1 Slide *'kernel-level-memory-management'*

1.1 NUMA

Linux 2.6 supports the *Non-Uniform Memory Access* (**NUMA**) model, **in which the access times for different memory locations from a given CPU may vary**. According to that architecture, physical memory is partitioned in several **nodes**.

To represent and manage a NUMA nodes, Linux uses a singly linked list of NUMA node descriptors of type `pg_data_t`, whose first element is pointed to by the `pgdat_list` variable.

Be careful to the fact that this data structure is used by Linux kernel even if the architecture is based on *Uniform Memory Access* (**UMA**): in fact Linux makes use of a single node that includes all system physical memory. Thus, the `pgdat_list` variable points to a list consisting of a single element (node 0) stored in the `contig_page_data` variable.

A node descriptor has several fields, the most important of them is:

`struct page *node_mem_map` Array of page descriptors of the node

However **Linux 2.6 partitions the physical memory of every memory node into three zones** which are:

`ZONE_DMA` Contains page frames of memory below 16 MB, that is page frames that can be used by old ISA-based devices (*Direct Memory Access* (DMA) processors).

`ZONE_NORNMAL` Contains page frames of memory at and above 16 MB and below 896 MB.

`ZONE_HIGHMEM` Contains page frames of memory at and above 896 MB

The `ZONE_DMA` and `ZONE_NORNMAL` zones include page frames that can be directly accessed by the kernel. The `ZONE_HIGHMEM` zone is always empty on 64-bit architectures.

2 Part 1

Since a data dependence can limit the amount of instruction-level parallelism we can exploit, a major focus of this chapter is overcoming these limitations. A dependence can be overcome in two different ways: maintaining the dependence but avoiding a hazard, and eliminating a dependence by transforming the code. Scheduling the code is the primary method used to avoid a hazard without altering a dependence, and such scheduling can be done both by the compiler and by the hardware.

Top and Bottom Halves One of the main problems with interrupt handling is how to perform lengthy tasks within a handler. Often a substantial amount of work must be done in response to a device interrupt, but interrupt handlers need to finish up quickly and not keep inter- rupts blocked for long. These two needs (work and speed) conflict with each other, leaving the driver writer in a bit of a bind. Linux (along with many other systems) resolves this problem by splitting the inter- rupt handler into two halves. The so-called top half is

the routine that actually responds to the interrupt—the one you register with request_irq. The bottom half is a routine that is scheduled by the top half to be executed later, at a safer time. The big difference between the top-half handler and the bottom half is that all interrupts are enabled during execution of the bottom half—that's why it runs at a safer time. In the typical scenario, the top half saves device data to a device-specific buffer, sched- ules its bottom half, and exits: this operation is very fast. The bottom half then per- forms whatever other work is required, such as awakening processes, starting up another I/O operation, and so on. This setup permits the top half to service a new interrupt while the bottom half is still working.

ksoftirqd is a per-cpu kernel thread which runs in background (as a deamon): is triggered to handle the software interrupts in process context.

NR_CPUS structures (the default value for this macro is 32; it denotes the maximum number of CPUs in the system