UNIVERSITY of WASHINGTON | BOTHELL

CSS430
Process Management

Textbook Chapter 3

Instructor: Stephen G. Dame
e-mail: sdame@uw.edu

These slides were adapted from the OSC textbook slides (Silberschatz, Galvin, and Gagne), Professor Munehiro Fukuda and the instructor's class materials.

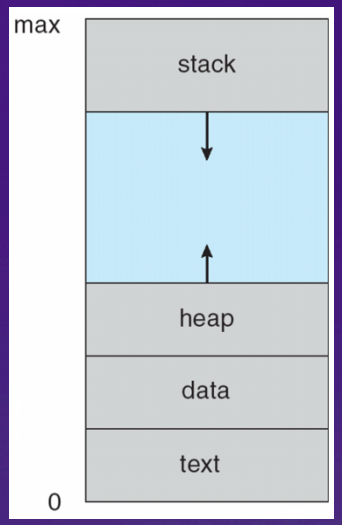
WKP 17

"Simple design, intense content."

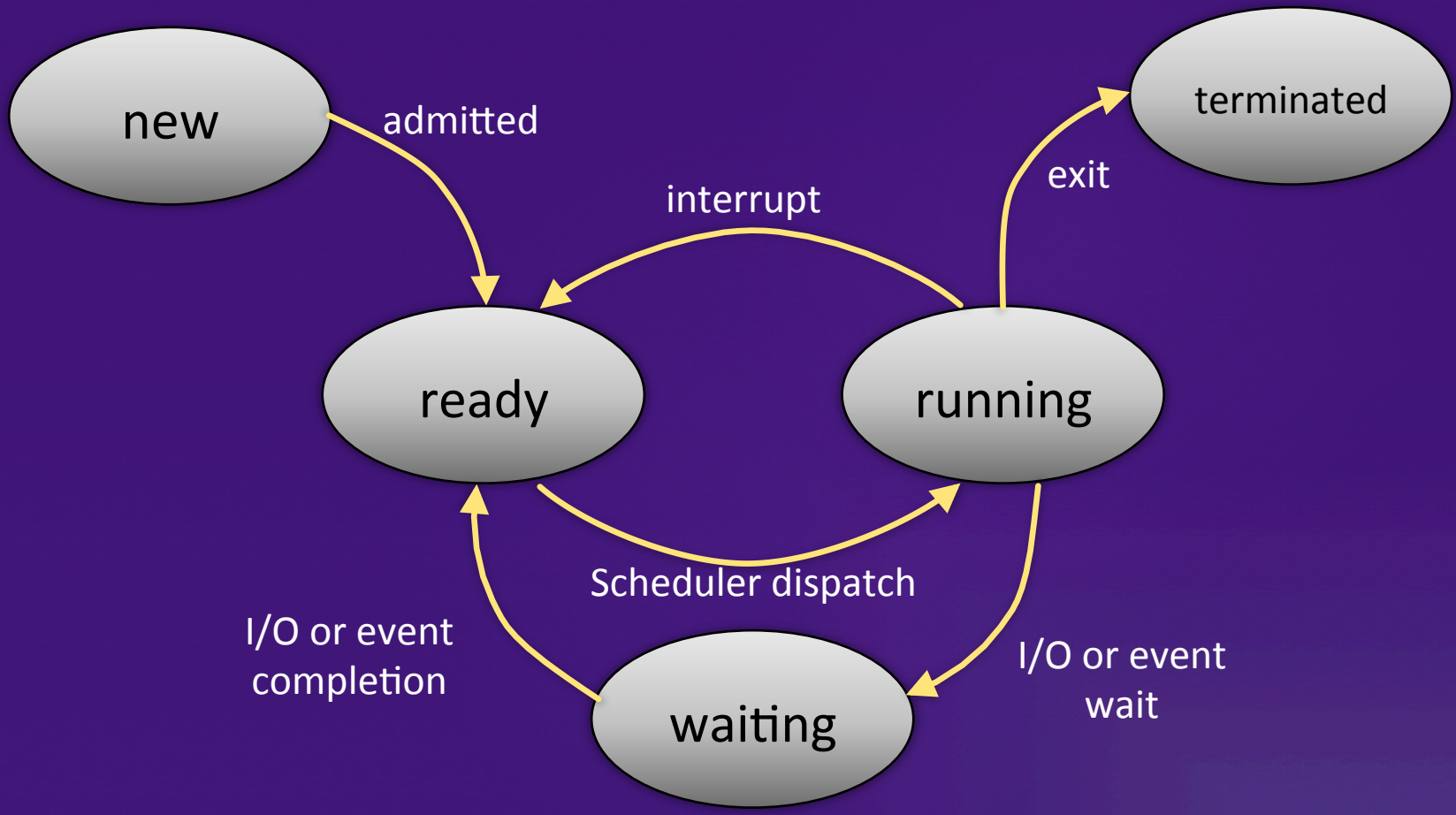
Edward Rolf Tufte is an American statistician and professor emeritus of political science, statistics, and computer science at Yale University. He is noted for his writings on information design and as a pioneer in the field of data visualization.

Process Concept

- Process – a program in execution; process execution must progress in sequential fashion.
- Textbook uses the terms ***job*** and ***process*** interchangeably.
- A process includes:
  - ✓ Program counter
  - ✓ Stack (local variables)
  - ✓ Data section (global data)
  - ✓ Text (code)
  - ✓ Heap (dynamic data)
  - ✓ Files (stdin, stdout, stderr, other file descriptors)

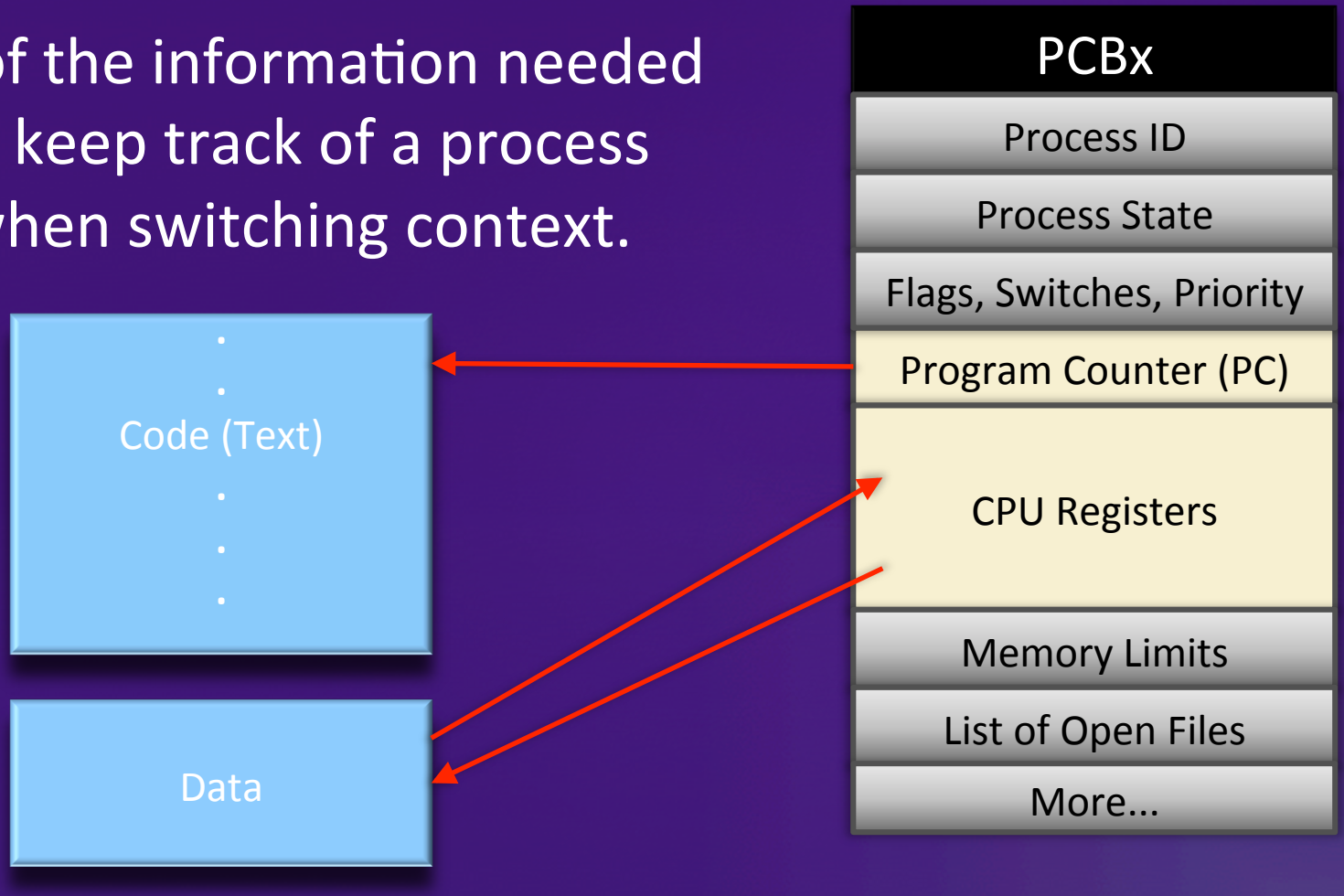


Process State

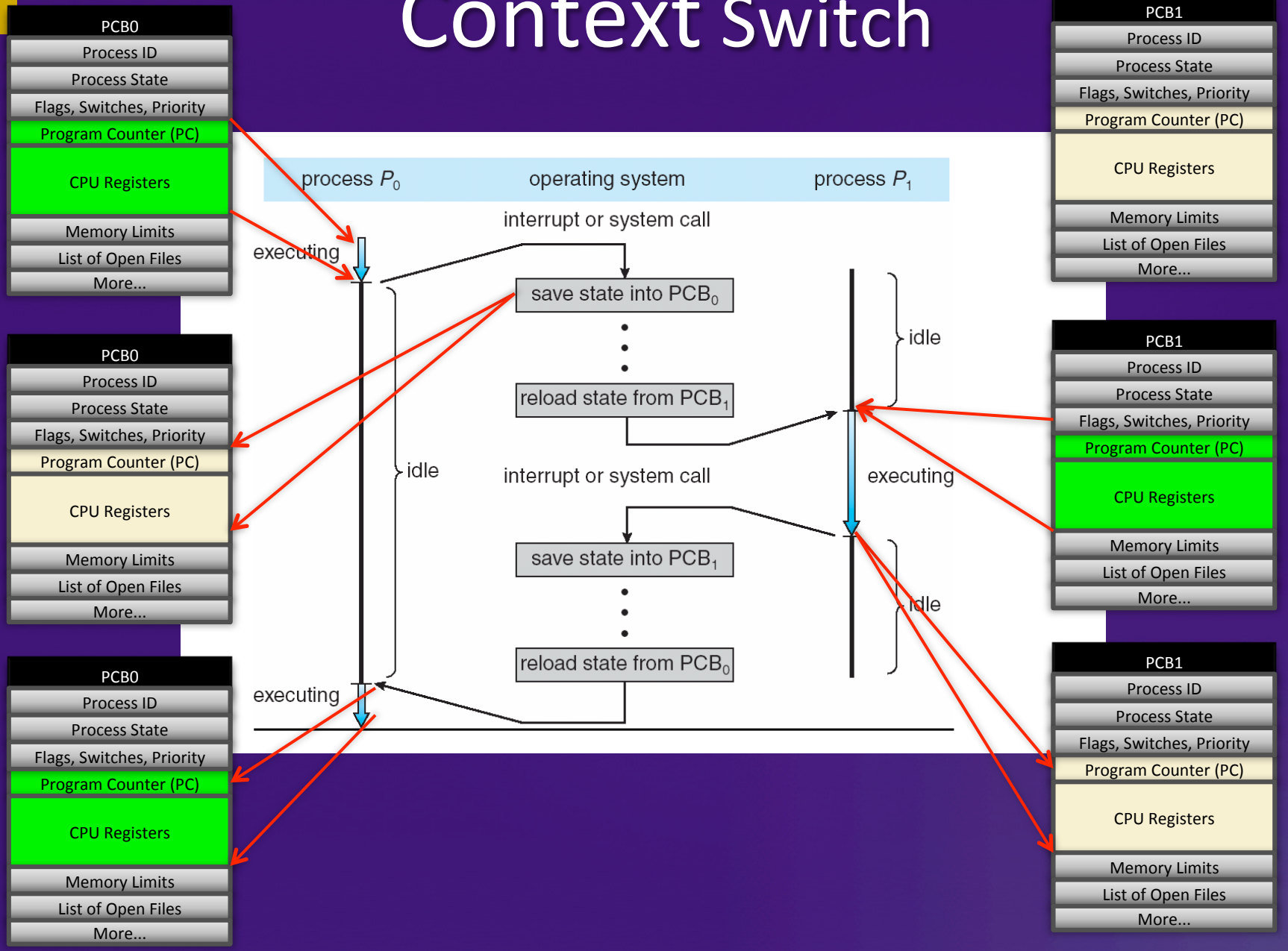


Process Control Block

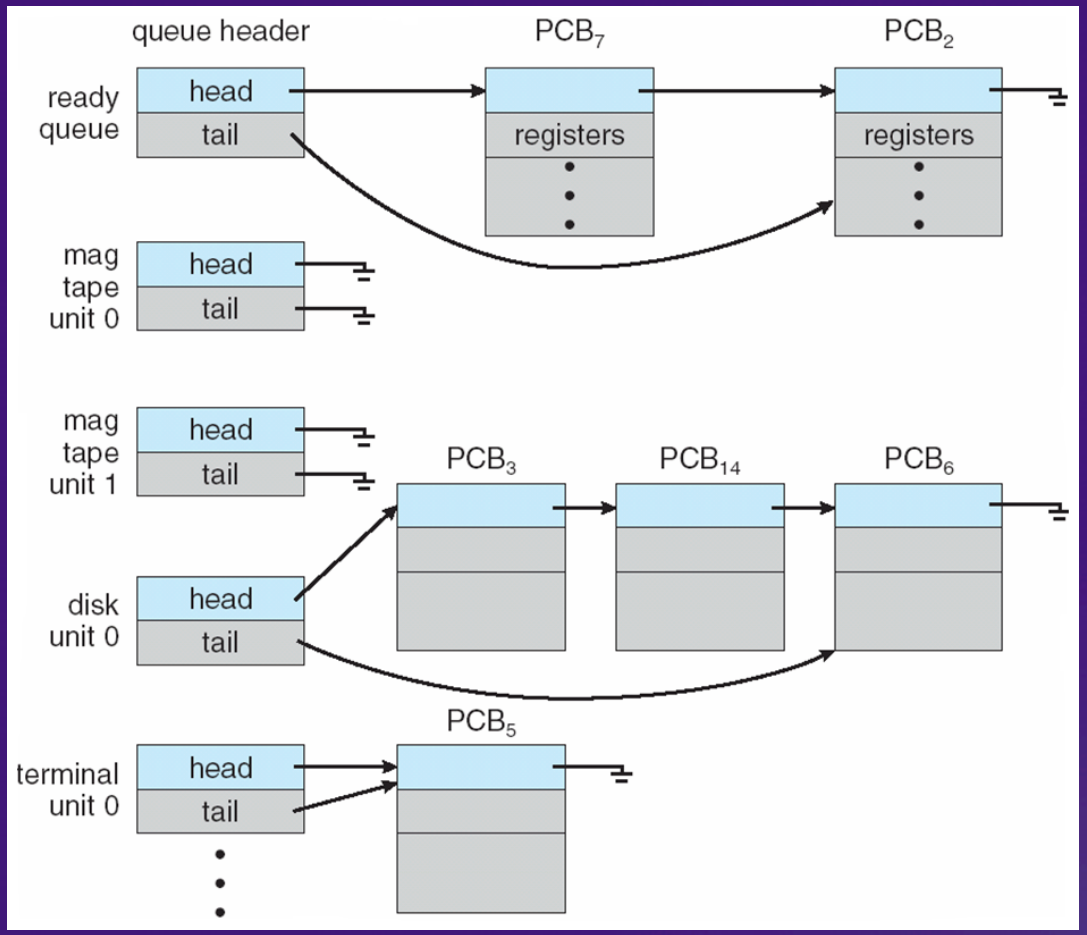
All of the information needed to keep track of a process when switching context.



Context Switch



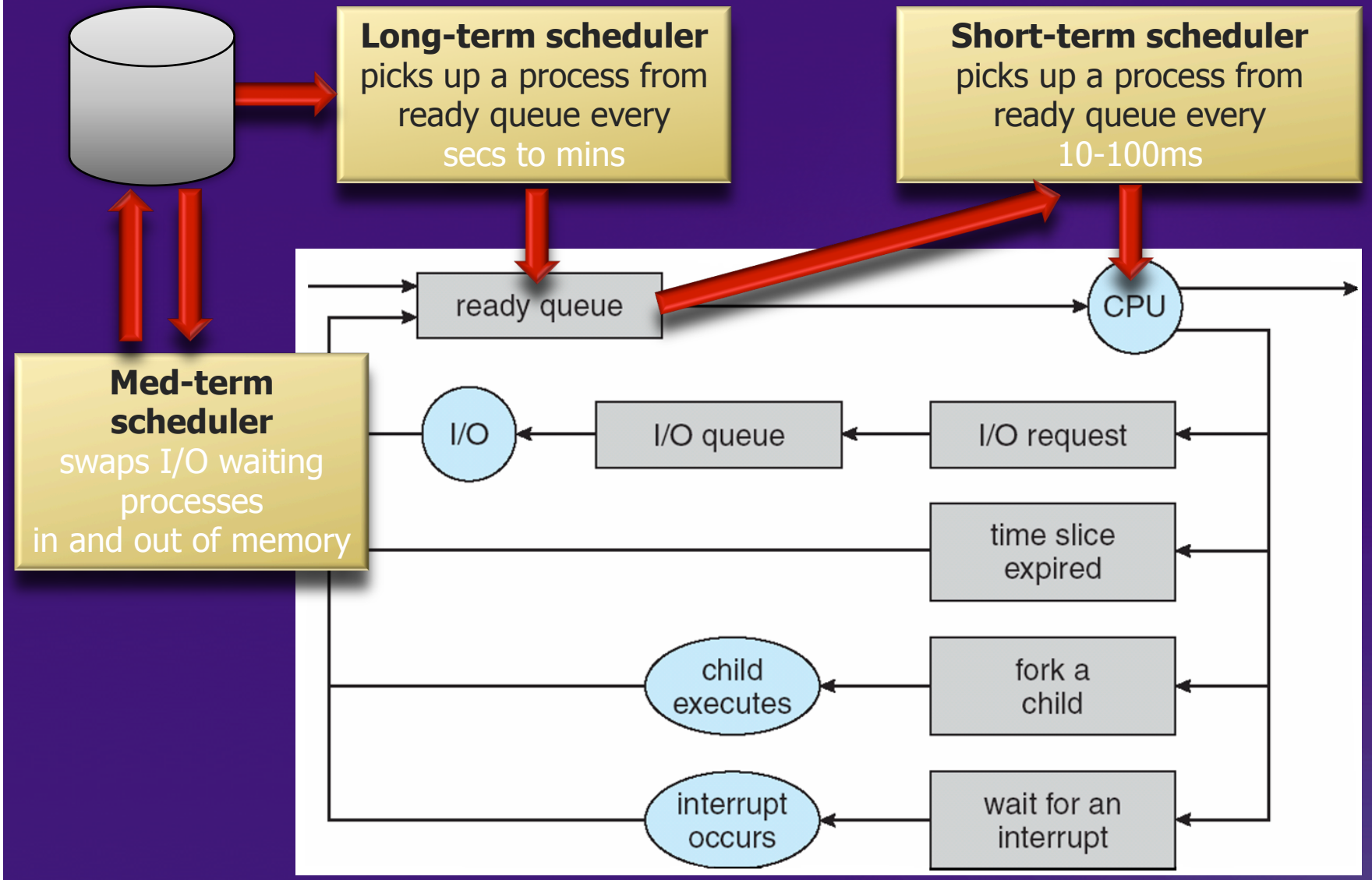
Process Scheduling Queues



Process Schedulers

- Long-Term Scheduler ("Job Scheduler")
  - ✓ Brings processes into the READY list
  - ✓ Period = seconds to minutes
- Medium Term Scheduler ("Swapper")
  - ✓ Swaps inactive processes to disk
  - ✓ Brings back swapped processes on demand
- Short-Term Scheduler ("CPU Scheduler")
  - ✓ Selects processes from the READY list
  - ✓ Allocates CPU time to the process
  - ✓ Period = milliseconds

Representation of Process Scheduling

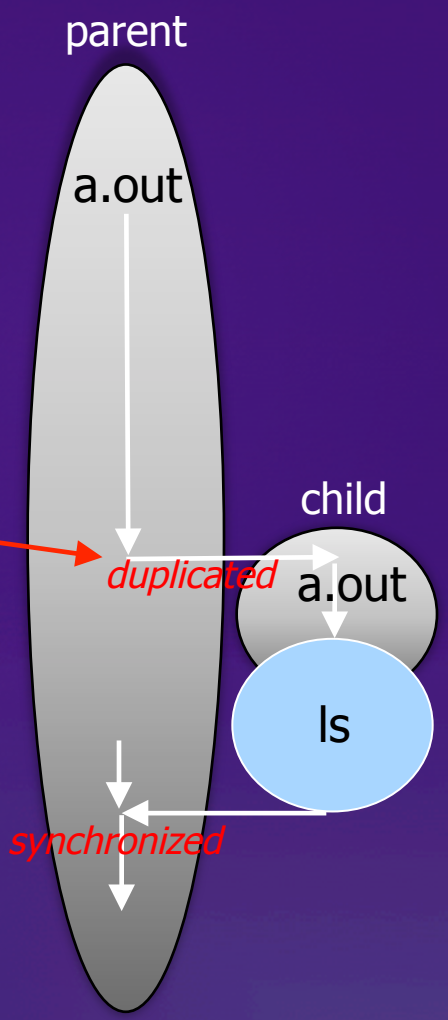


Process Creation

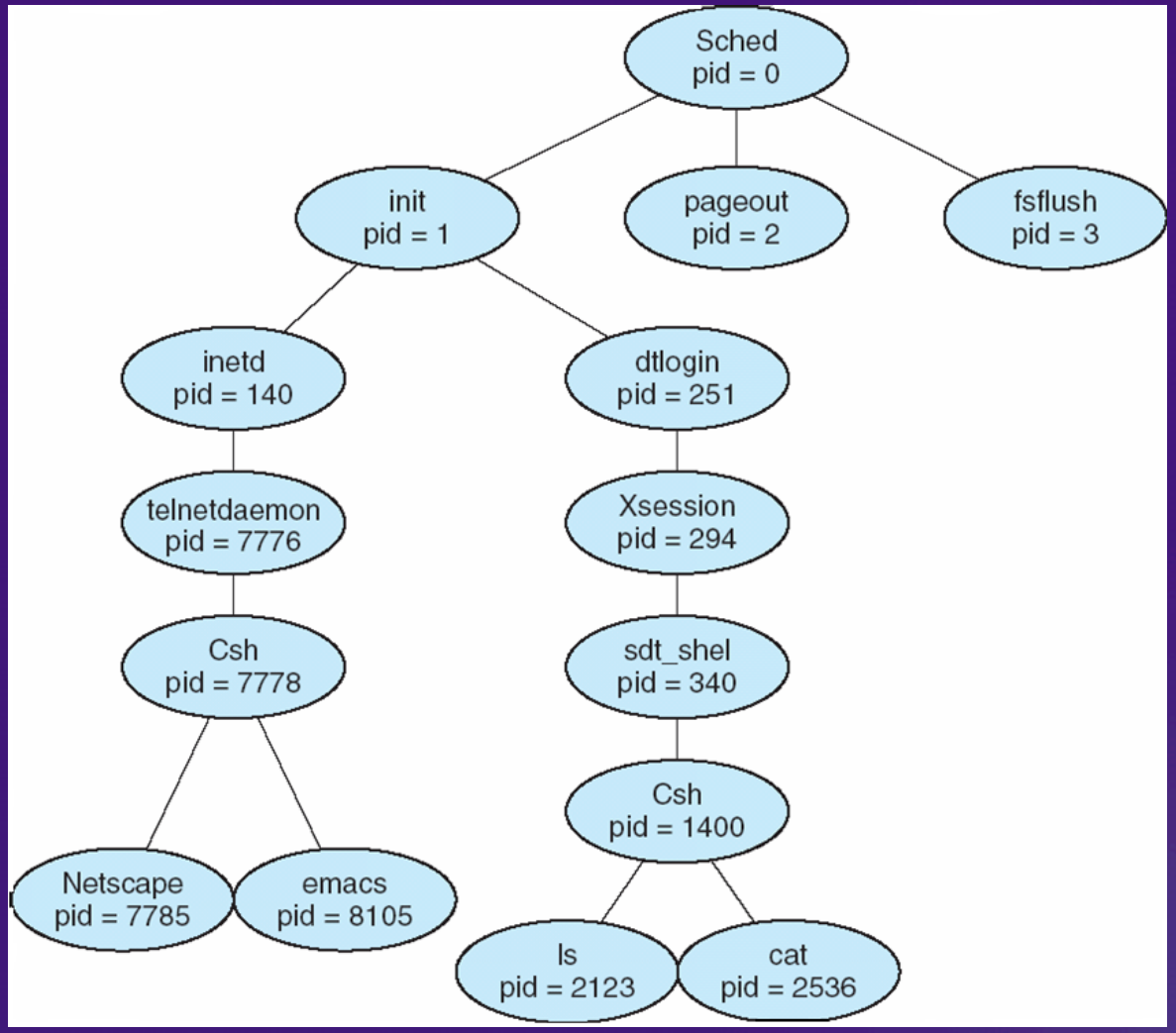
- Parent process creates children processes.
- Resource sharing
  - ✓ Resource inherited by children: file descriptors, shared memory and system queues
  - ✓ Resource not inherited by children: address space
- Execution
  - ✓ Parent and children execute concurrently.
  - ✓ Parent blocks on wait() system call until children terminate.
- UNIX examples
  - ✓ fork() system call creates new process.
  - ✓ execlp system call used after a fork() to replace the process' memory space with a new program.
- CSS430-unique ThreadOS: SysLib.exec and Syslib.join

C - Forking a Separate Process

```c
#include <stdio.h>   // for printf
#include <stdlib.h>  // for exit
#include <unistd.h>  // for fork, execlp

int main(int argc, char *argv[])
{
   int pid; // process ID
   // fork another process
   pid = fork();
   if (pid < 0) { // error occurred
      fprintf(stderr, "Fork Failed");
      exit(EXIT_FAILURE);
   }
   // ---------- CHILD SECTION -----------
   else if (pid == 0) {
      execlp("/bin/ls","ls","-l",NULL);
   }
   // ---------- PARENT SECTION ----------
   else {
      // parent will wait for the child to complete
      wait(NULL);
      printf("Child Complete");
      exit(EXIT_SUCCESS);
   }
}
```



A Tree of Processes On A Typical Unix System



Process Termination

- Process termination occurs when
  - ✓ last statement is executed
  - ✓ **exit()** system called explicitly
- Upon process termination
  - ✓ Termination code is passed from:
    child (via **exit()**) → parent (via **wait()**).
  - ✓ Process resources are deallocated by OS.
- Parent may terminate execution of children processes (via **kill()**) when:
  - ✓ Child has exceeded allocated resources.
  - ✓ Task assigned to child is no longer required.
  - ✓ Parent is exiting (cascading termination).
    - ❖ Some operating systems do not allow child to continue if its parent terminates.

Discussion 1

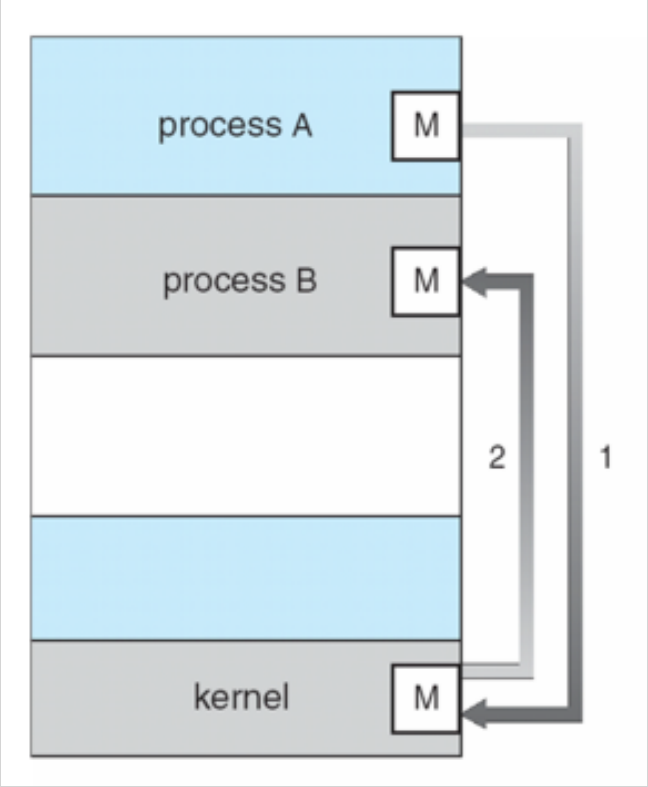
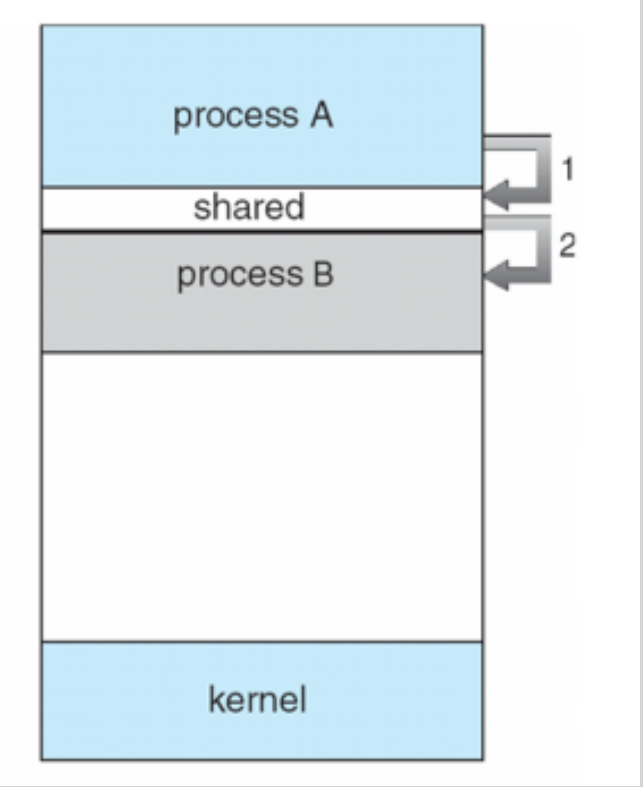
① What are the differences between CPU bound and I/O bound processes?
② What is another name for the *text section* in a process and what does it contain?
③ Discuss the differences between short-term, medium-term, and long-term scheduling in the following table:

Scheduler Type	Source of Processes	(approx.) period of execution (sec)	Key Role
Short Term			
Medium Term			
Long Term			

Cooperating Processes

- *Process independency:* Processes belonging to different users do not affect each other unless they give each other certain access permissions
- *Process Cooperation*: Processes spawned from the same user process share some resources and communicate with each other through these resources (e.g. shared memory, message queues, pipes, and files)
- Advantages of process cooperation
  - ✓ Information sharing: (sharing files)
  - ✓ Computation speed-up: (parallel programming)
  - ✓ Modularity: (like who | wc –l, one process lists current users and another counts the number of users.)
  - ✓ Convenience: (e.g. web-surfing while working on programming with vim and g++)

Communication Models

Message Passing	Shared Memory
	

Message Passing

- Message system – processes communicate with each other without resorting to shared variables.
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

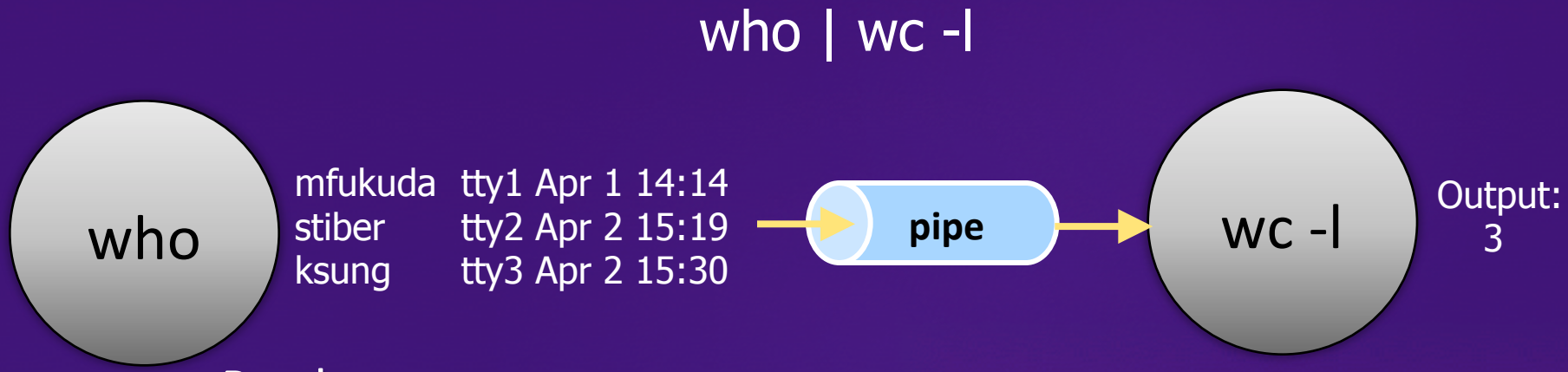
Direct Communication

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- How can a process locate its partner to communicate with?
  - Processes are created and terminated dynamically and thus a partner process may have gone.
  - Direct communication takes place between a parent and its child process in many cases.

    Example: pipe(fd)

Producer-Consumer Problems

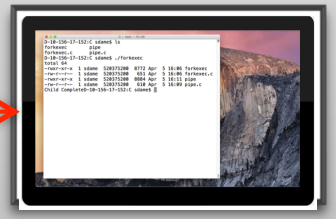


- Producer process:
  - ✓ **who** produces a list of current users.
- Consumer process
  - ✓ **wc** receives it for counting # of users (i.e. *lines*).
- Communication link:
  - ✓ OS provides a pipe.

File Descriptors

File Descriptors		
Name	**File No**	**Description**
Std In	0	Key Input
Std Out	1	Out from Console
Std Error	2	Error Output to Console
Hdrive file	3	R/W File
…	…	…
CD ROM file	7	R/O File
…	…	…
Hdrive file	42	W/O File

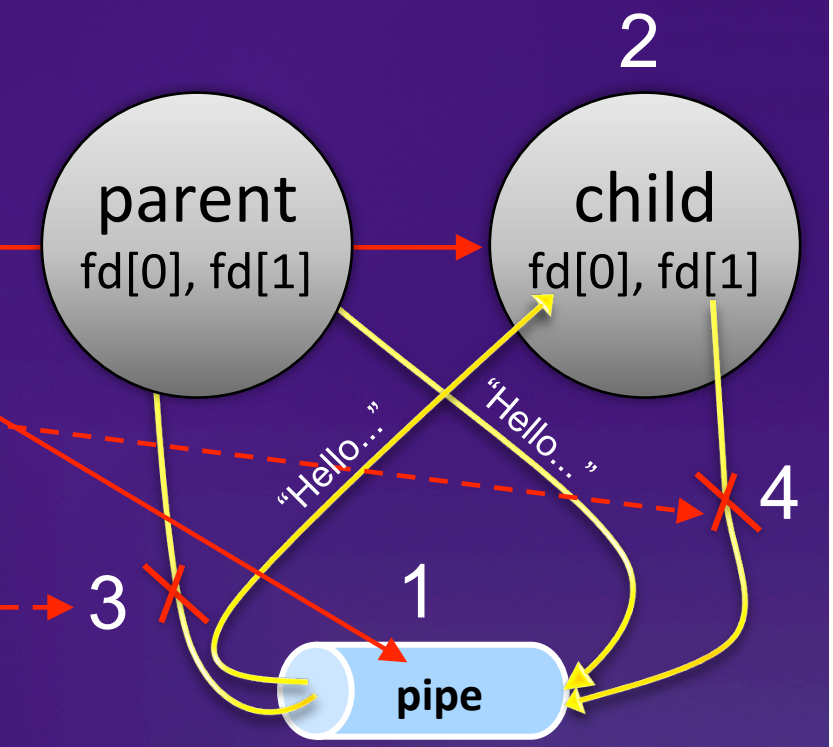
Device Layer
/dev/input
/dev/tty
/dev/sda
/dev/sr0



Direct Communication

Filename: pipe.c

```c
#include <unistd.h>     // for fork, pipe

int main( void ) {
   enum {RD, WR}; // pipe fd index RD=0, WR=1
   int n, fd[2];
   pid_t pid;
   char buf[100];

   if( pipe(fd) < 0 ) // 1: pipe created
      perror("pipe error");
   else if ((pid = fork()) < 0) // 2: child forked
      perror("fork error");

   else if (pid == 0) {                    Child process
      close(fd[WR]);// 4: child's fd[1] closed
      n = read(fd[RD], buf, 100);
      write(STDOUT_FILENO, buf, n);
   }

   else {
      close(fd[RD]); // 3: parent's fd[0] closed
      write(fd[WR], "Hello my child\n", 15);
      wait(NULL);
   }
                                        Parent process
}
```



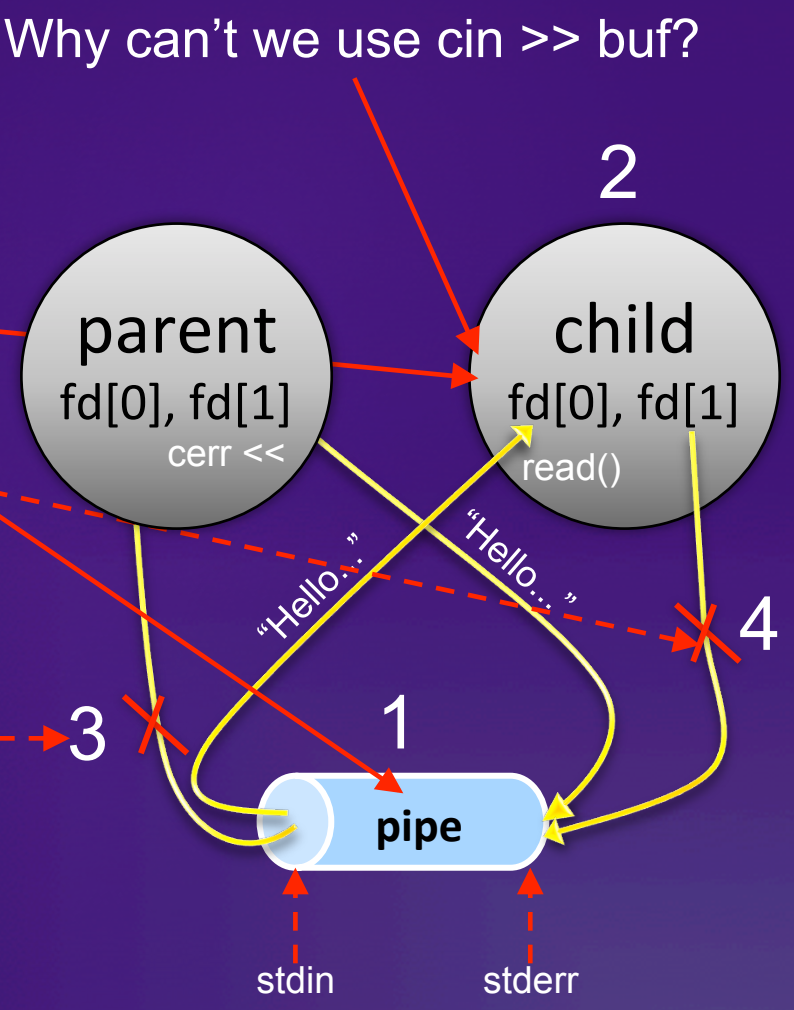
pipe, dup2 in C++

Filename: pipe.cpp

```
#include <stdlib.h>     // for exit
#include <stdio.h>      // for perror
#include <unistd.h>     // for fork, pipe
#include <sys/wait.h>   // for wait
#include <iostream>     // for cerr, cout
using namespace std;

int main( void ) {
   enum {RD, WR}; // pipe fd index RD=0, WR=1
   int n, fd[2];
   pid_t pid;

   if( pipe(fd) < 0 ) // 1: pipe created
      perror("pipe error");
   else if ((pid = fork()) < 0) // 2: child forked
      perror("fork error");
   else if (pid == 0) {

      close(fd[WR]);// 4: child's fd[1] closed
      dup2(fd[RD], 0); // stdin(0) --> childs pipe read

      char buf[256];
      n = read(fd[RD], buf, 256); // use this raw read!
      // cin >> buf; <-- *caution with cin and white space!
      cout << buf;       // write to stdout
      cout << "Child Done!" << endl;
   }
   else {

      close(fd[RD]); // 3: close parent's read end of pipe
      dup2(fd[WR], 2); // stderr(2) --> parent's pipe write

      cerr <<  "Hello my child" << endl;
      wait( NULL );
      cout << "Parent Done!" << endl;
   }
   exit(EXIT_SUCCESS);
}
```

Child process

Parent process



pipe, dup2, execlp in C++

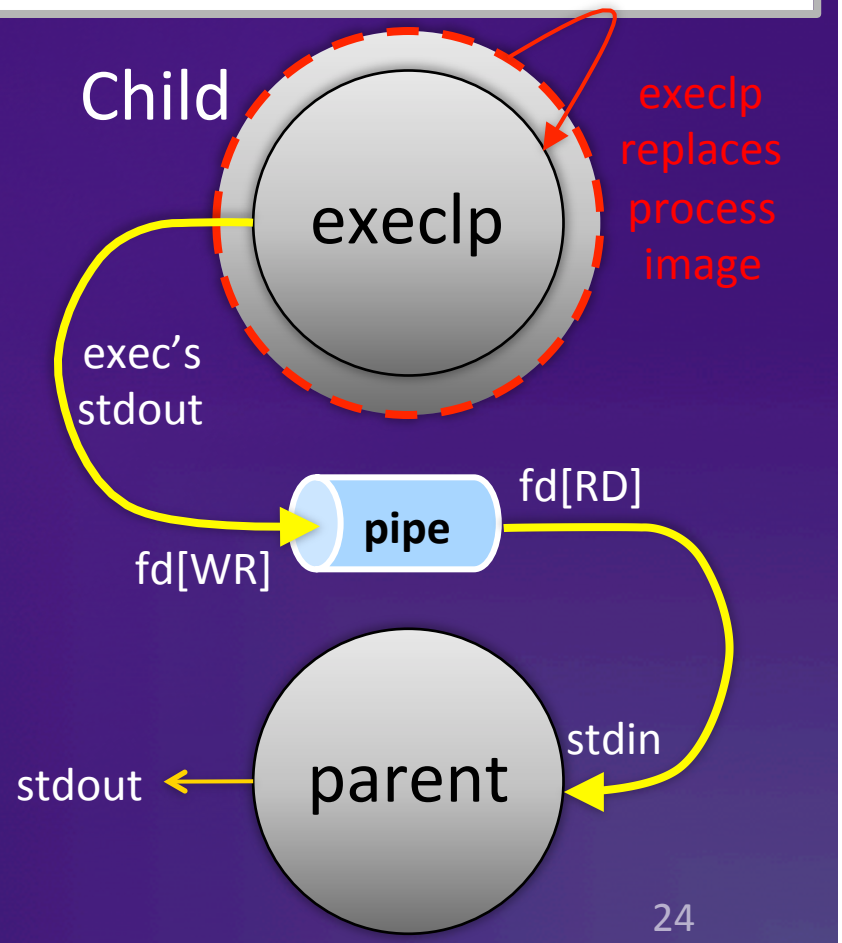
Filename: pipe_exec.cpp

```
#include <stdlib.h>     // for exit
#include <stdio.h>      // for perror
#include <unistd.h>     // for fork, pipe
#include <sys/wait.h>   // for wait
#include <iostream>     // for cerr, cout
#define MAXSIZE 4096
using namespace std;

int main( void ) {
   enum {RD, WR}; // pipe fd index RD=0, WR=1
   int n, fd[2];
   pid_t pid;

   if( pipe(fd) < 0 ) // 1: pipe created
      perror("pipe error");
   else {
      switch(fork()) {
         case -1:
            perror("fork error");
         case 0:
         {
            close(fd[RD]);// 4: child's fd[0] closed
            dup2(fd[WR], 1); // stdout --> childs pipe write
            execlp("/bin/ls","ls","-l", NULL);
            // Never returns here!!
         }
         default:
         {
            close(fd[WR]); // 3: close parent's write end of pipe
            dup2(fd[RD], 0); // stdin --> parent's pipe read
            wait( NULL );

            char buf[MAXSIZE];
            n = read(fd[RD], buf, MAXSIZE); // use this raw read!
            buf[n] = '\0'; // make sure cstring is terminated
            cout << buf;   // write to stdout
            cout << "Parent Done!" << endl;
         }
      }
   }
   exit(EXIT_SUCCESS);
}
```

Child process

Parent process

```
# ./pipe_exec
total 128
-rwxr-xr-x  1 Steve  Steve  9788 Apr  6 18:27 pipe
-rw-r--r--  1 Steve  Steve  1576 Apr  6 17:11 pipe.cpp
-rwxr-xr-x  1 Steve  Steve  9784 Apr  6 18:52 pipe_exec
-rw-r--r--  1 Steve  Steve  1723 Apr  6 18:51
pipe_exec.cpp
Parent Done!
```



Indirect Communication (non-hard coded IDs)

Indirect Communication (Message Queues)

- Messages are directed and received from mailboxes (also referred to as ports).
  - ✓ Each mailbox has a unique id.
  - ✓ Processes can communicate only if they share a mailbox.
- Processes must know only a mailbox id. They do not need to locate their partners
  - ✓ Example: message queue

Example: Message Queues

msg_snd.cpp

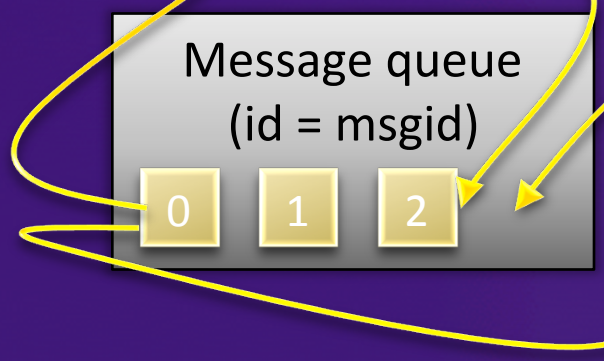
```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <iostream>
using namespace std;

struct mymesg {
   long mytype;
   char mtext[512];
} message_body;

int main( void ) {
   int msgid = msgget( 100, IPC_CREAT );
   strcpy( message_body.mtext, "hello world\n" );
   msgsnd( msgid, &message_body, 512, 0 );
}
```

key=100

msg_rcv.cpp

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <iostream>
using namespace std;

struct mymesg {
   long mytype;
   char mtext[512];
} message_body;

int main( void ) {
   int msgid = msgget( 100, IPC_CREAT );
   msgrcv( msgid, &message_body, 512, 0, 0 );
   cout << message_body.mtext << endl;
}
```

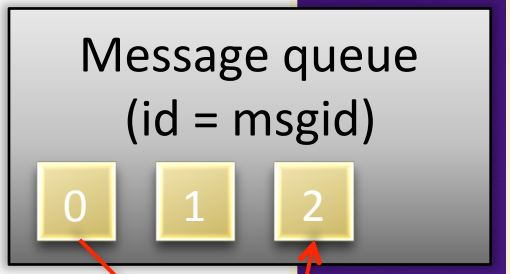
key=100



Some other process can also enqueue and dequeue a message (on the same key)

msg_snd.cpp → msg_rcv.cpp

```cpp
#include <stdlib.h>  // for exit
#include <stdio.h>   // for perror
#include <sys/ipc.h> // for IPC_CREAT
#include <sys/msg.h> // for msgget, msgrcv
#include <iostream>  // for cin, cout, cerr
#include <string.h>  // for strcpy, strlen
using namespace std;
#define MSGSZ 128

typedef struct {
   long mtype;
   char mtext[MSGSZ];
} message_buf;

main() {
   int msqid;
   size_t msgsz;
   int msgflg = IPC_CREAT | 0666;
   message_buf mymsg;

   key_t key = 74563;
   if((msqid = msgget(key, msgflg)) < 0) {
      perror("msgget");
      exit(1);
   }
   cout << "Sending to Msg Queue:" << key << endl;
   strcpy( mymsg.mtext, "Hello Huskies!");
   msgsz = strlen(mymsg.mtext) + 1;
   mymsg.mtype = 1;

   if(msgsnd(msqid,&mymsg,msgsz,IPC_NOWAIT) < 0)
   {
      perror("msgsnd");
   }
}
```




```cpp
#include <stdlib.h>  // for exit
#include <stdio.h>   // for perror
#include <sys/ipc.h> // for IPC_CREAT
#include <sys/msg.h> // for msgget, msgrcv
#include <iostream>  // for cin, cout, cerr
using namespace std;
#define MSGSZ 128

typedef struct {
   long mtype;
   char mtext[MSGSZ];
} message_buf;

main() {
   int msqid;
   size_t msgsz;
   int msgflg = IPC_CREAT | 0666;
   message_buf mymsg;

   key_t key = 74563;
   if((msqid = msgget(key, msgflg)) < 0) {
      perror("msgget");
      exit(1);
   }

   if(msgrcv(msqid, &mymsg, MSGSZ, 1, 0) < 0) {
      perror("msgrcv");
      exit(1);
   }
   cout << mymsg.mtext << endl;
}
```

Shared Mailbox (Java)

```java
import java.util.Vector;

public class MessageQueue<E> implements Channel<E>
{
    private Vector<E> queue;

    // a "generic" vector queue
    public MessageQueue() {
        queue = new Vector<E>();
    }
    // nonblocking send
    public void send(E item) {
        queue.addElement(item);
    }
    // nonblocking receive
    public E receive() {
        if (queue.size() == 0)
            return null;
        else
            return queue.remove(0);
    }
}
```

```java
public interface Channel<E>
{
    public void send(E item);
    public E receive();
}
```

Makes use of generics <E> And Vector<E> to implement a simple Date message mailbox.

```java
import java.util.Date;

public class Test
{
    public static void main(String[] args) {
        Channel<Date> mailBox = new MessageQueue<Date>();
        mailBox.send(new Date());

        Date rightNow = mailBox.receive();
        System.out.println(rightNow);
    }
}
```

Next chapter will show sharing of this mailbox between threads.

Synchronization

- Sending Process
  - ✓ Non-Blocking – Sends and resumes execution
  - ✓ Blocking – Sender is blocked until message is received or accepted by buffer.
- Receiving Process
  - ✓ Non-Blocking – receives valid or NULL
  - ✓ Blocking – Waits until message arrives

Discussion 2

① In the previous example, what will happen if you log into uw1-320-lab run msg_rcv(), then log in from a second shell to uw1-320-lab and run msg_snd()?

② What do Vector<E> and Java Generic do for us?

③ What would be two methods of connecting machines together geographically separated from an indirect ID perspective?

Buffering**

- Established by creating a shared memory area between a *Producer* and a *Consumer*.
- Queue of messages attached to the link; implemented in one of three ways:
  1. Zero capacity – 0 messages
     Producer must wait for Consumer (rendezvous).
  2. Bounded capacity – finite length of *n* messages
     Producer must wait if link is full (This happens in practical world like sockets).
  3. Unbounded capacity – "infinite" length
     Producer never waits. (Non-blocking send)

** Applies to both Direct and Indirect Communications

Shared Memory

Bounded Buffer

- Communication link or media has a bounded space to buffer in-transfer data.

```
import java.util.*;

public class BoundedBuffer
{
   public static final int BUFFER_SIZE = 5;
   private int count;   // #items in buffer
   private int in;      // points to the next free position
   private int out;     // points to the next full position
   private Object[] buffer;   // a reference to buffer

   public BoundedBuffer( ) {
      count = 0;
      in = 0;
      out = 0;
      buffer = new Object[BUFFER_SIZE];
   }
   public void enter( Object item ) {...}
   public object remove( ) {...}
}
```

See also (for C++ references):

Beej's Guide to Unix IPC - shm

Producer and Consumer Processes

Producer Process

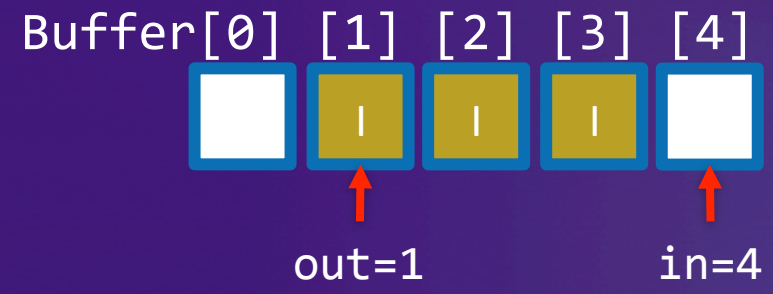
```
for(int i = 0;; i++ ) {
   BoundedBuffer.enter(new Integer(i));
}
```

```
public void enter( Object item ) {
    while ( count == BUFFER_SIZE )
       ; // buffer is full! Wait till buffer is consumed
    ++count;
    buffer[in] = item; // add an item
    in = ( in + 1 ) % BUFFER_SIZE; // circular buffer
}
```

Consumer Process

```
for(int i = 0; ; i++ ) {
   BoundedBuffer.remove( );
}
```

```
public object remove( ) {
    Object item;
    while ( count == 0 )
       ; // buffer is empty! Wait till buffer is filled
    -- count;
    item = buffer[out]; // pick up an item
    out = ( out + 1 ) % BUFFER_SIZE; // circular buffer
}
```



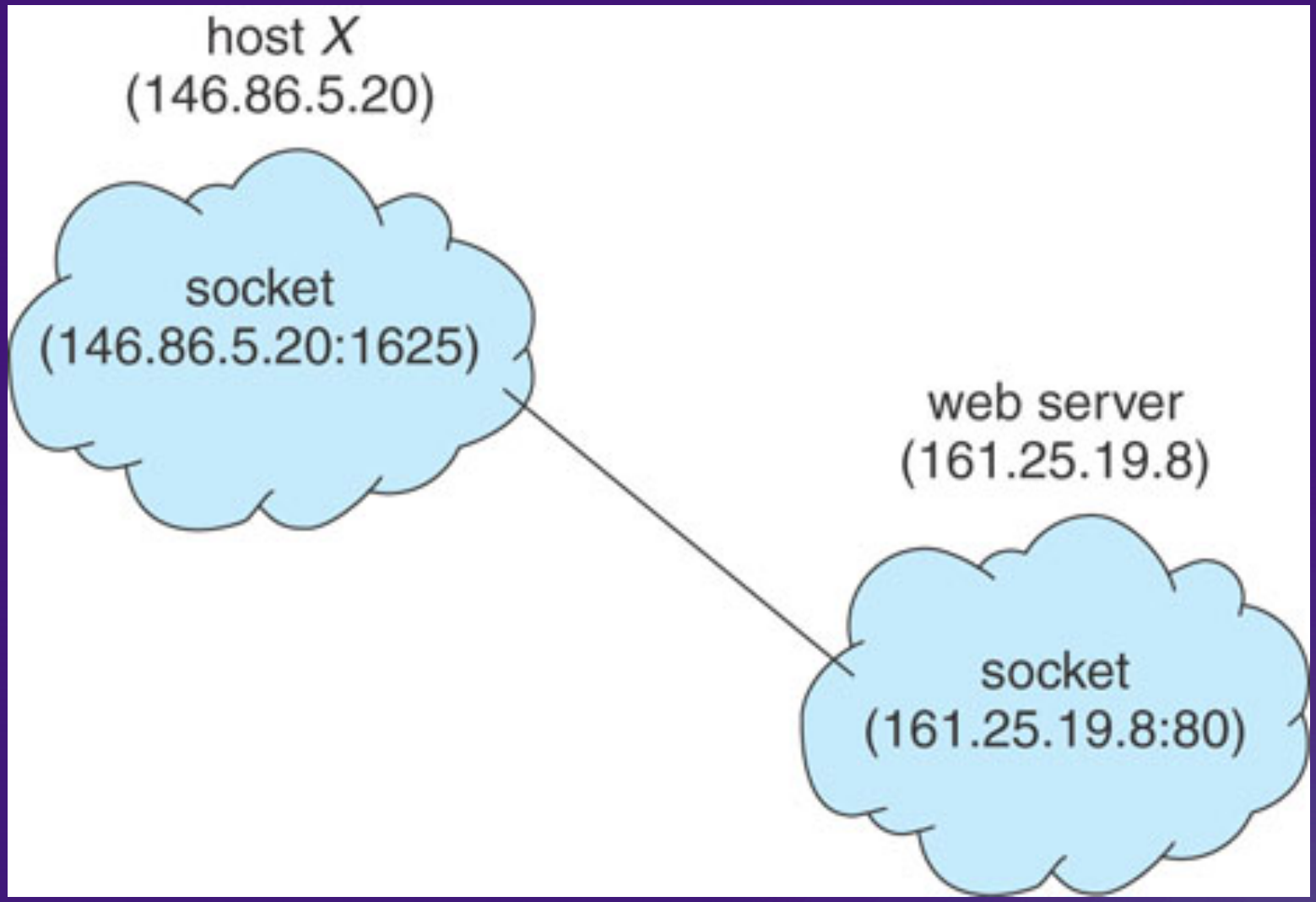
Client-Server Systems

- Sockets
- Remote Procedure Calls (RPC)
- Remote Method Invocation (RMI)

Sockets

- Common way to connect hosts at different locations and/or IP
- Low-level form of communication between distributed processes
- Allows only an unstructured stream of data

Socket Connection



Datagram Sockets

- User Datagram Protocol (UDP)
- Also know as "connectionless" sockets
- (can be) Point to multi-point
- Not guaranteed to be delivered
- Java Classes for UDP:

  DatagramPacket, DatagramSocket, MulticastSocket

Stream Sockets

- Transmission Control Protocol (TCP)
- Also know as "connection-oriented" sockets
- Point to point
- Guaranteed to be delivered
- (ftp, http, telnet, ssh, etc.)
- Java Classes for TCP:

  URL, URLConnection, Socket, SocketServer

(TCP) Socket Client-Server

```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);
      }
   }
}
```

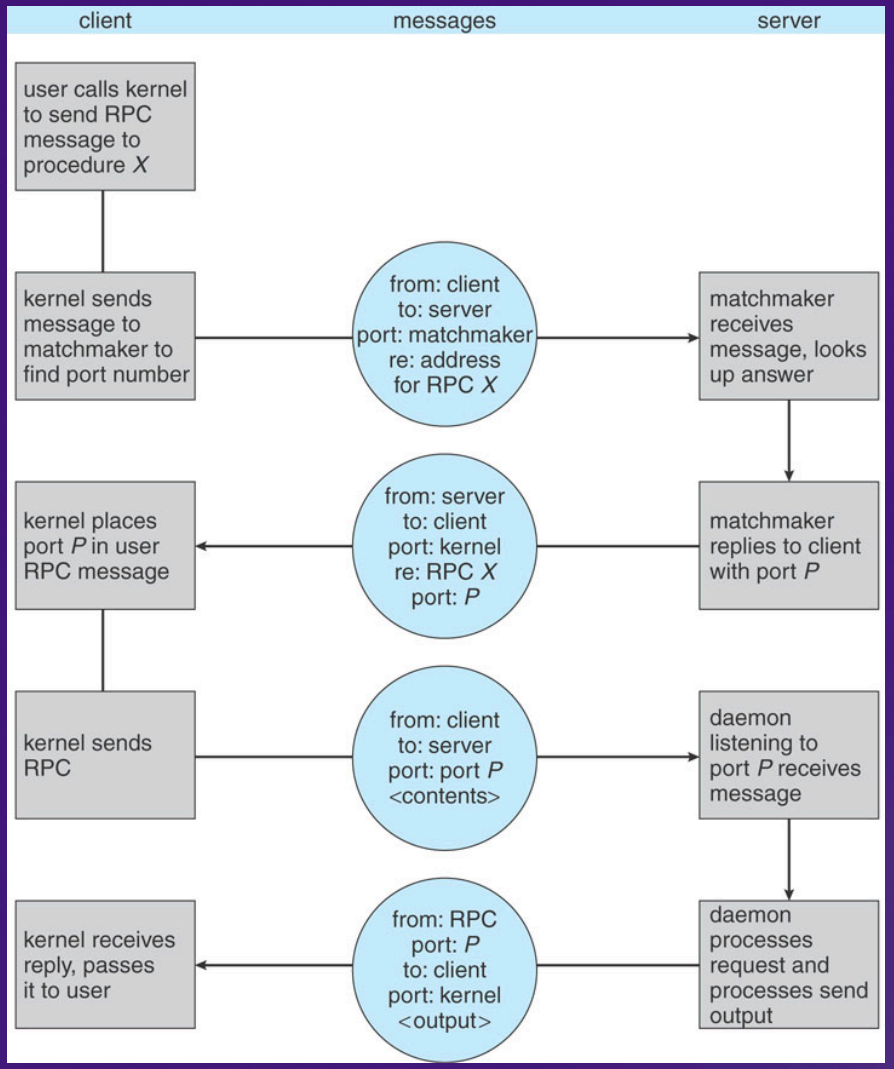
Server

```java
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);
      }
   }
}
```

Client

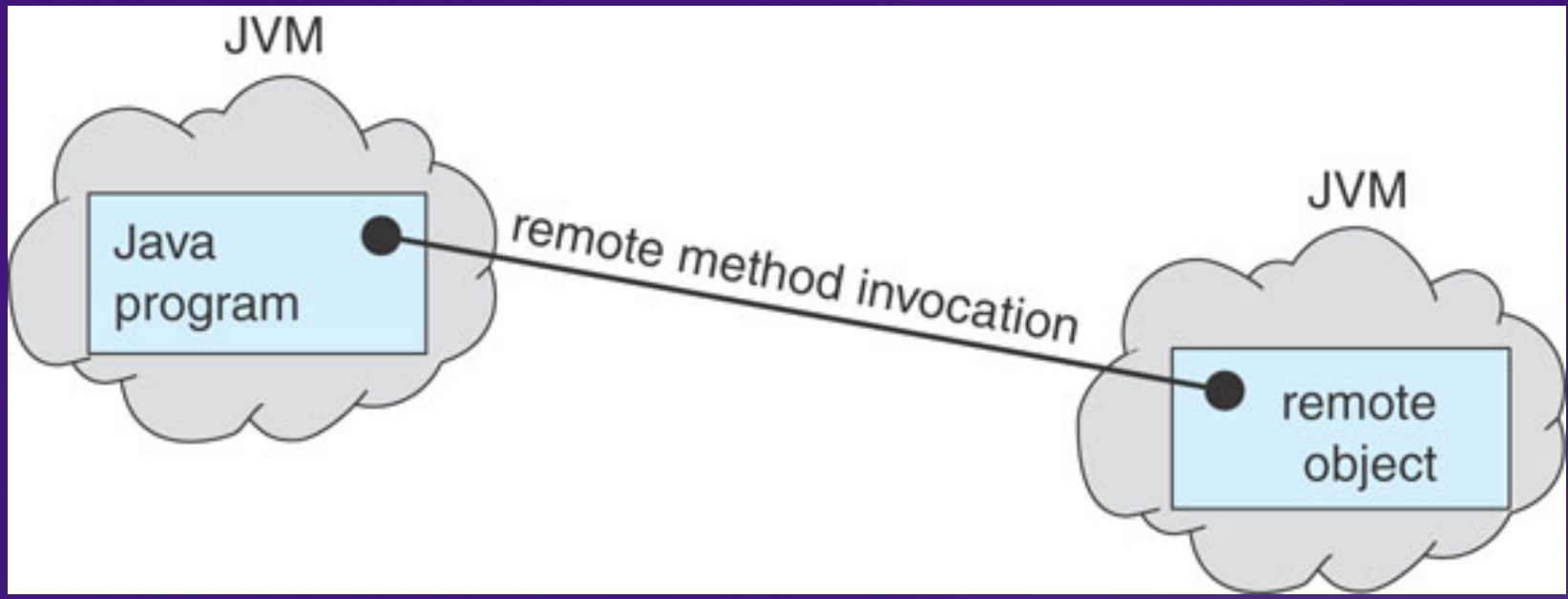
RPC

- C/C++ Only
- UDP Based

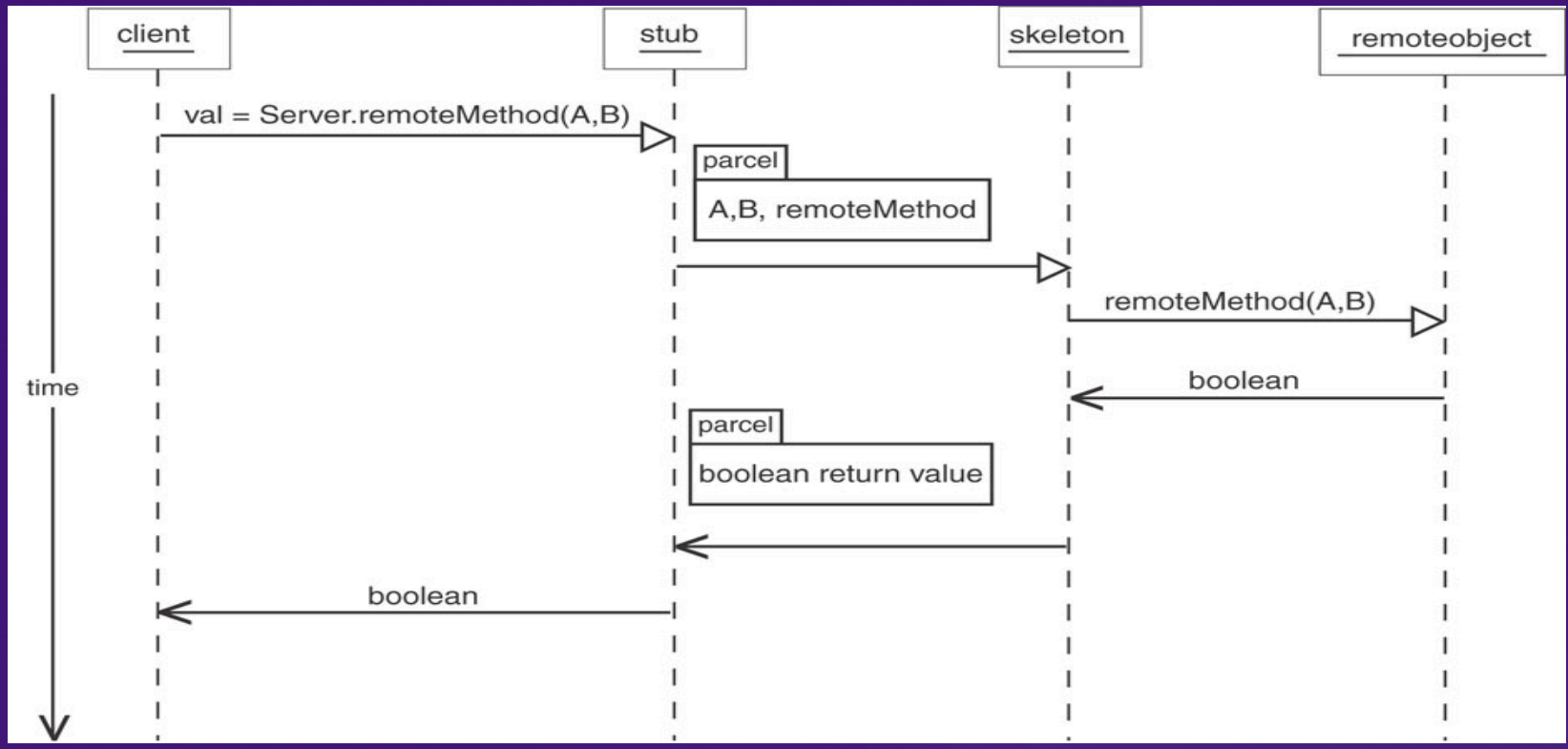


Remote Method Invocation (RMI)

- Java Based
- TCP Based Communications (i.e. more reliable)



Sequence Diagram (RMI)



- Stub provides an interface on the client side, which maps to a port on the remote side.
- Marshalling - involves packaging the parameters into a form that can be transmitted over a network

ThreadOS SysLib

```
public interface SysLib {
    public static int exec( String args[] );
    public static int join( );
    public static int boot( );
    public static int exit( );
    public static int sleep( int milliseconds );
    public static int disk( );
    public static int cin( StringBuffer s );
    public static int cout( String s );
    public static int cerr( String s );
    public static int rawread( int blkNumber, byte[] b );
    public static int rawwrite( int blkNumber, byte[] b );
    public static int sync( );
    public static int cread( int blkNumber, byte[] b );
    public static int cwrite( int blkNumber, byte[] b );
    public static int flush( );
    public static int csync( );
    public static String[] stringToArgs( String s );
    public static void short2bytes( short s, byte[] b, int offset );
    public static short bytes2short( byte[] b, int offset );
    public static void int2bytes( int i, byte[] b, int offset );
    public static int bytes2int( byte[] b, int offset );
}
```

Shell.java (starter example)

```
import java.io.*;
import java.util.*;

class Shell extends Thread
{
   //command line string to contain the full command
   private String cmdLine;

   // constructor for shell
   public Shell( ) {
      cmdLine = "";
   }

   // required run method for this Shell Thread
   public void run( ) {

      // build a simple command that invokes PingPong
      cmdLine = "PingPong abc 100";

      // must have an array of arguments to pass to exec()
      String[] args = SysLib.stringToArgs(cmdLine);
      SysLib.cout("Testing PingPong\n");

      // run the command
      int tid = SysLib.exec( args );
      SysLib.cout("Started Thread tid=" + tid + "\n");

      // wait for completion then exit back to ThreadOS
      SysLib.join();
      SysLib.cout("Done!\n");
      SysLib.exit();
   }
}
```

Week 2 Homework

1. In Unix, the first process is called **init**. All the others are descendants of "init". The **init** process spawns a **sshd** process that detects a new secure ssh requested connection (WKPort 22). Upon a new connection, **sshd** spawns a login process that then loads a **shell** on it when a user successfully logs into the system. Now, assume that the user types **who | grep <uwnetid> | wc –l**. Draw a process tree from **init** to those three commands. Add **fork**, **exec**, **wait**, and **pipe** system calls between any two processes affecting each other.

2. **Consider four different types of inter-process communication (IPC).**

a)	Pipe:	implemented with pipe, read, and write
b)	Socket:	implemented with socket, read, and write
c)	Shared memory:	implemented shmget, shmat, and memory read/write
d)	Shared message queue:	implemented with msgget, msgsnd, and msgrcv

   1. Which types are based on direct communication?
   2. Which types of communication do not require parent/child process relationship?
   3. If we code a produce/consumer program, which types of communication require us to implement process synchronization?
   4. Which types of communication can be used to communicate with a process running on a remote computers?
   5. Which types of communication must use file descriptors?
   6. Which types of communication need a specific data structure when transferring data?