CS5223
Distributed Systems

Lecture 3: OS Support for Distributed Systems -- Processes and Threads

Instructor: YU Haifeng

Today’s Roadmap

- Chapter 3 of textbook

- Review of processes in operating systems
- Threads and light-weight process (LWP)

- Multi-threaded servers
- Designing servers

- Code migration

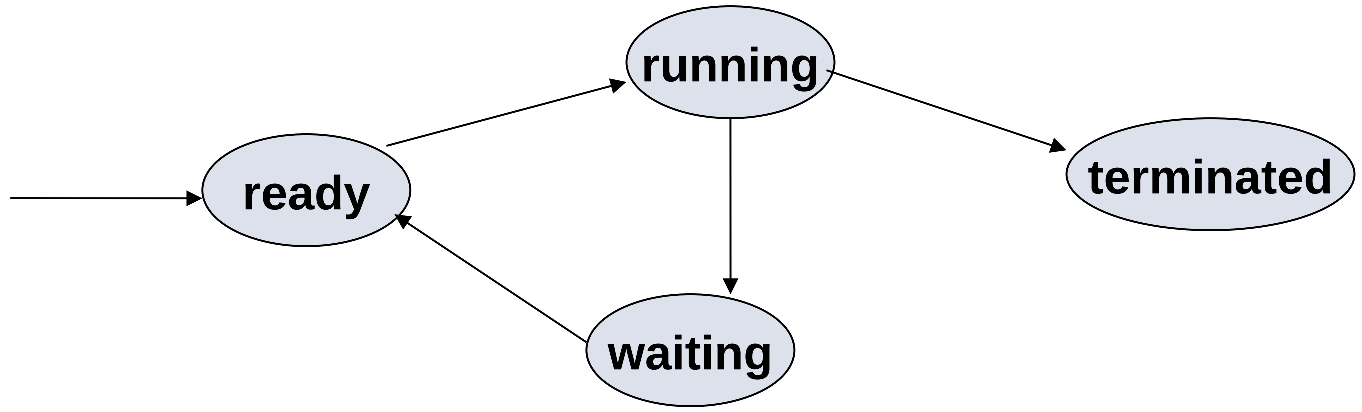
A Brief Review on Processes

- Process: a program being executed in its own virtual memory space
  - Historical goal: Allow multiple users to share the same CPU (computers were expensive!)
  - Emphasis: Isolation and protection

- Process control block (PCB): contains information associated with a process
  - program counter (PC)
  - stack pointer (SP)
  - CPU registers
  - open files
  - memory allocation information (code, stack and data)

A Brief Review on Process State

- Running: **instructions are being executed**
- Waiting: **process is waiting for some event to occur**
- Terminated: **process has finished execution**
- Ready: **process is waiting to be assigned to a processor**



Processes Are Expensive: Creation Cost

- Need to create independent address space
  - New mapping between physical memory pages to virtual memory pages
- Need to allocate and maintain open file information for the process
- A typical OS can thus only support small number of processes (e.g., a few hundreds)
- Historical perspective: We did not care because processes were long running (e.g., long running simulation)

Processes are Expensive: Context Switching

- When a CPU switches from process P1 to P2, it
  - saves the state of P1, loads the state of P2, starts execution of P2
- Switching requires
  - Trapping into the OS (why?)
  - Saving and changing memory mapping
  - Saving and changing open file table
  - Saving and changing CPU context (e.g., register values, program counters, stack pointer)

- Historical perspective: We did not care because processes do not tend to switch so often

Processes are Expensive: Inter-Process Communication

- Processes often need to communicate/collaborate with other processes
  - Word may hand over the file to a spooler (daemon) process which sends the file to the printer
  - In a simple distributed game, we may have two processes (one for each player), and each process may need to periodically update the other process
- Inter-process communication is usually expensive
  - Because we wanted to isolate / protect processes from each other
  - A process takes extra precaution when communicating with other processes
  - Usually involve trapping to the OS – directly updating some common data is not usually an option

Why We Care

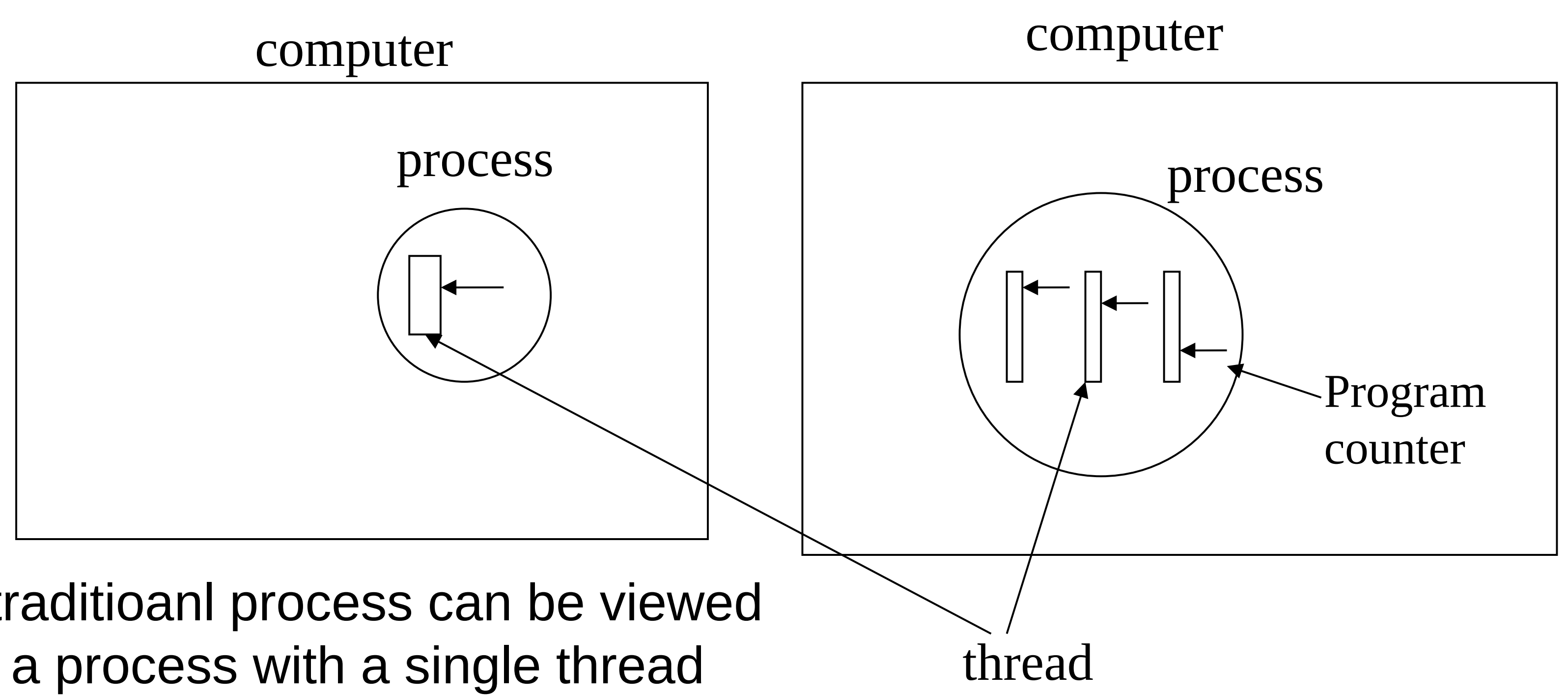
- Interactive applications (such as graphic user interface)
  - Each window may corresponds to a process
  - Each mouse click may trigger a context switch
  - MS Windows
- Distributed applications (such as web servers)
  - Each web request may require the creation of a new process
  - Each message sent over the network is an I/O operation and may trigger a context switch
  - Apache web server
- Fundamental reason:
  - "Embarrassingly parallel" applications can no longer afford the overhead
- Same application: Protection/isolation no longer needed

Solution: Threads

- Each process contains multiple threads of execution
- Goal of threads: Allow parallelism with a low cost
  - (Goal of processes: Isolation and protection)

- The state of a thread contains
  - Value of program counter (PC)
  - Value of CPU registers
  - Value of stack space
- All threads in the same process share
  - Virtual memory space
  - Code section
  - Data section (heap space)
  - Operating system resources, e.g., open file

Processes and Threads



A traditioanl process can be viewed as a process with a single thread

All threads within the same process are supposed to be cooperative – No protection is provided!

Will you run downloaded code in a process or thread?

Threads Are Cheap: Creation Cost

- No need to create independent address space
  - Use the same address space with other threads
  - Cost amortized
- No need to allocate and maintain open file information
  - All threads in the same process
- Threads can be implemented entirely in the user-level (not visible to the OS)
  - Java Virtual Machine being a prominent example
  - Can have thousands of threads

Threads Are Cheap: Context Switching

- Process context switching requires
  - Trapping into the OS
  - Saving and changing memory mapping
  - Saving and changing open file table
  - Saving and changing CPU context (e.g., register values, program counters, stack pointer)

- Thread context switch requires
  - Saving and changing CPU context (e.g., register values, program counters, stack pointer)

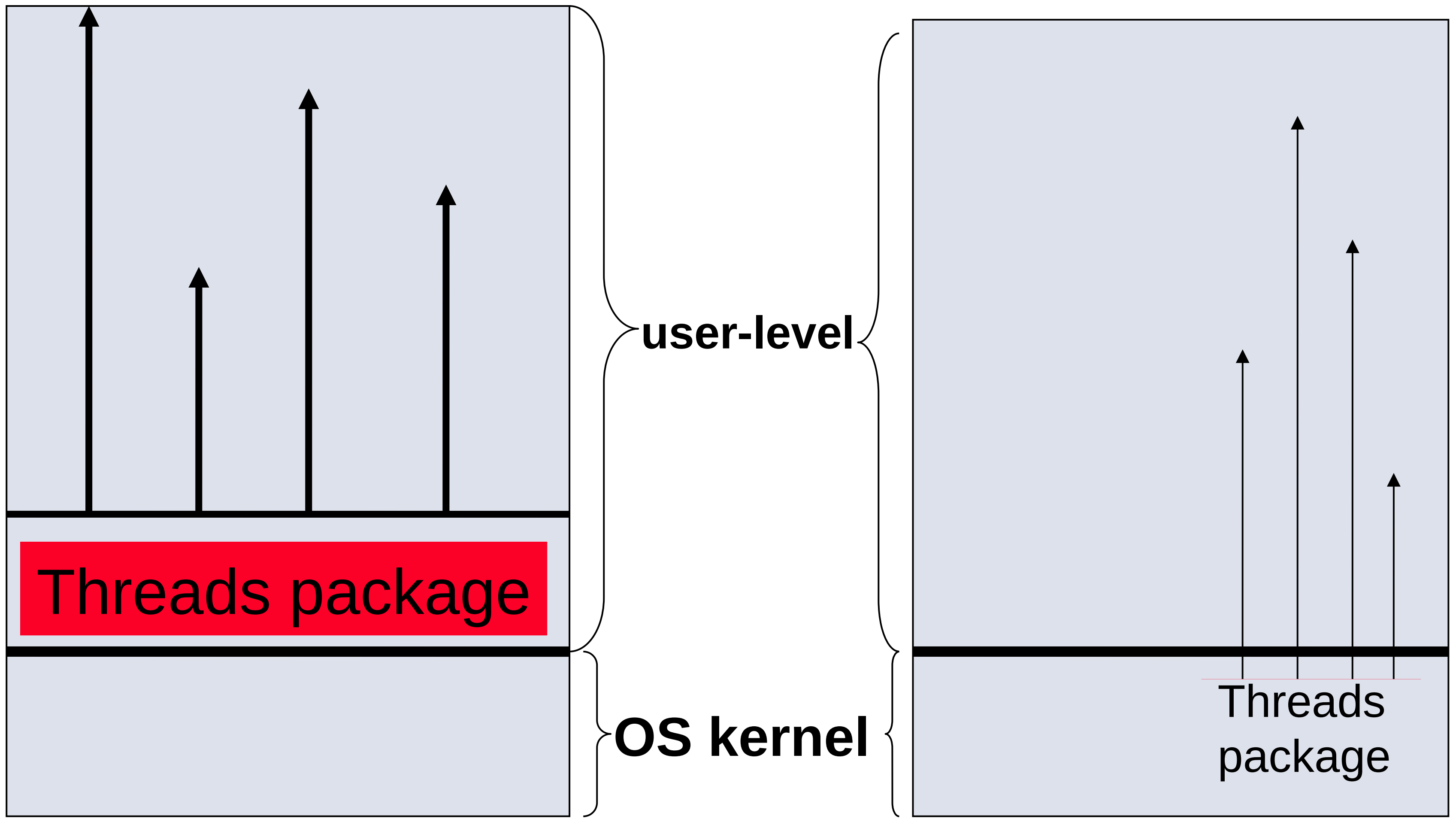
Threads Are Cheap: Inter-Process Communication

- Threads in the same process see the same virtual memory (shared memory)
  - Can exchange information by updating the shared memory
  - Extremely simple (and cheap) memory read/write operations
  - No OS intervention is needed
  - In Java, a thread can access any shared memory that it has a pointer to
  - A thread may corrupt the shared memory and cause all threads to fail

Implementing Threads

- User-level threads
  - Usually implemented as a thread library or a thread package
- Kernel-level threads
- Hybrid (lightweight processes or LWP)
  - The term LWP has been defined by different people with different meaning
  - We will stick with the definition on Tanenbaum book

Kernel-level Vs User-level Threads



Implementing User-level Threads

- Creation, deletion, context switch of threads are done at user level as a library (i.e., thread package)
  - Threads are not visible to OS
- Example:
  - Some UNIX have user-level thread packages
  - Java
- Pros:
  - No OS modification required
  - Cheap to create, destroy, and context switch threads
- Cons:
  - User CANNOT interrupt a thread (no way to preempt a thread if it hangs)
  - Invocation of blocking system call will block all threads in that process

Avoiding Blocking All Threads

- OS can provide non-blocking systems calls
  - Example: non-blocking file read and write and then provide a polling mechanism

- OS can provide upcall to the thread package
  - Tell the thread package that the thread is blocked
  - Will also notify the thread package when the thread is unblocked

```
read(file, buffer);
// return immediately
if (file.readComplete)
    continue;
else
    switch to another thread
```

Both approaches need OS support – most modern OS do have such support

Implementing Kernel-level Threads

- Example: OS/2
- Pros:
  - No problem with blocking system calls
- Cons:
  - Need extensive support from the OS
  - Thread creation/deletion entails system calls (more expensive that user-level thread creation/deletion)
  - User does not have control over thread scheduling (especially bad for real-time applications)
  - Scalability (OS kernel space sometimes cannot grow too big)

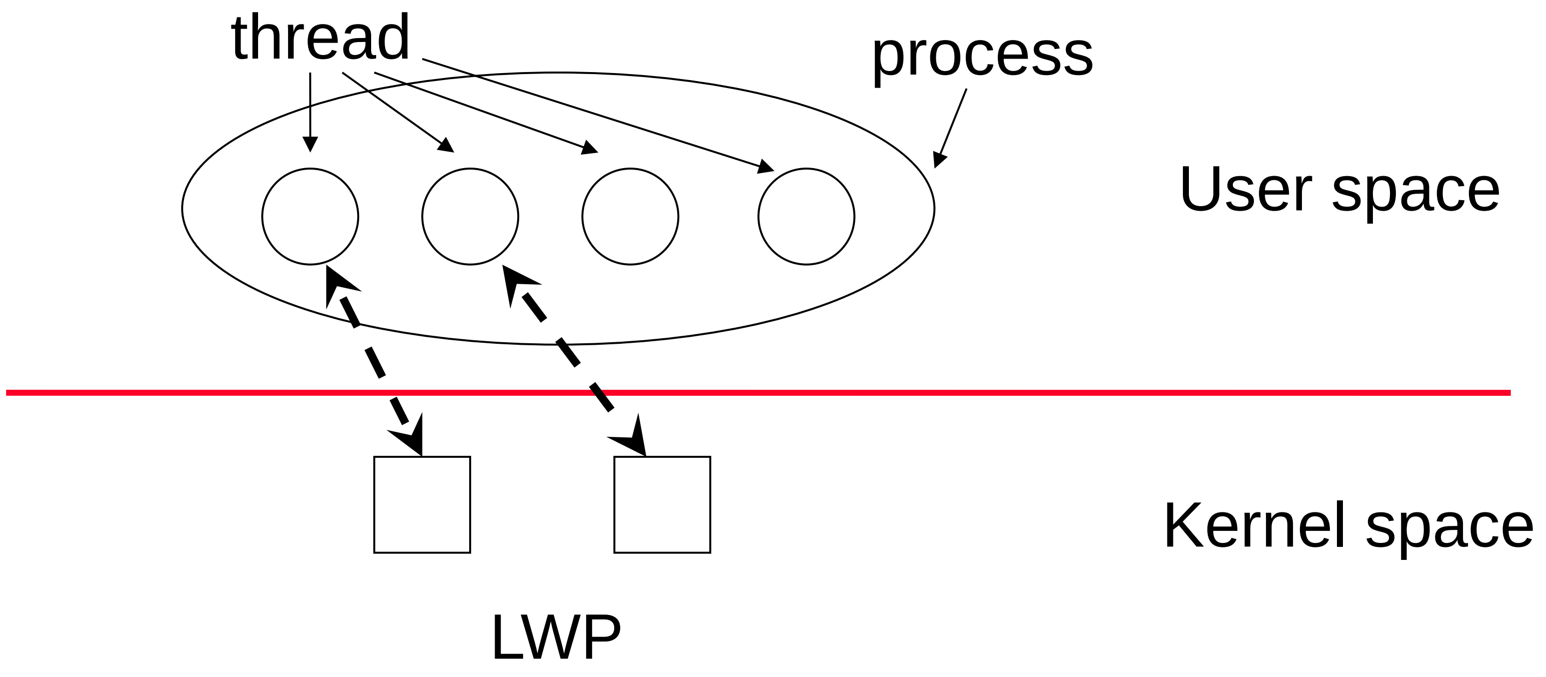
Implementing Threads Using the Hybrid Approach

- Key problem with user-level threads: Will block on blocking system calls
  - OS does not see the n threads in a process P1 – it only sees a single process P1
  - After P1 invoking the blocking system call, OS will pick another process P2 to run – the thread package (in P1) does not get control back

- Idea: We create multiple lightweight processes, so that P2 is created by the thread package as well.
  - The thread package still gets control back
  - How many LWP should we use? PollEv.com/haifengyu229

The Hybrid Approach

- Example: Solaris 2
- OS has the notion of LWP as the unit of scheduling
- A user-level process may have n threads and are associated with k LWP



Comparing the Hybrid Approach with User-Level Threads

- The hybrid approach is a compromise
- Inherits most of the pros of user-level threads
- But still have the scalability concern
  - k needs to be larger than the number of potentially blocked threads

Using Threads in Distributed Systems

- Multi-threading the client
  - Same motivation as using multiple process in non-distributed systems
  - Communication over the network is one kind of I/O
- Prominent examples:
  - Retrieving a web page
  - Both fair and unfair
- Multi-threading the server (e.g., web server) is more critical
  - Simplified software structure
  - Improves performance

Multi-threaded Server: A Programming Paradigm

- Multi-threaded software are usually harder to design / understand than single-threaded software
  - But server is an exception
- Server accepts client request and send back a reply accordingly
  - Conceptually, each request is a “job” and the “job” is simple and small
  - Use one thread for each request
- Much easier to comprehend than having a single thread that works on a “pool of jobs”
  - More on this subject later…

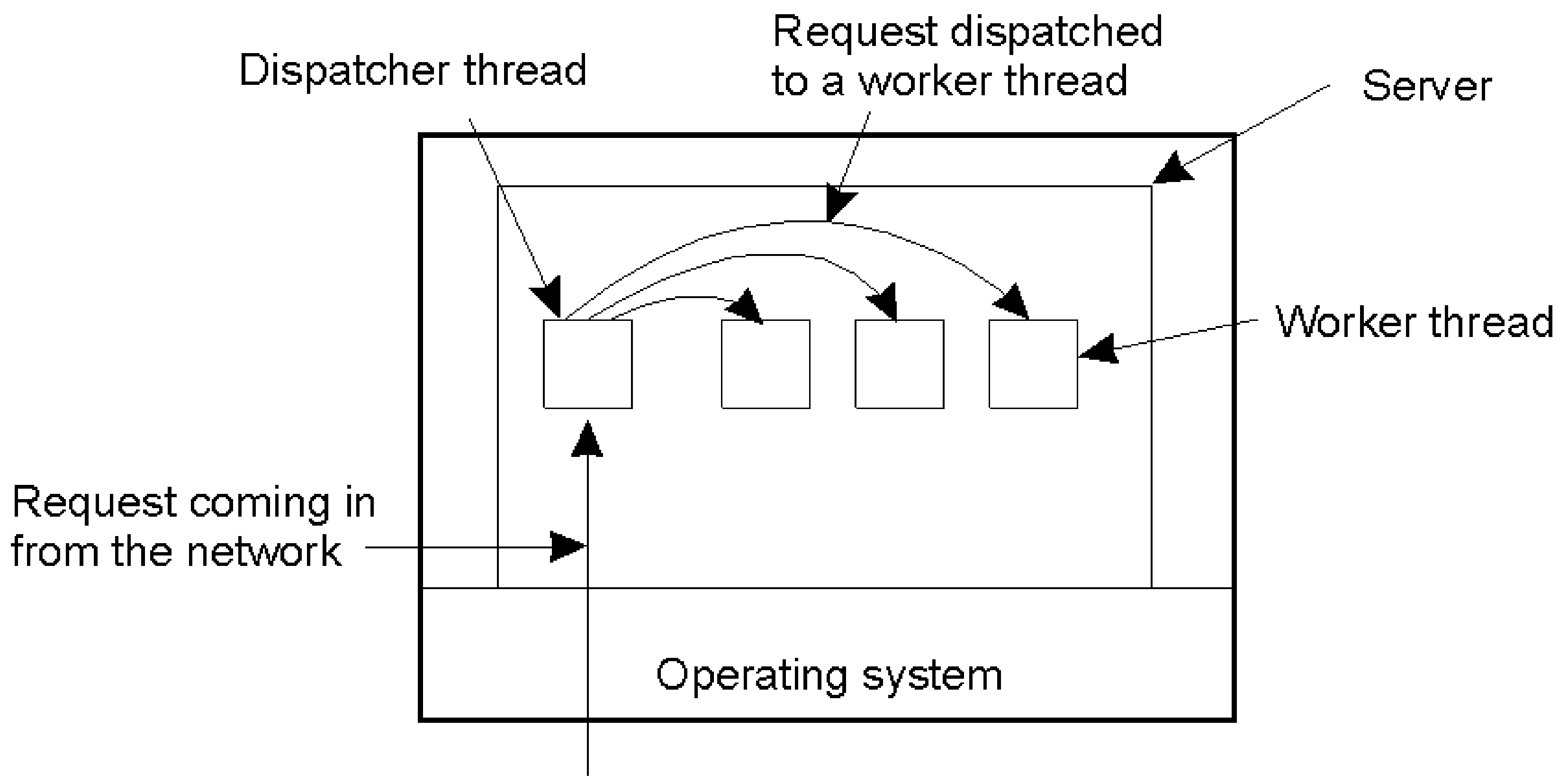
Multi-threaded Server: Exploiting “Embarrassingly Parallel Workload”

- The server needs to do a larger number of small “jobs”
  - Each job may block on disk access and network access

- Threads naturally exploits such parallelism
  - Processes are too expensive to exploit such parallelism
  - The overhead of doing a “job” may even be smaller than the overhead of creating a process

Multi-threaded Server Architecture

Not the only possible structure, but is indeed used prevalently (e.g., in Apache web server)


Dispatcher thread
Request dispatched to a worker thread
Server
Worker thread
Request coming in from the network
Operating system

The Worker Thread Pool

- Even creating threads are not free
  - The thread pool avoids overhead of creating and destroying threads

- If thread pool is a good idea, why not use it in all multi-threaded software (such as your web browser, which is a multi-threaded client)? PollEv.com/haifengyu229

- Maintaining a thread pool ensures system stability over perk load

Event-Driven Server Architecture

- Even (user-level) threads have context switch overheads
  - Saving & restoring CPU state
  - Saving & restoring stack pointers
  - how to eliminate all these
- A single thread working on pool of requests
  - Assumes OS has non-blocking system calls
  - So-called event-driven architecture
  - The thread reacts to events (e.g., new jobs coming or completed I/O)

Pros and Cons of Event-Driven Architecture

- Pros:
  - Better performance (e.g., a event-driven web server written in Java has 30% more throughput than a multi-threaded web server written in C)
- Cons:
  - Hard to understand – most people are more comfortable with multi-threaded servers
  - There is a fundamental reason…later
- Open question:
  - Whether the performance improvement is worth the complexity (and potentially more bugs)

Deeper Discussion on Event-Driven Architecture

- What exactly is the different between multi-threaded server and event-driven architecture?
  - In event-driven architecture, the single thread needs to maintain stage information for each request
  - This is essentially a special case of the thread state in multi-threaded architecture

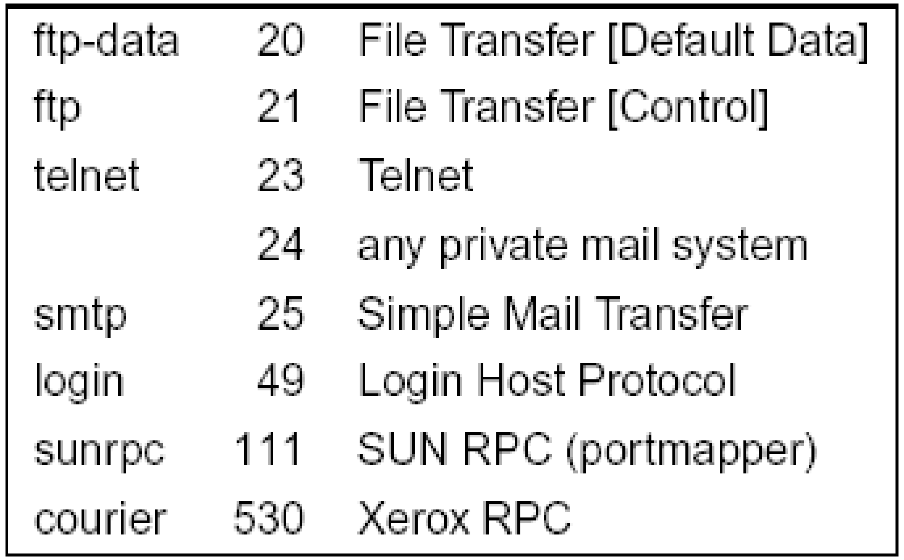
- Even-driven architecture = A highly specialized user-level “thread” package
  - Source of performance gain: specialization
  - No need to save & restore irrelevant CPU states
  - Source of complexity: You are basically writing a simple thread package

History Readings (Non-compulsory)

- SEDA:
  - Original paper on event-driven web servers
  - http://www.sosp.org/2001/papers/welsh.pdf

More on Server Design

- We have discussed the architecture of a server
  - Multi-threaded and event-driven
- Locating a server:
  - Finding the IP address – DNS (later)
  - Finding the port – well-known or use superserver



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

Out-of-Band Request

- 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 (possibly per service request)
  - Server has a separate thread (or process) waiting for incoming urgent messages
  - When urgent message comes in, the associated service request is put on hold
  - Note: we require OS supports high-priority scheduling of specific threads or processes
  - We may also rely on out-of-band communication in TCP

Stateless and Stateful Servers

- Stateful servers – record information about what previous requests a client have made
- Stateless servers – do not record such information
  - May record some bookkeeping information which is not needed for correctness
- Examples of stateful servers:
  - FTP servers (the server remember which directory you are in)
  - File servers (keeps track of which files you have opened)
  - ATM / Banking systems

Stateless Servers

- Examples of stateless servers:
  - Web servers

- Theoretically, we can always convert a stateful server into a stateless server
  - By requiring the client to include all past requests
  - Sometime this is indeed what we do when banking records are messed up
  - Web server send back cookies to clients, which the client will include in the next request

Pros and Cons of Stateful Servers

- Pros:
  - Simplifies the job for the client
  - More efficient in some cases (e.g., file server and FTP server)

- Cons:
  - Much more complex than stateless server
  - Extremely difficult to deal with server failures
  - E.g., to deal with failures in banking systems, we need a whole theory for database recovery
  - Extremely difficult to deal with client failures as well
  - E.g., client may crash when holding locks to a file

Choosing between Stateful and Stateless

- Different states may have different requirements
  - The server can be stateful for some states but stateless for other

- There are also intermediate design points: Session states
  - The state is only maintained within a session
  - Example: FTP (but not banking)

- Generally, try to be stateless unless you have reasons not to be

Server Cluster/Farm

- Many larger-scale services are provided by server clusters
  - Example: Search engine (google), Internet data center, large-scale E-commerce
  - Thousands of nodes or more

- Typically use a three-tier architecture, but not always
  - Tiering provide modular design and a E-commerce site may have its own software for 2nd tier but 3rd party software for other tiers

- Design choice: How to allocate machines to different tiers
- Design choice: How to allocate requests to different machines
- A lot of research on these (driven by vast commercial interests)

Server Cluster/Farm (continued)

- Managing a server cluster/farm can be a nightmare
  - Nobody has a perfect answer
- Google probably has the most experience and expertise on that
  - But proprietary not published
- Challenges:
  - Heterogeneity
  - Frequent failures (there’s never a time when all nodes are up)
  - Lack of supporting software
- An area with a lot of money

Code Migration: Motivation

- Performance improvement
  - Process moved from heavily-loaded to lightly-loaded machine
  - Prominent examples: Java applet and game
- Dynamic code migration: Migrate code based on current observation
  - Why we cannot do load balancing well enough
  - Load changing and unpredictable
- Reduce communication
  - E.g. Database client-server system, move client application to operate on data to reduce network communication costs
  - Helps only if the code size is smaller than the data size

Code Migration: Challenges

- Security
- Heterogeneity
- Dynamic code migration may cause threshing
- Migration decisions are often not trivial to make
- Dynamic code migration needs proper preservation of execution environment

Rule of Thumb: Code migration is frequently used for small pieces of code (such as java Applet), but less so for complex programs. Dynamically migrating an entire running process is possible but needs strong justifications due to its complexity.

Today’s Summary

- Review of processes in operating systems
- Threads and light-weight process (LWP)

- Multi-threaded clients
- Designing servers

- Code migration