Course TÖL401G: Stýrikerfi / Operating Systems 3. Processes

Mainly based on slides and figures subject of copyright by Silberschatz, Galvin and Gagne

Chapter Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using POSIX system calls that perform these operations.
- Describe and contrast interprocess communication using shared memory and message passing.
- Describe programs that use POSIX pipes and POSIX shared memory to perform interprocess communication.
- Describe client–server communication using sockets and including how to create client/server programs using the Java socket API.

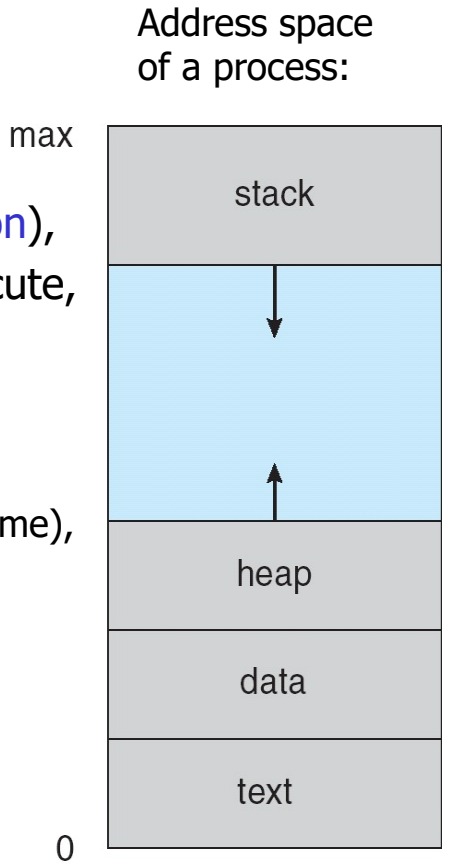
Contents

1. Process Concept
2. Process Scheduling
3. Operations on Processes
4. Interprocess Communication
5. Communication in Client-Server Systems
6. Summary

For users of any edition of Silberschatz et al.:
The section on Interprocess communication differs slightly from the book and contains differing material.

3.1 Process Concept

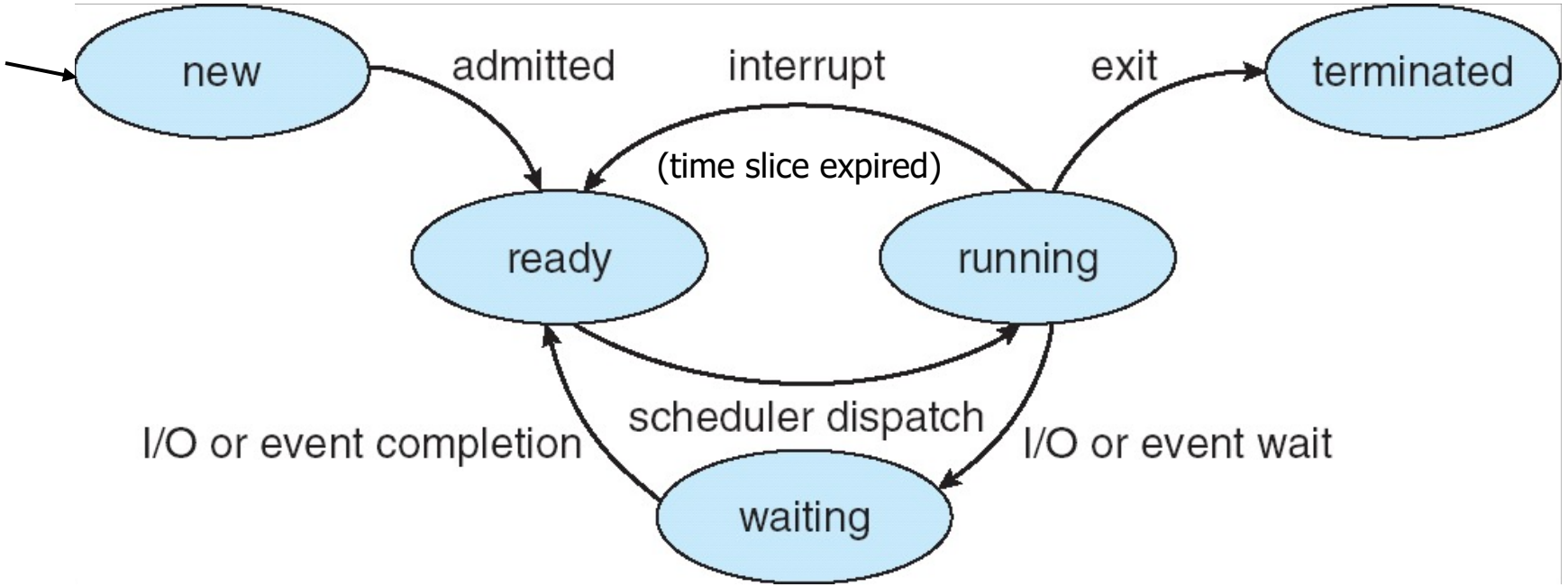
- Process: a program in execution.
  - A program is a passive entity:
    - E.g. a list of instructions stored on a hard disk.
  - A process is an active entity:
    - List of instructions loaded into main memory (text section),
    - Program counter (PC) specifying next instruction to execute,
    - Stack (for calling sub-routines or storing local variables),
    - Data section (for global variables)
    - Set of further associated resources, e.g.
      - Heap (memory dynamically allocated during process run time),
      - Opened files, …
- Synonyms for process:
  - Batch system world: job,
  - Time-shared systems: task.



Process State

- As a process executes, it changes its state:
  - New: The process is being created.
  - Running: Instructions are being executed.
    - On a single processor system, at most one process may be in state running. (On multi processor/multi core systems, each processor/core may execute a process.)
  - Waiting: The process cannot execute instructions, because it is waiting for some event to occur.
    - E.g. waiting for I/O completion.
  - Ready: The process is waiting to be assigned to a processor.
  - Terminated: The process has finished execution.
  - Note: depending on the OS, the name of states may vary (e.g. "blocked" instead of "waiting") or further more detailed states may exist.

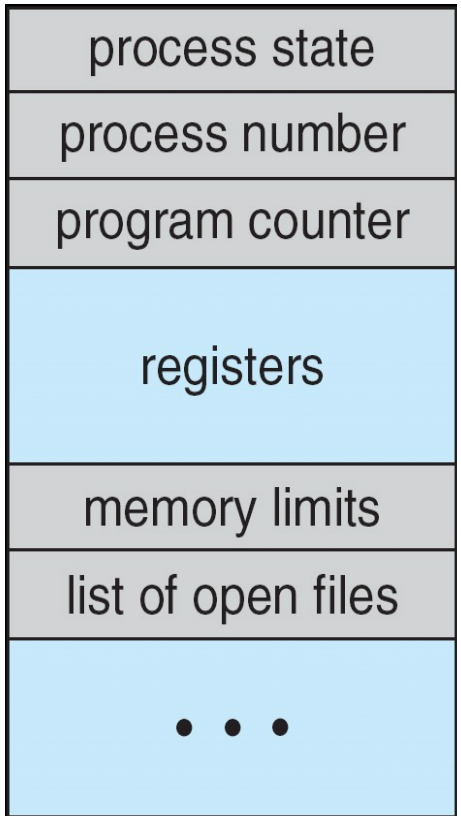
Process State Transition Diagram



- State changes from running to ready, if time slice expires.
  - I.e. interrupted by timer of scheduler in order to give CPU time to another process and therefore interrupts current running process.
  - Each process gets a "slice" of the CPU time (=time sharing →3.2).

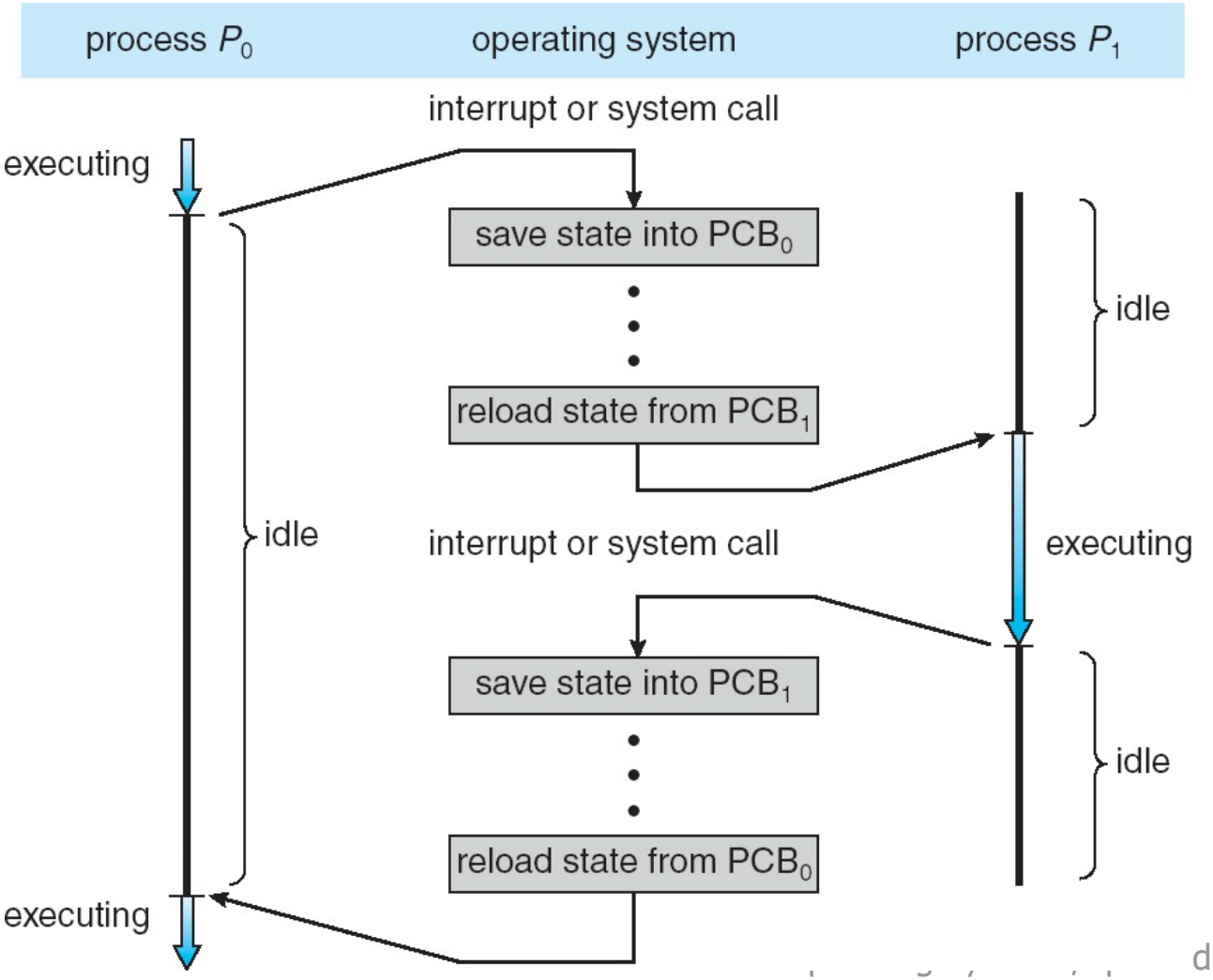
Process Control Block (PCB)

- State of process (and further information) stored by OS in process control block (PCB):
  - Process state,
  - Process number/Process Id,
  - Program counter and all other CPU registers,
    - Saved when interrupt occurs, restored when execution continues.
  - CPU scheduling information,
    - Process priority, pointer to scheduling queues, …
  - Memory-management information,
    - Allocated memory, address space (memory limits/ memory mapping) of process, …
  - Accounting information,
    - Used CPU time, …
  - I/O status information.
    - I/O devices reserved by process, opened files, …



- Program counter, CPU registers, and memory management information are called hardware context of a process.

CPU Switch From Process to Process (Context Switch)



Note: here, "save/reload state" refers not to the process state but to the hardware context.

(The process state changes of course as well as part of context switch, e.g. from "running" to "ready" and vice versa.)

3.2 Process Scheduling

- Process scheduler is responsible for switching between processes:
  - Multiprogramming (batch system, e.g. High-Performance Computing/Supercomputing):
    - Goal: maximise CPU (& other resource's) utilisation.
      - Switch to another process as soon as a process is waiting.
  - Time sharing (multi tasking system):
    - Goal: allow interaction between users and processes.
      - Switch frequently between processes (interrupt process even if it is not waiting). Each process gets a time slice of the CPU.
  - (More on scheduling algorithms in chapter 5…)

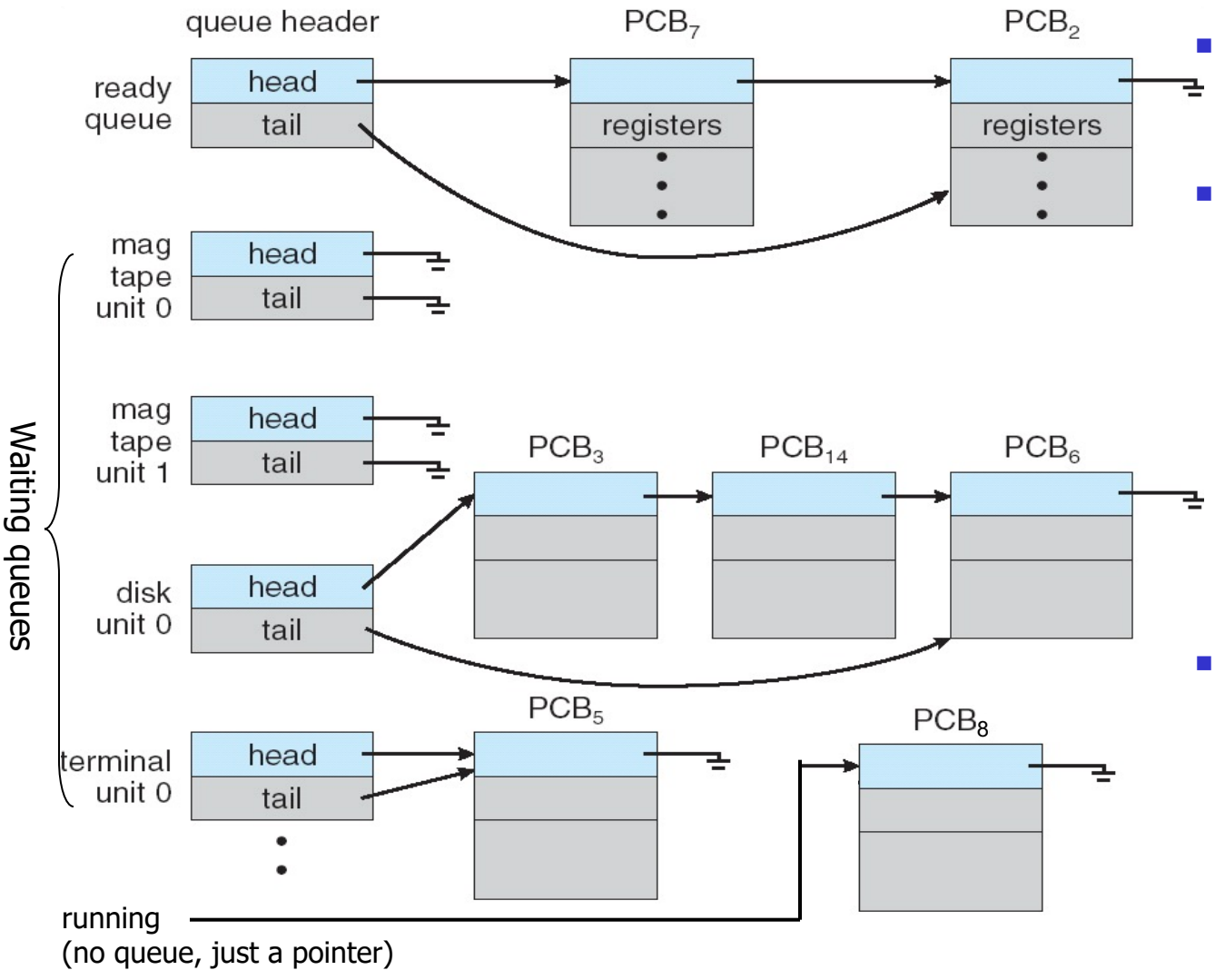
Scheduler Activations

- Process scheduler is activated when:
  1. Process gets blocked because it is waiting for some event, e.g. completion of I/O:
     - Transition from running to waiting state.
  2. Time slices expires, i.e. timer interrupt occurred (in case of timesharing system, i.e. preemptive):
     - Transition from running to ready state.
  3. I/O completes, i.e. interrupt of I/O device signal completion (preemptive):
     - Transition from waiting to ready.
  4. Process terminates (e.g. makes **`exit()`** system call or abnormal termination) (**`exit()`** is nonpreemptive, but abnormal termination may be preemptive):
     - Transition from running to terminated state/removal of process from system.

Process Scheduling Queues

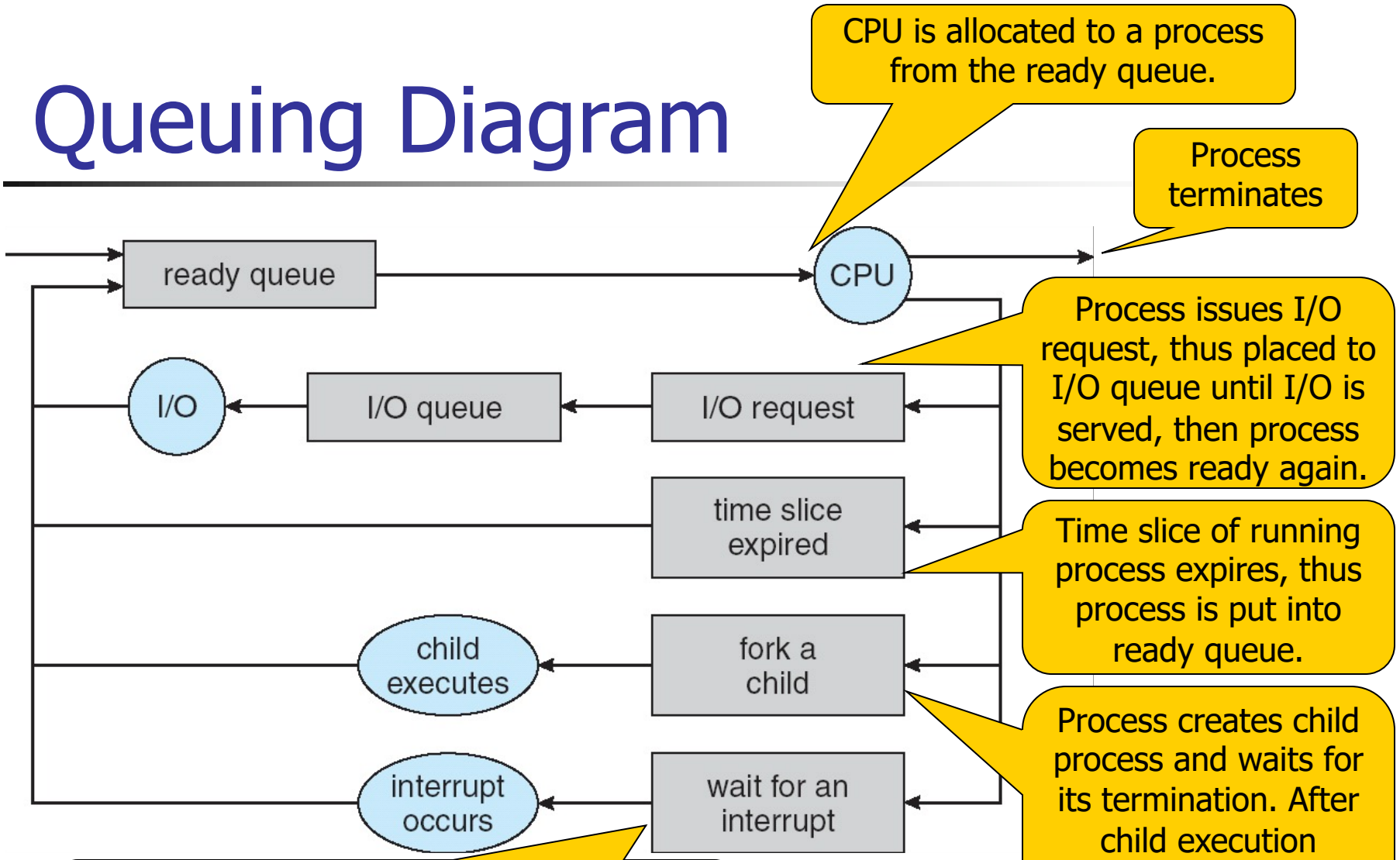
- When switching to another process, the process scheduler selects (dispatches) a ready process that will be executed by the CPU.
  - Scheduler needs to be able to find all ready processes.
- Scheduler maintains several queues for an efficient scheduling:
  - Job queue: set of all processes in the system.
    - As soon as a process enters the system it is added to job queue.
  - Ready queue: set of all processes in main memory, ready to execute.
  - Device queues: set of processes waiting for an I/O device.
    - Several processes may be waiting for the same I/O device: one queue for each device.
  - Processes migrate among the various queues depending on whether they are ready or waiting.
- Queues are implemented as a linked list data structure:
  - Queue header is pointing to PCB of first (and final) process in the queue.
  - Each PCB contains a pointer to the next process in the particular queue.
  - (See next slide for example...)

Ready Queue, Various I/O Device Queues, Pointer to Running Process



- Queues allow scheduler to manage ready and waiting processes.
- Linked list data structure instead of an array data structure allows to move a process from one queue to another without copying PCBs, but just by adjusting pointers.
- In case of multi-processor/-core, each processor/core has a pointer pointing to the PCB of the currently running process.

Queuing Diagram

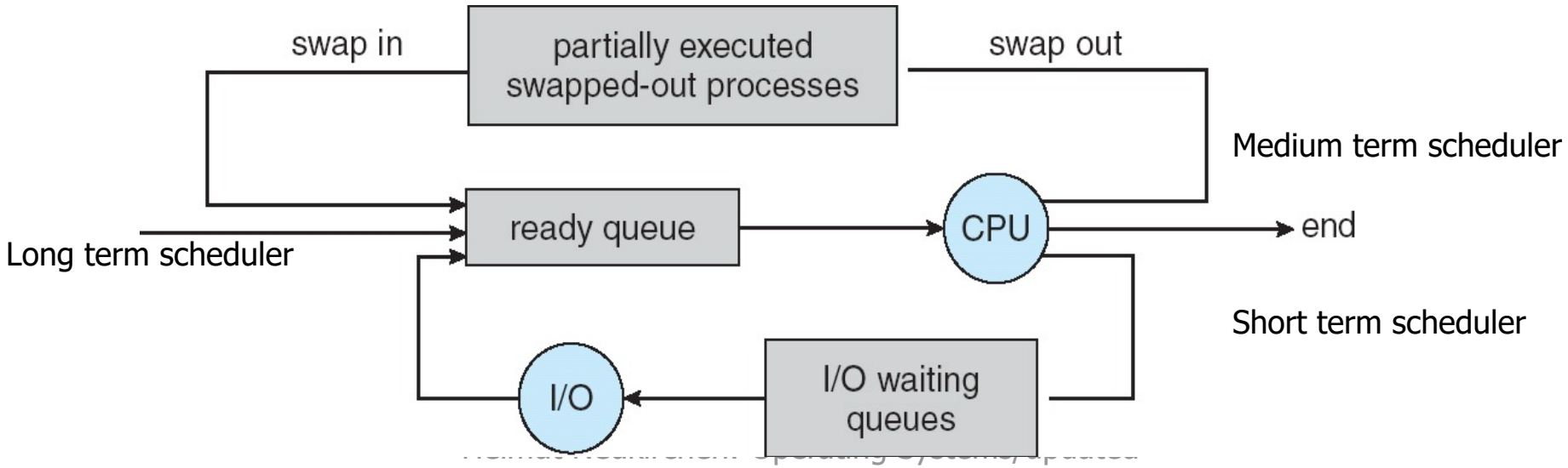


Batch System Schedulers

- In multi tasking systems, all processes started by a user are kept in memory at the same time.
  - Usually only one scheduler (the CPU scheduler).
- However, in batch systems only a selected number (=degree of multiprogramming) of the submitted batch jobs are loaded into memory and started as process. (Once a process terminates, a new batch job is loaded into memory and started.) Two schedulers:
  - Long-term scheduler (or job scheduler): selects which processes from the submitted batch jobs should be loaded into memory (and thus brought into the ready queue).
  - Short-term scheduler (or CPU scheduler): selects which process from the ready queue should be executed next and get the CPU.

Medium Term Scheduling/Swapping

- Sometimes, degree of multiprogramming needs to be temporarily reduced (e.g. if processes require a lot of memory).
  - A medium term scheduler may be used to identify processes that should not be considered by short-term scheduler for some time:
    - Processes are removed from main memory to hard disk (swapped out) and later-on, e.g. if another process terminates, the swapped-out state is put into main memory again (swapped in).



Process Schedulers

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ must be fast.
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ may be slow.
- The long-term (/medium term) scheduler controls the degree of multi-programming (how many program are executed at the same time).
- Processes can be described as either:
  - I/O-bound process: spends more time doing I/O than computations, many short CPU bursts.
  - CPU-bound process: spends more time doing computations; few, but very long CPU bursts.
  - Long-term scheduler should try to achieve a good process mix of I/O and CPU-bound processes to utilise both CPU and I/O devices.
    - E.g.: submitters of batch jobs need to classify their jobs as either I/O- or CPU-bound

Context Switch

- When CPU switches to another process, the system must save the (hardware) state of the old process (also: interrupt processing) and load the saved (hardware) state for the new process: Context switch.
  - Status of CPU (program counter and all other CPU registers),
  - Status of memory controller hardware (memory limits/memory mapping).
  - (Other information from the PCB usually needs not to be saved/restored as part of context switch, because it refer to states where the PCB is anyway the primary location for maintaining and storing them, e.g. process Id.)
- Time needed for context-switch is overhead.
  - Time slices between 10ms and 100ms mean between 100 and 10 context switches per second.
  - Usually, a context switch takes between 3µs (=0.003ms) and 1ms (depending on scenarios, OS and hardware: nowadays typically 30µs).
    - http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html
  - Overhead may be reduced by using threads (switches only CPU status, but not memory controller hardware status). →Chapter 4

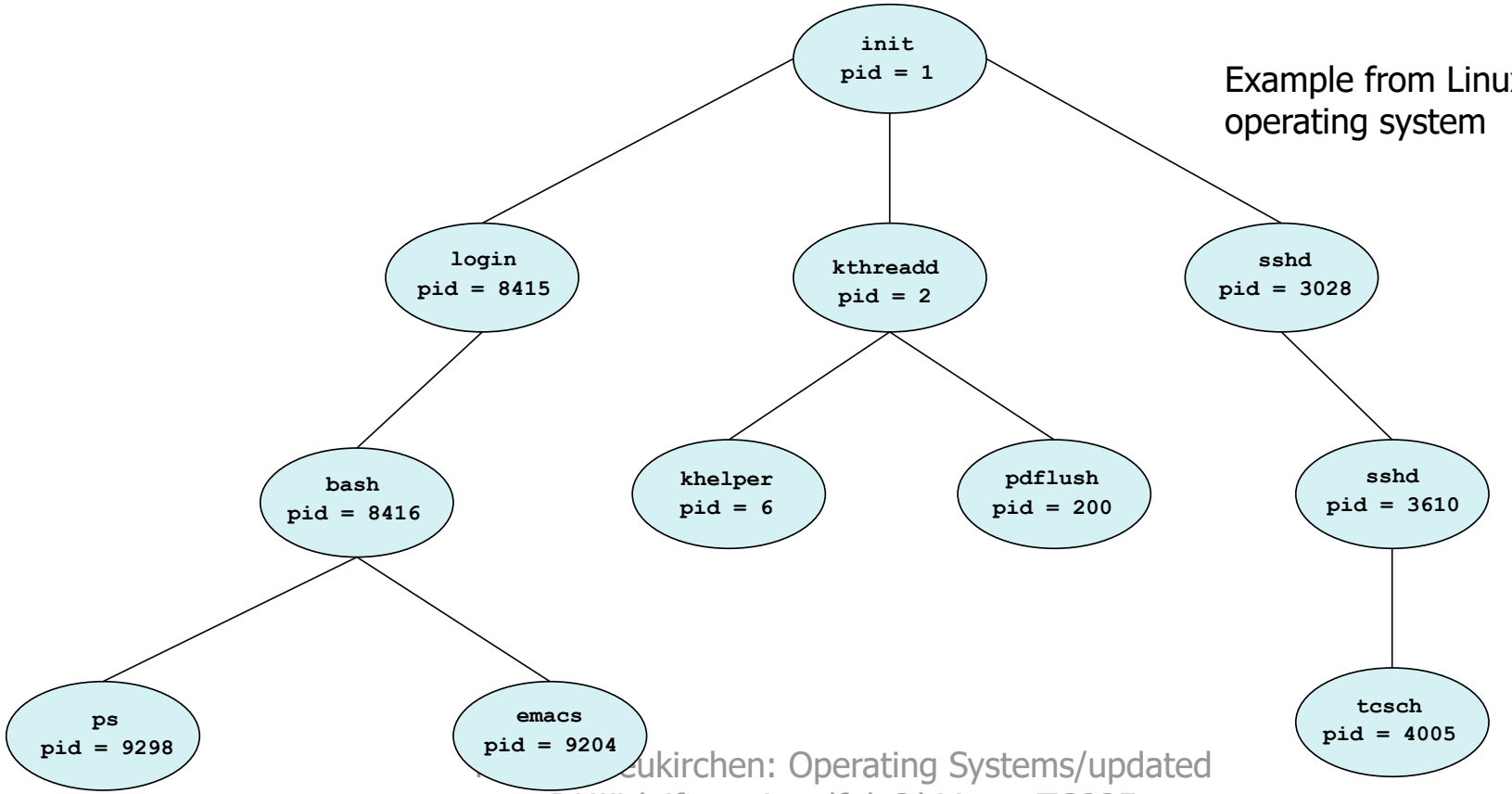
3.3 Operations on Processes

- Operating system must include support to create and terminate processes dynamically.
- To identify a process, each process has a process identifier (PId).
  - PId is just an integer number.
  - Usually, the first process created by the operating system ("init" process) has PId 1, all further processes get increasing PIds.
- Usually, there is a parent-child relationship between processes.

I.Hjörleifsson ingolfuh@hi.is st. TG225

Process Creation: Tree of processes

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.

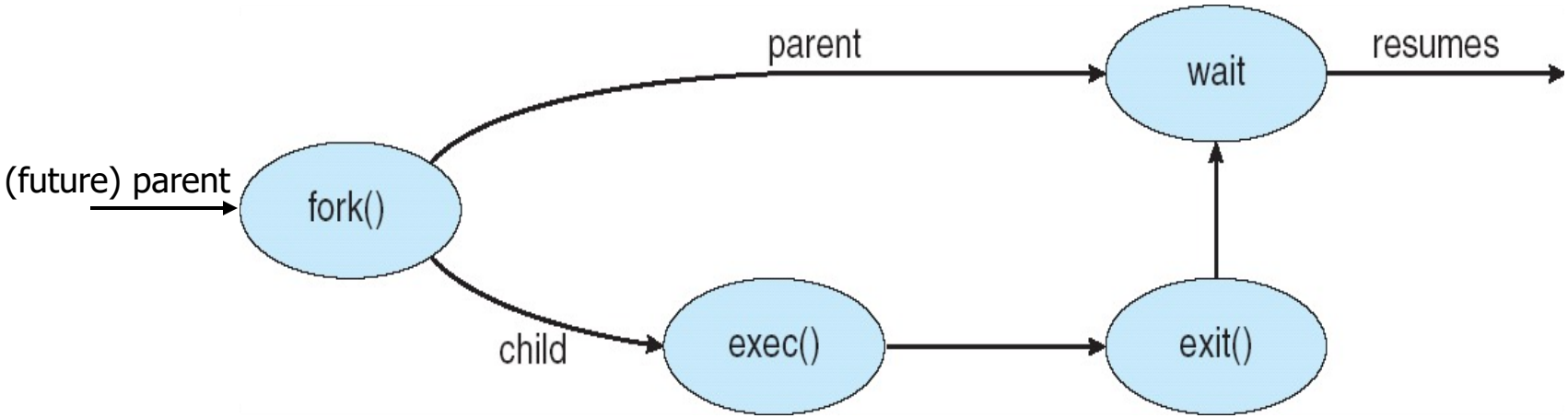


Process Creation: Design decisions

- Several possibilities concerning sharing of resources, execution, and address space:
- Resource sharing:
  - Parent and children share all resources or
  - Children share subset of parent's resources or
  - Parent and child share no resources.
- Execution:
  - Parent and children execute concurrently or
  - Parent waits until children terminate.
- Address space:
  - Child address space (instructions, data, heap, stack) duplicate of parent or
  - Child has a new program loaded into it.

Process Creation: POSIX

- Approach (chosen design decision) depends on operating system.
- POSIX example:
  - `fork` system call creates new process as a copy of parent process.
  - `exec` system call used after a fork to replace the process' memory space with a new program.



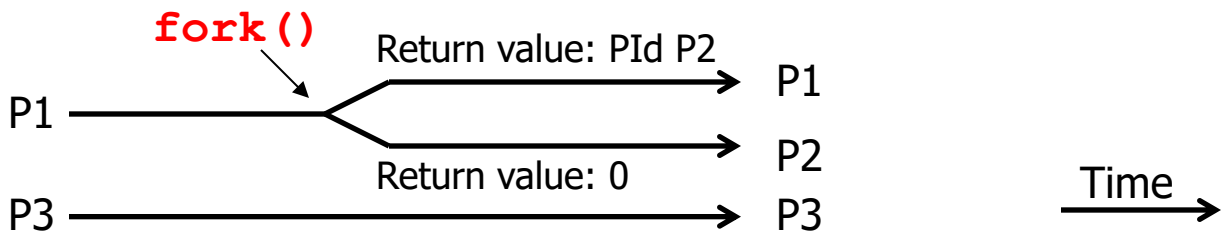
Process Termination: POSIX

- Process executes last statement and voluntarily requests from operating system to be deleted (**`exit`**).
  - Return data from child to parent (return value provided by **`exit`** system call – may be retrieved by parent via **`wait`**).
  - Process' resources are deallocated by operating system.
- Parent may terminate execution of children processes (**`kill`**), e.g. because
  - child has exceeded allocated resources,
  - task assigned to child is no longer required,
  - parent is exiting.
    - Some operating system do not allow child to continue if its parent terminates. (However, POSIX does.)
      - In that case, all children would be automatically terminated if parent terminates (cascading termination).

POSIX System Calls for Process Creation in Detail (1)

- POSIX `fork()` system call for creating a new process:
  - New process (child) is an exact copy of the creating process (parent), but has a new PId.

  ⇒ Child and Parent execute the same instructions after the **`fork()`**.
  - Return value of **`fork()`** system call differs for parent and child: based on this, a process can identify whether it is the child (return value: 0) or the parent (return value: PId of Child).

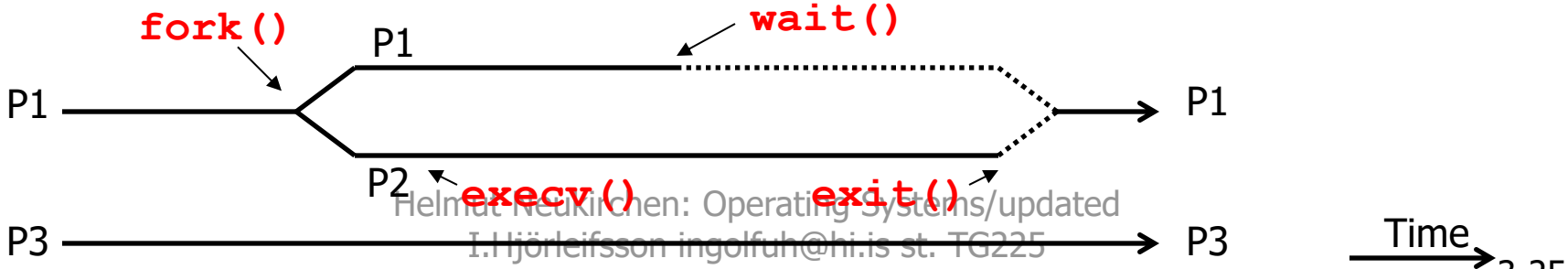


POSIX System Calls for Process Creation in Detail (2)

- "Exact copy" of processes when **`fork`**ing means:
  - Copy of instructions (text section), data, stack, heap.
  - Partial copy of PCB, e.g.:
    - CPU registers,
    - Opened files.
      - E.g. for communication between parent and child using pipe files (→3.5).
  - Other parts of PCB are not copied but adjusted, e.g.:
    - New Process Identifier (PId) for child,
    - Address space (child process is a copy: address space is copied into another part of the RAM)
- If it is not intended that child process executes after a **`fork()`** the same instructions as the parent:
  - (Child-)Process may use one of the **`exec()`** system calls (e.g. **`execv()`**) to replace its instructions and data with those from a programme file.
    - E.g. used by a shell to start other programs.

POSIX System Calls for Process Termination in Detail (1)

- Possible process terminations:
  - Normal termination using `exit()` system call (intentionally by client),
  - Termination to indicate an error using `exit()` system call (intentionally by client),
  - Termination due to a severe process error, e.g. division by 0 (by operating system),
  - Termination using `kill()` system call (by another process).
- Parent may wait for the termination of child process using `wait()` system call.
  - The status code provided by child process when calling `exit()` is passed back to parent as return value: this status code is stored in a table of processes.
    - If parent has not yet called `wait()`, it might do so in future and process table entry for child (storing the status code) needs to be kept: child process is during that time in process state zombie until `wait()` is called.
    - If parent process terminates without calling `wait()`, the init process (PId=1) becomes parent of that child. (Init calls anyway periodically `wait()` thus removing any zombies.)



POSIX System Calls for Process Termination in Detail (2)

- Usage of **`kill()`** system call in detail:
  - **`kill (pid, SIGKILL)`**
    sends signal **`SIGKILL`** to process **`pid`** , thus terminating it.

  - Signals send via **`kill()`** do not necessarily kill a process. E.g.:
    - **`SIGSTOP`** puts a process to sleep,
    - **`SIGCONT`** awakes process again.

- Operating system considers protection:
  - A process of user **`A`** is not allowed to kill process of user **`B`**.

Example of using POSIX system calls: fork(), exit(), execv(), wait()

- C program using fork(), etc.
  - (Java does only support creating threads, but not forking processes.)

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int err, status;

if (fork() == 0) {
   /* Child process */
   err = execv("/bin/program", NULL);
   exit(err);
}
else {
   /* Parent process */
   wait(&status);
   ...
}
```

Create new process

Replace instruction and data of process with those of program in file **/bin/program**

This line of code should never get executed, except that **execv()** fails, e.g. because file does not exist.

Wait for child to terminate. **status** will contain the parameter passed by child to **exit()**.

Multiprocess applications: Web browser example

- In the past, many web browsers ran as single process:
  - If one of the web sites (in a tab) causes trouble, entire browser can hang or crash.
- Nowadays, Webbrowsers are multiprocess, e.g. Google Chrome, with three different types of processes:
  - One main Browser process manages user interface, disk and network I/O, and creates/coordinates the other processes.
  - For each web page (e.g. in a tab): Renderer process for each web pages dealing with HTML and Javascript.
  - For each plug-in (e.g. Flash): Plug-in process running the plug-in code.
  - Separate Renderer and Plug-in processes: provide sandbox, minimizing effect of security exploits (problem occurs only in that process).
- Interprocess communication needed between processes (e.g. HTML to be rendered to Renderer process) → next section.

3.4 Interprocess Communication (IPC)

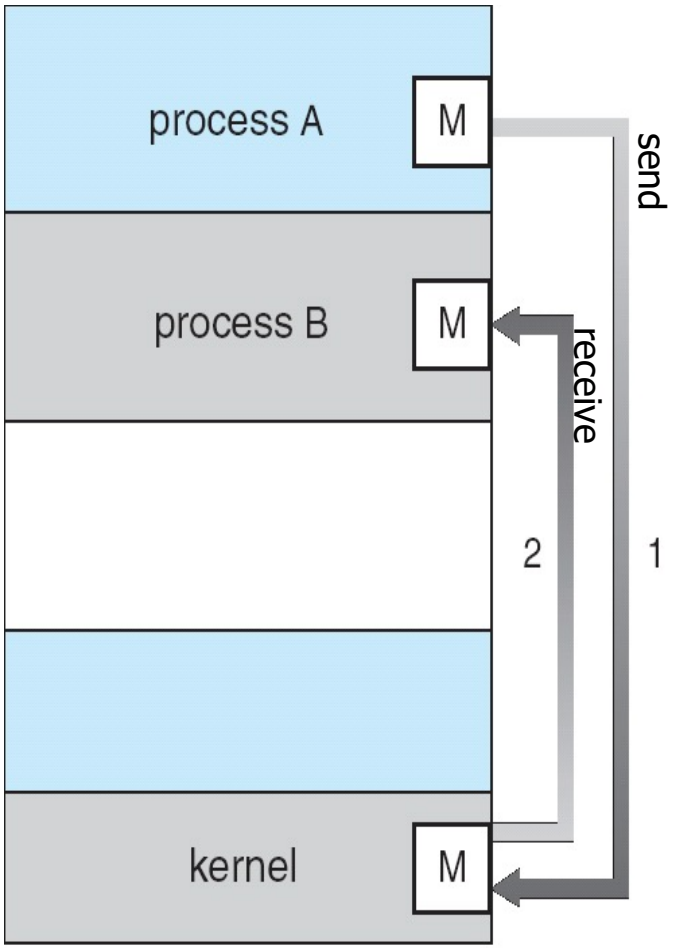
- Concurrently running processes may be independent or cooperating.
- An independent process is not related to any other process.
  - Does not communicate or share data with any other processes.
- A cooperating process communicates or shares data with other processes (directly, not via files).
  - Reasons for process cooperation:
    - Information sharing: several users work together on the same data.
    - Computation speed-up: using parallel processing on a multi-processor/-core system.
    - Modularity: as, e.g., used in microkernel OS design.
    - Convenience: e.g. Compiler process translates source code to textual assembly instructions, assembler process takes them and converts them to machine code.
  - Process cooperation requires mechanisms for interprocess communication (IPC) and synchronisation of actions (process synchronisation).
    - IPC will be discussed in this and the following section.
    - Process synchronisation will be treated later in a chapter on its own (ch. 6).

Two fundamental models of IPC: Message Passing vs. Shared Memory (1)

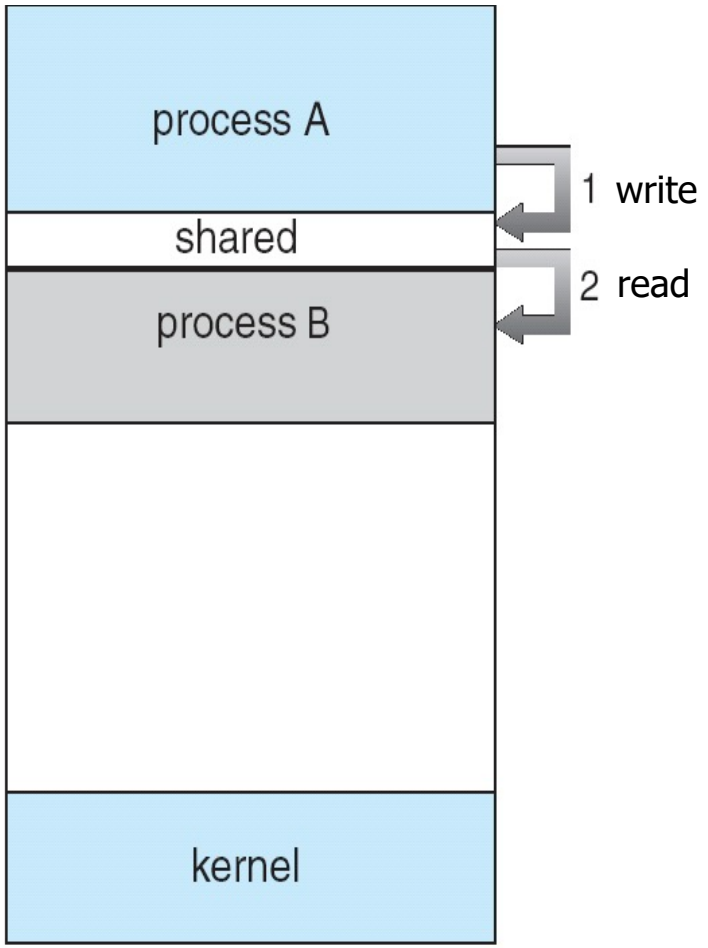
- For communication, processes need to exchange data:
  - Either using shared memory that is requested by the communicating processes from the operating system via system calls.
    - Once the shared memory area has been established, ordinary (=fast) memory accesses can be used to read and write shared data without further OS support.
    - Processes need to synchronise access to shared memory: suitable infrastructure needs to be implemented by processes.
    - Preferable if huge chunks of data need to be exchanged.
  - Or using message passing facilities of operating system:
    - OS has an internal buffer that can be accessed by different processes via send & receive system calls to exchange data.
    - Operating system provides communication infrastructure and thus synchronises access to exchanged data.
    - Preferable if only smaller chunks of data need to be exchanged.

Message Passing vs. Shared Memory (2)

Message passing:



Shared memory:



Shared-Memory Systems, e.g. POSIX

- Usually, an OS prevents two processes from accessing each other's memory. System calls are needed to remove memory protection and share memory between processes.
- POSIX system calls for shared memory, e.g.:
  - `int shmget (long key, int size, int flag)`: get an Id (will be the return value) for a shared memory area using a **`key`** (on which communicating processes have to agree on – e.g. using message passing).
  - `void* shmat (int id, char* addr, int flag)`: map shared memory area to process memory at **`addr`** using the **`id`** that identifies the previously created shared memory area.
- (We will have a look in later chapters on how an OS does realise protected and shared memory and on how to synchronise access to a shared buffer.)
- While communication via shared memory is fast, it requires that the communicating processes reside on the same machine.

Message-Passing Systems

- Mechanism provided by OS to allow processes to communicate and to synchronize their actions without using shared variables/shared memory: Messages are exchanged between processes.
- IPC provides two operations to processes that want to communicate:
  - **send**(*message*)
  - **receive**(*message*)
- If processes *P* and *Q* wish to communicate, they need to:
  1. Establish a communication link between them.
     - Communicating processes on two different machines: network connection.
     - Communicating processes on the same machines: usually a buffer of the operating system.
       - However, this buffer is no shared memory between processes that can directly accessed by the processes. (Rather, only send & receive can be used to add and retrieve data from the head and tail of the buffer.)
  2. Exchange messages via the communication link using send & receive.

Design Decisions when Implementing an OS with Message Passing IPC

- Communication:
  - direct (processes need to know each other) vs.
  - indirect communication (processes communicate via a well-known mailbox):
- Communication:
  - synchronous (send & receive may block if buffer full or empty respectively) vs.
  - asynchronous (non-blocking send & receive).
- Buffer size:
  - no buffer,
  - automatic buffering (either bounded or unbounded).
- Further details on following slides…

Direct vs. Indirect Communication: Direct communication

- 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.
  - Disadvantage: processes must know the PId of the other process.
    - In practice, e.g. a server process may write its PId to a file with a well-known file name. All client processes may then retrieve PId of server process from that file and send message to server.

- 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.
    - Not possible to have multiple independent links between a pair. (Because no further information than the PId would be available to distinguish links.)
  - The link may be unidirectional, but is usually bi-directional.

Direct vs. Indirect Communication: Indirect Communication (1)

- Indirect communication: Messages are directed and received from mailboxes (also referred to as ports):
  - **send**(*A, message*): send a message to mailbox A
  - **receive**(*A, message*): receive a message from mailbox A
  - Advantage: processes only need to agree on Id of mailbox.
    - Mailbox Ids must be well-known to avoid that different application accidentally use the same Id. Mailbox must be created first (→next slide)

- Properties of communication link:
  - Link established only if processes decide to share a common mailbox.
  - A link/mailbox may be associated with many processes.
  - Each pair of processes may share several communication links (by using multiple mailboxes at the same time).
  - Link may be unidirectional or bi-directional.

Direct vs. Indirect Communication: Indirect Communication (2)

- For supporting indirect communication, an OS must provide the following operations:
  - Create a new mailbox using a certain Id,
  - Send and Receive messages through a mailbox,
  - Destroy a mailbox.

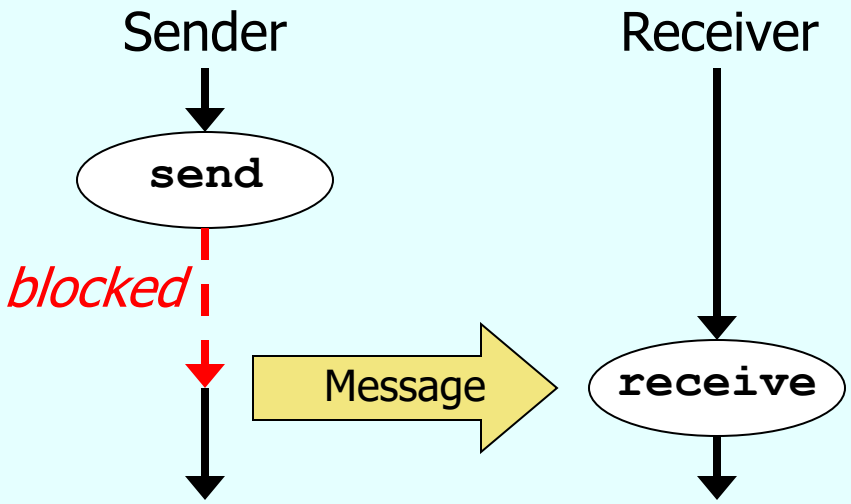
Synchronization: Blocking vs. Non-Blocking Send & Receive

- Send & receive operations may be blocking or non-blocking:
  - Blocking:
    - Blocking send: sender is blocked until the message is received by the receiving process or put by the OS into some buffer/mailbox.
    - Blocking receive: a process that issues a receive operation is blocked until a message is available.
  - Non-blocking:
    - Non-blocking send: sender continues immediately after sending.
    - Non-blocking receive: receiver receives a valid message or gets "null" (or some error return value) if no message is available.

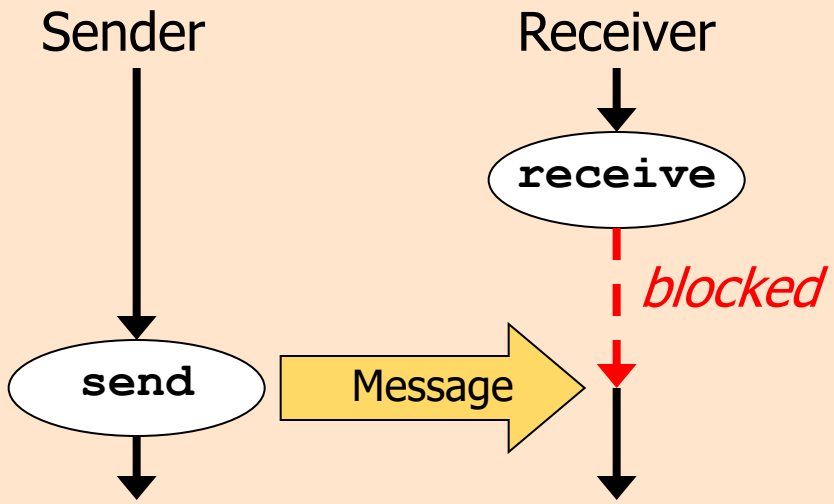
(More examples on next slides…)

Synchronization: Blocking Send & Receive

- Blocking Send
  - Sender waits/blocks until receiver (or buffer) is ready to receive.

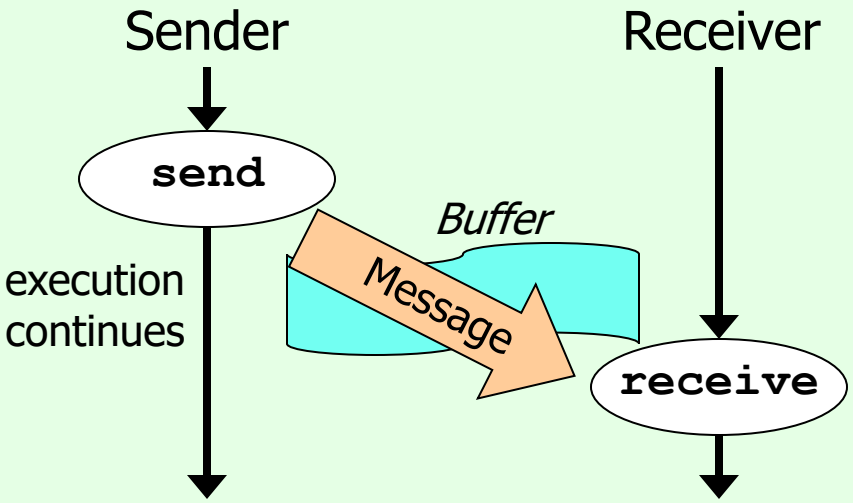


- Blocking Receive
  - Receiver waits/blocks until Sender sends a message.

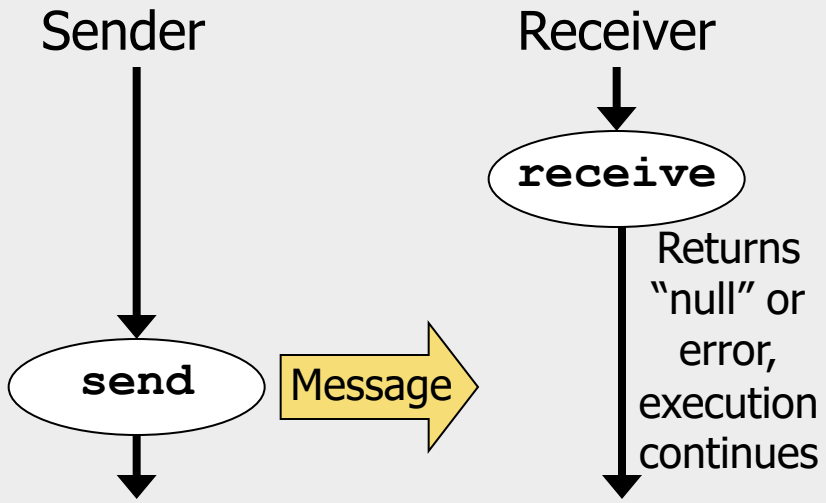


Synchronization: Non-Blocking Send & Receive

- Non-Blocking Send
  - Sender does not wait for receiver. Message is stored in buffer or (in case of no buffering): message gets lost if receiver is not ready.



- Non-Blocking Receive
  - Receiver does not wait for message. If no message is available, receive operation returns "null" or an error.



Buffering: Buffer size

- Buffer queue provided by OS for each communication link.
- Possible buffer capacities:
  1. Zero capacity: 0 messages buffer size, i.e. no buffer at all.
     - If blocking send is used, sender must wait for receiver until receiver retrieves/receives the message.
  2. Bounded capacity: finite length of *n* messages.
     - If blocking send is used, sender must wait if communication link (or its buffer respectively) capacity exceeded.
  3. Unbounded capacity: infinite length.
     - Even if blocking send is used, sender never hast to wait.

Synchronous vs. Asynchronous communication

- Synchronous communication:
  - Close temporal coupling between send and reception events.
  - E.g. telephone: receiver hears immediately what sender says.
  - Can be used for synchronisation between both sender and receiver.
- Asynchronous communication:
  - No close temporal coupling send and reception events.
  - E.g. mail: it may take some time until receiver gets what sender posted.
  - Only partially suitable for synchronisation between sender and receiver:
    - Receiver does not know exactly when message was sent.
    - Sender does not know when receiver actually receives message.

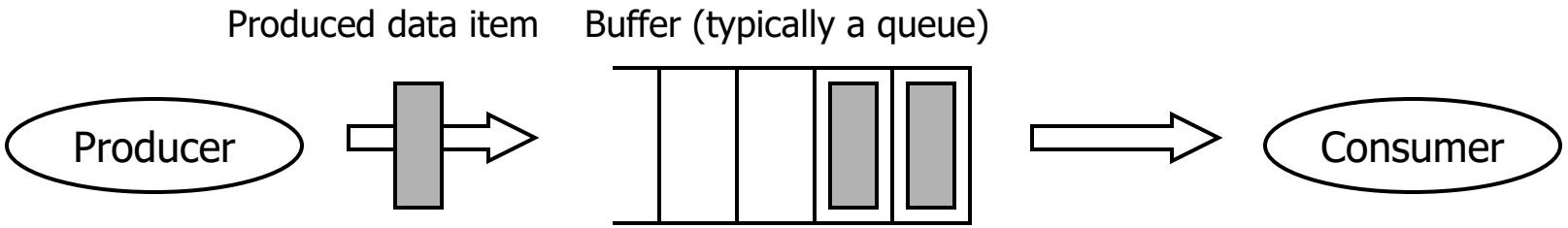
Synchronization of Processes

- Rendezvous communication:
  - Blocking send und blocking Receive with zero capacity buffer.
  - Often synonymously called synchronous communication, because can be used for synchronisation of processes:
    - Sender only continues when receiver received message.
    - Receiver only continues when sender sent message.

- By default, most message passing communication mechanisms use non-blocking send (or blocking send with a buffer) and blocking receive.
  - Often, this is called asynchronous communication. Can only be used to take care that receiver does not start an action before sender does.
    - Receiver only knows that sender must have sent message at any time in the past. Sender knows nothing about the reception.

Producer-Consumer Problem

- A classical example to illustrate the usage/study the applicability of an interprocess communication/synchronisation mechanism:
- Concurrent producer and consumer processes share a common buffer.
  - Producer:
    - Puts (inserts) produced data items into buffer until it is full.
    - Waits if buffer is full.
    - Resumes when buffer is not full any more.
  - Consumer:
    - Consumes (removes) data items from buffer.
    - Waits if buffer is empty (no items available).
    - Resumes when new items are available.



- Example: Output of **`dir`** command is piped as input to **`more`** command.

Variants of the Producer-Consumer Problem

- There are two flavours of the producer-consumer problem:
  - Unbounded-buffer: Buffer is infinitely large (i.e. producer never has to wait).
    - Note: Computers are finite machines with in particular a finite amount of memory, hence an infinitely large buffer cannot be implemented. However, "unbounded-buffer" simply means that the buffer is large enough for all considered cases. (E.g. a Java vector that can grow instead of an array of fixed size.)
  - Bounded-buffer: assumes that there is a fixed buffer size (i.e. a producer may have to wait if buffer is full).

Solution of Producer-Consumer Problem Using Message Passing

- Assumption:
  - non-blocking send with unbounded buffer, non-blocking receive.
    - (Other solutions with other combinations of blocking/non-blocking possible, too.)
- Excerpt of a possible Java solution:
  - Producer process:

```
Mailbox mbox = new Mailbox("ProducerConsumerMailbox");

while (true) {
  Data message = new Data();
  mbox.send(message);
}
```

  - Consumer process:

```
Mailbox mbox = new Mailbox("ProducerConsumerMailbox");

while (true) {
  Data message = (Data) mbox.receive();
  if (message!=null) {
    // do something with consumed message
  }
}
```

Infinite buffer renders problem trivial (sender never has to wait)!

Use Mailbox with string as ID.

Produce data.

Put data into unbounded buffer.

Use Mailbox with string as ID.

Non-blocking receive

If receive did return data, otherwise ignore.

Solution of Producer-Consumer Problem Using Message Passing

- The Java code on the previous slide assumes that some class **`Mailbox`** is available (e.g. provided by the Operating System).
  - A possible implementation of class **`Mailbox`** could look like shown below (code of constructor for creating/using a mailbox with some ID not shown):

```
public class Mailbox {
  private Vector queue; // Unbounded buffer

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

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

3.5 Communication in Client-Server Systems

- Scope of different interprocess communication mechanisms:
  - Shared memory: same machine.
    - Based on sharing physical memory.
  - Message passing: same machine or different machines connected via a network possible, but same type of operating system required.
    - Based on message passing mechanism that is specifically implemented by each operating system (may be different on each OS).
- How to communicate between processes running on different machines having different operating systems?
  - Standardised communication mechanisms required, e.g. the Internet Protocol (IP).
    - Heavily used in client-server systems:
    - Server, e.g. running on Unix machine, offers service (e.g. HTTP server delivering WWW pages).
    - Client, e.g. running on an MS Windows machine accesses service (e.g. via a web browser).

Sockets

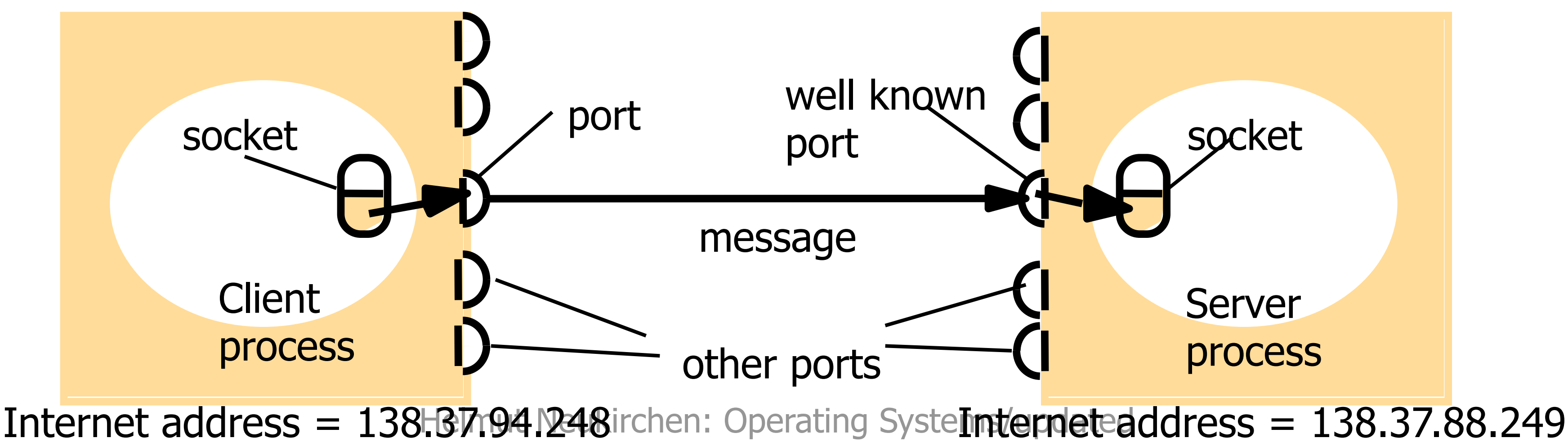
- A generic programming model for non-local and local interprocess communication. Available on all major operating systems:
  - Allows to write source code that runs a many operating systems.
  - Usually provided by the kernel.
    - (But may also be provided by a library on top of the kernel.)
- Sockets are endpoints of communication.
  - Communication takes place between a pair of sockets.
  - The operating system binds the sockets of the sending and receiving process to a port and address. (→slides after next slide)
    - (Comparable to indirect communication using mailboxes).
  - Different communication styles supported:
    - Datagram type (connection-less unreliable) and Stream type (connection-oriented reliable). (→next slide)

Excursion: Reliable Connection-oriented vs. Unreliable Connection-less Communication

- Unreliable, connection-less communication:
  - A single message is immediately sent from one process to another (datagram). (Datagram has maximum size, e.g. ≈64KB)
    - Like sending a letter/telegram: sender & intended receiver do not know whether message was lost (e.g. due to network problems) or not.
    - Fast: no overhead involved.
    - Example: User Datagram Protocol (UDP) from the IP family.
- Reliable, connection-oriented communication:
  - First, a connection needs to be set up. Using this, an endless stream of data can be transmitted between a pair of processes.
    - Connection is reliable, because reception of transmitted data needs to be acknowledged by receiver. If sender does not get an acknowledgement, data is retransmitted.
    - Slower: Connection set-up and acknowledgment significant overhead.
    - Example: Transmission Control Protocol (TCP) from the IP family.

Sockets, addresses and ports

- Each IP address identifies a machine on the Internet.
  - IPv4 address: 4 x 8 Bits, IPv6: 16 x 8 Bits.
- For making a server process reachable, server needs to listen at a port.
  - Clients contact the agreed well known port number at the server's IP address.
- To enable the server to reply to each client, the client socket has also a port number.
  - Client port number is dynamically assigned by the operating system.
- A process may use several sockets (& ports) at the same time.



More about port numbers

- Port numbers: 16 Bit integer, i.e. 65536 different ports available per machine.
- Well Known Port Numbers are used for standard services.
  - Example: HTTP server process typically binds port 80 to its socket.
    - All client processes will use as well port 80 as target address for contacting that HTTP server process.
  - Well Known Port Numbers are in the range [0,1023].
    - An operating system does only allow processes running with superuser privileges to bind their socket to a port number <1024.
      - Prevents that any ordinary (malicious) user can start a process that claims to be an "official" service on that machine.
- The range [1024, 49151] ("registered ports") can be used by server processes started by ordinary users.
- Dynamically assigned port number of client sockets are in the range [49152, 65535].

Java API for Socket Communication Port numbers and IP addresses

- Java Application Programming Interface (API) supports interprocess communication based on sockets.
- How to to specify IP address and Port numbers in the Java world:
  - Port numbers are represented by Integers: Java built-in type **`int`**
  - IP Addresses have an own data type: class **`InetAddress`** from package **`java.net.InetAddress`** (IPv4 and IPv6 supported).
  - For translating domain names (e.g. **`www.hi.is`**) into IP addresses using the Domain Names Service (DNS), the static method **`getByName(String host) throws UnknownHostException`** can be applied to type **`InetAddress`**:

```
import java.net.InetAddress;
import java.net.UnknownHostException;
public class InetAddressDemo {
  public static void main(String[] args) {
    try {
      InetAddress anInetAddress = InetAddress.getByName("www.hi.is");
      System.out.println(anInetAddress.getHostAddress());
    } catch (UnknownHostException e) {
    }
  }
}
```

Java API for Connection-less Sockets

- Class **`DatagramSocket`** from package **`java.net.DatagramSocket`** represents a socket of datagram type.
  - Once a socket has been constructed, the port number stays fixed.
  - The same socket can be used for sending and receiving via the assigned port.

- Constructors and Methods:
  - **`DatagramSocket`**`()`: creates a socket on local machine and assigns free port to it.
  - **`DatagramSocket`**`(int port)`: creates a socket that is bound to local port **`port`**.
  - `void` **`send`**`(DatagramPacket dp) throws IOException`: sends datagram packet **`dp`** that must have set a destination IP address and destination port.
  - `void` **`receive`**`(DatagramPacket dp) throws IOException`: receives datagram packet and stores it in **`dp`** that provides space for storing the message. IP address and port of the originating socket can be retrieved from **`dp`** afterwards.
  - `void` **`close`**`()`: close socket.
- (More about datagram packets on next slide…)

Java API for Packets Transmitted via a Connection-less Socket

- Class **`DatagramPacket`** from package **`java.net.DatagramPacket`** represents a datagram packet.
  - May be used for sending and receiving via a **`DatagramSocket`**.
- Constructors and Methods:
  - **`DatagramPacket`**`(byte[] buf, int length)`: creates a packet that can be used for receiving. **`buf`** must provide space for received message. If received message is longer than **`length`**, it will be truncated. Origin of message can be retrieved after reception (see getters below).
  - **`DatagramPacket`**`(byte[] buf, int length, InetAddress address, int port)`: creates a packet that can be used for sending to IP address **`address`** and port **`port`**.
  - `byte[]` **`getData`**`()`: returns the contents of the message.
  - `int` **`getLength`**`()`: returns the length of the message.
  - `InetAddress` **`getAddress`**`()`: returns IP address to which datagram is being sent or from which the datagram was received.
  - `int` **`getPort`**`()`: returns the port number to which datagram is being sent or from which the datagram was received.

Java Example for Connection-less Server Process

```
import java.net.*;
import java.io.*;

public class ConnectionLessServer {
  public static void main(String args[]) {
    DatagramSocket aSocket = null;
      try {
      aSocket = new DatagramSocket(6789); // create socket at agreed port
      byte[] buffer = new byte[1000];
      while (true) {
        DatagramPacket request = new DatagramPacket(buffer, buffer.length);
        aSocket.receive(request);
        DatagramPacket reply = new DatagramPacket(request.getData(), request.getLength(),
                                                  request.getAddress(), request.getPort());
        aSocket.send(reply);
      }
    } catch (SocketException e) {
      System.out.println("Socket: " + e.getMessage());
    } catch (IOException e) {
      System.out.println("IO: " + e.getMessage());
    } finally {
      if (aSocket != null)
        aSocket.close();
    }
  }
}
```

- Server just sends the received message back to the same IP address and port from where it has been received.

Server runs in an endless loop. (Abort with e.g. CTRL-C)

Needs to be changed if already used by some other running server (Error "Socket: Address already in use").

Java Example for Connection-less Client Process

```
import java.io.*;
import java.net.*;
public class ConnectionLessClient {
  public static void main(String args[]) {
    // args[0]: message contents, args[1]: destination hostname
    DatagramSocket aSocket = null;
    try {
      aSocket = new DatagramSocket();
      byte[] message = args[0].getBytes();
      InetAddress aHost = InetAddress.getByName(args[1]);
      int serverPort = 6789; // agreed port
      DatagramPacket request = new DatagramPacket(message, message.length,
                                                  aHost, serverPort);
      aSocket.send(request);
      byte[] buffer = new byte[1000];
      DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
      aSocket.receive(reply);
      System.out.println("Reply: " + new String(reply.getData()));
    } catch (SocketException e) {
      System.out.println("Socket: " + e.getMessage());
    } catch (IOException e) {
      System.out.println("IO: " + e.getMessage());
    } finally {
    if (aSocket != null)
      aSocket.close();
    }
  }
}
```

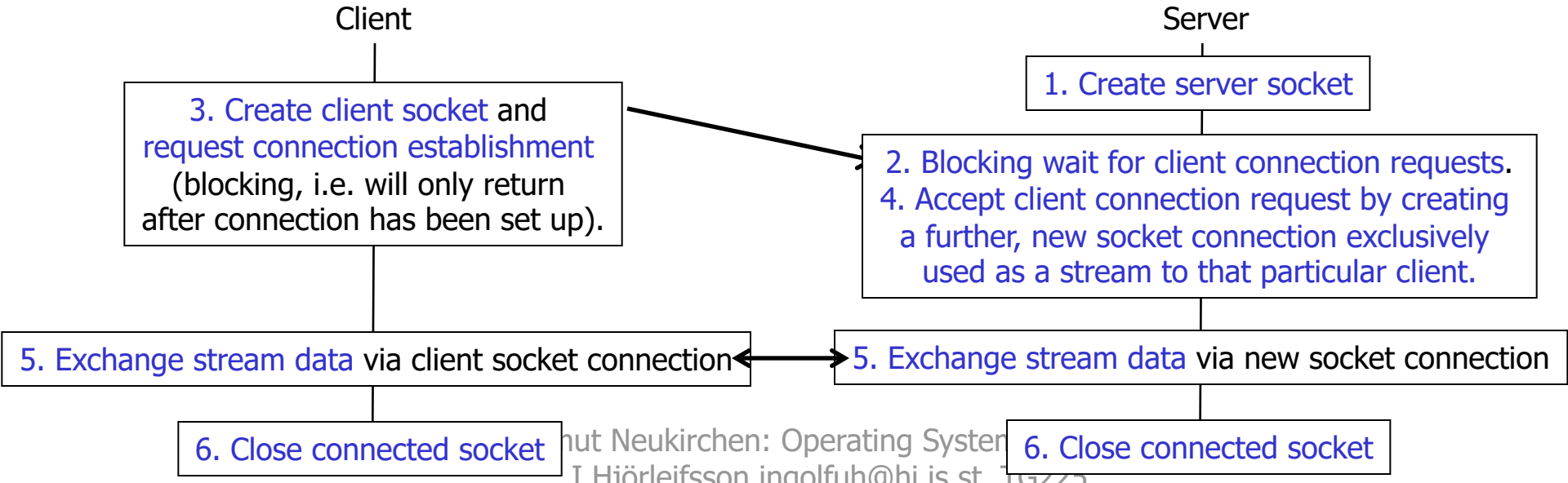
- Client sends message via socket to agreed port and waits on same socket for reply.

Command line parameters.

Needs to be the same as the server's port.

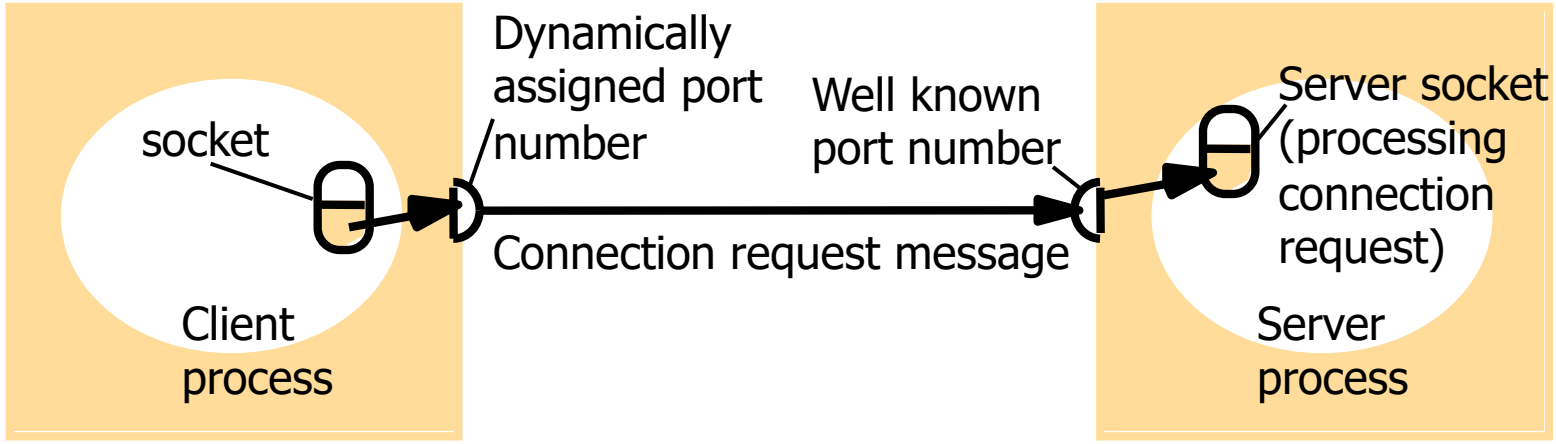
Connection-oriented Sockets for Servers and Clients (1)

- A connection (=bi-directional stream for endless amount of bytes) is initiated by the client and accepted by the server that is waiting for connection requests.
  - Server socket is listening for connection requests from a client socket.
  - Once this connection is established (=request is accepted), a new "connected" socket is created at the server side via which client (using the same client socket via which connection was requested) and server may exchange streams of data.
    - Original server socket can still be used to accept further connection requests.

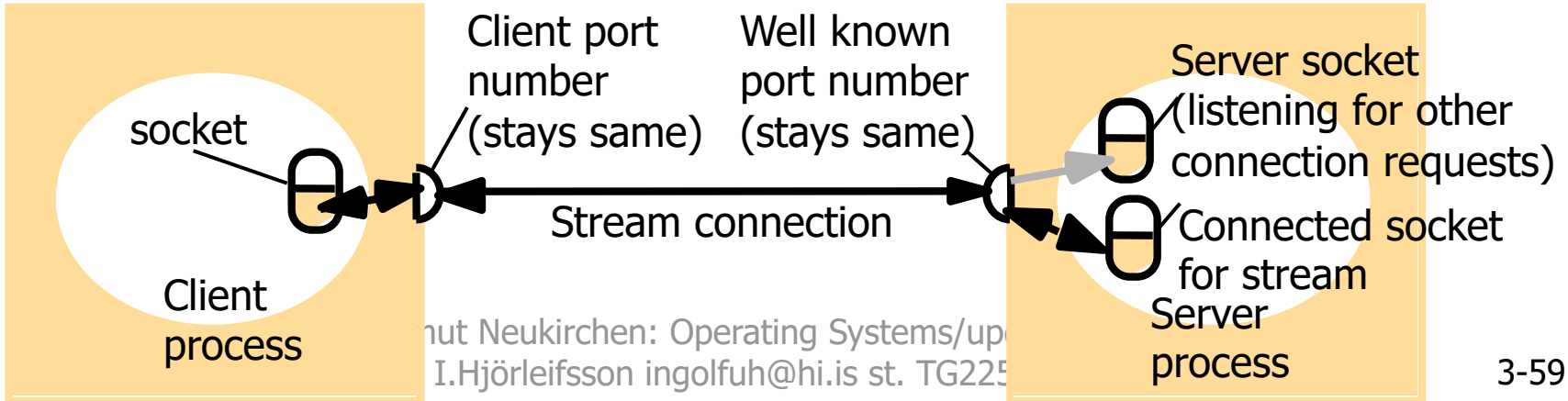


Connection-oriented Sockets for Servers and Clients (2)

- 1. Establishing connection by connecting to server socket listening for connection requests:

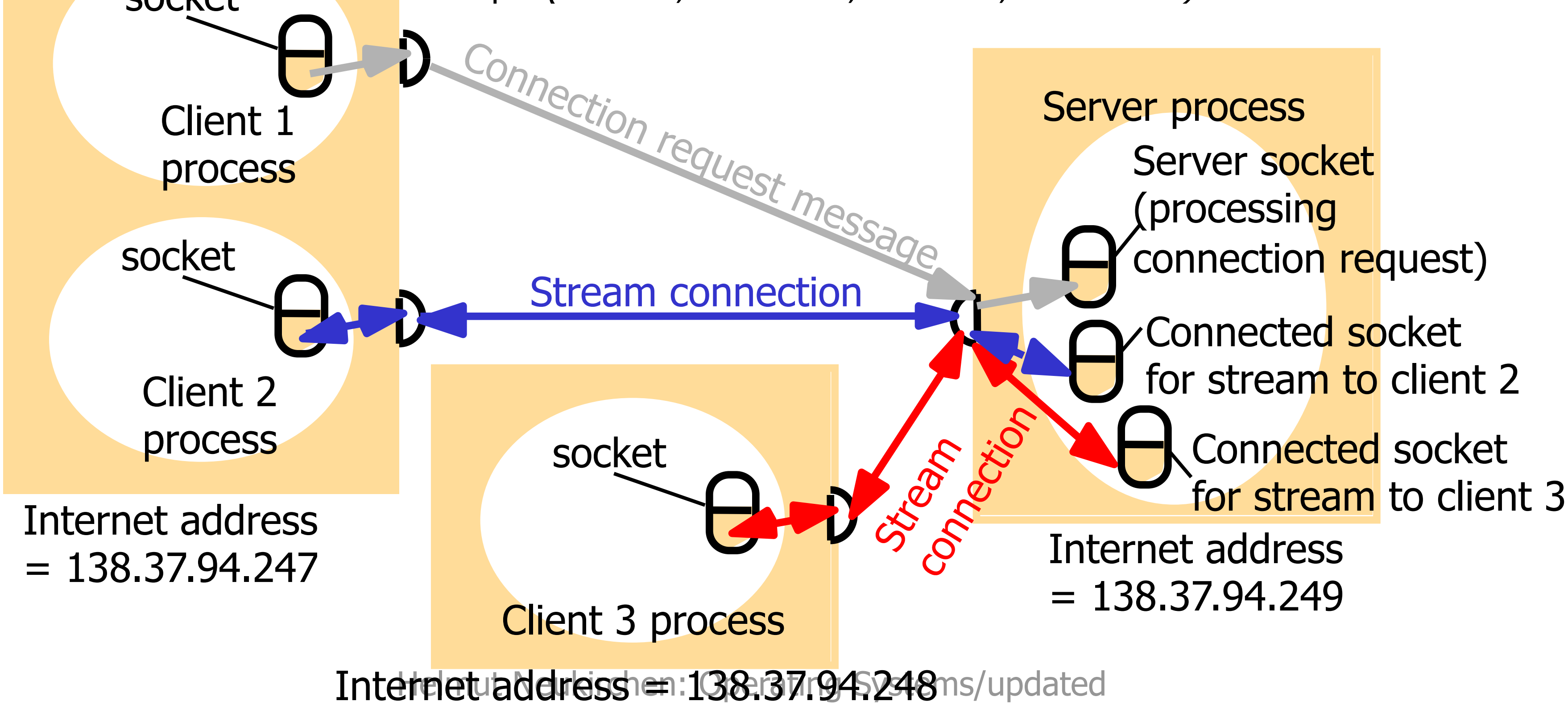


- 2. Established connection via created connected socket at server side, same socket at client side (further connection requests to server socket possible, resulting each time in a further connected socket at server for handling that new stream):



Connection-oriented Sockets for Servers and Clients (3)

- Multiple connections to same server port possible at the same time.
  - Server OS (in charge of ports) can keep connections apart and route them to the right socket of server process, because the clients differ in port number and/or IP number.
    - A connection is uniquely identified by the tuple *(Client IP, Client Port, Server IP, Server Port)*.



Java API for Connection-oriented Server Sockets

- Class **`ServerSocket`** from package **`java.net.ServerSocket`** represents a connection-oriented server socket.
  - Is only used for listening to connection request (that is used to establish the connection used for sending/receiving the actual data stream).

- Constructors and Methods:
  - **`ServerSocket`**`(int port) throws IOException`: creates a socket that is bound to local port **`port`**.
  - `Socket` **`accept()`** `throws IOException`: listens in a blocking-style for a connection to be accepted. Creates and returns a further connected socket that can be used for the actual stream communication. (See next slide…)

Java API for Connection-oriented Sockets and Stream Communication

- Class **`Socket`** from package **`java.net.Socket`** represents a plain socket.
- Constructor:
  - **`Socket`**`(String host, int port) throws UnknownHostException, IOException`: Creates a socket for a client and connects it to target host and port of a server socket. Such a connected socket can be used for stream communication.
  - If socket has been constructed and returned by the **`ServerSocket.accept()`** method, it is also a connected one that can be used for stream communication.
- Methods for applying ordinary Java stream communication (read, write etc.):
  - `InputStream` **`getInputStream`**`() throws IOException`: gets input stream of socket.
  - `OutputStream` **`getOutputStream`**`() throws IOException`: gets output stream of socket.
  - The stream objects returned by **`getInputStream()`** and **`getOutputStream()`** can be used as arguments of constructors that create suitable Java input and output streams, e.g. **`DataInputStream`** and **`DataOutputStream`** which allow binary representations of primitive Java data type values to be read and written in a machine independent manner using **`read`** and **`write`** methods (stream can be used to transmit infinite amount of bytes).

Java Example for Connection-Oriented Server Process

```
import java.net.*;
import java.io.*;

public class ConnectionOrientedServer {
  public static void main(String args[]) {
    try {
      int serverPort = 7896; // the server port
      ServerSocket listenSocket = new ServerSocket(serverPort);
      while (true) {
        Socket clientSocket = listenSocket.accept();
        Connection c = new Connection(clientSocket); // Handle request
      }
    } catch (IOException e) {
      System.out.println("Listen socket:" + e.getMessage());
    }
  }
}
```

- Server just accepts connection and creates class to handle further processing (→next slide).

Needs to be changed if already used by some other server (Error "Socket: Address already in use") or your old server is still running and needs to be terminated first.

The `ConnectionOrientedServer` class is only responsible for accepting the initial connection of each client. The actual communication (& service provision) is handled by a separate class `Connection` (→next slide).

Java Example for Connection-Oriented Server Class

```
import java.net.*;
import java.io.*;
class Connection {
  DataInputStream in;
  DataOutputStream out;
  Socket clientSocket;
  public Connection(Socket aClientSocket) {
    try {
      clientSocket = aClientSocket;
      in = new DataInputStream(clientSocket.getInputStream());
      out = new DataOutputStream(clientSocket.getOutputStream());
      handleRequest();
    } catch (IOException e) { System.out.println("Connection:" + e.getMessage()); }
  }
  public void handleRequest() {
    try { // an echo server
      String data = in.readUTF(); // read a Unicode-encoded text string from the stream
      out.writeUTF(data);
    } catch (EOFException e) { System.out.println("EOF:" + e.getMessage());
    } catch (IOException e) { System.out.println("readline:" + e.getMessage());
    } finally {
      try {
        clientSocket.close();
      } catch (IOException e) {/* close failed */
      }
    }
  }
}
```

- Actual communication (& service provision) after accepting connection: send the received stream data back via the established connection.

Connecting to the input and output stream of the socket.

The actual server routine (is called as part of the constructor).

Read/write stream operations are used.

Note: `readUTF`/`writeUTF` have the string size as a first value, then the actual text string follows – therefore, `readUTF` knows how many bytes it needs to receive from the stream before returning the string.

Java Example for Connection-oriented Client Process

- Client sends message via socket to agreed port and waits on same socket for reply.

```
import java.net.*;
import java.io.*;

public class ConnectionOrientedClient {
  public static void main(String args[]) {
    // args[0]: message contents, args[1]: destination hostname
    Socket aSocket = null;
    try {
      int serverPort = 7896;
      aSocket = new Socket(args[1], serverPort);
      DataInputStream in = new DataInputStream(aSocket.getInputStream());
      DataOutputStream out = new DataOutputStream(aSocket.getOutputStream());
      out.writeUTF(args[0]); // UTF is a Unicode-based string encoding
      String data = in.readUTF(); // read a UTF string from the stream
      System.out.println("Received: " + data);
    } catch (UnknownHostException e) { System.out.println("Socket:" + e.getMessage());
    } catch (EOFException e) { System.out.println("EOF:" + e.getMessage());
    } catch (IOException e) { System.out.println("readline:" + e.getMessage());
    } finally {
      if (aSocket != null)
      try {
        aSocket.close();
      } catch (IOException e) { System.out.println("close:" + e.getMessage());
      }
    }
  }
}
```

Needs to be the same as the server's port.

Read/write stream operations are used.

Excursion: Java package concept used in samples

- Source code of these Java examples is provided in Canvas (*.zip) and in video.
- To separate the different Java examples used in the different chapters and sections, the hierarchical Java package concept is used: e.g., classes **`ConnectionLessClient`** and **`ConnectionLessServer`** are contained in package **`ch3Processes.connectionLessSocket`**.
  - To start a class using the JVM (**`java`** command), the package name needs to be used as prefix of that class: **`java ch3Processes.connectionLessSocket.ConnectionLessClient`**
  - In the file system, each package is mapped to a directory, e.g. class **`ch3Processes.connectionLessSocket.ConnectionLessClient`** is contained in a file with path **`ch3Processes/connectionLessSocket/ConnectionLessClient.class`** (or **`.java`**)
  - The package file path will get added to the Java Classpath, e.g. if the bytecode of class **`ConnectionLessClient`** is contained in file **`/home/helmut/workspace/OperatingSystems/bin/ch3Processes/connectionLessSocket/ConnectionLessClient.class`**, the JVM needs to be started using **`java -cp /home/helmut/workspace/OperatingSystems/bin ch3Processes.connectionLessSocket.ConnectionLessClient`**

No line break here

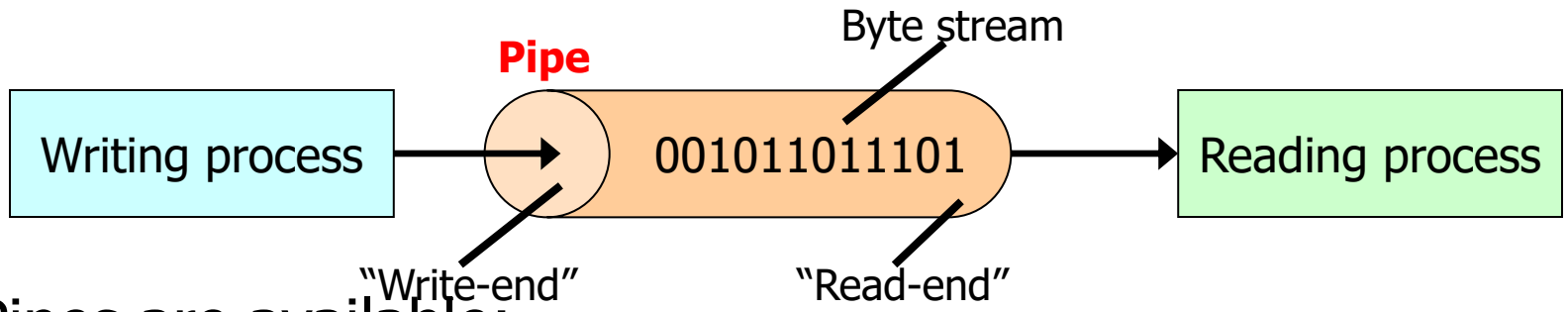
  - Note: Microsoft systems use \ than / as directory separator. For example, if the folder **`ch3Processes`** is located in **`C:\Users\jon ingi`**, then, you have to use:

```
java -cp "C:\Users\jon ingi" ch3Processes.connectionLessSocket.ConnectionLessClient
```

Double quotes if path contains blanks. On MS Windows, it must not end with "\", i.e. …**`ingi"`**, not …**`ingi\"`**

Pipes

- Pipes are the most simple interprocess communication mechanism:
  - Transmit a stream of bytes in FIFO (First In, First Out) processes.
  - Ordinary read/write operations like for files can be used.



- Pipes are available:
  - to a programmer using system calls of POSIX and MS Windows.
  - to the end user using the pipe operator "**|**" in a command line shell, e.g.: **`dir | more`**

Characteristics of Pipes

- Pipes
  - have a limited size (Linux: 4KByte) and thus their content can be easily kept in main memory (in contrast to sharing data via a file).
    - If pipe full: reader has to read before further data can be written.
  - may be named or unnamed/anonymous:
    - A named pipe is mapped to a file name in the file system. Any process that knows the file name can access that pipe (which is not a real file).
    - An unnamed/anonymous pipe requires a parent-child relationship between the communicating processes to be able to refer to and share the same pipe (→process creation using fork).
  - may allow unidirectional communication only or bidirectional communication.
  - may be restricted to processes running on the same machine or may support communication over a network (e.g. POSIX named pipes together with the `netcat` system program).

POSIX
Unnamed Pipes

- Unidirectional & Anonymous pipes:
  - System call: `pipe(int fd[2])`
    - Creates pipe (stored in memory managed by OS); returns a file handle ("file descriptor") for the read-end (`fd[0]`), and a file handle for the write-end (`fd[1]`).
    - These handles can be used like opened files.
    - Write-end may only be used to write to pipe, read-end to read from.
  - Usually, parent process creates pipe using above system call and then forks.
    - Child process is a copy of parent, thus it inherits the file handles that refer to the same anonymous pipe. Pipe (owned by OS) not copied, just the handles to it.
  - What one process writes to the pipe can be read by the other process from the pipe.
    - Pipe has limited size: if a process is writing faster to the pipe than the other process is reading, pipe fills up: writing process gets blocked until some space is available again (due to reading). Reading process gets blocked if pipe is empty.
  - A process that is writing shall close its read-end, a process that is reading shall close its write-end (to let the OS know the direction of the pipe).

POSIX Unnamed Pipes: C example

```
main() {
  char buffer[5];/* Buffer for received data */
  int fd[2];     /* File descriptors for read-end (fd[0])
                                    and write-end (fd[1]) of pipe */
  pipe(fd); /* Create unnamed, directional pipe: fd[0] and fd[1] are returned */
  if (fork() == 0) /* Fork child process that gets a copy of file descriptors */
  { /* Child process as writer */
    close(fd[0]); /* Close unused read-end (refers only to read-end of child */
    write(fd[1], "Test", 5); /* Write 5 byte string to write-end of pipe */
    ...
    exit(0); /* Terminate child process */
  }
  else
  {/* Parent process as reader */
    close(fd[1]); /* Close write-end (refers only to write-end of parent) */
    read(fd[0], buffer, 5); /* Read 5 bytes from read-end of pipe */
    printf("Read: %s\n", buffer);
    ...
    exit(0); /* Terminate parent process */
  }
} /* When processes terminate, OS will close all opened files. Once the last
     file descriptor for a pipe is closed, unnamed pipe ceases to exist. */
```

3.6 Summary

- Process: program in execution.
  - Process Control Block (PCB) represents a process by storing all relevant information of a process.
  - Process states: new, ready, running, waiting, terminated.
  - Non-running processes placed in queues.
- Scheduler responsible for switching between processes (context switch).
- System calls for creating and terminating processes.
- Process cooperation requires interprocess communication:
  - Shared memory,
  - Message passing,
  - Sockets,
    - Only Java Socket API covered – POSIX socket API different: not object-oriented.
  - *(Remote Procedure Call/Remote Method Invocation),*
  - Pipes.