Introduction to Pintos

Benedikt Huber
Roland Kammerer

Institut für Technische Informatik
Technische Universität Wien
-
Programmieren von Betriebssystemen UE

13. März 2012

```
# Runs in real mode, which is a 16-bit segment.
        .code16
# Set up segment registers.
        sub %ax, %ax
        mov %ax, %ds
        mov %ax, %ss
        mov $0xf000, %esp
# Configure serial port so we can report progress w/o connected VGA.
        sub %dx, %dx                    # Serial port 0.
        mov $0xe3, %al                  # 9600 bps, N-8-1.
        int $0x14                       # Destroys AX.
        call puts
        .string "PiLo"
# Read the partition table on each system hard disk
        mov $0x80, %dl                  # Hard disk 0.
read_mbr:
        sub %ebx, %ebx                  # Sector 0.
        mov $0x2000, %ax                # Use 0x20000 for buffer.
        mov %ax, %es
        call read_sector
        jc no_such_drive
        # Print hd[a-z].
        call puts
        .string " hd"
        mov %dl, %al
        add $'a' - 0x80, %al
        call putc
```

What is PintOS?

- Previous slide shows a code snippet from the x86 boot loader
- Fortunately, you do not need to write assembler code in this course :)
- But you can read well-documented, existing assembler code, if you like
- Most of Pintos is written in C, however

What is PintOS?

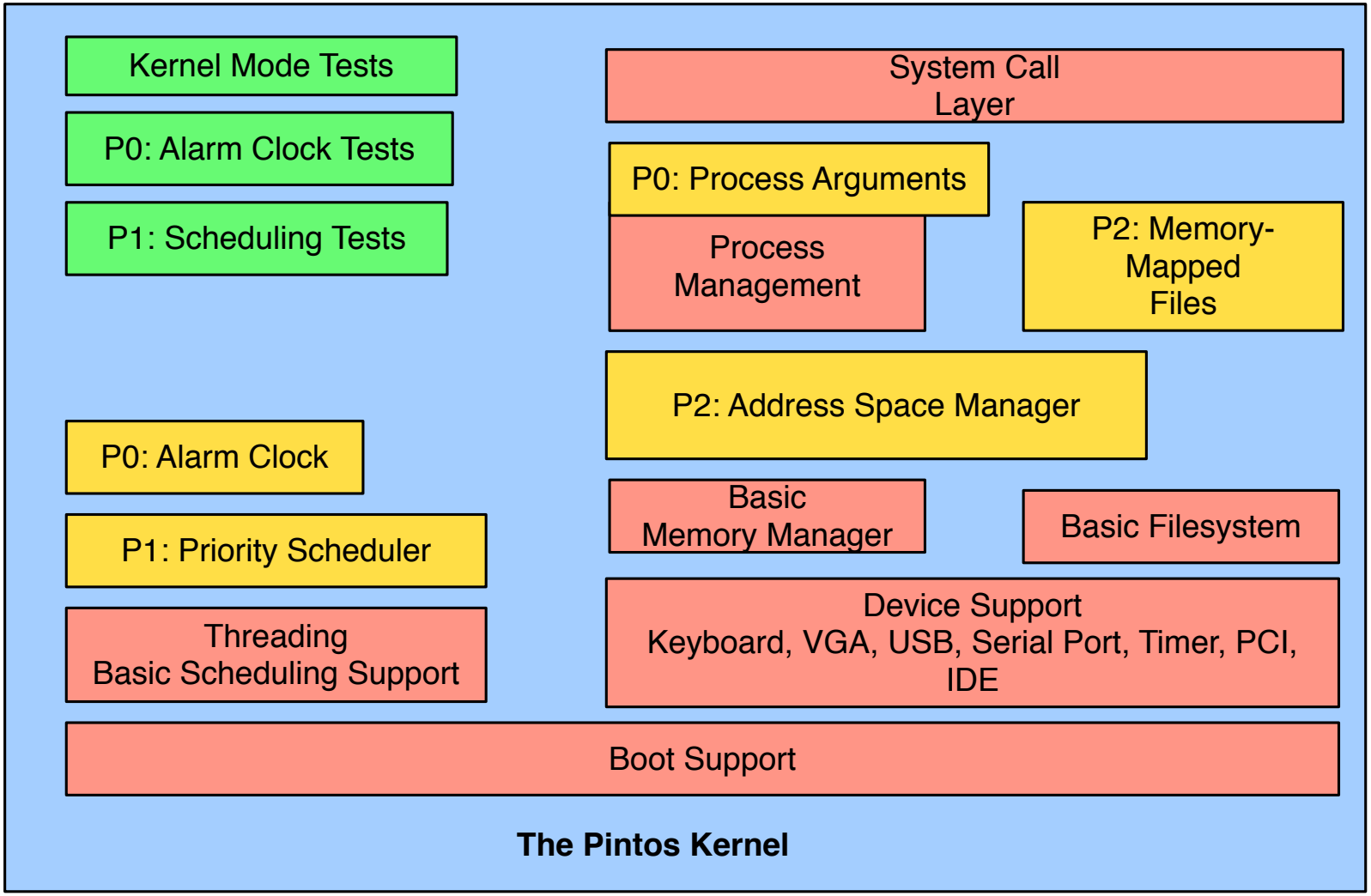
Pintos is an *instructional* operating system,

- running on *x86*,
- written in *C*, by Ben Pfaff,
- emphasizing concepts and preferring simple implementations,
- encouraging *test driven development*,
- that you will improve during this course

Outline

- Some Basics
- Threads and Scheduling
- Synchronization
- User Programs
- Memory Management
- Pintos Infrastrucure
- Assignments

Pintos Structure



Adapted from http://www.pintos-os.org/wp-content/files/SIGCSE2009-Pintos.pdf

Figure: High-Level View on the Pintos Kernel

Setup

- Set `PATH` environment variable to include `pintos-progos/utils`
- The `pintos` command-line tool is used to start the emulator with pintos
- Supports both `bochs` (recommended for project 1) and `qemu` (recommended for project 2)
- Synopsis: `pintos [OPTION...] -- [ARGUMENT...]`
- Arguments before `--` are passed to the pintos script
- Arguments after `--` are passed to the pintos kernel

Running a kernel test

- Kernel tests are linked into the kernel
- You need to pass `--kernel-test` to pintos to run a kernel test
- Kernel tests may not access the file system
- They are used for the alarm clock assignment and project 1

```
cd pintos-progos/intro
make
pintos --bochs --kernel-test -- -q run alarm-single
```

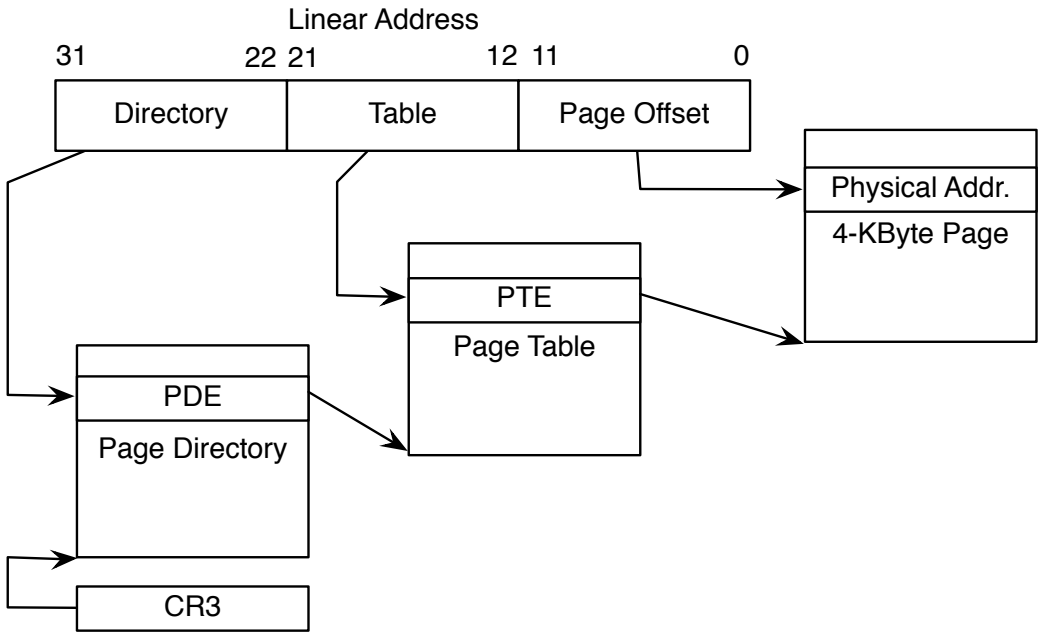
Running a user program

- User programs are loaded from the (virtual) disk and executed by the main kernel thread
- You need to prepare the filesystem to run a user program

```
cd pintos-progos/intro
make
pintos --qemu --filesys-size=2 \
       -p tests/intro/userprog-args/args-none \
       -a args-none -- -q -f run args-none
```

Paging

- PintOS uses paging facilities of x86
- Virtual memory is divided in pages, 4KByte each
- Virtual memory address: page number + page offset
- Processor translates virtual address to physical address, consulting the page directory to find the right page table, and the page table to find the physical address



Adapted from Intel's IA-32 Architectures Software Developer's Manual

Figure: Paging

Threads

- Pintos is multi-threaded; exactly one thread is running at each time (`thread_current()`).
- Each thread is represented by one kernel page
- Thread page includes stack and additional data at the beginning of the page

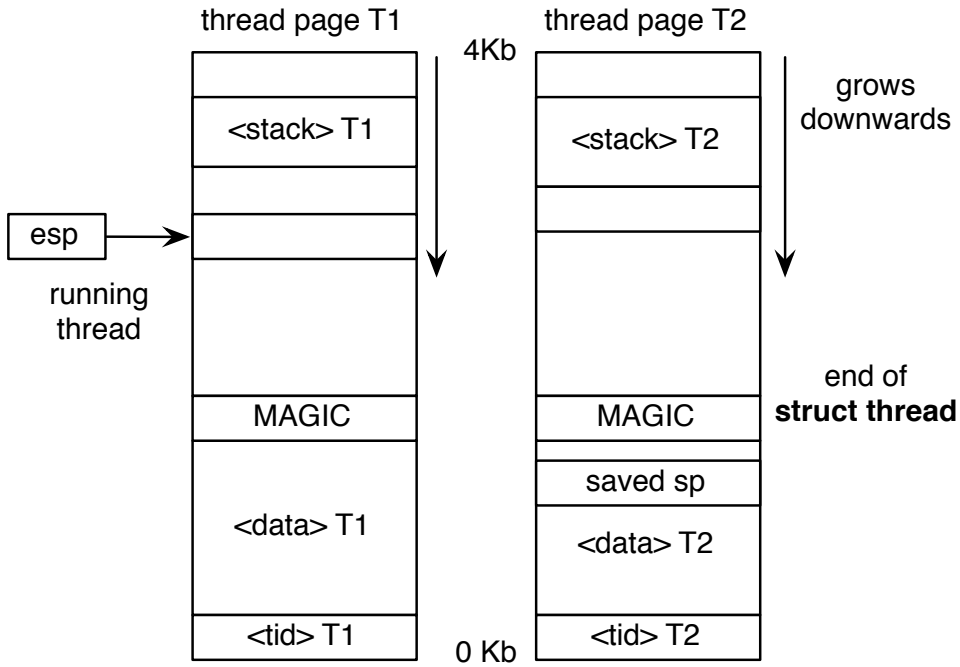


Figure: Threads

Interrupts

- Interrupts notifies CPU of an event
- CPU saves context, then executes interrupt handler routine
- Either internal (caused by the CPU itself) or external
  - Internal: “Belongs” to running thread, interrupts should be enabled, nesting is possible. Examples: Page Fault, System Call.
  - External, such as Timer Interrupt: interrupts disabled, no nesting
- Interrupt frames provide information on the CPU state before the interrupt

Lists

- Doubly linked lists are *ubiquitous* in Pintos (e.g., semaphore wait queue)
- List pointers need to be embedded in all structures which are potential members of a list (sharing possible)
- No need for dynamic memory allocation for list operations

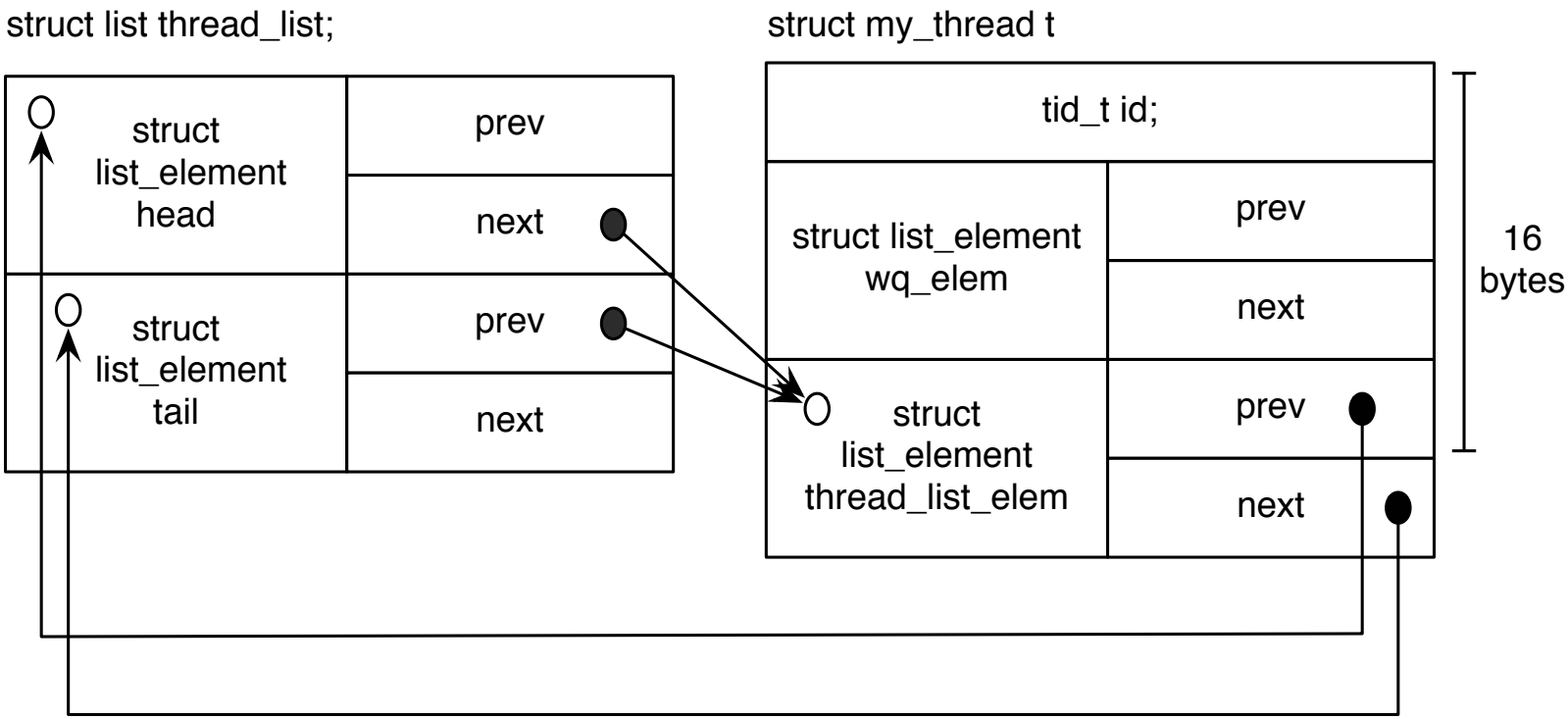


Figure: List Datastructure

Lists

```
struct list_elem
  {
    struct list_elem *prev;      /* Previous element. */
    struct list_elem *next;      /* Next list element. */
  };
struct list
  {
    struct list_elem head;       /* List head. */
    struct list_elem tail;       /* List tail. */
  };

struct list all_threads;

struct thread
  {
    tid_t tid;
    struct list_elem all_threads_elem;
  };
```

List Operations (1)

- We need to use address arithmetic to obtain data of stored elements
- The macro `list_entry` [1] calculates pointer to structure embedding list pointers

```
for (e = list_begin (&all_blocks); e != list_end (&all_blocks);
     e = list_next (e))
 {
   struct block *block = list_entry (e, struct block, list_elem);
   if (!strcmp (name, block->name))
     return block;
 }
```

[1] cf. Linux, macro `container_of`

List Operations (2)

- Operations to initialize list, navigate, insert and delete elements, splice and reverse list
- Use list as a priority queue: insert into ordered list, and remove from front or back
- Be careful when removing elements!

```
for (e = list_begin (&list); e != list_end (&list);
     e = list_remove (e))
  {
    /* WRONG */
    free ( list_entry (e, struct thread, elem) );
  }
```

Threads

- System executes one kernel thread at the time
- Each thread is uniquely identified by
  1. its identifier `tid`
  2. the (virtual) address of its kernel page
  3. its stack pointer (directly accessible in running thread)
- Some threads execute user processes
- *Scheduling* algorithm decides which thread to run
- **Source:** `threads/thread.c`

Thread Page (1)

```
struct thread {
 tid_t tid;                          /* Thread identifier. */
 enum thread_status status;          /* Thread state. */
 char name[16];                      /* Name (debugging). */
 uint8_t *stack;                     /* Saved stack pointer. */
 int priority;                       /* Priority. */
 struct list_elem allelem;           /* List element (all) */

 /* Shared between thread.c (ready list) and
                        synch.c. (waiting queue) */
 struct list_elem elem;
 ...
 unsigned magic;                     /* Detects stack overflow */
 /* Stack ends here */
};
```

Thread Page (2)


```
struct thread {
 tid_t tid;                          /* Thread identifier. */
 ...
 /* Process Structure; NULL if no process started */
 struct process* process;

 /* User processes created by this thread */
 struct list children;

 /* Page directory for user processes */
 uint32_t *pagedir;

 unsigned magic;                     /* Detects stack overflow */
 /* Stack ends here */
};
```

Scheduling

- Threads status is one out of
  - `THREAD_RUNNING`: the currently active thread
  - `THREAD_READY`: ready to be scheduled (ready list)
  - `THREAD_BLOCKED`: thread is waiting for an event
  - `THREAD_DYING`: destroyed on next context switch
- Scheduling is triggered by either
  - Timer interrupt (`thread_tick()`)
  - The running thread, for example, when thread is blocked, its priority was changed, etc.

Switching Threads

- Switching to another thread is accomplished by changing the stack pointer in `switch_threads`
  1. Scheduling algorithm selects the next thread to run
  2. The running thread calls `switch_threads`
  3. The necessary registers and the stack pointer of the running thread are saved
  4. The registers and the stack pointer of the next thread are restored
  5. The function returns, the next thread is now running
- This works because all threads which are not running *have been preempted* in `switch_threads`
- New threads need to carefully setup stack frame to fake this

Synchronization

- Synchronization is *really* important
- Synchronization Mechanisms
  - Disabling interrupts
  - Semaphores
  - Locks and condition variables
- Checklist
  - Is data thread-local or shared?
  - Which thread is responsible for allocating and destroying?
  - Does owner of data change?
  - Which pointers do potentially reference data?
  - Which lock protects shared data?
- Source: `threads/synch.c`

Disabling Interrupts

- Critical sections can be protected by disabling interrupts (+spinlocks on multiprocessor)
- Avoid this form of synchronization (less responsive scheduling)
- Necessary if blocking is impossible in the current context (external interrupt handler)

Locks

- Clean abstraction to protect shared data structures
- At any time, at most one thread holds a lock
- Only the lock owner may release the lock
- When *acquiring* lock: if lock is held by another thread, acquiring thread is blocked
- When *releasing*: oldest blocked thread is unblocked (FIFO)

```
struct lock list_lock;
lock_init(&list_lock);
...
lock_acquire(&list_lock);
modify_list(&the_list);
lock_release(&list_lock);
```

Semaphores

- Shared integer variable
- `sema_down`: block until positive, then decrement
- `sema_up`: increment, potentially unblocking other threads
- Semaphore usage example:
  - Initialize semaphore to 0
  - One thread waits (down on semaphore) until an action is completed
  - Another thread signals (up) the completion of the action
- Locks are restricted semaphores (initialized to 1), but simpler to understand

Monitors

- Another synchronization mechanism implemented in Pintos
- Problem: Only want to acquire lock if some condition is true (e.g., buffer not empty)
- Monitors allow to give up lock until signalled by another thread

```
lock_acquire (&lock);
while (n == 0) /* empty buffer */
 cond_wait (&not_empty, &lock);
n--;
cond_signal (&not_full, &lock); /* buf not full*/
lock_release (&lock);
```

User Programs

- Executed by a kernel thread by setting up the initial environment (memory, stack pointer, etc.) and then switching to user mode
- User programs access kernel functionality by means of system calls (internal interrupts)
- User processes may create other processes, access the file system, shutdown the computer,...
- ...but never crash or corrupt the kernel
- **Source:** `userprog/process.c`

Spawning new processes

- The parent process creates a new thread and passes it all necessary information to load a program (thread start argument)
- Next, the parent process waits for the loading process to complete (`sema_down`)
- The new thread loads program segments (code, data) from disk into memory
- The new thread sets up the stack
- The new threads signals the parent process that loading is complete, and starts to execute the user program

Destroying processes

- When a process terminates, it checks whether the parent process is alive (race condition requires lock here)
  - If the parent process is alive, the process signals (`sema_up`) that it terminated
  - Otherwise, it cleans up the resources it used itself
- Parent process cleans up terminated child processes
- Parent process may wait until a child process signals its termination (`sema_down`)
- Be careful not to leak memory: Some thread/process needs to be responsible for deallocating memory

Protection from misbehaving user programs

- Memory Protection
  - Paging is used to ensure only valid virtual memory addresses are accessed
  - An invalid access causes a page fault (internal interrupt)
- Accessing user-space memory in kernel mode
  - You must not trust addresses from user space
  - The current implementation uses the page fault handler and specialized functions to access user memory

System Calls

- User programs communicate with the kernel by triggering the syscall interrupt (0x30)
- The system call handler analyzes the stack, and carries out the appropriate action
- One must not trust addresses and especially strings from user space

```
#define syscall2(NUMBER, ARG0, ARG1)
        ({
          int retval;
          asm volatile
            ("pushl %[arg1]; pushl %[arg0]; "
             "pushl %[number]; int $0x30; addl $12, %%esp
               : "=a" (retval)
               : [number] "i" (NUMBER),
                 [arg0] "r" (ARG0),
                 [arg1] "r" (ARG1)
               : "memory");
          retval;
        })
```

Memory

- Pintos provides up to 64Mb of physical memory
- Memory is always accessed using virtual addresses
- Command line switch `-m` to select available memory
- Divided into kernel (above 3Gb) and user memory
- Pages are managed in kernel and user pool (`palloc.c`)

Memory Allocation

- In kernel mode: either page allocator or block allocator
- Page allocator `palloc`
  - Allocates one or more consecutive pages
  - Possible to allocate user space memory (`PAL_USER`)
  - Internal and external fragmentation
- Block allocator `malloc`
  - Memory efficient for blocks less than 4Kb
  - External fragmentation for larger blocks
- Avoid memory leaks!

Paging

- Most significant bits determine page associated with memory address
- Functionality for paging is already available (see `threads/pte.h` and `userprog/pagetable.c`)
- Each user process has it own page directory (pointing to zero or more page tables)
- Page Flags
  - Present: Whether page is present in physical memory; access page faults if it is not
  - Read/Write: Whether page is writable; access page faults on write if it is read only
  - User/Kernel: Whether page is accessible in user mode
  - Accessed, Dirty: Useful to implement swapping and memory mapped pages
- Page fault handler can setup missing pages (Project 2)

Simple File System

- Setup using the pintos command line utility
- Accessed when loading user programs, during system calls, loading pages for memory mapped files,...
- In this course: only simple synchronization (global lock)
- For user programs, open files are identified by a file descriptor
- In kernel space, handles of open files are of type `struct file*`
- A file may be opened more than once; the internal data is deleted when (1) the file is deleted and (2) no process has a handle to that file

```
void process_lock_filesys (void);
void process_unlock_filesys (void);
```

Filesystem Restrictions

The pintos filesystem is simple, and restricted:

- No internal synchronization (just global lock)
- File size is fixed at creation time
- Limited number of files
- No directories
- Data in a single file must occupy a contiguous range of disk sectors
- No subdirectories
- File names are limited to 14 characters
- No filesystem repair tool ;)

File System Tools

```
* Create a 2 MB hard disk for pintos
   # [src/userprog/build]
   pintos-mkdisk filesys.dsk --filesys-size=2
* Format Disk
   # -f ... format virtual disk
   pintos -f -q
* Copy file to filesystem
   # -p FILE ... file to put on virtual disk
   # -a FILE ... newname on virtual disk
   pintos -p ../../examples/echo -a echo -- -q
* Execute echo, and get file 'echo' from disk
   pintos -g echo -- -q run 'echo x'
```

Build System

- Modular build system using Makefiles
- Files of interest
  - `intro/Make.vars`: defines the build and testing process for project 0 (intro)
  - `Make.config`: Configures the build tools
  - `Makefile.build`: Includes the list of all files necessary to build the kernel; you need to modify this file if you add new files
  - `Makefile.kernel`: Included by the Makefile in subdirectories
  - `Makefile.userprog`: Defines how to build user programs
- Binaries
  - Build process creates `loader.bin` and `kernel.bin`
  - loader.bin: 512 byte boot loader at 0*x*7*c*00
  - kernel.bin: ELF, limited to 512K, entry at start, linking controlled by threads/kernel.lds.S (linker script)

Intro Project

- Two small tasks, a lot of reading
- Alarm Clock
  - Current implementation of `sleep` uses busy wait
  - Instead, block the thread until time has expired
  - *Recommended:* write a least one kernel test
- Argument passing and stack setup
  - Currently, stack setup is not implemented
  - Moreover, user programs do not accept arguments
  - Implement both argument parsing and stack setup
  - The stack page is user space memory; access it correctly
  - Dump the stack (`hex_dump`) to debug your implementation
  - *Recommended:* write and run at least one user program (see `examples`)

Requirements and Design Document

- For each of the projects, there is a document [2] including
  - An introduction to the topic
  - Overview of affected files
  - Requirement list
  - A link to the design document
  - A FAQ to help you out
- To get a perfect score, you need to meet all requirements, and pass all tests for the assignment

[2] http://pan.vmars.tuwien.ac.at/progos/doc/progos/pintos.html

Design Document Example

- Alarm Clock
  - Data Structures
    - Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.
  - Algorithms
    - Briefly describe what happens in a call to timer_sleep(), including the effects of the timer interrupt handler.
    - What steps are taken to minimize the amount of time spent in the timer interrupt handler?
  - Synchronization
    - How are race conditions avoided when multiple threads call timer_sleep() simultaneously?
  - Rationale
  - Survey Questions

Automated Tests

- There is a set of automated tests for each project
- For each project, there is one directory to
  - Build the kernel (`make`)
  - Run automated tests (`make check`)
  - Run grading script (`make grade`)

```
[~]        cd pintos-progos/intro
[intro]    make
[intro]    make check
[intro]    make grade
           # Project 1
[threads] make check
           # Project 2
[vm]       make check
```

Project 1: Priority Scheduling

- Goal: Implement priority scheduling
- Scheduler should select thread with highest priority
- Prevent priority inversion by implementing priority donation for locks
- Priority donation is tricky!
  - Design and verify a correct strategy before starting to implement
  - Check your design against the test cases provided for priority scheduling

Priority Donation Test

```
ASSERT (thread_get_priority () == PRI_DEFAULT);
lock_acquire (&a);
lock_acquire (&b);

/* thread a needs lock a, boosts main thread */
thread_create ("a", PRI_DEFAULT + 3, a_thread_func, &a);
msg ("Main thread should have priority %d.  Actual priority: %d.",
     PRI_DEFAULT + 3, thread_get_priority ());

/* thread c does not need a lock, runs until completion */
thread_create ("c", PRI_DEFAULT + 1, c_thread_func, NULL);

/* thread b needs lock b, boosts main thread */
thread_create ("b", PRI_DEFAULT + 5, b_thread_func, &b);
msg ("Main thread should have priority %d.  Actual priority: %d.",
     PRI_DEFAULT + 5, thread_get_priority ());

/* after release, main threads priority is still highest */
lock_release (&a);
/* after release, main thread priority is default */
lock_release (&b);

msg ("Threads b, a, c should have just finished, in that order.");
```

Project 2: Virtual Memory Management

- Goal: improve the virtual memory management of pintos
- You need to manage additional information on pages (where to get the initial data, whether you need to write back to disk)
- Pages should be loaded into memory on demand (page fault handler)
- Stack should grow if necessary (heuristic to detect stack accesses)
- Support for mmaping files into memory (write-back on munmap)

Registering as a group

- Submit your group member suggestions until Friday, 16.3
- Groups will be assembled until next Monday
- Each group needs to register itself for one tutor
  - In myTI, register your group for “Gruppenanmeldung Tutor”
  - *Ignore* the actual timeslots for this myTI date, but pick a tutor which has compatible time constraints
- Each group is assigned to one particular tutor
- The tutor will contact you via email as soon as the assignment is complete

Working as a group

- Get to know your team members (email)
- Setup a shared git repository
  - As soon as group accounts are available
  - We will provide a script to setup the shared repository
  - One team member initializes the repository and shares the git clone URL
  - Then all team members clone their local copies from this URL
- Attend the design talks with your Tutor as a group
- Have fun!