```
**********************************************************
   15454 Fri Aug 19 10:01:36 2016
new/usr/src/uts/common/exec/elf/elf_notes.c
7307 Fixing 5780 introduced a regression
**********************************************************
______unchanged_portion_omitted_

 166 int
 167 write_elfnotes(proc_t *p, int sig, vnode_t *vp, offset_t offset,
 168     rlim64_t rlimit, cred_t *credp, core_content_t content)
 169 {
 170         union {
 171                 psinfo_t        psinfo;
 172                 pstatus_t       pstatus;
 173                 lwpsinfo_t      lwpsinfo;
 174                 lwpstatus_t     lwpstatus;
 175 #if defined(__sparc)
 176                 gwindows_t      gwindows;
 177                 asrset_t        asrset;
 178 #endif /* __sparc */
 179                 char            xregs[1];
 180                 aux_entry_t     auxv[__KERN_NAUXV_IMPL];
 181                 prcred_t        pcred;
 182                 prpriv_t        ppriv;
 183                 priv_impl_info_t prinfo;
 184                 struct utsname  uts;
 185         } *bigwad;

 187         size_t xregsize = prhasx(p)? prgetprxregsize(p) : 0;
 188         size_t crsize = sizeof (prcred_t) + sizeof (gid_t) * (ngroups_max - 1);
 189         size_t psize = prgetprivsize();
 190         size_t bigsize = MAX(psize, MAX(sizeof (*bigwad),
 191             MAX(xregsize, crsize)));

 193         priv_impl_info_t *prii;

 195         lwpdir_t *ldp;
 196         lwpent_t *lep;
 197         kthread_t *t;
 198         klwp_t *lwp;
 199         user_t *up;
 200         int i;
 201         int nlwp;
 202         int nzomb;
 203         int error;
 204         uchar_t oldsig;
 205         uf_info_t *fip;
 206         int fd;
 207         vnode_t *vroot;

 209 #if defined(__i386) || defined(__i386_COMPAT)
 210         struct ssd *ssd;
 211         size_t ssdsize;
 212 #endif  /* __i386 || __i386_COMPAT */

 214         bigsize = MAX(bigsize, priv_get_implinfo_size());

 216         bigwad = kmem_alloc(bigsize, KM_SLEEP);

 218         /*
 219          * The order of the elfnote entries should be same here
 220          * and in the gcore(1) command.  Synchronization is
 221          * needed between the kernel and gcore(1).
 222          */

 224         /*
```



```
 225          * Get the psinfo, and set the wait status to indicate that a core was
 226          * dumped.  We have to forge this since p->p_wcode is not set yet.
 227          */
 228         mutex_enter(&p->p_lock);
 229         prgetpsinfo(p, &bigwad->psinfo);
 230         mutex_exit(&p->p_lock);
 231         bigwad->psinfo.pr_wstat = wstat(CLD_DUMPED, sig);

 233         error = elfnote(vp, &offset, NT_PSINFO, sizeof (bigwad->psinfo),
 234             (caddr_t)&bigwad->psinfo, rlimit, credp);
 235         if (error)
 236                 goto done;

 238         /*
 239          * Modify t_whystop and lwp_cursig so it appears that the current LWP
 240          * is stopped after faulting on the signal that caused the core dump.
 241          * As a result, prgetstatus() will record that signal, the saved
 242          * lwp_siginfo, and its signal handler in the core file status.  We
 243          * restore lwp_cursig in case a subsequent signal was received while
 244          * dumping core.
 245          */
 246         mutex_enter(&p->p_lock);
 247         lwp = ttolwp(curthread);

 249         oldsig = lwp->lwp_cursig;
 250         lwp->lwp_cursig = (uchar_t)sig;
 251         curthread->t_whystop = PR_FAULTED;

 253         prgetstatus(p, &bigwad->pstatus, p->p_zone);
 254         bigwad->pstatus.pr_lwp.pr_why = 0;

 256         curthread->t_whystop = 0;
 257         lwp->lwp_cursig = oldsig;
 258         mutex_exit(&p->p_lock);

 260         error = elfnote(vp, &offset, NT_PSTATUS, sizeof (bigwad->pstatus),
 261             (caddr_t)&bigwad->pstatus, rlimit, credp);
 262         if (error)
 263                 goto done;

 265         error = elfnote(vp, &offset, NT_PLATFORM, strlen(platform) + 1,
 266             platform, rlimit, credp);
 267         if (error)
 268                 goto done;

 270         up = PTOU(p);
 271         for (i = 0; i < __KERN_NAUXV_IMPL; i++) {
 272                 bigwad->auxv[i].a_type = up->u_auxv[i].a_type;
 273                 bigwad->auxv[i].a_un.a_val = up->u_auxv[i].a_un.a_val;
 274         }
 275         error = elfnote(vp, &offset, NT_AUXV, sizeof (bigwad->auxv),
 276             (caddr_t)bigwad->auxv, rlimit, credp);
 277         if (error)
 278                 goto done;

 280         bcopy(&utsname, &bigwad->uts, sizeof (struct utsname));
 281         if (!INGLOBALZONE(p)) {
 282                 bcopy(p->p_zone->zone_nodename, &bigwad->uts.nodename,
 283                     _SYS_NMLN);
 284         }
 285         error = elfnote(vp, &offset, NT_UTSNAME, sizeof (struct utsname),
 286             (caddr_t)&bigwad->uts, rlimit, credp);
 287         if (error)
 288                 goto done;

 290         prgetcred(p, &bigwad->pcred);
```

```
292         if (bigwad->pcred.pr_ngroups != 0) {
293                 crsize = sizeof (prcred_t) +
294                     sizeof (gid_t) * (bigwad->pcred.pr_ngroups - 1);
295         } else
296                 crsize = sizeof (prcred_t);

298         error = elfnote(vp, &offset, NT_PRCRED, crsize,
299             (caddr_t)&bigwad->pcred, rlimit, credp);
300         if (error)
301                 goto done;

303         error = elfnote(vp, &offset, NT_CONTENT, sizeof (core_content_t),
304             (caddr_t)&content, rlimit, credp);
305         if (error)
306                 goto done;

308         prgetpriv(p, &bigwad->ppriv);

310         error = elfnote(vp, &offset, NT_PRPRIV, psize,
311             (caddr_t)&bigwad->ppriv, rlimit, credp);
312         if (error)
313                 goto done;

315         prii = priv_hold_implinfo();
316         error = elfnote(vp, &offset, NT_PRPRIVINFO, priv_get_implinfo_size(),
317             (caddr_t)prii, rlimit, credp);
318         priv_release_implinfo();
319         if (error)
320                 goto done;

322         /* zone can't go away as long as process exists */
323         error = elfnote(vp, &offset, NT_ZONENAME,
324             strlen(p->p_zone->zone_name) + 1, p->p_zone->zone_name,
325             rlimit, credp);
326         if (error)
327                 goto done;


330         /* open file table */
331         vroot = PTOU(p)->u_rdir;
332         if (vroot == NULL)
333                 vroot = rootdir;

335         VN_HOLD(vroot);

337         fip = P_FINFO(p);

339         for (fd = 0; fd < fip->fi_nfiles; fd++) {
340                 uf_entry_t *ufp;
341                 vnode_t *fvp;
342                 struct file *fp;
343                 vattr_t vattr;
344                 prfdinfo_t fdinfo;

346                 bzero(&fdinfo, sizeof (fdinfo));

348                 mutex_enter(&fip->fi_lock);
349                 UF_ENTER(ufp, fip, fd);
350                 if (((fp = ufp->uf_file) == NULL) || (fp->f_count < 1)) {
351                         UF_EXIT(ufp);
352                         mutex_exit(&fip->fi_lock);
353                         continue;
354                 }

356                 fdinfo.pr_fd = fd;
```



```
357                 fdinfo.pr_fdflags = ufp->uf_flag;
358                 fdinfo.pr_fileflags = fp->f_flag2;
359                 fdinfo.pr_fileflags <<= 16;
360                 fdinfo.pr_fileflags |= fp->f_flag;
361                 if ((fdinfo.pr_fileflags & (FSEARCH | FEXEC)) == 0)
362                         fdinfo.pr_fileflags += FOPEN;
363                 fdinfo.pr_offset = fp->f_offset;


366                 fvp = fp->f_vnode;
367                 VN_HOLD(fvp);
368                 UF_EXIT(ufp);
369                 mutex_exit(&fip->fi_lock);

371                 /*
372                  * There are some vnodes that have no corresponding
373                  * path.  Its reasonable for this to fail, in which
374                  * case the path will remain an empty string.
375                  */
376                 (void) vnodetopath(vroot, fvp, fdinfo.pr_path,
377                     sizeof (fdinfo.pr_path), credp);

379                 if (VOP_GETATTR(fvp, &vattr, 0, credp, NULL) != 0) {
380                         /*
381                          * Try to write at least a subset of information
382                          */
383                         fdinfo.pr_major = 0;
384                         fdinfo.pr_minor = 0;
385                         fdinfo.pr_ino = 0;
386                         fdinfo.pr_mode = 0;
387                         fdinfo.pr_uid = (uid_t)-1;
388                         fdinfo.pr_gid = (gid_t)-1;
389                         fdinfo.pr_rmajor = 0;
390                         fdinfo.pr_rminor = 0;
391                         fdinfo.pr_size = -1;

393                         error = elfnote(vp, &offset, NT_FDINFO,
394                             sizeof (fdinfo), &fdinfo, rlimit, credp);
395                         VN_RELE(fvp);
396                         if (error) {
397 #endif /* ! codereview */
398                                 VN_RELE(vroot);
396                         if (error)
399                                 goto done;
400                         }
401 #endif /* ! codereview */
402                         continue;
403                 }

405                 if (fvp->v_type == VSOCK)
406                         fdinfo.pr_fileflags |= sock_getfasync(fvp);

408                 VN_RELE(fvp);

410                 /*
411                  * This logic mirrors fstat(), which we cannot use
412                  * directly, as it calls copyout().
413                  */
414                 fdinfo.pr_major = getmajor(vattr.va_fsid);
415                 fdinfo.pr_minor = getminor(vattr.va_fsid);
416                 fdinfo.pr_ino = (ino64_t)vattr.va_nodeid;
417                 fdinfo.pr_mode = VTTOIF(vattr.va_type) | vattr.va_mode;
418                 fdinfo.pr_uid = vattr.va_uid;
419                 fdinfo.pr_gid = vattr.va_gid;
420                 fdinfo.pr_rmajor = getmajor(vattr.va_rdev);
421                 fdinfo.pr_rminor = getminor(vattr.va_rdev);
```

```
422                 fdinfo.pr_size = (off64_t)vattr.va_size;

424                 error = elfnote(vp, &offset, NT_FDINFO,
425                     sizeof (fdinfo), &fdinfo, rlimit, credp);
426                 if (error) {
427                         VN_RELE(vroot);
428                         goto done;
429                 }
430         }

432         VN_RELE(vroot);

434 #if defined(__i386) || defined(__i386_COMPAT)
435         mutex_enter(&p->p_ldtlock);
436         ssdsize = prnldt(p) * sizeof (struct ssd);
437         if (ssdsize != 0) {
438                 ssd = kmem_alloc(ssdsize, KM_SLEEP);
439                 prgetldt(p, ssd);
440                 error = elfnote(vp, &offset, NT_LDT, ssdsize,
441                     (caddr_t)ssd, rlimit, credp);
442                 kmem_free(ssd, ssdsize);
443         }
444         mutex_exit(&p->p_ldtlock);
445         if (error)
446                 goto done;
447 #endif  /* __i386 || defined(__i386_COMPAT) */

449         nlwp = p->p_lwpcnt;
450         nzomb = p->p_zombcnt;
451         /* for each entry in the lwp directory ... */
452         for (ldp = p->p_lwpdir; nlwp + nzomb != 0; ldp++) {

454                 if ((lep = ldp->ld_entry) == NULL)      /* empty slot */
455                         continue;

457                 if ((t = lep->le_thread) != NULL) {     /* active lwp */
458                         ASSERT(nlwp != 0);
459                         nlwp--;
460                         lwp = ttolwp(t);
461                         mutex_enter(&p->p_lock);
462                         prgetlwpsinfo(t, &bigwad->lwpsinfo);
463                         mutex_exit(&p->p_lock);
464                 } else {                                /* zombie lwp */
465                         ASSERT(nzomb != 0);
466                         nzomb--;
467                         bzero(&bigwad->lwpsinfo, sizeof (bigwad->lwpsinfo));
468                         bigwad->lwpsinfo.pr_lwpid = lep->le_lwpid;
469                         bigwad->lwpsinfo.pr_state = SZOMB;
470                         bigwad->lwpsinfo.pr_sname = 'Z';
471                         bigwad->lwpsinfo.pr_start.tv_sec = lep->le_start;
472                 }
473                 error = elfnote(vp, &offset, NT_LWPSINFO,
474                     sizeof (bigwad->lwpsinfo), (caddr_t)&bigwad->lwpsinfo,
475                     rlimit, credp);
476                 if (error)
477                         goto done;
478                 if (t == NULL)          /* nothing more to do for a zombie */
479                         continue;

481                 mutex_enter(&p->p_lock);
482                 if (t == curthread) {
483                         /*
484                          * Modify t_whystop and lwp_cursig so it appears that
485                          * the current LWP is stopped after faulting on the
486                          * signal that caused the core dump.  As a result,
487                          * prgetlwpstatus() will record that signal, the saved
```



```
488                          * lwp_siginfo, and its signal handler in the core file
489                          * status.  We restore lwp_cursig in case a subsequent
490                          * signal was received while dumping core.
491                          */
492                         oldsig = lwp->lwp_cursig;
493                         lwp->lwp_cursig = (uchar_t)sig;
494                         t->t_whystop = PR_FAULTED;

496                         prgetlwpstatus(t, &bigwad->lwpstatus, p->p_zone);
497                         bigwad->lwpstatus.pr_why = 0;

499                         t->t_whystop = 0;
500                         lwp->lwp_cursig = oldsig;
501                 } else {
502                         prgetlwpstatus(t, &bigwad->lwpstatus, p->p_zone);
503                 }
504                 mutex_exit(&p->p_lock);
505                 error = elfnote(vp, &offset, NT_LWPSTATUS,
506                     sizeof (bigwad->lwpstatus), (caddr_t)&bigwad->lwpstatus,
507                     rlimit, credp);
508                 if (error)
509                         goto done;

511 #if defined(__sparc)
512                 /*
513                  * Unspilled SPARC register windows.
514                  */
515                 {
516                         size_t size = prnwindows(lwp);

518                         if (size != 0) {
519                                 size = sizeof (gwindows_t) -
520                                     (SPARC_MAXREGWINDOW - size) *
521                                     sizeof (struct rwindow);
522                                 prgetwindows(lwp, &bigwad->gwindows);
523                                 error = elfnote(vp, &offset, NT_GWINDOWS,
524                                     size, (caddr_t)&bigwad->gwindows,
525                                     rlimit, credp);
526                                 if (error)
527                                         goto done;
528                         }
529                 }
530                 /*
531                  * Ancillary State Registers.
532                  */
533                 if (p->p_model == DATAMODEL_LP64) {
534                         prgetasregs(lwp, bigwad->asrset);
535                         error = elfnote(vp, &offset, NT_ASRS,
536                             sizeof (asrset_t), (caddr_t)bigwad->asrset,
537                             rlimit, credp);
538                         if (error)
539                                 goto done;
540                 }
541 #endif /* __sparc */

543                 if (xregsize) {
544                         prgetprxregs(lwp, bigwad->xregs);
545                         error = elfnote(vp, &offset, NT_PRXREG,
546                             xregsize, bigwad->xregs, rlimit, credp);
547                         if (error)
548                                 goto done;
549                 }

551                 if (t->t_lwp->lwp_spymaster != NULL) {
552                         void *psaddr = t->t_lwp->lwp_spymaster;
553 #ifdef _ELF32_COMPAT
```

```
554                         /*
555                          * On a 64-bit kernel with 32-bit ELF compatibility,
556                          * this file is compiled into two different objects:
557                          * one is compiled normally, and the other is compiled
558                          * with _ELF32_COMPAT set -- and therefore with a
559                          * psinfo_t defined to be a psinfo32_t.  However, the
560                          * psinfo_t denoting our spymaster is always of the
561                          * native type; if we are in the _ELF32_COMPAT case,
562                          * we need to explicitly convert it.
563                          */
564                         if (p->p_model == DATAMODEL_ILP32) {
565                                 psinfo_kto32(psaddr, &bigwad->psinfo);
566                                 psaddr = &bigwad->psinfo;
567                         }
568 #endif

570                         error = elfnote(vp, &offset, NT_SPYMASTER,
571                             sizeof (psinfo_t), psaddr, rlimit, credp);
572                         if (error)
573                                 goto done;
574                 }
575         }
576         ASSERT(nlwp == 0);

578 done:
579         kmem_free(bigwad, bigsize);
580         return (error);
581 }
```