Communication, Interrupts and Protection

David May: May 14, 2017

Communication and Input-Output

Computers need to communicate with each other, and with input and output devices. This involves:

- Encoding the data to be communicated
- Establishing a communication *protocol*

For communication between computers the most important point about the protocol is that it must provide for *handshaking*. The receiver must periodically confirm that it has taken the data and is able to accept more. This applies both at the level of transferring a few bits of data at a time and at the level of sending large blocks of data such as Ethernet packets. This enables the communication to be reliable even when the speed of operation of the two communicating computers varies.

A simple scheme is to use a set of wires for the data, a request wire (req) and an acknowledge wire (ack). The sequence is:

1. sender places data on data wires
2. sender sets $req = 1$
3. receiver observes $req = 1$ and takes data
4. receiver sets $ack = 1$
5. sender observes $ack = 1$ and sets $req = 0$
6. receiver observes $req = 0$ and sets $ack = 0$
7. sender observes $ack = 0$

The above is a *return-to-zero* protocol. It is possible to omit two of the steps using a *non-return-to-zero* protocol:

1. sender places data on data wires
2. sender changes req
3. receiver observes change of req and takes data
4. receiver changes ack
5. sender observes change of ack

Note that when communicating with an i-o device, the req and ack are not always needed.

It is important that the request signal does not overtake the data signals as a result of wire or logic delays. It is possible to eliminate this problem using a *delay insensitive* scheme.

For example, using two request signals (req_0 and req_1) and an acknowledge signal (ack) a 0 bit can be sent by changing req_0 and a 1 bit by changing req_1; in either case the receiver replies by changing ack.

This is a very simple example of an n-of-m code (it is 1-of-2); these involve the receiver waiting for n signals to change state before replying by changing ack, after which the sender changes n signals again. Other examples are 1-of-4 which can encode 4 symbols (2 bits) per transition or 3-of-6 which can encode 20 symbols per 3 transitions.

It is often important to minimise the number of signal wires and there are also a number of communication schemes that transfer data serially (one bit at a time) using a clock signal that accompanies the serial data signal. This clock signal is used to store a block of data (a packet) at the receiver; only when the whole block has been received is it handed over to the receiving processor. It is also possible to avoid the use of the separate clock signal by using an encoding that enables both clock and data to be recovered from the same serial stream.

Metstability

Usually, there is no simple relationship between the clocks of the two communicating devices; both the frequencies and the phases will differ. Incoming data is sampled by using it as the data source of a flip-flop; the local clock of the receiving processor is used as the clock for the flip-flop.

This arrangement gives rise to the possibility that the data will change state at the same time as the clock, giving rise to *metastability*. The flip-flop enters a state in which it is balanced but unstable (like a pencil standing on its point). After a time,

it will settle into a set state or a reset state. In designing input-output systems, it is important that sufficient time is allowed to ensure that the flip-flops have settled. Otherwise, the processor itself can enter a state in which it is simultaneously trying to execute the next instruction and handle an input-output request.

Interrupts

It is often important to respond rapidly to external input and output requests. A common way to support this is to enable execution of a program to be *interrupted*. An interrupt causes the processor to temporarily suspend execution of a program, transfer control to a short sequence of instructions (an interrupt routine) to perform the input or output operation, finally returning to execute the interrupted program.

It is common for there to be several input and output devices, each with an associated interrupt routine. The collection of these, along with other software for allocating resources such as memory form the operating system *Kernel*.

In order to be able to return to the interrupted program after executing an interrupt routine, the pc must be stored when control is transferred to the interrupt routine. It is also important that interrupts are disabled when the interrupt routine is entered. This can be done using two additional registers:

register	**use**
spc	the saved program counter
ink	the *executing in kernel* flag

When an interrupt request is received from a device, control is transferred to the corresponding interrupt routine; the address of this can read from a memory location associated with the device. This address is known as an *interrupt vector*.

Each device normally has some associated registers; these often appear to the programmer as a small set of consecutive memory locations. This is referred to as *memory mapped* input-output. Typically there will be a data register and a control register. The control register will contain a *ready* flag to indicate that the device is ready to transfer data. For input, *ready* will be set when the data register is full; for output, *ready* will be set when the data register is empty. The control register will also contain an *enable* flag which is set and cleared by the program. When *enable* is set and *ready* is set, an interrupt request is made to the processor and the processor will transfer control to the interrupt routine as soon as the ink flag is clear.

The vectors associated with interrupt requests are normally stored in low memory

locations, along with those associated with kernel calls.

In addition to entering the kernel as a result of an interrupt request, it is normal to provide a *kernel call* or *system call* instruction. A *kernel return* instruction is also needed to return control from the kernel to the application program.

$intreq$	$spc \leftarrow pc$; $pc \leftarrow mem[kint + n]$ $ink \leftarrow true$	kernel entry - input-output request from device n
KCALL	$spc \leftarrow pc$; $pc \leftarrow mem[kcall + areg]$ $ink \leftarrow true$	kernel entry from kernel call
KRET	$pc \leftarrow spc$ $ink \leftarrow false$	kernel exit

An interrupt can, in principle, occur between any two instructions. However, there are places where this would cause difficulties:

- during entry to the kernel as a result of a system call or an error, but before the spc (for example) has been saved
- during a kernel instruction sequence which is modifying a data structure (such as a buffer) used to communicate with the interrupting input-output device.

One way to avoid these problems is not to permit interrupts when executing the kernel. An interrupt request from an input-output device is accepted by the processor only when ink is $false$. This means that, for responsive input-output, the kernel procedures must be kept short; some of the longer procedures can be treated as application programs.

Direct Memory Access (DMA)

In order to speed up input and output of large amounts of data, it is common to provide hardware in the input-output device that can directly access memory. Instead of each item of data being transferred between the processor and the input-output device, the processor supplies information about a region of memory (typically this a a base address and a length). The device then carries out the transfer. The *ready* and *enable* flags can be used in the same way as for non-DMA devices to signal that the device is ready and to control interrupts.

Protection

Most computers include hardware to *contain* errors in application programs, preventing them giving rise to further errors in other programs or in the kernel itself, for example by overwriting critical data or instructions.

The instruction set architecture has to:

- protect the kernel from an error in an application program
- protect an application program from an error in another application program
- enable the kernel to remove failed application programs
- protect the external environment from errors in applications programs (by performing input and output via the kernel)
- enable the kernel to allocate resources such as memory and input-output to applications programs

A starting point is to provide some registers to define the region of memory used by a currently executing application program, along with a (boolean) register to record whether the processor is executing kernel software or application software:

register	**use**
ab	the base address of the application memory
as	the size of the application memory
ink	the *executing in kernel* flag

Some instructions are needed to set these registers. They must only be executed by the kernel; otherwise an application program could change its own memory region.

SETAB	if ink then $ab \leftarrow s$ else $error$	set application base
SETAS	if ink then $as \leftarrow s$ else $error$	set application size

This makes it possible to re-define some of the instructions so as to prevent an application program from corrupting (or branching into) the kernel; for example:

STAM if ink
then $mem[oreg] \leftarrow areg$
else
if $oreg < as$
then $mem[ab + oreg] \leftarrow areg$
else $error$

LDAM if ink
then $areg \leftarrow mem[oreg]$
else
if $oreg < as$
then $areg \leftarrow mem[ab + oreg]$
else $error$

STAI if ink
then $mem[breg + oreg] \leftarrow areg$
else
if $(breg + oreg) < as$
then $mem[ab + breg + oreg] \leftarrow areg$
else $error$

LDAI if ink
then $areg \leftarrow mem[areg + oreg]$
else
if $(areg + oreg) < as$
then $areg \leftarrow mem[ab + areg + oreg]$
else $error$

BRU if ink
then $pc \leftarrow pc + oreg$
else
if $(pc + oreg) < as$
then $pc \leftarrow pc + oreg$
else $error$

All of the instructions that access memory and all of the branch instructions must be modified in this way.

Notice that this has resulted in application programs being *relocatable*; they can be moved around in memory by the kernel because all of the addresses they use are offsets relative to the ab register. It is also possible to have several programs in memory at the same time, and to move them to and from a disc.

Switching to the Kernel

There are three potential reasons for switching to the kernel:

- An error has been detected in an application program
- An application program has made a request to the kernel (a *kernel call*)
- An input-output device has made a request to the processor

All of these perform a similar operation on kernel entry, selecting different vectors within the kernel corresponding to the different reasons for entry. Entry to the kernel as a result of an error is similar to entry as a result of an interrupt request:

$$\begin{array}{lll} error & spc \leftarrow pc; & \text{kernel entry - error number } n \\ & pc \leftarrow mem[kerr + n] & \\ & ink \leftarrow true & \end{array}$$

The same kernel return instruction (KRET) can be used to return to an application program (which may be different from the one on kernel entry) regardless of the reason for switching to the kernel.

Finally, it should be possible to write the kernel itself in a high level language using a stack in the normal way. This requires a separate location in memory to use as the kernel stack pointer.

Scheduling

In addition to the device drivers, a kernel usually allocates memory and input-output devices to application programs. It also shares the processor's time between several programs, so that the processor appears to be executing several programs at the same time. This is known as *scheduling*.

A simple scheduler operates using a queue of programs. This queue contains programs that are able to proceed; there will normally also be other programs that are not able to proceed. For example, a program will be unable to proceed if:

- it is waiting for input data from a device which is not ready
- it is waiting to output data to a device which is not ready
- it is waiting until a specified time
- it is waiting for a page in virtual memory to be moved to physical memory as a result of a page fault (see notes on the memory hierarchy)

In these cases, the currently running program will have entered the kernel either as a result of a kernel call or as a result of an error (page fault). When an executing program becomes unable to proceed, it is *de-scheduled*. Its state (the register contents) is saved, then the next program is taken from the queue and its state is restored.

A waiting program will become able to proceed again as a result of an interrupt request from a device causing entry to the kernel. The scheduler will identify the program waiting for the device and *re-schedule* it by adding it to the end of the queue.

Note that it is possible that the queue is empty when a program is de-scheduled in which case the processor must wait for an interrupt. The instruction set should include an instruction to do this; if not, an *idling loop* can be used.

An example of the operation of a simple scheduler is:

- **Interrupt from timer:** update current time; if the current program has exceeded its allocated time-slice, de-schedule it and re-schedule it; also re-schedule any programs waiting until the (new) current time; otherwise return.
- **Interrupt from input device:** transfer data to buffer; if buffer full disable interrupts; if there is a de-scheduled program waiting, re-schedule it.
- **Kernel call for input:** if buffer not empty, return data; if buffer empty, de-schedule and enable interrupts
- **Interrupt from output device:** transfer data to device; if buffer empty disable interrupts; if there is a de-scheduled program waiting, re-schedule it.
- **Kernel call for output:** transfer data to buffer, enable interrupts; if buffer full, de-schedule
- **Error - page-fault:** initiate page transfer, de-schedule; this will subsequently result in an interrupt as a result of the page transfer completing at which point the program will be re-scheduled.
- **Other kernel call:** there are usually many of these to provide access to files, initiating and ending programs etc.
- **Other error:** write the program to disc and de-schedule.