```
**********************************************************
   27054 Fri May 22 16:26:10 2015
new/usr/src/uts/i86pc/os/cpr_impl.c
XXXX pat_sync is clever enough to check for X86FSET_PAT
**********************************************************
______unchanged_portion_omitted_

 920 void
 921 i_cpr_start_cpu(void)
 922 {

 924         struct cpu *cp = CPU;

 926         char *str = "i_cpr_start_cpu";
 927         extern void init_cpu_syscall(struct cpu *cp);

 929         PMD(PMD_SX, ("%s() called\n", str))

 931         PMD(PMD_SX, ("%s() #0 cp->cpu_base_spl %d\n", str,
 932             cp->cpu_base_spl))

 934         mutex_enter(&cpu_lock);
 935         if (cp == i_cpr_bootcpu()) {
 936                 mutex_exit(&cpu_lock);
 937                 PMD(PMD_SX,
 938                     ("%s() called on bootcpu nothing to do!\n", str))
 939                 return;
 940         }
 941         mutex_exit(&cpu_lock);

 943         /*
 944          * We need to Sync PAT with cpu0's PAT. We have to do
 945          * this with interrupts disabled.
 946          */
 947         if (is_x86_feature(x86_featureset, X86FSET_PAT))
 947                 pat_sync();

 949         /*
 950          * If we use XSAVE, we need to restore XFEATURE_ENABLE_MASK register.
 951          */
 952         if (fp_save_mech == FP_XSAVE) {
 953                 setup_xfem();
 954         }

 956         /*
 957          * Initialize this CPU's syscall handlers
 958          */
 959         init_cpu_syscall(cp);

 961         PMD(PMD_SX, ("%s() #1 cp->cpu_base_spl %d\n", str, cp->cpu_base_spl))

 963         /*
 964          * Do not need to call cpuid_pass2(), cpuid_pass3(), cpuid_pass4() or
 965          * init_cpu_info(), since the work that they do is only needed to
 966          * be done once at boot time
 967          */


 970         mutex_enter(&cpu_lock);
 971         CPUSET_ADD(procset, cp->cpu_id);
 972         mutex_exit(&cpu_lock);

 974         PMD(PMD_SX, ("%s() #2 cp->cpu_base_spl %d\n", str,
 975             cp->cpu_base_spl))

 977         if (tsc_gethrtime_enable) {
```



```
 978                 PMD(PMD_SX, ("%s() calling tsc_sync_slave\n", str))
 979                 tsc_sync_slave();
 980         }

 982         PMD(PMD_SX, ("%s() cp->cpu_id %d, cp->cpu_intr_actv %d\n", str,
 983             cp->cpu_id, cp->cpu_intr_actv))
 984         PMD(PMD_SX, ("%s() #3 cp->cpu_base_spl %d\n", str,
 985             cp->cpu_base_spl))

 987         (void) spl0();          /* enable interrupts */

 989         PMD(PMD_SX, ("%s() #4 cp->cpu_base_spl %d\n", str,
 990             cp->cpu_base_spl))

 992         /*
 993          * Set up the CPU module for this CPU.  This can't be done before
 994          * this CPU is made CPU_READY, because we may (in heterogeneous systems)
 995          * need to go load another CPU module.  The act of attempting to load
 996          * a module may trigger a cross-call, which will ASSERT unless this
 997          * cpu is CPU_READY.
 998          */

1000         /*
1001          * cmi already been init'd (during boot), so do not need to do it again
1002          */
1003 #ifdef PM_REINITMCAONRESUME
1004         if (is_x86_feature(x86_featureset, X86FSET_MCA))
1005                 cmi_mca_init();
1006 #endif

1008         PMD(PMD_SX, ("%s() returning\n", str))

1010         /* return; */
1011 }
______unchanged_portion_omitted_
```

```
**********************************************************
   50607 Fri May 22 16:26:10 2015
new/usr/src/uts/i86pc/os/mp_startup.c
XXXX pat_sync is clever enough to check for X86FSET_PAT
**********************************************************
______unchanged_portion_omitted_

1620 /*
1621  * Startup function for 'other' CPUs (besides boot cpu).
1622  * Called from real_mode_start.
1623  *
1624  * WARNING: until CPU_READY is set, mp_startup_common and routines called by
1625  * mp_startup_common should not call routines (e.g. kmem_free) that could call
1626  * hat_unload which requires CPU_READY to be set.
1627  */
1628 static void
1629 mp_startup_common(boolean_t boot)
1630 {
1631         cpu_t *cp = CPU;
1632         uchar_t new_x86_featureset[BT_SIZEOFMAP(NUM_X86_FEATURES)];
1633         extern void cpu_event_init_cpu(cpu_t *);

1635         /*
1636          * We need to get TSC on this proc synced (i.e., any delta
1637          * from cpu0 accounted for) as soon as we can, because many
1638          * many things use gethrtime/pc_gethrestime, including
1639          * interrupts, cmn_err, etc.  Before we can do that, we want to
1640          * clear TSC if we're on a buggy Sandy/Ivy Bridge CPU, so do that
1641          * right away.
1642          */
1643         bzero(new_x86_featureset, BT_SIZEOFMAP(NUM_X86_FEATURES));
1644         cpuid_pass1(cp, new_x86_featureset);

1646         if (boot && get_hwenv() == HW_NATIVE &&
1647             cpuid_getvendor(CPU) == X86_VENDOR_Intel &&
1648             cpuid_getfamily(CPU) == 6 &&
1649             (cpuid_getmodel(CPU) == 0x2d || cpuid_getmodel(CPU) == 0x3e) &&
1650             is_x86_feature(new_x86_featureset, X86FSET_TSC)) {
1651                 (void) wrmsr(REG_TSC, 0UL);
1652         }

1654         /* Let the control CPU continue into tsc_sync_master() */
1655         mp_startup_signal(&procset_slave, cp->cpu_id);

1657 #ifndef __xpv
1658         if (tsc_gethrtime_enable)
1659                 tsc_sync_slave();
1660 #endif

1662         /*
1663          * Once this was done from assembly, but it's safer here; if
1664          * it blocks, we need to be able to swtch() to and from, and
1665          * since we get here by calling t_pc, we need to do that call
1666          * before swtch() overwrites it.
1667          */
1668         (void) (*ap_mlsetup)();

1670 #ifndef __xpv
1671         /*
1672          * Program this cpu's PAT
1673          */
1674         if (is_x86_feature(x86_featureset, X86FSET_PAT))
1674         pat_sync();
1675 #endif

1677         /*
```



```
1678          * Set up TSC_AUX to contain the cpuid for this processor
1679          * for the rdtscp instruction.
1680          */
1681         if (is_x86_feature(x86_featureset, X86FSET_TSCP))
1682                 (void) wrmsr(MSR_AMD_TSCAUX, cp->cpu_id);

1684         /*
1685          * Initialize this CPU's syscall handlers
1686          */
1687         init_cpu_syscall(cp);

1689         /*
1690          * Enable interrupts with spl set to LOCK_LEVEL. LOCK_LEVEL is the
1691          * highest level at which a routine is permitted to block on
1692          * an adaptive mutex (allows for cpu poke interrupt in case
1693          * the cpu is blocked on a mutex and halts). Setting LOCK_LEVEL blocks
1694          * device interrupts that may end up in the hat layer issuing cross
1695          * calls before CPU_READY is set.
1696          */
1697         splx(ipltospl(LOCK_LEVEL));
1698         sti();

1700         /*
1701          * Do a sanity check to make sure this new CPU is a sane thing
1702          * to add to the collection of processors running this system.
1703          *
1704          * XXX  Clearly this needs to get more sophisticated, if x86
1705          * systems start to get built out of heterogenous CPUs; as is
1706          * likely to happen once the number of processors in a configuration
1707          * gets large enough.
1708          */
1709         if (compare_x86_featureset(x86_featureset, new_x86_featureset) ==
1710             B_FALSE) {
1711                 cmn_err(CE_CONT, "cpu%d: featureset\n", cp->cpu_id);
1712                 print_x86_featureset(new_x86_featureset);
1713                 cmn_err(CE_WARN, "cpu%d feature mismatch", cp->cpu_id);
1714         }

1716         /*
1717          * We do not support cpus with mixed monitor/mwait support if the
1718          * boot cpu supports monitor/mwait.
1719          */
1720         if (is_x86_feature(x86_featureset, X86FSET_MWAIT) !=
1721             is_x86_feature(new_x86_featureset, X86FSET_MWAIT))
1722                 panic("unsupported mixed cpu monitor/mwait support detected");

1724         /*
1725          * We could be more sophisticated here, and just mark the CPU
1726          * as "faulted" but at this point we'll opt for the easier
1727          * answer of dying horribly.  Provided the boot cpu is ok,
1728          * the system can be recovered by booting with use_mp set to zero.
1729          */
1730         if (workaround_errata(cp) != 0)
1731                 panic("critical workaround(s) missing for cpu%d", cp->cpu_id);

1733         /*
1734          * We can touch cpu_flags here without acquiring the cpu_lock here
1735          * because the cpu_lock is held by the control CPU which is running
1736          * mp_start_cpu_common().
1737          * Need to clear CPU_QUIESCED flag before calling any function which
1738          * may cause thread context switching, such as kmem_alloc() etc.
1739          * The idle thread checks for CPU_QUIESCED flag and loops for ever if
1740          * it's set. So the startup thread may have no chance to switch back
1741          * again if it's switched away with CPU_QUIESCED set.
1742          */
1743         cp->cpu_flags &= ~(CPU_POWEROFF | CPU_QUIESCED);
```

```
1745         /*
1746          * Setup this processor for XSAVE.
1747          */
1748         if (fp_save_mech == FP_XSAVE) {
1749                 xsave_setup_msr(cp);
1750         }

1752         cpuid_pass2(cp);
1753         cpuid_pass3(cp);
1754         cpuid_pass4(cp, NULL);

1756         /*
1757          * Correct cpu_idstr and cpu_brandstr on target CPU after
1758          * cpuid_pass1() is done.
1759          */
1760         (void) cpuid_getidstr(cp, cp->cpu_idstr, CPU_IDSTRLEN);
1761         (void) cpuid_getbrandstr(cp, cp->cpu_brandstr, CPU_IDSTRLEN);

1763         cp->cpu_flags |= CPU_RUNNING | CPU_READY | CPU_EXISTS;

1765         post_startup_cpu_fixups();

1767         cpu_event_init_cpu(cp);

1769         /*
1770          * Enable preemption here so that contention for any locks acquired
1771          * later in mp_startup_common may be preempted if the thread owning
1772          * those locks is continuously executing on other CPUs (for example,
1773          * this CPU must be preemptible to allow other CPUs to pause it during
1774          * their startup phases).  It's safe to enable preemption here because
1775          * the CPU state is pretty-much fully constructed.
1776          */
1777         curthread->t_preempt = 0;

1779         /* The base spl should still be at LOCK LEVEL here */
1780         ASSERT(cp->cpu_base_spl == ipltospl(LOCK_LEVEL));
1781         set_base_spl();         /* Restore the spl to its proper value */

1783         pghw_physid_create(cp);
1784         /*
1785          * Delegate initialization tasks, which need to access the cpu_lock,
1786          * to mp_start_cpu_common() because we can't acquire the cpu_lock here
1787          * during CPU DR operations.
1788          */
1789         mp_startup_signal(&procset_slave, cp->cpu_id);
1790         mp_startup_wait(&procset_master, cp->cpu_id);
1791         pg_cmt_cpu_startup(cp);

1793         if (boot) {
1794                 mutex_enter(&cpu_lock);
1795                 cp->cpu_flags &= ~CPU_OFFLINE;
1796                 cpu_enable_intr(cp);
1797                 cpu_add_active(cp);
1798                 mutex_exit(&cpu_lock);
1799         }

1801         /* Enable interrupts */
1802         (void) spl0();

1804         /*
1805          * Fill out cpu_ucode_info.  Update microcode if necessary.
1806          */
1807         ucode_check(cp);

1809 #ifndef __xpv
```



```
1810         {
1811                 /*
1812                  * Set up the CPU module for this CPU.  This can't be done
1813                  * before this CPU is made CPU_READY, because we may (in
1814                  * heterogeneous systems) need to go load another CPU module.
1815                  * The act of attempting to load a module may trigger a
1816                  * cross-call, which will ASSERT unless this cpu is CPU_READY.
1817                  */
1818                 cmi_hdl_t hdl;

1820                 if ((hdl = cmi_init(CMI_HDL_NATIVE, cmi_ntv_hwchipid(CPU),
1821                     cmi_ntv_hwcoreid(CPU), cmi_ntv_hwstrandid(CPU))) != NULL) {
1822                         if (is_x86_feature(x86_featureset, X86FSET_MCA))
1823                                 cmi_mca_init(hdl);
1824                         cp->cpu_m.mcpu_cmi_hdl = hdl;
1825                 }
1826         }
1827 #endif /* __xpv */

1829         if (boothowto & RB_DEBUG)
1830                 kdi_cpu_init();

1832         /*
1833          * Setting the bit in cpu_ready_set must be the last operation in
1834          * processor initialization; the boot CPU will continue to boot once
1835          * it sees this bit set for all active CPUs.
1836          */
1837         CPUSET_ATOMIC_ADD(cpu_ready_set, cp->cpu_id);

1839         (void) mach_cpu_create_device_node(cp, NULL);

1841         cmn_err(CE_CONT, "?cpu%d: %s\n", cp->cpu_id, cp->cpu_idstr);
1842         cmn_err(CE_CONT, "?cpu%d: %s\n", cp->cpu_id, cp->cpu_brandstr);
1843         cmn_err(CE_CONT, "?cpu%d initialization complete - online\n",
1844             cp->cpu_id);

1846         /*
1847          * Now we are done with the startup thread, so free it up.
1848          */
1849         thread_exit();
1850         panic("mp_startup: cannot return");
1851         /*NOTREACHED*/
1852 }
```

_____***unchanged_portion_omitted_***