Theory Assignment 1 - CS3510

Problem 1.20

Direct memory access is used for high-speed I/O devices in order to avoid increasing the CPU's execution load.

a) How does the CPU interface with the device to coordinate the transfer?

In DMA, an I/O device is allocated buffers, pointers and counters which it uses to directly transfer a block of data to memory. No CPU interrupts are raised during this transfer.

b) How does the CPU know when the memory operations are complete?

At the end of the transfer, a single interrupt is raised to tell the CPU that the operation completed.

c) The CPU is allowed to execute other programs while the DMA controller is transferring data. Does this process interfere with the execution of the user programs? If so, describe what forms of interference are caused.

This process makes the I/O device unavailable till the transfer is complete.

Problem 1.24

Discuss, with examples, how the problem of maintaining coherence of cached data manifests itself in the following processing environments:

Single-processor systems

Cache coherency isn't an issue in single-processor systems, as only one processor reads and writes from the cache; cache coherency is mantained as usual.

Multi-processor systems

For multi-processor systems, each CPU also has a local cache. If the same variables exist in more than CPU's local cache, and they need to updated across all CPUs. This is handled by the hardware itself (and is dependent on the CPU architecure).

Distributed systems

In a distributed setting, several copies of a file can be kept on different machines. As these can be accessed and updated concurrently, the distributed system needs to ensure that they're updated across all machines as soon as possible.

Problem 3.2

How many total processes are created by the following program?

```
#include <stdio.h>
#include <unistd.h>

int main() {
/* fork a child process */
    fork();
/* fork another child process */
    fork();
/* and fork another */
    fork();
    return 0;
}
```

This code generates a total of *8 processes*, as can be seen here - http://i.imgur.com/m0KyRLG.png (http://i.imgur.com/m0KyRLG.png)

In general, consecutive calls on `fork()` will generate 2^n processes.

Problem 3.14

Using the program in Figure 3.34, identify values of `pid` at lines A, B, C and D, given that `pid` of parent and child are `2600` and `2603`.

At line A: We're printing the return value of the `fork()` call from the child process, which is 0.

```
child: pid = 0
```

At line B: We're calling `getpid()` from the child, and printing the result. This will return the true `pid` of the child.

```
child: pid1 = 2603
```

At line C: We're printing the return value of the `fork()` call from the parent process. This will return the `pid` of the child.

```
parent: pid = 2603
```

At line D: We're calling `getpid()` from the parent process. This will return the `pid` of the parent process itself.

```
parent: pid = 2600
```

Extra Credit Question

Describe any 10 fields in the Linux Process Control Block (PCB)

The Linux PCB is internally called `task_struct` in the source code. The members of `task_struct` are defined in the sched.h file (http://elixir.freeelectrons.com/linux/latest/source/include/linux/sched.h) in the source tree. Here, we attempt to understand some 12 fields of the struct.

- `volatile long state`
  This field keeps track of the state of the process. In general, if it is -1, the process is unrunnable, if it's 0 it's runnable, if it's >0, it's stopped. The exact states are defined using `TASK_RUNNING`, `TASK_INTERRUPTIBLE`, `TASK_UNINTERRUPTIBLE`, `TASK_STOPPED`, `TASK_ZOMBIE` and `TASK_DEAD`.

- `int prio, static_prio`
  This defines the priority of the process when scheduled.

- `unsigned long sleep_avg`
  Stores the average sleep time of the process.

- `unsigned long long timestamp`
  Stores the time when the process is activated. It's used for things such as recalculating the task's priority.

- `cpumask_t cpus_allowed`
  It's a mask that indicates on which CPUs the process is allowed to run.

- `int exit_code, exit_signal`
  Holds the code or signal when a process exits. `exit_code` can be `SIGHUP`, `SIGINT`, `SIGQUIT` and so on that are sent to the process to stop it. `exit_signal` is used with `SIGCHILD` to signal parent process on exit.

- `pid_t pid`
  Holds the process id, which acts as its global identifier.

- `struct task_struct *parent`
  A pointer to the PCB (`task_struct`) of the parent process.

- `struct list_head children`
  A list of all the children of the process.

- `struct list_head sibling`
  A link to the parent's `struct list_head children`, i.e, the parent's children list.

- `struct fs_struct *fs`
  Pointer to the structure containing the filesystem information.

- `struct files_struct *files`
  List of all the files currently opened by the process.