Operating Systems Concepts: Revision Lecture

Olav Beckmann
Room 449 Huxley

Some General Comments
- Make sure you have all the lecture handouts, especially make sure you look at tutorial exercises and notes on solutions - check web page
- Revision Guide - check web page

Some Useful Facts
- Kilo, Mega, Giga:
  1K = 2^10 = 1024
  1M = 2^20 = 1024 * 1024 = ...
  1G = 2^30 = 1024 * 1024 * 1024 = ...

- Storage layout of arrays in memory
  ```
  int x[5];
  ```

- Storage layout of structures
  ```
  typedef struct { int y, int z[2] } mydata;
  ```

- Address 0  y  z[0]  z[1]

### NARC - Not A Real Computer

**Processor**

**Memory**

- Processor operates on data items - "words" stored in memory
- Each word consists of 32 binary (i.e. 0 or 1) digits - "bits"
- A word may represent a number (or characters or anything...)
- A word may represent an instruction for the processor
- Processor is a machine which interprets ("executes") instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>operand(D)</td>
</tr>
</tbody>
</table>

- Maximum addressable memory ?

### The NARC Instruction Set

<table>
<thead>
<tr>
<th>Instruction opcode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOADC</td>
<td>Acc := D</td>
</tr>
<tr>
<td>LOADM</td>
<td>Acc := Memory(D)</td>
</tr>
<tr>
<td>STOREM</td>
<td>Memory(D) := Acc</td>
</tr>
<tr>
<td>ADDC</td>
<td>Acc := Acc + D</td>
</tr>
<tr>
<td>ADDM</td>
<td>Acc := Acc + Memory(D)</td>
</tr>
<tr>
<td>SUBC</td>
<td>Acc := Acc - D</td>
</tr>
<tr>
<td>SUBM</td>
<td>Acc := Acc - Memory(D)</td>
</tr>
<tr>
<td>JMP</td>
<td>PC := D</td>
</tr>
<tr>
<td>JMPZ</td>
<td>If Acc=0 then PC := D</td>
</tr>
<tr>
<td>JMPN</td>
<td>If Acc&lt;0 then PC := D</td>
</tr>
<tr>
<td>CALL</td>
<td>Lnk := PC; PC := D</td>
</tr>
<tr>
<td>RET</td>
<td>PC := Lnk</td>
</tr>
<tr>
<td>HALT</td>
<td></td>
</tr>
</tbody>
</table>

### What exactly does the processor do?

```c
int Mem[MAX]; // main memory
int Acc; // accumulator
int Lnk; // link register
int PC; // program counter
int Op; // current opcode
int D; // current operand

void fetch() {
    int W;
    W = Mem[PC];
    Op = Opcode(W); // most significant 8 bits of W
    D = Operand(W); // least significant 24 bits of W
}
```

This piece of C code describes what the processor does to fetch an instruction

### Execute

```c
void execute() {
    switch(op) { // (opcode)
        case 1: Acc = D; // loadc
        case 2: Acc = Mem[D]; // loadm
        case 3: Mem(D) := Acc; // storem
        case 4: Acc = Acc+D; // addc
        case 5: Acc = Acc+Mem[D]; // addm
        case 6: Acc = Acc-D; // subc
        case 7: Acc = Acc-Mem[D]; // subm
        case 8: PC := D; // jmp
        case 9: if (Acc=0) PC := D; // jmpz
        case 10: if (Acc<0) PC := D; // jmpn
        case 11: Lnk := PC; PC := D; // call
        case 12: PC := Lnk // ret
    }
}
```

This C function describes what the processor does to execute an instruction
The “Fetch-Execute Cycle”

PC = 0;
do {
    fetch();
    PC = PC + 1;
    execute();
} forever;

- Execution starts at location zero (when NARC boots)
- Instructions are stored in memory
- So are the computation’s working variables
- Fetch and execute are realised as digital circuits
  (see next slide)

The “Fetch-Execute Cycle”

NARC - Control

- Each instruction in the NARC instruction set is implemented by a sequence of steps in which data is moved around the data path
- Eg. addm D:
  1: AddressIn=PC; Read
  2: Op&D = DataOut; PC=PC+1
  3: Switch(Op) {
      4: AddressIn = D;
      5: Acc = Acc + DataOut
  }
- These steps are sometimes called “microinstructions”

NARC - Control

An alternative to polling: Interrupts

- While we check the spelling of the preceding word, we want the keystrokes to be stashed in a buffer
- “No matter how fast you type!”
- There is a limit…
  – incredibly brief keystrokes might not be noticed
  – two keystrokes in very quick succession...
  – if the buffer is filled faster than it is emptied, it will fill up
- I suppose we could build a special electronic box to fix it… or….

Interrupting the NARC processor

- When a key is pressed, keyboard requests the processor to interrupt what it’s doing and collect the keystroke

Modifying the fetch-execute cycle

PC = 0;
do {
    fetch();
    PC = PC + 1;
    execute();
} forever;

PC = 0;
do {
    fetch();
    PC = PC + 1;
    execute();
    if (InterruptRequest) {
        save(PC);
        PC = ipc;
    }
} forever;

Not finished yet… why?
Enabling and disabling interrupts

PC = 0;
do {
    fetch();
    PC=PC+1;
    execute();
    if (InterruptRequest && InterruptEnabled) {
        InterruptEnabled = false;
        save(PC);
        PC = ipc;
    }
} forever;

Saving the PC during an interrupt, and restoring it on return

PC = 0;
do {
    fetch();
    PC=PC+1;
    execute();
    if (InterruptRequest && InterruptEnabled) {
        InterruptEnabled = false;
        Mem[0] = PC;
        Mem[1] = PC;
    }
} forever;

Device driver structure - top-half, bottom-half

- Top half of device driver is called by client process. It initiates device operation.
- The two halves interact via a queue.
- With an input device, the bottom half adds items to the queue, the top half removes them.
- With an output device, the top half adds items to be output, the bottom half removes them.

Bottom half is initiated by interrupt signalling device completion.
Bottom half reinitiates device operation if there is more work to do.
The top half can allow other processes to run while it is waiting.

Device driver structure

char getchar() {
    while (KbdBuffer.isEmpty())
        wait; // maybe let another process run
    return KbdBuffer.remove();
}

void InterruptHandler() {
    saveProcessorState();
    char ch = *KBD_PORT_ADDRESS;
    KbdBuffer.add(scanToAscii(ch));
    restoreProcessorState();
    InterruptEnabled = true;
    PC=Mem[0];
}

The Interrupt Handler

void InterruptHandler() {
    saveProcessorState();
    char ch = *KBD_PORT_ADDRESS;
    KbdBuffer.add(scanToAscii(ch));
    restoreProcessorState();
    rti; // restore PC from Mem[0] and re-enable interrupts
}
An interrupt can occur after the execution of any instruction.

```c
PC = 0;
do{
fresh();
if (InterruptRequest && InterruptEnabled) {
    InterruptEnabled = false;
    Mem[0] = PC;
    PC = Mem[1];
}
} forever;
```

Recall...

Saving the PC during an interrupt, and restoring it on return.

```c
void InterruptHandler() {
    char ch = *KBD_PORT_ADDRESS;
    KbdBuffer.add(scanToAscii(ch));
%
```
Using Semaphores: Mutual Exclusion

```
var d: int //shared variable
var s: semaphore
InitSema(s,1) // initialise to 1

process p(n)
  // s = 1
  P(s) // s = 0
  d := d + 1
  V(s)
end
```

Process can only enter critical section if s == 1. Only one process at a time can be executing its critical section – so get mutual exclusion.

Using Semaphores: Synchronisation

```
Process A:
  ...........
  ...........
L1: V(proceed)
  ...........
  ...........

A

Event

B

Process B:
  ...........
  ...........
L2: P(proceed)
  ...........
  ...........

- Process B must wait at L2 – until Process A reaches L1 and signals that B can proceed by executing V(proceed).
- What value must the semaphore proceed be initialised to?
```

Using Semaphores: Communication

```
Producer

Buffer (N Items)

Consumer

Three semaphores for three “resources”:
• Space in buffer is resource needed by Producer
  – allow deposit only when buffer not full (items in buffer < N)
• Item in buffer is resource needed by Consumer
  – allow fetch only when buffer not empty (items in buffer > 0)
• Mutual exclusion for buffer access is resource needed by everyone
  – allow buffer access only when no one else accessing it
```

Semaphore Solution

```
var mutex: semaphore // initialise to 1
var space: semaphore // initialise to N
var item: semaphore // initialise to 0

process Producer
  loop
    produce item
    P(space) // “I want space”
    P(mutex) // “I want mutual exclusion”
    deposit item
    V(mutex) // “Here is mutual exclusion”
    V(item) // “Here is item”
  end loop
end

process Consumer
  loop
    fetch item
    P(item) // “I want item”
    P(mutex) // “I want mutual exclusion”
    consume item
    V(mutex) // “Here is mutual exclusion”
    V(space) // “Here is space”
  end loop
end
```

Two Pass Assembler

```
A

B

C

label

main

loadm A

jmpz end

loadm C

addm B

storem C

loadm A

subc 1

storem A

jmp main

symbol table

<table>
<thead>
<tr>
<th>label</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>main</td>
<td>5</td>
</tr>
<tr>
<td>loadm A</td>
<td>6</td>
</tr>
<tr>
<td>loadm C</td>
<td>7</td>
</tr>
<tr>
<td>addm B</td>
<td>8</td>
</tr>
<tr>
<td>storem C</td>
<td>9</td>
</tr>
<tr>
<td>loadm A</td>
<td>10</td>
</tr>
<tr>
<td>loadm A</td>
<td>11</td>
</tr>
<tr>
<td>loadm A</td>
<td>12</td>
</tr>
</tbody>
</table>
```

The Job of the Linker

```
- Suppose we want our new program to use the functionality provided by another program
- E.g. newly-purchased word processor needs to access software driver for new keyboard
- Simpler working example: ‘main’ program uses separately-provided ‘incr’ procedure (next slide)

Two issues:
• relocation: concatenate the binary code - and adjust symbol references according to new addresses
• name binding: resolve names used but undefined in ‘main’ with names defined in ‘incr’
```
The Linker

External Symbol Table

incr
Loc Value
0 addc 1
1 ret
2 0
4 loadc 23
5 call 0
6 storem 2
7 binc 25
8 call 0
9 storem 2
10 ret

External Symbol Table

main
Loc Value
0 addc 1
1 ret
2 0
4 loadc 23
5 call 0
6 storem 2
7 binc 25
8 call 0
9 storem 2
10 ret

Suppose a program has been assembled starting at address 0. Suppose it is now loaded into store starting at address 100.

Need to relocate absolute address operands

Table 1: Concatenate

0.1
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0

Table 2: Relocate

0.1
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0

Table 3: Resolve

0.1
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0

Note: NARC uses only absolute addressing. Relative addresses (e.g. PC relative jumps) do not have to be relocated.

Which locations will need to be adjusted?
Object code file must record this information.

Table 1: Object Code File

Loc Value Relocate?
A .word 23
B .word 45
C .word 0
main loadm A
jmpz end
loadm C
addm B
storem C
loadm A
subc 1
storem A
end

Table 2: External Symbol Table

Loc Value Rel?
main T 2
incr U

The symbol table entry for an undefined label points to a linked list of entries which use that label

Note two stages of relocation:
- at link time and load time

Linker

memory

Relocating loader
Linking, Loading and Relocation - Summary

- Assembler translates human-readable representation of instructions into object file:
  - object file includes list of external names defined, and names used but not defined
  - object file also includes relocation information
- Linker combines object files:
  - concatenate, relocate so internal name references OK
  - resolve name binding
- Loader finds sufficiently-large free memory region, interprets object file format, loads object file instructions and initialises data into memory
  - relocates so that internal name references are right

Enabling and Disabling Interrupts on Intel i386

Flags Register (Status Word)

<table>
<thead>
<tr>
<th>IF</th>
<th>ZF</th>
<th>CF</th>
</tr>
</thead>
</table>

- Interrupt Flag: when set (equal to 1) interrupts are enabled otherwise they are disabled.
- The only way to change the IF flag is through the following instructions:
  - cli - clear IF (disable interrupts)
  - sti - set IF (enable interrupts)
- The eflags register controls whether interrupts are enabled.
- For reasons of instruction set design, the eflags register cannot be directly addressed.

Kernel Exit & Entry (cont.)

For convenience we will use the following macros:

```
#define ENTERKERNEL
int psd; psd = int_mutexon();
```

```
#define EXITKERNEL
int_mutexoff(psd);
```

See include/icos/intP.h in Simple Kernel source code.

The assembler routines int_mutexon and int_mutexoff (see previous slide) can be found in int_asm.S.
Semaphore Implementation

```c
typedef struct Semaphore {
    int count;
    Queue waiting;
} Semaphore;
```

```c
void P(Semaphore *s) {
    ENTERKERNEL
    if (s->count > 0)
        s->count--;
    else
        addTail(s->waiting, running);
    dispatch();
    EXITKERNEL
}
```

```c
void V(Semaphore *s) {
    ENTERKERNEL
    if (!isEmpty(s->waiting))
        s->count++;
    else
        setReady(getHead(s->waiting));
        setReady(running);
        dispatch();
    EXITKERNEL
}
```

```c
void initSema(Semaphore *s, int value) {
    ENTERKERNEL
    s->count = value;
    initQ(s->waiting);
    EXITKERNEL
}
```

Key Datastructure in Kernel: FIFO Queue

```
typedef struct Queue {
    Process head, tail;
} Queue;
```

Advantages of this data structure?

- Add processes at tail of queue
- Remove processes from head

Ready Queue - Static multi-level priority

```
#define HIGH_PRIORITY 3
#define NORMAL_PRIORITY 2
#define LOW_PRIORITY 1
#define IDLE_PRIORITY 0
```

ReadyQ[4]: /one queue for each priority level

Process running: //points to currently running process

Scheduling Operations

```
void setReady(Process p) {
    addTail(readyQ[p->prior], p);
}
```

```
void dispatch() {
    Process oldrunning = running;
    int pr = 3;
    while (!isEmpty(readyQ[pr])) {
        pr--;
    }
    running = getHead(readyQ[pr]);
    psswitch(oldrunning->savearea, running->savearea);
}
```

```
void initSched() {
    for (int pr=0; pr<4; pr++)
        InitQ(readyQ[pr]);
    create(null, 0, idle);
    running = getHead(readyQ[idle]);
}
```

Process State Transitions

- Delayed and suspended are both blocked states.

- A process will be in exactly one of these states. The kernel maintains data structures to keep track of the processes in each state:
  - A ready process - is on one of the four ready queues, readyQ[priority] (see below)
  - A delayed process - is on delayQ
  - A suspended process - is on the queue associated with a semaphore S
  - The running process - is pointed to by the variable running
    (Just one processor, so only one process at a time in the running state.)
UNIX: Dynamic Priority Scheduling

- UNIX uses two-level scheduling (see lecture notes Ch 6).
- The UNIX low-level scheduling algorithm is a dynamic priority scheduling algorithm.
- Priority is an integer in the range [-20, 20], with lower number meaning higher priority. Kernel mode processes negative, user mode processes positive.
- Nice ("niceness") is also integer in range [-20, 20]; added to priority. Users can choose positive niceness – lower priority, only superuser can choose negative niceness.
- Assume there are 40 priority queues; first process from highest queue gets to run.
- Quantum typically 100ms. After process uses up its quantum, rejoins its own priority queue -> round-robin for equal priorities.
- CPU usee = \((\text{quanta used in last second}) \times \frac{\text{CPU usage}}{2}\)

In real life: Linux Scheduling

- Linux scheduling is different from UNIX. Three classes:
  - "real-time" FIFO
  - "real-time" round-robin
  - Timesharing
- Users can alter (lower) priority by altering (raising) niceness.
- Quantum ("dynamic priority") starts at value of static priority, 20
  - Quantum of running process is decremented by 1 at clock tick.
  - Timer interrupt gives a clock tick every 10ms – "jiffy".
- Goodness = quantum + priority if quantum > 0, else 0.
- When the scheduler is called, highest goodness is selected to run.

Memory Management

- Dynamic partition store allocation: each job is allocated a contiguous region of memory
  - Advantage over fixed partition: avoids wasted space
  - Disadvantage: fragmentation – several problems
  - Cannot run jobs larger than available memory
- Dynamic partition store management with compaction means that we need virtual addressing
- Placement algorithms (first fit, best fit etc) are about where to allocate a job, given a number of memory regions that are large enough.

Basic Paging Scheme

- 32-bit address
- Assume 4k page -> 12-bit page offset
- 20-bit page number
- Each page table entry consists of 20-bit phys page no. plus status
- There are \(2^{20}\) page table entries
- Page table size is at least \(\frac{32}{20}\)

Basic Paging Scheme - Example

- The virtual page number is the index into the page table.
- Page table entry is example only – could be any value.
Paging with TLB

• TLB for 4k pages (data) has 64 entries.

• TLB index is the same as (virtual page no modulo TLB size).

Paging with TLB: Example

Address 12291 (decimal)

Index 011 (binary) gives line number 3

Tag indicates that correct page table entry is held in TLB

TLB entry is correct

Summary

• Please work hard…

• When the exam starts –
  – Smile…
  – Think about the weighting of question parts
  – Manage your time!
  – It is much easier to get the first 30-40% of a question than to get the last 20% - therefore, make very sure you do not miss out on relatively easy-to-get marks by leaving too little time for one of the questions.
  – Take a look at the parts of the question you are answering – it should be obvious which part(s) are the easy parts.
  – Bring a watch, and allocate (at least initially) equal time to each question – they all carry equal weight!

• When answering questions – think about the weighting: it is an indication of how much we expect you to say on a question.
  • Questions are marked out of 20 (per question)
  • I.e. if a question-part counts 10%, that means 2 marks