



T-106.420
Concurrent Programming
Implementation of the Kernel



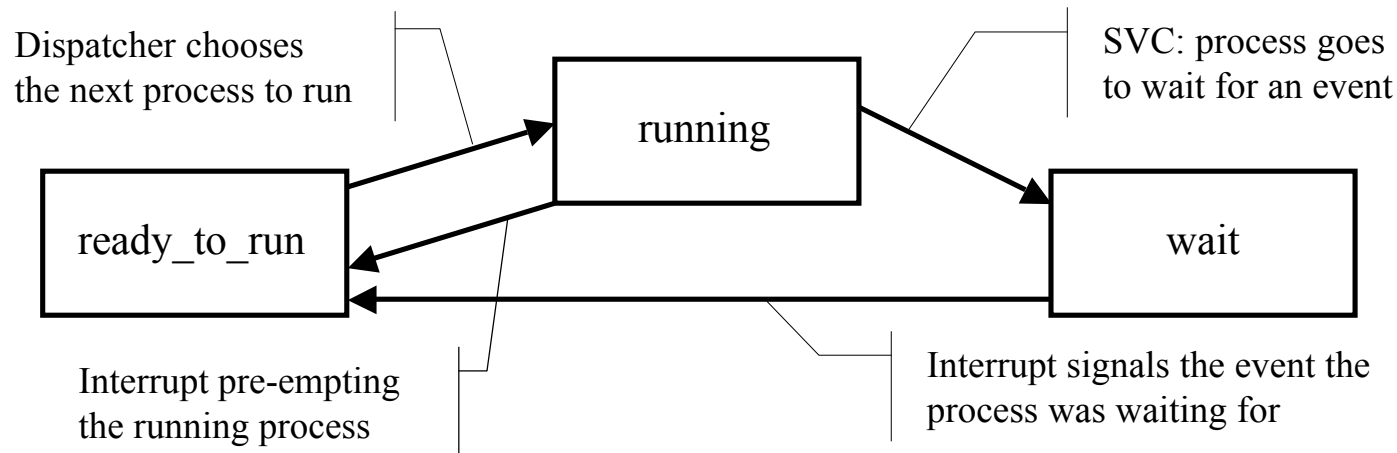
Implementing the Kernel

- It's about implementing the process (thread) abstraction
- Basic primitives:
 - `fork()` the create new (child) process
 - A special “hard wired” root process (mother of all others) will start the system at its bootstrap.
 - `quit()` terminate this process
 - `join()` wait for the child to terminate
- Basic data structures for each process:
 - In the kernel:
 - Process descriptor, i.e. metadata for the kernel to manage the process like execution state: running, waiting, ready-to-run
 - Processor image of “the virtual processor” of a non running process = register values, program counter, stack pointer etc.
 - In process' own address space:
 - stack and possible other private state data structures, heap...



Changes in the Process State

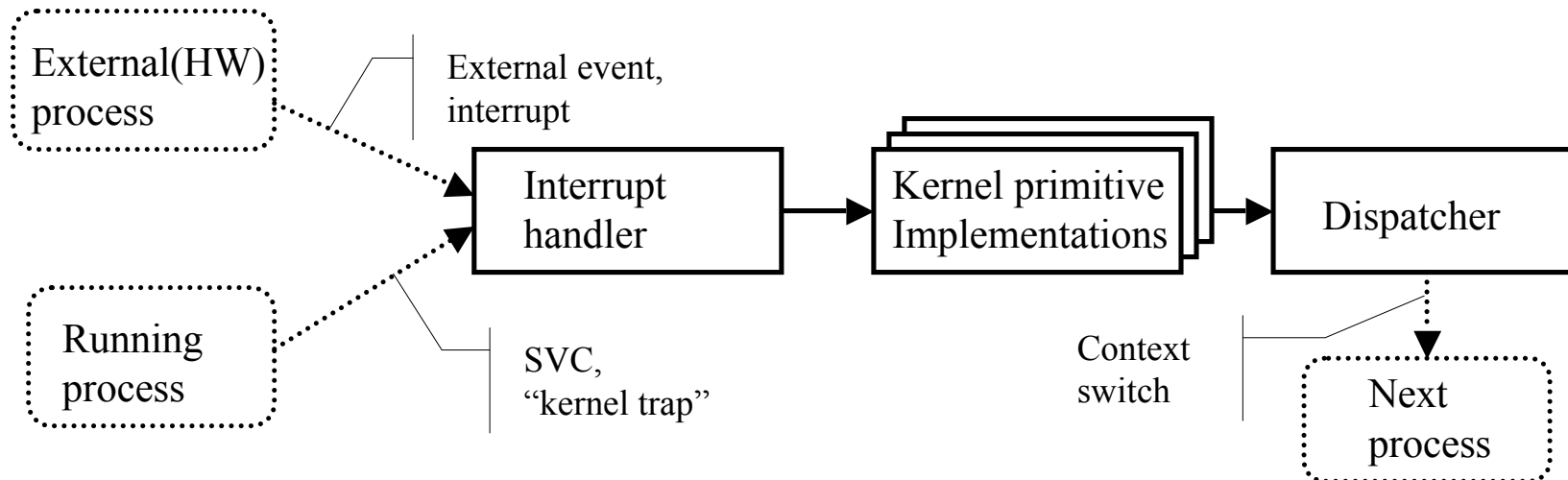
- Switch the processor from one process to another (s.c. multitasking) because of two reasons: SuperVisorCall (SVC) , i.e “kernel trap” and external event (interrupt)
- Process gives up the processor “voluntarily” by making a SVC, because it has to wait for an event. Process is put to a wait queue and the dispatcher chooses another one, which is ready-to-run. Context is switched to it.
- An external event causes an interrupt, pre-empting the running process and moving it to ready-to-run. It also moves a process waiting for the event to ready-to-run. Dispatcher chooses a new process from ready-to-run queue and switches context to it.





Flow of control in the Kernel

- Kernel takes care which process to run next. It's entered because two reasons:
 - Process gives up the process “voluntarily” by making a (SVC) causing a “kernel trap”
 - Process is pre-empted because some external event causes an HW-interrupt
- For security reasons Kernel operates in it's own address space:
 - Kernel is always entered via one interrupt “trap”, with all parameters, including the cause to enter, is passed via processor registers
- Interrupt handler examines the cause to enter and delegates control to the right kernel primitive. Before exit dispatcher chooses the next process to run and context is switched to it.
- A special dummy “idle process” (one per processor) is picked to run in case there is nothing else





Single processor vs. multiprocessor

- Complexity challenge: How to keep the consistency of the kernel data structures, like various process queues, and at the same time strive for fine grain atomicity to guarantee low kernel reaction latencies ?
- In a *single processor* disabling interrupts is an easy way to implement atomic actions. Straightforward solution to secure the kernel: Disable interrupts for the whole duration of all kernel operations.
- In a *multiprocessor* spin-locks with special HW, like test-and-set are needed to synchronize several processors
- In both cases non-constant time atomic operations on kernel data structures cause unpredictable latencies => Real-time response can not be guaranteed.
- In a multiprocessor long atomic operations may cause lot of busy waiting on spin locks implying memory contention and waist cycles => Waist of energy.
- In a multiprocessor one idle (slow running)process per processor is an easy way to cope temporary over abundance of processors.



Implementing semaphores in the kernel

- Data structure for each semaphore:
 - semaphore value(int)
 - waiting queue
- Operations implemented as kernel primitives:
 - `s = CreateSem(initvalue)`,
 - `Psem(s)`
 - `Vsem(s)`
- To provide low latencies and to reduce memory contention each semaphore should be protected with its own spin lock
- Monitor mechanisms, like mutual exclusion and condition queues can be implemented likewise



Levels of atomicity

- Policy vs. mechanism:
 - Lower level policy is a higher level mechanism
 - For the higher level mechanism means unavoidable “overhead”
 - On one level :
 - policy means “intelligence”, reflection and wise decisions: algorithm
 - how to minimize the overhead caused to higher levels and still run a vice policy ?

- Example levels:

- Specific synchronization algorithms: eg. dining philosophers, readers-writers, barriers etc.

- Synchronization primitives: Semaphores, monitor mechanisms

- Spin locks and interrupt disabling

- Machine instructions

- Application specific scheduling algorithms

- Strong fairness, e.g. FIFO

- Weakly fair scheduling

- HW-arbitration

↑
Longer interactions with
lower frequency and smaller
and more specific set of peers.
More intelligent scheduling
and overhead



Levels of abstraction and atomicity

- Challenge: Suppose we have enough processor capacity. How to provide required reaction times for external processes subject to external resources ?
- Minimize internal overhead caused by internal process interaction
- Provide appropriate scheduling decisions of external resources w.r.t external needs
- Process interaction requires shared data structures and atomic operations on them.
- Delay caused by atomic actions depends on two factors:
 - How long are the actions ?
 - How many processes are competing for them ?
- Levels of abstraction:
 - On a low level short and frequent interactions with minimal overhead shared between a large set of processes are often needed
 - On a higher level longer, less frequent, more elaborate interactions between a smaller, more dedicated set of processes can be afforded
 - Higher level interaction is based on lower level interactions.