



Project01 - Make gpid Systemcall

첫 번째 프로젝트입니다. 크게 4 부분으로 나뉘고, 각각의 단계에 대해서 기술해놓았습니다.

- Design
 - 명세에서 요구하는 조건에 대한 구현 계획을 서술하고 있습니다.
- Implement
 - 새롭게 구현하거나 수정한 부분이 기존과 어떻게 다른지, 해당 코드의 목적이 무엇인지에 대해 구체적으로 서술하고 있습니다.
- Result
 - 컴파일 및 실행과정과, 명세의 요구부분이 정상적으로 동작하는 실행결과를 첨부하고, 동작 과정에 대해 설명하고 있습니다.
- Trouble shooting
 - 과제 수행 과정에서 겪은 문제와, 해당 문제와 해결 과정을 서술하고 있습니다.

Design

- 우리는 gpid 시스템 콜을 구현할 것입니다.
- gpid 시스템 콜은 현재 프로세스의 부모의 부모, 즉 조부모의 pid를 호출하는 시스템 콜입니다.
- 이 시스템콜을 구현하기 전에, 먼저 xv6의 시스템 콜엔 어떤게 있는지 살펴봅시다.
- xv6의 파일 내부는 다음과 같이 이루어져 있습니다.

◦ \$ ls

```
root@83d0a9e98fab:/OS/xv6-public# ls
BUGS          bio.o        date.h      forktest.d  initcode.asm  ln.asm    mp.c      proc.h      sleeplock.d  sysproc.c  vectors.pl
LICENSE       bootasm.S   def.h       forktest.o  initcode.d   ln.c     mp.d      proc.o      sleeplock.h  sysproc.d  vm.c
Makefile      bootasm.d   dot-bochsrc fs.c       initcode.o  ln.d     mp.h      project01.asm  sleeplock.o  sysproc.o  vm.d
Notes         bootasm.o   echo.asm   fs.d       initcode.out ln.o     mp.o      project01c.asm  spinlock.c  toc.ftr   vm.o
README        bootblock   echo.c    fs.h       ioapic.c   ln.sym   my_userapp.asm  project01.d  spinlock.d  toc.hdr   wc.asm
TRICKS        bootblock.asm echo.d   fs.img    ioapic.d   log.c    my_userapp.c  project01.o  spinlock.h  trap.c    wc.c
_cat          bootblock.o  echo.o    fs.o       ioapic.o   log.d    my_userapp.d  project01.sym  spinlock.o  trap.d    wc.d
_echo         bootblockother.o echo.sym gdbutil  kalloc.c log.o    my_userapp.o  project1.c  spinlp  trap.o  wc.o
_forktest     bootmain.c  elf.h     gpid.c   kalloc.o  ls.asm   my_userapp.sym rm.asm  stat.h   trapasm.S  wc.sym
_grep         bootmain.d  entry.S   gpid.d   kalloc.o  ls.c    param.h   rm.c    stressfs.asm trapasm.o  x86.h
_init         bootmain.o  entry.o   gpid.o   kbd.c    ls.d    picirq.c  rm.d    stressfs.c  traps.h   xv6.img
_kill         bootxv6.sh  entryother grep.asm kbd.o    ls.o    picirq.d  rm.o    stressfs.d  types.h   zombie.asm
_in           buf.h      entryother.S grep.c   kbd.o    ls.sym   picirq.o  rm.o    stressfs.d  user.o   zombie.c
_ls           cat.c      entryother.d grep.o  kernel main.d  pipe.d   runoff.list string.c  uart.o   zombie.d
_mkdir        cat.d      entryother.d grep.o  kernel main.o  pipe.o   runoff.spec string.d   ulib.o   zombie.e.o
_my_userapp   cat.d      entryother.o grep.sym kernel.asm main.o  pipe.o   runoff.spec string.d   ulib.d   zombie.sym
_project01    cat.o      exec.c    ide.c   kernel.ld memide.c pr_pl  runofff  string.o  ulib.c   zombie.e.o
_project1     cat.o      exec.c    ide.d   kernel.sym memlayout.h prac_syscall.c sh.asm  swtch.S  ulib.o
_rm           console.c  exec.o    ide.o   kill.asm mkdirr.asm prac_syscall.d sh.c   swtch.o  umalloc.c
_sh           console.d  fcntl.h  init.asm kill.c   mkdirr.c prac_syscall.o sh.d   syscall.c  umalloc.d
_stressfs    console.o  file.c   init.c   kill.d   mkdirr.d printf.c   sh.o   syscall.d  umalloc.o
_wc           cscope.files file.d   init.d   kill.o   mkdirr.o printf.d   sh.sym  syscall.h  user.h
_zombie      cscope.in.out file.h  init.o   kill.sym mkdirr.sym printf.o  show1   syscall.o  usys.S
asm.h        cscope.out  file.o   init.sym lapic.c  mkfs.c  printps  sign_pl  syscall.c  usys.o
bio.c        cscope.po.out forktst.asm initcode lapic.d  mkfs.c  proc.c   sleep1.p  sysfile.d  vectors.S
bio.d        cuth       forktst.c initcode.s lapic.o mmu.h  proc.d   sleeplock.c sysfile.o  vectors.o
```

- 자, 이제 xv6의 파일 내부에서 syscall.h를 열어봅시다. 이 파일안엔 xv6 내부에서 쓸 수 있는 시스템콜이 내장되어 있을 겁니다.

◦ \$ vim syscall.h

```

1 // System call numbers
2 #define SYS_fork 1
3 #define SYS_exit 2
4 #define SYS_wait 3
5 #define SYS_pipe 4
6 #define SYS_read 5
7 #define SYS_kill 6
8 #define SYS_exec 7
9 #define SYS_fstat 8
10 #define SYS_chdir 9
11 #define SYS_dup 10
12 #define SYS_getpid 11
13 #define SYS_sbrk 12
14 #define SYS_sleep 13
15 #define SYS_uptime 14
16 #define SYS_munmap 15
17 #define SYS_write 16
18 #define SYS_mknod 17
19 #define SYS_unlink 18
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_myfunction 22
24 #define SYS_gpid 23

```

- 확인해보니, 현재 프로세스 아이디를 불러올 수 있는 getpid()라는 시스템콜이 이미 구현되어있는걸 확인할 수 있습니다.
- getpid()함수를 통해 gpid를 구현할 수 있을테니, 저희는 이전에 깔았던 cscope를 활용하여 getpid()가 어디서 호출되어 있고, 어떻게 구현되어 있는지 확인해 봅시다.
- vim의 Command모드에서 :cs find c getpid 를 입력해 getpid()함수가 어디서 호출되어있느지 확인해보고, :cs find s getpid 를 입력해 getpid가 어디서 심볼로 확인되어있는지 확인해 봅시다.

```

7 int wait(void);
8 int pipe(int);
9 int write(int, const void*, int);
10 int read(int, void*, int);
11 int close(int);
12 int kill(int);
13 int exec(char*, char**);
14 int open(const char*, int);
15 int mknod(const char*, short, short);
16 int unlink(const char*);
17 int fstat(int fd, struct stat*);
18 int chmod(const char*, const char*);
19 int mkdir(const char*);
20 int chdir(const char*);
21 int dup(int);
22 int getpid(void);
23 char* sbrk(int);
24 int sleep(int);
25 int uptime(void);
26 int myfunction(char*);
27 int gpid(void);
28
29 // ulib.c
30 int stat(const char*, struct stat*);
31 char* strcpy(char*, const char*);
32 void *memmove(void*, const void*, int);
33 char* strchr(const char*, char c);
34 int strcmp(const char*, const char*);
35 void* prints(char*, ...);
36 void* strlens(char*, int max);
37 uint strlen(const char*);
38 void* memset(void*, int, uint);
39 void* malloc(uint);
40 void* free(void* );
41 int atoi(const char* );
~ ~ ~
COMMAND user.h
Cscope tag: getpid
# line filename / context / line
1   22 user.h <>exit>
        int getpid(void );
2   28 usys.S <>SYSCALLS>
        SYSCALL(getpid)
Type number and <Enter> (q or empty cancels): 

```

- 확인해보니, getpid는 user.h안에서 함수를 호출하고 있고, user.h는 다양한 곳에서 호출되고 있습니다. 자 일단, syscall.h안에 getpid가 있으니, syscall.h를 include 하고 함수를 찾아봅시다

```
○ Cscope tag: syscall.h
# line filename / context / line
 1   4 include.S <>GLOBAL>
      #include <syscall.h>
 2   8 syscall.c <>GLOBAL>
      #include <syscall.h>
 3   1 usys.S <>GLOBAL>
      #include <syscall.h>
```

- syscall.c에서 syscall.h를 호출하고 있군요. 자, 그럼 syscall.c를 살펴봅시다.

```
136 void
137 syscall(void)
138 {
139     int num;
140     struct proc *curproc = myproc();
141
142     if(num >= 0 && num < NELEM(syscalls) && syscalls[num]) {
143         curproc->tf->eax = syscalls[num]();
144     } else {
145         printf("%d %s: unknown sys call %d\n",
146                ! curproc->pid, curproc->name, num);
147         curproc->tf->eax = -1;
148     }
149 }
150 }
```

- 맨 끝의 함수에서, pid라는 변수를 사용하는 걸 볼 수 있고, 이 pid가 저희가 알고 있는 process id가 되겠군요. 이 함수를 본격적으로 분석해봅시다.

- pid는 curproc라는 구조체에서 가져오고, 이 curproc는 myproc()이라는 함수를 통해 값을 받고 있습니다. 그럼, 이 myproc()함수를 뜯어보면 되겠군요.
 - **:cs find d myproc**으로 검색하면 myproc()은 proc.c라는 함수 안에 있음을 알 수 있습니다.

```
136 void
137 syscall(void)
138 {
139     int num;
140     struct proc *curproc = myproc();
141
142     num = curproc->tf->eax;
143     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
144         curproc->tf->eax = syscalls[num]();
145     } else {
146         sprintf("%d %s: unknown sys call %d\n",
147             curproc->pid, curproc->name, num);
148         curproc->tf->eax = -1;
149     }
150 }
```

~
~
~
~
~
~
~
~
~
~

COMMAND syscall.c
Scope tag: myproc
line filename / context / line
1 61 proc_c <>pushcli>>
 pushcli();
2 62 proc_c <>mycpu>>
 c = mycpu();
3 64 proc_c <>popcli>>
 popcli();

Type number and <Enter> (o or empty cancels): ■

- 자 그럼, proc.c를 들어가봅시다. myproc()은 함수고, 여기서 proc라는 구조체를 사용하고 있군요. myproc는 포인터 구조체를 반환하고 있습니다. 그럼 이 proc라는 구조체를 분석하면 답을 얻을 수 있을 듯 합니다.
 - **:cs find g proc**으로 검색해보면, proc구조체는 proc.h라는 파일 안에 정의되어 있습니다.

```
0x0000000000000000 p_start: command,
COMMAND proc.c
Scope tag: proc
# line filename / context / line
1   6 defns.h <>proc>
    struct proc;
2  12 proc.c <>proc>
    struct proc proc[NPROC];
3  10 proc.h <>proc>
    struct proc *proc;
4  38 proc.h <>proc>
    struct error_s;
```

- 그럼, proc.h라는 파일 안을 들어가보면 되겠습니다. vim으로 proc.h를 켜봅시다.

- proc구조체를 뜯어봅시다. proc구조체 안에 pid가 존재하고, 여기서 parent process구조체를 불러올 수 있겠군요.
 - 그럼 myproc()함수로 내 프로세스를 불러온 다음, myproc()->parent->parent 가 내 조부모의 process고 여기서 ->pid를 한다면 내 조부모의 pid를 얻을 수 있겠군요!
 - 자 이제 그럼 구현만 하면 끝날 것 같습니다!

Implement

- Design이 끝났으니, 이제 구현만 하면 됩니다.
 - 파일 이름은 gpid.c로 하면 될 것 같고, 이건 Project01로 불러오는 것이니 project01.c라는 파일도 필요하겠군요.
 - gpid.c를 먼저 만들어 봅시다
 - 아까 Design대로, myproc()->parent->parent->pid만 하면 끝나겠군요.
 - 헤더파일을 잘 불러오는게 중요할 것 같습니다.

```
◦ int gpid() {
    int pid = myproc()->parent->parent->pid;
    return p;
}
```

- gpid구현은 이걸로 끝입니다. 다만 나의 Wrapper function이 필요할 것 같습니다. 실습에서 따라해본 대로 Wrapper function을 만들어 봅시다.

- //Wrapper function for my gpid

```

int sys_gpid(void) {
    int pid = myproc()->pid;
    if(pid < 0)
        return -1;
    return gpid();
}

```

```

1 #include "types.h"
2 #include "mmu.h"
3 #include "param.h"
4 #include "defs.h"
5 #include "proc.h"
6 #include "x86.h"
7
8
9 int gpid() {
10     int p = myproc()->parent->parent->pid;
11     return p;
12 }
13
14 //Wrapper for my gpid
15 int sys_gpid(void) {
16     int pid = myproc()->pid;
17     if(pid < 0)
18         return -1;
19     return gpid();
20 }

```

NORMAL gpid.c c utf-8[unix] 85% In :17/20%

- 자, Wrapper function까지 끝났습니다. 이제 설정 파일만 추가하면 되겠군요.

- 먼저 Makefile를 열어봅시다
- Makefile의 OBJS 밑에 저희의 시스템 콜을 넣읍시다.

```

OBJS = \
    bio.o \
    console.o \
    exec.o \
    file.o \
    fs.o \
    ide.o \
    ioapic.o \
    kalloc.o \
    kbd.o \
    lapic.o \
    log.o \
    main.o \
    mp.o \
    picirq.o \
    pipe.o \
    proc.o \
    sleeplock.o \
    spinlock.o \
    string.o \
    switch.o \
    sysdev.o \
    sysfile.o \
    sysproc.o \
    trapasm.o \
    trap.o \
    uart.o \
    vectors.o \
    vm.o \
    prac_syscall.o \
    gpid.o \

```

\$ make clean
\$ make | grep gpid

- 여기까지 끝냈다면 make를 해, gpid.o파일이 생겼을 겁니다.
- 그 다음, defs.h, syscall.h, syscall.c에 저희의 gpid 시스템 콜을 추가해 줍시다

```

192 //gpid.c
193 int gpid();
194
195 // number of elements in fixed-size array
196 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500

```

- 마지막으로 user.h 와 usys.S에 설정을 추가해주면 끝납니다.

```

11 SYSCALL(uptime)
12 SYSCALL(myfunction)
133 #SYSCALL(gpid)

NORMAL usys.S
27 int gpid(void);
28
29 // ublic
30 int stat(const char*, struct stat*);
31 char* strcpy(char*, const char*);
32 void* memmove(void*, const void*, int);
33 char* strchr(const char*, char c);
34 int strcmp(const char*, const char*);
35 void printf(int, const char*, ...);
36 char* gets(char*, int max);
37 uint strlen(const char*);
38 void* memset(void*, int, uint);
39 void* malloc(uint);
40 void free(void*);
41 int atoi(const char*);

NORMAL user.h

```

- 이제 project01.c 파일만 만들어 저희의 앱으로 만들면 되겠네요!

- project01.c를 다음과 같이 만들어 줍시다

```
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4
5 int main(int argc, char* argv[]) {
6     int sid = 2020028377;
7     printf(1, "My Student id is %d\n", sid);
8     printf(1, "My pid is %d\n", getpid());
9     printf(1, "My gpid is %d\n", gpid());
10    exit();
11 }
```

- 마지막으로 Makefile에 설정만 추가해봅시다!

```
170 UPROGS=\
171   _cat\
172   _echo\
173   _forktes\
174   _grep\
175   _init\
176   _kill\
177   _ln\
178   _ls\
179   _mkdir\
180   _rm\
181   _sh\
182   _stressfs\
183   _wc\
184   _zombie\
185   _my_userapp\
186   _project01
```

- 마지막으로 make만 해줍시다!

```
$ make clean  
$ make  
$ make fs.img
```

- 오류가 나지 않고 성공적으로 make가 된다면 이제 모두 끝입니다!

Result

- 이제 구현이 끝났습니다!
- 실행이 잘 되는지 확인해보겠습니다.

```
root@83d0a9e98fab:/OS/xv6-public# make clean
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*.o *.d *.asm *.sym vectors.S bootblock entryother \
initcode initcode.out kernel xv6.img fs.img kernelmemfs \
xv6memfs.img mkfs .gdbinit \
_cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _wc _zombie _my_userapp _project01
```

- 이제 xv6를 켜봅시다.

- \$./bootxv6.sh

- \$ project01

- 다음과 같이 실행된다면 성공적으로 만들어진 것 입니다!

```
$ project01
My Student id is 2020028377
My pid is 6
My gpid is 1
```

Trouble shooting.

- 개인적으로 힘들었던 점은 역시 **cscope** 사용법이었습니다.
- 자주 쓰던 방법이 아니니 좀 어려웠네요. 하지만 실습과제와 구글링을 통해 해결했습니다.
- 특히 ****:cs find s ~, :cs find c ~, :cs find d ~** 방법이 꽤나 좋았습니다.
- 이 화면만 질리게 봤습니다.

```
~ NORMAL syscall.h
E259: no matches found for cscope query g getpid of getpid
                                         cpp  utf-8[unix] 100% ln :24/24=6:1
```

- 두 번째로 힘들었던 점은 헤더 파일이었습니다
- 헤더 파일이 순서가 꼬여서 자꾸 에러가 났었습니다.
- 여기 고치는데 꽤나 많은 시간을 썼습니다.

```
In file included from gpid.c:4:
proc.h:5:20: error: field 'ts' has incomplete type
  5 |   struct taskstate ts;           // Used by x86 to find stack for interrupt
     |   ^
proc.h:6:22: error: 'NSEGS' undeclared here (not in a function)
  6 |   struct segdesc gdt[NSEGS];    // x86 global descriptor table
     |   ^
make: *** [cbuiltin: gpid.o] Error 1
root@083d0a9e98fab:/OS/xv6-public#
```

- cscope를 이용해 ts가 어디있는지 보고, 헤더 파일의 위치를 수정하여 해결하였습니다.

```
>_ Command
1 #include "types.h"
2 #include "mmu.h"
3 #include "defs.h"
4 #include "param.h"
5 #include "proc.h"
6 #include "x86.h"
7
```

- 오류를 해결한 모습입니다.

```
root@083d0a9e98fab:/OS/xv6-public# make | grep gpid
boot block is 451 bytes (max 510)
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o
ld -m elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o exec.o file.o fs.o ide.o ioapic.o kalloc.o kbd.o lapic.o log.o main.o mp.o picirq.o pipe.o pr
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0246113 s, 208 MB/s
1+0 records in
1+0 records out
512 bytes copied, 0.00149204 s, 343 kB/s
398+1 records in
398+1 records out
203984 bytes (204 kB, 199 KiB) copied, 0.00253254 s, 80.5 MB/s
```