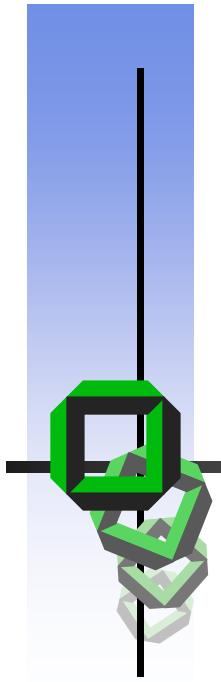


Systems Design and Implementation

I.6 – Threads and Scheduling

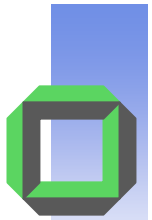
System Architecture Group, SS 2009

University of Karlsruhe

June 2, 2009

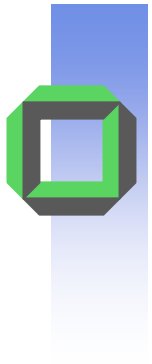
Jan Stoess

University of Karlsruhe



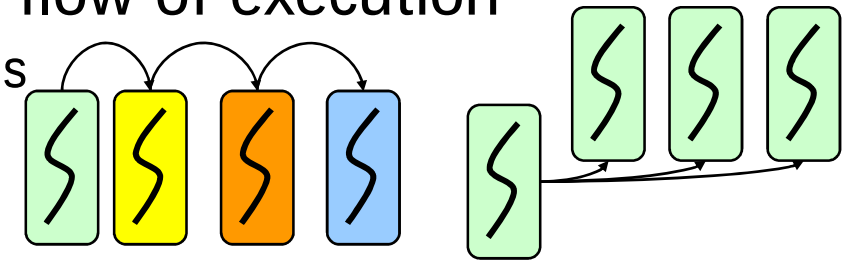
Overview

- Motivation
- Threads and Processes
  - Thread and Process Management
  - Usage scenarios
  - Program execution
- Thread Scheduling
  - Thread scheduling and accounting
  - Classic scheduling approaches
  - Scheduler activations et al.
- Multi-server systems
  - Scheduling issues in multi-server systems
  - Case study: Scheduling in K42

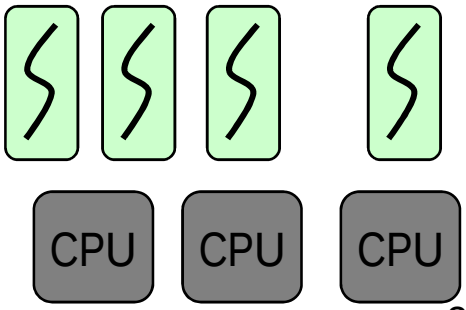


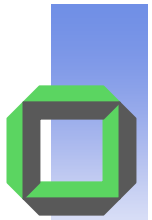
Motivation

- Definition of Thread
  - Short for “thread of execution”
  - Represents an independent flow of execution
- Purposes of threads
  - Expressing independent flow of execution
    - Dispatcher/Worker Models
    - Serialized Threads



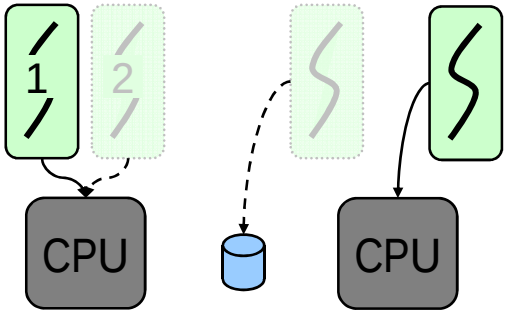
  - Expressing concurrency:
    - Multithreading via Time-Slicing
    - Multithreading via Multiprocessing

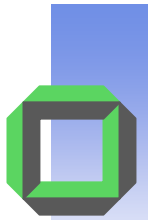




Motivation

- Purposes of threads
  - Resource management
    - Overlap I/O and CPU
    - Prioritize threads for QoS/RT/...
  - Security
    - Performance Isolation





Thread usage scenarios

- Structuring programs
  - Parallel loop

```
<parallel for> (i=1; i<n; i++)
          b[i] = (a[i] + a[i-1]) / 2.0
```

  - Parallel subprocedures

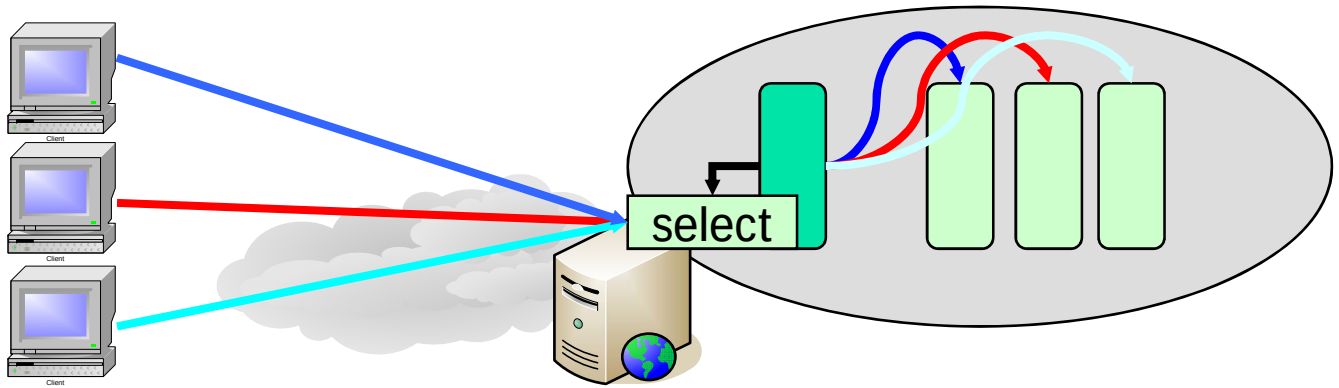
```
<parallel> {
    i_am = get_thread_num();
    n_threads = get_num_threads();
    /*
     * do stuff
     */
    }
```

Thread usage scenarios

- Structuring programs
  - Fork/Join

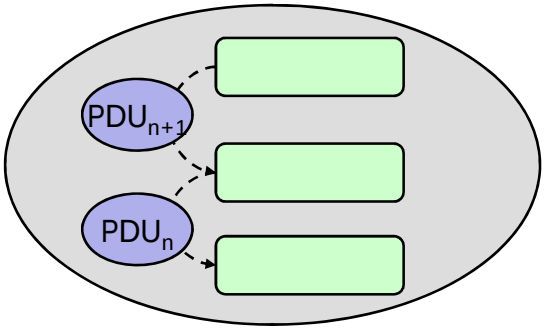
```
<parallel fork>  {
   aa = bb; // Unit 1
   cc = dd; // Unit 2
   ee = ff; // Unit 3
 } <parallel join>
```

  - Worker/Dispatcher
    - Example: Webserver
    - Dispatcher ready while work being done

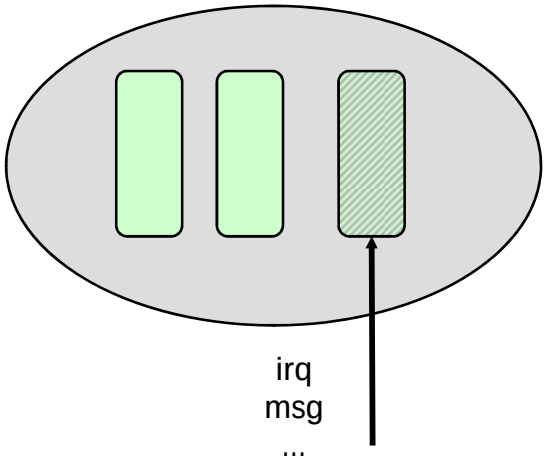


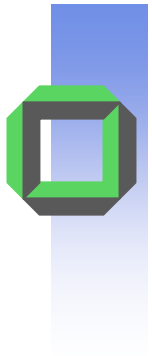
Thread usage scenarios

- Structuring programs
  - Pipelines
    - Difference to procedures?
    - Threads do parallel processing



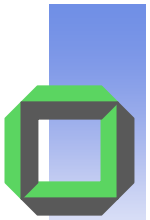
  - Signal/Call-back threads
    - Modeling asynchronous events
    - Thread can be created dynamically (pop-up thread)
    - Arriving messages, interrupts, ...





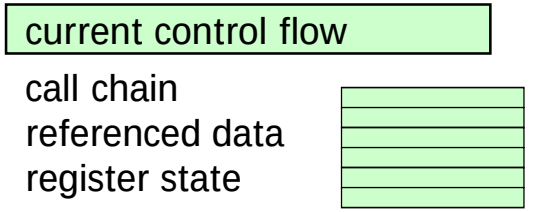
Motivation

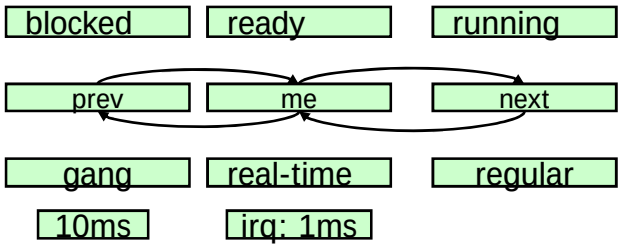
- Thread management
  - How to manage thread state
  - How to create, destroy, dispatch, ... threads
- Program execution
  - How to create threads from a file
- Thread scheduling
  - How to schedule threads among processors
- Thread accounting
  - How to track threads' processor usage
- Resource management
  - How to schedule and account threads on other resources

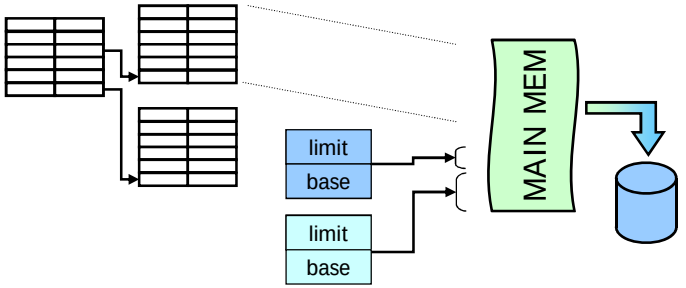


Thread and process management

- Thread state
  - Independent flow of *execution*
  - Execution state:
    - Instruction pointer
    - Stack pointer
    - ... Enough?
- Broader thread state
  - Scheduling/accounting state
    - thread state, ready queue
    - current processor
    - priority, scheduling class
    - timeslice, budget, latency
  - Memory state
    - reference to address space
    - (text/data/heap) segments
    - swap state

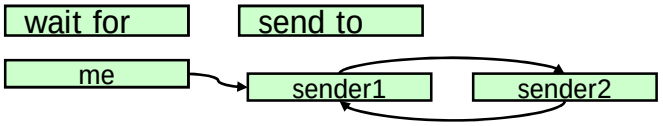


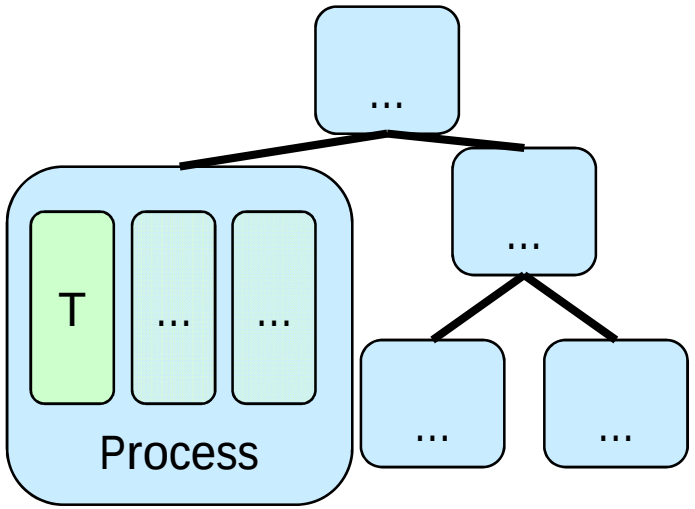




Thread and process management

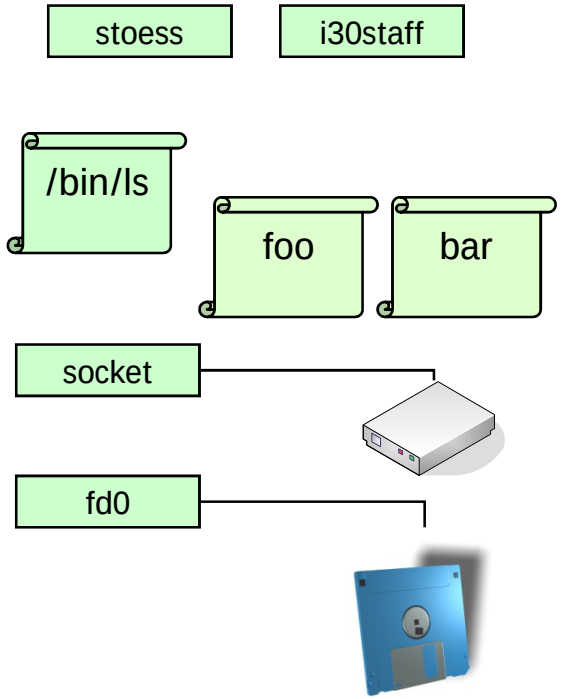
- Broader thread state
  - Communication state
    - Thread used as communication endpoint
    - communication state
    - wait queue, send queue
  - Process state
    - Process id
    - Process hierarchy

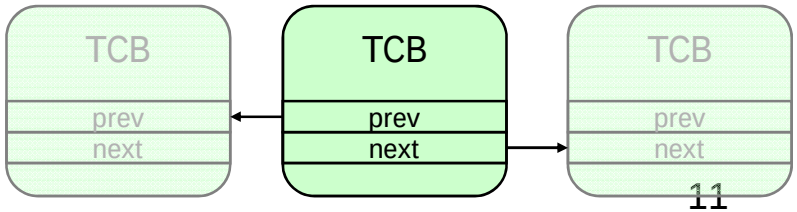


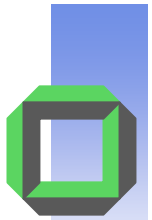


Thread and process management

- Broader thread state
  - Security
    - user, group, security class
  - Other resource state
    - File references
      - executing file
      - open files
    - I/O resources
      - Open network handles
      - Open peripheral devices
      - ...
- Storing thread state
  - Thread Control Block
  - Pointer lists
    - May be stored within TCB



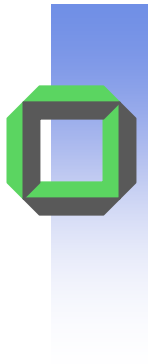




Thread and process management

- Basic Thread operations
  - Create
    - Like asynchronous procedure call
    - Allocate and initialize TCB
      - Initial IP, SP
    - Enqueue thread in ready queue
  - Startup
    - Remove from ready queue
    - Why not create and startup in one step?
  - Block (on resource, ipc, ...)
    - Save register state and IP on stack
    - Enqueue into wait queue
    - Update thread's state
    - Resume next thread

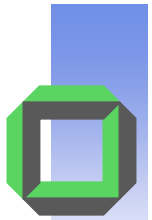
Source: T. Anderson et. al. ***The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors.*** IEEE Transactions of Computers 38:12 1989



Thread and process management

- Basic Thread operations
  - Signal Thread
    - Remove thread from wait queue
    - Place thread on ready queue
    - Update thread's state
  - Resume thread
    - Remove thread from ready queue
    - Restore register state
    - Continue executing at IP
  - Thread finish
    - Deallocate stack and TCB
    - Find and resume next thread

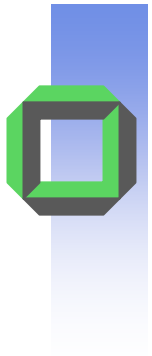
Source: T. Anderson et. al. ***The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors.*** IEEE Transactions of Computers 38:12 1989



Thread and process management

- Thread management considerations
  - Performance optimizations
    - Do not waste stack space
      - put initial Arguments in the TCB
    - Reduce overhead of finding free memory
      - Use free memory lists for stacks
      - Use free memory lists for TCBs
  - Synchronization
    - Must serialize concurrent access
    - Latency and throughput concerns
      - Latency: how fast is the uncontended case?
      - Throughput: how many operations per time are possible?

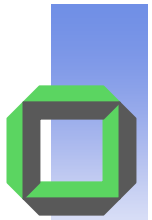
Source: T. Anderson et. al. ***The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors.*** IEEE Transactions on Computers 38:12 1989



Thread and process management

- Synchronization alternatives
  - Single lock
    - Single lock for all thread data structures
    - Low latency in non-contended case
    - Limits throughput
  - Multiple locks
    - Separate locks for ready queues, wait queues, free lists, ...
    - Higher latency, but better throughput expected
  - (Processor-)Local free lists:
    - Use local memory allocation pools
    - Reduces contention when allocating TCBs or stacks
    - Trades space for time (why?)
    - May induce additional memory allocation costs
    - May require to balance pools

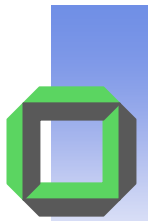
Source: T. Anderson et. al. ***The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors.*** IEEE Transactions on Computers 38:12 1989



Thread and process management

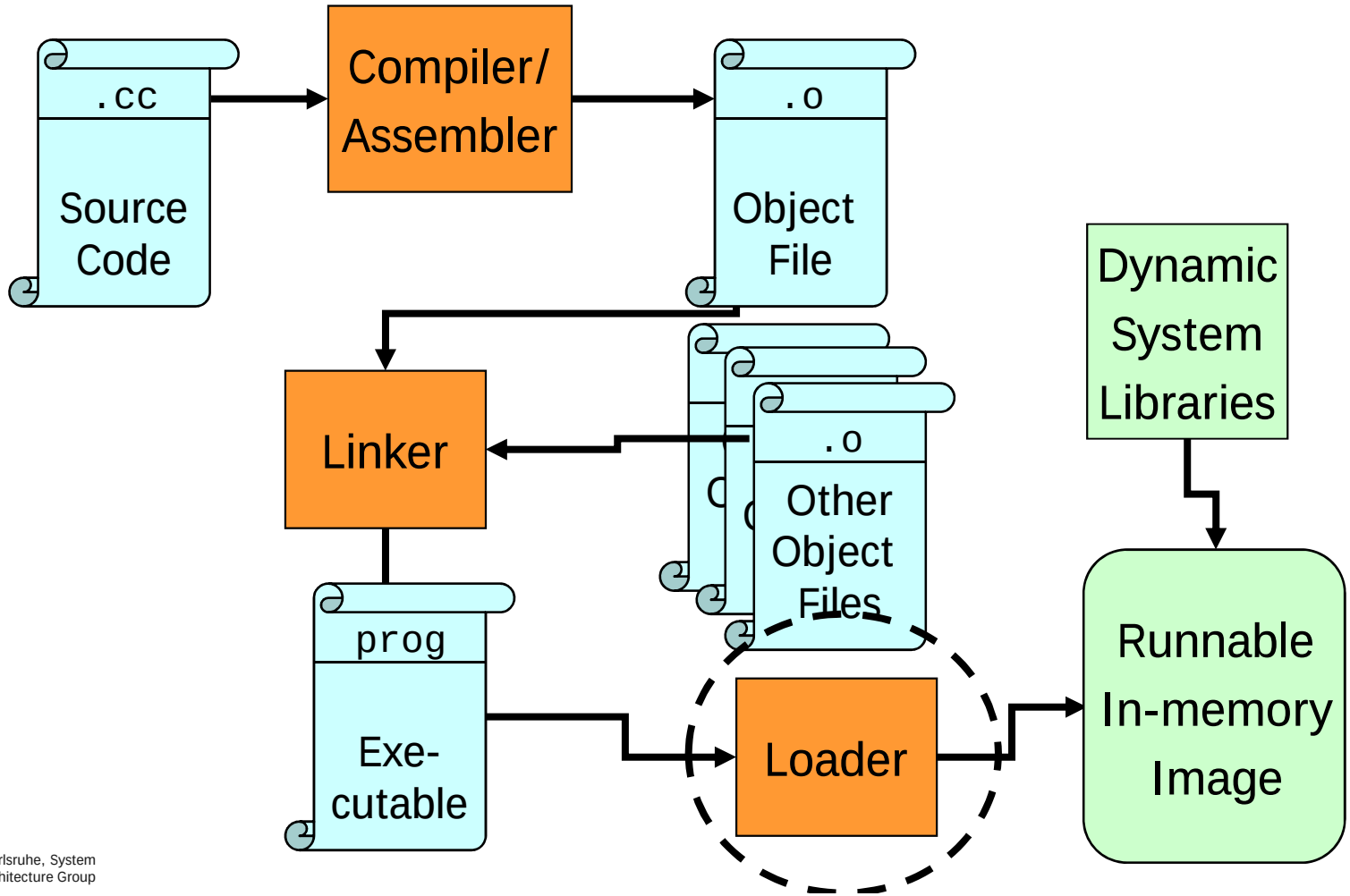
- Synchronization alternatives
  - Local ready queues:
    - Use local queues for starting/resuming threads
    - Reduces contention for queuing threads in the ready queue(s)
    - May require to balance thread load
    - Requires synchronization during load balancing
- Implementing synchronization
  - Cannot use threading (obviously)
  - Use spin locks
    - Locks are held shortly
  - Use hardware facilities
    - `cmpxchg` et al.
    - incurs bus locking overhead

Source: T. Anderson et. al. ***The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors.*** IEEE Transactions on Computers 38:12 1989



Program execution

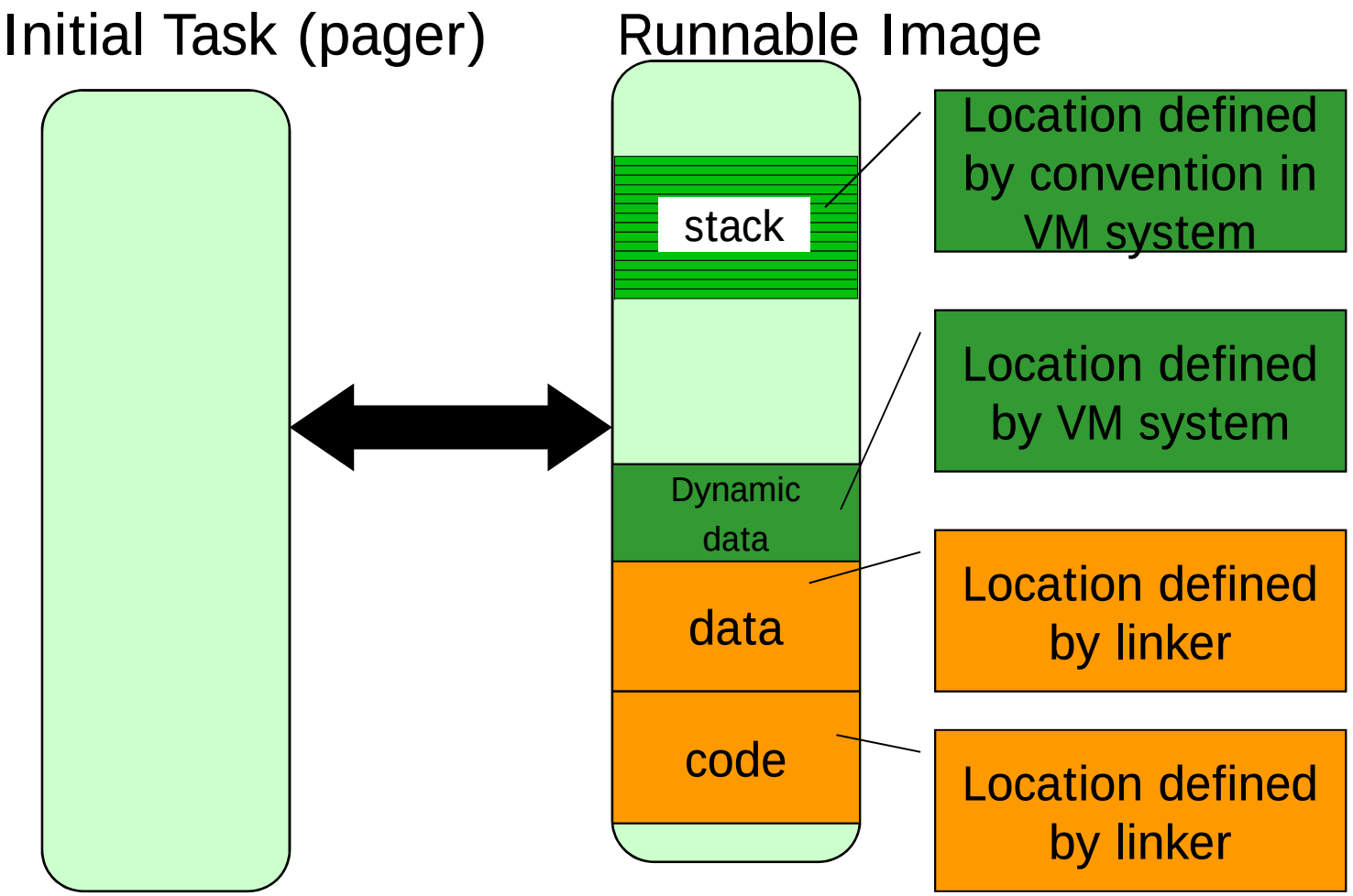
- The road to a running program:

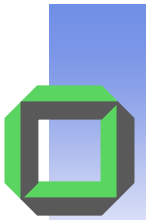


Program execution

- Loading an executable
  - What do we need to load?
    - Execution context
      - Initial instruction pointer
      - Initial stack pointer
      - Program arguments
    - Memory context
      - Code
      - Data
  - Where do we load it?
    - Static code/data specified by linker
      - Encoded in executable file
      - Developers can tweak linker to modify layout
    - Dynamic code/data specified by operating system
      - Virtual memory subsystem

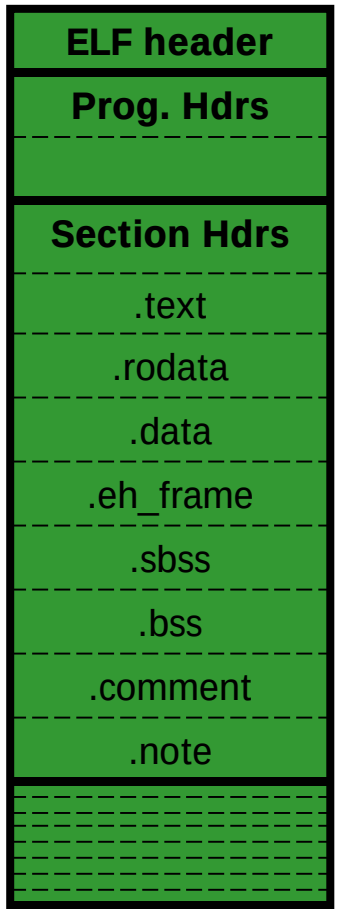
Example program layout

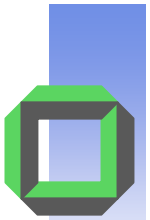




Executable file format

- Case study: ELF format
  - **E**xecutable and **L**inkable **F**ormat
  - Four major parts in ELF file
    - ELF header
      - roadmap
    - Program headers
      - describe segments directly related to program loading
    - Section headers
      - describe contents of the file
    - The data itself

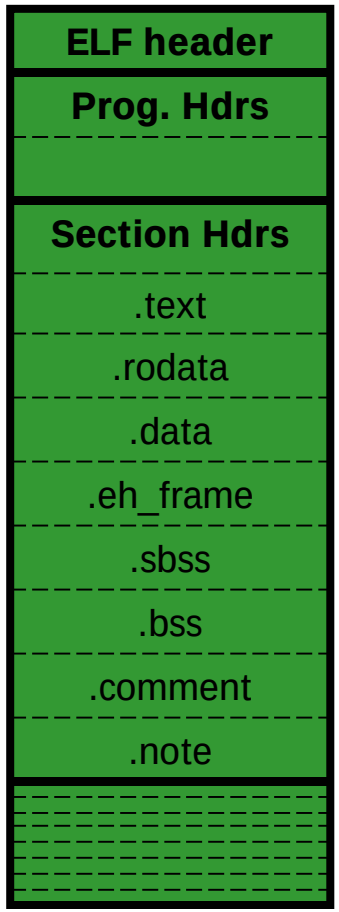


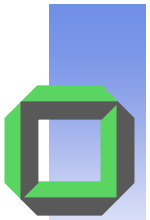


Executable file format

- ELF header

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT]; /* Magic number & other info */
  Elf32_Half    e_type;     /* Object file type */
  Elf32_Half    e_machine;  /* Architecture */
  Elf32_Word    e_version;  /* Object file version */
  Elf32_Addr    e_entry;    /* Entry point virtual address */
  Elf32_Off     e_phoff;    /* Program header table file offset */
  Elf32_Off     e_shoff;    /* Section header table file offset */
  Elf32_Word    e_flags;    /* Processor-specific flags */
  Elf32_Half    e_ehsize;   /* ELF header size in bytes */
  Elf32_Half    e_phentsize;/* Program header table entry size */
  Elf32_Half    e_phnum;    /* Program header table entry count */
  Elf32_Half    e_shentsize;/* Section header table entry size */
  Elf32_Half    e_shnum;    /* Section header table entry count */
  Elf32_Half    e_shstrndx; /* Section header string table index */
} Elf32_Ehdr;
```

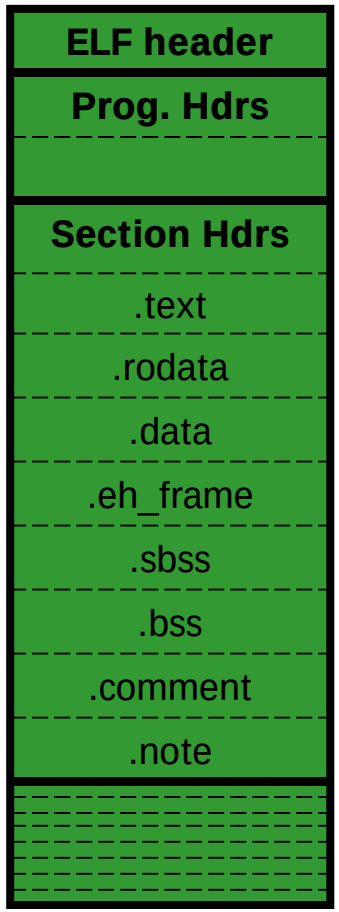




Executable file format

- Section headers

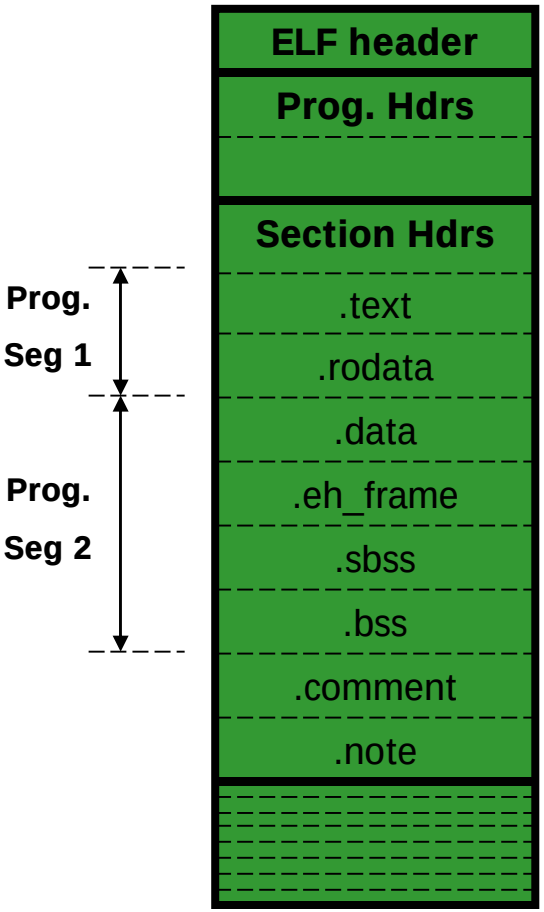
```
typedef struct
{
  Elf32_Word    sh_name;  /* Section name (str tbl index)*/
  Elf32_Word    sh_type;  /* Section type */
  Elf32_Word    sh_flags; /* Section flags */
  Elf32_Addr    sh_addr;  /* Section virtual addr */
  Elf32_Off     sh_offset;/* Section file offset */
  Elf32_Word    sh_size;  /* Section size in bytes */
  Elf32_Word    sh_link;  /* Link to another section */
  Elf32_Word    sh_info;  /* Additional section info */
  Elf32_Word    sh_addralign; /* Section alignment */
  Elf32_Word    sh_entsize; /* Entry size if section holds
                               table */
} Elf32_Shdr;
```

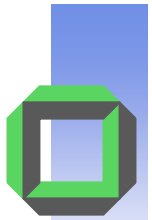


Executable file format

- Program headers

```
typedef struct
{
  Elf32_Word    p_type;   /* Segment type */
  Elf32_Off     p_offset; /* Segment file offset */
  Elf32_Addr    p_vaddr;  /* Segment virt. address */
  Elf32_Addr    p_paddr;  /* Segment phys. address */
  Elf32_Word    p_filesz; /* Segment size in file */
  Elf32_Word    p_memsz;  /* Segment size in mem */
  Elf32_Word    p_flags;  /* Segment flags */
  Elf32_Word    p_align;  /* Segment alignment */
} Elf32_Phdr;
```

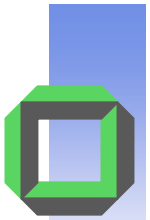




Executable file format

- Example: **`test_client`** section headers

```
$ objdump -h test_client
testclient:     file format elf32-i386
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000200d  00300000  00300000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000004b4  00302020  00302020  00003020  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00002038  00303000  00303000  00004000  2**12
                  CONTENTS, ALLOC, LOAD, CODE
  3 .ctors        00000000  00306000  00306000  00007000  2**0
                  CONTENTS
  4 .dtors        00000000  00306000  00306000  00007000  2**0
                  CONTENTS
  5 .bss          00000404  00306000  00306000  00007000  2**2
                  ALLOC
  6 .debug_abbrev 000008b5  00000000  00000000  00007000  2**0
                  CONTENTS, READONLY, DEBUGGING.
```

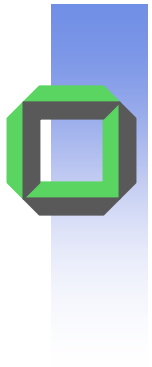


Executable file format

- Example: **`test_client`** program headers

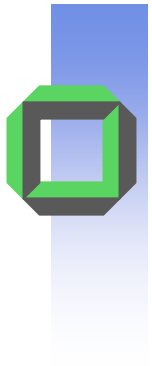
```
$ objdump -p test_client

testclient:     file format elf32-i386

Program Header:
    LOAD off    0x00001000 vaddr 0x00300000 paddr 0x00300000 align 2**12
         filesz 0x000024d4 memsz 0x000024d4 flags r-x
    LOAD off    0x00004000 vaddr 0x00303000 paddr 0x00303000 align 2**12
         filesz 0x00002038 memsz 0x00003404 flags rwx
$
```

- Note: memsz > filesz
  - implicit .bss segment
    - ~~batman's shameful secret~~
    - block started by symbol
  - Reserved but uninitialized data
  - Must be zero-filled



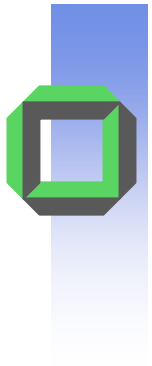
Thread scheduling and accounting

- Threads imply scheduling problems
  - Who runs next on a processor?
  - Which processor should a thread run?
  - How long should a thread run?
  - ...
- Wanted: separation of policy & mechanisms
  - Mechanism:
    - (re-)dispatching, preemption, migration, accounting
  - Policy:
    - Allocation, Budgeting, priorities, scheduling classes latency constraints, ...



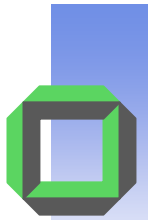
Thread scheduling and accounting

- Problems:
  - Scheduling may span multiple subsystems and layers
    - Distributed/hierarchical scheduling
    - E.g., Web server: application-directed, OS-enforced scheduling
    - distributed resource managers → distributed accounting
  - Scheduling is tied to other OS-abstractions
    - *may imply* scheduling decision
    - Examples
      - blocking I/O, IPC, interrupts, exceptions, ...
      - resource consumption/exhaustion (memory, energy, ...)
    - Policy and mechanism hard to distinguish



Thread scheduling and accounting

- Problems:
  - Scheduling policies are complex
    - Different environments
    - Different policies
    - Multidimensional problems
  - Performance implications
    - Scheduling distributed and entangled
    - Diverse but frequently invoked scheduling-related services
    - Need simple and clean abstractions
    - Separation of policy and mechanism may be complex and inefficient



Thread scheduling policies

- Processor-based scheduling
  - Service time
    - Working set, execution signature
  - Processor-internal characteristics
    - Performance counters, HW sensors
- Processor-associated resources
  - Caches, Memory
  - Pinned components (e.g., devices, drivers)
- Sharing
  - Communication facilities
  - IPC, shared memory, ...
- Load balancers
  - Run queue length
  - Idle time
  - Context switch rates

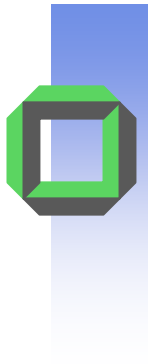
Time-Based Scheduling

Energy-aware Scheduling

Affinity Scheduling

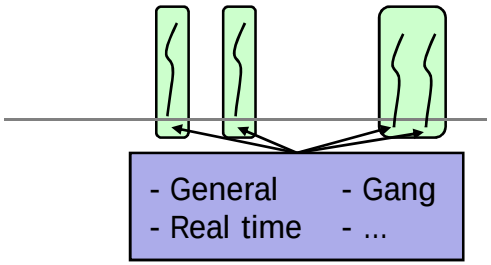
Co-/Gang-Scheduling

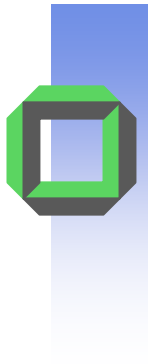
Load Balancing



Traditional scheduling approaches

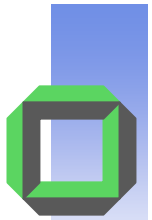
- Kernel level scheduling
  - Kernel provides a notion of executable entity
    - Kernel threads, processes, tasks, ...
    - Execution context initialized by the user
  - Kernel responsible for threads
    - Management
    - Scheduling
    - Dispatching
  - Central, in-kernel management
    - Global state and visibility
    - Low interaction with other subsystems/layers
    - Kernel-managed scheduling policies (+tweaking)
  - Execution contexts entangled with many other abstractions
    - Memory protection
    - Accounting
    - Communication
    - Other resources (files, I/O, ...)





Traditional scheduling approaches

- Analysis
  - Global scope
    - All applications are subject to scheduling
    - Kernel can enforce scheduling decisions
  - Low-overhead scheduling
    - central scheduling state
    - no boundary crossing needed
  - One-fits-it-all approach
    - Fixed scheduling policies, tweaking at most
    - Oblivious to application requirements
    - Problems:
      - Competing jobs? Different job requirements?
      - Different number of threads/job?
      - Different application QoS?



Traditional scheduling approaches

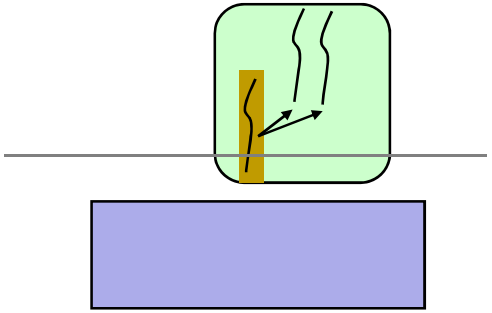
- Analysis
  - Oblivious to special environments
    - Linux as desktop system, Linux as scalable server?
    - response time vs. throughput
  - Complex and heavyweight abstractions
    - Process switch changes scheduling context, memory context, I/O context,....
  - OS oblivious to application boundaries
    - application spanning multiple processes, threads, address spaces?
  - Extensibility is difficult
    - Scheduling is deeply embedded within the kernel
    - No modularization, no component boundaries

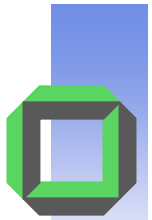
Further reading: *H. Franke et al.* ***PMQS: Scalable Linux Scheduling for High End Servers.*** *Proceedings of the 5th Annual Linux Showcase and Conference, 2001*

G. Banga et al. ***Resource containers: A new facility for resource management in server systems***. Proceedings of the 3rd Symposium on Operating Systans Design and Implementation (OSDI-99).

Traditional scheduling approaches

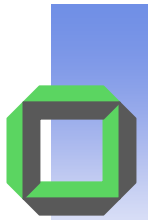
- Application level scheduling
  - Application provides *its own thread* notion
    - Transparently mapped one or more kernel-provided execution contexts
  - Application responsible for
    - Thread management
    - Thread scheduling
    - Thread dispatching
  - Decentralized, application-wise management
    - Local state and visibility
    - *No* interaction with other subsystems/layers
    - Arbitrary policies and extension possible





Traditional scheduling approaches

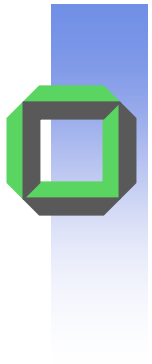
- Analysis
  - Library approach
  - Very low overhead abstractions
    - Only within application
    - Thread switch *only switches* execution context
  - Can respect application-level QoS
  - Local scope
    - OS-agnostic threads
    - Application scheduler cannot respect
      - OS-Kernel
      - Other subsystems
      - Other applications
    - And vice versa
    - Examples:
      - Blocking on I/O
      - Component-based systems



Scheduler activations

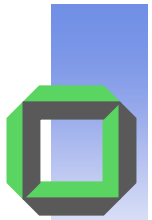
- Kernel-level threads alone
  - Heavy-weight abstractions
  - Not extensible, not customizable
- User-level threads alone
  - Oblivious to OS activity (pagefaults, I/O)
  - Oblivious to multi-programming
- Idea:
  - *Combine* user and kernel threads

Source: *T. E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy* ***Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism***. ACM Transactions on Computer Systems, Vol. 10, No. 1, February 1992, Pages 53-79.



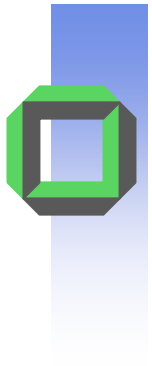
Scheduler activations

- Basic assumption:
  - Common case
    - Thread operations without kernel intervention
    - Communication, synchronization
    - Use (pure) user-level threads
  - Infrequent case
    - Kernel intervention needed
    - E.g., processor reallocation, page faults, ...
    - Use user-level threads but mimic kernel thread behavior
    - When a thread blocks, the processor can run another thread
- Requirement: Distributed scheduling control
  - Kernel needs to know application state
    - How much parallelism does the application contain?
  - Applications need to know kernel scheduling state
    - When does a thread block?



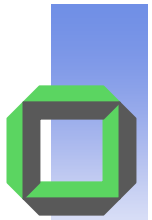
Scheduler activations

- Approach:
  - Abstractions:
    - Kernel provides virtual multiprocessors
    - Address spaces have a dedicated scheduler
  - Mechanisms:
    - Scheduler notifies the kernel on the thread operations *that affect processor allocation*
    - Kernel notifies scheduler on *all address-space-related* kernel scheduling events
    - The latter is termed *scheduler activation*



Scheduler activations

- Virtual processors (VPs):
  - One or more VPs per address space
  - AS scheduler freely allocates threads to VPs
  - Kernel vectors re-allocations to the AS scheduler
  - AS scheduler notifies kernel if it needs more/less VPs
- Upcall points:
  - Processor added
  - Processor preempted
  - Scheduler activation blocked
  - Scheduler activation unblocked

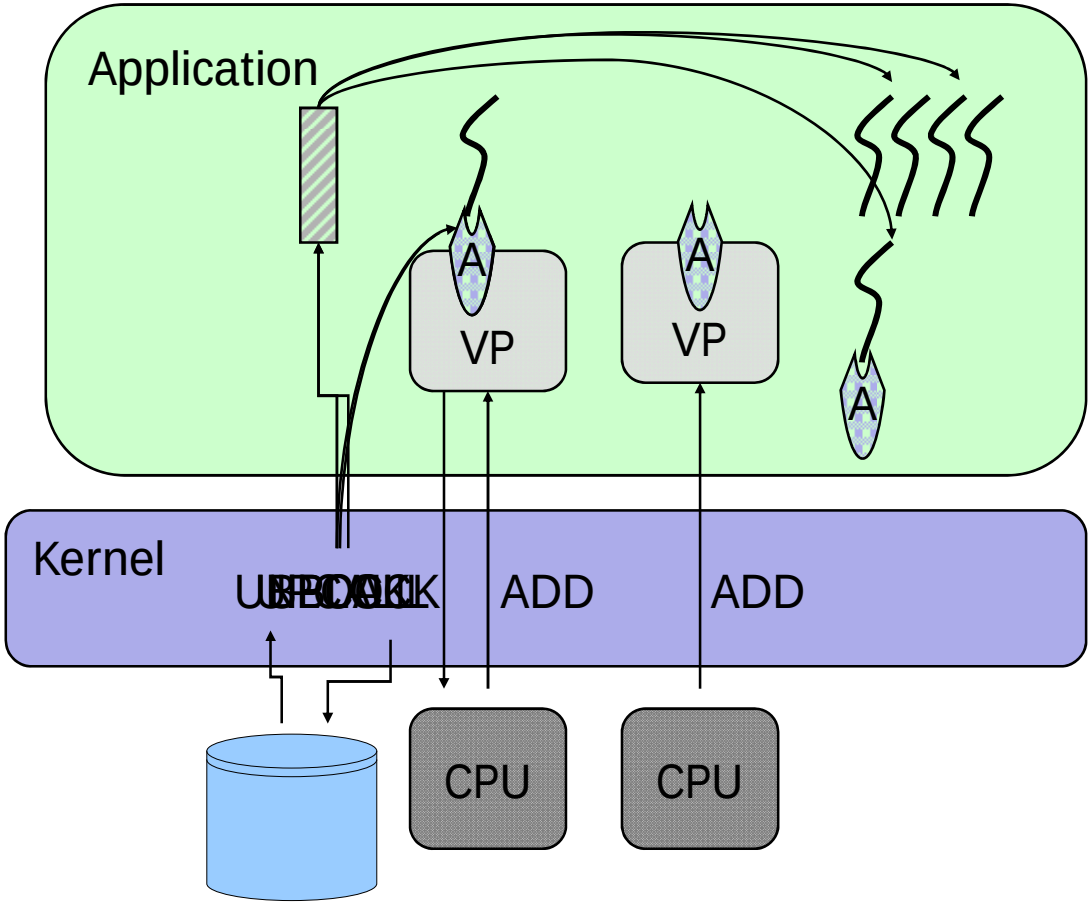


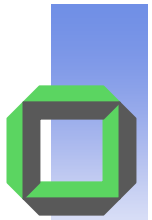
Scheduler activations

- Scheduler activation:
  - Execution context for user-level threads
  - Notification system for kernel
  - Scratch space for saving user-level contexts
- Application start:
  - Kernel creates scheduler activation
  - Assigns activation to a virtual processor
  - Upcalls into application at fixed entry (scheduler)
  - Scheduler initializes itself
  - Scheduler dispatches the first user-level thread to activation
- Scheduling-related kernel-events
  - Kernel creates a *new* scheduler activation
  - Assigns the activation to the VP
  - Upcalls into application at fixed entry (scheduler)
  - Scheduler processes the event

Scheduler activations

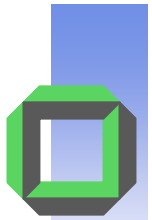
- Example: I/O request





Scheduler activations

- Conclusion & Analysis
  - Basic idea:
    - Combined application-level and kernel-level scheduler
    - Synchronous vectoring to notify user level scheduler
  - Extensible, keeps policy out of the kernel
    - Kernel only *dispatches*
    - Users may develop arbitrary policies
  - Limited to threads within a single address-space
    - One user-level scheduler per address-space
    - User-level scheduler can not dispatch threads in other protection domains
  - Performance penalties
    - Needs 2 user-kernel transitions per upcall
    - Depends on the number of scheduling related events

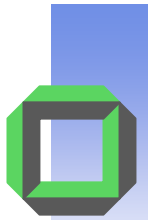


Scheduler activations: similar approaches

- Process control
  - Improve multiprocessing in multiprogrammed systems
    - Fair behavior while maximizing throughput
  - Basic idea:
    - Application performance best if
      - #application processes = #processors
    - Employ centralized scheduler
      - Calculates optimal number of processes for each application
      - Requests applications to dynamically change their number of processes
    - Relies on polling and cooperativeness of applications

Sources: *A. Tucker et al.* ***Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors.*** Proceedings of the 12th ACM Symposium on Operating Systems

B.D. Marsh et al. ***First-Class User-Level Theads****.* Proceedings of the 13th ACM Symposium on Operating Systems Principles, Operating Systems Review, 25(5), pp. 110-121, 1991

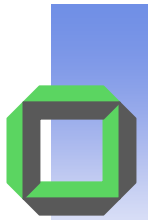


Scheduler activations: similar approaches

- 1st class user-level threads
  - Kernel-threads are heavy-weight
  - User-level threads have a second-class status
    - Not known by kernel
    - Lack of interaction methodology *between different* user-level thread packages
  - Basic idea:
    - Grant user-level threads 1st-class status
    - Use shared memory for interaction between kernel and user
    - Use software-interrupts for events that require synchronous interaction
    - Develop scheduler interface convention for interaction between schedulers

Sources: *A. Tucker et al.* ***Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors.*** Proceedings of the 12th ACM Symposium on Operating Systems

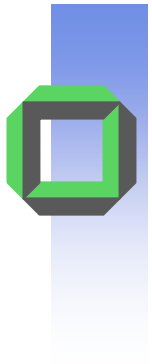
B.D. Marsh et al. ***First-Class User-Level Theads****.* Proceedings of the 13th ACM Symposium on Operating Systems Principles, Operating Systems Review, 25(5), pp. 110-121, 1991



Scheduler activations: similar approaches

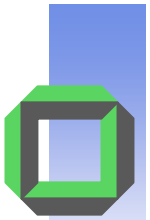
- CPU inheritance scheduling
  - Most OSes only support rigid scheduling
    - Set of scheduling classes
    - Implementations tied together in the OS
  - Basic idea:
    - Threads can *donate* their time to other threads
    - Threads can wait for timing events
      - interrupts, timers, blocking of time donatees,,,
    - Hierarchical control over processing time
    - Threads schedule themselves
    - Allow for arbitrary relations and administrative domains
  - Limitations:
    - Timing hard to virtualize
    - Not fully evaluated

Source: *B.Ford et al.* ***CPU inheritance scheduling.*** *Proceedings of the 2nd Usenix Symposium on Operating Systems Design and Implementation (OSDI), p91-105. 1996*



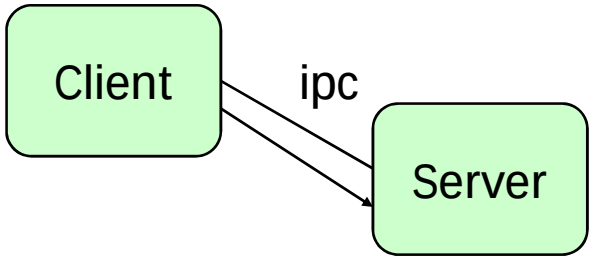
Scheduling in multi-server systems

- Threads represent different entities
  - Servers, Clients, Resources
- IPC is “the” mechanism for everything
  - Requests
  - Dispatching
  - Synchronization
  - Interrupts
  - Resource allocations
  - Permission faults
- IPC is tied to scheduling
  - IPC operations may block/unblock entities
    - Who runs after a unblocking operation?
    - Who runs after a blocking operation?
  - IPC operations may imply scheduling decisions
    - Depends on the usage scenario



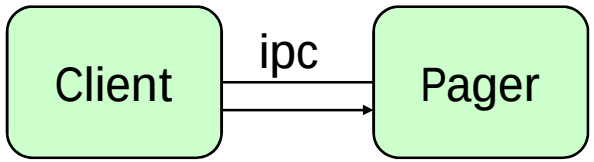
Scheduling in multi-server systems

- Server requests

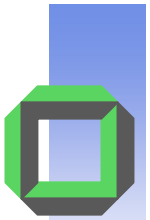


- Servers model resources
- Requests imply resource donation
- Do not change scheduling context on ipc

- Resource faults

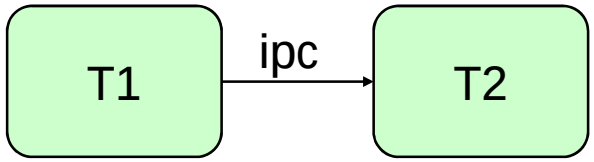


- Resource managers shouldn't run in clients' contexts only
- They may have insufficient resources
- Do change client contexts



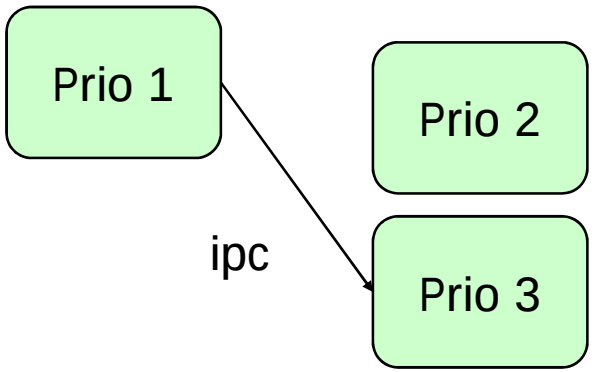
Scheduling in multi-server systems

- Synchronization

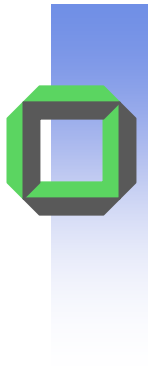


- Threads are independent
- IPC used for notification
- Do change scheduling context

- RT subsystem

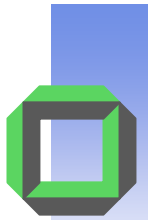


- May want strict priorities
- Perform scheduling decision during IPC



Scheduling in multi-server systems

- The Dilemma:
  - Policy should be controlled by applications
  - IPC performance is critical
- Alternatives:
  - Policy upcalls:
    - Vector every scheduling event (i.e., every IPC)
    - expensive
  - Type-safe description languages:
    - Download scheduling code into the kernel
    - complex, insecure, no timing guarantees
  - Default policies (L4 state of the art behavior)
    - Pistachio:
      - Kernel-level round robin scheduler
      - Always donate time slice
    - Fiasco:
      - In-kernel RT scheduling policy
    - Fast but often inappropriate



Case study: scheduling in K42

- K42 is
  - A high performance, open source, general-purpose research operating system kernel for cache-coherent multiprocessors
  - Linux-compatible
- Main goals:
  - Scalability and performance
  - Adaptability
  - Extensibility and maintainability
  - Open-source compatibility
- Approach:
  - Modular, object-oriented code
  - No centralized code-paths, global data structures, locks
  - Move system functionality from kernel to servers and users

Case study: scheduling in K42

- K42 structure
- User-mode libraries
  - Thread scheduling
  - Linux emulation
  - Application libraries (GLIBC, ...)
- System layers
  - NFS
  - K42 scalable FS
  - Name server
  - Socket server
  - Pipe server
- Kernel
  - Memory management
  - IPC
  - Base scheduling
  - Networking
  - Devices

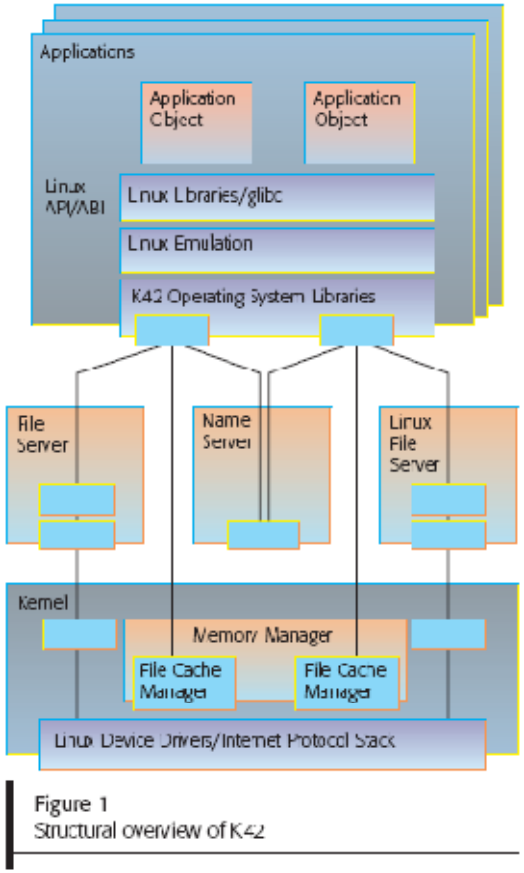
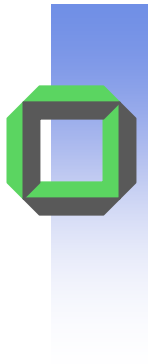


Figure 1
Structural overview of K42

Sources J. Appavoo et al. ***Experience with K42, an Open Source, Linux-compatible, Scalable Operating-system Kernel.*** *IBM Systems Journal, 44:2 2005*



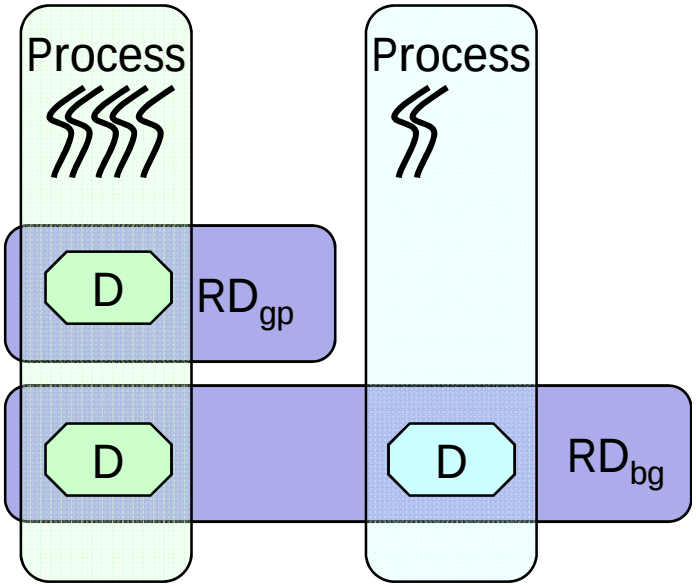
Case study: scheduling in K42

- Scheduling in K42
  - Partitioned between kernel and user
  - Processes consist of
    - One address space and
    - One or more dispatchers
  - Kernel schedules *dispatchers*
    - Uses kernel resources (e.g., pinned memory)
  - *Dispatchers* schedule threads
    - Threads are oblivious to the kernel
    - Applications have a customizable thread model
    - Can create arbitrary number of threads
    - Can not exhaust kernel resources
  - Multiple dispatchers
    - Attain parallelism
    - Establish different scheduling paradigms (QoS, Prio, ...)
  - Similar to scheduler-activations

Source: M. Auslander et al. ***Scheduling in K42***. IBM Research, http://www.research.ibm.com/K42, August 2002

Case study: scheduling in K42

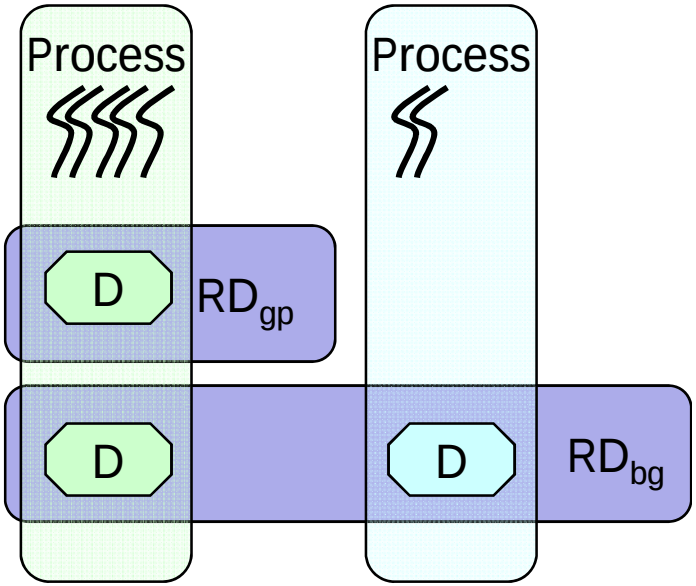
- Kernel scheduling
  - Dispatchers
    - bound to a processor
    - belong to *resource domains*
  - Resource domains
    - own resource rights (e.g., processor share)
    - are accountable entities
    - belong to one of five *scheduling classes*



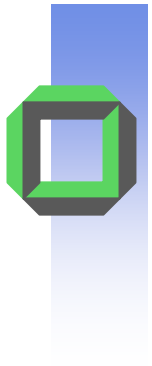
Source: M. Auslander et al. ***Scheduling in K42***. IBM Research, http://www.research.ibm.com/K42, August 2002

Case study: scheduling in K42

- Kernel scheduling
  - Scheduling classes
    1. system, hard real time
    2. gang-scheduled
    3. soft real time
    4. general purpose
    5. Background
    - Classes are strictly prioritized
    - Within scheduling classes
      - Weighted proportional sharing
      - Admission controls for real time and gang scheduling
  - On scheduling events
    - Kernel chooses resource domain based on kernel policies
    - Kernel chooses a dispatcher from that domain
    - Kernel *does not ensure fairness* within a domain



Source: M. Auslander et al. ***Scheduling in K42***. IBM Research, http://www.research.ibm.com/K42, August 2002



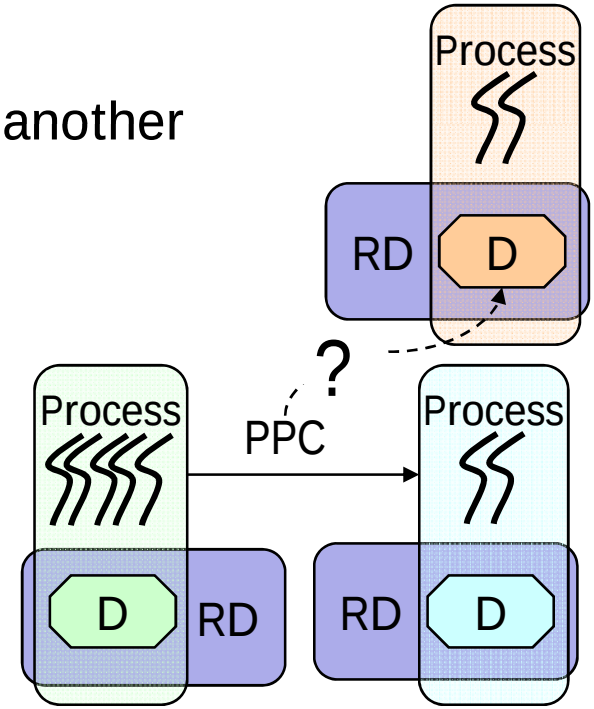
Case study: scheduling in K42

- Kernel scheduling
  - Soft preemptions:
    - Approach:
      - Upcall to dispatcher
      - Dispatcher saves its state
      - Dispatcher yields voluntarily after short time
    - Assumptions:
      - Current and next dispatcher reside in the same scheduling class **and**
      - Current dispatcher is well-behaving
  - Hard preemptions:
    - Assumptions:
      - Current dispatcher resides in a lower-prioritized class than the next dispatcher **or**
      - Current dispatcher hasn't responded timely to upcall

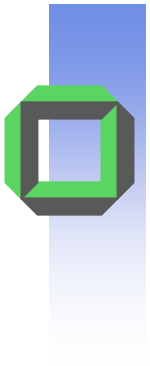
Source: M. Auslander et al. ***Scheduling in K42***. IBM Research, http://www.research.ibm.com/K42, August 2002

Case study: scheduling in K42

- Kernel IPC
  - Protected procedure call (PPC) semantics
  - RPC invocation and return plus address space switch
    - syscall → switch AS → sysret
- Kernel IPC and scheduling:
  - PPC is an explicit handoff from one to another dispatcher
  - How to integrate this handoff?
  - Pure Variant 1: *strict scheduling*
    - Switch dispatcher
    - Switch resource domain
    - Pursue scheduling decision in between
  - Analysis
    - Respects kernel scheduling
    - Degrades performance!

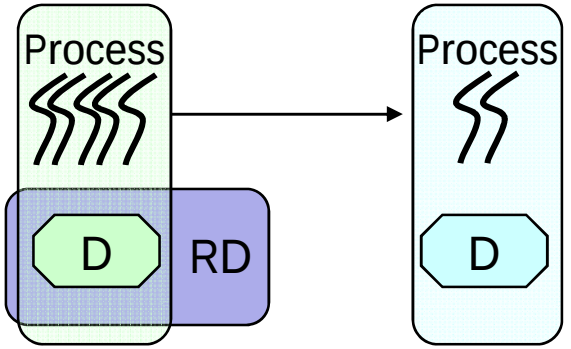


Source: M. Auslander et al. ***Scheduling in K42***. IBM Research, http://www.research.ibm.com/K42, August 2002

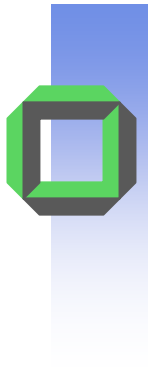


Case study: scheduling in K42

- Kernel IPC and scheduling:
  - Pure Variant 2: *resource donation*
    - Switch dispatcher but
    - Keep caller's resource domain
      - Clients donate their domain
      - Server's consumption accounted to clients
  - Analysis
    - Allows for efficient implementation
    - Complex
      - Kernel must schedule server dispatchers in client domains
      - Servers must be able to switch resource domains
    - Biggest problem: Priority inversion (!)
      - Resource-constrained client calls server
      - Server acquires a lock and suffers resource exhaustion
      - Only solution: enhance locking; too expensive



Source: M. Auslander et al. ***Scheduling in K42***. IBM Research, http://www.research.ibm.com/K42, August 2002



Case study: scheduling in K42

- Kernel IPC and scheduling:
  - Final approach: mix variants 1 and 2
    - Switch dispatcher
    - Keep caller's resource domain *until next in-kernel scheduling*
    - Then switch resource domain
  - Analysis:
    - Efficient in the common case
    - Avoids priority inversion
    - Reduces the precision and determination of accounting

Source: M. Auslander et al. ***Scheduling in K42***. IBM Research, http://www.research.ibm.com/K42, August 2002

Case study: scheduling in K42

- Kernel/Dispatcher interface
  - Dispatcher structure
    - Mostly based on shared memory
    - Disable-interrupts bit
    - Pending interrupts vector
    - Machine-state save area
    - Control registers, message buffers
  - Current dispatcher location
    - Stored in read-only page at fixed virtual address
    - Different mappings per processor
  - Entry points
    - Code addresses for different kernel events
    - Changed via system call
    - Initialized by dispatcher

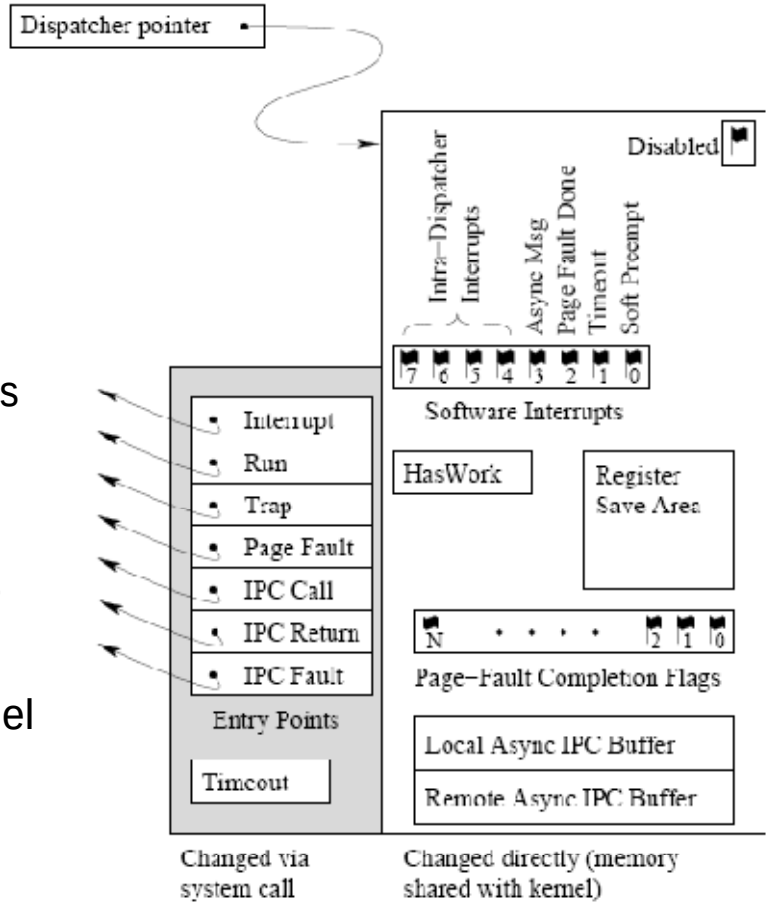
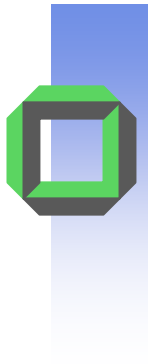


Figure 2. Dispatcher Structure

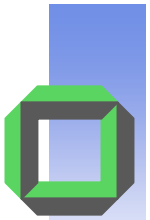
Source: M. Auslander et al. ***Scheduling in K42***. IBM Research, http://www.research.ibm.com/K42, August 2002



Case study: scheduling in K42

- User-Level Scheduling
  - Library code providing thread implementations
  - Thread object
    - Contains current Stack pointer
    - Other state saved on stack
  - CurrentThread
    - Points to currently running thread
    - Can be special register or hard-coded virtual address
  - ThreadIDs
    - 64-bit handle
    - Identifies thread and dispatcher
      - ID changes on migration

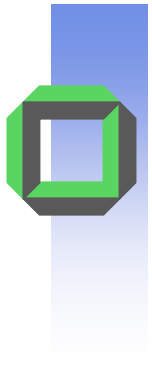
Source: M. Auslander et al. ***Scheduling in K42***. IBM Research, http://www.research.ibm.com/K42, August 2002



Case study: scheduling in K42

- User-Level Scheduling
  - Thread operations
    - Creation
      - Allocate thread object from free list, initialize it
      - Put into dispatcher ready queue
    - Block
      - Saves threadID in some data structure
      - (Thread must prevent migration for ID to remain valid)
    - Unblock
      - Resume threadID from data structure
      - Call dispatcher to resume thread
    - Migration
      - Load balancing: migrate to idle dispatcher
      - QoS change: migrate to dispatcher with other resource domain

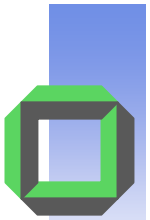
Source: M. Auslander et al. ***Scheduling in K42***. IBM Research, http://www.research.ibm.com/K42, August 2002



Case study: scheduling in K42

- Conclusion:
  - Two-level scheduling approach
    - Dispatchers
    - Kernel-level scheduling classes and domains
    - User-level threads
    - Soft preemptions (acitvations) for user-visible scheduling
    - Hard preemptions for enforcement
  - IPC/PPC and scheduling:
    - Restricted resource donation model
    - Default in-kernel policy that assumes client/server relations
    - Trades off accuracy for performance

Source: M. Auslander et al. ***Scheduling in K42***. IBM Research, http://www.research.ibm.com/K42, August 2002



Thursday

- File / Task Service Design Presentations