1

Processes

COS450 - Fall 2018

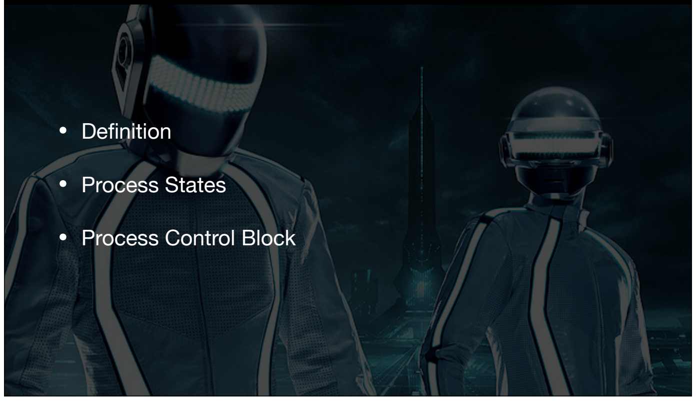
2

Processes

- What is a ***process***?
- ***Scheduling*** processes
- ***Cooperating*** and ***Communication***

3

What is a Process



- Definition
- Process States
- Process Control Block

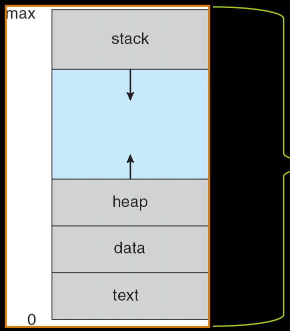
Definition

4

A **Process** is a **program** in execution.

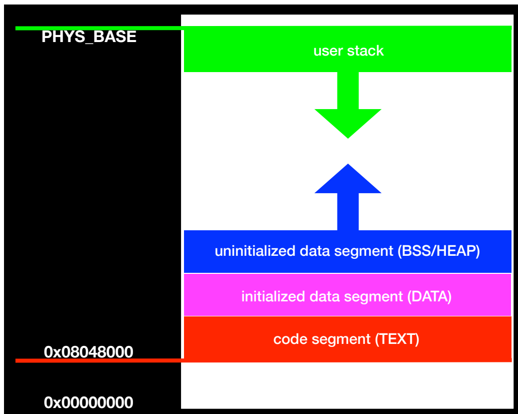
a process has:
- a program counter
- a stack
- data
- code, files, ...

5



Process in Memory

6



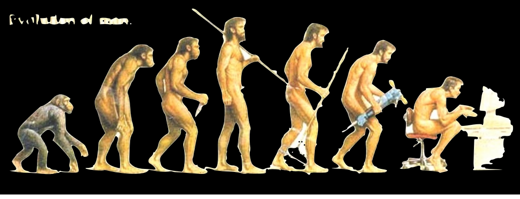
PHYS_BASE
user stack
uninitialized data segment (BSS/HEAP)
initialized data segment (DATA)
code segment (TEXT)
0x08048000
0x00000000

Pintos Process Memory

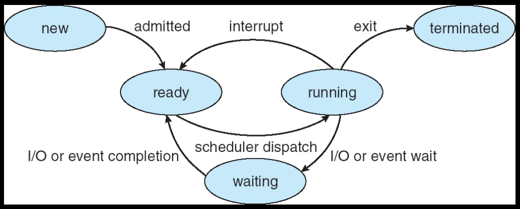
7

Process State

new	...being created
running	...being executed
waiting	...waiting for IO
ready	...waiting to execute
terminated	...waiting to die



8



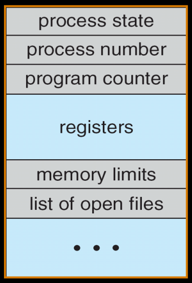
Process State

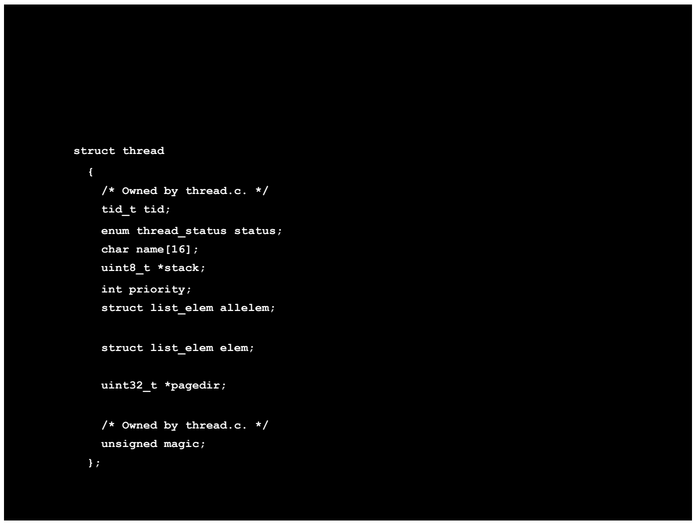
9

Process Control Block (PCB)

Every process has:

- state (as described)
- program counter
- registers (saved or active)
- Scheduling information
- Memory allocation
- File and IO allocation

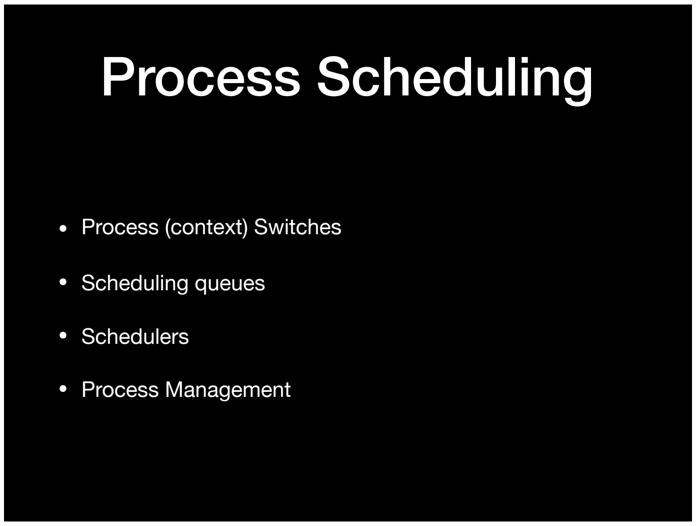




```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;
    enum thread_status status;
    char name[16];
    uint8_t *stack;
    int priority;
    struct list_elem allelem;

    struct list_elem elem;

    uint32_t *pagedir;

    /* Owned by thread.c. */
    unsigned magic;
  };
```

10

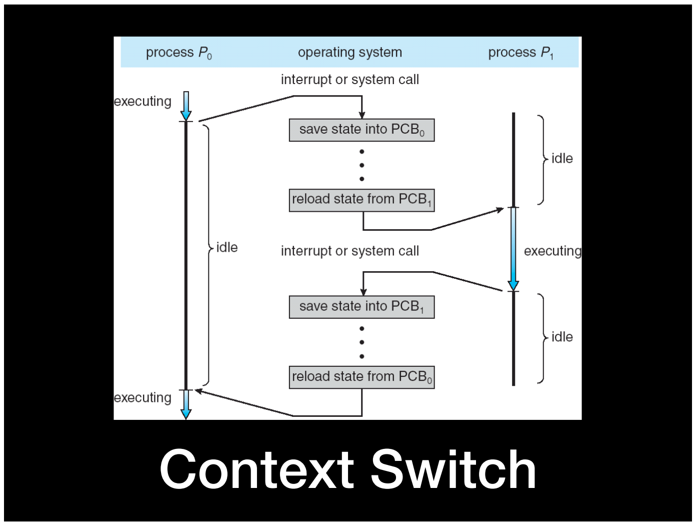
Registers get pushed on stack
Open file and other information is not here (yet) - Projects add it.



Process Scheduling

- Process (context) Switches
- Scheduling queues
- Schedulers
- Process Management

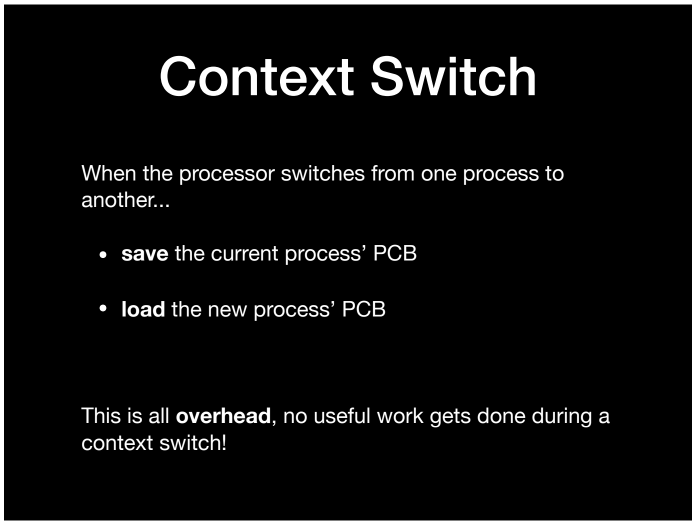
11



Context Switch

12

schedule() and switch_threads() in Pintos



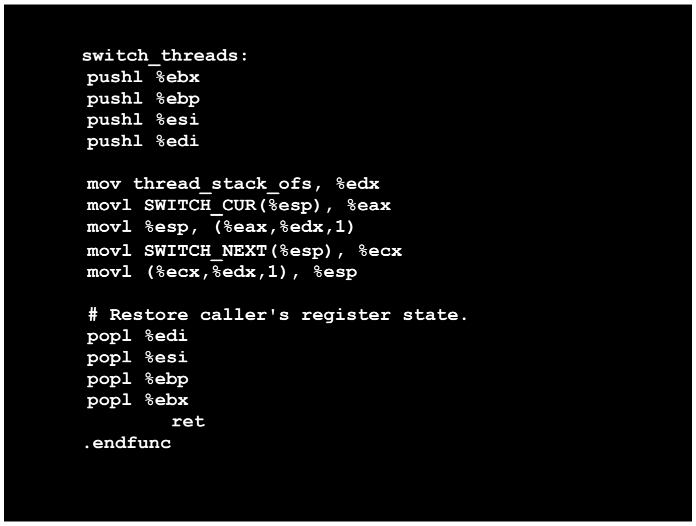
13

Context Switch

When the processor switches from one process to another...

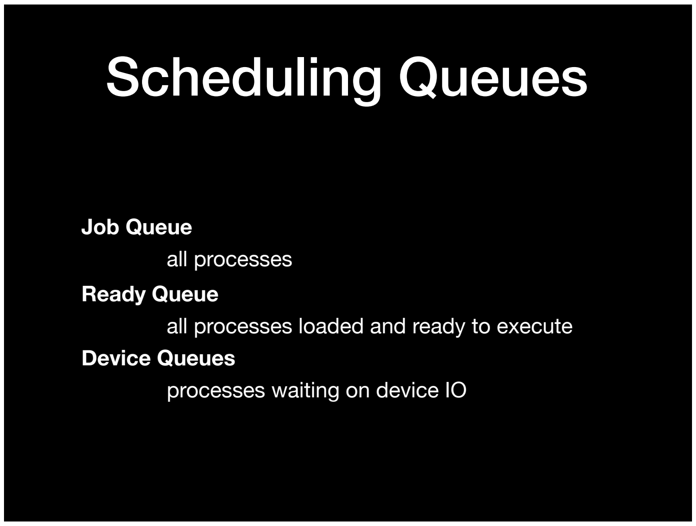
- **save** the current process' PCB
- **load** the new process' PCB

This is all **overhead**, no useful work gets done during a context switch!



14

```
switch_threads:
 pushl %ebx
 pushl %ebp
 pushl %esi
 pushl %edi

 mov thread_stack_ofs, %edx
 movl SWITCH_CUR(%esp), %eax
 movl %esp, (%eax,%edx,1)
 movl SWITCH_NEXT(%esp), %ecx
 movl (%ecx,%edx,1), %esp

 # Restore caller's register state.
 popl %edi
 popl %esi
 popl %ebp
 popl %ebx
        ret
.endfunc
```



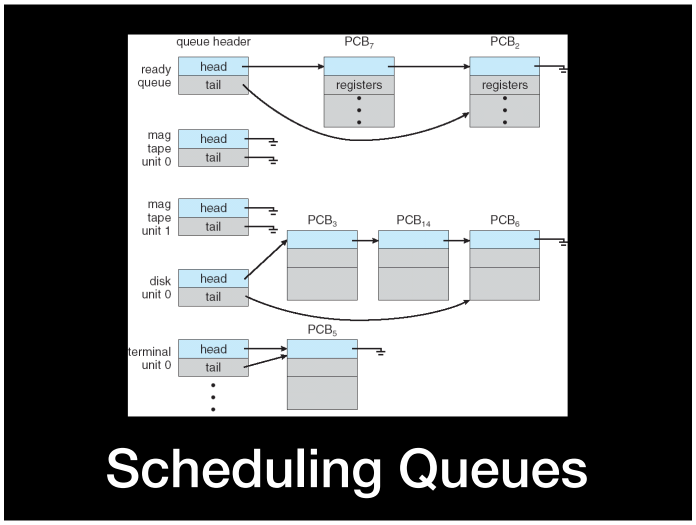
15

Scheduling Queues

Job Queue
all processes

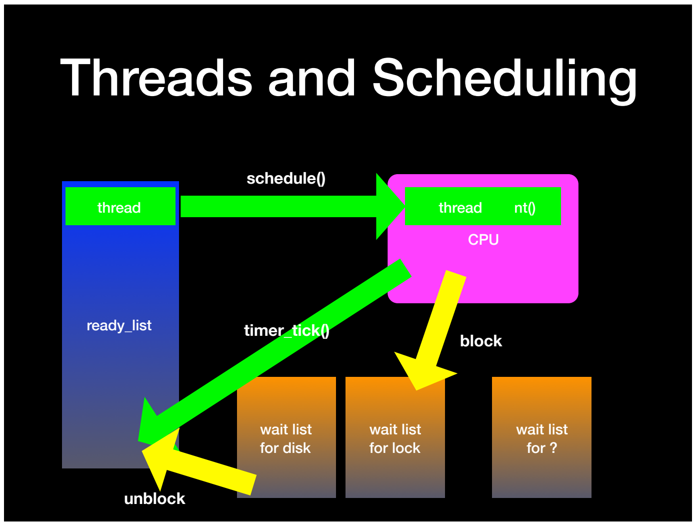
Ready Queue
all processes loaded and ready to execute

Device Queues
processes waiting on device IO



Scheduling Queues

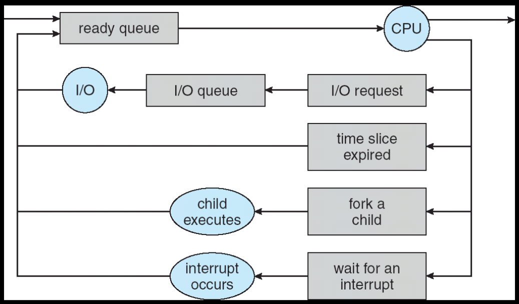
16



Threads and Scheduling

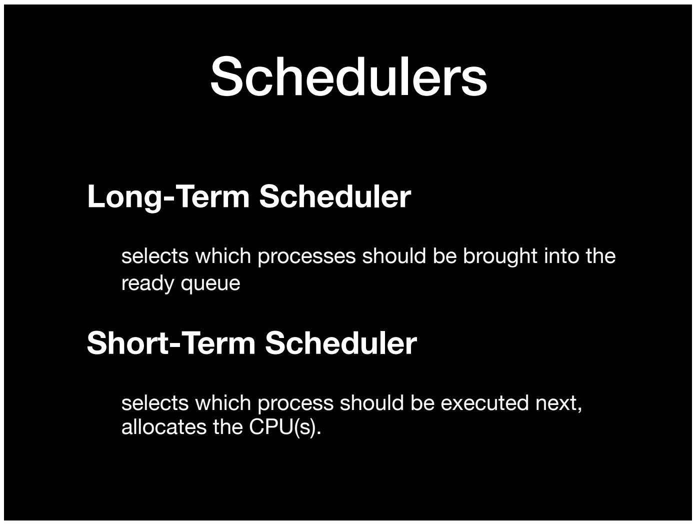
17

A rough sketch of Pintos' scheduling



Scheduling Queues

18



Schedulers

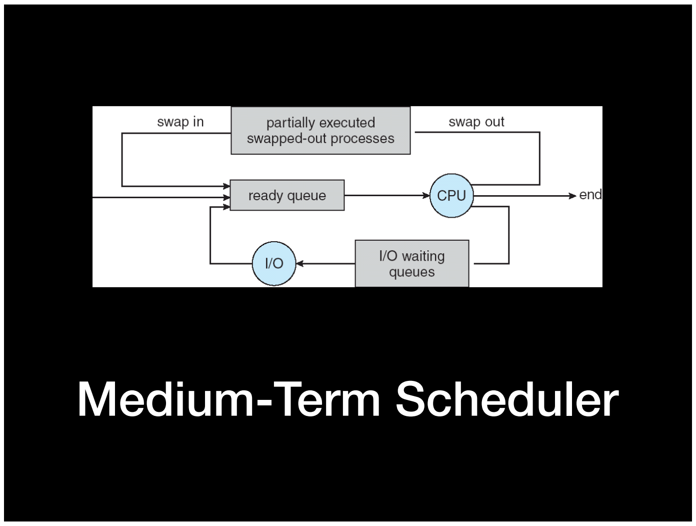
Long-Term Scheduler

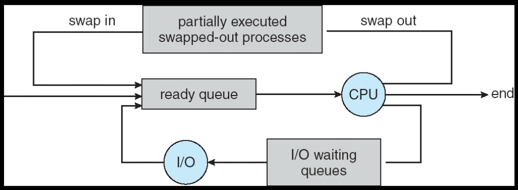
selects which processes should be brought into the ready queue

Short-Term Scheduler

selects which process should be executed next, allocates the CPU(s).

19

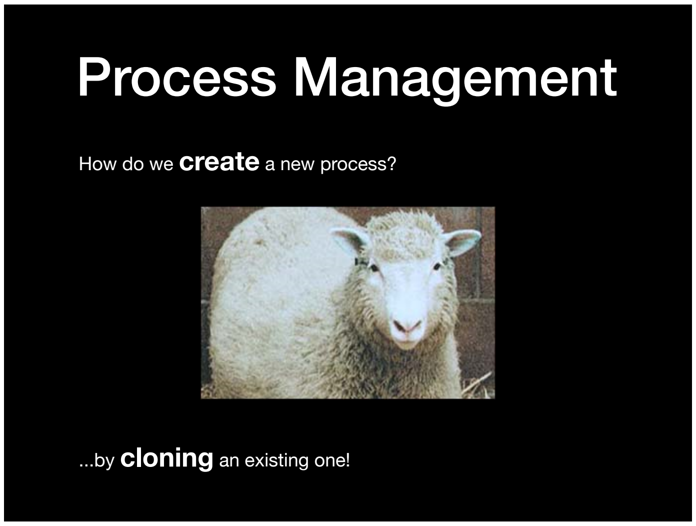




Medium-Term Scheduler

20

A medium-term scheduler is not common in today's operating systems.

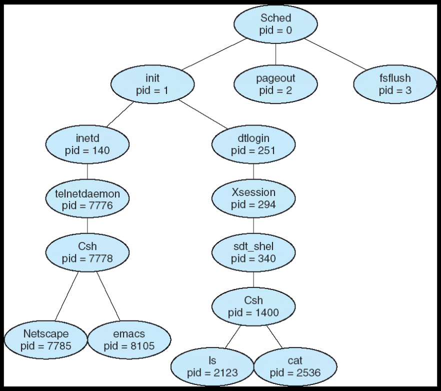


Process Management

How do we **create** a new process?

...by **cloning** an existing one!

21



Process Tree

Process Creation

Resource Sharing Options

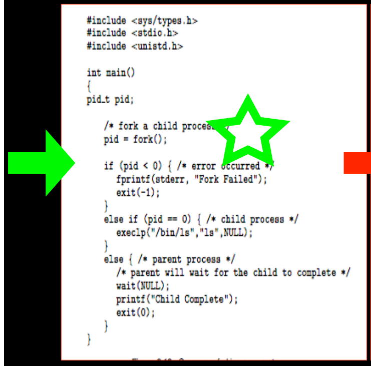
- Parent & Child share all
- Parent & Child share subset
- Parent & Child share none

Execution Options

- Concurrent execution
- Parent waits for child to finish

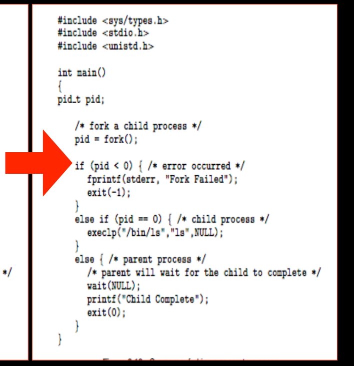
POSIX

Parent (pid=102)

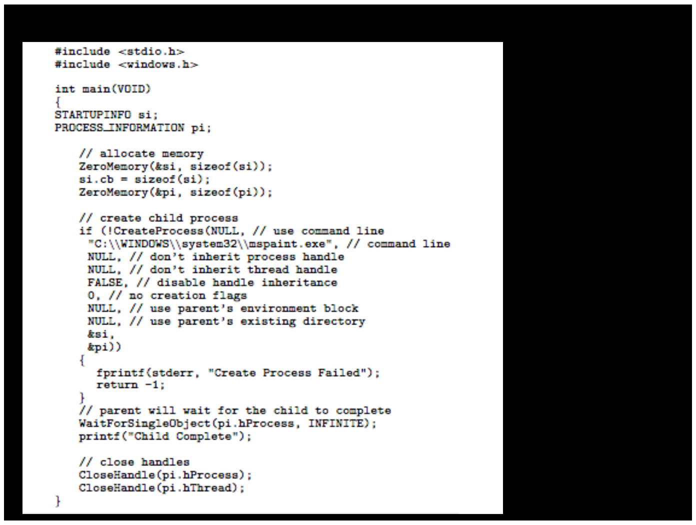


```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

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

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

Child (pid=0)

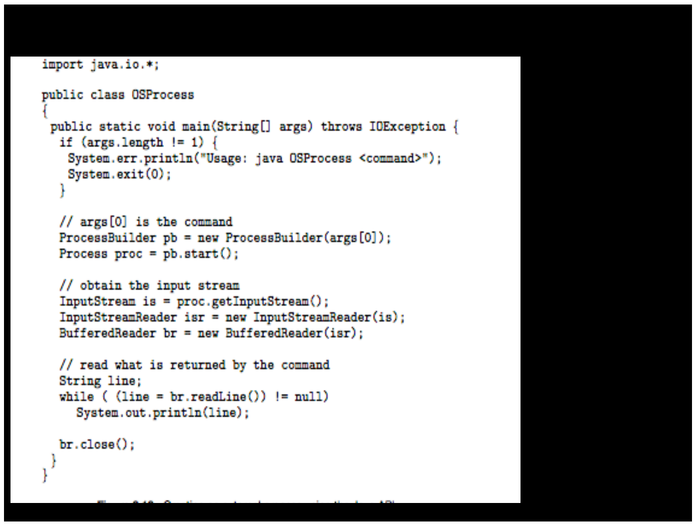


```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

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

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



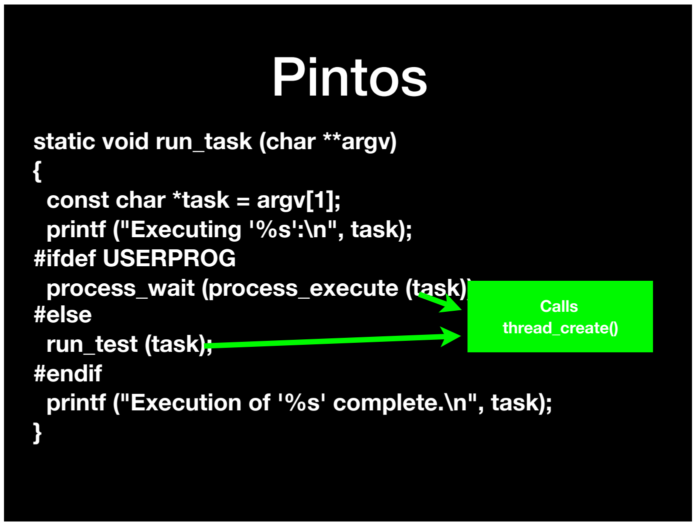
```
#include <stdio.h>
#include <windows.h>

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 line
     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);
}
```

25



```
import java.io.*;

public class OSProcess
{
 public static void main(String[] args) throws IOException {
  if (args.length != 1) {
   System.err.println("Usage: java OSProcess <command>");
   System.exit(0);
  }

  // args[0] is the command
  ProcessBuilder pb = new ProcessBuilder(args[0]);
  Process proc = pb.start();

  // obtain the input stream
  InputStream is = proc.getInputStream();
  InputStreamReader isr = new InputStreamReader(is);
  BufferedReader br = new BufferedReader(isr);

  // read what is returned by the command
  String line;
  while ( (line = br.readLine()) != null)
    System.out.println(line);

  br.close();
 }
}
```

26



Pintos

```
static void run_task (char **argv)
{
  const char *task = argv[1];
  printf ("Executing '%s':\n", task);
#ifdef USERPROG
  process_wait (process_execute (task));
#else
  run_test (task);
#endif
  printf ("Execution of '%s' complete.\n", task);
}
```

Calls
thread_create()

27

28

Processes

✓ What is a ***process***?

✓ ***Scheduling*** processes

- ***Cooperating*** and ***Communication***

29

Communication

- Inter Processes Communication
  - Shared Memory
  - Message Passing
    - Direct vs. indirect
- Synchronization Details
- Network Communications (Sockets)

30

Interprocess Communication

Processes communicate to get work done.

Sometimes they get it done faster.

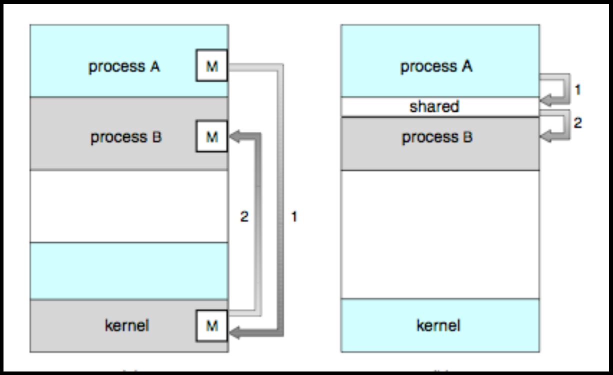
31

Producer - Consumer

Cooperating Processes

32

Message Passing Shared Memory



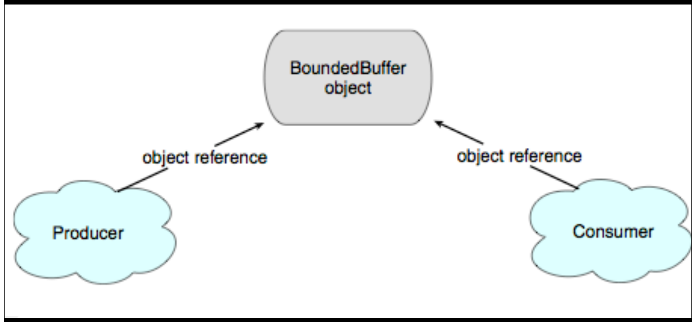
Message Passing

33

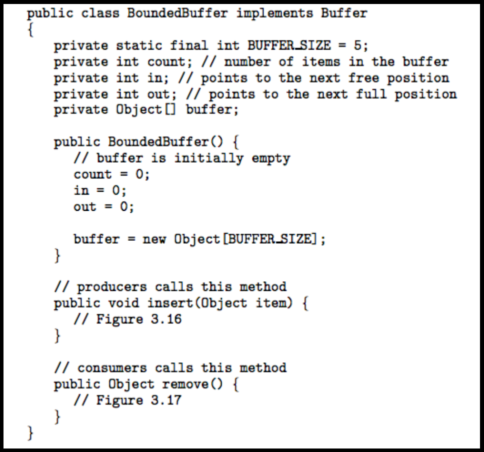
Shared Memory

- Processes **share** a section of memory
  - Producer adds items to buffer
  - Consumer removes them

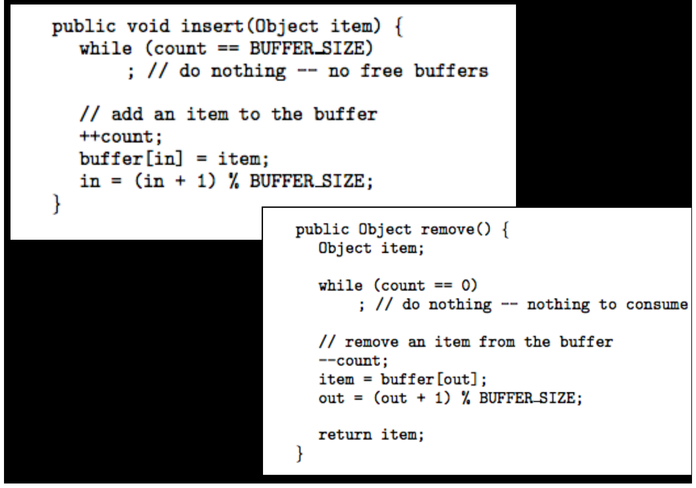
...lets look at some code



Example (Java)



```
public class BoundedBuffer implements Buffer
{
   private static final int BUFFER_SIZE = 5;
   private int count; // number of items in the buffer
   private int in; // points to the next free position
   private int out; // points to the next full position
   private Object[] buffer;

   public BoundedBuffer() {
      // buffer is initially empty
      count = 0;
      in = 0;
      out = 0;

      buffer = new Object[BUFFER_SIZE];
   }

   // producers calls this method
   public void insert(Object item) {
      // Figure 3.16
   }

   // consumers calls this method
   public Object remove() {
      // Figure 3.17
   }
}
```



```
public void insert(Object item) {
   while (count == BUFFER_SIZE)
       ; // do nothing -- no free buffers

   // add an item to the buffer
   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
}
```

```
public Object remove() {
   Object item;

   while (count == 0)
       ; // do nothing -- nothing to consume

   // remove an item from the buffer
   --count;
   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;

   return item;
}
```

37

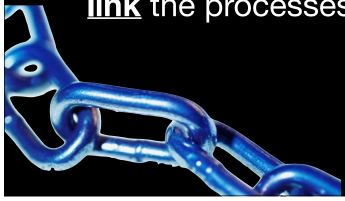
Message Passing

- Processes do not share any memory or variables.
  - Producer sends messages
  - Consumer receives messages

38

Message Passing

Before sending messages we need to **link** the processes together



39

Direct Address/Link

Processes can explicitly name each other

- send(Consumer, *message*)
- receive(Producer, *message*)

Two-party link, bidirectional(?), automatic

Indirect Address/Link

Processes use an OS *mailbox*

- send(mbox, message)
- receive(mbox, message)

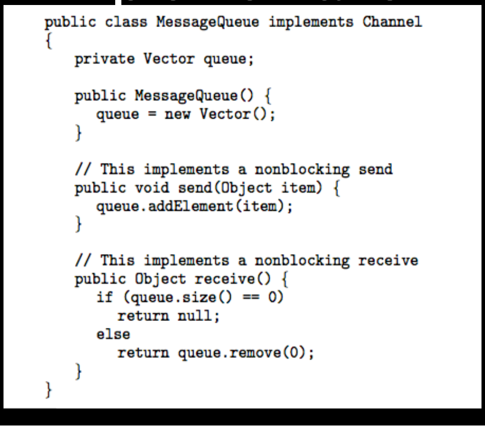
Mailboxes are unique across the system

Can they have multiple receivers? senders?

Mailboxes

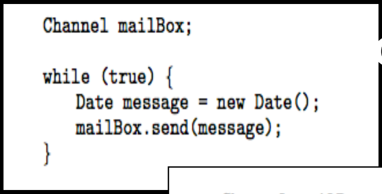
- Sharing: who owns the mailbox?
- Blocking vs. Non-blocking calls
- Mailbox size: 0, bounded, unbounded
- Implementations...

Implementation

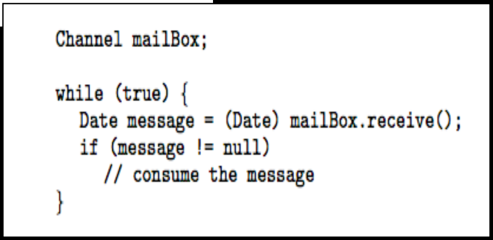


```
public class MessageQueue implements Channel
{
   private Vector queue;

   public MessageQueue() {
      queue = new Vector();
   }

   // This implements a nonblocking send
   public void send(Object item) {
      queue.addElement(item);
   }

   // This implements a nonblocking receive
   public Object receive() {
      if (queue.size() == 0)
         return null;
      else
         return queue.remove(0);
   }
}
```

cer

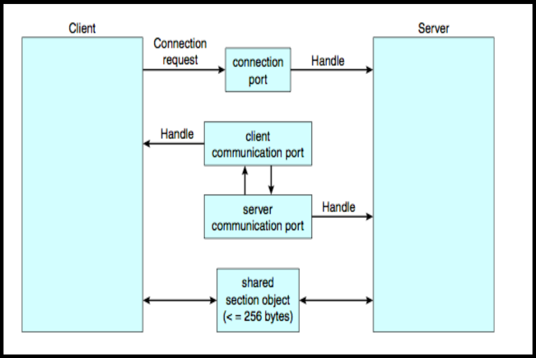


```
Channel mailBox;

while (true) {
   Date message = new Date();
   mailBox.send(message);
}
```



```
Channel mailBox;

while (true) {
   Date message = (Date) mailBox.receive();
   if (message != null)
      // consume the message
}
```

Windows XP


Client
Server
Connection request
connection port
Handle
Handle
client communication port
server communication port
Handle
shared section object (< = 256 bytes)

Client-Server

- Sockets
- Remote Procedure Calls
- Remote Method Invocation

Sockets (IP)

46

A *socket* defines an endpoint for communication.

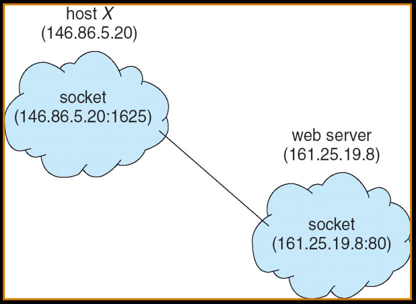
simple version; IP address and port number

130.111.125.26:80

Communication happens over a <u>socket pair</u>.

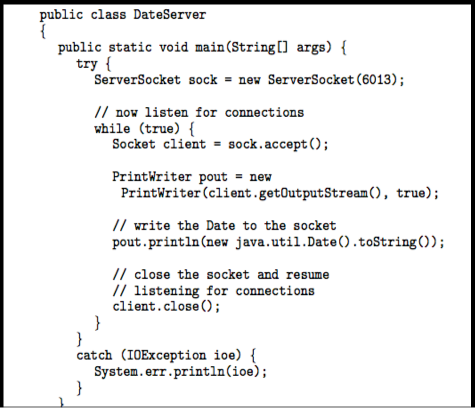
Cloud Sockets

47



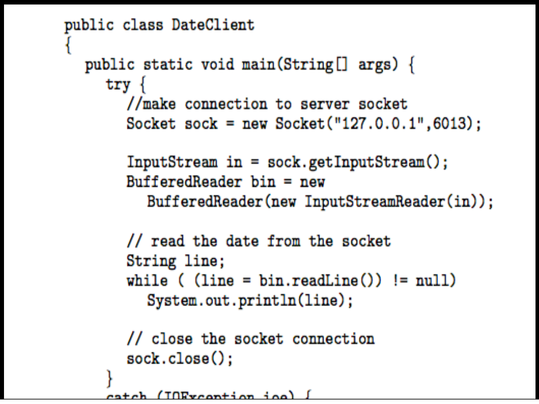
Socket-based Server

48



```
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);
    }
```

Socket-based Client



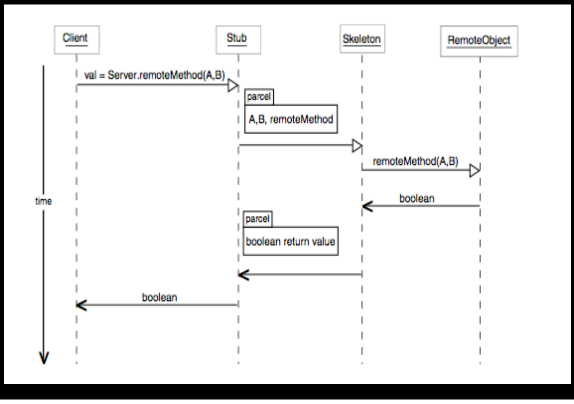
```
public class DateClient
{
  public static void main(String[] args) {
    try {
      //make connection to server socket
      Socket sock = new Socket("127.0.0.1",6013);

      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();
    }
```

Remote Procedure Calls (RPC)

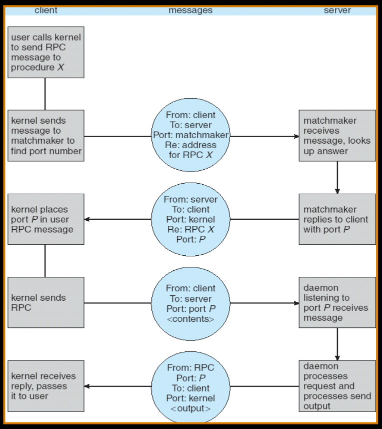
Idea: Make calls on remote process look like local calls.

- **stubs** on client proxies to server
- **skeleton** on server dispatches to procedures

The (required) UML



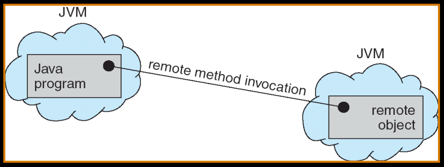
52


client
messages
server
user calls kernel to send RPC message to procedure X
kernel sends message to matchmaker to find port number
From: client To: server Port: matchmaker Re: address for RPC X
matchmaker receives message, looks up answer
kernel places port P in user RPC message
From: server To: client Port: kernel Re: RPC X Port: P
matchmaker replies to client with port P
kernel sends RPC
From: client To: server Port: port P <contents>
daemon listening to port P receives message
kernel receives reply, passes it to user
From: RPC Port: P To: client Port: kernel <output>
daemon processes request and processes send output

53

Remote Method Invocation (RMI)

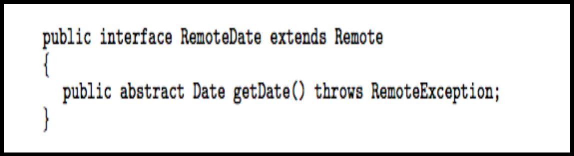
Idea: Manipulate objects that are on a remote process.



Object-oriented version of RPC

54

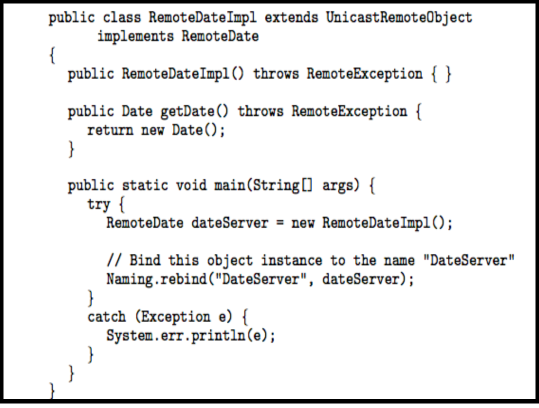
Interface



```
public interface RemoteDate extends Remote
{
   public abstract Date getDate() throws RemoteException;
}
```

Server

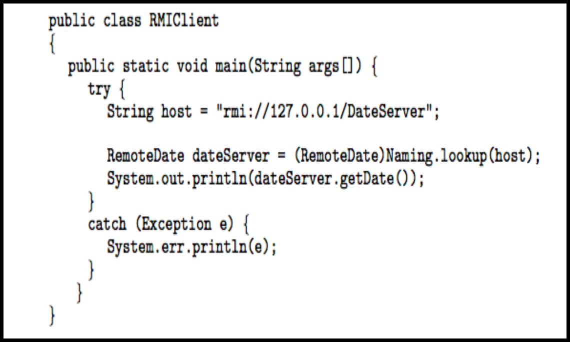
55



```
public class RemoteDateImpl extends UnicastRemoteObject
      implements RemoteDate
{
   public RemoteDateImpl() throws RemoteException { }

   public Date getDate() throws RemoteException {
      return new Date();
   }

   public static void main(String[] args) {
      try {
         RemoteDate dateServer = new RemoteDateImpl();

         // Bind this object instance to the name "DateServer"
         Naming.rebind("DateServer", dateServer);
      }
      catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

Client

56



```
public class RMIClient
{
   public static void main(String args[]) {
      try {
         String host = "rmi://127.0.0.1/DateServer";

         RemoteDate dateServer = (RemoteDate)Naming.lookup(host);
         System.out.println(dateServer.getDate());
      }
      catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

Communications

57

- Definitions
- Shared Memory
- Message Passing
- Client-Server and Examples

Processes

- ✓ What is a ***process***?
- ✓ ***Scheduling*** processes
- ✓ ***Cooperating*** and ***Communication***

End

Processes