HW 3 Hints: Core Dump File Reader

Discussion Session 11/24 & 11/25

Eric Bergstrom

HW 3: Core files

•Signals are generated from the kernel or by the kill() syscall from the user-level.

•Process is terminated if a signal is caught that cannot be handled.

•For the purpose of debugging, the core image of the process will be stored in a file called "**`core`**".

- `data`
- `stack`
- `text`
- `process table entry`

Some signals that generate core files

```
//generates a core dump via SIGSEGV (memory access violation)
 int main ( void ) {
        int array[2];
        array[222222] = 1;
        return 0;
}

// generates a core dump via SIGFPE (floating point exception)
int main ( void ) {

        int i = 0, j = 7;
        j = j / i;
}
```

HW 3 - Core File

`dump_core:` described in lines 18399-18468 (pgs 779-780) under the file `src/mm/signal.c.`

Three items are being dumped in the following order:

- The memory map of all the segments
- The process table entry for the process being terminated
- The data in every segment

/usr/src/mm/signal.c

```
/*===========================================================================*
 *                                 dump_core                                  *
 *===========================================================================*/
PRIVATE void dump_core(rmp)
register struct mproc *rmp;       /* whose core is to be dumped */
{
/* Make a core dump on the file "core", if possible. */

  int fd, fake_fd, nr_written, seg, slot;
  char *buf;
  vir_bytes current_sp;
  phys_bytes left;                /* careful; 64K might overflow vir_bytes */
  unsigned nr_to_write;           /* unsigned for arg to write() but < INT_MAX */
  long trace_data, trace_off;

  slot = (int) (rmp - mproc);

  /* Can core file be written?  We are operating in the user's FS environment,
   * so no special permission checks are needed.
   */
  if (rmp->mp_realuid != rmp->mp_effuid) return;
  if ( (fd = open(core_name, O_WRONLY | O_CREAT | O_TRUNC | O_NONBLOCK,
                                                  CORE_MODE)) < 0) return;
  rmp->mp_sigstatus |= DUMPED;
```

Memory Maps : /usr/include/minix/type.h

```
#ifndef _TYPE_H
#define _TYPE_H
#ifndef _MINIX_TYPE_H
#define _MINIX_TYPE_H

/* Type definitions. */
typedef unsigned int vir_clicks; /* virtual  addresses and lengths in clicks */
typedef unsigned long phys_bytes;/* physical addresses and lengths in bytes */
typedef unsigned int phys_clicks;/* physical addresses and lengths in clicks */

struct mem_map {
  vir_clicks mem_vir;             /* virtual address */
  phys_clicks mem_phys;           /* physical address */
  vir_clicks mem_len;             /* length */
};
```

/usr/src/mm/signal.c

1) The Memory Map of all the segments.

```
/* Make sure the stack segment is up to date.
   * We don't want adjust() to fail unless current_sp is preposterous,
   * but it might fail due to safety checking.  Also, we don't really want
   * the adjust() for sending a signal to fail due to safety checking.
   * Maybe make SAFETY_BYTES a parameter.
   */
  sys_getsp(slot, &current_sp);
  adjust(rmp, rmp->mp_seg[D].mem_len, current_sp);

  /* Write the memory map of all segments to begin the core file. */
  if (write(fd, (char *) rmp->mp_seg, (unsigned) sizeof rmp->mp_seg)
      != (unsigned) sizeof rmp->mp_seg) {
         close(fd);
         return;
  }
```

/usr/src/mm/mproc.h

```
/* This table has one slot per process.  It contains all the memory management
 * information for each process.  Among other things, it defines the text, data
 * and stack segments, uids and gids, and various flags.  The kernel and file
 * systems have tables that are also indexed by process, with the contents
 * of corresponding slots referring to the same process in all three.
 */

EXTERN struct mproc {
  struct mem_map mp_seg[NR_SEGS];/* points to text, data, stack */
  char mp_exitstatus;              /* storage for status when process exits */
  char mp_sigstatus;               /* storage for signal # for killed procs */
  pid_t mp_pid;                    /* process id */
  pid_t mp_procgrp;                /* pid of process group (used for signals) */
  pid_t mp_wpid;          /* pid this process is waiting for */
  int mp_parent;          /* index of parent process */

…code omitted ...

  message mp_reply;                /* reply message to be sent to one */
} mproc[NR_PROCS];
```

/usr/src/mm/signal.c

2) The Process Table Entry of Process being terminated

```
/* Write out the whole kernel process table entry to get the regs. */
trace_off = 0;
while (sys_trace(3, slot, trace_off, &trace_data) == OK) {
        if (write(fd, (char *) &trace_data, (unsigned) sizeof (long))
            != (unsigned) sizeof (long)) {
                close(fd);
                return;
        }
        trace_off += sizeof (long);
}
```

sys_trace() traps to /usr/src/kernel/system.c

```
sys_trace(3, slot, trace_off, &trace_data)

 ==> /usr/src/lib/syslib/sys_trace.c

sys_trace(req, procnr, addr, *data_ptr)
```

invokes a _taskcall and gets handled in system.c: do_trace()

Essentially returns contents of proc entry from kernel space to MM, returned in trace_data.

Trace_offset controls loop, when entire proc structure is copied, it exits loop (reads in one long at a time).

/usr/src/mm/signal.c

3) The data in every segment

```
 /* Loop through segments and write the segments themselves out. */
  for (seg = 0; seg < NR_SEGS; seg++) {
        rw_seg(1, fd, slot, seg,
                 (phys_bytes) rmp->mp_seg[seg].mem_len << CLICK_SHIFT);
  }
  close(fd);
}
```

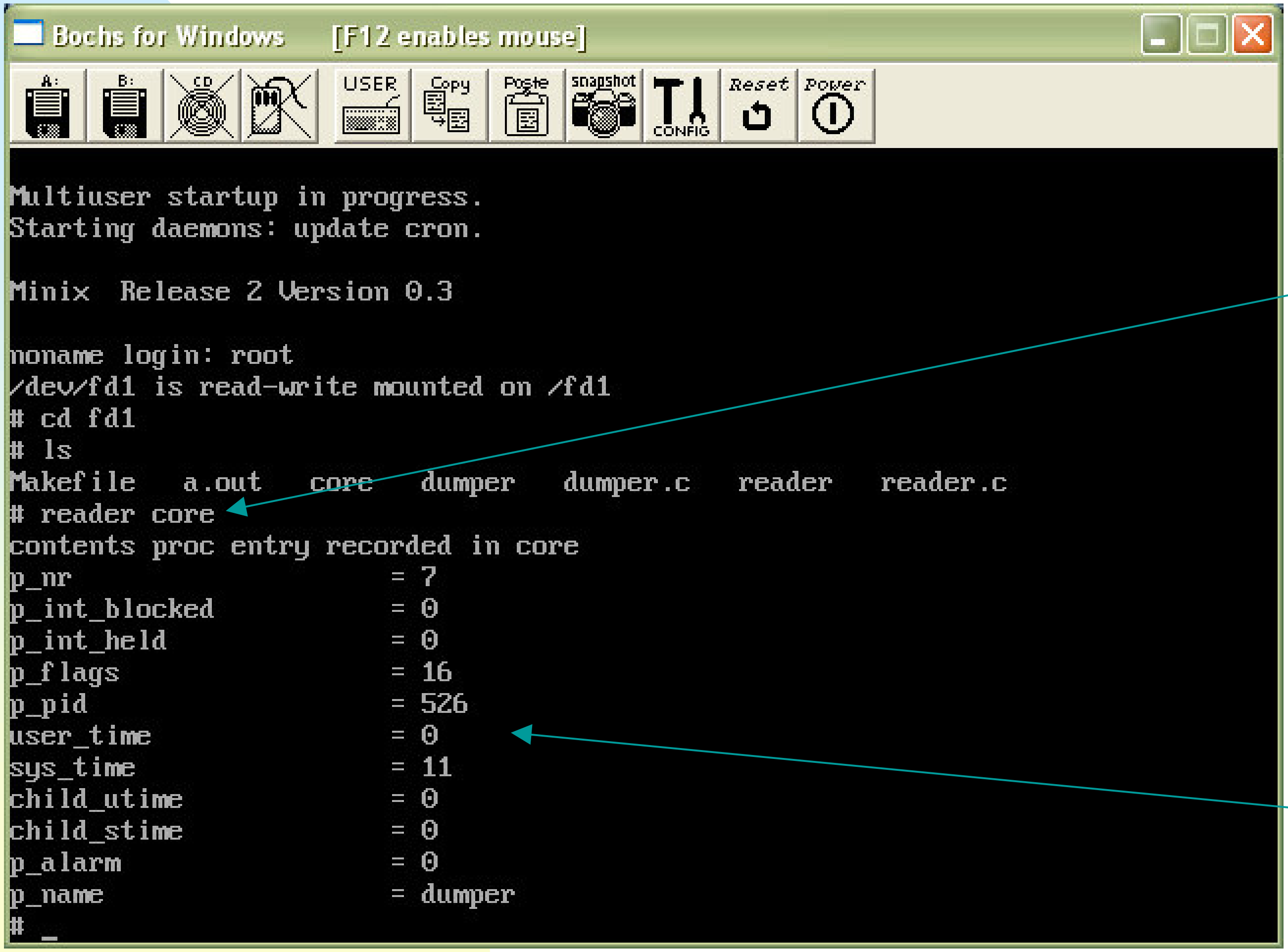
For the assignment:

This data is not required to be displayed.

For the curious: rw_seg is defined /usr/src/mm/exec.c

Found by using grep command: grep rw_seg *

HW 3: Sample Input / Output



```
Multiuser startup in progress.
Starting daemons: update cron.

Minix  Release 2 Version 0.3

noname login: root
/dev/fd1 is read-write mounted on /fd1
# cd fd1
# ls
Makefile   a.out   core   dumper   dumper.c   reader   reader.c
# reader core
contents proc entry recorded in core
p_nr                    = 7
p_int_blocked           = 0
p_int_held              = 0
p_flags                 = 16
p_pid                   = 526
user_time               = 0
sys_time                = 11
child_utime             = 0
child_stime             = 0
p_alarm                 = 0
p_name                  = dumper
# _
```

Pass core filename using command-line argument

Display values from process table entry

Using command-line parameters in C

We will be using a script to test multiple core files in grading so your program needs to take a core filename as input.

Recall:

```
#include <stdio.h>

int main ( int argc, char *argv[] ) {
        printf("Name of executable: %s\n", argv[0] );
        if ( argc == 2 )
                printf("first commandline parameter: %s\n", argv[1] );
        return 0;
}
```

System calls needed

int **open**(char *filename, int flags);

example (opening file for read and write):

```
int fd = open( filename, O_RDWR );
```

int **read**(int fd, void *buf, size_t nbytes);

example (reading into a structure):

```
struct test {
int a;
char b;
}

struct test b;

read( fd, &b, sizeof(struct test));
```

More details on man 2 pages.

HW 3: Approaches

(1) Define the proc structure and mem_map structures within your program.

(2) Or, include the .h files that define the proc structure and mem_map structures (both of these structures require more than 1 header to include constants that are within the headers)

HW 3: Approaches

After reading in a proc structure, print out the appropriate fields as defined in assignment:

`p_nr`

`p_int_blocked`

`p_int_held`

`p_flag   <===` **NOTE**: **p_flag** should be **p_flags**

`p_pid`

`user_time`

`sys_time`

`child_utime`

`child_stime`

`p_alarm`

`p_name[16]`

HW 3: Approaches

Sample output for a process called mem_violation that dumped into a file called **core**.

```
# cc dumper.c  -o mem_violation
# mem_violation
Memory fault - core dumped
* reader core
contents proc entry recorded in core
p_nr           = 7
p_int_blocked      = 0
p_int_held       = 0
p_flags         = 16
p_pid          = 351
user_time        = 1
sys_time        = 10
child_utime       = 0
child_stime       = 0
p_alarm         = 0
p_name          = mem_violation
#
```