"Learn the truth and never reject it."

47 Writing Operating System

Operating System is nothing but collection of programs for managing system resources like CPU, memory, storage device etc. Study of the Operating System is one of the vastest areas. This chapter does not deal with the details about Operating System. And in this chapter I would like to show you how OS can be written in Turbo C. However you may not be able to code your Operating System without depth knowledge of memory management, processor scheduling etc. So I strongly recommend you to go through a good Operating System book for indepth knowledge. According to me most of the people are not using Turbo C to write OS, because Turbo C is 16bit. Also people mainly hangout with Assembly language for a better and tight code.

47.1 EZOS_86

EZOS_86 is a simple multitasking kernel written in Turbo C by Scott A. Christensen for x86 machines in 1996-97. Operating Systems are usually protected and licensed according to GNU's General Public License and so this EZOS_86! So if you modify or rewrite this source code, you must acknowledge the author Scott A. Christensen and you are expected to keep the name of the revised OS as EZOS_86, but you can change the version. Regarding OS and other software, violation of copyright is treated as high offense. So *beware* of the licenses!

47.1.1 Notes

The author **Scott A. Christensen** added following note:

EZOS_86 is a simple multitasking kernel for the x86 family. It is written in 100% C source (it uses Turbo C extensions to access the registers). If you need a tight, fast, hand-coded, assembly kernel, forget this one!

The main emphasis here is to keep it simple: no linked lists, no dynamic allocation, no complicated task scheduling, no assembly language, etc. Yes, this can be embedded!

The scheduler is very rudimentary. It is preemptive, but with a strictly prioritized order. There is no protection from starvation; if a higher priority task spins the CPU, the lower priority tasks will never execute. Programs for embedded applications are often event driven and properly written will work fine. On the other hand, it wouldn't be that hard to change the scheduler to a round robin method if desired.

The scheduler always traverses the Task Control Block (TCB) array from the beginning (&tcb[0]). The first task encountered that is eligible to run is the one executed. At least one task

MUST always be eligible to run; hence the "null" task, which is created as the lowest priority and NEVER, sleeps.

The same task function can have multiple instances. For example you could call `OsTaskCreate( )` three times and pass task0 as the function all three times. Of course you must specify a unique stack and tcb. The parameter passed to task0 can identify the particular instance of the function.

Reentrancy issues:

- use the runtime library at your own risk (reason for direct video)
- floating point is not reentrant; use semaphore protection or only do floating point in one task.

Semaphores:

- clearing semaphore does not cause task switch; call `OsSchedule( )` to yield. This can throttle throughput. One could have null task continuously scan TCBs for eligible task and yield.
- `OsSemClear( )` returns TRUE if higher priority task waiting on sem
- multiple tasks can sleep on same semaphore
- ok to clear semaphore from within interrupt routine

As written this code will run a demo on an IBM clones and even clean up upon exit returning nicely backs to DOS. It creates the file "out" to dump the stack contents. Interrupt routines use the current task's stack. Be careful not to exceed your allocated stack space; very strange results can occur. Compile it with Turbo C with optimization off.

Wishlist:

- simple file functions to read/write directly to IDE HD with FAT16
- multitasking capable floating point support
- some sort of built in debugging capability (TBUG.ZIP looks like a good start)
- runtime calculation of cpu utilization
- a _simplified_ malloc for embedded applications

47.1.2 Kernel Source Code

```
/*
 *    ezos_86.c
 *
 *    Copyright (c) 1996-7 Scott A. Christensen
 *    All Rights Reserved
 *
 *    This file is part of the EZOS_86 multitasking kernel.
 *
 *
```

```
 *    version     description
 *    -----------------------------------------------------------------
 *    0.01.00     initial release
 *
 */

#include <dos.h>
#include <stdio.h>
#include <conio.h>
#include <stdarg.h>

/*-----------------------------------------------------------------*/
#define TRUE                    (0 == 0)
#define FALSE                   (0 != 0)

#define RUNNING                 0
#define RUN_ASAP                1
#define SLEEPING                2
#define PENDING                 3
#define SUSPENDED               4
#define KILLED                  5

#define ALL_KILLED              -2
#define NOT_STARTED             -1

#define TICK_VECT               8

#define MAX_TASKS               10
#define STACKSIZE               1024

#define PENDING_SEM_REQUEST     0
#define PENDING_SEM_WAIT        1

#define TSK_ERR_                -1000
#define TSK_ERR_TIMEOUT         (TSK_ERR_ - 0)

#define OS_INFINITE_WAIT        -1L
#define OS_IMMEDIATE_RETURN     0L

#define OsEnable()              enable()
#define OsDisable()             disable()

#define ATTR                    ((unsigned int) (((BLACK<<4)|WHITE)<<8))

#define schedule()                                              \
     {                                                          \
       int                    si;                              \
```

```
        static PTCB_REC       pTCBsi;                       \
        static PTCB_REC       pTCBsc;                       \
                                                            \
        if(killedTasks == numTasks)                         \
        {                                                   \
          _SP       = mainSP;                               \
          _SS       = mainSS;                               \
          mainSleep = FALSE;                                \
          curTask   = ALL_KILLED;                           \
        }                                                   \
        else                                                \
        {                                                   \
          for(si = 0, pTCBsi = tcb; si < numTasks; si++, pTCBsi++) \
          {                                                 \
            if(pTCBsi->taskStatus == RUNNING)               \
              break;                                        \
            if(pTCBsi->taskStatus == RUN_ASAP)              \
            {                                               \
              pTCBsc = &tcb[curTask];                      \
              if(pTCBsc->taskStatus == RUNNING)            \
                pTCBsc->taskStatus = RUN_ASAP;             \
              pTCBsc->taskSP     = _SP;                    \
              pTCBsc->taskSS     = _SS;                    \
              pTCBsi->taskStatus = RUNNING;                \
              _SP                = pTCBsi->taskSP;         \
              _SS                = pTCBsi->taskSS;         \
              curTask            = si;                     \
              break;                                        \
            }                                               \
          }                                                 \
        }                                                   \
      }

/*-------------------------------------------------------------*/
typedef void (far cdecl *FUNCPTR)();

typedef struct
{
    unsigned int     r_bp;
    unsigned int     r_di;
    unsigned int     r_si;
    unsigned int     r_ds;
    unsigned int     r_es;
    unsigned int     r_dx;
    unsigned int     r_cx;
    unsigned int     r_bx;
    unsigned int     r_ax;
```

```
    FUNCPTR          taskStartAddr;
    unsigned int     r_flags;
    FUNCPTR          taskExitReturn;
    void *           pTaskParam;
} STACK_REC;

typedef struct
{
    unsigned int     taskStatus;
    unsigned int     taskSP;
    unsigned int     taskSS;
    long             ticks;
    int              semState;
    int *            pSem;
} TCB_REC, *PTCB_REC;

/*-------------------------------------------------------------*/
void far interrupt   OsTickIsr(void);
int  far interrupt   OsSchedule(void);
void far             OsTaskKill(void);
void                 OsTaskCreate(PTCB_REC, FUNCPTR, void *,
                                  unsigned char far *, int);
long                 OsTranslateMilsToTicks(long);
void                 OsInstall(void);
void                 OsRun(void);
void                 OsDeinstall(void);
void                 OsSleep(long);
void                 OsSleepTicks(long);
int                  OsSemClear(int *);
void                 OsSemSet(int *);
int                  OsSemWait(int *, long);
int                  OsSemSetWait(int *, long);
int                  OsSemRequest(int *, long);
int                  OsDisableStat(void);

void                 dumpStack(FILE *, unsigned char *, int);
void                 tprintf(const char *, ...);
void                 tputs(const char *);
void                 sout(char *);
void                 incRow(void);
void far             task0(void *);
void far             task1(void *);
void far             task2(void *);
void far             taskNull(void *);

/*-------------------------------------------------------------*/
void             (far interrupt *oldTickIsr)(void);
```

```
int               numTasks = 0;
int               killedTasks = 0;
int               curTask = NOT_STARTED;
int               mainSleep = TRUE;
unsigned int      mainSP;
unsigned int      mainSS;

TCB_REC           tcb[MAX_TASKS];
unsigned int      _stklen = (STACKSIZE * MAX_TASKS) + 1024;
int               itick = 0;
unsigned int      (far *screen)[80];
int               row = 0;
int               col = 0;
int               tickSem = 1;
int               goSem = 1;
int               screenSem = 0;

/*-------------------------------------------------------------*/
/*-------------------------------------------------------------*/
/*-------------------------------------------------------------*/
void main()
{
  unsigned char   stack0[STACKSIZE];
  unsigned char   stack1[STACKSIZE];
  unsigned char   stack2[STACKSIZE];
  unsigned char   stackNull[STACKSIZE];
  FILE *          f;

  clrscr();
  puts("\n\n          EZOS_86 multitasking kernel");
  puts("          Copyright (C) 1996-97 Scott A. Christensen");
  delay(5000);

  clrscr();
  gotoxy(1, 24);
  screen = MK_FP(0xB800, 0);

  OsTaskCreate(&tcb[0], task0, (void *) 100, stack0, STACKSIZE);
  OsTaskCreate(&tcb[1], task1, (void *) 101, stack1, STACKSIZE);
  OsTaskCreate(&tcb[2], task2, (void *) 102, stack2, STACKSIZE);
  OsTaskCreate(&tcb[3], taskNull, NULL, stackNull, STACKSIZE);

  OsInstall();

  OsRun();

  OsDeinstall();
```

```
  f = fopen("out", "wb");

  dumpStack(f, stack0, STACKSIZE);
  dumpStack(f, stack1, STACKSIZE);
  dumpStack(f, stack2, STACKSIZE);
  dumpStack(f, stackNull, STACKSIZE);

  fclose(f);

  puts("done, hit key to continue...");
  getch();
}

/*---------------------------------------------------------------*/
void dumpStack(
    FILE *              f,
    unsigned char *     stack,
    int                 size
    )
{
  int             i;
  char            buf[80];
  char            string[80];

  string[0] = 0;
  for(i = 0; i < size; i++)
  {
    if(i % 16 == 0)
      fprintf(f, "%04X:%04X  ", FP_SEG(&stack[i]), FP_OFF(&stack[i]));
    fprintf(f, "%02X ", stack[i]);
    if(isalnum(stack[i]) || stack[i] == ' ')
    {
      buf[0] = stack[i];
      buf[1] = 0;
      strcat(string, buf);
    }
    else
      strcat(string, ".");
    if(i % 16 == 15)
    {
      fprintf(f, " %s\r\n", string);
      string[0] = 0;
    }
  }
  fprintf(f, "\r\n");
}
/*---------------------------------------------------------------*/
```

```
void OsInstall()
{

  oldTickIsr = getvect(TICK_VECT);
  setvect(TICK_VECT, OsTickIsr);
}

/*-------------------------------------------------------------------*/
void OsRun()
{

  while(mainSleep);
}

/*-------------------------------------------------------------------*/
void OsDeinstall()
{

  setvect(TICK_VECT, oldTickIsr);
}

/*-------------------------------------------------------------------*/
void far interrupt OsTickIsr()
{
  int                i;
  static PTCB_REC    pTCBi;

  switch(curTask)
  {
    case ALL_KILLED:
      break;

    case NOT_STARTED:
      mainSP            = _SP;
      mainSS            = _SS;
      pTCBi             = tcb;
      pTCBi->taskStatus = RUNNING;
      _SP               = pTCBi->taskSP;
      _SS               = pTCBi->taskSS;
      curTask           = 0;
      break;

    default:
      itick++;
```

```
      for(i = 0, pTCBi = tcb; i < numTasks; i++, pTCBi++)
      {
        if((pTCBi->taskStatus == SLEEPING) ||
                              (pTCBi->taskStatus == PENDING))
          if(pTCBi->ticks > 0L)
            if(--(pTCBi->ticks) == 0L)
              pTCBi->taskStatus = RUN_ASAP;
      }
      schedule();
      break;
  }

  oldTickIsr();
}

/*-------------------------------------------------------------*/
int far interrupt OsSchedule()
{
  OsDisable();
  schedule();
  return _AX;                /* dummy value */
}

/*-------------------------------------------------------------*/
void far OsTaskKill()
{

  OsDisable();

  killedTasks++;

  tcb[curTask].taskStatus = KILLED;

  OsSchedule();
}

/*-------------------------------------------------------------*/
void OsTaskCreate(
    PTCB_REC               pTCB,
    FUNCPTR                func,
    void *                 pTaskParam,
    unsigned char far *    pStack,
    int                    stackSize
    )
{
  STACK_REC far *    pStackRec;
  int                i;
```

```
  for(i = 0; i < stackSize; i++)
    pStack[i] = 0xFF;

  pStackRec = (STACK_REC far *) (pStack + stackSize -
sizeof(STACK_REC));

  pStackRec->r_bp           = 0;
  pStackRec->r_di           = 0;
  pStackRec->r_si           = 0;
  pStackRec->r_ds           = _DS;
  pStackRec->r_es           = _DS;
  pStackRec->r_dx           = 0;
  pStackRec->r_cx           = 0;
  pStackRec->r_bx           = 0;
  pStackRec->r_ax           = 0;
  pStackRec->taskStartAddr  = func;
  pStackRec->r_flags        = 0x0200;
  pStackRec->taskExitReturn = OsTaskKill;
  pStackRec->pTaskParam     = pTaskParam;

  pTCB->taskStatus = RUN_ASAP;
  pTCB->taskSP     = FP_OFF(pStackRec);
  pTCB->taskSS     = FP_SEG(pStackRec);

  numTasks++;
}

/*-------------------------------------------------------------*/
long OsTranslateMilsToTicks(
    long          mils
    )
{
  long            x;

  if(mils < 0L)
    return -1L;

  if(!mils)
    return 0L;

  x = ((mils * 91L) / 5000L) + 1L;       /* 18.2 ticks per sec */

  return x;
}

/*-------------------------------------------------------------*/
void OsSleep(
```

```
    long          mils
    )
{
  long            ticks;

  ticks = OsTranslateMilsToTicks(mils);

  OsSleepTicks(ticks);
}

/*---------------------------------------------------------------*/
void OsSleepTicks(
    long          ticks
    )
{
  PTCB_REC        pTCB;

  if(ticks <= 0L)
    return;

  OsDisable();

  pTCB = &tcb[curTask];

  pTCB->taskStatus = SLEEPING;
  pTCB->ticks      = ticks;

  OsSchedule();
}

/*---------------------------------------------------------------*/
int OsSemClear(
    int *         pSem
    )
{
  int                 i;
  STACK_REC far *     pStackRec;
  int                 processedRequest;
  PTCB_REC            pTCB;
  int                 higherEligible;
  int                 intsEnabled;

  intsEnabled = OsDisableStat();
  if(!*pSem)
  {
    if(intsEnabled)
      OsEnable();
```

```
    return FALSE;
  }

  *pSem = 0;

  processedRequest = FALSE;
  higherEligible = FALSE;

  for(i = 0, pTCB = tcb; i < numTasks; i++, pTCB++)
  {
    if((pTCB->taskStatus == PENDING) && (pTCB->pSem == pSem))
    {
      switch(pTCB->semState)
      {
        case PENDING_SEM_REQUEST:
          if(processedRequest)
            break;
          processedRequest = TRUE;
          *pSem = 1;
          /* !!! no break here !!! */

        case PENDING_SEM_WAIT:
          pStackRec = MK_FP(pTCB->taskSS, pTCB->taskSP);
          pStackRec->r_ax = 0;
          pTCB->taskStatus = RUN_ASAP;
          if(i < curTask)
            higherEligible = TRUE;
          break;
      }
    }
  }

  if(intsEnabled)
    OsEnable();

  return higherEligible;
}

/*-------------------------------------------------------------*/
void OsSemSet(
    int *         pSem
    )
{
  int             intsEnabled;

  intsEnabled = OsDisableStat();
  *pSem = 1;
```

```
  if(intsEnabled)
    OsEnable();
}

/*-------------------------------------------------------------*/
int OsSemWait(
    int *         pSem,
    long          mils
    )
{
  long            ticks;
  PTCB_REC        pTCB;

  OsDisable();

  if(!*pSem)
  {
    OsEnable();

    return 0;
  }

  ticks = OsTranslateMilsToTicks(mils);

  if(!ticks)
  {
    OsEnable();

    return TSK_ERR_TIMEOUT;
  }

  pTCB = &tcb[curTask];

  pTCB->taskStatus = PENDING;
  pTCB->semState   = PENDING_SEM_WAIT;
  pTCB->pSem       = pSem;
  pTCB->ticks      = ticks;

  _AX = TSK_ERR_TIMEOUT;

  return OsSchedule();
}

/*-------------------------------------------------------------*/

int OsSemSetWait(
    int *         pSem,
```

```
    long          mils
    )
{

  OsDisable();

  OsSemSet(pSem);

  return OsSemWait(pSem, mils);
}

/*-------------------------------------------------------------*/
int OsSemRequest(
    int *         pSem,
    long          mils
    )
{
  long            ticks;
  PTCB_REC        pTCB;

  OsDisable();

  if(!*pSem)
  {
    *pSem = 1;
    OsEnable();
    return 0;
  }

  ticks = OsTranslateMilsToTicks(mils);

  if(!ticks)
  {
    OsEnable();
    return TSK_ERR_TIMEOUT;
  }

  pTCB = &tcb[curTask];

  pTCB->taskStatus = PENDING;
  pTCB->semState   = PENDING_SEM_REQUEST;
  pTCB->pSem       = pSem;
  pTCB->ticks      = ticks;

  _AX = TSK_ERR_TIMEOUT;
```

```
  return OsSchedule();
}

/*-------------------------------------------------------------*/
int OsDisableStat()
{
  unsigned int    flags;

  flags = _FLAGS;

  OsDisable();

  return flags & 0x0200;
}

/*-------------------------------------------------------------*/
void tprintf(
    const char *     format,
    ...
    )
{
  va_list         argPtr;
  char            buf[100];
  struct time     t;

  va_start(argPtr, format);
  vsprintf(buf + 18, format, argPtr);
  va_end(argPtr);

  OsSemRequest(&screenSem, OS_INFINITE_WAIT);

  gettime(&t);

  sprintf(buf, "-T%02d(%02d:%02d:%02d.%02d)",
          curTask, t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);

  buf[17] = ' ';

  sout(buf);

  OsSemClear(&screenSem);
}


/*-------------------------------------------------------------*/

void tputs(
```

```
    const char *     string
    )
{
  struct time     t;
  char            buf[100];

  OsSemRequest(&screenSem, OS_INFINITE_WAIT);

  gettime(&t);

  sprintf(buf, "-T%02d(%02d:%02d:%02d.%02d) %s\n",
          curTask, t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund, string);

  sout(buf);

  OsSemClear(&screenSem);
}

/*-------------------------------------------------------------*/
void sout(
    char *        p
    )
{

  while(*p)
  {
    switch(*p)
    {
      case '\r':
        col = 0;
        break;

      case '\n':
        col = 0;
        incRow();
        break;

      case '\t':
        sout("    ");
        break;

      default:
        screen[row][col] = ATTR | ((unsigned int) *p);
        if(++col > 79)
        {
          col = 0;
```

```
          incRow();
        }
        break;
    }
    p++;
  }
}

/*-----------------------------------------------------------------*/
void incRow()
{
  int             r;
  int             c;

  if(++row > 24)
  {
    for(r = 0; r < 24; r++)
      for(c = 0; c < 80; c++)
        screen[r][c] = screen[r + 1][c];

    for(c = 0; c < 80; c++)
      screen[24][c] = ATTR | ((unsigned int) ' ');

    row = 24;
  }
}

/*-----------------------------------------------------------------*/
void far task0(
    void *        pTaskParam
    )
{
  int             val = (int) pTaskParam;
  int             i;
  long            j;
  int             rc;

  OsSemWait(&goSem, OS_INFINITE_WAIT);

  tprintf("init val passed = %d\n", val);

  for(i = 0; i < 7; i++)
  {
    rc = OsSemWait(&tickSem, 300L);
    switch(rc)
    {
      case 0:
```

```
        tputs("OsSemWait successful");
        OsSleep(150L);
        break;

      case TSK_ERR_TIMEOUT:
        tputs("OsSemWait failed, error = TSK_ERR_TIMEOUT");
        break;

      default:
        tprintf("OsSemWait failed, error = %d\n", rc);
        break;
    }

    OsSleep(100L);
  }
}

/*-------------------------------------------------------------*/
void far task1(
    void *        pTaskParam
    )
{
  int             val = (int) pTaskParam;
  int             i;

  OsSemWait(&goSem, OS_INFINITE_WAIT);

  tprintf("init val passed = %d\n", val);

  for(i = 0; i < 3; i++)
  {
    OsSleep(500L);

    tputs("");
  }

  tputs("clearing tickSem");

  OsSemClear(&tickSem);

  OsSleep(1000L);

  tputs("");
}

/*-------------------------------------------------------------*/
void far task2(
```

```
    void *        pTaskParam
    )
{
  int             val = (int) pTaskParam;
  int             i;
  int             j;

  tprintf("init val passed = %d\n", val);

  OsSleep(2000L);

  OsSemClear(&goSem);

  for(i = 0; i < 3; i++)
  {
    OsSleepTicks(18L);
    tputs("");
  }
}

/*-------------------------------------------------------------------*/
void far taskNull(
    void *        pTaskParam
    )
{

  while(killedTasks != numTasks - 1);
}
```

47.2 Good Luck!

Because of the success of Linux, many people are hanging out with the creation of OS. Writing an efficient and neat OS is considered to be tough task because you may need to know more OS fundamentals and hardware details. If you could be able to come out with a new OS, the World would really appreciate you! Good Luck!