样需成对出现，但可以分别处于不同的并发子程序中
- 每个并发子程序中的多个wait操作的顺序不能颠倒，即资源信号量wait操作执行在前而互斥信号量wait操作执行在后，否则可能引起死锁
- 每个并发子程序中的多个signal 操作的执行顺序无关紧要
- 非临界资源访问操作无需放到临界区中

基于AND信号量的生产/消费者子程序设计

```
producer_i:
  begin
    repeat
      produce an item in nextp;
      Swait(empty, mutex);
      buffer[in] := nextp;
      in := (in+1) mod n;
      Ssignal(mutex, full);
    until false;
  end
```

```
consumer_j:
  begin
    repeat
      Swait(full, mutex);
      nextc:=buffer[out];
      out := (out+1) mod n;
      Ssignal(mutex, empty);
      consume the item in nextc;
    until false;
  end
```

生产者－消费者主程序设计ZGS版

实现生产者进程之间关于缓冲区的互斥访问

实现消费者进程之间关于缓冲区的互斥访问

```
Var b: f
    in, out: g
    mutexP, mutexC, empty, full : semphore := 1, 1, n, 0 ;
begin
  parbegin
    producer1; …; produceri; …; producerY;
    consumer1; …; consumerj; …; consumerX;
  parend
end
```

生产者子程序设计ZGS版

```
producer_i:
Var nextp: item;
  begin
    repeat
      produce an item in nextp;
      wait(empty);
      wait(mutexP);
      buffer[in] := nextp;   in := (in+1) mod n;
      signal(mutexP);
      signal(full);
    until false;
  end
```

消费者子程序设计ZGS版

```
consumer_j:
Var nextc: item;
  begin
    repeat
      wait(full);
      wait(mutexC);
      nextc:=buffer[out];   out := (out+1) mod n;
      signal(mutexC);
      signal(empty);
      consume the item in nextc;
    until false;
  end
```

2.4 经典进程同步问题

2.4.1 生产者—消费者问题

2.4.2 哲学家进餐问题

2.4.3 读者—写者问题

哲学家进餐问题描述

- 哲学家进餐问题是典型的同步问题
  - 五个哲学家共用一张圆桌，分别坐在环桌均匀摆放的五张椅子上，并全部奉行交替地进行思考和进餐的生活方式
  - 圆桌上放有五支筷子，均匀排放在哲学家之间的位置上
  - 哲学家饥饿时便试图去取用圆桌上最靠近他左右两端的两支筷子，且只有在同时拿到两支筷子时方可进餐，进餐完毕则把筷子放回原处，并继续进行思考

哲学家进餐问题剖析(待续)

- 筷子是临界资源
  - 信号量数组chopstick[0..4]，初始值均为1
- 第i个哲学家活动

wait(chopstick[i]);

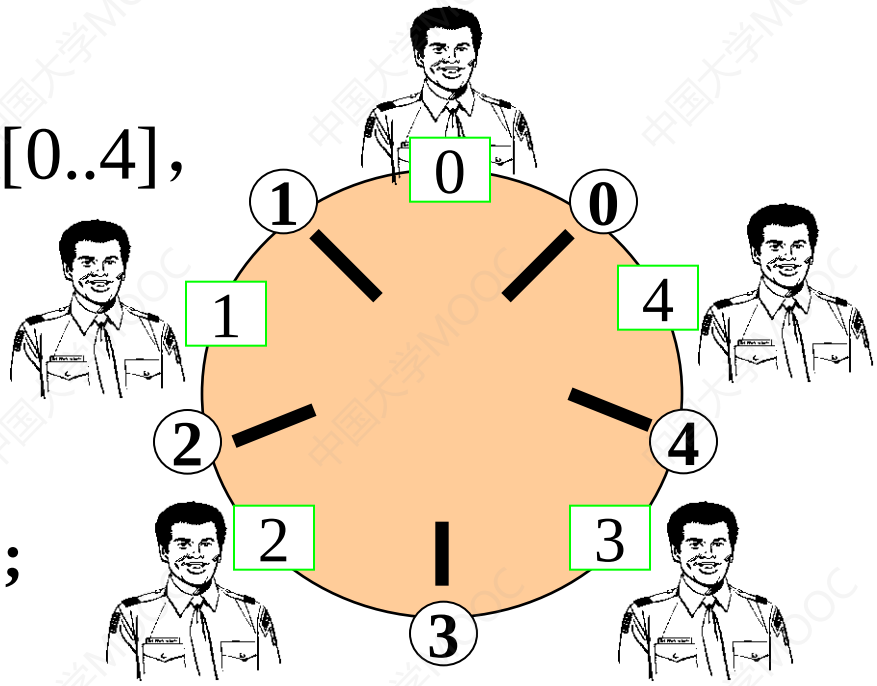
wait(chopstick[(i+1)mod 5]);

Eat;

signal(chopstick[i]);

signal(chopstick[(i+1)mod 5]);

Think;



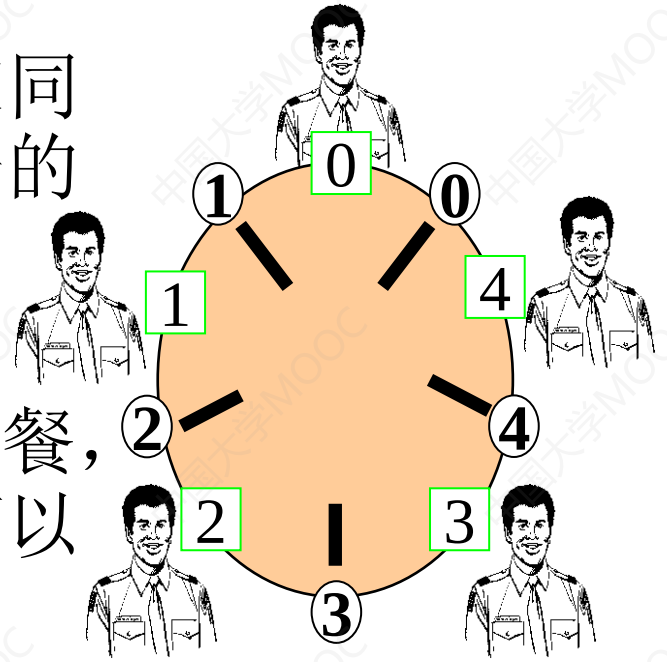
哲学家进餐问题剖析(续)

- 上述解决方案在五个哲学家同时饥饿且各自拿起左边筷子的情况下会引起死锁
- 避免死锁的三种方法

① 至多允许四个哲学家同时进餐，以保证至少有一个哲学家可以同时拿到两支筷子而进餐

② **仅当哲学家左右两支筷子均可使用时，才允许他拿筷进餐**

③ 奇数号哲学家先拿左筷后拿右筷；而偶数号哲学家则相反



哲学家进餐主程序设计

Var chopstick: array[0..4] of semphore:=(1,1,1,1,1);
begin
  parbegin
    philosophy$_0$;
    … ; philosophy$_i$; …
    philosophy$_4$;
  parend
end

基于AND信号量机制的哲学家进餐子程序设计

```
philosophy_i :
  begin
    repeat
      Think;
      Swait(chopstick[(i+1)mod 5], chopstick[i]);
      Eat;
      Ssignal(chopstick[(i+1)mod 5], chopstick[i]);
    until false;
  end
```

2.4 经典进程同步问题

2.4.1 生产者—消费者问题

2.4.2 哲学家进餐问题

2.4.3 读者—写者问题

读者—写者问题描述

- 读者—写者问题是指保证任何写者进程必须与其它进程互斥地访问共享数据对象（数据文件或记录）的同步问题。
  - 存在多个进程共享一个数据对象
  - 只要求读的进程称为读者进程
  - 拥有写或修改要求的进程称为写者进程
  - 允许多个读者进程同时执行读操作
  - 任何写者进程的执行具有排它性
- 读者—写者问题常用于测试新同步原语

读者—写者程序信号量及变量设计

- 写者进程与其它进程的互斥执行
  - 写互斥信号量wmutex，初始值为1
- 读者进程之间的并发执行
  - 读者进程计数变量readercount，表示正在执行的读者进程数量，其初始值为0
- 读者进程计数变量的互斥访问
  - readercount对于多个读者进程而言是临界资源，应为之设置读互斥信号量rmutex，其初始值为1

读者－写者主程序设计

Var readercount: integer := 0;

rmutex, wmutex : semphore := 1, 1;

begin

parbegin

$reader_1$; … ; $reader_i$; … ; $reader_m$;

$writer_1$; … ; $writer_j$; … ; $writer_n$;

parend

end

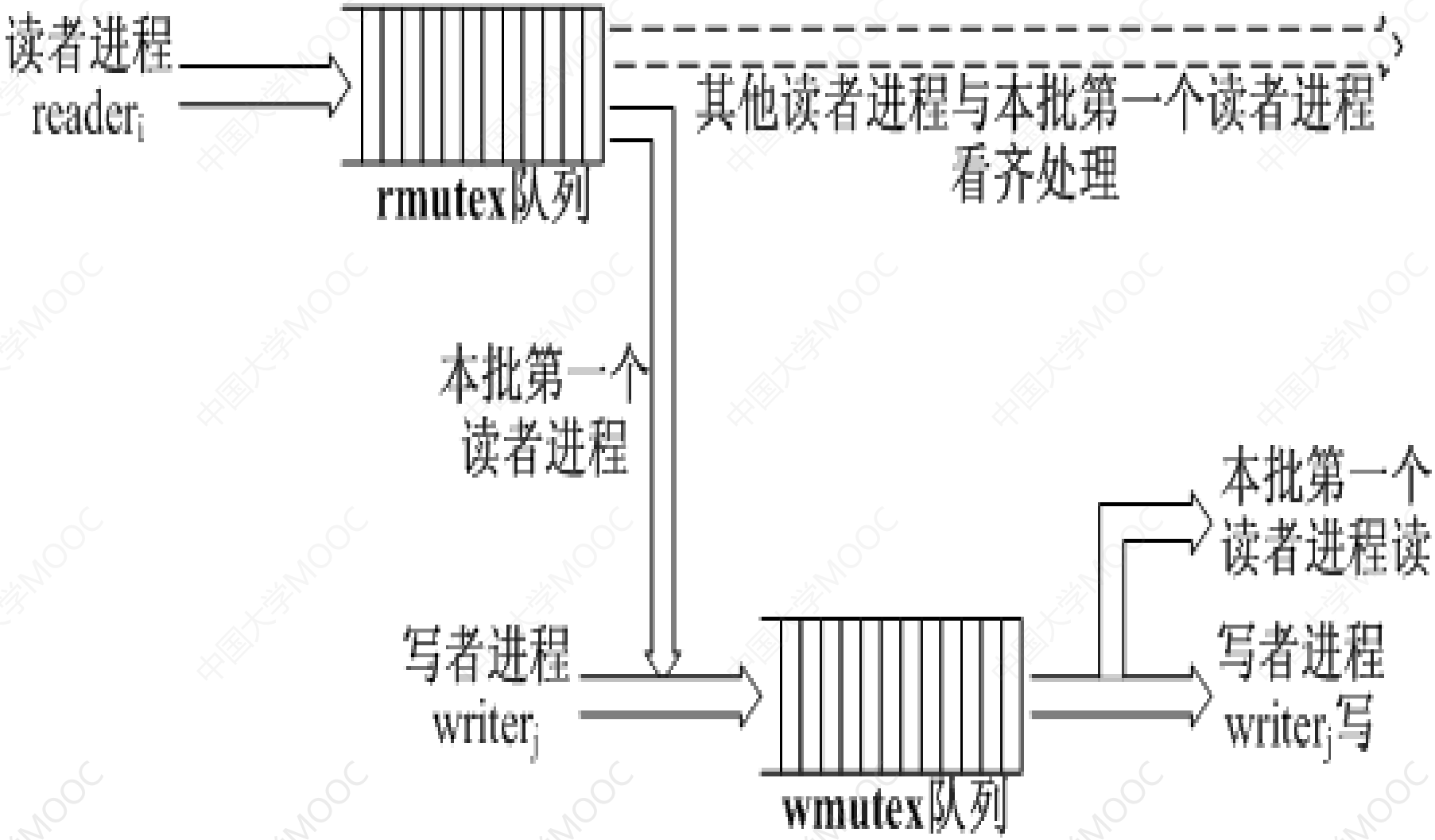
写者子程序设计

```
writer_j:
  begin
    repeat
      wait(wmutex);
      Perform write operation;
      signal(wmutex);
    until false;
  end
```

读者子程序设计

```
reader_i:
  begin
    repeat
        wait(rmutex);
        if readercount=0 then wait(wmutex);
        readercount := readercount +1;
        signal(rmutex);
        Perform read operation;
        wait(rmutex);
        readercount := readercount -1;
        if readercount=0 then signal(wmutex);
        signal(rmutex);
    until false;
  end
```

读者—写者问题解决方案反思



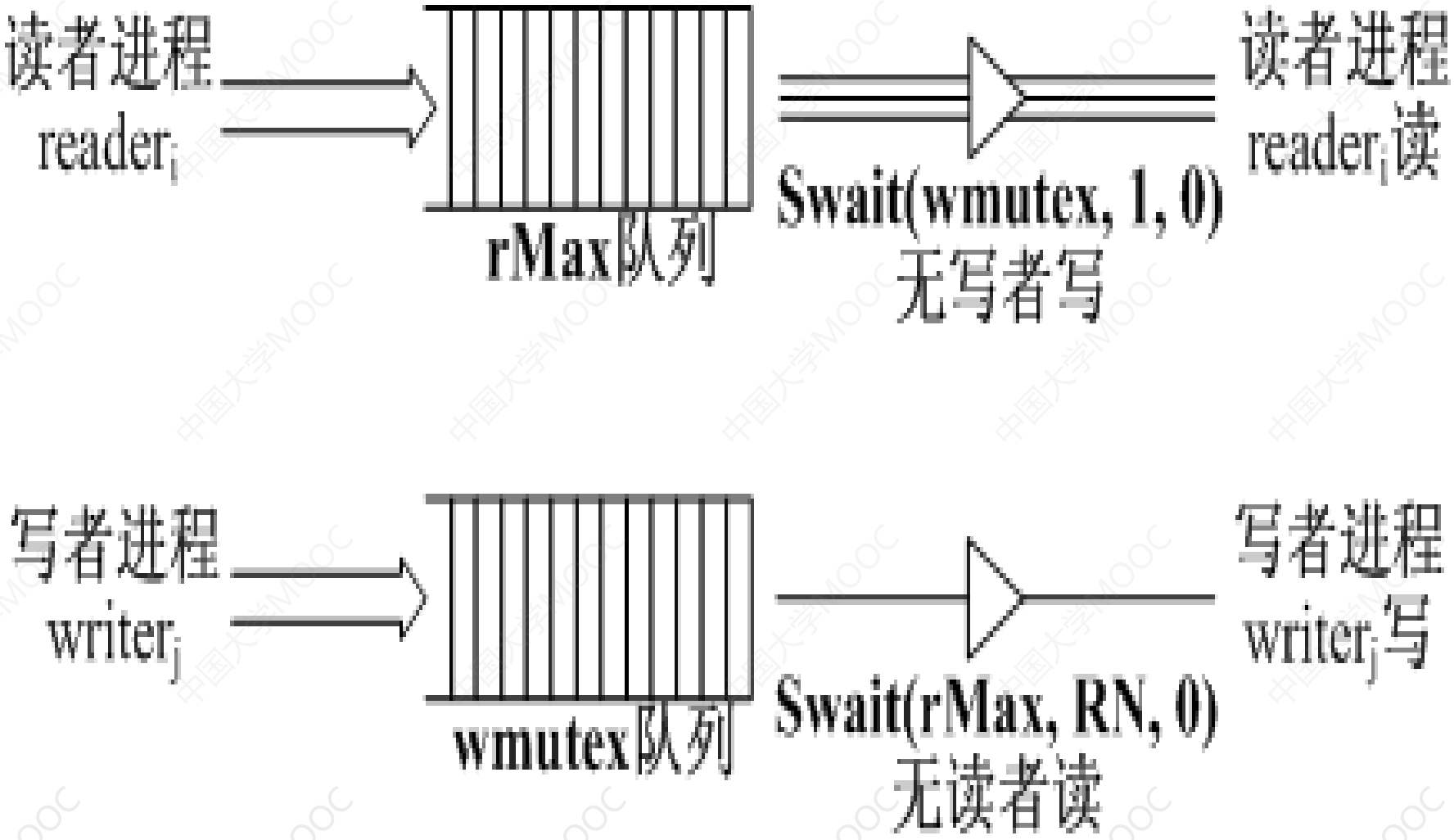
写者优先的读者—写者问题

- 一旦有写者到达，则后续的读者必须等待（无论当时是否有读者在读）

课后作业

- 如果读者到来
  - 有写者写或有写者等，则新读者等待
  - 无写者写且无写者等，则新读者可读
- 如果写者到来
  - 无读者读且无写者写，则新写者可写
  - 有读者读或有写者写，则新写者等待

读者数限定的读者—写者问题



读者数限定的读者－写者主程序设计

```
Var RN: integer := 10;
      rMax, wmutex : semphore := RN, 1;
begin
   parbegin
      reader1; … ; readeri ; … ; readerm;
      writer1; … ; writerj ; … ; writern;
   parend
end
```

读者数限定的读者子程序设计

```
readeri:
   begin
      repeat
         Swait(rMax, 1, 1);
         Swait(wmutex, 1, 0);
         ……
         Perform read operation;
         ……
         Ssignal(rMax, 1);
      until false;
   end
```

读者数限定的写者子程序设计

```
writer_j:
  begin
    repeat
      Swait(wmutex, 1, 1, rMax, RN, 0);
      Perform write operation;
      Ssignal(wmutex, 1);
    until false;
  end
```

2.4 经典进程同步问题

2.4.1 生产者—消费者问题

2.4.2 哲学家进餐问题

2.4.3 读者—写者问题

作业题

- **2.8-2.9、2.10[必做]、2.11、2.12[必做]、2.13[必做]**

  课本作业23、24、25、26、27、28

- **2.14[必做]** 给出基于记录型信号量机制的写者优先的读者-写者问题的同步解决方案。

实验课题

实验课题6

典型同步问题模拟处理编程设计与实现

第二章 进程管理

2.1 进程的基本概念

2.2 进程控制

2.3 进程同步

2.4 经典进程同步问题

2.5 管程

2.6 进程通信

2.7 线程

2.5 管程

2.5.1 管程的引入及定义

2.5.2 管程内在机制实现要领

2.5.2 利用管程解决生产者—消费者问题

管程的引入

- 基于信号量的进程同步机制的弊端
  - 访问临界资源的各进程均须自备同步操作
  - 大量同步操作分散不利于系统管理
  - 同步操作使用不当可能导致系统死锁
- 对策
  - 软硬件资源及操作抽象描述为管程
  - 并发进程间的同步操作，分别集中于相应的管程中，由管程专职负责同步方案

像什么?

管程的语法

TYPE monitor_name = MONITOR;

<管程变量说明>

PROCEDURE entry P_1（形参表）;

<过程局部变量说明>

BEGIN

<语句序列>

END;

……　　//其它入口过程定义

BEGIN

<初始化语句序列>

END

管程举例说明-资源请求与释放

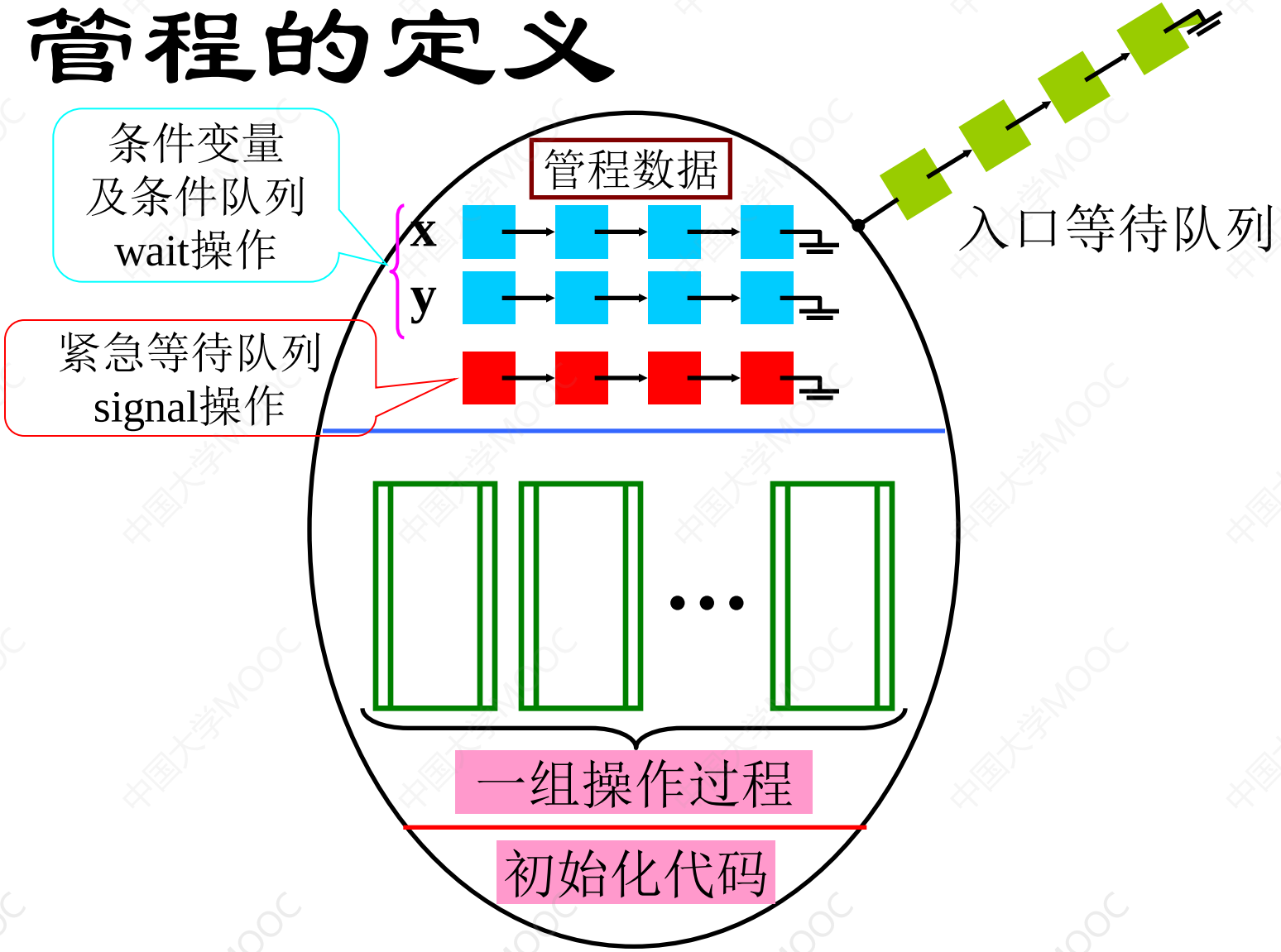
```
TYPE SSU = MONITOR
VAR busy: boolean;
        nobusy: condition;
define Qingqiu(), Shifang();
use wait(), signal();
PROCEDURE entry Qingqiu();
BEGIN
    if busy then nobusy.wait();
    busy := true;
END;
```

```
PROCEDURE entry Shifang();
BEGIN
    busy := false;
    nobusy.signal();
END;

BEGIN
    busy := false;
END
```

```
Process
BEGIN
  ……
  SSU.Qingqiu();
  …//访问资源
  SSU.Shifang();
  ……
END;
```

管程的定义


条件变量
及条件队列
wait操作
紧急等待队列
signal操作
管程数据
x
y
入口等待队列
一组操作过程
初始化代码

2.5 管程

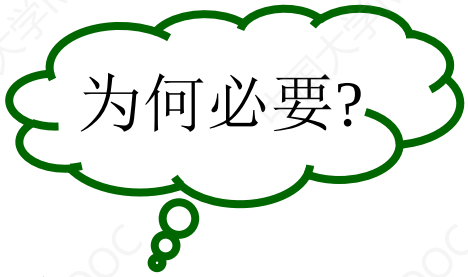
2.5.1 管程的引入及定义

2.5.2 管程内在机制实现要领

2.5.2 利用管程解决生产者—消费者问题

管程关于进程同步互斥的保证

- 管程特性
  - 数据结构与操作代码的封装性
  - 管程每次只准许一个进程进入以保证互斥
- 条件变量
  - 进程同步操作必需设置原语wait与signal
  - 为区别等待原因而引入**条件变量**
  - 不同条件变量设有不同进程等待队列
  - **signal操作处理方式**

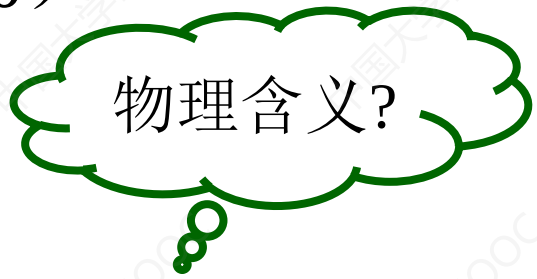


signal操作后管程互斥的保证

- 假定进程P执行x.signal()操作，而条件变量x队列中存在阻塞进程Q等，则当P执行x.signal()操作后，进程Q被释放。为避免管程中同时有两个进程在执行，可采取如下处理策略之一：
  - P等待直到Q退出管程或者Q等待另一条件
  - Q等待直到P退出管程或者P等待另一条件
  - Hoare选用策略一，而Hanson则规定管程中的过程所执行的signal操作为最后一个操作

Hoare管程实现方案（4-1）

- TYPE **interface** = RECORD包括以下分量
  - 控制管程互斥进入的信号量mutex（初值为1）
  - 紧急队列信号量semCQ（初值为0）
  - 紧急队列长度lenCQ（初值为0）
- 管程定义及成员组成
  - **interface**类型成员数据变量IM
  - 所谓的**wait/signal条件变量操作函数**，均设计为基于信号量的可被中断的过程（由于为成员函数/过程，所以可直接访问IM）



Hoare管程实现方案（4-2）

- 并发进程关于管程入口过程的调用框架，也即**管程入口过程**调用语句将转换为：

```
PROCEDURE entry RuKouGuoCheng();
BEGIN
wait(IM.mutex) ⇔WaitMutex();
<过程体>                    ⇔SignalCQ();
if IM.lenCQ >0 then signal(IM.semCQ);
⇔CriticalQueue()        else signal(IM.mutex);
                                ⇔SignalMutex();
END
```

Hoare管程实现方案（4-3）

- 条件变量的wait操作函数

```
procedure wait(var xSem: semaphore, xCount: integer);
begin
   xCount := xCount + 1;
   if IM.lenCQ > 0 then signal(IM.semCQ);
                   else signal(IM.mutex);
   wait(xSem);
   xCount := xCount - 1;
end;
```

Hoare管程实现方案（4-4）

- 条件变量的signal操作函数

```
procedure signal(var xSem: semaphore, xCount: integer);
begin
  if xCount > 0 then
  begin
     IM.lenCQ := IM.lenCQ + 1;
     signal(xSem);
     wait(IM.semCQ);
     IM.lenCQ := IM.lenCQ - 1;
  end;
end;
```

基于Hoare管程的哲学家就餐6-1

```
cobegin
    process philosopher_i:
        begin
            WaitMutex();
            dining_philosophers.PickUp(i);
            if CriticalQueue()>0 then SignalCQ();
                                 else SignalMutex();
            吃饭;
            WaitMutex();
            dining_philosophers.PutDown(i);
            if CriticalQueue()>0 then SignalCQ();
                                 else SignalMutex();
        end;
coend.
```

基于Hoare管程的哲学家就餐6-2

- 哲学家状态细化
  - 思考、饥饿、吃饭
  - var state: array [0..4] of (thinking, hungry, eating);
- i#哲学家进入吃饭状态的前提条件
  - 其相邻两座哲学家不处于吃饭状态
  - state[(i-1)mod5]≠eating AND state[(i+1)mod5]≠eating
- 引入关于哲学家等待的条件变量（对于Hoare管程实现方案，其实就是信号量）及其队列长度
  - var self: array [0..4] of semaphore;　//i#等吃条件变量
  - selfCount: array[0..4] of integer;

基于Hoare管程的哲学家就餐6-3

```
TYPE dining_philosophers = MONITOR
VAR state: array [0..4] of (thinking, hungry, eating);
        self: array [0..4] of semaphore := 0, 0, 0, 0, 0;
        selfCount: array[0..4] of integer:= 0, 0, 0, 0, 0;
procedure entry Test(k: 0..4);
procedure entry PickUp(k: 0..4);
procedure entry PutDown(k: 0..4);
use wait(semaphore,integer),  signal(semaphore,integer);
BEGIN
    for i:=0 to 4 do
       state[i] := thinking;
END;
```

基于Hoare管程的哲学家就餐6-4

```
procedure PickUp(k: 0..4);
BEGIN
   state[k] := hungry;
   Test(k);
   if state[k]≠eating
   then wait(self[k], selfCount[k]);
END;
```

基于Hoare管程的哲学家就餐6-5

```
procedure PutDown(k: 0..4);

BEGIN

  state[k] := thinking;

  Test((k-1) mod 5);

  Test((k+1) mod 5);

END
```

基于Hoare管程的哲学家就餐6-6

```
procedure Test(k: 0..4);
BEGIN
  if state[(k-1)mod5]≠eating
    AND state[k]=hungry
    AND state[(k+1)mod5]≠eating
  then
  begin
     state[k] := eating;
     signal(self[k], selfCount[k]);
  end;
END;
```

Hanson管程实现方案(5-1)

条件变量物理含义及初值？

- 引入用于管程控制的四条原语
  - 实现管程互斥访问原语 check(); / release();
  - 条件变量等待原语 wait(condition);
  - 条件变量释放原语 signal(condition);
  - ✓ 条件变量核心原语 W(condition);/R(condition);
- 引入用于开放和关闭管程的条件变量lock
- 引入关于等待进入管程的进程计数变量lenEntryQ
- 引入关于已进入管程且不处于等待状态的进程计数变量countActive

Hanson管程实现方案(5-2)

- 管程检查进入原语

```
procedure check();
begin
    if countActive = 0
    then countActive := countActive + 1;
    else
    begin
        lenEntryQ := lenEntryQ +1;
        W(lock);
    end;
end;
```

Hanson管程实现方案(5-3)

- 管程退出开放原语

```
procedure release();
begin
    countActive := countActive - 1;
    if countActive = 0 AND lenEntryQ >0  then
    begin
        lenEntryQ := lenEntryQ -1;
        countActive := countActive + 1;
        R(lock);
    end;
end;
```

Hanson管程实现方案(5-4)

- 条件变量等待原语

```
procedure wait(var zC: condtion);
begin
    zC := zC + 1;
    countActive := countActive - 1;
    if lenEntryQ >0 then
    begin
        lenEntryQ := lenEntryQ -1;
        countActive := countActive + 1;
        R(lock);
    end;
    W(zC);
end;
```

Hanson管程实现方案(5-5)

- 条件变量释放原语

```
procedure signal(var zC: condtion);
begin
    if zC > 0 then
    begin
        zC := zC -1;
        countActive := countActive + 1;
        R(zC);
    end;
end;
```

Hanson管程应用

- 基于Hanson管程的进程同步问题解决方案
  - 入口（entry）类型过程隐含以check()开始，以release()结束
- 基于Hanson管程的进程同步问题解决示例
  - 生产者-消费者问题

2.5 管程

2.5.1 管程的引入及定义

2.5.2 管程内在机制实现要领

2.5.2 利用管程解决生产者—消费者问题

基于管程PC的生产者子程序设计

```
producer:
Var nextp: item;
  begin
    repeat
      produce an item in nextp;
      wait(empty);   wait(mutex);
      buffer[in] := nextp;   in := (in+1) mod n;
      signal(mutex);   signal(full);
    until false;
  end
```

producer-consumer.**put(nextp);**

基于管程PC的消费者子程序设计

```
consumer:
Var nextc: item;
  begin
    repeat
      wait(full);   wait(mutex);
      nextc:=buffer[out];   out := (out+1) mod n;
      signal(mutex);   signal(empty);
       consume the item in nextc;
    until false;
  end
```

producer-consumer.get(nextc);

生产者-消费者管程设计要领

- 循环缓冲及操作机制
  - 循环缓冲 buffer: array [0..n-1] of item;
  - 生产/消费指针 in, out: integer := 0, 0;
  - 已有有效数据数量 **count**: integer := 0;
- 条件变量设计
  - noEmpty/noFull
- 针对每个条件变量
  - 设置进程等待队列
  - 设置同步操作原语wait与signal

生产者-消费者管程设计(待续)

```
TYPE producer-consumer = monitor;
  var in, out, count : integer;
      buffer: array [0..n-1] of item;
      noEmpty, noFull: condition;
  PROCEDURE entry put(nextp: item);
    BEGIN
      if count=n then wait(noEmpty);
      buffer[in] := nextp;   in := (in+1) mod n;
      count := count + 1;
      signal(noFull);
    END;
```

生产者-消费者管程设计(续)

```
PROCEDURE entry get(nextc: item);
  BEGIN
    if count=0 then wait(noFull);
    nextc:=buffer[out];   out := (out+1) mod n;
    count := count - 1;
    signal(noEmpty);
  END;

BEGIN
   in := out := count := 0;
END
```

2.5 管程

2.5.1 管程的引入及定义

2.5.2 管程内在机制实现要领

2.5.2 利用管程解决生产者—消费者问题

第二章 进程管理

2.1 进程的基本概念

2.2 进程控制

2.3 进程同步

2.4 经典进程同步问题

2.5 管程

2.6 进程通信

2.7 线程

2.6 进程通信

2.6.1 进程通信概念及分类

2.6.2 消息传递通信实现方式

2.6.3 消息传递系统实现若干问题

2.6.4 消息缓冲队列通信机制

进程通信概念与实现机制

- 进程通信概念
  - 指进程之间的信息交换
- 实现机制
  - 低级进程通信：效率低，主要针对控制信息的传送。典型实例为信号量机制。
  - 高级进程通信：能传送大量数据，效率高，进程通信实现细节由操作系统提供，整个通信过程对用户透明，通信程序编制简单

进程通信的类型

网络环境：
套接字方式

- 共享存储器系统
  - 基于共享数据结构的通信方式
  - 基于共享存储区的通信方式
- **消息传递系统**
  - 直接/间接通信方式
- 管道通信系统
  - 管道概念
  - 协调机制：互斥、同步、通信前提

2.6 进程通信

2.6.1 进程通信概念及分类

2.6.2 消息传递通信实现方式

2.6.3 消息传递系统实现若干问题

2.6.4 消息缓冲队列通信机制

直接通信方式

- 通信原语
  - Send(Receiver, message)
  - Receive(Sender, message)
- 一个接收进程可与多个发送进程通信
  - 打印进程
  - Sender无法事先指定，故为变参性质
- 基于进程直接通信原语的应用
  - 生产者-消费者通信过程

基于通信原语的生产者子程序设计

```
producer:
Var nextp: item;
  begin
    repeat
      produce an item in nextp;
      wait(empty);   wait(mutex);
      buffer[in] := nextp;   in := (in+1) mod n;
      signal(mutex);   signal(full);
    until false;
  end
```

Send(consumer,nextp);

基于通信原语的消费者子程序设计

```
consumer:
Var nextc: item;
  begin
    repeat
      wait(full);   wait(mutex);
      nextc:=buffer[out];   out := (out+1) mod n;
      signal(mutex);   signal(empty);
       consume the item in nextc;
    until false;
  end
```

Receive(producer, nextc);

间接通信方式

- 信箱
  - 进程间通信有关共享数据结构的中间实体
  - 由操作系统或用户进程创建
  - 私有/公有/共享信箱
  - 可实现实时/非实时通信
- 通信原语
  - 信箱的创建和撤销、消息的发送和接收
- 发送/接收进程间存在的四种关系
  - 一对一、多对一、一对多、多对多

2.6 进程通信

2.6.1 进程通信概念及分类

2.6.2 消息传递通信实现方式

2.6.3 消息传递系统实现若干问题

2.6.4 消息缓冲队列通信机制

消息传递系统中的几个问题

- 通信链路
  - 显式/隐式建立（计算机网络/单机）
  - 点-点或多点连接通信链路
  - 单向/双向通信链路
  - 无容量/有容量通信链路（缓冲区）
- 消息格式
  - 有消息头和消息正文构成，分定/变长两种
- 进程同步方式
  - 发送/接收进程阻塞与否（三种情况）

2.6 进程通信

2.6.1 进程通信概念及分类

2.6.2 消息传递通信实现方式

2.6.3 消息传递系统实现若干问题

2.6.4 消息缓冲队列通信机制

消息缓冲队列通信机制-数据结构

- 消息缓冲区

```
type MessageBuffer = record
  Sender;
        // 发送者进程标识符
  Size;  // 消息长度
  Text;  // 消息正文
  Next;
   // 指向下一缓冲区的指针
End;
```

- PCB通信数据项

```
type PCB = record
  messageQueue; //队首指针
  mutexOfMQ;
     // 消息队列互斥信号量
  semphoreOfMQ;
     // 消息队列资源信号量
  ……
End;
```

初始值?

消息缓冲队列通信机制示意图

消息缓冲队列通信机制-发送原语

```
Procedure Send(Receiver, SA)
Begin
  getbuf(SA.Size, Buffer_i);
  Buffer_i.Sender:=SA.Sender; Buffer_i.Size := SA.Size;
  Buffer_i.Text := SA.Text; Buffer_i.Next := 0;
  getid(PCB_Set, Receiver, PID);
  wait(PID.mutexOfMQ);
  insert(PID.messageQueue,Buffer_i);
  signal(PID.mutexOfMQ); signal(PID.semphoreOfMQ);
End;
```

消息缓冲队列通信机制-接收原语

```
Procedure Receive(RB)
Begin
  PID := InternalNameOfProcess();
  wait(PID.semphoreOfMQ); wait(PID.mutexOfMQ);
  remove(PID.messageQueue, Buffer_j);
  signal(PID.mutexOfMQ);
  RB.Sender := Buffer_j.Sender; RB.Size:=Buffer_j.Size;
  RB.Text := Buffer_j.Text; putbuf(Buffer_j);
End;
```

2.6 进程通信

2.6.1 进程通信概念及分类

2.6.2 消息传递通信实现方式

2.6.3 消息传递系统实现若干问题

2.6.4 消息缓冲队列通信机制

作业题

- **2.15** 简明扼要地谈谈你对各种进程通信方式的认识与理解，并着重就消息缓冲队列通信机制进行分析与描述。
- **2.16** 为什么要引入管程？并就管程的组成和同步互斥机理展开简明扼要的讨论。

第二章 进程管理

2.1 进程的基本概念

2.2 进程控制

2.3 进程同步

2.4 经典进程同步问题

2.5 管程

2.6 进程通信

2.7 线程

2.7 线程

2.7.1 线程的基本概念

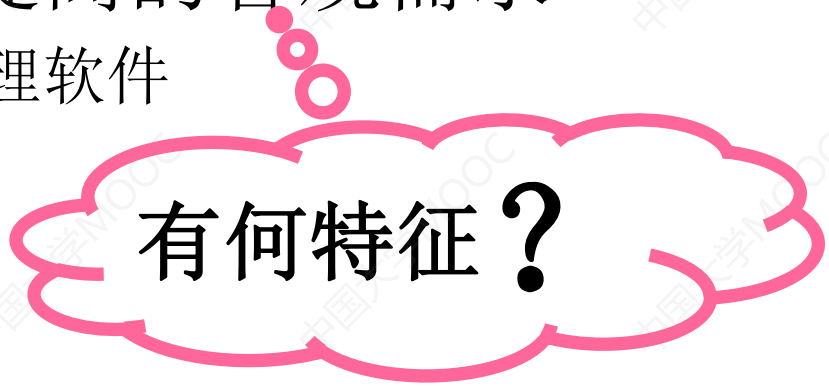
2.7.2 线程的控制

2.7.3 线程间的同步与通信

2.7.4 线程的实现机制

线程的引入

- 进程并发机制缺陷分析
  - 进程并发执行基础（资源拥有与调度分派基本单位）
  - 进程创建、切换和撤销等操作时空开销较大
  - 进程并发执行程度及进程间通信效率受限
- 系统并发程度进一步提高的客观需求
  - 事务处理软件、数据库处理软件
  - 窗口系统及操作系统自身
  - 多处理机系统
- 对策
  - 资源拥有与调度分派两种属性的分离



线程的特征

与进程有何区别与联系？

- 轻型实体及共享进程资源
- 独立调度和分派的基本单位
- 创建、撤销、切换等系统开销
- 地址空间共享及通信效率
- 系统并发执行程度大大提高

？

2.7 线程

2.7.1 线程的基本概念

2.7.2 线程的控制

2.7.3 线程间的同步与通信

2.7.4 线程的实现机制

线程的控制

- 线程基本属性
  - 线程标识符、寄存器状态、堆栈及专有存储器
  - 线程状态、优先级与信号屏蔽
- 线程的创建和终止
  - 初始化线程
  - 线程创建函数或系统调用
  - 线程完成性终止与被迫强制性终止
  - 被终止线程重新恢复运行

某些系统线程一旦建立恒运行

“等待线程终止”连接命令

2.7 线程

2.7.1 线程的基本概念

2.7.2 线程的控制

2.7.3 线程间的同步与通信

2.7.4 线程的实现机制

线程间的同步和通信

- 信号量机制
  - 私有信号量/公有信号量
- 互斥锁
  - 实现资源互斥使用的简单机制
  - lock/unlock两种状态和原语操作
  - 适合于高频使用的关键共享数据和程序段
- 条件变量与互斥锁
  - 互斥锁与特定条件变量相联系
  - 互斥锁用于短期锁定，保证互斥进入临界区
  - 条件变量用于长期等待，直至所等资源可用

基于互斥锁和条件变量的线程同步

Lock(mutex);
Check data structure of Resource;
if Resource is busy
  begin
    Unlock(mutex);
    wait(condition_variable);
    Lock(mutex);
  end

Mark resource as busy;
Unlock(mutex);

请求访问资源入口

访问资源结束出口

Lock(mutex);
Mark resource as free;
Unlock(mutex);
wakeup(condition_variable);

2.7 线程

2.7.1 线程的基本概念

2.7.2 线程的控制

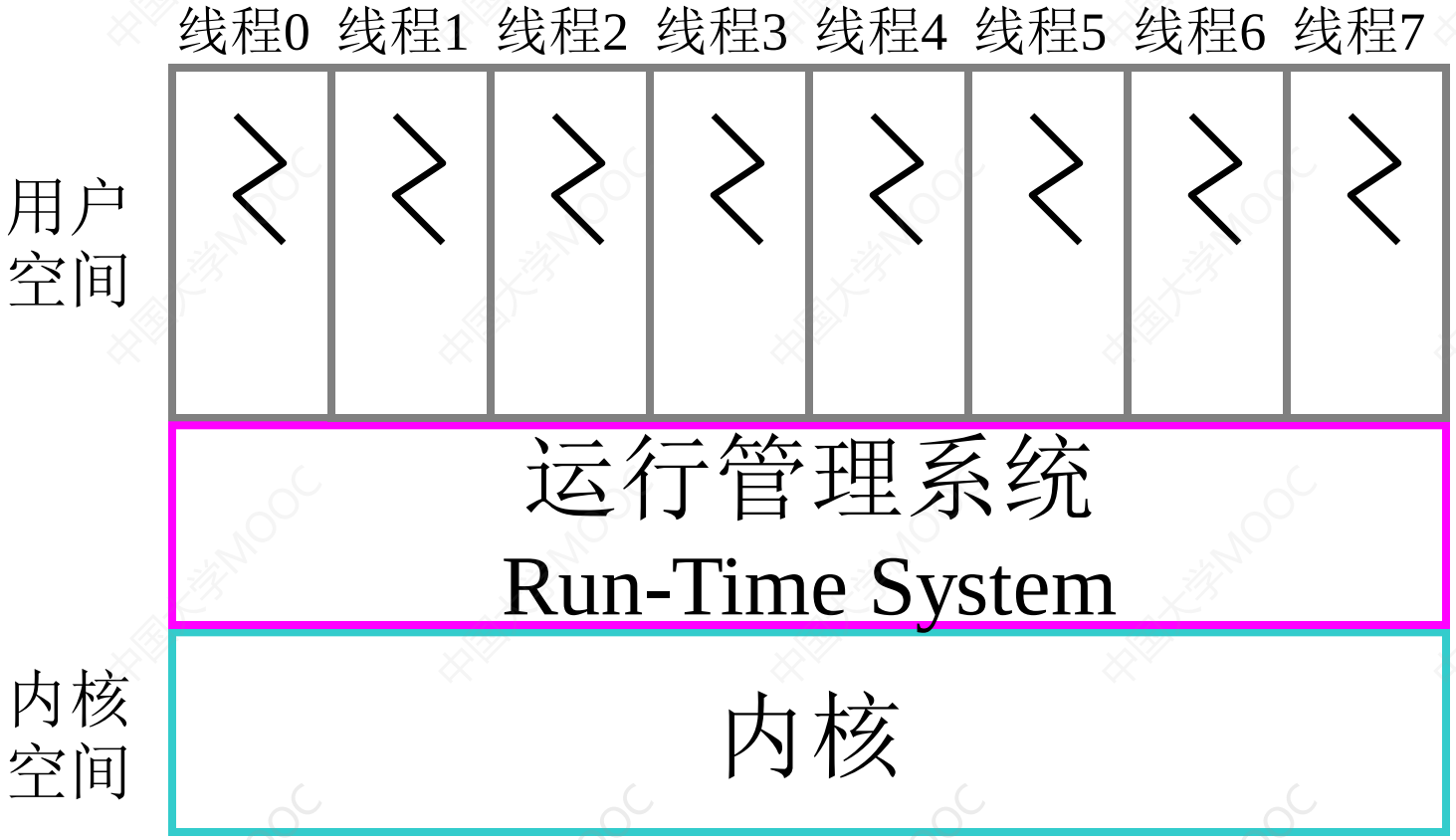
2.7.3 线程间的同步与通信

2.7.4 线程的实现机制

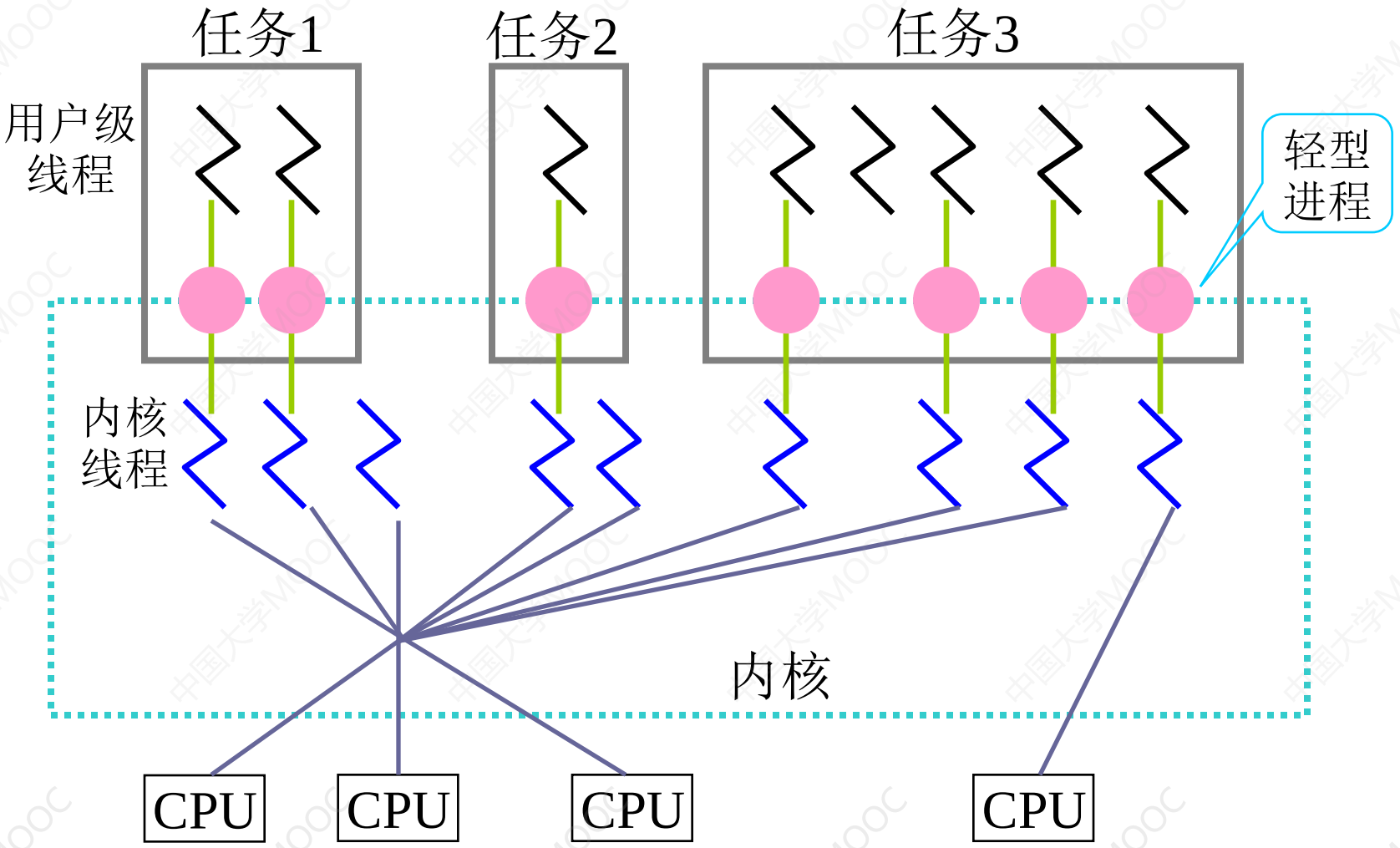
线程实现方式

- 内核支持线程
  - 依赖于内核及其中的线程控制块
- 用户级线程
  - 仅存在于用户空间中，与内核无关
- 实现方式比较
  - 操作系统要求、系统开销与性能、调度算法灵活性、并发线程数量及实际效果
  - 系统调用阻塞链锁效应
  - 时钟中断与轮转调度可行性
  - 线程调度与执行时间（**例：**进程A[1线程]｜B[100线程]）

用户级线程模型1—运行管理系统



用户级线程模型2–轻型进程



2.7 线程

2.7.1 线程的基本概念

2.7.2 线程的控制

2.7.3 线程间的同步与通信

2.7.4 线程的实现机制

作业题

- **2.17** 为什么要引入线程？其与进程之间存在哪些区别与联系？从实现方式看，有哪些类型的线程，并加以扼要说明。
- **2.18** 简述线程间的同步与互斥机制。

第二章 进程管理

2.1 进程的基本概念
2.2 进程控制
2.3 进程同步
2.4 经典进程同步问题
2.5 管程
2.6 进程通信
2.7 线程

第二章习题讲解

作业题

- **2.4** 试举例说明引起进程创建、撤消、阻塞或被唤醒的主要事件分别有哪些？
- **2.3** 试对进程的状态及状态转换进行总结，注意状态转换的物理含义及转化条件。
- **2.2** 比较程序和进程。
- **2.1** 比较程序的顺序执行和并发执行。

作业题

- **2.5** 试根据你自己的理解，采用类C语言设计和描述操作系统关于进程控制块的数据结构、组织方式及管理机制。在此基础上，给出进程的创建、终止、阻塞、唤醒、挂起与激活等函数原型及函数代码。注意，对于过于复杂的功能或你无法解决的细节可采用指定功能的函数模块来替代。如处理机调度scheduler()

作业题

- **2.7** 试阐述你对整型信号量机制与记录型信号量机制的完整理解以及AND型信号量机制与一般信号量集机制的基本思想。
- **2.6** 什么是临界资源和临界区？试举例说明。并谈谈你对进程同步机制准则的理解。

作业题

- **2.8** 在生产者—消费者问题中，如果缺少了signal(full)或signal(empty)，对执行结果会有何影响？
- signal(full)
  - 生产者
  - 消费者

开始 → 生产者生产数据填满n个缓冲区时

- signal(empty)
  - 生产者
  - 消费者

开始 → 消费者取走n个缓冲区的数据时

作业题

- **2.9** 在生产者—消费者问题中，如果将两个wait操作即wait(full)和wait(mutex)互换位置；或者是将signal(mutex)与signal(full)互换位置，结果会如何？
- **wait(full)和wait(mutex)互换位置**
  - 消费者 **wait(mutex) => wait(full)**
  - 生产者 **wait(empty) => wait(mutex)**
  - **时间节点：循环缓冲均为空缓冲区时**
- **signal(mutex)与signal(full)互换位置**

作业题

- **2.10[必做]** 我们为临界区设置一把锁W，当W=1时，表示关锁；W=0时，表示锁已打开。试写出开锁原语与关锁原语，并利用它们去实现互斥。

作业题2.10参考答案

```
int W=0, wcount = 0;
Semaphore ws = 0;
```

关锁原语Lock(W)：

```
if (W==1)
{ wcount++; wait(ws); }
W = 1;
```

开锁原语Unlock(W)：

```
W:=0;
if (wcount>0)
{ wcount--; signal(ws); }
```

互斥实现：

```
repeat
……
  Lock(W);
  临界区
  Unlock(W);
……
until false
```

作业题

- **2.11** 试修改下面生产者—消费者问题解法中的错误：

$producer_i$:	$consumer_j$:

Var nextp: item;
begin
repeat
produce an item in nextp;
wait(mutex); wait(full);
buffer[in] := nextp;
signal(mutex);
until false;
end
	Var nextc: item;
	begin
	repeat
	wait(mutex); wait(empty);
	nextc:=buffer[out]; out := out+1;
	signal(mutex);
	consume the item in nextc;
	until false;
	end

作业题

- **2.12[必做]** 试利用记录型信号量写出一个不会出现死锁的哲学家进餐问题的算法。
- **2.13[必做]** 课本在测量控制系统中的数据采集任务，把所采集数据送一单缓冲区；计算任务从该单缓冲区中取出数据进行计算。写出利用信号量机制实现两者共享单缓冲的同步算法。

主程序设计

```
Var buffer: data;
      empty, full : semphore := 1, 0 ;
begin
   parbegin
      ProcessOfCaiJiShuJu;
      ProcessOfJiSuan;
   parend
end
```

数据采集任务子程序设计

```
ProcessOfCaiJiShuJu:
Var nextp: data;
  begin
    repeat
      collect data in nextp;
      wait(empty);
      buffer := nextp;
      signal(full);
    until false;
  end
```

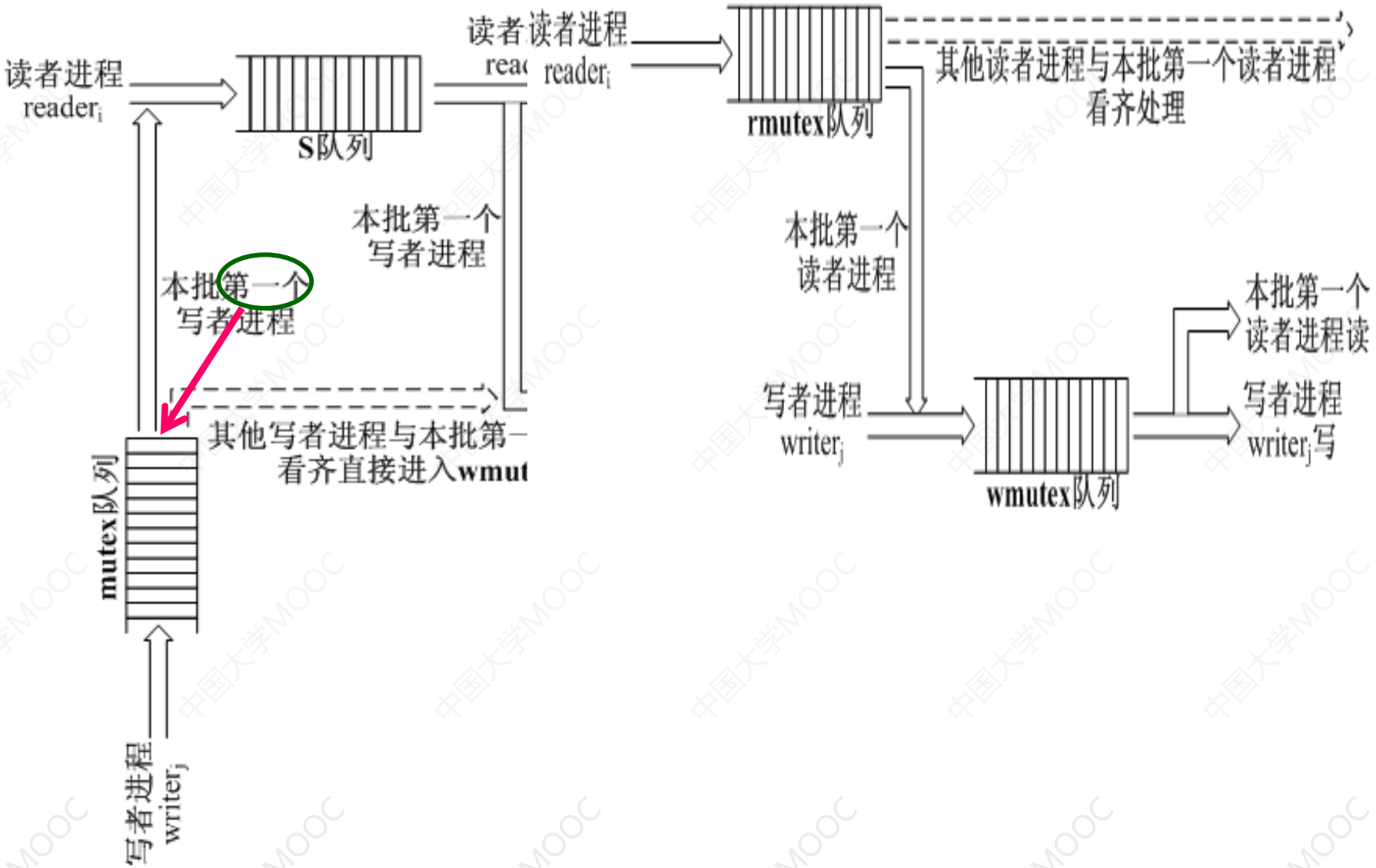
计算任务子程序设计

```
ProcessOfJiSuan :
Var nextc: data;
  begin
    repeat
      wait(full);
      nextc:=buffer;
      signal(empty);
      calculate using the data in nextc;
    until false;
  end
```

作业题

- **2.15** 简明扼要地谈谈你对各种进程通信方式的认识与理解，并着重就消息缓冲队列通信机制进行分析与描述。
- **2.14[必做]** 给出基于记录型信号量机制的写者优先的读者-写者问题的同步解决方案。

写者优先读者-写者问题同步解决方案



读者－写者主程序设计

Var readercount, **writercount**: integer := 0, **0**;

S, **mutex**,rmutex,wmutex : semphore := **1**,**1**,1,1;

begin

parbegin

$reader_1$; … ; $reader_i$; … ; $reader_m$;

$writer_1$; … ; $writer_j$; … ; $writer_n$;

parend

end

读者子程序设计

reader$_i$:
begin
repeat

wait(S);
wait(rmutex);
if readercount=0 then wait(wmutex);
readercount := readercount +1;
signal(rmutex);
signal(S);
Perform read operation;
wait(rmutex);
readercount := readercount -1;
if readercount=0 then signal(wmutex);
signal(rmutex);

until false;
end

写者子程序设计

writer$_i$:
begin
repeat

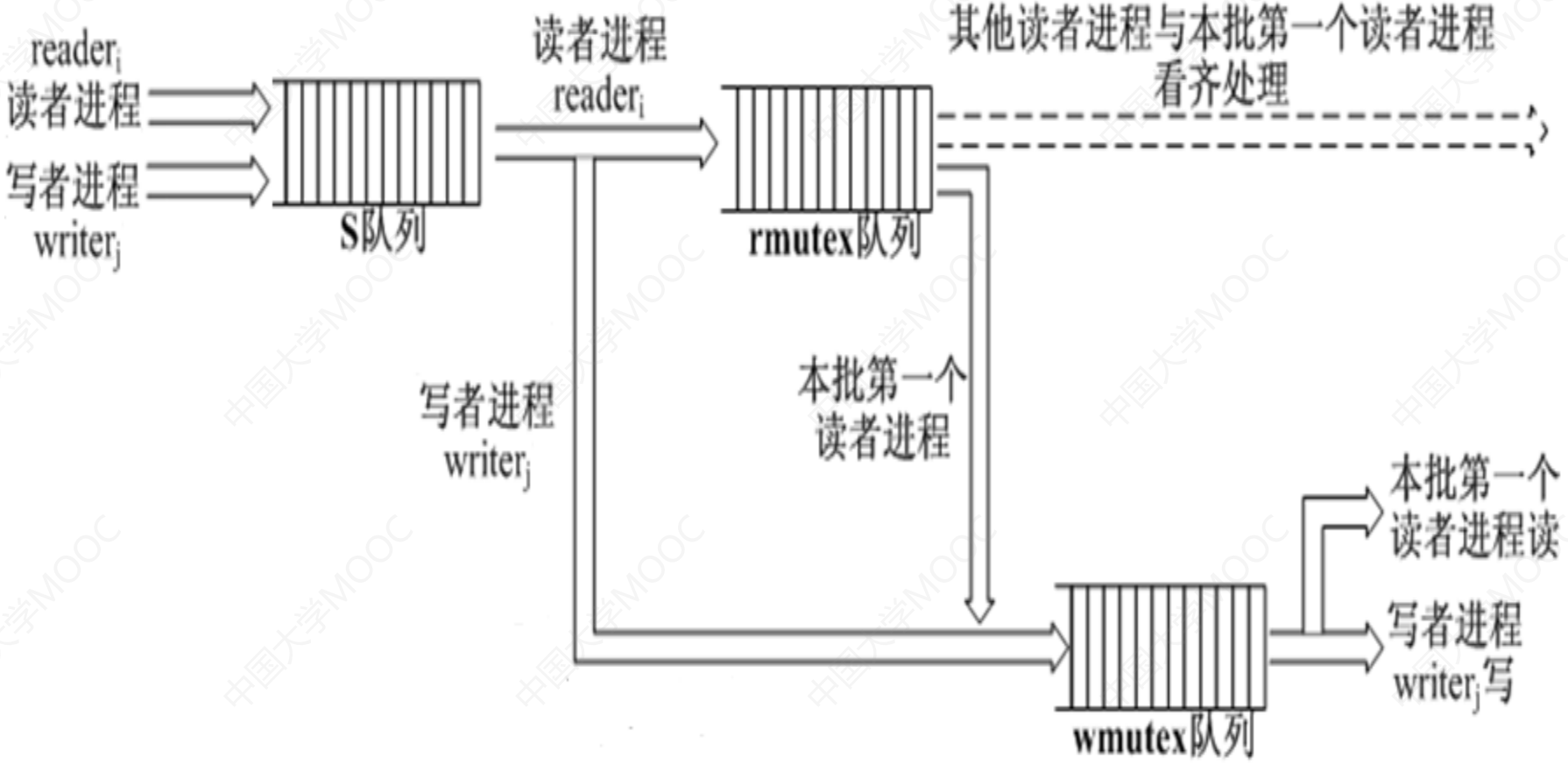
wait(mutex);
if writercount=0 then wait(S);
writercount := writercount +1;
signal(mutex);

wait(wmutex);Perform write operation; **signal(wmutex);**

wait(mutex);
writercount := writercount -1;
if writercount=0 then signal(S);
signal(mutex);

until false;
end

公平型读者-写者问题同步解决方案



公平型读者－写者主程序设计

```
Var readercount: integer := 0;
      S, rmutex,wmutex : semphore := 1,1,1;
begin
   parbegin
      reader1; … ; readeri ; … ; readerm;
      writer1; … ; writerj ; … ; writern;
   parend
end
```

公平型读者子程序设计

```
readeri:
  begin
    repeat
        wait(S);
        wait(rmutex);
        if readercount=0 then wait(wmutex);
        readercount := readercount +1;
        signal(rmutex);
        signal(S);
        Perform read operation;
        wait(rmutex);
        readercount := readercount -1;
        if readercount=0 then signal(wmutex);
        signal(rmutex);
    until false;
  end
```

公平型写者子程序设计

```
writeri:
  begin
    repeat
      wait(S);
      wait(wmutex);
      Perform write operation;
      signal(wmutex);
      signal(S);
    until false;
  end
```

作业题

- **2.16** 为什么要引入管程？并就管程的组成和同步互斥机理展开简明扼要的讨论。
- **2.17** 为什么要引入线程？其与进程之间存在哪些区别与联系？从实现方式看，有哪些类型的线程，并加以扼要说明。
- **2.18** 简述线程间的同步与互斥机制。

同学们，再见！

2021年4月27日星期二