Chapter 3: Processes

Outline

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

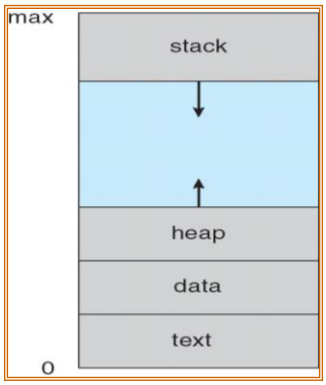
Process Concept

- **The name of CPU activities varies in different systems:**
  - Batch system – *jobs*.
  - Time-shared systems – *user programs* or *tasks*.
- **Textbook uses the terms *job* and *process* almost interchangeably, but prefers *process*.**
  - However, much of operating-system theory and terminology was developed during the batch era.
  - *Job* (such as job scheduling) would be used to avoid misunderstanding.

3

Process Concept (cont'd)

- **Process (active):**
  - A program (passive) in execution.
  - Execution must progress in **sequential** fashion.
- **A process in memory includes:**
  - ***Text section***: the program code.
  - ***Stack***: contains temporary data (local variable, function parameters …).
  - ***Data section***: contains global variables.
  - ***Heap***: used for dynamical memory allocation.
  - ***Program counter***: the address of next instruction.
  - ***Register context***



Process in memory

4

Process Concept (cont'd)

- **A program is not a process:**
  - A program is a ***passive*** entity; a list of instruction stored on disk.
  - A process is an ***active*** entry.
    - A program becomes a process when an executable file is loaded into memory.
- **Two (or more) processes may be associated with the same program.**
  - For example, a user may invoke many copies of the web browser program.
  - Each of these is a separate process.
    - **While the text sections are equivalent, the data, heap, and stack sections vary.**

5

Process State

- **The *state* of a process is defined by the current activity of that process.**
- **Each process may be in one of the following states:**
  - ***New***: the process is being created.
  - ***Running***: instructions are being executed.
  - ***Waiting***: the process is waiting for some event to occur.
    - Such as an I/O completion.
  - ***Ready***: the process is waiting to be assigned to a process.
  - ***Terminated***: the process has finished execution.
- **Only one process can be running on any processor at any instance.**
  - Many processes may be *ready* and *waiting*.

6

Process State (cont'd)

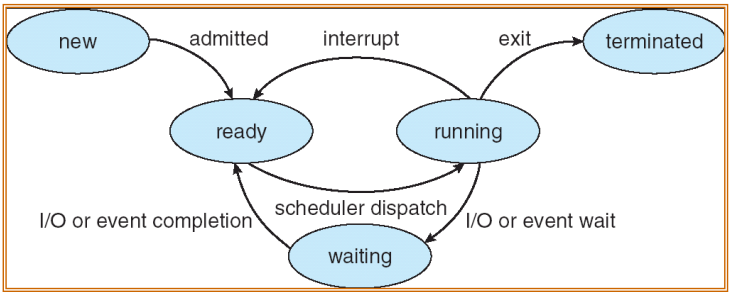
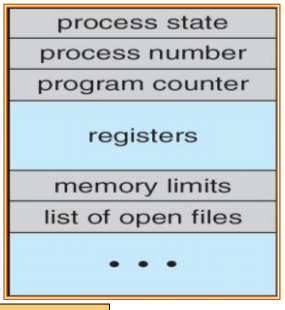


Diagram of Process State

7

Process Control Block

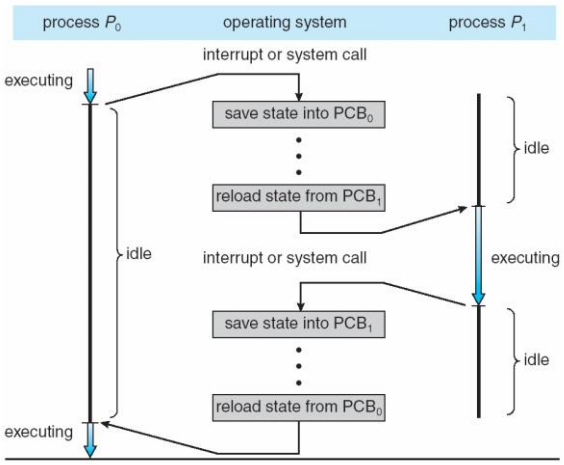
- ***Process control block*** **(PCB): a representation of a process in the operating system.**
- **Information associated with each PCB:**
  - **Process state**.
  - **Program counter**.
    - The address of next instruction.
  - **CPU registers**.
  - **CPU scheduling information**.
    - Process priority.
  - **Memory-management information**.
  - **Accounting information**.
    - E.g., amount of CPU time.
  - **I/O status information**.
    - Allocated devices, files, and so on.



Different processes have different PCB information.

8

CPU Switches From Process to Process



Threads

- **So far, we focus on *single-thread* process.**
  - **Only one task** can be performed at one time.
  - For example, a single-thread word program can not run the spell checker when the user simultaneously type in characters.
- **Most modern operating system allow *multiple-thread* execution.**
  - A process can have multiple threads of execution and thus can **perform more than one task at a time**.

10

Process Scheduling

- **Multiprogramming:**
  - To have some process running at all time.
  - To maximize CPU utilization.
- **Time sharing:**
  - To switch the CPU among processes so frequently.
  - Users can interact with each program while it is running.
- **To do so … we need …**
  - ***Process scheduler***:
    - Select an available process for execution on the CPU.
    - The rest have to wait until the CPU is free and can be rescheduled.

11

Scheduling Queues and Schedulers

- **The operating system usually contains a lot of queues (FIFO).**
  - Contain pointers to the first and final PCBs in the list (queue).
  - Each PCB includes a pointer field that points to the next PCB in the queue.
  - ***Job queue***:
    - Processes are initially spooled to a mass-storage device (e.g., a disk), where they are kept for later execution.
    - Job scheduler later selects processes from this pool and load them into memory for execution.
    - Often in batch system. **On some operating systems, job queue and job scheduler may be absent**.
      - Such as UNIX and MS Windows.
      - Just put every new process in memory for execution.

12

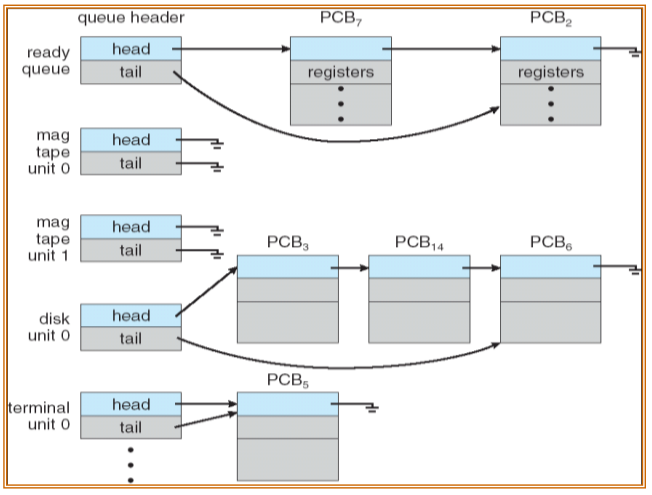
Scheduling Queues and Schedulers (cont'd)

- ***Ready queue***:
  - Contain the processes residing in main memory.
  - Those processes are **ready** and waiting to execute.

- **Operating systems also include other queues.**
  - ***Device queue***:
    - A list of processes waiting for a particular I/O device.
      - For instance, there are many processes make requests to a disk.

13

Scheduling Queues and Schedulers (cont'd)



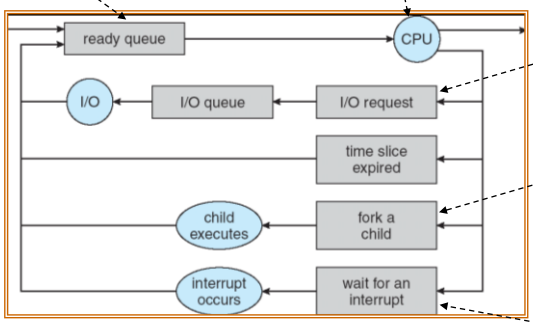
Ready Queue And Various I/O Device Queues

14

Scheduling Queues and Schedulers (cont'd)

A new process is initially put in the ready queue (**ready** state).

Eventually, it will be selected (dispatched) for execution (**running** state).



The process may issue an I/O request, and switches to the **waiting** state.

The process may create a new child process and wait for the child process's termination (**waiting** state).

The process may be removed from the CPU by an interrupt, and be put back in the ready queue (**ready** state).

Processes migrate among the various queues.

When the process terminates, it is removed from all Queues and has its PCB and resources deallocated.

15

Scheduling Queues and Schedulers (cont'd)

- **A process migrates among the various scheduling queues throughout its lifetime.**
  - The operating system must select process from these queues in some fashion.
- **Two main schedulers:**
  - ***Job scheduler*** (or **long-term** scheduler) – selects which processes should be brought into the ready queue.
  - ***CPU scheduler*** (or **short-term** scheduler) – selects which process should be executed next and allocates CPU.
- **The primary difference between the schedulers: frequency of execution.**
  - Often, the short-term scheduler executes at leas once every 100 ms.
  - The long-term scheduler executes much less frequently.

16

Scheduling Queues and Schedulers (cont'd)

- **The short-term scheduler must be fast.**
  - If it takes 10 ms to decide to execute a process for 100 ms, then 10 / (10 + 100) = 9% of the CPU is being wasted for scheduling the work.
- **However, because of the longer interval between executions, the long-term scheduler can afford to take more time to decision.**
- **The long-term scheduler controls the degree of multiprogramming.**
- **In general, processes can be described as either *I/O bound* or *CPU bound*.**
  - A **I/O-bound process** spends more of its time doing I/O.
  - A **CPU-bound process** uses more of its time doing computations.
  - A good long-term scheduler should select a good process **mix** of I/O-bound and CPU-bound processes that the system will be balanced.

17

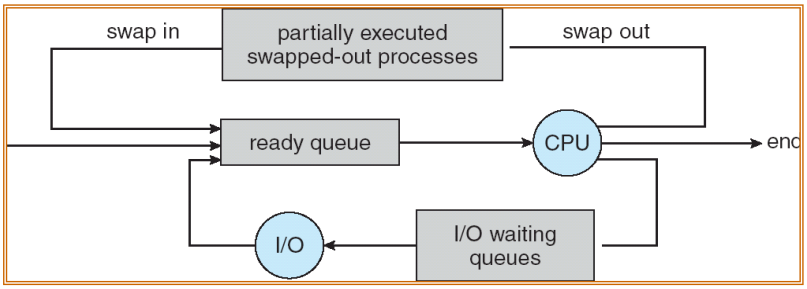
Scheduling Queues and Schedulers (cont'd)

- **For systems with no long-term schedulers:**
  - The stability of the systems may depend on the self-adjusting nature of human users.
    - If the performance declines to unacceptable levels on a multi-users system, some users will simply quit.
- **Some operating systems may introduce a *medium-term scheduler*.**
  - It **removes** processes from memory and thus reduce the degree of multi-programming (**swap out**).
  - Later, the processes can be **reintroduced** into memory for execution (**swap in**).
  - Swapping may improve the process mix.
    - Or for better memory management.

18

Scheduling Queues and Schedulers (cont'd)

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



Addition of medium-term scheduling to the queueing diagram.

19

Multitasking in Mobile Systems

- **Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended**
- **Due to screen real estate, user interface limits iOS provides for a**
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- **Android runs foreground and background, with fewer limits**
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

Context Switch

- **When CPU switches to another process, the system must save the context of the old process and load the saved context for the new process.**
  - E.g., interrupts cause the operating system to change a CPU from its current task to run a kernel routine.
- **Such a switching is known as a *context switch*.**
- **The context is represented in the PCB of the process.**
  - The value of the CPU registers.
  - The process state.
  - Program counter.
  - …

21

Context Switch (cont'd)

- **Context-switch time is overhead; the system does no useful work while switching.**
- **Dependent on hardware support.**
  - Some processors provide multiple sets of registers.
    - Multiple contexts loaded at once
    - A context switch simply requires changing the pointer to the current register set.

22

Operations on Processes

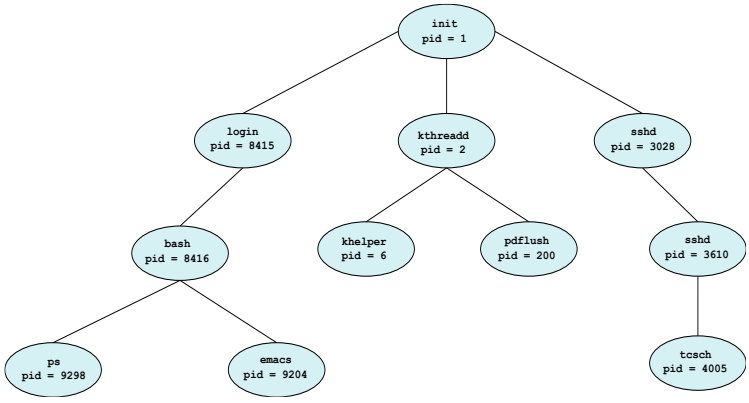
23

Process Creation

- **A process may create several new processes, via a create-process system call.**
- **The creating process is called a parent process, and the new processes are called the children of that process.**
  - New processes may in turn create other processes, forming a **tree** of processes.
- **Most operating system (UNIX and MS Windows) identify processes according to a unique process identifier (pid).**
  - Usually an integer number
- **Resource sharing options**
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- **Execution options**
  - Parent and children execute concurrently
  - Parent waits until children terminate

24

A Tree of Processes in Linux



systemd is a system and service manager for Linux operating systems. When run as first process on boot (as PID 1), it acts as init system that brings up and maintains user space services.

Process Creation (cont'd)

- **On UNIX, the `ps` command can be used to obtain the information of all process.**
- **You may use pstree to see a tree of processes**
  - pstree –np
  - pstree –p 1

26

Process Creation (cont'd)

- **When parent process holds certain resources (files, I/O devices).**
  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no resources.
- **When a process creates a new process, two possibilities exist in terms of execution:**
  - Parent and children execute concurrently.
  - Parent waits until children terminate.
- **There are also two possibilities in terms of the address space of the new process:**
  - Child duplicate of parent.
  - Child has **a program loaded into it**.

27

Process Creation (cont'd)

- **UNIX examples:**
  - **`Fork()`** system call creates new process (POSIX).
  - The new process consists of **a <u>copy</u> of the address space of the original process**.
  - Both process **continue (concurrently) execution** at the instruction after the `fork()`.
  - But … how to distinguish?
    - The return code for the `fork()` is zero for the new (child) process.
    - The process identifier (PID) of the child is returned to the parent.

28

Process Creation (cont'd)

fork_test.c

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

int main()
{
    pid_t  pid;

    pid = fork(); /* fork another process */
    if (pid < 0) { /* error occurred */
           printf("Fork Failed\n");
           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\n");
           exit(0);
    }
}
```

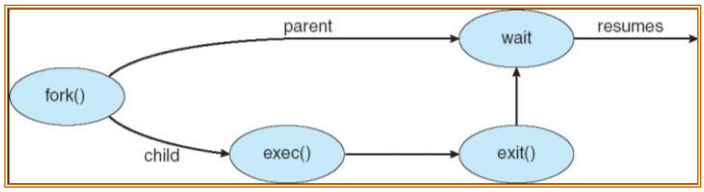
29

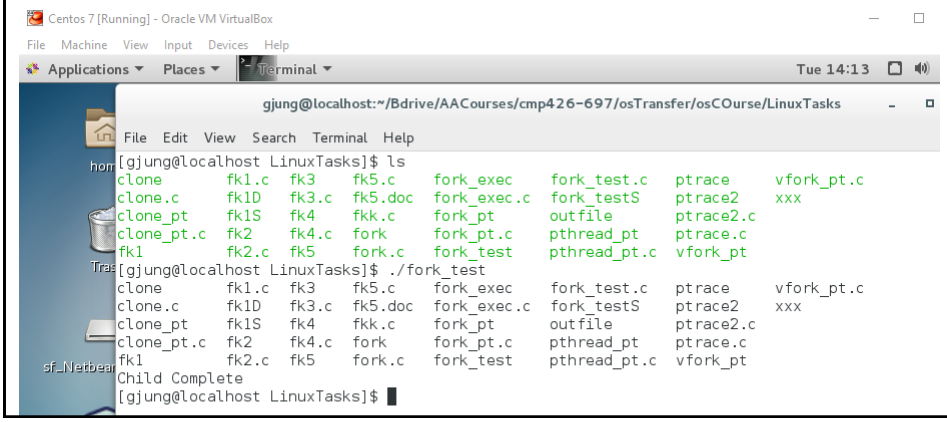
Process Creation (cont'd)

- Typically, the **`exec()`** (or its family, such as **`execlp()`**) system call (POSIX) is used after a `fork()` by **one of** the two processes to **replace the process's memory space with a new program**.
  - For example, the child process overlays its address space with UNIX command `/bin/ls` using the `execlp()` system call.
- The parent can wait for the child process to complete with the **`wait()`** system call (POSIX).
  - When the child process completes, the parent process resumes from the call to `wait`.
- **`exit()`** system call: cause normal process termination (POSIX).

30

Process Creation (cont'd)





Process Termination

- **Process executes last statement and asks the operating system to delete it by using the `exit()` system call.**
  - Can return a status value (typically an integer) to its parent process (via the `wait()` system call).
  - Process' resources are deallocated by operating system.

- **Parent may terminate execution of children processes (`kill()` or abort()) when:**
  - 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***.

Creating a Separate Process via Windows API

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
      "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
      NULL, /* don't inherit process handle */
      NULL, /* don't inherit thread handle */
      FALSE, /* disable handle inheritance */
      0, /* no creation flags */
      NULL, /* use parent's environment block */
      NULL, /* use parent's existing directory */
      &si,
      &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Multiprocess Architecture – Chrome Browser

- **Many web browsers ran as single process (some still do)**
  - If one web site causes trouble, entire browser can hang or crash
- **Google Chrome Browser is multiprocess with 3 different types of processes:**
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

Interprocess Communication

35

Cooperating Processes

- **Processes may be independent processes or cooperating processes.**
  - **Independent** process cannot affect or be affected by the execution of another process.
  - **Cooperating** process can affect or be affected by the execution of another process.
- **Reasons for process cooperation:**
  - Information sharing.
  - Computation speed-up.
    - Can be achieved only if the system have multiple CPUs.
  - Modularity.
  - Convenience.
    - Users may work on many tasks (or sub-tasks) at the same time.

36

Cooperating Processes (cont'd)

- ***Interprocess communication*** **(IPC) is a mechanism that allows cooperating processes to exchange data and information.**
- **Two typical models of IPC:**
  - ***Shared memory***.
    - A region of memory is shared by cooperating processes.
    - Processes can exchange information by reading and writing data to the region.
  - ***Message passing***.
    - Communication takes place by means of messages exchanged between the processes.
  - Both of the models are common and implemented in many operating systems.

37

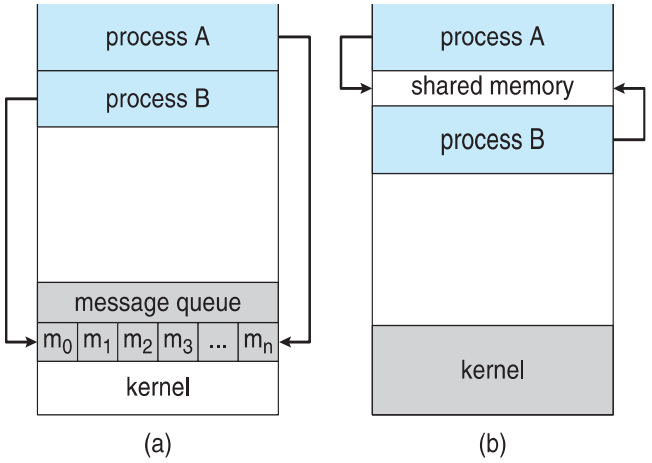
Cooperating Processes (cont'd)

- **Message passing is useful for exchanging smaller amounts of data.**
  - Because **no conflicts** need be avoided.
  - Usually **slow**, because message-passing systems are generally implemented as system calls.
    - Require the more time-consuming task of kernel intervention.
- **Shared memory allows maximum speed and convenience of communication.**
  - It can be done at memory speeds when within a computer.
  - Is **faster** than message passing.
    - Once shared memory is established, all accesses are treated as routine memory access.
    - No assistance from the kernel is required.

38

Cooperating Processes (cont'd)

(a) Message passing. (b) shared memory.



Shared-Memory Systems

- **A process creates a shared-memory region residing in its address space.**
- **Other processes that wish to communicate using this shared-memory segment must attach it to their address space.**
- **Then, they can exchange information by reading and writing data in the shared area.**
- **The processes are responsible for ensuring that they are not writing to the same location simultaneously.**
  - Chapter 5 will discuss synchronization to synchronize concurrent accesses.

40

Shared-Memory Systems (cont'd)

- **Shared memory as a *producer-consumer problem*:**
  - **producer** process produces information that is consumed by a **consumer** process.
  - A **buffer** (shared memory) can be filled by the producer and emptied by the consumer.
- **Two types of buffers:**
  - *unbounded-buffer* places no practical limit on the size of the buffer.
    - The consumer may have to wait for new items.
    - But the producer can always produce new items.
  - *bounded-buffer* assumes that there is a fixed buffer size.
    - The consumer may have to wait for new items.
    - The producer have to wait if the buffer is full.

41

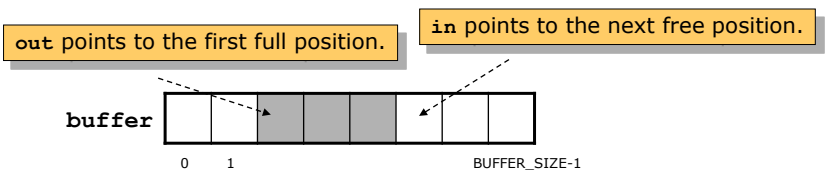
Shared-Memory Systems (cont'd)

- **Implementation:**

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

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- **The buffer is implemented as a circular array:**
  - `in=(in++)%BUFFER_SIZE`
  - `out=(out++)%BUFFER_SIZE`
- **At most `BUFFER_SIZE-1` items in the buffer at the same time.**
- **Empty: `in == out`**
- **Full: `((in+1)%BUFFER_SIZE) == out`**



42

Shared-Memory Systems (cont'd)

- **The producer process:**

```
item nextProduced;

while (true) {
    /* Produce an item */
    while (((in = (in + 1) % BUFFER_SIZE)  == out)
                ;   // do nothing
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER SIZE;
}
```

- **The consumer process**

```
item nextConsumed

while (true) {
    while (in == out)
                ;   /* do nothing
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER SIZE;
}
```


POSIX Shared Memory Example

- **A process must first create a shared memory segment using the `shmget()` system call (Shared Memory GET).**

set to `IPC_PRIVATE` to create a new shared-memory segment.

Size in bytes of the shared memory

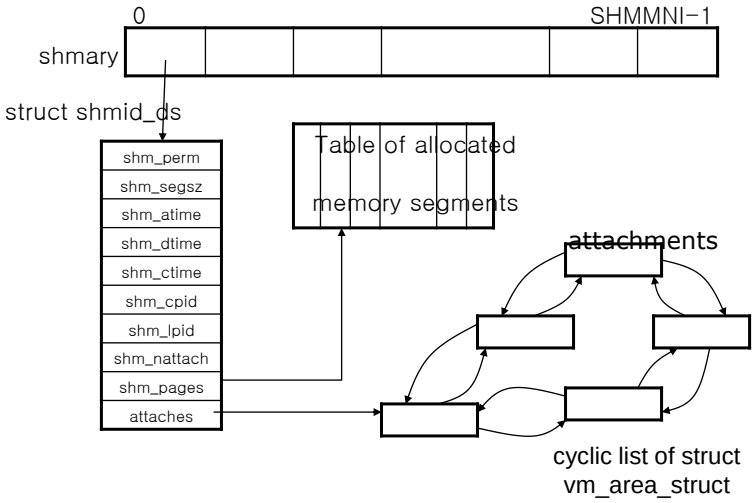
```
segment_id = shmget(IPC_PRIVATE, size,
                    S_IRUSR | S_IWUSR);
```

Return an integer identifier for the shared-memory segment.

The permission, S_IRUSR | S_IWUSR indicates that the owner may read or write.

POSIX Shared Memory Example (cont'd)

http://www.tldp.org/LDP/lpg/node68.html



POSIX Shared Memory Example (cont'd)

structure shmid_ds in "include/linux/shm.h"

```
struct shmid_ds {
    struct ipc_perm shm_perm;     /* operation perms */
    int shm_segsz;                /* size of segment (bytes) */
    __kernel_time_t shm_atime;    /* last attach time */
    __kernel_time_t shm_dtime;    /* last detach time */
    __kernel_time_t shm_ctime;    /* last change time */
    __kernel_ipc_pid_t shm_cpid; /* pid of creator */
    __kernel_ipc_pid_t shm_lpid; /* pid of last operator */
    unsigned short shm_nattch;    /* no. of current attaches */
    unsigned short shm_unused;    /* compatibility */
    void *shm_unused2;            /* ditto - used by DIPC */
    void *shm_unused3;            /* unused */
};
```

POSIX Shared Memory Example (cont'd)

- **Processes that wish to access a shared-memory segment must attach it to their address space using the `shmat()` (Shared Memory Attach) system call.**

A pointer, points to the beginning location of the attached shared memory.

The integer identifier of the shared-memory segment being attached.

```
shared_memory = (char *) shmat(segment_id,
                               NULL, 0);
```

Cast it as a character string.

A location where the shared memory will be attached, if NULL, the system chooses a suitable address at which to attach the segment.

Mode flag: 0 allows both reads and writes to the shared region.

POSIX Shared Memory Example (cont'd)

- **Then, the process can access the shared memory as a routine memory access using the pointer returned from `shmat()`.**

```
sprintf(shared_memory, "hello");
```

- **Other processes sharing this segment would see the updates to the shared memory segment.**

POSIX Shared Memory Example (cont'd)

- **When a process no longer requires access to the shared-memory segment, it detaches the segment from its address space.**
  - `shmdt(shared_memory);`

Pass the pointer of the shared-memory region to detach.

- **Finally, a shared-memory segment can be removed from the system with the `shmctl()` system call.**
  - `Shmctl(segment_id, IPC_RMID, NULL);`

The integer identifier of the shared-memory segment.

Destroy the shared segment

49

POSIX Shared Memory Example (cont'd)

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

int main()
{
  /* the identifier for the shared memory segment */
  int segment_id;
  /* a pointer to the shared memory segment */
  char* shared_memory;
  /* the size (in bytes) of the shared memory segment */
  const int segment_size = 4096;

  /* allocate  a shared memory segment */
  segment_id = shmget(IPC_PRIVATE, segment_size, S_IRUSR | S_IWUSR);

  /* attach the shared memory segment */
  shared_memory = (char *) shmat(segment_id, NULL, 0);

  /* write a message to the shared memory segment   */
  sprintf(shared_memory, "Hi there!");

  /* now print out the string from shared memory */
  printf("*%s\n", shared_memory);

  /* now detach the shared memory segment */
  shmdt(shared_memory);

  /* now remove the shared memory segment */
  shmctl(segment_id, IPC_RMID, NULL);

  return 0;
}
```

```
[gjung@localhost ShMemThreads]$ gcc -o shm_test shm_test
[gjung@localhost ShMemThreads]$ ./shm_test
*Hi there!
[gjung@localhost ShMemThreads]$
```

50

POSIX Shared Memory Example -- producer

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

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

POSIX Shared Memory Example -- consumer

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

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Message-Passing Systems

- **Processes communicate with each other without resorting to shared variables.**
- **The communication actions are synchronized.**
- **Useful in a distributed (network) environment.**

- **A message-passing facility provides at least two operations:**
  - ***send(message)***
  - ***receive(message)***
  - message size can be **fixed** or **variable**.
    - **fixed-sized messages** are easy to implement, but makes the programming task difficult.
    - **Variable-sized messages** require a more complex system-level implementation, but the programming task becomes simpler.

53

Message-Passing Systems (cont'd)

- **Processes that want to communicate must have a way to refer to each other.**
- ***Direct-symmetry-communication*:**
  - processes that want to communicate must explicitly name the **recipient** or **sender** of the communication.
  - `send(P, message)` – send a message to process `P`.
  - `receive(Q, message)` – receive a message from process `Q`.
- ***Direct-asymmetry-communication*:**
  - Only the sender names the recipient.
  - `send(P, message)` – send a message to process `P`.
  - `receive(id, message)` – receive a message from **any** process; the variable `id` is set to the name of the sender.

54

Message-Passing Systems (cont'd)

- **The disadvantage in both of these *hard-coding* schemes (symmetric and asymmetric).**
  - Changing the identifier of a process may necessitate examining all other processes.
  - All references to the old identifier must be modified to the new identifier.
- ***Indirect communication*:**
  - The messages are sent to and received from ***mailboxes***, or ***ports***.
  - Each mailbox has a unique identification.
  - A process can communicate with some other process via **a number of** different mailboxes.
  - A mailbox may be owned either by a process of by the operating system.
  - `send(A, message)` – send a message to mailbox `A`.
  - `receive(A, message)` – receive a message from mailbox `A`.

55

Message-Passing Systems (cont'd)

- **In indirect communication scheme, a communication link has the following properties:**
  - A link is established between a pair of processes only if both members of the pair have a shared mailbox.
  - A link may associated with associated with more than two processes.
  - Between each pair of communicating processes, there may be a number of different links.

56

Message-Passing Systems (cont'd)

- **A practical problem of 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.

57

Message-Passing Systems (cont'd)

- **There are different design options for implementing the `send()` and `receive()` primitives.**
  - **Blocking** (***synchronous***) **send**: the sending process is blocked until the message is received by the receiving process or by the mail box.
  - **Nonblocking** (***asynchronous***) **send**: the sending process sends the message and resumes operation.
  - **Blocking receive**: the receiver blocks until a message is available.
  - **Nonblocking receive**: the receiver retrieves either a valid message or a null.
  - Different combinations of `send()` and `receive()` are possible.
    - E.g., **rendezvous**: `send()` and `receive()` are blocking, a trivial solution to the producer-consumer problem.

58

Message-Passing Systems (cont'd)

- ***Buffering*:**
  - Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary **queue**.
  - Queue can be:
    - **Zero capacity**: can not have any messages waiting in it.
      - The sender must block until the recipient receives the message.
    - **Bounded capacity**: the queue has finite length *n*.
      - If the queue is full, the sender must block until space is available in the queue.
    - **Unbounded capacity**: the sender never blocks.

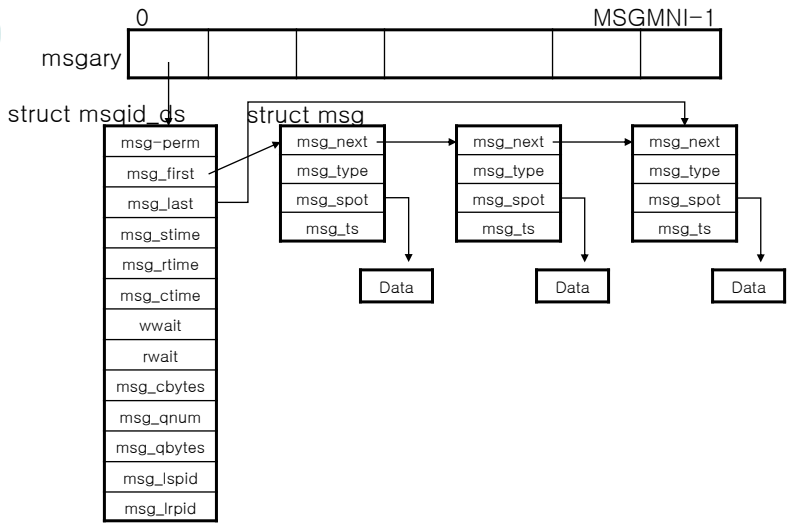
59

Message Based Communication-Linux

- **commnunicating with messages**
  - Processes can
    - send messages to the message queues.
    - receive messages from the message queues.
  - FIFO like
    - Messages are received in the same order in which they are entered in the message queue.
  - struct msqid_ds
    - Linux uses the structure msqid_ds to handle the message queues.
  - Below operations are used to handle a message queue.
    - msgget(), msgsnd(), msgrcv(), msgctl()

60

message queues



61

msqid_ds

- http://tldp.org/LDP/lpg/node32.html
- **structure msqid_ds** in "include/linux/msg.h"

```
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first;          /* first message on queue */
    struct msg *msg_last;           /* last message in queue */
    __kernel_time_t msg_stime;      /* last msgsnd time */
    __kernel_time_t msg_rtime;      /* last msgrcv time */
    __kernel_time_t msg_ctime;      /* last change time */
    struct wait_queue *wwait;
    struct wait_queue *rwait;
    unsigned short msg_cbytes;      /* current number of bytes on queue */
    unsigned short msg_qnum;        /* number of messages in queue */
    unsigned short msg_qbytes;      /* max number of bytes on queue */
    __kernel_ipc_pid_t msg_lspid;/* pid of last msgsnd */
    __kernel_ipc_pid_t msg_lrpid;/* last receive pid */
};
```

62

msqid_ds (cont'd)

- http://tldp.org/LDP/lpg/node31.html
- **structure msg** in "include/linux/msg.h"

```
struct msg {
   struct msg *msg_next;        /* next message on queue */
   long  msg_type;
   char *msg_spot;              /* message text address */
   time_t msg_stime;            /* msgsnd time */
   short msg_ts;                /* message text size */
};
```

63

Example- mserver.c mclient.c

- **mserver.c**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <sys/errno.h>
#include <stdio.h>
#include <stdlib.h>

#define MSGKEY 75
#define ACK "msgserv received the following message:  "

struct msgform {
  long mtype;
  char mtext[2048];
} sndbuf, rcvbuf, *msgp;

int msgid;
```

64

mserver.c (cont'd)

```
main()
{
 //  extern int errno;
  int i, rtrn;

  msgid = msgget(MSGKEY, 0777|IPC_CREAT);

  for(;;) {
    msgp = &rcvbuf;
    msgrcv(msgid, msgp, 2048, 1, 0);
    printf("\n%s\n", rcvbuf.mtext);

    msgp = &sndbuf;
    strcpy(sndbuf.mtext, ACK);
    strcat(sndbuf.mtext, rcvbuf.mtext);
    msgp->mtype = 10;
    rtrn=msgsnd(msgid, msgp, sizeof(sndbuf.mtext), 0);
    if (rtrn == -1) {
      printf("\n msgsnd() system call failed, Error # = %d\n", errno);
      exit(0);
    }
  }
}
```

65

mclient.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

#define MSGKEY 75


struct msgform {
  long mtype;
  char mtext[2048];
} sndbuf, rcvbuf, *msgp;
```

66

mclient.c (cont'd)

```
main()
{ int i, c, msgid;
  int rtrn, msgsz;

  msgid = msgget(MSGKEY, 0777);
  msgp = &sndbuf;
  msgp->mtype = 1;
  printf("\n Enter your message to be delivered to the server:\n");
  for (i = 0; ((c=getchar())!=EOF); i++)
    sndbuf.mtext[i] = c;
  msgsz = i + 1;

  rtrn = msgsnd(msgid, msgp, msgsz, 0);
  if (rtrn == -1) {
    printf("\n msgsnd() system call failed, error # = %d\n", errno);
    exit(0);
  }

  msgp = &rcvbuf;
  msgrcv(msgid, msgp, 2048, 10, 0);
  printf("\n%s\n", rcvbuf.mtext);
}
```

67

Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer

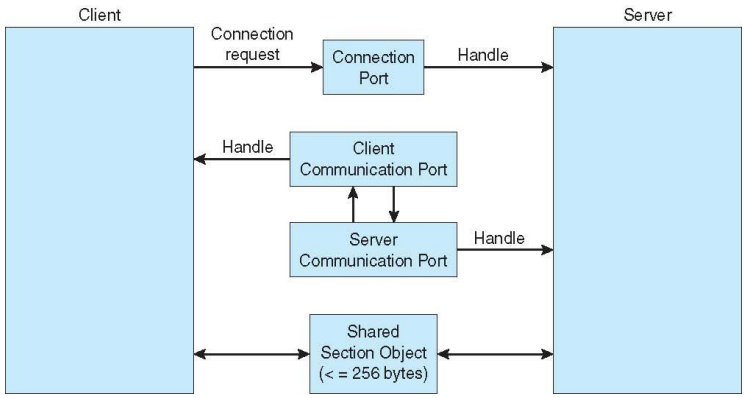
    `msg_send(), msg_receive(), msg_rpc()`
  - Mailboxes needed for commuication, created via

    `port_allocate()`
    - Send and receive are flexible, for example four options if mailbox full:
      - Wait indefinitely
      - Wait at most n milliseconds
      - Return immediately
      - Temporarily cache a message

Examples of IPC Systems – Windows

- **Message-passing centric via advanced local procedure call (LPC) facility**
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle to the subsystem's **connection port** object.
    - The client sends a connection request.
    - The server creates two private **communication ports** and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Local Procedure Calls in Windows



ipcs, ipcrm in Linux

- **ipcs command**
  - provides information on the ipc facilities for which the calling process has read acccess.
  - options
    - -s: shows information about current semaphores.
    - -m: shows information about current shared memory.
    - -q: shows information about current message queues.
- **ipcrm command**
  - removes the resource specified by id.
  - synopsis
    - ipcrm [shm | msg | sem] id

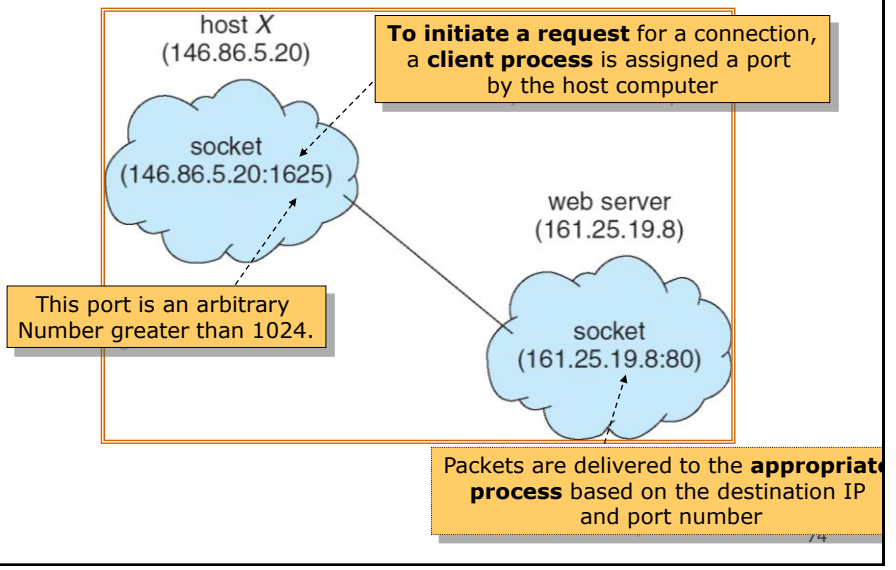
Communication in Client-Server Systems

Sockets

- **A *socket* is defined as an *endpoint for communication*.**
- **A socket is identified by an *IP address* and a *port number*.**
  - The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**.
- **A pair of processes communicating over a network employ a pair of sockets – one for each process.**
  - The server waits for incoming client requests by listening to a specified port.
  - Once a request is received, the server accepts (constructs) a connection from the client socket to complete the connection.
- **Why port?**
  - A way to communicate is to know the process ID of the hosts.
  - However, IDs change frequently.
  - Each port represents a network services.
  - All ports below 1024 are considered ***well known***.
    - http – 80, telnet – 23, ftp – 21 …
    - Special IP address 127.0.0.1 (loopback) to refer to system on which a process is running

73

Sockets (cont'd)



Java Sockets Example

- **Java provides three different types of sockets:**
  - ***Connection-oriented*** (TCP) sockets – implemented with the `Socket` class.
  - ***Connectionless*** (UDP) sockets – implemented with the `DatagramSocket` class.
  - ***Multicast sockets*** – implemented with `MulticastSocket` class.

- **Example Server:**
  - Use connection-oriented TCP sockets.
  - Listen to port 6013.
  - When a client request is received, the server returns the data and time to the client.

75

Java Sockets Example (cont'd)

DateServer.java

```
import java.net.*;
import java.io.*;

public class DateServer
{
  public static void main(String[] args)  {
   try {
     ServerSocket sock = new ServerSocket(6013);

     // now listen for connections
     while (true) {
         Socket client = sock.accept();
         // we have a connection

         PrintWriter pout = new PrintWriter(client.getOutputStream(), true);
         // write the Date to the socket
         pout.println(new java.util.Date().toString());

         // close the socket and resume listening for more connections
         client.close();
     }
   } catch (IOException ioe) {
      System.err.println(ioe);
   }
  }
}
```

This class implements server sockets

Listens for (and **block till**) a connection to be made to this socket and accepts it.

Return a socket that the server can use to communicate with the client.

The server closes the socket to the client and resumes listening for more requests.

Java Sockets Example (cont'd)

DateClient.java

```
import java.net.*;
import java.io.*;

public class DateClient
{
   public static void main(String[] args)  {
     try {
         // this could be changed to an IP name
         // or address other than the localhost
         Socket sock = new Socket("127.0.0.1",6013);
         InputStream in = sock.getInputStream();
         BufferedReader bin = new BufferedReader(new
                                          InputStreamReader(in));

         String line;
         while( (line = bin.readLine()) != null)
            System.out.println(line);

         sock.close();
     } catch (IOException ioe) {
         System.err.println(ioe);
     }
   }
}
```

Create a socket and request a connection with the server on port 6013

77

Java Sockets Example (cont'd)

- **The IP address 127.0.0.1 is a special IP address known as the *loopback*.**
  - **it is referring to itself.**
  - The example runs the client and server on the same host to communicate using the TCP/IP protocol.

78

UNIX sockets

- **communication via a network**
  - The socket programming supports communication between processes via a network as well as locally on a computer.
- **long-established services**
  - It allows network applications to be programmed using the long-established UNIX concept of file descriptors.
- **two types**
  - UNIX domain sockets
  - INET domain sockets

79

Remote Procedure Calls

- **The *remote procedure call* (RPC) was designed as a way to abstract the procedure-call mechanism for use between systems with network connections.**
  - Allow a client to invoke a procedure on a remote host as it would invoke a procedure locally.
- **Is a message-based communication because the communicating processes are executing on separate systems.**
- **The message are well structured and are thus no longer just packets of data.**

80

Remote Procedure Calls (cont'd)

- **The RPC takes place by providing a *stub* on the client side.**
  - Stub: **client-side proxy** for the actual procedure on the server.
  - The server-side stub receives this message, unpack the marshalled parameters, and performs the procedure on the server.
  - A separate stub exists for each separate remote procedure.
  - On Windows, stub code compile from the specification written in MIDL (Microsoft Interface Definition Language)
- **RPC operations:**
  1. When the client invokes a remote procedure, the PRC system calls the appropriate stub, passing it the required parameters.
  2. The stub **locates the port** on the server and **marshals the parameters**, and then **transmits a message** to the server.
  3. Server has a similar stub that receives this message and invokes the procedure.
  4. If necessary, return values are passed back to the client using the same technique.

81

Remote Procedure Calls (cont'd)

- ***Parameter marshalling*:**
  - Different systems use different data representation.
    - Big/little-endian: use the high/low memory address to store the most signification byte.
  - Many RPC systems define a machine-independent representation of data.
    - For example, external data representation (XDR).
  - Parameter marshalling involves converting the machine-dependent data into XDR before they are sent to the server.
  - On the server side, the XDR data are **unmarshalled** and converted to the machine-dependent representation for the server.

82

Remote Procedure Calls

- **Problem 1 – RPCs can fail, or be executed more than once, as a result of network errors.**
  - The operating system have to ensure that messages are acted on ***exactly once***, ***rather than at most one***.
- **For at most once:**
  - Each message is attached a timestamp (in the sender side).
  - The server keep a large history of all the timestamps of messages it has already processed to detect repeated messages.
- **For exactly once:**
  - the server must acknowledge to the client that the RPC call was received and executed.
  - The client must resend each RPC call periodically until it receives the ACK for that call.
  - Of course, the server must implement the "at most once" protocol.

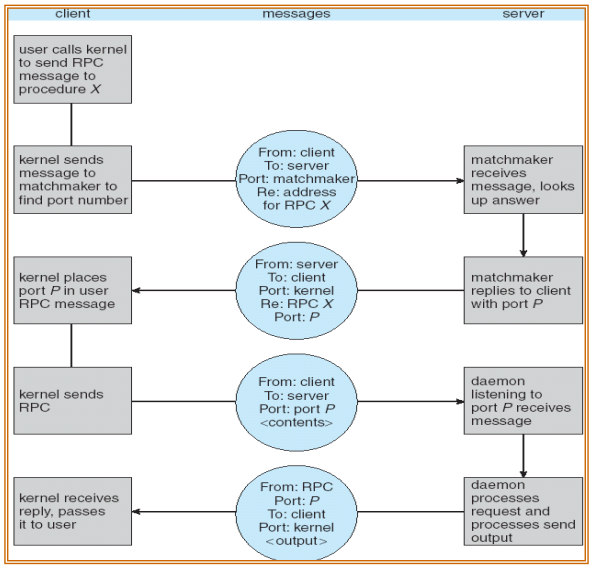
83

Remote Procedure Calls (cont'd)

- **Problem 2 – How does a client know the port numbers on the server?**
  - Approach 1: the information may be predetermined, in the form of **fixed port addresses**.
    - The server can not change the port number of the requested service.
  - Approach 2: the operating system provides a ***matchmaker*** daemon on a fixed RPC port.
    - A client then sends a message containing the name of the RPC to the matchmaker requesting the port address of the RPC it needs to execute.
    - More flexible, but requires the extra overhead.

84

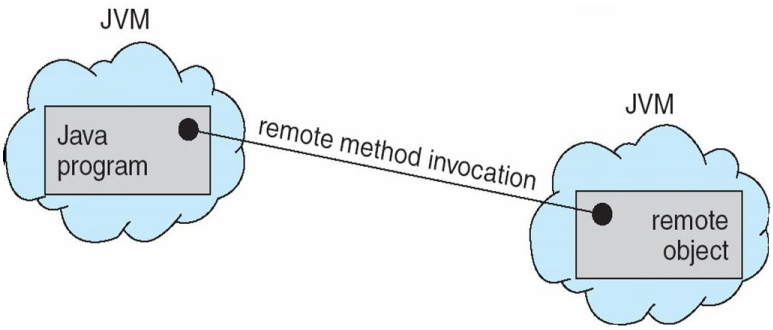
Remote Procedure Calls (cont'd)



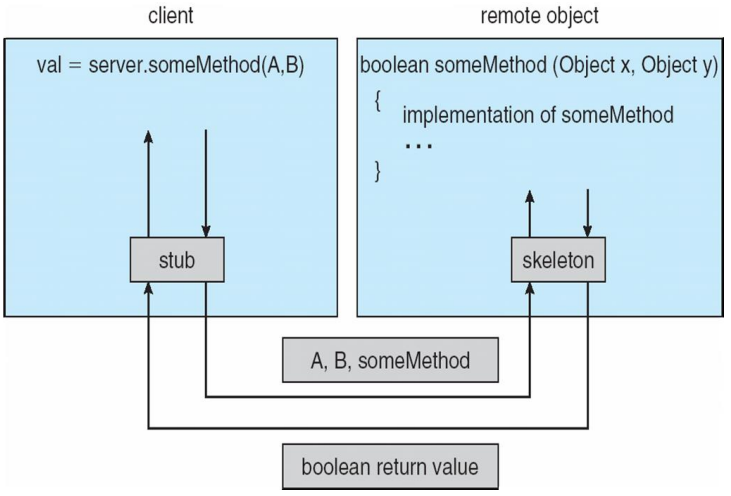
85

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



Appendix – How to use `man`

- **Typical syntax:**

```
man [section] command_or_function
```

- **Section:**

1. **User commands.**
2. **System calls and error numbers .**
3. **Functions in the C libraries.**
4. Device drivers.
5. File formats.
6. Games and other diversions.
7. Miscellaneous information.
8. System maintenance and operation commands.
9. Kernel interface documentation.

88