C Program Forking Separate Process

- The C program illustrates how to create a new process UNIX.
- After **`fork()`** there are two different processes running copies of the same program. The only difference is that the value of pid for the child process is zero, while that for the parent is an integer value greater than zero
- The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files.
- The child process then overlays its address space with the UNIX command “ls” (used to get a directory listing) using the execlp() (a version of the exec() system call).
- The parent waits for the child process to complete with the wait() system call.
- When the child process completes, the parent process resumes from the call to wait(), where it completes using the exit() system call.

Process Termination

- A process terminates when it finishes executing its final statement and it asks the operating system to delete it by using the `exit()` system call.
  - At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call.
  - All the resources of the process are deallocated by the operating system.
- A parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
  - 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 (Cont.)

- Some operating systems do not allow a child process 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);
```

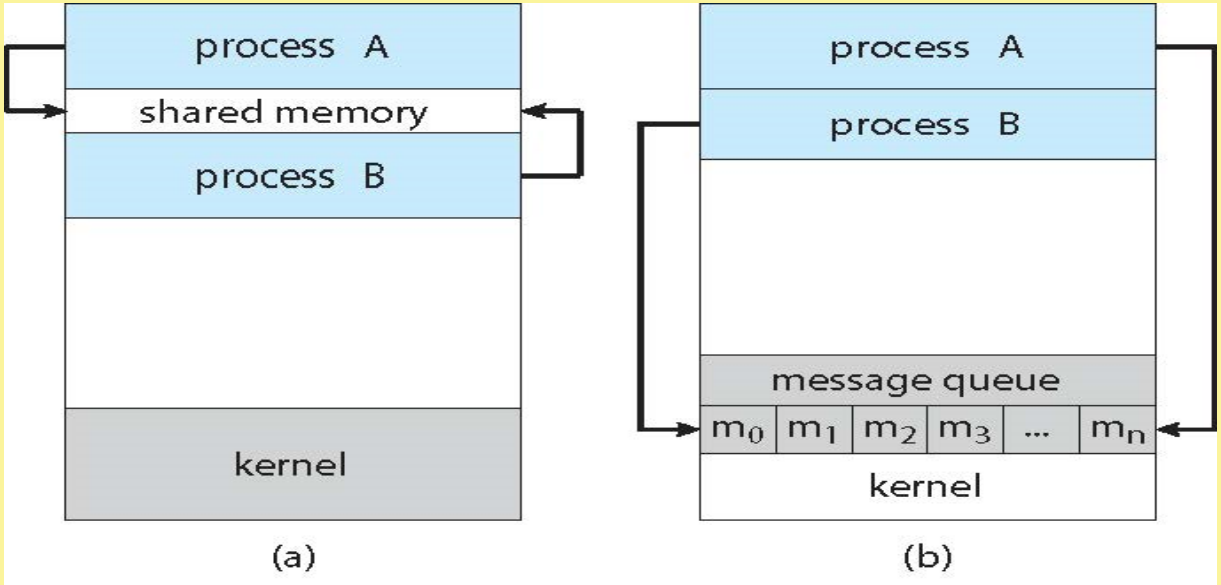
- If no parent waiting (did not invoke `wait()`) process is **zombie**
- If parent terminated without invoking `wait`, process is **orphan**


Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
  - Cooperating processes can affect or be affected by other processes, including sharing data
  - Independent processes cannot affect other processes
- Reasons for having cooperating processes:
  - Information sharing
  - Computation speedup (multiple processes running in parallel)
  - Modularity
  - Convenience
- Cooperating processes need **interposes communication** (**IPC**)

Communications Models

- Two models of IPC
  - **Shared memory**
  - **Message passing**



Shared Memory Systems

- 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

- Cooperating processes that access shared data need to synchronize their actions to ensure data consistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem – The producer-Consumer problem
  - Producer process produces information that is consumed by a Consumer process.
  - The information is passed from the Producer to the Consumer via a buffer.
  - Two types of buffers can be used:
    - **unbounded-buffer** places no practical limit on the size of the buffer
    - **bounded-buffer** assumes that a fixed buffer size


Bounded-Buffer Solution

- Shared data

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

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

- Solution presented in the next two slides is correct, but only 9 out of 10 buffer elements can be used

Bounded-Buffer – 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;
}
```

Bounded Buffer – Consumer

```
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 */

}
```

Message Passing Systems

- Mechanism for processes to communicate and to synchronize their actions
  - 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
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?


Implementation of Communication Link

- Physical:
  - Shared memory
  - Hardware bus
  - Network
- Logical:
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering

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
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional


Indirect Communication

- 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
- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - delete a mailbox
- Primitives are defined as:
  - **`send`**(*A, message*) – send a message to mailbox A
  - **`receive`**(*A, message*) – receive a message from mailbox A

Indirect Communication (Cont.)

- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication Issues

- 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.


Blocking and Non-blocking schemes

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message

n Different combinations possible

l If both send and receive are blocking, we have a **rendezvous**

Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.
- Such queues can be implemented in three ways:
  1. Zero capacity – no messages are queued on a link. Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages Sender must wait if link full
  3. Unbounded capacity – infinite length Sender never waits


Producer-Consumer with Rendezvous

- Producer-consumer synchronization becomes trivial with rendezvous (blocking send and receive)

```
message next_produced;
while (true) {
        /* produce an item in next produced */
    send(next_produced);
}
```

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

    /* consume the item in next consumed */
}
```


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.
- Named pipes – can be accessed without a parent-child relationship.

Ordinary Pipes

- Ordinary Pipes allow two process to 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 unidirectional, allowing only one-way communication.
- If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction.
- Require parent-child relationship between communicating processes

Ordinary Pipes

On UNIX systems, ordinary pipes are constructed using the function

pipe (int fd[])

This function creates a pipe that is accessed through the

int fd[]

file descriptors:

fd[0] is the read-end of the pipe

fd[1]} is the write-end of the pipe

UNIX treats a pipe as a special type of file. Thus, pipes can be accessed using ordinary read() and write() system calls.

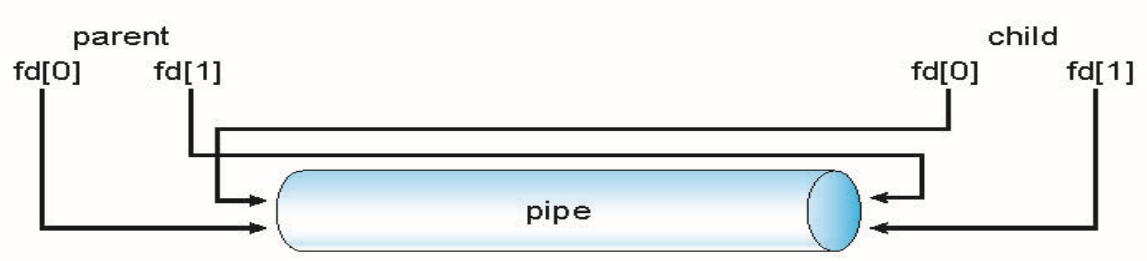
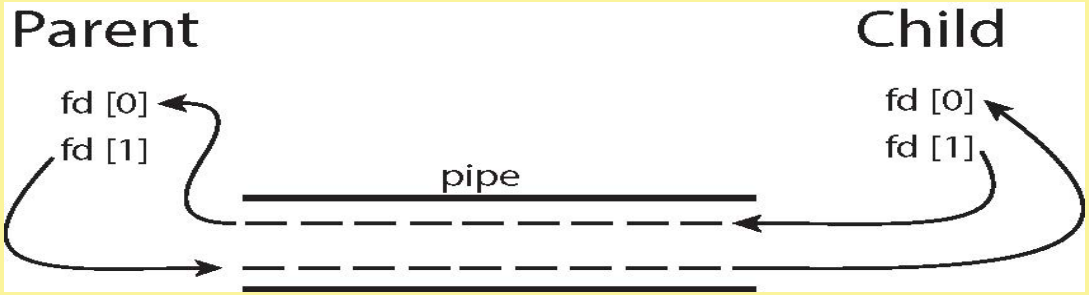


Figure Pipe



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

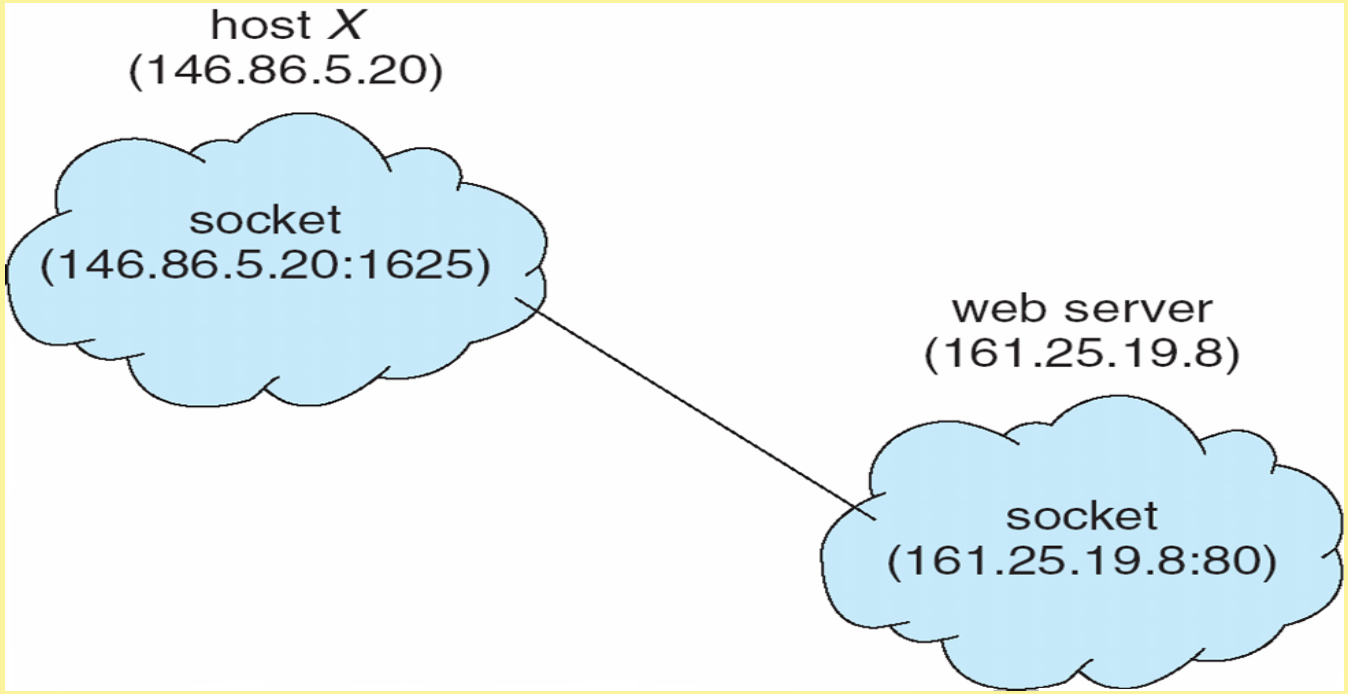
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – 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**) is used to refer to system on which process is running. That is, when a computer refers to address 127.0.0.1, it is referring to itself.

Socket Communication



Remote Procedure Calls

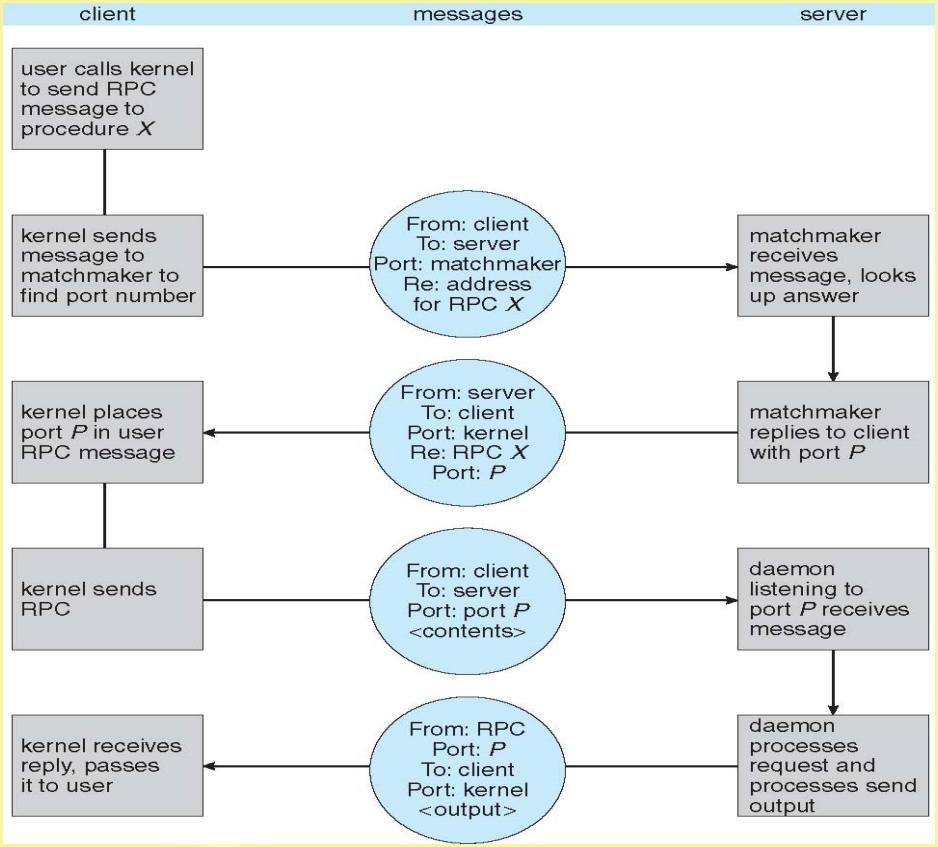
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters (marshalling involves packaging the parameters into a form that can be transmitted over a network).
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)

Remote Procedure Calls (Cont.)

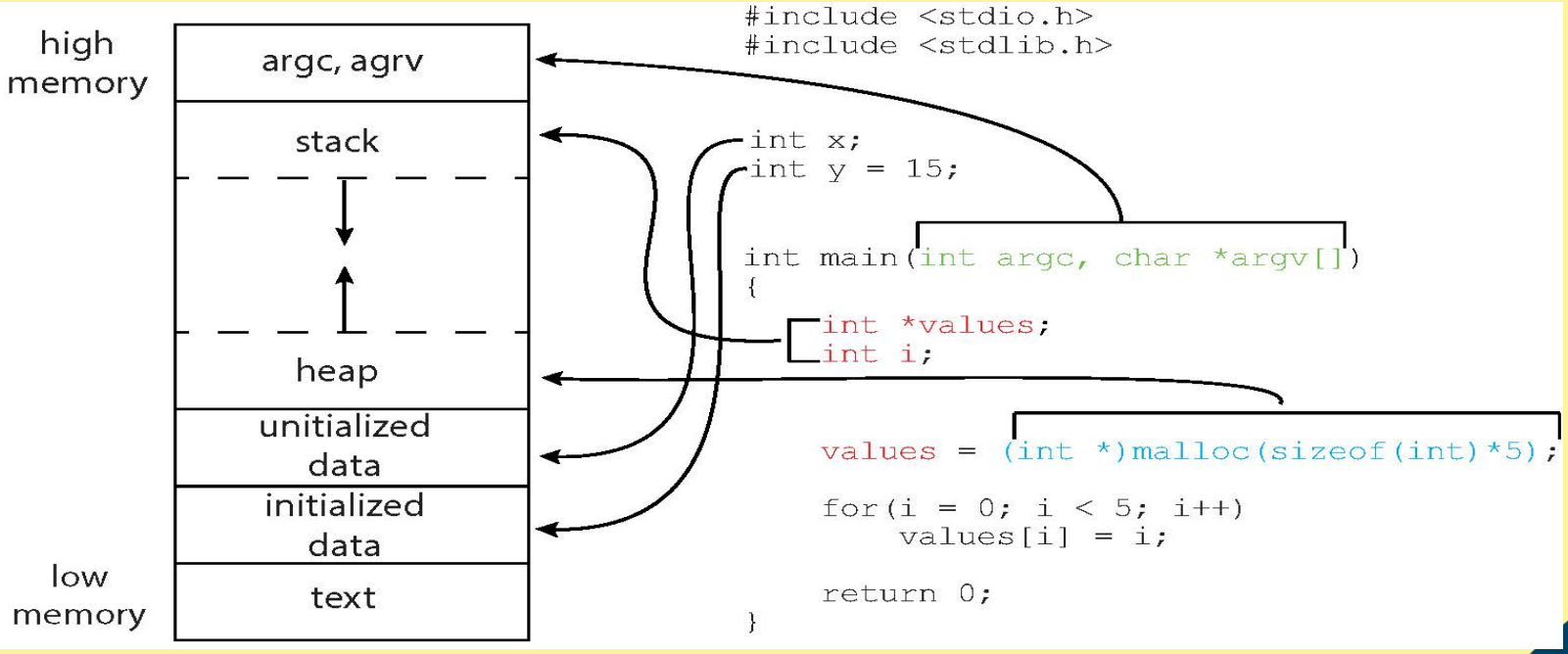
- Data representation handled via **External Data Representation** (**XDL**) format to account for different architectures.
- Must be dealt with concerns differences in data representation on the client and server machines.
- Consider the representation of 32-bit integers.
  - **Big-endian.** Store the most significant byte first
  - **Little-endian.** Store the least significant byte first.

  Big-endian is the most common format in data networking . It is also referred to as **network byte order**.
- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server

Execution of RPC



Memory Layout



Conclusion

Conclusion:

- A process is a program in execution
- Each process has a PCB
- A process may be in one of the states – new, ready, running, waiting, terminated
- Ready queue contains all the ready processes waiting
- A parent process may create children processes
- Processes need synchronization and communication

NPTEL ONLINE CERTIFICATION COURSES

Thank you

Operating System Fundamentals

Santanu Chattopadhyay
Electronics and Electrical Communication Engg.

Threads

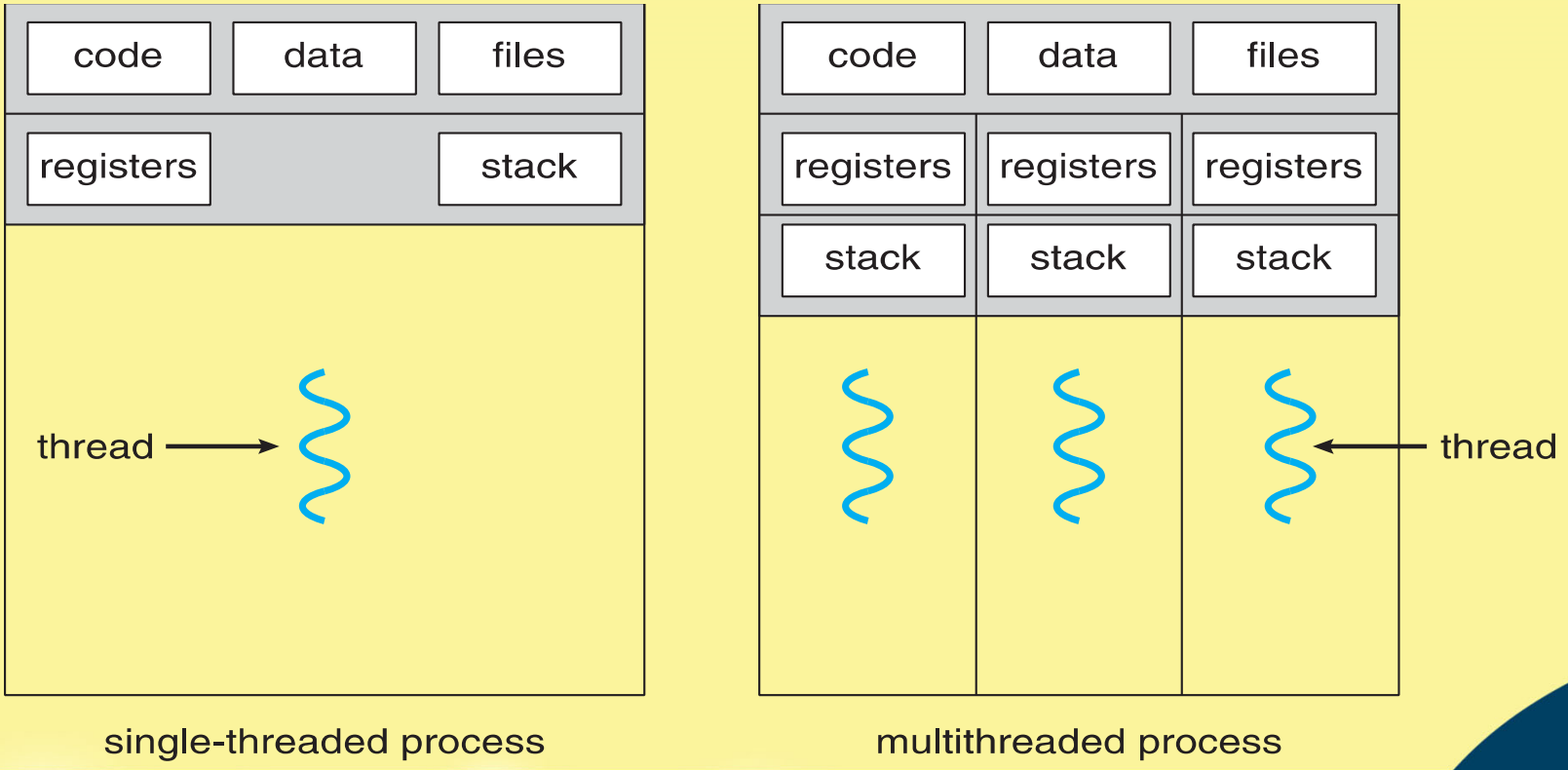
Concepts Covered:

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues

Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

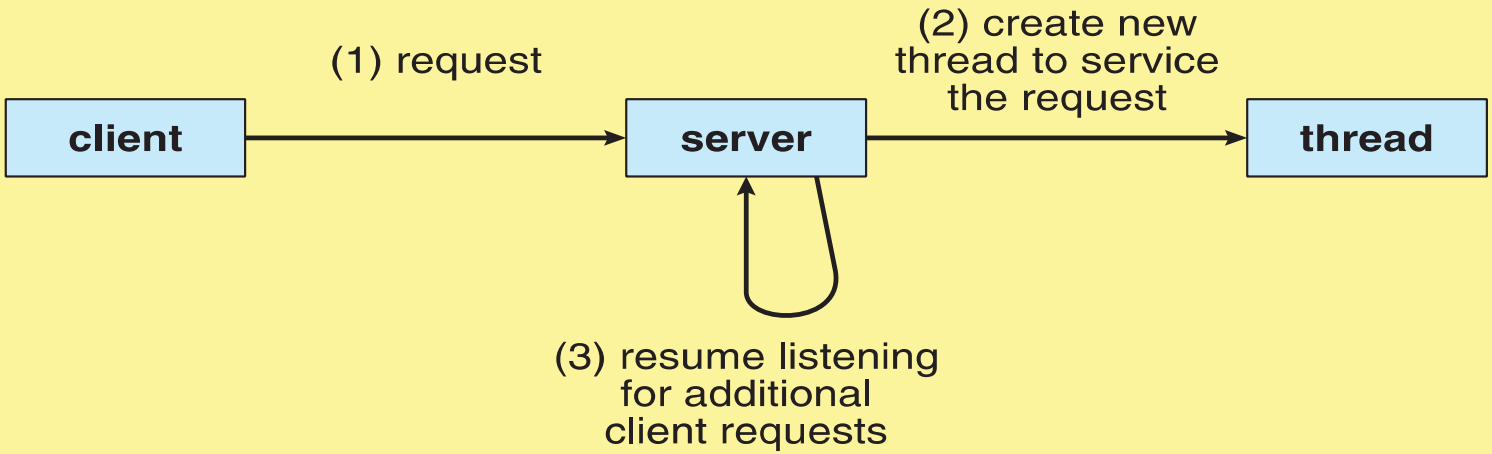

Single and Multithreaded Processes



single-threaded process

multithreaded process

Multithreaded Server Architecture



Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

Multicore Programming

- **Multi-CPU systems**. Multiple CPUs are placed in the computer to provide more computing performance.
- **Multicore systems**. Multiple computing cores are placed on a single processing chip where each core appears as a separate CPU to the operating system
- Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency.
- Consider an application with four threads.
  - On a system with multiple cores, concurrency means that some threads can run in parallel, because the system can assign a separate thread to each core

Multicore Programming (Cont.)

- There is a fine but clear distinction between concurrency and parallelism.
- A concurrent system supports more than one task by allowing all the tasks to make progress.
- In contrast, a system is parallel if it can perform more than one task simultaneously.
- Thus, it is possible to have concurrency without parallelism

Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As number of threads grows, so does architectural support for threading
  - CPUs have cores as well as ***hardware threads***
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Multicore Programming

- **Multicore** or **multiprocessor** systems are placing pressure on programmers. Challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency