Real-time Operating Systems and Systems Programming

Summary

ARVUTITEADUSE INSTITUUT
DEPARTMENT OF COMPUTER SCIENCE

Grades

- Programming project(s) (40%)

  Huffman
  Labs

- Exam (60%)
  Terminology, some functions, code reading, coding on paper

Exam

- 2 programming tasks – think of practice lessons
- "what does this function do?" – 2-3 sentences max
- operator precedence puzzle
- memory layout puzzle (pointer/array things)
- some general questions

C keywords

- Types
  - `char double enum float int long short struct union void`
- Parameters to variables
  - `auto const extern register signed static unsigned volatile`
- Flow control
  - `break case continue default do else for goto if return switch while`
- Operators
  - `sizeof, typeof`

Operator precedence

```
>>    ()   []  ->  .
<<    !  ~  ++  -- + - * & (type) sizeof
>>    *  /  %
>>    +  -
>>    <<  >>
>>    <  <=  >  >=
>>    ==  !=
>>    &
>>    ^
>>    |
>>    &&
>>    ||
<<    ? :
<<    =  +=  -=  *=  /=  %=  &=  ^=  |=  <<=  >>=
>>    ,
```

Variables

- Name to an address.
- Type says amount of memory to reserve
- Must be declared before use

Real-Time Systems

- Hardware or software which has a time constraint for reactions
- For our purposes, also embedded systems
  - What would be the difference?

Characteristics

- Specified limit on system response latency
- Event-driven scheduling
- Low-level programming
- Software coupled to special hardware

- Volatile Data
- Multi-tasking implementation
- Unpredictable environment
- Runs continously
- Life-critical applications

Terminology

- System: black box with n inputs and m outputs
- Response time: time between presentation of a set of inputs and the appearance of the corresponding outputs
- Events: Changes of state which cause changes in flow-of-control of a progam
    - Synchronous: events occur at predictable times
    - Asynchronous: events interrupt flow-of-control

State vs Event based

- State based:
  - System constantly reads system inputs and reacts to their combination
- Event based
  - System is in standby and events “wake” it to make it work

Deterministic RTS

- A deterministic RTS: you can determine a unique set of outputs and next state from a given set of possible states and inputs.

Real time Correctness

- Correctness depends on result and the time of delivery.
- Soft – missing some deadlines not a problem
- Firm – missing deadline: result worthless, but not a problem
- Hard – missing a deadline makes result worthless and is a problem

Static Predictability

- RT system: satisfying time constraints
  - Assumptions about workload and sufficient resources
  - Certified at design time, that all constraints will be met
- For static systems, 100% guarantees can be given at design time
  - Requires immutable workload and system resources
  - System must be re-certified on any change

Dynamic Predictability

- Dynamic systems: not statically defined
  - Systems configurations might change
  - Workload might change
- Dynamic predictability
  - Under appropriate assumptions (sufficient resources)
  - Tasks will satisfy time contstraints

Latency minimization

- Latency is the time between an event and the system's reaction to it.
- We want to minimize latencies
  - For different applications, different latencies are required.
  - 10 ms might be barely enough (probably a dedicated system)
  - 500 ms might be enough (might use an external kernel)

Multiple Requirements

- Real-time
- Power constraints
- Size constraints
- Cost limits
- Security requirements
- Fault tolerance

- *Often conflicting*

Operating systems

- Interface between hardware and software
- Provide services for applications
- Provide an abstraction layer for hardware

What services?

- Processes
- Multitasking
- Interrupts
- Memory management
  - Virtual memory
- Protected/supervisor mode
- Disk & Files

- Booting the computer
- Device drivers
- Networking
- Users / authentication
- Graphical UI

What applies for Real-time?

Usually not included in RTOS

- Paged & swappable virtual memory management
- Disk filing system
- Full networking facilities
- Intertask security
- Multi-user support
- GUI

More power (and responsibility)

- Interrupts can be masked
  - Can used only if max. int. latency (by specification) longer than longest critical section path
- Memory allocation
  - Fixed-size blocks
  - Re-entrant core libraries (allocation on stack)

Other services

- HW initialization
- Real-time clock management
- Critical resource protection
- Intertask communication
- Intertask synchronization

- I/O management
- Multiple interrupt servicing
- Memory allocation and recovery
- Assistance for debugging

POSIX

- POSIX (**P**ortable **O**perating **S**ystem **I**nterface [for Uni**X]**)
- Standard for Unix, defines core specifications-command-line, shell, some programs, basic IO.Threading API.

Posix IO

- Program has two inputs:
  - Command line arguments to main()
  - Standard input (keyboard connected to file by default)
- Two outputs
  - Standard output (connected to terminal by default)
  - Standard error (connecterd to terminal by default)
- Standard streams can be redirected

I/O for RT Systems

- Can be complex
  - Desktop computing hides the fact successfully
- Need to understand
  - Port address mappings
  - Register functionalities

Hardware access

- Done by accessing HW ports & registers
  - Memory mapped
  - I/O mapped

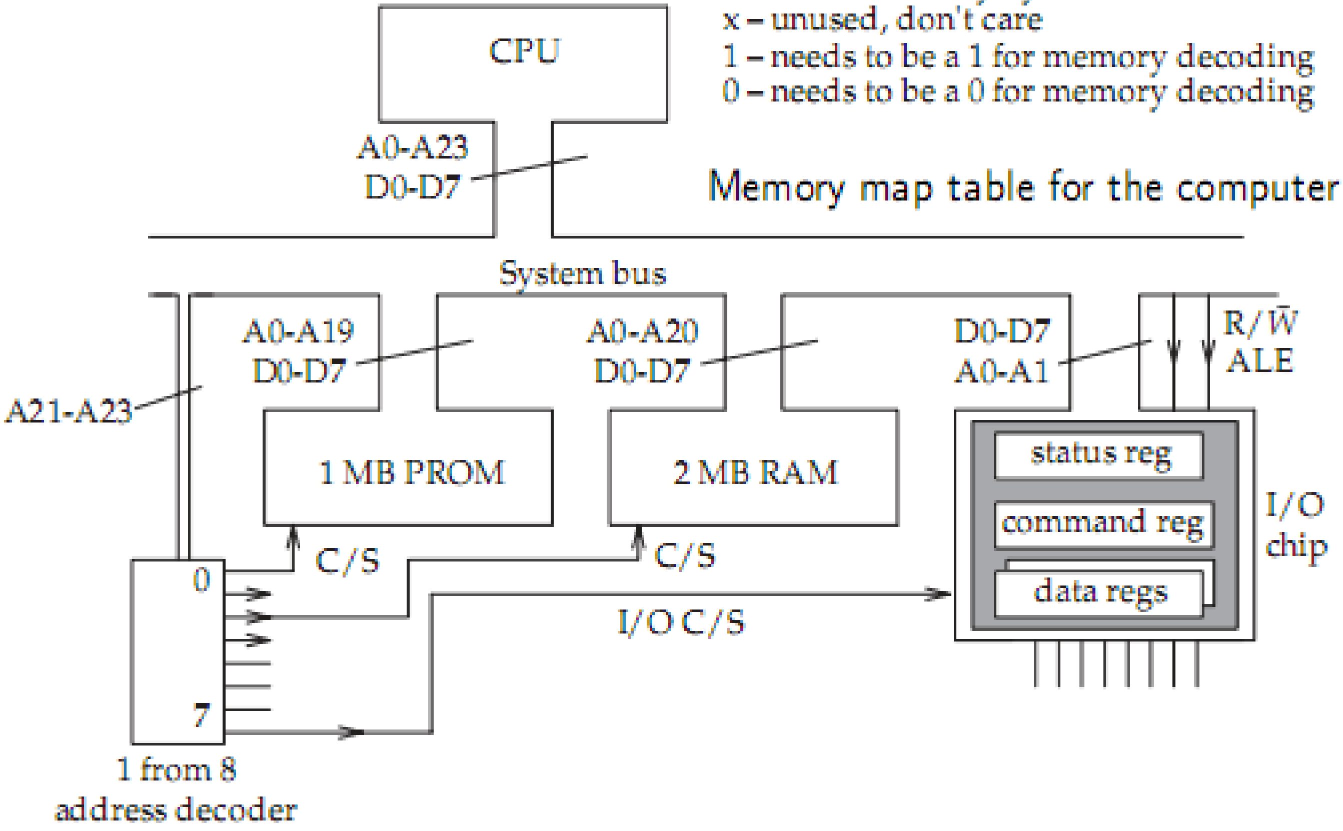
Memory mapped

- I/O registers behave like memory locations

Device	Size	Address pins	24 bit address bus	Address range
PROM1	1 MB	20	000x ++++ ++++ ++++ ++++ ++++	00 0000–0F FFFF
RAM1	2 MB	21	001+ ++++ ++++ ++++ ++++ ++++	20 0000–3F FFFF
RAM2	2 MB	21	010+ ++++ ++++ ++++ ++++ ++++	40 0000–5F FFFF
RAM3	2 MB	21	011+ ++++ ++++ ++++ ++++ ++++	60 0000–7F FFFF
I/O	4 B	2	111x xxxx xxxx xxxx xxxx xx++	E0 0000–E0 0003
				E0 0004–E0 0007
			Aliases	E0 0008–E0 000B
				E0 000C–E0 000F
				. . .

+ – used internally by device
x – unused, don't care
1 – needs to be a 1 for memory decoding
0 – needs to be a 0 for memory decoding

Memory map table for the computer system



Memory-mapped I/O, showing the address decoding for a 24 bit CPU

IO Mapped

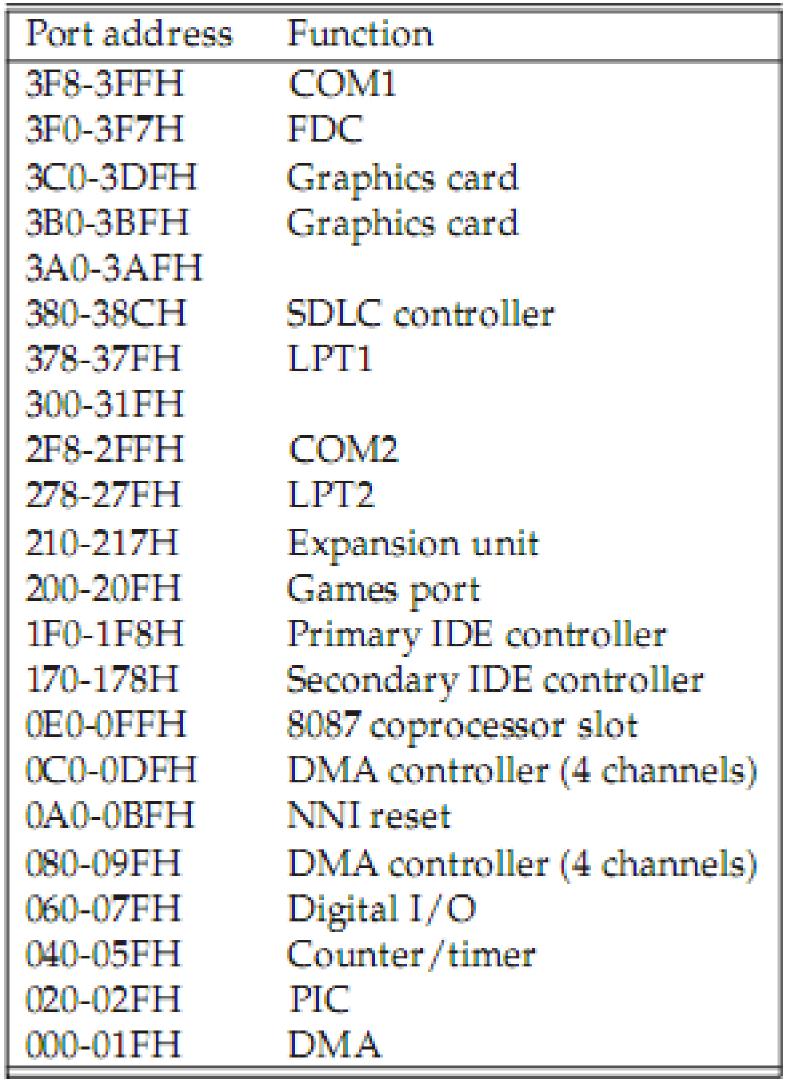
- More bus control lines, extra instructions
- Independent address space for I/O ports
- Intel (IN & OUT instructions)
- Better caching: we need to read “raw data” for I/O
- C lang extensions: inb() outb() functions.

Programmers view of ports

- For direct I/O:
  - Base address of I/O chip
  - Memory map and function of its registers

Need to Identify:
Command
Status
Data

PC I/O mapped port addresses



Port address	Function
3F8-3FFH	COM1
3F0-3F7H	FDC
3C0-3DFH	Graphics card
3B0-3BFH	Graphics card
3A0-3AFH	
380-38CH	SDLC controller
378-37FH	LPT1
300-31FH	
2F8-2FFH	COM2
278-27FH	LPT2
210-217H	Expansion unit
200-20FH	Games port
1F0-1F8H	Primary IDE controller
170-178H	Secondary IDE controller
0E0-0FFH	8087 coprocessor slot
0C0-0DFH	DMA controller (4 channels)
0A0-0BFH	NNI reset
080-09FH	DMA controller (4 channels)
060-07FH	Digital I/O
040-05FH	Counter/timer
020-02FH	PIC
000-01FH	DMA

A listing of PC I/O-mapped port addresses with standard function

Port polling

- Poll until data arrives
- Problem: CPU fast, devices slow
- Dedicated (spin) vs intermittent (timed) polling

Blocking & nonblocking

- IO operations wait until complete: blocking
- Simple: read only when data waiting (kbhit() - DOS/Win)
- Possible to turn off blocking & buffering for keyboard
- fd = open(“/dev/ttyS0”, O_RDWR | O_NOCTTY | O_NONBLOCK);
- Ioctl() & fcntl()

Blocking

- Device blocking often necessary for fair scheduling
- Threading possible
- select() function for multiple sockets.

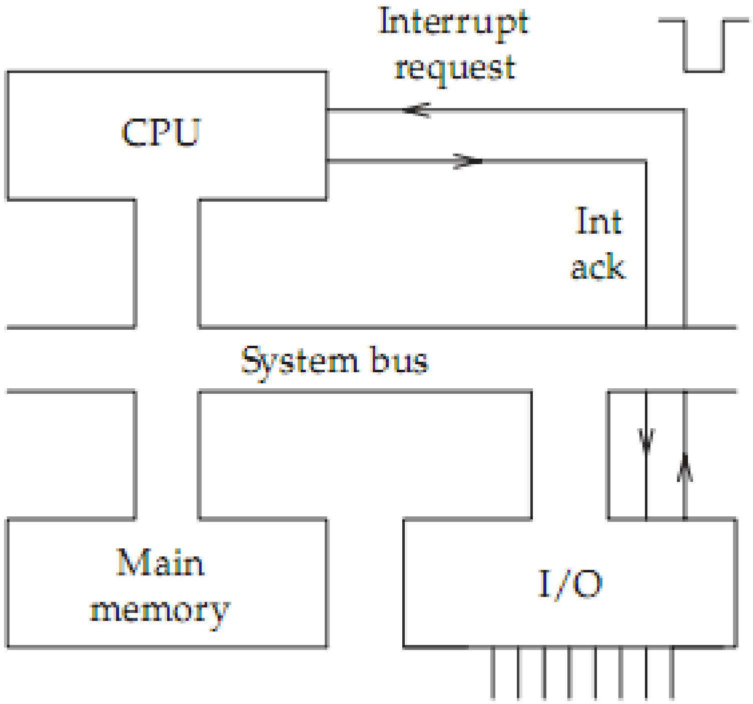
Interrupts

- Interrupt method good for occasional attention
- Requires hardware support, quite common

CPU level

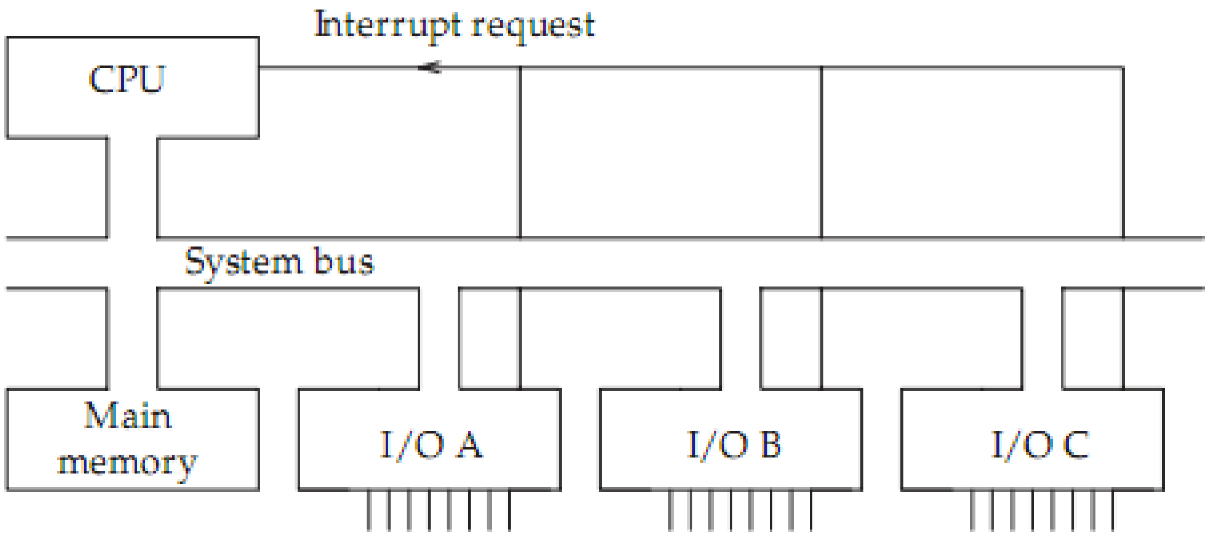
- On every instruction, interrupt line is checked
- On interrupt, selected service routine executed after saving the instruction pointer
- Gets restored afterwards.
- Response in 10μs

System diagram



Source detection

- Often only one interrupt line
- How to find the source?



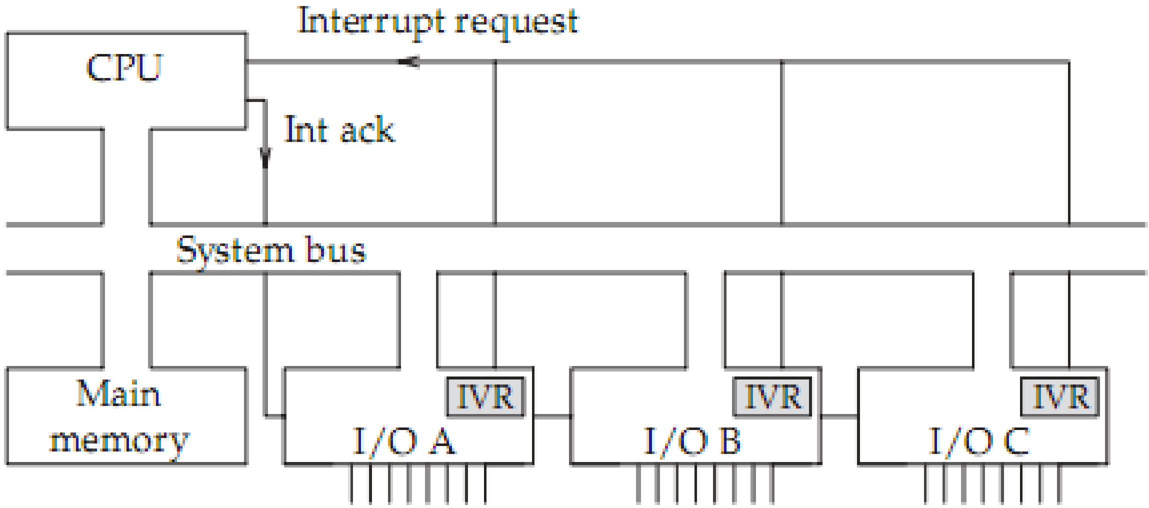
Connection of many interrupting devices to the CPU

Polling

- Slow since all devices must be polled individually
- Does not require extra hardware
- Adequate for small number of devices

Vector interrupts

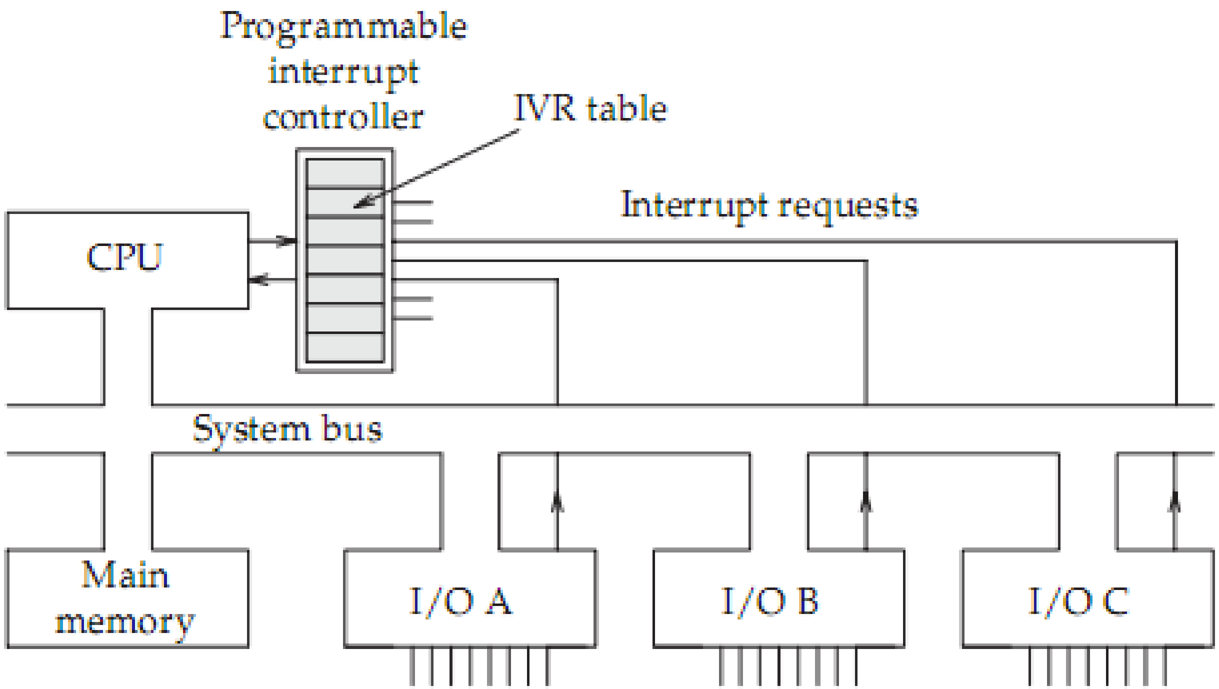
- Interrupt Vector Registers (IVR) in devices
- Motorola Mc68000 family



Vectored interrupts using a daisy-chain prioritization line

PIC interrupts

- Needs Programmable Interrupt Controller (PIC)
- PC method. Centralized prioritizing encoder.



Vectored interrupts using PIC hardware

Actions

- Interrupt
- CPU saves program counter (PC) & CPU status register to stack
- Entry address for Inter. Service Routine (ISR) from Interrupt Vector Table (IVT), written to PC
- ISR starts

ISR

- Store register contents to stack
- Verify source (test device flag for example)
- Remove cause to prevent further interruption
- Reinitialize device?
- …
- POP saved registers from stack, RTE instruction to restore Instruction Pointer & status

Memory

- Processor registers (hidden in C)
- RAM
- Devices (hard disk)
- Internet?
- People??
- Books???

Stack

- Simple data structure
- Efficient implementations
- FILO (as opposed to FIFO)
- Operations: Push, Pop
- Important to us due to call stack
- Often supported in hardware

C implementation

```
typedef struct {
    int size;
    int items[STACKSIZE];
} STACK;
```

```
void push(STACK *ps, int x)
{
    if (ps->size == STACKSIZE) {
        fputs("Error: stack overflow\n", stderr);
        abort();
    } else
        ps->items[ps->size++] = x;
}
```

```
int pop(STACK *ps)
{
    if (ps->size == 0){
        fputs("Error: stack underflow\n", stderr);
        abort();
    } else
        return ps->items[--ps->size];
}
```

Hardware implementation

- Special stack register (can be read/written)
  - We will name it %esp in further examples
- Assembly instructions to manipulate it

Short introduction to assembly

- Mainly moves data around (mov series)
- Jumps, conditional jumps (jmp series)
- Arithmetics
- Management (push, pop, call, return)
- Examples from IA32
  - Word = 16 bit due to ancient history
  - Double word for 32 bits

Registers

- 8 registers for 32bit values
- General purpose: %eax %ecx %edx %ebx %edi %esi (Historical names, would be simpler)
- Fun registers: %esp %ebp (Stack pointer & Frame pointer)
- Can be addressed also in smaller segments
- %eax[%ax[%ah[ ] %al[ ]]

Aside: C numeric constants

- Decimal: 10; -10
- Octal: 037 0431 (leading 0)
- Hexadecimal: 0xf1 0xdada (leading 0x)
- Unsigned: 10u, 0xafU
- Long: 10l 10L; Short: 10s 10S
- Floating-point: 0.04 4e-2 10.0 1e2

Stack operations

%eax = 0x123 %edx = 0 %esp = 0x108

pushl %eax

%eax = 0x123 %edx = 0 %esp = 0x104

popl %edx

%eax = 0x123 %edx = 123 %esp = 0x108

Stack frame

- Uses frame pointer to keep track of previous frame
- Stack pointer tracks “top” of stack

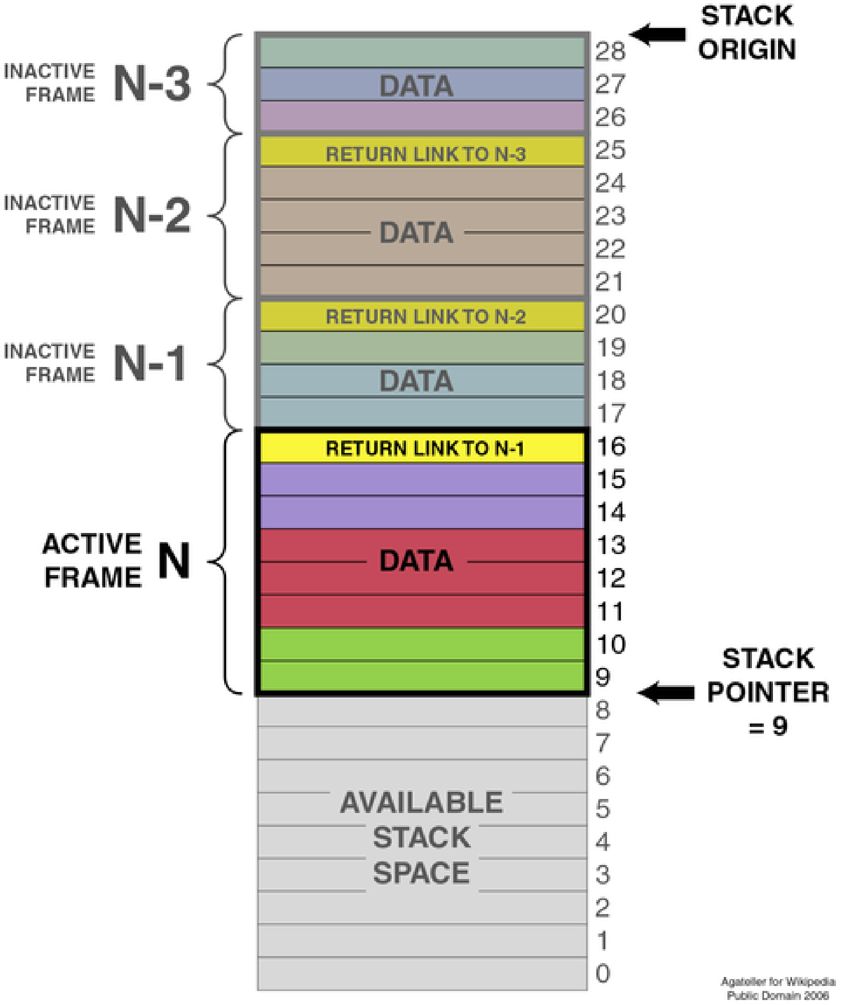


Image source: Wikipedia

One frame contains

- Address of last %ebp
  - Current frame pointer points to it (data accessed in relation to it)
- Saved registers
- Local variables (out of registers; array; &)
- Any temporary data
- Argument building area
- Return address (only if not active frame)

Transfer of control

- For procedure calls, processor supports the following instructions:
  - **call *Label*** / **call **Operand*** – calls procedure
  - **leave** – prepares stack for return
  - **ret** – return from call

1. Prepare stack
2. call procedure
3. Profit

Call instruction

- Can start executing from an address or a label
- Pushes return address to stack (return address is next instruction from the call)
- Jumps to called address (= set program counter to the start of a procedure)

Ret instruction

- Pop an address from stack
- Go to the address with program counter
- *To use properly, stack pointer must point to the “bookmark” address that* call instruction *stored.*
- For preparation, leave instruction is used

```
Leave:
movl %ebp, %esp
popl %ebp
     # note: %ebp == stack frame
```

Recap

- *call* pushes return address to stack, jumps

- new procedure saves old stack frame to stack
- Copies current stack pointer to frame pointer
- …
- Copy frame pointer to stack pointer
- Restore old frame pointer
- Return to stored bookmark

Register conventions

- %eax, %edx, %ecx – Caller save
  - Procedures can overwrite them as want, but must restore them after return, as they may get overwritten
- %ebx, %esi, %edi – Callee save
  - Procedures can overwrite them only if they save them and restore them before returning
- %eax is the return register

What reflects to C?

- Automatic variables live on stack
- Function arguments are copied to stack before calling (call by value)
- Using pointers as arguments to functions can make calls by reference
- Uninitialized variables contain garbage
- Pointers to freed stack contain garbage
- Writing over a stack frame pointer is Not Good
- Writing over the return address is worse

Buffer overflow exploitation

- When a buffer overflows, it is possible to write over the return pointer to point within the buffer itself
- The buffer gets executed
- Newer C implementations protect stack for desktop compilation

How to remove variable from stack?

- Easy, declare it as *static*.
  - static int i = 0xf00;
- Moves the variable to heap

Long Jump

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

- setjmp ja longjmp can aid interrupt handling
- setjmp() saves the *stack* into env buffer, for use by longjmp() function. The env is usable only once.
- setjmp() returns 0 on the first call and a different value on the second call after longjmp() has been called. It can return "twice"!

Long Jump (2)

```
void longjmp(jmp_buf env, int val);
```

- longjmp() restores the saved environment. After longjmp() the program behaves like setjmp() would have returned the value val. longjmp() cannot send 0 since it will be replaced with 1.

```
jmp_buf env;
if ((val=setjmp(env)) == 0)
   printf("Now we have set long jump\n");
else
   printf("Long jump has returned value
%d\n",val);
.........…
longjump(env, 3);
```

Heap

- Section of memory for dynamic structures
- Bounded by brk pointer in kernel
- Function for allocation and deallocation:
  void *sbrk()
- Normally not used directly
  alloc(), malloc(), calloc(), free()
- Allocators divide heap into blocks

Why dynamic allocation?

- Programs often know the amount of memory needed and sizes for data structures runtime
- RTOS note: you might still prefer static allocation for predictability

Constraints for allocators

- Handling arbitrary request sequences
- Making immediate responses for requests
- Use only heap
- Block alignment must be kept
- Cannot modify allocated blocks

Fragmentation problem

- Allocation and deallocation sequences can result in “holes”.
  - Internal fragmentation: the holes within memory blocks themselves
  - External fragmentation: happens when there would be enough free memory for a block, but a single block cannot hold it.

Implementation

- Most naïve: just allocate, never reuse
- More clever:
  - Organize free blocks
  - Deal with placement of blocks
  - Splitting of blocks
  - Joining of blocks

Organizing blocks

- Implicit free list
- Blocks have headers which include
  - Block size
  - Allocated/Free field
- Header size: 1 word
- Return the pointer to content, use header internally

Header

- Due to alignment, the block sizes are multiple of 8
    - 3 lowest order bits are free!
    - Last bit used for free/allocated
- Terminating header with size 0
- “Contents” are located on double word alignment boundaries
- We have minimum block size

Where to place?

- When searching for a free block, one can have policies for placement:
  - First fit – end of list is often free; fragments
  - Next fit – spreads allocation; fragments worse
  - Best fit – good, but slower

Should we split?

- Option to use entire block
- Or split
- If the fit is “good”, do not split

How to get free memory?

- Ask for more (mmap() or sbrk())
- Merge adjacent blocks upon freeing
  - Can also be done when needed

Merging

- Merging next block is simple: just add
- How to find the previous block?
  - Boundary tags (block footer)
  - Block header has 2 free bits, use one to show that the previous block is free (then only free blocks have footers)

Implementation details

- Initialize block list
- Decide policies
- Blocks may behave like data structures (linked or double linked lists)
- For faster allocation, keep free lists
- Segregation of free lists (see next)

Simple Segregation

- For memory storage, a memory class will store blocks up to size X (malloc({17-32}) → 32)
- If new memory is needed, allocate a page
- Split it into equal blocks sized according to the storage class
- Do not merge blocks
- Link them into free list
- Problems: extreme fragmentation
  (sounds like a grenade)

Segregated fit

- Allocator has an array of free lists, according to size classes
- Allocate according to class, first fit
- Split if needed
- If not found, search larger classes or ask more
- Thought to work well since GNU malloc() behaves like this

Array memory management

- Dynamically defined 2d array needs 2 allocations with malloc() and some tricky pointer initialization

Fixed 2d array

- ***Stack* allocation**
  Allocation: int fixed[50][100];
- **Access:** fixed[5][9] = 1; /* või */
  fixed[0][5*100+9] = 1; /* või */
  fixed[1][4*100+9] = 1; /* jne */
- **Initialization:**

```
for(i=0;i<50;i++) for(j=0;j<100;j++) fixed[i][j] = 0; /*
   slooow */

int *ptr = fixed[0]; int *end = fixed[49]+99; *end = 0;
   while(ptr != end) *ptr++=0;
```

- **Passing to a function:**
  Prototype: void func(int fixed[50][100]);

Dynamic 2d array

- Stored in *heap.*

Allocation

```
int **dynamic;
dynamic = (int**)malloc(sizeof(int*)*50);
dynamic[0] = (int*)malloc(sizeof(int)*50*100);
for (i=1;i<50;i++) dynamic[i]=dynamic[i-1]+100;
```

Access

```
dynamic[5][9] = 1; /* või */
dynamic[0][5*100+9] = 1;      /* või */
dynamic[1][4*100+9] = 1;      /* jne... */
```

Initialization

```
int *ptr = dynamic[0];
int *end = dynamic[49] + 99; *end = 0;
while (ptr !=end) *ptr++=0;
```

Prototype

```
func(int** vec);
```

Dynamic 2d array

- Stored in *heap.*

Allocation

```
int **dynamic;
dynamic = (int**)malloc(sizeof(int*)*50);
dynamic[0] = (int*)malloc(sizeof(int)*50*100);
for (i=1;i<50;i++) dynamic[i]=dynamic[i-1]+100;
```

Access

```
dynamic[5][9] = 1; /* või */
dynamic[0][5*100+9] = 1;      /* või */
dynamic[1][4*100+9] = 1;      /* jne... */
```

Initialization

```
int *ptr = dynamic[0];
int *end = dynamic[49] + 99; *end = 0;
while (ptr !=end) *ptr++=0;
```

Prototype

```
func(int** vec);
```

Trap & System calls

- Processors provide syscall *n* – trap instruction
- System calls encode arguments, execute syscall to run service *n*
- Then system call decoded & executed on kernel level
- Seem identical to normal functions to programmer
- man syscalls for complete list

Signals

- Interface to interrupts & other conditions on user level
- Sent for 2 reasons:
  - Kernel has detected an event such as divide-by-zero error, illegal memory access etc
  - A process uses kill() system call to send a signal. Can be sent to process itself (shortcut: raise()).

Life of a signal

- *Pending signal* – signal was sent, not received
    - At most one pending signal of any single type for a process, others discarded
- Blocked signals wait
- Ignored signals are discarded
- Other signals are delivered to the process (only once).

Received signal

- Each signal has predefined default action:
  - Process terminates
  - Process termiantes and dumps core
  - Process stops until restarted by SIGCONT signal
  - Process ignores the signal
- Process continues where it was.
  - Unless a system call was interrupted, which sometimes is an error

Signal handling issues

- Pending signals are blocked while handling the same type of signal.
- Pending signals are not queued: if one SIGINT is already pending, other is discarded
- System calls are interrupted on some systems (read(), wait(), accept(), set errno to EINTR)
- For more portable signal handling sigaction() is defined in Posix-compilant systems

Blocking signals

- Processes can explicitly block and unblock signals with sigprocmask() function.
  - sigprocmask(), sigemptyset(), sigfullset(), sigaddset(), sigdelset(), sigismember() - returns 1 if member, 0 if not, -1 on error

Signal handling patterns

- Make the handler as small as possible, possibly changing only one global variable
- Block some of the signals during handling
- Handler which quits might do only cleanup, then re-assign default action to current signal, then raise() the same signal again.

Definition of Thread

- A thread is a unit of execution, associated with a process, with its own thread ID, stack, stack pointer, program counter, condition codes, and general-purpose registers.
- Multiple threads associated with a process run concurrently in the context of that process, sharing its code, data, heap, shared libraries, signal handlers, and open files.

Process vs Thread

- Process – unit of resource ownership:
  - a virtual address space which holds the process image.
  - protected access to processors, other processes, files, and I/O resources.
- Thread – unit of dispatching:
  - Has an execution state (running, ready, etc.)
  - Saves thread context when not running
  - Has an execution stack and some per-thread static storage for local variables
  - Has access to the memory address space and resources of its process

Benefits of using threads instead of processes

- Properly implemented, threads take:
  - Less time to create a new thread than a process, because the newly created thread uses the current process address space.
  - Less time to terminate a thread than a process.
  - Less time to switch between two threads within the same process, partly because the newly created thread uses the current process address space.
  - Less communication overheads -- communicating between the threads of one process is simple the threads share almost everything: address space, in particular. So, data produced by one thread is immediately available to all the other threads.

Benefits of multi-threading

- Improve application responsiveness
- Use multiprocessors more efficiently
- Improve program structure
- Use fewer system resources

Thread Libraries

- Provide interface for thread manipulation:
  - creating and destroying threads
  - passing messages and data between threads
  - scheduling thread execution
  - saving and restoring thread contexts
- Are not a part of C standard
- Example libraries:
  - POSIX threads
  - SOLARIS threads

Thread Control

- Pthreads defines about 60 functions that allow C programs to create, kill, and reap threads, to share data safely with peer threads, and to notify peers about changes in the system state.
- However, most threaded programs use only a small subset of the functions defined in the interface.

Creating threads

```
#include <pthread.h>

typedef void *(func)(void *);

int pthread_create(pthread_t *tid,
pthread_attr_t *attr, func *f,
void *arg);
```

returns: 0 if OK, non-zero on error

```
pthread_t pthread_self(void);
```

Terminating Threads

A thread terminates in one of the following ways:

- The thread terminates *implicitly* when its top-level thread routine returns.
- The thread terminates *explicitly* by calling the pthread_exit() function, which returns a pointer to the return value thread return. If the main thread calls pthread_exit, it waits for all other peer threads to terminate, and then terminates the main thread and the entire process with a return value of thread return.
- Some peer thread calls the Unix exit() function, which terminates the process and all threads associate with the process.
- Another peer thread terminates the current thread by calling the pthread_cancel() function with the ID of the current thread.

```
int pthread_exit(void *thread_return);
```

- Returns 0 if OK, nonzero on error

```
int pthread_cancel(pthread_t tid);
```

- Returns 0 if OK, nonzero on error

Reaping terminated threads

- Threads wait for other threads to terminate by calling the pthread_join function.
- int pthread_join(pthread_t tid, void **thread_return);
- The pthread_join function blocks until thread tid terminates,
- There is no way to instruct pthread_join to wait for an arbitrary thread to terminate.

Detaching threads

- At any point in time, a thread is *joinable* or *detached*. A joinable thread can be reaped and killed by other threads. Its memory resources (such as the stack) are not freed until it is reaped by another thread.
- In contrast, a detached thread cannot be reaped or killed by other threads. Its memory resources are freed automatically by the system when it terminates.
- By default, threads are created joinable. In order to avoid memory leaks, each joinable thread should either be explicitly reaped by another thread, or detached by a call to the pthread_detach function.
- `int pthread_detach(pthread t tid);`
- Note:

```
    pthread_detach(pthread_self()) // used to detach self
```

- Generally threads are detached

Shared variables

- Sharing variables is one of the most attractive features of threads
- It is also most dangerous for creating bugs that are difficult to detect
- Global variables are shared
- Local automatic variables (stack) are not shared but are not protected either (share common virtual address space)
- Local static variables are shared as globals
- Generally: a variable is shared if and only if one of its instances is referenced by more than one thread.

Incorrect sharing

```
#include <pthread.h>
#define NITERS 10000000
void *count(void *arg);
/* shared variable */
unsigned int cnt = 0;
int main() {
   pthread_t tid1, tid2;
   pthread_create(&tid1, NULL, count, NULL);
   pthread_create(&tid2, NULL, count, NULL);
   pthread_join(tid1, NULL);
   pthread_join(tid2, NULL);
   if (cnt != (unsigned)NITERS*2)
     printf("BOOM! cnt=%d\n", cnt);
   else
     printf("OK cnt=%d\n", cnt);
 }
void *count(void *arg)  { // thread routine
 int i;
 for (i=0; i<NITERS; i++)
     cnt++;
 return NULL; }
```

Sharing problem

Code for thread:

```
for (i=0; i<NITERS; i++)
    ctr++;
```

Is actually:

```
     LOAD ctr
     INCREMENT ctr
     STORE ctr
```

Mutexes

- A mutex is synchronization variable that is used to protect the access to shared variables. There are three basic operations defined on a mutex.
  - Init, Lock, Unlock
- int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
- Compile time initialization

  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

Mutex lock and unlock

- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
- These are atomic operations
- Locking is also called aquiring the mutex, unlocking is called releaseing
- At any moment only on thread can hold a mutex

Using mutexes

```
// general code
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);

// thread code
pthread_mutex_lock(&mutex);

// critical section
// access shared variables
pthread_mutex_unlock(&mutex);
```

Correct thread routine

```
/* thread routine */
void *count(void *arg)
{
  int i;

  for (i=0; i<NITERS; i++) {
    pthread_mutex_lock(&mutex);
    cnt++;
    pthread_mutex_unlock(&mutex);
}
return NULL;
```

Deadlocks

- Locking order might cause issues when threads hold mutexes mutually

Classical scheduling

- Two goals:
  - Maximize processor usage
  - Minimize response time of tasks
- Evaluation:
  - Task waiting time
  - Processor throughput
  - Total execution time of tasks
  - Average response time of tasks

Scheduling decisions

- Preemptive or non-preemptive
- Static or Dynamic
- Soft or Hard (Best effort vs Strict)

Scheduling strategies

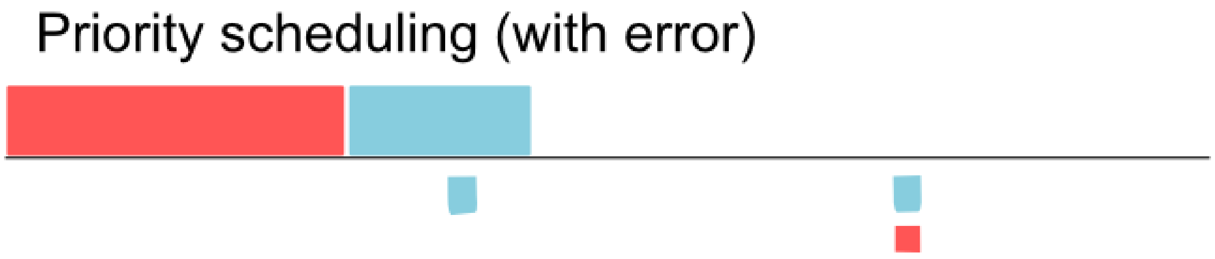
- Round-robin
- First come first served (FIFO queue)
- Prioritized scheduling
- Deadline prioritization
- Shortest first

Implementation details

- Election table
- Priority queue list
- For complex scheduling, *two level* scheduling can be implemented
  - High level decisions on general policy that affect longer periods
  - Lower level scheduler decides reordering for immediate future

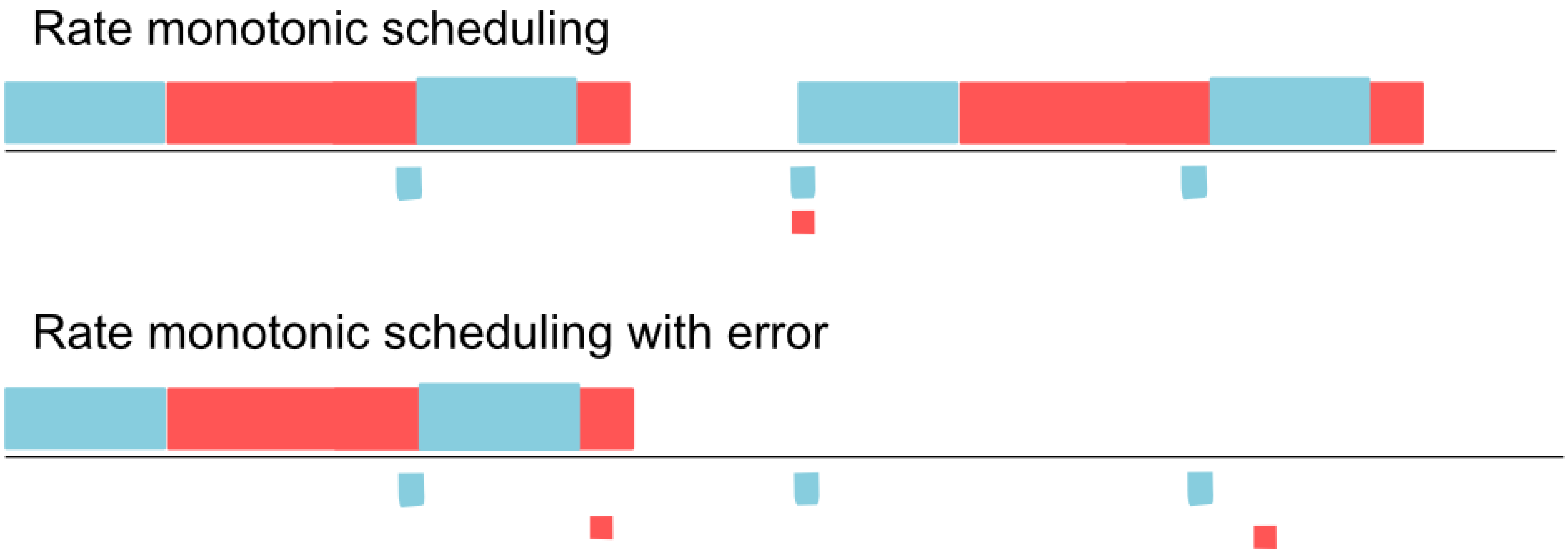
Priority scheduling

- Red is of higher priority, but with longer deadline



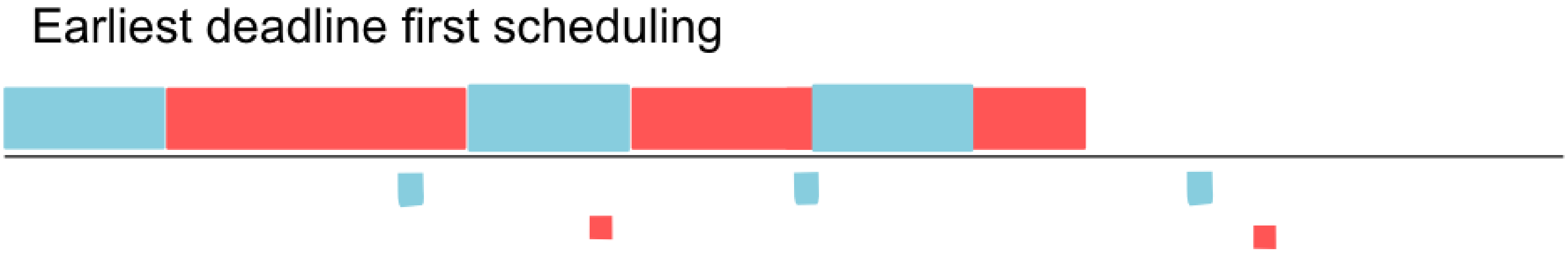
Rate monotonic scheduling

- Priority is inverse of the period
  - Short periods are high priority and vice versa

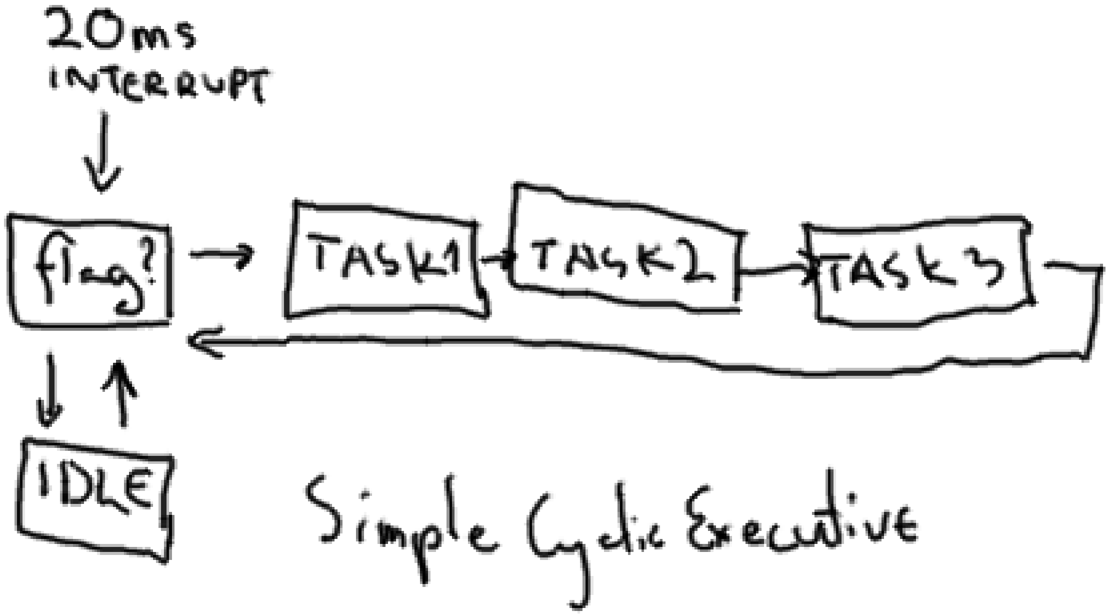

Rate monotonic scheduling
Rate monotonic scheduling with error

Earliest deadline first

- Priorities are assigned according to deadlines



Cyclic Executive



Cyclic Executive

- Offers 3 priorities for tasks
- Interrupt if pre-emption is needed (high priority)
- HW Clock with cooperative scheduling (middle priority)
- Base code with first come first served priority (low priority)
- Inflexible, since does not offer task “aging” or dynamic scheduling
- Simple to test

Tasks exeeding time-slot

- Might be possible to split larger tasks.
- Starts on even ticks, then yields time to others
- Finishes on odd ticks

Posix scheduling

- Phtread scheduling with function:
  - sched_setscheduler()
  - SCHED_FIFO – realtime
  - SCHED_RR – realtime with timeslots
  - SCHED_OTHER – normal scheduling
  - SCHED_BATCH – less prioritized normal (Linux 2.6.16+)
  - SCHED_IDLE – lowest possible priority (Linux 2.6.23+)

SCHED_FIFO

- FIFO processes pre-empt any OTHER and BATCH processes.
- If FIFO process is pre-empted it wil resume as soon as higher-priority processes are blocked
- If becomes runnable, inserted to queue
- sched_setscheduler() puts in front of queue if runnable (ignoring POSIX)
- sched_yield() to send self to end of list

SCHED_RR

- Round Robin enchances FIFO
- Each process gets maximum time quantum
- If runs longer, put to end of queue
- sched_rr_get_interval() will return the quantum

SCHED_OTHER

- Normal way of things
- Processes run according to the nice value
  - nice() or setpriority() used to set the value
- The priority increases each quantum processes are ready, but denied time
- SCHED_BATCH assumes CPU-intensive process which is not interactive and it gets a penalty in priority

Permissions

- CAP_SYS_NICE permission needed
- Unprivileged processes can set SCHED_OTHER for the same user
- Can be overriden
- Processes running under SCHED_IDLE cannot change to something other without permissions

Miscellaneous

- Child processes inherit their parents' scheduling policy
- Real-time processes need memory locking (mlock() and munlock()) to avoid paging delays
- A non-blocking loop in _FIFO or _RR priority will lock the computer unless a shell is scheduled on same level prior to running it for killing it off. Remember when debugging.

Process creation and destruction

- Unix offers 4 system calls for process creation, destruction and waiting for them to finish:
  - exec() family
  - fork()
  - wait()
  - exit()

Loading of a process

- Binary executable contains header, (program) text, data, relocation information and symbol table. Text and data will be loaded with program

Executable file	Process memory
HEADER	TEXT
TEXT (program)	DATA
DATA (initialized)	(BSS)
BSS (=uninitialized data)	free mem
RELOCATION	STACK
SYMBOL TABLE (can be stripped)	USER BLOCK (in kernel adr space)

exec() family

- Exec loads a binary executable into memory and starts a process.

```
extern char **environ;
int execl( const char *path, const char* arg, ...);
int execv (const char *path,  char *const argv[]);
int execle(const char *path,
         const char *arg, ..., char * const envp[]);
```

- execl : full file path, arguments as chars
- execv : full file path, arguments as array
- execle : full file path, arguments as chars, environment

Environment

- getenv()
- see also putenv()

exec() family (2)

- The real function is execve()

fork()

wait()

waitpid()

system()

atexit()

Demon

Zombie

Process

Files in UNIX

- Everything is a file:
  - File is a file
  - Directory is a file
  - FIFO special file (named pipe)
  - Character special file (subtype of a device file)
  - Block special file (subtype of a device file)
  - Symbolic link file

File attributes

Attribute	**Value meaning**
File type	Type of file (regular, directory, fifo, …)
Access permission	File access permissions for different users
Hard link count	Number of hard links of a file
UID	User ID of file owner
GID	Group ID of file
File size	File size in bytes
Last access time	Time the file was last accessed
Last modification time	Time the file was last modified
Last change time	Time the file attribute was last changed
Inode number	Inode number of the file
File system ID	File system ID where the file is stored

NB: File name is not an attribute!

Inode

- File data is held in a structure called inode.
- File-system keeps them in a table called *inode table*.
- Inode number uniquely defines a file

File status

```
int stat(const char *file_name, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
```

- Returns information about the file node
- fstat() is analogous to stat(), but takes the file descriptor as the first argument.
- lstat() is like stat(), but returns the data about symbolic link in stead of linked file

Stat structure

```
#include <sys/stat.h>, <sys/types.h>
struct stat
{
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device) */
    off_t         st_size;     /* total size, in bytes */
    unsigned long st_blksize;  /* blocksize for filesystem I/O */
    unsigned long st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last data modification */
    time_t        st_ctime;    /* time of last inode change */

};
```

There are macros to manipulate the data more easily!

Directory

- In UNIX a directory is a file which contains pairs:
    - inode: name
- It contains the files "." and "..", which correspond to the same directory and to the one above the current, correspondingly. They contain their inodes.
- Root directory / has .. file which points to itself
- If inode is marked as 0, the entry is free to write

Unix IO

- Why necessary to know
  - Helps to understand other concepts such as process creation anomalies
  - Sometimes necessary to use: file metadata, network programming risks

File Descriptor

- File for a process is a small positive integer named "File Descriptor"
- Sometimes the word "channel" is used.
- Predefined descriptors:
  - 0 standard input
  - 1 standard output
  - 2 standard error
- Descriptor is a file table index

File sharing

- Open files in kernel are in 3 structures:
  - Descriptor Table: unique for a process points to:
  - File Table: shared by processes, holds file position, reference count, points to V-node table:
  - V-node Table: shared by processes, basically holds most of stat() information
- Fork() copies Descriptor Table

Unix IO

- File: sequence of bytes
- Open files: open()
- Change current file position: seek()
- Read and write: read(), write()
- Close: close()

Notes about read() and write()

- They return how many bytes were moved.
- Sometimes these calls return before you have sent all of the data (network reading would be a prime example)
- Buffering happens only on file system level

fopen(), fread(), fwrite() etc

- Wrap open(), read(), write() to create streams
- They are buffered and therefore preferred.
- Stream from fileno:

```
FILE *fdopen (int fd, const char *mode);
```

- File number from stream:

```
int fileno(FILE *stream);
```

dup(), dup2(), dup3

- Copy the open file descriptor
- dup() - new descriptor is the lowest one
- dup2(int oldfd, int newfd) – close new;
  copy old to new
- dup3(int oldfd, int newfd, int flags) –
  close new; copy old to new
  flags : O_CLOEXEC – prevents race conditions on threaded programs

Example of output redirection

- Open file to get descriptor
- Use dup2() to replace descriptor of stdout (1)
- Printf sends data to a file now
- If you exec a program; output also sent to file

What to use

- Use Standard IO calls if you can
- Use Unix IO if you must
    - Meaning: mostly for networking & control/speed

Hardlink

```
int link(const char *oldpath, const char *newpath);
```

- link() hardlinks oldpath to newpath.
- When newpath exists, it is not overwritten
- The new file acts as a pointer to data. The files are identical as they point to the same inode.
- Returns 0 on success, -1 on error and sets an errno

Softlink

```
int symlink(const char *oldpath, const char
  *newpath);
```

- Creates a softlink (or Symbolic link)
- Acts like a Windows shortcut and can be either:
  - Relative: ../text.txt
  - Absolute: /home/irve/text.txt

unlink()

```
int unlink(const char *pathname);
```

- Changes the inode of pointed file to 0, then decreases the reference counter to the file. If the counter reaches 0, deallocates the data.
- PS! In reality the data gets replaced only when the last process which uses the file is stopped. You can open a temporary file, unlink it and still access it!

File permissions

```
int access(const char *pathname, int mode);
```

- access() chechs if the processes have the privilege to access the file for either writing, reading or verifying its existence.
- mode is a bitmask of permissions R_OK, W_OK, X_OK, ja F_OK (latter the existence of the file)
- Returns 0, when the access is granted, -1 otherwise.

Granting access to files

```
int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
```

- Defines the file permissions to the file
- mode is the result of OR with the following constants:

```
#define S_IRWXU 0000700    /* RWX mask for owner */
#define S_IRUSR 0000400    /* R for owner */
#define S_IWUSR 0000200    /* W for owner */
#define S_IXUSR 0000100    /* X for owner */
#define S_IRWXG 0000070    /* RWX mask for group*/
#define S_IRGRP 0000040    /* R for group */
#define S_IWGRP 0000020    /* W for group */
/* jne... */
```

Setting the owner

```
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

- Change owner and/or group of the file pointed by its path or file descriptor
- Owner can only be changed by the root user.
- User can set the group of the file to the groups to which it belongs.
- The root can set the group as needed

Permissions to new files

```
mode_t umask(mode_t mask);
```

- Umask sets permissions to the new files

  umask = mask & 0777

- The bits set in umask are *removed.*
- Permission bits = mode & ~(umask)
- Widely used umask is 022, which creates files with permissions of 0666 & ~022 = 0644 = rw-r--r--
- The function always succeeds and it returns the previous umask value.

Directory management

```
int mkdir(const char *path, mode_t mode);
```

- mkdir() creates a new directory

```
int rmdir(const char *path);
```

- rmdir() removes the directory
- The directory must be empty and can only contain the files . and ..

Working directory

```
int chdir(const char *path);
int fchdir(int fildes);
```

- chdir() changes the working directory to the pointed one.
- Returns a pointer to current working directory

```
char *getcwd(char *buf, size_t size);
```

- buffer size must be at least 1 larger than directory name.
- When buf is NULL, the function allocates size bytes, otherwise stores it to pointer

Directory Content Stream

- You can open directory streams

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *dirname);
```

- Opens dirname as a stream. Positioned on the first entry.

```
int closedir(DIR *dirp);
```

- Close the stream *stream*

Listing stream entries

```
struct dirent *readdir(DIR *dirp);
```

- Return the structure to dirent pointer, which contains the current entry
- NULL is returned after the last entry
    - <dirent.h> file describes the data structure
    - readdir() overwrites the data after running
    - POSIX standard states that dirent contains the field char d_name[], which has no determined length, to a maximum of NAME_MAX chars, null terminated. Other fields are not portable, while d_namelen field is often present

Programming an Operating System

- Programming
  - Tasks
  - Scheduling
  - Context Switches
  - Mutexes
- Michael Barr, Programming Embedded Systems in C and C++

Os under discussion ADEOS

ftp://ftp.oreilly.com/examples/nutshell/embedded_c/

- A Decent Embedded Operating System

Task States

- Ready <> Running > Waiting > Ready
- Running-Ready – task switching
- Waiting = Blocking
- Only one running at time

Task Creation

- ADEOS allows only task creation
- When (and if) a task function returns, task is deleted
- Construction needs a function, priority and stack size.

Scheduler

- Decides which task gets to run
- Example uses priority list for task scheduling
- FIFO behaviour in case of conflicts
- 255 priority levels

Scheduling Points

- Events during which scheduler is invoked
- (Task creation has one)
- os.schedule() runs

Clock Tick

- Runs on timer interrupts
- Waking the tasks which wait for timer to expire

Ready List

- Ordinary linked list
- Priority queue
- Next task always on top

Idle task

- Empty loop
- Hidden from application developer
- Has id and priority of 0
- Always ready

schedule()

- Looks for top task, if it is the running task, good
- Otherwise switches tasks
- Note that tasks start only after initialization of scheduler (since scheduler is invoked on task creation too)

Context Switch

- Architecture specific
- Must be written in assembler language
- restoreContext() and saveContext()
- Tasks wake in saveContext() and a clever jump is done to distinguish between saving and restoring.

Mutexes

- Multitasking aware binary flag
- Setting and clearing are atomic
- Interrupts are disabled
- Implementation: flag + waiting list
- Initialization simple

Mutex setting and clearing

- Setting
  - If taken, process goes to waiting state until released
  - Scheduling called
- Releasing
  - Does not block
  - On release a context switch might occur due to scheduling

Optimizing compilers

- Same code has different representations
- Some are more efficient (yet less readable)
- Handwritten assembler code is often optimal

How to optimize: use -O option for gcc

Why not the default option?

Limitations

- Never alter the correct program behaviour
- Their understanding of program behaviour is limited
- Compilation must be fast

Optimization blockers

```
void foo1(int *xp, int *yp)
{
    *xp += *yp;
    *xp += *yp;

}
void foo2(int *xp, int *yp)
{
    *xp += 2* *yp;

}
```

- Similar code
- First uses 6 memory references, second 3
- Would be possible to optimize?
- What happens if pointers are equal?

Optimization blockers

```
int f(int);
int func1(x) {
    return f(x) + f(x) \
                + f(x)+ f(x);
}

int func2(x) {
    return 4*f(x)
}
```

- func2() faster
- but only in case f() is without side effects
- Usually not tested

Program performance assessment

- Speed of processors can vary
- Useful measure for examples: *cycles per element.*
- What is the code overhead for any array element

Base example

```
typedef struct {
	int len;
	data_t *data;
} vec_rec, *vec_ptr;

typedef int data_t; //or float for experiments

#define IDENT 0 // or 1
#define OPER + //or *

// actual implementation of vectors less interesting
```

Base implementation

```
void combine1(vec_ptr v, data_t *dest) {
    int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OPER val;
    }
}

//                              int          float
// unoptimized          +42 *41      +41 *160
// optimized -O2             +31 *33      +31 *143
```

Moving calculations from loop

```
void combine2(vec_ptr v, data_t *dest) {
    int i;
    int length = vec_length(v)
    *dest = IDENT;
    for (i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OPER val;
    }
}

// old optimized -O2        +31 *33     +31 *143
// move vec_len         +22 *21     +21     *135
```

Reducing function calls

```
void combine3(vec_ptr v, data_t *dest) {
        int i;
        int length = vec_length(v)
        data_t *data = get_vec_start(v);
        *dest = IDENT;
        for (i = 0; i < length; i++) {
                *dest = *dest OPER data[i];
        }
}

// move vec_len                +22 *21      +21      *135
// direct data access                  +6  *9   +8  *117

// Note: we gained speed by losing in abstraction & modularity
```

Decompilation analysis

Compare 3
dest in edi, data in ecx, i in edx, length in esi

```
.L18                                  :loop
movl (%edi), %eax                     Read dest
imull (%ecx, %edx, 4), %eax           Multiply data
movl %eax, (%edi)                     Write *dest
incl %edx                         i++
cmpl %esi, %edx                       Compare
i:length
jl .L18                           if < goto loop
```

Compare 4
data in eax, x in ecx, i in edx, length in esi

```
.L24                                  :loop
imull (%eax, %edx, 4) %ecx            Multiply by data[i]
incl %edx                         i++
cmpl %esi, %edx                       Compare i:length
jl .L24                           If <, goto loop
```

Storage variable

```
void combine4(vec_ptr v, data_t *dest) {
    int i;
    int length = vec_length(v)
    data_t *data = get_vec_start(v);
    data_t x = IDENT;
    *dest = IDENT;
    for (i = 0; i < length; i++) {
        x = x OPER data[i];
    }
    *dest = x;
}

// direct data access        +6  *9   +8  *117
// temporary variable        +2  *4   +3  *5

// Why not automatic?
```

Different functions

- combine3(v, get_vec_start(v) + 2);
- combine4(v, get_vec_start(v) + 2);
- Last element used for destination

c3	c4
2 3 5	2 3 5
2 3 1	2 3 5
2 3 2	2 3 5
2 3 6	2 3 5
2 3 36	2 3 5
2 3 36	2 3 30

Aside: further optimizations

- Modern processors use pipelining, parallelization
- Can be used for advantage

Loop unrolling

```
void combine5(vec_ptr v, data_t *dest) {
    int i;
    int length = vec_length(v)
    int limit = length - 2;
    data_t *data = get_vec_start(v);
    data_t x = IDENT;
    *dest = IDENT;
    for (i = 0; i < limit; i += 3) {
        x = x OPER data[i] OPER data[i+1] OPER data[i+2];
    }
    for(; i < length; i++) {
        x = x OPER data[i];
    }
    *dest = x;
}

// temporary variable          +2   *4  +3  *5
// loop unroll      x3         +1.3 *4  +3  *5

// How many unrollings optimal?
```

Pointer code

- Use pointer code for speedups

```
void combine4p(vec_ptr v, data_t *dest) {
    int length = vec_length(v)
    data_t *data = get_vec_start(v);
    data_t *dend = data +length;
    data_t x = IDENT;
    for (; data < dend; data++) {
        x = x OPER *data;
    }
    *dest = x;
}

// temporary variable        +2  *4   +3  *5
// pointer code          +3  *4   +3  *5

// Mostly useless here, but really depends on compiler/platform
// Readability often priority
```

Parallelism

- It's often good to parallelize code

```
void combine6(vec_ptr v, data_t *dest) {
        int length = vec_length(v);
        int limit = length - 1;
        data_t = *data = get_vec_start(v);
        data_t x0 = IDENT; data_t x1 = IDENT;
        int i;
        for(i = 0; i < limit; i+=2) {
                x0 = x0 OPER data[i];
                x1 = x1 IOPER data[i+1];
        }
        for(;i < length; i++) {
                x0 = x0 OPER data[i];
        }
        *dest = x0 OPER x1;
}
// loop unroll      x3                  +1.3 *4 +3  *5
// parallelize by 2                     +1.5 *2 +2  *2.5
```

End results

- For most things unroll x8, parallel x4 is best
- For integer addition, best unroll x16
- **on Pentium III**

- Your results would be different

Less predictable features

- Cached data
- Load/store latency
- Branch prediction (predictive execution)

Cache

```
struct a {
    int a;
    int b;
    int c;
    int d;
};
```

```
struct a {
    int a;
    int b;
};
```

This is faster!

Row vs Column based access

- For two-dimensional arrays, visit by rows, not by columns!

What to do in real-life?

- High level design: choose appropriate algorithms and data structures.
- Basic coding principles
    - Eliminate exessive function calls, move computations out from loops, compromise on modularity
    - Eliminate unnecessary memory references. Use temporary variables to hold intermediate results. Store results only when final value calculated

Real-Life 2

- Low level optimizations
  - Try different pointer-array code
  - Reduce loop overhead by unrolling them
  - Pipelined architecture: Find ways to split iterations when needed
- Avoid introducing errors by unittesting. Benchmark to find anomalies

Code Efficiency

- To avoid macro definitions, C99 has keyword: inline
- When checking for alternatives, use switch carefully:
  - Put more popular cases first
  - Use function pointer arrays
- Inline assembly
- Gloal variables (but maintenance nightmare)

Code Efficiency

- Fixed-vs-floating point: former is faster
  - Small amount of decimal places: val << 2
- Use native word size (bus + registers are faster)

Code Size

- Standard library routines refer to other functions. Write your own printf()
- Goto is bad, yet good for jumping out of nested loops.

```
Int fun(void) {
    /* working */
    goto CLEANING; /* in case of error */
    /* more work */
    return SUCCESS;
CLEANING:
    /* cleanup here */
    return FAILURE;
}
```

Memory Usage

- Reduce dependence on stack & heap by using ROM for constant values (declare them as const)
- Some constants change: use flash memory & technicians
- Stack space estimation: fill memory with some pattern; check changes after running

Power-saving

- Necessary for battery-powered devices
- Processor modes (PXA255 example)
  - Turbo – minimize memory access due to waiting
  - Run – default mode
  - Idle – processor not clocked, peripherals operate
  - Sleep – lowest power state
- Clock frequency – tricky, needs HW knowledge
- Reduce external memory access – cache, processor memory

Optimization problems

- Dead code elimination
  - Declare variables as volatile
- Debugging more difficult: breakpoints missing, functions split and code different

Networking

- TCP/IP protocol
- On hardware level we have network adapter which uses system bus to communicate with memory (usually with DMA)

Getting and creating addresses

- Addresses have structure:

```
struct in_addr {
      unsigned int s_addr; /* network byte order */
};
```

- Network byte order: big-endian

```
unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int
hostshort);

unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned long int
netshort);
```

Hostname conversion

```
int inet_aton(const char *cp,
                    struct in_addr * inp);
char *inet_ntoa(struct in_addr in);
```

Domain names

- Domains are structured
  - dijkstra.cs.ttu.ee -> ee > ttu > cs > dijkstra
- Host entry structures

```
struct hostent {
        char *h_name; /* official name */
        char **h_aliases; /* null-terminated array of domains */
        int  h_addrtype; /* address type AF_INET */
        int  h_length;    /* address length */
        char      **h_addr_list; /* null terminated array of in_addr
structs*/
};
```

- Retrieval and query

```
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const char *addr, int
len, 0);
```

Domain name mappings

- One to one
  - Host has only one name and address
- Multiple domains to one address
  - dragon.ee www.dragon.ee
- Multiple addresses to multiple domains
  - most of google
- Consider when working with host entries

Internet connection

- Communication done by sending streams of bytes over the wire
- Full duplex: you can both read and write
- Point to point: connects a pair of processes
- Socket: endpoint for communication
  - address:port
- Connection is a pair of sockets

Socket interface

- Berkeley sockets
  - developed by their researchers, distributed with Unix 4.2 BSD kernel and distributed to universities and labs
- Socket from the view of kernel: communication endpoint
- Socket from a programs view: an open file

Socket addresses

- Socket address; general and specific

```
struct sockaddr {
    unsigned short  sa_family;   /* protocol family */
    char                     sa_data;          /* address data */
}

struct sockaddr_in {
    unsigned short  sin_family; /* address family AF_INET */
    unsigned short  sin_port; /* port number in network byte order
*/
    struct in_addr   sin_addr; /* IP address in network byte order */
    unsigned char   sin_zero[8]; /* pad to sizeof(struct sockaddr) */
}
```

Overview of interface

- Client
  - socket()
  - connect()
  - read()/write()
  - close()

- Server
  - socket()
  - bind()
  - listen()
  - accept()
  - read()/write()
  - close()

socket()

- Creates a socket descriptor

```
// int socket(int domain, int type, int protocol);

clientfd = socket(AF_INET, SOCK_STREAM, 0);
```

connect()

- Establish a connection with given socket address
- Blocks until successful or error occurs

```
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

bind()

- Associate a socket with an address and port

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

- Convert active socket to listening socket

```
int listen(int sockfd, int backlog);
```

- Accept incoming connection
  - note that a new file descriptor is returned; why?

```
int accept(int listenfd, struct sockaddr *addr, int
*addrlen);
```

Notes

- When a connection is terminated while it is read, a signal is generated
  - EPIPE: Broken pipe, program terminates unless handled
- There are additional functions to replace read() and write() with sockets
  - send(), recv(): et specify additional flags for sending and receiving data
- For UDP you can use recvfrom() sendto()
  - connect() or bind()/listen() are not needed for them

Bit fields

- An alternative to flag variables. It is possible to "pack" values into a structure using : notation to signify the amount of bits allocated. Assignment and use like a reg

```
struct {
   unsigned int is_keyword: 1;
   unsigned int is_extern: 1;
   unsigned int is_static: 1;
} flags;

flags.is_static = 1;
flags.is_keyword = 0;
```

Bit-fields (2)

- Bit-fields are declared as unsigned integers to avoid sign problems.
- For use in expressions they are cast into an integer automatically.
- Internal implementation depends on architecture.

Variable arguments

- To declare functions with variable arguments, just type … like you would not care about what goes there...

```
void foo( int arg1, int arg2, ...) {
```

Ellipsis here...

- How to get to them?

Getting the other arguments using a macro (easier method)

- #include <stdarg.h>
- Macros:

```
void va_start(va_list ap, last);
type va_arg(va_list ap, type);
void *va_end(va_list ap);
```

- Possible implementation

```
typedef char *va_list;
#define va_start(ap, v)  ((void) (ap = (va_list) &v + sizeof(v)))
#define va_arg(ap,type)   (*((type *)(ap))++)
#define va_end(ap)  ((void) (ap = 0))
```

More info: 'man stdarg' or 'man vararg'

Var args (example)

```
 #include <stdarg.h>
 #define MAXARGS     31
 void f1(int n_ptrs, ...) {
     va_list ap;
     char *array[MAXARGS];
     int ptr_no = 0;

     if (n_ptrs > MAXARGS)
         n_ptrs = MAXARGS;
     va_start(ap, n_ptrs);
     while (ptr_no < n_ptrs)
          array[ptr_no++] = va_arg(ap, char*);
     va_end(ap);
     f2(n_ptrs, array);
}
```

Posssible problems

- Disruption of work
- Data integrity
- Privilege escalation
- Data leakage

(CIA triad: confidentiality, integrity, availability)

Guard your inputs

- Input is a lie!

Command line

- Execve() lets you add \0 chars where not expected
- Setuid/setgid problems

Environment variables

- You have full control over environment
- IFS variable (telling what character separates the commands in a shell)
- When you use system() function, causes problems
- Solution: purify the environment; use only what needed
- (setuid/setgid problems)
- User gets to include random .so files using LD_PRELOAD (and change it in ~/.environment variable)

Filenames

- Sneaky . .. and / possibilities
- Buffer overrun with PATH_MAX problems
- ../*/../*/../*/../* denial of service when using glob() function

Passwords

- Problem: how to ask password so that it does not reach the screen of the user.
- "Solution":

```
#include <unistd.h>
char * = getpasswd(char * prompt)
```

- Connects to "real" terminal /dev/tty , if cannot, tries stdin ja stderr . Blocks INTR, QUIT and SUSP commands in terminal.
- Terminal is flushed before and after password is typed

What does getpass() do?

- Prints the prompt
- Goes into noncanonical mode, turns off echo, restores the terminal state after function
- Due to lack of *thread-safety*, and exclusion from POSIX standard, general recommendation not to use it. Write yourself (or find a working solution).
- For important applications the good practice is to encrypt the password upon recieval and overwrite the original buffer.

Encryption: crypt()

- Encrypts using DES (broken) or MD5 (broken soon); Blowfish or SHA-256 / SHA-

```
char * crypt(constchar* key, const char* salt);
```

- Belief that hash, once calculated cannot be reversed in a reasonable time.
- salt: if two letters, chooses DES, if MD5, start the string as 1 + 8 chars, which end in $ or \0
- For Blowfish etc see manpage; you change the id

Salt

- Salting prevents dictionary attacks using rainbow tables.
- Output being salt + $ (when missing) + hash
- Salt should be a random string when the password is stored
- For checking the password, provide the previous output of crypt() as the salt, and compare *salt* to crypt() result. (As $ ends salt, you can provide the whole result for salt argument)

Storage of passwords

- Hash is problematic; MD5 has over 9000 million tries per second
- You can calculate the hash repetitively on existing hashes: try it 100 times to send attacker away
- The attacker using a GPU will be thwarted
- Don't invent stuff, use the bcrypt library

Stack smashing

- Canary
  - Ubuntus uses by default, others not
- Address space randomization (ASLR)

Standard library problems

- Mostly the lack of input length checks

Malloc

- Double free() really problematic
- You can control the behaviour by setting MALLOC_CHECK 2 environment variable
- After the release, use a macro to set the pointer to NULL

Non-negative values

- Use an unsigned type

Compilation steps

  - Source code
- Preprocessing
- Compiler
  - Assembly code
- Assembler
  - Object code
- Linker

Makefiles

- Compilation must be an atomic process
  - Otherwise the programmer debugs larger chunks
- Save time on compiling large projects
- Help with modularity
- Compile unfamiliar programs without thinking

Makefile layout

- File uses tabs instead of spaces
- Named either "makefile" or "Makefile"

```
target: prerequisite1 prerequisite2
	commmand

myprog: myprog.c myprog.h
	gcc myprog.c -o myprog
```

Laying out a program

- Modules:
  - Spread the program over several .c files
  - Use .h files for function prototypes and data
- For .h:

```
#define _header_h_
#ifndef _header_h_
...
#endif
```

.h files

- Describe the "interface"
- Function prototypes
- Data types and structures declared
- const and #define
- #includes for other headers

Makefile with separate linking

- Simple makefile which compiles in several steps
- Note first and last directives

```
# Makefile for the sample
sample:  sample.o  my_math.o
        gcc –o sample sample.o my_math.o
sample.o:  sample.c  my_math.h
        gcc –c sample.c
my_math.o:  my_math.c my_math.h
        gcc –c my_math.c
clean:
        rm sample *.o core
```

Makefile (2)

- Make checks upon running the command whether it needs to compile anything by looking at file dates and their dependencies
- So it tries to only compile the minimal set

clean Convention

- Makefiles often specify (and programmers expect) a way to clean out the temporary files

  ```
  make clean
  ```

  clears the files if specified

- If for some reason you need to recompile and make does not want to:

  ```
    touch filename.h
  ```

Implicit rules

- Make can compile when some rules are omitted
- It "knows" how to compile from .c to .o, for example, if the names match and only target and prerequisites are present

Implicit Rule Example

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

# Note special .PHONY keyword here!!!
.PHONY : clean
clean :
        rm edit $(objects)
```

.PHONY

- Make clean does not have prerequisites and thus will always run
- If someone makes a file named "clean" into directory, cleaning will fail
- .PHONY tells that we are dealing with a command, not a target file

Macros

- You can define macros in a makefile to avoid repeating yourself
- Macros are defined as:

  ```
  name = value
  ```
- Used as:
  $(name) or
  ${name}

Multiple directories

- Sometimes you need to split program modules into directories
- Every module has its own makefile
- Program has a directory for every module and one for all of the .h files
- Main Makefile creates the program
- Makefiles in modules make the corresponding object files

Directory Example

- C program uses Stack module and Queue module and has a main.
- Program has 7 files: StackTypes.h, StackInterface.h, QueueTypes.h, QueueInterface.h, StackImplementation.c, QueueImplementation.c and Main.c
- The target is a program in a directory which contains subdirectories Stack, Queue and Include (containing every .h file)

Stack dir

- StackImplementation.c and the makefile:

```
export: StackImplementation.o

StackImplementation.o: StackImplementation.c \
                       ../Include/StackTypes.h \
                       ../Include/StackInterface.h
           gcc -I../Include -c StackImplementation.c
   # substitute a print command of your choice for lpr below
   print:
           lpr StackImplementation.c
   clean:
           rm -f *.o
```

Queue dir

- QueueImplementation.c and the makefile:

```
export: QueueImplementation.o
   QueueImplementation.o: QueueImplementation.c \
                          ../Include/QueueTypes.h \
                          ../Include/QueueInterface.h
           gcc -I../Include -c QueueImplementation.c
   # substitute a print command of your choice for lpr
     below
   print:
           lpr QueueImplementation.c
   clean:
           rm -f *.o
```

Notes

- -I (capital i) tells where the library includes can be found; use commas for multiple; don't use spaces
- This enables us to gather .h files in one location for ease of reference
- The \ symbol before line-end escapes it.

Main directory

- Main includes main.c and makefile:

```
 export: Main
 Main: Main.o StackDir QueueDir
        gcc -o Main Main.o ../Stack/StackImplementation.o \
                           ../Queue/QueueImplementation.o
Main.o: Main.c ../Include/*.h
        gcc -I../Include -c Main.c
StackDir:
        (cd ../Stack; make export)
QueueDir:
         (cd ../Queue; make export)

#continues
```

Main directory (2)

```
print:
        lpr Main.c
printall:
        lpr Main.c
        (cd ../Stack; make print)
        (cd ../Queue; make print)

clean:
        rm -f *.o  Main  core
cleanall:
        rm -f *.o  Main  core
        (cd ../Stack; make clean)
        (cd ../Queue; make clean)
```

Notes

- Unix command sequence in brackets makes them run as a subprocess
- So the directory changes apply, but only for the subprocess itself

Let's Add Macros

```
CC = gcc
HDIR = ../Include
INCPATH = -I$(HDIR)
DEF =  $(HDIR)/StackTypes.h  $
  (HDIR)/StackInterface.h
SOURCE = StackImplementation
export:  $(SOURCE).o

$(SOURCE).o:  $(SOURCE).c  $(DEF)
            $(CC)  $(INCPATH)  -c  $(SOURCE).c
print:
            lpr  $(SOURCE).c
clean:
```

GNU Make

- GNU Make has a ton of features such as:
  - Control structures and conditional clauses, cycles
  - Simple text modifying features
  - Automatic variables referring to target/source
- Gmake manual: http://www.gnu.org/software/make/manual/make.html

Valgrind

- Memory debugging and profiling tool
- Makes your program really slow, but documents it while it runs
- Usage:

```
valgrind --tool=memcheck prog args
```

- Tools: memcheck, callgrind, cachegrind
- For callgrind run callgrind_annotate

Don't forget

- gdb
  - and (somewhat) graphical ddd
- hexdump
- objdump

I18n ja L10n

- Internationalization – enabling translation support for a program
- Localization – translation and modifying a program to suit local idioms and customs

ASCII

- Time before ASCII luckily outside of our scope
- ASCII standard: characters with value of less than 32 are non-printable (bell sound or feeding a new paper into the printer)
- Characters above 127 free for anyone to use

IBM PC codepage (437)

- ASCII compatible
- For some European languages é and è letters
- Horizontal and vertical table-drawing characters
- Remember the older cashier screens
  - (For those you can use the curses library)
- What about Hebrew, Asian languages, Russian?

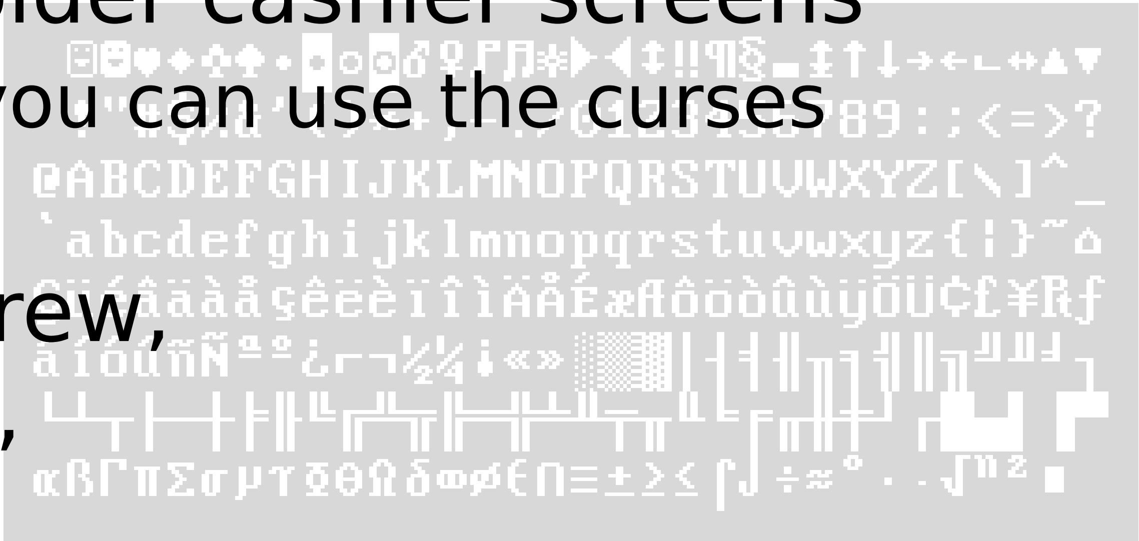

☺☻♥♦♣♠•◘○◙♂♀♪♫☼►◄↕‼¶§▬↨↑↓→←∟↔▲▼
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~⌂
ÇüéâäàåçêëèïîìÄÅÉæÆôöòûùÿÖÜ¢£¥₧ƒ
áíóúñÑªº¿⌐¬½¼¡«»░▒▓│┤╡╢╖╕╣║╗╝╜╛┐
└┴┬├─┼╞╟╚╔╩╦╠═╬╧╨╤╥╙╘╒╓╫╪┘┌█▄▌▐▀
αßΓπΣσµτΦΘΩδ∞φε∩≡±≥≤⌠⌡÷≈°∙·√ⁿ²■

Illustration: https://en.wikipedia.org/wiki/File:Codepage-437.png

Unicode to the rescue

- Unicode is a collection of *Code Points*
- Every *Code Point* refers to a symbol which sometimes is a character in some language like A or Õ, or something else like ﬃ (U+FB03)
- They exist in a rather plentiful manner (cat faces etc)
- You can refer to the *Code Points* using some specific encoding

Coding: UTF-8

- A specific coding
- Lower 127 characters are ASCII compatible
- Further bytes represent multibyte characters
- Linux has mostly completed standardization to UTF-8; Use of anything other than this should be considered problematic

In practice

- GNU library: libiconv
  http://www.gnu.org/software/libiconv/
- fopen("file.txt", "r, ccs=UTF-8");
- wchar_t data-type
- fgetc() >> wint_t fgetwc(FILE * stream)
- EOF >> WEOF

Linux support

- The input from the keyboard (what you get from terminal stdin) is converted to UTF-8 stream (console driver does this work)
- The output to console is decoded using a UTF-8 decoder and is presented using a 16-bit font
- BOM does not exist (the FE FF)

Two approaches

- Keep internal data in UTF-8
- Keep data in its decoded form and convert only upon outputting it
  - A character would be an object in memory in this case

Usage

- Define locale in environment:
  LANG=et_EE (for output in ISO 8859-1)
  LANG=et_EE.UTF-8 (for output in UTF-8)
- #include <locale.h>
- setlocale() - LC_CTYPE or LC_ALL arguments
- command:
  locale -a shows the locales installed into system

Gettext

- Solutionf from Sun Microsystems
- Copied by GNU project
- Quite standard and widely used

Workflow

- Write your program using gettext() function and locale registration
- Use xgettext program to gather your strings into .pot file
- Create translation files for your target language using msginit command
- Translate
- Cinvert translation into binary using msgfmt program
- Put the result into /usr/share/locale/XX/LC_MESSAGES (XX is language; et or de, for example)

Hello.c

```
#include <libintl.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  setlocale( LC_ALL, "" );
  bindtextdomain( "hello", "/usr/share/locale" );
  textdomain( "hello" );
   printf( gettext( "Hello, world!\n" ) );
   exit(0);
 }
```

Source: http://oriya.sarovar.org/docs/gettext_single.html

Explanation

- setlocale() gathers the users preferences for language and its customs (date formats, week starting date, currency, etc)
- bindtextdomain() tells that „hello“ program can find its translation under /usr/share/locale (this is the default and could be skipped)
- textdomain() tells that language set is named "hello" in all of the languages
- **gettext()** should wrap all the strings; alias

Virtual memory

- Virtual memory gives you a large address space (from 0 to "really large", latter depending on system)
- It is not continous: every address is not usable
- Divided into pages. (usually 4kB)
- Every page is located in the memory (*frame*) or somewhere else (swap disk)
  - Emptied memory marked as: "contains zeroes"
- Virtual addresses are mapped to either real frames or swap space

Page fault

- As there is more virtual memory than real memory, we must swap pages between "real" and "backup" memory
- It's called paging
- Page fault: an attempt to read memory which is not in the RAM
- When page fault happens, the couple of milliseconds needed for memory access suddenly take a far greater amount of time
- Hard disks become noisy

How to get memory

- There are two ways of getting memory
  - upon starting your program (exec) when your program gets its memory space and is allocated space in there for its constants, code text and stack space
  - in your program:
    - auto variables
    - malloc
    - mmap: map a file into virtual memory
  - fork: *copy on write trikk*
- When program stops, its memory space collapses

Tracing memory

- You can trace memory allocation using

```
void mtrace(void);
void muntrace(void);
```

- Use environment variable named MALLOC_TRACE to specify the file which will store the statistics about memory allocation and release
- The first activates, the second deactivates trace
- C

```
mtrace progamname mtrace-log
```

- Result is not human-readable – use a command:

mmap()

- mmap() maps a file into virtual memory (or creates an anonymous mapping)
- Sometimes usefil:
  - We can read only parts of file which we use
  - mmap() lets you write changes back to disk
  - we can open files larger than mem+swap

```
void * mmap (void *address, size_t length, int protect,
               int flags, int filedes, off_t offset)
```

  - Parameters: desired start of mapping, length, protection data, management data, file descriptor and file offset

mmap() parameters

- prot: PROT_READ, PROT_WRITE, PROT_EXEC bits
  - depending on system: *write* is usually *read* or write protected files can not be written when PROT_READ is missing
- flags: refine mapping:
  - MAP_PRIVATE: don't write changes into file
  - MAP_SHARED: changes visible in file and other processes
  - MAP_FIXED: get this address or fail
  - MAP_ANONYMOUS: don't open a file (some systems expand heap using this trick)

munmap() & msync() & madvise()

- munmap(): removes mapped space starting from an addressto given address (may remove several); can handle unmapped segments.
- msync(): write mapping to file from given point
- madvise(): suggests how you want to use an address region: for random access, sequental access; will we need it all eventually or is the contents becoming irrelevant and when anything happens to it, the client won't leave the room in

Overview

- OSE
- OSEK/VDX
- Nucleus
- VRTX
- VxWorks
- QNX
- µC/OS

OSE by ENEA

- Operating System Embedded
- One of the most widely used RT operating systems
- 1.5 billion run-times deployed
- Swedish company
- Characterized by event messages for interprocess communication
- Wide support for processors
- Over half of 3G base stations

OSEK/VDX

- Automotive standard with many implementations (consider POSIX)
- *Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen*
- “Open Systems and their Interfaces for the Electronics in Motor Vehicles”
- Runs without memory protection
- Open standard

OSEK/VDX Specifies

- Static tasks, stacks, mutexes
- Tasks
  - Basic: never block
  - Enhanced: can wait on objects
- Priorities, round robin for same level
- Deadlock prevention with Priority Ceiling
  - Task which holds a priority object will prevent lower priority tasks from running

Nucleus

- 2.1 billion run-times deployed (according to developer)
- Closed source
- Windowing system
- Products
  - Motorola, Samsung, LG cellular phones
  - Creative Zen soundcards
  - Many iPhone clones

VRTX

- Versatile Real-Time Executive
- Two kernels: microcontroller and scalable
- Runs Hubble Space Telescope
- Also used by Motorola

VxWorks

- Started as improvement on VTRX (1985)
- Kernel was replaced later
- Name is probably a pun on VTRX
- Development done on host such as Linux, Unix, Windows with cross-compiling to ease testing
- Honda ASIMO; Boeing 787, 747-8; Linksys WRT54G, Canon DIGIC-II & DIGIC-III; Apache Longbow attack helicopter; Spirit&Opportunity rovers

QNX

- Microkernel architecture
- cars, mobile phones
- Owned by BlackBerry
- Amount of (optional) "services"
- proc – system service for task management
- message passing passes CPU time

µC/OS-II & -III

- Started as a two part article in Embedded Systems Programming magazine
- Free for noncommercial use
- $10000 per end product otherwise
- Also certified for use in safety critical contexts
- The Book!

How to Choose?

- Main characteristics:
  - Worst case performance documented
  - Interrupt latency known
  - Context switch time
- Exclude from the list:
  - **Processor**
  - **Real-time performance**
  - **Price**

Last optimization

- Compatibility of compiler
- Debugging issues
- Development tools
- Memory requirements
- Add-on software (network, usb, filesystems)
- Vendor experience

That's all, folks

- *Insert copyrighted image here*