9 Interrupt Processing

- Interrupt processing takes place in layer 5 of the layered design of XINU:
  0. chapter 2: **run-time environment** and **hardware**
  1. chapter 8: **memory management** (`getstk`, `freestk`, ...)
  2. chapter 3, 4, 5: **list manipulation** plus **context switching** (`resched`, `ctxsw`, `ready`, process table) and **process management** (`suspend`, `resume`, `kill`, `create`)
  3. chapter 6: process coordination by **semaphores** (`wait`, `signal`, `screate`, `sdelete`)
  4. chapter 7: **asynchronous message passing** (`send`, `receive`)
  5. chapter 9: **interrupt processing** (`intmap` table plus interrupt dispatchers)
- Without an **interrupt mechanism** the operating system cannot guarantee that it will ever **regain control** once user program is started!

9.1 Dispatching Interrupts

- The 11/2 processor uses *vectored interrupts* to handle exceptions and interrupts from external devices.
  - Whenever an external device needs to communicate with the processor, it places a signal on the **interrupt bus line**
  - If the processor is running with **interrupts enabled**, it checks this line **after every time an instruction has been executed** to see whether an interrupt needs processing.
  - To handle the interrupt, the processor sends an acknowledgement over the interrupt bus line.
  - The **first external device** with an pending request receives the acknowledgement and returns its interrupt vector address v along the bus.
  - When the processor receives v, the processor pushes the current PC and PS on the stack, loads the PC and PS from two words in memory, starts at location v and continues executing instructions at the new location.

Dispatching Interrupts

- Each external device is assigned a **unique interrupt vector address** v before it is inserted into the bus, thus enabling the system to distinguish among them. I.e., every device is **identified** by its unique interrupt vector address v.
- The interrupt vector locations contain a valid PC and PS. This PC points to an **interrupt handling routine** (interrupt handler) for the interrupting device.
- The processor execute the code of the **interrupt routine** before **returning** to the place where the user's program has interrupted.
- To make the interrupt **transparent** (i.e., invisible) to the running program, the interrupt routine saves and restores the **stack** of the machine – the values of R_0 to R_6 (since hardware saves PC and PS).

Dispatching Interrupts

- In practice SP ($= R_6$) **needs no saving** – provided the interrupt routine pops off the stack whenever it pushed the stack before returning (i.e., the stack is restored to its original position).
- To prevent the interrupt routine from **being interrupted itself** by another device, the PS stored in the interrupt vector specifies a priority level of 7, i.e. `034i`. Thus interrupt processing begins with interrupts disabled.
- Interrupt processing ends by executing a **return from interrupt** instruction: `rti` or `rtt`, restoring – in a **single step** – the old PC and PS from the stack and returning the stack pointer SP to its original value.

Handling Exceptions

- **Exceptions** are handled similarly, except that they can **never be turned off** (i.e., disabled).
- Examples of exceptions occur when **nonexistent memory** is addressed, upon an **array-bound** error, execution of **invalid code** or **power failure**.

Dispatching Interrupts

- There is no procedure call instruction `jsr` in the interrupt program for jumping to the interrupt handler. The processor **simulates** a call to the appropriate interrupt routine **in between** the execution of instructions of the user's program.
- Comer uses a two-level strategy for processing interrupts:

  1st level: Interrupt brach to **low-level interrupt dispatch routines** written in **assembly language**. They save and restore register info, identify the interrupting device, ahndle return from interrupts and call the second level:

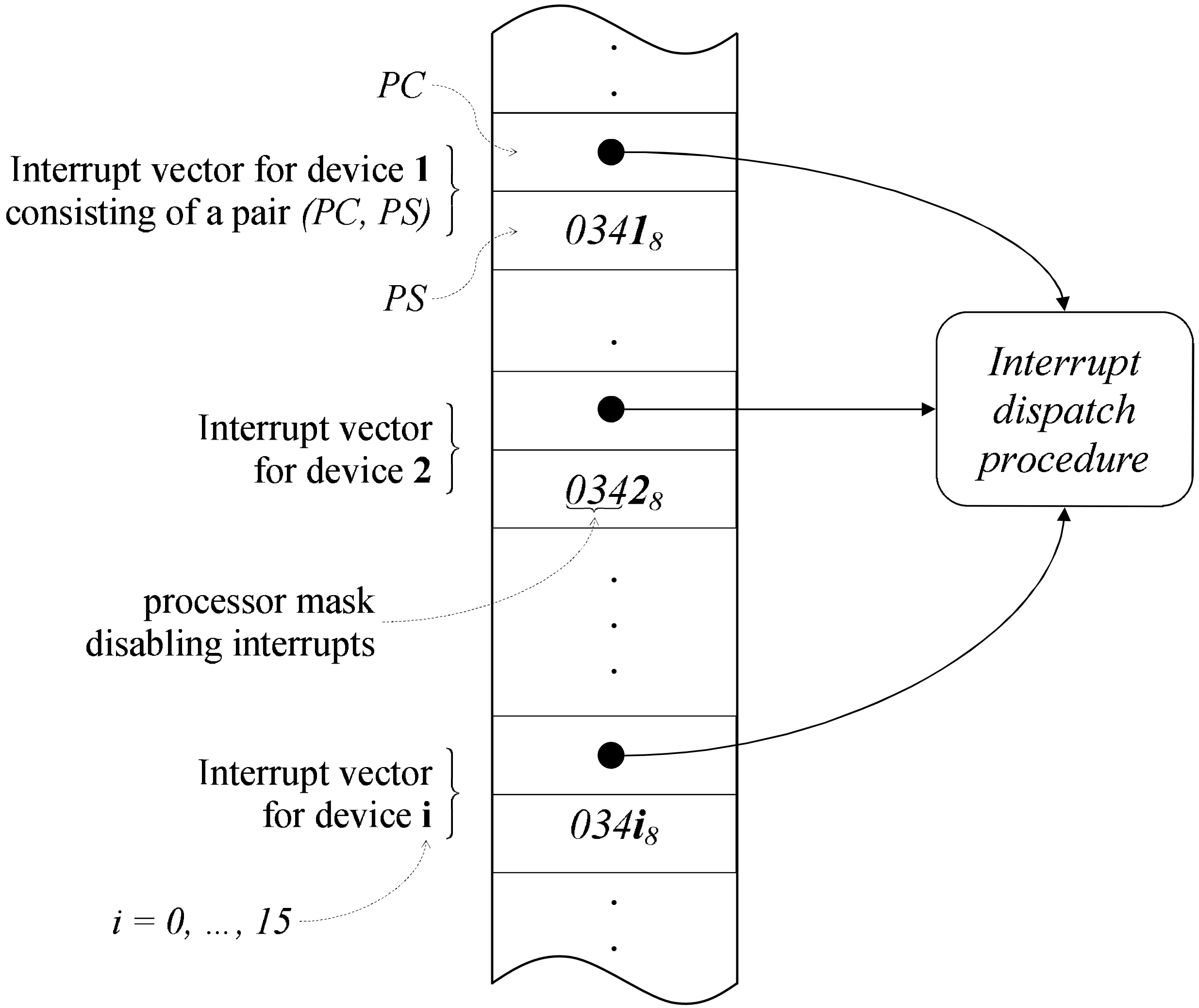
  2nd level: **High-level routines** (for r/t, i/o processing, disc processing) **do the real work**, using the info passed from level 1.

- Xinu contains three different interrupt dispatchers:
  - for handling **clock interrupts** – chapter 10
  - for handling **input interrupts** – this chapter
  - for handling **output interrupts** – this chapter

9.2 Input and Output Interrupt Dispatchers

- Devices **connect to computer system** through hardware mechanisms called **controllers**.
- **Controllers** reside on boards plugged into the system bus; their hardware converts digital data to wave forms and signals controlling peripheral devices s.a. terminals and disks
- Xinu allows both input and output operations to the **console** device with output to console **displayed on screen** and input from console **read from keyboard**.
- Input interrupts for characters typed on a terminal's keyboard trap on a **different vector** from **output interrupts** (for characters sent to the display)

Vectored Interrupt Table



Input and Output Interrupt Dispatchers

- **When an interrupt occurs**, the old PC and PS are **put on the stack,** new PC and PS are **loaded from interrupt vector**; while the new PC points to interrupt dispatch routine.
- The **interrupt dispatch routine** saves a copy of PS (because condition codes change lower-order bits of register) and masks off low-order 4 bits to compute an **index into a global interrupt dispatch table** which contains the **addresses of the high-level interrupt handlers** for each connected device (see the code for `intmap` later).
- I/O interrupt routines defined in `intmap` are `getchar()`, `putchar(ch)`, `fgetchar()`, and `fputchar(ch)`. They receive an argument, **identifying the interrupting device**. I/O interrupt dispatchers are combined in file `ioint.s`.

Input and Output Interrupt Dispatchers

Stack image after calling `jsr pc,*(r0)` in `ioinit`-routine:

	Stack	
	...	
	old PC	stacked by hardware
	old PS	
	new PS	$\longleftarrow$ to identify device number
	old R_0	since these are not saved by `csv`-routine and the high-level interrupt routines obey normal procedure call conventions
	old R_1	
	`icode/ocode`	$\longleftarrow$ parameter for interrupt routine
$SP \longrightarrow$	return address	$\longleftarrow$ return address of `move 2(sp),r1`

Input and Output Interrupt Dispatchers

After returning from interrupt-routine:

$SP + 12$	$\longrightarrow$	$\ldots$
		old PC
$SP + 8$	$\longrightarrow$	old PS
		new PS
$SP + 4$	$\longrightarrow$	old R_0
$SP + 2$	$\longrightarrow$	old R_1
SP	$\longrightarrow$	`icode`/`ocode`
		return address

The call to `rtt` is popping old PS and PC from stack, adding 4 to SP, returning to address pointed at by old PC.

Input and Output Interrupt Dispatchers

```
/* io.h -  fgetc, fputc, getchar, isbaddev, putchar */


#define INTVECI inint           /* input interrupt dispatch routine     */
#define INTVECO outint          /* output interrupt dispatch routine    */
extern  int     INTVECI();
extern  int     INTVECO();

struct  intmap  {               /* device-to-interrupt routine mapping  */
        int     (*iin)();       /* address of input interrupt routine   */
        int     icode;          /* argument passed to input routine     */
        int     (*iout)();      /* address of output interrupt routine  */
        int     ocode;          /* argument passed to output routine    */
        };

#ifdef  NDEVS
extern  struct  intmap intmap[NDEVS];
#define isbaddev(f)     ( (f)<0 || (f)>=NDEVS )
#endif
#define BADDEV          -1

/* In-line I/O procedures */

#define getchar()       getc(CONSOLE)
#define putchar(ch)     putc(CONSOLE,(ch))
#define fgetc(unit)     getc((unit))
#define fputc(unit,ch)  putc((unit),(ch))

struct  vector  {
        char    *vproc;         /* address of interrupt procedure       */
        int     vps;            /* saved process status word            */
        };
```

Input and Output Interrupt Dispatchers

```
/* ioint.s - inint, outint */

/* I/O interrupts trap here.  Original PC and PS are on top of the      */
/* stack upon entry.  Low order 4 bits of the current PS contain the    */
/* device descriptor.  Interrupts are disabled.                         */

        .globl  _inint,_outint,_intmap
_outint:                                / Output interrupt entry point
        mfps    -(sp)                   / Save device descriptor from PS
        mov     r0,-(sp)                / Save r0 (csv does not)
        mov     $_intmap+4,r0           / point r0 to output in intmap
        br      ioint                   / Go do common part of code
_inint:                                 / Input interrupt entry point
        mfps    -(sp)                   / Save device code from PS
        mov     r0,-(sp)                / Save r0 (csv does not)
        mov     $_intmap,r0             / point r0 to input in intmap
ioint:                                  / Code common to input & output
        mov     r1,-(sp)                / Save r1 (csv does not)
        mov     4(sp),r1                / Get saved PS in r1
        bic     $177760,r1              / Mask off device descriptor
        ash     $3,r1                   / pick correct entry in intmap
        add     r1,r0                   / Form pointer to intmap entry
        mov     2(r0),-(sp)             / Push "code" from intmap as arg
        jsr     pc,*(r0)                / Call interrupt routine
        mov     2(sp),r1                / Restore r1 and R0 from stack
        mov     4(sp),r0
        add     $8,sp                   / Pop arg, saved r0, r1, and PS
        rtt                             / Return from interrupt
```

9.3 The Rules for Interrupt Processing

1) Interrupt routines examine and modify global data structures such as I/O buffers, so **interference from over processes** must be prevented. This is essentially done in two ways:

- Disabling interrupts, with interrupts still disabled after interrupt routine returns; so only after `ioint` returns are interrupts enabled again.
- Disabling interrupts, but: **high level interrupt routines may enable interrupts** by calling `resched`.

In case CPU switches to a process with interrupts enabled (`send` called, starting the receiver waiting for a message, or `signal` making first process on waiting queue ready to run):

$\Longrightarrow$ Interrupt routines should **leave global data** in a **valid state** before calling `resched()`.

$\Longrightarrow$ **No process enables interrupts** unless it **previously disabled them**.

The Rules for Interrupt Processing

2) **Interrupts should not be disabled too long.** If they do, devices will fail to perform correctly, e.g., when processor does not accept character from an I/O device before the next one arrives, **data will be lost**.

$\Longrightarrow$ Interrupt routines must be designed to **enable further interrupts as quickly as possible**.

3) Interrupt code is executed by **whatever process is running when the interrupt occurs**. This is especially the `NULL` process. Recall that `resched()` requires that there's **at least one process ready to run**, so the `NULL` process must always be current or ready.

$\Longrightarrow$ Interrupt routines can **only call procedures** that **leave the executing process in the current or ready states**.

E.g., `signal` or `send` may be called in an interrupt routine, but not `wait` or `receive`!

9.4 Rescheduling while processing an Interrupt

This section explains **why no infinite stack-overflow can occur because of the combination of interrupt routines with `resched()` calls**:

- Assume a running process P is interrupted. Executing the dispatcher and high-level routine is done by process P, with its PS still **disabling interrupts**.
- Unless the high-level interrupt routines indirectly call `resched()` through which **another process**, Q, with possibly **enabled interrupts** is executed. However, $Q \neq P$ in case Q enables interrupts because P disables interrupts.
- Interrupting Q leads to **disabling interrupts** in Q, but, as a result of a call to `resched()`, may **enable interrupts** of a new process R. Important is that $R \neq Q$ and $R \neq P$.

... **and so on** ...

- So, **no more interrupts** can be in state of being executed than the **number of processes** executed in parallel on the ready queue and executing as current.
- For this to be sure, no procedure should **enable interrupts** unless it **previously disabled them** (when calling `resched()`).