OS LAB4 - REPORT

1.

به منظور جلوگیری از بن بست وقفه‌ها را غیرفعال می‌کنیم. اگر وقفه‌ها فعال باشند، کد در حال اجرا در kernel mode می‌تواند در هر لحظه متوقف شود تا یک interrupt handler اجرا شود. برای مثال فرض کنید که iderw، idelock را گرفته است و در حین اجرا ناگهان وقفه‌ای ایجاد می‌شود که می‌تواند منجر به اجرای ideintr شود؛ ideintr برای اجرا می‌کوشد که idelock را بگیرد که از آنجا که این قفل در اختیار iderw است و iderw تنها زمانی به ادامه‌ی اجرا شدن می‌پردازد که از interrupt handler خارج شده باشیم که این شرایط منجر به بن بست می‌شود. برای حل این مشکل، اگر یک interrupt handler نیاز به یک lock داشته باشد، هنگام اجرای آن lock نباید توسط پردازه‌ی دیگری گرفته شده باشد که به منظور حل آن وقفه‌ها را غیرفعال می‌کنیم.
برای حل مشکل نواحی بحرانی تودرتو، از توابع pushcli در acquire و popcli در release استفاده می‌شود. این توابع برای نگهداری این مورد که در کدام سطح از نواحی بحرانی تودرتو قرار داریم استفاده می‌شوند) شماره‌ی سطح در متغیر ncliهر پردازنده استفاده می‌شود .(هنگام آزاد کردن یک قفل ncli یک واحد کاسته شده و هنگام گرفتن یک قفل یک واحد افزایش می‌یابد. این کار تا زمانی ادامه می‌یابد که ncli صفر شود و در این حالت وقفه‌ها فعال می‌شوند. توابع cli وsti برای غیرفعال کردن و فعال کردن وقفه‌ها استفاده می‌شوند که از این منظر مشابه popcli و pushcli هستند اما تفاوت آن‌ها در این است که هرجا از pushcli استفاده شده است، باید از popcli هم استفاده شود و این دو تابع با هم استفاده می‌شوند در حالی که برای cli و sti این مورد لزوما برقرار نیست. همچنین در صورتی که ncli به صفر نرسد، pushcli و popcli تغییری در وضعیت فعال یا غیرفعال بودن وقفه‌ها ایجاد نمی‌کنند در حالی که cli و sti در هربار فراخوانی این وضعیت را تغییر می‌دهند.

2.1

Whether the process is allocated, ready to run, running, waiting
for I/O, or exiting.

2.2

One possibility at the end of each interrupt is that trap calls yield. Yield in turn calls sched, which calls swtch to save the current context in proc->context and switch to the scheduler context previously saved in cpu->scheduler

A process that wants to give up the CPU must acquire the process table lock ptable.lock, release any other locks it is holding, update its own state (proc->state), and then call sched. Yield follows this convention, as do sleep and exit, which we will examine later. Sched double-checks those conditions and then an implication of those conditions: since a lock is held, the CPU should be running with interrupts disabled. Finally, sched calls swtch to save the current context in proc->context and switch to the scheduler context in cpu->scheduler.

A kernel thread always gives up its processor in sched and always switches to the
same location in the scheduler, which (almost) always switches to a process in sched.

3.

Mutex is like a spinlock, but you may block holding a mutex. If you can't lock a mutex, your task will suspend itself, and be woken up when the mutex is released. This means the CPU can do something else while you are waiting.

4.
Hardware schema can be divided into two categories :

- **Directory Protocols**
- **Snoopy Protocols**

Directory Protocols

In this approach we collect and maintain information about where copies of lines reside in each cache.

Typical implementation has a centralized controller which is a part of the main memory controller. There is a directory that is stored in main memory which contains global state information about the contents of the various local caches.

When an individual cache controller makes a request, the centralized controller checks and issues necessary commands for data transfer between memory and caches or between caches themselves.

It is also responsible for keeping the state information up to date, therefore, every local action that can affect the global state of a line must be reported to the central controller. The controller maintains information about which processors have a copy of which lines.

Before a processor can write to a local copy of a line, it must request exclusive access to the line from the controller.

Let’s see the flow when an individual cache controller tries to update its local copy of the cache line.

- Local cache controller request for an exclusive access to the line from the centralized cache controller

- Before granting the exclusive access, the controller sends a message to all processors with a cached copy of this time, forcing each processors to invalidate its copy.

- Centralized controller receives the acknowledgement back from each processor

- Controller grants exclusive access to the requesting processor.

- When another processor tries to read a line that is exclusively granted to another processors, this results in a cache miss, it will send a miss notification to the controller.

- Upon receiving the cache miss notification controller will issue command to the processor holding the exclusive access to write-back to main memory.

Drawbacks of directory schema :

- Central cache controller bottleneck
- Communication overhead between various cache controller and central controller.

Snoopy Protocols

Snoopy protocols distribute the responsibility for maintaining cache coherence among all of the cache controllers in a multiprocessor system.

A cache must recognize when a line that it holds is shared with other caches. When an update action is performed on a shared cache line, it must be announced to all other caches by a broadcast mechanism.

Each cache controller is able to ***"snoop"*** on the network to observed these broadcasted notification and react accordingly.

Snoopy protocols are ideally suited to a bus-based multiprocessor, because the shared bus provides a simple means for broadcasting and snooping.

There are two basic approaches to snoopy protocols explored :

- ***Write-invalidate snoopy protocol***
- ***Write-update (write-broadcast) snoopy protocol***

Write-invalidate protocol

There can be multiple readers but only one write at a time.

Initially, a line may be shared among several caches for reading purposes.
When one of the caches wants to perform a write to the line it first issues a notice that invalidates that tine in the other caches, making the line exclusive to the writing cache.

Once the line is exclusive, the owning processor can make local writes until some other processor requires the same line.

Write with update protocol

There can be multiple writers as well as multiple readers.

When a processors wishes to update a shared line, the word to be updated is distributed to all others, and caches containing that line can update it.

5.

Non-Scalable

6.
Defining a per-CPU variable at compile time is quite easy: DEFINE_PER_CPU(type, name); This creates an instance of a variable of type type, named name, for each processor on the system. If you need a declaration of the variable elsewhere, to avoid compile warnings, the following macro is your friend: DECLARE_PER_CPU(type, name); You can manipulate the variables with the get_cpu_var() and put_cpu_var() routines.A call to get_cpu_var() returns an lvalue for the given variable on the current processor. It also disables preemption, which put_cpu_var() correspondingly enables. get_cpu_var(name)++; /* increment name on this processor */ put_cpu_var(name); /* done; enable kernel preemption */
You can obtain the value of another processor's per-CPU data, too: per_cpu(name, cpu)++; /* increment name on the given processor */ You need to be careful with this approach because per_cpu() neither disables kernel preemption nor provides any sort of locking mechanism.The lockless nature of per-CPU data exists only if the current processor is the only manipulator of the data. If other processors touch other processors' data, you need locks. Be careful. Chapter 9,"An Introduction to Kernel Synchronization," and Chapter 10,"Kernel Synchronization Methods," discuss locking. Another subtle note:These compile-time per-CPU examples do not work for modules because the linker actually creates them in a unique executable section (for the curious, .data.percpu). If you need to access per-CPU data from modules, or if you need to create such data dynamically, there is hope.