Processes

Distributed Systems [3]

殷亚凤

Email: yafeng@nju.edu.cn

Homepage: http://cs.nju.edu.cn/yafeng/

Room 301, Building of Computer Science and Technology

Review

- **Architecture Styles**: Layered, object-oriented, event-based, shared data spaces-based
- **System Architecture :** Centralized, Decentralized, Hybrid
- **Middleware**
- **Self-managing Distributed Systems**

This lesson

- **Process**
- **Thread**
- **Client – Server**
- **Code Migration**

What is a Process?

- Process: An **execution stream** in the context of a **process state** (a program in execution)
- **Execution stream**
  - Stream of executing instructions
  - Running piece of code
  - Sequential sequence of instructions
  - “thread of control”
- **Process state**
  - Everything that the running code can affect or be affected by

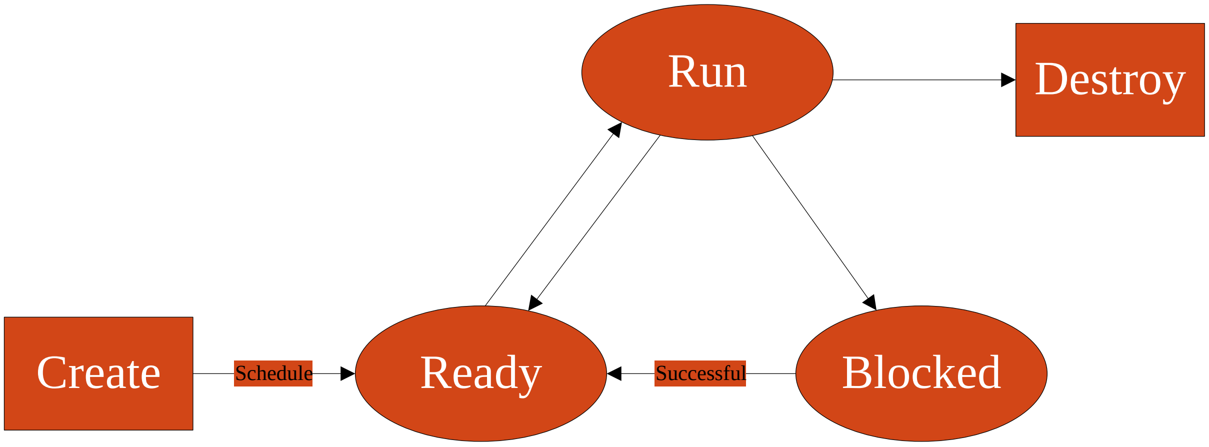
Processes vs. Programs

- A process is different than a program
  - Program: Static code and static data
  - Process: Dynamic instance of code and data

- No one-to-one mapping between programs and processes
  - **One process** can execute **multiple programs**
  - **One program** can invoke **multiple processes**

Processes

- Each process is in one of three modes:
  - **Running**: On the **CPU** (only one on a uniprocessor)
  - **Ready**: Waiting for the **CPU**
  - **Blocked** (or asleep): Waiting for I/O or synchronization to complete



Low Performance

- **Resource management**
  - When creating a new process, assign address space, copy data
- **Scheduling**
  - Context switch
    - Process Context: CPU context and storage context
- **Cooperation**
  - IPC, interprocess communication
  - Shared memory

Introduction to Threads

- **Thread**: A minimal software processor in whose context a series of instructions can be executed.

- **Saving** a thread context implies stopping the current execution and saving **all the data needed to** continue the execution at a later stage.

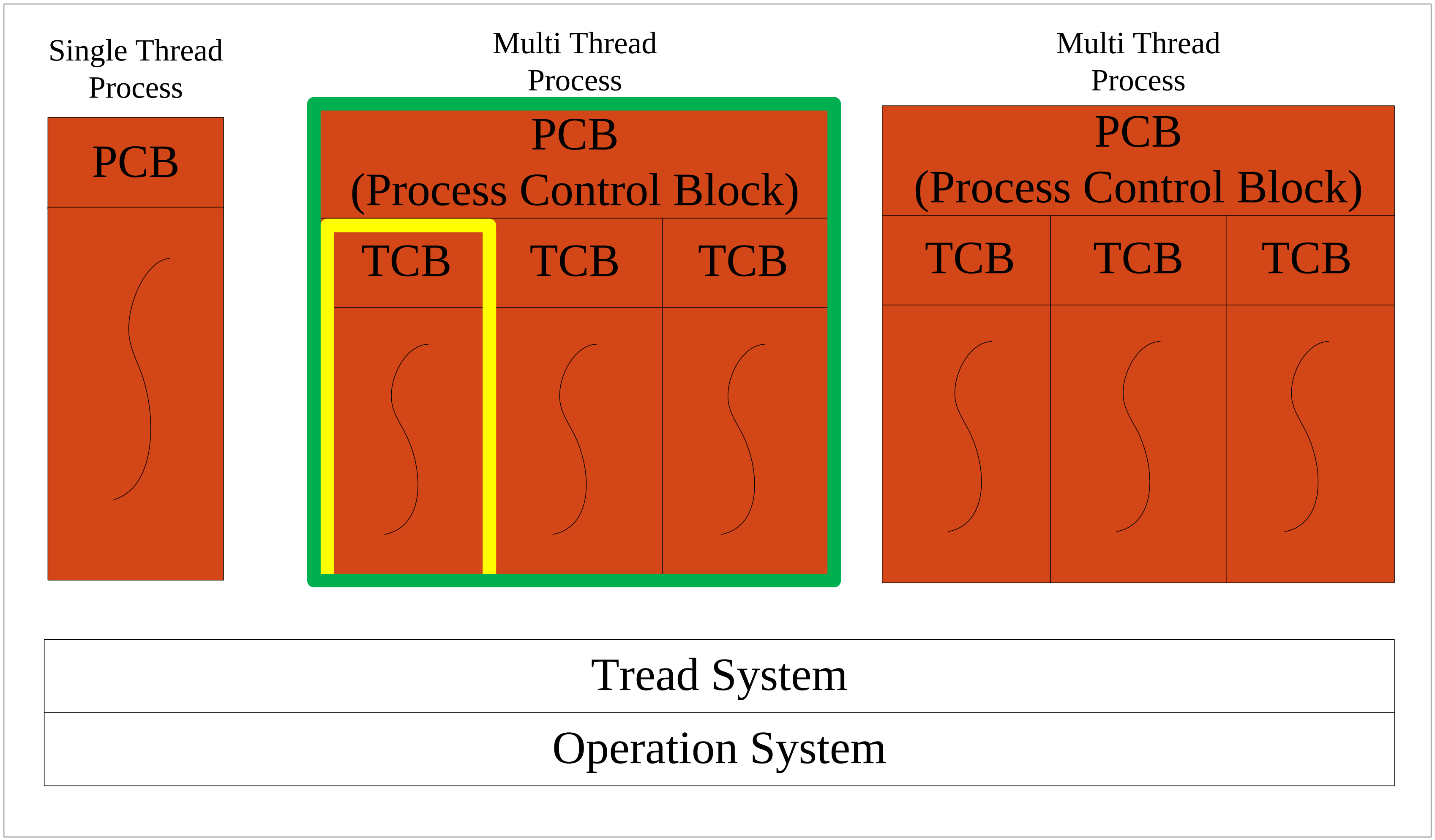
Process and thread : Context Switching

- **Processor context**: The minimal collection of values stored in the **registers of a processor** used for the execution of a series of **instructions** (e.g., stack pointer, addressing registers, program counter).
- **Thread context**: The minimal collection of values stored in **registers and memory**, used for the execution of a series of **instructions** (i.e., processor context, state).
- **Process context**: The minimal collection of values stored in **registers and memory**, used for the execution of a **thread** (i.e., thread context, but now also at least MMU register values).

Context Switching

- **Threads** share the same address space. Thread context switching can be done entirely independent of the operating system.
- **Process** switching is generally more expensive as it involves getting the OS in the loop, i.e., to the kernel.
- **Creating and destroying** threads is much cheaper than doing so for processes.

Thread and Process



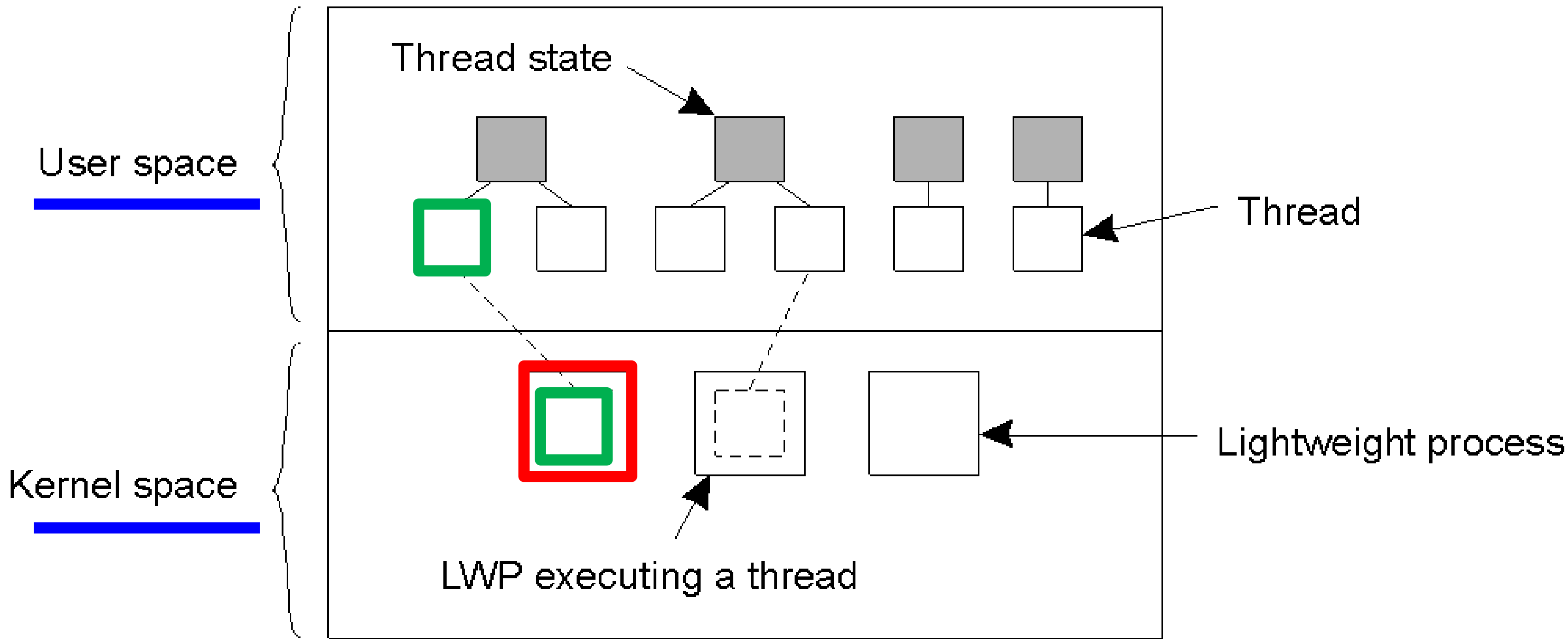
User-Level Thread

- All the threads are created in user processes' address spaces.
- **Advantage**
  - All operations can be completely handled within a single process $\Rightarrow$ implementations can be extremely efficient.
- **Disadvantage**
  - Difficult to get the support from OS, block
    - All services provided by the kernel are done on behalf of the process in which a thread resides $\Rightarrow$ if the kernel decides to block a thread, the entire process will be blocked.

Threads and Operating Systems

- Have the **kernel** contain the implementation of a thread package. This means that all operations return as system calls:
  - Operations that block a thread are no longer a problem: the **kernel schedules another available thread** within the same process.
  - Handling **external events** is simple: the kernel (which catches all events) schedules the thread associated with the event.
  - The problem is (or used to be) **the loss of efficiency** due to the fact that each thread operation requires a trap to the kernel.

Thread Implementation



Light weight processes, LWP

Threads and Distributed Systems

- Hiding network latencies:
  - Web browser scans an incoming HTML page, and finds that more files need to be fetched.
  - Each file is fetched by a separate thread, each doing a (blocking) HTTP request.
  - As files come in, the browser displays them.
- Multiple request-response calls to other machines(RPC)
  - A client does several calls at the same time, each one by a different thread.
  - It then waits until all results have been returned.
  - Note: if calls are to different servers, we may have a linear speed-up.

Multithreaded Clients

- Example, web browser

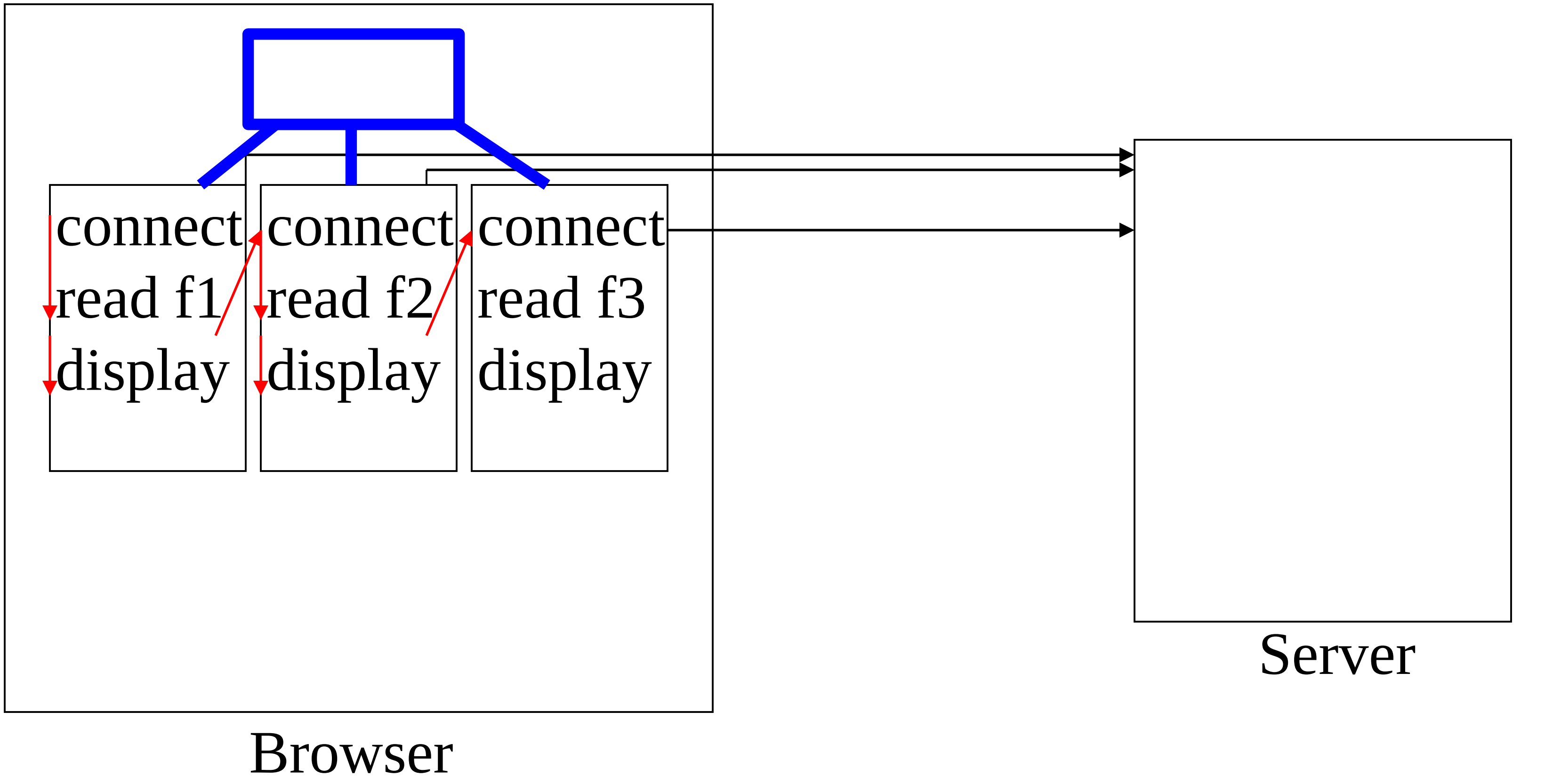
  A web document $\supset$ plain text, a collection of images.

  To fetch a HTML file:

  connect server, read a file1, display

  connect server, read a file2, display

  … …

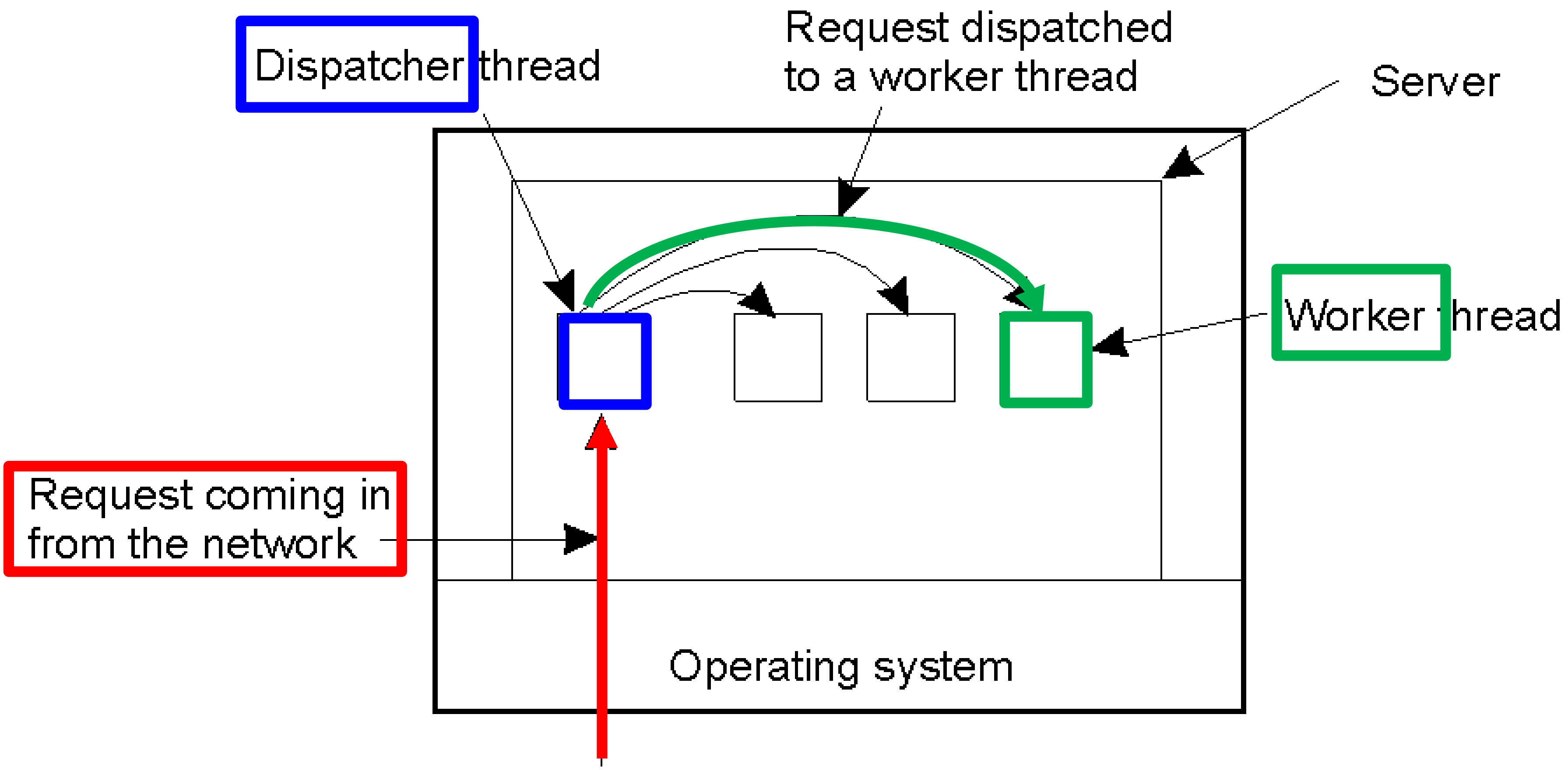

connect
read f1
display
connect
read f2
display
connect
read f3
display
Browser
Server

Threads and Distributed Systems

- **Improve performance**:
  - Starting a thread is much cheaper than starting a new process.
  - Having a single-threaded server prohibits simple scale-up to a multiprocessor system.
  - As with clients: hide network latency by reacting to next request while previous one is being replied.
- **Better structure**
  - Most servers have high I/O demands. Using simple, well-understood blocking calls **simplifies the overall structure**.
  - Multithreaded programs tend to be smaller and easier to understand due to **simplified flow of control**.

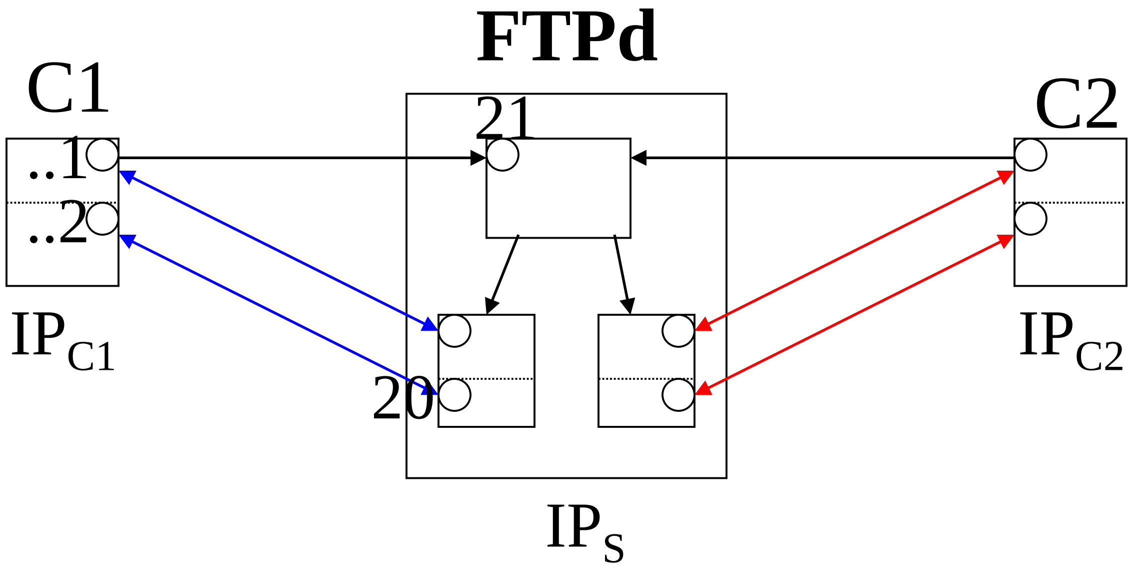
Multithreaded Servers

- A multithreaded server organized in a dispatcher/worker model.



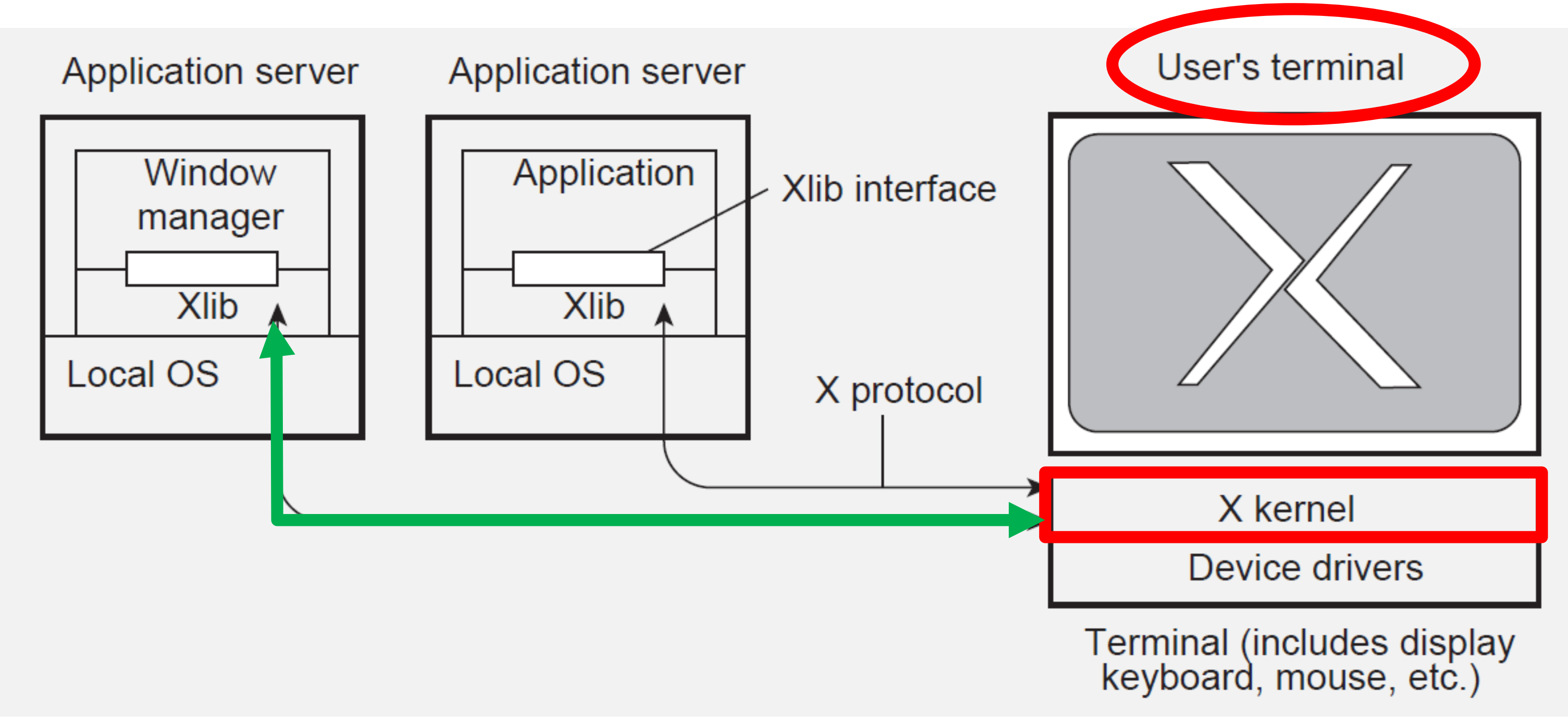
Multithreaded servers

- Example, file server



Clients: User Interfaces

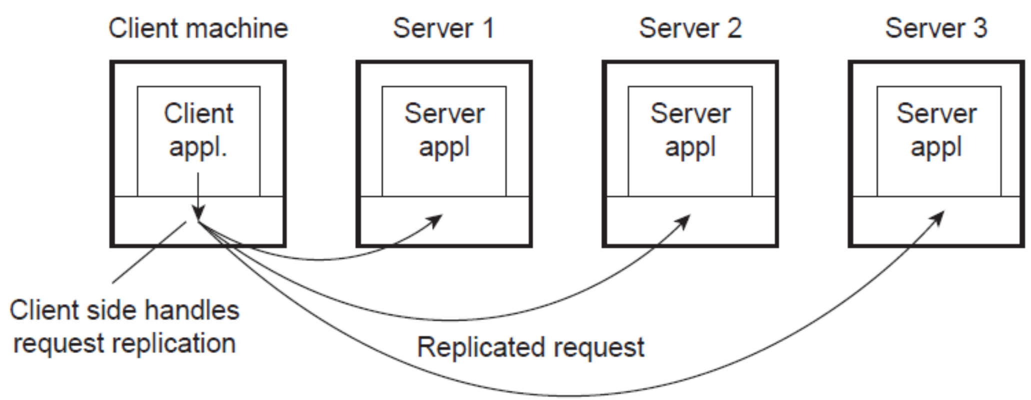
- A major part of client-side software is focused on (graphical) user interfaces.



X Window

Client-Side Software

- Generally tailored for **distribution transparency**
  - access transparency: client-side stubs for RPCs
  - location/migration transparency: let client-side software keep track of actual location
  - failure transparency: can often be placed only at client (we're trying to mask server and communication failures).
  - replication transparency: multiple invocations handled by client stub



Servers: General organization

- Basic model:
  - A server is a process that **waits for incoming service** requests at a specific transport address. In practice, there is a one-to-one mapping between **a port and a service**.

ftp-data	20	File Transfer [Default Data]
ftp	21	File Transfer [Control]
telnet	23	Telnet
	24	any private mail system
smtp	25	Simple Mail Transfer
login	49	Login Host Protocol
sunrpc	111	SUN RPC (portmapper)
courier	530	Xerox RPC

Servers: General organization

- Type of servers:
  - Superservers: Servers that listen to **several ports**, i.e., provide several independent services. In practice, when a service request comes in, they start a **subprocess** to handle the request (UNIX inetd)
  - Iterative vs. concurrent servers: Iterative servers can handle only **one client at a time**, in contrast to **concurrent servers**

Out-of-band communication

- Is it possible to **interrupt** a server once it has accepted (or is in the process of accepting) a service request?
- Use a separate port for urgent data:
  - Server has a separate thread/process for urgent messages
  - Urgent message comes in $\Rightarrow$ associated request is put on hold
  - Note: we require OS supports priority-based scheduling
- Use out-of-band communication facilities of the transport layer:
  - Example: TCP allows for urgent messages in same connection
  - Urgent messages can be caught using OS signaling techniques

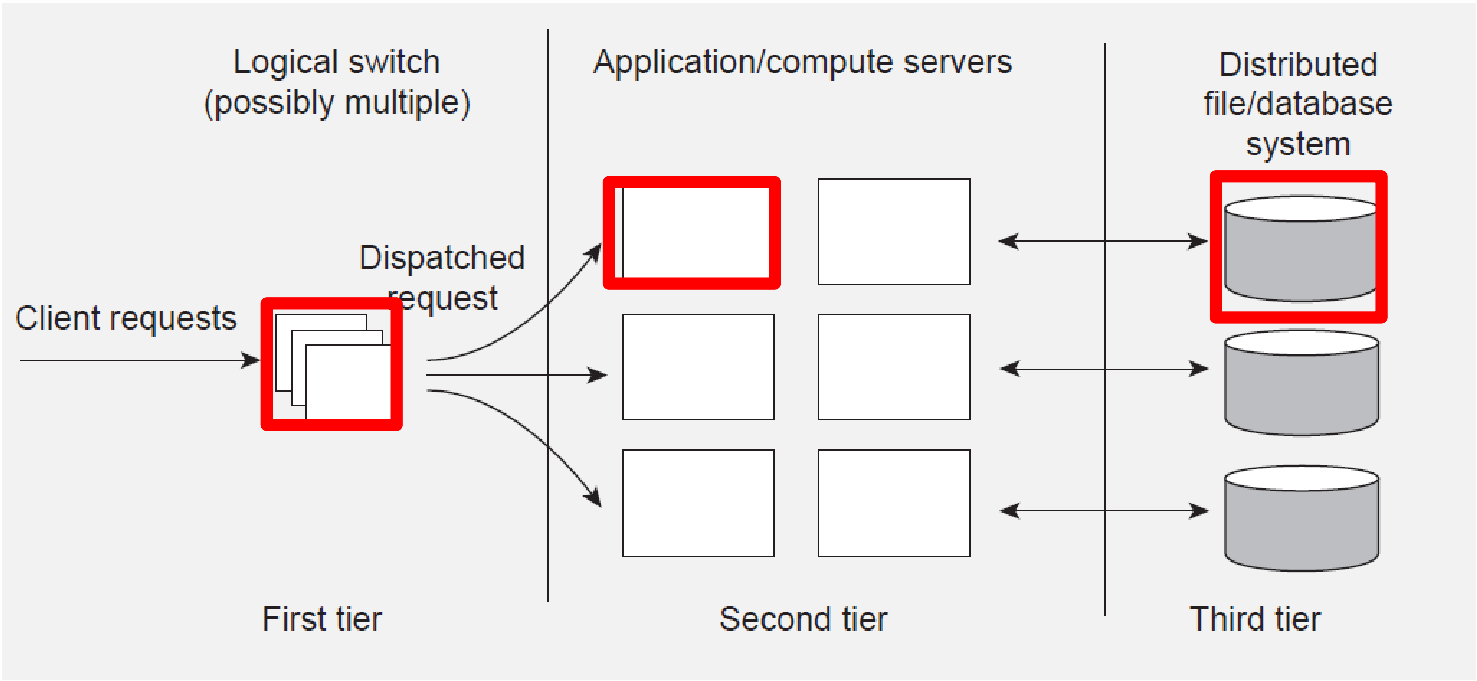
Stateless

- Never keep accurate information about the status of a client after having handled a request:
  - Don't record whether a file has been opened (simply close it again after access)
  - Don't promise to invalidate a client's cache
  - Don't keep track of your clients
- Consequences
  - Clients and servers are completely independent
  - State inconsistencies due to client or server crashes are reduced
  - Possible loss of performance because, e.g., a server cannot anticipate client behavior (think of prefetching file blocks)

Stateful

- Keeps track of the status of its clients:
  - Record that a file has been opened, so that prefetching can be done
  - Knows which data a client has cached, and allows clients to keep local copies of shared data
- The performance of stateful servers can be extremely high, provided clients are allowed to keep local copies. As it turns out, reliability is not a major problem.

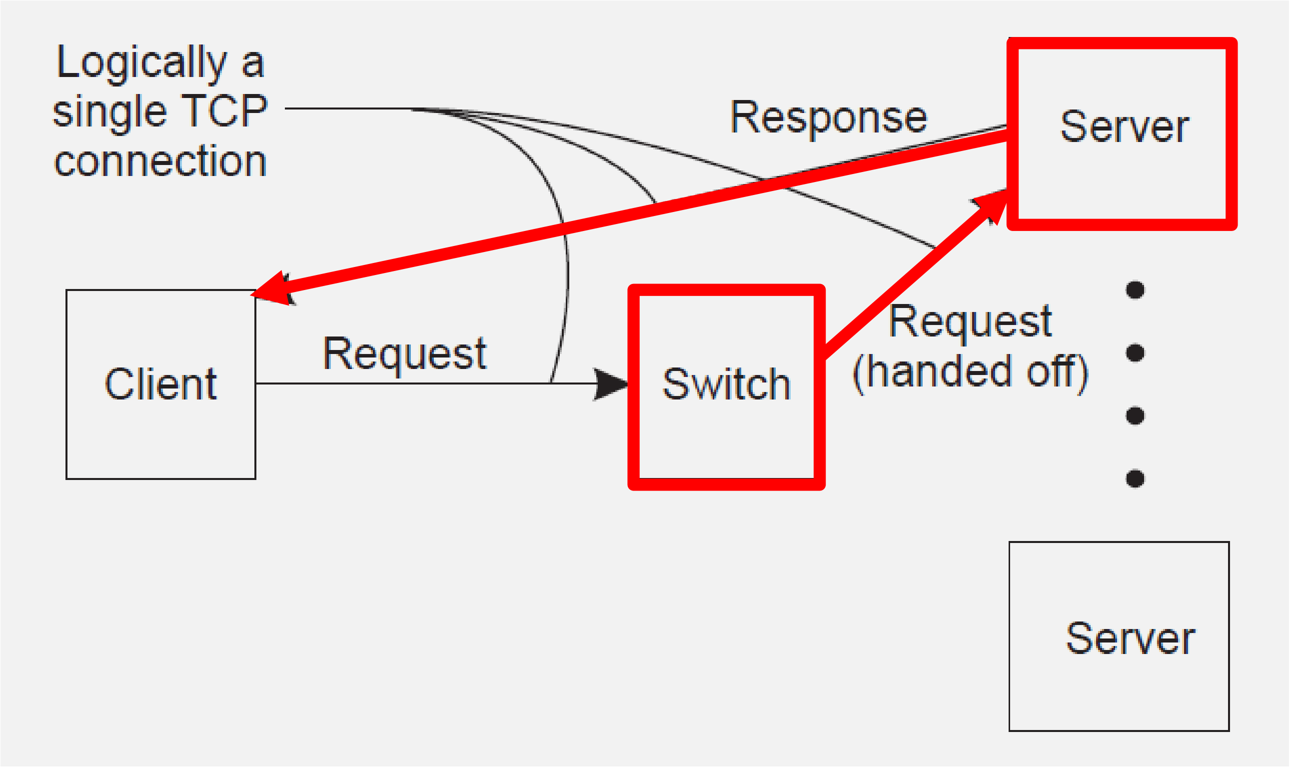
Server clusters: three different tiers



- Crucial element
  - The first tier is generally responsible for passing requests to an appropriate server.

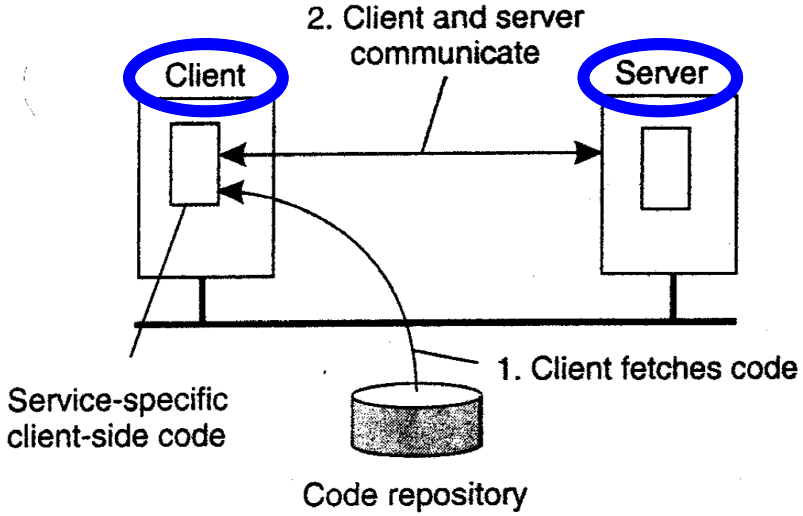
Request Handling

- The first tier handle all communication from/to the.
- The one popular one is TCP-handoff



Code Migration

- Approaches to code migration



- Migration and local resources
- Migration in heterogeneous systems

Migration model

- **Code segment**: contains the executing instructions
- **Resource segment**: contains the pointers pointing to the external resources
- **Execution segment**: contains executing states

Weak and strong mobility

Weak

- Move only **code and data segment** (and reboot execution):
  - Relatively simple, especially if code is portable
  - Distinguish code shipping (push) from code fetching (pull)

Strong

- Move component, including **execution state**
  - Migration: move entire object from one machine to the other
  - Cloning: start a clone, and set it in the same execution state

Managing local resources

- Object-to-resource binding
  - By identifier: the object requires a specific instance of a resource (e.g. a specific database)
  - By value: the object requires the value of a resource (e.g. the set of cache entries)
  - By type: the object requires that only a type of resource is available (e.g. a color monitor)

Managing local resources

- An object uses local resources that may or may not be available at the target site
- Resource types
  - Fixed: the resource cannot be migrated, such as local hardware
  - Fastened: the resource can, in principle, be migrated but only at high cost
  - Unattached: the resource can easily be moved along with the object (e.g. a cache)

Managing local resources (2/2)

	Unattached	Fastened	Fixed
ID	MV (or GR)	GR (or MV)	GR
Value	CP (or MV, GR)	GR (or CP)	GR
Type	RB (or MV, GR)	RB (or GR, CP)	RB (or GR)

GR = Establish global systemwide reference

MV = Move the resource

CP = Copy the value of the resource

RB = Re-bind to a locally available resource

Migration in heterogeneous systems

- Main problem
  - The target machine may not be suitable to execute the migrated code
  - The definition of process/thread/processor context is highly dependent on local hardware, operating system and runtime system
- Make use of an abstract machine that is implemented on different platforms
  - Interpreted languages, effectively having their own VM
  - Virtual VM (as discussed previously)