0117401: Operating System
计算机原理与设计

Chapter 3: Process

陈香兰
xlanchen@ustc.edu.cn
http://staff.ustc.edu.cn/~xlanchen

Computer Application Laboratory, CS, USTC @ Hefei
Embedded System Laboratory, CS, USTC @ Suzhou

May 31, 2017

温馨提示：

为了您和他人的工作学习，
请在课堂上**关机或静音**。

不要在课堂上**接打电话**。

Overview

- 多道程序技术和程序并发执行的条件
- Process Concept
- Process Scheduling
- Operation on processes
- Interprocess Communication (进程间通信, IPC)
- Example of IPC Systems
- Communication in C/S Systems
- 小结和作业

Outline

- 多道程序技术和程序并发执行的条件
  - 多道程序技术的难点
  - Seriel execution of programs（程序的顺序执行）
  - Concurrent execution of programs（程序的并发执行）

Some easily confused terms

- In our course:
  - Program(程序): passive entity, usually a file containing a list of instructions stored on disk (often called an executable file).
  - Tasks(任务): a general reference
  - Jobs(作业): in batch system, user programs (and data) waiting to be loaded and executed
  - Processes(进程): a program in execution
- Usually, the term job and process are used interchangeably.

Outline

多道程序技术和程序并发执行的条件

- 多道程序技术的难点
- Seriel execution of programs（程序的顺序执行）
- Concurrent execution of programs（程序的并发执行）

Multiprogramming(多道程序) techniques

- From Simple Batch system →Multiprogramming system
  - Memory must be **shared** by multiple programs
  - CPU must be **multiplexing(复用)** by multiple programs
  - 4 basic components:
    1. Process management
    2. Memory management
    3. I/O system management
    4. file management

Difficulties of multiprogramming techniques

- 与单道相比，在多道系统中，进程之间的运行随着调度的发生而
  具有无序性，那么
  - How to ensure correct concurrent?

- Related theory:
  - Conditions of the concurrent execution of program
  - Theoretical model: Precedence graph (前趋图)
  - Analysis on the serial execution of programs based on precedence graph
  - Analysis on the concurrent execution of programs based on precedence graph

Precedence Graph（前趋图）

- **Goal**：
  准确的描述语句、程序段、进程之间的执行次序

Definition

Precedence graph（前趋图） is a **Directed Acyclic Graph（有向无环图，DAG）**.

- **Node(结点)**：
  一个执行单元（如一条语句、一个程序段或进程）
- **Edge(边**，directed edge(有向边))：
  The precedence relation（前趋关系）“$\rightarrow$”，
  $\rightarrow = \{(P_i, P_j) \mid P_i\text{必须在}P_j\text{开始执行前执行完}\}$

Precedence Graph（前趋图）

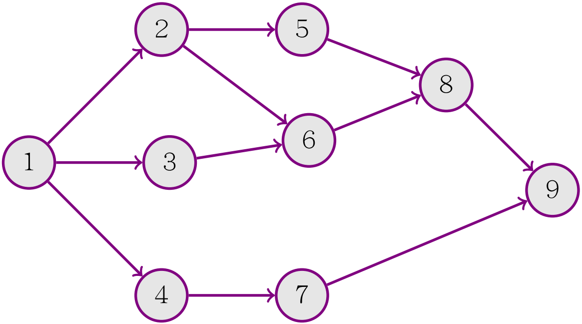
- If $(P_i, P_j) \in \rightarrow$, then $P_i \rightarrow P_j$
  Here,
  P_i is called the **predecessor(前趋)** of P_j, and
  P_j the **subsequent(后继)** of P_i

- 没有前趋的结点称为**初始结点**（initial node）
- 没有后继的结点称为**终止结点**（final node）

- 结点上使用一个**权值**（weight）表示
  该结点所含的程序量或结点的执行时间

Precedence Graph (前趋图)

- Example:



Outline

- 多道程序技术和程序并发执行的条件
  - 多道程序技术的难点
  - Seriel execution of programs（程序的顺序执行）
  - Concurrent execution of programs（程序的并发执行）

Seriel execution of programs(程序的顺序执行)

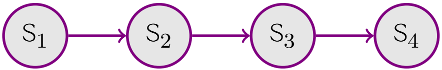
- 一个较大的程序通常包含若干个程序段。程序在执行时，必须按照
  某种先后顺序逐个执行，仅当前一个程序段执行完，后一个程序段
  才能执行。
  例如

  $$I_1 \rightarrow C_1 \rightarrow P_1 \rightarrow I_2 \rightarrow C_2 \rightarrow P_3$$

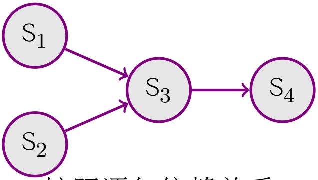
  其中
  - I代表用户程序和数据的输入;
  - C代表计算;
  - P代表输出结果

Seriel execution of programs(程序的顺序执行)

- 在一个程序段中，多条语句也存在执行顺序的问题。
在下面的例子中，S1和S2必须在S3执行前执行完。
类似的，S4必须在S3执行完才能执行。
    1. S1：a＝x＋3
    2. S2：b＝y＋4
    3. S3：c＝a＋b
    4. S4：d＝a＋c



按指令地址顺序执行



按照语句依赖关系

程序顺序执行时的特征

1 顺序性

- 严格按照程序规定的顺序执行

2 封闭性

- 程序是在封闭的环境下运行的。独占全机资源。一旦开始运行，
  结果不受外界因素的影响。

3 可再现性

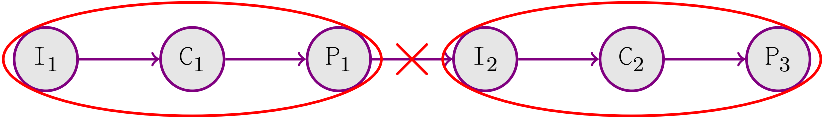
- 只要程序执行时的环境和初始条件相同，都将获得相同的结果。

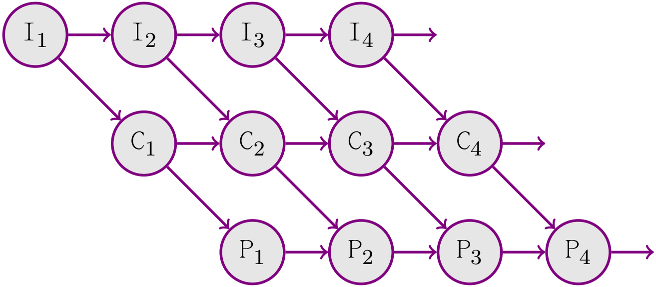
Outline

- 多道程序技术和程序并发执行的条件
  - 多道程序技术的难点
  - Seriel execution of programs (程序的顺序执行)
  - Concurrent execution of programs (程序的并发执行)

Concurrent execution of programs (程序的并发执行)

- P_i与I_{i+1}之间不存在内在的前趋关系





程序并发执行时的前趋图

程序并发执行时的特征

1 间断性

- 并发程序“执行——暂停执行——执行”

2 失去封闭性

- 由于资源共享，程序之间可能出现相互影响的现象

3 不可再现性

- 原因同上。
- 举例：变量N的共享，设某时刻N＝n，则若执行顺序为：

1. N:＝N＋1；print(N)；N:=0；N的值依次为n＋1；n＋1；0
2. print(N)；N:=0；N:＝N＋1； N的值依次为n；0；1
3. print(N)；N:＝N＋1；N:=0；N的值依次为n；n＋1；0

程序并发执行的条件(Bernstein's conditions)

- 在上述3个特性中，必须防止“不可再现性”。
- 为使并发程序的执行保持“可再现性”，引入并发执行的条件。
  - **思路**：分析程序或语句的输入信息和输出信息，考察它们的相关性
  - Definitions, notation and terminology:
    - **读集**$R(p_i)$，表示程序p_i在执行时需要参考的所有变量的集合
    - **写集**$W(p_i)$，表示程序p_i在执行期间要改变的所有变量的集合
  - 1966, Bernstein: if programs p_1 and p_2 meet the following conditions, they can be executed **concurrently**, and have **reproducibility (可再现性)**

    1. If process p_i writes to a memory cell M_i, then no process p_j can read the cell M_i.
    2. If process p_i read from a memory cell M_i, then no process p_j can write to the cell M_i.
    3. If process p_i writes to a memory cell M_i, then no process p_j can write to the cell M_i.

$$R(p_1)\bigcap W(p_2)\bigcup R(p_2)\bigcap W(p_1)\bigcup W(p_1)\bigcap W(p_2)=\varnothing$$

Outline

Outline

the processes

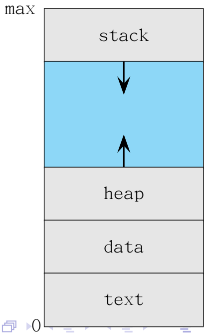
- 进程需要使用某种方法加以描述，原因
  1. 进程运行的间断性，要求在进程暂停运行时记录该程序的现场，
     以便下次能正确的继续运行
  2. 资源的共享，要求能够记录进程对资源的共享情况
  3. 为保证程序“正确”的并发执行，必须将进程看成某种对象，
     对其进行描述并加以控制

Process Concept I

- An OS executes a variety of programs:
  - Batch system - **jobs**
  - Time-shared systems - **user programs or tasks**
  - PC - **several programs**: a word processor, a web browser, etc.
- we call all of them **process**
  - a program in execution;
  - process execution must progress in sequential fashion

A process includes:

- text section⇐program code
- program counter + other registers⇐current activity
- stack⇐temporary data
- data section⇐global variables
- heap



Process Concept II

COMPARE: Program vs. Process?

- Program: a passive entity (静态的)
- Process: a active entity (活动的)

进程的五大特征

1. 动态性：最基本的特性
2. 并发性
3. 独立性
4. 异步性
5. 结构特征

进程的五大特征

1. **动态性：最基本的特性**

   “它由创建而产生，由调度而执行，因得不到资源而暂停执行，以及由撤销而消亡”

   - 具有生命期

2. **并发性**
3. **独立性**
4. **异步性**
5. **结构特征**

进程的五大特征

1. **动态性：最基本的特性**
2. **并发性**
   - 多道
   - 既是进程也是OS的重要特征
3. **独立性**
4. **异步性**
5. **结构特征**

进程的五大特征

1. **动态性：最基本的特性**
2. **并发性**
3. **独立性**
   - 进程是一个能独立运行的基本单位，也是系统中独立获得资源和独立调度的基本单位。
4. **异步性**
5. **结构特征**

进程的五大特征

1. **动态性：最基本的特性**
2. **并发性**
3. **独立性**
4. **异步性**
   - 进程按各自独立的、不可预知的速度向前推进。
   - 导致“不可再现性”
   - OS必须采取某种措施来保证各程序之间能协调运行。
5. **结构特征**

进程的五大特征

1. **动态性：最基本的特性**
2. **并发性**
3. **独立性**
4. **异步性**
5. **结构特征**
   - 从结构上看，进程实体是由程序段、数据段及进程控制块三部分组成

$$\text{进程映像} = \text{程序段} + \text{数据段} + \text{进程控制块}$$

Outline

Process State

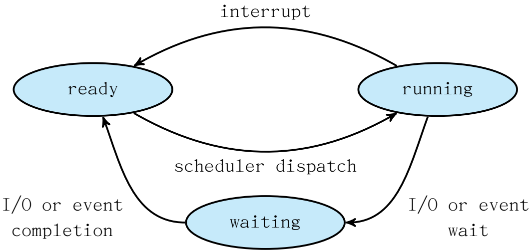
- As a process executes, it changes **its** **state**.

State Models（状态模型）

1. 最基本的“三状态”模型
2. 引入“新”和“终止”态的“五状态”模型
3. 引入“挂起”状态的“七状态”模型

1 “三状态”模型

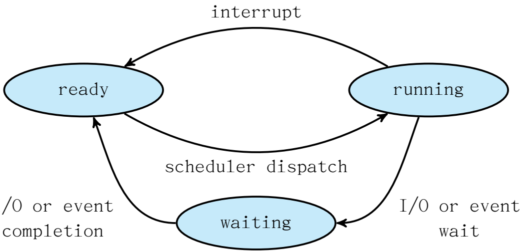
- 三种最基本的状态
  1. ready (就绪)： “万事具备，只欠CPU”
  2. running (执行)
  3. waiting (等待，also blocked(阻塞)，sleeping(睡眠))



4 types of state transferring

1 "三状态"模型

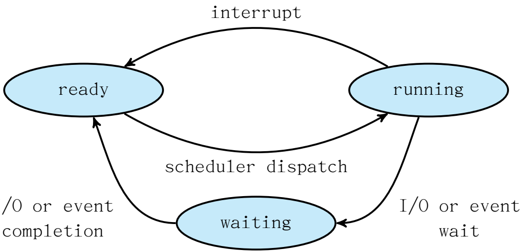
- 三种最基本的状态
  1. ready (就绪): "万事具备, 只欠CPU"
     - DataStructure: ready queue
  2. running (执行)
  3. waiting (等待, also blocked(阻塞), sleeping(睡眠))



4 types of state transferring

1 "三状态"模型

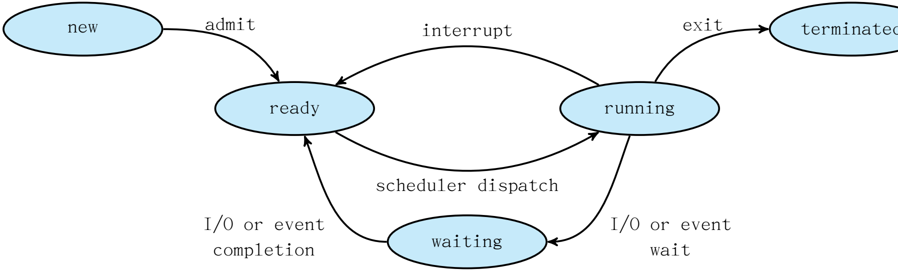
- 三种最基本的状态
  1. ready (就绪): "万事具备，只欠CPU"
  2. running (执行)
  3. waiting (等待, also blocked(阻塞), sleeping(睡眠))
     The process is waiting for some event to occur:
     - I/O completion, reception of a signal, resource allocation, etc.
     - DataStructure: waiting queue



4 types of state transferring

2 “五状态”模型

- Two more states is added to the “three state” model.
  1. **new** (新状态): The process is beig created
     - initialization, resource preallocation, etc.
  2. **terminated** (终止状态): The process has finished execution, normally or abnormally.
     - removed from ready queue, but still not destroyed.
     - other process may gather some information from the terminated processes



6 types of state transferring

3 “Seven state” model

- 进程因自身内部的一些原因，无法继续运行时，暂时进入“等待”状态，当等待的原因消除后，就可以返回就绪状态；
  但有时会因进程外部的一些原因，使得进程暂时不能继续运行。
  外部原因主要有
  1. 终端用户的需要
  2. 父进程的需求
  3. 操作系统的需要
  4. 对换(swapping)的需要
  5. 负载(work load)调节的需要

3 “Seven state” model

- 进程因自身内部的一些原因，无法继续运行时，暂时进入“等待”状态，当等待的原因消除后，就可以返回就绪状态；
  但有时会因进程外部的一些原因，使得进程暂时不能继续运行。
  外部原因主要有
  1. 终端用户的需要
  2. 父进程的需求
  3. 操作系统的需要
  4. 对换(swapping)的需要
  5. 负载(work load)调节的需要

引入“挂起”状态

3 “Seven state” model

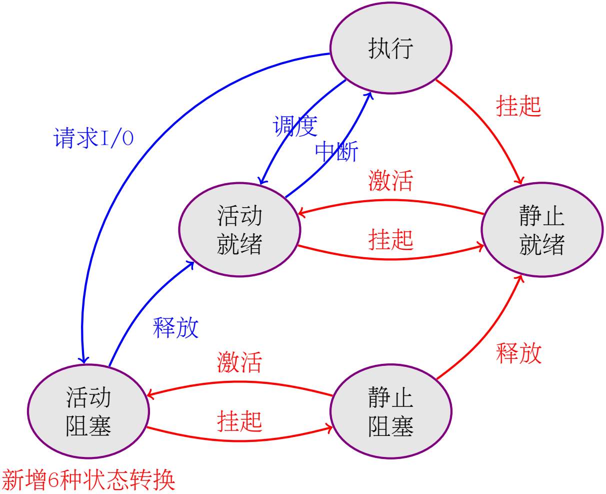
- 进程因自身内部的一些原因，无法继续运行时，暂时进入“等待”状态，当等待的原因消除后，就可以返回就绪状态；
  但有时会因进程外部的一些原因，使得进程暂时不能继续运行。
  外部原因主要有
  1. 终端用户的需要
  2. 父进程的需求
  3. 操作系统的需要
  4. 对换(swapping)的需要
  5. 负载(work load)调节的需要

引入“挂起”状态

- “挂起”状态不是一种状态，而是**一类状态**
  - 挂起后处于静止状态：**静止就绪**，**静止阻塞**
  - 非挂起的活动状态：**活动就绪**，**活动阻塞**，还包括**执行态**

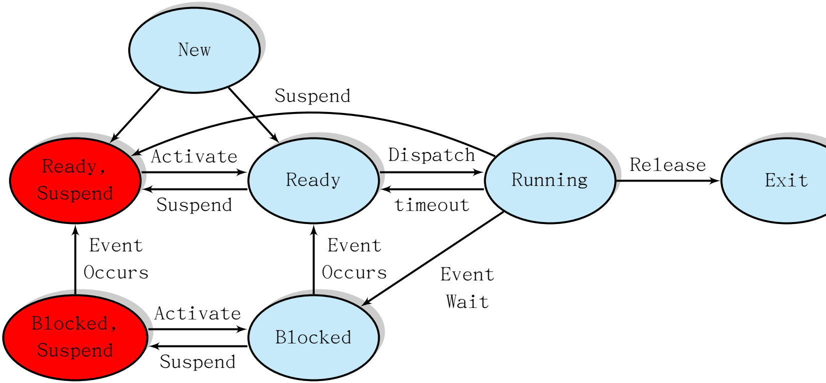
3 “Seven state” model

- 在状态转换中，增加了活动状态与静止状态之间、静止状态内部之间的状态转换



3 “Seven state” model

- 包含“挂起”状态的“7状态”模型



Outline

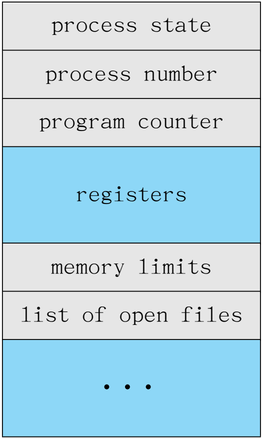
Process Control Block（进程控制块，PCB）

- Each process is represented in the OS by a PCB, also called Task Control Block，TCB
  是操作系统中的一种关键数据结构
  - 由操作系统进程管理模块维护
  - 常驻内存
- 操作系统根据PCB来控制和管理并发执行的进程

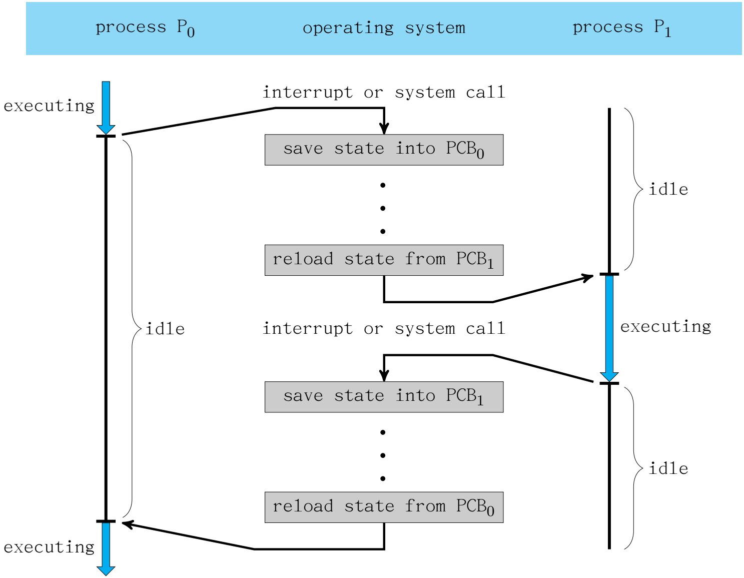
PCB是进程存在的唯一标志

Process Control Block (进程控制块，PCB)

- Information associated with each process
  - Process **state** (...)
  - **Program counter**
  - CPU **registers**
  - CPU-scheduling information
  - Memory-management information
  - Accounting information: time used, time limit, ...
  - I/O status information



CPU Switch From Process to Process



Examples I

- 观察：数据结构和状态
  - struct task_struct in Linux0.11 & Linux 2.6.26
  - struct OS_TCB in μC/OS-II

```
typedef struct os_tcb {
    OS_STK *OSTCBStkPtr; /* Pointer to current top of stack */
#if OS_TASK_CREATE_EXT_EN > 0
    void *OSTCBExtPtr; /* Pointer to user definable data for TCB
extension */
    OS_STK *OSTCBStkBottom; /* Pointer to bottom of stack */
    INT32U OSTCBStkSize; /* Size of task stack (in number of stack
elements) */
    INT16U OSTCBOpt; /* Task options as passed by OSTaskCreateExt() */
    INT16U OSTCBId; /* Task ID (0..65535) */
#endif
    struct os_tcb *OSTCBNext; /* Pointer to next TCB in the TCB list */
    struct os_tcb *OSTCBPrev; /* Pointer to previous TCB in the TCB
list */
#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0) || (OS_SEM_EN > 0)
|| (OS_MUTEX_EN > 0)
    OS_EVENT *OSTCBEventPtr; /* Pointer to event control block */
#endif
```

Examples II

```
#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
    void *OSTCBMsg; /* Message received from OSMboxPost() or OSQPost() */
#endif
#if (OS_VERSION >= 251) && (OS_FLAG_EN > 0) && (OS_MAX_FLAGS > 0)
#if OS_TASK_DEL_EN > 0
    OS_FLAG_NODE *OSTCBFlagNode; /* Pointer to event flag node */
#endif
    OS_FLAGS OSTCBFlagsRdy; /* Event flags that made task ready to run */
#endif
    INT16U OSTCBDly; /* Nbr ticks to delay task or, timeout waiting for
event */
    INT8U OSTCBStat; /* Task status */
    INT8U OSTCBPrio; /* Task priority (0 == highest, 63 == lowest) */
    INT8U OSTCBX; /* Bit position in group corresponding to task priority
(0..7) */
    INT8U OSTCBY; /* Index into ready table corresponding to task priority
*/
    INT8U OSTCBBitX; /* Bit mask to access bit position in ready table */
    INT8U OSTCBBitY; /* Bit mask to access bit position in ready group */
#if OS_TASK_DEL_EN > 0
    BOOLEAN OSTCBDelReq; /* Indicates whether a task needs to delete
itself */
#endif
} OS_TCB;
```

Outline

Process Scheduling

- Process Scheduling Queues
- Schedulers
- Context Switch(上下文切换)

Process Scheduling

The objective of multiprogramming

to have some process running at all times, to maximize CPU utilization.

The objective of time sharing

to switch the CPU among processes so frequently that users can interact with each program whilt it is running.

What the system need?

the process scheduler selects an available process to execute on the CPU.

Process Scheduling

The objective of multiprogramming

to have some process running at all times, to maximize CPU utilization.

The objective of time sharing

to switch the CPU among processes so frequently that users can interact with each program whilt it is running.

What the system need?

the process scheduler selects an available process to execute on the CPU.

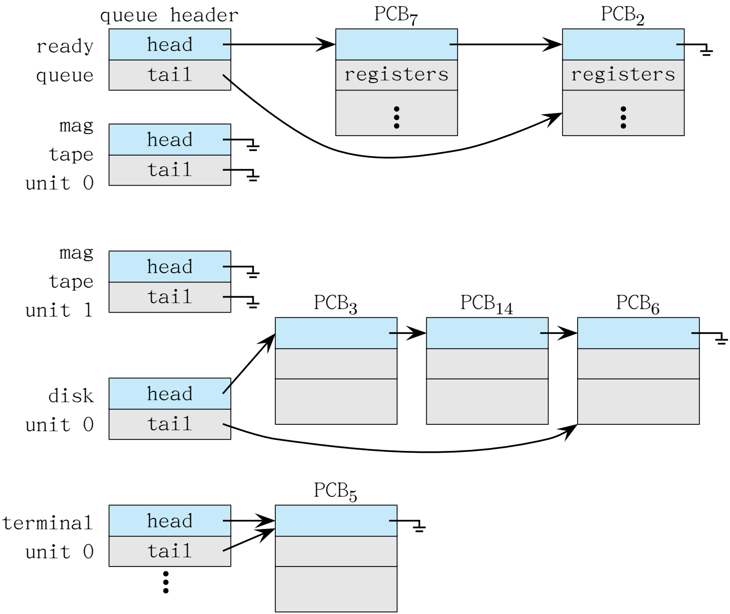
Outline

Process Scheduling Queues

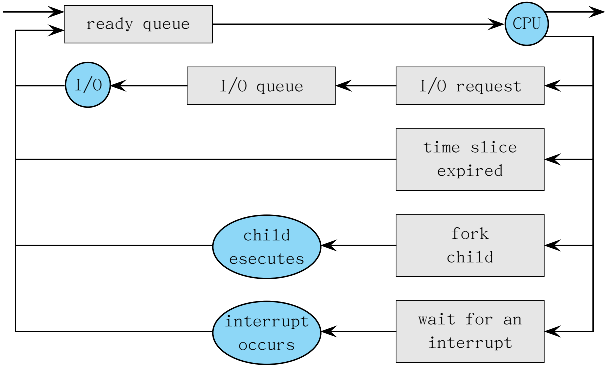
Processes migrate among the various queues

- **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

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



Queueing-diagram representation of process scheduling

Outline

Schedulers I

Long-term (长期) scheduler (or job scheduler)

- selects which processes should be brought into the ready queue

Short-term (短期) scheduler (or CPU scheduler)

- selects which process should be executed next and allocates CPU

The primary distinction between long-term & short-term schedulers I

- The prilmary distinction between long-term & short-term schedulers lies in frequency of execution
  - Short-term scheduler is invoked very frequently (UNIT: ms) $\Rightarrow$ must be fast
  - Long-term scheduler is invoked very infrequently (UNIT: seconds, minutes) $\Rightarrow$ may be slow
  - WHY?
- The long-term scheduler controls the degree of multiprogramming (多道程序度)
  - the number of processes in memory.
  - stable?

The primary **distinction** between long-term & short-term schedulers II

- Processes can be described as either:

I/O-bound (I/O密集型) process

  - spends more time doing I/O than computations, many short CPU bursts

CPU-bound (CPU密集型) process

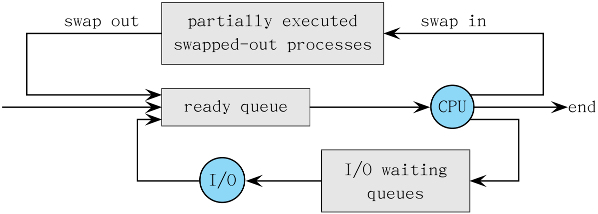
  - spends more time doing computations; few very long CPU bursts

- **IMPORTANT** for long-term scheduler:
  - A good process mix of I/O-bound and CPU-bound processes.

- The long-term scheduler may **be absent** or minimal
  - UNIX, MS Windows, ...
  - The stability depends on
    - physical limitation
    - self-adjusting nature of human users

Addition of Medium Term (中期) Scheduling

- **Medium-Term (中期) Scheduler**
  - can **reduce** the degree of multiprogramming
  - the scheme is called **swapping (交换)**: swap in VS. swap out



Addition of medium-term scheduling to the queueing diagram

Outline

Context Switch (上下文切换)

- CONTEXT (上下文)
  - when an interrupt occurs; When scheduling occurs

the context is represented in the PCB of the process

  - CPU registers
  - process state
  - memory-management info
  - ...

  - operation: state save VS. state restore

Context Switch (上下文切换)

- 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**
  - Context-switch time is **overhead**; the system does no useful work while switching
  - **Time dependent on hardware support** (typical: n μs)
    - CPU & memory speed
    - N of registers
    - the existence special instructions

Code reading

- 观察
  - 队列的组织
  - 上下文的内容和组织
  - 上下文切换
- linux-0.11
- linux-2.6.26
- uC/OS-II

Outline

Operation on processes

- The processes in most systems can execute **concurrently**, and they may be created and deleted **dynamically**.
- The OS must provide a mechanism for
  - **process creation**
  - **process termination**

Outline

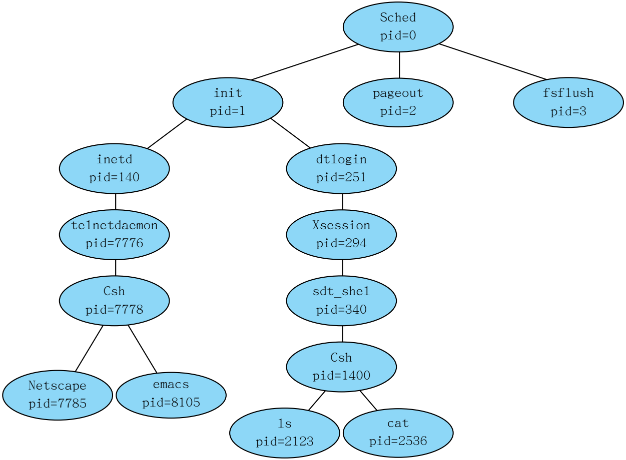
Process Creation I

- Parent process (父进程) create children processes (子进程), which, in turn create other processes, forming a tree of processes
- Most OSes identify processes according to a unique process identifier (pid).
  - typically an integer number
- UNIX & Linux

Command:

```
ps -el
pstree
```

Process Creation II



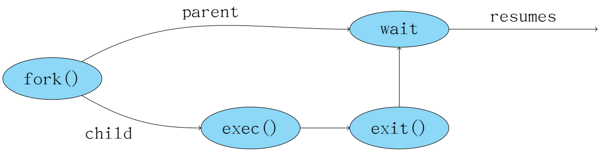
A tree of processes on a typical Solaris

Parent and children

- Resource sharing
  - In general, a process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.
  - When a process creates a subprocesses
    - Parent and children may share all resources, or
    - Children may share subset of parent's resources, or
    - Parent and child may share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it

UNIX examples: fork + exec

- **fork** system call creates new process
- **exec** system call used after a **fork** to replace the process' memory space with a new program




```
#include <unistd.h>
pid_t fork(void);
```

```
#include <unistd.h>
extern char **environ;
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const
envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

C Program Forking Separate Process

```
int main(void) {
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) {      /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    } else if (pid == 0) {      /* child process */
        execlp("/bin/ls", "ls", NULL);
    } else {      /* parent process */
        /* parent will wait for the child to complete */
        wait (NULL);
        printf ("Child Complete");
        exit(0);
    }
}
```

Outline

Process Termination

1. [Self]Process executes last statement and asks the OS to delete it by using the **exit()** system call.
   - Output data (**a status value**, typically an integer) **from child to parent** (via **wait()**)
   - Process' **resources** are **deallocated** by the OS
2. [Other]Termination can be caused by another process
   - Example: TerminateProcess() in Win32
3. [User]Users could **kill** some jobs.
   - See command "kill" and "pkill"

Process Termination

4. [Parent]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

- UNIX Example:
  - If the parent terminates, all its children have assigned as their **new parent** the **init** process.

```
#include <stdlib.h>
void exit(int status);
```

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

Example: echo.

```
#include <stdio.h>
int main(void){
    char string[80];
    int i;
    printf("HELLO! NICE TO MEET YOU!\n");
    for (i=0;i<10;i++){
        printf("Input %d: ",i);
        scanf("%s",string);
        printf("You say: %s\n",string);
    }
    printf("GOODBYE!\n");
}
```

Describe the whole life of a process executing echo.

Outline

- Interprocess Communication (进程间通信, IPC)
  - Shared-Memory systems
  - Message-Passing Systems

Interprocess Communication (进程间通信, IPC)

- Processes executing concurrently in the OS may be either **independent processes** or **cooperating processes**
  - **Independent process cannot** affect or be affected by the execution of other processes
  - **Cooperating process can** affect or be affected by the execution of other processes
- **Advantages** of allowing process cooperation
  - **Information sharing**: a shared file VS. several users
  - **Computation speed-up**: 1 task VS. several subtasks in parallel with multiple processing elements (such as CPUs or I/O channels)
  - **Modularity**
  - **Convenience**: 1 user VS. several tasks
- Cooperating processes require an **IPC mechanism** that will allow them **to exchange data and information**.

Interprocess Communication (进程间通信, IPC)

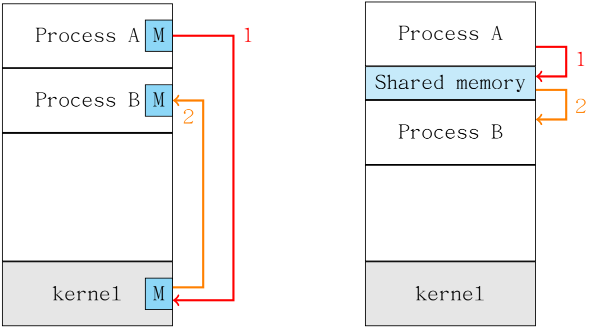
Two fundamental models of IPC:

1. Message-passing (消息传递) model
   - useful for exchange **smaller amount of data**, because no conflicts need be avoided.
   - **easier to implement**
   - **exchange information via system calls** such as send(), receive()
2. Shared-memory (共享内存) model

Interprocess Communication (进程间通信, IPC)

Two fundamental models of IPC:

1. Message-passing (消息传递) model
2. Shared-memory (共享内存) model
   - faster at memory speed via memory accesses.
   - system calls only used to establish shared memory regions



Outline

- Interprocess Communication (进程间通信, IPC)
  - Shared-Memory systems
  - Message-Passing Systems

Shared-Memory systems

- Normally, the OS tries to prevent one process from accessing another process's memory.
- Shared memory requires that two or more processes agree to remove this restriction.
  - exchange information by R/W data in the shared areas.
  - The form of data and the location are determined by these processes and not under the OS' s control.
  - The processes are responsible for ensuring that they are not writing to the same location simultaneously.

Example: Producer-Consumer Problem (生产者-消费者问题)

- **Producer-Consumer Problem (生产者-消费者问题, PC问题)**: Paradigm for cooperating processes
  - **producer (生产者)** process produces information that is consumed by a **consumer (消费者)** process.
- Shared-Memory solution
  - a buffer of items shared by producer and consumer
- Two types of buffers:
  1. unbounded-buffer places no practical limit on the size of the buffer
  2. bounded-buffer assumes that there is a fixed buffer size

Example: Producer-Consumer Problem (生产者-消费者问题)

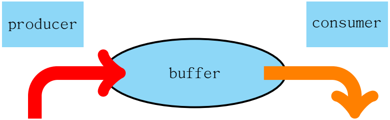
- **Producer-Consumer Problem (生产者-消费者问题, PC问题)**: Paradigm for cooperating processes
  - **producer (生产者)** process produces information that is consumed by a **consumer (消费者)** process. Example:

$$\text{complier} \xrightarrow{\text{assembly code}} \text{assembler} \xrightarrow{\text{object models}} \text{loa}$$

- **Shared-Memory solution**
  - a buffer of items shared by producer and consumer
- **Two types of buffers:**
  1. **unbounded-buffer** places no practical limit on the size of the buffer
  2. **bounded-buffer** assumes that there is a fixed buffer size

Example: Producer-Consumer Problem (生产者-消费者问题)

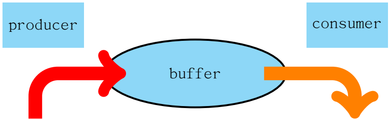
- **Producer-Consumer Problem (生产者-消费者问题, PC问题)**: Paradigm for cooperating processes
  - **producer (生产者)** process produces information that is consumed by a **consumer (消费者)** process.
- **Shared-Memory solution**
  - a buffer of items shared by producer and consumer



- **Two types of buffers:**
  1. **unbounded-buffer** places no practical limit on the size of the buffer
  2. **bounded-buffer** assumes that there is a fixed buffer size

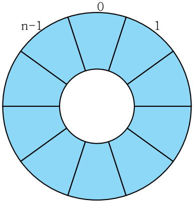
Example: Producer-Consumer Problem (生产者-消费者问题)

- **Producer-Consumer Problem (生产者-消费者问题, PC问题)**: Paradigm for cooperating processes
  - **producer (生产者)** process produces information that is consumed by a **consumer (消费者)** process.
- **Shared-Memory solution**
  - a buffer of items shared by producer and consumer



- **Two types of buffers:**
  1. **unbounded-buffer** places no practical limit on the size of the buffer
  2. **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer — Shared-Memory Solution



Insert() Method

```
while (true) {
      /* Produce an item */
      while (((in + 1) % BUFFER_SIZE) == out)
           ; /* do nothing — no free buffers */
      buffer[in] = item;
      in = (in + 1) % BUFFER SIZE;
}
```

Shared variables reside in a shared region

```
#define BUFFER_SIZE 10
typedef struct {
      ...
} item;

item buffer[BUFFER_SIZE];
int in = 0; // index of the next empty buffer
int out = 0; // index of the next full buffer
```

Remove() Method

```
while (true) {
      while (in == out)
           ; // do nothing — nothing to consume

      // remove an item from the buffer
      item = buffer[out];
      out = (out + 1) % BUFFER SIZE;
      return item;
}
```

- all empty? all full? ⇒Solution is "correct", but can only use BUFFER_SIZE-1 elements

Outline

Interprocess Communication (进程间通信, IPC)

- Shared-Memory systems
- Message-Passing Systems

Message-Passing Systems

- Message passing (消息传递)
  - provides a mechanism for processes to communicate and to synchronize their actions without sharing the same address space.
  - processes communicate with each other without resorting to shared variables
  - particularly useful in a distributed environmet.
- IPC facility provides at least two operations:
  1. send(message) — message size fixed or variable
  2. receive(message)
- If process P and Q wish to communicate, they need to:
  1. establish a communication link between them
  2. exchange messages via send/receive
- Implementation of communication link
  1. physical (e.g., shared memory, hardware bus)
  2. logical (e.g., logical properties)

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?

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 in this scheme
  - 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
- **Symmetry** VS **asymmetry**
  - send(P, message)
  - receive(id, message) - receive a message **from any process**

Indirect Communication

- Messages are directed and received from **mailboxes** (also referred to as **ports**)
  - Each mailbox has **a unique id** (such as POSIX message queues)
  - Processes can communicate only if they share a mailbox
  - Primitives are defined as:
    - send(A, message) — send a message to mailbox A
    - receive(A, message) — receive a message from mailbox A
- **Properties** of communication link in this scheme
  - Link established only if processes share a common mailbox
  - **A link may be associated with more than two processes**
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- **Mailbox sharing problem**
  - P1, P2, and P3 share mailbox A
  - P1, sends; P2 and P3 receive
  - Who gets the message?
- **Solutions to choose**
  - 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.

Indirect Communication

- Who is the owner of a mailbox?
  - a process
    - only owner can receive messages through its mailbox, others can only send messages to the mailbox.
    - when the process terminates, its mailbox disappears.
  - the OS
    - the mailbox is independent and is not attached to any particular process.
- Operations
  1. create a new mailbox
  2. send/receive messages through mailbox
  3. destroy a mailbox

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
- Difference combinations are possible.
  - If both are blocking ≡**rendezvous(集合点)**
- **The solution to PC problem** via message passing is trivial when we use blocking send()/receive().

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

Outline

- Example of IPC Systems
  - POSIX Shared Memory
  - Mach (by yourself)
  - Windows XP

Outline

POSIX API for shared memory

```
#include<sys/ipc.h>
#include<sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);

#include<sys/types.h>
#include<sys/shm.h>
void* shmat(int shmid, const void* shmaddr, int shmflg);
int shmdt(const void* shmaddr);
```

C program illustrating POSIX shared-memory API

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main(){
    int segment_id;
    char* shared_memory;
    const int size = 4096;

    segment_id = shmget(IPC_PRIVATE, size, S_IRUSR|
S_IWUSR);
    shared_memory = (char*) shmat(segment_id, NULL, 0);

    sprintf(shared_memory, "Hi there!");
    printf("%s\n",shared_memory);

    shmdt(shared_memory);
    shmctl(segment_id, IPC_RMID, NULL);
    return 0;
}
```

Two program using POSIX shared-memory: program1

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main(void) {
    key_t key;
    int shm_id;
    char * shm_addr;

    key=ftok(”.”,’m’);
    shm_id=shmget(key,4096,IPC_CREAT|IPC_EXCL|S_IRUSR|
S_IWUSR);

    shm_addr=(char*)shmat(shm_id,0,0);

    sprintf(shm_addr,”hello, this is 11111111\n”);
    printf(”111111: %s”,shm_addr);
    sleep(10);
    printf(”111111: %s”,shm_addr);
    shmdt(shm_addr);
    shmctl(shm_id,IPC_RMID,0);
    return 0;
}
```

Two program using POSIX shared-memory: program2

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main(void) {
    key_t key;
    int shm_id;
    char * shm_addr;

    key=ftok(".",'m');
    shm_id=shmget(key,4096,S_IRUSR|S_IWUSR);

    shm_addr=(char*)shmat(shm_id,0,0);

    printf("22222222:",shm_addr);
    sprintf(shm_addr,"this is 22222222\n");
    shmdt(shm_addr);
    return 0;
}
```

Outline

Outline

LPC in Windows XP

- Subsystems
  - **application** programs can be considered **clients** of the Windows XP **subsystems server**.
  - application programs communicate via a message-passing mechanism: **local procedure-call (LPC)** facility.
- Port object: two types
  - connection ports: named objects, to set up communication channels
  - communication ports
    - for small message, use the port' s message queue
    - for a larger message, use a section object, which sets up a region of shared memory.
      this can avoids data copying

LPC in Windows XP

- Local procedure calls in Windows XP.

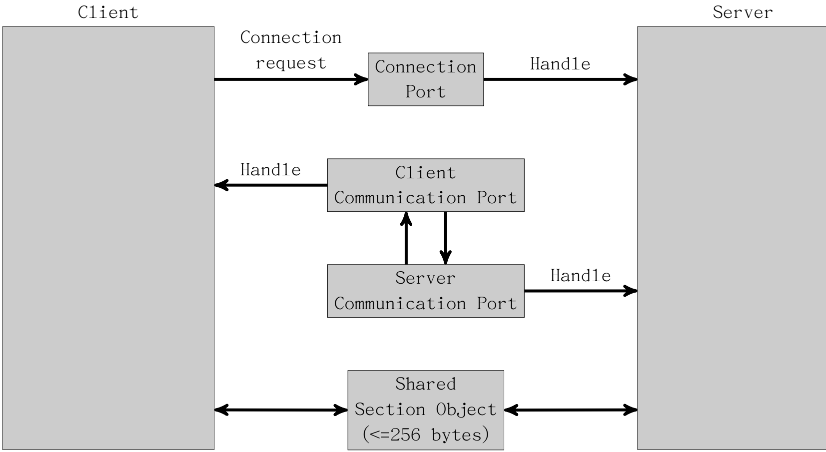


Figure 3.17 Local procedure calls in Windows XP.

Outline

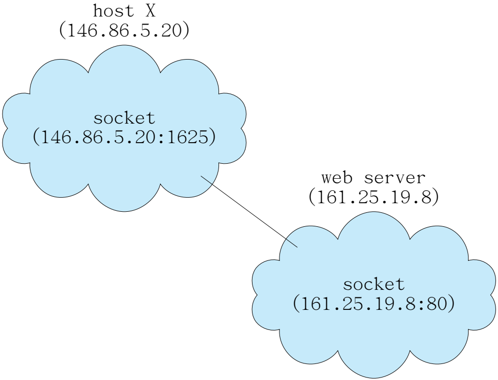
Communication in C/S Systems

Client-Server Communication

- Sockets（套接字）
- Remote Procedure Calls（远程过程调用，RPC）
- Remote Method Invocation（远程方法调用，RMI）（Java）

Sockets（套接字）

- 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

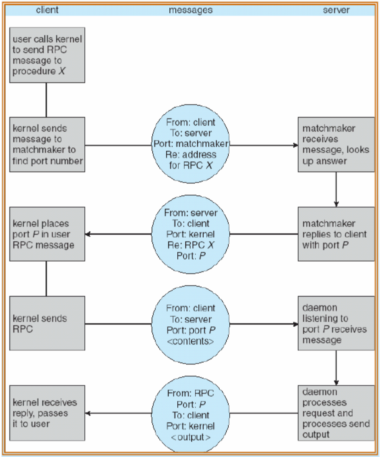


Remote Procedure Calls(远程过程调用，RPC)

- 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 peforms the procedure on the server.

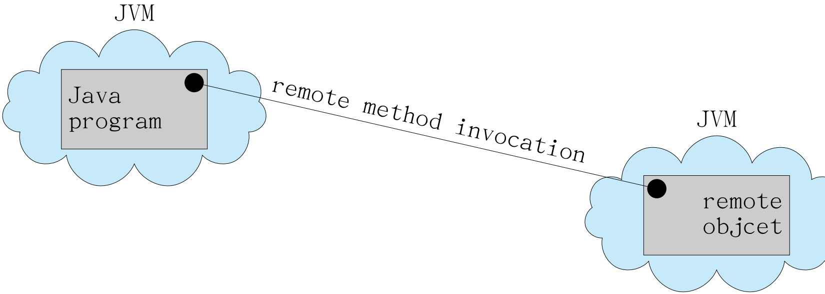
Remote Procedure Calls(远程过程调用，RPC)

- Execution of a remote precedure call (RPC)



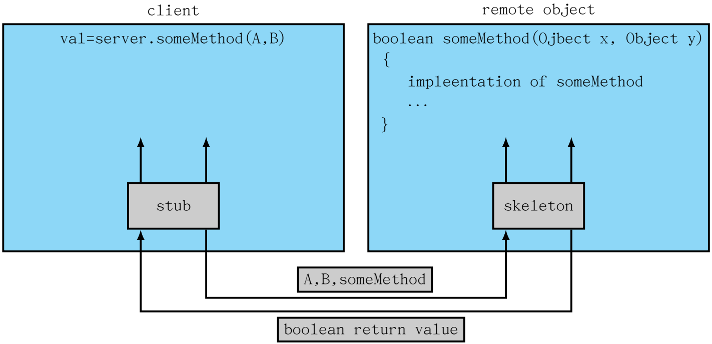
Remote Method Invocation(远程方法调用，RMI)

- 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.



Remote Method Invocation(远程方法调用，RMI)

- Marshalling Parameters



Outline

小结和作业

小结

多道程序技术和程序并发执行的条件

- 多道程序技术的难点
- Seriel execution of programs (程序的顺序执行)
- Concurrent execution of programs (程序的并发执行)

Process Concept

- the Processes
- Process State
- Process Control Block (PCB)

Process Scheduling

- Process Scheduling Queues
- Schedulers
- Context Switch(上下文切换)

Operation on processes

- Process Creation
- Process Termination

Interprocess Communication (进程间通信, IPC)

- Shared-Memory systems
- Message-Passing Systems

Example of IPC Systems

- POSIX Shared Memory
- Mach (by yourself)
- Windows XP

Communication in C/S Systems

小结和作业

阅读

- Read related code in Linux or uC/OS-II
- Subsubsection "An Example: Mach" of subsection "Examples of IPC Systems"
- Subsubsection "An Example: Windows XP" of subsection "Examples of IPC Systems"
- Subsection "Communication in Client-Server Systems"

作业 I

- 程序的顺序执行和并发执行有什么异同之处?
- 什么是Bernstein条件?
- 对于下面的语句:

  S_1: $a = 5 - x$;

  S_2: $b = a \cdot x$;

  S_3: $c = 4 \cdot x$;

  S_4: $d = b + c$;

  S_5: $e = d + 3$

  1. 画出前趋图
  2. 证明S_2和S_3是可以并发执行的，而S_3和S_4是不能并发执行的。

- 阅读至少2本操作系统相关书籍，给出这些书中关于进程的定义，
  要列出出处。

作业 II

- 阅读linux-0.11的内核代码，找到其进程数据结构加以分析。
  说明linux-0.11中进程的状态及其转换关系。
- 名词解释：
  - 长／中／短期调度
  - 多道程序度
  - IO密集型／CPU密集型
  - 进程上下文
- 如果一个main函数中连续调用三次fork()，那么该函数运行时在系统中实际上一共创建了多少个进程？为什么？请画出父子关系图。

谢谢！