

进程&线程

[零 参考资料 3](#_Toc49978515)

[一 进程(Process) 3](#_Toc49978516)

[1 概述 3](#_Toc49978517)

[2 进程调度(Process Scheduling) 4](#_Toc49978518)

[2.1 概述 4](#_Toc49978519)

[2.2 Schedulers 5](#_Toc49978520)

[3 Operations on Processes 6](#_Toc49978521)

[3.1 进程创建(Process creation) 6](#_Toc49978522)

[3.2 进程终止(Process termination) (UNIX/LINUX) 7](#_Toc49978523)

[4 进程间通信(Interprocess Communication) 7](#_Toc49978524)

[4.1 概述 7](#_Toc49978525)

[4.2 Shared-memory Systems 8](#_Toc49978526)

[4.3 Message-Passing Systems 8](#_Toc49978527)

[二 线程(Thread) 9](#_Toc49978528)

[1 概述 9](#_Toc49978529)

[2 多线程模型(Multithreading Models) 9](#_Toc49978530)

[3 多线程问题(Threading Issues) 9](#_Toc49978531)

[三 CPU调度(CPU Scheduling) 9](#_Toc49978532)

[1 概述 9](#_Toc49978533)

[2 调度算法(Scheduling Algorithms) 9](#_Toc49978534)

[3 多处理器调度(Multiple-Processor Scheduling) 9](#_Toc49978535)

[4 实时系统CPU调度(Real-Time CPU Scheduling) 9](#_Toc49978536)

[5 算法评估(Algorithm Evaluation) 9](#_Toc49978537)

[四 进程同步(Process synchronization) 9](#_Toc49978538)

[1 概述 9](#_Toc49978539)

[2 临界区(critical section) 9](#_Toc49978540)

[2.1 概述 9](#_Toc49978541)

[2.2 Peterson 9](#_Toc49978542)

[2.3 硬件同步(synchronization hardware) 9](#_Toc49978543)

[2.4 互斥锁(Mutex Locks) 9](#_Toc49978544)

[2.5 信号量(Semaphores) 9](#_Toc49978545)

[3 同步问题(Classic Problems of Synchronization) 9](#_Toc49978546)

[3.1 有界缓冲区(The Bounded-Buffer Problem) 9](#_Toc49978547)

[3.2 读-写(The Readers –Writers Problem) 9](#_Toc49978548)

[3.3 哲学家就餐(The Dining-Philosophers Problem) 9](#_Toc49978549)

[4 管程(Monitors) 9](#_Toc49978550)

[5 替代同步(Alternative Approaches) 9](#_Toc49978551)

[五 死锁(Deadlock) 9](#_Toc49978552)

[1 概述 9](#_Toc49978553)

[2 预防(Deadlock Prevention) 9](#_Toc49978554)

[3 死锁避免算法(Deadlock Avoidance) 9](#_Toc49978555)

[3.1 安全状态(Safe State) 10](#_Toc49978556)

[3.2 资源分配图算法(Resource-Allocation-Graph Algorithm) 10](#_Toc49978557)

[3.3 银行家算法(Banker's Algorithm) 10](#_Toc49978558)

[4 死锁检测(Deadlock Detection) 10](#_Toc49978559)

[5 从死锁中恢复(Recovery from Deadlock) 10](#_Toc49978560)

[六 LINUX 进程调度 10](#_Toc49978561)

[七 LINUX 进程地址空间 10](#_Toc49978562)

[八 LINUX 进程通信 10](#_Toc49978563)

[九 JAVA Thread 10](#_Toc49978564)

[1 概述 10](#_Toc49978565)

[1.1 原语 10](#_Toc49978566)

[1.2 内存模型 10](#_Toc49978567)

[1.3 原理 10](#_Toc49978568)

[2 Thread 10](#_Toc49978569)

[3 Thread Pool 10](#_Toc49978570)

[3.1 概述 10](#_Toc49978571)

[3.2 API 10](#_Toc49978572)

[4 ThreadLocal 10](#_Toc49978573)

[5 Future 10](#_Toc49978574)

[6 Fork 11](#_Toc49978575)

[7 ReentrantLock 11](#_Toc49978576)

[8 其他线程工具 11](#_Toc49978577)

[8.1 CountDownLatch 11](#_Toc49978578)

[8.2 CyclicBarrier 11](#_Toc49978579)

# 零 参考资料

《操作系统概念》International Student Version第九版

《深入理解LINUX内核》第三版

《深入Linux内核架构》

《Linux内核设计与实现》

《垃圾回收算法手册》

《深入理解计算机系统》

Oracle java 官方文档：

<https://docs.oracle.com/javase/8/docs/api/index.html>

<https://docs.oracle.com/en/java/javase/14/docs/api/index.html>

# 进程(Process)

## 概述

<https://en.wikipedia.org/wiki/Process_management_(computing)>

* + - 1. process是active entity，具有一个指定下个执行指令的program counter和一组相关资源，不只是包括程序代码：
         1. 还包括当前activity,比如program counter值、处理器的registers的内容等
         2. 通常还包括

包含function parameters、return address、local variables等临时数据的process stack、

包含global variables的data section

* + - * 1. 可能还包括在process 运行时动态分配的heap
        2. process 本身也可以作为一个用于执行其他代码的环境，比如JVM
        3. 任何时候一个processor只有一个process处于running状态(It is important to realize

that only one process can be running on any processor at any instant)，其他进程处于ready或waiting状态



* + - 1. 程序本身不是process，其只是 passive entity，
         1. 一个可执行文件被加载到memory时，对应的程序就成为了一个process
         2. 即使两个process与同一个程序关联，但仍当两个独立execution sequences
      2. process state 在执行process时发生改变，部分取决于process当前activity
         1. 每个系统的process至少有五个状态(以下状态名在各个系统中可能有差异)：

New：process被创建

Running：程序指令正在执行

Waiting： process正等待某个事件(比如I/O完成或收到信号)

Ready: process正等待被关联到某个processor

Terminated：process已经完成执行



* + - 1. Process Control Block(PCB)也称为task control block，在OS中每个process都用PCB表示，本身包含了特定process相关信息，包括但不限于：
         1. process state: 状态查看第c点
         2. program counter: 指定当前进程下个执行指令的地址
         3. CPU register: 根据计算机架构不同，CPU register的类型和数量也会不同

通常包括accumulators、index register、 stack pointer、general-purpose registers、其他一些condition-code information

CPU-scheduling information: 包括 process priority、pointers to scheduling queues、其他scheduling parameters

memory-management information： 根据OS的memory system，这部分信息可能包括s the value of the base and limit registers、page tables、segment table

accounting information：包括 amount of CPU and real time used, time limits, account numbers, job/process numbers

I/O status information：包括分配到的I/O devices列表、list of open files



## 进程调度(Process Scheduling)

### 概述

* + - 1. Process scheduler将一个有效process放到CPU上
      2. job queue，当process加入到系统时，都会被加入到这个queue，这个queue包含了系统内所有的process
      3. ready queue：
         1. 驻留在memory中、状态为ready或waiting的process，会被加入到这个queue
         2. 通常使用链表实现

头部包含指向链表的第一个PCB和指向最后个PCB的两个指针

每个PCB都包含指向下个PCB的pointer field

* + - 1. device queue，等待特定I/O device的process 都会被加入到这个队列，每个device 都有自己的device queue
      2. process scheduling 通常用queueing diagram展示



* + - 1. Context switch: 切换CPU到另一个process需要保存当前process状态(包括kernel和user mode)然后恢复另外一个process的状态的任务过程
         1. process context 保存在PCB中，通常包括CPU register的值、process状态、memory-management信息
         2. Context switch时间是纯粹的开销，期间系统不做任何相关有用工作

### Schedulers

<https://en.wikipedia.org/wiki/CPU-bound>

<https://en.wiktionary.org/wiki/I/O_bound>

* + - 1. process被适当的scheduler选中并执行
      2. long-term scheduler或者job scheduler 从mass-storage device中选择脱机保存到此处的process(These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution)，加载到memory执行
         1. 控制着degree of multiprogramming，即memory 中process数量
         2. 可能只有当process离开系统时，才需要long-term scheduler，因为保证degree of multiprogramming 稳定, 创建process的平均创建速率必须等于process离开系统的平均departure rate
         3. 选择process是十分仔细地
         4. 如UNIX和Windows等time-sharing system 通常没long-term scheduler，只是简单地将新process放到memory给short-term scheduler选择
      3. short-term scheduler或者CPU scheduler从ready状态的processes中选择一个process，并分配CPU资源给它
         1. 通常每100ms必须至少执行一个次，本身执行必须要快
      4. process 通常可以分为I/O-bound process 和 CPU-bound process
         1. I/O-bound，执行I/O比执行计算需要更多时间,如果大部分process都是I/O-bound,ready queue 几乎为空
         2. CPU-bound, 执行计算比执行I/O需要更多时间，如果大部分process都是CPU-bound，I/O waiting queue几乎为空，从而设备几乎空置
      5. medium-term scheduler
         1. 部分系统会有，如time-sharing system，
         2. 主要实现方案是,swapping：

将process从memory或CPU 竞争中移出(swap out)，降低the degree of multiprogramming

之后process可能被重新调入内存(swap in)，从之前中断的地方继续执行



## Operations on Processes

### 进程创建(Process creation)

* + - 1. process tree：每个新process(parent process)可以再创建其他process(child process),从而形成process tree
      2. process identifier(PID)：大多数OS（如UNIX、Linux、Win）识别process的唯一标识，
         1. 通常是个整数数字
         2. 每个process都有唯一pid，作为访问kernel中process各种属性的索引
      3. 当process创建一个child process
         1. child process 可以从OS那里直接获得资源
         2. child process 也可以只从parent process获得资源子集，parent process可以划分资源给child，或者跟child共享资源
         3. 将child process限制只能访问parent的资源，可以防止创建过多process，导致系统overloading
         4. 有两种执行方式：

parent 和child 并发执行

parent 进入wait状态，直到部分或全部child执行完毕

### 进程终止(Process termination) (UNIX/LINUX)

* + - 1. 通常发生在当process执行完最后语句并通过调用exit()请求系统删除自己时，自身资源由OS释放
      2. process可以通过相应的system call终止另一个process
         1. 通常只有parent才能通过system call终止child process
         2. parent需要直到child的pid，才能终止child，因此创建child process要将child 的pid传给parent
      3. Process 终止时可以返回状态值(通常是整数)给parent
      4. 当child process终止时，资源由OS释放，
         1. 但在parent调用wait()之前， 其entry仍在process table中
         2. wait()使parent等待child终止，并让parent获得child的pid和exit status，使parent感知到哪个child终止了
      5. Cascading termination:在不允许child process在parent process已经terminate时继续存在的系统中，如果process 终止(正常OR异常)，其所有child也必须终止
      6. Zombie process：指的是本身已经终止，但parent process还没调用wait()的process
      7. 当parent process没调用wait()就终止，会导致child成为孤儿进程(orphan process)，Linux/UNIX的init process便成为orphan process的parent，会定期执行wait(),收集orphan的exit状态，并释放orphan的pid和process-table entry

## 进程间通信(Interprocess Communication)

### 概述

* + - 1. 独立进程(Independent process): 指不影响其他process，也不被其他process影响的process
      2. 协作进程(Cooperating process): 指能影响其他process或受其他process影响的process
      3. Inteprocess Communication(IPC): 允许process之间交互数据和信息的一种机制，常见的有两种模型：
         1. shared memory：建立一块共享的memory区域，给cooperationg process交互数据和信息。仅在建立共享区域时需要system call，后续的数据共享存储，不需要内核参与。在多核系统中，存在cache一致性问题
         2. 消息传递(message passing)：通信通过cooperationg process之间交互信息进行；因为无需避免冲突，很适合少量数据交互。对于分布式系统，更易于实现。但因为消息传递系统通常通过system calls实现，因此速度比shared memory模型慢



### Shared-memory Systems



### Message-Passing Systems

# 线程(Thread)

## 概述

## 多线程模型(Multithreading Models)

## 多线程问题(Threading Issues)

# CPU调度(CPU Scheduling)

## 概述

## 调度算法(Scheduling Algorithms)

## 多处理器调度(Multiple-Processor Scheduling)

## 实时系统CPU调度(Real-Time CPU Scheduling)

## 算法评估(Algorithm Evaluation)

# 进程同步(Process synchronization)

## 概述

## 临界区(critical section)

### 概述

### Peterson

### 硬件同步(synchronization hardware)

### 互斥锁(Mutex Locks)

### 信号量(Semaphores)

## 同步问题(Classic Problems of Synchronization)

### 有界缓冲区(The Bounded-Buffer Problem)

### 读-写(The Readers –Writers Problem)

### 哲学家就餐(The Dining-Philosophers Problem)

## 管程(Monitors)

## 替代同步(Alternative Approaches)

# 并发

## 概述

## 死锁(Deadlock)

## 预防死锁(Deadlock Prevention)

## 死锁避免算法(Deadlock Avoidance)

### 安全状态(Safe State)

### 资源分配图算法(Resource-Allocation-Graph Algorithm)

### 银行家算法(Banker's Algorithm)

## 死锁检测(Deadlock Detection)

## 从死锁中恢复(Recovery from Deadlock)

# LINUX 进程调度

# LINUX 进程地址空间

# LINUX 进程通信

# JAVA Thread

## 概述

### 内存模型

### 原理

## Thread

<https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html>

<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/Thread.html>

## Thread Pool

<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/concurrent/ThreadPoolExecutor.html>

<https://docs.oracle.com/javase/8/docs/api/index.html>

### 概述

### API

## ThreadLocal

<https://docs.oracle.com/javase/8/docs/api/java/lang/ThreadLocal.html>

<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/ThreadLocal.html>

## Future

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/FutureTask.html>

<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/concurrent/FutureTask.html>

## Fork

<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/concurrent/ForkJoinTask.html>

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html>

## ReentrantLock

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html>

<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/concurrent/locks/ReentrantLock.html>

## 其他线程工具

### CountDownLatch

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CopyOnWriteArraySet.html>

<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/concurrent/CountDownLatch.html>

### CyclicBarrier

<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/concurrent/CyclicBarrier.html>

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html>