Chapter 2: Process Management

2.1 Processes

GV: Nguyễn Thị Thanh Vân

Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations. mo ta tien trinh duoc tao ra va ket thuc nhu the nao
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that uses pipes and POSIX shared memory to perform interprocess communication.
- Describe client-server communication using sockets and remote procedure calls.
- Design kernel modules that interact with the Linux operating system.

Outlines

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

Process Concept

- An OS executes a variety of programs that run as a process.
- **Process**
  - a program in execution - must progress in sequential fashion.
  - No parallel execution of instructions of a single process
- One program can be several processes
  - Consider multiple users executing the same program
    - Compiler, Text editor
- The memory layout of a process is typically divided into multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters,
    - return addresses,
    - local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

Process in Memory

bien cuc bo trong ham

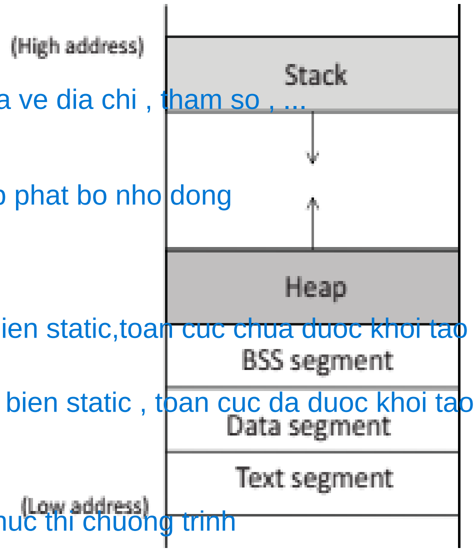
Stack: store local variables in func, store data related to function calls: return address, arguments, (LIFO) tra ve dia chi , tham so , ...

Heap: provide space for dynamic cap phat bo nho dong memory allocation. This area is managed by malloc, calloc …

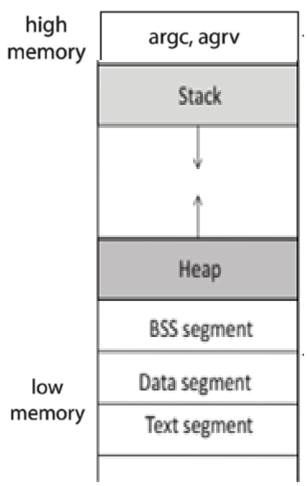
BSS segment: stores uninitialized static/global variables (zero) bien static,toan cuc chua duoc khoi tao

Data segment : stores static/global variables that are initialized by the programmer bien static , toan cuc da duoc khoi tao

Text: stores the executable code of the program (read-only) chua ma thuc thi chuong trinh



Memory Layout of a C Program



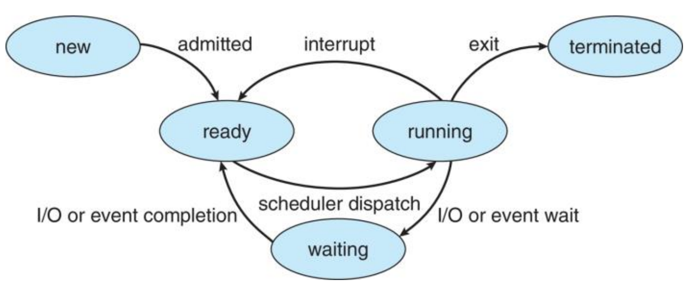

```
#include <stdio.h>
#include <stdlib.h>

int x;  bss segment
int y = 15;  data Segment

int main(int argc, char *argv[])
{   local variable
    int *values; stack
    int i;  stack
                          heap
    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

Process

- A process includes:
  - Text chua phan code chuong trinh
  - Data
  - Heap
  - Stack
  - PC – Program counter: a register that manages the memory address of the instruction to be executed next bo dem chuong trinh (thanh ghi) quan li bo nho cua thuong trinh tiep theo se thuc hien
  - PSW – Program status word: a register that performs the function of a status register and program counter thanh ghi (trang thai) luu gia tri
  - SP – Stack poiter
  - Registers nhung thanh ghi
- Four principal events cause processes to be created:
  - System initialization.
  - Execution of a process-creation system call by a running process.
  - A user request to create a new process. yeu cau ng dung tao tien trinh moi
  - Initiation of a batch job. khoi tao cong viec theo dang low

Process State

- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor da chuan bi de thuc thi chi can he thong goi
  - **Terminated**: The process has finished execution



xóa

Process Control Block (PCB)

khoi dieu khien process

- PCB: a data structure used by computer OS to store all the information about a process. luu tru thong tin ve tien trinh
- Each process is represented in OS by PCB, also called **task control block**
- It contains many pieces of information associated with a specific process:

trang thai cua tien trinh

- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute vi tri cua lenh tiep theo dc thuc thi
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

process state
process number
program counter
registers
memory limits
list of open files
• • •

gioi han ve bo nho

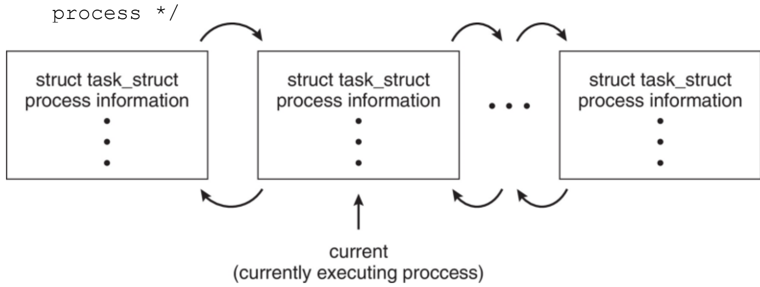
Threads

- So far, process has a single thread of execution ngay truoc tien trinh thuc hien don luong
- Most modern OSs have extended the process concept to allow a process to have multiple threads of execution cac OS hien dai cho phep chay da luong
  - thus to perform more than one task at a time
  - Multiple threads can run in parallel
- The PCB is expanded to include information for each thread.
  - Must then have storage for thread details,
- Explore in detail later – next chapter

PCB in Linux OS

PCB is represented by the C structure `task_struct`, which is found in <include/linux/sched.h>

```
pid t_pid;                    /* process identifier */
long state;                   /* state of the process */
unsigned int time_slice       /* scheduling information */
struct task_struct *parent;/* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files;/* list of open files */
struct mm_struct *mm;         /* address space of this
process */
```



Process Scheduling

- Goals of:
  - Multiprogramming: some process running at all times so as to maximize CPU utilization
  - Time sharing: switch a CPU core among processes so frequently that users can interact with each program while it is running

    chuyen doi CPU giua cac tien trinh

lap lich tien trinh

- => **Process scheduler** selects among available processes for next execution on CPU core
- The number of processes currently in memory is known as the **degree of multiprogramming**
- Maintains **scheduling queues** of processes

  quan li hai hang doi

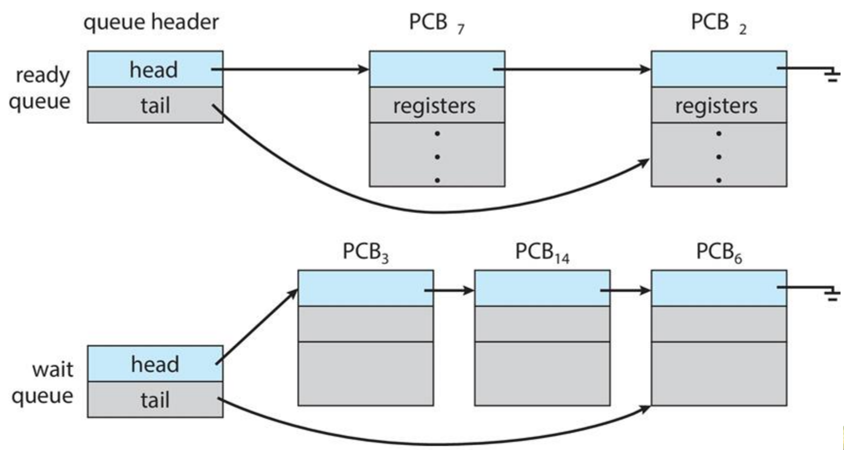
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute

    nhung tien trinh chi can dc cap cpu la dc thuc thi lien
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - Processes migrate among the various queues

    tien trinh cho mot cai gi do moi dc thuc thi

Ready and Wait Queues

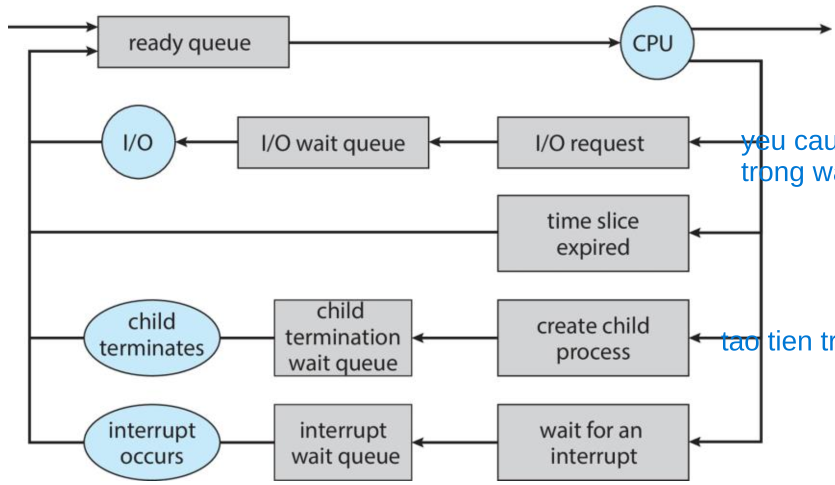
- Ready and Wait Queues are generally stored as linked lists duoc to chua theo danh sach lien ket
-



CPU Scheduling

- A process migrates among the ready queue and various wait queues throughout its lifetime.
- CPU scheduler:
  - select from among the processes that are in the ready queue and allocate a CPU core to one of them
  - select a new process for the CPU frequently.
- Queueing-diagram Representation of Process Scheduling: Once the process is allocated the CPU and is executing, one of several events could occur:
  - The process could issue an I/O request and then be placed in an I/O queue.
  - The process could create a new child process and wait for the child's termination.
  - The process could be removed forcibly from the CPU,as a result of an interrupt, and be put back in the ready queue.

Queueing-diagram Representation of Process Scheduling



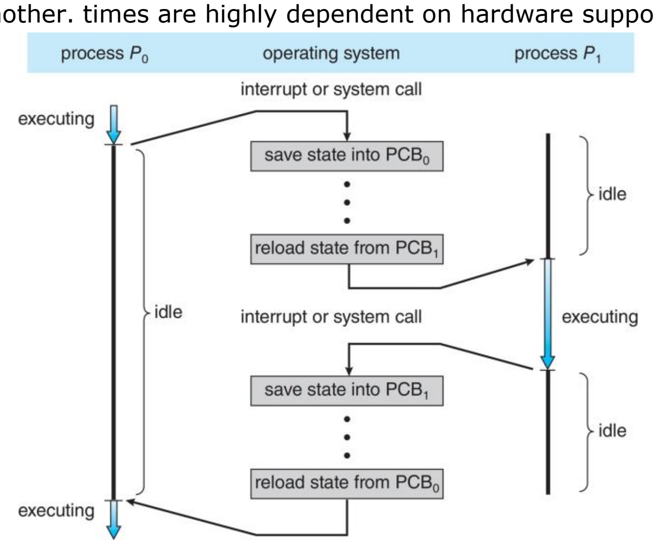
yeu cau ve Input Ouputnam trong wait queue

tao tien trinh con

CPU Switch From Process to Process

su chuyen doi cac tien trinh duoc goi la context switch

A **context switch** occurs when the CPU switches from one process to another. times are highly dependent on hardware support.



Context Switching

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
  - The more complex the OS and the PCB, the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU, multiple contexts loaded at once

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 tien trinh mat truoc
  - Multiple **background** processes– in memory, running, but not on the display, and with limits tien trinh mat sau
  - 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

Operations on Processes

- System must provide mechanisms for: he thong phai cung cap co che de
  - Process creation tao tien trinh
  - Process termination ngat tien trinh

Process Creation

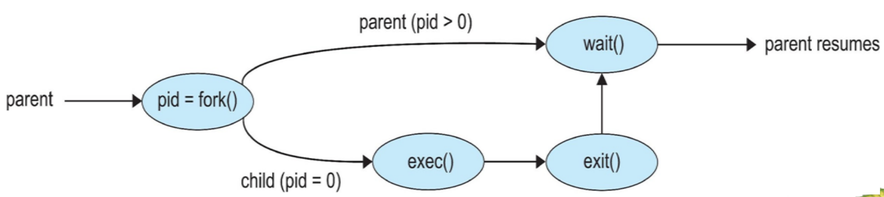
tien trinh con tao ra cac tien trinh con khac tao thanh 1 cay

tien trinh cha tao ra cac tien trinh con

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes mot cay cac tien trinh
- Generally, process identified and managed via a **process identifier** (**pid**) dinh danh cac tien trinh de ho tro cho cac tien trinh theo quan he parent child
- 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

Process Creation (Cont.)

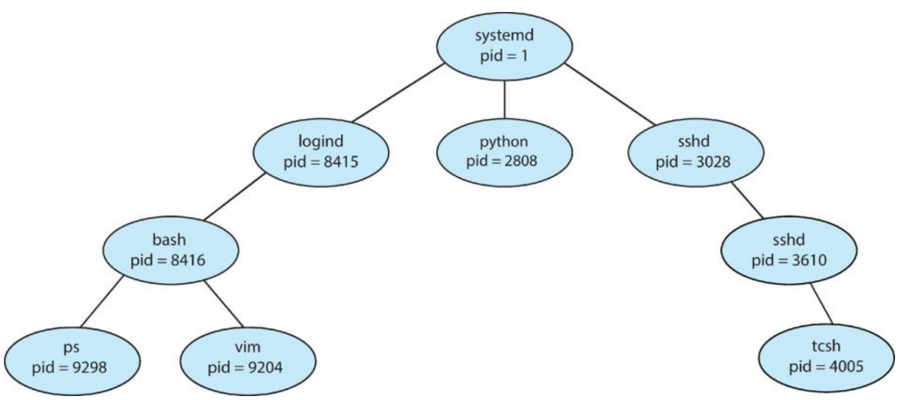
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program thay the bo nho cua tien trinh thanh chuong trinh moi
  - Parent process calls **`wait()`** waiting for the child to terminate



khi ham fork duoc goi parent se tao ra tien trinh child , child la mot ban sao chep cua parent , parent va child duoc thuc thi song song

so tien trinh = 2^n voi n la so ham fork() duoc goi

A Tree of Processes in Linux



C Program Forking Separate Process

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

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 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");
    }

    return 0;
}
```


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);
}
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call. yeu cau he dieu hanh xoa tien trinh
  - Returns status data from child to parent (via **`wait()`**) tai nguyen cua cac tien trinh se
  - Process' resources are deallocated by operating system duoc thu hoi va cap phat cho cac tien trinh khac
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so: tien trinh cha co the ket thuc tien trinh con
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting, and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc., are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
- If parent terminated without invoking **`wait()`**, process is an **orphan**

Ex

- Detail in Process_syscal_expl file

Android Process Importance Hierarchy

tat tien trinh

- Mobile operating systems often have to terminate processes to
lay lai reclaim system resources such as memory. From **most** to **least** important:
  - Foreground process
  - Visible process
  - Service process
  - Background process
  - Empty process
- Android will begin terminating processes that are least important.

tat tien trinh it quan trong nhat

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

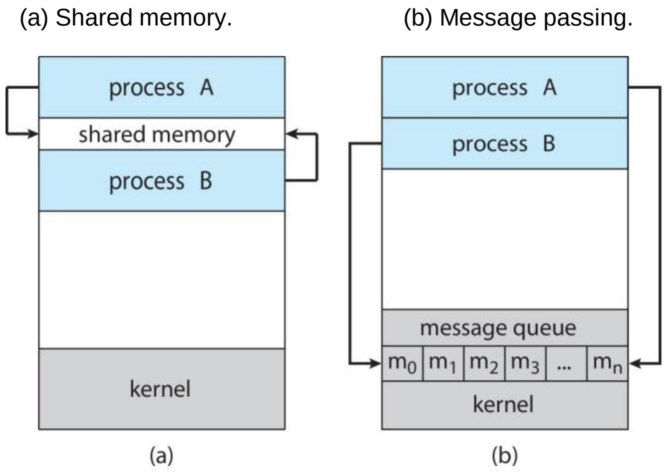
Each tab represents a separate process.

Interprocess Communication

giao tiep giua cac tien trinh

- Processes within a system may be **independent** or **cooperating**
- **Cooperating process** can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory** (under the control of users) duoi su quan li cua user
  - **Message passing** (under the control of OS) duoi su quan li cua he dieu hanh

Communications Models

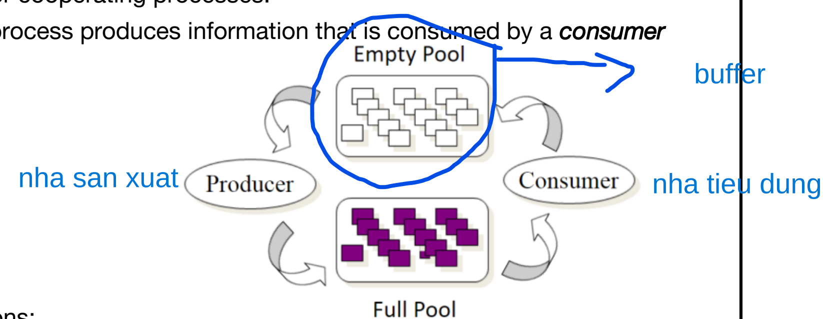


buffer : vung luu tru du lieu tam thoi trong RAM

shared memory :

ex: Producer-Consumer Problem

- Paradigm for cooperating processes:
  - ***producer*** process produces information that is consumed by a ***consumer*** process



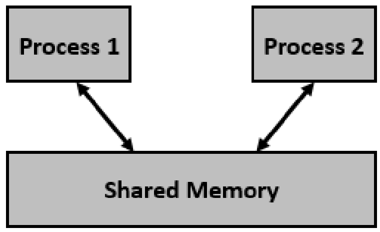
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer: buffer vo han
    - Producer never waits nha cung cap khong can doi cu tiep tuc san xuat
    - Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - Producer must wait if all buffers are full phai doi neu buffer full
    - Consumer waits if there is no buffer to consume

ki nang chinh la cung cap co che cho phep hanh dong tien trinh cua nguoi dung thuc thi dong bo khi no truy cap vao shared memory

Shared Memory Solution

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapters 6 & 7.

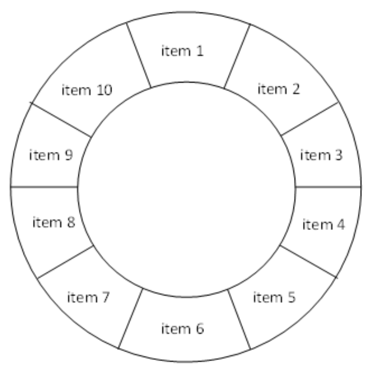
dong bo duoc thao luan o chap 6 7



Bounded-Buffer – Shared-Memory Solution

- Shared data

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

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



- Solution presented in next slides is correct, but can only use `BUFFER_SIZE-1` items; that is: 9 items

Producer/ Consumer Process – Shared Memory

Producer

```
item next_produced;
while (true) {
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE) == out)
   ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```

Customer

```
item next_consumed;
while (true) {
        while (in == out)
        ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
/* consume the item in next consumed */
}
```


What about Filling all the Buffers?

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.
- We can do so by having an integer **`counter`** that keeps track of the number of full buffers.
- Initially, **`counter`** is set to 0.
- The integer **`counter`** is incremented by the producer after it produces a new buffer.
- The integer **`counter`** is and is decremented by the consumer after it consumes a buffer.

Producer/ Consumer Process

Producer

```
while (true) {
      /* produce an item in next produced
   while (counter == BUFFER_SIZE)
      ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

Customer

```
while (true) {
      while (counter == 0)
            ; /* do nothing */
      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
       counter--;
/* consume the item in next consumed */
}
```


Race Condition

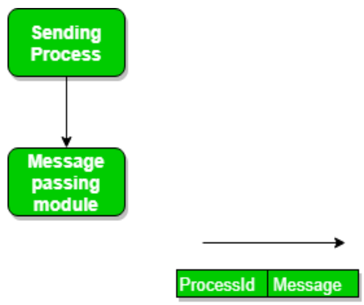
- **`counter++`** could be implemented as
```
register1 = counter
register1 = register1 + 1
counter = register1
```
- **`counter--`** could be implemented as
```
register2 = counter
register2 = register2 - 1
counter = register2
```
- Consider this execution interleaving with "count = 5" initially:

S0: producer execute **`register1 = counter`**	{register1 = 5}
S1: producer execute **`register1 = register1 + 1`**	{register1 = 6}
S2: consumer execute **`register2 = counter`**	{register2 = 5}
S3: consumer execute **`register2 = register2 - 1`**	{register2 = 4}
S4: producer execute **`counter = register1`**	{counter = 6 }
S5: consumer execute **`counter = register2`**	{counter = 4}

- Question – why was there no race condition in the first solution (where at most N – 1) buffers can be filled?

IPC – Message Passing

- Processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- The *message* size is either fixed or variable



Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a **communication link** between them can phai thiet lap link giao tiep giua cac tien trinh
  - Exchange messages via send/receive
- Implementation issues:
  - How are **links** established? cac duong link duoc thiet lap nhu the nao
  - Can a link be associated with more than two processes? mot duong link co the co 2 tien trinh tro len khong
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link? kha nang cua duong link
  - Is the size of a message that the link can accommodate fixed or variable? co the chua bao nhieu thong diep tren duong link
  - Is a link unidirectional or bi-directional?

Implementation of Communication Link

- Physical:
  - Shared memory
  - Hardware bus bang cac duong bus
  - Network
- Logical:
  - Direct or indirect truc tiep hoac gian tiep
  - Synchronous or asynchronous
  - Automatic or explicit buffering

tu dong thuc hien ro rang tren bo dem

Direct Communication

- Processes must name each other explicitly:
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes giua hai tien trinh cho co 1 duong link
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

1 chieu 2 chieu

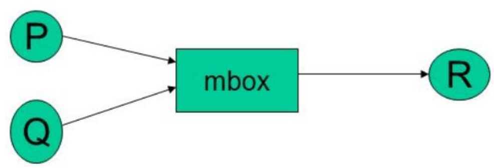
gian tiep

Indirect Communication

thong qua trung gian mailbox

- Messages are directed and received from mailboxes (also is ports)
  - Each mailbox has a unique id moi thung thu co ma dinh danh
  - Processes can communicate only if they share a mailbox muon giao tiep phai chia se chung mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox duong link se duoc thiet lap neu tien trinh chia se chung mail bõ
  - A link may be associated with many processes 1 duong link co the ket noi nhieu hon hai tien trinh
  - Each pair of processes may share several communication links giua 2 tien trinh co the co nhieu duong link
  - Link may be unidirectional or bi-directional

1 huong 2 huong



Indirect Communication

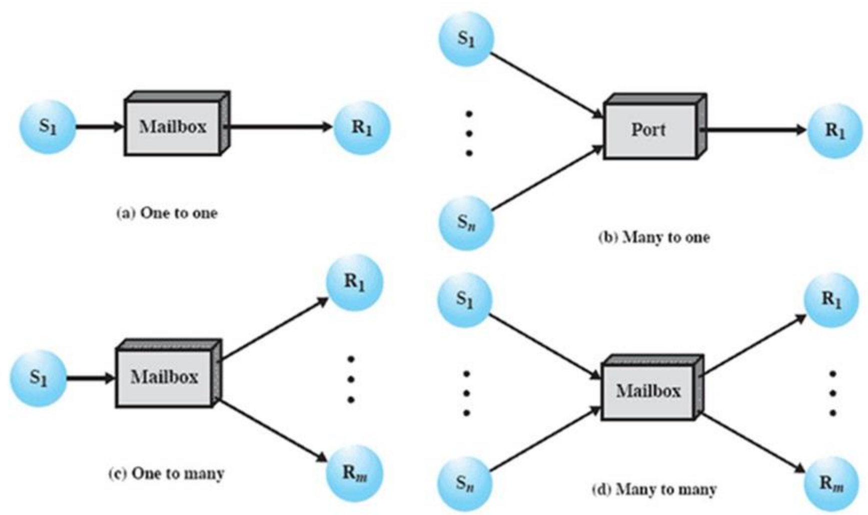
- Operations
  - Create a new mailbox (port)
  - Send and receive messages through mailbox
  - Delete a mailbox
- Primitives are defined as: cac ham nguyen tu khong the chia nho cac ham , khong co ham nao duoc xen vao giua send va receive
  - **`Send`**(*A, message*) – send a message to mailbox A
  - **`receive`**(*A, message*) – receive a message from mailbox A

Indirect Communication (Cont.)

- 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.

1 thoi diem chi co 1 tien trinh duoc nhan

Indirect Communication (Cont.)



Synchronization

Message passing may be either blocking or non-blocking

- **Blocking** is considered **synchronous** dong bo
  - **Blocking send** -- the sender is blocked until the message is received bi block den khi nao tin nhan gui da duoc nhan
  - **Blocking receive** -- the receiver is blocked until a message is available den khi co 1 mess gui , receive san sang nhan
- **Non-blocking** is considered **asynchronous** bat dong bo
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Producer-Consumer: Message Passing

- Producer

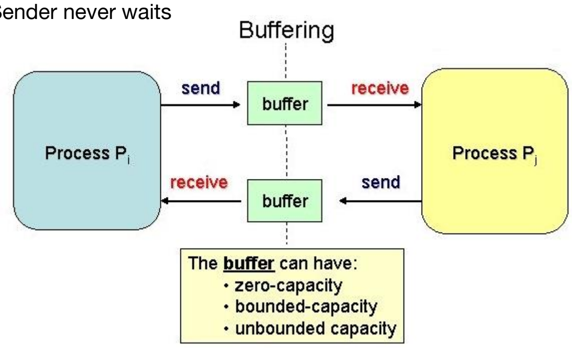
```
message next_produced;
while (true) {
  /* produce an item in next_produced */

  send(next_produced);
}
```

- Consumer

```
message next_consumed;
while (true) {
   receive(next_consumed)

  /* consume the item in next_consumed */
}
```

Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways
  1. Zero capacity – no messages are queued on a link. khong co kha nang luu tru
     Sender must wait for receiver (rendezvous) nguoi gui phai doi nguoi nhan
  2. Bounded capacity – finite length of *n* messages co kha nang luu tru co gioi han
     Sender must wait if link full ng gui phai doi neu link bi day
  3. Unbounded capacity – infinite length co kha nang luu tru vo han
     Sender never waits



Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment
    `shm_fd = shm_open(name, O CREAT | O RDWR, 0666);`
  - Also used to open an existing segment
  - Set the size of the object

  `ftruncate(shm_fd, 4096);`
  - Use `mmap()` to memory-map a file pointer to the shared memory object
  - Reading and writing to shared memory is done by using the pointer returned by `mmap()`.

IPC POSIX 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;
}
```


IPC POSIX 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;
}
```

Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two ports at creation- Kernel and Notify
  - Messages are sent and received using the `mach_msg()` function
  - Ports needed for communication, created via `mach_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

Mach Messages

```
#include<mach/mach.h>

struct message {
       mach_msg_header_t header;
       int data;
};

mach port t client;
mach port t server;
```

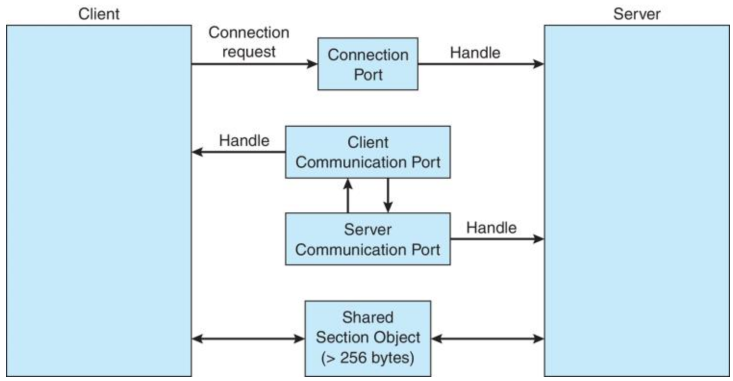
Mach Message Passing – Client/Server

```
/* Client Code */

struct message message;

// construct the header
message.header.msgh_size = sizeof(message);
message.header.msgh_remote_port = server;
message.header.msgh_local_port = client;

// send the message
mach_msg(&message.header, // message header
   MACH_SEND_MSG, // sending a message
   sizeof(message), // size of message sent
   0, // maximum size of received message - unnecessary
   MACH_PORT_NULL, // name of receive port - unnecessary
   MACH_MSG_TIMEOUT_NONE, // no time outs
   MACH_PORT_NULL // no notify port
);

/* Server Code */

struct message message;

// receive the message
mach_msg(&message.header, // message header
   MACH_RCV_MSG, // sending a message
   0, // size of message sent
   sizeof(message), // maximum size of received message
   server, // name of receive port
   MACH_MSG_TIMEOUT_NONE, // no time outs
   MACH_PORT_NULL // no notify port
);
```


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.

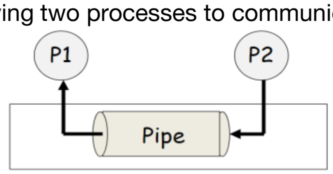
Local Procedure Calls in Windows



cach cac tien trinh trao doi voi nhau

Pipes

- Acts as a conduit allowing two processes to communicate

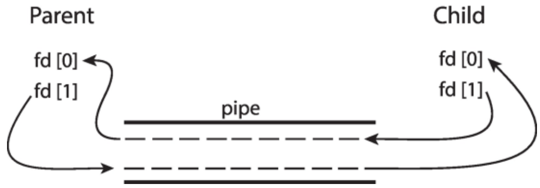


- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- **Ordinary pipes** – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created. yeu cau 2 tien trinh phai co quan he cha con
- **Named pipes** – can be accessed without a parent-child relationship.

khong can moi quan he cha con

Ordinary Pipes

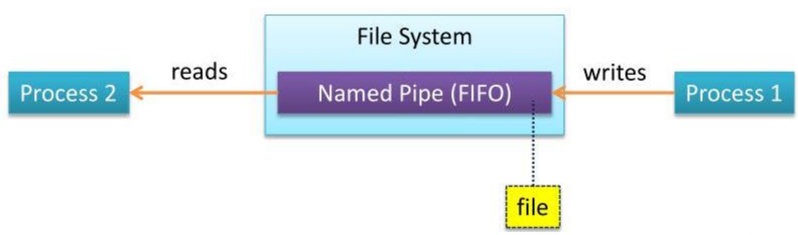
- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional 1 chieu
- Require parent-child relationship between communicating processes



- Windows calls these **anonymous pipes** con co the duoc goi la anonymous pipes

Named Pipes

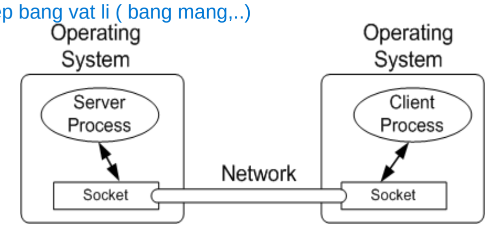
- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems



Communications in Client-Server Systems

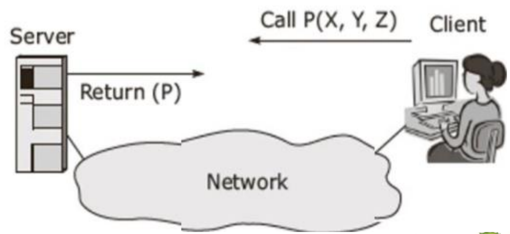
- Sockets

giao tiep bang vat li (bang mang,..)



- Remote Procedure Calls

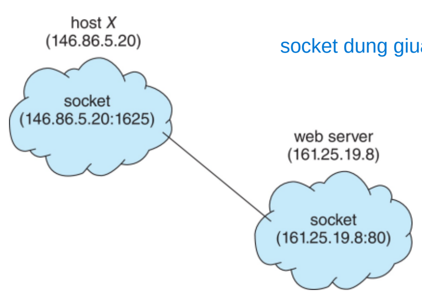
goi thu tuc tu xa



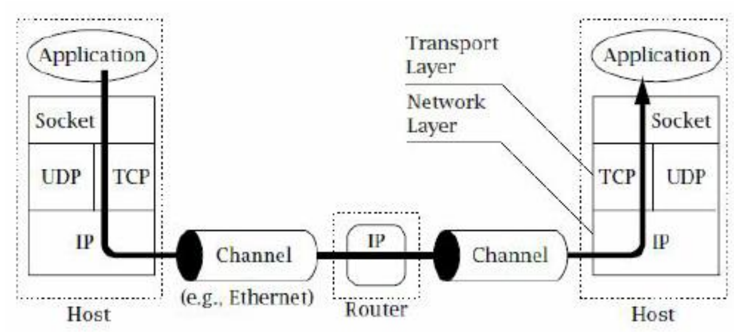
Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port**
  - port is a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are ***well known***, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running
- Communication:

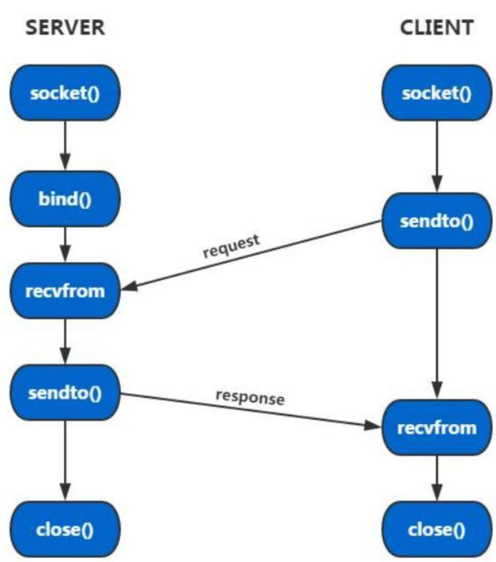
socket dung giua host va sever



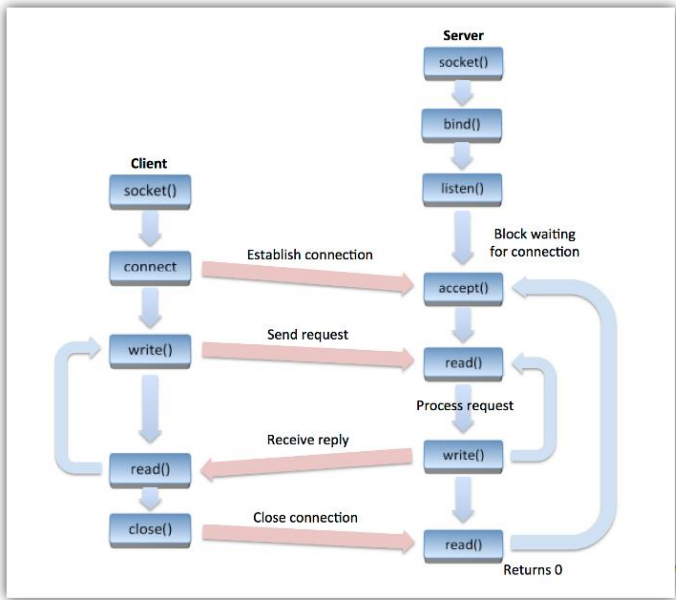
Sockets



UDP Socket



TCP Socket



Sockets in Java

- Three types of sockets
  - **Connection-oriented** (**TCP**)
  - **Connectionless** (**UDP**)
  - **`MulticastSocket`** class– data can be sent to multiple recipients
- Consider this "Date" server in 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();

        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 connections */
        client.close();
      }
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

Sockets in Java

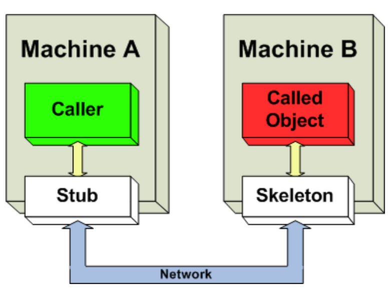
The equivalent Date client

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

public class DateClient
{
  public static void main(String[] args) {
    try {
      /* make connection to server socket */
      Socket sock = new Socket("127.0.0.1",6013);

      InputStream in = sock.getInputStream();
      BufferedReader bin = new
        BufferedReader(new InputStreamReader(in));

      /* read the date from the socket */
      String line;
      while ( (line = bin.readLine()) != null)
        System.out.println(line);

      /* close the socket connection*/
      sock.close();
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

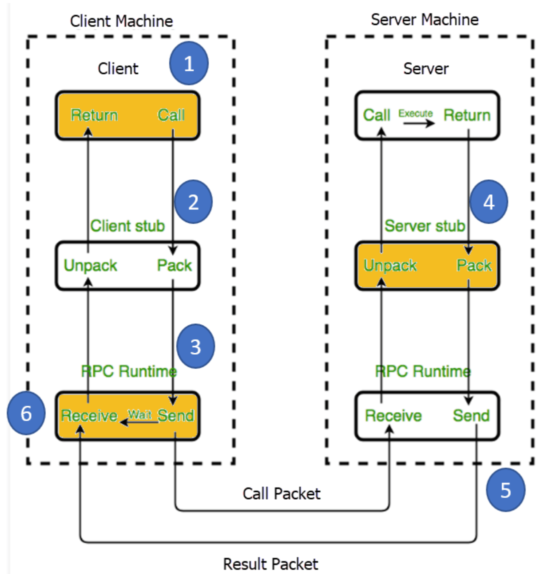

Remote Procedure Calls

goi thu tuc tu xa

- RPC abstracts procedure calls between processes on networked systems
  - Again, uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server



Execution of RPC



co 2 mo hinh mo ta giao tiep giua hai tien trinh
+sharedmemory : duoc quan li boi nguoi dung
+mess pass : kernel
tien trinh giao tiep qua link : +physical:network
+logic
(network)theo mo hinh TCP/IP
-4layers
Application : socket
Trans : port , co 2 loai ket noi la TCP va UDP
Net : IP
Link

End of Chapter 2.1