第二章　进程管理

翁楚良

https://chuliangweng.github.io

2023 春　ECNU

2.1 进程

- 进程模型
- 进程创建和终止
- 进程层次结构
- 进程状态及转换
- 进程的实现
- 线程及实现

进程的实现

- 进程控制块(PCB, process control block)
  - 进程控制块是由OS维护的用来记录进程相关信息的一块内存
  - 每个进程在OS中的登记表项（可能有总数目限制），OS据此对进程进行控制和管理（PCB中的内容会动态改变）
- 进程表(process table)
  - 操作系统维护的一个结构体数组
  - 该数组的一个元素(即进程表项)，对应于一个进程控制块

MINIX3进程控制块

- 在MINIX3中进程通信、内存管理和文件管理是由系统中的几个模块分别处理，左图列出进程控制块(进程表)一些重要的域。

Kernel	Process management	File management
Registers	Pointer to text segment	UMASK mask
Program counter	Pointer to data segment	Root directory
Program status word	Pointer to bss segment	Working directory
Stack pointer	Exit status	File descriptors
Process state	Signal status	Real id
Current scheduling priority	Process ID	Effective UID
Maximum scheduling priority	Parent process	Real GID
Scheduling ticks left	Process group	Effective GID
Quantum size	Children's CPU time	Controlling tty
CPU time used	Real UID	Save area for read/write
Message queue pointers	Effective UID	System call parameters
Pending signal bits	Real GID	Various flag bits
Various flag bits	Effective GID	
Process name	File info for sharing text	
	Bitmaps for signals	
	Various flag bits	
	Process name	

Minix的进程表项(kernel维护)

```
17: struct proc {
18:   struct stackframe_s p_reg; /* process' registers saved in stack frame */
19:   reg_t p_ldt_sel;/* selector in gdt with ldt base and limit */
20:   struct segdesc_s p_ldt[2+NR_REMOTE_SEGS]; /* CS, DS and remote segments */
21:
22:   proc_nr_t p_nr; /* number of this process (for fast access) */
23:   struct priv *p_priv;/* system privileges structure */
24:   char p_rts_flags;/* SENDING, RECEIVING, etc. */
25:
26:   char p_priority;/* current scheduling priority */
27:   char p_max_priority;/* maximum scheduling priority */
28:   char p_ticks_left; /* number of scheduling ticks left */
29:   char p_quantum_size;/* quantum size in ticks */
30:
31:   struct mem_map p_memmap[NR_LOCAL_SEGS];/* memory map (T, D, S) */
32:
33:   clock_t p_user_time;/* user time in ticks */
34:   clock_t p_sys_time;/* sys time in ticks */
35:
36:   struct proc *p_nextready; /* pointer to next ready process */
37:   struct proc *p_caller_q; /* head of list of procs wishing to send */
38:   struct proc *p_q_link; /* link to next proc wishing to send */
39:   message *p_messbuf;/* pointer to passed message buffer */
40:   proc_nr_t p_getfrom;/* from whom does process want to receive? */
41:   proc_nr_t p_sendto;/* to whom does process want to send? */
42:
43:   sigset_t p_pending;/* bit map for pending kernel signals */
44:
45:   char p_name[P_NAME_LEN]; /* name of the process, including \0 */
46: } « end proc » ;
```

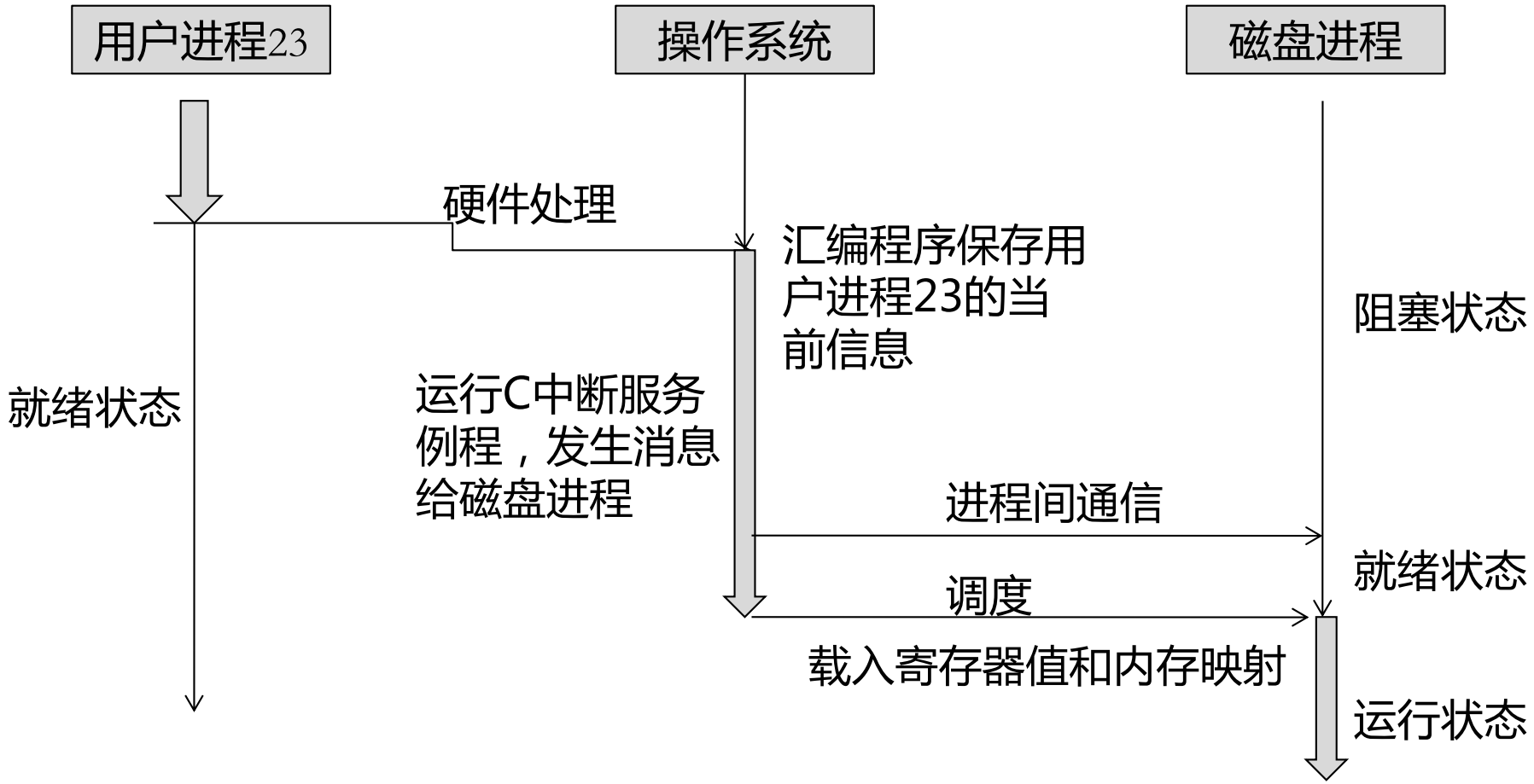
Minix的进程表项(kernel维护)

```
89: /* The process table and pointers to process table slots. The pointers allow
90:  * faster access because now a process entry can be found by indexing the
91:  * pproc_addr array, while accessing an element i requires a multiplication
92:  * with sizeof(struct proc) to determine the address.
93:  */
94: EXTERN struct proc proc[NR_TASKS + NR_PROCS]; /* process table */
95: EXTERN struct proc *pproc_addr[NR_TASKS + NR_PROCS];
96: EXTERN struct proc *rdy_head[NR_SCHED_QUEUES]; /* ptrs to ready list headers */
97: EXTERN struct proc *rdy_tail[NR_SCHED_QUEUES]; /* ptrs to ready list tails */
98:
```

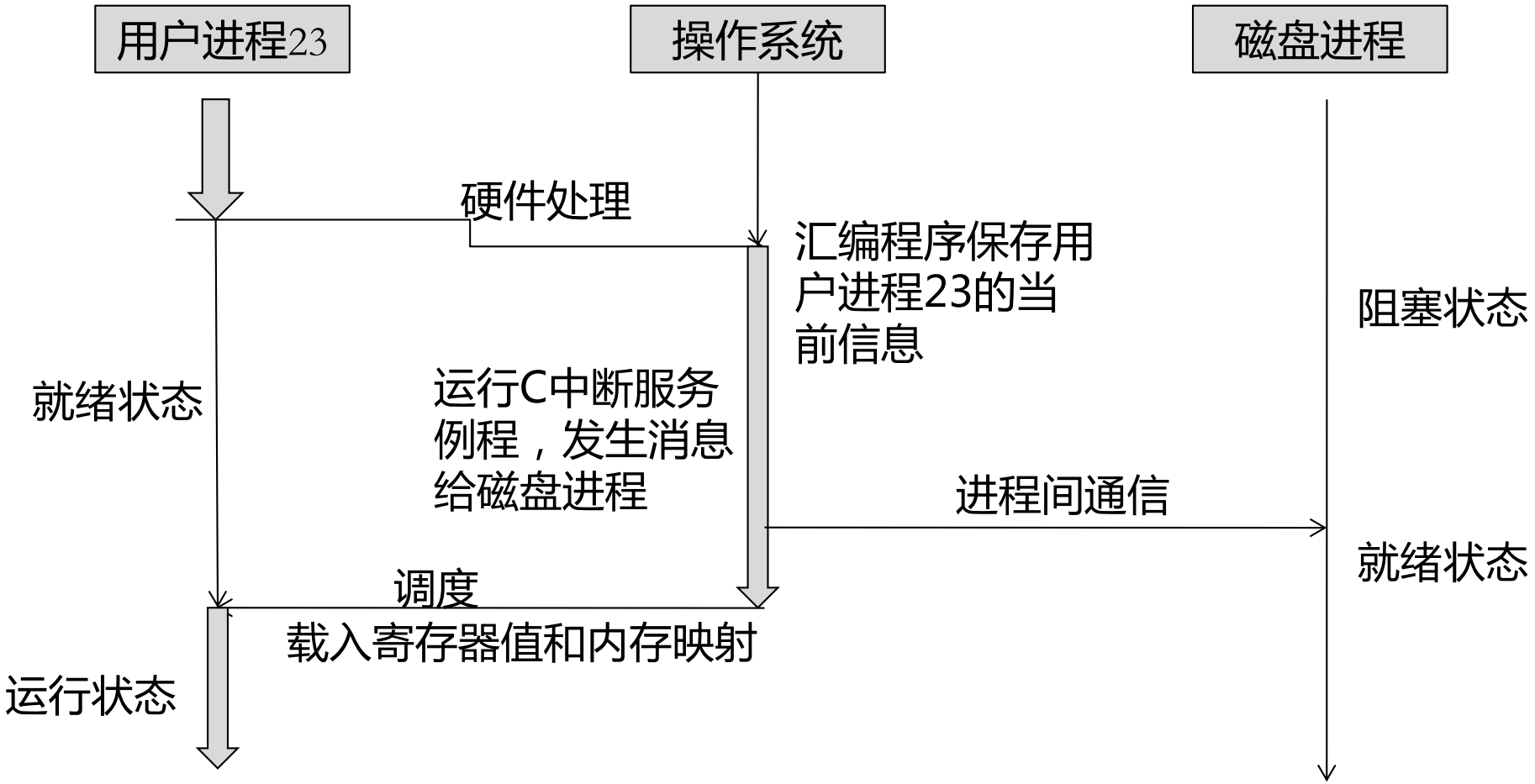
中断处理步骤

- 1.硬件压栈程序计数器等
- 2.硬件按中断向量下载新的程序计数器
- 3.汇编语言程序存储寄存器值 (至当前进程的PCB)
- 4.汇编语言程序设置新的栈 (为运行处理程序作准备)
- 5.C语言中断服务程序运行
- 6.对等待的就绪任务作标识 (发生中断后相关进程状态发生变化)
- 7.调度程序决定哪个进程是下一个将运行的
- 8.C语言中断服务程序返回汇编代码
- 9.汇编语言程序开始运行新的当前进程 (载入寄存器值和内存映射)

情况一


用户进程23
操作系统
磁盘进程
硬件处理
汇编程序保存用户进程23的当前信息
阻塞状态
就绪状态
运行C中断服务例程，发生消息给磁盘进程
进程间通信
就绪状态
调度
载入寄存器值和内存映射
运行状态

情况二


用户进程23
操作系统
磁盘进程
硬件处理
汇编程序保存用户进程23的当前信息
阻塞状态
就绪状态
运行C中断服务例程，发生消息给磁盘进程
进程间通信
就绪状态
调度
载入寄存器值和内存映射
运行状态

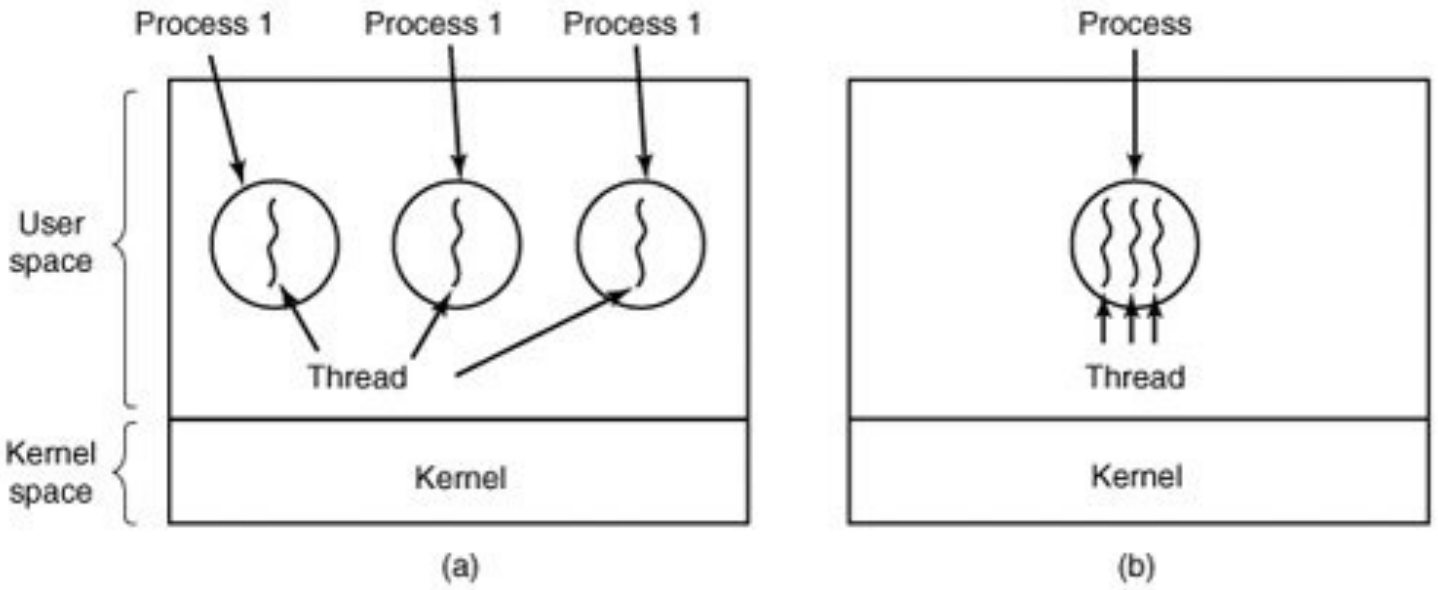
2.1 进程

- 进程模型
- 进程创建和终止
- 进程层次结构
- 进程状态及转换
- 进程的实现
- 线程及实现

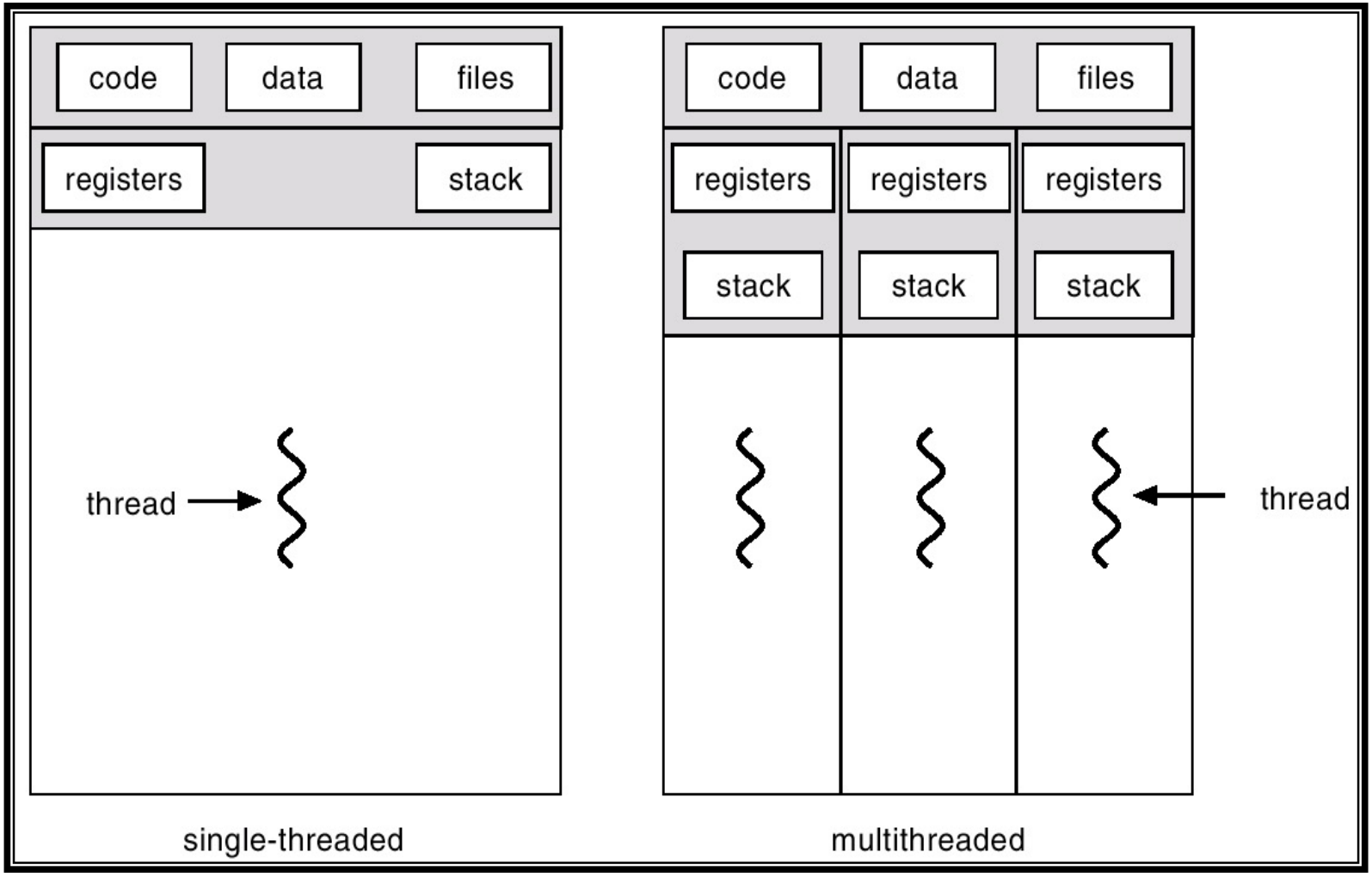
线程

- 通常情况下，一个进程只有一个地址空间和一个控制流
- 有些情况下，一个进程在一个地址空间中可以有多个控制流，即线程(thread)
- 线程：作为CPU调度单位，而进程作为除CPU之外其他资源分配单位。
  - 只拥有必不可少的资源，如：线程状态、寄存器上下文和栈
  - 同样具有就绪、阻塞和执行三种基本状态
- 线程的优点：减小并发执行的时间和空间开销（线程的创建、退出和调度），因此容许在系统中建立更多的线程来提高并发程度。
  - 线程的创建时间比进程短；
  - 线程的终止时间比进程短；
  - 同进程内的线程切换时间比进程短；
  - 由于同进程内线程间共享内存和文件资源，可直接进行不通过内核的通信；

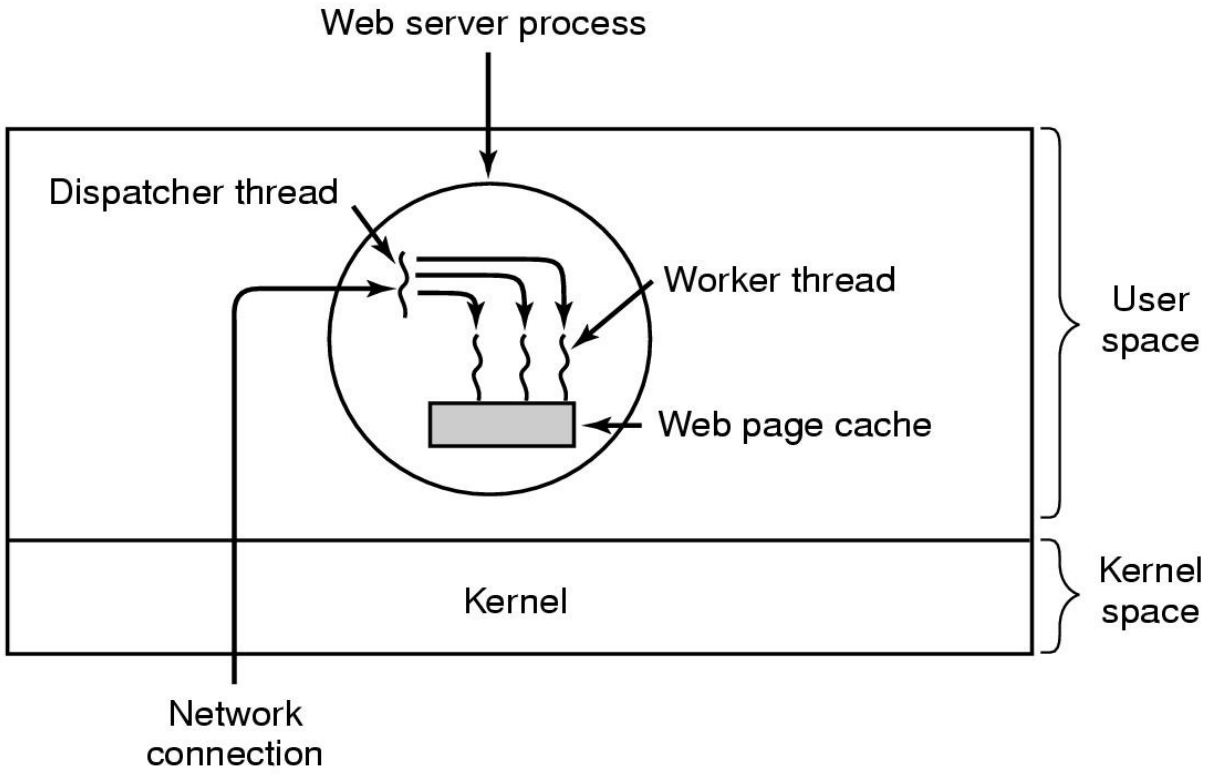
进程与线程关系


Process 1
Process 1
Process 1
User space
Thread
Kernel space
Kernel
(a)
Process
Thread
Kernel
(b)

单线程进程与多线程进程



Thread usage example: Web Server



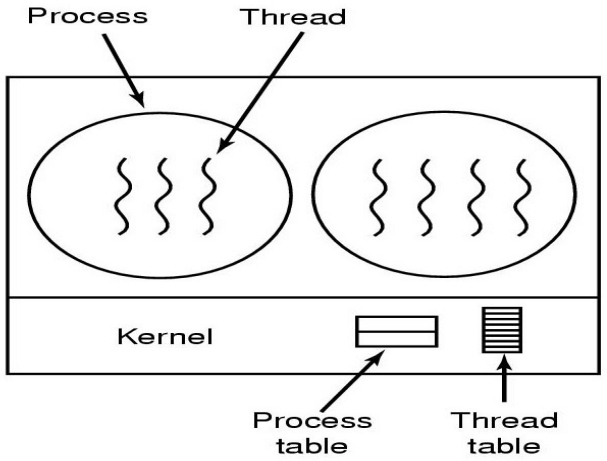
- It needs to handle several files/pages requests over a short period.
- Hence more efficient to create (and destroy) a single thread for each request.
- On a SMP machine: multiple threads can possibly be executing simultaneously on different processors.

线程的表示

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

线程实现方式(1)

- 内核线程(kernel-level thread)
  - 依赖于OS核心，由内核的内部需求进行创建和撤销，用来执行一个指定的函数
    - 内核维护进程和线程的上下文信息；
    - 线程切换由内核完成；
    - 一个线程发起系统调用而阻塞，不会影响其他线程的运行。
    - 时间片分配给线程，所以多线程的进程获得更多CPU时间。



- Windows NT/2000/XP
- OS/2
- Solaris 2
- Tru64 UNIX
- BeOS
- Linux

例子：Linux线程

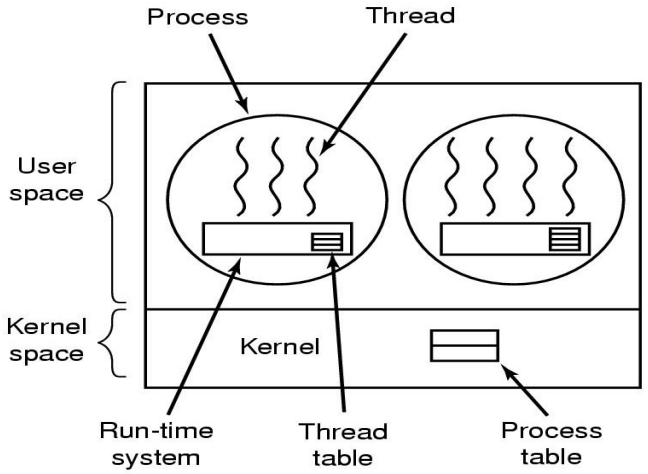
```c
#include <sys/types.h>
#include <pthread.h>

long  shared = 0;

int cal(void){
  long i = 0;
  while(i < 50000) i++;
  return 5;
}

void * fun(void * args){
  long i;
  for(i = 0; i < 10000; i++) {
    shared = cal() + shared + cal();
  }

  return NULL;
}

int main(){

  pthread_t thread_id_1;

  pthread_create(&thread_id_1, NULL, fun, NULL);

  pthread_join(thread_id_1, NULL);

  printf("shared: %d\n", shared);

  return 0;
}
```

1.c

线程实现方式(2)

- 用户线程(user-level thread)
  - 不依赖于OS核心，应用进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。
  - 调度由应用软件内部进行，通常采用非抢先式和更简单的规则，也无需用户态/核心态切换，所以速度特别快
  - 一个线程发起系统调用而阻塞，则整个进程在等待。时间片分配给进程，多线程则每个线程就慢
- 用户线程的维护由应用进程完成
- 内核不了解用户线程的存在
- 用户线程切换不需要内核特权
- 用户线程调度算法可针对应用优化

- POSIX Pthreads
- Mach C-threads
- Solaris UI-threads



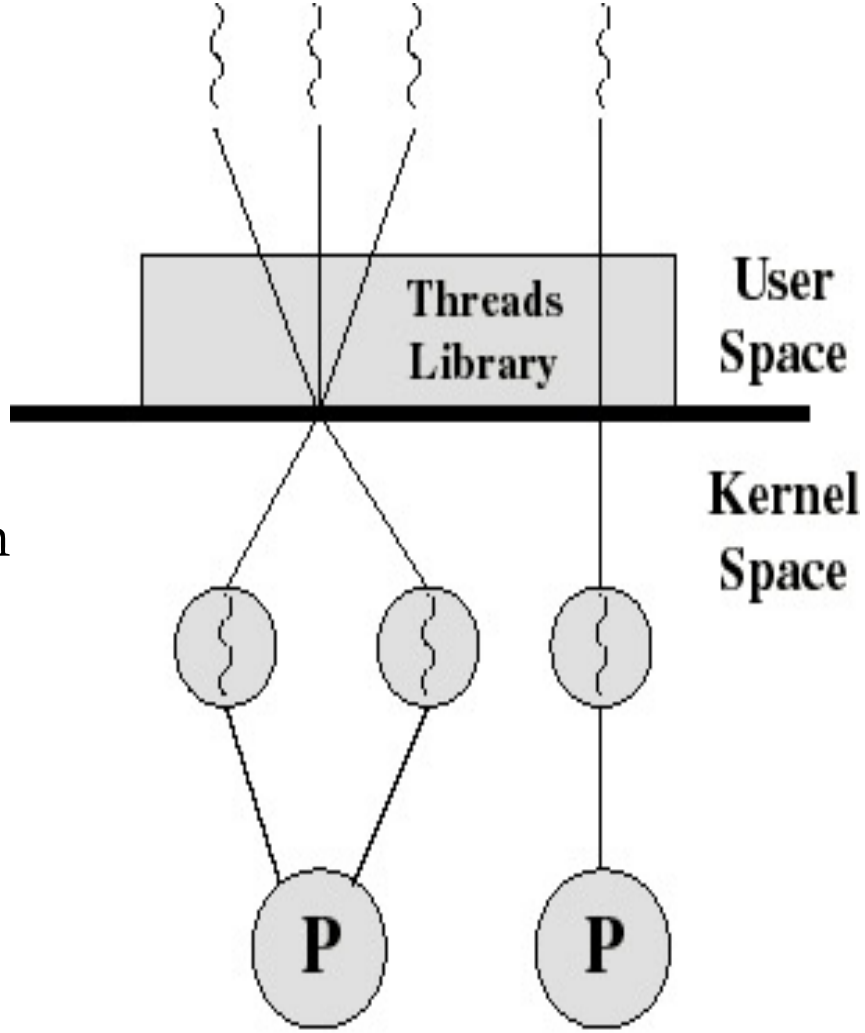
线程操作时延(us)

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

T. Anderson, et al, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", ACM TOCS, February 1992.

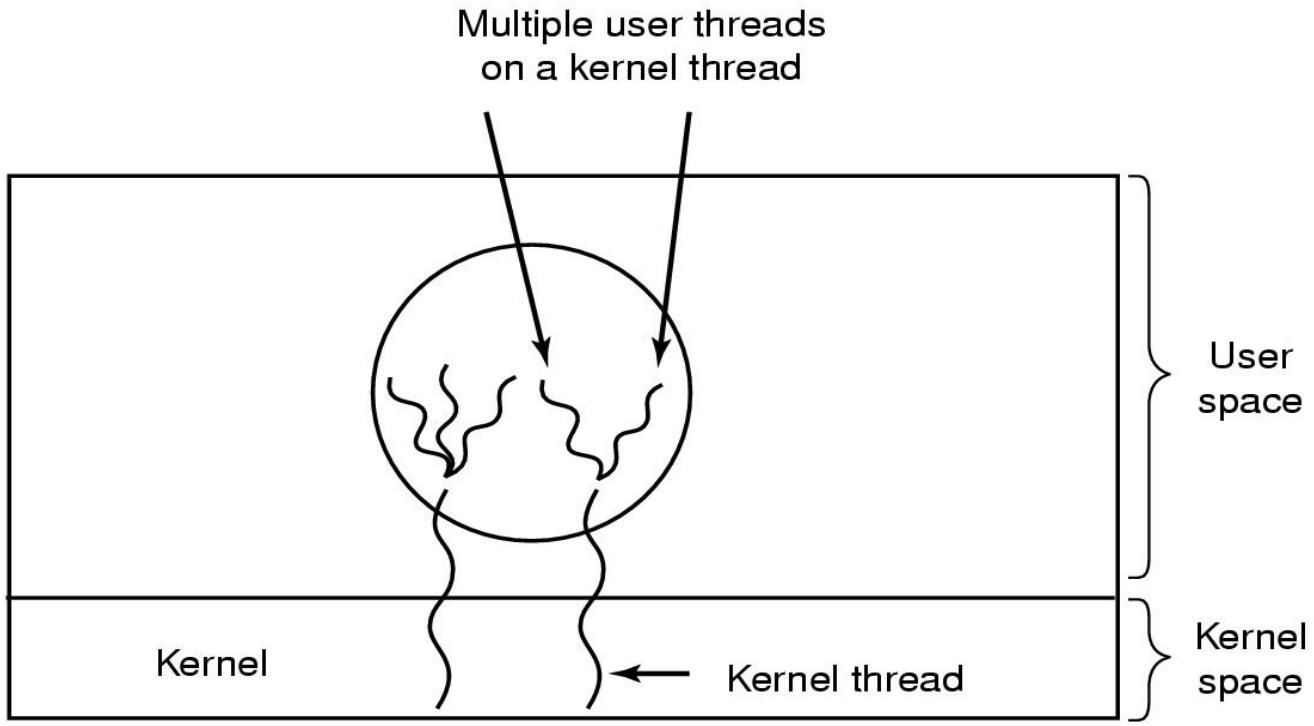
线程实现方式(3)

- 混合(hybrid)方案
  - Thread creation done in the user space.
  - Bulk of scheduling and synchronization of threads done in the user space.
  - The programmer may adjust the number of KLTs.
  - May combine the best of both approaches.
  - Example is Solaris.



Hybrid Implementation (Solaris)

- Multiplexing user-level threads onto kernel-level threads.



进程和线程的比较

- 地址空间和其他资源（如打开文件）：进程间相互独立，同一进程的各线程间共享 — 某进程内的线程在其他进程不可见
- 通信：进程间通信IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信 — 需要进程同步和互斥手段的辅助，以保证数据的一致性
- 调度：线程上下文切换比进程上下文切换要快得多；

多线程的优点和缺点实际上是对立统一的，由于线程共享进程的地址空间，因此可能会导致竞争，因此对某一块有多个线程要访问的数据需要一些同步技术。

第二章进程管理 提纲

- 2.1 进程
- 2.2 进程间通信
- 2.3 经典并发问题
- 2.4 进程调度
- 2.5 MINIX3进程概述
- 2.6 MINIX3进程实现
- 2.7 MINIX3系统任务
- 2.8 MINIX3时钟任务

进程间通信

- 进程间通信涉及三方面问题
  - 一个进程如何向另一个进程传送信息
  - 两个或多个进程在涉及临界活动时不会彼此影响
  - 相互依存关系的进程需要保证正确的执行次序

对于线程间通信，通过共享的地址空间可以很容易实现传递信息；进程间通信中的另两方面问题解决方法亦适用于线程。

进程间通信

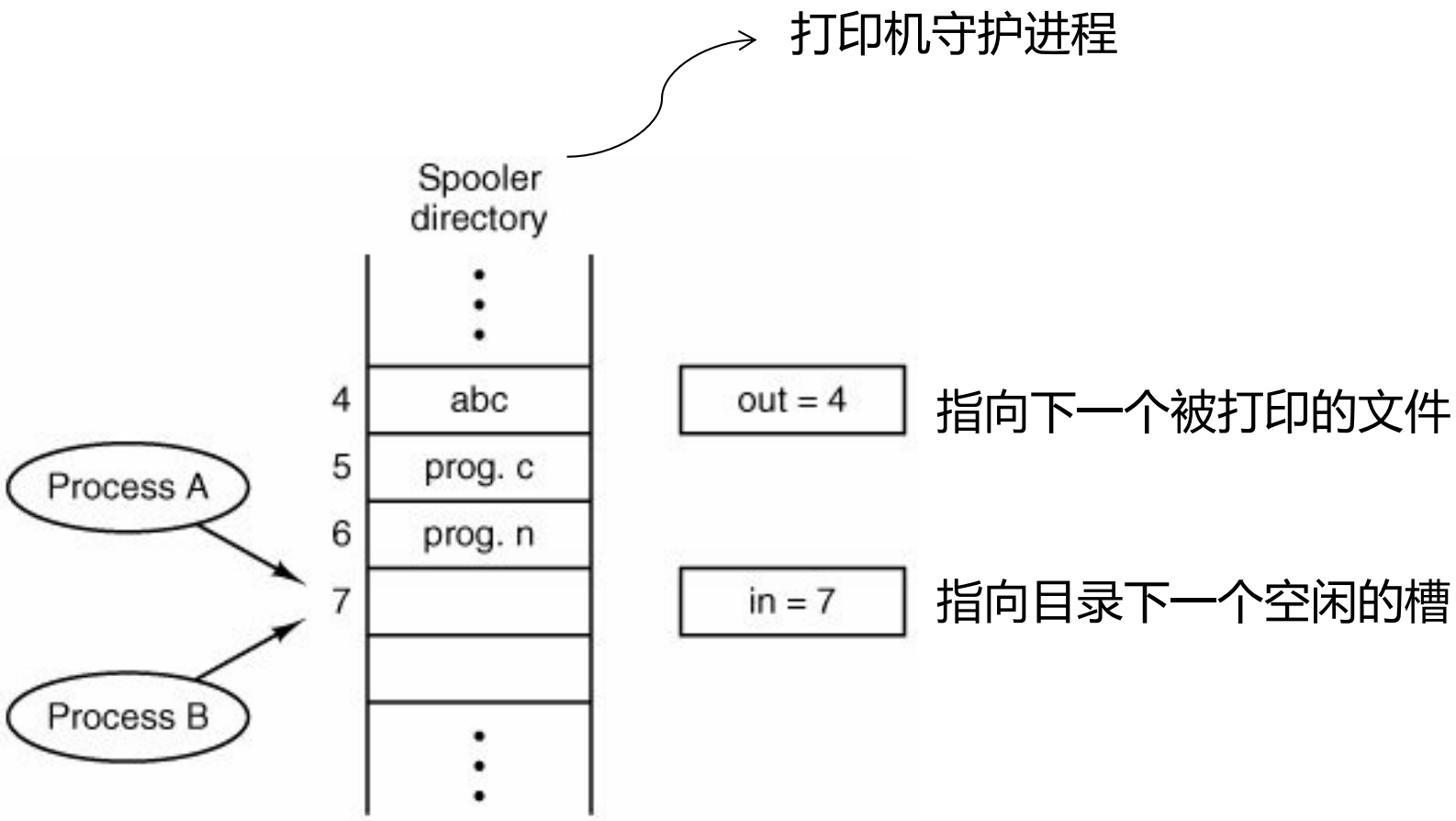
- 竞争条件
- 临界区
- 忙等待形式互斥
- 睡眠和唤醒
- 信号量
- 互斥
- 管程
- 消息传递

竞争条件(race conditions)

- 两个或多个进程读写某些共享数据，而最后的结果取决于进程运行的精确时序，就称为竞争条件(race conditions)。

- 存在竞争条件的多进程，大多数的运行结果都很好，但在极少数情况下发生一些无法解释的奇怪的事情。

例子: 假脱机打印程序


打印机守护进程
Spooler directory
4
abc
5
prog. c
6
prog. n
7
Process A
Process B
out = 4
指向下一个被打印的文件
in = 7
指向目录下一个空闲的槽

例子：线程共享变量

```c
#include <sys/types.h>
#include <pthread.h>

long  shared = 0;

int cal(void){
  long i = 0;
  while(i < 50000) i++;
  return 5;
}

void * fun(void * args){
  long i;
  for(i = 0; i < 10000; i++) {
    shared = cal() + shared + cal();
  }

  return NULL;
}

int main(){

  pthread_t thread_id_1;

  pthread_create(&thread_id_1, NULL, fun, NULL);

  pthread_join(thread_id_1, NULL);

  printf("shared: %d\n", shared);

  return 0;
}
```

1.c

```c
long  shared = 0;

int cal(void){
  long i = 0;
  while(i < 50000) i++;
  return 5;
}

void * fun(void * args){
  long i;
  for(i = 0; i < 10000; i++) {
    shared = cal() + shared + cal();
  }

  return NULL;
}

int main(){

  pthread_t thread_id_1;
  pthread_t thread_id_2;

  pthread_create(&thread_id_1, NULL, fun, NULL);
  pthread_create(&thread_id_2, NULL, fun, NULL);

  pthread_join(thread_id_1, NULL);
  pthread_join(thread_id_2, NULL);

  printf("shared: %d\n", shared);

  return 0;
}
```

2.c

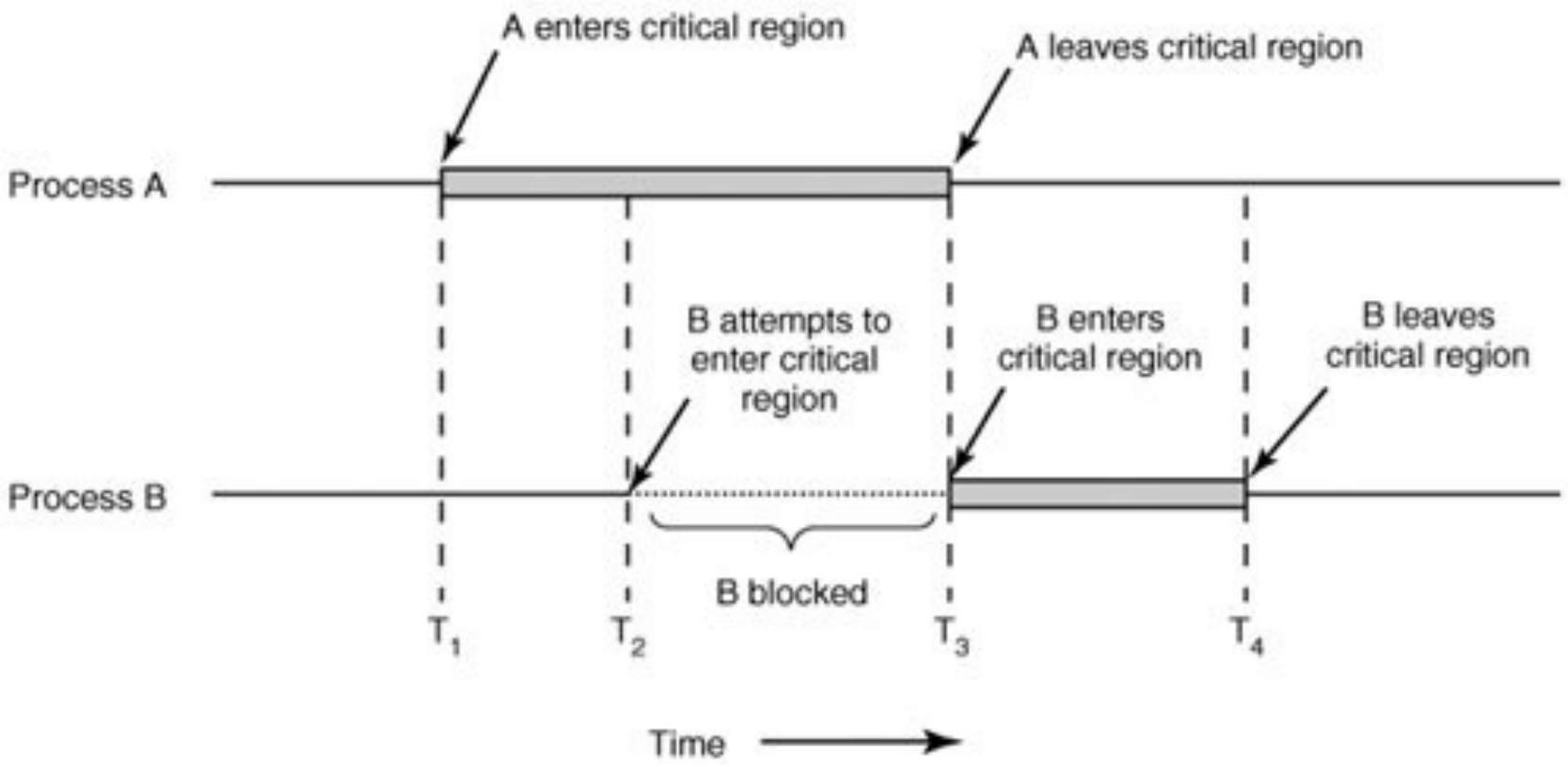
进程间通信

- 竞争条件
- 临界区
- 忙等待形式互斥
- 睡眠和唤醒
- 信号量
- 互斥
- 管程
- 消息传递

临界区

- 对共享内存进行访问的程序片段称作临界区(critical region)，或临界段(critical section)
- 避免竞争条件，一种好的解决方案需具备以下条件：
  - 任何两个进程不能同时处于临界区
  - 不应对CPU的速度和数目作任何假设
  - 临界区外的进程不得阻塞其他进程
  - 不得使进程在临界区外无休止地等待

例子


A enters critical region
A leaves critical region
Process A
B attempts to enter critical region
B enters critical region
B leaves critical region
Process B
B blocked
T1
T2
T3
T4
Time