操作系统

L03 进程

胡燕
大连理工大学 软件学院

2.E-OS结构回顾

核心内容

- OS的几大服务
- OS提供给应用层的最原始接口
- OS结构设计的几种典型类型

单选题 3分

设置

以下关于微内核的描述，描述不正确的是（ ）。

A 微内核系统结构清晰，利于编程开发

B 微内核代码量少，有良好的可移植性

C 微内核功能代码可以互相调用，性能高

D 微内核有良好的伸缩性，可扩展性高

提交

单选题 4分

设置

以下哪种操作系统的设计结构是综合了宏内核和微内核特性的混合式内核结构：（ ）。

A DOS

B Mach

C MacOS

D Multics

提交

单选题 3分

设置

用户程序必须通过（ ）向操作系统提出访问外部设备的请求。

A I/O指令

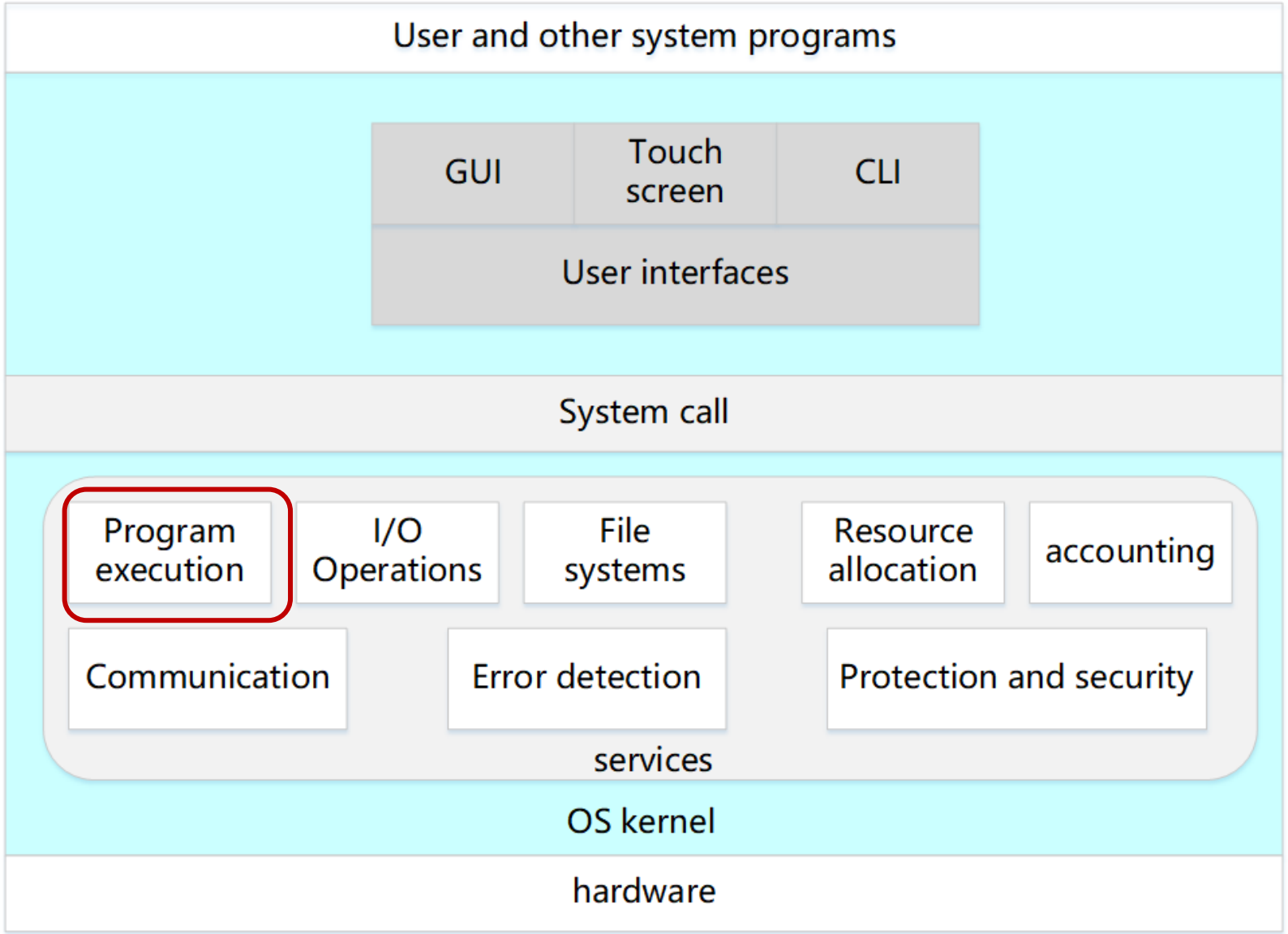
B 系统调用

C 中断

D 创建新的进程

提交

什么是进程



程序的执行

是一个过程

- 启动
- 初始化
- 执行代码中的指令
- 退出

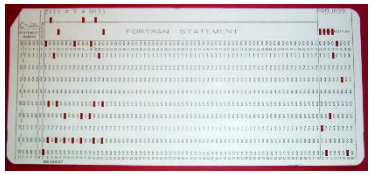
OS将其抽象为一个核心概念

进程

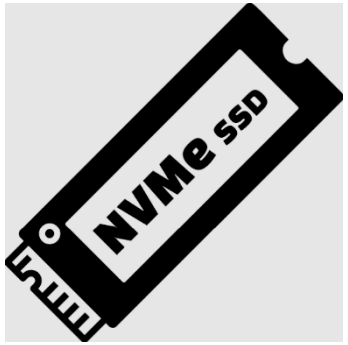
Process

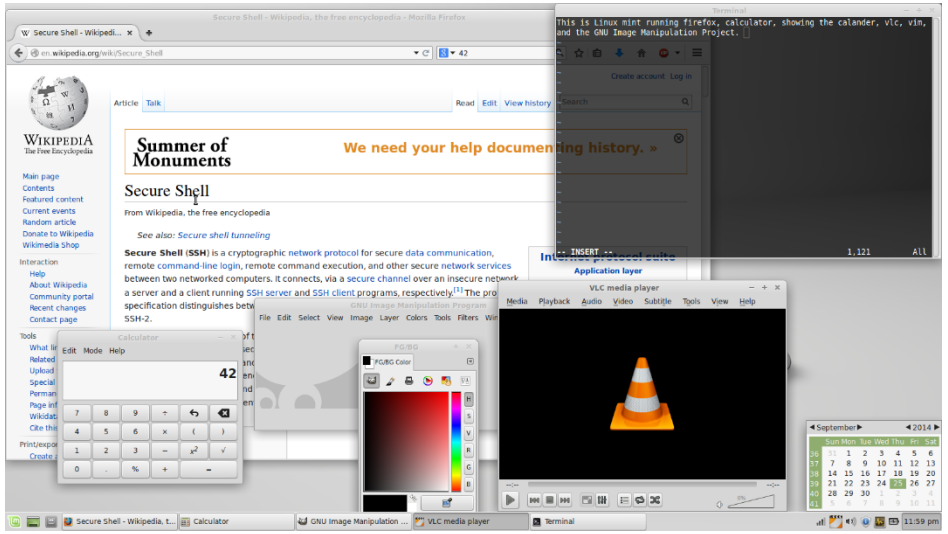
什么是进程

程序与进程



```
Current date is Tue 1-01-1980
Enter new date:
Current time is 21:35:24.18
Enter new time:


The IBM Personal Computer DOS
Version 2.00 (C)Copyright IBM Corp 1981, 1982, 1983

A>dir

 Volume in drive A has no label
 Directory of A:\

COMMAND  COM  17664  3-08-83  12:00p
FORMAT   COM   6016  3-08-83  12:00p
CHKDSK   COM   6400  3-08-83  12:00p
SYS      COM   1408  3-08-83  12:00p
DEBUG    COM  11904  3-08-83  12:00p
SLOOP              32  1-01-80   7:44p
      6 File(s)    292864 bytes free

A>_
```





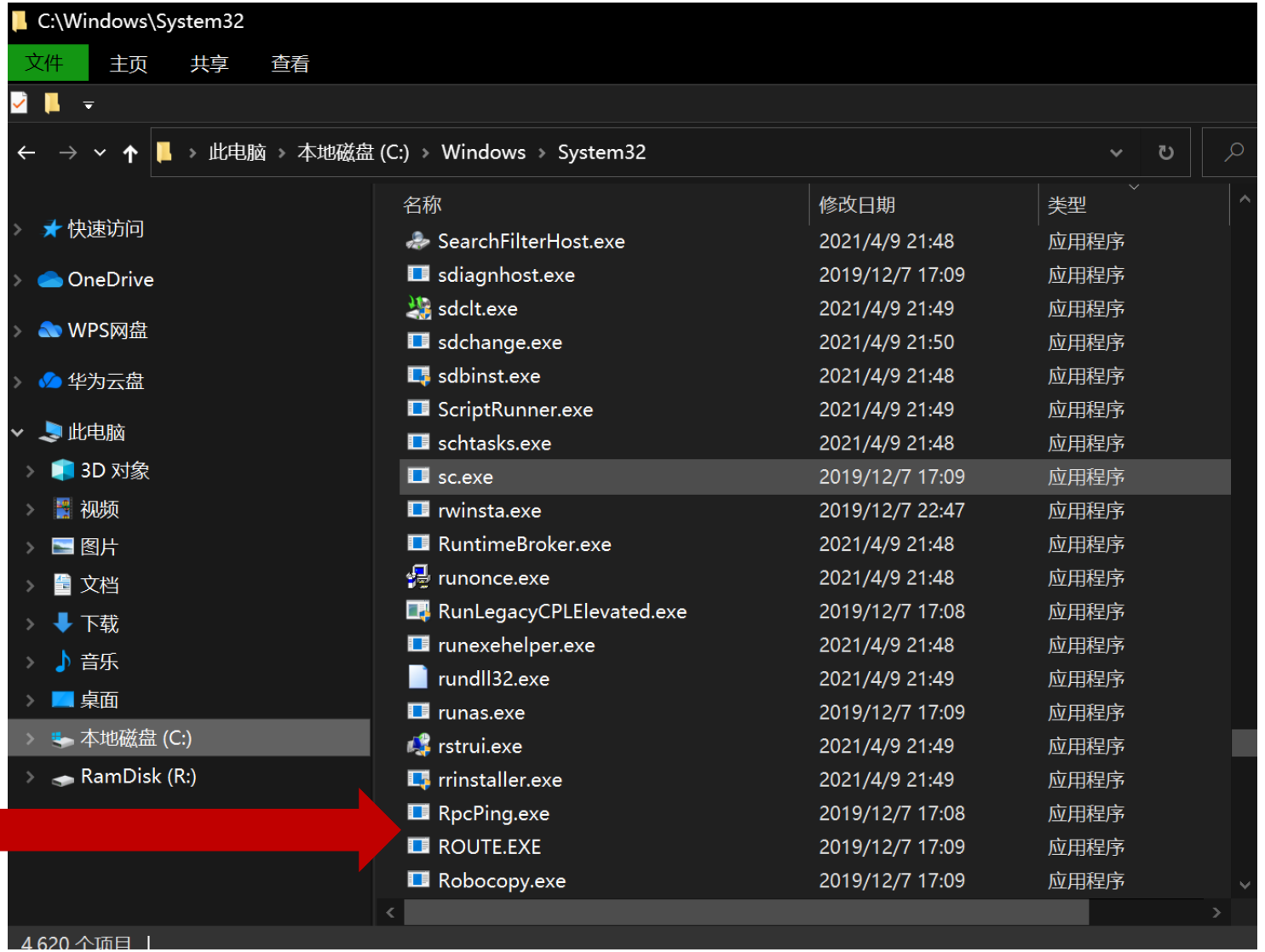
程序：静态的存在
=>纸带上的0和1，软盘/硬盘上的可执行文件

进程：动态的对象
=>程序的执行

什么是进程

Q1:什么是程序 (What is a program?)

例如:
C:\Windows\System32下可以找到很多.exe文件
（它们是Windows下的可执行程序）



什么是进程

Q2:程序从何而来?

开发人员完成源码开发

编译器编译源代码，生成可执行文件

什么是进程

Q3:如何执行一个程序?

方式1：CLI，键入命令

方式2：GUI，鼠标点击

什么是进程

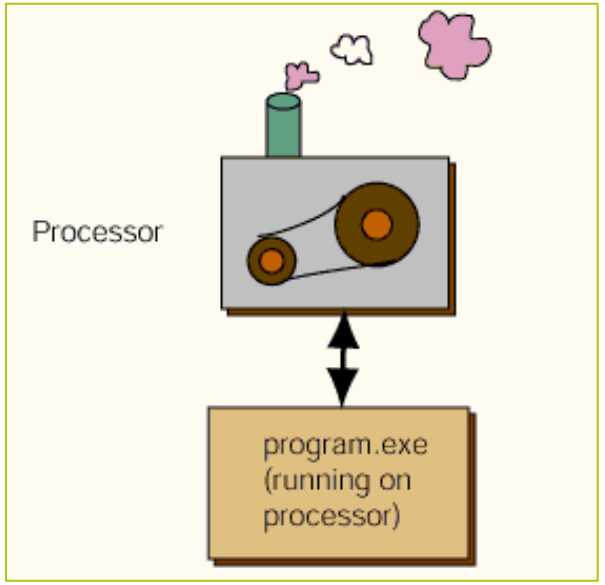
Q4:如何描述一个程序的执行过程?

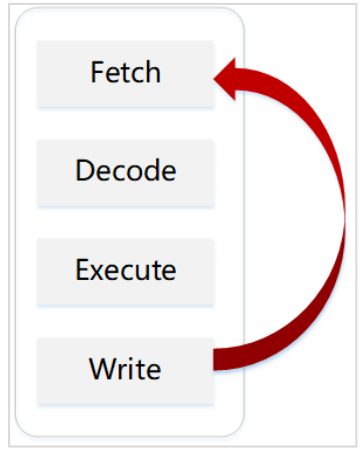
程序执行过程中经历了什么?

什么是进程

程序

启动





方式1：CLI，键入命令

方式2：GUI，鼠标点击

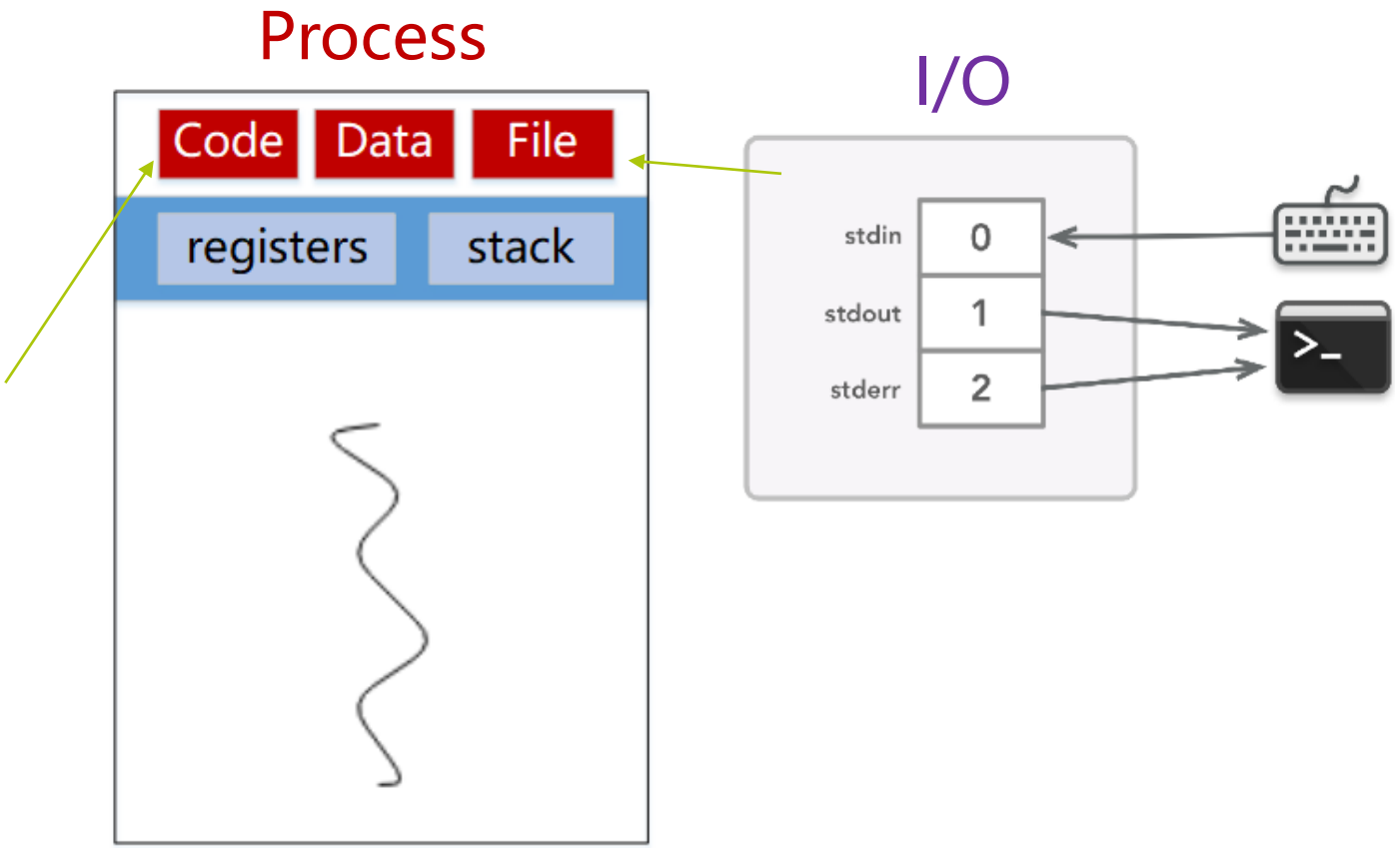
Program in Execution =?

什么是进程

- **进程（Process）：运行中的程序（A program in execution）**
  - 程序在给定输入下的一次执行
  - 进程是一个动态的概念。进程从开始到执行结束，有一个完整的生命周期

Program

```c
/* Demonstrates the  getchar() function. */

#include<stdio.h>

main()
 {
   int ch;

   while ((ch = getchar()) != '\n')
   putchar(ch);

   return 0;
 }
```



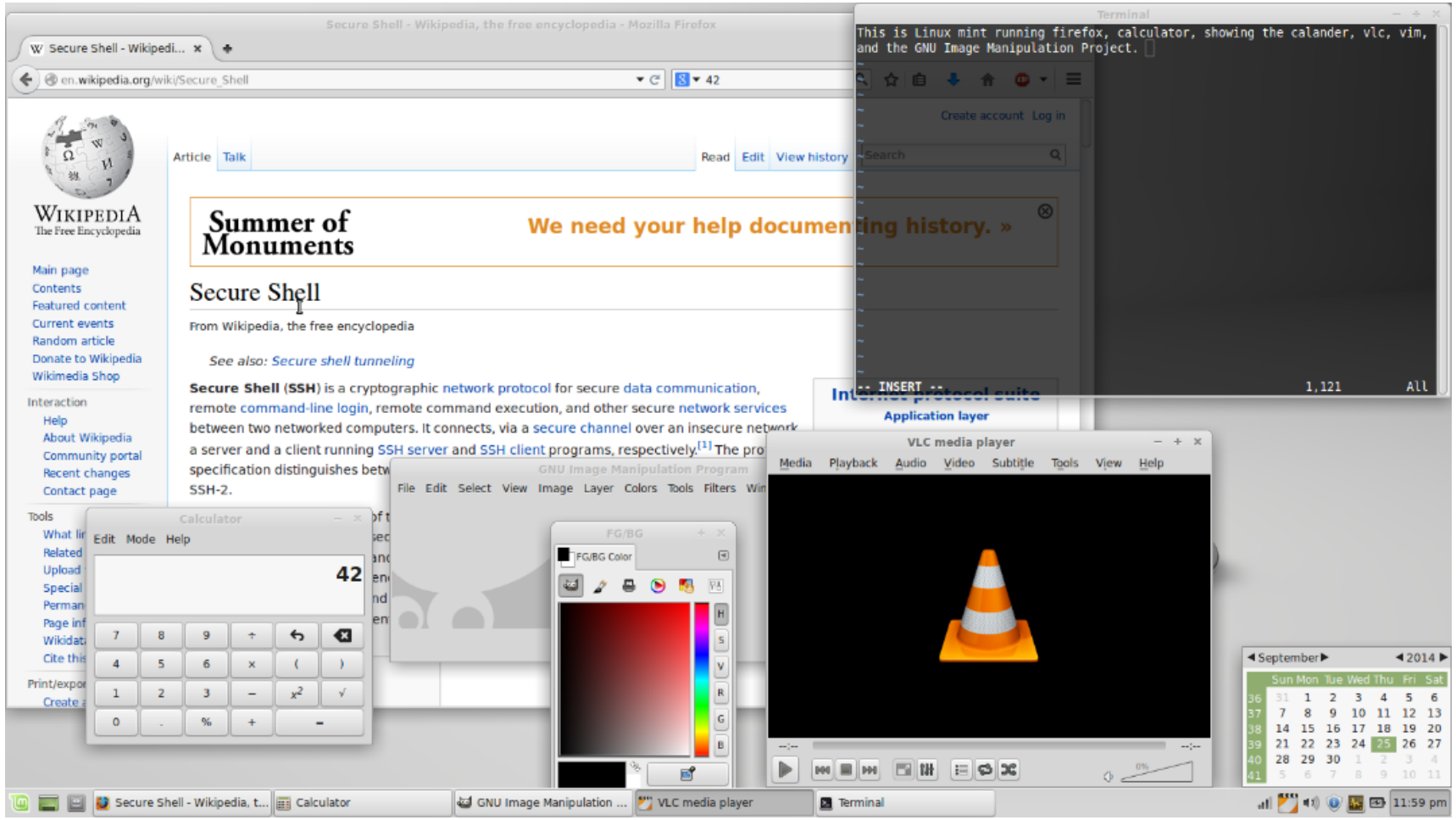
什么是进程

- **相似的概念：作业（Job），任务（Task）,进程（Process）**

作业	任务	进程
批处理系统	Unix/Linux	Windows

为什么需要进程概念

- Q:为何需要专门建立进程概念



核心需求来源于：
多任务（Multitasking）

现代计算机硬件日渐强大，需要多任务来充分利用

任务多了，就会产生资源竞争

于是，OS为每个任务（程序的执行）建立进程概念，作为分配资源的基本单位。

多道程序v.s.多任务

- **批处理系统（Batch System）**

单道批处理系统 → 多道批处理系统

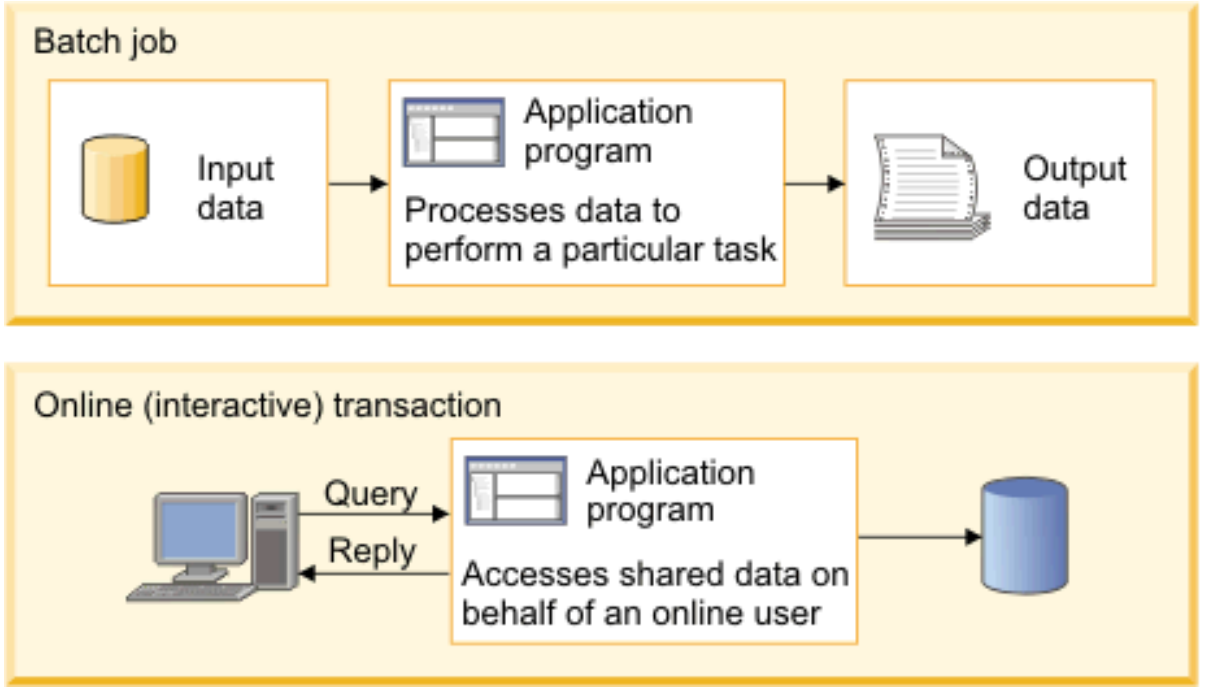
多道程序（Multi-Programming）

多道程序v.s.多任务

- **多任务系统（MultiTasking System）**

增加交互性

多道批处理系统 → 多任务系统



多道程序v.s.多任务

- **多道程序度（Degree of Multiprogramming, Dm）**
- **多道批处理系统或多任务系统中，自然地就引入了一个新概念：**
  - **多道程序度 Dm**

单道程序操作系统	多道程序操作系统
Dm=1	Dm > 1

多道程序v.s.多任务

- **讨论：**

- **引入多道程序的意义是什么？多道程序度 Dm是不是越大越好，为什么?**

多任务

- **现代多任务系统的挑战**
  - **多道程序度 Dm > 1时，系统如何为进程分配CPU、内存等资源**

分配CPU → 调度(Scheduling)

分配内存 → 进程地址空间（映像结构）

支持交互 → 为不同的业务场景设计适用的调度算法

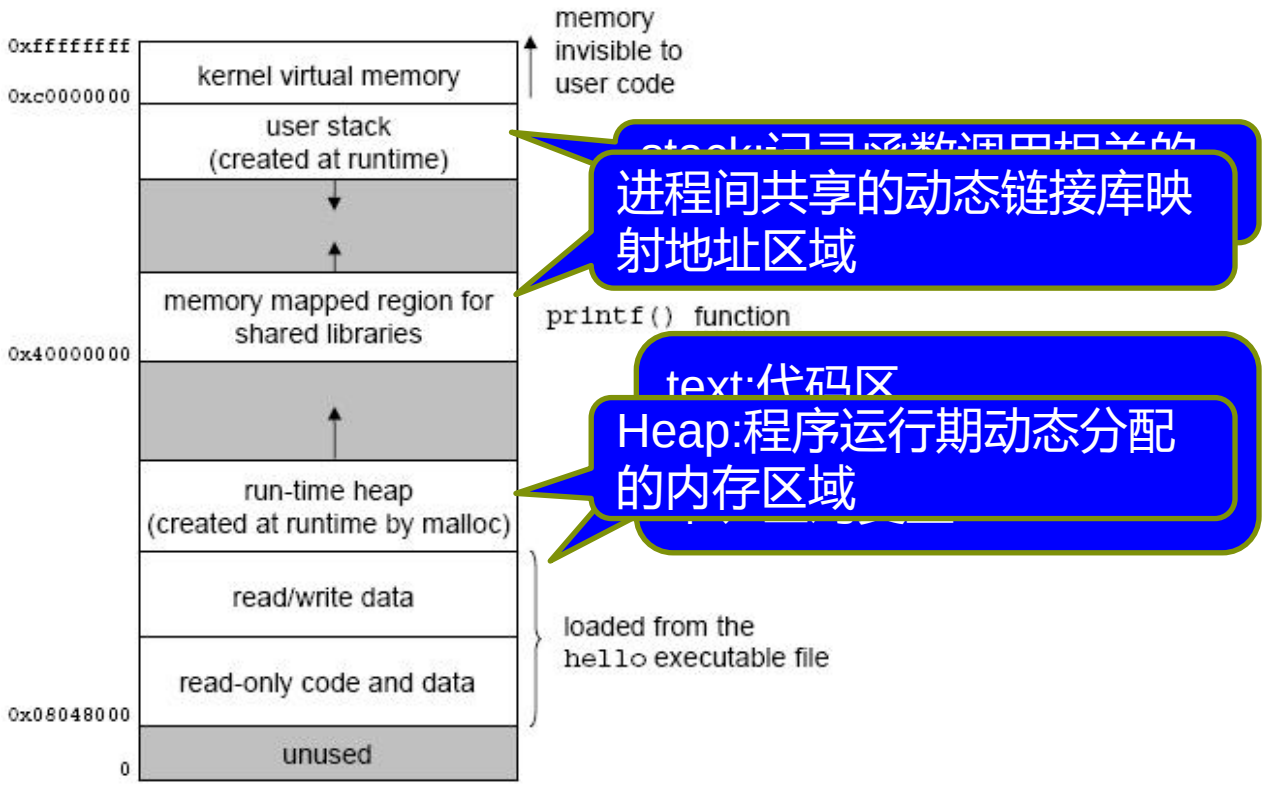
进程映像

- **多任务环境下，为每个进程分配独立的地址空间**
  - **保证对内存资源的有序分配，避免无序竞争**

-为每个进程对象设置专门的管理数据结构（PCB）
-为每个进程设立相对独立的进程地址空间（进程映像），形成隔离的效果

进程映像

- 进程地址空间构成
  - 代码、数据、栈、堆、共享库映射区域



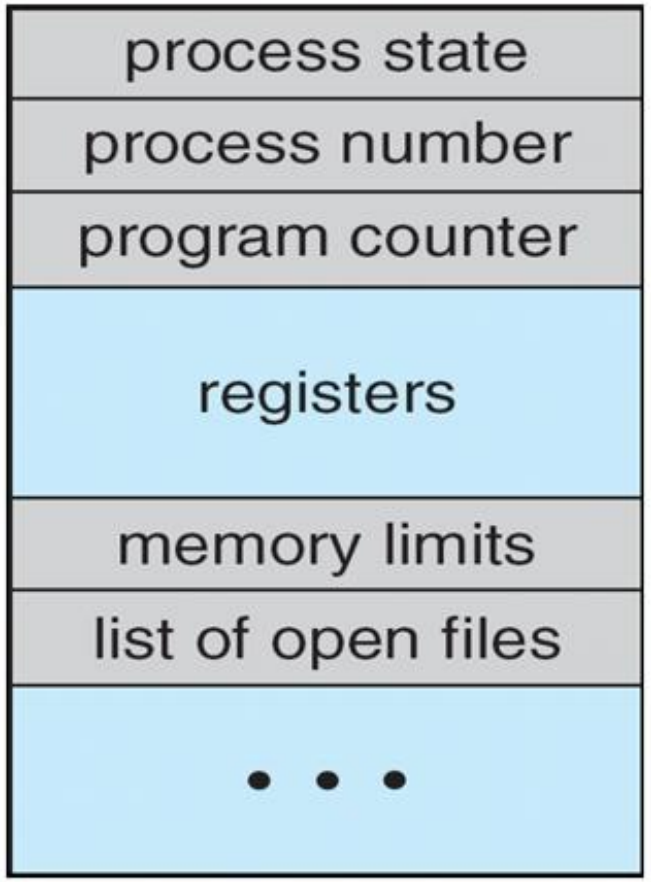
进程控制操作

- **进程控制操作，影响进程状态变化**

进程控制操作	说明
创建进程	创建新的进程
撤销进程	终止一个进程
阻塞进程	在进程提出的系统服务请求（如I/O操作请求），因某种原因无法得到内核的及时响应，而采取的进程控制操作
唤醒进程	阻塞进程所等待的事件发生（如I/O操作完成），系统唤醒进程的操作

进程控制块

- **Process Control Block (PCB)**

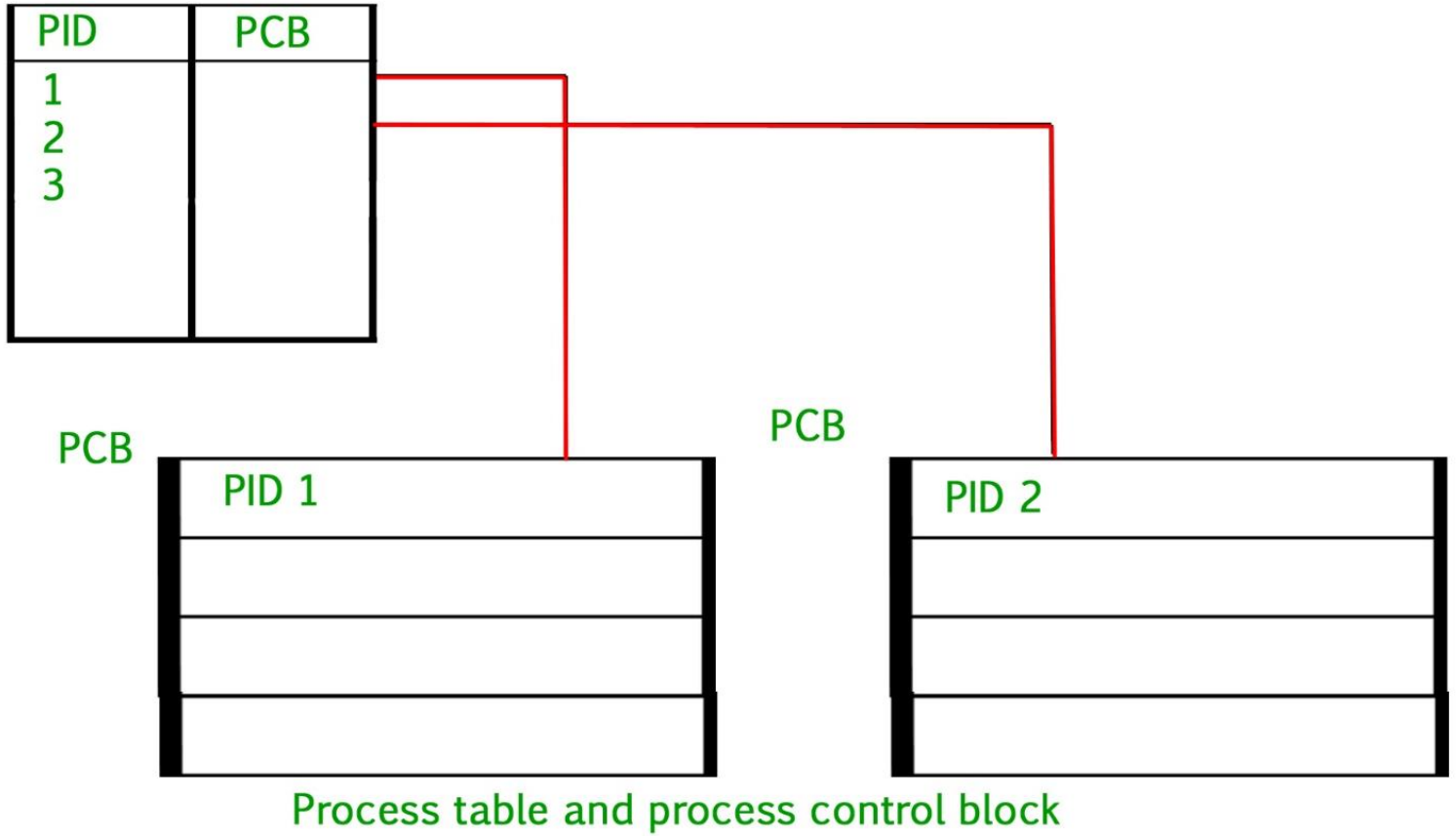


- PCB是OS内核中用来表示进程的唯一数据结构

PCB是OS为控制进程而设计的专门数据结构
OS通过PCB来获取进程相关的信息和对进程实施控制

进程控制块

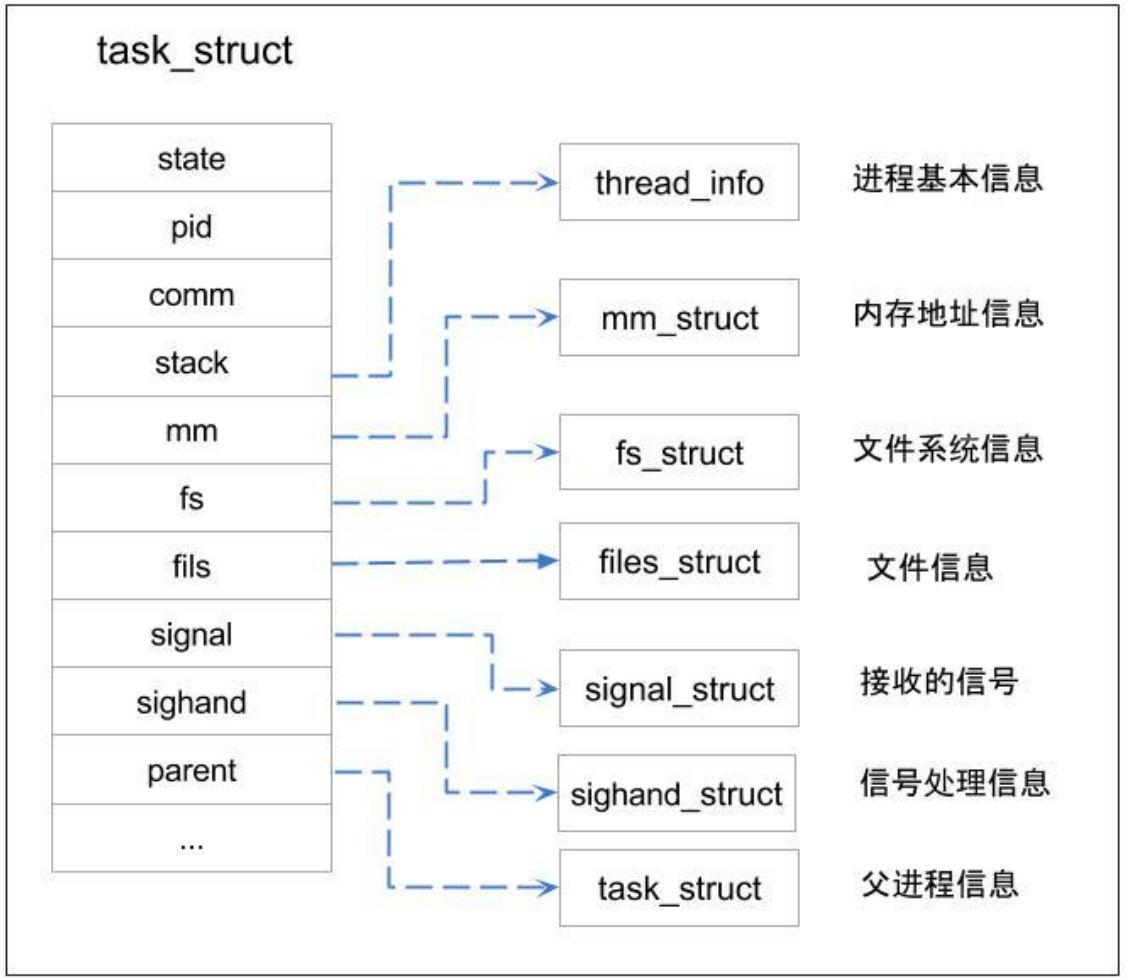
- 内核通过PCB列表（通常实现为链表），将进程组织在一起



Process table and process control block

进程控制块

- 示例：Linux PCB - task_struct



- 讨论：
  - 简述进程的概念
  - 为何要引入进程概念?
  - 进程与程序的关系

进程与程序

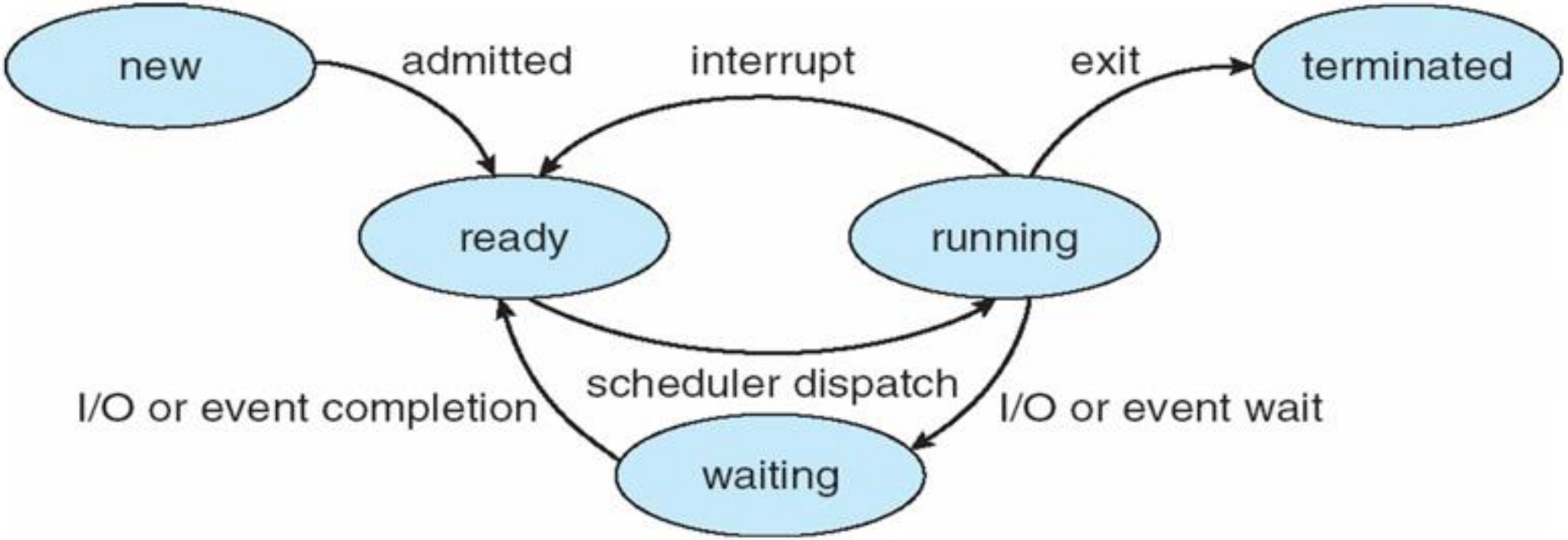
程序是一个静态的概念

进程是一个动态的概念

- 存在生命周期
- 生命周期中，会经历状态变化

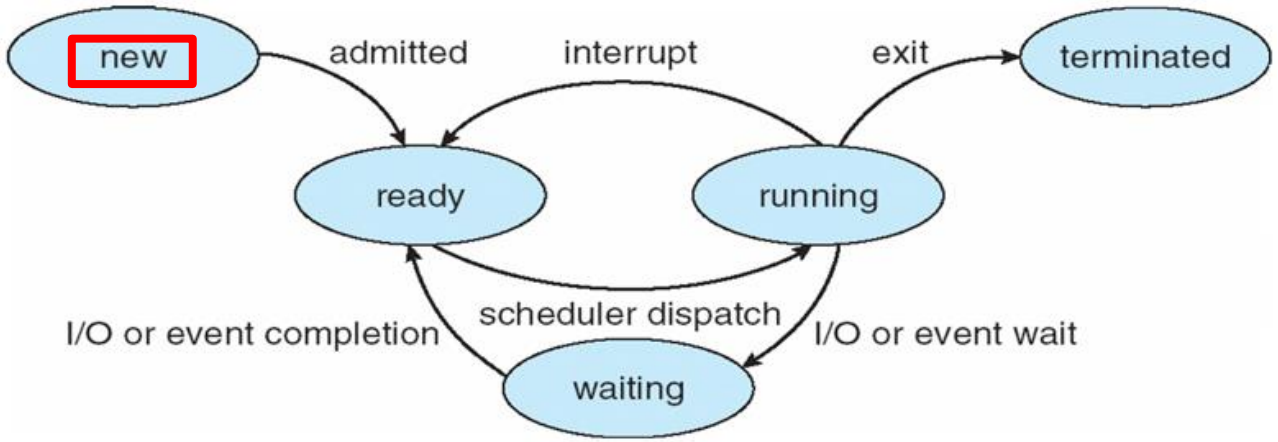
进程的5状态迁移图

- 进程是动态的概念，生命周期内会经历各种状态变化



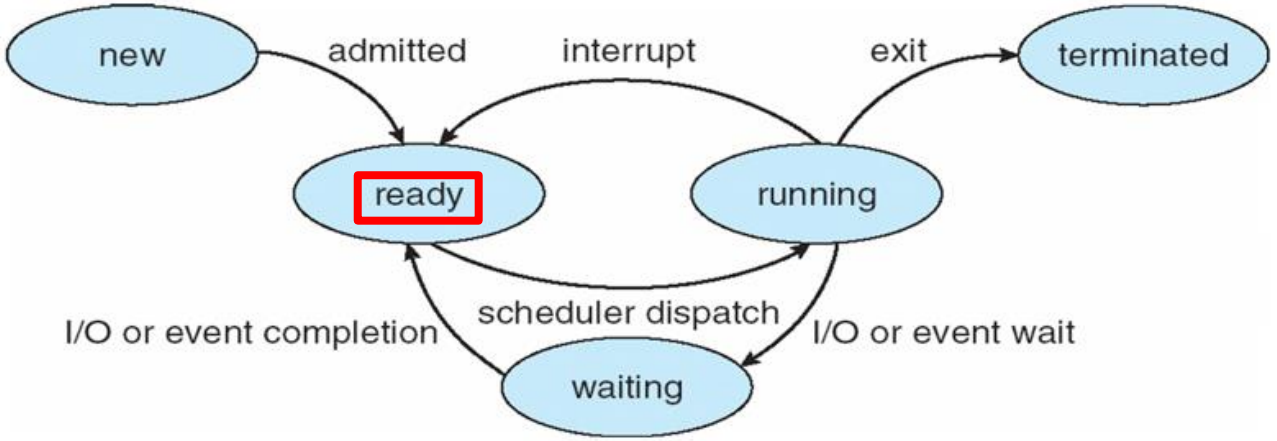
进程的5种典型状态

- **进程在其整个生命周期中，会经历诸多状态**



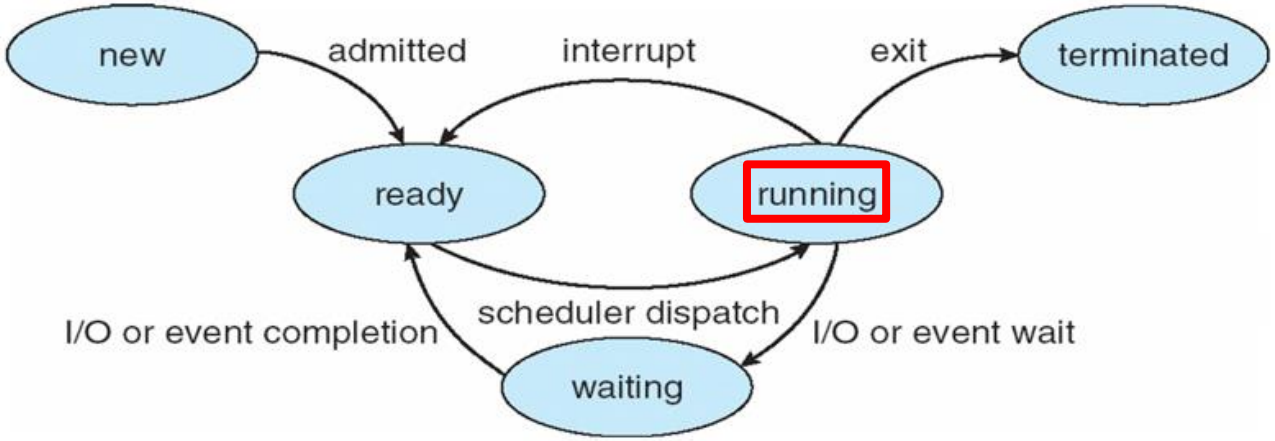
- new(新建状态)
  - 进程刚被创建好时，处于new状态
  - 等待被系统接纳

进程的5种典型状态



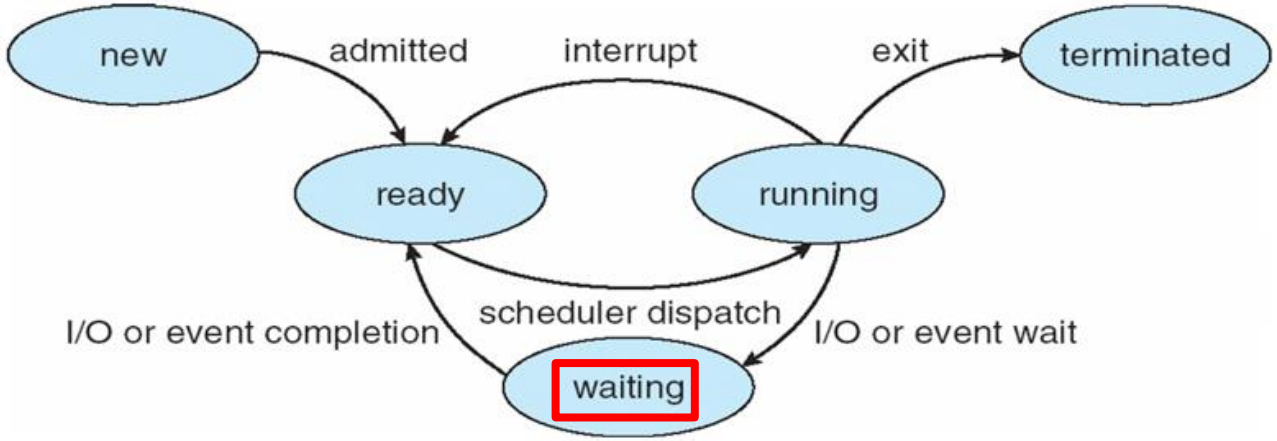
- ready(就绪状态)
  - 已经被成功加载进内存并初始化完毕，等待系统分配CPU资源

进程的5种典型状态

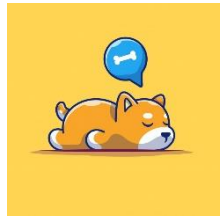


- running(运行状态)
  - 已经从就绪状态被调度器选中，正在利用CPU执行

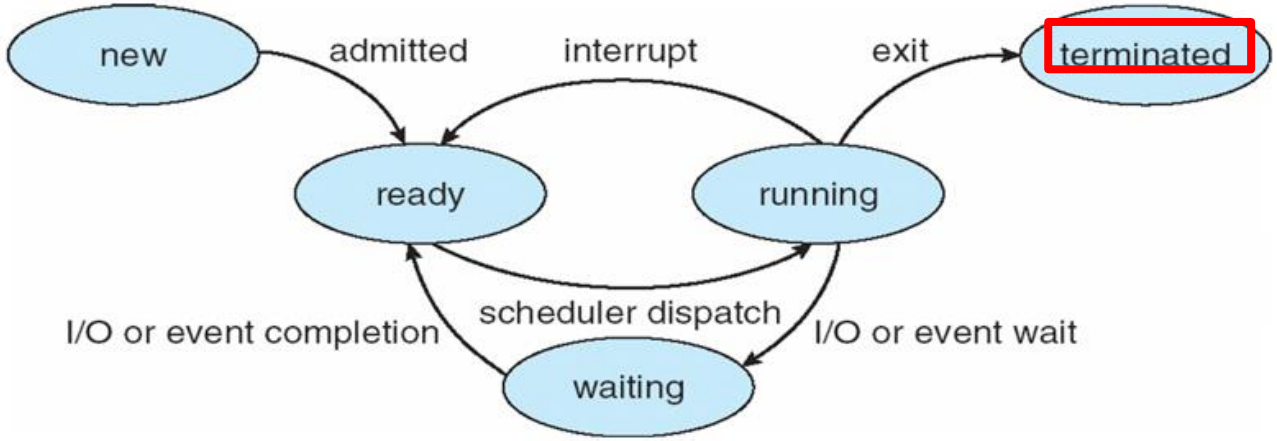
进程的5种典型状态



- waiting(阻塞状态)
  - 进程执行受到阻碍，必须暂停的状态
  - 阻碍进程继续执行的因素可能有：
    I/O，等待某个事件发生



进程的5种典型状态

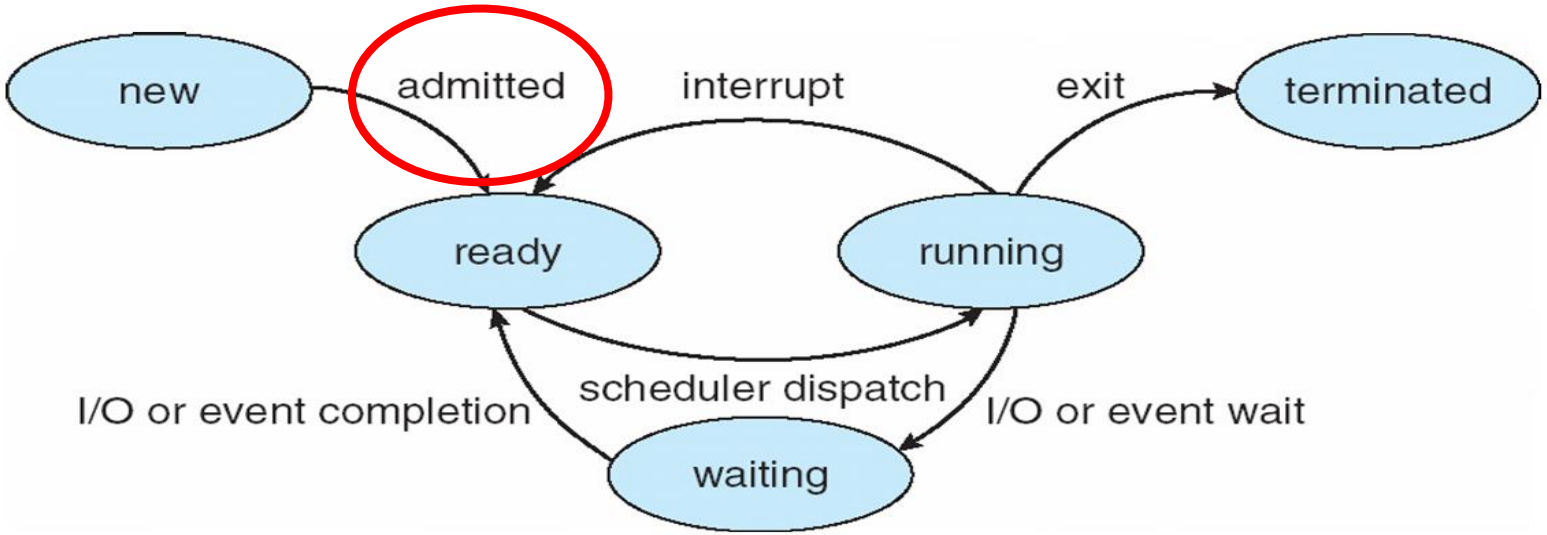


- Terminated(终止状态)
  - 进程执行完毕后等待被系统清除的状态

new ➡ ready

new（新建）➡ ready（就绪）

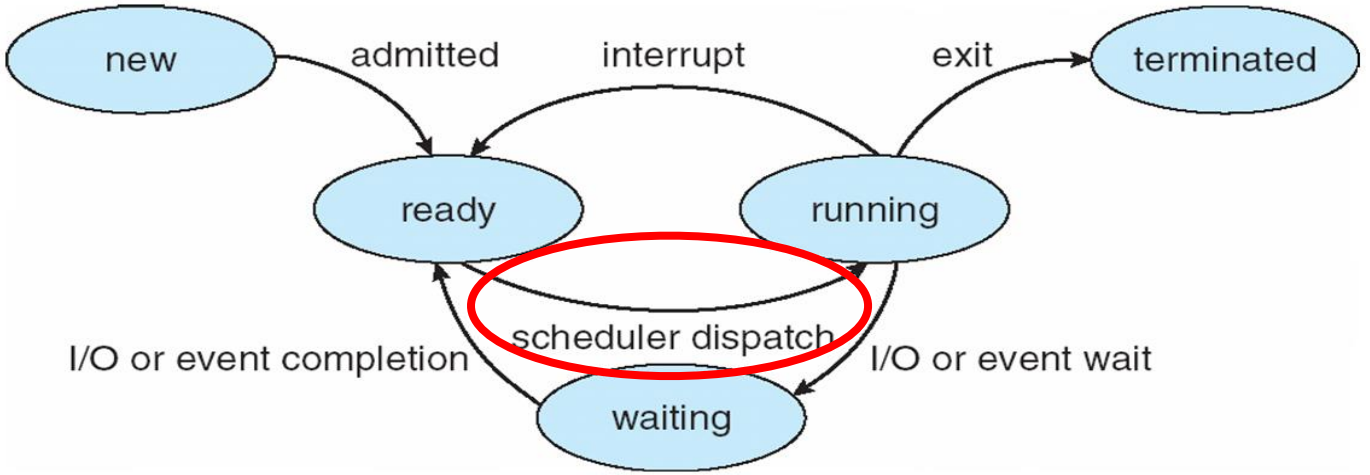
- 进程的数据结构创建完毕，初始化好之后，操作系统将其状态标记为就绪，将进程控制块插入进程就绪队列。



ready ➡ running

ready（就绪） ➡ running（运行）

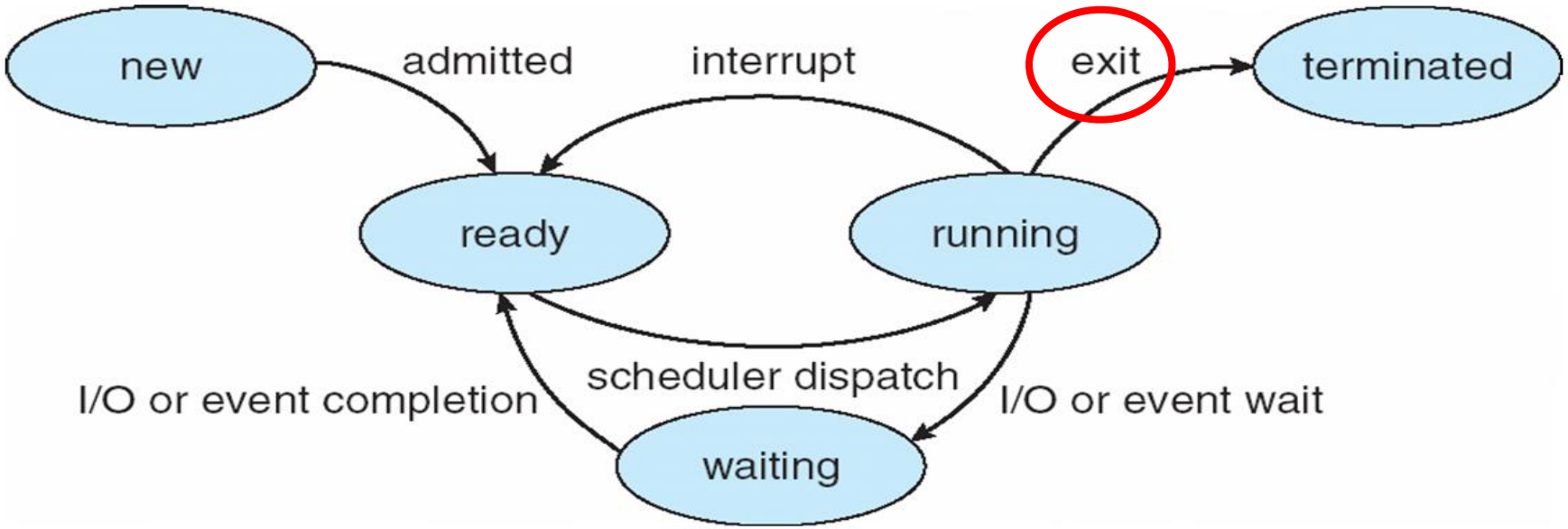
- 就绪进程被派遣程序安排到CPU上运行。



running ➡ terminated

running（运行） ➡ terminated（终止）

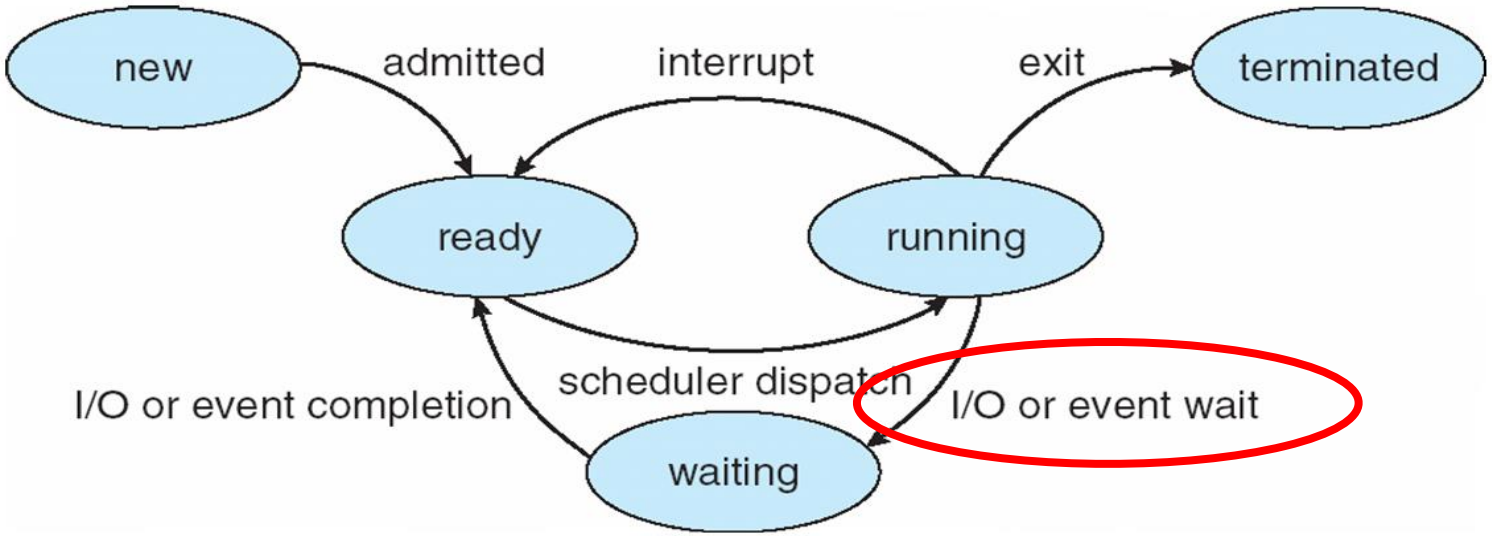
- 进程运行结束，或因出错被异常终止。



running ➡ waiting

running（运行态） ➡ waiting（阻塞态）

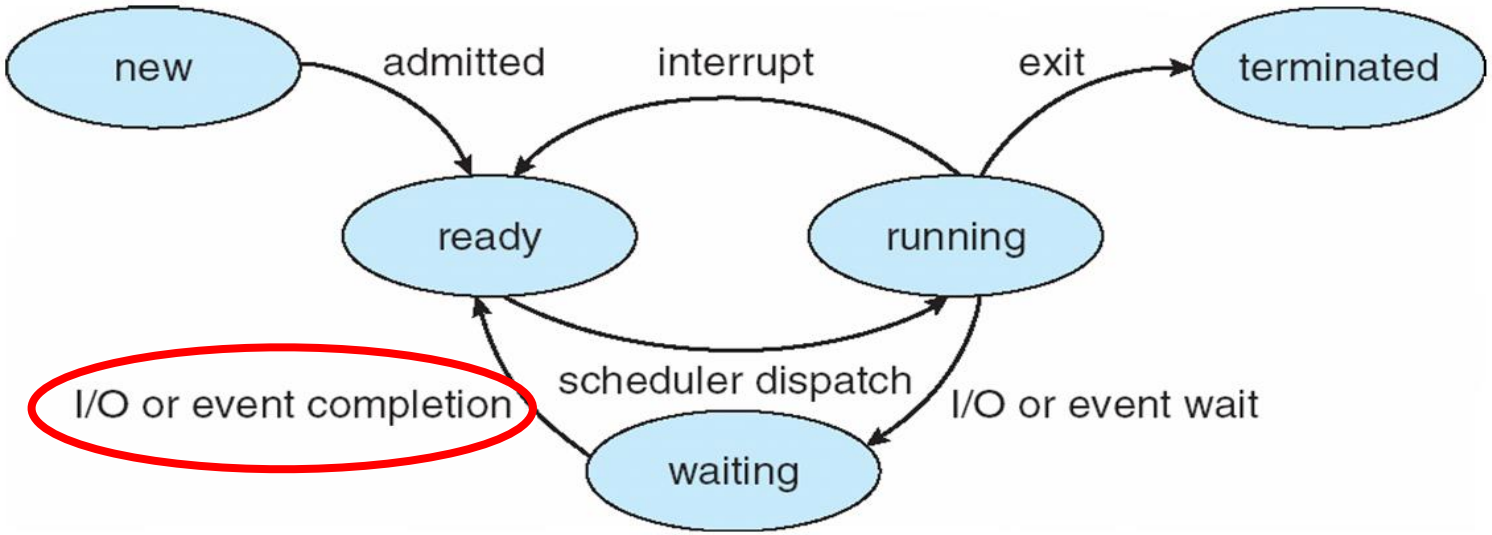
- 进程因等待I/O或某种事件，转入阻塞状态。



waiting ➡ ready

waiting（阻塞） ➡ ready（就绪）

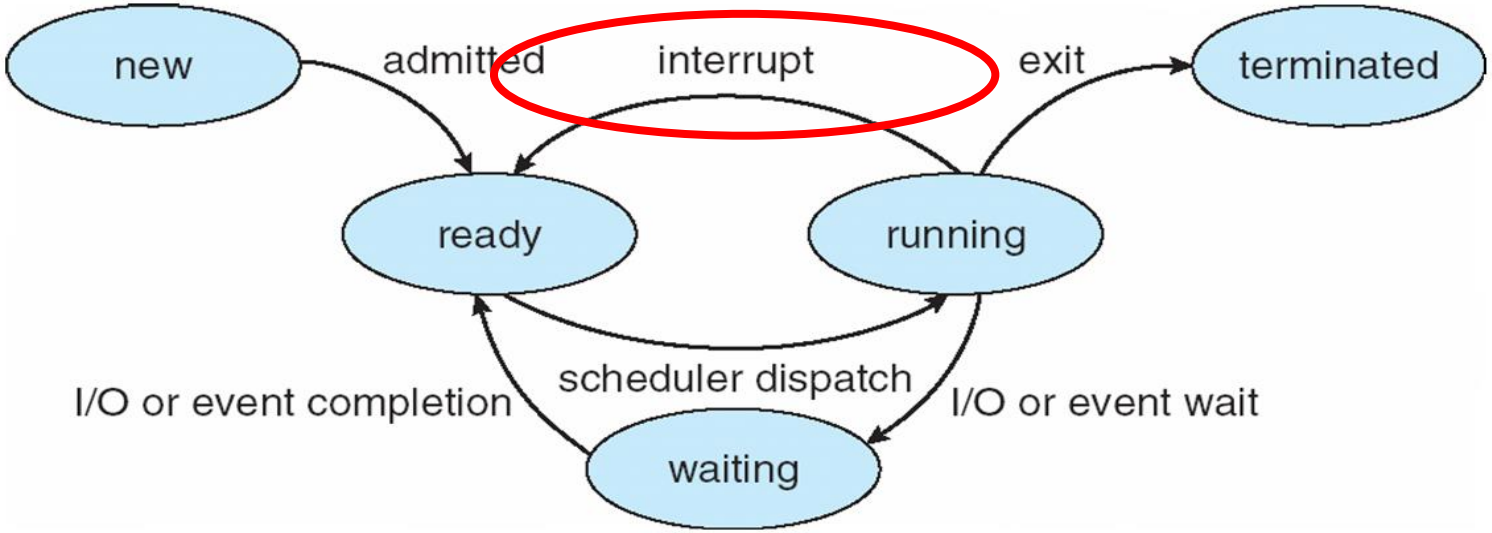
- IO完成的事件，会使等待该IO的进程被唤醒，转入就绪状态。



running ➡ ready

running（运行） ➡ ready（就绪）

- 进程因中断事件，从运行态切换至就绪态。



进程的状态迁移

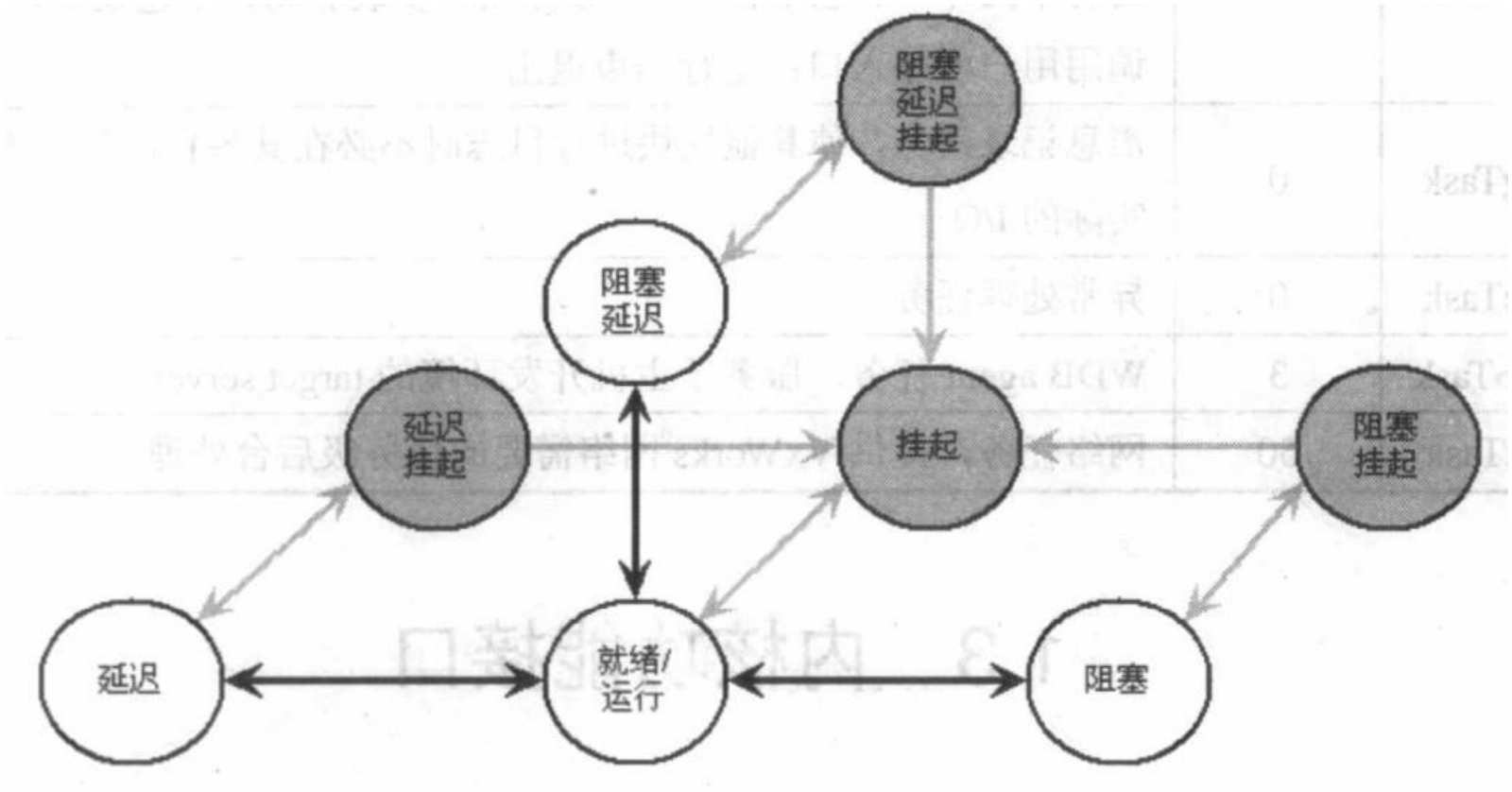
- 讨论：
  - 通用操作系统中都会重点实现进程管理模块
  - 是不是所有的操作系统设计中都会实现进程概念?
  - 进程或任务的状态是否只会实现刚刚介绍的5种状态的迁移?

进程的状态迁移

- **示例：**
  - VxWorks、RT-Thread均以多任务共享同一全局地址空间的形式实现
    - 核心：高效率
    - 也提供了支持进程管理的选项（进程管理作为可以选择的模块）

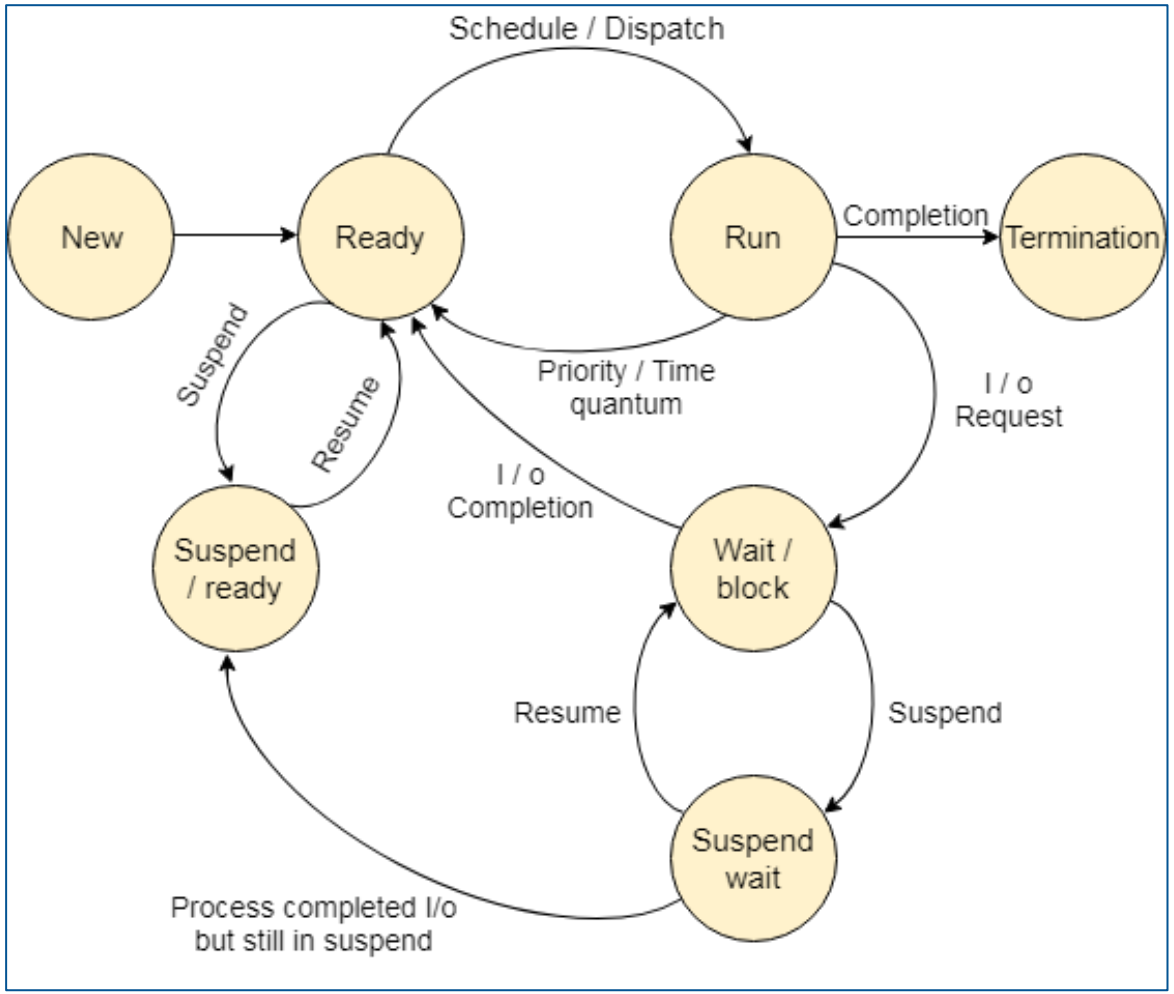
进程的状态迁移

- **示例：VxWorks的任务状态迁移图**



进程的状态迁移

- **示例：任务的七状态迁移图**



Linux下典型的进程控制操作

- **Linux下的基本进程控制操作**
  - 创建进程: fork()
  - 撤销进程: exit(), abort()
  - 阻塞进程: wait(), waitpid()
  - 唤醒进程: signal()

- **获取进程的PID**

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    pid_t my_pid;
    my_pid = getpid();
    printf("My process ID is: %d\n", my_pid);
    return 0;
}
```

- **A simple fork() example**

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    pid_t child_pid;
    child_pid = fork();
    if (child_pid < 0) {
        // Fork failed
        perror("Fork failed");
        return 1;
    } else if (child_pid == 0) {
        // Child process
        printf("Child process: My PID is %d\n", getpid());
    } else {
        // Parent process
        printf("Parent process: My PID is %d\n", getpid());
        printf("Parent process: Child process ID is %d\n", child_pid);
    }
    return 0;
}
```

- **Parent Process part with waitpid() call**

```
else {
    // Code executed by parent process
    printf("Parent process: Child process ID is %d\n", child_pid);
    // Wait for the child process to complete
    int status;
    waitpid(child_pid, &status, 0);
    if (WIFEXITED(status)) {
        printf("Parent process: Child exited with status %d\n", WEXITSTATUS(status));
    }
}
```

- **Fork代码实践：例1**

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

- **Fork代码实践：例2**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int x = 1;
    pid_t p = fork();
    if(p<0){
        perror("fork fail");
        exit(1);
    }
    else if (p == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);

    return 0;
}
```

Summary

小结：

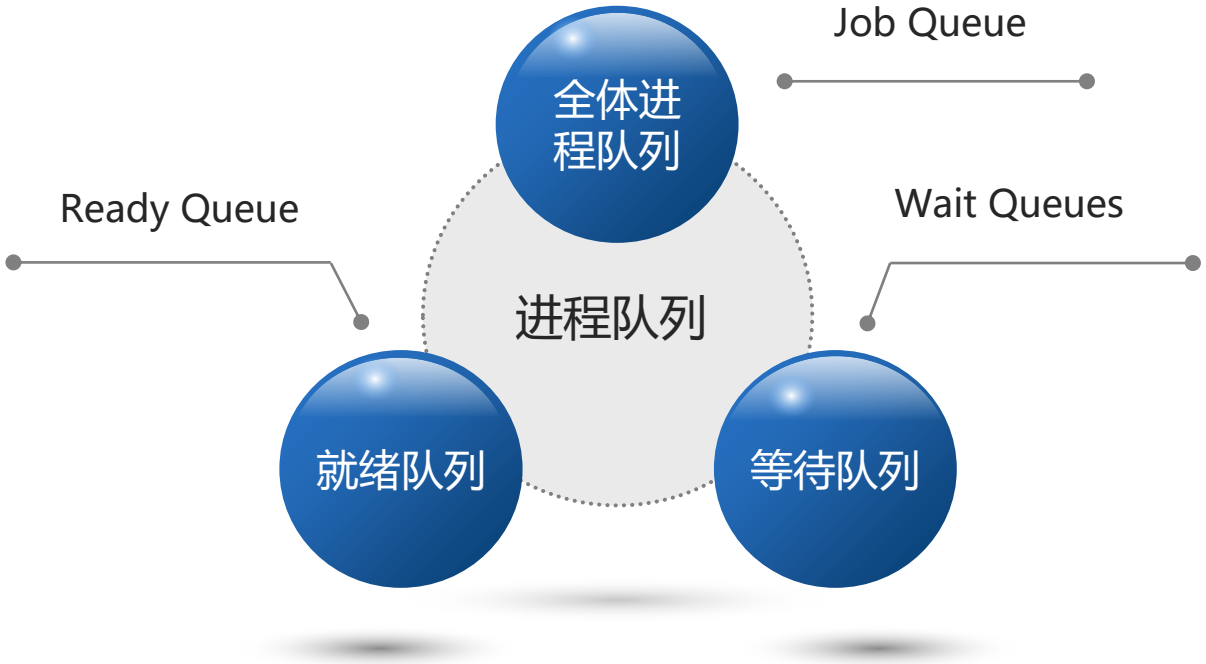
- 进程概念
- 进程状态迁移

为何需要进程调度

- **为什么要进行调度**
  - 多任务竞争使用CPU资源，需要对它们进行协调
  - 进程不断在变换状态，操作系统需要对不同状态的任务进行管理
- **调度的基本要求**
  - 主要目标：提高CPU利用率 (Maximize CPU Usage)
  - 在不同进程间快速切换，使得多个进程可以分时共享CPU资源
  - 调度器要在合适的时间点，从就绪的进程中选择下一个在CPU上执行的进程

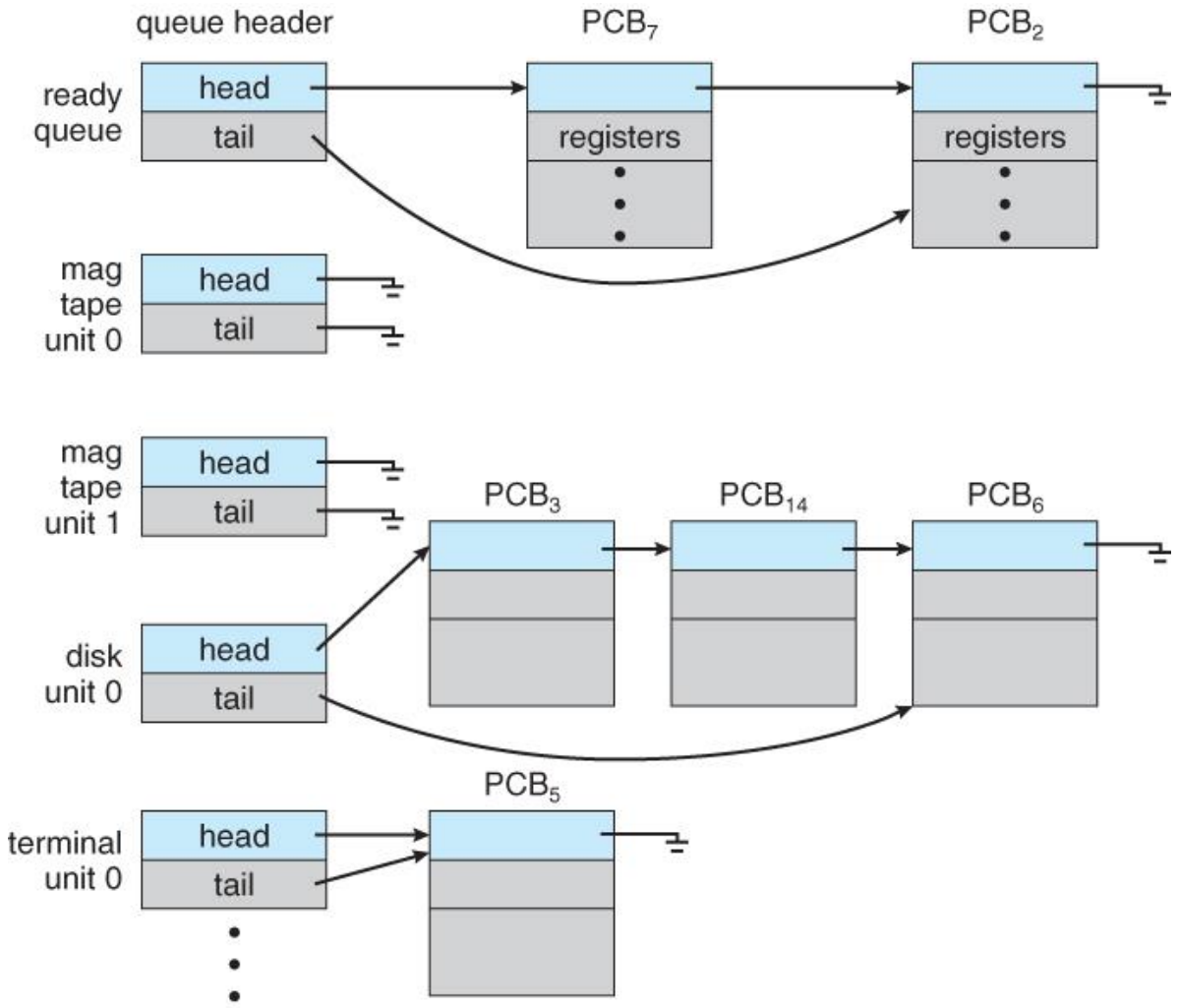
进程队列

- 通过各种队列对不同状态的进程进行管理



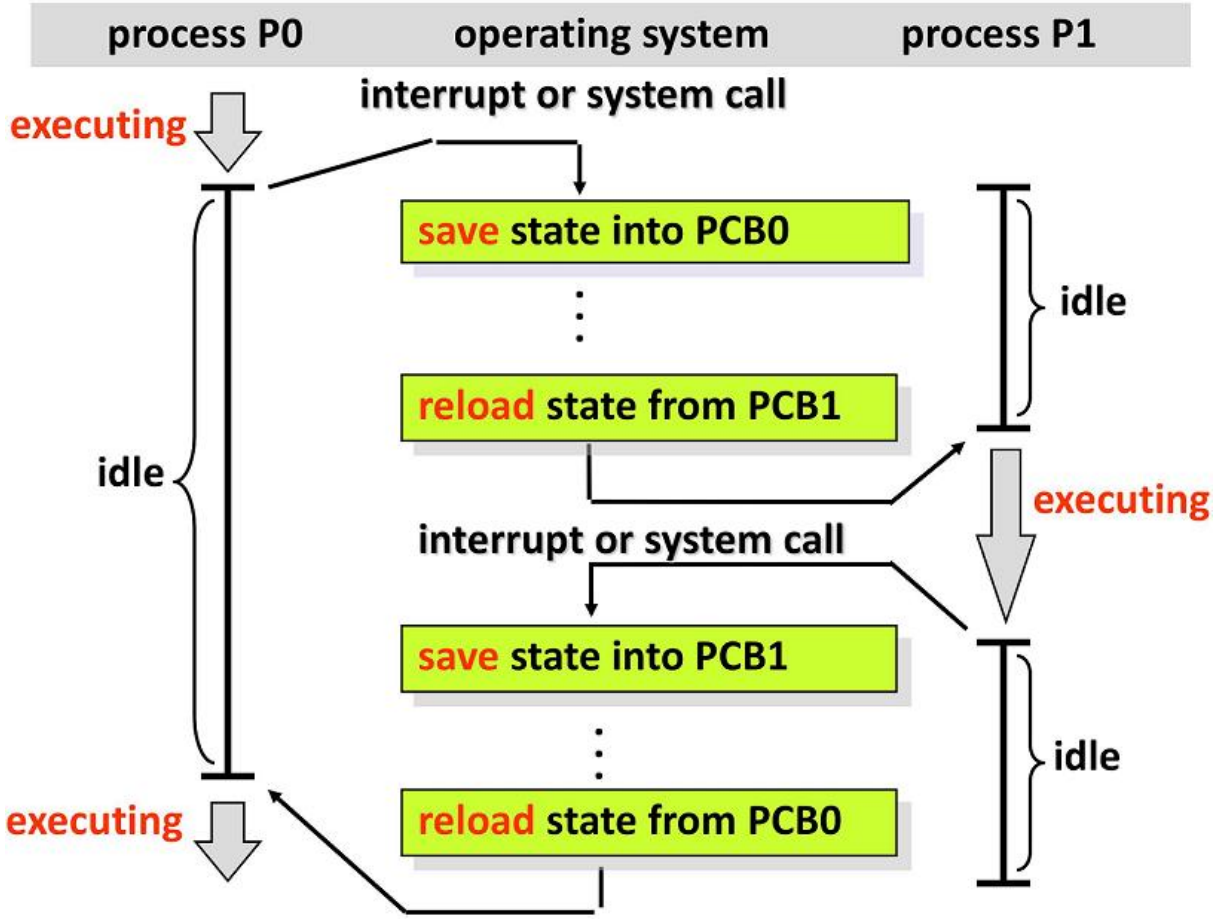
进程队列

- 就绪队列与等待队列示意图



上下文切换

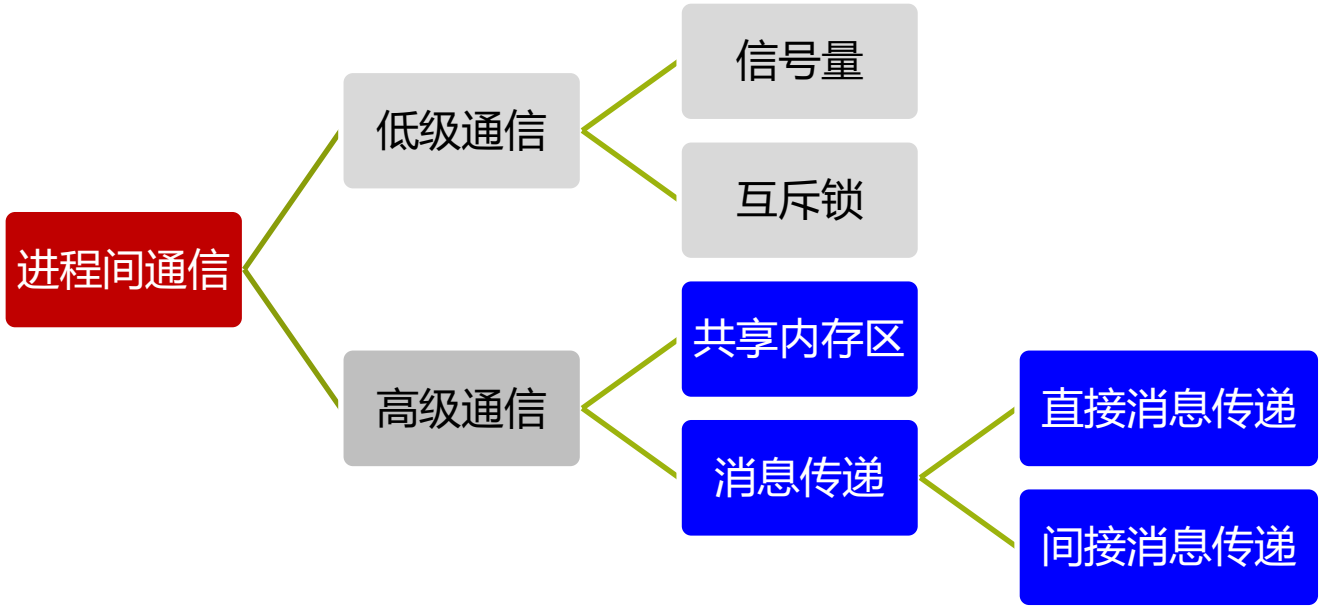
- 当进程获取对CPU控制，以及进程退出对CPU控制时，要保存进程执行的现场（又称**上下文**）



什么是进程间通信

- **为什么需要进程间通信（Inter-Process Communication, IPC）：**
  - 进程之间的关系可能是独立(independent)，也可能是相互协作（cooperating）。
  - 进程间的协作需要互相传递信息，因此需要专门的通信机制支持

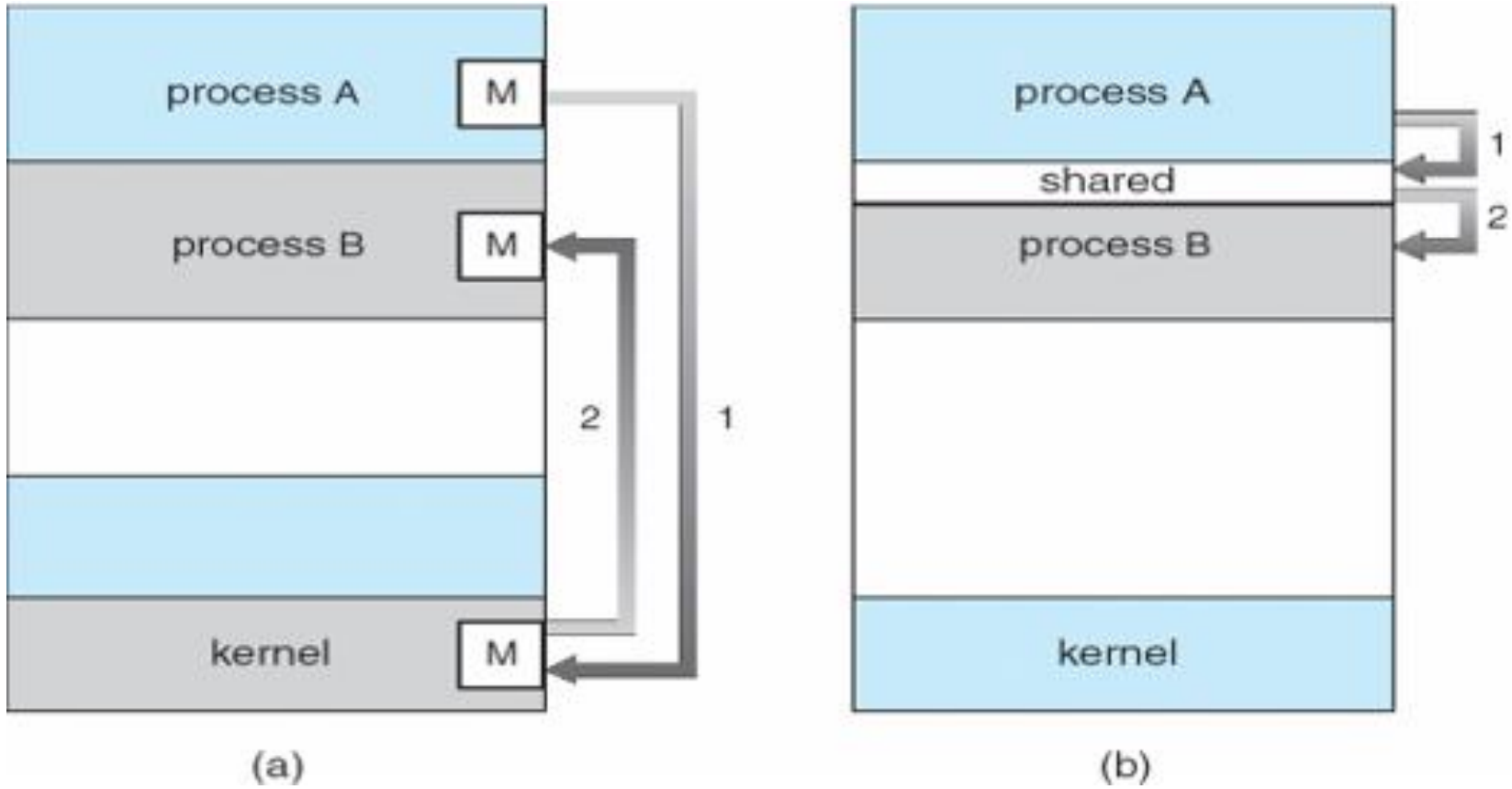
IPC的分类



- 低级通信（Low-Level IPC）：用于进程控制信息的传递，传输信息量相对较小
- 高级通信：主要用于进程间信息的交换与共享，传输信息量相对较大

IPC的分类

- **消息传递与共享内存原理示意**
  - (a) 消息传递　(b) 共享内存

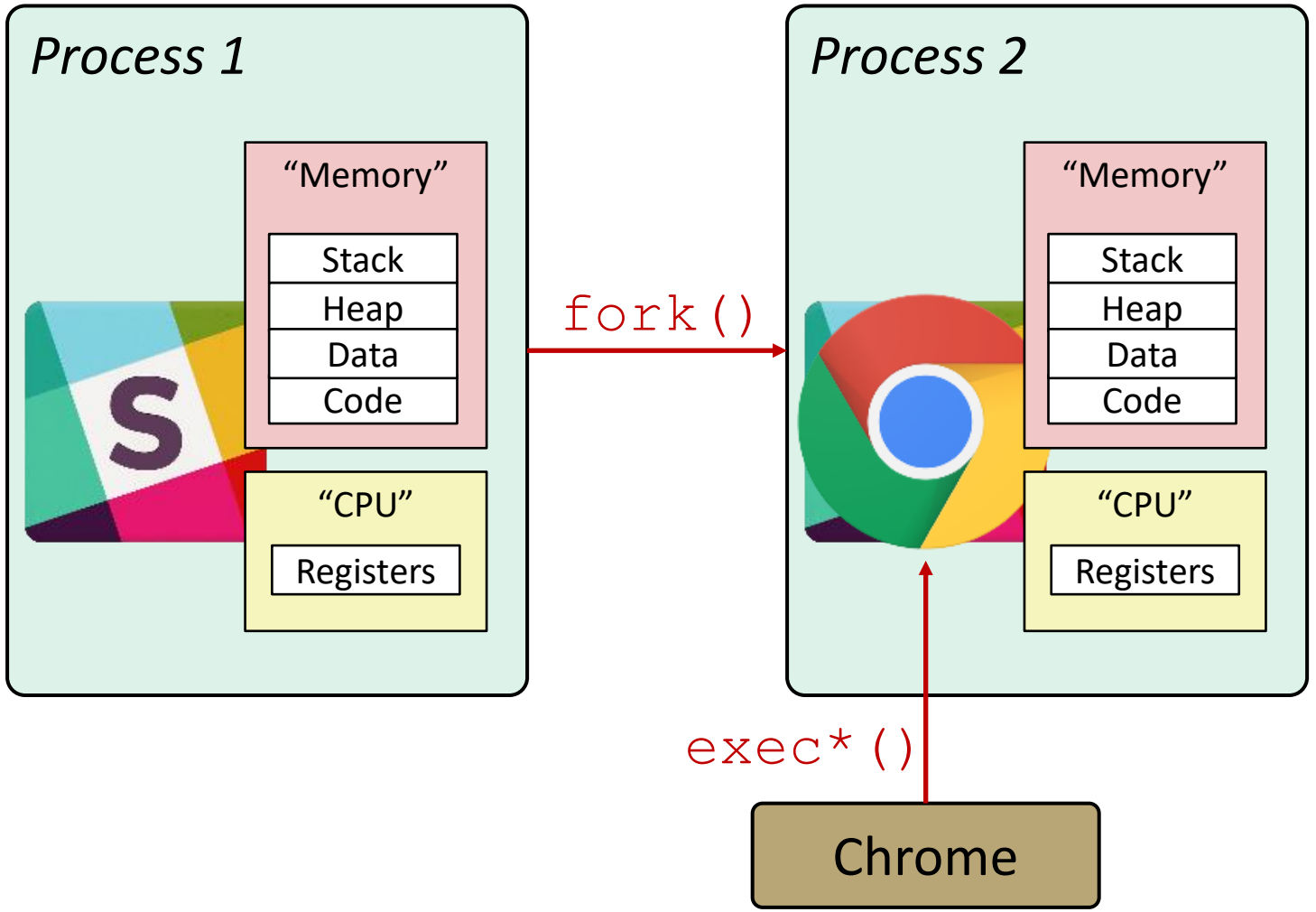


Summary

小结：

- 进程概念
- 进程状态迁移
- 进程调度基础概念
- 进程间通信

进程创建实践


Process 1
"Memory"
Stack
Heap
Data
Code
"CPU"
Registers
fork()
Process 2
"Memory"
Stack
Heap
Data
Code
"CPU"
Registers
exec*()
Chrome

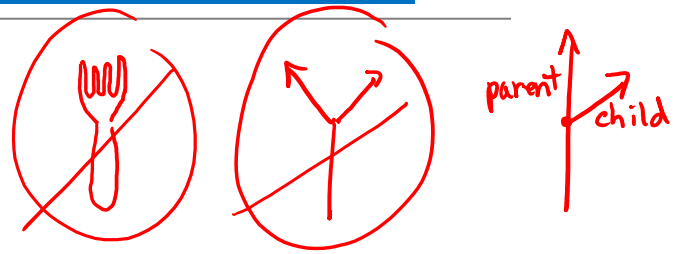
进程创建实践

- fork-exec model (Linux):
  - **`fork()`** creates a copy of the current process
  - **`exec*()`** replaces the current process' code and address space with the code for a different program
    - Family: `execv`, `execl`, `execve`, `execle`, `execvp`, `execlp`
  - `fork()` and `execve()` are *system calls*
- Other system calls for process management:
  - **`getpid()`**
  - **`exit()`**
  - **`wait()`, `waitpid()`**

进程创建实践

returns a PID

```
pid_t fork(void)
```



Returns child's process ID (PID) to the parent process

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

parent gets child's PID
child gets 0

Understanding fork

Process X (parent)

PID X

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

fork

Process Y (child)

PID Y

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Understanding fork

Process X (parent)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Process Y (child)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

PID X

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = Y

PID Y

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

Understanding fork

Process X (parent)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Process Y (child)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();          pid = Y
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();          pid = 0
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

hello from parent

hello from child

Which one appears first?

non-deterministic!

Understanding fork

```
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

splits here

← child only

← parent only

← both

- **该Linux程序的输出结果是什么?**

Fork-Exec

```
// Example arguments: path="/usr/bin/ls",
//     argv[0]="/usr/bin/ls", argv[1]="-ahl", argv[2]=NULL
void fork_exec(char *path, char *argv[]) {
   pid_t pid = fork();
   if (pid != 0) {
      printf("Parent: created a child %d\n", pid);
   } else {
      printf("Child: about to exec a new program\n");
      execv(path, argv);
   }
   printf("This line printed by parent only!\n");
}
```

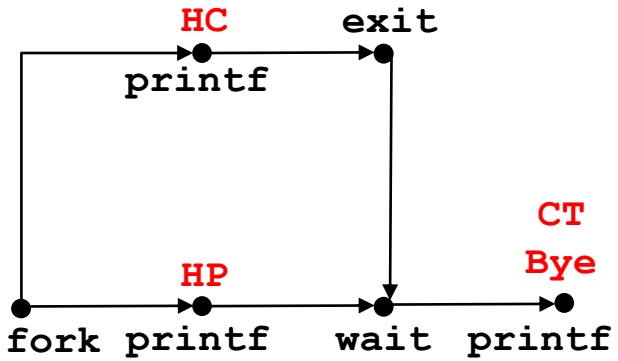
Wait: Synchronizing with Children

```
void fork_wait() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

forks.c

child

parent



Feasible output:

HC HP
HP HC
CT CT
Bye Bye

Infeasible output:

HP
CT
Bye
HC

谢 谢!
Thank you!