Operating Systems Structure
and
Processes

Otto J. Anshus
University of Tromsø/Oslo

The Architecture of an OS

- Monolithic
- Layered
- Virtual Machine, Library, Exokernel
- Micro kernel and Client/Server
- Hybrids

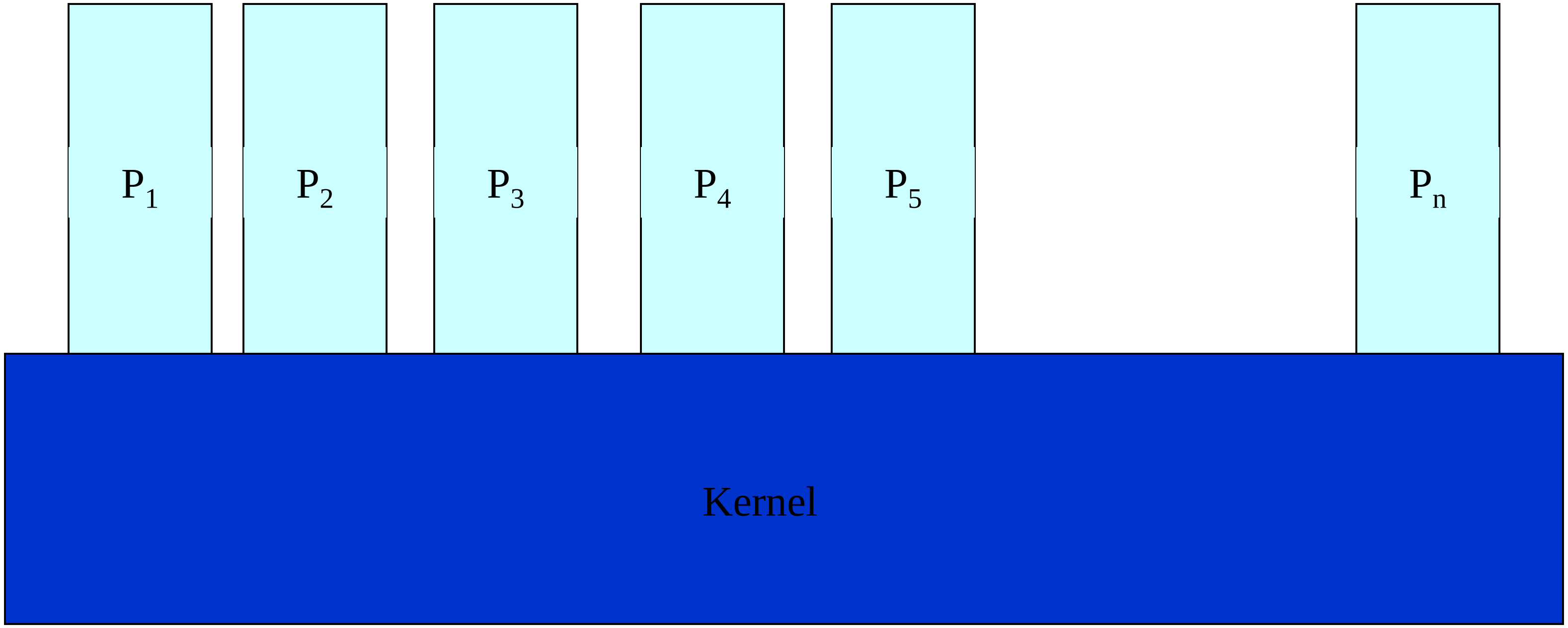
Goals of the architecture

- OS as Resource Manager
- OS as Virtual Machine (abstractions)
- Efficiency, flexibility, size, security, … as discussed earlier

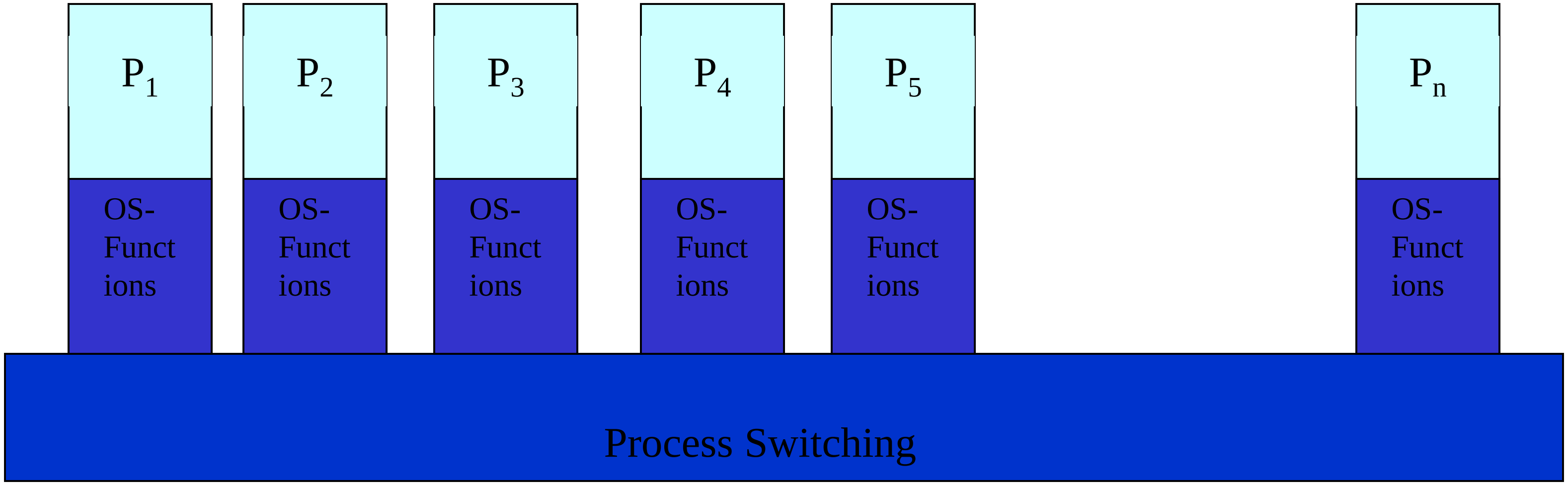
Operating System Use of Processes:
Where is the OS executing?

Illustrations: Stallings: Operating System: Internals and Design

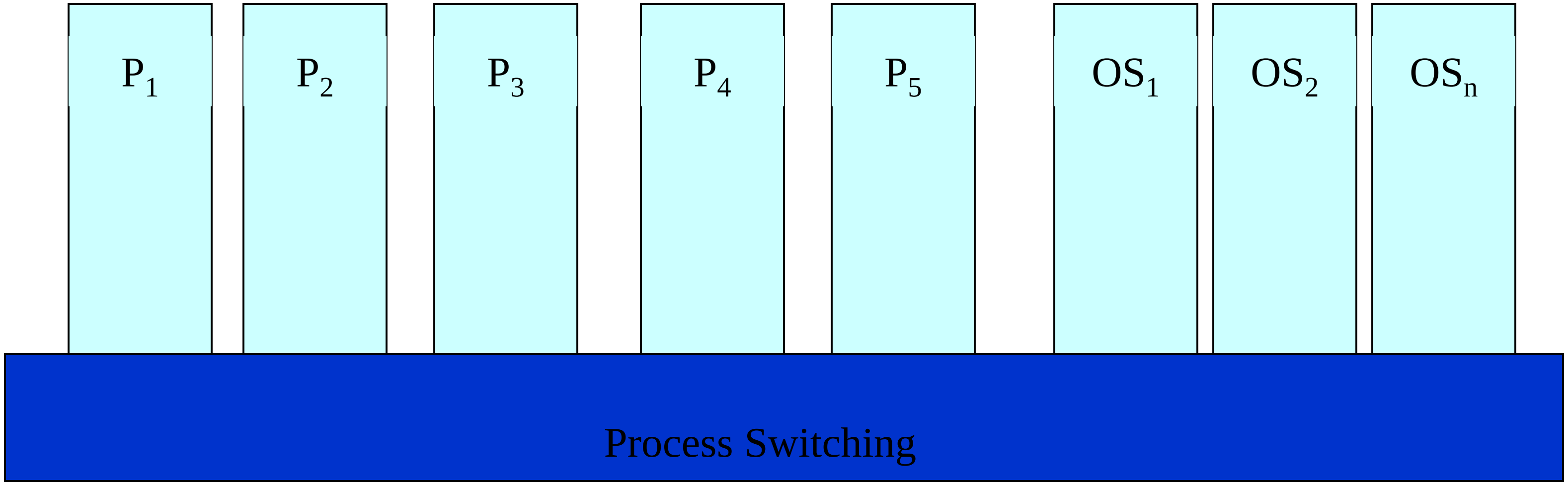
Separate Kernel

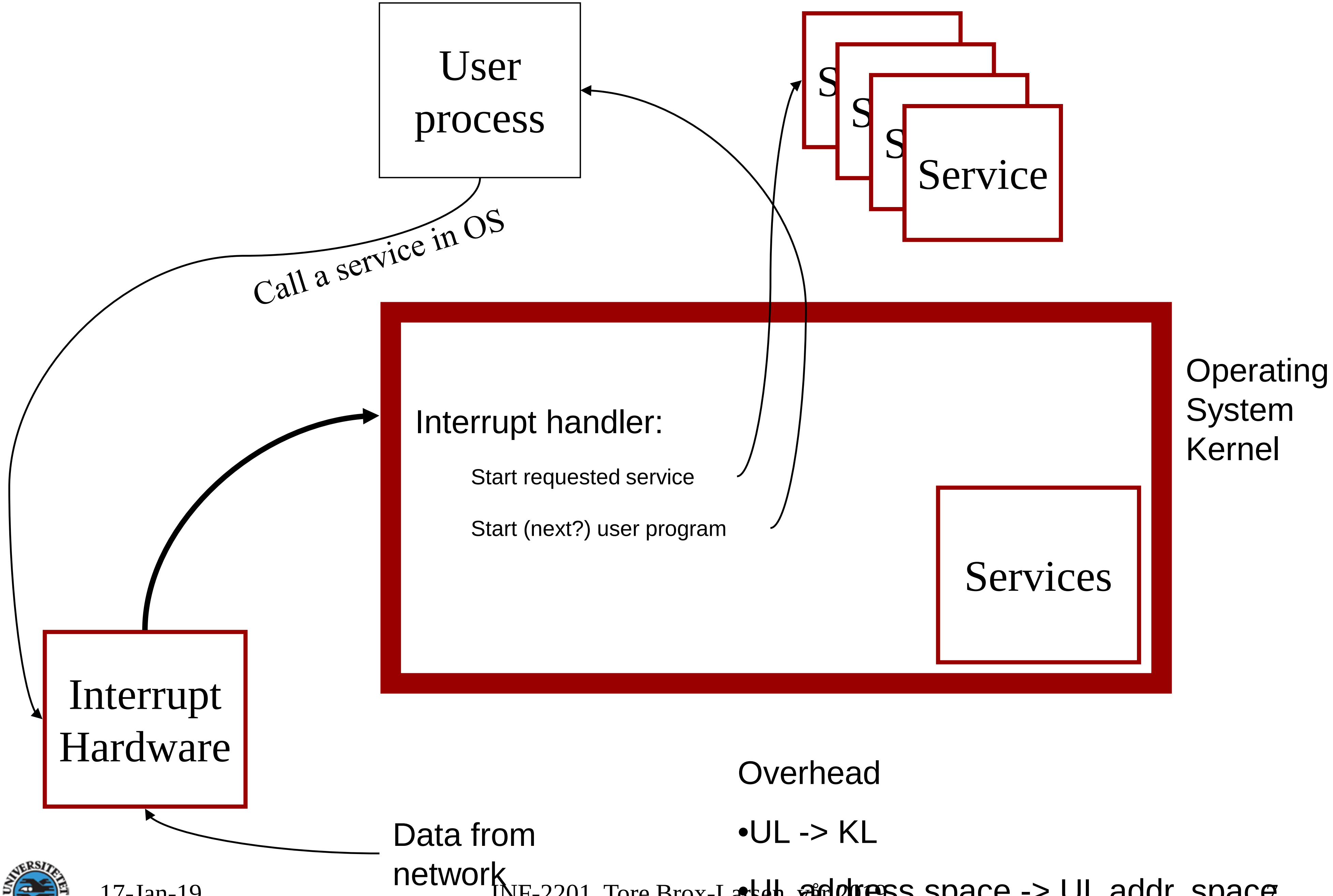


OS-Functions Executing within Processes



OS-Functions Executing in Separate Processes



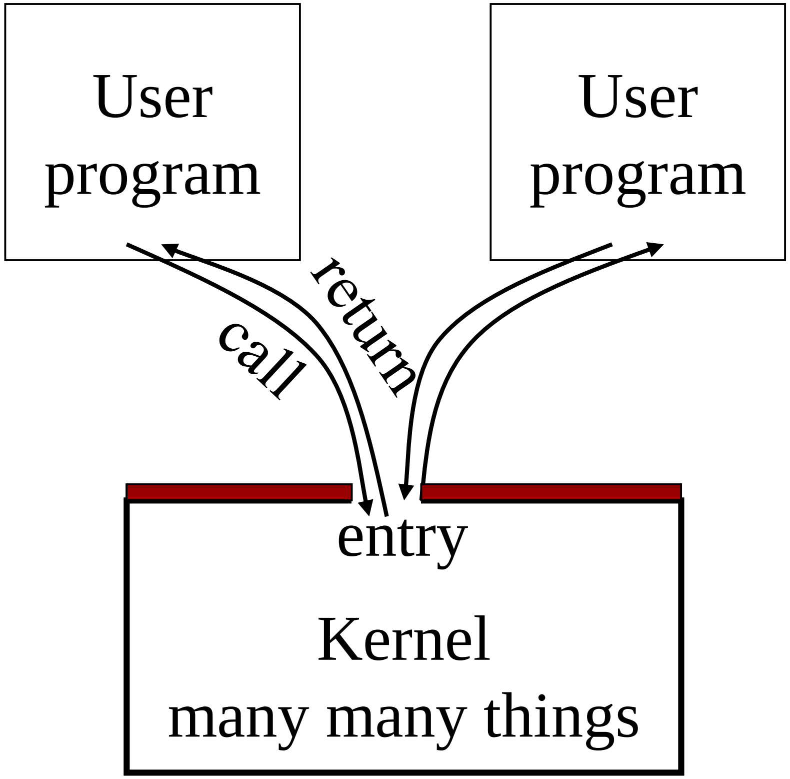


Overhead

- UL -> KL
- UL address space -> UL addr. space

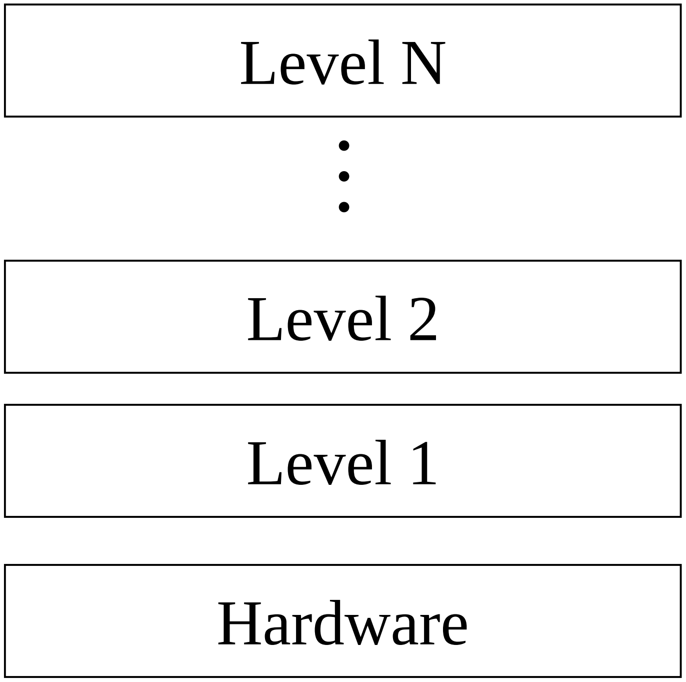
Monolithic

- All kernel routines are together
- A system call interface
- Examples:
  - Linux
  - Most Unix OS
  - NT (hybrid)



Layered Structure

- Hiding information at each layer
- Develop a layer at a time
- Examples
  - THE (6 layers, semaphores, Dijkstra 1968)
  - MS-DOS (4 layers)



Level N

⋮

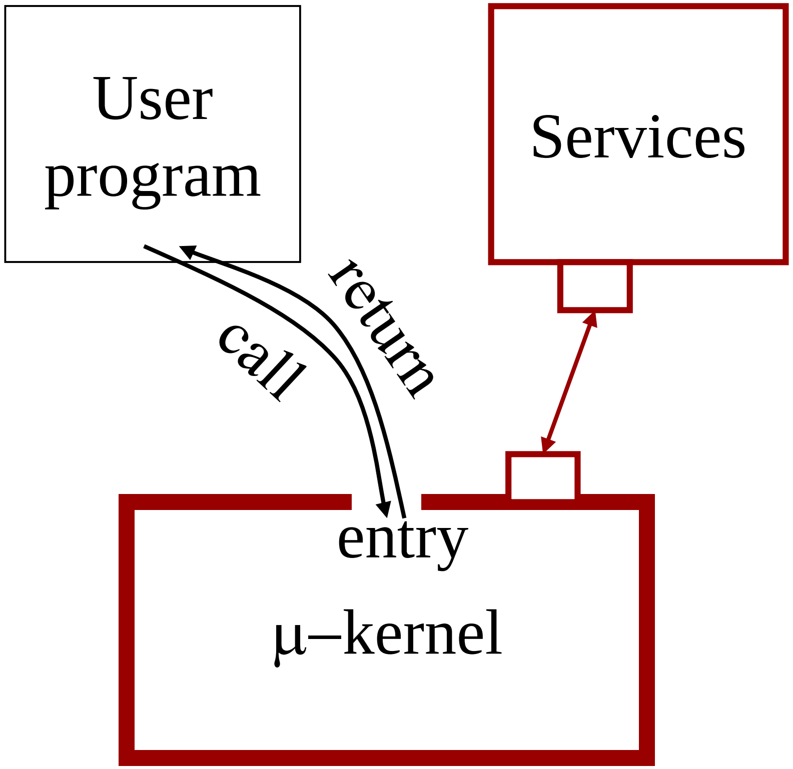
Level 2

Level 1

Hardware

Microkernel and Client/Server

- Micro-kernel is “micro”
- Services are implemented as user level processes
- Micro-kernel get services on behalf of users by messaging with the service processes
- Example: L4, Nucleus, Taos, Mach, Mach, NT (hybrid)



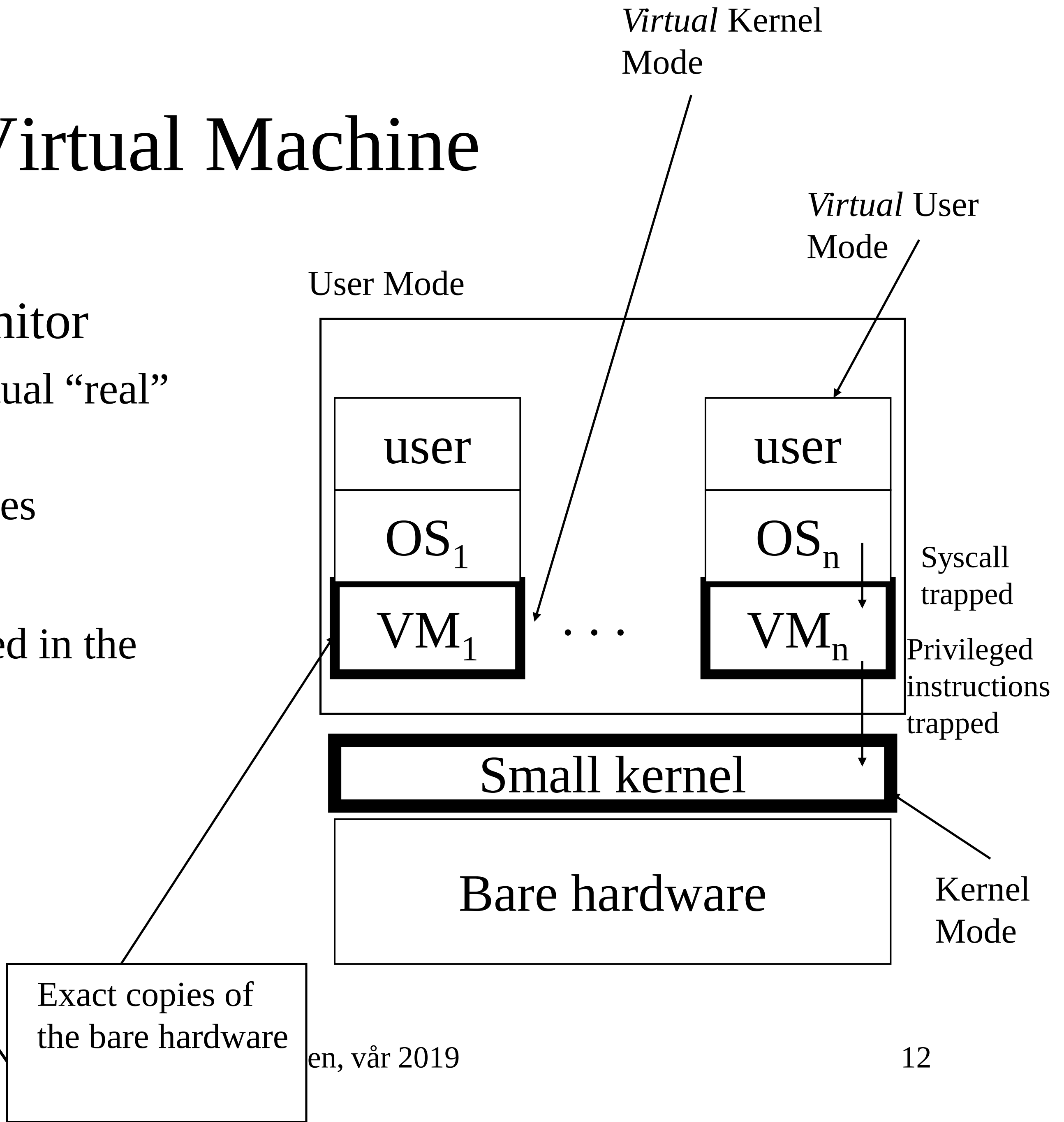
Virtual Machine

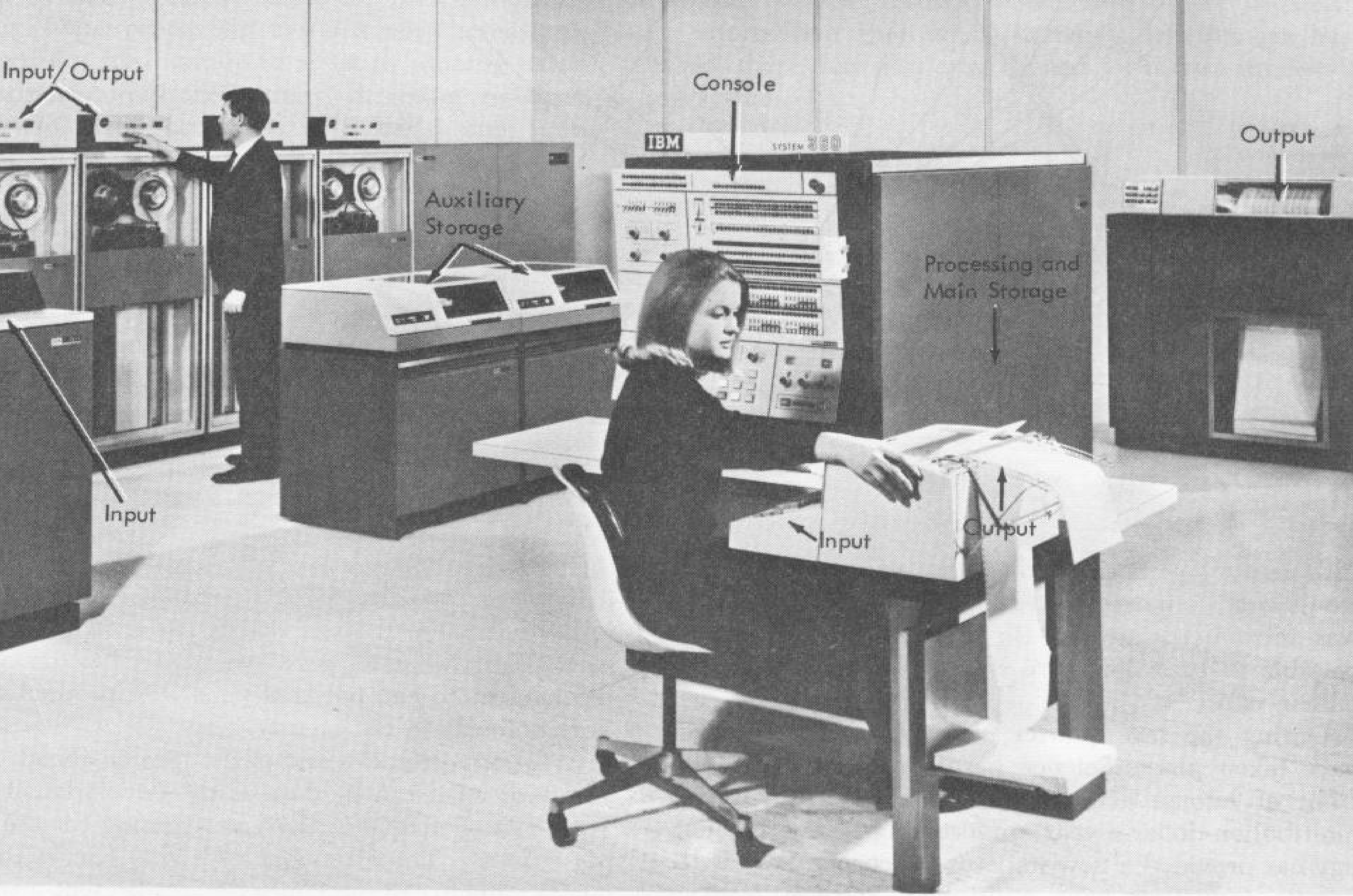
"A running program is often referred to as a virtual machine - a machine that doesn't exist as a matter of actual physical reality. The virtual machine idea is itself one of the most elegant in the history of technology and is a crucial step in the evolution of ideas about software. To come up with it, scientists and technologists had to recognize that a computer running a program isn't merely a washer doing laundry. A washer is a washer whatever clothes you put inside, but when you put a new program in a computer, it becomes a new machine.... The virtual machine: A way of understanding software that frees us to think of software design as machine design."

From David Gelernter's "Truth, Beauty, and the Virtual Machine," Discover Magazine, September 1997, p. 72.

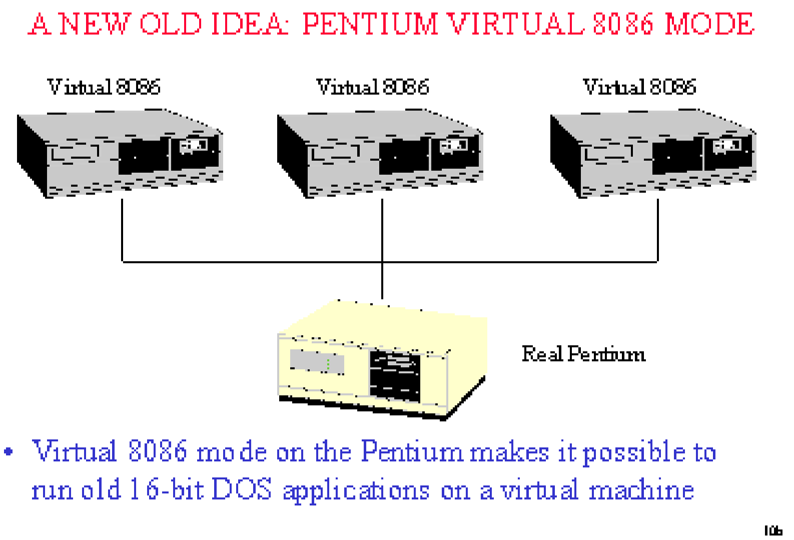
Virtual Machine

- Virtual machine monitor
  - provide multiple virtual “real” hardware
  - run different OS codes
- Example
  - IBM VM/370: Started in the 70’s. Check out
  - virtual 8086 mode
  - Java VM
  - VMware
  - Exokernel




Input/Output
Console
IBM
SYSTEM 360
Output
Auxiliary Storage
Processing and Main Storage
Input
Input
Output

Virtual 8086



A NEW OLD IDEA: PENTIUM VIRTUAL 8086 MODE

- Virtual 8086 mode on the Pentium makes it possible to run old 16-bit DOS applications on a virtual machine

Java VM

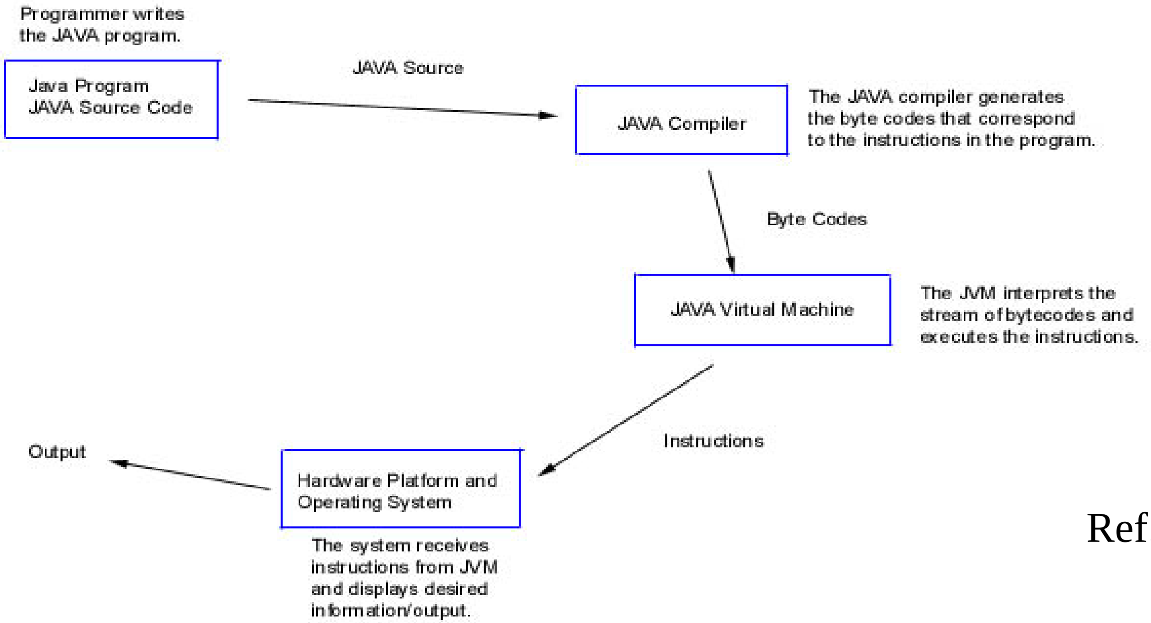
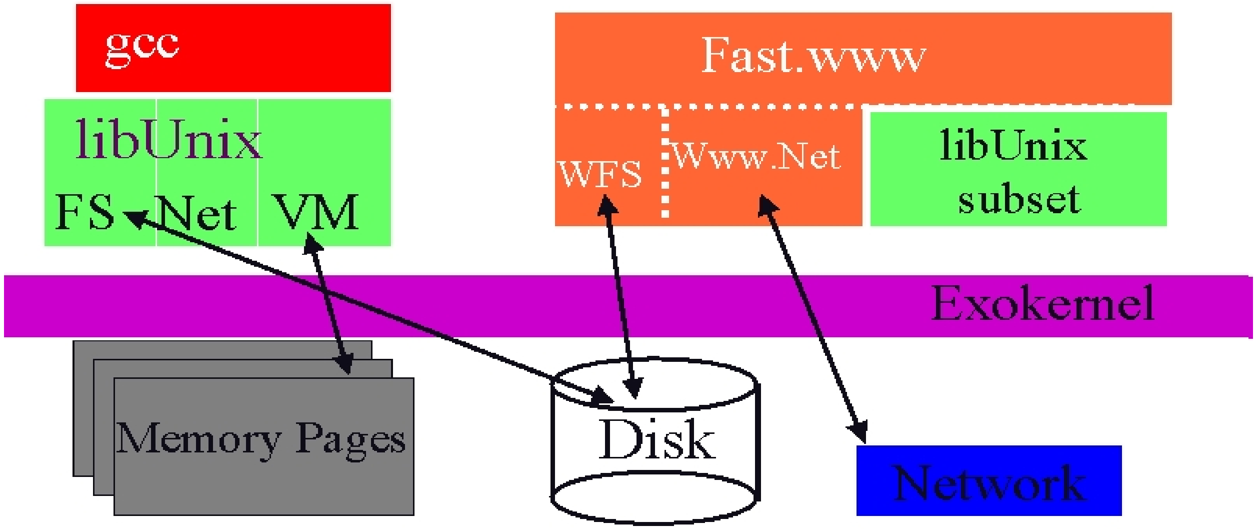

Programmer writes the JAVA program.
Java Program JAVA Source Code
JAVA Source
JAVA Compiler
The JAVA compiler generates the byte codes that correspond to the instructions in the program.
Byte Codes
JAVA Virtual Machine
The JVM interprets the stream of bytecodes and executes the instructions.
Instructions
Hardware Platform and Operating System
The system receives instructions from JVM and displays desired information/output.
Output

Figure 1.1: Diagram of Java Program Execution

Ref. Pascal P-Code

Exokernel Architecture



Hardware Support

- What is the minimal support?
  - 2 modes
  - Exception and interrupt trapping
- Can virtual machine be protected without such support?
  - Yes, emulation instead of executing on real machine

Pro et Contra

Monolithic	Layered	VM	C/S	Micro kernel
•Performance	•Clean, less bugs •Clear division of labour	•Many virtual computers with different OS’es •Test of new OS while production work continues •All in all: flexibility	•Clear division of labour	•More flexible •Small means less bugs+manageable •Distributed systems •Failure isolation of services at Kernel Level
•Less structured	•Are layers really sepaparated? •Performance issues?	•Performance issues? •Complexity issues?	•Performance issues?	•Flexibility issues? •Performance issues?

“Truths” on Micro Kernel Flexibility and Performance

- A micro kernel restricts application level flexibility.
- Switching overhead kernel-user mode is inherently expensive.
- Switching address-spaces is costly.
- IPC is expensive.
- Micro kernel architectures lead to memory sy
- Kernel should be portable (on top of a small hardware-dependent layer).

NO: Can be <50 cycles

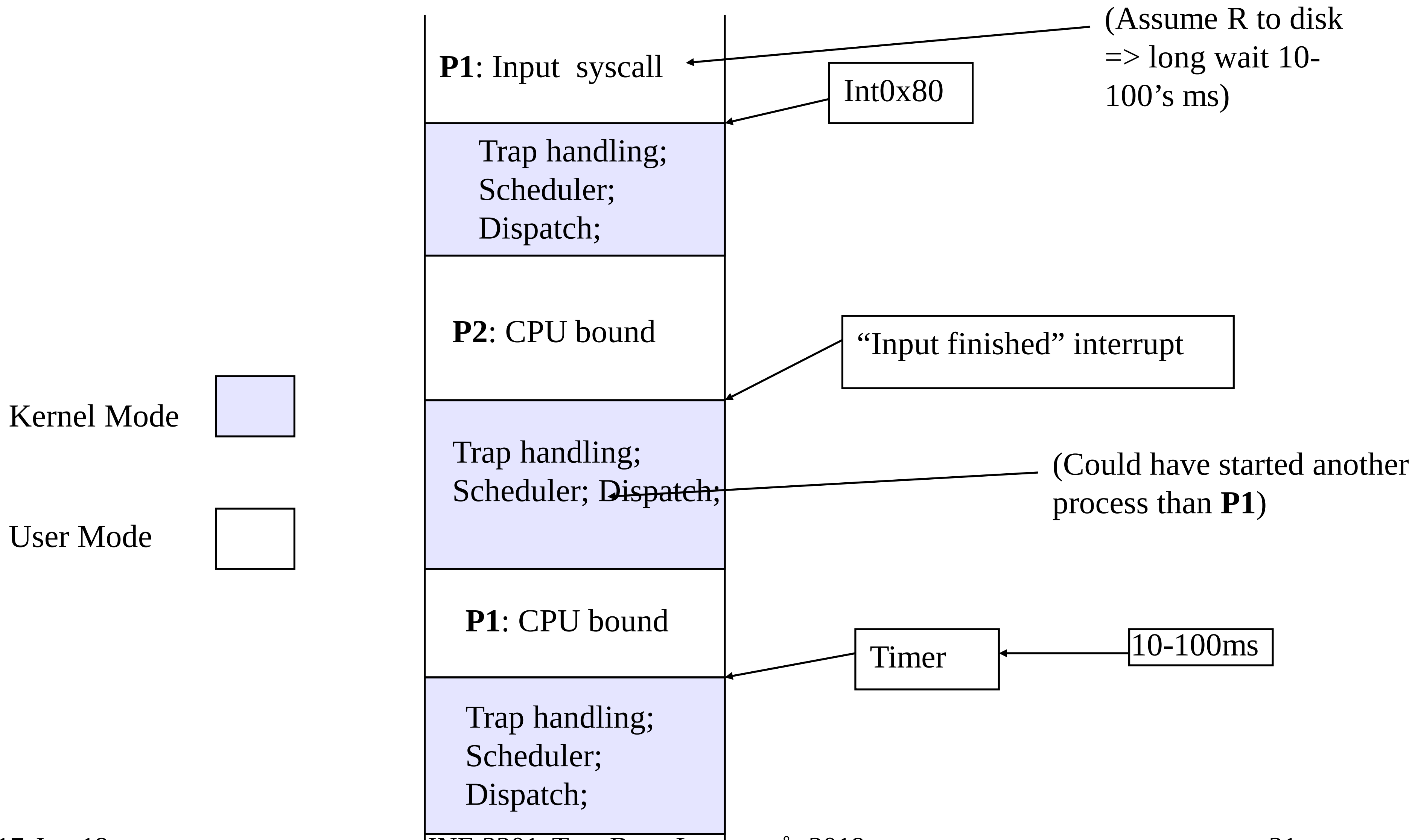
NO: 6-20 microsec round-trip, 53-500 cycles/IPC one way

Taken from Jochen Liedtke, SOSP 15 paper: ”On micro kernel construction”

Concurrency and Process

- Problem to solve
  - A shared CPU, many I/O devices and lots of interrupts
  - Users feel they have machine to themselves
- Strategy
  - Decompose hard problems into simple ones
  - Deal with one at a time
  - Process is such a unit

Flow of Execution



P1: Input syscall

(Assume R to disk => long wait 10-100’s ms)

Int0x80

Trap handling;
Scheduler;
Dispatch;

P2: CPU bound

“Input finished” interrupt

Trap handling;
Scheduler; Dispatch;

(Could have started another process than **P1**)

P1: CPU bound

Timer

10-100ms

Trap handling;
Scheduler;
Dispatch;

Kernel Mode ▨

User Mode □

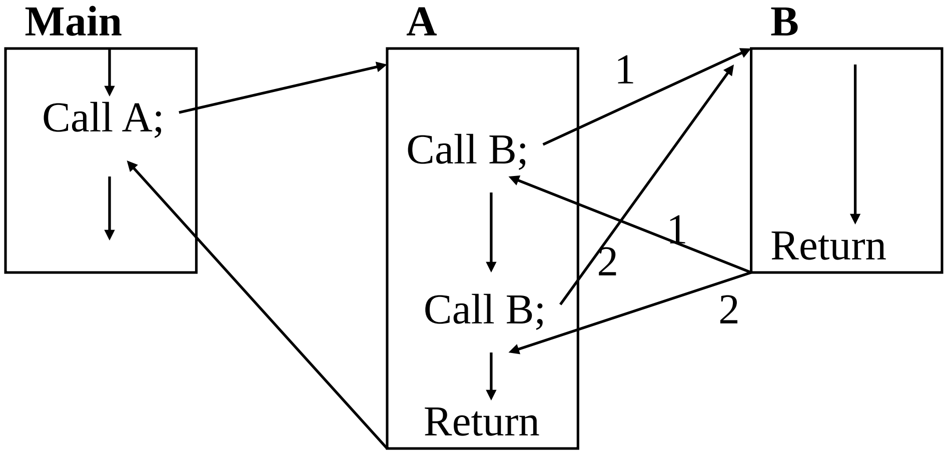
Procedure, Co-routine, Thread, Process

- Procedure, Function, (Sub)Routine
    - Call-execute all-return nesting
- Co-routine
    - Call-resumes-return

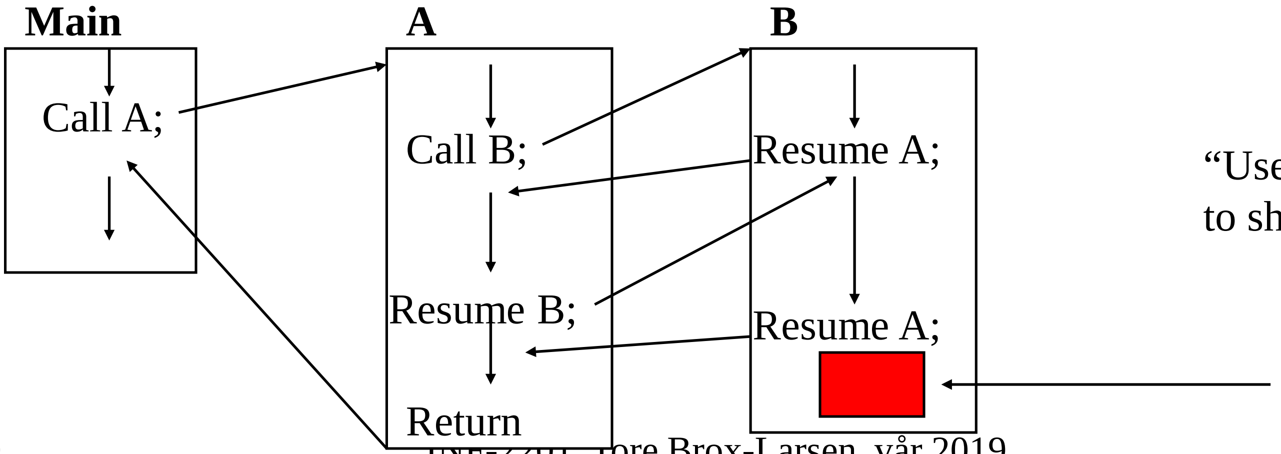
User level non preemptive “scheduler” in user code

- Thread (more later)
- Process
    – Single threaded ↓
    – Multi threaded ↓↓↓

Procedure and Co-routine



"User Yield when finished"



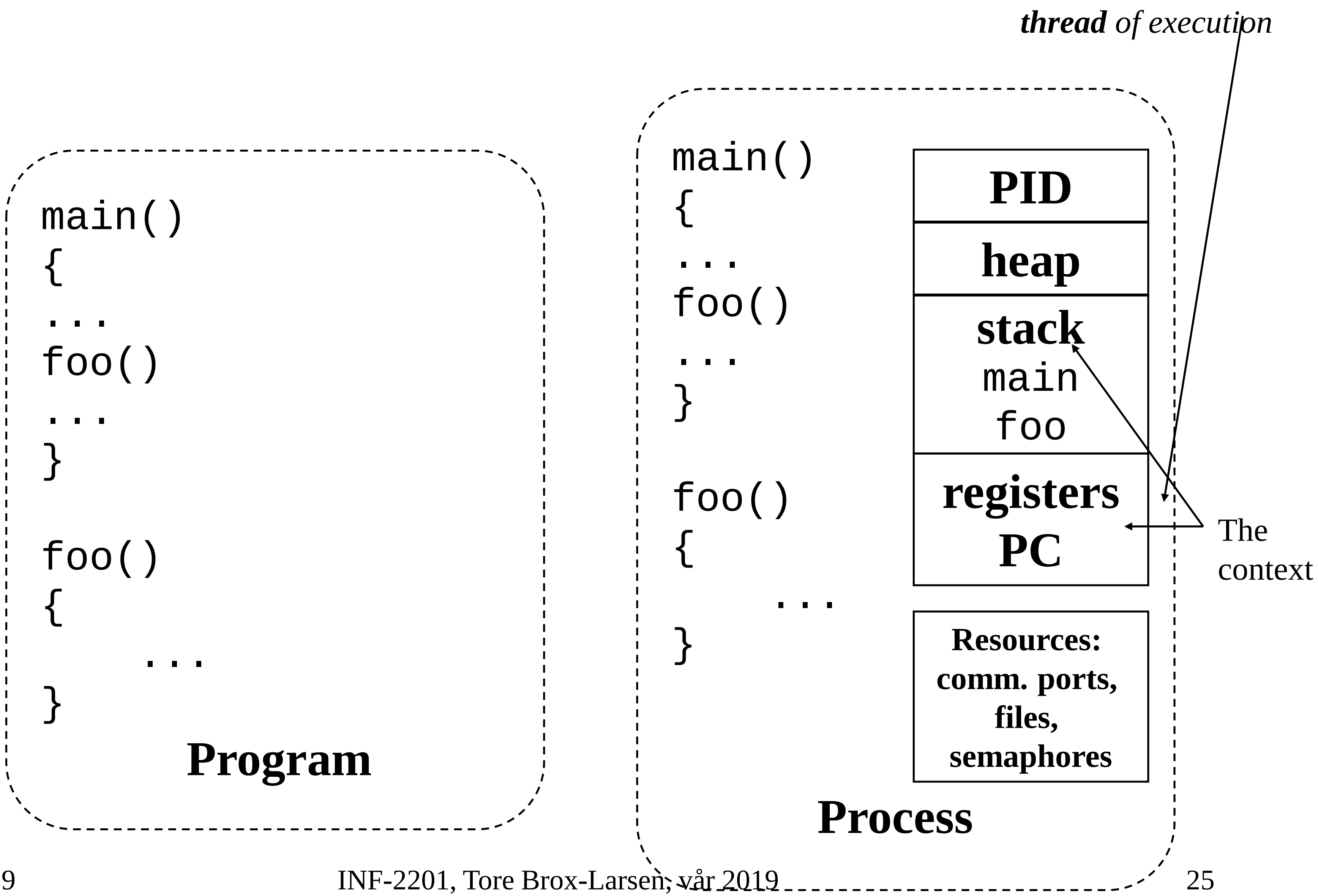
"User Yield during execution to share CPU"

Never executed

Process

- Sequential execution of operations
  - No concurrency inside a (**single** threaded) process
  - Everything happens sequentially
- Process state
  - Registers
  - Stack(s)
  - Main memory
  - Open files in UNIX
  - Communication ports
  - Other resources

Program and Process



For at least one *__thread__ of execution*

```
main()
{
...
foo()
...
}

foo()
{
    ...
}
```

Program

```
main()
{
...
foo()
...
}

foo()
{
    ...
}
```

PID

heap

stack

main

foo

registers

PC

The context

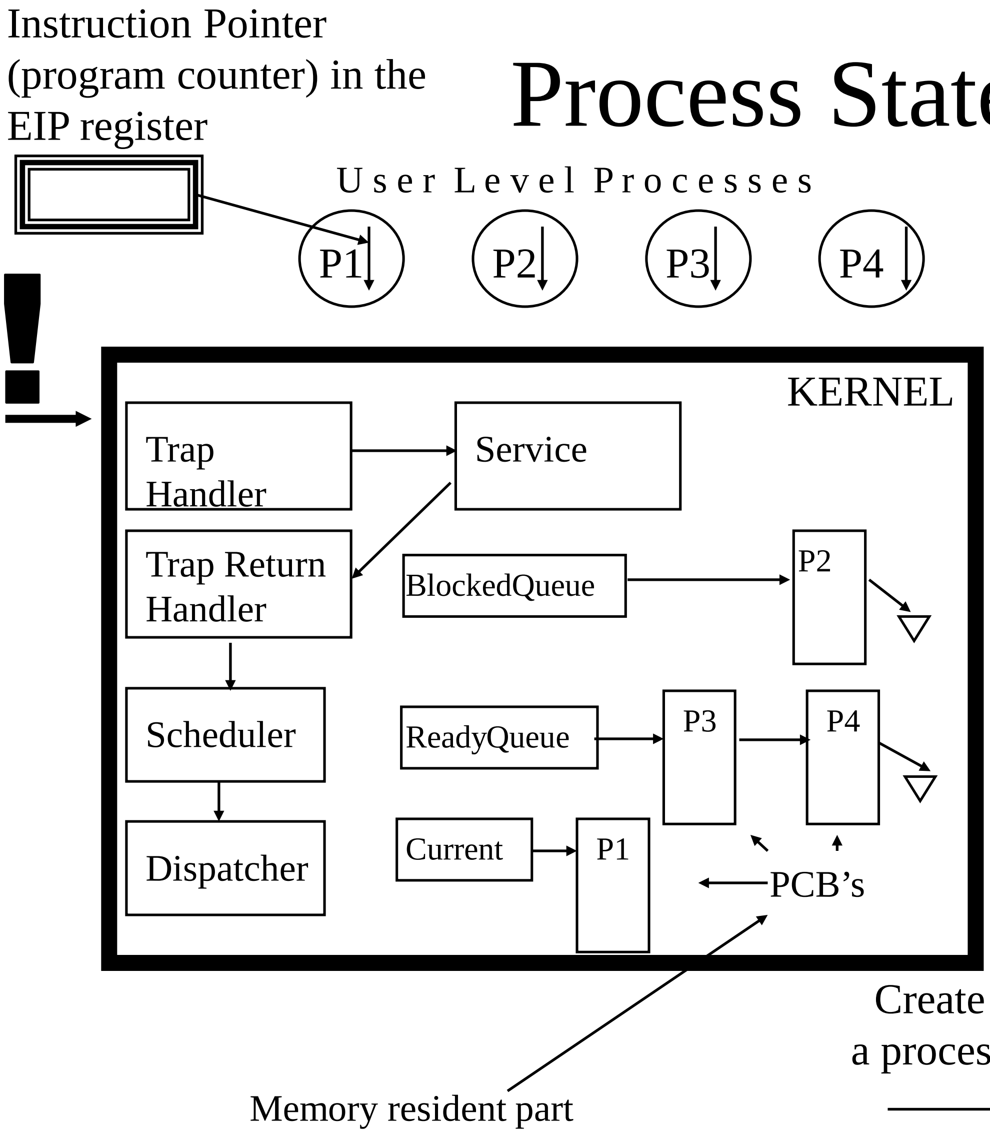
Resources: comm. ports, files, semaphores

Process

Process vs. Program

- Process > program
  - – Program is just part of process state
  - – Example: many users can run the same program
- Process < program
  - – A program can invoke more than one process
  - – Example: Fork off processes to lookup webster

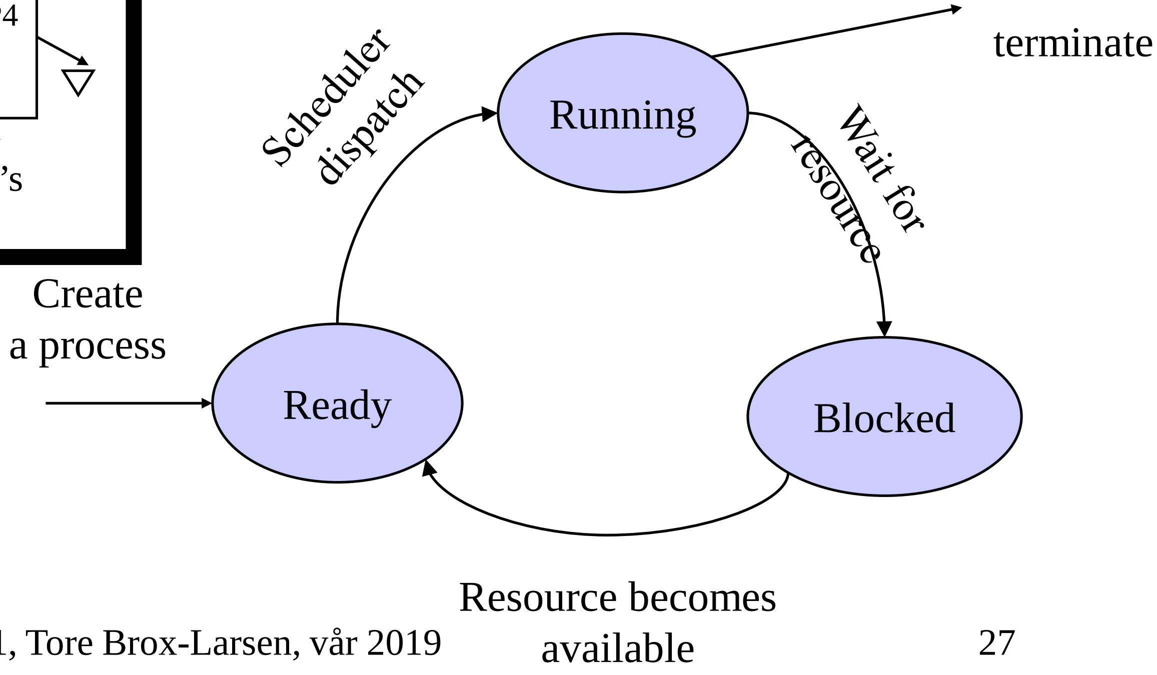
Process State Transitions



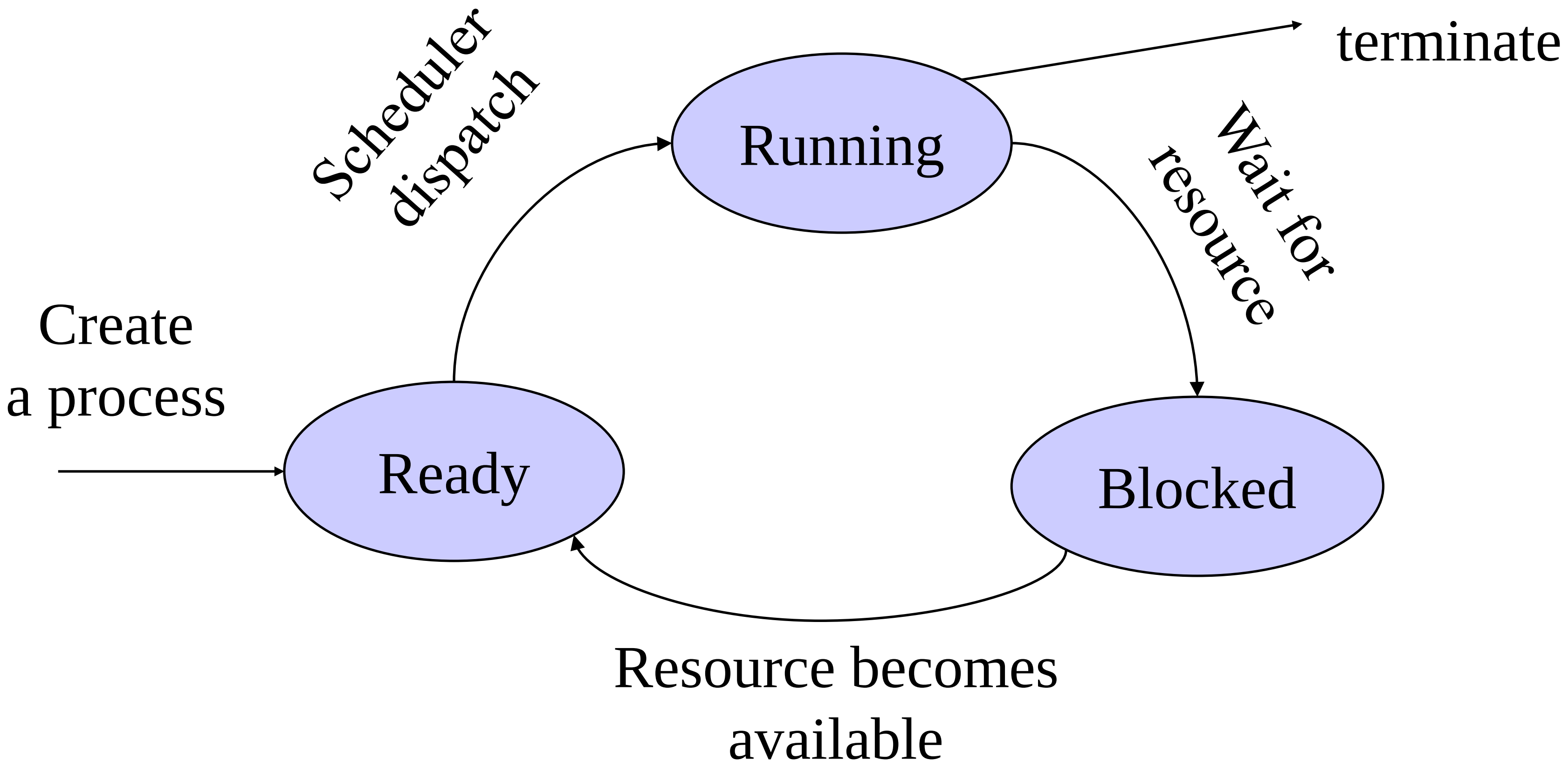
MULTIPROGRAMMING

•Uniprocessor: *Interleaving* (“pseudoparallelism”)

•Multiprocessor: *Overlapping* (“true paralellism”)



Process State Transition



Process Control Block (Process Table)

- What
  - Process management info
    - State (ready, running, blocked)
    - Registers, PSW, parents, etc
  - Memory management info
    - Segments, page table, stats, etc
  - I/O and file management
    - Communication ports, directories, file descriptors, etc.

Discussion: What needs to be saved and restored on a context switch?

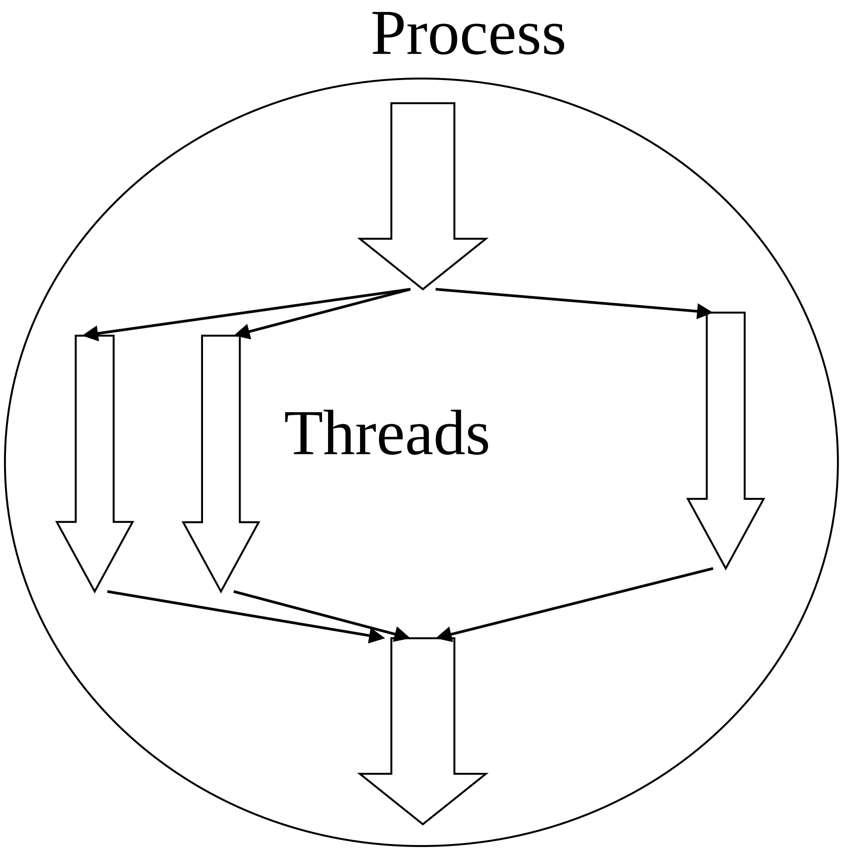
- Volatile state
    - Program counter (Program Counter (PC) also called Instruction Pointer (Intel: EIP))
    - Processor status register
    - Other register contents
    - User and kernel stack pointers
    - A pointer to the address space in which the process runs
        - the process's page table directory

…and how?

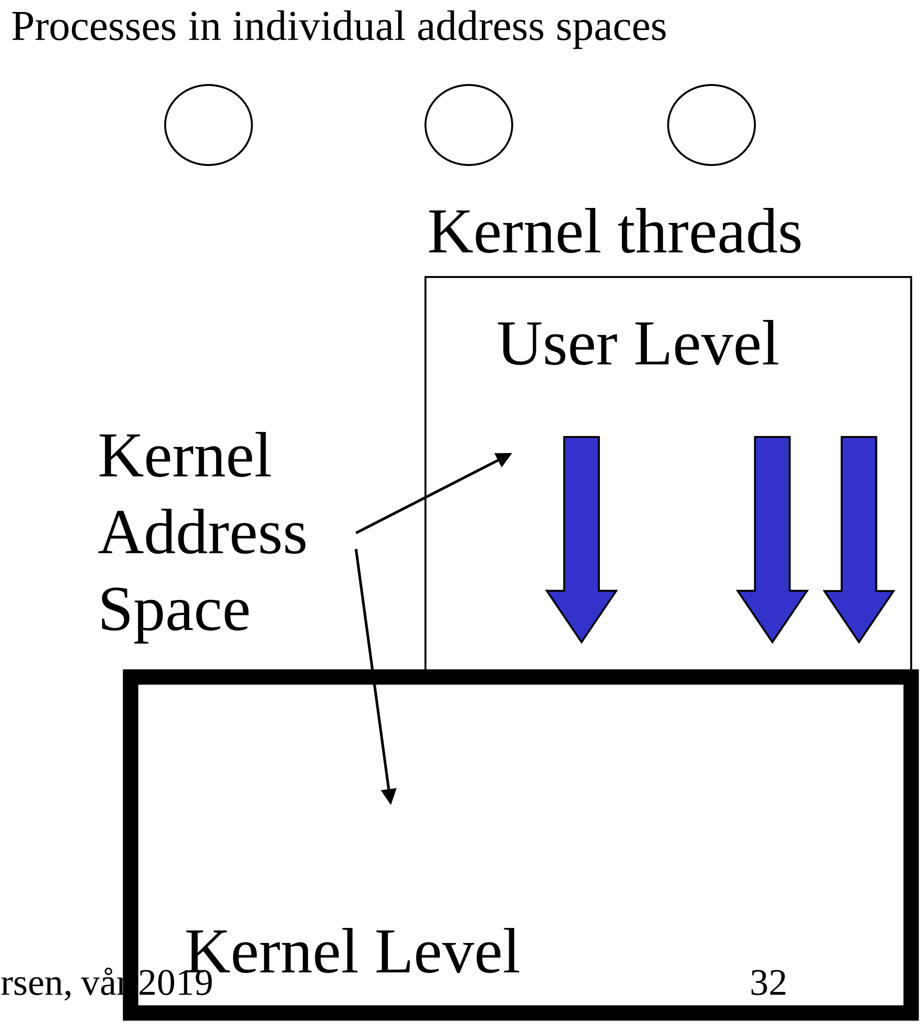
- **Save**(volatile machine state, current process);
- **Load**(another process’s saved volatile state);
- **Start**(new process);

Threads and Processes

Trad. Threads



Project OpSys



Links

- Virtual machine
    - http://whatis.techtarget.com/definition/0,,sid9_gci213305,00.html
- Exokernel
    - https://en.wikipedia.org/wiki/Exokernel
- THE
    - http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD196.PDF
- L4
    - http://os.inf.tu-dresden.de/L4/
- VM
    - http://www.vm.ibm.com/

Course Variants

- UiO – INF3151 – Thomas Plagemann
  - https://www.uio.no/studier/emner/matnat/ifi/INF3151/v19/timeplan/index.html#FOR
- Princeton – COS 318 – Jaswinder P. Singh
  - http://www.cs.princeton.edu/courses/archive/fall18/cos318/schedule.html