1-1-1992

# User-level Threads

Sanujit Senapati

INSTITUTE FOR SIMULATION AND TRAINING

# USER- LEVEL THREADS

SANUJIT SENAPATI

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF CENTRAL FLORIDA

DECEMBER 11, 1992

```
/*****************************************************************************
*                              EMISSION PDU                                  *
*                                                                            *
*                              DIS VERSION                                   *
*                                                                            *
*  Emission PDU Transmission                                                 *
*       When an Emission PDU is received by the protocol manager,            *
*       messages containing EMISSION_INFOs with one Emitter Systems are      *
*       created.                                                             *
*       Emission PDU is created from the info in EMISSION_INFO and sent      *
*       over the network.                                                    *
*                                                                            *
*  Responsibility:  Sanujit Senapati                                         *
*****************************************************************************/

// *** INCLUDES ***

#include <stdio.h>
#include <string.h>
#include "sim.h"
#include "util.h"
#pragma hdrstop
#include <assert.h>
#include <limits.h>
#include <mem.h>
#include "dr_emit.h"
#include "exec.h"
#include "loc_dis.h"
#include "p_num.h"
#include "simaddr.h"

// *** CONSTANTS ***

#define MAX_PACKET 1500

// *** TYPES ***

typedef struct
{
    UINT                    mem_check;       // checked for memory overwrite
    char                    comm_area[MPHS]; // for Network headers
    DIS_PDU_HEADER          header;          // Generic DIS PDU header
    DIS_EMISSION_VARIANT    var;             // The PDU proper
} DIS_EMIS_PACKET;

// *** STATIC DATA ***

static char               *buffer;
static DIS_EMIS_PACKET    *whole;
static EMISSION_INFO      *emit_info;

// *** LOCAL PROTOTYPES ***

// *** FUNCTIONS ***


/*****************************************************************************
*   Initialize the Emission PDU and allocates memory for the packet          *
*****************************************************************************/
void InitEmission()
```

```
{
    buffer      = (char *)malloc(MAX_PACKET);
    whole       = (DIS_EMIS_PACKET *)buffer;
    emit_info   = (EMISSION_INFO *)buffer;
}

/***************************************************************************
*   Copy info Beam Parm data to packet Beam Parm data.                    *
***************************************************************************/
static void DISBeamFromBeamInfo(DIS_BEAM_PARMS *dest, EMIT_FUND *src)
{
    dest->frequency           = src->frequency;
    dest->freq_range          = src->freq_range;
    dest->erp                 = src->erp;
    dest->prf                 = src->prf;
    dest->pulse_width         = src->pulse_width;
}

/***************************************************************************
*   Copy info Beam dir data to packet Beam dir data.                      *
***************************************************************************/
static void DISBeamDirFromBeamDir
(
    DIS_BEAM_DIR      *dest,
    EMIT_DIRECTION    *src_dir,
    EMIT_SWEEP        *src_sweep
)
{
    dest->azimuth_center      = src_dir->azimuth_center;
    dest->eleva_center        = src_dir->eleva_center;
    dest->azimuth_sweep       = src_sweep->azimuth_sweep;
    dest->eleva_sweep         = src_sweep->eleva_sweep;
    dest->beam_sweep_sync     = src_sweep->sync;
}

/***************************************************************************
*   Copy packet Beam Parm data to info Beam Parm data                     *
***************************************************************************/
static void BeamInfoFromDISBeam(EMIT_FUND *dest, DIS_BEAM_PARMS *src)
{
    dest->frequency          = src->frequency;
    dest->freq_range         = src->freq_range;
    dest->erp                = src->erp;
    dest->prf                = src->prf;
    dest->pulse_width        = src->pulse_width;
}

/***************************************************************************
*   Copy packet Beam dir data to info Beam dir data                       *
***************************************************************************/
static void BeamDirFromDISBeamDir
(
    EMIT_DIRECTION    *dest_dir,
    EMIT_SWEEP        *dest_sweep,
    DIS_BEAM_DIR      *src
)
{
    dest_dir->azimuth_center       = src->azimuth_center;
    dest_dir->eleva_center         = src->eleva_center;
    dest_sweep->azimuth_sweep      = src->azimuth_sweep;
    dest_sweep->eleva_sweep      = src->eleva_sweep;
    dest_sweep->sync             = src->beam_sweep_sync;
}

/***************************************************************************
*   Copy packet emitter parms to emitter parms                            *
***************************************************************************/
static void DISEmitDescToEmitDesc(EMITTER_DESC *dest,DIS_EMITTER_DESC *src)
{
    dest->freq_band = src->freq_band;
    dest->name      = src->name;
    dest->category  = src->category;
    dest->id        = src->id;
}

/***************************************************************************
*   Copy emitter parm to packet emitter parms                             *
***************************************************************************/
static void EmitDescToDISEmitDesc(DIS_EMITTER_DESC *dest,EMITTER_DESC *src)
{
    dest->freq_band = src->freq_band;
    dest->name      = src->name;
    dest->category  = src->category;
    dest->id        = src->id;
}

/***************************************************************************
*   Send an emission pdu on the network.                                  *
***************************************************************************/
void SendEmis(EMISSION_INFO *emis_info,TIME rightnow)
{
    VehicleID       id;
    EMITTER_INFO    *emitter;
    DIS_EMITTER     *dis_emitter;
    int             i,j,k,
                    pdu_size, info_size;

    pdu_size    = sizeof(DIS_PDU_HEADER) + sizeof(DIS_EMISSION_VARIANT);
    info_size   = sizeof(EMISSION_INFO);

    //  Assignment done to detect memory overwrites while building headers
    whole->mem_check           = CHECK_VAL;

    //  Build DIS envelope in whole.header
    whole->header.version    = DISProtocolVersionJan91;
    whole->header.exercise   = getExerciseID();
    whole->header.kind       = EMISSIONPDUKIND;

    //  Fill in non-header DIS fields(fixed information)
    whole->var.emitting_id             = emis_info->entity_id;
    whole->var.timestamp               = simToDIStime(rightnow);
    whole->var.event_id.simulator      = emis_info->entity_id.simulator;
    whole->var.event_id.event          = 1;
    whole->var.emitter_count           = emis_info->emitter_count;

    //  pointer to emitter info and packet area
    dis_emitter  = (DIS_EMITTER *)(whole+1);
    emitter      = (EMITTER_INFO *)(emis_info + 1);

    for (i=emis_info->emitter_count; i>0; i--)
```

```
{
    DIS_BEAM    *dis_beam    = (DIS_BEAM *)(dis_emitter+1); // beam pckt area
    BEAM_INFO   *beam        = (BEAM_INFO *)(emitter+1);    // beam info area

    //  Fill in emitter data
    //
    dis_emitter->rel_loc     = emitter->rel_loc;
    dis_emitter->beam_count  = emitter->beam_count;
    EmitDescToDISEmitDesc(&dis_emitter->desc,&emitter->desc);

    SwapDISEmitter(dis_emitter);

    //  Fill in information related to beams in the emitter
    //
    for(j=emitter->beam_count; j>0; j--)
    {
        EMIT_TARGET     *target;
        DIS_BEAM_TARGET *dis_target;
        UINT            target_count = beam->target_count;

        dis_beam->mode          = beam->mode;
        dis_beam->function      = beam->function;
        dis_beam->id            = beam->id;
        dis_beam->target_count  = target_count;

        DISBeamFromBeamInfo(&dis_beam->parms,&beam->parms);
        DISBeamDirFromBeamDir(&dis_beam->dir, &beam->dir, &beam->sweep);

        SwapDISEmissionBeam(dis_beam);

        dis_target  = (DIS_BEAM_TARGET *)(dis_beam + 1);
        target      = (EMIT_TARGET *)(beam + 1);

        //  Fill in information related to targets in the beams
        //
        for (k=target_count; k > 0; k--,target++,dis_target++)
        {
            dis_target->id          = target->id;
            dis_target->emitter_id  = target->emitter;
            dis_target->beam_id     = target->beam;

            SwapDISEmissionTarget(dis_target);
        }

        pdu_size  += sizeof(DIS_BEAM) +
                     target_count*sizeof(DIS_BEAM_TARGET);
        info_size += sizeof(BEAM_INFO) + target_count*sizeof(EMIT_TARGET);

        dis_beam    = (DIS_BEAM *)dis_target;
        beam        = (BEAM_INFO *)target;
    }

    pdu_size    += sizeof(DIS_EMITTER);
    info_size   += sizeof(EMITTER_INFO);

    emitter      = (EMITTER_INFO *)beam;
    dis_emitter  = (DIS_EMITTER *)dis_beam;
}

//  Update the distribution manager
```

```
    setAddressToThisNode(&id.simulator);
    id.vehicle = DIST_MGR;

    msg_build_and_send(&id,EMISSION_DATA,info_size,emis_info,NO_DELAY,FALSE,0);

    //  Internal send is done, now complete the packet
    SwapDISEmissionVariant(&whole->var);

    //  All finished, ship it out
    netSend(&whole->header,
            pdu_size,
            DISProtocolNumber,
            (int)whole->header.exercise,
            disPort());

    //  watch out for header overwriting memory.
    assert(whole->mem_check == CHECK_VAL);
}

/*****************************************************************************
*   Build an emission structure from an arriving DIS EMIS pdu.              *
*   Local data area is used to build info.                                  *
*****************************************************************************/
void BuildSendEmission(DIS_EMISSION_VARIANT *evar)
{
    VehicleID       id;
    DIS_EMITTER     *dis_emitter;
    EMITTER_INFO    *emitter;
    int             i,j,k,
                    info_size = sizeof(EMISSION_INFO);

    //  Swap all multi-octet fields
    //
    SwapDISEmissionVariant(evar);

    //  Build Internal representation of emittingID
    //
    emit_info->entity_id        = evar->emitting_id;
    emit_info->timestamp        = evar->timestamp;
    emit_info->emitter_count    = evar->emitter_count;

    // pointer to emitter packet area
    //
    dis_emitter   = (DIS_EMITTER *)(evar+1);
    emitter       = (EMITTER_INFO *)(emit_info+1);

    for(i=evar->emitter_count;i>0;i--)
    {
        DIS_BEAM    *dis_beam;
        BEAM_INFO   *beam;

        SwapDISEmitter(dis_emitter);

        //  Fill in emitter information in EMISSION_INFO
        //
        emitter->rel_loc    = dis_emitter->rel_loc;
        emitter->beam_count = dis_emitter->beam_count;
        DISEmitDescToEmitDesc(&emitter->desc,&dis_emitter->desc);

        // pointer to beam pckt area and beam info area
        //
        beam          = (BEAM_INFO *)(emitter + 1);
        dis_beam      = (DIS_BEAM *)(dis_emitter + 1);

        for (j=emitter->beam_count; j>0; j--)
        {
            DIS_BEAM_TARGET *dis_target;
            EMIT_TARGET     *target;
            UINT            target_count;

            //  Fill in beam information in EMISSION_INFO
            //
            SwapDISEmissionBeam(dis_beam);

            beam->mode              = dis_beam->mode;
            beam->function          = (EMIT_FUNC )dis_beam->function;
            beam->id                = dis_beam->id;
            beam->target_count      =
            target_count            = dis_beam->target_count;

            BeamInfoFromDISBeam(&beam->parms,&dis_beam->parms);
            BeamDirFromDISBeamDir (&beam->dir, &beam->sweep, &dis_beam->dir);

            dis_target   = (DIS_BEAM_TARGET *)(dis_beam + 1);
            target       = (EMIT_TARGET *)(beam + 1);

            for (k=target_count; k > 0; k--, target++, dis_target++)
            {
                SwapDISEmissionTarget(dis_target);

                target->id      = dis_target->id;
                target->emitter = dis_target->emitter_id;
                target->beam    = dis_target->beam_id;
            }

            info_size += sizeof(BEAM_INFO) + target_count*sizeof(EMIT_TARGET);

            dis_beam     = (DIS_BEAM *)dis_target;
            beam         = (BEAM_INFO *)target;
        }

        info_size += sizeof(EMITTER_INFO);

        dis_emitter = (DIS_EMITTER *)dis_beam;
        emitter     = (EMITTER_INFO *)beam;
    }

    assert(info_size < MAX_PACKET);

    //  Address of simulator and ID of vehicle
    //
    setAddressToThisNode(&id.simulator);
    id.vehicle = DIST_MGR;

    //  Send it in the name of the protocol manager
    proto_msg_send(&id,EMISSION_DATA,info_size,emit_info);
}

// **** End of file ****
```

```
/***************************************************************************\
*                            EMITTER DEAD RECKONING                          *
*                                                                            *
*   This module maintains the DR model of an emitter.                        *
*                                                                            *
*   When the distibution manager receives a message from the protocol        *
*   manager with EMISSION info, it checks the emitting entity list. If an    *
*   emitting entity exists then it is updated else a new emitting entity     *
*   is created.                                                              *
*                                                                            *
* Responsibility: Sanujit Senapati                                           *
***************************************************************************/

// *** INCLUDES ***

#include <stdio.h>
#include <mem.h>
#include "sim.h"
#include "ent_mgr.h"
#include "objects.h"
#include "util.h"
#pragma hdrstop
#include "dist_mgr.h"
#include "dr_emit.h"
#include "simaddr.h"

// *** CONSTANTS ***

// *** TYPES ***

// *** STATIC DATA ***

static CHAIN_HEAD emitting_entity_list = {0};
static CHAIN_HEAD detection_entry_free = {0};

// *** LOCAL PROTOTYPES ***

// *** FUNCTIONS ***

#define ReceiveEmission(recv_const,emit_const,dist) \
                    (dist <= (recv_const)*(emit_const))

/**************************************************************************
 * Calculate the 'emitter dr constant' from the emitter parameters.  The  *
 * formula used for detection by a receiver is based on the parameters of *
 * the receiver and the candidate emitter.  The receiver's parameters are *
 * Antenna Gain(AG), Minimum Discernable Signal (MDS), and Front End Loss *
 * (FEL).  The emitter's parameters are Frequency (F) and Effective Radiated*
 * Power (ERP).  The receiver precalculates a constant containing factors *
 * contributed by its parameters.  The emitter parameters can be similarly *
 * precalculated.                                                         *
 *                                                                        *
 * The complete equation follows:                                         *
 *                                                                        *
 * Detection Range (km) =                                                 *
 *   (7.549 * 10^2) *                                                     *
 *   square root                                                          *
 *   (                                                                    *
 *      (antilog (ERP/10) * antilog (GR/10) * antilog (MDS/10)) /         *
 *      (F^2 * antilog (FEL/10))                                          *
 *   )                                                                         *
 *                                                                             *
 * The follow 'emitter dr constant' is extracted from the above equation:      *
 *                                                                             *
 * Emitter dr constant = square_root ( antilog (ERP/10.0) / F^2)               *
 *                                                                             *
 * This simplified detection range equation results:                           *
 *                                                                             *
 * Detection Range = receiver_constant * emitter_dr_constant.                  *
 *                                                                             *
 * Return:  the 'emitter dr constant' value explained above                    *
 *****************************************************************************/
static float EmitterDRConstant (float erp, float rf)
{
    return ( sqrt ( pow (10, erp / 10.0) / (rf * rf) ) );
}

/*****************************************************************************
*   This routine initializes the Emitting Entity List.                        *
*****************************************************************************/
void InitEmittingEntityList(void)
{
    DLL_init(&emitting_entity_list);
    DLL_init(&detection_entry_free);
}

/*****************************************************************************
*   Creates a new Emitting Entity Entry and points at the DR of the entity. *
*****************************************************************************/
EMITTING_ENTITY *MakeEmittingEntity(VehicleID *entity_id)
{
    EMITTING_ENTITY *emit=0;
    DEAD_REC *dr_model;

    // Do not add emitter if there is no dr model
    //
    if (dr_model = get_dr_model(entity_id))
    {
        EMITTING_ENTITY *previous;

        emit = (EMITTING_ENTITY *) malloc(sizeof(EMITTING_ENTITY));

        // Initialize emitter
        //
        emit->dr_model  = dr_model;
        emit->entity_id = *entity_id;

        // Put emitter in ascending order
        //
        previous = (EMITTING_ENTITY *) DLL_getnxt(&emitting_entity_list);

        while ( previous &&
                RelationID(&previous->entity_id,LESS_THAN,entity_id))
            previous = (EMITTING_ENTITY *) DLL_getnxt(previous);

        if (previous)
            DLL_prepend(&previous->link,&emit->link);
        else
            DLL_append(CLink(&emitting_entity_list),&emit->link);
    }
    return( emit );
}

/*****************************************************************************
*   This routine locates an Emitting Entity structure in the                *
*   emitting_entity list and returns its address (NULL if it can't match    *
*   the vehicle ID).                                                        *
*****************************************************************************/
EMITTING_ENTITY *GetEmittingEntity(VehicleID *entity_id)
{
    EMITTING_ENTITY *emit =    (EMITTING_ENTITY *)
                                DLL_getnxt(&emitting_entity_list);

    while (emit && !match_ids(&emit->entity_id,entity_id))
        emit = (EMITTING_ENTITY *)DLL_getnxt(emit);

    return(emit);
}

/*****************************************************************************
*   This routine locates an emitter with the given ID in the emitter list.  *
*   A pointer to the emitter is returned (NULL if it can't find a match).   *
*****************************************************************************/
EMITTER_DR *GetEmitterDR(CHAIN_HEAD *emitter_list, UCHAR id)
{
    EMITTER_DR *emitter = (EMITTER_DR *) DLL_getnxt(emitter_list);

    while (emitter && emitter->desc.id != id)
        emitter = (EMITTER_DR *)DLL_getnxt(emitter);

    return(emitter);
}

/*****************************************************************************
*   Update the DR beam parameters using the beam info.                      *
*   New beams are created and added to list.                                *
*****************************************************************************/
static void UpdateDRBeam(  CHAIN_HEAD *head, BEAM_INFO *beam_info)
{
    EMITTER_DR_BEAM *beam = (EMITTER_DR_BEAM *) DLL_getnxt(head);

    // Search for beam in linked list.
    while (beam && beam->id != beam_info->id)
        beam = (EMITTER_DR_BEAM *)DLL_getnxt(beam);

    if (beam && beam_info->mode == OFF)
    {
        // Beam is OFF, delete from list

        DLL_release(&beam->link);
    }
    else
    if (!beam && beam_info->mode != OFF)
    {
        // New beam, add to list

        beam = (EMITTER_DR_BEAM *) malloc(sizeof(EMITTER_DR_BEAM));
        DLL_append(CLink(head),&beam->link);
    }
    // Update parameters of active beams.
    //
    if (beam && beam_info->mode != OFF)
    {
        beam->mode          = beam_info->mode;
        beam->function      = beam_info->function;
        beam->id            = beam_info->id;
        beam->parms         = beam_info->parms;
        beam->dir           = beam_info->dir;
        beam->sweep         = beam_info->sweep;
        beam->parm_const    = EmitterDRConstant (beam->parms.erp,
                                                 beam->parms.frequency);

        if (beam_info->target_count)
            beam->target = *((EMIT_TARGET *)(beam_info + 1));
    }
}

/*****************************************************************************
*   Update the emitting entity chain as needed based on a new emission      *
*   info.  If the emission PDU is for a new entity, a new emitting entity   *
*   entry is created; otherwise the existing emitting entity list is        *
*   updated.                                                                *
*****************************************************************************/
void NewEmitter(EMISSION_INFO *info)
{
    EMITTING_ENTITY *emit;

    // Get the emitting entity entry or else attempt to make one; otherwise
    // forget about emission (no dr model exist for entity).
    //
    if ((emit = GetEmittingEntity(&info->entity_id)) ||
        (emit = MakeEmittingEntity(&info->entity_id)))
        UpdateEmissionDR(&emit->dr_model->emitters,info);
}

/*****************************************************************************
*   Update the emitter chain in the entity dead reckoning model as          *
*   needed based on a new emission info. If the emitter does not            *
*   exist, then a new emitter is created. The beams and parameters are also *
*   updated.                                                                *
*****************************************************************************/
void UpdateEmissionDR(CHAIN_HEAD *emitter_list, EMISSION_INFO *info)
{
    EMITTER_DR      *emitter;
    EMITTER_INFO    *emit_info = (EMITTER_INFO *)(info + 1);
    BEAM_INFO       *beam;
    UINT            i,j;

    for (i=info->emitter_count; i>0; i--)
    {
        if (!(emitter = GetEmitterDR(emitter_list,emit_info->desc.id)))
        {
            emitter = (EMITTER_DR *) malloc(sizeof(EMITTER_DR));
            emitter->beam_count = 0;
            DLL_init(&emitter->beams);
            DLL_append(CLink(emitter_list),&emitter->link);
        }
        //  Update emitter
        //
        emitter->timestamp  = info->timestamp;
        emitter->desc       = emit_info->desc;
        emitter->rel_loc    = emit_info->rel_loc;

        //  Update beams
        //
        beam = (BEAM_INFO *)(emit_info + 1);
        for (j=emit_info->beam_count; j>0; j--)
        {
            UpdateDRBeam(&emitter->beams,beam);

            beam = (BEAM_INFO *)((EMIT_TARGET *)beam + beam->target_count)+1;
        }

        emit_info = (EMITTER_INFO *)beam;
    }
}
/******************************************************************************
*   Release the emitter data in dr model. Also release the                    *
*   corressponding entry in the emitting entity list.                         *
******************************************************************************/
void ReleaseEmitterDR(CHAIN_HEAD *emitters)
{
    EMITTER_DR *emitter;

    while (emitter = (EMITTER_DR *) DLL_getnxt(emitters))
    {
        EMITTER_DR_BEAM *beam;

        while (beam = (EMITTER_DR_BEAM *) DLL_getnxt(&emitter->beams))
            DLL_release(&beam->link);

        DLL_release(&emitter->link);
    }
}

/******************************************************************************
*   Builds a list of Emitters detectable by a Receiver based upon its         *
*   receiver parameters, emitter parameters, and distance between the         *
*   receiver and emitter.                                                     *
*   Reception equation variables have been rolled up into constants for       *
*   receiver and emitter.  The multiplication of these constants gives the    *
*   maximum detection range.                                                  *
******************************************************************************/
void DetectableEmissions(LOC *recv_loc, float recv_const, CHAIN_HEAD *list)
{
    EMITTING_ENTITY *emit_ent = (EMITTING_ENTITY *) &emitting_entity_list;

    DLL_init(list);

    while (emit_ent = (EMITTING_ENTITY *) DLL_getnxt(emit_ent))
    {
        DEAD_REC    *dr         = emit_ent->dr_model;
        EMITTER_DR  *emitter    = (EMITTER_DR *) &dr->emitters;

        while (emitter = (EMITTER_DR *) DLL_getnxt(emitter))
        {
            EMITTER_DR_BEAM *beam;
            LOC             emitter_loc;
            float           dist;

            RelativeToActual(&emitter_loc,emitter->rel_loc,dr);

            dist = distance(recv_loc,&emitter_loc);

            /* Determine if at least one beam can be received */

            /* Evaluate beams */
            beam = (EMITTER_DR_BEAM *) &emitter->beams;

            while ((beam = (EMITTER_DR_BEAM *) DLL_getnxt(beam)) &&
                    !ReceiveEmission(recv_const,beam->parm_const,dist));

            if (beam)
            {
                E_DR_ENTRY *emit_dr;

                //  Get or create an emission detection entry
                //
                if (emit_dr = (E_DR_ENTRY *) DLL_getnxt(&detection_entry_free))
                    DLL_delete(CLink(emit_dr));
                else
                    emit_dr = (E_DR_ENTRY *)malloc(sizeof(E_DR_ENTRY));

                //  assign dr model and emitter system
                //  to entry and append to entry list
                //
                emit_dr->dr = dr;
                emit_dr->e_dr = emitter;
                DLL_append(CLink(list),&emit_dr->link);
            }
        }
    }
}

/******************************************************************************
*   Append an emitter detection entry list to the detection free list.        *
******************************************************************************/
void ReleaseEDREntryList(CHAIN_HEAD *e_dr_entry_list)
{
    DLL_AppendList(&detection_entry_free,e_dr_entry_list);
}

/******************************************************************************
*   Append an emitter detection entry to the detection free list.             *
******************************************************************************/
void ReleaseEDREntry(E_DR_ENTRY *edr_entry)
{
    DLL_delete(CLink(edr_entry));
    DLL_append(CLink(&detection_entry_free),CLink(edr_entry));
}

// *** END OF FILE ***
```

```
/*****************************************************************************
*   TRACK BEAM SIMULATION                                                    *
*                                                                            *
*   Track beam scan.                                                         *
*                                                                            *
*   Responsibility: Sanujit Senapati                                         *
*****************************************************************************/

/*** INCLUDES ***/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sim.h"
#include "objects.h"
#include "util.h"
#pragma hdrstop
#include "dist_mgr.h"
#include "ent_mgr.h"
#include "extrepor.h"
#include "fsm.h"
#include "loc_emis.h"

/*** CONSTANTS ***/


/*** TYPES ***/

/*** STATIC DATA ***/

static float dir_threshold = M_PI/360;      // Radian value of 1/2 degree

//  Indicates if the header information is put in the Emission Info
//
static int push_header = FALSE;

/*** LOCAL PROTOTYPES ***/

/*** FUNCTIONS ***/

/*****************************************************************************
*   Set the value of direction threshold value to the given angle value.    *
*****************************************************************************/
static void setDirTH(int angle)
{
    dir_threshold = degrees_to_radians(angle);
}

/*****************************************************************************
*   Determine the minimum interval when one of an emitter's track beams     *
*   will require an update of its direction.                                 *
*                                                                            *
*   The interval for each beam is calculated as the time required for the   *
*   target to traverse the chord across the direction threshold angle       *
*   assumming worst case (i.e. the target and observer are traveling in     *
*   opposite directions and are adjacent to each other).                     *
*                                                                            *
*   The interval value is given by the following formula (t=m/v):           *
*                                                                            *
*       I   = interval value                                                 *
*       D   = distance from observer to target                                 *
*       TA  = tangent of direction threshold angle                             *
*       CL  = Length of chord                                                  *
*       S   = relative speed between observer and target                       *
*                                                                              *
*       CL = D * TA                                                            *
*       I = CL/S = D * TA / S                                                  *
******************************************************************************/
static UINT TrackInterval(ENTITY_CB *e_cb, EMITTER *emitter)
{
    int         i,
                interval    = e_cb->ew.repeat_rate;
    float       tan_dir     = tan(dir_threshold),
                ent_speed   = abs(e_cb->dynamics.loc_dyn.speed);
    LOC         emit_loc;
    TRACK_BEAM  *beam       = emitter->track.beams;
    DEAD_REC    *dr;

    RelativeToActual(&emit_loc,emitter->rel_loc,e_cb->own_dr);

    for ( i=emitter->track.beam_count; i>0; i--, beam++ )
    if ( beam->active && (dr = get_dr_model(&beam->target.id)) )
    {
        float   dist    = distance(&dr->dr_loc,&emit_loc),
                vel_x   = dr->dynamics.vel[0],
                vel_y   = dr->dynamics.vel[1],
                vel_z   = dr->dynamics.vel[2],
                speed   = ent_speed +
                            sqrt(vel_x*vel_x + vel_y*vel_y + vel_z*vel_z);

        if (speed)
        {
            int delay;

            delay = dist*tan_dir / speed;

            if (interval > delay)
                interval = delay;
        }
    }

    return(interval);
}

/******************************************************************************
*   Update the Track Direction Parameters (azimuth and elevation center)     *
*   based on the current location of the tracked target with respect to the *
*   observing emitter.                                                        *
*   Return boolean: change greater than the threshold value?                 *
******************************************************************************/
int UpdateTrackDir(LOC *emit_loc, LOC *target_loc, TRACK_BEAM *beam)
{
    EMIT_DIRECTION *last_dir    = &beam->last_dir,
                   *dir         = &beam->dir;

    dir->azimuth_center = bearing(emit_loc,target_loc);
    dir->eleva_center   = elevation(emit_loc,target_loc);

    return
    (
        (abs_diff(dir->azimuth_center, last_dir->azimuth_center)
            > dir_threshold)
        ||
        (abs_diff(dir->eleva_center, last_dir->eleva_center) > dir_threshold)
    );
}

/*****************************************************************************
*   Track targets and update the direction and tracking status. Report any  *
*   change in tracking status. Build a PDU for all the track beams with     *
*   direction changes. Return TRUE if direction of one or more track beams  *
*   is changed.                                                             *
*****************************************************************************/
static int EmitterTrackScan(ENTITY_CB *e_cb, EMITTER *emitter)
{
    int             i,
                    push_emitter    = FALSE,
                    track_changed   = FALSE;
    EMITTER_DESC    *emit_desc      = &emitter->desc;
    EMITTER_TRACK   *track          = &emitter->track;
    TRACK_BEAM      *beam           = track->beams;
    LOC             emit_loc;

    RelativeToActual(&emit_loc,emitter->rel_loc,e_cb->own_dr);

    for (i=track->beam_count; i>0; i--, beam++)
    if (beam->active)
    {
        LOC target_loc;

        if (mob_loc_from_id(&target_loc,&beam->target.id) &&
            distance(&target_loc,&emit_loc) <= beam->range)
        {
            track_changed = UpdateTrackDir (&emit_loc, &target_loc, beam);
        }
        else
        {
            //  Lost track on target, deactivate track beam

            SendRadarReport(e_cb, emit_desc, &beam->target.id,
                                RADAR_DEACTIVATE);
            track->active_beam_count--;
            beam->active    = FALSE;
            track_changed   = TRUE;
        }

        if (track_changed)
        {
            if (!push_header)
            {
                //  No Emission Info header, push header
                //  and set the header pushed flag TRUE
                //
                PushHeader(&e_cb->dynamics.vehicleID,e_cb->control.useTime);
                push_header = TRUE;
            }

            if (!push_emitter)
            {
                //  Emitter Information not inserted,
            //  push emitter info and set flag
            //
            PushEmitterSystem(emitter);
            push_emitter = TRUE;
        }

        //  Push the changed tracked beam and parameters
        //
        PushTrackBeam(beam,beam->active ? ON : OFF);
      }
    }

    return(track_changed);
}

/*****************************************************************************
*   Emitter Track Scan FSM which tracks targets for all the active emitter   *
*   track beams within the emitter tracking capability.                      *
*****************************************************************************/
void EWTrack(FSM_RECORD *fsm)
{
    ENTITY_CB *e_cb = (ENTITY_CB *)fsm->cb;

    fsm->name = "EWTrack";

    if (Destroyed(e_cb->dynamics.appearance))
        FSM_suicide(INCAPABLE);
    else
    {
        ENTITY_EW   *ew         = &e_cb->ew;
        EMITTER     *emitter    = (EMITTER *) &ew->active_emitters;
        int         active      = FALSE,
                    changed     = FALSE,
                    track_interval,
                    repeat_rate = e_cb->ew.repeat_rate;

        // Process each emitter in active emitter list
        //
        while (emitter = (EMITTER *) DLL_getnxt(emitter))
        {
            if (emitter->track.active_beam_count)    // track function
                changed |= EmitterTrackScan(e_cb,emitter);

            track_interval = TrackInterval(e_cb,emitter);

            if (repeat_rate > track_interval)
                repeat_rate = track_interval;

            if (emitter->track.active_beam_count)
                active = TRUE;
            else
            if (!emitter->search.active)
            {
                // This emitter is no longer active.
                // Remove it from active list.

                DLL_delete(CLink(emitter));
                DLL_append(CLink(&ew->emitters),&emitter->link);
            }
        }
        if (changed)
        {
            //  Send the Emission Info
            //
            SendEmitter();
            push_header = FALSE;
        }

        // Repeat state if the track beam is still active.
        if (active)
            FSM_repeat(repeat_rate);
    }
}

/*** END OF FILE ***/
```

# User-Level Threads

**Sanujit Senapati**

Department of Computer Science

University of Central Florida

December 11, 1992

**Abstract**

Threads are the vehicles of concurrency in many approaches to parallel programming. Threads separate the notion of a sequential execution stream from other traditional processes, such as address spaces and I/O descriptors. The objective of this separation is to make the expression and control of parallelism simpler for use. Threads are supported either by the operating system or by an user-level library. In this term paper we will study the various characteristics and functionality of user-level and kernel threads, the various approaches of user-level thread design, the various implementation issues and the adaptability and use of threads in real-time operating systems.

## 1 Introduction

Threads are a part of most operating systems, because of their efficiency, portability and clarity in implementing parallelism in programming. Traditional processes are not suitable for parallel programming considering the efficiency and system overhead in creating and synchronizing kernel processes. Therefore many run-time environments have implemented threads which are basically lightweight processes with the required data structures and functionality to implement parallelism. Threads have been implemented in operating systems at kernel and user-level.

In the following sections, we will discuss the various functionality of threads in general, the advantages and disadvantages of user-level threads over kernel threads, the various approaches used in the design and implementation of user-level threads, the adaptability of threads for use in real-time operating systems.

## 2 Functionality of Threads

The main functionality of an user-level threads package can be divided into calls regarding thread manipulation (e.g., creation), thread control (e.g., synchronization), and higher level utilities (e.g., condition variables). The various data structures a threads package usually maintains consists of:

- stack pool, for locally executing threads.
- a pool of thread descriptors.
- scheduling information.
- a memory pool for future memory allocation/deallocation requests.

The various thread manipulation calls can be call for forking operations. Thread control calls are sleep, wakeup and lock operations. Other memory allocation operations can be memory allocation. memory deallocation. High level operations can be wait and signal.

## 3 User-Level vs Kernel Threads

Threads can be supported either at user level or in the kernel. Each of these approaches has its advantages and disadvantages. Before comparing the two classes of threads, we will discuss the primary characteristics of both the approaches. Kernel threads are built in the operating system and have their own address space, priority, private file descriptors, etc like any other kernel process. Scheduling operations on these threads are done by the kernel. Kernel threads run in the kernel space. On the other hand, user-level threads are implemented as run-time library routines which can be linked to an application just like any other functions in a programming language. This approach of thread management does not require any kernel intervention, thereby has the flexibility of implementing parallelism in programming at the user-level. Besides user-level threads run in the user space.

## 3.1 Kernel Threads

The kernel based approach treats threads similar to a process, thus resulting in degradation of performance due to the overhead of process management, like unique address space, registers, program counter, etc and synchronization. A thread may or may not need its own address space, priority, private file descriptors or signal interface. These are unused in the user space and can incur unwarranted costs. Kernel threads support a limited number of user and application thread requirements and semantically inflexible.

## 3.2 User-Level Threads

On the other hand, user-level avoid the problems associated with kernel threads, but they also introduce new problems. User-level threads require that the kernel provide a set of non-blocking system calls. Otherwise execution of other threads may be prevented. Another problem with user-level threads is the lack of coordination between scheduling and synchronization. Synchronization between threads, either in the same address space or in overlapping address space may be adversely affected by kernel scheduling decisions. A thread that is preempted by the kernel may be performing operations for which other, running threads may hold a mutual exclusion lock forcing other threads to spin or block, thereby slowing execution.

Considering the above factors, proper approaches with flexibility and better performance are required in user-level thread design. The approaches should address the following problems:

- Functionality: the user level thread should exhibit these behaviour of a kernel thread management system:

  - No processor idles in the presence of ready threads.

  - No high priority thread waits for a processor while a low priority thread runs.

  - When a thread traps to the kernel to block, the processor on which the thread was running can be used to another thread from the same or from a different address space.

- Performance: the cost of thread management operations should be within an order of magnitude of a procedure call, essentially the same as that achieved by the best existing user-level thread management system.

- Flexibility: the user-level part should be structured to simplify application specific customization.

## 4 Related Work

Several approaches supporting user-level threads, have been implemented in the past. We will discuss a few of them in this section.

### 4.1 Scheduler Activations Model

The approach given in [1] provides each application with a virtual multiprocessor. Each application knows exactly how many (and which) processors have been allocated to it and has complete control over which threads are running on these processors. The operating system kernel has control over the allocation of processors among addresses including the ability to change the number of processors assigned to an application during its execution. To achieve this, the kernel notifies the address space thread scheduler of every event affecting the address space. The kernel's role is to vector events that influence user-level scheduling to the address space for the thread scheduler to handle, rather than to interpret these events on its own. Also, the thread system in each address space notifies the kernel of the subset of user-level events that affect processor allocation decisions. By communicating all kernel events, functionality is improved because the application has complete knowledge of its scheduling state. By communicating downward only those events that affect processor allocation, good performance is preserved, since most events do not need to be reflected to the kernel. These kernel mechanisms realizing these ideas are called *scheduler activations*. A scheduler activation is the execution context for an event vectored from the kernel to an address space; the address space thread scheduler uses this context to handle the event to modify user-level thread data structures, to execute user-level threads, and to make requests of the kernel.

## 4.2 Psyche Model

Another similar approach in which the above concept of virtual multiprocessor is given in [2]. A design of a kernel interface to be used by user-level thread packages has been implemented. Short-term scheduling takes place in the user space. The kernel remains in charge of the resource allocation and protection. The various operations performed by the threads package including creation, destruction, synchronization, and context switching, occur in user space without entering the kernel.

The kernel has access to thread state information maintained by the threads package through shared data structures, which makes it easier for conveying information efficiently in both directions. The kernel provides the thread package with software interrupts (signals, upcalls) whenever a scheduling decision may be required. Timer interrupts support the time-slicing of threads. Warnings prior to preemption allow the thread package to coordinate synchronization with kernel-level scheduling. An interrupt delivered each time a thread blocks in the kernel makes every system call non-blocking by default, without modifying or replacing the kernel interface, and provides an uniform entry mechanism into the user-level scheduler when a thread has blocked or unblocked. The operating system establishes a standard interface for user-level schedulers, and provides locations in which to list the functions that implement the interface. Abstractions shared between thread packages can then invoke appropriate operations to block and unblock different kinds of threads. Although the kernel never calls these operations it identifies them in the kernel/user data area so that user-level code can invoke them without depending on the referencing environment of any programming language or thread package.

## 4.3 Continuation Model

The approach discussed in [3] is an improvement of the user-level threads of Mach threads and interprocess communication with the use of *continuations* as a basis for control transfer between execution contexts.

In this system, a thread blocks in the kernel in one of the two ways. It either preserves its register state and stack and resumes execution by restoring this state, or it specifies its resumption context as a continuation, a function that the thread should execute when it next runs. by allowing a thread to block with a continuation, the kernel programmer can space and time during thread management. Continuations enable a thread to discard its stack while blocked, thereby reducing the space required to support threads in the kernel. Continuations also allow a thread to present a high-level representation of its execution state while blocked, reducing control transfer overhead because the state can be examined and acted upon.

The kernel has been restructured so that a thread can use either the process model or the interrupt model when blocking. When a thread blocks using the interrupt model, it records the execution context in which it should be resumed in an auxiliary data structure, called a continuation. The blocked thread is resumed by means of a call to the saved continuation. The process has advantages because a thread may block with its stack context intact at any time within the kernel. On the other hand when the thread has little or no context, say because it is waiting to receive a message from another thread, or because the next instruction it should execute is in user space, it may relinquish its kernel stack entirely. Furthermore, since a continuation is accessed through a machine-independent interface, it is often possible to examine a continuation at runtime and avoid using it, because the system's current state makes it unnecessary. Another advantage is that many low-level optimizations associated with control transfer in operating systems can be recast in terms of continuations.

## 5 Adapting User-level threads for real-time operating systems

Real-time operations are time bound in nature and have high timing requirements. Besides real-time thread design should consider the requirements of predictability and high efficiency for real-time kernels. The essential idea underlying real-time threads is that the schedulability of real-time threads may be determined at the time of program compilation (static scheduling) or at the time of threads creation (dynamic scheduling). Dynamic scheduling of threads cannot be scheduled off-line. Static scheduling may be performed at the the time of program initialization or by simply initializing the package's data structures containing scheduling information based on the decisions of off-line algorithms. The typical use of real-time threads in an actual system, then, is to make static (initialization time or compile-time) guarantees for the application's minimal task set while the runtime guarantees are made for threads with unpredictable arrival and execution times.

## 6 Conclusion

In the effort for making threads perform efficiently and offer flexibility to a variety of user and application requirements, the various approaches discussed in this term paper, have shared the functions of typical threads between the user level and kernel implementations. Functions which do not need much of kernel support like memory allocation and scheduling have been implemented at the user-level and the functions like resource allocation have been implemented at the kernel level. The various interfaces between the kernel and user-level data structures have been properly studied.

## 7 References

[1] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M., "Scheduler Activations: Effective Kernel Support for User-Level Management of Parallelism", *In Proceedings of the 13th ACM Symposium of Operating System Principles*, October 1991.

[2] Marsh, B. D., Scott, M. L., LeBlanc, T. J., and Markatos, E. P., "First-Class User-Level Threads", *In Proceedings of the 13th ACM Symposium of Operating System Principles*, October 1991.

[3] Draves, R. P., Bershad, B. N., Rashid, R. F. and Dean, R. W., "Using Continuations to Implement Thread Management and Communication in Operating Systems", *In Proceedings of the 13th ACM Symposium of Operating System Principles*, October 1991.

[4] Schwan, K., Zhou, H., Gheith, A., "Real-Time Threads", *Operating System Review*, April 1991.

0000133