合肥學院
HEFEI UNIVERSITY

Distributed Computing
Lesson 8: Threads and Parallelism

Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://www.it-weise.de

Hefei University, South Campus 2
Faculty of Computer Science and Technology
Institute of Applied Optimization
230601 Shushan District, Hefei, Anhui, China
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区
计算机科学与技术系
应用优化研究所
中国 安徽省 合肥市 蜀山区 230601
经济技术开发区 锦绣大道99号

Outline

website

Overview

- Servers need to deal with multiple clients at the same time.
- Dealing with clients may involve I/O to/from the disk or communication with other processes, meaning that at times, the CPU does no real work for a task (but waits for I/O completion)?
- The CPU time can be used more efficiently if "shared" between clients.
- Threads allow for having multiple, independent, (quasi-)parallel streams of execution in a program.
- Threads are resources that can be pooled and cached.

So far...

- Distributed systems are inherently parallel

So far...

- Distributed systems are inherently parallel
- On each node, some process may be running

So far...

- Distributed systems are inherently parallel
- On each node, some process may be running
- Multiple nodes may communicate at the same time

So far...

- Distributed systems are inherently parallel
- On each node, some process may be running
- Multiple nodes may communicate at the same time
- Multiple connection requests or data packets may arrive at a server socket at the same time or close to each other

So far...

- Distributed systems are inherently parallel
- On each node, some process may be running
- Multiple nodes may communicate at the same time
- Multiple connection requests or data packets may arrive at a server socket at the same time or close to each other
- The examples so far process one request (= job) at a time, roughly in FIFO order

So far...

- Distributed systems are inherently parallel
- On each node, some process may be running
- Multiple nodes may communicate at the same time
- Multiple connection requests or data packets may arrive at a server socket at the same time or close to each other
- The examples so far process one request (= job) at a time, roughly in FIFO order
- If we have a single processor and no blocking (system) calls during a job, this is OK

So far...

- Distributed systems are inherently parallel
- On each node, some process may be running
- Multiple nodes may communicate at the same time
- Multiple connection requests or data packets may arrive at a server socket at the same time or close to each other
- The examples so far process one request (= job) at a time, roughly in FIFO order
- If we have a single processor and no blocking (system) calls during a job, this is OK
- Usually, we have multiple (virtual) processors AND blocking system calls

So far...

- Distributed systems are inherently parallel
- On each node, some process may be running
- Multiple nodes may communicate at the same time
- Multiple connection requests or data packets may arrive at a server socket at the same time or close to each other
- The examples so far process one request (= job) at a time, roughly in FIFO order
- If we have a single processor and no blocking (system) calls during a job, this is OK
- Usually, we have multiple (virtual) processors AND blocking system calls
- We waste runtime.

So far...

- Distributed systems are inherently parallel
- On each node, some process may be running
- Multiple nodes may communicate at the same time
- Multiple connection requests or data packets may arrive at a server socket at the same time or close to each other
- The examples so far process one request (= job) at a time, roughly in FIFO order
- If we have a single processor and no blocking (system) calls during a job, this is OK
- Usually, we have multiple (virtual) processors AND blocking system calls
- We waste runtime.
- So what to do?

Single-Machine Parallelism

- First and second generation systems: only single program in execution

Single-Machine Parallelism

- First and second generation systems: only single program in execution
- For better resource utilization: multiple programs loaded, "switching" of active process

Single-Machine Parallelism

- First and second generation systems: only single program in execution
- For better resource utilization: multiple programs loaded, "switching" of active process
- Preemptive multitasking: "switching" performed by the operating system, e.g., via timer interrupt

Single-Machine Parallelism

- First and second generation systems: only single program in execution
- For better resource utilization: multiple programs loaded, “switching” of active process
- Preemptive multitasking: “switching” performed by the operating system, e.g., via timer interrupt
- Scheduling: decision about which activity should be execute when in order to optimize characteristics (response time, throughput)

Single-Machine Parallelism

- First and second generation systems: only single program in execution
- For better resource utilization: multiple programs loaded, "switching" of active process
- Preemptive multitasking: "switching" performed by the operating system, e.g., via timer interrupt
- Scheduling: decision about which activity should be execute when in order to optimize characteristics (response time, throughput)
- 3rd generation systems: virtualization: memory, processors

Processes

- Processes are the basic activities of a system

Processes

- Processes are the basic activities of a system
- Processes use resources exclusively

Processes

- Processes are the basic activities of a system
- Processes use resources exclusively, e.g.,
  1. (virtual) memory

Processes

- Processes are the basic activities of a system
- Processes use resources exclusively, e.g.,
  1. (virtual) memory:
     - each process has an own virtual address space

Processes

- Processes are the basic activities of a system
- Processes use resources exclusively, e.g.,
  1. (virtual) memory:
     - each process has an own virtual address space
     - other processes cannot read or write into this memory (except for shared memory)

Processes

- Processes are the basic activities of a system
- Processes use resources exclusively, e.g.,
  1. (virtual) memory:
     - each process has an own virtual address space
     - other processes cannot read or write into this memory (except for shared memory)
     - processes thus "think" that they are "alone" in the memory

Processes

- Processes are the basic activities of a system
- Processes use resources exclusively, e.g.,
  1. (virtual) memory:
     - each process has an own virtual address space
     - other processes cannot read or write into this memory (except for shared memory)
     - processes thus "think" that they are "alone" in the memory
     - see operating systems lectures [1–11] ...

Processes

- Processes are the basic activities of a system
- Processes use resources exclusively, e.g.,
  1. (virtual) memory
  2. (virtual) processor

Processes

- Processes are the basic activities of a system
- Processes use resources exclusively, e.g.,
  1. (virtual) memory
  2. (virtual) processor:
     - scheduling transparent for processes

Processes

- Processes are the basic activities of a system
- Processes use resources exclusively, e.g.,
  1. (virtual) memory
  2. (virtual) processor:
     - scheduling transparent for processes
     - processes think they have their own processor on which only they are executed

Processes

- Processes are the basic activities of a system
- Processes use resources exclusively, e.g.,
  1. (virtual) memory
  2. (virtual) processor:
     - scheduling transparent for processes
     - processes think they have their own processor on which only they are executed
     - see operating systems lectures [1–11] ...

Processes

- Processes are the basic activities of a system
- Processes use resources exclusively, e.g.,
  1. (virtual) memory
  2. (virtual) processor
  3. other resources

Processes

- Processes are the basic activities of a system
- Processes use resources exclusively, e.g.,
  1. (virtual) memory
  2. (virtual) processor
  3. other resources
     - usually via "handles", i.e., unique IDs identifying resource owner which are valid only inside the process which acquired them
     - sockets in `C` are such handles, socket objects in `Java` map to handles

Processes

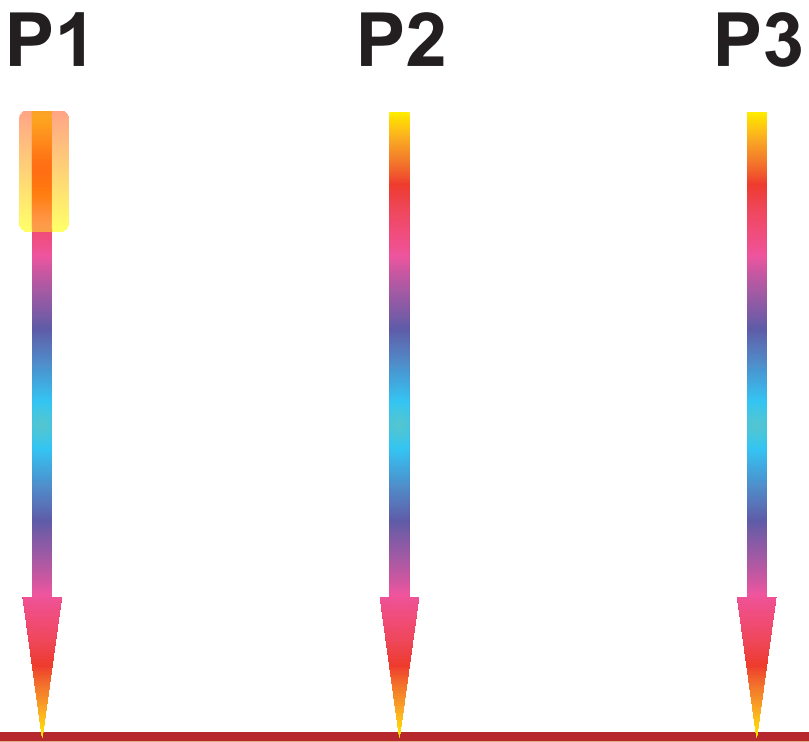
- Processes are the basic activities of a system
- Processes use resources exclusively, e.g.,
  1. (virtual) memory
  2. (virtual) processor
  3. other resources
     - usually via "handles", i.e., unique IDs identifying resource owner which are valid only inside the process which acquired them
     - sockets in `C` are such handles, socket objects in `Java` map to handles
     - handles not visible/useful for other processes

Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]

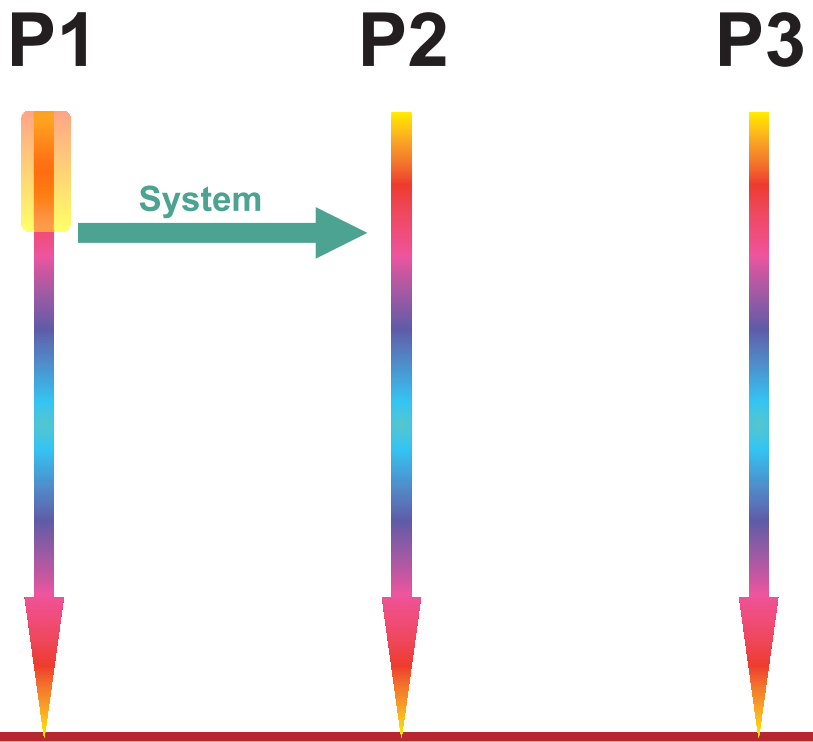
Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]



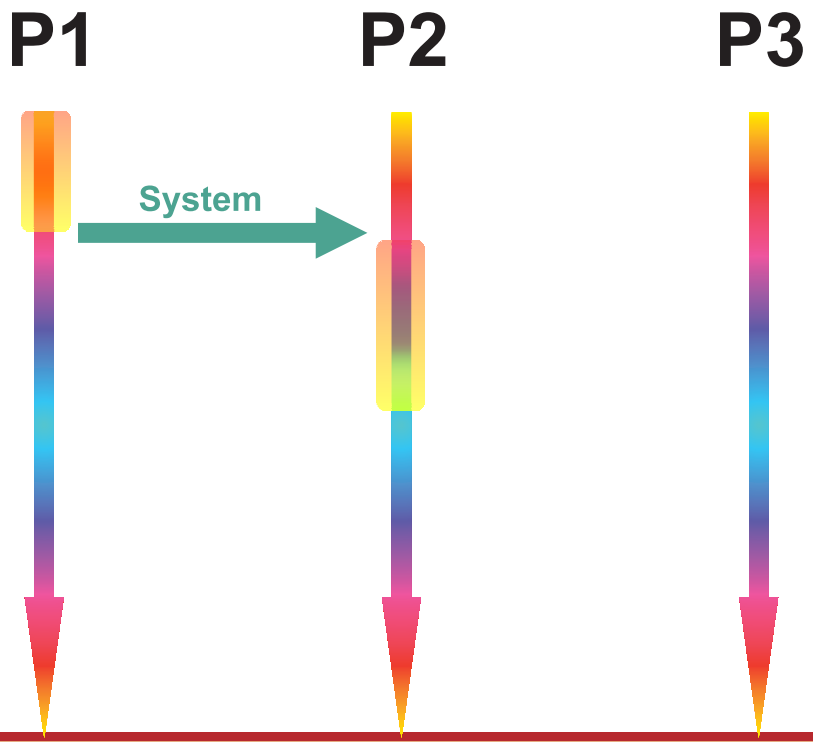
Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]



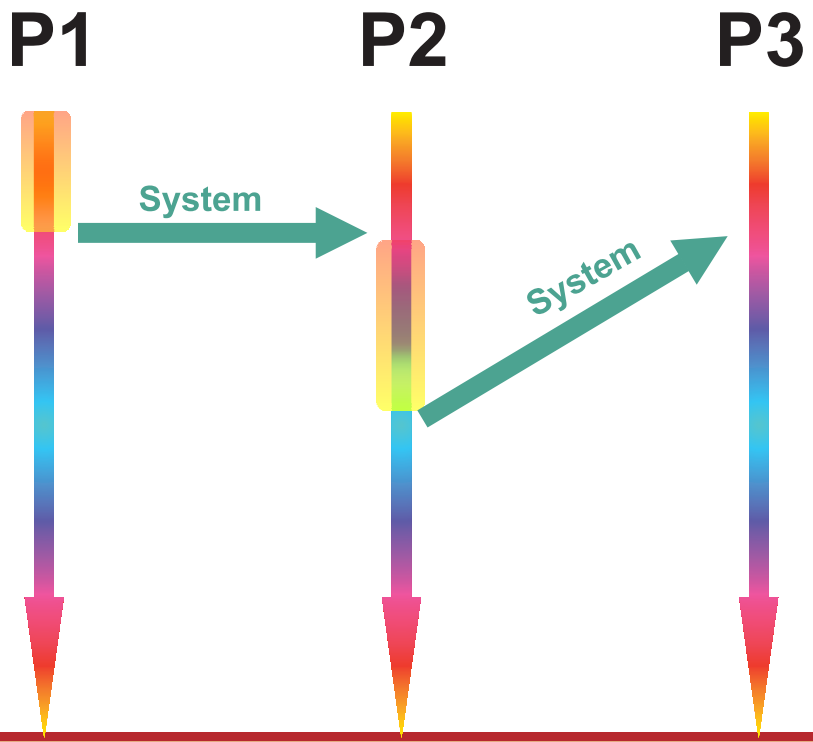
Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]



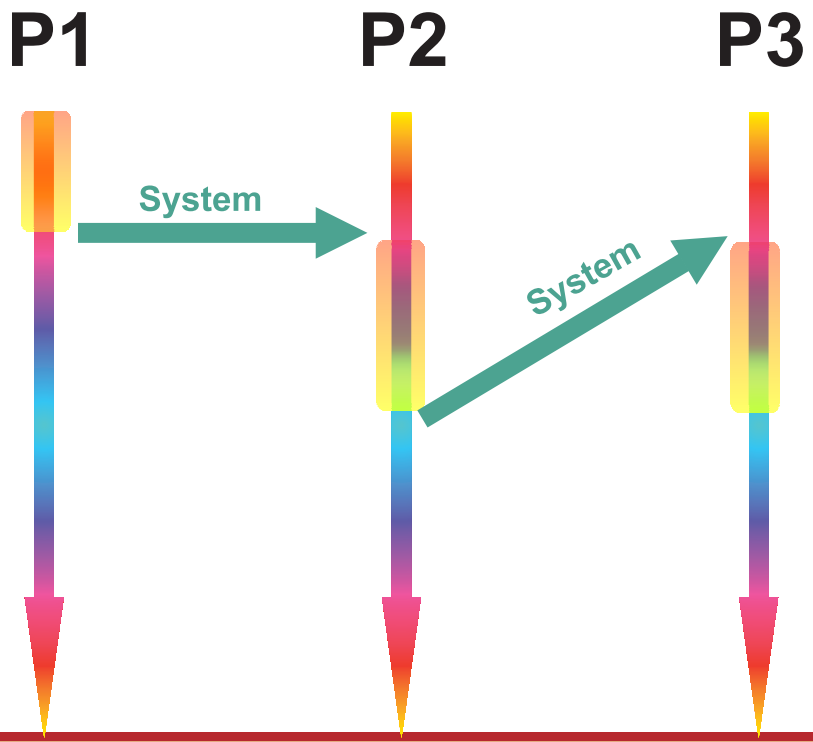
Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]



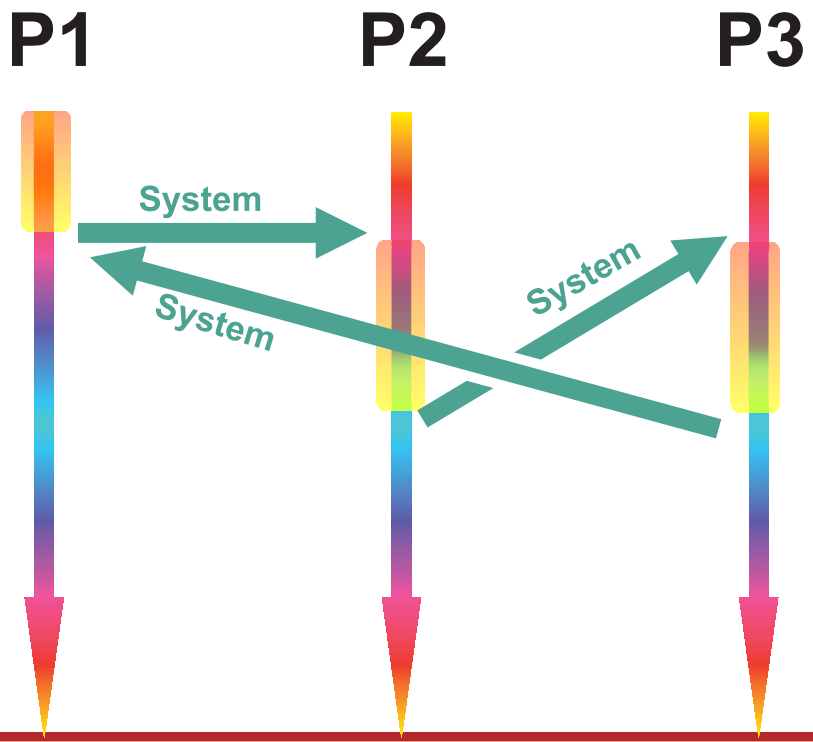
Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]



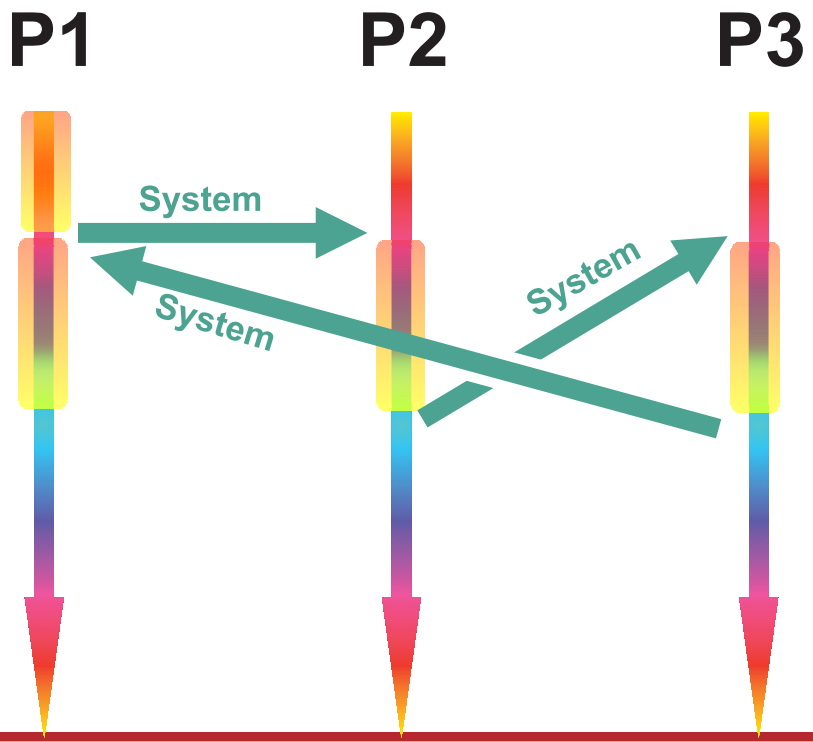
Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]



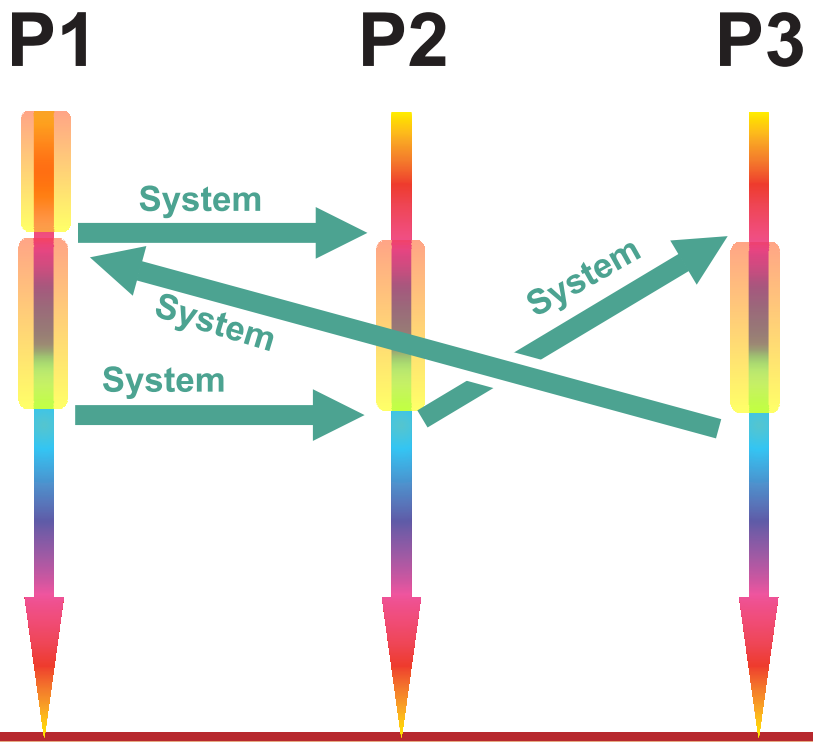
Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]



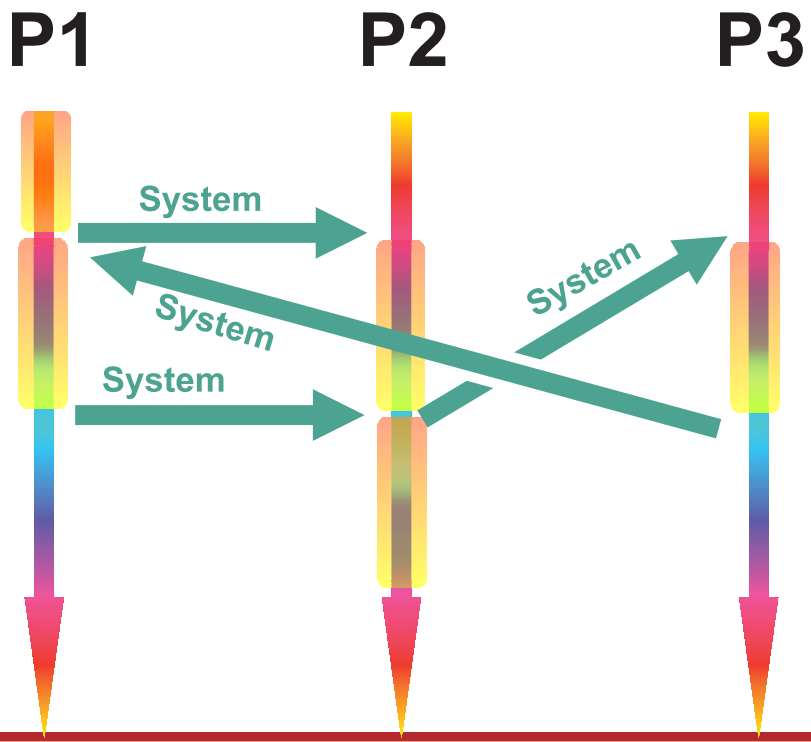
Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]



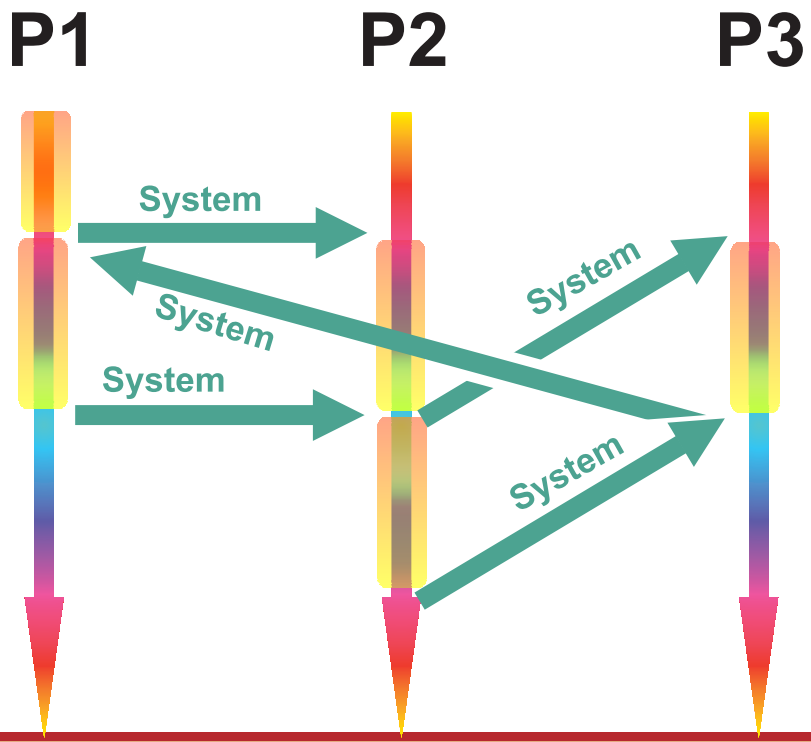
Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]



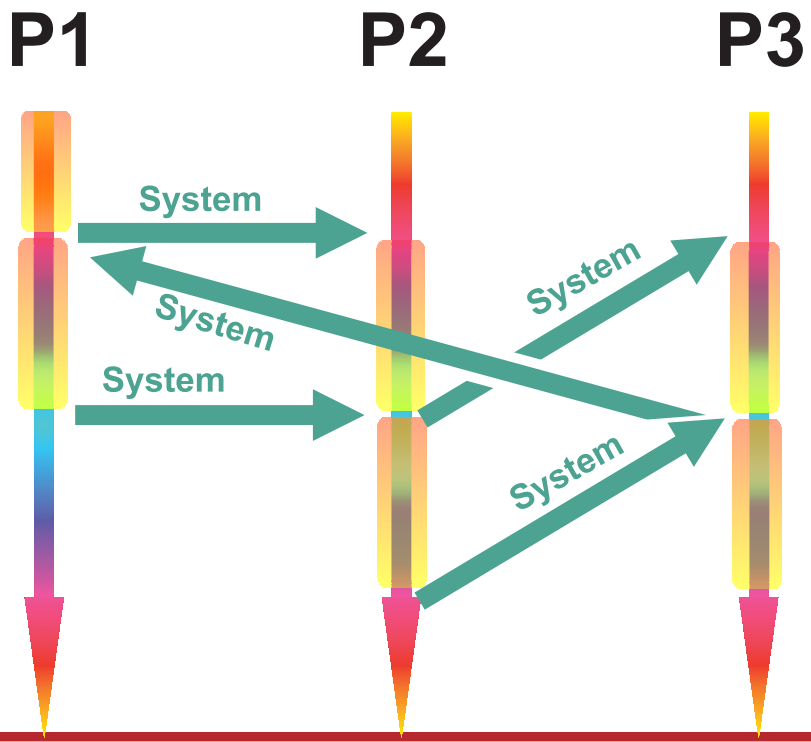
Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]



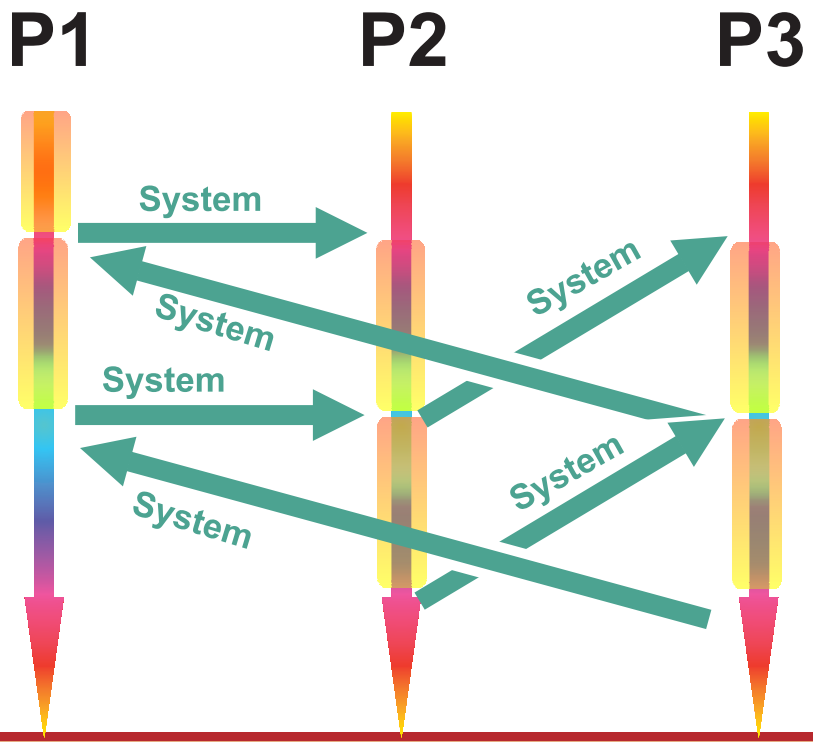
Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]



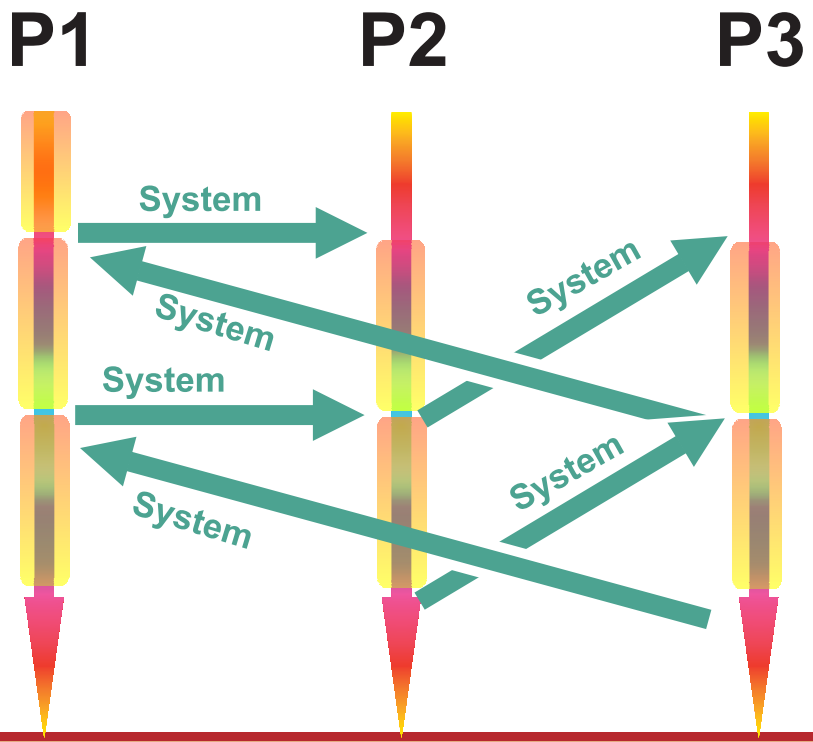
Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]



Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]



Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]
- storing registers and process counter in PCB

Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]
- storing registers and process counter in PCB
- selecting next process (PCB)

Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]
- storing registers and process counter in PCB
- selecting next process (PCB)
- restoring registers, instruction pointer, virtual memory table pointer

Parallelism: Context Switch

- Processes run quasi-parallel: OS performs context switches [1–11]
- storing registers and process counter in PCB
- selecting next process (PCB)
- restoring registers, instruction pointer, virtual memory table pointer
- flushing of caches

Process Creation

- Creation of a new process involves a couple of steps

Process Creation

- Creation of a new process involves a couple of steps:
  - allocating new Process Control Block (PCB)

Process Creation

- Creation of a new process involves a couple of steps:
  - allocating new Process Control Block (PCB)
  - initializing other data structures (e.g., for virtual memory)

Process Creation

- Creation of a new process involves a couple of steps:
  - allocating new Process Control Block (PCB)
  - initializing other data structures (e.g., for virtual memory)
  - loading first few pages from program code

Process Creation

- Creation of a new process involves a couple of steps:
  - allocating new Process Control Block (PCB)
  - initializing other data structures (e.g., for virtual memory)
  - loading first few pages from program code
  - loading required libraries

Processes: Advantages/Disadvantages

- **Advantages?**
- **Disadvantages?**

Processes: Advantages/Disadvantages

- **Advantages?**
  - Security: other processes cannot read memory / confidential data
- **Disadvantages?**

Processes: Advantages/Disadvantages

- **Advantages?**
  - Security: other processes cannot read memory / confidential data
  - Safety: if one process fails, it cannot influence other processes directly

- **Disadvantages?**

Processes: Advantages/Disadvantages

- **Advantages?**
  - Security: other processes cannot read memory / confidential data
  - Safety: if one process fails, it cannot influence other processes directly
  - Virtual memory: More memory can be used than actually physically available
- **Disadvantages?**

Processes: Advantages/Disadvantages

- **Advantages?**
  - Security: other processes cannot read memory / confidential data
  - Safety: if one process fails, it cannot influence other processes directly
  - Virtual memory: More memory can be used than actually physically available
- **Disadvantages?**
  - Inter-process communication (IPC) slow

Processes: Advantages/Disadvantages

- **Advantages?**
  - Security: other processes cannot read memory / confidential data
  - Safety: if one process fails, it cannot influence other processes directly
  - Virtual memory: More memory can be used than actually physically available
- **Disadvantages?**
  - Inter-process communication (IPC) slow
  - context switch slow

Processes: Advantages/Disadvantages

- **Advantages?**
  - Security: other processes cannot read memory / confidential data
  - Safety: if one process fails, it cannot influence other processes directly
  - Virtual memory: More memory can be used than actually physically available
- **Disadvantages?**
  - Inter-process communication (IPC) slow
  - context switch slow
  - explicit sharing of data/information complicated

Processes: Advantages/Disadvantages

- **Advantages?**
  - Security: other processes cannot read memory / confidential data
  - Safety: if one process fails, it cannot influence other processes directly
  - Virtual memory: More memory can be used than actually physically available
- **Disadvantages?**
  - Inter-process communication (IPC) slow
  - context switch slow
  - explicit sharing of data/information complicated
  - initialization/management time and resource consuming

Processes: Advantages/Disadvantages

- **Advantages?**
  - Security: other processes cannot read memory / confidential data
  - Safety: if one process fails, it cannot influence other processes directly
  - Virtual memory: More memory can be used than actually physically available
- **Disadvantages?**
  - Inter-process communication (IPC) slow
  - context switch slow
  - explicit sharing of data/information complicated
  - initialization/management time and resource consuming
  - passing of handles complex

Threads

- “Lightweight Processes”

Threads

- “Lightweight Processes”
  - a process can own an arbitrary number of threads

Threads

- “Lightweight Processes”
  - a process can own an arbitrary number of threads
  - all threads of a process run quasi-parallel to each other

Threads

- “Lightweight Processes”
  - a process can own an arbitrary number of threads
  - all threads of a process run quasi-parallel to each other
  - scheduling via OS (kernel-mode threads) or owning process (usermode threads)

Threads

- “Lightweight Processes”
  - a process can own an arbitrary number of threads
  - all threads of a process run quasi-parallel to each other
  - scheduling via OS (kernel-mode threads) or owning process (usermode threads)
  - all threads of one process share the resources of this process

Threads

- “Lightweight Processes”
  - a process can own an arbitrary number of threads
  - all threads of a process run quasi-parallel to each other
  - scheduling via OS (kernel-mode threads) or owning process (usermode threads)
  - all threads of one process share the resources of this process
  - all threads of one process reside in this processes address space

Threads

- “Lightweight Processes”
  - a process can own an arbitrary number of threads
  - all threads of a process run quasi-parallel to each other
  - scheduling via OS (kernel-mode threads) or owning process (usermode threads)
  - all threads of one process share the resources of this process
  - all threads of one process reside in this processes address space
  - each thread has private stack and virtual processor

Threads

- "Lightweight Processes"
  - a process can own an arbitrary number of threads
  - all threads of a process run quasi-parallel to each other
  - scheduling via OS (kernel-mode threads) or owning process (usermode threads)
  - all threads of one process share the resources of this process
  - all threads of one process reside in this processes address space
  - each thread has private stack and virtual processor
- Context switch much faster: caches do not need to be flushed, virtual memory does not need to be switched

Threads

- "Lightweight Processes"
  - a process can own an arbitrary number of threads
  - all threads of a process run quasi-parallel to each other
  - scheduling via OS (kernel-mode threads) or owning process (usermode threads)
  - all threads of one process share the resources of this process
  - all threads of one process reside in this processes address space
  - each thread has private stack and virtual processor
- Context switch much faster: caches do not need to be flushed, virtual memory does not need to be switched
- No security threat: all threads in one process are part of the same program

What to use?

- Traditionally: Web Servers `fork`ed in Unix

What to use?

- Traditionally: Web Servers `fork`ed in Unix:
  - multi-process systems

What to use?

- Traditionally: Web Servers `fork`ed in Unix:
  - multi-process systems
  - safe

What to use?

- Traditionally: Web Servers `fork`ed in Unix:
  - multi-process systems
  - safe
  - high resource consumption

What to use?

- Traditionally: Web Servers `fork`ed in Unix:
  - multi-process systems
  - safe
  - high resource consumption
  - for each request, a new process is created, which costs time before the request is processed

What to use?

- Traditionally: Web Servers `fork`ed in Unix:
  - multi-process systems
  - safe
  - high resource consumption
  - for each request, a new process is created, which costs time before the request is processed
- Multi-threaded servers

What to use?

- Traditionally: Web Servers `fork`ed in Unix:
  - multi-process systems
  - safe
  - high resource consumption
  - for each request, a new process is created, which costs time before the request is processed
- Multi-threaded servers:
  - multiple threads process client requests in parallel
  - faster
  - less secure/safe: 1 compromised thread can compromise the whole server process

Threads in Java

- Java has built-in, easy-to-use support for multi-threadding [12–16]

Threads in Java

- Java has built-in, easy-to-use support for multi-threadding [12–16]
- Class `Thread`

Threads in Java

- Java has built-in, easy-to-use support for multi-threadding [12–16]
- Class `Thread`:
  - has method `void run()` which does the work and can be overridden

Threads in Java

- Java has built-in, easy-to-use support for multi-threadding [12–16]
- Class `Thread`:
  - has method `void run()` which does the work and can be overridden
  - is started with `void start()`

Threads in Java

- Java has built-in, easy-to-use support for multi-threadding [12–16]
- Class `Thread`:
  - has method `void run()` which does the work and can be overridden
  - is started with `void start()`
  - we can wait until it is finished with `void join()`

Threads in Java

- Java has built-in, easy-to-use support for multi-threadding [12–16]
- Class `Thread`:
  - has method `void run()` which does the work and can be overridden
  - is started with `void start()`
  - we can wait until it is finished with `void join()`
- Interface `Runnable`

Threads in Java

- Java has built-in, easy-to-use support for multi-threadding [12–16]
- Class `Thread`:
  - has method `void run()` which does the work and can be overridden
  - is started with `void start()`
  - we can wait until it is finished with `void join()`
- Interface `Runnable`
  - has method `void run()` which may do some work

Threads in Java

- Java has built-in, easy-to-use support for multi-threadding [12–16]
- Class `Thread`:
  - has method `void run()` which does the work and can be overridden
  - is started with `void start()`
  - we can wait until it is finished with `void join()`
- Interface `Runnable`
  - has method `void run()` which may do some work
  - can be passed into the constructor of `Thread`, thread will then execute `run()` when started

Listing: MinHTTPServerMultiThread.java Multi-Threaded HTTP Server / Java

```
import java.io.BufferedReader;      import java.io.File;        import java.io.FileInputStream;   import java.io.InputStreamReader;
import java.io.OutputStreamWriter;  import java.io.PrintWriter; import java.net.ServerSocket;     import java.net.Socket;

public class MinHTTPServerMultiThread {
  public static final void main(final String[] args) {
    ServerSocket    server;     Socket          client;
    try {
      server = new ServerSocket(9995); //create server socket 1 + 2)

      for (;;) {  //forever...
        client = server.accept(); //wait for and accept incoming connection 3)
        new Thread(new Job(client)).start(); //create and start a new thread to process the request

      }
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }

  private static final class Job implements Runnable { //the job class: process one request; Runnable is the key interface
    private final Socket m_client; //the client socket to process
    Job(final Socket client){ //create a job for a given socket
      this.m_client=client;
    }

    @Override //this method is executed by the thread that was created with this object as constructor parameter
    public final void run(){ //process the client socket: exactly the same as in the MinHTTPServer example
      BufferedReader  br;   PrintWriter     pw;   String    s;    File  f;
      byte[]          bs;   FileInputStream fis;  Throwable x;    int   i;

      try{
      br = new BufferedReader(new InputStreamReader(this.m_client.getInputStream())); // read character data
      pw = new PrintWriter(new OutputStreamWriter(this.m_client.getOutputStream(), "ISO_8859-1")); //chose the right encoding! [17, 18]

      process: { //4 + 3)
        x = null;
        try {
          while ((s = br.readLine()) != null) { //read text from connection line-by-line until end
            if (s.startsWith("GET ")) {  // try to find the GET command in the HTTP request [17, 18]
              f   = new File(s.substring(4, s.indexOf(' ', 4)).replace('/', File.separatorChar)); //in a very crude way, extract the requested path from that command
              bs  = new byte[(int) (f.length())];  //allocate a buffer of the right size
              fis = new FileInputStream(f);        //open the file
              i   = fis.read(bs);                  //read the complete file into memory
              fis.close();                         //close the file
              pw.write("HTTP/1.1 200 OK\r\n\r\n");  pw.flush(); //send "success" according to [17, 18]
              this.m_client.getOutputStream().write(bs, 0, i); //...and the file content  (4 + 3))
              break process;                       //ok, we are finished here
            }
          }
        } catch (Throwable t) { x= t; } //if request fails, remember why

        //hm, we did not find the file or had an error [17, 18]
        pw.write("HTTP/1.1 404 Not Found\r\n\r\n<html><head><title>404</title></head><body><h1>404 - Not found</h1><pre>");
        if(x != null) { x.printStackTrace(pw); } //write the error message (notice the <pre>...</pre> wrapper)
        pw.write("</pre></body></html");         //end the html body
        pw.flush(); //and flush (4 + 3))
      }

      this.m_client.close(); }//4)
      catch(Throwable error) { error.printStackTrace(); }
    }
  }
}
```

Thread Pools in Java

- Now we create a new thread for every single request

Thread Pools in Java

- Now we create a new thread for every single request
  - Wasteful: many threads are created and used only once (threads are OS resources)

Thread Pools in Java

- Now we create a new thread for every single request
  - Wasteful: many threads are created and used only once (threads are OS resources)
  - What if very many requests come at a time? Fraction of runtime/thread will go down to 0

Thread Pools in Java

- Now we create a new thread for every single request
  - Wasteful: many threads are created and used only once (threads are OS resources)
  - What if very many requests come at a time? Fraction of runtime/thread will go down to 0
- A *thread pool* keeps n threads and has a job queue

Thread Pools in Java

- Now we create a new thread for every single request
  - Wasteful: many threads are created and used only once (threads are OS resources)
  - What if very many requests come at a time? Fraction of runtime/thread will go down to 0
- A *thread pool* keeps n threads and has a job queue
  - If a thread is idle, it takes a job out of the queue and processes it

Thread Pools in Java

- Now we create a new thread for every single request
  - Wasteful: many threads are created and used only once (threads are OS resources)
  - What if very many requests come at a time? Fraction of runtime/thread will go down to 0
- A *thread pool* keeps n threads and has a job queue
  - If a thread is idle, it takes a job out of the queue and processes it
  - If no job is in the queue, it waits for the next job

Thread Pools in Java

- Now we create a new thread for every single request
  - Wasteful: many threads are created and used only once (threads are OS resources)
  - What if very many requests come at a time? Fraction of runtime/thread will go down to 0
- A *thread pool* keeps n threads and has a job queue
  - If a thread is idle, it takes a job out of the queue and processes it
  - If no job is in the queue, it waits for the next job
  - After it is finished, it becomes idle again

Thread Pools in Java

- Now we create a new thread for every single request
  - Wasteful: many threads are created and used only once (threads are OS resources)
  - What if very many requests come at a time? Fraction of runtime/thread will go down to 0
- A *thread pool* keeps n threads and has a job queue
  - If a thread is idle, it takes a job out of the queue and processes it
  - If no job is in the queue, it waits for the next job
  - After it is finished, it becomes idle again
  - A job is executed by at most 1 thread

Thread Pools in Java

- Now we create a new thread for every single request
  - Wasteful: many threads are created and used only once (threads are OS resources)
  - What if very many requests come at a time? Fraction of runtime/thread will go down to 0
- A *thread pool* keeps n threads and has a job queue
  - If a thread is idle, it takes a job out of the queue and processes it
  - If no job is in the queue, it waits for the next job
  - After it is finished, it becomes idle again
  - A job is executed by at most 1 thread
- `ExecutorService` is an interface of classes that can execute `Runnable`s

Thread Pools in Java

- Now we create a new thread for every single request
  - Wasteful: many threads are created and used only once (threads are OS resources)
  - What if very many requests come at a time? Fraction of runtime/thread will go down to 0
- A *thread pool* keeps n threads and has a job queue
  - If a thread is idle, it takes a job out of the queue and processes it
  - If no job is in the queue, it waits for the next job
  - After it is finished, it becomes idle again
  - A job is executed by at most 1 thread
- `ExecutorService` is an interface of classes that can execute `Runnable`s
- `Executors.newFixedThreadPool(n)` creates a thread-pool based `ExecutorService`

Listing: MinHTTPServerThreadPool.java Thread-Pooled HTTP Server / Java

```
import java.io.BufferedReader;    import java.io.File;            import java.io.FileInputStream;   import java.io.InputStreamReader;            import java.io.OutputStreamWriter;
import java.io.PrintWriter;       import java.net.ServerSocket;   import java.net.Socket;           import java.util.concurrent.ExecutorService; import java.util.concurrent.Executors;

public class MinHTTPServerThreadPool {
  public static final void main(final String[] args) {
    ServerSocket    server;     Socket          client;   ExecutorService pool;
    try {
      pool = Executors.newFixedThreadPool(10); //create a pool of 10 threads waiting to execute something
      server = new ServerSocket(9994); //(1 + 2)

      for (;;) {
        client = server.accept(); //wait for and accept new connection (3)
        pool.execute(new Job(client)); //enqueue the job into the pool's job queue, it will be executed when a thread is ready
      }
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }

  private static final class Job implements Runnable { //the job class: process one request; Runnable is the key interface
    private final Socket m_client; //the client socket to process
    Job(final Socket client){ //create a job for a given socket
      this.m_client=client;
    }

    @Override //this method is executed by a thread in the thread pool
    public final void run(){
      BufferedReader  br;   PrintWriter     pw;   String    s;    File  f;
      byte[]          bs;   FileInputStream fis;  Throwable x;    int   i;

      try{
      br = new BufferedReader(new InputStreamReader(this.m_client.getInputStream())); // read character data
      pw = new PrintWriter(new OutputStreamWriter(this.m_client.getOutputStream(), "ISO_8859-1")); //chose the right encoding! [17, 18]

      process: { //(4 + (3))
        x = null;
        try {
          while ((s = br.readLine()) != null) { //read text from connection line-by-line until end
            if (s.startsWith("GET␣")) {  // try to find the GET command in the HTTP request [17, 18]
              f   = new File(s.substring(4, s.indexOf('␣', 4)).replace('/', File.separatorChar)); //in a very crude way, extract the requested path from that command
              bs  = new byte[(int) (f.length())];  //allocate a buffer of the right size
              fis = new FileInputStream(f);        //open the file
              i   = fis.read(bs);                  //read the complete file into memory
              fis.close();                         //close the file
              pw.write("HTTP/1.1␣200␣OK\r\n\r\n");  pw.flush(); //send "success" according to [17, 18]
              this.m_client.getOutputStream().write(bs, 0, i); //...and the file content  (4 + (3))
              break process;                       //ok, we are finished here
            }
          }
        } catch (Throwable t) { x= t; } //if request fails, remember why

        //hm, we did not find the file or had an error [17, 18]
        pw.write("HTTP/1.1␣404␣Not␣Found\r\n\r\n<html><head><title>404</title></head><body><h1>404␣-␣Not␣found</h1><pre>");
        if(x != null) { x.printStackTrace(pw); } //write the error message (notice the <pre>...</pre> wrapper)
        pw.write("</pre></body></html");         //end the html body
        pw.flush(); //and flush (4 + (3))
      }

      this.m_client.close(); }//(4)
      catch(Throwable error) { error.printStackTrace(); }
    }
  }
}
```

Summary

- Parallelism may increase server performance significantly.
- The concept of threads allows for pre-emptive multi-tasking of different (quasi-)parallel strands of a process.
- The class `Thread` implements this in Java.
- Each client of a server can be processed by a different thread.
- Since `Thread`s are expensive system resources, thread pools can hold a set of threads to be re-used for future clients after having completed a task.

谢谢

Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://www.it-weise.de

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui,
China

Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog

Bibliography

Bibliography I

1. William Stallings. *Operating Systems: Internals and Design Principles*. Gradience Online Accelerated Learning Series (GOAL). Upper Saddle River, NJ, USA: Pearson Education and Upper Saddle River, NJ, USA: Prentice Hall International Inc., 2004. ISBN 0029464919, 0130319996, 0131479547, 0131809776, 0136006329, 0138874077, 0139179984, 9780029464915, 9780130319999, 978-0131479548, 9780131809772, 978-0136006329, 9780138874070, and 9780139179983. URL `http://books.google.de/books?id=dBQFXs5NPEYC`.
2. Gary J. Nutt. *Operating Systems: A Modern Perspective*. Reading, MA, USA: Addison-Wesley Publishing Co. Inc., 2002. ISBN 0201741962 and 9780201741964. URL `http://books.google.de/books?id=AHBGAAAAYAAJ`.
3. Andrew Stuart Tanenbaum. *Modern Operating Systems*. Upper Saddle River, NJ, USA: Pearson Education and Upper Saddle River, NJ, USA: Prentice Hall International Inc., 2008. ISBN 0136006639 and 978-0136006633. URL `http://books.google.de/books?id=y22rPwAACAAJ`.
4. Jean Bacon and Tim Harris. *Operating Systems: Concurrent and Distributed Software Design*. International Computer Science Series. Reading, MA, USA: Addison-Wesley Publishing Co. Inc., 2003. ISBN 0321117891 and 9780321117892. URL `http://books.google.de/books?id=kkaaH3Q19Z4C`.
5. Harvey M. Deitel, Paul J. Deitel, and David R. Choffnes. *Operating Systems*. Upper Saddle River, NJ, USA: Pearson Education and Upper Saddle River, NJ, USA: Prentice Hall International Inc., 2004. ISBN 0131828274 and 9780131828278. URL `http://books.google.de/books?id=M45QAAAAMAAJ`.
6. Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill Series in Computer Science: Systems and Languages. Maidenhead, England, UK: McGraw-Hill Ltd., 1994. ISBN 007057572X and 9780070575721. URL `http://books.google.de/books?id=sLlQAAAAMAAJ`.
7. Andrew Stuart Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. Upper Saddle River, NJ, USA: Pearson Education, 2009. ISBN 0135053765 and 9780135053768. URL `http://books.google.de/books?id=MOQhQAAACAAJ`.
8. David A. Solomon and Mark E. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Programming Series. Redmond, WA, USA: Microsoft Press, 2000. ISBN 0735610215 and 9780735610217. URL `http://books.google.de/books?id=3xqBRAAACAAJ`.
9. Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Reading, MA, USA: Addison-Wesley Professional, 2003. ISBN 0321154959 and 9780321154958. URL `http://books.google.de/books?id=mdiIu8Kk1WMC`.
10. Winfried Kalfa. *Betriebssysteme*, volume 24 of *Informatik, Kybernetik, Rechentechnik*. Berlin, Germany: Akademie Verlag, 1988. ISBN 3055004779 and 9783055004773. URL `http://books.google.de/books?id=Pm8mAAAACAAJ`.

Bibliography II

11. Jürgen Grothe and Winfried Kalfa. *Grundlagen der Betriebssysteme*, volume 2-1500 of *Lehrbriefe für das Hochschulfernstudium*. Dresden, Sachsen, Germany: Zentralstelle für das Hochschulfernstudium des Ministeriums für Hoch- und Fachschulwesen, 1988. URL `http://books.google.de/books?id=-62uPgAACAAJ`.
12. Scott Oaks and Henry Wong. *Java Threads*. The Java Series. Upper Saddle River, NJ, USA: Prentice Hall International Inc., Santa Clara, CA, USA: Sun Microsystems Press (SMP), and Reading, MA, USA: Addison-Wesley Professional, 3rd edition, 2004. ISBN 0596007825 and 9780596007829. URL `http://books.google.de/books?id=mB_92VqJbsMC`.
13. James Gosling, William Nelson Joy, Guy Lewis Steele Jr., and Gilad Bracha. *The Java™ Language Specification*. The Java Series. Upper Saddle River, NJ, USA: Prentice Hall International Inc., Santa Clara, CA, USA: Sun Microsystems Press (SMP), and Reading, MA, USA: Addison-Wesley Professional, 3rd edition, May 2005. ISBN 0-321-24678-0 and 978-0321246783. URL `http://java.sun.com/docs/books/jls/`.
14. Guido Krüger. *Handbuch der Java-Programmierung*. 4. aktualisierte edition. ISBN 3-8273-2361-4 and 3-8273-2447-5. URL `http://www.javabuch.de/`.
15. Christian Ullenboom. *Java ist auch eine Insel – Programmieren mit der Java Standard Edition Version 6*. Bonn, North Rhine-Westphalia, Germany: Galileo-Press, 6. aktualisierte und erweiterte edition, 2007. ISBN 3-89842-838-9 and 978-3-89842-838-5. URL `http://www.galileocomputing.de/openbook/javainsel6/`.
16. Santa Clara, CA, USA: Sun Microsystems, Inc. *Java™ 2 Platform Standard Edition 5.0 – API Specification*, October 19, 2010.
17. Timothy John Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*, volume 1945 of *Request for Comments (RFC)*. Network Working Group, May 1996. URL `http://tools.ietf.org/html/rfc1945`.
18. R. Fielding, J. Gettys, Jeffrey Mogul, H. Frystyk, L. Masinter, P. Leach, and Timothy John Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*, volume 2616 of *Request for Comments (RFC)*. Network Working Group, June 1999. URL `http://tools.ietf.org/html/rfc2616`.