关键数据结构：

//dmpbox对应的是整个DPG，一个DPG只有一个dmpbox

struct dmpbox {

uint32\_t magic; /\* Constant to test for validity \*/

/\* Globals \*/

spinlock\_t lock; /\* Guards all reads and writes of a deterministic

\* box except for local accesses to a dmptaskinfo \*/

unsigned mode; /\* execution mode \*/

unsigned shimmed; /\* 1 if there are shim threads, 0 otherwise \*/

unsigned ntasks; /\* total # of tasks in this box \*/

unsigned next\_id; /\* next deterministic pid to assign \*/

/\* Scheduling: shared \*/

uint64\_t logical\_time; /\* even:parallel-mode, odd:serial-mode \*/

int quantum\_size; /\* current quantum ticks budget \*/

int quantum\_cpu\_scaling; /\* cpu.local\_quantum\_size = box.quantum\_size \* box.quantum\_cpu\_scaling \*/

unsigned nrunnable; /\* total # of currently runnable tasks \*/

struct list\_head all; /\* all tasks \*/

struct list\_head activating; /\* all tasks with state == ACTIVATING \*/

/\* Scheduling: DMP\_SERIAL \*/

struct list\_head roundrobin; /\* all tasks have state == RUNNING|READY|ENDQUANTUM \*/

/\* this is essentially the 'runnable' queue \*/

/\* Scheduling: DMP\_MOT \*/

unsigned short runnable\_cpus; /\* # of cpus with cpu->first != NULL \*/

unsigned short pwork\_cpus; /\* # of cpus that have parallel work left \*/

struct dmpcpu cpus[NR\_CPUS];

struct list\_head processgroups; /\* list of dmpschedgroups (see above) \*/

struct list\_head zombiegroups; /\* list of dmpschedgroups (see above) \*/

/\*

\* Shim notification

\* List of dmpqbarriers, sorted in by increasing logical\_time.

DPG中待处理的的barrier集合，按照逻辑时间增加的顺序排列

\*/

struct list\_head pending\_quantum\_barriers;

/\* Misc \*/

struct list\_head handoffs;

struct dmpmot\_bucket mot[DMP\_MOT\_HASH\_BUCKETS];

struct dmpboxstats stats;

/\* Debugging \*/

int \_\_box\_bugging; /\* Set if the box is executing a DMP\_BUG\_ON \*/

};

//dmptaskinfo结构在task\_struct结构中，task\_struct结构用来存放每个线程的信息，每个线程对应一个唯一task\_struct，每个task\_struct对应唯一一个dmptaskinfo

struct dmptaskinfo {

uint32\_t magic; /\* Constant to test for validity \*/

unsigned dmp\_id; /\* deterministic pid, relative to this group \*/

int state; /\* DMP\_TASK state \*/

int quantum\_ticks; /\* this quantum: accumulated ticks (including penalties) \*/

int context; /\* current execution context: SYS\_ or DMP\_CTX\_ \*/

/\* Where do I belong? \*/

struct dmpbox\* box; //对应整个dmp线程组

struct dmpschedgroup\* schedgroup;

struct list\_head box\_list; /\* link for box->all \*/

struct list\_head serial\_list; /\* link for serial-mode lists

- for DMP\_SERIAL: box->roundrobin

- for DMP\_MOT: cpu->roundrobin \*/

struct list\_head state\_list; /\* link for per-state lists

- for DMP\_SERIAL: box->activating

- for DMP\_MOT: schedgroup->runnable or box->activating \*/

/\* Shim stuff \*/

struct task\_struct\* shim; /\* shimmee: thread shimming us

shimmer: thread being shimmed \*/

struct dmpshiminfo shiminfo;

/\* What am I serializing on? need at least 3 of these \*/

struct dmpser\_object serobjects[4];

int did\_check\_serobjects; /\* if unset, the task should recapture ownership of serobjects \*/

/\* State used by optimizations \*/

struct list\_head handoff\_list;

int smser\_first\_fault;

int smser\_mot\_faults;

int smser\_max\_faults;

int smser\_end\_quantum;

/\* Debug stuff \*/

int pgflt\_serial\_ct;

uint64\_t saved\_box\_mode;

uint64\_t saved\_box\_round;

/\* Task statistics \*/

struct dmptaskstats stats;

};

## Shim相关的系统调用函数

Shim主要是通过一个shim线程和一个确定线程一一对应来工作的，sys\_dmp\_shim\_attach()函数将一个确定线程绑定到当前正在执行的shim线程上，sys\_dmp\_shim\_trace()函数用来接收下一个要执行的事件信息，并根据事件类型的不同执行相应的操作。sys\_dmp\_shim\_set\_barrier()函数根据其参数的不同，设置在某个逻辑时间点让某个线程挂起。

\* Syscall API for shim tasks

首先检测pid对应的线程是否是确定的，如果是则将其attach到当前线程。

shim和pid相互绑定，当pid对应的线程执行不确定的操作时，跳转到shim执行，

shim执行完后再返回到pid

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

asmlinkage long sys\_dmp\_shim\_attach(pid\_t pid)

{

struct task\_struct \*child;

long ret = 0, flags;

struct dmpbox \*box;

/\*

\* This task is 'put' by the shim thread when it receives an DMP\_SHIM\_EXIT

\* event from the DMP thread.

\*/

child = ptrace\_get\_task\_struct(pid);

if (IS\_ERR(child)) {

ret = PTR\_ERR(child);

goto out;

}

/\*

\* TODO: Find a cleaner way to do this

\* Wait for child to become a DMP thread

\*/

DMPDBG(SHIM, current,

"shim\_attach: pid=%d waiting for thread pid=%d,%d to \"determinize\"...\n",

current->pid, child->pid, child->dmp.dmp\_id);

while (!IS\_DMP(child))

schedule();

DMPDBG(SHIM, current,

"shim\_attach: pid=%d attaching to pid=%d,%d\n",

current->pid, child->pid, child->dmp.dmp\_id);

box = child->dmp.box;

BUG\_ON(!box);

/\* Now attach \*/

spin\_lock\_irqsave(&box->lock, flags);//加锁可能是为了防止线程切换

if (!box->shimmed) {

DMPDBG(SHIM, current, "shim\_attach: trying to shim non-shimmed box\n");

ret = -EINVAL;

} else if (child->dmp.shim != NULL) {

DMPDBG(SHIM, current, "shim\_attach: trying to shim an already shimmed thread\n");

ret = -EALREADY;

} else {

child->dmp.shim = current;

current->dmp.shim = child;

}

spin\_unlock\_irqrestore(&box->lock, flags);

if (dmp\_opts.optims & (PIN\_DMP|PIN\_SHIM)) {

sched\_setaffinity(current->pid, child->dmp.schedgroup->cpu->cpumask);

}

out:

return ret;

}

/\*

\* To be called by a shim thread to receive the next event from

\* the caller's corresponding dmp thread.

shim线程收到下一个要执行的事件信息(为什么是shim接收?)

用事件信息填充event，根据事件类型不同执行相应操作

\*/

asmlinkage long sys\_dmp\_shim\_trace(struct dmp\_shim\_event \_\_user \*event)

{

struct task\_struct \*dmptask = current->dmp.shim;

struct semaphore \*dmpsem = &dmptask->dmp.shiminfo.sem;

struct task\_struct \*shimtask = current;

struct semaphore \*shimsem = &shimtask->dmp.shiminfo.sem;

long ret = 0;

if (!dmptask)

return -EINVAL;

if (!access\_ok(VERIFY\_WRITE, event, sizeof \*event))

return -EACCES;

DMP\_BUG\_ON(shimtask, !IS\_DMP(dmptask));

/\* Don't clear these: they need to be preserved across a dmp\_shim\_sleep() notification \*/

// TODO: revisit when we deal with signals

//shimtask->dmp.shiminfo.regs = NULL;

/\* Let the dmptask continue until the next event \*/

/\* If we put the task to sleep, this is unnecessary \*/

if (!shimtask->dmp.shiminfo.need\_to\_sleep) {

DMPDBG(SHIM, shimtask, "signaling pid=%d,%d (on: %p, ct: %d, empty? %d)\n", dmptask->pid, dmptask->dmp.dmp\_id, dmpsem, atomic\_read(&dmpsem->count), !waitqueue\_active(&dmpsem->wait));

up(dmpsem);//将dmpsem置位up，即将控制权交给shim

} else {

shimtask->dmp.shiminfo.need\_to\_sleep = 0;

}

/\* Sleep and wait for the next event to happen \*/

DMPDBG(SHIM, shimtask, "waiting (on: %p, ct: %d, empty? %d)\n", shimsem, atomic\_read(&shimsem->count), !waitqueue\_active(&shimsem->wait));

down(shimsem);

/\* Return this event \*/

const uint8\_t event\_type = shimtask->dmp.shiminfo.event\_type;

const uint64\_t logical\_time = shimtask->dmp.shiminfo.logical\_time;

const uint32\_t barrier\_type = shimtask->dmp.shiminfo.barrier\_type;

const uint32\_t prev\_sleep = shimtask->dmp.shiminfo.saved\_sleep\_state;

const uint64\_t qticks = shimtask->dmp.shiminfo.qticks; /\* For debugging \*/

struct pt\_regs \*regs = shimtask->dmp.shiminfo.regs;

DMPDBG(SHIM, shimtask, "event @%lld.%lld %d %p\n", logical\_time/2, logical\_time%2, event\_type, regs);

\_\_put\_user(event\_type, &event->event\_type);

\_\_put\_user(logical\_time, &event->logical\_time);

\_\_put\_user(prev\_sleep, &event->prev\_sleep);

\_\_put\_user(qticks, &event->qticks);

switch (event\_type) {

case DMP\_SHIM\_SIGNAL:

DMPDBG(SHIM, shimtask, "SIGNAL signr:%d rax:%#lx\n", shimtask->dmp.shiminfo.siginfo->si\_signo, regs->orig\_rax);

DMP\_BUG\_ON(shimtask, !regs);

DMP\_BUG\_ON(shimtask, !shimtask->dmp.shiminfo.siginfo);

copy\_siginfo\_to\_user(&event->siginfo, shimtask->dmp.shiminfo.siginfo);

copy\_regs\_to\_user(dmptask, regs, &event->regs);

break;

case DMP\_SHIM\_SYSCALL\_ENTER:

case DMP\_SHIM\_SYSCALL\_LEAVE:

case DMP\_SHIM\_CALL:

case DMP\_SHIM\_VFORK\_WAIT:

case DMP\_SHIM\_RDTSC:

DMP\_BUG\_ON(shimtask, !regs);

copy\_regs\_to\_user(dmptask, regs, &event->regs);

DMPDBG(SHIM, shimtask, "SYSCALL rax:0x%lx\n", regs->orig\_rax);

break;

case DMP\_SHIM\_BARRIER:

\_\_put\_user(barrier\_type, &event->barrier\_type);

break;

case DMP\_SHIM\_EXIT:

/\* Release our hold on the task so it can be reclaimed \*/

put\_task\_struct(dmptask);

shimtask->dmp.shim = NULL;

break;

default:

DMPERR(dmptask, "bad event: 0x%x\n", event\_type);

ret = -EBADMSG;

break;

}

/\*

unsigned int flags;

spin\_lock\_irqsave(&dmptask->dmp.box->lock, flags);

DMPDBG(VMDBG, dmptask, "VMDBG lpid: %d, cpu: %d\n", dmptask->pid, task\_cpu(dmptask));

//DMPDBG(VMDBG, dmptask, "VMDBG stack: %p\n", dmptask->stack);

//DMPDBG(VMDBG, dmptask, "VMDBG cpu: %d\n", task\_cpu(dmptask));

dmp\_verify\_task(dmptask);

spin\_unlock\_irqrestore(&dmptask->dmp.box->lock, flags);

\*/

return ret;

}

/\*

\* To be called by a shim thread to register a barrier on a dmp thread.

\* Return the absolute logical\_time at which the barrier was registered.

\*

\* Type is one of the following:

\* SHIM\_BARRIER\_FLAG\_IO

\* SHIM\_BARRIER\_FLAG\_SYSCALL

\* SHIM\_BARRIER\_FLAG\_DISTQ

whence和val共同决定barrier发生的时间，其中whence决定其类型，

若发生时间为当前时间加上一个偏移量时，val表示偏移量

\*/

asmlinkage long sys\_dmp\_shim\_set\_barrier(pid\_t pid, uint16\_t type, int whence, uint64\_t val)

{

struct task\_struct \*dmptask = current->dmp.shim;

struct task\_struct \*shimtask = current;

struct dmpbox \*box = NULL;

struct dmpqbarrier \*bar = NULL, \*cur = NULL;

unsigned int flags, needs\_serial;

uint64\_t when;

long ret = 0;

if (pid != 0)

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

if (!dmptask)

return -EINVAL;

if (type >= SHIM\_BARRIER\_FLAG\_MAX)

return -EINVAL;

DMP\_BUG\_ON(shimtask, !IS\_DMP(dmptask));

box = dmptask->dmp.box;

spin\_lock\_irqsave(&box->lock, flags);

needs\_serial = 0;

//根据whence类型的不同，设置在不同的逻辑时间点增加barrier

switch (whence) {

case SHIM\_BARRIER\_NEXT\_MODE:

/\* Next mode \*/

when = box->logical\_time + 1;

break;

case SHIM\_BARRIER\_NEXT\_SERIAL:

/\* Next serial mode \*/

if (IS\_DMP\_SERIAL\_MODE(box)) {

when = box->logical\_time + 2;

} else {

when = box->logical\_time + 1;

}

needs\_serial = 1;

break;

case SHIM\_BARRIER\_OFFSET\_PARALLEL:

/\* Advance N quantums, in parallel mode \*/

when = ((box->logical\_time / 2) + val) \* 2;

break;

case SHIM\_BARRIER\_OFFSET\_SERIAL:

/\* Advance N quantums, in serial mode \*/

when = ((box->logical\_time / 2) + val) \* 2 + 1;

needs\_serial = 1;

break;

case SHIM\_BARRIER\_FIXED\_TIME:

when = val;

needs\_serial = (when % 2 == 1);

break;

default:

ret = -EINVAL;

goto out\_unlock;

}

DMPDBG(SHIM, shimtask, "shim\_set\_barrier(@%llu.%llu), %llu\n", when/2, when%2, when);

/\*

\* Do not allow setting a barrier in the past

\* The current time is "in the past" because the mode transition has already passed.

\*/

if (when <= box->logical\_time) {

DMPDBG(SHIM, shimtask, "failed! (in the past: %llu)\n", when);

ret = -EINVAL;

goto out\_unlock;

}

/\*

\* First see if a barrier for this thread already exists for the given

\* quantum; if so, we can reuse it if one of the shim events is a distq

\* barrier.

\*/

DMPDBG(SHIM, shimtask, "Process %p checking for existing barriers for %p to reuse:\n", shimtask, dmptask);

list\_for\_each\_entry(bar, &box->pending\_quantum\_barriers, qlist) {

DMPDBG(SHIM, shimtask, " @%llu.%llu (%llu) p:%p type:%u\n", bar->logical\_time/2, bar->logical\_time%2, bar->logical\_time, bar->p, bar->type);

/\*

\* Barriers are in sorted order; if we pass our time, then we

\* won't find a barrier later. Short-circuit the loop, and

\* point bar to the 'end' of the list.

\*/

if (bar->logical\_time > when) {

bar = NULL;

DMPDBG(SHIM, shimtask, " Reached a later barrier, done\n");

break;

}

if (bar->p == dmptask && bar->logical\_time == when) {

DMPDBG(SHIM, shimtask, " Found a compatible barrier\n");

break;

}

}

/\*

\* See if we reached the end of the list without finding a valid barrier

\*/

if (bar && (&bar->qlist == &box->pending\_quantum\_barriers)) {

DMPDBG(SHIM, shimtask, "Reached the end of the list\n");

bar = NULL;

}

/\*

\* See if we've found a barrier we can piggyback on

\*/

if (bar != NULL) {

DMPDBG(SHIM, shimtask, "Found a barrier we can reuse\n");

/\*

\* First make sure we're compatible with the existing barrier

\* event; to be compatible, exactly one of the two barriers

\* must be of type DISTQ.

\*/

if (!((bar->type == SHIM\_BARRIER\_FLAG\_DISTQ && type != SHIM\_BARRIER\_FLAG\_DISTQ)

|| (bar->type != SHIM\_BARRIER\_FLAG\_DISTQ && type == SHIM\_BARRIER\_FLAG\_DISTQ))) {

DMPERR(shimtask, "Task %p already has a barrier for this round: (new:%d, prev:%d)\n", shimtask, type, bar->type);

DMP\_BUG(shimtask);

}

/\*

\* Set the new type flag in this barrier

\*/

bar->type |= type;

} else {

DMPDBG(SHIM, shimtask, "Can't reuse; creating a new one\n");

/\*

\* We couldn't reuse one, so allocate a new one

\*/

bar = kmalloc(sizeof(\*bar), GFP\_KERNEL);

if (bar == NULL) {

DMPDBG(SHIM, shimtask, "failed! (out of memory)\n");

ret = -ENOMEM;

goto out\_unlock;

}

/\*

\* Initialize its values

\*/

memset(bar, 0, sizeof(\*bar));

INIT\_LIST\_HEAD(&bar->qlist);

bar->type = type;

bar->logical\_time = when;

bar->needs\_serial = needs\_serial;

bar->p = dmptask;

/\*\*

\* If we're setting a barrier for the serial mode of the current round, we

\* need to make sure we don't reset the quantum\_ticks value when the task is

\* begins executing again. If restore\_qticks is set, when the task is woken,

\* its quantum\_ticks will be set to the given value.

\*/

if (box->logical\_time % 2 == 0 && when == box->logical\_time + 1) {

DMPDBG(SYS, dmptask, "Barrier set for current round; preserving ticks value of %d\n", dmptask->dmp.quantum\_ticks);

bar->restore\_qticks = 1;

bar->qticks = dmptask->dmp.quantum\_ticks;

}

/\*

\* Append bar to box->pending\_quantum\_barriers in sorted order

\* NB: Races in the shim can make this list nondeterministic (>1

\* barriers added at the same time but for different tasks, but the

\* tasks are ordered nondeterministically). This is okay because we

\* sort tasks by dmp\_id when activating to deliver the barrier.

\*/

list\_for\_each\_entry(cur, &box->pending\_quantum\_barriers, qlist) {

if (cur->logical\_time >= when)

break;

}

/\* The new barrier belongs between cur->prev and cur \*/

list\_add(&bar->qlist, cur->qlist.prev);

DMPDBG(SHIM, shimtask, "barrier successfully set\n");

}

/\* This may initiate a fast-forward \*/

dmp\_quantum\_fast\_forward(box, NULL);

ret = when;

out\_unlock:

spin\_unlock\_irqrestore(&box->lock, flags);

return ret;

}

/\*\*

\* To be called by a shim thread to put its corresponding dmp thread

\* to sleep immediately.

\*

\* Parameters:

\*/

asmlinkage long sys\_dmp\_shim\_sleep(int sleep\_for\_wake)

{

struct task\_struct \*dmptask = current->dmp.shim;

struct semaphore \*dmpsem = &dmptask->dmp.shiminfo.sem;

struct task\_struct \*shimtask = current;

struct semaphore \*shimsem = &shimtask->dmp.shiminfo.sem;

if (dmptask == NULL)

return -EINVAL;

DMP\_BUG\_ON(shimtask, !IS\_DMP(dmptask));

DMPDBG(SHIM, shimtask, "Putting dmp task %p to sleep\n", dmptask);

/\*

\* Signal the dmp thread that it needs to sleep.

\*/

dmptask->dmp.shiminfo.need\_to\_sleep = 1;

shimtask->dmp.shiminfo.need\_to\_sleep = 1;

DMPDBG(SHIM, shimtask, "Forcing dmp state to TASK\_UNINTERRUPTIBLE (%d)\n", TASK\_UNINTERRUPTIBLE);

dmptask->dmp.shiminfo.saved\_sleep\_state = TASK\_UNINTERRUPTIBLE;

/\*

\* Allow it to actually go to sleep

\*/

DMPDBG(SHIM, shimtask, "Allowing dmp task %p to go to sleep (on: %p, ct: %d, empty? %d)\n", dmptask, dmpsem, atomic\_read(&dmpsem->count), !waitqueue\_active(&dmpsem->wait));

up(dmpsem);

/\*

\* Wait for the thread to acknowledge sleeping

\*/

DMPDBG(SHIM, shimtask, "Waiting for sleep acknowledgment... (on: %p, ct: %d, empty? %d)\n", shimsem, atomic\_read(&shimsem->count), !waitqueue\_active(&shimsem->wait));

down(shimsem);

DMPDBG(SHIM, shimtask, "Got sleep acknowledgment\n");

unsigned int flags;

spin\_lock\_irqsave(&dmptask->dmp.box->lock, flags);

DMPDBG(VMDBG, dmptask, "VMDBG lpid: %d, cpu: %d\n", dmptask->pid, task\_cpu(dmptask));

//DMPDBG(VMDBG, dmptask, "VMDBG stack: %p\n", dmptask->stack);

//DMPDBG(VMDBG, dmptask, "VMDBG cpu: %d\n", task\_cpu(dmptask));

dmp\_verify\_task(dmptask);

spin\_unlock\_irqrestore(&dmptask->dmp.box->lock, flags);

return 0;

}

asmlinkage long sys\_dmp\_shim\_wake(void)

{

DMPERR(current, "dmp\_shim\_wake not implemented\n");

return 0;

}