/\*

\* linux/kernel/sched.c

\*

\* Kernel scheduler and related syscalls

\*

\* Copyright (C) 1991, 1992 Linus Torvalds

\*

\* 1996-12-23 Modified by Dave Grothe to fix bugs in semaphores and

\* make semaphores SMP safe

\* 1998-11-19 Implemented schedule\_timeout() and related stuff

\* by Andrea Arcangeli

\* 1998-12-28 Implemented better SMP scheduling by Ingo Molnar

\*/

/\*

\* 'sched.c' is the main kernel file. It contains scheduling primitives

\* (sleep\_on, wakeup, schedule etc) as well as a number of simple system

\* call functions (type getpid()), which just extract a field from

\* current-task

\*/

#include <linux/config.h>

#include <linux/mm.h>

#include <linux/init.h>

#include <linux/smp\_lock.h>

#include <linux/nmi.h>

#include <linux/interrupt.h>

#include <linux/kernel\_stat.h>

#include <linux/completion.h>

#include <linux/prefetch.h>

#include <linux/compiler.h>

#include <asm/uaccess.h>

#include <asm/mmu\_context.h>

extern void timer\_bh(void);

extern void tqueue\_bh(void);

extern void immediate\_bh(void);

/\*

\* scheduler variables

\*/

unsigned securebits = SECUREBITS\_DEFAULT; /\* systemwide security settings \*/

extern void mem\_use(void);

/\*

\* Scheduling quanta.

\*

\* NOTE! The unix "nice" value influences how long a process

\* gets. The nice value ranges from -20 to +19, where a -20

\* is a "high-priority" task, and a "+10" is a low-priority

\* task.

\*

\* We want the time-slice to be around 50ms or so, so this

\* calculation depends on the value of HZ.

\*/

#if HZ < 200

#define TICK\_SCALE(x) ((x) >> 2)

#elif HZ < 400

#define TICK\_SCALE(x) ((x) >> 1)

#elif HZ < 800

#define TICK\_SCALE(x) (x)

#elif HZ < 1600

#define TICK\_SCALE(x) ((x) << 1)

#else

#define TICK\_SCALE(x) ((x) << 2)

#endif

#define NICE\_TO\_TICKS(nice) (TICK\_SCALE(20-(nice))+1)

/\*

\* Init task must be ok at boot for the ix86 as we will check its signals

\* via the SMP irq return path.

\*/

struct task\_struct \* init\_tasks[NR\_CPUS] = {&init\_task, };

/\*

\* The tasklist\_lock protects the linked list of processes.

\*

\* The runqueue\_lock locks the parts that actually access

\* and change the run-queues, and have to be interrupt-safe.

\*

\* If both locks are to be concurrently held, the runqueue\_lock

\* nests inside the tasklist\_lock.

\*

\* task->alloc\_lock nests inside tasklist\_lock.

\*/

spinlock\_t runqueue\_lock \_\_cacheline\_aligned = SPIN\_LOCK\_UNLOCKED; /\* inner \*/

rwlock\_t tasklist\_lock \_\_cacheline\_aligned = RW\_LOCK\_UNLOCKED; /\* outer \*/

static LIST\_HEAD(runqueue\_head);

/\*

\* We align per-CPU scheduling data on cacheline boundaries,

\* to prevent cacheline ping-pong.

\*/

static union {

struct schedule\_data {

struct task\_struct \* curr;

cycles\_t last\_schedule;

} schedule\_data;

char \_\_pad [SMP\_CACHE\_BYTES];

} aligned\_data [NR\_CPUS] \_\_cacheline\_aligned = { {{&init\_task,0}}};

#define cpu\_curr(cpu) aligned\_data[(cpu)].schedule\_data.curr

#define last\_schedule(cpu) aligned\_data[(cpu)].schedule\_data.last\_schedule

struct kernel\_stat kstat;

extern struct task\_struct \*child\_reaper;

#ifdef CONFIG\_SMP

#define idle\_task(cpu) (init\_tasks[cpu\_number\_map(cpu)])

#define can\_schedule(p,cpu) \

((p)->cpus\_runnable & (p)->cpus\_allowed & (1 << cpu))

#else

#define idle\_task(cpu) (&init\_task)

#define can\_schedule(p,cpu) (1)

#endif

void scheduling\_functions\_start\_here(void) { }

/\*

\* This is the function that decides how desirable a process is..

\* You can weigh different processes against each other depending

\* on what CPU they've run on lately etc to try to handle cache

\* and TLB miss penalties.

\*

\* Return values:

\* -1000: never select this

\* 0: out of time, recalculate counters (but it might still be

\* selected)

\* +ve: "goodness" value (the larger, the better)

\* +1000: realtime process, select this.

\*/

static inline int goodness(struct task\_struct \* p, int this\_cpu, struct mm\_struct \*this\_mm)

{

int weight;

/\*

\* select the current process after every other

\* runnable process, but before the idle thread.

\* Also, dont trigger a counter recalculation.

\*/

weight = -1;

if (p->policy & SCHED\_YIELD)

goto out;

/\*

\* Non-RT process - normal case first.

\*/

if (p->policy == SCHED\_OTHER) {

/\*

\* Give the process a first-approximation goodness value

\* according to the number of clock-ticks it has left.

\*

\* Don't do any other calculations if the time slice is

\* over..

\*/

weight = p->counter;

if (!weight)

goto out;

#ifdef CONFIG\_SMP

/\* Give a largish advantage to the same processor... \*/

/\* (this is equivalent to penalizing other processors) \*/

if (p->processor == this\_cpu)

weight += PROC\_CHANGE\_PENALTY;

#endif

/\* .. and a slight advantage to the current MM \*/

if (p->mm == this\_mm || !p->mm)

weight += 1;

weight += 20 - p->nice;

goto out;

}

/\*

\* Realtime process, select the first one on the

\* runqueue (taking priorities within processes

\* into account).

\*/

weight = 1000 + p->rt\_priority;

out:

return weight;

}

/\*

\* the 'goodness value' of replacing a process on a given CPU.

\* positive value means 'replace', zero or negative means 'dont'.

\*/

static inline int preemption\_goodness(struct task\_struct \* prev, struct task\_struct \* p, int cpu)

{

return goodness(p, cpu, prev->active\_mm) - goodness(prev, cpu, prev->active\_mm);

}

/\*

\* This is ugly, but reschedule\_idle() is very timing-critical.

\* We are called with the runqueue spinlock held and we must

\* not claim the tasklist\_lock.

\*/

static FASTCALL(void reschedule\_idle(struct task\_struct \* p));

static void reschedule\_idle(struct task\_struct \* p)

{

#ifdef CONFIG\_SMP

int this\_cpu = smp\_processor\_id();

struct task\_struct \*tsk, \*target\_tsk;

int cpu, best\_cpu, i, max\_prio;

cycles\_t oldest\_idle;

/\*

\* shortcut if the woken up task's last CPU is

\* idle now.

\*/

best\_cpu = p->processor;

if (can\_schedule(p, best\_cpu)) {

tsk = idle\_task(best\_cpu);

if (cpu\_curr(best\_cpu) == tsk) {

int need\_resched;

send\_now\_idle:

/\*

\* If need\_resched == -1 then we can skip sending

\* the IPI altogether, tsk->need\_resched is

\* actively watched by the idle thread.

\*/

need\_resched = tsk->need\_resched;

tsk->need\_resched = 1;

if ((best\_cpu != this\_cpu) && !need\_resched)

smp\_send\_reschedule(best\_cpu);

return;

}

}

/\*

\* We know that the preferred CPU has a cache-affine current

\* process, lets try to find a new idle CPU for the woken-up

\* process. Select the least recently active idle CPU. (that

\* one will have the least active cache context.) Also find

\* the executing process which has the least priority.

\*/

oldest\_idle = (cycles\_t) -1;

target\_tsk = NULL;

max\_prio = 0;

for (i = 0; i < smp\_num\_cpus; i++) {

cpu = cpu\_logical\_map(i);

if (!can\_schedule(p, cpu))

continue;

tsk = cpu\_curr(cpu);

/\*

\* We use the first available idle CPU. This creates

\* a priority list between idle CPUs, but this is not

\* a problem.

\*/

if (tsk == idle\_task(cpu)) {

#if defined(\_\_i386\_\_) && defined(CONFIG\_SMP)

/\*

\* Check if two siblings are idle in the same

\* physical package. Use them if found.

\*/

if (smp\_num\_siblings == 2) {

if (cpu\_curr(cpu\_sibling\_map[cpu]) ==

idle\_task(cpu\_sibling\_map[cpu])) {

oldest\_idle = last\_schedule(cpu);

target\_tsk = tsk;

break;

}

}

#endif

if (last\_schedule(cpu) < oldest\_idle) {

oldest\_idle = last\_schedule(cpu);

target\_tsk = tsk;

}

} else {

if (oldest\_idle == -1ULL) {

int prio = preemption\_goodness(tsk, p, cpu);

if (prio > max\_prio) {

max\_prio = prio;

target\_tsk = tsk;

}

}

}

}

tsk = target\_tsk;

if (tsk) {

if (oldest\_idle != -1ULL) {

best\_cpu = tsk->processor;

goto send\_now\_idle;

}

tsk->need\_resched = 1;

if (tsk->processor != this\_cpu)

smp\_send\_reschedule(tsk->processor);

}

return;

#else /\* UP \*/

int this\_cpu = smp\_processor\_id();

struct task\_struct \*tsk;

tsk = cpu\_curr(this\_cpu);

if (preemption\_goodness(tsk, p, this\_cpu) > 0)

tsk->need\_resched = 1;

#endif

}

/\*

\* Careful!

\*

\* This has to add the process to the \_beginning\_ of the

\* run-queue, not the end. See the comment about "This is

\* subtle" in the scheduler proper..

\*/

static inline void add\_to\_runqueue(struct task\_struct \* p)

{

list\_add(&p->run\_list, &runqueue\_head);

nr\_running++;

}

static inline void move\_last\_runqueue(struct task\_struct \* p)

{

list\_del(&p->run\_list);

list\_add\_tail(&p->run\_list, &runqueue\_head);

}

static inline void move\_first\_runqueue(struct task\_struct \* p)

{

list\_del(&p->run\_list);

list\_add(&p->run\_list, &runqueue\_head);

}

/\*

\* Wake up a process. Put it on the run-queue if it's not

\* already there. The "current" process is always on the

\* run-queue (except when the actual re-schedule is in

\* progress), and as such you're allowed to do the simpler

\* "current->state = TASK\_RUNNING" to mark yourself runnable

\* without the overhead of this.

\*/

static inline int try\_to\_wake\_up(struct task\_struct \* p, int synchronous)

{

unsigned long flags;

int success = 0;

/\*

\* We want the common case fall through straight, thus the goto.

\*/

spin\_lock\_irqsave(&runqueue\_lock, flags);

p->state = TASK\_RUNNING;

if (task\_on\_runqueue(p))

goto out;

add\_to\_runqueue(p);

if (!synchronous || !(p->cpus\_allowed & (1 << smp\_processor\_id())))

reschedule\_idle(p);

success = 1;

out:

spin\_unlock\_irqrestore(&runqueue\_lock, flags);

return success;

}

inline int wake\_up\_process(struct task\_struct \* p)

{

return try\_to\_wake\_up(p, 0);

}

static void process\_timeout(unsigned long \_\_data)

{

struct task\_struct \* p = (struct task\_struct \*) \_\_data;

wake\_up\_process(p);

}

/\*\*

\* schedule\_timeout - sleep until timeout

\* @timeout: timeout value in jiffies

\*

\* Make the current task sleep until @timeout jiffies have

\* elapsed. The routine will return immediately unless

\* the current task state has been set (see set\_current\_state()).

\*

\* You can set the task state as follows -

\*

\* %TASK\_UNINTERRUPTIBLE - at least @timeout jiffies are guaranteed to

\* pass before the routine returns. The routine will return 0

\*

\* %TASK\_INTERRUPTIBLE - the routine may return early if a signal is

\* delivered to the current task. In this case the remaining time

\* in jiffies will be returned, or 0 if the timer expired in time

\*

\* The current task state is guaranteed to be TASK\_RUNNING when this

\* routine returns.

\*

\* Specifying a @timeout value of %MAX\_SCHEDULE\_TIMEOUT will schedule

\* the CPU away without a bound on the timeout. In this case the return

\* value will be %MAX\_SCHEDULE\_TIMEOUT.

\*

\* In all cases the return value is guaranteed to be non-negative.

\*/

signed long schedule\_timeout(signed long timeout)

{

struct timer\_list timer;

unsigned long expire;

switch (timeout)

{

case MAX\_SCHEDULE\_TIMEOUT:

/\*

\* These two special cases are useful to be comfortable

\* in the caller. Nothing more. We could take

\* MAX\_SCHEDULE\_TIMEOUT from one of the negative value

\* but I' d like to return a valid offset (>=0) to allow

\* the caller to do everything it want with the retval.

\*/

schedule();

goto out;

default:

/\*

\* Another bit of PARANOID. Note that the retval will be

\* 0 since no piece of kernel is supposed to do a check

\* for a negative retval of schedule\_timeout() (since it

\* should never happens anyway). You just have the printk()

\* that will tell you if something is gone wrong and where.

\*/

if (timeout < 0)

{

printk(KERN\_ERR "schedule\_timeout: wrong timeout "

"value %lx from %p\n", timeout,

\_\_builtin\_return\_address(0));

current->state = TASK\_RUNNING;

goto out;

}

}

expire = timeout + jiffies;

init\_timer(&timer);

timer.expires = expire;

timer.data = (unsigned long) current;

timer.function = process\_timeout;

add\_timer(&timer);

schedule();

del\_timer\_sync(&timer);

timeout = expire - jiffies;

out:

return timeout < 0 ? 0 : timeout;

}

/\*

\* schedule\_tail() is getting called from the fork return path. This

\* cleans up all remaining scheduler things, without impacting the

\* common case.

\*/

static inline void \_\_schedule\_tail(struct task\_struct \*prev)

{

#ifdef CONFIG\_SMP

int policy;

/\*

\* prev->policy can be written from here only before `prev'

\* can be scheduled (before setting prev->cpus\_runnable to ~0UL).

\* Of course it must also be read before allowing prev

\* to be rescheduled, but since the write depends on the read

\* to complete, wmb() is enough. (the spin\_lock() acquired

\* before setting cpus\_runnable is not enough because the spin\_lock()

\* common code semantics allows code outside the critical section

\* to enter inside the critical section)

\*/

policy = prev->policy;

prev->policy = policy & ~SCHED\_YIELD;

wmb();

/\*

\* fast path falls through. We have to clear cpus\_runnable before

\* checking prev->state to avoid a wakeup race. Protect against

\* the task exiting early.

\*/

task\_lock(prev);

task\_release\_cpu(prev);

mb();

if (prev->state == TASK\_RUNNING)

goto needs\_resched;

out\_unlock:

task\_unlock(prev); /\* Synchronise here with release\_task() if prev is TASK\_ZOMBIE \*/

return;

/\*

\* Slow path - we 'push' the previous process and

\* reschedule\_idle() will attempt to find a new

\* processor for it. (but it might preempt the

\* current process as well.) We must take the runqueue

\* lock and re-check prev->state to be correct. It might

\* still happen that this process has a preemption

\* 'in progress' already - but this is not a problem and

\* might happen in other circumstances as well.

\*/

needs\_resched:

{

unsigned long flags;

/\*

\* Avoid taking the runqueue lock in cases where

\* no preemption-check is necessery:

\*/

if ((prev == idle\_task(smp\_processor\_id())) ||

(policy & SCHED\_YIELD))

goto out\_unlock;

spin\_lock\_irqsave(&runqueue\_lock, flags);

if ((prev->state == TASK\_RUNNING) && !task\_has\_cpu(prev))

reschedule\_idle(prev);

spin\_unlock\_irqrestore(&runqueue\_lock, flags);

goto out\_unlock;

}

#else

prev->policy &= ~SCHED\_YIELD;

#endif /\* CONFIG\_SMP \*/

}

asmlinkage void schedule\_tail(struct task\_struct \*prev)

{

\_\_schedule\_tail(prev);

}

/\*

\* 'schedule()' is the scheduler function. It's a very simple and nice

\* scheduler: it's not perfect, but certainly works for most things.

\*

\* The goto is "interesting".

\*

\* NOTE!! Task 0 is the 'idle' task, which gets called when no other

\* tasks can run. It can not be killed, and it cannot sleep. The 'state'

\* information in task[0] is never used.

\*/

asmlinkage void schedule(void)

{

struct schedule\_data \* sched\_data;

struct task\_struct \*prev, \*next, \*p;

struct list\_head \*tmp;

int this\_cpu, c;

spin\_lock\_prefetch(&runqueue\_lock);

if (!current->active\_mm) BUG();

need\_resched\_back:

prev = current;

this\_cpu = prev->processor;

if (unlikely(in\_interrupt())) {

printk("Scheduling in interrupt\n");

BUG();

}

release\_kernel\_lock(prev, this\_cpu);

/\*

\* 'sched\_data' is protected by the fact that we can run

\* only one process per CPU.

\*/

sched\_data = & aligned\_data[this\_cpu].schedule\_data;

spin\_lock\_irq(&runqueue\_lock);

/\* move an exhausted RR process to be last.. \*/

if (unlikely(prev->policy == SCHED\_RR))

if (!prev->counter) {

prev->counter = NICE\_TO\_TICKS(prev->nice);

move\_last\_runqueue(prev);

}

switch (prev->state) {

case TASK\_INTERRUPTIBLE:

if (signal\_pending(prev)) {

prev->state = TASK\_RUNNING;

break;

}

default:

del\_from\_runqueue(prev);

case TASK\_RUNNING:;

}

prev->need\_resched = 0;

/\*

\* this is the scheduler proper:

\*/

repeat\_schedule:

/\*

\* Default process to select..

\*/

next = idle\_task(this\_cpu);

c = -1000;

list\_for\_each(tmp, &runqueue\_head) {

p = list\_entry(tmp, struct task\_struct, run\_list);

if (can\_schedule(p, this\_cpu)) {

int weight = goodness(p, this\_cpu, prev->active\_mm);

if (weight > c)

c = weight, next = p;

}

}

/\* Do we need to re-calculate counters? \*/

if (unlikely(!c)) {

struct task\_struct \*p;

spin\_unlock\_irq(&runqueue\_lock);

read\_lock(&tasklist\_lock);

for\_each\_task(p)

p->counter = (p->counter >> 1) + NICE\_TO\_TICKS(p->nice);

read\_unlock(&tasklist\_lock);

spin\_lock\_irq(&runqueue\_lock);

goto repeat\_schedule;

}

/\*

\* from this point on nothing can prevent us from

\* switching to the next task, save this fact in

\* sched\_data.

\*/

sched\_data->curr = next;

task\_set\_cpu(next, this\_cpu);

spin\_unlock\_irq(&runqueue\_lock);

if (unlikely(prev == next)) {

/\* We won't go through the normal tail, so do this by hand \*/

prev->policy &= ~SCHED\_YIELD;

goto same\_process;

}

#ifdef CONFIG\_SMP

/\*

\* maintain the per-process 'last schedule' value.

\* (this has to be recalculated even if we reschedule to

\* the same process) Currently this is only used on SMP,

\* and it's approximate, so we do not have to maintain

\* it while holding the runqueue spinlock.

\*/

sched\_data->last\_schedule = get\_cycles();

/\*

\* We drop the scheduler lock early (it's a global spinlock),

\* thus we have to lock the previous process from getting

\* rescheduled during switch\_to().

\*/

#endif /\* CONFIG\_SMP \*/

kstat.context\_swtch++;

/\*

\* there are 3 processes which are affected by a context switch:

\*

\* prev == .... ==> (last => next)

\*

\* It's the 'much more previous' 'prev' that is on next's stack,

\* but prev is set to (the just run) 'last' process by switch\_to().

\* This might sound slightly confusing but makes tons of sense.

\*/

prepare\_to\_switch();

{

struct mm\_struct \*mm = next->mm;

struct mm\_struct \*oldmm = prev->active\_mm;

if (!mm) {

if (next->active\_mm) BUG();

next->active\_mm = oldmm;

atomic\_inc(&oldmm->mm\_count);

enter\_lazy\_tlb(oldmm, next, this\_cpu);

} else {

if (next->active\_mm != mm) BUG();

switch\_mm(oldmm, mm, next, this\_cpu);

}

if (!prev->mm) {

prev->active\_mm = NULL;

mmdrop(oldmm);

}

}

/\*

\* This just switches the register state and the

\* stack.

\*/

switch\_to(prev, next, prev);

\_\_schedule\_tail(prev);

same\_process:

reacquire\_kernel\_lock(current);

if (current->need\_resched)

goto need\_resched\_back;

return;

}

/\*

\* The core wakeup function. Non-exclusive wakeups (nr\_exclusive == 0) just wake everything

\* up. If it's an exclusive wakeup (nr\_exclusive == small +ve number) then we wake all the

\* non-exclusive tasks and one exclusive task.

\*

\* There are circumstances in which we can try to wake a task which has already

\* started to run but is not in state TASK\_RUNNING. try\_to\_wake\_up() returns zero

\* in this (rare) case, and we handle it by contonuing to scan the queue.

\*/

static inline void \_\_wake\_up\_common (wait\_queue\_head\_t \*q, unsigned int mode,

int nr\_exclusive, const int sync)

{

struct list\_head \*tmp;

struct task\_struct \*p;

CHECK\_MAGIC\_WQHEAD(q);

WQ\_CHECK\_LIST\_HEAD(&q->task\_list);

list\_for\_each(tmp,&q->task\_list) {

unsigned int state;

wait\_queue\_t \*curr = list\_entry(tmp, wait\_queue\_t, task\_list);

CHECK\_MAGIC(curr->\_\_magic);

p = curr->task;

state = p->state;

if (state & mode) {

WQ\_NOTE\_WAKER(curr);

if (try\_to\_wake\_up(p, sync) && (curr->flags&WQ\_FLAG\_EXCLUSIVE) && !--nr\_exclusive)

break;

}

}

}

void \_\_wake\_up(wait\_queue\_head\_t \*q, unsigned int mode, int nr)

{

if (q) {

unsigned long flags;

wq\_read\_lock\_irqsave(&q->lock, flags);

\_\_wake\_up\_common(q, mode, nr, 0);

wq\_read\_unlock\_irqrestore(&q->lock, flags);

}

}

void \_\_wake\_up\_sync(wait\_queue\_head\_t \*q, unsigned int mode, int nr)

{

if (q) {

unsigned long flags;

wq\_read\_lock\_irqsave(&q->lock, flags);

\_\_wake\_up\_common(q, mode, nr, 1);

wq\_read\_unlock\_irqrestore(&q->lock, flags);

}

}

void complete(struct completion \*x)

{

unsigned long flags;

spin\_lock\_irqsave(&x->wait.lock, flags);

x->done++;

\_\_wake\_up\_common(&x->wait, TASK\_UNINTERRUPTIBLE | TASK\_INTERRUPTIBLE, 1, 0);

spin\_unlock\_irqrestore(&x->wait.lock, flags);

}

void wait\_for\_completion(struct completion \*x)

{

spin\_lock\_irq(&x->wait.lock);

if (!x->done) {

DECLARE\_WAITQUEUE(wait, current);

wait.flags |= WQ\_FLAG\_EXCLUSIVE;

\_\_add\_wait\_queue\_tail(&x->wait, &wait);

do {

\_\_set\_current\_state(TASK\_UNINTERRUPTIBLE);

spin\_unlock\_irq(&x->wait.lock);

schedule();

spin\_lock\_irq(&x->wait.lock);

} while (!x->done);

\_\_remove\_wait\_queue(&x->wait, &wait);

}

x->done--;

spin\_unlock\_irq(&x->wait.lock);

}

#define SLEEP\_ON\_VAR \

unsigned long flags; \

wait\_queue\_t wait; \

init\_waitqueue\_entry(&wait, current);

#define SLEEP\_ON\_HEAD \

wq\_write\_lock\_irqsave(&q->lock,flags); \

\_\_add\_wait\_queue(q, &wait); \

wq\_write\_unlock(&q->lock);

#define SLEEP\_ON\_TAIL \

wq\_write\_lock\_irq(&q->lock); \

\_\_remove\_wait\_queue(q, &wait); \

wq\_write\_unlock\_irqrestore(&q->lock,flags);

void interruptible\_sleep\_on(wait\_queue\_head\_t \*q)

{

SLEEP\_ON\_VAR

current->state = TASK\_INTERRUPTIBLE;

SLEEP\_ON\_HEAD

schedule();

SLEEP\_ON\_TAIL

}

long interruptible\_sleep\_on\_timeout(wait\_queue\_head\_t \*q, long timeout)

{

SLEEP\_ON\_VAR

current->state = TASK\_INTERRUPTIBLE;

SLEEP\_ON\_HEAD

timeout = schedule\_timeout(timeout);

SLEEP\_ON\_TAIL

return timeout;

}

void sleep\_on(wait\_queue\_head\_t \*q)

{

SLEEP\_ON\_VAR

current->state = TASK\_UNINTERRUPTIBLE;

SLEEP\_ON\_HEAD

schedule();

SLEEP\_ON\_TAIL

}

long sleep\_on\_timeout(wait\_queue\_head\_t \*q, long timeout)

{

SLEEP\_ON\_VAR

current->state = TASK\_UNINTERRUPTIBLE;

SLEEP\_ON\_HEAD

timeout = schedule\_timeout(timeout);

SLEEP\_ON\_TAIL

return timeout;

}

void scheduling\_functions\_end\_here(void) { }

#ifndef \_\_alpha\_\_

/\*

\* This has been replaced by sys\_setpriority. Maybe it should be

\* moved into the arch dependent tree for those ports that require

\* it for backward compatibility?

\*/

asmlinkage long sys\_nice(int increment)

{

long newprio;

/\*

\* Setpriority might change our priority at the same moment.

\* We don't have to worry. Conceptually one call occurs first

\* and we have a single winner.

\*/

if (increment < 0) {

if (!capable(CAP\_SYS\_NICE))

return -EPERM;

if (increment < -40)

increment = -40;

}

if (increment > 40)

increment = 40;

newprio = current->nice + increment;

if (newprio < -20)

newprio = -20;

if (newprio > 19)

newprio = 19;

current->nice = newprio;

return 0;

}

#endif

static inline struct task\_struct \*find\_process\_by\_pid(pid\_t pid)

{

struct task\_struct \*tsk = current;

if (pid)

tsk = find\_task\_by\_pid(pid);

return tsk;

}

static int setscheduler(pid\_t pid, int policy,

struct sched\_param \*param)

{

struct sched\_param lp;

struct task\_struct \*p;

int retval;

retval = -EINVAL;

if (!param || pid < 0)

goto out\_nounlock;

retval = -EFAULT;

if (copy\_from\_user(&lp, param, sizeof(struct sched\_param)))

goto out\_nounlock;

/\*

\* We play safe to avoid deadlocks.

\*/

read\_lock\_irq(&tasklist\_lock);

spin\_lock(&runqueue\_lock);

p = find\_process\_by\_pid(pid);

retval = -ESRCH;

if (!p)

goto out\_unlock;

if (policy < 0)

policy = p->policy;

else {

retval = -EINVAL;

if (policy != SCHED\_FIFO && policy != SCHED\_RR &&

policy != SCHED\_OTHER)

goto out\_unlock;

}

/\*

\* Valid priorities for SCHED\_FIFO and SCHED\_RR are 1..99, valid

\* priority for SCHED\_OTHER is 0.

\*/

retval = -EINVAL;

if (lp.sched\_priority < 0 || lp.sched\_priority > 99)

goto out\_unlock;

if ((policy == SCHED\_OTHER) != (lp.sched\_priority == 0))

goto out\_unlock;

retval = -EPERM;

if ((policy == SCHED\_FIFO || policy == SCHED\_RR) &&

!capable(CAP\_SYS\_NICE))

goto out\_unlock;

if ((current->euid != p->euid) && (current->euid != p->uid) &&

!capable(CAP\_SYS\_NICE))

goto out\_unlock;

retval = 0;

p->policy = policy;

p->rt\_priority = lp.sched\_priority;

if (task\_on\_runqueue(p))

move\_first\_runqueue(p);

current->need\_resched = 1;

out\_unlock:

spin\_unlock(&runqueue\_lock);

read\_unlock\_irq(&tasklist\_lock);

out\_nounlock:

return retval;

}

asmlinkage long sys\_sched\_setscheduler(pid\_t pid, int policy,

struct sched\_param \*param)

{

return setscheduler(pid, policy, param);

}

asmlinkage long sys\_sched\_setparam(pid\_t pid, struct sched\_param \*param)

{

return setscheduler(pid, -1, param);

}

asmlinkage long sys\_sched\_getscheduler(pid\_t pid)

{

struct task\_struct \*p;

int retval;

retval = -EINVAL;

if (pid < 0)

goto out\_nounlock;

retval = -ESRCH;

read\_lock(&tasklist\_lock);

p = find\_process\_by\_pid(pid);

if (p)

retval = p->policy & ~SCHED\_YIELD;

read\_unlock(&tasklist\_lock);

out\_nounlock:

return retval;

}

asmlinkage long sys\_sched\_getparam(pid\_t pid, struct sched\_param \*param)

{

struct task\_struct \*p;

struct sched\_param lp;

int retval;

retval = -EINVAL;

if (!param || pid < 0)

goto out\_nounlock;

read\_lock(&tasklist\_lock);

p = find\_process\_by\_pid(pid);

retval = -ESRCH;

if (!p)

goto out\_unlock;

lp.sched\_priority = p->rt\_priority;

read\_unlock(&tasklist\_lock);

/\*

\* This one might sleep, we cannot do it with a spinlock held ...

\*/

retval = copy\_to\_user(param, &lp, sizeof(\*param)) ? -EFAULT : 0;

out\_nounlock:

return retval;

out\_unlock:

read\_unlock(&tasklist\_lock);

return retval;

}

asmlinkage long sys\_sched\_yield(void)

{

/\*

\* Trick. sched\_yield() first counts the number of truly

\* 'pending' runnable processes, then returns if it's

\* only the current processes. (This test does not have

\* to be atomic.) In threaded applications this optimization

\* gets triggered quite often.

\*/

int nr\_pending = nr\_running;

#if CONFIG\_SMP

int i;

// Subtract non-idle processes running on other CPUs.

for (i = 0; i < smp\_num\_cpus; i++) {

int cpu = cpu\_logical\_map(i);

if (aligned\_data[cpu].schedule\_data.curr != idle\_task(cpu))

nr\_pending--;

}

#else

// on UP this process is on the runqueue as well

nr\_pending--;

#endif

if (nr\_pending) {

/\*

\* This process can only be rescheduled by us,

\* so this is safe without any locking.

\*/

if (current->policy == SCHED\_OTHER)

current->policy |= SCHED\_YIELD;

current->need\_resched = 1;

spin\_lock\_irq(&runqueue\_lock);

move\_last\_runqueue(current);

spin\_unlock\_irq(&runqueue\_lock);

}

return 0;

}

asmlinkage long sys\_sched\_get\_priority\_max(int policy)

{

int ret = -EINVAL;

switch (policy) {

case SCHED\_FIFO:

case SCHED\_RR:

ret = 99;

break;

case SCHED\_OTHER:

ret = 0;

break;

}

return ret;

}

asmlinkage long sys\_sched\_get\_priority\_min(int policy)

{

int ret = -EINVAL;

switch (policy) {

case SCHED\_FIFO:

case SCHED\_RR:

ret = 1;

break;

case SCHED\_OTHER:

ret = 0;

}

return ret;

}

asmlinkage long sys\_sched\_rr\_get\_interval(pid\_t pid, struct timespec \*interval)

{

struct timespec t;

struct task\_struct \*p;

int retval = -EINVAL;

if (pid < 0)

goto out\_nounlock;

retval = -ESRCH;

read\_lock(&tasklist\_lock);

p = find\_process\_by\_pid(pid);

if (p)

jiffies\_to\_timespec(p->policy & SCHED\_FIFO ? 0 : NICE\_TO\_TICKS(p->nice),

&t);

read\_unlock(&tasklist\_lock);

if (p)

retval = copy\_to\_user(interval, &t, sizeof(t)) ? -EFAULT : 0;

out\_nounlock:

return retval;

}

static void show\_task(struct task\_struct \* p)

{

unsigned long free = 0;

int state;

static const char \* stat\_nam[] = { "R", "S", "D", "Z", "T", "W" };

printk("%-13.13s ", p->comm);

state = p->state ? ffz(~p->state) + 1 : 0;

if (((unsigned) state) < sizeof(stat\_nam)/sizeof(char \*))

printk(stat\_nam[state]);

else

printk(" ");

#if (BITS\_PER\_LONG == 32)

if (p == current)

printk(" current ");

else

printk(" %08lX ", thread\_saved\_pc(&p->thread));

#else

if (p == current)

printk(" current task ");

else

printk(" %016lx ", thread\_saved\_pc(&p->thread));

#endif

{

unsigned long \* n = (unsigned long \*) (p+1);

while (!\*n)

n++;

free = (unsigned long) n - (unsigned long)(p+1);

}

printk("%5lu %5d %6d ", free, p->pid, p->p\_pptr->pid);

if (p->p\_cptr)

printk("%5d ", p->p\_cptr->pid);

else

printk(" ");

if (p->p\_ysptr)

printk("%7d", p->p\_ysptr->pid);

else

printk(" ");

if (p->p\_osptr)

printk(" %5d", p->p\_osptr->pid);

else

printk(" ");

if (!p->mm)

printk(" (L-TLB)\n");

else

printk(" (NOTLB)\n");

{

extern void show\_trace\_task(struct task\_struct \*tsk);

show\_trace\_task(p);

}

}

char \* render\_sigset\_t(sigset\_t \*set, char \*buffer)

{

int i = \_NSIG, x;

do {

i -= 4, x = 0;

if (sigismember(set, i+1)) x |= 1;

if (sigismember(set, i+2)) x |= 2;

if (sigismember(set, i+3)) x |= 4;

if (sigismember(set, i+4)) x |= 8;

\*buffer++ = (x < 10 ? '0' : 'a' - 10) + x;

} while (i >= 4);

\*buffer = 0;

return buffer;

}

void show\_state(void)

{

struct task\_struct \*p;

#if (BITS\_PER\_LONG == 32)

printk("\n"

" free sibling\n");

printk(" task PC stack pid father child younger older\n");

#else

printk("\n"

" free sibling\n");

printk(" task PC stack pid father child younger older\n");

#endif

read\_lock(&tasklist\_lock);

for\_each\_task(p) {

/\*

\* reset the NMI-timeout, listing all files on a slow

\* console might take alot of time:

\*/

touch\_nmi\_watchdog();

show\_task(p);

}

read\_unlock(&tasklist\_lock);

}

/\*\*

\* reparent\_to\_init() - Reparent the calling kernel thread to the init task.

\*

\* If a kernel thread is launched as a result of a system call, or if

\* it ever exits, it should generally reparent itself to init so that

\* it is correctly cleaned up on exit.

\*

\* The various task state such as scheduling policy and priority may have

\* been inherited fro a user process, so we reset them to sane values here.

\*

\* NOTE that reparent\_to\_init() gives the caller full capabilities.

\*/

void reparent\_to\_init(void)

{

struct task\_struct \*this\_task = current;

write\_lock\_irq(&tasklist\_lock);

/\* Reparent to init \*/

REMOVE\_LINKS(this\_task);

this\_task->p\_pptr = child\_reaper;

this\_task->p\_opptr = child\_reaper;

SET\_LINKS(this\_task);

/\* Set the exit signal to SIGCHLD so we signal init on exit \*/

this\_task->exit\_signal = SIGCHLD;

/\* We also take the runqueue\_lock while altering task fields

\* which affect scheduling decisions \*/

spin\_lock(&runqueue\_lock);

this\_task->ptrace = 0;

this\_task->nice = DEF\_NICE;

this\_task->policy = SCHED\_OTHER;

/\* cpus\_allowed? \*/

/\* rt\_priority? \*/

/\* signals? \*/

this\_task->cap\_effective = CAP\_INIT\_EFF\_SET;

this\_task->cap\_inheritable = CAP\_INIT\_INH\_SET;

this\_task->cap\_permitted = CAP\_FULL\_SET;

this\_task->keep\_capabilities = 0;

memcpy(this\_task->rlim, init\_task.rlim, sizeof(\*(this\_task->rlim)));

this\_task->user = INIT\_USER;

spin\_unlock(&runqueue\_lock);

write\_unlock\_irq(&tasklist\_lock);

}

/\*

\* Put all the gunge required to become a kernel thread without

\* attached user resources in one place where it belongs.

\*/

void daemonize(void)

{

struct fs\_struct \*fs;

/\*

\* If we were started as result of loading a module, close all of the

\* user space pages. We don't need them, and if we didn't close them

\* they would be locked into memory.

\*/

exit\_mm(current);

current->session = 1;

current->pgrp = 1;

current->tty = NULL;

/\* Become as one with the init task \*/

exit\_fs(current); /\* current->fs->count--; \*/

fs = init\_task.fs;

current->fs = fs;

atomic\_inc(&fs->count);

exit\_files(current);

current->files = init\_task.files;

atomic\_inc(&current->files->count);

}

extern unsigned long wait\_init\_idle;

void \_\_init init\_idle(void)

{

struct schedule\_data \* sched\_data;

sched\_data = &aligned\_data[smp\_processor\_id()].schedule\_data;

if (current != &init\_task && task\_on\_runqueue(current)) {

printk("UGH! (%d:%d) was on the runqueue, removing.\n",

smp\_processor\_id(), current->pid);

del\_from\_runqueue(current);

}

sched\_data->curr = current;

sched\_data->last\_schedule = get\_cycles();

clear\_bit(current->processor, &wait\_init\_idle);

}

extern void init\_timervecs (void);

void \_\_init sched\_init(void)

{

/\*

\* We have to do a little magic to get the first

\* process right in SMP mode.

\*/

int cpu = smp\_processor\_id();

int nr;

init\_task.processor = cpu;

for(nr = 0; nr < PIDHASH\_SZ; nr++)

pidhash[nr] = NULL;

init\_timervecs();

init\_bh(TIMER\_BH, timer\_bh);

init\_bh(TQUEUE\_BH, tqueue\_bh);

init\_bh(IMMEDIATE\_BH, immediate\_bh);

/\*

\* The boot idle thread does lazy MMU switching as well:

\*/

atomic\_inc(&init\_mm.mm\_count);

enter\_lazy\_tlb(&init\_mm, current, cpu);

}