1 Process Context

1.1 What is context?

A process is sometimes called a task, subroutine or program. Process context is all the information that the process needs to keep track of its state.

Registers Temporary storage locations for a few bytes. Typical CPU has 8-64 registers.

Program Counter A special register which points to the next instruction to be executed.

Stack Pointer Another special register which points to the top (bottom) of a stack of data

Processor Flags Hardware bits in the CPU which detect errors or arithmetic results.

Sometimes also called “processor state”.

Example: reading a book and the phone rings.

1.2 Context Switching

When we call a subroutine (function) or when an interrupt happens, we need to save the current context because we are starting a new one.

This happens a lot so it must be efficient.

- After function completes, must know where to resume.
- New process can mess up information in old process. (i.e. what if they both use the same register?)
- Saving the context and restoring it is called context switching.

Procedure:

1. Control is passed from Proc1 to OSH
2. OSH saves processor state from CPU registers (immediately!)
3. OSH finds stored state of Proc2 and restores it into CPU registers. IR is restored last.
4. CPU jumps to Proc1 code according to IR.

OSH = Operating System or Hardware.

1.3 Types of Context Switches

- Subroutine Call. Either to user routine or to OS.
- Interrupt. Hardware initiated context switch.

1.4 Calls

- User subroutine (e.g. \( \text{sin}(\theta) \)). Main program context is saved, sin routine is loaded.
- System Calls Examples
  - I/O such as write block of data to disk or apply a value to D/A converter.
  - Process Control: halt, wait (pend), start, kill, or unblock another process.
  - Interprocess Communication: pipes, mailboxes, semaphores.
  - Return from Interrupt.
1.5 The Stack

Problem: How to allocate space for context storage. Consider nesting of subroutine calls. Solution: A stack is a data structure modeled on a stack of plates(!). (spring balanced).

- **Push:** a piece of information on the stack.
- **Pop:** a piece of information off of the stack.
- Always put/take info from current stack top.

Stacks are implemented with a **Stack Pointer**. Enter the proper C code on the blank lines according to class discussion.

- **Push:** \_\_\_ = piece_of_information
- **Pop:** piece_of_information = \_\_\_
- **Initialize Stack:** SP = stack_top
- Stack grows down.
- Have to allocate space for maximum stack depth.

1.6 Subroutine Call

How to call a subroutine (function in C). (the compiler generates instructions to do all this so you don’t have to!)

1. Push i argument *values* to stack:
   
   ```c
   foreach i \_\_\_ = argument(i);
   ```

2. Generate JSR machine instruction.
   - (a) Push machine registers onto stack.
   - (b) Load subroutine addr into PC
   - (c) execute next instruction (i.e. jump to *PC).

3. Subroutine looks for arguments at *(SP - MACH_REG_SIZE)*

4. When Subr. completes, execute **RET** instruction.
   - (a) Pop machine registers from stack.
   - (b) Clean up stack from function args: `foreach i SP++;
   - (c) Load old value into PC
   - (d) execute next instruction (i.e. jump to *PC).

2 Interrupts

A context switch caused by an **external hardware event**.

Types of inputs that can cause an interrupt:

- User pushes a button or moves a mouse.
- Packet arrives on network interface.
- A sensor detects an event.
- Disk drive completes an operation and has a bunch of data ready for CPU.
- A clock can be programmed to set up interrupts at regular intervals.
- A hardware timer can count clock pulses and generate an interrupt when it gets to zero.
- A thermostat indicates that a temperature setpoint is reached.
Connections  Interrupts are logic signals physically wired to the CPU (or interrupt processor peripheral). There are various schemes for wiring multiple interrupts:

Priority
- 7 lines leading into CPU.
- peripherals can pull one line low for example using open collector “wired OR”.
- A small address bus identifies the specific interrupt within a line.
- Lines are ranked by priority e.g. ISR for Line 0 cannot be interrupted by any other interrupt, ISR for Line 6 can be interrupted by ANY other interrupt.

Interrupt Vectors  Each interrupt has a specific code number which is applied to the processor interrupt address bits (maybe 8bits).

CPU has a specific hard-wired memory address called the interrupt vector table. The interrupt vector table is an array of function pointers. The functions they point to are called interrupt service routines, ISRs.

<table>
<thead>
<tr>
<th>Memory Addr</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>isr_0()</td>
</tr>
<tr>
<td>1002</td>
<td>isr_1()</td>
</tr>
<tr>
<td>1004</td>
<td>isr_2()</td>
</tr>
<tr>
<td>1006</td>
<td>isr_3()</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1100</td>
<td>isr_{N}()</td>
</tr>
</tbody>
</table>

Typical Interrupt Vector Table

Meaning of ISRs  Each hardware device is hooked up to a specific interrupt vector table address. We need to write each ISR to handle the proper device.

2.1 ISRs

Servicing an interrupt

Hardware Level  The hardware does the following steps when an interrupt is detected:

1. Push processor context onto stack.
2. Load appropriate address from IVT into program counter (PC).
3. jump to *PC.

Software Level  The ISR is now in control. The ISR must do the following.

1. (There are no arguments)
2. block (“mask”) all other interrupts if necessary
3. “service” the device.
4. unmask other interrupts
5. Execute a return from interrupt instruction
Hardware Level  Now the hardware takes over again to clean up the interrupt.

1. Pop processor context from the stack.
2. Last item is PC
3. continue whatever was being done when interrupt occurred.

2.2 ISR Programming
Some tips for successful ISR programming

Problems with ISRs

- ISRs are virtually impossible to debug!
- The debugger uses “software interrupts” to handle tricks like breakpoints and single stepping.
- thus even correct ISRs will break the debugger OR
- the debugger will break your ISRs
- ISRs take over the processor – even pre-emptive schedulers.
- ISRs can easily mess up other processes if they
  - write in wrong global variables
  - take up a lot of CPU time
  - play around with the stack

ISR Solutions

- Keep your ISRs small, simple, & quick
- No loops inside ISR’s
- No complex logic (no nested if’s )
- Don’t do unrelated I/O in the ISR (for example, no \texttt{printf(“ISR message”);}).
- Guidelines: Max: 1/2 page of C, 1 page of ASM
- Do not trust / use debugger.
- Do absolute minimum of stuff in ISR, do the rest in a regular task.
- Check and recheck your use of interrupt masking and how you save and restore the context.

3 Real Time Control

3.1 Basic Task
Map inputs to outputs (through a control law) at regular time intervals.
\% Initialization
set_up(timer0);
set_up(input,output);

\% endless control loop
while(1)
{
    wait(timer0);
    x=read(input);
    y=control_function(x);
    output(y,output);
}

3.2 Timing
Different ways to ensure regular time sampling.

- While loop + Interrupt
  - “main” program is while(1);
  - Timer is set up for regular interrupts every T seconds.
  - ISR is I/O and control law:
    x=read(input);
    y=control_function(x);
    output(y,output);
    return();

- “spin-lock”:
  wait(timer0) \rightarrow while(timer0 == 0);

- OS Call:
  wait(event)
  return control to OS until \texttt{event} occurs. \texttt{event} is timer0.