操作系统原理与设计

第 3 章 Processes（进程）2

陈香兰

中国科学技术大学计算机学院

March 19, 2014

提纲

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.

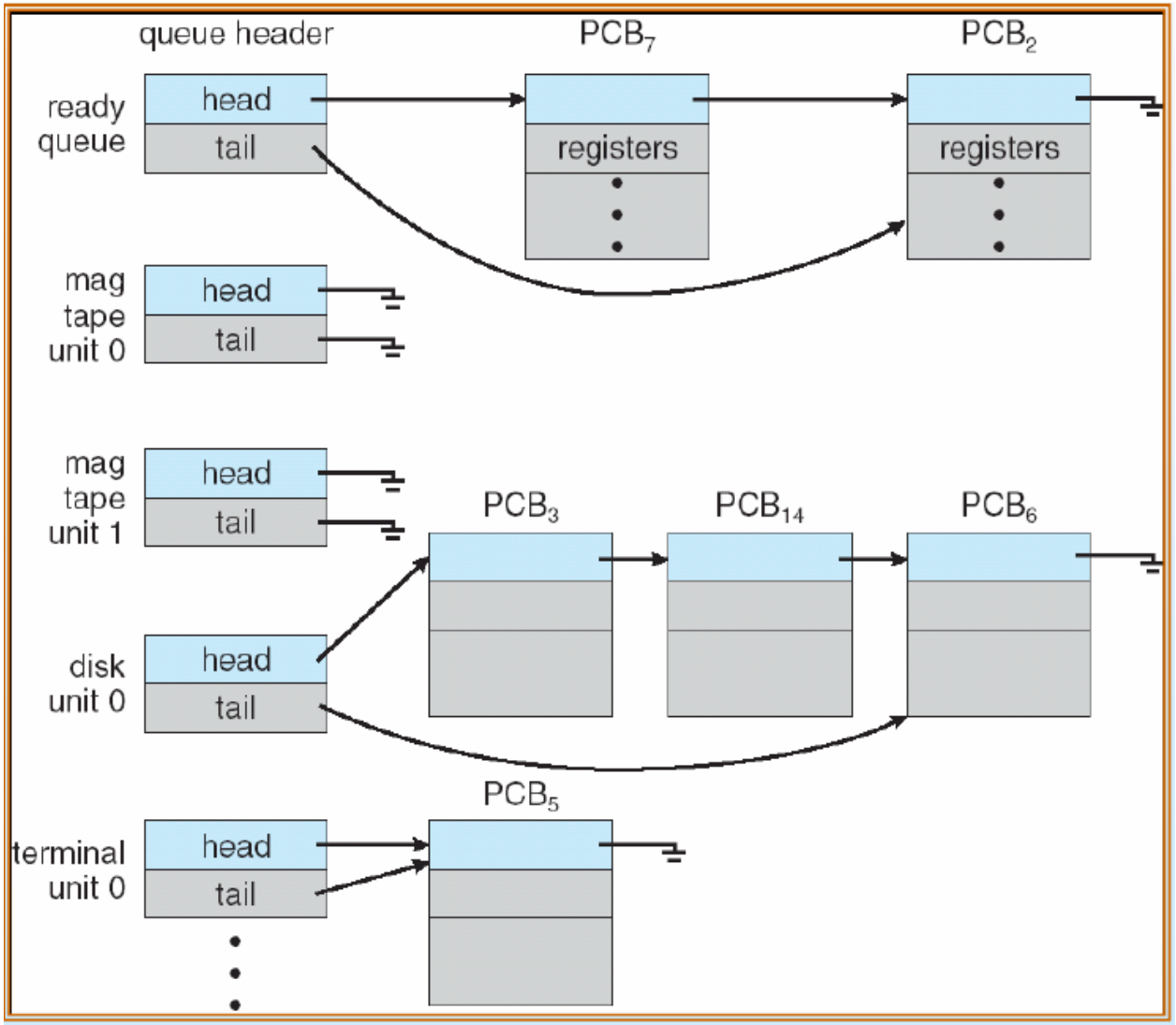
Outline

Process Scheduling Queues

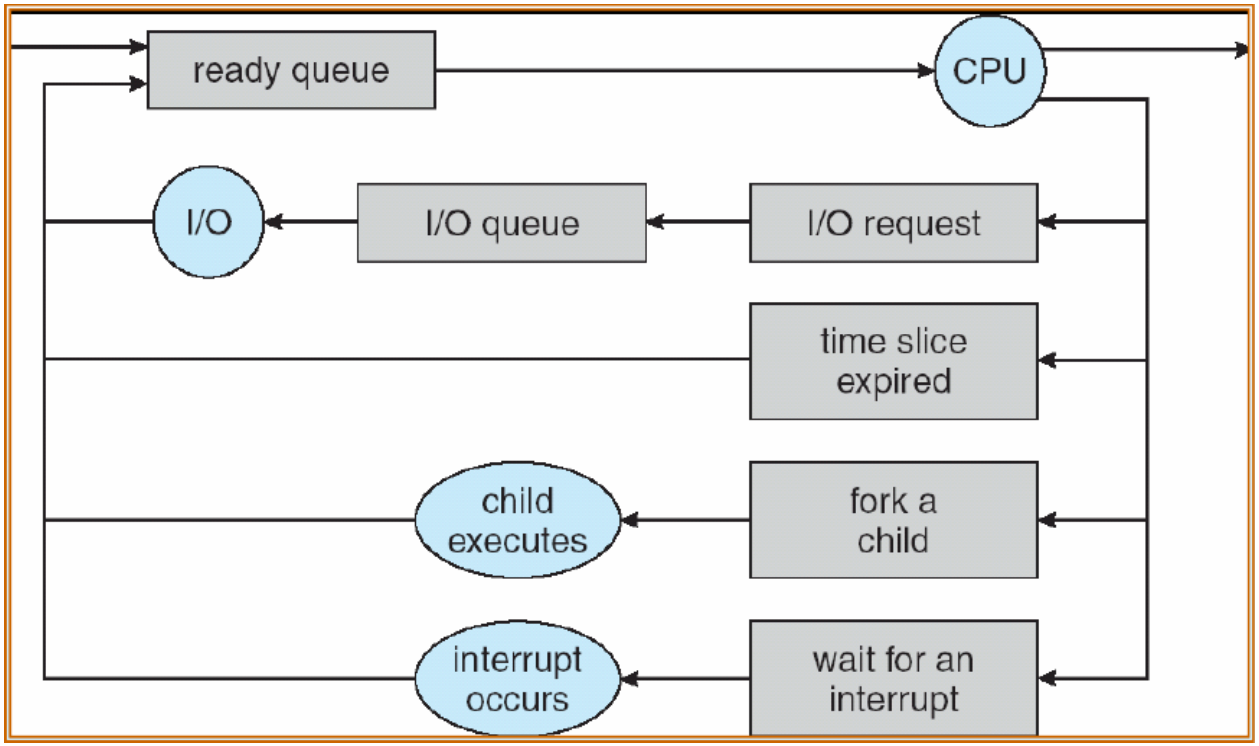
Processes migrate among the various queues

- **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 pri1mary **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

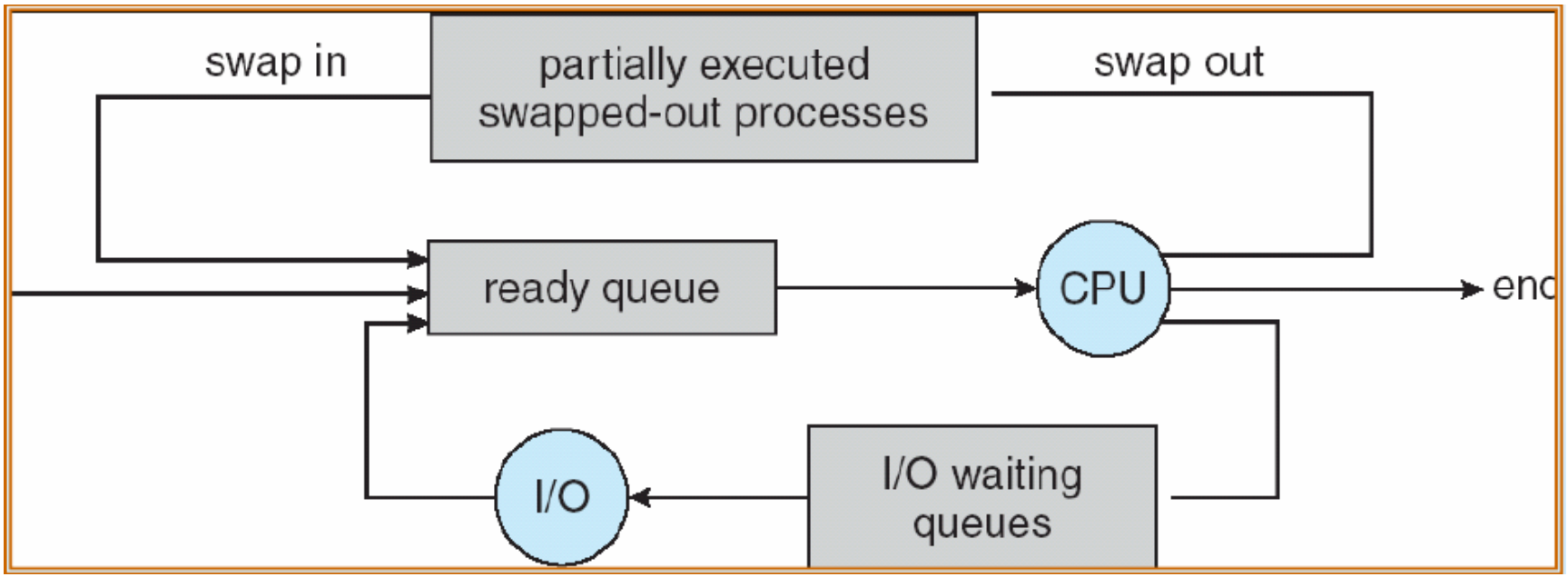
- spends more time doing computations; few very long CPU bursts

- **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) I

- **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**

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

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

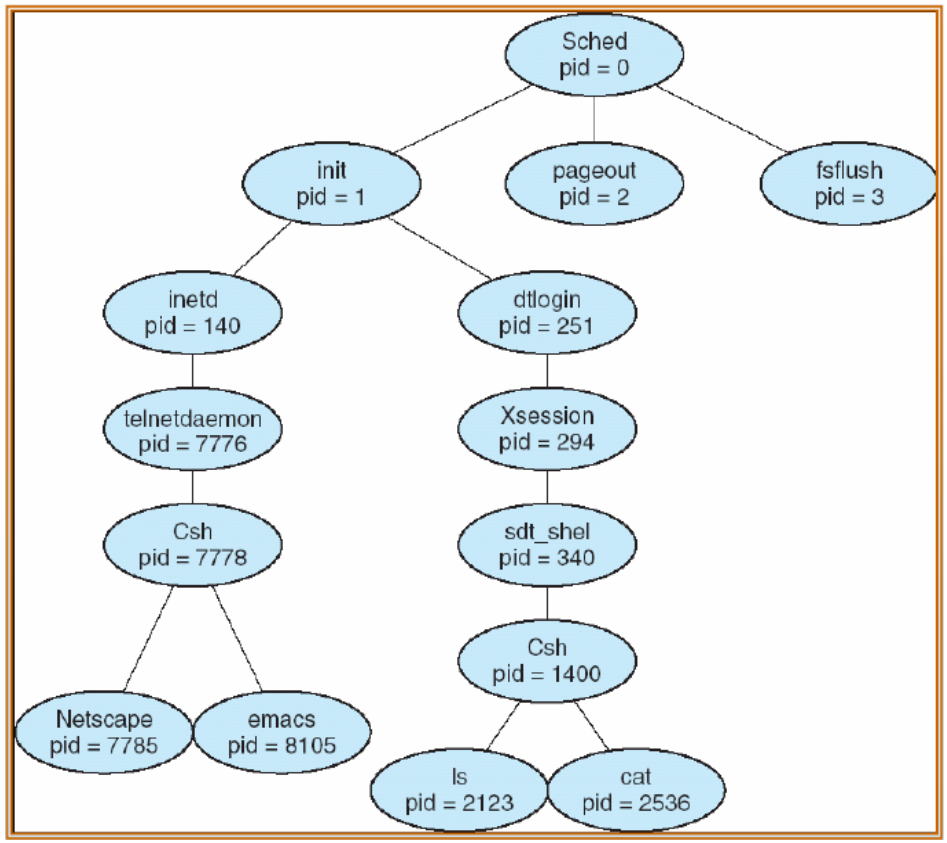
Process Creation I

- **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
```

Process Creation II



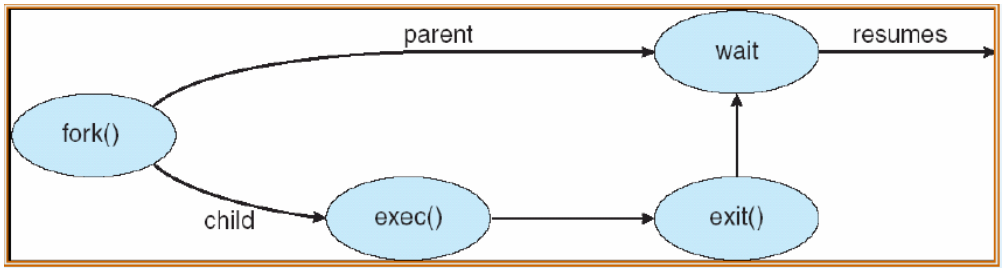
A tree of processes on a typical Solaris

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. 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. Termination can be caused by another process
   - Example: TerminateProcess() in Win32
3. Users could **kill** some jobs.

Process Termination

- **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**

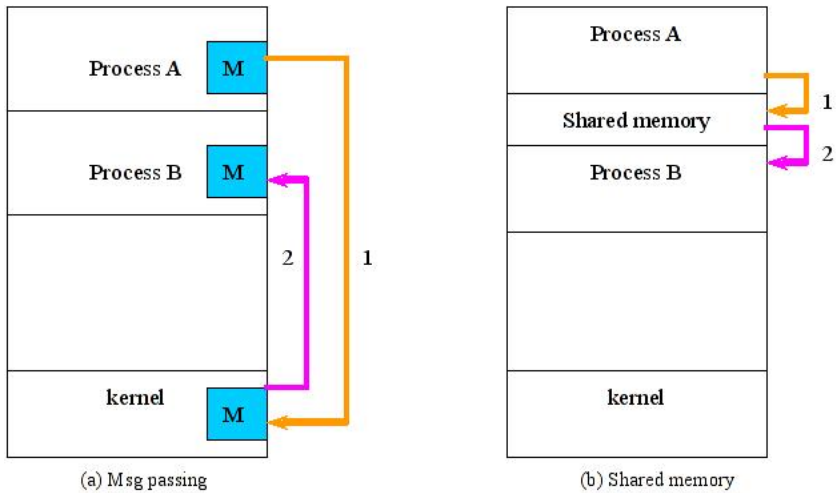
- UNIX Example:
  - exit(), wait()
  - If the parent terminates, all its children have assigned as their **new parent** the **init** process.

Interprocess Communication (进程间通信，IPC) I

- 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) II

- **Two fundamental models of IPC**
  - **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()
  - **Shared-memory (共享内存) model**
    - **faster** at memory speed via memory accesses.
    - system calls only used to establish shared memory regions



Outline

Shared-Memory systems

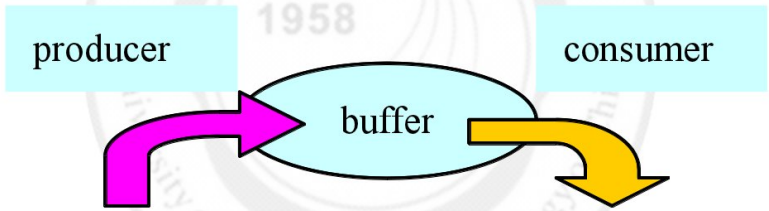
- Normally, the OS tries to **prevent one process from accessing another process's memory**.
- **Shared memory** requires that two or more processes agree to **remove this restriction**.
  - They can exchange information by **R/W data in the shared areas**.
  - **The form of data and the location** are **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**.

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:

$$\textit{complier} \xrightarrow{\textit{assembly code}} \textit{assembler} \xrightarrow{\textit{object models}} \textit{loader}$$

- **Shared-Memory solution**
  - a buffer of items shared by producer and consumer



- **Two types of buffers**
  - **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 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
```

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;
}
```

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

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**:
  - **send(message)** – message size fixed or variable
  - **receive(message)**
- If process P and Q wish to communicate, they need to:
  - **establish a communication link** between them
  - exchange messages via **send/receive**
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - 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**
  - **create** a new mailbox
  - **send**/**receive** messages through mailbox
  - **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

POSIX Shared Memory

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);
```

POSIX Shared Memory

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;
}
```

POSIX Shared Memory

Two program using shared memory: program1

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

int main(void) {
    key_t key;
    int shm_id;
    const int shm_size=4096;
    char * shm_addr;

    key=ftok(".",'m');
  shm_id=shmget(key,shm_size,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:");
    printf(shm_addr);
    sleep(10);
    printf("111111:");
    printf(shm_addr);
    shmdt(shm_addr);
    shmctl(shm_id,IPC_RMID,0);
    return 0;
}
```

POSIX Shared Memory

Two program using shared memory: program2

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

int main(void) {
    key_t key;
    int shm_id;
    const int shm_size=4096;
    char * shm_addr;

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

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

    printf("22222222:");
    printf(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.

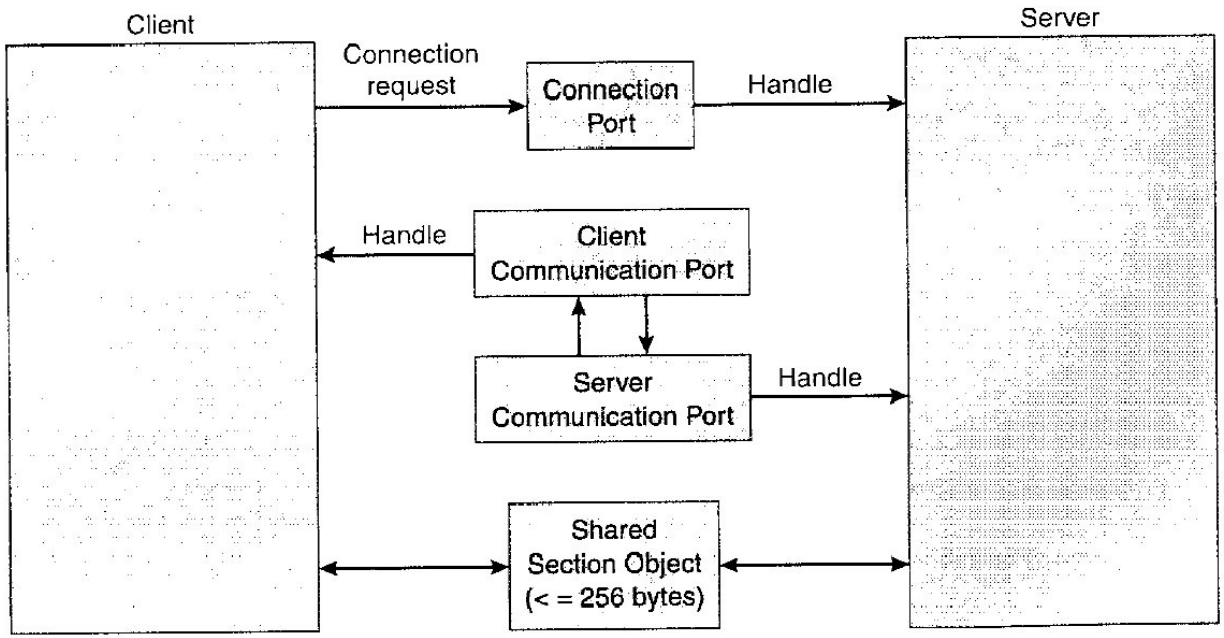


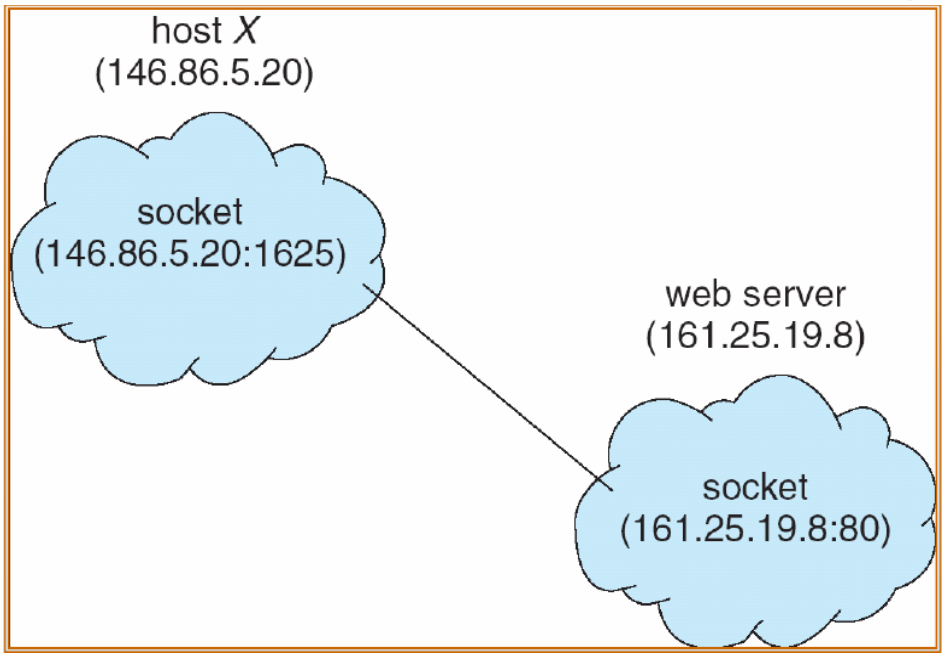
Figure 3.17 Local procedure calls in Windows XP.

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

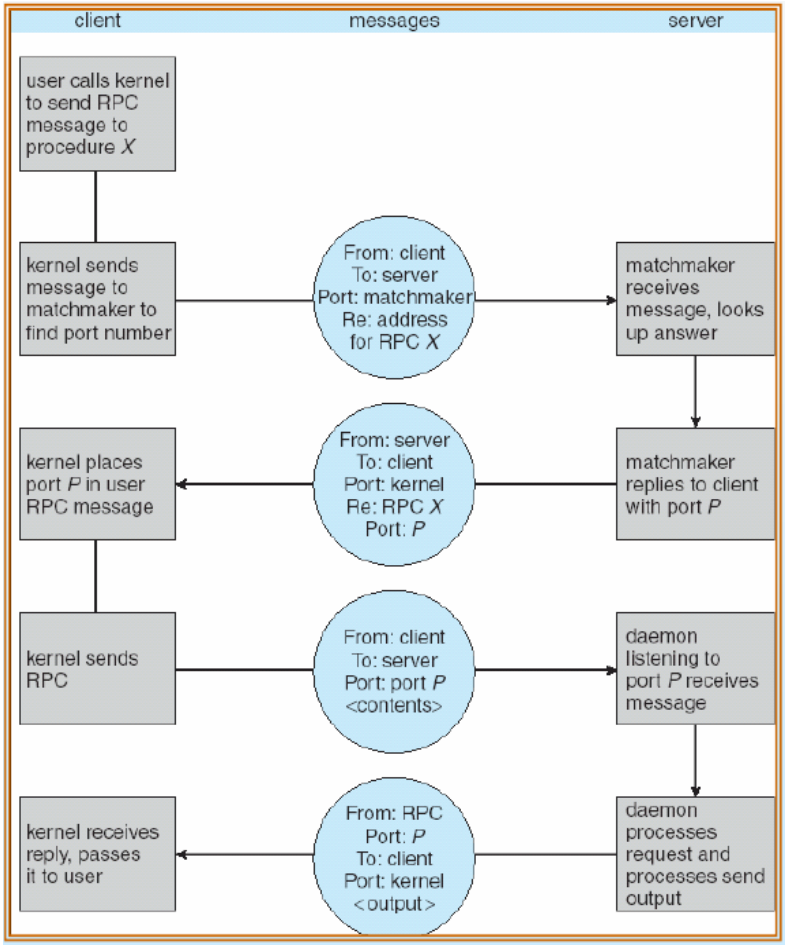


Remote Procedure Calls(远程过程调用, RPC)

- 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.

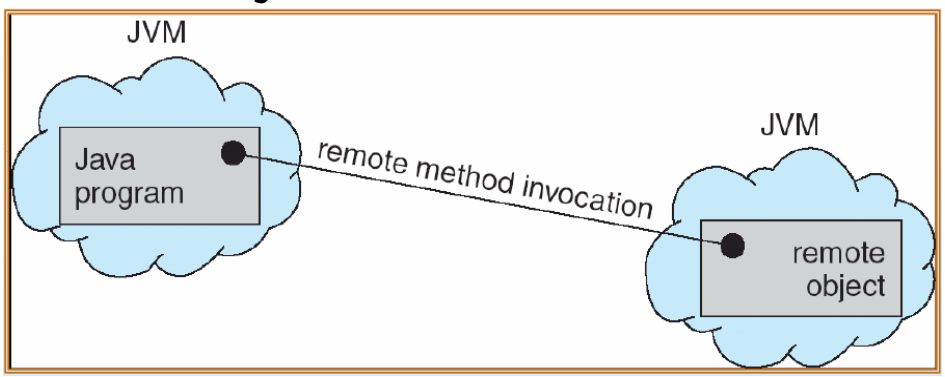
Remote Procedure Calls(远程过程调用, RPC)

- Execution of a remote precedure call (RPC)



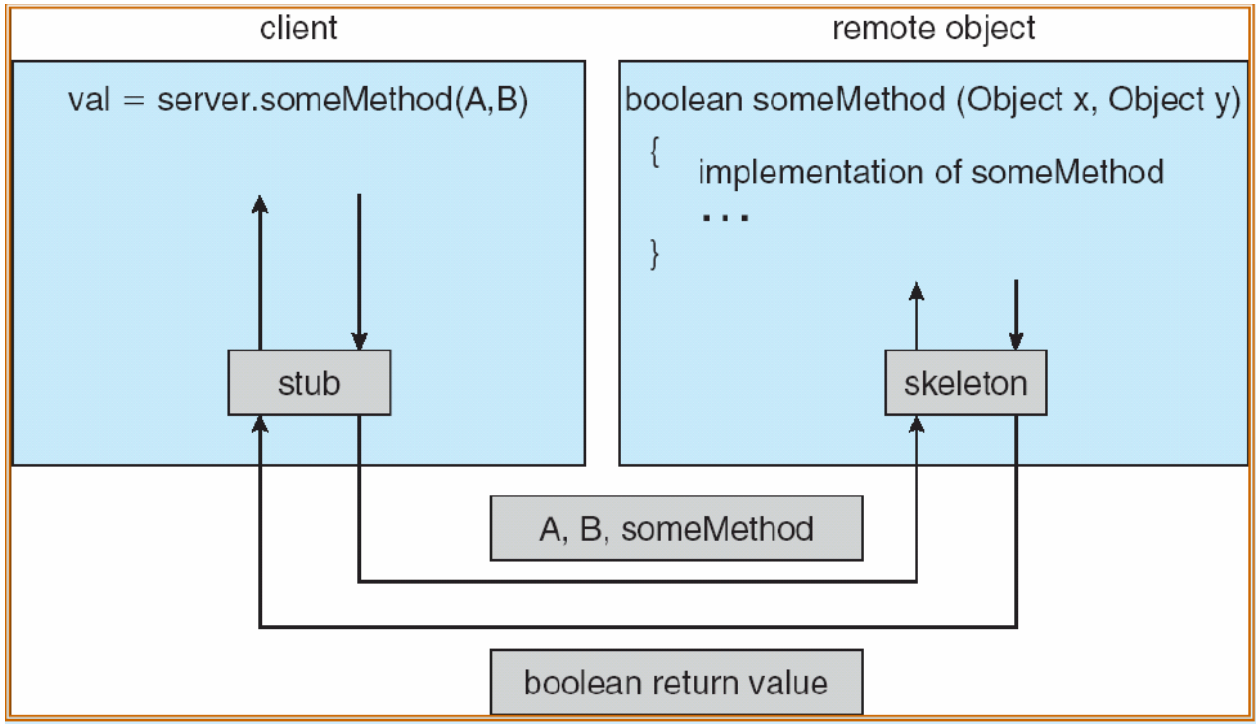
Remote Method Invocation(远程方法调用, RMI)

- 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



小结

1. Process Scheduling
   - Process Scheduling Queues
   - Schedulers
   - Context Switch(上下文切换)
2. Operation on processes
   - Process Creation
   - Process Termination
3. Interprocess Communication (进程间通信, IPC)
   - Shared-Memory systems
   - Message-Passing Systems
4. Example of IPC Systems
   - POSIX Shared Memory
   - Mach
   - Windows XP
5. Communication in C/S Systems
6. 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”

谢谢！