程序设计语言

Programming Languages

裘宗燕

数学学院信息科学系

2012年2-6月

第9章 并发性

- 为什么需要并发性?
- 简单历史
- 并发程序的基本问题
- 并发程序设计基础
- 进程与线程
- 共享存储器模型
- 消息传递模型
- 进程间通讯

为什么（需要）并发

顺序程序：在运行中只有一个活动的执行上下文

顺序执行是命令式程序的基本特性，也出现在说明性程序的执行中

并发程序运行中可能存在多个同时活动的执行上下文，多个“控制线程”

三方面重要原因：

- 为更好反映问题的逻辑结构。许多程序里，特别是各种服务器和图形应用，需要维护一批基本上相互独立的“作业”的轨迹。构造这种程序时，最简单最合逻辑的方式是用一个独立的控制线程表示一个作业
- 需要应对多台独立设备。OS 可能在任何时候被中断，为此需要用一个上下文表示中断前的工作，用另一上下文表示中断本身。实时控制系统常需要处理多台独立的外部设备（它们各有独立的控制线程），还需要与其他处理器上运行的线程交互，以实现系统的整体行为
- 通过并发可能提高系统性能。有些程序本质上可能不需要并行，但使用多个处理器同时工作可能提高系统性能，得到很大速度提升

并发性

并发性不是新想法

- 有关并发程序设计的大部分最基础的理论工作从 20 世纪 60 年代开始提出和研究，70 年代其基础理论框架已经基本完成
- Algol 68已包含了有关并发性的程序设计特征

- 对并发性的广泛兴趣是近年的新现象，原因：低价的多处理器系统，图形、多媒体和互联网应用的发展（用并发线程描述特别自然）等

并发性问题出现在许多不同层次上

- 数字逻辑层几乎所有事情都是并行的，信号在大量连线上同时传播
- 在现代处理器里，流水线和超标量特征就是为了利用指令级并行
- 专用向量处理器实现一种中层的数据并行；各种多处理器系统中存在很多并行运行的进程；在互联网上一切事情都并发地进行
- 本章集中关注中尺度和大尺度的并发性，这些并发性由程序员可见的结构的语义表示，多处理器的机器可以利用这种并发性

并发性：简单历史

- 早期计算机是单用户机器，一个用户工作时占用整个计算机硬件
- 单用户模式成本太高。计算机系统转到另一模式：用户在线下创建作业（输入源代码）；操作员维护一批作业，启动作业执行。每个程序最后把控制转回驻留的监控程序，它立即读入下一程序后令其执行
- 商务应用需要读入大量数据，计算相对简单。程序执行到 I/O 时处理器给设备发命令后进入忙等待，可能导致大量时间闲置
- 为利用忙等待的闲置周期，开发了中断驱动的 I/O 和多道程序设计技术（多个应用可同时驻留内存）。两项发明都需要硬件支持。前者要求实现中断，后者要求内存保护，保证一个程序的错误不破坏其他程序
- OS 维持多个程序的轨迹，知道哪些正在等 I/O 完成，哪些可运行
- 程序在运行中需要 I/O 时把控制传给 OS，OS 给设备发命令后立即把控制传给另一可运行程序。设备完成时发中断使控制回到 OS。OS 知道等待这个 IO 的程序又可运行了，并从可运行程序里选一个执行。这样，只有内存里的所有程序都在等 I/O 时处理器才会闲置

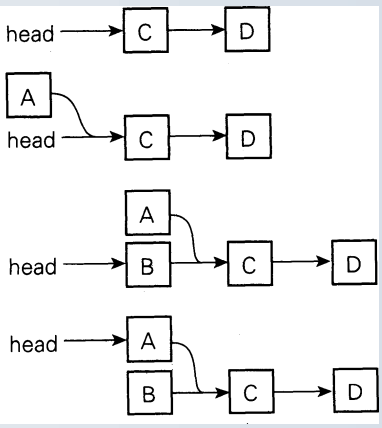
中断和竞态

- 中断驱动的 I/O 把并发性引进 OS：中断可能随时发生，包括控制已经位于 OS 里的情况，中断处理器和 OS 的主要功能部分是并发的
- 如果中断发生时 OS 正在修改某数据结构（例如可运行程序队列），中断处理器就可能看到数据结构处于不一致的状态

```
main program:
    read head
    A→next := head

        handler:
            read head
            B→next := head
            head := B

    head := A
```



竞态问题：并行线程在某些点上出现“竞争”，此时它们都要接触某些公共对象，系统行为依赖于哪个线程在前

为保证正确行为，必须对相关线程进行同步（设法控制各种动作发生的顺序）

并非所有竞态条件都是坏的，有时多种运行结果都可接受

同步的目标是消除“坏”竞态条件，消除可导致程序产生错误结果的情况

分时系统和分布式系统

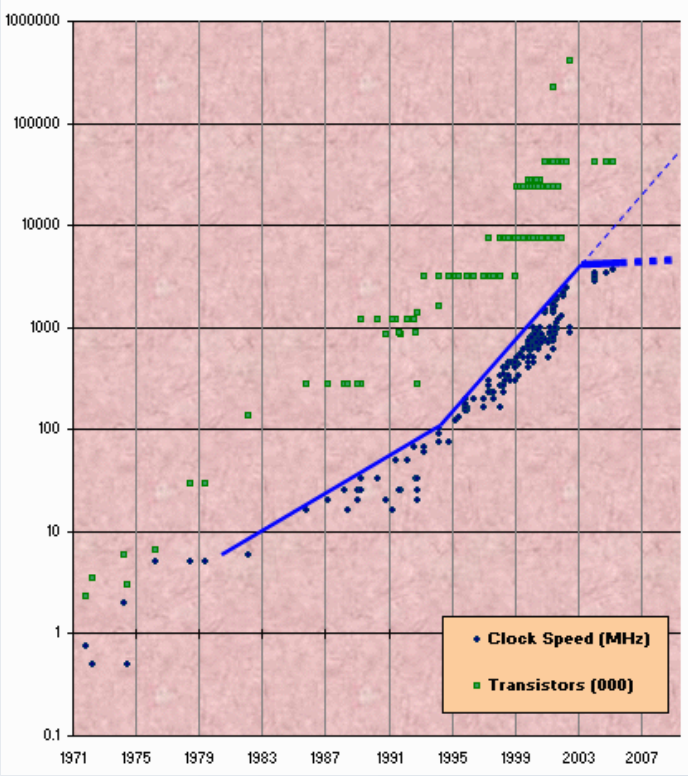
随着内存增大和虚存技术发展，系统已能允许任意多个程序同时运行

- 在批处理系统里，仅在一个程序因 I/O 而阻塞时才转到另一程序
- 强占式分时系统常规地每秒完成几次切换，防止计算量大的程序连续占据计算机太长时间，导致其他程序无法得到运行的机会

- 分时系统在 1970 年代早期使用很多。其中扩充了数据共享机制和其他支持线程间通信的机制，允许用户在应用层面进行并发编程
- 计算机网络以分布式系统的方式支持真正的并行，程序在物理上相互无关的不同机器里运行，程序之间通过消息的方式通信

  多数分布式系统通过并发来有效利用多台物理设备。也有少数系统里的分布就是为利用多处理器提供的加速性能
- 多处理器系统在 60 年代前后发明，80 年代之前不多见。近年这种系统开始出现在用户级桌面机器里。由于单处理器的性能提高遇到严重挑战，今后大部分桌面系统都将使用包含多个处理器的芯片

多处理器和并行



通过提高主频的方式提高计算机性能的时代已经结束了

大约 2003 年以来，常规的计算机主频增长已经停止

单个处理器芯片的性能提高完全依靠片上多处理器技术

目前流行的高端商品处理器上单片处理器数为 8-16 核，研究或作为原型的有几十核到数千核的多核/众核处理器

多核发展方兴未艾，对并行程序设计技术和语言提出许多新要求

并行性：个人计算机

在个人计算机的发展历史中，OS 的发展历史又重复了一遍

- 早期 PC 执行忙等待 I/O，同时只运行一个应用
- 随着微软 Windows 和 MacOS 的 Multifinder 的开发，PC 系统有了同时在内存里维护多个程序的能力，并能在它们与 I/O 之间切换
- PC 是单用户机器， 开始用户对强占的要求不迫切，可以接受一个程序长期霸占处理器的情况，因为它就是当时用户运行的程序

  随着 PC 功能日益强大，应用日益复杂，用户开始要求线程并发执行，如浏览器，“后台”线程更新窗口、检查电子邮件、照顾打印机等。如果程序都能在运行中适当位置自愿交出控制，就能满足后台计算需要

  Windows 3.1/MacOS v7 可以看到 “合作式多道程序设计”
- 程序不能常规地交出控制就无法保证合理响应时间。由于用户越来越不满意，Windows 95 把强占功能 加入32 位应用，Windows NT 和 MacOS X 把强占加入所有程序，并让程序在独立地址空间里运行。这样一个程序的错误就不会破坏另一个程序，也不会导致系统垮台

并行性：硬件

并行计算机硬件的情况：

- 单机箱系统可分为两类：处理器共享公共存储器（常称为**多处理器系统**），常放在一个机柜里，共享内存、磁盘、电源和一份操作系统拷贝；另一类系统的处理器只能通过消息相互通信
- 计算机**集群**：一批物理上完全独立的单处理器或小型多处理器安在一组机架中，通过高速系统级网络连接，作为整体管理（Google、Amazon或eBay，都用成百成千台处理器构成的集群 ）
- **多机系统**：采用消息传递的单机架机器。上世纪80年代在科学计算与数据库应用领域很流行，已基本上被集群和大型的多处理器取代
- **向量处理器**：提供一些特殊指令，能把同样操作应用于数组的每一元素。向量指令很容易流水线化，在科学计算程序里很有用

  向量机的思想也在微处理器里发挥作用，如奔腾的MMX扩充，目前广泛使用的图形处理器，GPU等
- 问题：处理器之间的通讯网络（拓扑结构，路由），共享和分布式存储器的有效利用（存储器一致性问题）等等

多线程程序的一些实例

一些应用的逻辑结构及其对并行的需求：

- 离散事件模拟：用并发线程表示真实世界的一批主动物体。事件在特定时间点自动发生。由基础实现确定哪个线程在何时运行
- 服务器端需要同时为许多客户服务，需要多线程支持
- 视频游戏：需要在连续更新屏幕图像的同时处理击键/鼠标/游戏棒动作。应该用并发线程处理输入，同时有一个或多个线程更新屏幕
- 基于万维网的应用对多线程的需求特别明显。例如浏览器：
  - 通常有多个活动线程与远程服务器通信。点击链接创建新线程，线程接收消息“包”并将其展示，需访问字型、拼装词序列、分割行等
  - 浏览中可能生成许多线程：图像、背景、表格常用独立线程处理，框架可能用多个线程。派生线程都可能与服务器通信，获取信息（如图像内容）。用户还可能访问菜单项，创建窗口，编辑书签，修改首选项等，这些都与页面绘制同时进行
  - 下面看一个并发实现的程序框架


```
procedure parse_page(address : url)
    contact server, request page contents
    parse_html_header
    while current_token in {"<p>", "<h1>", "<ul>", . . . ,
            "<background", "<image", "<table", "<frameset", . . . }
        case current_token of
            "<p>"     :  break_paragraph
            "<h1>"    :  format_heading; match("</h1>")
            "<ul>"    :  format_list; match("</ul>")
            . . .
            "<background" :
                        a : attributes := parse_attributes
                        fork render_background(a)
            "<image"  :  a : attributes := parse_attributes
                        fork render_image(a)
            "<table"  :  a : attributes := parse_attributes
                        scan forward for "</table>" token
                        token_stream s :=. . .     -- table contents
                        fork format_table(s, a)
            "<frameset" :
                        a : attributes := parse_attributes
                        parse_frame_list(a)
                        match("</frameset>")
        . . .
    . . .
procedure parse_frame_list(a1 : attributes)
    while current_token in {"<frame", "<frameset", "<noframes>"}
        case current_token of
            "<frame"  :  a2 : attributes := parse_attributes
                        fork format_frame(a1, a2)
        . . .
```

浏览器的线程代码框架：

parse_page 是处理**HTML**页面的分析器的根过程

对某些结构（背景、图像、表格和框架等）处理与页面本身的语法分析并发进行

并发线程用**fork**操作创建

需要响应键盘或鼠标动作时也可能创建其他线程

多线程可以保证快操作（如正文显示）不必等慢操作（如显示大图像）。如果某线程**阻塞**（等消息或**I/O**），就自动切换到其他线程

基于强占式线程包，实现还可以经常做线程间切换，保证任何线程都不会霸占处理器

指派循环

没有线程支持时也可用顺序结构实现浏览器，通常用一个**指派循环**把所有可能引起延迟的事件集中到这里处理：

- 用数据结构保存未完成**作业**的轨迹，作业状态可能很复杂
- 高层页面绘制作业的状态需记录已收数据包情况，标识各种子作业（图像、表格、框架等），以便遇到用户点击“停止”按钮时处理它们的状态
- 为实现较好响应效果，必须保证每个连续进行的子动作（ **continue_task** 的子动作，见下页）执行时间很短
  - 等消息时应暂停当前动作，读文件时也暂停当前动作（磁盘慢）
  - 如果某作业所需的时间长于 **1/10** 秒（人可观察的典型阈值），就必须分段，在片段执行之间保存状态并回到指派循环开始。还需保证循环的条件覆盖了所有可能的同步事件
  - 循环条件的求值必须与所有由于计算时间长而分段的作业交错执行。（实际中可能需要比交错更复杂的机制，以保证输入驱动的作业或计算密集的作业都不会长期占用共享资源）


```
type task_descriptor = record
    -- fields in lieu of thread-local variables, plus control-flow information
    ...
ready_tasks : queue of task_descriptor
...
procedure dispatch
    loop
        -- try to do something input-driven
        if a new event E (message, keystroke, etc.) is available
            if an existing task T is waiting for E
                continue_task(T, E)
            else if E can be handled quickly, do so
            else
                allocate and initialize new task T
                continue_task(T, E)
        -- now do something compute bound
            if ready_tasks is nonempty
                continue_task(dequeue(ready_tasks), 'ok')

procedure continue_task(T : task, E : event)
    if T is rendering an image
        and E is a message containing the next block of data
            continue_image_render(T, E)
    else if T is formatting a page
        and E is a message containing the next block of data
            continue_page_parse(T, E)
    else if T is formatting a page
        and E is 'ok'       -- we're compute bound
            continue_page_parse(T, E)
    else if T is reading the bookmarks file
        and E is an I/O completion event
            continue_goto_page(T, E)
    else if T is formatting a frame
        and E is a push of the "stop" button
            deallocate T and all tasks dependent upon it
    else if E is the "edit preferences" menu item
        edit_preferences(T, E)
    else if T is already editing preferences
        and E is a newly typed keystroke
            edit_preferences(T, E)
    ...
```

非线程浏览器的指派循环：

continue_task 的各个子句必须覆盖作业状态和触发事件的所有可能组合

主要问题是破坏了程序的算法结构：

- 执行时间长的操作需要不时返回循环开始，作业（接收图像、绘制图像、处理嵌套的菜单等）都不能用标准的控制流结构描述
- 程序内部的作业管理变成显式的，而作业的内在控制流变成隐含的

基于并发线程包实现，可以将程序转回正确的形式

- 使作业内部控制流变成显式的
- 作业（线程）管理变成隐式的，由线程管理器自动完成

并发程序设计基础

并发（**concurrency**）和并行（**parallelism**）的概念（书上的说明）

- 并发性指一大类程序的一组特性，在这类程序里，两个或更多的执行上下文同时处于活动状态。并行指并发程序里的一种特性，在其中的多个上下文里正在同时实际处于执行中
- 真正的并行性需要并行硬件。从语义观点看，在并行与强占式并发系统里（系统会在不可预期的时刻在不同执行环境之间切换）的“准并行”之间没有根本区别，两种情况需要同样的程序设计技术

另一种理解（我的理解）：

- 并行强调的是多个执行活动同时处于运行状态之中，强调其相互独立性。这里关心并行算法、并行系统、并行体系结构等等
- 并发强调的是多个执行活动之间的关系和相互作用，这里关心的是互斥、同步、通讯、资源的共享和竞争等等

基本概念：线程和作业

下面把并发程序的正在活动的执行上下文称为线程（动态概念）

- 操作系统设计中通常区分重量级进程（有独立地址空间）和轻量级进程（可能共享地址空间）。后者在 **20** 世纪 **80-90** 年代为迎合共享内存多处理器系统的快速发展而加入**Unix**的各种版本
- 特定程序的一组线程在操作系统提供的一个或多个进程上实现
- 如果没有轻量级进程，并发程序里的线程就必须在多个重量级线程上运行，语言实现必须保证线程的共享数据能映射到所有进程的地址空间

作业指良好定义的、必须由线程执行的程序工作单元（是静态概念）

- 常见情况是一组进程共享一个公共“作业袋”，即一组需要完成的作业
- 各进程反复地从袋中取出作业并执行之（每个执行形成一个作业线程），完成后去再取一个。作业线程工作中可能把新作业加入袋中

讨论并发程序的术语比较混乱，要注意具体文献里的说法。

基本概念：通讯和同步

并发程序设计模型里，需要处理的最关键问题就是通信和同步

- 通信指线程可以用于获得其他线程产生的信息的（各种）机制。命令式程序的通信机制通常基于*共享存储器或消息传递*
  - 在共享存储器模型里，存在一些可以被多个线程访问的变量（不同线程之间有共享的存储区，共享变量）。一个线程把值写入某变量，另一线程去读该变量，也就实现了通讯
  - 在消息传递模型里线程没有公共状态。如果一对线程要通信，必须有一个线程明确执行一次 **send** 操作，把消息传送给另一个线程
- 同步指可用于控制不同线程间操作发生的相对顺序的各种机制
  - 消息传递模型里的同步是隐式的，接收必须在发送之后。要求接收消息的线程必须等到发送方实际发送之后才能完成动作
  - 在共享存储器模型里同步通常不是隐式的，除非明确说明做某些事，否则“接收方”就可能在某变量被“发送方”修改前读其原来的值

基本概念：忙等待同步和阻塞同步

共享存储器模型和消息传递模型都需要同步。两种主要实现方法：

- 自旋（忙等待）：等待同步的线程继续占据处理器，不断检查某个同步条件是否成立（例如某队列是否为空，某个变量是否大于 **0**）
- 自旋等待实际上假定了存在其他线程，它们可能来修改相关状态，从而改变本线程所关注或等待的条件

  如果没有强占，在单一处理器上忙等待没有意义（因为不可能有其他活动的事物来改变线程的被等待状态）
- 阻塞式同步（基于调度器的同步）：

  需要等待某个同步条件的线程它需要准备等待的条件上登记，然后把它当前占用的处理器上交给调度器

  其他线程的活动可能使这一线程等待条件变为成立的，这时某些内部机制可找出正等待的线程（集合），采取行动唤醒正在它们

后面要更仔细地研究同步问题

并发：语言和库

提供并发的方式：设计有并发功能的语言；通过传统语言扩充或并发库

- 后两种方式更常见
- 目前在用的很多并行程序都是用针对向量机的 **Fortran**语言扩充，或用调用并发库的 **C/C++** 语言写出
- 随着 **Java** 和 **C#** 的影响增加，情况正在发生变化。但显式并行语言要在高性能并行应用方面取代 **Fortran**、**C** 和 **C++** 还需要一段时间

许多 **OS** 提供并发库

如 **Unix** 系统包含基于 **POSIX** 的 **pthread** 标准的库

支持消息传递的库可分为两类：

- 一类主要为程序内部进程间通信
- 另一类为跨程序通信，通常实现某种标准互联网协议，像文件 **I/O**

并发：语言和库

- 最流行的程序内消息传递包是 **PVM** 和 **MPI**
  - 两个包功能类似，各有所长
  - 存在针对 **C**、**C++** 和 **Fortran** 的 **PVM** 和 **MPI** 实现
- 对客户端对于服务器请求的通信，有一种技术称为远程过程调用（**RPC**），这种技术概念清晰，很有吸引力
  - 如果客户端需要调用服务器提供的某个功能（ **RPC**），就在的客户端建立与该远程过程对应的本地桩过程（用一个本地过程包装起来的远程过程接口）
  - 当客户调用本地桩过程时，这个过程将把实际参数包装成消息送到服务器相应端口，而后等待服务端消息
    - 服务器接到远程调用消息后执行相应的过程，完成有关计算后把得到的结果发回
    - 桩过程得到服务端回复后，将其结果参数的方式返回给客户端

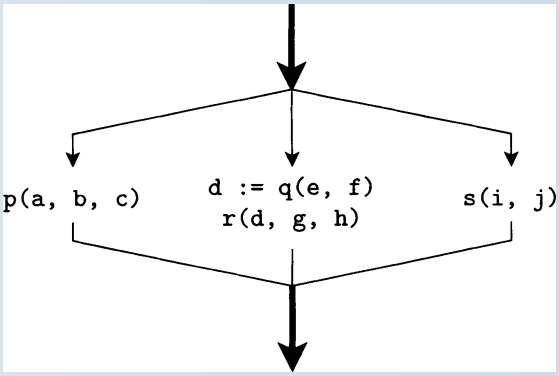
线程创建的形式

最简单的并发程序就是静态定义的一组线程。这种构造方式太受限制

语言或库一般都支持在运行中动态创建新线程。常见形式有下面几种：

Co-begin（Algol-68，Occam 等）

```
par begin                    # concurrent #
    p(a, b, c),
    begin                    # sequential #
        d := q(e, f);
        r(d, g, h)
    end,
    s(i, j)
end
```



特点：用一种语法结构围起一组可以并行执行的“作业”。这些作业的线程同时启动；所有作业都完成时整个语法结构的执行结束

包含一系列并发执行的语句的结构都可以称为 **co-begin**

线程创建的形式

并行循环（SR、Occam，Fortran 的一些方言）

```
co (i := 5 to 10) ->
    p(a, b, i)          # six instances of p, each with a different i
oc
```

性质：并行循环中的不同迭代生成一组同时执行的线程

HPF 是 Fortran 一种方言， Fortran 90 的 1995 修订接受其 forall 循环

为消除竞态条件，要求迭代的动同步：循环中语句只能是赋值或嵌套 **forall**；迭代中任何赋值语句中变量的所有读操作都必须在迭代里对左部的写操作之前完成；对左部的所有写操作必须在随后赋值的读操作之前完成

```
forall (i=1:n-1)
    A(i) = B(i) + C(i)
    A(i+1) = A(i) + A(i+1)
end forall
```

第一个语句读 **B** 的 $n-1$ 个元素和 **C** 的 $n-1$ 个元素，后更新 **A** 的 $n-1$ 个元素；第二个赋值语句读 **A** 的全部 n 个元素后更新其中 $n-1$ 个

线程创建的形式

加工时启动（Ada、SR）

用类似无参子程序的形式声明作业，在声明加工时创建其线程

```
procedure P is
    task T is
        ...
    end T;
begin -- P
    ...
end P;
```

作业 T 用一个 begin…end 块描述，控制进入过程 P 时同时启动一个线程执行 T 的块（和主进程并行）

如果 P 递归，就可能出现 T 的多个并发实例与执行 P 的进程（的当前实例）并发执行的情况

主程序的行为就像是一个初始的默认进程

P 和 T 的实例都完成是过程 P 的执行完成并返回

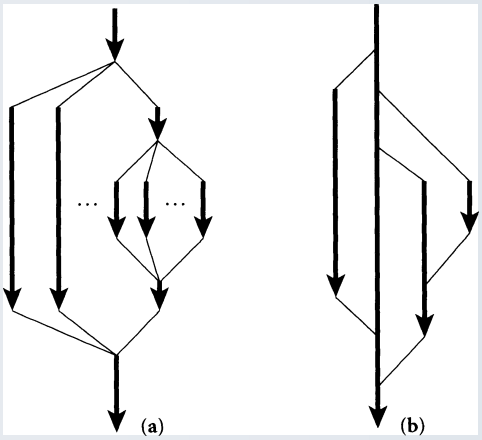
如果过程 P 的执行先完成，它在返回之前将等待 T 的相应实例（这个 P 实例开始执行时创建的那个 T 实例）完成

这一规则可以保证 P 的局部变量（按静态作用域规则，这些变量在 T 中可见）在 T 结束前不会被释放，因此不会出现悬空引用

线程创建的形式

Fork/Join（Ada、Java、Modula-3、SR、Unix 系统）

前几种创建机制生成的线程都是完好嵌套的，fork/join 支持更自由的并发执行模式。fork 的功能是立即创建线程，join 等待指定线程完成



(a) 完好嵌套的并行执行模式

(b) fork/join 生成的自由模式

Ada，定义作业类型后创建作业执行实例

```
task type T is
    ...
begin
    ...
end T;

pt : access T := new T;
```

用指向作业的指针引用新创建的作业实例（一个线程），相当于 fork

无对应 join 的操作

线程创建的形式

Java 创建线程的方式也是 fork/join

```
class image_renderer extends Thread {
    ...
    image_renderer( args ) {
        // constructor
    }
    public void run() {
        // code to be run by the thread
    }
}
...
image_renderer rend = new image_renderer( constructor_args );
rend.start();     rend.join();          // wait for completion
```

Java 还有 java.util.concurrent并行库。实际上不鼓励程序员创建线程，而是建议把作业（支持 Callable/Runnable 接口的对象）送给 Executor 对象，放入线程池的方式。这样做使作业与线程的概念分离，因为 Executor 类可以根据基础平台的特点优化线程的并行执行和调度

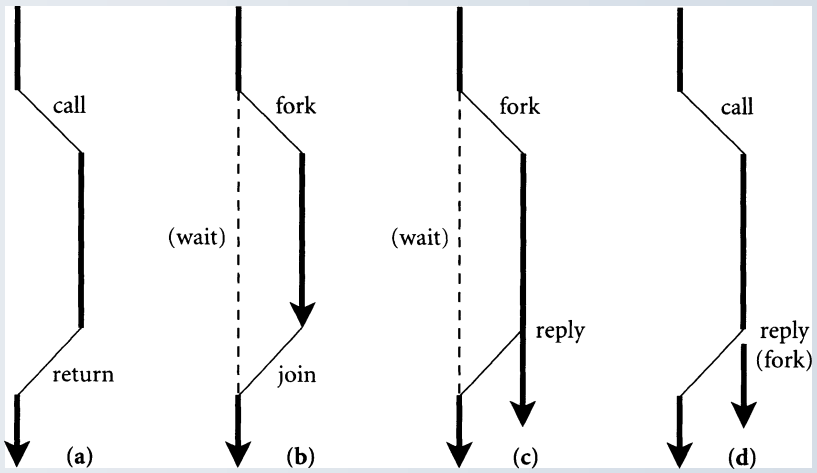
线程创建的形式

隐式接收（**RPC，远程过程调用**）

前面机制都是在现存线程的同一个地址空间里创建新线程。在RPC系统里，为响应来自另外地址空间的请求，常常需要自动创建新线程

服务器不是用接到请求的线程执行操作，而是把通道（链接、套接字）约束到一个局部线程上，接到请求时自动生成一个新线程去处理这个请求

早回复



普通过程调用

过程调用的双线程实现

分叉返回后继续执行

早回复后继续执行（做一点工作后分支）

线程创建的形式

引入早回复的主要原因是应用的需要，父线程（接到请求的线程）需要保证新线程在父线程继续执行前已完成了某些初始化工作

万维网浏览器的例子：

- 页面格式化线程通常要为内嵌图像处理创建子线程，子线程创建服务器连接后接受图像数据。服务器将先送过来图像的大小，页面线程在知道了这一大小后才能正确安置页面中的其他文字和其他图像
- 早回复机制使父线程创建绘图子线程后等它送回图像大小。得到图像大小的信息后两个进程就并行地各自继续执行

Java 中进程创建（对象创建）与 **start** 调用分开，很容易处理这种情况

在 **Java** 中实现浏览器实例：

- 页面格式化线程创建图像描绘线程后先调用它的 **get_size** 方法，该方法负责创建服务器连接，并取得图像的大小信息返回
- 作为子线程成员函数的 **get_size** 把数据保存在线程对象里，包括图像大小和服务器连接，线程的 **run** 方法可以直接使用它们