Exception:

- interrupt 通常来自IO asyn.
- trap 系统调用 syn.
- fault syn.
- abort syn

exception 实际上是由核控制子进程的一种方式.

Status of a Process

- New
- Running
- Ready
- Stopped
- Blocked 等待IO
- Terminated

Process:

creating processes: fork 0 to child; child pid to parent
-1 on error

exec

reaping: pid_t waitpid(pid_t pid, int *status, int options)

pid: -1, 0, n, -n

pid 的值	含义
> 0	等待进程 ID 等于 pid 的子进程结束。
== 0	等待任何一个与当前进程在同一进程组的子进程结束。
< -1	等待进程组 ID 等于 -pid 的任何一个子进程结束。
== -1	等待任何一个子进程结束（等价于 wait() 的行为）。

status: WIFEXITED(status) & WEXITSTATUS(status)
WIFSIGNALED(status) & WTERMSIG(status)

option: 0: 直到有子进程终止

WNOHAN (no hang): 如果无子进程终止 则立刻返回0.

WUNTRACED: 挂起，一直等到有子进程终止或停止

WCONTINUED: 挂起，直到有子进程停止，或有已停止的进程重新开始.

WUNTRACED | WNOHAN: 立刻返回，如果有进程停止或中止，则返回PID；如果没有，返回0

```
pid_t wait(int* status)
    = waitpid(-1, &status, 0)
```

sleep & pause: sleep: 挂起一段时间

pause: 挂起直到接收到信号

竞争条件：原因：

- Shell一行执行多个命令，当父进程结束，就开始执行下一个命令，不管子进程是否执行完。
- 因此child的数据会与parent的随机交叉
- 先子后父，不会交叉

int execve (const char* filename, const char *argv[], const char* envp[])

Signal

发送信号：
- 硬件
- 内核
- shell命令
- 函数：kill. alarm . raise ...

接收信号：从内核态->用户态时

sighandler_t signal (int signum, sighandler_t handler)

阻塞和解除：
- sigprocmask
- sigemptyset
- sigfillset
- sigaddset
- sigdelset

信号处理：回收子进程 while (waitpid(-1, NULL, 0)>0)

为避免父子进程竞争，要屏蔽sigchld.

CPU调度：进程切换的决策

优化目标：

1. Turnaround Time 周转时间

$$T_{turnaround} = T_{completion} - T_{arrival}$$

- FIFO：缺：先到的任务特别长。
- SJF（最短优先）
  ↓
- STCF（时间片抢占）

只考虑周转时间会带来公平性问题

2. Response time 响应时间

$$T_{response} = T_{firstrun} - T_{arrival}$$

RR：时间片队列

3. 考虑IO

MLFQ（Multi-Level Feedback Queue）

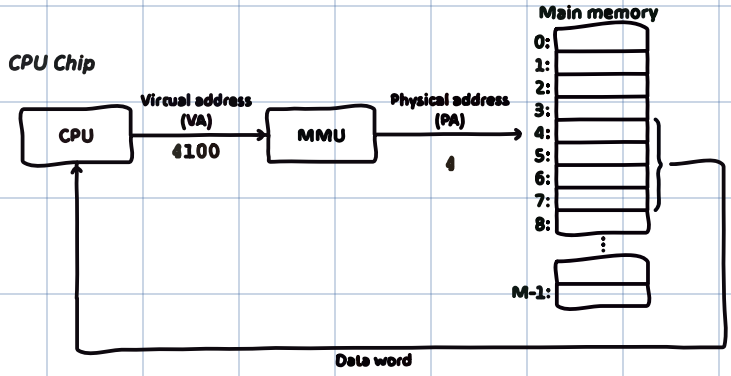
- Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
- Rule 2: If Priority(A) = Priority(B), A & B run in RR.
- Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).
- Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- Rule 5: After some time period S, move all the jobs in the system to the topmost queue.

随机调度算法：Lottery Scheduling

多核CPU调度

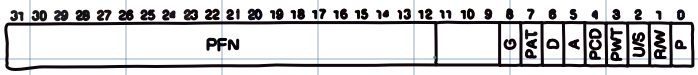
Virtual Memory

A System Using Virtual Addressing



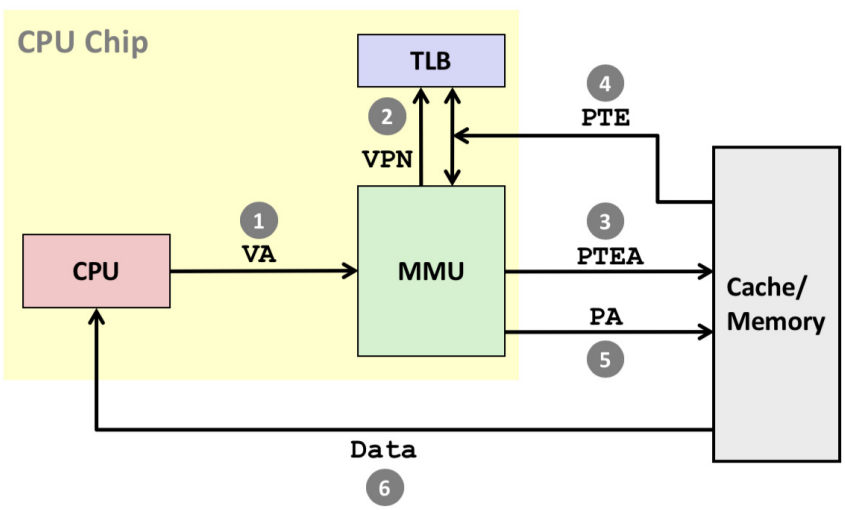
- Used in all modern servers, desktops, and laptops
- One of the great ideas in computer science

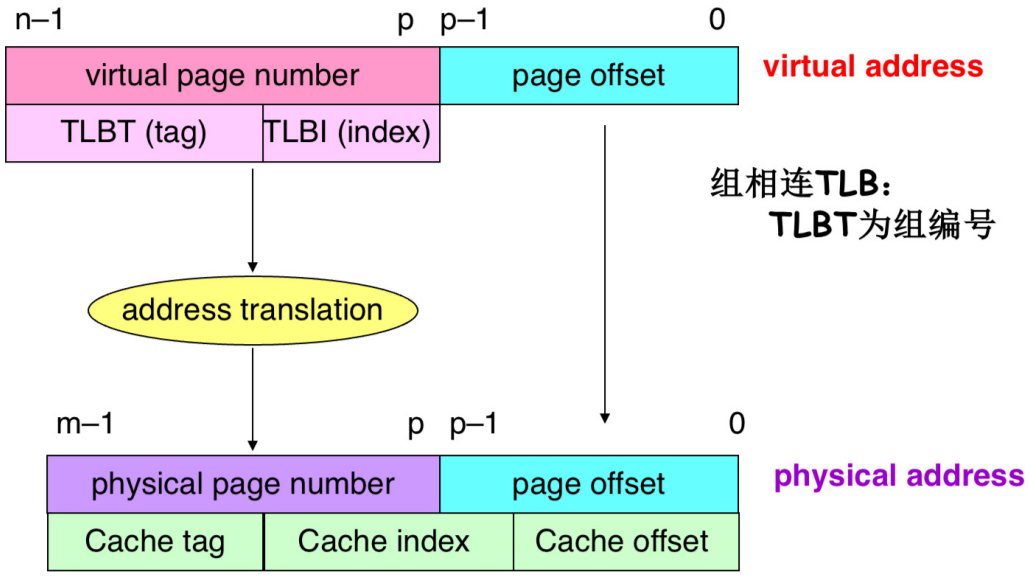
真实的PTE:



- valid bit (P)
- protection bits (R/W) (r/w/e权限)
- present bit (P) (on disk/ in physical memory)
- dirty bit (D) (与disk版本是否一致)
- reference bit/accessed bit (A) (体现page的流行度，与replacement有关)
- U/S: user/supervisor，用户态是否可以访问
- PAT/PWT/PCD/G: 与硬件cache相关

TLB:





组相连TLB：
TLBT为组编号

多级页表：

动态内存分配

碎片：
- 内部碎片
- 外部碎片

Keeping Track of Free Blocks

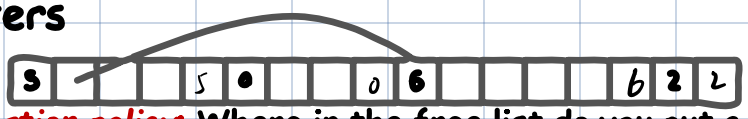
Method 1. *Implicit list* using length—links all blocks

(可加footer，方便合并)

eg：由于双字对齐，header的最后三位可作标志位。

Find a free block: First fit, Next fit, Best fit, Worst fit.

Method 2: *Explicit list* among the free blocks using pointers



Insertion policy: Where in the free list do you put a newly freed block?

LIFO (last-in-first-out) policy
- Insert freed block at the beginning of the free list
- *Pro:* simple and constant time
- *Con:* studies suggest fragmentation is worse than address ordered

Address-ordered policy
- Insert freed blocks so that free list blocks are always in address order:
  addr(prev) < addr(curr) < addr(next)
- *Con:* requires search
- *Pro:* studies suggest fragmentation is lower than LIFO

Method 3: *Segregated free list*

- Different free lists for different size classes

· #1. 简单分离存储（Simple Segregated Storage）
每个链表中装大小相同的块，不分割，不合并。

· #2. 分离适配（Segregated Fit）
每个链表中是一个空间范围的free blocks

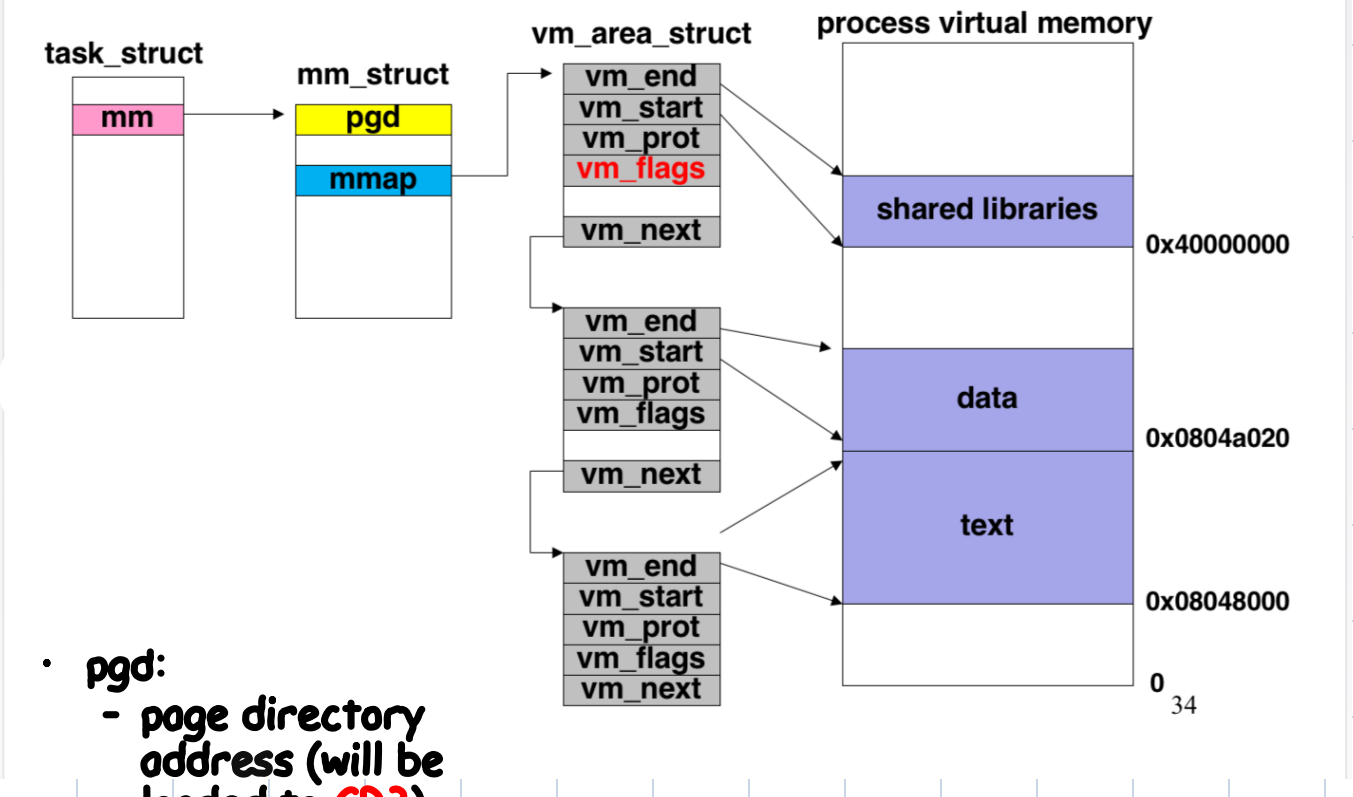
· #3. 伙伴系统（Buddy System）
每个表的大小都是2的幂

Method 4: **Blocks sorted by size**

- Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block and the length used as a key

垃圾回收：递归遍历指针，找到所有能到达的块并标记。

内存映射：vm_area_struct：



- pgd:
  - page directory address (will be loaded to CR3)
- vm_prot:
  - read/write permissions for this area
- vm_flags
  - shared with other processes or private to this process

mmap 函数

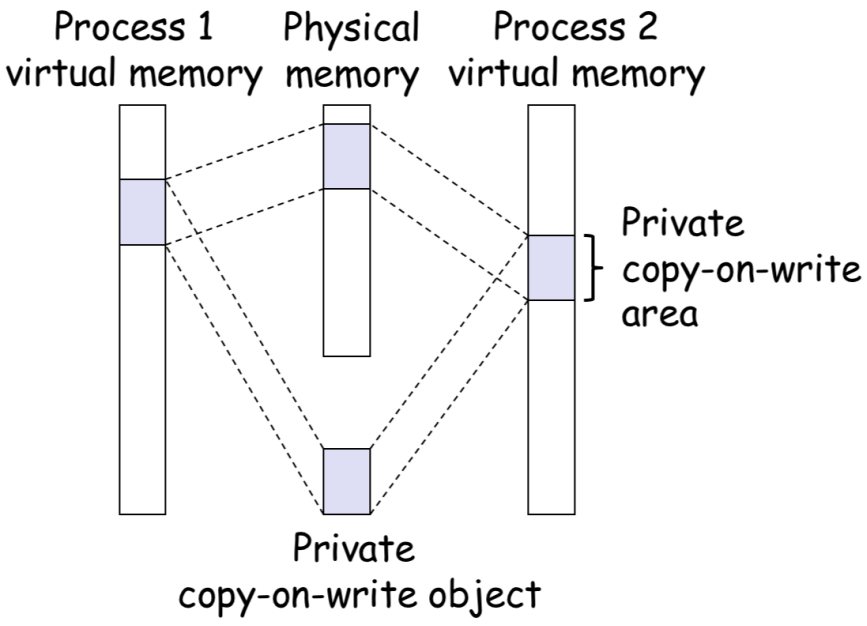
```
void *mmap(void *start, int len, int prot,
           int flags, int fd, int offset)
```

Map ***len*** bytes starting at offset ***offset*** of the file specified by file description ***fd***, preferably at address ***start*** (usually 0 for don't care).

- **prot**: **PROT_READ, PROT_WRITE**
- **flags**: **MAP_PRIVATE, MAP_SHARED**

Return a pointer to the mapped area

cow: copy on write



I/O

Devices：磁盘　　寻道时间　等待时间　读取时间

调度算法
- F-scan　由→外　外→内
- C-scan　只有由→外 读
- SSTF (Shortest Seeking Time First)

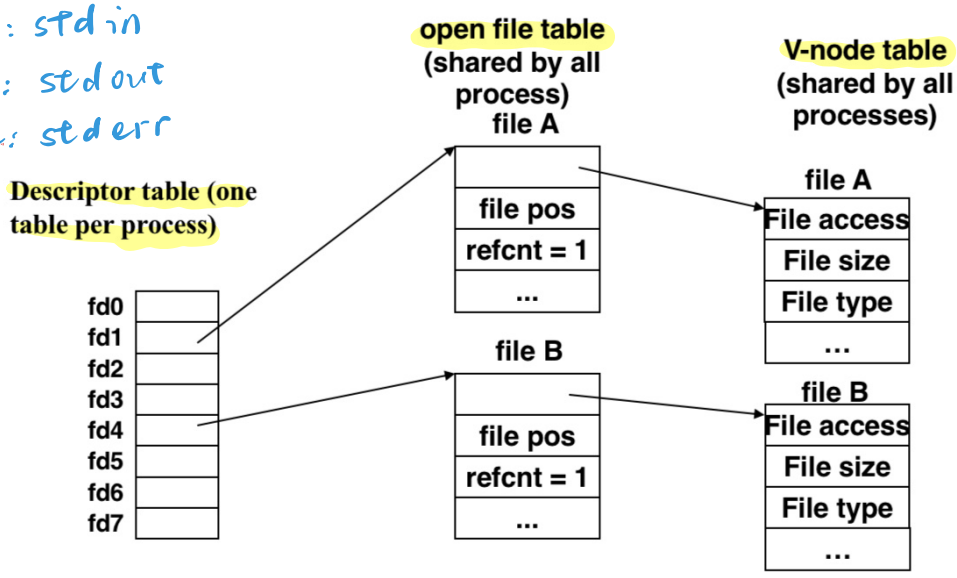
磁盘
- RAID（独立磁盘冗余阵列）
  - RAID 0　无备份.
  - RAID 1　镜像备份.　?个一组是否固定.
  - RAID 5　旋转奇偶校验
  - JBOD　?
- SSD　　映射策略
  - page - level
  - block - level
  - hybrid：log table + data table.
    - partial merge
    - full merge

此类型题目不统一，需注意 merge 的要求.

Unix I/O

Descriptor table → open file table → V-node table

0: stdin
1: stdout
2: stderr



接口函数

```
int open(char *filename, int flags, mode_t mode);
      Returns: new file descriptor if OK, -1 on error

flags
    - O_RDONLY, O_WRONLY, O_RDWR (must have one)
    - O_CREAT, O_TRUNC, O_APPEND, O_DIRECT (optional)
mode
    - S_IRUSR, S_IWUSR, S_IXUSR
    - S_IRGRP, S_IWGRP, S_IXGRP
    - S_IROTH, S_IWOTH, S_IXOTH
```

```
int close(int fd) ;
          Returns: zero if OK, -1 on error
```

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
                  returns: number of bytes read if OK,
                                  0 on EOF, -1 on error

ssize_t write(int fd, const void *buf, size_t count);
                returns: number of bytes written if OK,
                                        -1 on error
```

读取文件的元数据.

```
#include <unistd.h>
#include <sys/stat.h>
int stat(const char *filename, struct stat *buf);
int fstat(int fd, struct stat *buf) ;
                                returns: 0 if OK, -1 on error
```

硬链接 不同文件名但相同inode.

软链接 不同文件名,数据块中存的是指向对应文件的路径字符串.

```
#include <sys/types.h>
#include <dirent.h>
```

空目录实际上包含一个引用自身和一个引用父目录的条目.

```
DIR *opendir (const char *name);
        returns: 成功则返回处理的指针；若出错，则返回NULL
struct dirent *readdir (DIR *dirp);
returns: 成功则返回指向下一个目录项的指针；没有更多目录或出错则返回NULL
DIR *closedir (DIR *dirp);
                returns: 成功则返回0；若出错，则返回-1
```

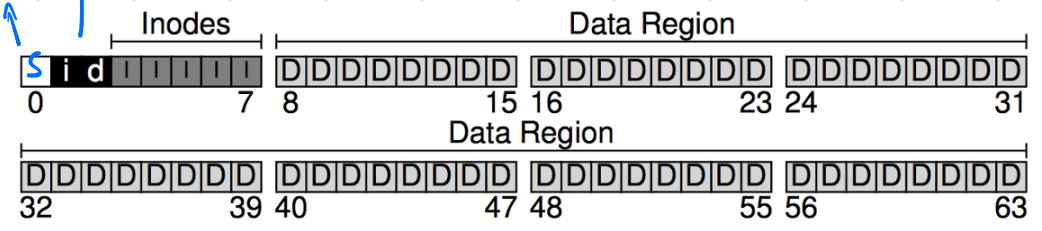
重定向 修改的是newfd

```
int dup2(int oldfd, int newfd);
        returns: nonnegative descriptor if OK,
                                        -1 on error
```

File System

bitmap.

super block ↑



Multi-level index 用大量空间索引数据块

Extent point + length.

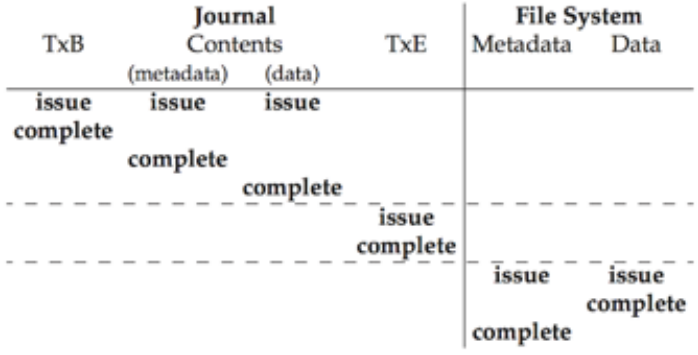
Linked-based inode中只包含一个指针

解决崩溃一致性问题的方法：

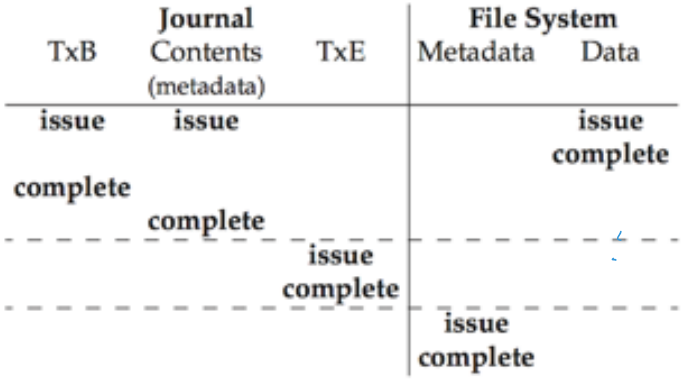
FSCK 文件系统检查程序

日志
- data journal
- meta data journal

- 总结时间线：data journaling

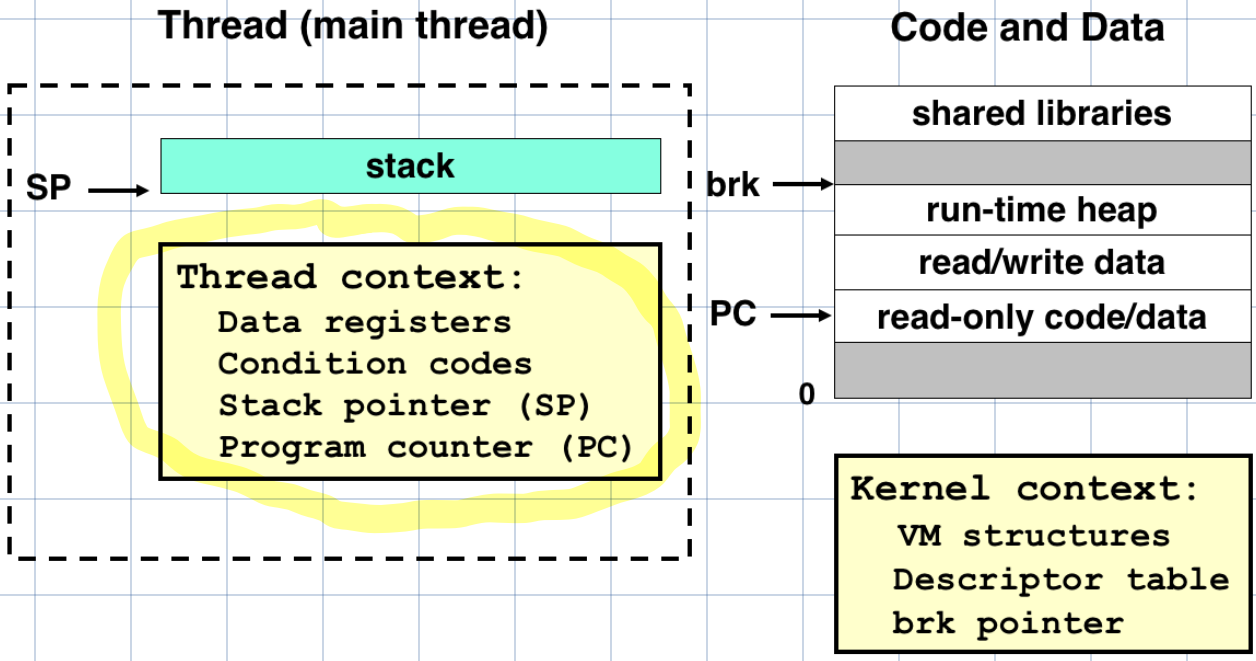


- 总结时间线：metadata journaling



并行编程：

线程的基本概念：进程 - code . data . heap . kernel context



Posix threads (Pthreads) interface

- Creating and reaping threads
  - **pthread_create**
  - **pthread_join**
- Determining your thread ID
  - **pthread_self**
- Terminating threads
  - **pthread_cancel** 发送中止请求给某线程
  - **pthread_exit** 线程主动退出
  - **exit** [terminates all threads] 会导致所有进程退出
  - **return** [terminates current thread]

eg：不join必须用线程的detached模式：

pthread_detach(tID)

信号量：

```
#include <semaphore.h>

int sem_init(sem_t *sem, 0, unsigned int value);
int sem_wait(sem_t *s);  /* P(s) */
int sem_post(sem_t *s);  /* V(s) */

#include "csapp.h"

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_wait */
```

再进程图来分析死锁.

并行编程模型

生产-消费者 核心数据结构:

```
struct {
  int *buf;      /* Buffer array */
  int n;         /* Maximum number of slots */
  int front;     /* buf[(front+1)%n] is first item */
  int rear;      /* buf[rear%n] is last item */
  sem_t mutex;   /* protects accesses to buf */
  sem_t slots;   /* Counts available slots */
  sem_t items;   /* Counts available items */
} sbuf_;
```

读写者问题

- 读优先
- 读写公平
- 写优先

线程安全 可重入函数:不涉及任何共享变量.

Race,当程序依赖于多个线程的执行顺序时.

缓存统一.

并行数据结构

Sloppy counter：每个cpu一个lock，还有个global lock.
本地累加，定期汇总.

链表：一个global lock，对insert和look_up上锁

队列：head_lock. tail_lock 分别对头尾上锁.

哈希表：每个桶1个锁 一个桶就是一个链表.

复用

无锁化：CAS.

锁的实现：

Test and set

Compare and swap.

Load-Linked and store-Conditional

eg：基本都是依赖一些原子操作