```
**********************************************************
    4433 Thu Nov  1 17:19:26 2012
new/usr/src/tools/findunref/findunref.1
3272 findunref should support git
**********************************************************
```

```
.\" CDDL HEADER START
.\"
.\" The contents of this file are subject to the terms of the
.\" Common Development and Distribution License (the "License").
.\" You may not use this file except in compliance with the License.
.\"
.\" You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
.\" or http://www.opensolaris.org/os/licensing.
.\" See the License for the specific language governing permissions
.\" and limitations under the License.
.\"
.\" When distributing Covered Code, include this CDDL HEADER in each
.\" file and include the License file at usr/src/OPENSOLARIS.LICENSE.
.\" If applicable, add the following below this CDDL HEADER, with the
.\" fields enclosed by brackets "[]" replaced with your own identifying
.\" information: Portions Copyright [yyyy] [name of copyright owner]
.\"
.\" CDDL HEADER END
.\"
.\" Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
.\" Use is subject to license terms.
.TH findunref 1 "Oct 30, 2012"
```

```
.\" " CDDL HEADER START
.\" "
.\" " The contents of this file are subject to the terms of the
.\" " Common Development and Distribution License (the "License").
.\" " You may not use this file except in compliance with the License.
.\" "
.\" " You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
.\" " or http://www.opensolaris.org/os/licensing.
.\" " See the License for the specific language governing permissions
.\" " and limitations under the License.
.\" "
.\" " When distributing Covered Code, include this CDDL HEADER in each
.\" " file and include the License file at usr/src/OPENSOLARIS.LICENSE.
.\" " If applicable, add the following below this CDDL HEADER, with the
.\" " fields enclosed by brackets "[]" replaced with your own identifying
.\" " information: Portions Copyright [yyyy] [name of copyright owner]
.\" "
.\" " CDDL HEADER END
.\" "
.\" "Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
.\" "Use is subject to license terms.
.TH findunref 1 "11 Aug 2009"
.I findunref
\- find unused files in a source tree
.SH SYNOPSIS
findunref [\fB-s\fP \fIsubtree\fP] [\fB-t\fP \fItstampfile\fP]
[\fB-S\fP \fBhg\fP|\fBtw\fP|\fBgit\fP] \fIsrcroot\fP \fIexceptfile\fP
[\fB-S\fP \fBhg\fP|\fBtw\fP] \fIsrcroot\fP \fIexceptfile\fP
.LP
.SH DESCRIPTION
.IX "OS-Net build tools" "findunref" "" "\fBfindunref\fP"
.LP
The findunref utility lists the files in a source tree which have not been
accessed more recently than a particular timestamp file.  Although
findunref may be used on its own, it is usually invoked by
\fBnightly\fP(1) to find files that are never referenced during a given
build (see \fB-f\fP in \fBnightly\fP(1)).
.LP
The root of the source tree to examine is specified by \fIsrcroot\fP.  To
```



```
simplify comparing findunref output from different source trees, findunref
outputs all filenames relative to \fIsrcroot\fP.
.LP
Some files in a source tree may be intentionally unreferenced (e.g.,
documentation) or only referenced during specialized types of builds.
Accordingly, \fIexceptfile\fP names a file containing a list of pathname
globs that will be ignored by findunref.  Within \fIexceptfile\fP, any
lines consisting solely of whitespace or starting with \fB#\fP will be
ignored.  Directory globs may also be specified, which will cause any
matching directories to be skipped entirely.  If no exceptions are
desired, \fIexceptfile\fP can be \fB/dev/null\fP.
.LP
Depending on how findunref is invoked, it can either check all files, or
limit its checks to files under control of a specific source code
management (SCM) system.
.LP
To limit checks to files managed by Mercurial, the \fBhg\fP(1) utility must be
present in \fB$PATH\fP and any relevant repositories must be located at or
under \fIsrcroot\fP.  Nested Mercurial repositories are supported.
.LP
To limit checks to files managed by Git, the \fBgit\fP(1) utility must be
present in \fB$PATH\fP and any relevant repositories must be located at or
under \fIsrcroot\fP.  Nested Git repositories are \fInot\fR supported.
management (SCM) system.  To limit checks to files managed by Mercurial,
the \fBhg\fP(1) utility must be present in \fB/usr/bin\fP and any relevant
repositories must be located at or under \fIsrcroot\fP.  Nested Mercurial
repositories are supported.
.SH OPTIONS
.TP 10
.B -s \fIsubtree\fP
Only look under \fIsubtree\fP for unreferenced files.  By default, all
directories under \fIsrcroot\fP are examined.
.TP 10
.B -t \fItstampfile\fP
Consider files older than \fItstampfile\fP to be unreferenced.
By default, \fIsrcroot\fB/.build.tstamp\fR is used.
.TP 10
.B -S \fBhg\fP|\fBtw\fP|\fBgit\fP
.B -S \fBhg\fP|\fBtw\fP
Only check files that are managed by the specified SCM.  To simplify
interaction with \fBwhich_scm\fP(1), the SCM names "mercurial" and
"teamware" may also be specified for "hg" and "tw", respectively.
By default, all files are checked.
"teamware" may also be specified.  By default, all files are checked.
.SH SEE ALSO
.LP
\fBgit\fP(1),
#endif /* ! codereview */
\fBhg\fP(1),
\fBnightly\fP(1),
\fBwhich_scm\fP(1)
.SH NOTES
Since many files are only used when building for a particular ISA (e.g.,
Makefiles that are specific to x86 or SPARC), builds must be done on all
applicable ISAs and the results merged.  For instance, if nightly builds
(with \fB-f\fP) are done on both SPARC and x86, \fBusr/src\fP will be
populated with a corresponding \fBunref-\fIisa\fB.out\fR file, which can
be merged with \fBcomm\fP(1):
.LP
.nf
comm -12 /path/to/unref-i386.out
         /path/to/unref-sparc.out > unref.out
.fi
.LP
This merged file can then be compared against the gate's latest
unreferenced file list (e.g. \fB/ws/onnv-gate/usr/src/unrefmaster.out\fP).
```

```
 99 .LP
100 Different gates have different unreferenced file policies.  Any changes to
101 \fIexceptfile\fP that would define new unreferenced file policies for a
102 given gate must be cleared with the appropriate gatekeepers.
```

```
**********************************************************
   13045 Thu Nov  1 17:19:27 2012
new/usr/src/tools/findunref/findunref.c
3272 findunref should support git
**********************************************************
______unchanged_portion_omitted_

 79 static checkscm_func_t check_tw, check_hg, check_git;
 80 static chdirscm_func_t chdir_hg, chdir_git;
 79 static checkscm_func_t check_tw, check_hg;
 80 static chdirscm_func_t chdir_hg;
 81 static int      pset_add(pset_t *, const char *);
 82 static int      pset_check(const pset_t *, const char *);
 83 static void     pset_empty(pset_t *);
 84 static void     pset_free(pset_t *);
 85 static int      checkpath(const char *, const struct stat *, int, struct FTW *);
 86 static pset_t  *make_exset(const char *);
 87 static void     warn(const char *, ...);
 88 static void     die(const char *, ...);

 90 static const scm_t scms[] = {
 91         { "tw",         check_tw,       NULL            },
 92         { "teamware",   check_tw,       NULL            },
 93         { "hg",         check_hg,       chdir_hg        },
 94         { "mercurial",  check_hg,       chdir_hg        },
 95         { "git",        check_git,      chdir_git       },
 96 #endif /* ! codereview */
 97         { NULL,         NULL,           NULL            }
 98 };

100 static const scm_t      *scm;
101 static hgdata_t         hgdata;
102 static pnset_t          *gitmanifest = NULL;
103 #endif /* ! codereview */
104 static time_t           tstamp;         /* timestamp to compare files to */
105 static pnset_t          *exsetp;        /* pathname globs to ignore */
106 static const char       *progname;

108 int
109 main(int argc, char *argv[])
110 {
111         int c;
112         char path[MAXPATHLEN];
113         char subtree[MAXPATHLEN] = "./";
114         char *tstampfile = ".build.tstamp";
115         struct stat tsstat;

117         progname = strrchr(argv[0], '/');
118         if (progname == NULL)
119                 progname = argv[0];
120         else
121                 progname++;

123         while ((c = getopt(argc, argv, "as:t:S:")) != EOF) {
124                 switch (c) {
125                 case 'a':
126                         /* for compatibility; now the default */
127                         break;

129                 case 's':
130                         (void) strlcat(subtree, optarg, MAXPATHLEN);
131                         break;

133                 case 't':
134                         tstampfile = optarg;
135                         break;
```



```
137                 case 'S':
138                         for (scm = scms; scm->name != NULL; scm++) {
139                                 if (strcmp(scm->name, optarg) == 0)
140                                         break;
141                         }
142                         if (scm->name == NULL)
143                                 die("unsupported SCM '%s'\n", optarg);
144                         break;

146                 default:
147                 case '?':
148                         goto usage;
149                 }
150         }

152         argc -= optind;
153         argv += optind;

155         if (argc != 2) {
156 usage:          (void) fprintf(stderr, "usage: %s [-s <subtree>] "
157                     "[-t <tstampfile>] [-S hg|tw|git] <srcroot> <exceptfile>\n",
 95                     "[-t <tstampfile>] [-S hg|tw] <srcroot> <exceptfile>\n",
158                     progname);
159                 return (EXIT_FAILURE);
160         }

162         /*
163          * Interpret a relative timestamp path as relative to srcroot.
164          */
165         if (tstampfile[0] == '/')
166                 (void) strlcpy(path, tstampfile, MAXPATHLEN);
167         else
168                 (void) snprintf(path, MAXPATHLEN, "%s/%s", argv[0], tstampfile);

170         if (stat(path, &tsstat) == -1)
171                 die("cannot stat timestamp file \"%s\"", path);
172         tstamp = tsstat.st_mtime;

174         /*
175          * Create the exception pathname set.
176          */
177         exsetp = make_exset(argv[1]);
178         if (exsetp == NULL)
179                 die("cannot make exception pathname set\n");

181         /*
182          * Walk the specified subtree of the tree rooted at argv[0].
183          */
184         if (chdir(argv[0]) == -1)
185                 die("cannot change directory to \"%s\"", argv[0]);

187         if (nftw(subtree, checkpath, 100, FTW_PHYS) != 0)
188                 die("cannot walk tree rooted at \"%s\"\n", argv[0]);

190         pnset_empty(exsetp);
191         return (EXIT_SUCCESS);
192 }

194 /*
195  * Load and return a pnset for the manifest for the Mercurial repo at 'hgroot'.
196  */
197 static pnset_t *
198 load_manifest(const char *hgroot)
199 {
200         FILE    *fp = NULL;
```

```
201         char    *hgcmd = NULL;
202         char    *newline;
203         pnset_t *pnsetp;
204         char    path[MAXPATHLEN];

206         pnsetp = calloc(sizeof (pnset_t), 1);
207         if (pnsetp == NULL ||
208             asprintf(&hgcmd, "hg manifest -R %s", hgroot) == -1)
146             asprintf(&hgcmd, "/usr/bin/hg manifest -R %s", hgroot) == -1)
209                 goto fail;

211         fp = popen(hgcmd, "r");
212         if (fp == NULL)
213                 goto fail;

215         while (fgets(path, sizeof (path), fp) != NULL) {
216                 newline = strrchr(path, '\n');
217                 if (newline != NULL)
218                         *newline = '\0';

220                 if (pnset_add(pnsetp, path) == 0)
221                         goto fail;
222         }

224         (void) pclose(fp);
225         free(hgcmd);
226         return (pnsetp);
227 fail:
228         warn("cannot load hg manifest at %s", hgroot);
229         if (fp != NULL)
230                 (void) pclose(fp);
231         free(hgcmd);
232         pnset_free(pnsetp);
233         return (NULL);
234 }

236 static void
237 chdir_git(const char *path)
238 {
239         FILE *fp = NULL;
240         char *gitcmd = NULL;
241         char *newline;
242         char fn[MAXPATHLEN];
243         pnset_t *pnsetp;

245         pnsetp = calloc(sizeof (pnset_t), 1);
246         if ((pnsetp == NULL) ||
247             (asprintf(&gitcmd, "git ls-files %s", path) == -1))
248                 goto fail;

250         if ((fp = popen(gitcmd, "r")) == NULL)
251                 goto fail;

253         while (fgets(fn, sizeof (fn), fp) != NULL) {
254                 if ((newline = strrchr(fn, '\n')) != NULL)
255                         *newline = '\0';

257                 if (pnset_add(pnsetp, fn) == 0)
258                         goto fail;
259         }

261         (void) pclose(fp);
262         free(gitcmd);
263         gitmanifest = pnsetp;
264         return;
265 fail:
```



```
266         warn("cannot load git manifest");
267         if (fp != NULL)
268                 (void) pclose(fp);
269         if (pnsetp != NULL)
270                 free(pnsetp);
271         if (gitcmd != NULL)
272                 free(gitcmd);
273 }

275 #endif /* ! codereview */
276 /*
277  * If necessary, change our active manifest to be appropriate for 'path'.
278  */
279 static void
280 chdir_hg(const char *path)
281 {
282         char hgpath[MAXPATHLEN];
283         char basepath[MAXPATHLEN];
284         char *slash;

286         (void) snprintf(hgpath, MAXPATHLEN, "%s/.hg", path);

288         /*
289          * Change our active manifest if any one of the following is true:
290          *
291          *   1. No manifest is loaded.  Find the nearest hgroot to load from.
292          *
293          *   2. A manifest is loaded, but we've moved into a directory with
294          *      its own hgroot (e.g., usr/closed).  Load from its hgroot.
295          *
296          *   3. A manifest is loaded, but no longer applies (e.g., the manifest
297          *      under usr/closed is loaded, but we've moved to usr/src).
298          */
299         if (hgdata.manifest == NULL ||
300             strcmp(hgpath, hgdata.hgpath) != 0 && access(hgpath, X_OK) == 0 ||
301             strncmp(path, hgdata.root, hgdata.rootlen - 1) != 0) {
302                 pnset_free(hgdata.manifest);
303                 hgdata.manifest = NULL;

305                 (void) strlcpy(basepath, path, MAXPATHLEN);

307                 /*
308                  * Walk up the directory tree looking for .hg subdirectories.
309                  */
310                 while (access(hgpath, X_OK) == -1) {
311                         slash = strrchr(basepath, '/');
312                         if (slash == NULL) {
313                                 if (!hgdata.rootwarn) {
314                                         warn("no hg root for \"%s\"\n", path);
315                                         hgdata.rootwarn = B_TRUE;
316                                 }
317                                 return;
318                         }
319                         *slash = '\0';
320                         (void) snprintf(hgpath, MAXPATHLEN, "%s/.hg", basepath);
321                 }

323                 /*
324                  * We found a directory with an .hg subdirectory; record it
325                  * and load its manifest.
326                  */
327                 (void) strlcpy(hgdata.hgpath, hgpath, MAXPATHLEN);
328                 (void) strlcpy(hgdata.root, basepath, MAXPATHLEN);
329                 hgdata.manifest = load_manifest(hgdata.root);

331                 /*
```

```
332                  * The logic in check_hg() depends on hgdata.root having a
333                  * single trailing slash, so only add it if it's missing.
334                  */
335                 if (hgdata.root[strlen(hgdata.root) - 1] != '/')
336                         (void) strlcat(hgdata.root, "/", MAXPATHLEN);
337                 hgdata.rootlen = strlen(hgdata.root);
338         }
339 }

341 /*
342  * Check if a file is under Mercurial control by checking against the manifest.
343  */
344 /* ARGSUSED */
345 static int
346 check_hg(const char *path, const struct FTW *ftwp)
347 {
348         /*
349          * The manifest paths are relative to the manifest root; skip past it.
350          */
351         path += hgdata.rootlen;

353         return (hgdata.manifest != NULL && pnset_check(hgdata.manifest, path));
354 }
355 /* ARGSUSED */
356 static int
357 check_git(const char *path, const struct FTW *ftwp)
358 {
359         path += 2;              /* Skip "./" */
360         return (gitmanifest != NULL && pnset_check(gitmanifest, path));
361 }
362 #endif /* ! codereview */

364 /*
365  * Check if a file is under TeamWare control by checking for its corresponding
366  * SCCS "s-dot" file.
367  */
368 static int
369 check_tw(const char *path, const struct FTW *ftwp)
370 {
371         char sccspath[MAXPATHLEN];

373         (void) snprintf(sccspath, MAXPATHLEN, "%.*s/SCCS/s.%s", ftwp->base,
374             path, path + ftwp->base);

376         return (access(sccspath, F_OK) == 0);
377 }

379 /*
380  * Using 'exceptfile' and a built-in list of exceptions, build and return a
381  * pnset_t consisting of all of the pathnames globs which are allowed to be
382  * unreferenced in the source tree.
383  */
384 static pnset_t *
385 make_exset(const char *exceptfile)
386 {
387         FILE            *fp;
388         char            line[MAXPATHLEN];
389         char            *newline;
390         pnset_t         *pnsetp;
391         unsigned int    i;

393         pnsetp = calloc(sizeof (pnset_t), 1);
394         if (pnsetp == NULL)
395                 return (NULL);

397         /*
```



```
398          * Add any exceptions from the file.
399          */
400         fp = fopen(exceptfile, "r");
401         if (fp == NULL) {
402                 warn("cannot open exception file \"%s\"", exceptfile);
403                 goto fail;
404         }

406         while (fgets(line, sizeof (line), fp) != NULL) {
407                 newline = strrchr(line, '\n');
408                 if (newline != NULL)
409                         *newline = '\0';

411                 for (i = 0; isspace(line[i]); i++)
412                         ;

414                 if (line[i] == '#' || line[i] == '\0')
415                         continue;

417                 if (pnset_add(pnsetp, line) == 0) {
418                         (void) fclose(fp);
419                         goto fail;
420                 }
421         }

423         (void) fclose(fp);
424         return (pnsetp);
425 fail:
426         pnset_free(pnsetp);
427         return (NULL);
428 }

430 /*
431  * FTW callback: print 'path' if it's older than 'tstamp' and not in 'exsetp'.
432  */
433 static int
434 checkpath(const char *path, const struct stat *statp, int type,
435     struct FTW *ftwp)
436 {
437         switch (type) {
438         case FTW_F:
439                 /*
440                  * Skip if the file is referenced or in the exception list.
441                  */
442                 if (statp->st_atime >= tstamp || pnset_check(exsetp, path))
443                         return (0);

445                 /*
446                  * If requested, restrict ourselves to unreferenced files
447                  * under SCM control.
448                  */
449                 if (scm == NULL || scm->checkfunc(path, ftwp))
450                         (void) puts(path);
451                 return (0);

453         case FTW_D:
454                 /*
455                  * Prune any directories in the exception list.
456                  */
457                 if (pnset_check(exsetp, path)) {
458                         ftwp->quit = FTW_PRUNE;
459                         return (0);
460                 }

462                 /*
463                  * If necessary, advise the SCM logic of our new directory.
```

```
464                  */
465                 if (scm != NULL && scm->chdirfunc != NULL)
466                         scm->chdirfunc(path);
467
468                 return (0);
469
470         case FTW_DNR:
471                 warn("cannot read \"%s\"", path);
472                 return (0);
473
474         case FTW_NS:
475                 warn("cannot stat \"%s\"", path);
476                 return (0);
477
478         default:
479                 break;
480         }
481
482         return (0);
483 }
484
485 /*
486  * Add ‘path’ to the pnset_t pointed to by ‘pnsetp’.
487  */
488 static int
489 pnset_add(pnset_t *pnsetp, const char *path)
490 {
491         char **newpaths;
492         unsigned int maxpaths;
493
494         if (pnsetp->npath == pnsetp->maxpaths) {
495                 maxpaths = (pnsetp->maxpaths == 0) ? 512 : pnsetp->maxpaths * 2;
496                 newpaths = realloc(pnsetp->paths, sizeof (char *) * maxpaths);
497                 if (newpaths == NULL)
498                         return (0);
499                 pnsetp->paths = newpaths;
500                 pnsetp->maxpaths = maxpaths;
501         }
502
503         pnsetp->paths[pnsetp->npath] = strdup(path);
504         if (pnsetp->paths[pnsetp->npath] == NULL)
505                 return (0);
506
507         pnsetp->npath++;
508         return (1);
509 }
510
511 /*
512  * Check ‘path’ against the pnset_t pointed to by ‘pnsetp’.
513  */
514 static int
515 pnset_check(const pnset_t *pnsetp, const char *path)
516 {
517         unsigned int i;
518
519         for (i = 0; i < pnsetp->npath; i++) {
520                 if (fnmatch(pnsetp->paths[i], path, 0) == 0)
521                         return (1);
522         }
523         return (0);
524 }
525
526 /*
527  * Empty the pnset_t pointed to by ‘pnsetp’.
528  */
529 static void
```



```
530 pnset_empty(pnset_t *pnsetp)
531 {
532         while (pnsetp->npath-- != 0)
533                 free(pnsetp->paths[pnsetp->npath]);
534
535         free(pnsetp->paths);
536         pnsetp->maxpaths = 0;
537 }
538
539 /*
540  * Free the pnset_t pointed to by ‘pnsetp’.
541  */
542 static void
543 pnset_free(pnset_t *pnsetp)
544 {
545         if (pnsetp != NULL) {
546                 pnset_empty(pnsetp);
547                 free(pnsetp);
548         }
549 }
550
551 /* PRINTFLIKE1 */
552 static void
553 warn(const char *format, ...)
554 {
555         va_list alist;
556         char *errstr = strerror(errno);
557
558         if (errstr == NULL)
559                 errstr = "<unknown error>";
560
561         (void) fprintf(stderr, "%s: ", progname);
562
563         va_start(alist, format);
564         (void) vfprintf(stderr, format, alist);
565         va_end(alist);
566
567         if (strrchr(format, ’\n’) == NULL)
568                 (void) fprintf(stderr, ": %s\n", errstr);
569 }
570
571 /* PRINTFLIKE1 */
572 static void
573 die(const char *format, ...)
574 {
575         va_list alist;
576         char *errstr = strerror(errno);
577
578         if (errstr == NULL)
579                 errstr = "<unknown error>";
580
581         (void) fprintf(stderr, "%s: fatal: ", progname);
582
583         va_start(alist, format);
584         (void) vfprintf(stderr, format, alist);
585         va_end(alist);
586
587         if (strrchr(format, ’\n’) == NULL)
588                 (void) fprintf(stderr, ": %s\n", errstr);
589
590         exit(EXIT_FAILURE);
591 }
```

```
**********************************************************
   87532 Thu Nov  1 17:19:27 2012
new/usr/src/tools/scripts/nightly.sh
3272 findunref should support git
**********************************************************
______unchanged_portion_omitted_

2117 # Echo the SCM types of $CODEMGR_WS and $BRINGOVER_WS
2118 function child_wstype {
2119         typeset scm_type junk

2121         # Probe CODEMGR_WS to determine its type
2122         if [[ -d $CODEMGR_WS ]]; then
2123                 $WHICH_SCM | read scm_type junk || exit 1
2124         fi

2126         case "$scm_type" in
2127         none|subversion|git|teamware|mercurial)
2127         none|subversion|teamware|mercurial)
2128                 ;;
2129         *)      scm_type=none
2130                 ;;
2131         esac

2133         echo $scm_type
2134 }
______unchanged_portion_omitted_
```