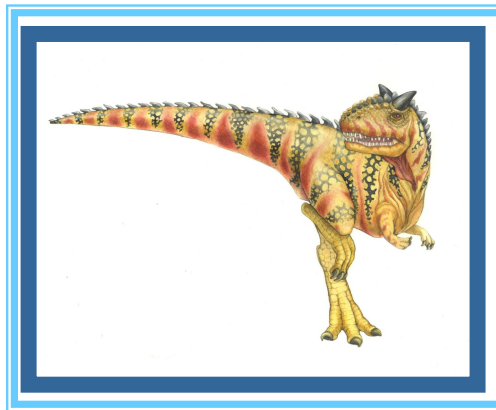


# Chapter 3: Processes

---





# Chapter 3: Processes

---

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems





# Objectives

---

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To describe Interprocesses communication





# Process Concept

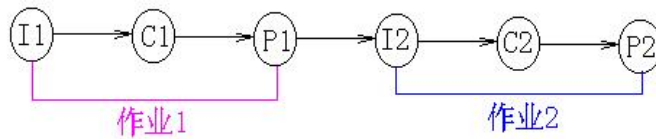
---

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Allow multiple programs to be loaded into memory and to be executed concurrently.

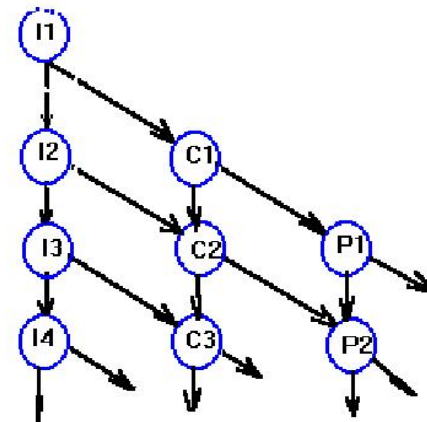




# Process Concept



**Sequential execution**



**Concurrent execution**





# Process Concept

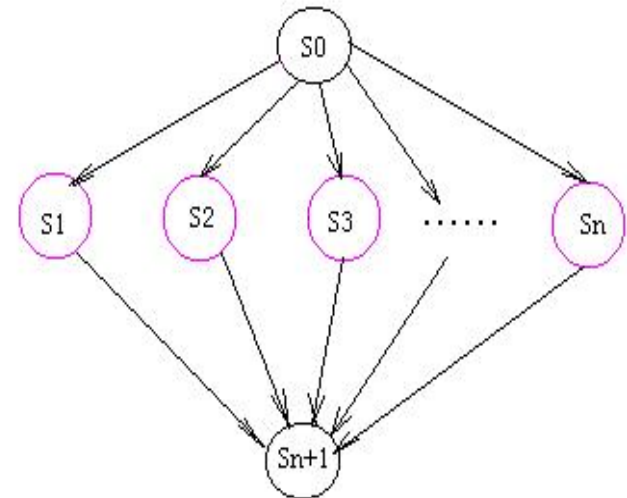
- Description for concurrent execution by Dijkstra:

Cobegin

$S_1; S_2; \dots; S_N$

Coend;

- $S_i (i=1, 2, 3, \dots, n)$
- Co:concurrent





# Process Concept

- 程序并发执行的特点:
- 1、中断性:程序的并发执行因竞争资源, 而呈现出“执行—暂停—执行”的间断性活动规律。 例如: 2个并发程序都要使用打印机, 其中1个就要等待。
- 2、失去封闭性:程序执行的结果不仅依赖于程序的初始条件, 还依赖于程序执行时的相对速度
- 3、不可再现性:由于失去封闭性, 初始条件相同, 但是结果不一定相同。
- 例如: 观察者/报告者, 有两个循环程序A和B, 它们共享一个变量N。程序A每执行一次时都要做 $N := N + 1$ 操作; 程序B每执行一次时, 都要做`print (N)`操作, 然后再将N置成“0”, 程序A和B以不同的速度运行。可能出现多报或漏报。(假定某时刻变量N的值为n)





# Process Concept

---

- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- A program is a passive entity; a process is an active entity.

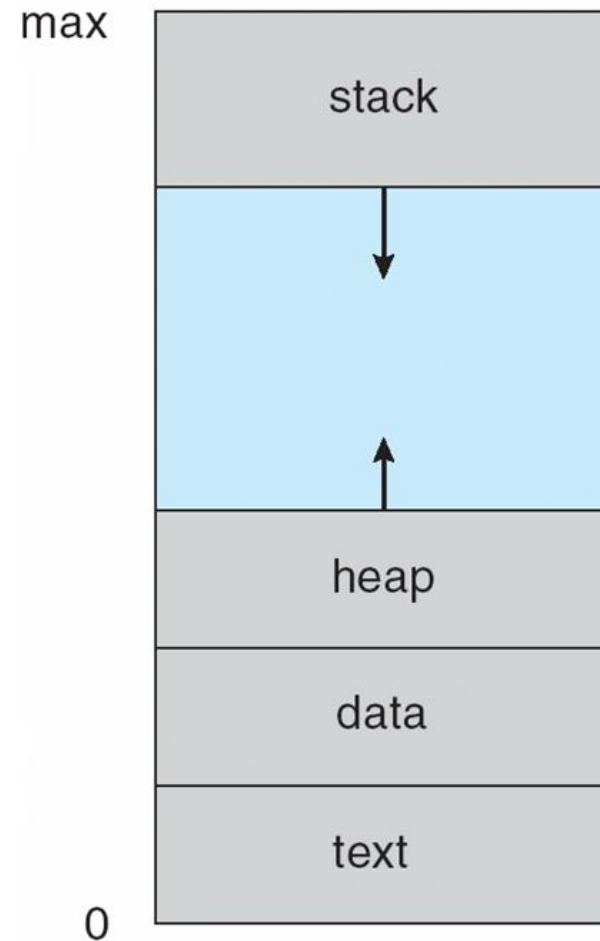






# Process Concept

- A process includes:
  - program counter
  - stack
  - data section



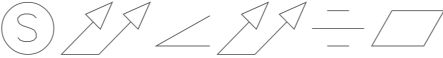


# Process Concept

在linux写一个程序，里面全局变量，局部变量，static，const，常数，数组，输出这些变量的地址：

```
[root@localhost 1018]# ./a.out
未赋值的全局变量的地址:0x80497bc
赋值的全局变量的地址: 0x80497a8
static修饰的全局变量的地址: 0x80497ac
const修饰的全局变量的地址: 0x8048540
数组的首地址: 0xbfd90564
未赋值的局部变量的地址: 0xbfd90544
赋值的局部变量的地址: 0xbfd90540
static修饰的局部变量的地址: 0x80497b0
const修饰的局部变量的地址: 0xbfd9053c
```

可以看出全局变量和

⑤  修饰的变量在同一个存储位置，局部变量有自己的存储位置，数组也是有自己的存储位置。





# Process Concept

---

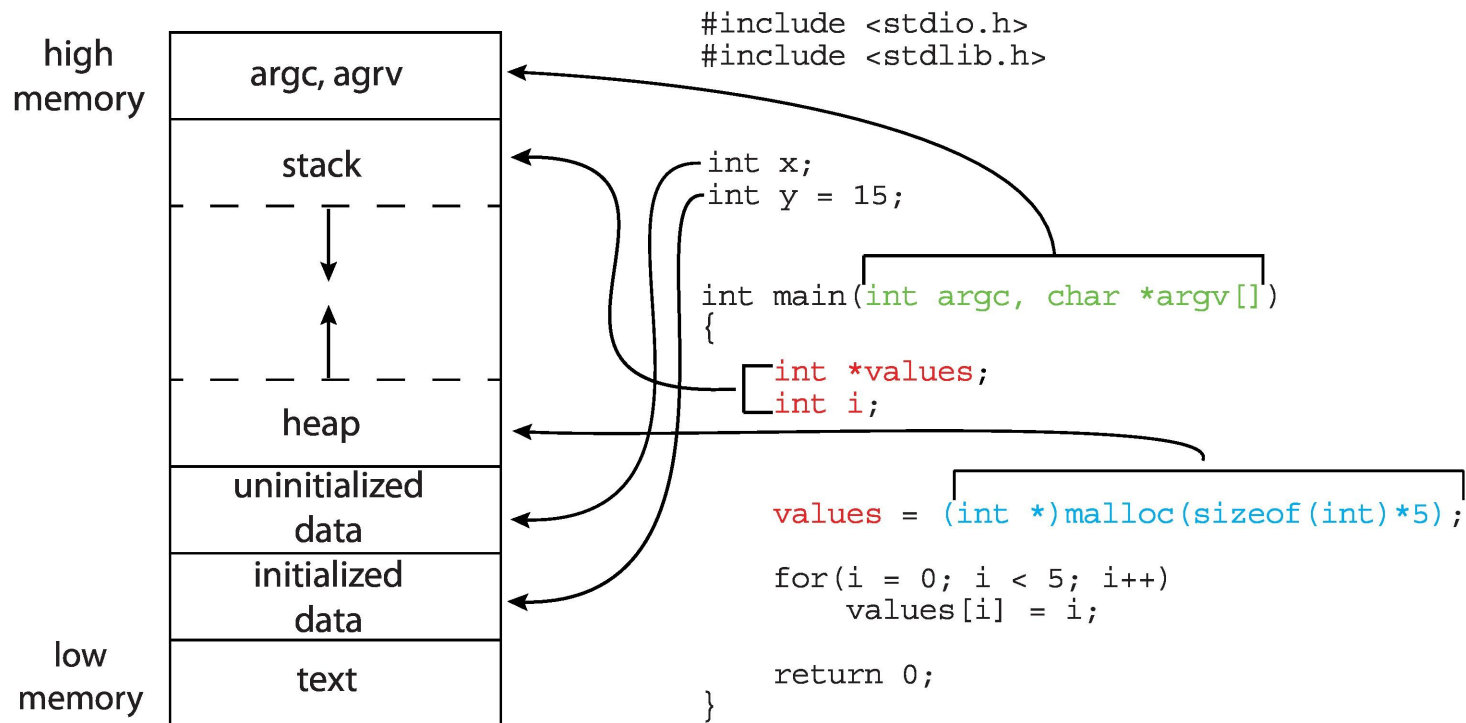
栈（操作系统）：由操作系统自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

堆（操作系统）：一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收，分配方式倒是类似于链表。





# Memory Layout of a C Program





# Process Concept

---

- 进程是动态的，程序是静态的：程序是有序代码的集合；进程是程序的执行。
- 进程是暂时的，程序是永久的：进程是一个状态变化的过程，程序可长久保存。
- 进程与程序的组成不同：进程的组成包括程序、数据和进程控制块（即进程状态信息）。
- 进程与程序的对应关系：通过多次执行，一个程序可对应多个进程；通过调用关系，一个进程可对应多个程序。





# Process Concept

- 结构特征：进程实体=程序段+相关的数据段+PCB。
- 动态性：进程的实质是进程实体的一次执行过程，因此动态性是进程的最基本的特征。
- 并发性：多个进程实体同存在于内存中，且能在一段时间内同时运行。是最重要的特征。
- 独立性：进程实体是一个能独立运行、独立分配资源和独立接受调度的基本单位。
- 异步性：进程按各自独立的、不可预知的速度向前推进。





# Process Concept

---

- Two types process
  - System process
  - User process





# Process Concept

- 系统进程与用户进程的区别：
- 资源的使用:系统进程被分配一个初始的资源集合，这些资源可以为它独占，也能以最高优先权的资格使用。用户进程通过系统服务请求的手段竞争使用系统资源；
- I/O操作:用户进程不能直接做I/O操作，而系统进程可以做显式的、直接的I/O操作。
- CPU工作状态:系统进程在管态下活动，而用户进程则在用户态下活动。







# Process Concept

---

- A process is related to:
  - PCB
  - A executable file
  - Some state
  - A queue (ready, run, waiting)
  - Some resources (memory, file, CPU, device)





# Process State

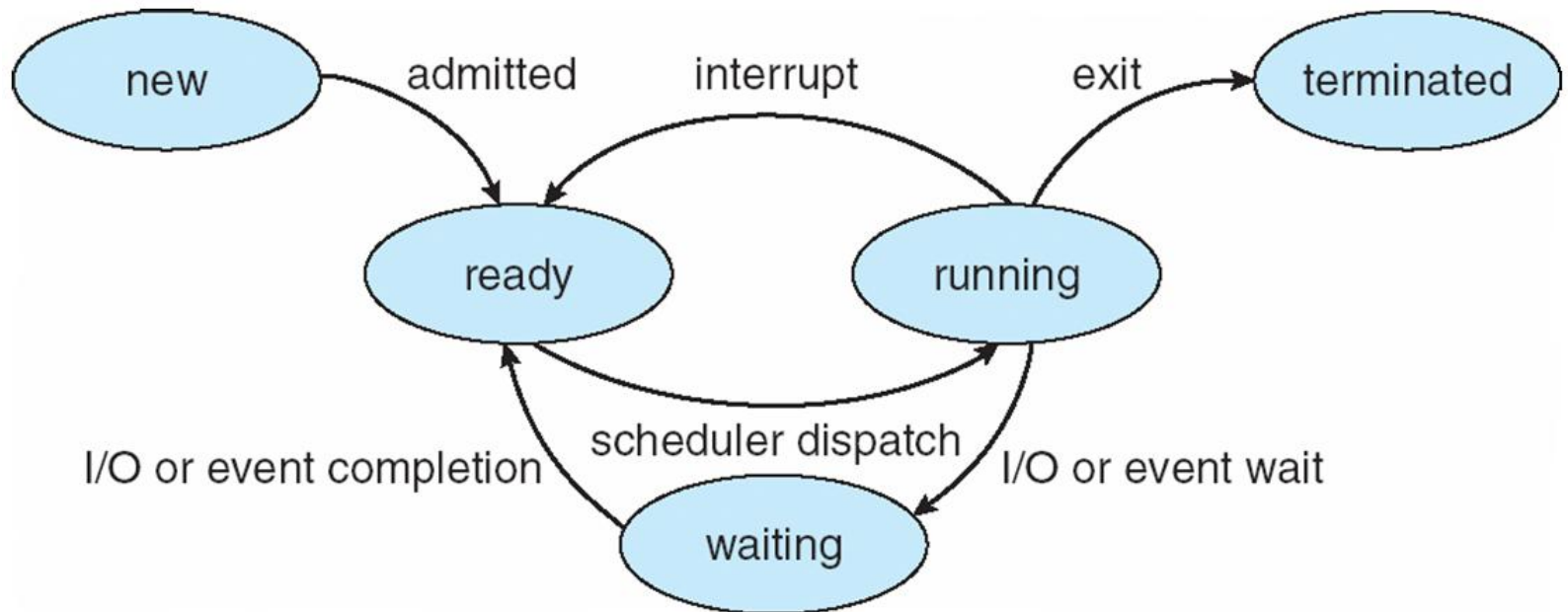
---

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution





# Diagram of Process State





# Process Control Block (PCB)

---

Information associated with each process

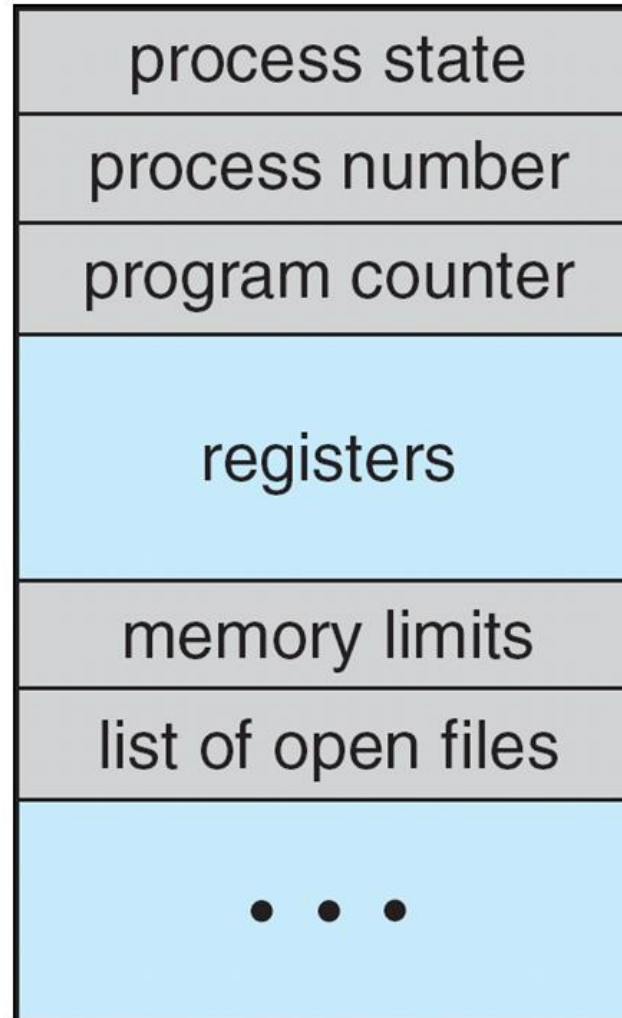
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information





# Process Control Block (PCB)

---





# Process Control Block (PCB)

---

- PCB in Linux
- `task_struct`: PCB of a process
- Task array:
  - Saves all pointers of `task_struct`
  - default size is 512





# Process Control Block (PCB)

---

- Information in task\_struct of Linux
  - Id: process id, user id, group id
  - State:
  - Scheduling information:
  - Family information: father, son
  - Communication information
  - Timer
  - file opened
  - Context





# Process Control Block (PCB)

---

- Process Scheduling Information in task\_struct of Linux
  - Policy: realtime or normal
  - Priority : the whole time a process can execution
  - Rt\_priority : the relative priority of a realtime process
  - Counter: time a process can execution







# Process Scheduling Queues

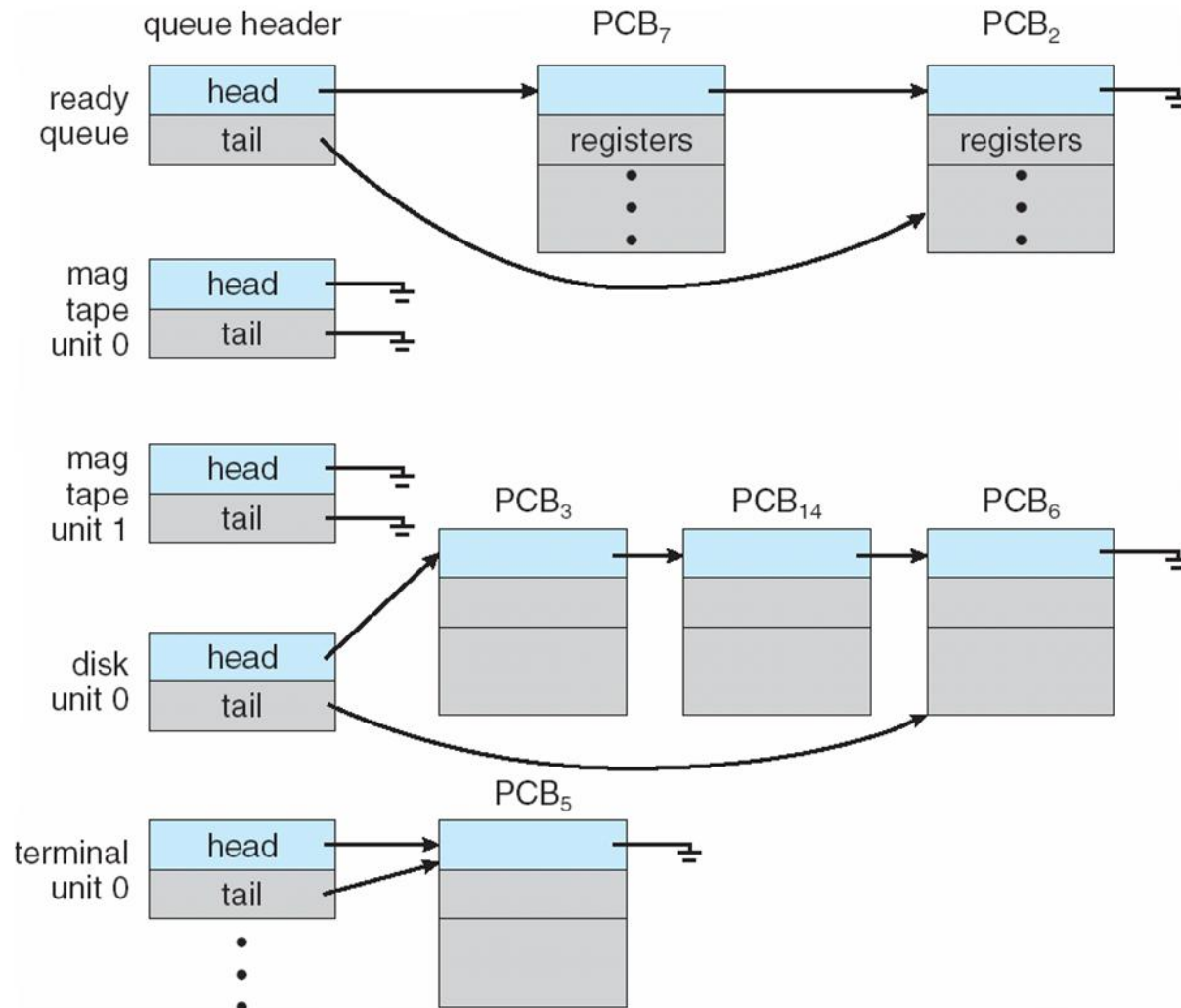
---

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues



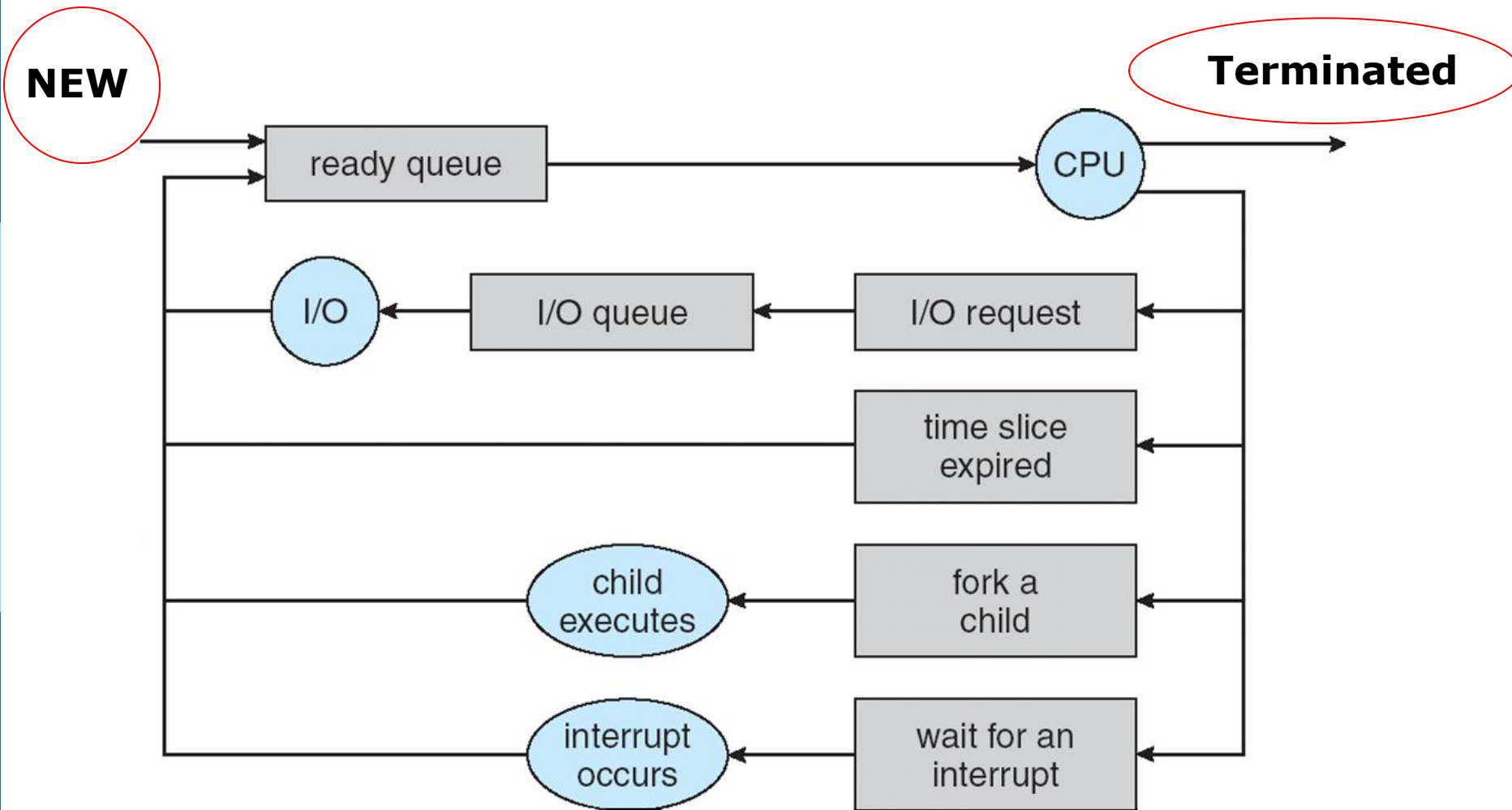


# Ready Queue And Various I/O Device Queues





# Representation of Process Scheduling





# Schedulers

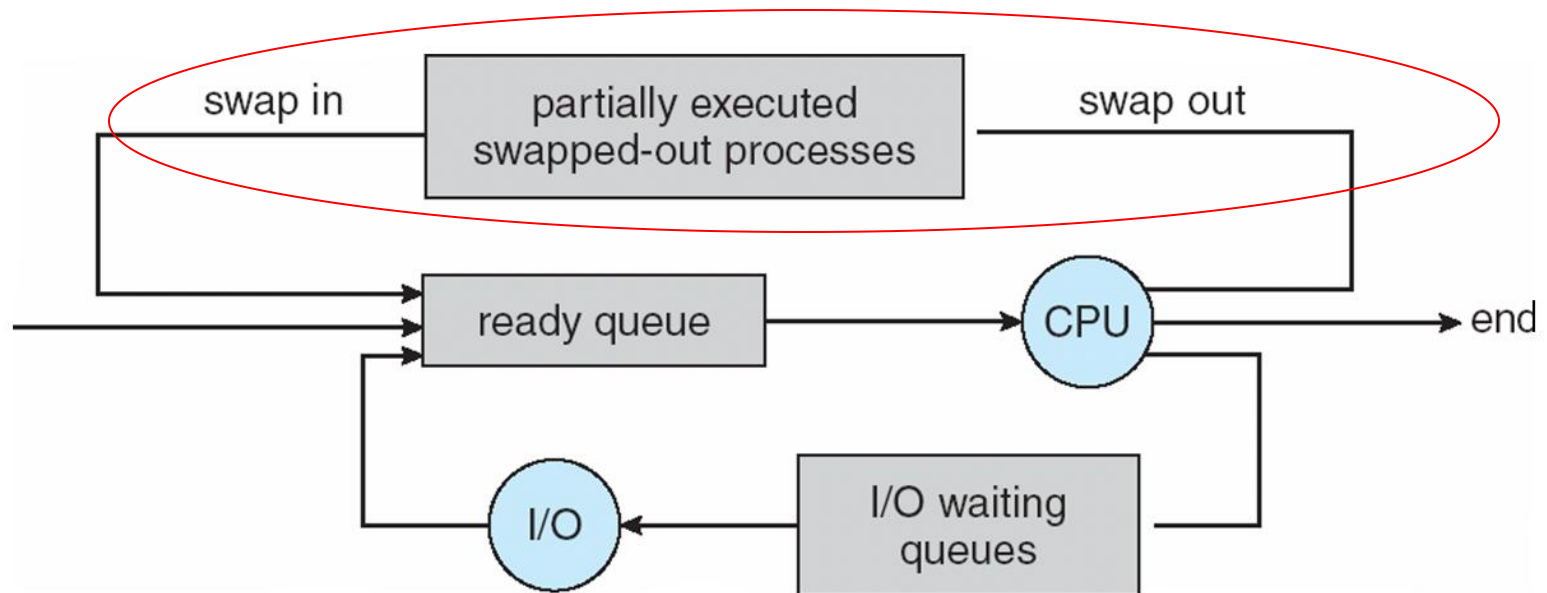
---

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the **ready queue (memory)**
- **Short-term scheduler** (or CPU scheduler) – selects which process should be **executed** next and allocates **CPU**
- Medium-term scheduling





# Addition of Medium Term Scheduling





# Schedulers (Cont)

---

- Short-term scheduler is invoked very frequently (**milliseconds**)  $\Rightarrow$  (must be fast)
- Long-term scheduler is invoked very infrequently (**seconds, minutes**)  $\Rightarrow$  (may be slow)
- The long-term scheduler controls the *degree* of *multiprogramming*





## Schedulers (Cont)

---

- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler controls the process **mix** of I/O-bound process and CPU-bound process





# Context Switch

---

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a **context switch**
- **Context** of a process represented in the PCB, it includes:
  - The values of CPU **registers**
  - The **process state**
  - **Memory-management** information







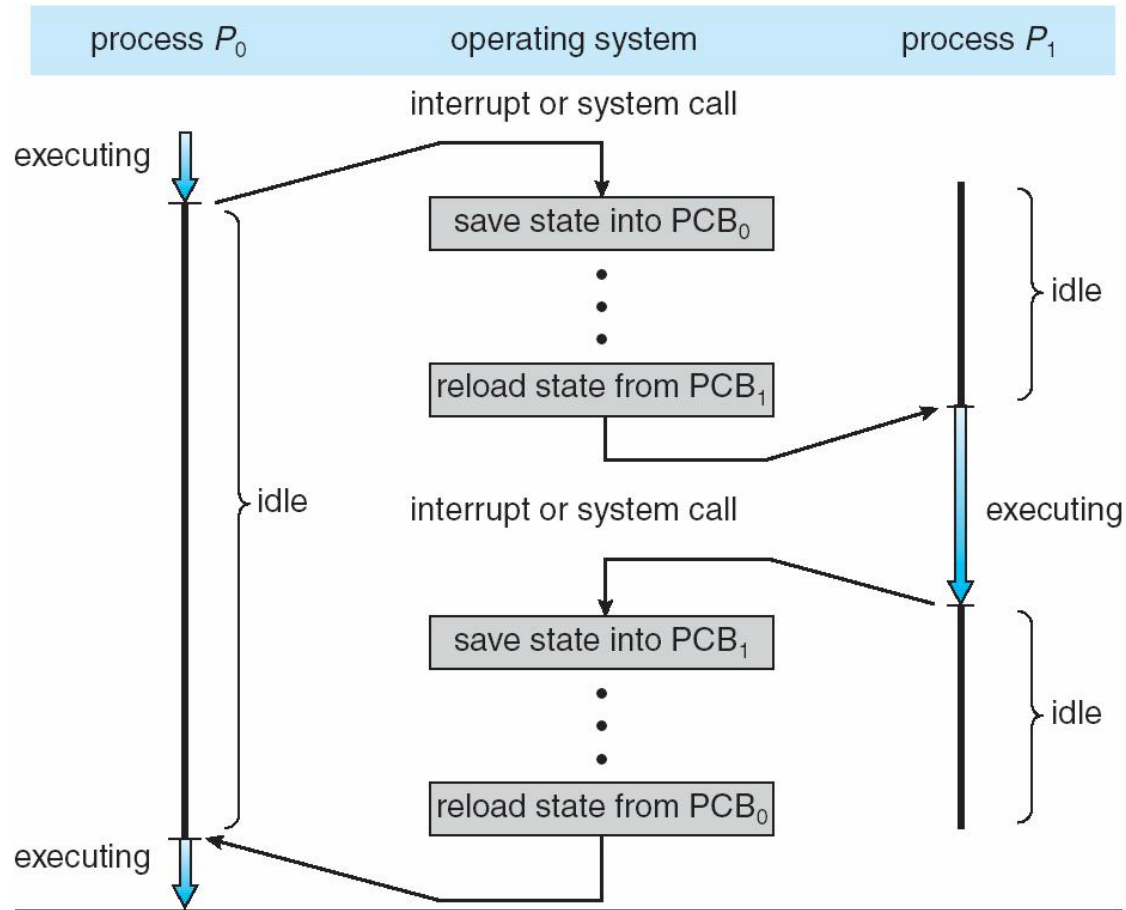
# Context Switch

- context由进程的用户地址空间内容、硬件寄存器内容及与该进程相关的核心数据结构组成
  - 用户级上下文：进程的用户地址空间（包括用户栈各层次），包括用户正文段、用户数据段和用户栈
  - 寄存器级上下文：程序计数器、程序状态寄存器、栈指针、通用寄存器的值
  - 系统级上下文：
    - 静态部分（PCB和资源表格）
    - 动态部分：核心栈（核心过程的栈结构，不同进程在调用相同核心过程时有不同核心栈）





# CPU Switch From Process to Process





# Context Switch

---

- Context-switch time is **overhead**; the system does no useful work while switching
- Time dependent on hardware support
  - Memory speed
  - The number of registers that must be copied
    - ▶ E.g. Some processors ( such as the Sun UltraSPARC) provides multiple sets of registers, a context switch simply includes changing the pointer to the current register sets
  - Typical speed: 1-1000ms





# Context Switch

---

- Process context switch in Linux:
  - Saving user data
    - ▶ Code, data, stack, shared memory
  - Saving Register data
    - ▶ PC\PSW\SP(Stack Pointer)\PCBP(PCB指针)\ISP(中断栈指针)
  - Saving memory management information
    - ▶ Virtual memory management data, e.g. Page table





# Question 1

- 下列各项步骤中，哪一个不是创建进程所必须的步骤？
  - A. 建立一个进程控制块PCB
  - B. 由CPU调度程序为进程调度CPU
  - C. 为进程分配内存等必要的资源
  - D. 将PCB链入进程就绪队列
- ✓ **解析：**进程创建的过程包括：①申请空白PCB；②为新进程分配资源；③初始化进程控制块；④如果进程就绪队列能够接纳新进程，便将新进程插入就绪队列。





## Question 2

● 一个进程被唤醒意味着\_\_\_\_\_？

- A. 该进程重新占有了CPU
- B. 它的优先权变为最大
- C. 其PCB移至等待队列队首
- D. 进程变为就绪状态

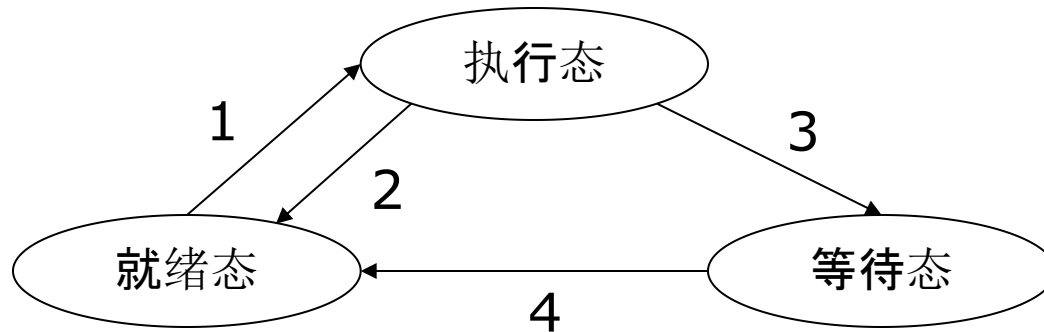
✓ **解析：**一个进程被唤醒意味着进程由阻塞态转换为就绪态，所以D选项正确。进程重新占有了CPU是从就绪态转换为执行态，所以A选项是错误的。进程被唤醒并不意味着优先权一定变为最大，所以B选项不正确。





## Question 3

- 某系统的进程状态转换图如图所示，请回答：



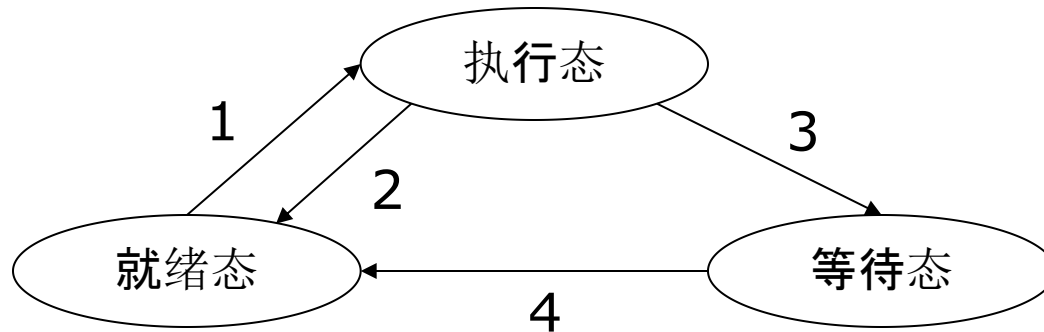
- 引起各种状态转换的典型事件有哪些？
- 当我们观察系统中某系进程时，能够看到某一进程产生的一次状态转换能引起另一个进程做一次状态转换。在什么情况下，当一个进程发生转换3时，能立即引起另一进程发生转换1？
- 试说明是否会发生这些因果转换：2->1; 3->2; 4->1。





## Question 3

- 某系统的进程状态转换图如图所示，请回答：



- 引起各种状态转换的典型事件有哪些？

- ✓ 在本题所给的进程状态转换图中，存在4种状态转换。当进程调度程序从就绪队列中选取一个进程投入运行时引起转换1；正在执行的进程如因时间片用完而被暂停执行就会引起转换2；正在执行的进程因等待的事件尚未发生而无法执行（如进程请求执行I/O）则会引起转换3；当进程等待的事件发生时（I/O）则会引起转换4。

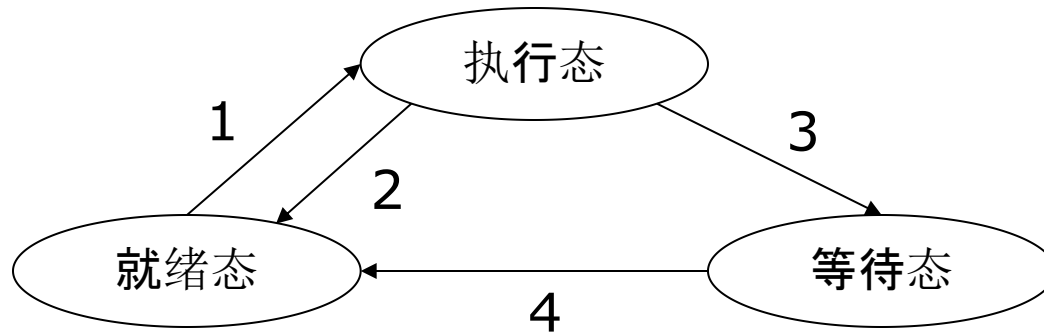






## Question 3

- 某系统的进程状态转换图如图所示，请回答：



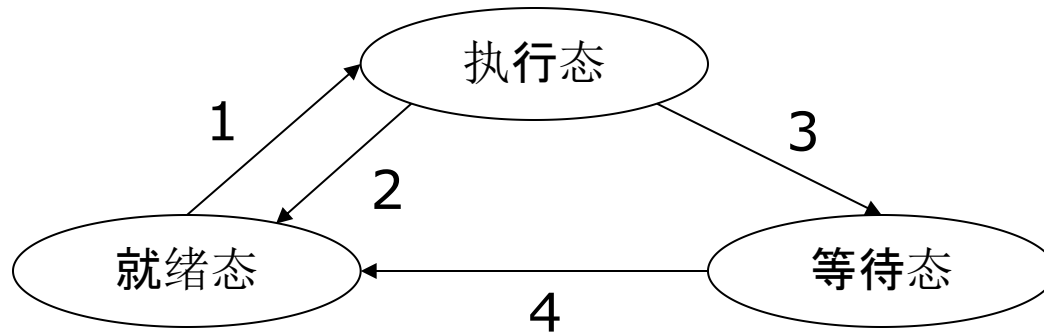
- 当我们观察系统中某系进程时，能够看到某一进程产生的一次状态转换能引起另一个进程做一次状态转换。在什么情况下，当一个进程发生转换3时，能立即引起另一进程发生转换1？
- ✓ 如果就绪队列非空，则一个进程的转换3会立即引起另一个进程的转换1。这是因为一个进程发生转换3意味着正在执行的进程由执行状态变为阻塞状态，这时处理机空闲，进程调度程序必然会从就绪队列中选取一个进程并将它投入运行，因此只要就绪队列非空，一个进程的转换3能立即引起另一个进程的转换1。



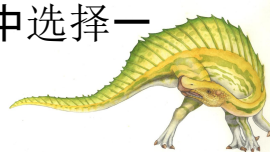


## Question 3

- 某系统的进程状态转换图如图所示，请回答：



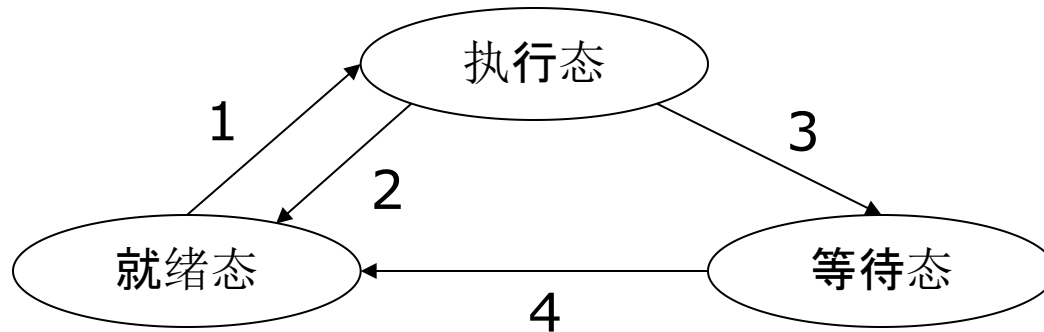
- 试说明是否会发生这些因果转换：2->1; 3->2; 4->1。
- ✓ 所谓因果转换指的是有两个转换，**一个转换的发生会引起另一个转换的发生**，前一个转换成为因，后一个转换称为果，这两个转换称为因果转换。当然，这种因果关系并不是什么时候都能发生，而是在一定条件下才会发生。
- ✓ 2->1: 当某进程发生转换2时，就必然引起另一进程的转换1。因为当发生转换2时，正在执行的进程从执行状态变为就绪状态，进程调度程序必然会从就绪队列中选择一个进程投入运行，即发生转换1。





## Question 3

- 某系统的进程状态转换图如图所示，请回答：



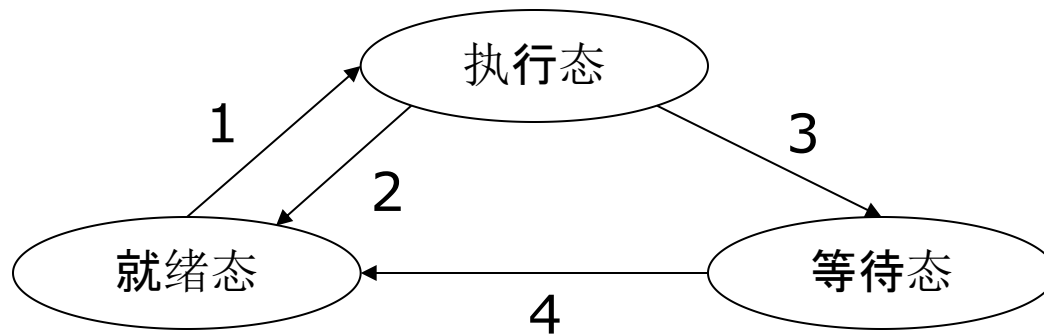
- 试说明是否会发生这些因果转换：2->1; 3->2; 4->1。
- ✓ 所谓因果转换指的是有两个转换，一个转换的发生会引起另一个转换的发生，前一个转换成为因，后一个转换称为果，这两个转换称为因果转换。当然，这种因果关系并不是什么时候都能发生，而是在一定条件下才会发生。
- ✓ 3->2: 某个进程的转换3绝不可能引起另一个进程发生转换2。这是因为当前执行进程从执行状态变为阻塞状态，不可能又从执行状态变为就绪状态。





## Question 3

- 某系统的进程状态转换图如图所示，请回答：



- 试说明是否会发生这些因果转换：2->1; 3->2; 4->1。
- ✓ 所谓因果转换指的是有两个转换，一个转换的发生会引起另一个转换的发生，前一个转换成为因，后一个转换称为果，这两个转换称为因果转换。当然，这种因果关系并不是什么时候都能发生，而是在一定条件下才会发生。
- ✓ 4->1: 当处理机空闲且就绪队列为空时，某一进程的转换4就会引起该进程的转换1。因为**此时处理机空闲**，一旦某个进程发生转换4，就意味着有一个进程从阻塞状态变为就绪状态，因而调度程序就会将就绪队列中的此进程投入运行。





# Operations on processes

---

- The processes in the system can execute **concurrently**, and they must be created and deleted **dynamically**.
- The operating system must provide a mechanism for process creation and termination.
  - The mechanism is process control.





# Operations on processes

---

- Process creation
- Process Termination
- Process blocking
- Process wakeup
- Process suspend
- Process activate





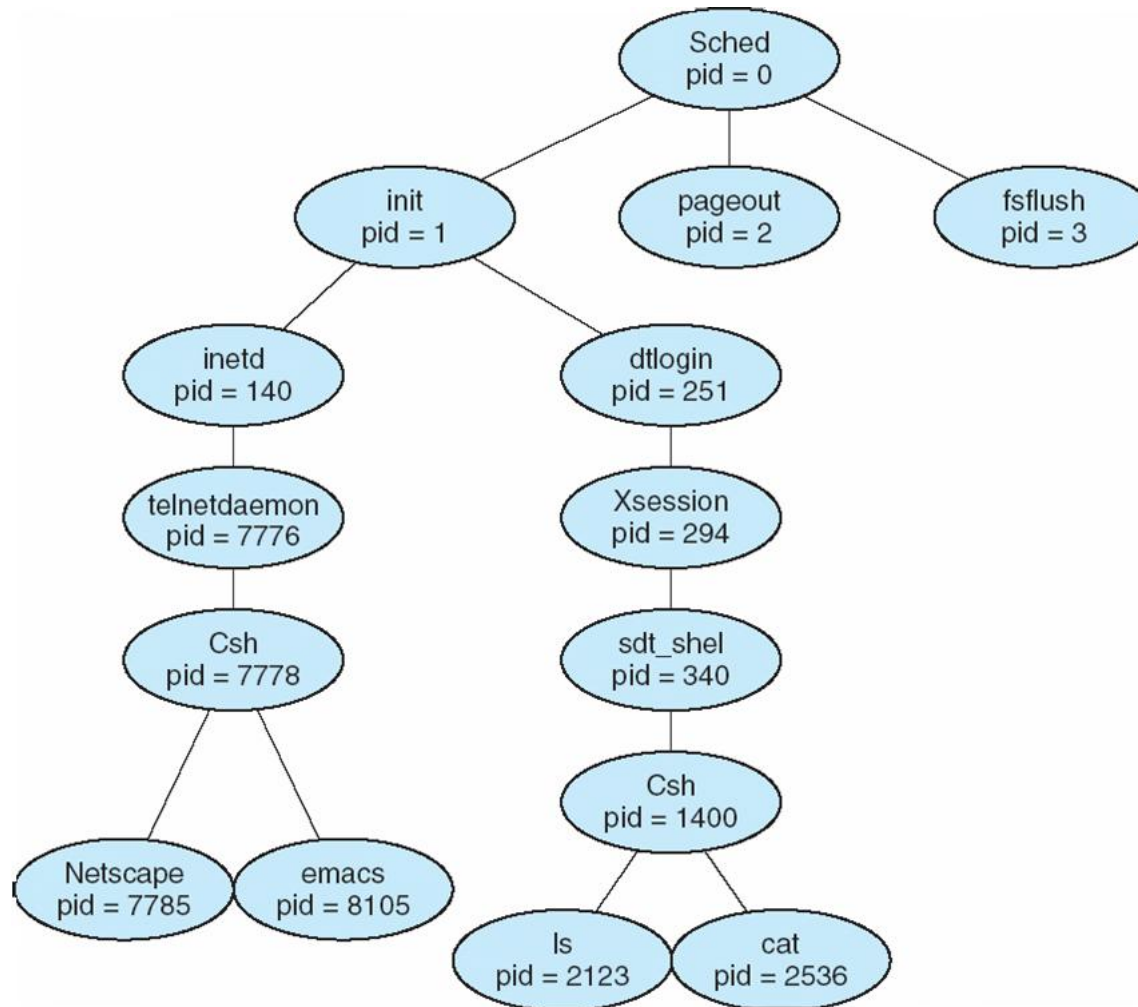
# Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing
  - Parent and children **share all** resources (嵌入式系统)
  - Children **share subset** of parent's resources (Unix系统)
  - Parent and child **share no** resources (Windows系统)
- Execution
  - Parent and children execute **concurrently**
  - Parent **waits** until children terminate





# A tree of processes on a typical Solaris







# Process Creation (Cont)

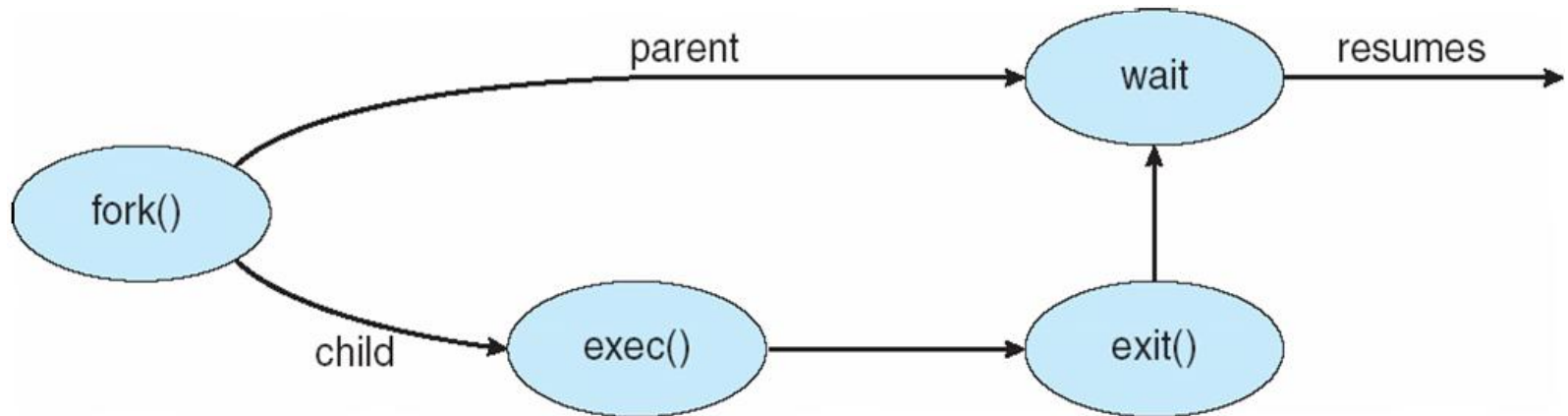
- Address space (共享/独立)
  - Child **duplicate** of parent (Unix)
  - Child has **a program loaded** into it (Windows)
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to **replace** the process' **memory space** with a new program
- Windows NT
  - Supports both models: the parent's address space may be duplicated, or the parent may specify the name of a program for the OS to load into the address of the new process



[illegible]



# Process Creation



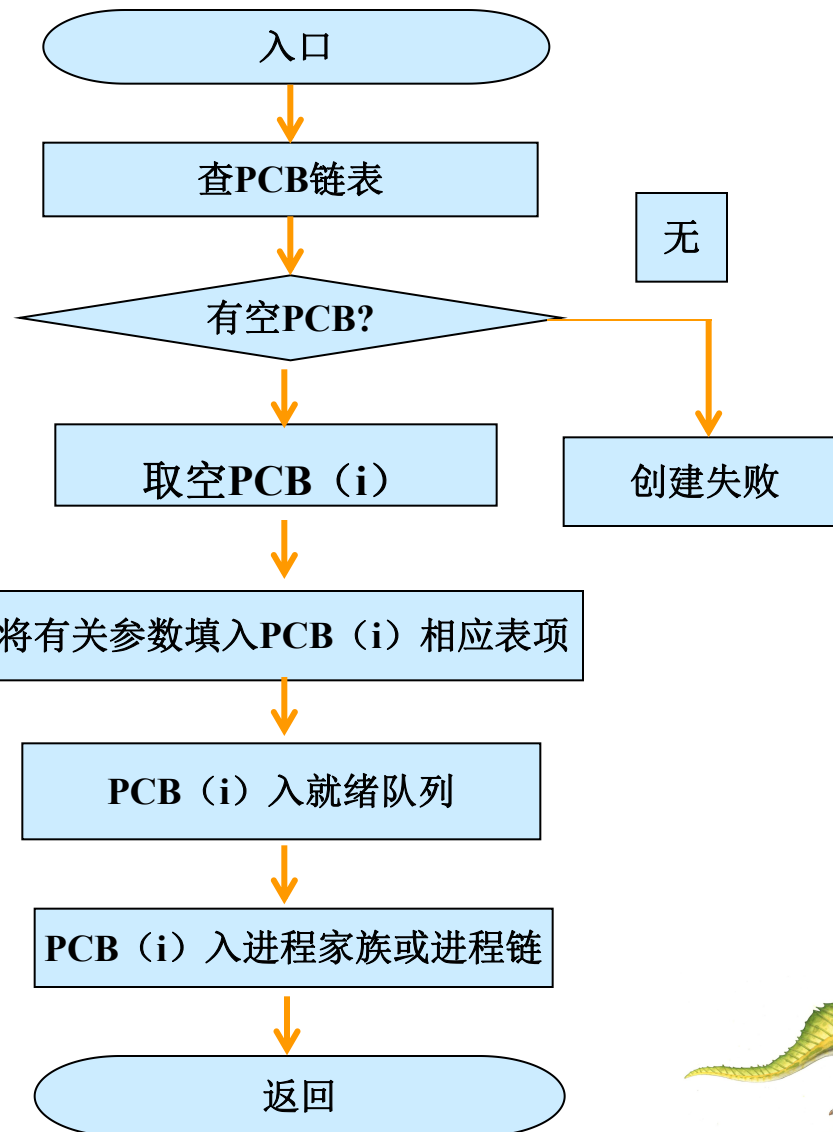


# Process creation

## ■ UNIX

`pid=fork();`

从系统调用fork返回时，CPU在父进程中时，pid值为所创建子进程的进程号(>0)，若在子进程中时，pid的值为零。





# Process Creation

$\Rightarrow \emptyset \nearrow \nearrow \quad \textcircled{M} \angle \Rightarrow \emptyset \quad ( \quad )$   
 $\textcircled{E}$

$\vee \nearrow \Rightarrow \textcircled{L} \bigcirc \quad (1)$   
 $\textcircled{E}$

$// \square \odot \frown ( \quad );$

$\textcircled{R}$

$\textcircled{R}$





# Process Creation

$\overline{\overline{\emptyset}} \nearrow \nearrow \quad \textcircled{M} \angle \overline{\overline{\emptyset}} \quad (\sqcup \square \overline{\overline{\cap}})$

$\textcircled{E}$

$\overline{\overline{\emptyset}} \nearrow \nearrow \quad \overline{\overline{\phantom{x}}};$

$// \square \odot \quad (\overline{\overline{\phantom{x}}} = 0; \overline{\overline{\phantom{x}}} \circ 3; \overline{\overline{\phantom{x}}} ++)$

$// \square \odot \cap ( \quad );$

$\textcircled{S} \textcircled{L} \bigcirc \bigcirc \textcircled{P}(3);$

$\textcircled{R}$





# C Program Forking Separate Process

```

≡∅ ↗ □ ∠ (L) — ○ ;
≡∅ ↗ (M) ∠ ≡∅ ( ) (E)
      (P) ≡ ∩ ↗ (P) ≡ ∩ ;
(P) ≡ ∩ = // □ ○ ∩ ( ) ;
≡ // (P) ≡ ∩ ↻ 0 (E) / *
○ ○ ○ □ ○ □ ▱ ▱ — ○ ○ ○ ∩ * /
      // (P) ○ ≡ ∅ ↗ // (S) ↗ ∩ ○ ○ ○ ,
" F □ ○ ∩ F ∠ ≡ (L) ○ ∩ " ) ;
      ○ ∇ ≡ ↗ (-1) ;

```

```

(R)
○ (L) (S) ○ ≡ // (P) ≡ ∩ ≡ ≡ 0 (E)
/ * ▱ ↗ ≡ (L) ∩ (P) ○ □ ▱ ○ (S) (S) * /
      // □ ○ ( ; ; ) (E)

```

```

(P) ○ ≡ ∅ ↗ // (" | ∅ ▱ ↗ ≡ (L) ∩ ,
□ ∠ (L) — ○ ≡ (S) → ∩ \ ∅ " ,
□ ∠ (L) — ○ + + ) ;
      (S) (L) ○ ○ (P) (2) ;

```

(R)

```

(R)
○ (L) (S) ○ (E) / * (P) ∠ ○ ○ ∅ ↗
(P) ○ □ ▱ ○ (S) (S) * / ∠ ○ ○ ∅ ↗
      // □ ○ ( ; ; ) (E)

```





# C Program Forking Separate Process

```

≡∅ ↗ □ ∠ (L) — ○;
≡∅ ↗ (M) ∠ ≡∅ ( ) (E)
      (P) ≡ ∩ ↗ (P) ≡ ∩;
(P) ≡ ∩ = // □ ○ ∩ ( );
≡ // (P) ≡ ∩ ↻ 0 (E) / *
○ ○ ○ □ ○ □ ▯ ▯ — ○ ○ ○ ∩ * /
      // (P) ○ ≡ ∅ ↗ // (S) ↗ ∩ ○ ○ ○,
" F □ ○ ∩ F ∠ ≡ (L) ○ ∩ );
      ○ ∇ ≡ ↗ (-1);

```

```

(R)
○ (L) (S) ○ ≡ // (P) ≡ ∩ ≡ ≡ 0 (E)
/ * ▯ ↗ ≡ (L) ∩ (P) ○ □ ▯ ○ (S) (S) * /
      // □ ○ ( ; ; ) (E)

```

```

(P) ○ ≡ ∅ ↗ // (" | ∅ ▯ ↗ ≡ (L) ∩,
□ ∠ (L) — ○ ≡ (S) → (P) \ ∅ ",
> < □ ∠ (L) — ○);
      (S) (L) ○ ○ (P) (2) ;

```

(R)

```

(R)
○ (L) (S) ○ (E) / * (P) ∠ ○ ○ ∅ ↗
(P) ○ □ ▯ ○ (S) (S) * / ∠ ○ ○ ∅ ↗
      // □ ○ ( ; ; ) (E)

```







# Process Creation

---

当父进程调用fork()创建子进程之后，下列哪些变量在子进程中修改之后，父进程里也会相应地作出改动？

- A.全局变量
- B.局部变量
- C.静态变量
- D.文件指针

答案：D

在fork创建多进程之后堆栈信息会完全复制给子进程内存空间，父子进程相互独立。文件描述符存在于系统中为所有进程所共享





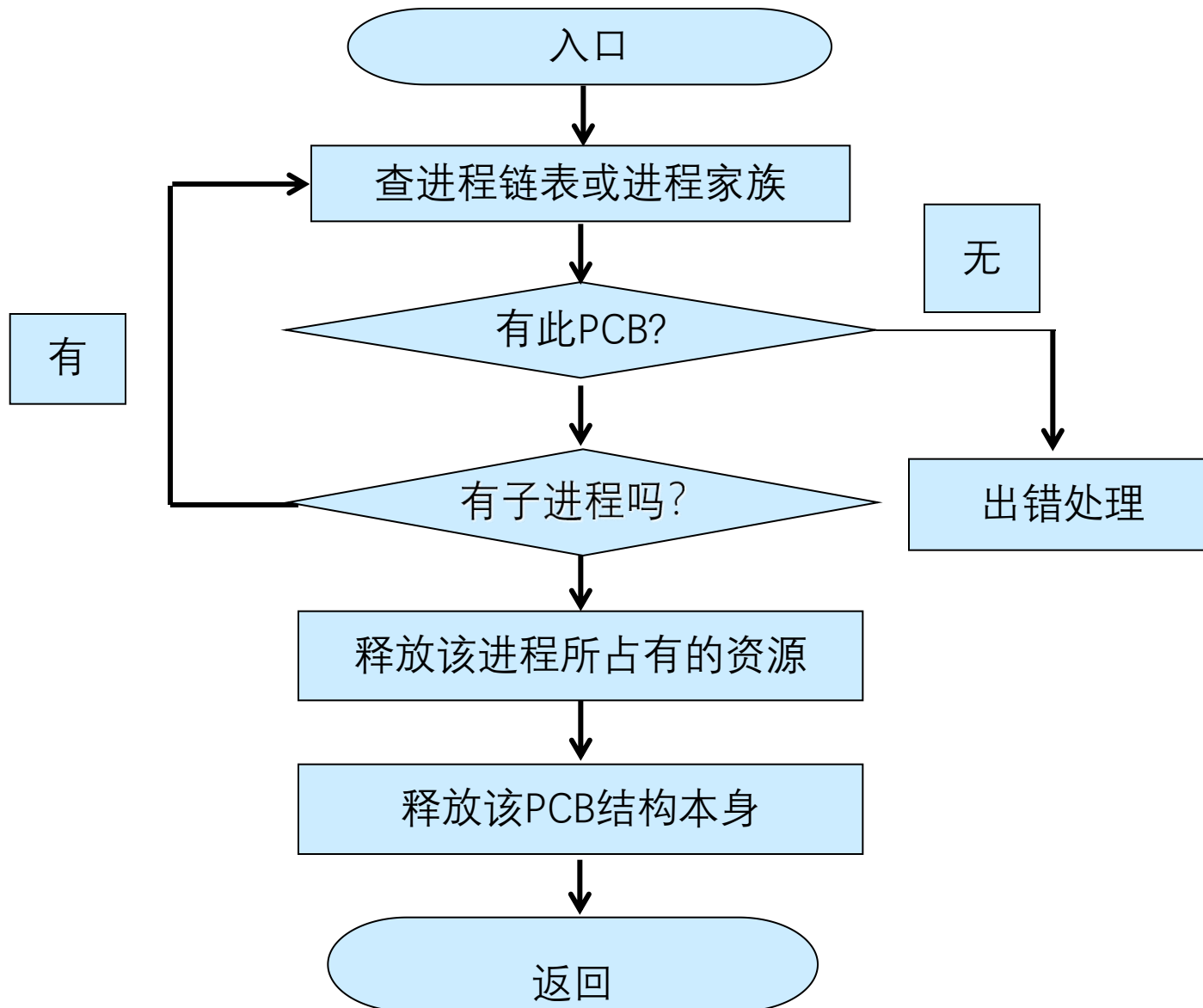
# Process Termination

- Process executes last statement and asks the operating system to **delete** it (**exit**)
  - **Output data** from child to parent (via **wait**)
  - Process' resources are **deallocated** by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded **allocated resources**
  - Task assigned to child is no longer required
  - If parent is exiting
    - ▶ **Some operating system do not allow child to continue if its parent terminates**
      - All children terminated - **cascading termination**





# Process Termination





# Process blocking and wakeup

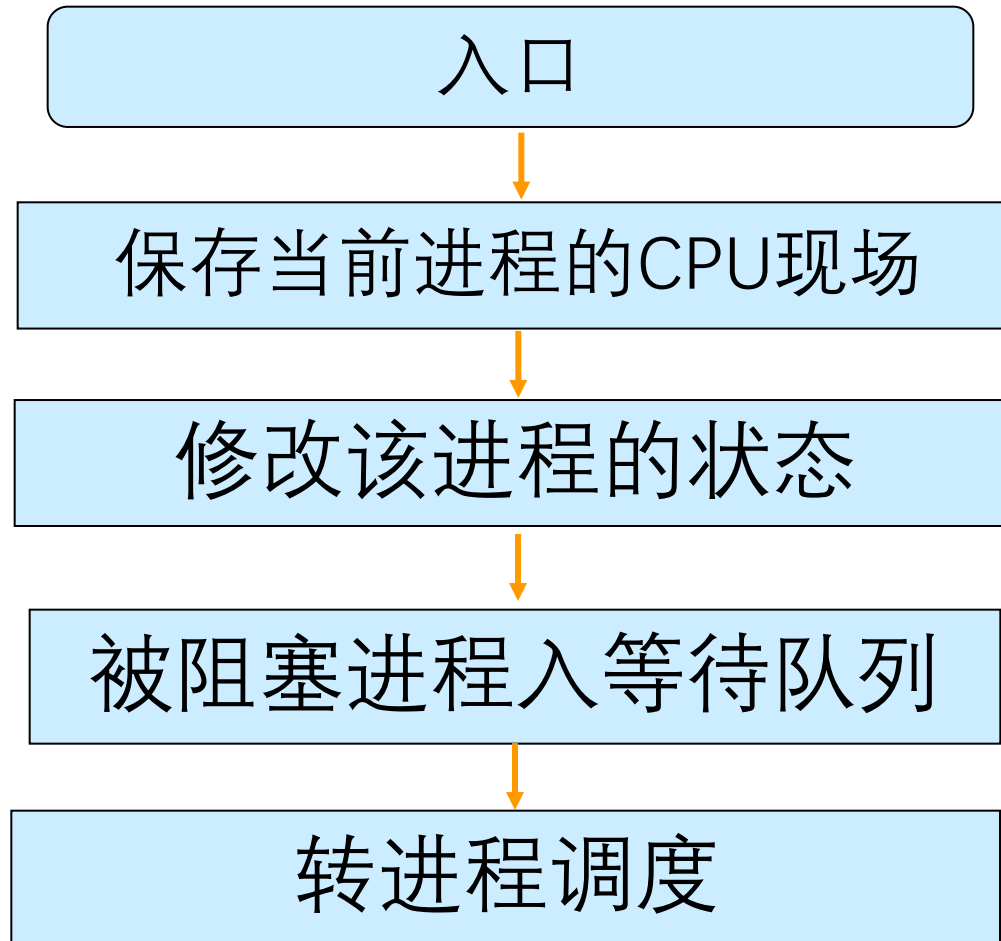
---

- A running process can not continue to run because of something: Running → blocked
  - I/O
  - Signal
- When the event happens, the waiting process will be wakeup
  - blocked → ready



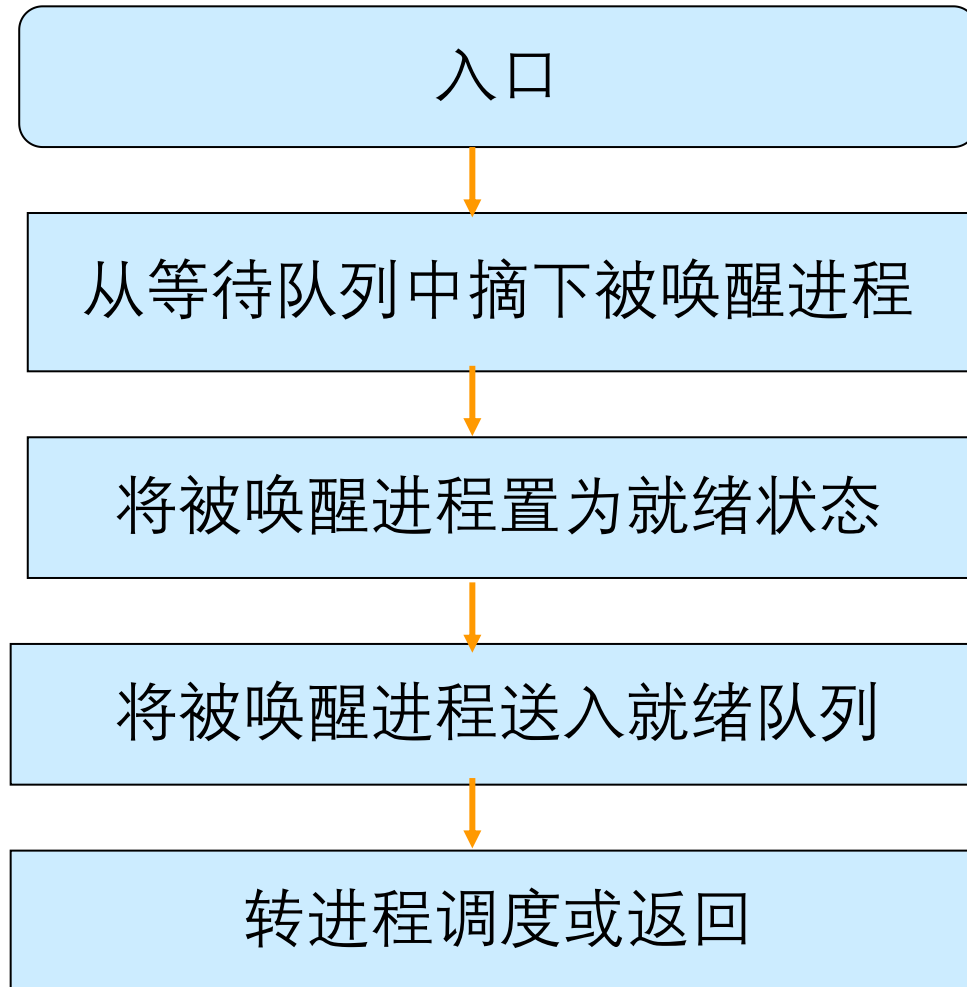


# Process Blocking



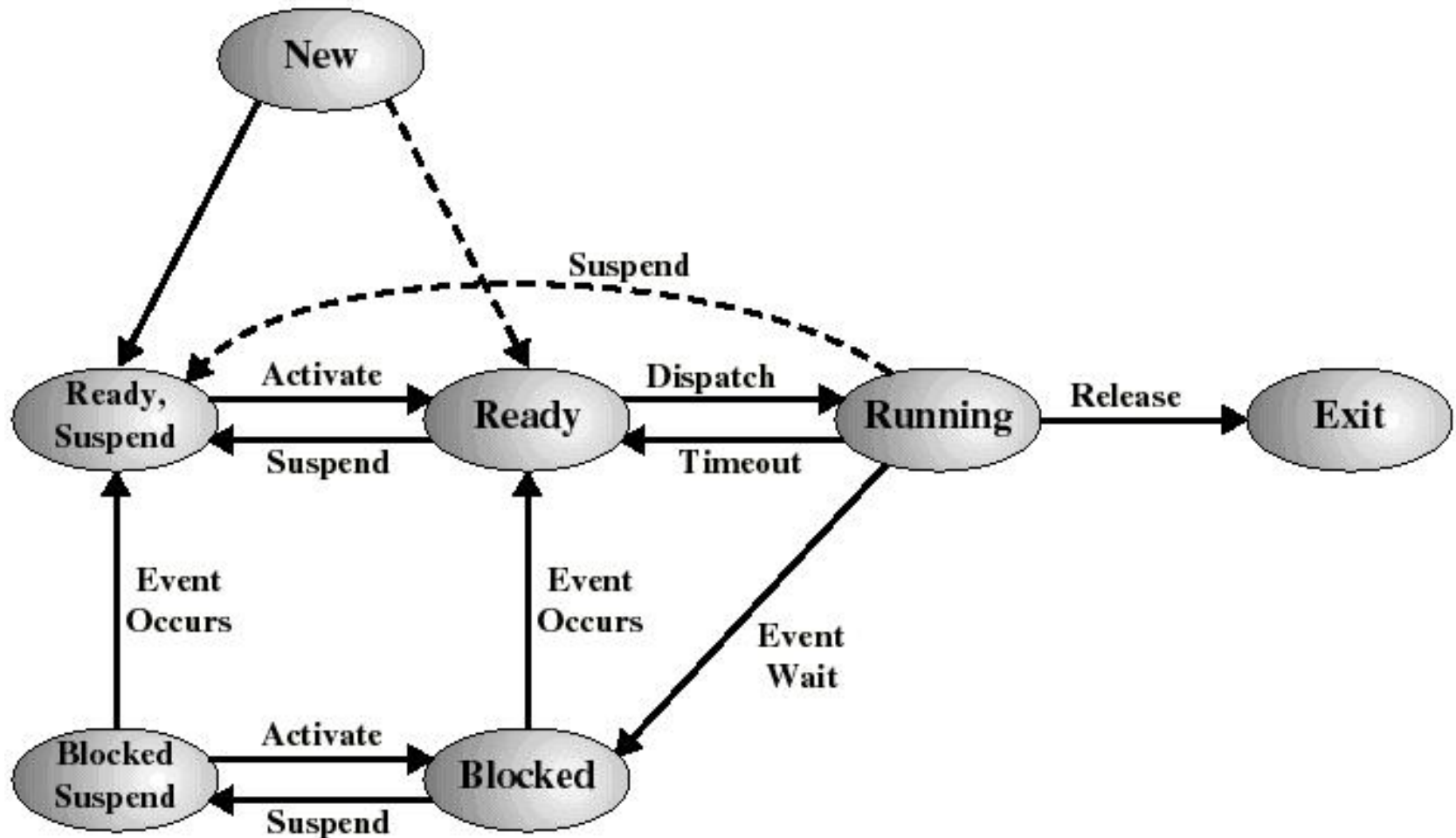


# Process wakeup





# Diagram of Process State





# Diagram of Process State

挂起状态的引入

## 1、终端用户的需要

当终端用户在自己的程序运行期间，发现有可疑问题时，希望暂时使自己的进程静止下来，以便研究其执行情况或对程序进行修改。

如果进程处于执行状态，则暂停执行

如果进程处于就绪状态，则暂时不接受调度

## 2、父进程的需求

父进程常常希望考察和修改子进程，或者需协调各子进程间的活动，要挂起自己的子进程。

## 3、操作系统的需要

操作系统有时需要挂起某些进程，检查运行中资源的使用情况及进行记帐，以便改善系统的运行性能。

## 4、对换的需要

为了缓和内存紧张的情况，将内存中处于阻塞状态的进程换至外存上。

## 5、负荷调节的需要

当实时系统中的工作负荷较重，可能影响到对实时任务的控制时，可由系统把一些不重要或不紧迫的进程挂起，以保证系统仍然能正常运行。







# Interprocess Communication

---

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience





# Interprocess Communication

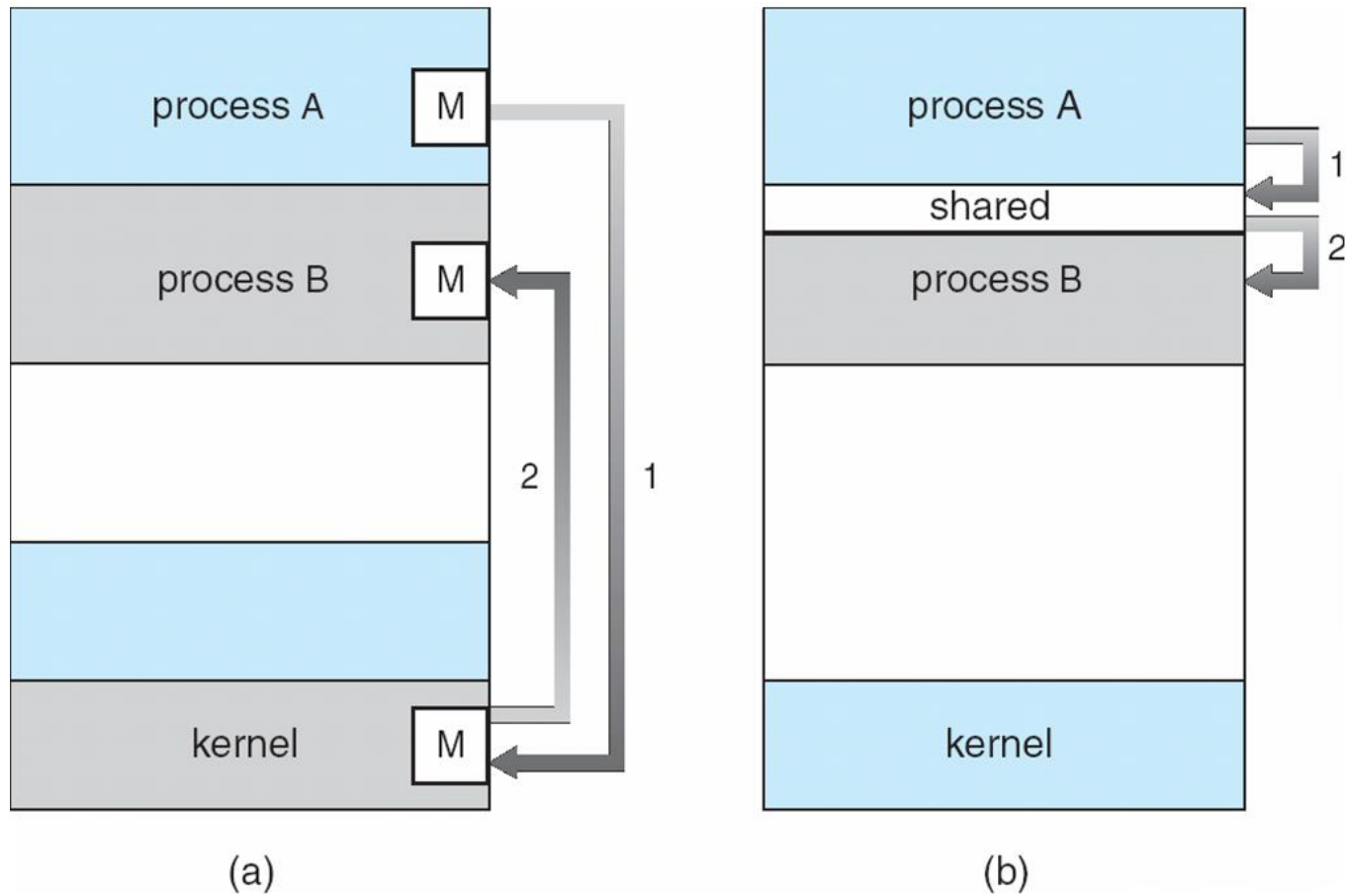
---

- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - Shared memory
  - Message passing





# Communications Models





# Shared memory

---

- Interprocess communication using Shared memory requires communicating processes to establish a shared memory
- The form of the exchanged data and location are **determined by the communicating processes and are not under the OS's control**





# Producer-Consumer Problem

---

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size





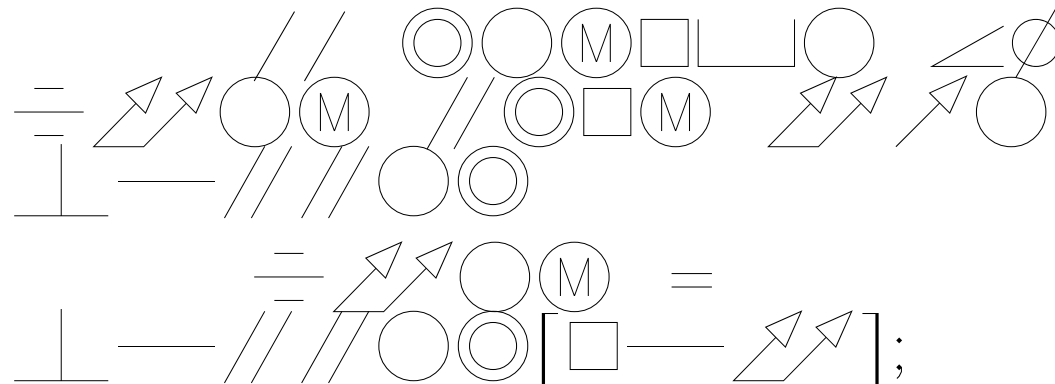
# Bounded-Buffer – Shared-Memory Solution

## ■ Shared data

```
#define BUFFER_SIZE 10  
typedef struct {  
    . . .  
} item;  
  
item buffer[BUFFER_SIZE];  
int in = 0;  
int out = 0;
```











# Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If  $P$  and  $Q$  wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive





# Implementation Questions

---

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?





# Message Passing

---

- Direct Communication
- Indirect Communication



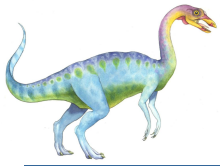


# Direct Communication

---

- Processes must name each other explicitly:
  - **send** ( $P$ , *message*) – send a message to process  $P$
  - **receive**( $Q$ , *message*) – receive a message from process  $Q$
- Properties of communication link
  - Links are established **automatically**
  - A link is associated with **exactly one pair** of communicating processes
  - Between each pair there **exists exactly one link**
  - The link may be unidirectional, but is **usually bi-directional**





# Indirect Communication

---

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each **mailbox** has a **unique id**
  - Processes can communicate only if they **share a mailbox**
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with **many processes**
  - Each pair of processes may **share several communication links**
  - Link may be unidirectional or bi-directional





# Indirect Communication

---

## ■ Operations

- create a new mailbox
- send and receive messages through mailbox
- destroy a mailbox

## ■ Primitives are defined as:

**send**(*A*, *message*) – send a message to mailbox *A*

**receive**(*A*, *message*) – receive a message from mailbox *A*





# Indirect Communication

## ■ Mailbox sharing

- $P_1$ ,  $P_2$ , and  $P_3$  share mailbox A
- $P_1$  sends;  $P_2$  and  $P_3$  receive
- Who gets the message?

## ■ Solutions

- Allow a link to be associated with at most **two processes**
- Allow **only one process at a time** to execute a receive operation
- Allow the system **to select arbitrarily the receiver**. Sender is **notified** who the receiver was.





# Message-passing---Synchronization

---

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** has the sender send the message and continue
  - **Non-blocking receive** has the receiver receive a valid message or null







# Buffering

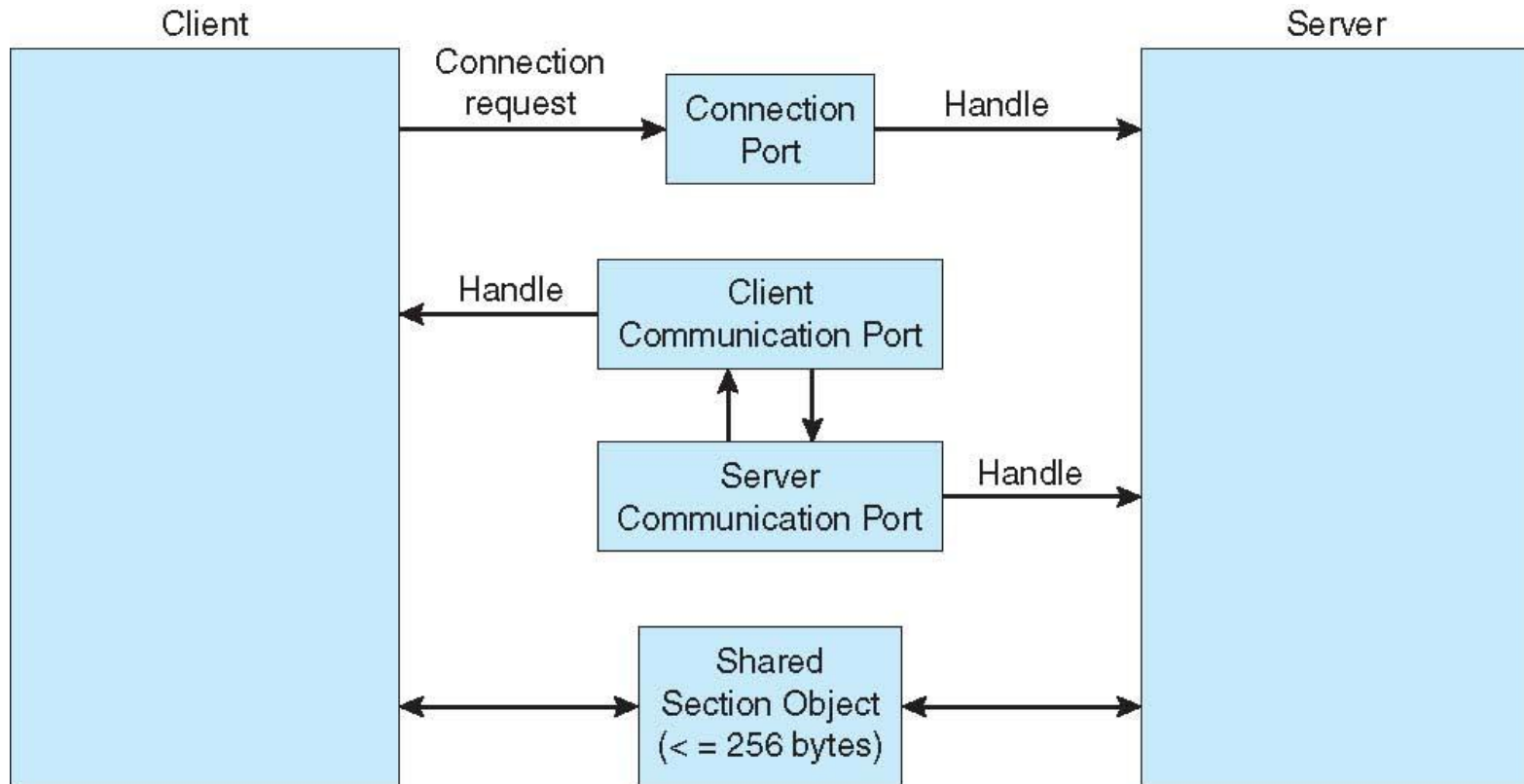
---

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages  
Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of  $n$  messages  
Sender must wait if link full
  3. Unbounded capacity – infinite length  
Sender never waits





# Local Procedure Calls in Windows XP





# Communications in Client-Server Systems

---

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)





# Sockets

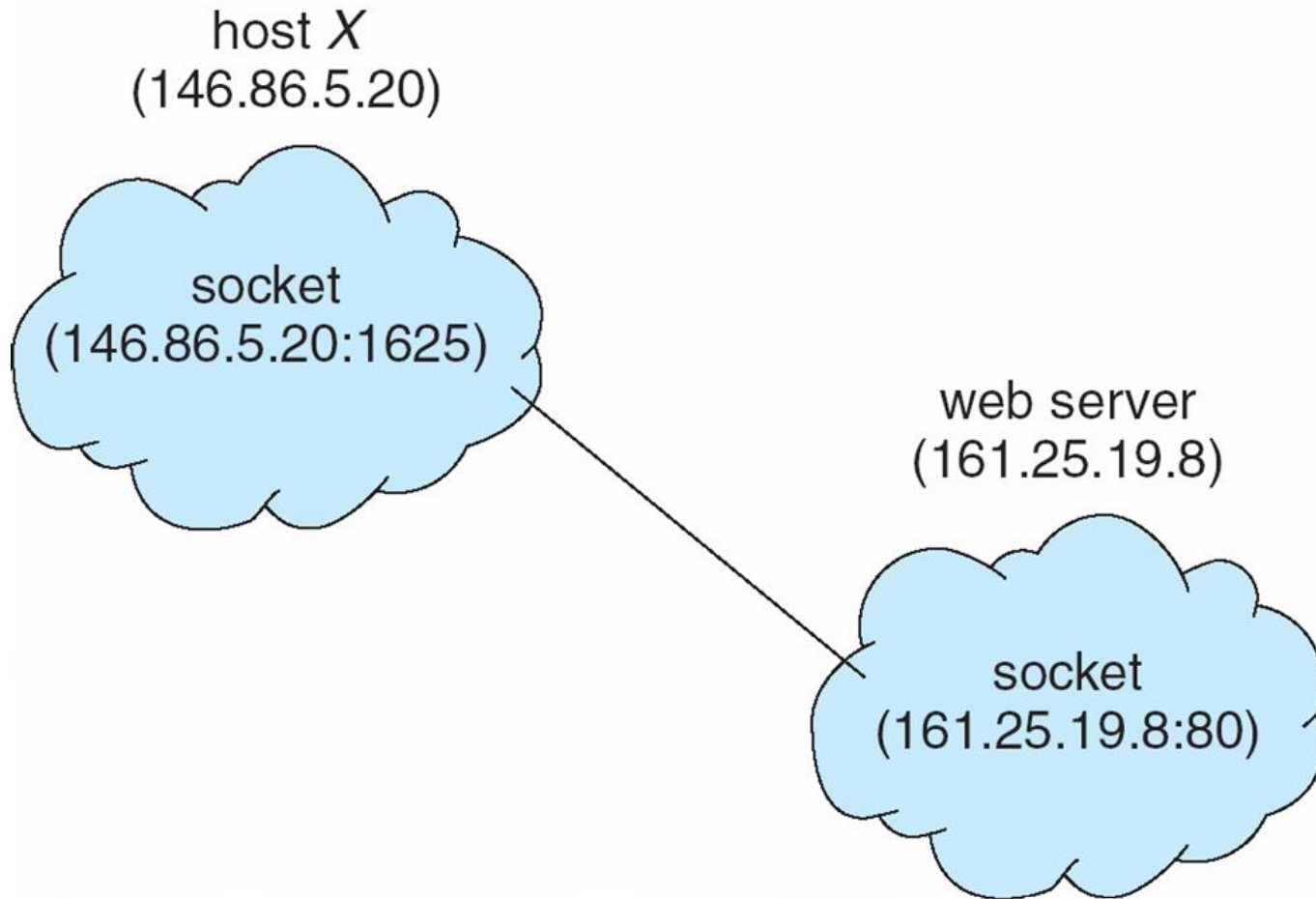
---

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets





# Socket Communication





# Remote Procedure Calls

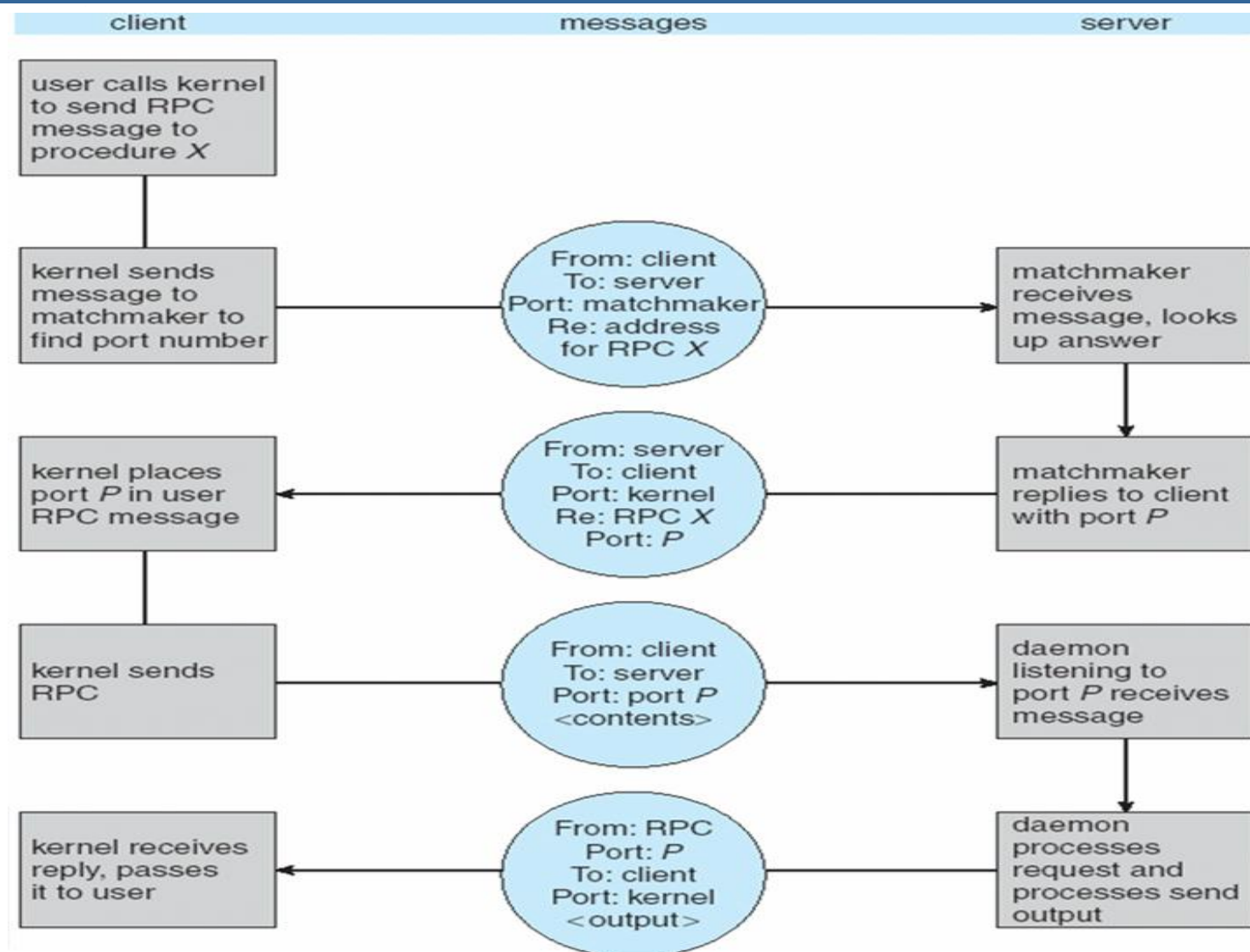
---

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and *marshalls* the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server





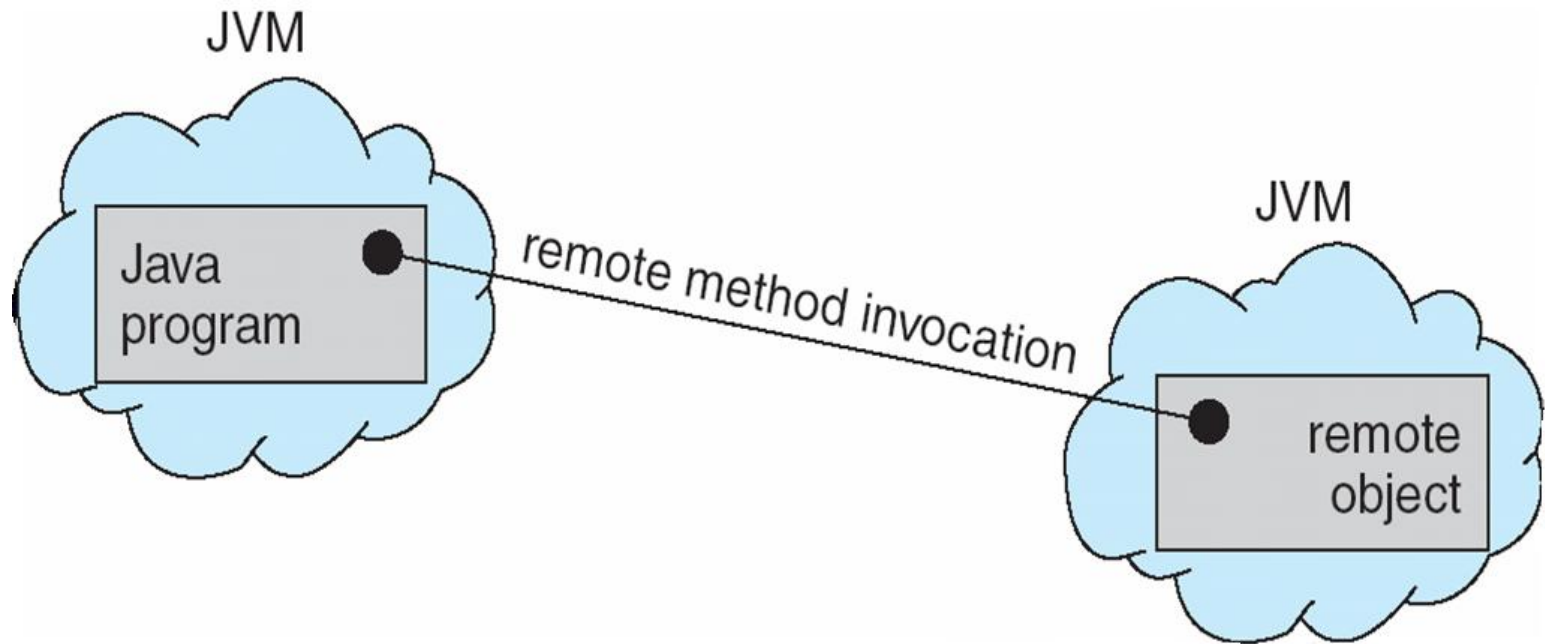
# Execution of RPC





# Remote Method Invocation

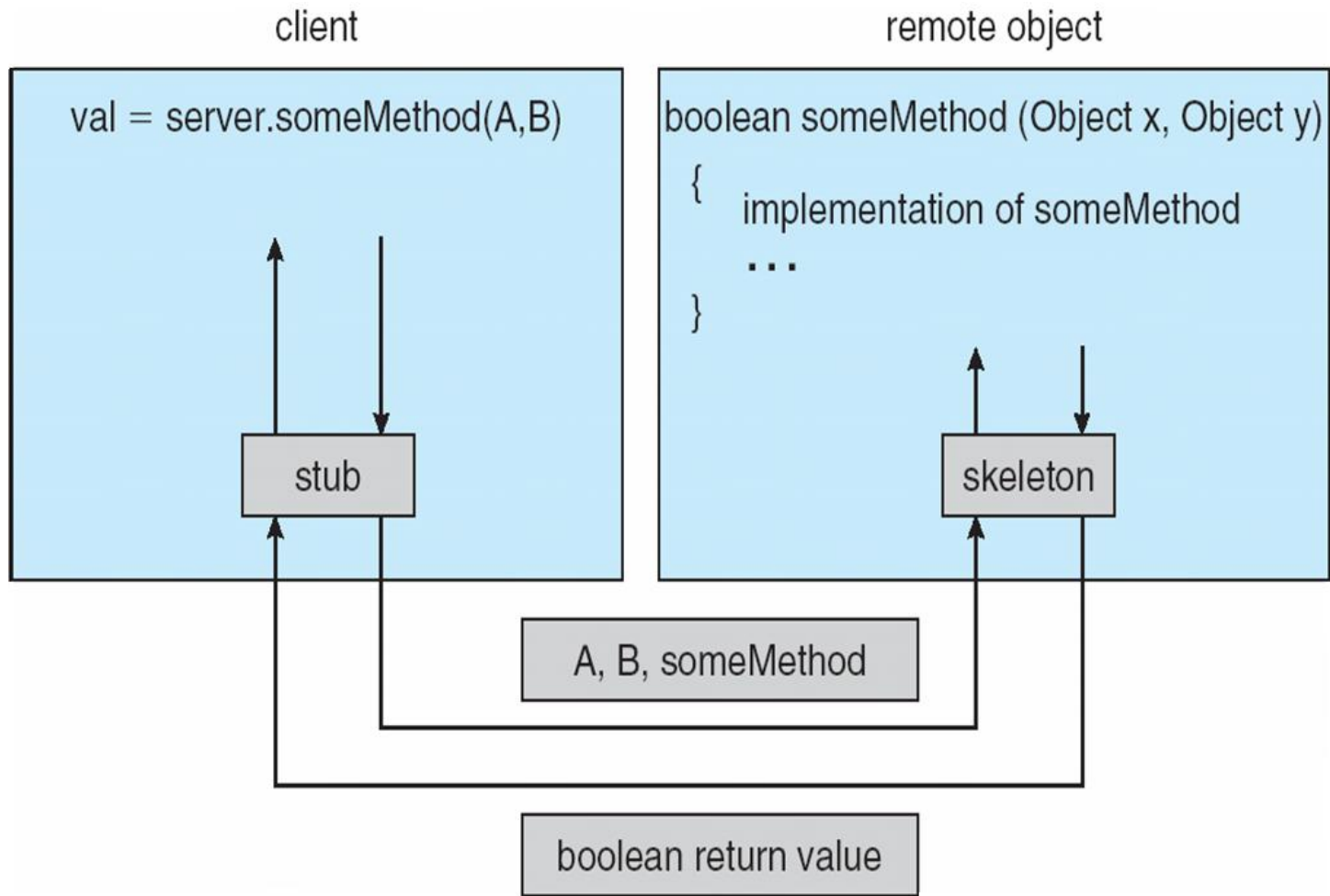
- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs
- RMI allows a Java program on one machine to invoke a method on a remote object







# Marshalling Parameters



# End of Chapter 3

---

