

```
1  /
*-----*
2
3          C ve Sistem Programcılar Derneği
4
5          UNIX/Linux Sistem Programlama Kursunda Yapılan Örnekler
6
7          Eğitmen: Kaan ASLAN
8
9  -----
*-----*/
10
11 /
*-----*
12
13          Merhaba UNIX/Linux Programı
14
15          -----
*-----*/
16
17 #include <stdio.h>
18
19 int main(void)
20 {
21     printf("Hello UNIX/Linux System Programming...\n");
22 }
23
24 /
*-----*
25
26          Programın komut satırı argümanları
27
28          -----
*-----*/
29
30 #include <stdio.h>
31
32 int main(int argc, char *argv[])
33 {
34     int i;
35
36     for (i = 0; i < argc; ++i)
37         puts(argv[i]);
38
39     return 0;
40 }
41
42          -----
*-----*
43
44          getopt örneği
45
46          -----
```

```
-----*/
43
44 #include <stdio.h>
45 #include <stdlib.h>
46 #include <unistd.h>
47
48 int main(int argc, char *argv[])
49 {
50     int result;
51     int a_flag = 0, b_flag = 0, c_flag = 0;
52     char *a_arg, *c_arg;
53     int i;
54
55     opterr = 0;
56     while ((result = getopt(argc, argv, "a:bc:")) != -1) {
57         switch (result) {
58             case 'a':
59                 a_flag = 1;
60                 a_arg = optarg;
61                 break;
62             case 'b':
63                 b_flag = 1;
64                 break;
65             case 'c':
66                 c_flag = 1;
67                 c_arg = optarg;
68                 break;
69             case '?':
70                 if (optopt == 'c')
71                     fprintf(stderr, "c switch without argument!..\n");
72                 else if (optopt == 'a')
73                     fprintf(stderr, "a switch without argument!..\n");
74                 else
75                     fprintf(stderr, "invalid switch: -%c\n", optopt);
76                 exit(EXIT_FAILURE);
77
78         }
79     }
80
81     if (a_flag)
82         printf("a switch specified --> %s\n", a_arg);
83
84     if (b_flag)
85         printf("b switch specified\n");
86
87     if (c_flag)
88         printf("c switch specified --> %s\n", c_arg);
89
90     printf("arguments without switch:\n");
91
92     for (i = optind; i < argc; ++i)
93         puts(argv[i]);
94
```

```
95     return 0;
96 }
97
98 /
*-----*
-----*
99 getopt örneği
100 -----*/
101
102 #include <stdio.h>
103 #include <stdlib.h>
104 #include <unistd.h>
105
106 int main(int argc, char *argv[])
107 {
108     int m_flag = 0, a_flag = 0, s_flag = 0, d_flag = 0;
109     int result;
110     double op_result, op1, op2;
111
112     while ((result = getopt(argc, argv, "masd")) != -1) {
113         switch (result) {
114             case 'm':
115                 m_flag = 1;
116                 break;
117             case 'a':
118                 a_flag = 1;
119                 break;
120             case 's':
121                 s_flag = 1;
122                 break;
123             case 'd':
124                 d_flag = 1;
125                 break;
126             case '?':
127                 fprintf(stderr, "invalid switch: -%c\n", optopt);
128                 exit(EXIT_FAILURE);
129         }
130     }
131
132     if (m_flag + a_flag + s_flag + d_flag != 1) {
133         fprintf(stderr, "only one switch must be specified!\n");
134         exit(EXIT_FAILURE);
135     }
136
137     if (argc - optind != 2) {
138         fprintf(stderr, "two arguments must be specified!\n");
139         exit(EXIT_FAILURE);
140     }
141
142     op1 = strtod(argv[optind], NULL);
143     op2 = strtod(argv[optind + 1], NULL);
```

```
145     if (m_flag)
146         op_result = op1 * op2;
147     else if (a_flag)
148         op_result = op1 + op2;
149     else if (s_flag)
150         op_result = op1 - op2;
151     else if (d_flag)
152         op_result = op1 / op2;
153
154     printf("%f\n", op_result);
155
156     return 0;
157 }
158 /
159 */
*-----*
-----*
160 getopt_long örneği
161 -----*/
162
163 #include <stdio.h>
164 #include <stdlib.h>
165 #include <unistd.h>
166 #include <getopt.h>
167
168 int main(int argc, char *argv[])
169 {
170     int result;
171     int a_flag = 0, b_flag = 0, c_flag = 0, head_flag = 0, tail_flag = 0;
172     char *b_arg, *head_arg;
173     int i;
174     struct option options[] = {
175         {"head", required_argument, NULL, 'h'},
176         {"tail", no_argument, NULL, 't'},
177         {0, 0, 0, 0}
178     };
179
180     opterr = 0;
181     while ((result = getopt_long(argc, argv, "ab:c", options, NULL)) != -1) {
182         switch (result) {
183             case 'a':
184                 a_flag = 1;
185                 break;
186             case 'b':
187                 b_flag = 1;
188                 b_arg = optarg;
189                 break;
190             case 'c':
191                 c_flag = 1;
192                 break;
193             case 'h':
```

```
194             head_flag = 1;
195             head_arg = optarg;
196             break;
197         case 't':
198             tail_flag = 1;
199             break;
200         case '?':
201             fprintf(stderr, "invalid option!\n");
202             eexit(EXIT_FAILURE);
203     }
204 }
205
206 if (a_flag)
207     printf("a switch given\n");
208 if (b_flag)
209     printf("b switch given with argument %s\n", b_arg);
210 if (c_flag)
211     printf("c switch given\n");
212 if (head_flag)
213     printf("head switch given with argument %s\n", head_arg);
214 if (tail_flag)
215     printf("tail switch given\n");
216
217 printf("Arguments without switch:\n");
218 for (i = optind; i < argc; ++i)
219     printf("%s\n", argv[i]);
220
221 /* .... */
222
223 return 0;
224 }
225
226 /
*-----*
-----*
227 getopt_long örneği
228 mycat [-oxt] [--top[=n] --header] <dosya listesi>
229 -----*/
230
231 #include <stdio.h>
232 #include <stdlib.h>
233 #include <unistd.h>
234 #include <getopt.h>
235
236 /* Symbolic Constans */
237
238 #define DEF_LINE          10
239 #define HEX_OCTAL_LINE_LEN 16
240
241 /* Function Prorotypes */
242
243 int print_text(FILE *f, int nline);
```

```
244 int print_hex_octal(FILE *f, int nline, int hexflag);
245
246 int main(int argc, char *argv[])
247 {
248     int result, err_flag = 0;
249     int x_flag = 0, o_flag = 0, t_flag = 0, top_flag = 0, header_flag = 0;
250     char *top_arg;
251     struct option options[] = {
252         {"top", optional_argument, NULL, 1},
253         {"header", no_argument, NULL, 'h'},
254         {0, 0, 0, 0}
255     };
256     FILE *f;
257     int i, nline = -1;
258
259     opterr = 0;
260     while ((result = getopt_long(argc, argv, "xoth", options, NULL)) != -1) ↵
261     {
262         switch (result) {
263             case 'x':
264                 x_flag = 1;
265                 break;
266             case 'o':
267                 o_flag = 1;
268                 break;
269             case 't':
270                 t_flag = 1;
271                 break;
272             case 'h':
273                 header_flag = 1;
274                 break;
275             case 1:
276                 top_flag = 1;
277                 top_arg = optarg;
278                 break;
279             case '?':
280                 if (optopt != 0)
281                     fprintf(stderr, "invalid switch: -%c\n", optopt);
282                 else
283                     fprintf(stderr, "invalid switch: %s\n", argv[optind - 1]); ↵
284                     /* argv[optind - 1] dokümente edilmemiş */
285                 err_flag = 1;
286             }
287         if (err_flag)
288             exit(EXIT_FAILURE);
289
290         if (x_flag + o_flag + t_flag > 1) {
291             fprintf(stderr, "only one option must be specified from -o, -t, -x ↵
292             \n");
293             exit(EXIT_FAILURE);
294     }
```

```
294     if (x_flag + o_flag + t_flag == 0)
295         t_flag = 1;
296
297     if (top_flag)
298         nline = top_arg != NULL ? (int) strtol(top_arg, NULL, 10) : DEF_LINE;
299
300     if (optind == argc) {
301         fprintf(stderr, "at least one file must be specified!..\n");
302         exit(EXIT_FAILURE);
303     }
304
305     for (i = optind; i < argc; ++i) {
306         if ((f = fopen(argv[i], "rb")) == NULL) {
307             fprintf(stderr, "cannot open file: %s\n", argv[i]);
308             continue;
309         }
310
311         if (header_flag)
312             printf("%s\n\n", argv[i]);
313
314         if (t_flag)
315             result = print_text(f, nline);
316         else if (x_flag)
317             result = print_hex_octal(f, nline, 1);
318         else
319             result = print_hex_octal(f, nline, 0);
320
321         if (i != argc - 1)
322             putchar('\n');
323
324         if (!result)
325             fprintf(stderr, "cannot read file: %s\n", argv[i]);
326
327         fclose(f);
328     }
329
330     return 0;
331 }
332
333 int print_text(FILE *f, int nline)
334 {
335     int ch;
336     int count;
337
338     if (nline == -1)
339         while ((ch = fgetc(f)) != EOF)
340             putchar(ch);
341     else {
342         count = 0;
343         while ((ch = fgetc(f)) != EOF && count < nline) {
344             putchar(ch);
345             if (ch == '\n')
```

```
346             ++count;
347         }
348     }
349
350     return !ferror(f);
351 }
352
353 int print_hex_octal(FILE *f, int nline, int hexflag)
354 {
355     int ch, i, count;
356     const char *off_str, *ch_str;
357
358     off_str = hexflag ? "%07X " : "%012o";
359     ch_str = hexflag ? "%02X%c" : "%03o%c";
360
361     if (nline == -1)
362         for (i = 0; (ch = fgetc(f)) != EOF; ++i) {
363             if (i % HEX_OCTAL_LINE_LEN == 0)
364                 printf(off_str, i);
365             printf(ch_str, ch, i % HEX_OCTAL_LINE_LEN == HEX_OCTAL_LINE_LEN ? - 1 ? '\n' : ' ');
366         }
367
368     else {
369         count = 0;
370         for (i = 0; (ch = fgetc(f)) != EOF && count < nline; ++i) {
371             if (i % HEX_OCTAL_LINE_LEN == 0)
372                 printf(off_str, i);
373             printf(ch_str, ch, i % HEX_OCTAL_LINE_LEN == HEX_OCTAL_LINE_LEN - 1 ? '\n' : ' ');
374             if (ch == '\n')
375                 ++count;
376         }
377     }
378
379     if (i % HEX_OCTAL_LINE_LEN != 0)
380         putchar('\n');
381
382     return !ferror(f);
383 }
384
385 /
*-----*
----- POSIX fonksiyonlarında hataların tespit edilmesi
----- */
388
389 #include <stdio.h>
390 #include <stdlib.h>
391 #include <string.h>
392 #include <errno.h>
393 #include <unistd.h>
```

```
394
395 int main(void)
396 {
397     if (chdir("xxxxx") == -1) {
398         fprintf(stderr, "error: %s (%d)\n", strerror(errno), errno);
399         exit(EXIT_FAILURE);
400     }
401     printf("Ok\n");
402
403     return 0;
404 }
405
406 /
*-----*
-----*
407     POSIX fonksiyonlarında hataların tespit edilmesi
408 -----*/
409
410 #include <stdio.h>
411 #include <stdlib.h>
412 #include <unistd.h>
413
414 int main(void)
415 {
416     if (chdir("xxxxx") == -1) {
417         perror("chdir");
418         exit(EXIT_FAILURE);
419     }
420     printf("Ok\n");
421
422     return 0;
423 }
424
425 /
*-----*
-----*
426     exit_sys fonksiyonu ile hata kontrolünün kısaltılması
427 -----*/
428
429 #include <stdio.h>
430 #include <stdlib.h>
431 #include <unistd.h>
432
433 void exit_sys(const char *msg);
434
435 int main(void)
436 {
437     if (chdir("xxxxx") == -1)
438         exit_sys("chdir");
439
440     printf("Ok\n");
```

```
441
442     return 0;
443 }
444
445 void exit_sys(const char *msg)
446 {
447     perror(msg);
448
449     exit(EXIT_FAILURE);
450 }
451
452 /
*-----*
-----*
453     open fonksiyonunun kullanilmasi
454 -----*/
455
456 #include <stdio.h>
457 #include <stdlib.h>
458 #include <fcntl.h>
459 #include <sys/stat.h>
460 #include <unistd.h>
461
462 void exit_sys(const char *msg);
463
464 int main(void)
465 {
466     int fd;
467
468     if ((fd = open("test.txt", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR|S_IRGRP|
469             S_IROTH)) == -1)
470         exit_sys("open");
471
472     printf("success\n");
473
474     close(fd);
475
476     return 0;
477 }
478
479 void exit_sys(const char *msg)
480 {
481     perror(msg);
482
483     exit(EXIT_FAILURE);
484 }
485 /
*-----*
-----*
486     open fonksiyonun kullanilmasi
487 -----*
```

```
-----*/
488
489 #include <stdio.h>
490 #include <stdlib.h>
491 #include <fcntl.h>
492 #include <unistd.h>
493
494 void exit_sys(const char *msg);
495
496 int main(void)
497 {
498     int fd;
499
500     if ((fd = open("test.txt", O_RDWR)) == -1)
501         exit_sys("open");
502
503     printf("success\n");
504
505     close(fd);
506
507     return 0;
508 }
509
510 void exit_sys(const char *msg)
511 {
512     perror(msg);
513
514     exit(EXIT_FAILURE);
515 }
516
517 /
*-----*
-----*
518     read fonksiyonun kullanımı
519 -----*/
-----*/
520
521 #include <stdio.h>
522 #include <stdlib.h>
523 #include <fcntl.h>
524 #include <unistd.h>
525
526 #define SIZE    100
527
528 void exit_sys(const char *msg);
529
530 int main(void)
531 {
532     int fd;
533     char buf[SIZE + 1];
534     ssize_t result;
535
536     if ((fd = open("test.txt", O_RDWR)) == -1)
```

```
537         exit_sys("open");
538
539     if ((result = read(fd, buf, SIZE)) == -1)
540         exit_sys("read");
541
542     buf[result] = '\0';
543     puts(buf);
544
545     close(fd);
546
547     return 0;
548 }
549
550 void exit_sys(const char *msg)
551 {
552     perror(msg);
553
554     exit(EXIT_FAILURE);
555 }
556
557 /
*-----*
-----*
-----*/
```

558 read fonksiyonunun EOF'a kadar döngü içerisinde çağrılmasına bir örnek

559 -----*/

```
560
561 #include <stdio.h>
562 #include <stdlib.h>
563 #include <fcntl.h>
564 #include <unistd.h>
565
566 #define CHUNK_SIZE      10
567
568 void exit_sys(const char *msg);
569
570 int main(void)
571 {
572     int fd;
573     char buf[CHUNK_SIZE + 1];
574     ssize_t result;
575
576     if ((fd = open("test.txt", O_RDWR)) == -1)
577         exit_sys("open");
578
579     while ((result = read(fd, buf, CHUNK_SIZE)) > 0) {
580         buf[result] = '\0';
581         printf("%s", buf);
582     }
583
584     if (result == -1)
585         exit_sys("read");
586
```

```
587     putchar('\n');
588
589     close(fd);
590
591     return 0;
592 }
593
594 void exit_sys(const char *msg)
595 {
596     perror(msg);
597
598     exit(EXIT_FAILURE);
599 }
600
601 /
*-----*
-----*
602     read fonksiyonun döngü içerisinde çağrılip okunan bilgilerin hex biçimde
       ekrana yazdırılması
603 -----*/
604
605 #include <stdio.h>
606 #include <stdlib.h>
607 #include <fcntl.h>
608 #include <unistd.h>
609
610 #define CHUNK_SIZE      10
611
612 void exit_sys(const char *msg);
613
614 int main(void)
615 {
616     int fd;
617     int i, k;
618     unsigned char buf[CHUNK_SIZE];
619     ssize_t result;
620
621     if ((fd = open("test.txt", O_RDWR)) == -1)
622         exit_sys("open");
623
624     k = 0;
625     while ((result = read(fd, buf, CHUNK_SIZE)) > 0)
626         for (i = 0; i < result; ++i) {
627             printf("%02X%c", buf[i], k % 16 == 15 ? '\n' : ' ');
628             ++k;
629         }
630
631     if (result == -1)
632         exit_sys("read");
633
634     putchar('\n');
```

```
636     close(fd);
637
638     return 0;
639 }
640
641 void exit_sys(const char *msg)
642 {
643     perror(msg);
644
645     exit(EXIT_FAILURE);
646 }
647
648 /
*-
-----*
-----*/
```

649 write fonksiyonun kullanımı

650 -----*/

```
651
652 #include <stdio.h>
653 #include <stdlib.h>
654 #include <fcntl.h>
655 #include <sys/stat.h>
656 #include <unistd.h>
657
658 #define CHUNK_SIZE      10
659
660 void exit_sys(const char *msg);
661
662 int main(void)
663 {
664     int fd;
665
666     if ((fd = open("test.txt", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|
667                 S_IRGRP|S_IROTH)) == -1)
668         exit_sys("write");
669
670     if (write(fd, "istanbul", 8) == -1)
671         exit_sys("write");
672
673     printf("Ok\n");
674
675     close(fd);
676
677     return 0;
678 }
679
680 void exit_sys(const char *msg)
681 {
682     perror(msg);
683
684     exit(EXIT_FAILURE);
685 }
```

```
685
686  /
687  *-----*
688  ----- Dosya kopyalama örneği (kaynak dosyadan blok blok okuma yapıp hedef dosyaya yazılıyor)
689  -----*/
690 /* mycp.c */
691
692 #include <stdio.h>
693 #include <stdlib.h>
694 #include <fcntl.h>
695 #include <sys/stat.h>
696 #include <unistd.h>
697
698 #define CHUNK_SIZE      4096
699
700 void exit_sys(const char *msg);
701
702 int main(int argc, char *argv[])
703 {
704     int fds, fdd;
705     char buf[CHUNK_SIZE];
706     ssize_t result_r, result_w;
707
708     if (argc != 3) {
709         fprintf(stderr, "wrong number of arguments!\n");
710         fprintf(stderr, "usage: mycp <source path> <destination path>\n");
711         exit(EXIT_FAILURE);
712     }
713
714     if ((fds = open(argv[1], O_RDONLY)) == -1)
715         exit_sys("open");
716
717     if ((fdd = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|
718                                     S_IRGRP|S_IROTH)) == -1)
719         exit_sys("open");
720
721     while ((result_r = read(fds, buf, CHUNK_SIZE)) > 0)
722         if ((result_w = write(fdd, buf, result_r)) != result_r) {
723             if (result_w == -1)
724                 exit_sys("write");
725             fprintf(stderr, "cannot write file!\n");
726             exit(EXIT_FAILURE);
727         }
728
729     if (result_r == -1)
730         exit_sys("read");
731
732     close(fds);
733     close(fdd);
```

```
733     return 0;
734 }
735 }
736
737 void exit_sys(const char *msg)
738 {
739     perror(msg);
740
741     exit(EXIT_FAILURE);
742 }
743
744 /
*-----*
-----*
745 Dosya göstéricisini EOF durumuna alıp dosyaya yazma yaparak dosyayı
    büyütme örneği
746 -----*/
747
748 #include <stdio.h>
749 #include <stdlib.h>
750 #include <string.h>
751 #include <fcntl.h>
752 #include <unistd.h>
753
754 void exit_sys(const char *msg);
755
756 int main(void)
757 {
758     int fd;
759     char buf[] = "this is a test\n";
760
761     if ((fd = open("test.txt", O_WRONLY)) == -1)
762         exit_sys("open");
763
764     if (lseek(fd, 0, SEEK_END) == -1)
765         exit_sys("lseek");
766
767     if (write(fd, buf, strlen(buf)) == -1)
768         exit_sys("write");
769
770     close(fd);
771
772     printf("Ok\n");
773
774     return 0;
775 }
776
777 void exit_sys(const char *msg)
778 {
779     perror(msg);
780
781     exit(EXIT_FAILURE);
```

```
782 }
783
784 /
*-----*
-----*
785 Dosya deliklerinin oluşturulması ve delikten okuma yapma
786 -----*/
787
788 #include <stdio.h>
789 #include <stdlib.h>
790 #include <string.h>
791 #include <fcntl.h>
792 #include <unistd.h>
793
794 void exit_sys(const char *msg);
795
796 int main(void)
797 {
798     int fd;
799     char wbuf[] = "this is a test\n";
800     char rbuf[64];
801     ssize_t result;
802     off_t pos;
803     int i;
804
805     if ((fd = open("test.txt", O_RDWR)) == -1)
806         exit_sys("open");
807
808     if ((pos = lseek(fd, 1000000, SEEK_SET)) == -1)
809         exit_sys("lseek");
810
811     if (write(fd, wbuf, strlen(wbuf)) == -1)
812         exit_sys("write");
813
814     if (lseek(fd, 100000, SEEK_SET) == -1)
815         exit_sys("lseek");
816
817     if ((result = read(fd, rbuf, 64)) == -1)
818         exit_sys("read");
819
820     for (i = 0; i < 64; ++i)
821         printf("%02X%c", (unsigned char)rbuf[i], i % 16 == 15 ? '\n' : ' ');
822
823     putchar('\n');
824
825     close(fd);
826
827     return 0;
828 }
829
830 void exit_sys(const char *msg)
831 {
```

```
832     perror(msg);
833
834     exit(EXIT_FAILURE);
835 }
836
837
838 /
*-----*
-----*
839 Linux sistemlerinde POSIX standartlarında bulunmayan copy_file_range
     isimli bir sistem fonksiyonu ve bunu sarmalayan bir
840 kütüphane fonksiyonu bulunmaktadır. Bu fonksiyon iki dosya
     betimleyicisini alarak hiç user mod tampon kullanmadan belli bir
     miktar
841 byte'ı bir betimleyicinin kaynağından diğer betimleyiciin hedefine
     kopyalamaktadır. Fonksiyonun prototipi şöyledir:
842
843 ssize_t copy_file_range(int fd_in, loff_t *off_in, fd_out, loff_t
     *off_out, size_t len, unsigned int flags);
844
845 Fonksiyonun ayrıntılı çalışması man sayfalarında açıklanmaktadır.
846
847 -----*/
848
849
850 /
*-----*
-----*
851 chmod POSIX fonksiyonun kullanımı
852 -----*/
853
854 #include <stdio.h>
855 #include <stdlib.h>
856 #include <sys/stat.h>
857
858 void exit_sys(const char *msg);
859
860 int main(void)
861 {
862     if (chmod("test.txt", S_IRUSR|S_IWUSR|S_IRGRP) == -1)
863         exit_sys("chmod");
864
865     return 0;
866 }
867
868 void exit_sys(const char *msg)
869 {
870     perror(msg);
871
872     exit(EXIT_FAILURE);
873 }
```

```
874
875  /
876  *-----*
877  ----- fchmod POSIX fonksiyonun kullanımı
878  -----*/
879 #include <stdio.h>
880 #include <stdlib.h>
881 #include <fcntl.h>
882 #include <unistd.h>
883 #include <sys/stat.h>
884
885 void exit_sys(const char *msg);
886
887 int main(void)
888 {
889     int fd;
890
891     if ((fd = open("test.txt", O_RDONLY)) == -1)
892         exit_sys("open");
893
894     if (fchmod(fd, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH) == -1)
895         exit_sys("chmod");
896
897     close(fd);
898
899     return 0;
900 }
901
902 void exit_sys(const char *msg)
903 {
904     perror(msg);
905
906     exit(EXIT_FAILURE);
907 }
908
909 /
910 *-----*
911 ----- chmod komutunun octal set etme işlemini yapan örnek biçimini
912 -----*/
913 #include <stdio.h>
914 #include <stdlib.h>
915 #include <string.h>
916 #include <errno.h>
917 #include <sys/stat.h>
918
919 void exit_sys(const char *msg);
920 int is_octal(const char *str);
```

```
921
922 int main(int argc, char *argv[])
923 {
924     int i;
925     long mode;
926     mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, ↵
927         S_IROTH, S_IWOTH, S_IXOTH};
928     mode_t result_mode;
929
930     if (argc < 3) {
931         fprintf(stderr, "wrong number of arguments!..\n");
932         exit(EXIT_FAILURE);
933     }
934
935     if (!is_octal(argv[1]) || (mode = (mode_t)strtoul(argv[1], NULL, 8)) > ↵
936         0x777) {
937         fprintf(stderr, "invalid octal digits!..\n");
938         exit(EXIT_FAILURE);
939     }
940
941     result_mode = 0;
942     for (i = 8; i >= 0; --i)
943         if (mode >> i & 1)
944             result_mode |= modes[8 - i];
945
946     for (i = 2; i < argc; ++i)
947         if (chmod(argv[i], result_mode) == -1)
948             fprintf(stderr, "%s: %s\n", argv[i], strerror(errno));
949
950     return 0;
951 }
952
953 int is_octal(const char *str)
954 {
955     int i;
956
957     for (i = 0; str[i] != '\0'; ++i)
958         if (str[i] < '0' || str[i] > '7')
959             return 0;
960
961     return 1;
962 }
963
964 void exit_sys(const char *msg)
965 {
966     perror(msg);
967
968     exit(EXIT_FAILURE);
969 }
```

```
970     chown fonksiyonun kullanımı (change own restricted özelliği)
971     -----
972     -----
973 #include <stdio.h>
974 #include <stdlib.h>
975 #include <unistd.h>
976
977 void exit_sys(const char *msg);
978
979 int main(void)
980 {
981     if (chown("test.txt", 1001, -1) == -1)
982         exit_sys("chown");
983
984     printf("Ok\n");
985
986     return 0;
987 }
988
989 void exit_sys(const char *msg)
990 {
991     perror(msg);
992
993     exit(EXIT_FAILURE);
994 }
995
996 /
997     Dizinler de dosyalar gibi open fonksiyonuyla açılabilirler
998     -----
999
1000 #include <stdio.h>
1001 #include <stdlib.h>
1002 #include <fcntl.h>
1003 #include <unistd.h>
1004
1005 void exit_sys(const char *msg);
1006
1007 int main(void)
1008 {
1009     int fd;
1010
1011     if ((fd = open("test", O_RDONLY)) == -1)
1012         exit_sys("open");
1013
1014     printf("Ok\n");
1015
1016     close(fd);
1017
1018     return 0;
```

```
1019 }
1020
1021 void exit_sys(const char *msg)
1022 {
1023     perror(msg);
1024
1025     exit(EXIT_FAILURE);
1026 }
1027
1028 /
*-----*
-----*
1029     Dizinlerde 'x' hakkı yol ifadesinde içinden geçilebilirlik anlamına
1030         gelmektedir. x hakkına sahip olmayan dizinlerle
1031     yol ifadesi oluşturmak istersek sistem fonksiyonları başarısız
1032         olacaktır. Aşağıdaki örnekte test dizininin x hakkı
1033         kaldırılmıştır. Bu nedenle open fonksiyonu başarısız olmaktadır.
1034 -----*/
1035
1036 #include <stdio.h>
1037 #include <stdlib.h>
1038 #include <fcntl.h>
1039 #include <unistd.h>
1040
1041 void exit_sys(const char *msg);
1042
1043 int main(void)
1044 {
1045     int fd;
1046
1047     if ((fd = open("test/test.txt", O_RDONLY)) == -1)
1048         exit_sys("open");
1049
1050     printf("Ok\n");
1051
1052     close(fd);
1053
1054     return 0;
1055 }
1056
1057 void exit_sys(const char *msg)
1058 {
1059     perror(msg);
1060
1061     exit(EXIT_FAILURE);
1062 }
1063
1064 /-----*
-----*
1065     unlink ve remove fonksiyonları
1066 -----*
```

```
-----*/
1065 #include <stdio.h>
1066 #include <stdlib.h>
1067 #include <unistd.h>
1068
1069 void exit_sys(const char *msg);
1070
1071 int main(void)
1072 {
1073     if (unlink("test.txt") == -1)
1074         exit_sys("unlink");
1075
1076     printf("Ok\n");
1077
1078     return 0;
1079 }
1080
1081 void exit_sys(const char *msg)
1082 {
1083     perror(msg);
1084
1085     exit(EXIT_FAILURE);
1086 }
1087
1088 /
*-----*/
1089     mkdir POSIX fonksiyonu (umask'ten etkilenmektedir)
1090 -----*/
1091
1092 #include <stdio.h>
1093 #include <stdlib.h>
1094 #include <sys/stat.h>
1095
1096 int is_octal(const char *str);
1097 void exit_sys(const char *msg);
1098
1099 int main(int argc, char *argv[])
1100 {
1101     int i;
1102     int mode;
1103     mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH};
1104     mode_t result_mode;
1105
1106     if (argc != 3) {
1107         fprintf(stderr, "wrong number of arguments!..\n");
1108         exit(EXIT_FAILURE);
1109     }
1110
1111     if (!is_octal(argv[1]) || (mode = (mode_t)strtoul(argv[1], NULL, 8)) > 0x777) {
```

```
1112         fprintf(stderr, "invalid octal digits!..\\n");
1113         exit(EXIT_FAILURE);
1114     }
1115
1116     result_mode = 0;
1117     for (i = 8; i >= 0; --i)
1118         if (mode >> i & 1)
1119             result_mode |= modes[8 - i];
1120
1121     umask(0);
1122     if (mkdir(argv[2], result_mode) == -1)
1123         exit_sys("mkdir");
1124
1125     return 0;
1126 }
1127
1128 int is_octal(const char *str)
1129 {
1130     int i;
1131
1132     for (i = 0; str[i] != '\\0'; ++i)
1133         if (str[i] < '0' || str[i] > '7')
1134             return 0;
1135
1136     return 1;
1137 }
1138
1139 void exit_sys(const char *msg)
1140 {
1141     perror(msg);
1142
1143     exit(EXIT_FAILURE);
1144 }
1145
1146 /
*-----*
----- Dizinlerin rmdir POSIX fonksiyonuyla silinmesi -----*/
1147
1148
1149 #include <stdio.h>
1150 #include <stdlib.h>
1151 #include <string.h>
1152 #include <errno.h>
1153 #include <unistd.h>
1154
1155
1156 int main(int argc, char *argv[])
1157 {
1158     int i;
1159
1160     if (argc == 1) {
1161         fprintf(stderr, "wrong number of arguments!..\\n");
```

```
1162         exit(EXIT_FAILURE);
1163     }
1164
1165     for (i = 1; i < argc; ++i)
1166         if (rmdir(argv[i]) == -1)
1167             fprintf(stderr, "%s: %s\n", argv[i], strerror(errno));
1168
1169     return 0;
1170 }
1171
1172 /
*-----*
-----*
1173     rename fonksiyonu dizin girişinin isminin değiştirilmesi
1174 -----*/
1175
1176 #include <stdio.h>
1177 #include <stdlib.h>
1178
1179 void exit_sys(const char *msg);
1180
1181 int main(int argc, char *argv[])
1182 {
1183     if (argc != 3) {
1184         fprintf(stderr, "wrong number of arguments!..\\n");
1185         exit(EXIT_FAILURE);
1186     }
1187
1188     if (rename(argv[1], argv[2]) == -1)
1189         exit_sys("rename");
1190
1191     return 0;
1192 }
1193
1194 void exit_sys(const char *msg)
1195 {
1196     perror(msg);
1197
1198     exit(EXIT_FAILURE);
1199 }
1200
1201 /
*-----*
-----*
1202     stat fonksiyonun kullanımı
1203 -----*/
1204
1205 #include <stdio.h>
1206 #include <stdlib.h>
1207 #include <string.h>
1208 #include <time.h>
```

```
1209 #include <errno.h>
1210 #include <unistd.h>
1211 #include <sys/stat.h>
1212
1213 const char *lsmode(mode_t mode);
1214
1215 int main(int argc, char *argv[])
1216 {
1217     struct stat finfo;
1218     struct tm *pt;
1219     char buf[50];
1220     int i;
1221
1222     if (argc == 1) {
1223         fprintf(stderr, "wrong number of arguments!..\n");
1224         exit(EXIT_FAILURE);
1225     }
1226
1227     for (i = 1; i < argc; ++i) {
1228         if (i != 1)
1229             printf("-----\n");
1230
1231         if (stat(argv[i], &finfo) == -1) {
1232             fprintf(stderr, "%s: %s\n", argv[i], strerror(errno));
1233             continue;
1234         }
1235
1236         printf("Name: %s\n", argv[i]);
1237         printf("Inode No: %lu\n", (unsigned long)finfo.st_ino);
1238         printf("Hard Link: %lu\n", (unsigned long)finfo.st_nlink);
1239         printf("Access modes: %s\n", lsemode(finfo.st_mode));
1240         printf("User Id: %ld\n", (unsigned long)finfo.st_uid);
1241         printf("Group Id: %lu\n", (unsigned long)finfo.st_gid);
1242         printf("File Size: %ld\n", (long)finfo.st_size);
1243         printf("File System Block (Cluster) Size: %ld\n", (long)
1244             finfo.st_blksize);
1245         printf("Number Of Blocks (512 Bytes): %ld\n", (long)
1246             finfo.st_blocks);
1247
1248         pt = localtime(&finfo.st_mtime);
1249         strftime(buf, 50, "%b %d %H:%M", pt);
1250         printf("Last Update: %s\n", buf);
1251
1252         pt = localtime(&finfo.st_atime);
1253         strftime(buf, 50, "%b %d %H:%M", pt);
1254         printf("Last Access: %s\n", buf);
1255
1256         pt = localtime(&finfo.st_atime);
1257         strftime(buf, 50, "%b %d %H:%M", pt);
1258         printf("Last Status Change: %s\n", buf);
1259     }
1260
1261     return 0;
1262 }
```

```
1260 }
1261
1262 const char *lsmode(mode_t mode)
1263 {
1264     static char modetxt[11];
1265     mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, ↵
1266         S_IROTH, S_IWOTH, S_IXOTH};
1267     int i;
1268
1269     if (S_ISREG(mode))
1270         modetxt[0] = '-';
1271     else if (S_ISDIR(mode))
1272         modetxt[0] = 'd';
1273     else if (S_ISCHR(mode))
1274         modetxt[0] = 'c';
1275     else if (S_ISBLK(mode))
1276         modetxt[0] = 'b';
1277     else if (S_ISFIFO(mode))
1278         modetxt[0] = 'p';
1279     else if (S_ISLNK(mode))
1280         modetxt[0] = 'l';
1281     else if (S_ISSOCK(mode))
1282         modetxt[0] = 's';
1283     else
1284         modetxt[0] = '?';
1285
1286     for (i = 0; i < 9; ++i)
1287         modetxt[1 + i] = (mode & modes[i]) ? "rwx"[i % 3] : '-';
1288
1289     return modetxt;
1290 }
1291 /
*-----*
-----*
1292     getpwnam fonksiyonu kullanici ismini alarak bize /etc/passwd      ↵
1293     dosyasindaki kullanici bilgilerini vermektedir
-----*/
```

```
1294
1295 #include <stdio.h>
1296 #include <stdlib.h>
1297 #include <pwd.h>
1298
1299 int main(int argc, char *argv[])
1300 {
1301     struct passwd *pass;
1302     int i;
1303
1304     if (argc == 1) {
1305         fprintf(stderr, "wrong number of arguments!..\n");
1306         exit(EXIT_FAILURE);
1307 }
```

```
1308
1309     for (i = 1; i < argc; ++i) {
1310         if (i != 1)
1311             printf("-----\n");
1312         if ((pass = getpwnam(argv[i])) == NULL) {
1313             fprintf(stderr, "cannot find user: %s\n", argv[i]);
1314             continue;
1315         }
1316
1317         printf("User name: %s\n", pass->pw_name);
1318         printf("Password: %s\n", pass->pw_passwd);
1319         printf("User id: %ld\n", (long)pass->pw_uid);
1320         printf("Group id: %ld\n", (long)pass->pw_gid);
1321         printf("User info: %s\n", pass->pw_passwd);
1322         printf("Home directory: %s\n", pass->pw_dir);
1323         printf("Login Prog: %s\n", pass->pw_shell);
1324     }
1325
1326     return 0;
1327 }
1328
1329 /
*-----*
-----*
1330     getpwuid fonksiyonu kullanıcı id'sini alarak bize /etc/passwd
      dosyasındaki kullanıcı bilgilerini vermektedir
1331 -----*/
1332
1333 #include <stdio.h>
1334 #include <stdlib.h>
1335 #include <sys/types.h>
1336 #include <pwd.h>
1337
1338 void exit_sys(const char *msg);
1339
1340 int main(int argc, char *argv[])
1341 {
1342     struct passwd *pass;
1343     int i;
1344     long uid;
1345
1346     if (argc == 1) {
1347         fprintf(stderr, "wrong number of arguments!..\n");
1348         exit(EXIT_FAILURE);
1349     }
1350
1351     for (i = 1; i < argc; ++i) {
1352         if (i != 1)
1353             printf("-----\n");
1354
1355         uid = strtol(argv[i], NULL, 10);
1356         if ((pass = getpwuid((uid_t)uid)) == NULL) {
```

```
1357         fprintf(stderr, "cannot find user: %s\n", argv[i]));
1358         continue;
1359     }
1360
1361     printf("User name: %s\n", pass->pw_name);
1362     printf("Password: %s\n", pass->pw_passwd);
1363     printf("User id: %ld\n", (long)pass->pw_uid);
1364     printf("Group id: %ld\n", (long)pass->pw_gid);
1365     printf("User info: %s\n", pass->pw_passwd);
1366     printf("Home directory: %s\n", pass->pw_dir);
1367     printf("Login Prog: %s\n", pass->pw_shell);
1368 }
1369
1370 return 0;
1371 }
1372
1373 void exit_sys(const char *msg)
1374 {
1375     perror(msg);
1376
1377     exit(EXIT_FAILURE);
1378 }
1379
1380 /
*-----*
-----*
1381     getpwent, setpwent ve endpwent fonksiyonlarının kullanımı
1382 -----*/
1383
1384 #include <stdio.h>
1385 #include <stdlib.h>
1386 #include <pwd.h>
1387
1388 void exit_sys(const char *msg);
1389
1390 int main(void)
1391 {
1392     struct passwd *pass;
1393
1394     while ((pass = getpwent()) != NULL)
1395         printf("User name: %s, User id:%lu\n", pass->pw_name, (unsigned
1396                                         long)pass->pw_uid);
1397
1398     printf("-----\n");
1399
1400     setpwent();
1401
1402     while ((pass = getpwent()) != NULL)
1403         printf("User name: %s, User id:%lu\n", pass->pw_name, (unsigned
1404                                         long)pass->pw_uid);
1405
1406     endpwent();
```

```
1405     return 0;
1406 }
1407 }
1408
1409 void exit_sys(const char *msg)
1410 {
1411     perror(msg);
1412
1413     exit(EXIT_FAILURE);
1414 }
1415
1416 /
*-----*
-----*
1417     getgrnam fonksiyonu grup ismini alarak /etc/group dosyasındaki grupla    ↵
      ilgili bilgileri bize vermektedir
1418 -----*/ ↵
1419
1420 #include <stdio.h>
1421 #include <stdlib.h>
1422 #include <grp.h>
1423
1424 int main(int argc, char *argv[])
1425 {
1426     struct group *grp;
1427     int i, k;
1428
1429     if (argc == 1) {
1430         fprintf(stderr, "wrong number of arguments!..\n");
1431         exit(EXIT_FAILURE);
1432     }
1433
1434     for (i = 1; i < argc; ++i) {
1435         if (i != 1)
1436             printf("-----\n");
1437
1438         if ((grp = getgrnam(argv[i])) == NULL) {
1439             fprintf(stderr, "cannot find user: %s\n", argv[i]);
1440             continue;
1441         }
1442
1443         printf("Group name: %s\n", grp->gr_name);
1444         printf("Group password: %s\n", grp->gr_passwd);
1445         printf("Group id: %ld\n", (long)grp->gr_gid);
1446         printf("Group members: ");
1447         for (k = 0; grp->gr_mem[k] != NULL; ++k) {
1448             if (k != 0)
1449                 printf(", ");
1450             printf("%s", grp->gr_mem[k]);
1451         }
1452         printf("\n");
1453     }
}
```

```
1454     return 0;
1455 }
1456 }
1457 */
1458 *
1459 *-----*
1460 *-----*
1461 *-----*
1462 *-----*
1463 *-----*
1464 *-----*
1465 *-----*
1466 #include <stdio.h>
1467 #include <stdlib.h>
1468 #include <grp.h>
1469
1470 int main(int argc, char *argv[])
1471 {
1472     struct group *grp;
1473     int i, k;
1474     long gid;
1475
1476     if (argc == 1) {
1477         fprintf(stderr, "wrong number of arguments!..\n");
1478         exit(EXIT_FAILURE);
1479     }
1480
1481     for (i = 1; i < argc; ++i) {
1482         if (i != 1)
1483             printf("-----\n");
1484
1485         gid = strtol(argv[i], NULL, 10);
1486         if ((grp = getgrgid((gid_t)gid)) == NULL) {
1487             fprintf(stderr, "cannot find user: %s\n", argv[i]);
1488             continue;
1489         }
1490
1491         printf("Group name: %s\n", grp->gr_name);
1492         printf("Group password: %s\n", grp->gr_passwd);
1493         printf("Group id: %ld\n", (long)grp->gr_gid);
1494         printf("Group members: ");
1495         for (k = 0; grp->gr_mem[k] != NULL; ++k) {
1496             if (k != 0)
1497                 printf(", ");
1498             printf("%s", grp->gr_mem[k]);
1499         }
1500         printf("\n");
1501     }
1502 }
```

```
*-----  
-----  
1503 Dosya bilgilerini ls -l stili ile yazdırın progam (birden fazla dosya  
     için hizalama uygulanmadı)  
1504 -----*/  
1505  
1506 #include <stdio.h>  
1507 #include <stdlib.h>  
1508 #include <grp.h>  
1509  
1510 void exit_sys(const char *msg);  
1511  
1512 int main(void)  
1513 {  
1514     struct group *grp;  
1515  
1516     while ((grp = getgrent()) != NULL)  
1517         printf("Group name: %s, Group id:%lu\n", grp->gr_name, (unsigned  
                     long)grp->gr_gid);  
1518  
1519     printf("-----\n");  
1520  
1521     setgrrent();  
1522  
1523     while ((grp = getgrent()) != NULL)  
1524         printf("Group name: %s, Group id:%lu\n", grp->gr_name, (unsigned  
                     long)grp->gr_gid);  
1525  
1526  
1527     endgrrent();  
1528  
1529     return 0;  
1530 }  
1531  
1532 void exit_sys(const char *msg)  
1533 {  
1534     perror(msg);  
1535  
1536     exit(EXIT_FAILURE);  
1537 }  
1538  
1539 /  
*-----  
-----  
1540 Dosya bilgilerini ls -l stili ile yazdırın progam (birden fazla dosya  
     için hizalama uygulanmadı)  
1541 -----*/  
1542  
1543 #include <stdio.h>  
1544 #include <stdlib.h>  
1545 #include <string.h>
```

```
1546 #include <time.h>
1547 #include <errno.h>
1548 #include <unistd.h>
1549 #include <sys/stat.h>
1550 #include <pwd.h>
1551 #include <grp.h>
1552
1553 const char *get_ls(const char *name, const struct stat *finfo);
1554
1555 int main(int argc, char *argv[])
1556 {
1557     int i;
1558     struct stat finfo;
1559     const char *lsi;
1560
1561     if (argc == 1) {
1562         fprintf(stderr, "wrong number od arguments!..\\n");
1563         exit(EXIT_FAILURE);
1564     }
1565
1566     for (i = 1; i < argc; ++i) {
1567         if (stat(argv[i], &finfo) == -1) {
1568             fprintf(stderr, "file not found: %s\\n", argv[i]);
1569             continue;
1570         }
1571         if ((lsi = get_ls(argv[i], &finfo)) == NULL)
1572             fprintf(stderr, "cannot get file info: %s\\n", argv[i]);
1573         puts(lsi);
1574     }
1575
1576     return 0;
1577 }
1578
1579 const char *get_ls(const char *name, const struct stat *finfo)
1580 {
1581     static char lstext[2048];
1582     char datebuf[32];
1583     struct passwd *pass;
1584     struct group *grp;
1585     struct tm *pt;
1586
1587     mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, ↵
1588                     S_IROTH, S_IWOTH, S_IXOTH};
1589     int i;
1590
1591     if (S_ISREG(finfo->st_mode))
1592         lstext[0] = '-';
1593     else if (S_ISDIR(finfo->st_mode))
1594         lstext[0] = 'd';
1595     else if (S_ISCHR(finfo->st_mode))
1596         lstext[0] = 'c';
1597     else if (S_ISBLK(finfo->st_mode))
1598         lstext[0] = 'b';
```



```
1645         exit_sys("open");
1646
1647     if ((result = read(fd, buf, SIZE)) == -1)
1648         exit_sys("read");
1649
1650     for (i = 0; i < result; ++i) {
1651         if (i % 16 == 0)
1652             printf("%08X ", i);
1653         printf("%02X%c", buf[i], i % 16 == 15 ? '\n' : ' ');
1654     }
1655     if (i % 16 != 0)
1656         putchar('\n');
1657
1658     close(fd);
1659
1660     return 0;
1661 }
1662
1663 void exit_sys(const char *msg)
1664 {
1665     perror(msg);
1666
1667     exit(EXIT_FAILURE);
1668 }
1669
1670 /
*-----*
-----*/
```

1671 Homework-4 için ipucu

```
1672 -----*/
```

1673
1674 #include <stdio.h>
1675 #include <stdlib.h>
1676 #include <string.h>
1677 #include <pwd.h>
1678
1679 #define MAX_LINE_LEN 4096
1680
1681 void exit_sys(const char *msg);
1682
1683 struct passwd *csd_getpwnam(const char *uname)
1684 {
1685 static struct passwd pass;
1686 static char buf[MAX_LINE_LEN];
1687 FILE *f;
1688 char *tok;
1689 int success = 0;
1690
1691 if ((f = fopen("/etc/passwd", "r")) == NULL)
1692 return NULL;
1693
1694 while (fgets(buf, MAX_LINE_LEN, f) != NULL) {

```
1695         tok = strtok(buf, ":\\n");
1696         if (tok == NULL)
1697             return NULL;
1698         if (!strcmp(tok, uname)) {
1699             success = 1;
1700             break;
1701         }
1702     }
1703
1704     if (!success)
1705         return NULL;
1706
1707     pass.pw_name = tok;
1708     tok = strtok(NULL, ":");
1709     if (tok == NULL)
1710         return NULL;
1711     pass.pw_passwd = tok;
1712     tok = strtok(NULL, ":");
1713     if (tok == NULL)
1714         return NULL;
1715     pass.pw_uid = (uid_t)strtoul(tok, NULL, 10);
1716 /* etc... */
1717
1718     return &pass;
1719 }
1720
1721 int main(void)
1722 {
1723     struct passwd *pass;
1724
1725     if ((pass = csd_getpwnam("csd")) == NULL) {
1726         fprintf(stderr, "cannot find user!..\\n");
1727         exit(EXIT_FAILURE);
1728     }
1729
1730     printf("%s, %lu\\n", pass->pw_name, (unsigned long)pass->pw_uid);
1731
1732     return 0;
1733 }
1734
1735 void exit_sys(const char *msg)
1736 {
1737     perror(msg);
1738
1739     exit(EXIT_FAILURE);
1740 }
1741
1742 /
*-----*
-----*
1743 dup fonksiyonu parametresiyle belirtilen betimleyicinin gösterdiği dosya →
    nesnesi ile aynı dosya nesnesini gösteren →
1744 yeni bir dosya betimleyicisi tahsis eder ve onun değerine geri döner. →
```

```
dup en düşük boş betimleyiciyi tahsis etmektedir.
```

```
1745 -----*----- →  
1746 -----*----- →  
1747 #include <stdio.h>  
1748 #include <stdlib.h>  
1749 #include <fcntl.h>  
1750 #include <unistd.h>  
1751  
1752 void exit_sys(const char *msg);  
1753  
1754 int main(void)  
1755 {  
1756     int fd1, fd2;  
1757     char buf[5 + 1];  
1758     ssize_t result;  
1759  
1760     if ((fd1 = open("test.txt", O_RDONLY)) == -1)  
1761         exit_sys("open");  
1762  
1763     if ((fd2 = dup(fd1)) == -1)  
1764         exit_sys("dup");  
1765  
1766     if ((result = read(fd1, buf, 5)) == -1)  
1767         exit_sys("read");  
1768  
1769     buf[result] = '\0';  
1770     puts(buf);  
1771  
1772     if ((result = read(fd2, buf, 5)) == -1)  
1773         exit_sys("read");  
1774  
1775     buf[result] = '\0';  
1776     puts(buf);  
1777  
1778     printf("fd1: %d, fd2: %d\n", fd1, fd2);  
1779  
1780     close(fd1);  
1781     close(fd2);  
1782  
1783     return 0;  
1784 }  
1785  
1786 void exit_sys(const char *msg)  
1787 {  
1788     perror(msg);  
1789  
1790     exit(EXIT_FAILURE);  
1791 }  
1792  
1793 /  
-----*----- →  
-----*----- →
```

```
1794     C'de değişken sayıda argüman alan fonksiyonların yazımı
1795     -----
1796
1797 #include <stdio.h>
1798 #include <stdlib.h>
1799 #include <fcntl.h>
1800 #include <unistd.h>
1801 #include <stdarg.h>
1802
1803 void exit_sys(const char *msg);
1804
1805 int add(int a, ...);
1806
1807 int main(void)
1808 {
1809     int result;
1810
1811     result = add(5, 10, 20, 30, 40, 50);
1812     printf("%d\n", result);
1813
1814     return 0;
1815 }
1816
1817 int add(int n, ...)
1818 {
1819     va_list arg;
1820     int result;
1821     int i;
1822
1823     va_start(arg, n);
1824
1825     result = 0;
1826     for (i = 0; i < n; ++i)
1827         result += va_arg(arg, int);
1828
1829     va_end(arg);
1830
1831     return result;
1832 }
1833
1834 void exit_sys(const char *msg)
1835 {
1836     perror(msg);
1837
1838     exit(EXIT_FAILURE);
1839 }
1840
1841 /
*-----
1842     fcntl fonksiyonu betimleyici üzerinde çeşitli get set işlemleri yapan
1843     genel amaçlı bir fonslyondur. Örneğin aşağıda dosya
```

```
1843     açılırken kullanılan erişim hakları daha sonra elde edilmiştir
1844     -----
1845     -----
1846 #include <stdio.h>
1847 #include <stdlib.h>
1848 #include <fcntl.h>
1849 #include <unistd.h>
1850
1851 void exit_sys(const char *msg);
1852
1853 int main(void)
1854 {
1855     int fd;
1856     int result;
1857
1858     if ((fd = open("test.txt", O_WRONLY|O_TRUNC)) == -1)
1859         exit_sys("open");
1860
1861     if ((result = fcntl(fd, F_GETFL)) == -1)
1862         exit_sys("fcntl");
1863
1864     printf("%s\n", (result & O_ACCMODE) == O_WRONLY ? "Yes\n" : "No\n");
1865
1866     close(fd);
1867
1868     return 0;
1869 }
1870
1871 void exit_sys(const char *msg)
1872 {
1873     perror(msg);
1874
1875     exit(EXIT_FAILURE);
1876 }
1877
1878 /
1879     fcntl fonksiyonu ile aslında biz dup işlemi de yapabiliriz. Bunun için
1880     F_DUPFD command kodu kullanılmaktadır.
1881     -----
1882     -----
1883
1884 #include <stdio.h>
1885 #include <stdlib.h>
1886 #include <fcntl.h>
1887 #include <unistd.h>
1888
1889 void exit_sys(const char *msg);
1890 }
```

```
1891     int fd;
1892     int result;
1893
1894     if ((fd = open("test.txt", O_WRONLY|O_TRUNC)) == -1)
1895         exit_sys("open");
1896
1897     if ((result = fcntl(fd, F_DUPFD, 0)) == -1)
1898         exit_sys("fcntl");
1899
1900     printf("New file descriptor: %d\n", result);
1901
1902     close(fd);
1903
1904     return 0;
1905 }
1906
1907 void exit_sys(const char *msg)
1908 {
1909     perror(msg);
1910
1911     exit(EXIT_FAILURE);
1912 }
1913
1914 /
*-----*
-----*
1915     1 Numaralı (Ctrl + Alt + F1) terminal aygıt sürücüsünü write only      ↗
      modunda açıp oraya bir şeyler yazmak
1916 -----*/*/
1917
1918 #include <stdio.h>
1919 #include <stdlib.h>
1920 #include <fcntl.h>
1921 #include <unistd.h>
1922
1923 void exit_sys(const char *msg);
1924
1925 int main(void)
1926 {
1927     int fd;
1928
1929     if ((fd = open("/dev/tty1", O_WRONLY)) == -1)
1930         exit_sys("open");
1931
1932     if (write(fd, "test\n", 5) == -1)
1933         exit_sys("write");
1934
1935     close(fd);
1936
1937     return 0;
1938 }
1939
```

```
1940 void exit_sys(const char *msg)
1941 {
1942     perror(msg);
1943
1944     exit(EXIT_FAILURE);
1945 }
1946
1947 /
*-----*
-----*
1948     0, 1, ve 2 numaralı dosya betimleyicileri terminal aygı sürücüsüne      ↵
     ilişkin dosya nesnelerini göstermektedir.
1949     0 Numaralı betimleyiciden okuma yapıldığında terminal okuma yapılmış      ↵
     olur, 1 betimleyici kullanılarak yazma yapıldığında
1950     yazılan şeyler terminale yani ekrana yazılırlar. 2 numaralı betimleyici ↵
     1 numaralı betimleyici dup yağılaraç elde edilmiştir.
1951     Dolayısıyla default durumda 2 numaralı betimleyici ile yazılanlar da      ↵
     ekrana çıkacaktır.
1952 -----*/
```

```
1953
1954 #include <stdio.h>
1955 #include <stdlib.h>
1956 #include <string.h>
1957 #include <unistd.h>
1958
1959 void exit_sys(const char *msg);
1960
1961 int main(void)
1962 {
1963     int fd;
1964     char buf[] = "This is a test\n";
1965
1966     if (write(1, buf, strlen(buf)) == -1)
1967         exit_sys("write");
1968
1969     return 0;
1970 }
1971
1972 void exit_sys(const char *msg)
1973 {
1974     perror(msg);
1975
1976     exit(EXIT_FAILURE);
1977 }
1978
1979 /
*-----*
-----*
1980     0 numaralı betimleyiciden read fonksiyonuyla okum ayaparsak aslında      ↵
     terminal aygit sürücüsünden yani
1981     klavyeden okuma yapmış oluruz.
1982 -----*
```

```
-----*/
1983
1984 #include <stdio.h>
1985 #include <stdlib.h>
1986 #include <unistd.h>
1987
1988 void exit_sys(const char *msg);
1989
1990 int main(void)
1991 {
1992     ssize_t result;
1993     char buf[1024 + 1];
1994
1995     if ((result = read(0, buf, 1024)) == -1)
1996         exit_sys("write");
1997
1998     buf[result] = '\0';
1999     puts(buf);
2000
2001     return 0;
2002 }
2003
2004 void exit_sys(const char *msg)
2005 {
2006     perror(msg);
2007
2008     exit(EXIT_FAILURE);
2009 }
2010
2011 /
*-----*
-----*
2012     stderr dosyasının anlamı. Programdaki bütün hata mesajlarını stderr      ↵
2013     dosyasına yazdırılmalıdır. Böylelikle      ↵
2014     normal çıktılarla hata mesajları yönlendirme yoluyla ayırtılabilir      ↵
2015
2016     */
2017
2018 #include <stdio.h>
2019
2020 int main(void)
2021 {
2022     printf("stdout\n");
2023     fprintf(stderr, "stderr\n");
2024
2025     return 0;
2026 }
2027
*-----*
-----*
2028     Aslında stdout dosyasının yönlendirilmesi çok kolaydır. Tek yapılacak      ↵
2029     şey 1 numaralı betimleyiciyi kapatıp sonra      ↵
```

```
2028     open ile yönlendirmenin yapılacak dosyayı açmaktadır. open'ın en düşük    ↵
        betimleyiciyi verdiğini anımsayınız
2029  -----
2030  -----
2031 #include <stdio.h>
2032 #include <stdlib.h>
2033 #include <fcntl.h>
2034 #include <sys/stat.h>
2035 #include <unistd.h>
2036
2037 void exit_sys(const char *msg);
2038
2039 int main(void)
2040 {
2041     int fd;
2042     int i;
2043
2044     close(1);
2045
2046     if ((fd = open("test.txt", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|    ↵
        S_IROGRP|S_IROTH)) == -1)
2047         exit_sys("open");
2048
2049     for (i = 0; i < 10; ++i)
2050         printf("%d\n", i);
2051
2052     return 0;
2053 }
2054
2055 void exit_sys(const char *msg)
2056 {
2057     perror(msg);
2058
2059     exit(EXIT_FAILURE);
2060 }
2061
2062 /
2063 *
2064 -----
2065 Dosya yönlendirmelerinin dup2 fonksiyonuyla yapılması daha iyi bir    ↵
        tekniktir. Çünkü dup2 yönlendirme işlemini atomik bir    ↵
        biçimde yapmaktadır. Dolayısıyla multithreaded uygulamalarda ve    ↵
        reentrant uygulamalarda herhangi bir sorun oluşmaz.
2066
2067 #include <stdio.h>
2068 #include <stdlib.h>
2069 #include <fcntl.h>
2070 #include <sys/stat.h>
2071 #include <unistd.h>
2072
```

```
2073 void exit_sys(const char *msg);
2074
2075 int main(void)
2076 {
2077     int fd;
2078     int i;
2079
2080     if ((fd = open("test.txt", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|
2081             S_IRGRP|S_IROTH)) == -1)
2082         exit_sys("open");
2083
2084     if (dup2(fd, 1) == -1)
2085         exit_sys("dup2");
2086
2087     for (i = 0; i < 10; ++i)
2088         printf("%d\n", i);
2089
2090     close(fd);
2091
2092     return 0;
2093 }
2094 void exit_sys(const char *msg)
2095 {
2096     perror(msg);
2097
2098     exit(EXIT_FAILURE);
2099 }
2100
2101 /
*-----*
-----*
2102     stdin dosyasının (0 numaralı betimleyicinin) yönlendirilmesi
2103 -----*/
2104
2105 #include <stdio.h>
2106 #include <stdlib.h>
2107 #include <fcntl.h>
2108 #include <unistd.h>
2109
2110 void exit_sys(const char *msg);
2111
2112 int main(void)
2113 {
2114     int fd;
2115     int i;
2116     int val;
2117
2118     if ((fd = open("test.txt", O_RDONLY)) == -1)
2119         exit_sys("open");
2120
2121     if (dup2(fd, 0) == -1)
```

```
2122         exit_sys("dup2");
2123
2124     while (scanf("%d", &val) != EOF)
2125         printf("%d\n", val);
2126
2127     close(fd);
2128
2129     return 0;
2130 }
2131
2132 void exit_sys(const char *msg)
2133 {
2134     perror(msg);
2135
2136     exit(EXIT_FAILURE);
2137 }
2138
2139 /
*-----*
----- Dizin listesini elde etmek için opendir, readdir ve closedir POSIX
----- fonksiyonları kullanılmaktadır.
2141 -----*/
2142
2143 #include <stdio.h>
2144 #include <stdlib.h>
2145 #include <errno.h>
2146 #include <dirent.h>
2147
2148 void exit_sys(const char *msg);
2149
2150 int main(void)
2151 {
2152     DIR *dir;
2153     struct dirent *ent;
2154
2155     if ((dir = opendir("/usr/include")) == NULL)
2156         exit_sys("opendir");
2157
2158     errno = 0;
2159     while ((ent = readdir(dir)) != NULL)
2160         printf("%s, %lu\n", ent->d_name, ent->d_ino);
2161
2162     if (errno)
2163         exit_sys("readdir");
2164
2165     closedir(dir);
2166
2167     return 0;
2168 }
2169
2170 void exit_sys(const char *msg)
```

```
2171  {
2172      perror(msg);
2173
2174      exit(EXIT_FAILURE);
2175  }
2176
2177 /
*-----*
-----*
2178     Aksi söylenmediği sürece bir POSIX fonksiyonu başarı durumunda da errno
2179     değişkenini et edebilir. Ancak pratikte GNU libc
2180     kütüphanesi böylesi gereksiz bir set işlemi yapmamaktadır. Yukarıdaki
2181     kodda bu anlamda küçük bir kusur vardır. printf başarı
2182     durumunda standart bağlamında errno değişkenini set edebileceği için
2183     errno değişkenine her yinelemeye yeniden 0 atanması uygun olur
2184 -----*/
2185
2186 #include <stdio.h>
2187 #include <stdlib.h>
2188 #include <errno.h>
2189 #include <dirent.h>
2190
2191 int main(void)
2192 {
2193     DIR *dir;
2194     struct dirent *ent;
2195
2196     if ((dir = opendir("/usr/include")) == NULL)
2197         exit_sys("opendir");
2198
2199     while (errno = 0, (ent = readdir(dir)) != NULL)
2200         printf("%s\n", ent->d_name);
2201
2202     if (errno)
2203         exit_sys("readdir");
2204
2205     closedir(dir);
2206
2207     return 0;
2208 }
2209
2210 void exit_sys(const char *msg)
2211 {
2212     perror(msg);
2213
2214     exit(EXIT_FAILURE);
2215 }
2216 /
*-----*
```

```
-----  
2217     link fonksiyonun kullanımı  
2218 ----- */  
2219  
2220 #include <stdio.h>  
2221 #include <stdlib.h>  
2222 #include <errno.h>  
2223 #include <unistd.h>  
2224  
2225 void exit_sys(const char *msg);  
2226  
2227 int main(int argc, char *argv[])  
2228 {  
2229     if (argc != 3) {  
2230         fprintf(stderr, "wrong number of arguments!..\\n");  
2231         exit(EXIT_FAILURE);  
2232     }  
2233  
2234     if (link(argv[1], argv[2]) == -1)  
2235         exit_sys("link");  
2236  
2237     return 0;  
2238 }  
2239  
2240 void exit_sys(const char *msg)  
2241 {  
2242     perror(msg);  
2243  
2244     exit(EXIT_FAILURE);  
2245 }  
2246  
2247 /-----  
-----  
2248     Dizin ağacını dolaşan bir örnek program  
2249 ----- */  
2250  
2251 #include <stdio.h>  
2252 #include <stdlib.h>  
2253 #include <string.h>  
2254 #include <errno.h>  
2255 #include <sys/stat.h>  
2256 #include <unistd.h>  
2257 #include <dirent.h>  
2258  
2259 void exit_sys(const char *msg);  
2260 void walk_dir(const char *path, int level);  
2261  
2262 int main(int argc, char *argv[])  
2263 {  
2264     walk_dir("/home/csd/Study", 0);
```

```
2265     return 0;
2266 }
2268 void exit_sys(const char *msg)
2269 {
2270     perror(msg);
2272     exit(EXIT_FAILURE);
2274 }
2275
2276 void walk_dir(const char *path, int level)
2277 {
2278     DIR *dir;
2279     struct dirent *ent;
2280     struct stat finfo;
2281
2282     if (chdir(path) == -1) {
2283         fprintf(stderr, "%s:%s\n", path, strerror(errno));
2284         return;
2285     }
2286
2287     if ((dir = opendir(".")) == NULL) {
2288         perror("opendir");
2289         return;
2290     }
2291
2292     while (errno = 0, (ent = readdir(dir)) != NULL) {
2293         if (!strcmp(ent->d_name, ".") || !strcmp(ent->d_name, ".."))
2294             continue;
2295         printf("%*s%s\n", level * 4, "", ent->d_name);
2296
2297         if (lstat(ent->d_name, &finfo) == -1) {
2298             perror("stat");
2299             continue;
2300         }
2301         if (S_ISDIR(finfo.st_mode)) {
2302             walk_dir(ent->d_name, level + 1);
2303             chdir("..");
2304         }
2305     }
2306     closedir(dir);
2307
2308 }
2309
2310 /
*-----*
-----*
2311     Yukarıdaki dizin ağacını dolaşma örneğinin fonksiyon göstericisi      *
2312     kullanılarak genelleştirilmiş hali
-----*/
```

```
2314 #include <stdio.h>
2315 #include <stdlib.h>
2316 #include <string.h>
2317 #include <errno.h>
2318 #include <sys/stat.h>
2319 #include <unistd.h>
2320 #include <dirent.h>
2321
2322 void exit_sys(const char *msg);
2323 void walk_dir(const char *path, int level, void (*proc)(const struct dirent *,
2324 *int));
2325 void disp(const struct dirent *ent, int level)
2326 {
2327     printf("%*s%s\n", level * 4, "", ent->d_name);
2328 }
2329
2330 int main(int argc, char *argv[])
2331 {
2332     walk_dir("/home/csd/Study", 0, disp);
2333
2334     return 0;
2335 }
2336
2337 void exit_sys(const char *msg)
2338 {
2339     perror(msg);
2340
2341     exit(EXIT_FAILURE);
2342 }
2343
2344 void walk_dir(const char *path, int level, void (*proc)(const struct dirent *,
2345 *int))
2346 {
2347     DIR *dir;
2348     struct dirent *ent;
2349     struct stat finfo;
2350
2351     if (chdir(path) == -1) {
2352         fprintf(stderr, "%s:%s\n", path, strerror(errno));
2353         return;
2354     }
2355
2356     if ((dir = opendir(".")) == NULL) {
2357         perror("opendir");
2358         return;
2359     }
2360
2361     while (errno = 0, (ent = readdir(dir)) != NULL) {
2362         if (!strcmp(ent->d_name, ".") || !strcmp(ent->d_name, ".."))
2363             continue;
2364         proc(ent, level);
2365         if (lstat(ent->d_name, &finfo) == -1) {
```

```
2365             perror("stat");
2366             continue;
2367         }
2368         if (S_ISDIR(finfo.st_mode)) {
2369             walk_dir(ent->d_name, level + 1, proc);
2370             chdir("../");
2371         }
2372     }
2373     closedir(dir);
2374 }
2375
2376 /
*-----*
-----*
2377     Yukarıdaki dizi nağacını dolaşan programın callback fonksiyon tarafından
2378     dolaşımının durdurulabildiği versiyonu
2379 -----*/
2380 #include <stdio.h>
2381 #include <stdlib.h>
2382 #include <string.h>
2383 #include <errno.h>
2384 #include <sys/stat.h>
2385 #include <unistd.h>
2386 #include <dirent.h>
2387
2388 void exit_sys(const char *msg);
2389 int walk_dir(const char *path, int level, int (*proc)(const struct dirent *, 
2390             int));
2391 int disp(const struct dirent *ent, int level);
2392 int main(int argc, char *argv[])
2393 {
2394     int result;
2395
2396     result = walk_dir("/home/csd/Study", 0, disp);
2397     printf("result: %d\n", result);
2398
2399     return 0;
2400 }
2401
2402 void exit_sys(const char *msg)
2403 {
2404     perror(msg);
2405
2406     exit(EXIT_FAILURE);
2407 }
2408
2409 int walk_dir(const char *path, int level, int (*proc)(const struct dirent *, 
2410             int))
2411 {
2412     DIR *dir;
```

```
2412     struct dirent *ent;
2413     struct stat finfo;
2414     int result;
2415
2416     result = 1;
2417     if (chdir(path) == -1) {
2418         fprintf(stderr, "%s:%s\n", path, strerror(errno));
2419         goto EXIT2;
2420     }
2421
2422     if ((dir = opendir(".")) == NULL) {
2423         perror("opendir");
2424         goto EXIT2;
2425     }
2426
2427     while (errno = 0, (ent = readdir(dir)) != NULL) {
2428         if (!strcmp(ent->d_name, ".") || !strcmp(ent->d_name, ".."))
2429             continue;
2430         if (!proc(ent, level)) {
2431             result = 0;
2432             goto EXIT1;
2433         }
2434
2435         if (lstat(ent->d_name, &finfo) == -1) {
2436             perror("stat");
2437             continue;
2438         }
2439         if (S_ISDIR(finfo.st_mode)) {
2440             result = walk_dir(ent->d_name, level + 1, proc);
2441             chdir("..");
2442             if (!result)
2443                 goto EXIT1;
2444         }
2445     }
2446 EXIT1:
2447     closedir(dir);
2448
2449 EXIT2:
2450     return result;
2451 }
2452
2453 int disp(const struct dirent *ent, int level)
2454 {
2455     printf("%*s%s\n", level * 4, "", ent->d_name);
2456
2457     if (!strcmp(ent->d_name, "sample.c")) {
2458         printf("Buldu\n");
2459         return 1;
2460     }
2461
2462     return 1;
2463 }
2464 }
```

```
2465
2466 /
*-----*
-----*
2467 scandir fonksiyonun kullanımı
2468 -----*/
2469
2470 #include <stdio.h>
2471 #include <stdlib.h>
2472 #include <dirent.h>
2473
2474 void exit_sys(const char *msg);
2475
2476 int fselect(const struct dirent *ent)
2477 {
2478     if (ent->d_name[0] == 'a' || ent->d_name[0] == 's' || ent->d_name[0] == 'd')
2479         return 1;
2480     return 0;
2481 }
2482
2483
2484 int main(int argc, char *argv[])
2485 {
2486     struct dirent **ent;
2487     int count;
2488     int i;
2489
2490     if ((count = scandir("/usr/include", &ent, fselect, alphasort)) == -1)
2491         exit_sys("scandir");
2492
2493     for (i = 0; i < count; ++i)
2494         puts(ent[i]->d_name);
2495
2496     for (i = 0; i < count; ++i)
2497         free(ent[i]);
2498
2499     free(ent);
2500
2501     return 0;
2502 }
2503
2504 void exit_sys(const char *msg)
2505 {
2506     perror(msg);
2507
2508     exit(EXIT_FAILURE);
2509 }
2510
2511 /-----*
```

```
2512     scandir fonksiyonunda son parametrenin (karşılaştırma fonksiyonun)  
     oluşturulması  
2513 -----  
-----*/  
2514  
2515 #include <stdio.h>  
2516 #include <stdlib.h>  
2517 #include <string.h>  
2518 #include <dirent.h>  
2519  
2520 void exit_sys(const char *msg);  
2521  
2522 int fcompare(const struct dirent **de1, const struct dirent **de2)  
2523 {  
2524     return -strcmp((*de1)->d_name, (*de2)->d_name);  
2525 }  
2526  
2527 int fselect(const struct dirent *ent)  
2528 {  
2529     if (ent->d_name[0] == 'a' || ent->d_name[0] == 's' || ent->d_name[0] == 'd')  
         return 1;  
2530     return 0;  
2531 }  
2532  
2533 }  
2534  
2535 int main(int argc, char *argv[])  
2536 {  
2537     struct dirent **ent;  
2538     int count;  
2539     int i;  
2540  
2541     if ((count = scandir("/usr/include", &ent, fselect, fcompare)) == -1)  
         exit_sys("scandir");  
2543  
2544     for (i = 0; i < count; ++i)  
         puts(ent[i]->d_name);  
2546  
2547     for (i = 0; i < count; ++i)  
         free(ent[i]);  
2549  
2550     free(ent);  
2551  
2552     return 0;  
2553 }  
2554  
2555 void exit_sys(const char *msg)  
2556 {  
2557     perror(msg);  
2558  
2559     exit(EXIT_FAILURE);  
2560 }  
2561
```

```
2562  /
*-----*
-----
2563      C standartlarına göre başlangıçta stdin ve stdout dosyaları interaktif    ↵
         bir aygıta yönlendirilmiş ise tam tamponlamalı modda
2564      olamazlar. Satır tamponlamalı ya da sıfır tamponlamalı modda    ↵
         olabilirler. Linux'ta standart C kütüphanesinde stdout ve stdin
2565      dosyaları başlangıçta satır tamponlamalı moddadır. stderr ise    ↵
         başlangıçta ister interaktif aygıta yönlendirilmiş olsun isterse
2566      normal bir dosyaya yönlendirilmiş olsun hiçbir zaman tam tamponlamalı    ↵
         olamamaktadır. Aynı zamanda standartlar stdin ve stdout
2567      dosyaları işin başında interaktif olmayan bir aygıta yönlendirilmişse    ↵
         kesinlikle bunların tam tamponlamalı modda olacağını
2568      söylemektedir. Aşağıdaki örnekte ekranda bir şey göremeyebilirsiniz
2569 -----
*-----*/
2570
2571 #include <stdio.h>
2572
2573 int main(int argc, char *argv[])
2574 {
2575     printf("this is a test");
2576
2577     for (;;)
2578         ;
2579
2580     return 0;
2581 }
2582
2583 /
*-----*
-----
2584      Yukarıdaki gibi bir durumda bizim yazdıklarımızın görünmesi için fflush    ↵
         fonksiyonunu çağrırmamız ya da yazının sonuna '\n'
2585      karakterini eklememiz gereklidir.
2586 -----
*-----*/
2587
2588 #include <stdio.h>
2589
2590 int main(int argc, char *argv[])
2591 {
2592     printf("this is a test");
2593     fflush(stdout);
2594
2595     for (;;)
2596         ;
2597
2598     return 0;
2599 }
2600
2601 /
*-----*
```

```
-----  
2602     Yine C standartlarına göre stdin dosyasından okuma yapan fonksiyonlar    ↵  
2603     istege bağlı olarak stdout dosyasını flush edebilirler.  
2604  
2605 #include <stdio.h>  
2606  
2607 int main(int argc, char *argv[])  
2608 {  
2609     printf("this is a test");  
2610     fflush(stdout);  
2611  
2612     getchar();  
2613  
2614     return 0;  
2615 }  
2616  
2617 /  
*-----  
2618     Aşağıdaki örnekte döngü süresince Linux sistemlerinde ekranda bir şey    ↵  
2619     göremeyebiliriz  
2620  
2621 #include <stdio.h>  
2622  
2623 int main(int argc, char *argv[])  
2624 {  
2625     int i;  
2626  
2627     for (i = 0; i < 20; ++i) {  
2628         printf("%d ", i);  
2629         sleep(1);  
2630     }  
2631     printf("\n");  
2632  
2633     return 0;  
2634 }  
2635  
2636 /  
*-----  
2637     Yukarıdaki anomaliyi ortadan kaldırmak için fflush yapmalıyız  
2638  
2639 #include <stdio.h>  
2640  
2641  
2642 int main(int argc, char *argv[])  
2643 {  
2644     int i;
```

```
2645
2646     for (i = 0; i < 20; ++i) {
2647         printf("%d ", i);
2648         fflush(stdout);
2649         sleep(1);
2650     }
2651     printf("\n");
2652
2653     return 0;
2654 }
2655
2656 /
*-----*
-----*
2657     setbuf standart C fonksiyonu açılan dosya için tamponu değiştirmekte ya da sıfır tamponlamalı moda geçmekte kullanılır.
2658 Aşağıdaki örnekte artık tampon olarak g_buf dizisi kullanılmaktadır.
g_buf dizisinin başında "this is a test" yazısı bulunacaktır.
2660 -----*/
2661
2662 #include <stdio.h>
2663
2664 char g_buf[BUFSIZ];
2665
2666 int main(int argc, char *argv[])
2667 {
2668     int i;
2669
2670     setbuf(stdout, g_buf);
2671
2672     printf("this is a test\n");
2673
2674     for (i = 0; i < 64; ++i) {
2675         if (i % 16 == 0)
2676             printf("%08X ", (unsigned int)i);
2677         printf("%02X%c", (unsigned int)g_buf[i], i % 16 == 15 ? '\n' : ' ');
2678     }
2679     if (i % 16 != 0)
2680         putchar('\n');
2681
2682     return 0;
2683 }
2684
2685 /
*-----*
-----*
2686     setbuf fonksiyonuyla dosyanın tamponlamalı modunu sıfır tamponlamalı mod haline getirebiliriz
2687 -----*/
2688
```

```
2689 #include <stdio.h>
2690 #include <unistd.h>
2691
2692 int main(int argc, char *argv[])
2693 {
2694     setbuf(stdout, NULL);
2695
2696     printf("this is a test");
2697     sleep(10);
2698
2699     return 0;
2700 }
2701
2702 /
*-----*
-----*
2703     setvbuf standart C fonksiyonu setbuf fonksiyonunun gelişmiş bir
2704         biçimidir. Bu fonksiyonla biz hem taponun büyütüğünü,
2705         hem de modunu ve yerini değiştirebilmekteyiz.
2706 -----*/
2707
2708 #include <stdio.h>
2709 #include <stdlib.h>
2710
2711 int main(int argc, char *argv[])
2712 {
2713     FILE *f;
2714     int ch;
2715
2716     if ((f = fopen("test.txt", "r")) == NULL) {
2717         fprintf(stderr, "cannot open file!\n");
2718         exit(EXIT_FAILURE);
2719     }
2720     if (setvbuf(f, NULL, _IONBF, BUFSIZ * 2) != 0) {
2721         fprintf(stderr, "cannot change buffer mode!\n");
2722         exit(EXIT_FAILURE);
2723     }
2724     while ((ch = fgetc(f)) != EOF)
2725         putchar(ch);
2726
2727     fclose(f);
2728
2729     return 0;
2730 }
2731
2732 /
*-----*
-----*
2733 Bir dosyayı byte byte kopyalayan iki programın zaman karşılaşması
2734     time komutıyla yapılmıştır. Bu işlemi tamponlu biçimde
2735         standart C fonksiyonlarıyla yaptığımızda POSIX read fonksiyonuna göre
```

```
    yaklaşık 10 kat daha hızlı çalışmaktadır.
```

```
2735 -----*  
2736  
2737 /* cp1.c */  
2738  
2739 #include <stdio.h>  
2740 #include <stdlib.h>  
2741 #include <fcntl.h>  
2742 #include <sys/stat.h>  
2743 #include <unistd.h>  
2744  
2745 void exit_sys(const char *msg);  
2746  
2747 int main(int argc, char *argv[])  
2748 {  
2749     int fds, fdd;  
2750     char buf[1];  
2751     ssize_t result;  
2752  
2753     if (argc != 3) {  
2754         fprintf(stderr, "wrong number of arguments!..\\n");  
2755         exit(EXIT_FAILURE);  
2756     }  
2757  
2758     if ((fds = open(argv[1], O_RDONLY)) == -1)  
2759         exit_sys(argv[1]);  
2760  
2761     if ((fdd = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|  
2762             S_IRGRP|S_IROTH)) == -1)  
2763         exit_sys(argv[2]);  
2764  
2765     while ((result = read(fds, buf, 1)) > 0)  
2766         if (write(fdd, buf, 1) == -1)  
2767             exit_sys("write");  
2768  
2769     if (result == -1)  
2770         exit_sys("read");  
2771  
2772     close(fds);  
2773     close(fdd);  
2774  
2775     return 0;  
2776 }  
2777 void exit_sys(const char *msg)  
2778 {  
2779     perror(msg);  
2780  
2781     exit(EXIT_FAILURE);  
2782 }  
2783  
2784 /* cp2.c */
```

```
2785
2786 #include <stdio.h>
2787 #include <stdlib.h>
2788
2789 int main(int argc, char *argv[])
2790 {
2791     FILE *fs, *fd;
2792     int ch;
2793
2794     if (argc != 3) {
2795         fprintf(stderr, "wrong number of arguments!..\\n");
2796         exit(EXIT_FAILURE);
2797     }
2798
2799     if ((fs = fopen(argv[1], "r")) == NULL) {
2800         fprintf(stderr, "cannot open file: %s\\n", argv[1]);
2801         exit(EXIT_FAILURE);
2802     }
2803
2804     if ((fd = fopen(argv[2], "w")) == NULL) {
2805         fprintf(stderr, "cannot open file: %s\\n", argv[2]);
2806         exit(EXIT_FAILURE);
2807     }
2808
2809     while ((ch = fgetc(fs)) != EOF)
2810         if (fputc(ch, fd) == -1) {
2811             fprintf(stderr, "cannot write file!..\\n");
2812             exit(EXIT_FAILURE);
2813         }
2814
2815     if (ferror(fs)) {
2816         fprintf(stderr, "cannot read file!..\\n");
2817         exit(EXIT_FAILURE);
2818     }
2819
2820     fclose(fs);
2821     fclose(fd);
2822
2823     return 0;
2824 }
2825
2826 /
*-----*
-----*
2827 Yukarıdaki örnekte cp2.c programında standart C dosyalarının tamponlama modunu Unbuffered hale getirirsek performans cp1.c ile yaklaşık aynı olmaktadır.
2828 -----*/
2829 -----*/
2830
2831 /* cp2.c */
2832
2833 #include <stdio.h>
```

```
2834 #include <stdlib.h>
2835
2836 int main(int argc, char *argv[])
2837 {
2838     FILE *fs, *fd;
2839     int ch;
2840
2841     if (argc != 3) {
2842         fprintf(stderr, "wrong number of arguments!..\\n");
2843         exit(EXIT_FAILURE);
2844     }
2845
2846     if ((fs = fopen(argv[1], "r")) == NULL) {
2847         fprintf(stderr, "cannot open file: %s\\n", argv[1]);
2848         exit(EXIT_FAILURE);
2849     }
2850
2851     if ((fd = fopen(argv[2], "w")) == NULL) {
2852         fprintf(stderr, "cannot open file: %s\\n", argv[2]);
2853         exit(EXIT_FAILURE);
2854     }
2855
2856     setbuf(fs, NULL);
2857     setbuf(fd, NULL);
2858
2859     while ((ch = fgetc(fs)) != EOF)
2860         if (fputc(ch, fd) == -1) {
2861             fprintf(stderr, "cannot write file!..\\n");
2862             exit(EXIT_FAILURE);
2863         }
2864
2865     if (ferror(fs)) {
2866         fprintf(stderr, "cannot read file!..\\n");
2867         exit(EXIT_FAILURE);
2868     }
2869
2870     fclose(fs);
2871     fclose(fd);
2872
2873     return 0;
2874 }
2875
2876 /
*-----
```

```
2881     dosyasından) okuyarak tampona yerleştirmektedir. Aşağıdaki programda →
2882         getchar ikinci çağrıda gerçek anlamda klavyeden →
2883         okuma istemeyecektir.
2883 -----
2884
2885 #include <stdio.h>
2886
2887 int main(void)
2888 {
2889     int ch;
2890
2891     ch = getchar();
2892     printf("%c\n", ch);
2893
2894     ch = getchar();
2895     printf("%c\n", ch);
2896
2897     return 0;
2898 }
2899
2900 /
2901 -----
2901     Her getchar fonksiyonunda yeniden klavyeden giriş istenmesini istiyorsak →
2902         bu durumda tampondaki byte'ları '\n'yi görene kadar →
2903         okumalıyız. Bu işlemin daha pratik bir yöntemi yoktur.
2903 -----
2904
2905 #include <stdio.h>
2906
2907 int main(void)
2908 {
2909     int ch;
2910
2911     ch = getchar();
2912     printf("%c\n", ch);
2913
2914     while ((ch = getchar()) != '\n' && ch != EOF)
2915         ;
2916
2917     ch = getchar();
2918     printf("%c\n", ch);
2919
2920     return 0;
2921 }
2922
2923 /
2923 -----
2924     stdin tamponunu boşaltan döngüyü bir fonksiyon olarak da yazabiliriz
2925 -----
```

```
2926
2927 #include <stdio.h>
2928
2929 void clear_stdin(void)
2930 {
2931     int ch;
2932
2933     while ((ch = getchar()) != '\n' && ch != EOF)
2934         ;
2935 }
2936
2937 int main(void)
2938 {
2939     int ch;
2940
2941     ch = getchar();
2942     printf("%c\n", ch);
2943
2944     clear_stdin();
2945
2946     ch = getchar();
2947     printf("%c\n", ch);
2948
2949     return 0;
2950 }
2951
2952 /
*-----*/
```

2953 Yukarıdaki fonksiyon bir makro biçiminde de yazılabiliyor (do-while'a dikkat)

```
2954 -----*/
```

2955
2956 #include <stdio.h>
2957
2958 #define clear_stdin() \
2959 do \
2960 { \
2961 int ch; \
2962 \
2963 while ((ch = getchar()) != '\n' && ch != EOF) \
2964 ; \
2965 } while (0)
2966
2967 int main(void)
2968 {
2969 int ch;
2970
2971 ch = getchar();
2972 printf("%c\n", ch);
2973

```
2974     clear_stdin();  
2975  
2976     ch = getchar();  
2977     printf("%c\n", ch);  
2978  
2979     return 0;  
2980 }  
2981  
2982 /  
*-----  
2983 gets fonksiyonu stdin dosyasından '\n' görene kadar ('\n' dahil olmak  
üzerde) ya da EOF'agelinene kadar okuma yapar. Okunanları  
parametresiyle belirtilen adresten itibaren yerleştirir. Sonuna da '\0' ekle.  
gets hiçbir okuma yapamadan EOF ile karşılaşırsa  
NULL adresi geri dönmektedir. Ancak en az 1 karakter okuma yapmış ise  
parametresiyle belirtilen adresin aynısına geri döner.  
gets fonksiyonu aşağıdaki gibi yazılmıştır  
2987 -----*/  
2988  
2989 #include <stdio.h>  
2990  
2991 char *mygets(char *buf)  
2992 {  
2993     int ch;  
2994     size_t i;  
2995  
2996     for (i = 0; (ch = getc(stdin)) != '\n' && ch != EOF; ++i)  
2997         buf[i] = ch;  
2998  
2999     if (i == 0 && ch == EOF)  
3000         return NULL;  
3001  
3002     buf[i] = '\0';  
3003  
3004     return buf;  
3005 }  
3006  
3007 int main(void)  
3008 {  
3009     char buf[1024];  
3010     char ch;  
3011  
3012     printf("Bir yazı giriniz:");  
3013     mygets(buf);  
3014  
3015     printf("Bir karakter giriniz:");  
3016     ch = getchar();  
3017  
3018     puts(buf);  
3019     printf("%c\n", ch);  
3020 }
```

```
3021     return 0;
3022 }
3023
3024 /
*-----*
-----+
3025     gets_s fonksiyonunun gerçekleştirmesi
3026 -----*/
3027
3028 #include <stdio.h>
3029
3030 char *mygets_s(char *buf, size_t size)
3031 {
3032     size_t i;
3033     int ch = 0;
3034
3035     for (i = 0; i < size - 1; ++i) {
3036         if ((ch = getc(stdin)) == '\n' || ch == EOF)
3037             break;
3038         buf[i] = ch;
3039     }
3040
3041     if (i == 0 && ch == EOF)
3042         return NULL;
3043
3044     buf[i] = '\0';
3045
3046     return buf;
3047 }
3048
3049 int main(void)
3050 {
3051     char buf[10];
3052     char ch;
3053
3054     printf("Bir yazı giriniz:");
3055     mygets_s(buf, 10);
3056     puts(buf);
3057
3058     return 0;
3059 }
3060
3061 /
*-----*
-----+
3062     scanf fonksiyonu başarılı biçimde yerleştirdiği parça sayısına geri
3063     döner. Baştaki leading space'leri atmaktadır. Ancak
3064     trailing space'lere dokunulmaz.
3065 -----*/
3066 #include <stdio.h>
```

```
3067
3068 int main(void)
3069 {
3070     int a = -1, b = -1;
3071     int result;
3072
3073     result = scanf("%d%d", &a, &b);
3074     printf("result = %d, a = %d, b = %d\n", result, a, b);
3075
3076     return 0;
3077 }
3078
3079 /
*-----*
-----*
3080     scanf fonksiyonundan sonra gets ya da gets_s kullanırken dikkat ediniz
3081 -----*/
3082
3083 #include <stdio.h>
3084
3085 int main(void)
3086 {
3087     int no;
3088     char name[1024];
3089
3090     printf("Numaranızı giriniz:");
3091     scanf("%d", &no);
3092
3093     printf("Adınızı giriniz:");
3094     gets(name);
3095
3096     printf("No: %d, Ad: %s\n", no, name);
3097
3098     return 0;
3099 }
3100
3101 /
*-----*
-----*
3102     Yukarıdaki programın düzeltilmiş hali şöyledir
3103 -----*/
3104
3105 #include <stdio.h>
3106
3107 void clear_stdin(void)
3108 {
3109     int ch;
3110
3111     while ((ch = getchar()) != '\n' && ch != EOF)
3112         ;
3113 }
```

```
3114
3115 int main(void)
3116 {
3117     int no;
3118     char name[1024];
3119
3120     printf("Numaranızı giriniz:");
3121     scanf("%d", &no);
3122
3123     clear_stdin();
3124
3125     printf("Adınızı giriniz:");
3126     gets(name);
3127
3128     printf("No: %d, Ad: %s\n", no, name);
3129
3130     return 0;
3131 }
3132
3133 /
*-----*
-----*
3134     scanf stdin dosyasından girdileri bir döngü içerisinde karakter karakter
3135         alıp işleme sokmaktadır. Eğer formak karakterine uygun
3136         bir girişle karşılaşmazsa o karakteri tampona geri bırakır ve işlemini
3137         sonlandırır
3138 -----*/
3139
3140 #include <stdio.h>
3141
3142 int main(void)
3143 {
3144     int a = -1, b = -1;
3145     int result;
3146     char buf[1024];
3147
3148     result = scanf("%d%d", &a, &b);      /* 10 ali */
3149     printf("result = %d, a = %d, b = %d\n", result, a, b); /* result = 1, a =
3150         = 10, b = -1 */
3151     gets(buf);
3152     printf("\'%s\'\n", buf);      /* "ali" */
3153
3154     return 0;
3155 }
3156
*-----*
-----*
3157     scanf fonksiyonuna uygun karakter girilmemesi sonucunda oluşacak hatalı
3158         durumun ele alınması
3159 -----*/

```

```
3157
3158 #include <stdio.h>
3159
3160 int get_menu_option(void);
3161 void clear_stdin(void);
3162
3163 int main(void)
3164 {
3165     int option;
3166
3167     for (;;) {
3168         option = get_menu_option();
3169         switch (option) {
3170             case 0:
3171                 printf("Geçersiz giriş!\n\n");
3172                 clear_stdin();
3173                 break;
3174             case 1:
3175                 printf("Kayıt ekleme işlemi\n\n");
3176                 break;
3177             case 2:
3178                 printf("Kayıt bul işlemi\n\n");
3179                 break;
3180             case 3:
3181                 printf("Kayıt sil işlemi\n\n");
3182                 break;
3183             case 4:
3184                 goto EXIT;
3185         }
3186     }
3187
3188 EXIT:
3189     return 0;
3190 }
3191
3192 int get_menu_option(void)
3193 {
3194     int option = 0;
3195
3196     printf("1) Kayıt Ekle\n");
3197     printf("2) Kayıt Bul\n");
3198     printf("3) Kayıt Sil\n");
3199     printf("4) Çıkış\n");
3200
3201     printf("Seçiminiz:");
3202     scanf("%d", &option);
3203
3204     return option;
3205 }
3206
3207 void clear_stdin(void)
3208 {
3209     int ch;
```

```
3210
3211     while ((ch = getchar()) != '\n' && ch != EOF)
3212         ;
3213 }
3214
3215 /
*-----*
-----*
3216     scanf fonksiyonunda format karakterlerinin yanın bir white space "white "
3217     space görmeyene kadar stdin'den okuma yap" anlamına gelmektedir.
3218     Bu nedenle sonda yanlışlıkla bırakılan white space'ler ciddi sorun
3219     oluşturur.
3220
3221 #include <stdio.h>
3222
3223 int main(void)
3224 {
3225     int a = -1, b = -1;
3226     int result;
3227
3228     result = scanf("%d%d\n", &a, &b);
3229     printf("result = %d, a = %d, b = %d\n", result, a, b);
3230
3231     return 0;
3232 }
3233 /
*-----*
-----*
3234     scanf fonksiyonunda format karakterlerinin arasında white space olmayan
3235     karakterler girişte de aynı pozisyonda bulundurulmak zorundadır.
3236
3237 #include <stdio.h>
3238
3239 int main(void)
3240 {
3241     int day, month, year;
3242
3243     scanf("%d/%d/%d", &day, &month, &year);
3244     printf("%02d/%02d/%02d\n", day, month, year);
3245
3246     return 0;
3247 }
3248
3249 /
*-----*
-----*
3250     scanf fonksiyonuyla yönlendirme drumunda EOF ya da geçersiz bir giriş
3251     görene kadar okuma yapma
```

```
3251 -----*/  
3252  
3253 #include <stdio.h>  
3254  
3255 int main(void)  
3256 {  
3257     int val;  
3258     int result;  
3259  
3260     while ((result = scanf("%d", &val)) != EOF && result != 0)  
3261         printf("%d\n", val);  
3262  
3263     return 0;  
3264 }  
3265  
3266 /-----*/  
3267     fileno fonksiyonu C tarzi stream'i parametre olarak alır. FILE yapısı  
3268     içerisindeki dosya betimleyicisi ile geri döner.  
3269 -----*/  
3270  
3271 #include <stdio.h>  
3272 #include <stdlib.h>  
3273 #include <unistd.h>  
3274  
3275 void exit_sys(const char *msg);  
3276  
3277 int main(void)  
3278 {  
3279     FILE *f;  
3280     int fd;  
3281     char buf[100 + 1];  
3282     int result;  
3283  
3284     if ((f = fopen("test.txt", "r")) == NULL)  
3285         exit_sys("fopen");  
3286  
3287     if ((fd = fileno(f)) == -1)  
3288         exit_sys("fileno");  
3289  
3290     if ((result = read(fd, buf, 100)) == -1)  
3291         exit_sys("read");  
3292     buf[result] = '\0';  
3293     puts(buf);  
3294  
3295     fclose(f);  
3296  
3297     return 0;  
3298 }
```

```
3299 void exit_sys(const char *msg)
3300 {
3301     perror(msg);
3302     exit(EXIT_FAILURE);
3303 }
3304 /
3305 *
3306 -----
3307     fdopen fonksiyonu işlevsel olarak fileno fonksiyonunun tersini
3308     yapmaktadır. Yani bizden bir betimleyici alıp bize FILE *
3309     verir. Dolayısıyla biz artık standart dosya fonksiyonlarını
3310     kullanabiliriz.
3311 -----
3312     */
3313
3314 #include <stdio.h>
3315 #include <stdlib.h>
3316 #include <fcntl.h>
3317 #include <unistd.h>
3318
3319 int main(void)
3320 {
3321     int fd;
3322     FILE *f;
3323     int ch;
3324
3325     if ((fd = open("test.txt", O_RDONLY)) == -1)
3326         exit_sys("open");
3327
3328     if ((f = fdopen(fd, "r")) == NULL)
3329         exit_sys("fdopen");
3330
3331     while ((ch = fgetc(f)) != EOF)
3332         putchar(ch);
3333
3334     fclose(f);
3335
3336     return 0;
3337 }
3338
3339 void exit_sys(const char *msg)
3340 {
3341     perror(msg);
3342     exit(EXIT_FAILURE);
3343 }
3344
3345
3346 /
```

```
*-----  
-----  
3347     Programın iki noktası arasında geçen zaman o anda sistemin yükgne bağlı →  
          olarak ciddi farklılıklar gösterebilir  
3348 -----*/  
3349  
3350 #include <stdio.h>  
3351 #include <time.h>  
3352  
3353 int main(void)  
3354 {  
3355     long i;  
3356     time_t start, end;  
3357  
3358     start = time(NULL);  
3359  
3360     for (i = 0; i < 10000000000; ++i)  
3361         ;  
3362  
3363     end = time(NULL);  
3364  
3365     printf("%lu\n", (unsigned long)(end - start));  
3366  
3367     return 0;  
3368 }  
3369  
3370 /  
-----  
-----  
3371 Thread'ler (ya da thread yoksa prosesler) IO yoğun ve CPU yoğun olmak →  
          üzere kabaca iki ayrılabilir.  
3372 IO yoğun thread'ler onalara verilen quanta süresini çok az kullanıp →  
          hemen bloke olurlar. CPU yoğun thread'ler kendilerine  
3373 verilen quanta süresini büyük ölçüde harcarlar. Genel olarak thread'ler →  
          IO yoğun olma eğilimindedir. Ancak bir döngü  
3374 içerisinde hiç bloke olmayan thread'ler CPU yoğundur. IO yoğun →  
          thread'ler yüzlerce olsa bile sistemde ciddi bir yavaşlığa  
3375 yol açmazlar. Örneğin aşağıdaki program IO yoğun bir thread'e örnektir.  
3376 -----*/  
3377  
3378 #include <stdio.h>  
3379  
3380 int main(void)  
3381 {  
3382     int val;  
3383  
3384     for (;;) {  
3385         scanf("%d", &val);  
3386         if (val == 0)  
3387             break;  
3388         printf("%d", val * val);
```

```
3389     }
3390
3391     return 0;
3392 }
3393
3394 /
*-
-----*
3395     Kütüphanelerdeki sleep gibi fonksiyonlar aslında CPU'yu mesgul ederek
3396     beklemeye yapmazlar. Blokeye yol açarak beklemeye yaparlar.
3397     Bu nedenle aşağıdaki gibi programlar IO yoğundur ve zaman harcama
3398     bakımından sisteme hiç yük bindirmezler
3399
3400 #include <stdio.h>
3401 #include <unistd.h>
3402
3403 int main(void)
3404 {
3405     int i;
3406
3407     for (i = 0; i < 10; ++i) {
3408         sleep(1);
3409         printf("%d ", i);
3410         fflush(stdout);
3411     }
3412     printf("\n");
3413
3414     return 0;
3415 }
3416 /
*-
-----*
3417     fork fonksiyonu ile yeni bir prosesin yaratılması
3418
3419
3420 #include <stdio.h>
3421 #include <stdlib.h>
3422 #include <unistd.h>
3423
3424 void exit_sys(const char *msg);
3425
3426 int main(void)
3427 {
3428     pid_t pid;
3429
3430     printf("before fork\n");
3431
3432     if ((pid = fork()) == -1)
3433         exit_sys("fork");
```

```
3434
3435     if (pid != 0) {
3436         /* parent */
3437         printf("parent process...\n");
3438     }
3439     else {
3440         /* child */
3441         printf("child process...\n");
3442     }
3443
3444     printf("common...\n");
3445
3446     return 0;
3447 }
3448
3449 void exit_sys(const char *msg)
3450 {
3451     perror(msg);
3452
3453     exit(EXIT_FAILURE);
3454 }
3455
3456 /
*-----*
-----*
```

3457 fork işlemi
3458 -----*/

```
3459
3460 #include <stdio.h>
3461 #include <stdlib.h>
3462 #include <unistd.h>
3463
3464 void exit_sys(const char *msg);
3465
3466 int main(void)
3467 {
3468     pid_t pid;
3469
3470     printf("before fork\n");
3471
3472     if ((pid = fork()) == -1)
3473         exit_sys("fork");
3474
3475     if (pid != 0) {
3476         printf("Parent's parent process id: %ld\n", (long)getppid());
3477         printf("Parent process process id: %ld\n", (long)getpid());
3478         printf("Parent process fork return value: %ld\n", (long)pid);
3479     }
3480     else {
3481         printf("Child process id: %ld\n", (long) getpid());
3482         printf("Child's parent process id: %ld\n", (long)getppid());
3483     }
```

```
3484     return 0;
3485 }
3487
3488 void exit_sys(const char *msg)
3489 {
3490     perror(msg);
3491     exit(EXIT_FAILURE);
3492 }
3494
3495 /
*-----*
-----*
3496     Aşağıdaki örnekte ekrana toplamda kaç tane "ends..." yazısı çıkar?
3497 -----*
-----*/
3498
3499 #include <stdio.h>
3500 #include <stdlib.h>
3501 #include <unistd.h>
3502
3503 void exit_sys(const char *msg);
3504
3505 int main(void)
3506 {
3507     if (fork() == -1)
3508         exit_sys("fork");
3509
3510     if (fork() == -1)
3511         exit_sys("fork");
3512
3513     printf("ends..\n");
3514
3515     return 0;
3516 }
3517
3518 void exit_sys(const char *msg)
3519 {
3520     perror(msg);
3521     exit(EXIT_FAILURE);
3522 }
3524
3525 /
*-----*
-----*
3526     Aşağıdaki örnekte ekrana toplamda kaç tane "ends..." yazısı çıkar?
3527 -----*
-----*/
3528
3529 #include <stdio.h>
3530 #include <stdlib.h>
```

```
3531 #include <unistd.h>
3532
3533 void exit_sys(const char *msg);
3534
3535 int main(void)
3536 {
3537     int i;
3538
3539     for (i = 0; i < 3; ++i)
3540         if (fork() == -1)
3541             exit_sys("fork");
3542
3543     printf("ends..\n");
3544
3545     return 0;
3546 }
3547
3548 void exit_sys(const char *msg)
3549 {
3550     perror(msg);
3551
3552     exit(EXIT_FAILURE);
3553 }
3554
3555 /
*-----*
-----*
3556 fork işleminden sonra artık üst proses ile alt prosesin sanal bellek
3557 alanları tamamen ayrılmıştır. Yani bunlar birbirlerinden
3558 izole edilmiş aynı kod ve data'ya sahip iki farklı proses durumundadır.
3559 Dolayısıyla fork işleminden sonra üst proseseki
3560 değişiklikler alt prosesi alt proseseki değişiklikler üst prosesi
3561 etkilemez. Yani fork işlemden sonra üst ve alt prosesin
3562 herhangi iki farklı prosesten çalışma anlamında bir farkı yoktur.
3563 -----*/
3564
3565 #include <stdio.h>
3566 #include <stdlib.h>
3567 #include <unistd.h>
3568
3569 void exit_sys(const char *msg);
3570
3571 int g_x;
3572
3573 int main(void)
3574 {
3575     pid_t pid;
3576
3577     g_x = 100;
3578     if ((pid = fork()) == -1)
3579         exit_sys("fork");
3580 }
```

```
3578     if (pid != 0) {
3579         g_x = 200;
3580     }
3581     else {
3582         sleep(1);
3583         printf("%d\n", g_x); /* 100 */
3584     }
3585 }
3586
3587 return 0;
3588 }
3589
3590 void exit_sys(const char *msg)
3591 {
3592     perror(msg);
3593
3594     exit(EXIT_FAILURE);
3595 }
3596
3597 /
*-----*
-----*
3598 fork işlemi sırasında üst prosesin dosya betimleyici tablosundaki file ↵
    nesne adresleri alt prosesin dosya betimleyici
3599 tablosuna kopyalanır. Yani fork işleminden sonra üst ve alt proseslerin ↵
    dosya betimleyici tabloları aynı dosya nesnesini (struct file)
3600 gösterir hale gelmektedir. Bu durumda bu proseslerden biri örneğin dosya ↵
    göstericisini konumlandırırsa diğeride bunu konumlandırmış
3601 olarak görecektir.
3602 -----*
-----*/
3603
3604 #include <stdio.h>
3605 #include <stdlib.h>
3606 #include <fcntl.h>
3607 #include <unistd.h>
3608
3609 void exit_sys(const char *msg);
3610
3611 int main(void)
3612 {
3613     pid_t pid;
3614     int fd;
3615
3616     if ((fd = open("test.txt", O_RDONLY)) == -1)
3617         exit_sys("open");
3618
3619     if ((pid = fork()) == -1)
3620         exit_sys("fork");
3621
3622     if (pid != 0) {
3623         lseek(fd, 200, SEEK_SET);
3624     }
```

```
3625     else {
3626         char buf[100 + 1];
3627         ssize_t result;
3628
3629         sleep(1);
3630
3631         if ((result = read(fd, buf, 100)) == -1)
3632             exit_sys("read");
3633         buf[result] = '\0';
3634         puts(buf);
3635     }
3636
3637     return 0;
3638 }
3639
3640 void exit_sys(const char *msg)
3641 {
3642     perror(msg);
3643
3644     exit(EXIT_FAILURE);
3645 }
3646
3647 /
*-----*
-----*
```

3648 execl Fonksiyonun kullanımı
3649 -----*/
3650
3651 #include <stdio.h>
3652 #include <stdlib.h>
3653 #include <unistd.h>
3654
3655 void exit_sys(const char *msg);
3656
3657 int main(void)
3658 {
3659 printf("program begins...\n");
3660
3661 if (execl("/bin/ls", "/bin/ls", "-l", "-i", (char *)NULL) == -1)
3662 exit_sys("execl");
3663
3664 printf("Unreachable code!..\n");
3665
3666 return 0;
3667 }
3668
3669 void exit_sys(const char *msg)
3670 {
3671 perror(msg);
3672
3673 exit(EXIT_FAILURE);
3674 }

```
3675
3676  /
3677  *-----*
3678  ----- execl fonksiyonun kullanımı
3679  -----*/
3680 /* sample.c */
3681
3682 #include <stdio.h>
3683 #include <stdlib.h>
3684 #include <unistd.h>
3685
3686 void exit_sys(const char *msg);
3687
3688 int main(void)
3689 {
3690     printf("sample begins...\n");
3691
3692     if (execl("mample", "mample", "ali", "veli", "selami", (char *)NULL) == -1)
3693         exit_sys("execl");
3694
3695     printf("Unreachable code!..\n");
3696
3697     return 0;
3698 }
3699
3700 void exit_sys(const char *msg)
3701 {
3702     perror(msg);
3703
3704     exit(EXIT_FAILURE);
3705 }
3706
3707 /* mample.c */
3708
3709 #include <stdio.h>
3710
3711 int main(int argc, char *argv[])
3712 {
3713     int i;
3714
3715     printf("mample is running...\n");
3716
3717     for (i = 0; i < argc; ++i)
3718         printf("%s\n", argv[i]);
3719
3720     return 0;
3721 }
3722
3723 /
```

```
*-----  
-----  
3724     exec sonrasında yeni bir proses yaratılmamaktadır. Yalnızca mevcut      ↵  
         proses hayatını başka bir program koduya  
3725     devam ettirmektedir  
3726 -----  
-----*/  
3727  
3728 /* sample.c */  
3729  
3730 #include <stdio.h>  
3731 #include <stdlib.h>  
3732 #include <unistd.h>  
3733  
3734 void exit_sys(const char *msg);  
3735  
3736 int main(void)  
3737 {  
3738     printf("sample begins...\n");  
3739  
3740     printf("sample process id: %ld\n", (long)getpid());  
3741  
3742     if (execl("mample", "mample", (char *)NULL) == -1)  
3743         exit_sys("execl");  
3744  
3745     printf("Unreachable code!..\n");  
3746  
3747     return 0;  
3748 }  
3749  
3750 void exit_sys(const char *msg)  
3751 {  
3752     perror(msg);  
3753  
3754     exit(EXIT_FAILURE);  
3755 }  
3756  
3757 /* mample.c */  
3758  
3759 #include <stdio.h>  
3760 #include <unistd.h>  
3761  
3762 int main(void)  
3763 {  
3764     printf("mample is running...\n");  
3765  
3766     printf("mample process id: %ld\n", (long)getpid());  
3767  
3768     return 0;  
3769 }  
3770  
3771 /  
-----*
```

```
-----  
3772     execv fonksiyonu çalıştırılacak programın komut satırı argümanlarını bir dizi biçiminde bizden ister.  
3773     Bu dizinin sonu NULL adresle bitmelidir.  
3774 ----- */  
3775  
3776 #include <stdio.h>  
3777 #include <stdlib.h>  
3778 #include <unistd.h>  
3779  
3780 void exit_sys(const char *msg);  
3781  
3782 int main(void)  
3783 {  
3784     char *args[] = {"./bin/ls", "-l", "-i", NULL};  
3785  
3786     printf("sample begins...\n");  
3787  
3788     if (execv("./bin/ls", args) == -1)  
3789         exit_sys("execl");  
3790  
3791     printf("Unreachable code!..\n");  
3792  
3793     return 0;  
3794 }  
3795  
3796 void exit_sys(const char *msg)  
3797 {  
3798     perror(msg);  
3799  
3800     exit(EXIT_FAILURE);  
3801 }  
3802  
3803 / -----  
*-----  
3804     Çalıştıracağı programı ve o programın komut satırı argümanlarını parametre olarak alan program örneği. Burada execl fonksiyonunun kullanılmamayağına onun yerine execv fonksiyonunun kullanılabileceğine dikkat ediniz.  
3805  
3806 ----- */  
3807  
3808 #include <stdio.h>  
3809 #include <stdlib.h>  
3810 #include <unistd.h>  
3811  
3812 void exit_sys(const char *msg);  
3813  
3814 int main(int argc, char *argv[])  
3815 {  
3816     if (execv(argv[1], &argv[1]) == -1)
```

```
3817         exit_sys("execv");
3818
3819     printf("Unreachable code!..\n");
3820
3821     return 0;
3822 }
3823
3824 void exit_sys(const char *msg)
3825 {
3826     perror(msg);
3827
3828     exit(EXIT_FAILURE);
3829 }
3830
3831 /
*-----*
-----*
3832     Aslında genellikle (ama her zaman değil) yalnızca fork ya da yalnızca      ↗
3833     exec kullanılmaz. fork ve exec birlikte kullanılır.                      ↗
3834     Üst proses çalışmaya devam edip alt prosesin başka bir kodu      ↗
3835     çalıştırılabilmesi için önce bir kez fork yapılır, alt prosese      ↗
3836     exec uygulanır.                                                       ↗
-----*/
```

```
3836
3837 #include <stdio.h>
3838 #include <stdlib.h>
3839 #include <unistd.h>
3840
3841 void exit_sys(const char *msg);
3842
3843 int main(void)
3844 {
3845     pid_t pid;
3846
3847     if ((pid = fork()) == -1)
3848         exit_sys("fork");
3849
3850     if (pid == 0)
3851         if (execl("/bin/ls", "/bin/ls", "-l", "-i", (char *)NULL) == -1)
3852             exit_sys("execl");
3853
3854     printf("parent continues...\n");
3855
3856     return 0;
3857 }
3858
3859 void exit_sys(const char *msg)
3860 {
3861     perror(msg);
3862
3863     exit(EXIT_FAILURE);
3864 }
```

```
3865
3866  /
3867      *-----*
3868      Yukarıdaki programda && operatörü de kullanabiliirdik. && operatörünün
3869      önce sol tarafındaki operand yapılır. Sol tarafındaki operand
3870      0 ise (yanlış ise) zaten sağ tarafındaki operand hiç yapılmaz. Aşağıdaki
3871      program yukarıdaki ile eşdeğerdir.
3872
3873      -----
3874
3875  #include <stdio.h>
3876  #include <stdlib.h>
3877  #include <unistd.h>
3878
3879  void exit_sys(const char *msg);
3880
3881  int main(void)
3882  {
3883      pid_t pid;
3884
3885      if ((pid = fork()) == -1)
3886          exit_sys("fork");
3887
3888      if (pid == 0 && execl("/bin/ls", "/bin/ls", "-l", "-i", (char *)NULL) == -1)
3889          exit_sys("execl");
3890
3891      printf("parent continues...\n");
3892
3893      return 0;
3894  }
3895
3896  void exit_sys(const char *msg)
3897  {
3898      perror(msg);
3899
3900      exec işleminde exec öncesi açılmış olan dosyalar default durumda
3901      kapatılmamaktadır. Aşağıdaki örnekte exec işleminde dosyanın
3902      aslında kapatılmadığı gösterilmeye çalışılmıştır.
3903
3904  /* sample.c */
3905
3906  #include <stdio.h>
3907  #include <stdlib.h>
```

```
3908 #include <fcntl.h>
3909 #include <unistd.h>
3910
3911 void exit_sys(const char *msg);
3912
3913 int main(void)
3914 {
3915     int fd;
3916     pid_t pid;
3917     char sfd[10];
3918
3919     if ((fd = open("test.txt", O_RDONLY)) == -1)
3920         exit_sys("open");
3921
3922     sprintf(sfd, "%d", fd);
3923     if (execl("mample", "mample", sfd, (char *)NULL) == -1)
3924         exit_sys("execl");
3925
3926     printf("Unreachable code...\n");
3927
3928     return 0;
3929 }
3930
3931 void exit_sys(const char *msg)
3932 {
3933     perror(msg);
3934
3935     exit(EXIT_FAILURE);
3936 }
3937
3938 /* mample.c */
3939
3940 #include <stdio.h>
3941 #include <stdlib.h>
3942 #include <unistd.h>
3943
3944 int main(int argc, char *argv[])
3945 {
3946     char buf[100 + 1];
3947     ssize_t result;
3948     int fd;
3949
3950     if (argc != 2) {
3951         fprintf(stderr, "wrong number of arguments!..\n");
3952         exit(EXIT_FAILURE);
3953     }
3954
3955     fd = (int)strtol(argv[1], NULL, 10);
3956
3957     if ((result = read(fd, buf, 100)) == -1) {
3958         perror("read");
3959         exit(EXIT_FAILURE);
3960     }
```

```
3961     buf[result] = '\0';
3962
3963     puts(buf);
3964
3965     return 0;
3966 }
3967
3968 /
*-----*
-----*
3969 Daha dosya açılırken eğer açış modeunda O_CLOEXEC bayrağı kullanılırsa dosyanın "close on exec" bayrağı set edilmiş olur.
3970 Böylece artık exec işlemlerinde dosya otomatik biçimde kapatılacaktır. Ancak O_CLOEXEC bayrağı POSIX standartlarında yoktur.
3971 Linux sistemlerinde bulumaktadır. Aşağıdaki programda çalıştırılan mample programının açık dosyayı kullanamadığına dikkat ediniz.
3972 -----*/
3973
3974 /* sample.c */
3975
3976 #include <stdio.h>
3977 #include <stdlib.h>
3978 #include <fcntl.h>
3979 #include <unistd.h>
3980
3981 void exit_sys(const char *msg);
3982
3983 int main(void)
3984 {
3985     int fd;
3986     pid_t pid;
3987     char sfd[10];
3988
3989     if ((fd = open("test.txt", O_RDONLY|O_CLOEXEC)) == -1)
3990         exit_sys("open");
3991
3992     sprintf(sfd, "%d", fd);
3993     if (execl("mample", "mample", sfd, (char *)NULL) == -1)
3994         exit_sys("execl");
3995
3996     printf("Unreachable code...\n");
3997
3998     return 0;
3999 }
4000
4001 void exit_sys(const char *msg)
4002 {
4003     perror(msg);
4004
4005     exit(EXIT_FAILURE);
4006 }
```

```
4008 /* mample.c */
4009
4010 #include <stdio.h>
4011 #include <stdlib.h>
4012 #include <unistd.h>
4013
4014 int main(int argc, char *argv[])
4015 {
4016     char buf[100 + 1];
4017     ssize_t result;
4018     int fd;
4019
4020     if (argc != 2) {
4021         fprintf(stderr, "wrong number of arguments!..\\n");
4022         exit(EXIT_FAILURE);
4023     }
4024
4025     fd = (int)strtol(argv[1], NULL, 10);
4026
4027     if ((result = read(fd, buf, 100)) == -1) {
4028         perror("read");
4029         exit(EXIT_FAILURE);
4030     }
4031     buf[result] = '\\0';
4032
4033     puts(buf);
4034
4035     return 0;
4036 }
4037
4038 /
*-----*
-----*
4039 Dosyanın "close on exec" bayrağını set etmenin taşınabilir bir yolu      ↗
4040      fcntl fonksiyonunu kullanmaktadır. fcntl fonksiyonunda           ↗
4041      command kod olarak F_SETFD girilirse dosyanın "betimleyici bayrakları"    ↗
4042      (file descriptor flags) set edilir. Simdilik POSIX
4043      standartlarında betimleyici bayrağı olarak yalnızca FD_CLOEXEC bayrağı    ↗
4044      tanımlanmıştır. Ancak ileriye doğru uyumu korumak için
4045      bu bayrağı set ederken önce get edip bir düzeyinde OR işlemi uygulamak    ↗
4046      iyi tekniktir.
4047 -----*/*/
4048
4049 /* sample.c */
4050
4051 #include <stdio.h>
4052 #include <stdlib.h>
4053 #include <fcntl.h>
4054 #include <unistd.h>
4055
4056 void exit_sys(const char *msg);
```

```
4054 int main(void)
4055 {
4056     int fd;
4057     pid_t pid;
4058     char sfd[10];
4059
4060     if ((fd = open("test.txt", O_RDONLY)) == -1)
4061         exit_sys("open");
4062
4063     if (fcntl(fd, F_SETFD, fcntl(fd, F_GETFD)|FD_CLOEXEC) == -1)
4064         exit_sys("fcntl");
4065
4066     sprintf(sfd, "%d", fd);
4067     if (execl("mample", "mample", sfd, (char *)NULL) == -1)
4068         exit_sys("execl");
4069
4070     printf("Unreachable code...\n");
4071
4072     return 0;
4073 }
4074
4075 void exit_sys(const char *msg)
4076 {
4077     perror(msg);
4078
4079     exit(EXIT_FAILURE);
4080 }
4081
4082 /* mample.c */
4083
4084 #include <stdio.h>
4085 #include <stdlib.h>
4086 #include <unistd.h>
4087
4088 int main(int argc, char *argv[])
4089 {
4090     char buf[100 + 1];
4091     ssize_t result;
4092     int fd;
4093
4094     if (argc != 2) {
4095         fprintf(stderr, "wrong number of arguments!..\n");
4096         exit(EXIT_FAILURE);
4097     }
4098
4099     fd = (int)strtol(argv[1], NULL, 10);
4100
4101     if ((result = read(fd, buf, 100)) == -1) {
4102         perror("read");
4103         exit(EXIT_FAILURE);
4104     }
4105     buf[result] = '\0';
4106 }
```

```
4107     puts(buf);
4108
4109     return 0;
4110 }
4111
4112 /
*-----*
-----*
4113 Prosesin sonlandırılması C'de normal olarak exit standart C fonksiyonu
4114     ile yapılmalıdır. exit standart C fonksiyonu açılmış olan
4115 stdio dosyalarını flush eder ve kapatır. Ayrıca başka birtakım
4116 sonlandırma işlemlerini de yapabilmektedir. Biz özellikle bir dosya
4117 açmışken, onun içine birşeyler yazmışsak doğrudan _exit fonksiyonunu
4118 kullanırken dikkat etmeliyiz.
4119 -----*/
4120
4121 #include <stdio.h>
4122 #include <stdlib.h>
4123 #include <unistd.h>
4124
4125 void exit_sys(const char *msg);
4126
4127 int main(void)
4128 {
4129     FILE *f;
4130
4131     if ((f = fopen("test.txt", "w")) == NULL)
4132         exit_sys("fopen");
4133
4134     fprintf(f, "this is a test\n");
4135
4136     _exit(0);
4137
4138     return 0;
4139 }
4140
4141 void exit_sys(const char *msg)
4142 {
4143     perror(msg);
4144
4145     exit(EXIT_FAILURE);
4146 }
4147
4148 /
*-----*
-----*
4149 Tabii programcı _exit fonksiyonunu çağrımadan önce dosyayı flush
4150     edebilir ya da kapatabilir. Bu durumda olumsuz bir durumla
4151     karşılaşılabilir.
4152 -----*/
4153
```

```
4150 #include <stdio.h>
4151 #include <stdlib.h>
4152 #include <unistd.h>
4153
4154 void exit_sys(const char *msg);
4155
4156 int main(void)
4157 {
4158     FILE *f;
4159
4160     if ((f = fopen("test.txt", "w")) == NULL)
4161         exit_sys("fopen");
4162
4163     fprintf(f, "this is a test\n");
4164     fclose(f);
4165
4166     _exit(0);
4167
4168     return 0;
4169 }
4170
4171 void exit_sys(const char *msg)
4172 {
4173     perror(msg);
4174
4175     exit(EXIT_FAILURE);
4176 }
4177
4178 /
*-----*
-----*
4179 Üst proses fork işlemi yaptığında alt prosese tüm stdio tamponlarının da kopyalarının oluşacağına dikkat ediniz. Bu
4180 nedenle alt prosese exit yerine çoğu kez _exit POSIX fonksiyonu tercih edilmelidir.
4181 -----*/
```

```
4198     /* fflush(f); */
4199
4200
4201     if ((pid = fork()) == -1)
4202         exit_sys("fork");
4203
4204     if (pid == 0) {
4205         printf("child\n");
4206         _exit(EXIT_FAILURE);
4207     }
4208
4209     printf("parent\n");
4210
4211     return 0;
4212 }
4213
4214 void exit_sys(const char *msg)
4215 {
4216     perror(msg);
4217
4218     exit(EXIT_FAILURE);
4219 }
4220
4221 /
*-----*
-----*
4222 Alt proseste exec işlemi yapıldığında zaten tüm bellek alanı      ↗
    boşaltılmaktadır. Dolayısıyla yukarıdaki örnekte olduğu gibi      ↗
bir flush problemi ortaya çıkımayacaktır. exec ile çalıştırılan program      ↗
    exit fonksiyonuyla sonlandırılmış olsa da bir problem      ↗
oluşmaz.
4224 -----*/      ↗
4225
4226
4227 #include <stdio.h>
4228 #include <stdlib.h>
4229 #include <unistd.h>
4230
4231 void exit_sys(const char *msg);
4232
4233 int main(void)
4234 {
4235     FILE *f;
4236     pid_t pid;
4237
4238     if ((f = fopen("test.txt", "r+")) == NULL)
4239         exit_sys("fopen");
4240
4241     fprintf(f, "xxxxx\n");
4242
4243     if ((pid = fork()) == -1)
4244         exit_sys("fork");
4245
```

```
4246     if (pid == 0) {
4247         if (execl("tample", "tample", (char *)NULL) == -1)
4248             _exit(EXIT_FAILURE);
4249     }
4250
4251     printf("parent\n");
4252
4253     return 0;
4254 }
4255
4256 void exit_sys(const char *msg)
4257 {
4258     perror(msg);
4259
4260     exit(EXIT_FAILURE);
4261 }
4262
4263
4264 /
*-----*
-----*
4265     wait fonksiyonu ilk alt proses sonlanana kadar beker. Alt prosesin      ↵
        sonlanma nedenini ve exit kodunu alarak geri döner.                   ↵
4266     Ancak normal sonlanmalarda exit kodu olusmaktadır. Alt prosesin normal    ↵
        bir bicimde sonlandigini anlayabilmek icin
4267     WIFEXITED makrosu kullanilmaktadir. Alt prosesin exit kod ise          ↵
        WEXITSTATUS makrosuyla elde edilir.
4268 -----*/
```

4269 /* sample.c */

4270 #include <stdio.h>

4271 #include <stdlib.h>

4272 #include <unistd.h>

4273 #include <sys/wait.h>

4274

4275 void exit_sys(const char *msg);

4276

4277 int main(void)

4278 {

4279 pid_t pid;

4280 int status;

4281

4282 printf("sample starts...\n");

4283

4284 if ((pid = fork()) == -1)
4285 exit_sys("fork");
4286
4287 if (pid == 0 && execl("mample", "mample", (char *)NULL) == -1)
4288 exit_sys("execl");
4289
4290 if (wait(&status) == -1)

```
4293         exit_sys("wait");
4294
4295     if (WIFEXITED(status))
4296         printf("Child terminated normally: %d\n", WEXITSTATUS(status));
4297     else
4298         printf("Child terminated abnormally!..\n");
4299
4300     return 0;
4301 }
4302
4303 void exit_sys(const char *msg)
4304 {
4305     perror(msg);
4306
4307     exit(EXIT_FAILURE);
4308 }
4309
4310 /* mample.c */
4311
4312 #include <stdio.h>
4313 #include <stdlib.h>
4314 #include <unistd.h>
4315
4316 int main(void)
4317 {
4318     int i;
4319
4320     printf("mample starts\n");
4321
4322     for (i = 0; i < 10; ++i) {
4323         printf("%d\n", i);
4324         sleep(1);
4325     }
4326
4327     return 100;
4328 }
4329
4330 /
*-----*
-----*  
4331     Üst proses kaç kere alt proses yaratmışsa o kadar wait uygulamalıdır. Bu
        durum biraz sıkıntılı olabilmektedir. Tabii
4332     eğer alt proses zaten sonlanmışsa wait hiç bekleme yapmaz.
4333 -----*/
4334
4335 /* sample.c */
4336
4337 #include <stdio.h>
4338 #include <stdlib.h>
4339 #include <unistd.h>
4340 #include <sys/wait.h>
4341
```

```
4342 void exit_sys(const char *msg);
4343
4344 int main(void)
4345 {
4346     pid_t pid1, pid2, pid_result;
4347     int status;
4348     int i;
4349
4350     printf("sample starts...\n");
4351
4352     if ((pid1 = fork()) == -1)
4353         exit_sys("fork");
4354
4355     if (pid1 == 0 && execl("mample", "mample", (char *)NULL) == -1)
4356         exit_sys("execl");
4357
4358     if ((pid2 = fork()) == -1)
4359         exit_sys("fork");
4360
4361     if (pid2 == 0 && execl("/bin/ls", "/bin/ls", "-l", (char *)NULL) == -1)
4362         exit_sys("execl");
4363
4364     for (i = 0; i < 2; ++i) {
4365         if ((pid_result = wait(&status)) == -1)
4366             exit_sys("wait");
4367
4368         if (WIFEXITED(status))
4369             printf("%s terminated normally: %d\n", pid_result == pid1 ?      ↵
4370                   "mample" : "ls", WEXITSTATUS(status));
4371         else
4372             printf("%s Child terminated abnormally!..\n", pid_result ==      ↵
4373                   pid1 ? "mample" : "ls");
4374     }
4375
4376
4377     return 0;
4378 }
4379
4380 void exit_sys(const char *msg)
4381 {
4382     perror(msg);
4383     exit(EXIT_FAILURE);
4384 }
4385
4386 /* mample.c */
4387
4388 #include <stdio.h>
4389 #include <stdlib.h>
4390 #include <unistd.h>
4391
4392 int main(void)
4393 {
4394     int i;
```

```
4393     printf("mample starts\n");
4394
4395     for (i = 0; i < 10; ++i) {
4396         printf("%d\n", i);
4397         sleep(1);
4398     }
4399
4400     return 100;
4401 }
4402 }
4403
4404 /
*-----*
-----*
4405     waitpid fonksiyonu wait fonksiyonun daha gelişmiş bir biçimidir. waitpid
4406     fonksiyonu ile biz proses id'sini bildiğimiz
4407     herhangi bir alt prosesin sonlanmasını bekleyebiliriz: waitpid
4408     fonksiyonun birinci parametresi değişik seçenekler sunmaktadır.
4409     Son parametre tipik olarak 0 geçilebilir ya da WNOHANG geçilebilir.
4410     WNOHANG waitpid fonksiyonunun bloke olmasını engellemektedir.
4411 -----*/
4410 /* sample.c */
4411
4412 #include <stdio.h>
4413 #include <stdlib.h>
4414 #include <unistd.h>
4415 #include <sys/wait.h>
4416
4417 void exit_sys(const char *msg);
4418
4419 int main(void)
4420 {
4421     pid_t pid1, pid2, pid_result;
4422     int status;
4423     int i;
4424
4425     printf("sample starts...\n");
4426
4427     if ((pid1 = fork()) == -1)
4428         exit_sys("fork");
4429
4430     if (pid1 == 0 && execl("mample", "mample", (char *)NULL) == -1)
4431         exit_sys("execl");
4432
4433     if ((pid2 = fork()) == -1)
4434         exit_sys("fork");
4435
4436     if (pid2 == 0 && execl("/bin/ls", "/bin/ls", "-l", (char *)NULL) == -1)
4437         exit_sys("execl");
4438
4439     if (waitpid(pid1, &status, 0) == -1)
```

```
4440         exit_sys("wait");
4441
4442     if (WIFEXITED(status))
4443         printf("mample terminated normally: %d\n", WEXITSTATUS(status));
4444     else
4445         printf("mample terminated abnormally!..\n");
4446
4447     if (waitpid(pid2, &status, 0) == -1)
4448         exit_sys("wait");
4449
4450     if (WIFEXITED(status))
4451         printf("ls terminated normally: %d\n", WEXITSTATUS(status));
4452     else
4453         printf("ls terminated abnormally!..\n");
4454
4455     return 0;
4456 }
4457
4458 void exit_sys(const char *msg)
4459 {
4460     perror(msg);
4461
4462     exit(EXIT_FAILURE);
4463 }
4464
4465 /* mample.c */
4466
4467 #include <stdio.h>
4468 #include <stdlib.h>
4469 #include <unistd.h>
4470
4471 int main(void)
4472 {
4473     int i;
4474
4475     printf("mample starts\n");
4476
4477     for (i = 0; i < 10; ++i) {
4478         printf("%d\n", i);
4479         sleep(1);
4480     }
4481
4482     return 100;
4483 }
4484
4485 /
*-----*-----*-----*
```

4486 Üst proses çalışmaya devam ederken alt proses sonlanmışsa ve üst proses ↵
wait fonksiyonlarını uygulamamışsa alt proses
4487 zombie durumda olur. İşletim sistemi sonlanan proseslerin exit kodlarını ↵
wait fonksiyonlarıyla onların üst prosesleri alır
4488 diye onlara ilişkin proses kontrol bloklarını ve prosess id değerlerini ↵

boşaltmamaktadır. Bu da patolojik bir durumdur.

4489 Zombie'lik tipik olarka wait fonksiyonlarıyla engellenebilir. Ancak →
SIGCHLD sinyali yoluyla da otomatik engelleme yöntemleri →
vardır. Aşağıdaki program zombie proses oluşturmaktadır. Bu programı →
çalıştırıp başka bir terminalden ps -al komutu ile →
alt prosesin durumuna dikkat ediniz.

4492 -----*/

4493

4494 #include <stdio.h>

4495 #include <stdlib.h>

4496 #include <unistd.h>

4497 #include <sys/wait.h>

4498

4499 void exit_sys(const char *msg);

4500

4501 int main(void)

4502 {

4503 pid_t pid;

4504

4505 printf("sample starts...\n");

4506

4507 if ((pid = fork()) == -1)

4508 exit_sys("fork");

4509

4510 if (pid == 0)

4511 exit(EXIT_SUCCESS);

4512

4513 printf("Child is zombie now. Press ENTER to exit...\n");

4514 getchar();

4515

4516 return 0;

4517 }

4518

4519 void exit_sys(const char *msg)

4520 {

4521 perror(msg);

4522

4523 exit(EXIT_FAILURE);

4524 }

4525

4526 /

-----*/

4527 exec fonksiyonları ile çalıştırılabilir (executable) olmayan dosyalar →
da çalıştırılabilir. exec fonksiyonları önce çalıştırılmak →
istenen dosyanın çalıştırılabilir olup olmadığına (yani ELF formatına →
sahip olup olmadığına) bakmaktadır. Eğer dosya çalıştırılabilir →
değilse bu durumda onun ilk satırını okuyup orada belirtilen programı →
çalıştırırlar. Asıl dosyanın yol ifadesini de o dosyaya →
komut satırı argümanı olarak verirler.

4531 -----*/

```
4532
4533 /* sample.c */
4534
4535 #include <stdio.h>
4536 #include <stdlib.h>
4537 #include <unistd.h>
4538 #include <sys/wait.h>
4539
4540 void exit_sys(const char *msg);
4541
4542 int main(void)
4543 {
4544     pid_t pid;
4545
4546     printf("sample starts...\n");
4547
4548     if ((pid = fork()) == -1)
4549         exit_sys("fork");
4550
4551     if (pid == 0 && execl("sample.py", "sample.py", (char *)NULL) == -1)
4552         exit_sys("execl");
4553
4554     if (wait(NULL) == -1)
4555         exit(EXIT_FAILURE);
4556
4557     return 0;
4558 }
4559
4560 void exit_sys(const char *msg)
4561 {
4562     perror(msg);
4563
4564     exit(EXIT_FAILURE);
4565 }
4566
4567 /* sample.py */
4568
4569 #! /usr/bin/python
4570
4571 for i in range(10):
4572     print(i)
4573
4574 /
*-----→
-----→
4575 Kabuk programı (bash) önce dosyayı fork ve exec yaparak çalıştırma →
    çalışır. Eğer exec başarısız olursa onu bir "shell script" →
4576 olarak düşünür ve doğrudan kendisi açarak bir scrip biçiminde →
    çalıştırır. Bu durumda biz shell script'lerin başına shebang →
4577 koymasak da onu komu satırında çalıştırabiliriz. Ancak exec yaparak →
    çalıştırıramayız. Aşağıda dosya bu nedenle komut satırında →
4578 çalıştırılabilir ancak exec yapılamaz →
4579 -----→
```

```
-----*/
4580
4581 # sample.sh
4582
4583 for i in 10 20 30 40 50
4584 do
4585     echo $i
4586 done
4587
4588 /
-----*
-----*
-----*
4589     Tabii bash script dosyalarının da yine shebang'e sahip olması iyi bir tekniktir
4590 -----
-----*/
4591
4592 #! /bin/bash
4593
4594 for i in 10 20 30 40 50
4595 do
4596     echo $i
4597 done
4598
4599 /
-----*
-----*
-----*
4600     Biz bir shebang'lı text dosyası çalıştırırken komut satırı argümanı da verebiliriz. Bu durumda bu komut satırı argümanları shebang'te belirtlen programın komut satırı argümanları olur. Örneğin test.txt dosyasında shebang olarak belirtilen dosya mample ise ./test.txt ali veli selami aslında mample test.txt ali veli selami durumuna gelir. Aşağıdaki programda sample programı execl ile test.txt dosyasını çalıştırmaktadır. Bu da mample programının çalıştırılmasına yol açacaktır.
4601
4602
4603
4604 -----
-----*/
4605
4606 /* sample.c */
4607
4608 #include <stdio.h>
4609 #include <stdlib.h>
4610 #include <unistd.h>
4611 #include <sys/wait.h>
4612
4613 void exit_sys(const char *msg);
4614
4615 int main(void)
4616 {
4617     pid_t pid;
4618
4619     printf("sample starts...\n");
4620
```

```
4621     if ((pid = fork()) == -1)
4622         exit_sys("fork");
4623
4624     if (pid == 0 && execl("test.txt", "test.txt", "ali", "veli", "selami", NULL) == -1)
4625         exit_sys("execl");
4626
4627     if (wait(NULL) == -1)
4628         exit(EXIT_FAILURE);
4629
4630     return 0;
4631 }
4632
4633 void exit_sys(const char *msg)
4634 {
4635     perror(msg);
4636
4637     exit(EXIT_FAILURE);
4638 }
4639
4640 /* test.txt */
4641
4642 #! /home/csd/Study/Unix-Linux-SysProg/mample
4643
4644 /* mample.c */
4645
4646 #include <stdio.h>
4647
4648 int main(int argc, char *argv[])
4649 {
4650     int i;
4651
4652     for (i = 0; i < argc; ++i)
4653         printf("%s\n", argv[i]);
4654
4655     return 0;
4656 }
4657
4658 /
*-----*
-----
```

4659 shebang'te belirtilen programa da shebanh satırında komut satırı
4660 argümanı verilebilir. Bu argümanlar tek bir komut satırı
4661 argümanı olarak shebang'te belirtilen programa arg[1] biçiminde
4662 aktarılmaktadır. Örneğin shabang şöyle olsun:

4663
4664#! /home/csd/Study/Unix-Linux-SysProg/mample ankara istanbul adana
4665
4666 Biz de test.txt dosyasını şöyle çalıştırılmış olalım:
4667
4668./test.txt ali veli selami
4669
4670 Bu durum aşağıdaki gibi bir çalıştırmayla eşdeğer olacaktır:

```
4669
4670     /home/csd/Study/Unix-Linux-SysProg/mample "ankara istanbul adana" ./    ↵
        test.txt ali veli selami
4671
4672 -----*/          ↵
4673 /* test.txt */
4674
4675 #! /home/csd/Study/Unix-Linux-SysProg/mample ankara istanbul adana
4676
4677 /
4678 *-----*/          ↵
4679 -----*/          ↵
4680
4681 /* test.txt
4682
4683 /* mycat.c */
4684
4685 #include <stdio.h>
4686 #include <stdlib.h>
4687
4688 int main(int argc, char *argv[])
4689 {
4690     FILE *f;
4691     int ch;
4692
4693     if (argc != 2) {
4694         fprintf(stderr, "wrong number of arguments!..\n");
4695         exit(EXIT_FAILURE);
4696     }
4697
4698     if ((f = fopen(argv[1], "r")) == NULL) {
4699         fprintf(stderr, "cannot open file: %s\n", argv[1]);
4700         exit(EXIT_FAILURE);
4701     }
4702
4703     while ((ch = fgetc(f)) != EOF)
4704         putchar(ch);
4705
4706     if (ferror(f)) {
4707         fprintf(stderr, "Cannot read file: %s\n", argv[1]);
4708         exit(EXIT_FAILURE);
4709     }
4710
4711     return 0;
4712 }
4713
4714 /* test.txt */
4715
```

```
4716  #! /home/csd/Study/Unix-Linux-SysProg/mycat
4717
4718  bugün hava çok güzel
4719  corona virüslerinin hepsi öldü
4720
4721 /
*-----*
-----*
4722      Shell programları -c ile tek bir komutu çalışırıp sonlanabilmektedir.    ↵
        Dolayısıyla biz komut satırında verdiğimiz tüm komutları
4723  aslında shell programına çalıştırabiliriz.
4724 -----
*-----*/
```

```
4725
4726 #include <stdio.h>
4727 #include <stdlib.h>
4728 #include <unistd.h>
4729 #include <sys/wait.h>
4730
4731 void exit_sys(const char *msg);
4732
4733 int main(int argc, char *argv[])
4734 {
4735     pid_t pid;
4736
4737     if (argc != 2) {
4738         fprintf(stderr, "wrong number of arguments!..\\n");
4739         exit(EXIT_FAILURE);
4740     }
4741
4742     if ((pid = fork()) == -1)
4743         exit_sys("fork");
4744
4745     if (pid == 0 && execl("/bin/bash", "/bin/bash", "-c", argv[1], (char *) NULL) == -1)
4746         exit_sys("execl");
4747
4748     if (wait(NULL) == -1)
4749         exit(EXIT_FAILURE);
4750
4751     return 0;
4752 }
4753
4754 void exit_sys(const char *msg)
4755 {
4756     perror(msg);
4757
4758     exit(EXIT_FAILURE);
4759 }
```

```
4760
4761 /
*-----*
```

```
4762     system standart bir C fonksiyonudur. /bin/sh shell programını -c      ↵
        seçeneği ile çalıştırır. Yani bizim fonksiyona verdığımız      ↵
        komut system tarafından aslında shell programına çalıştırılmaktadır.      ↵
        system fork ya da waitpid fonksiyonlarında başarısız      ↵
        olursa -1 değerine exec fonksiyonlarında başarısız olursa 127 değerine      ↵
        geri döner. Eğer başarılı olursa shell programının exit      ↵
        koduyla geri dönümektedir. Zaten shell de -c seçeneği ile son      ↵
        çalıştırıldığı komutun exit koduyla geri döner.      ↵
4766 -----
-----*/
```

```
4767
4768 #include <stdio.h>
4769 #include <stdlib.h>
4770 #include <unistd.h>
4771 #include <sys/wait.h>
4772
4773 void exit_sys(const char *msg);
4774
4775 int mysystem(const char *command)
4776 {
4777     pid_t pid;
4778     int status;
4779
4780     if (command == NULL)
4781         return 1;
4782
4783     if ((pid = fork()) == -1)
4784         return -1;
4785
4786     if (pid == 0 && execl("/bin/sh", "/bin/sh", "-c", command, (char *)NULL) == -1)
4787         _exit(127); /* Neden exit değil de _exit? */
4788
4789     if (waitpid(pid, &status, 0) == -1)
4790         return -1;
4791
4792     return status;
4793 }
4794
4795 int main(int argc, char *argv[])
4796 {
4797     int result;
4798
4799     result = mysystem("ls -l");
4800     if (result == -1 || result == 127)
4801         exit_sys("mysystem");
4802
4803     printf("mysystem terminated normally with return value %d\n", result);
4804
4805     return 0;
4806 }
4807
4808 void exit_sys(const char *msg)
```

```
4809  {
4810      perror(msg);
4811
4812      exit(EXIT_FAILURE);
4813 }
4814
4815 /
*-----*
-----*
4816     getenv standart C fonksiyonu (aynı zamanda POSIX fonksiyonu) çevre
        değişkeninin ismini (anahtarı) alıp ona karşı gelen
4817     değeri bize verir. Eğer öyle bir çevre değişkeni yoksa getenv NULL
        adresine geri dönmektedir. getenv fonksiyonu başarısızlık
4818     durumunda errno değerini set etmez.
4819 -----*/
4820
4821 #include <stdio.h>
4822 #include <stdlib.h>
4823
4824 int main(int argc, char *argv[])
4825 {
4826     char *result;
4827     int i;
4828
4829     if (argc == 1) {
4830         fprintf(stderr, "wrong number of arguments!..\\n");
4831         exit(EXIT_FAILURE);
4832     }
4833
4834     for (i = 1; i < argc; ++i) {
4835         if ((result = getenv(argv[i])) == NULL) {
4836             fprintf(stderr, "cannot get environment variable: %s\\n",
4837                     [i]);
4838             continue;
4839         }
4840         printf("%s --> %s\\n", argv[i], result);
4841     }
4842
4843     return 0;
4844 }
4845 /
*-----*
-----*
4846     setenv POSIX fonksiyonu prosesin çevre değişken listesine yeni bir
        anahtar-değer çifti ekler
4847 -----*/
4848
4849 #include <stdio.h>
4850 #include <stdlib.h>
4851
```

```
4852 void exit_sys(const char *msg);
4853
4854 int main(void)
4855 {
4856     char *result;
4857
4858     if (putenv("city=istanbul") == -1)
4859         exit_sys("setenv");
4860
4861     if ((result = getenv("city")) == NULL) {
4862         fprintf(stderr, "cannot get environment variable!..\n");
4863         exit(EXIT_FAILURE);
4864     }
4865     puts(result);
4866
4867     return 0;
4868 }
4869
4870 void exit_sys(const char *msg)
4871 {
4872     perror(msg);
4873
4874     exit(EXIT_FAILURE);
4875 }
4876
4877 /
*-----*
-----*
4878 putenv fonksiyonu da setenv fonksiyonuna benzemektedir. Aradaki fark
    putenv fonksiyonun "anahtar=değer" biçiminde tek bir yazı
4879 almasıdır. Eğer ilgili değişken zaten varsa değeri değiştirilir.
4880 -----*/
4881
4882 #include <stdio.h>
4883 #include <stdlib.h>
4884
4885 void exit_sys(const char *msg);
4886
4887 int main(void)
4888 {
4889     char *result;
4890
4891     if (putenv("city=istanbul") == -1)
4892         exit_sys("setenv");
4893
4894     if ((result = getenv("city")) == NULL) {
4895         fprintf(stderr, "cannot get environment variable!..\n");
4896         exit(EXIT_FAILURE);
4897     }
4898     puts(result);
4899
4900     return 0;
```

```
4901 }
4902
4903 void exit_sys(const char *msg)
4904 {
4905     perror(msg);
4906
4907     exit(EXIT_FAILURE);
4908 }
4909
4910 /
*-----*
----->
4911 Prosesin tüm çevre değişken listesinin yazdırılması. environ global
değişkeninin extern bildirimi hiçbir başlık dosyasında
yapılmamıştır.
4912 -----*
```

```
4913 -----*/
4914
4915 #include <stdio.h>
4916
4917 extern char **environ;
4918
4919 int main(void)
4920 {
4921     int i;
4922
4923     for (i = 0; environ[i] != NULL; ++i)
4924         puts(environ[i]);
4925
4926     return 0;
4927 }
4928
4929 /
*-----*
----->
4930 Çevre değişkenleri birtakım parametrik bilgilerin kolay oluşturulması
    için kullanılabilmektedir. Örneğin bir program database
4931 dosyasını DATALOC çevre değişkeni ile belirtilen bir dizinde arayacak
    biçimde yazılmış olabilir.
4932 -----*
```

```
4933 -----*/
4934
4935 #include <stdio.h>
4936 #include <stdlib.h>
4937
4938 int main(void)
4939 {
4940     FILE *f;
4941     char *val;
4942     char path[1024] = "datafile";
4943
4944     if ((val = getenv("DATALOC")) != NULL)
4945         sprintf(path, "%s/datafile", val);
```

```
4945
4946     if ((f = fopen(path, "r+")) == NULL) {
4947         fprintf(stderr, "cannot open file!..\\n");
4948         exit(EXIT_FAILURE);
4949     }
4950     /* .... */
4951
4952     fclose(f);
4953
4954     return 0;
4955 }
4956
4957 /
*-----*
-----*
4958     exec fonksiyonun p'li versiyonlarındaki yol ifadelerinde hiç '/'      ↵
        karakteri yoksa bu durumda bu fonksiyonlar prosesin      ↵
4959 PATH isimli çevre değişkenine başvururlar. Bu PATH çevre değişkenin      ↵
        değeri olan yazıyı ':' karakterlerinden parse ederler      ↵
4960 sonra sırasıyla ilgili dosyayı o dizinlerde ararlar. İlk bulunan      ↵
        dizindeki programı exec ederler. Eğer yol ifadesinde en az bir      ↵
4961 '/' karakteri varsa bu durumda fonksiyonun davranışını p'siz versiyonlarla      ↵
        tamamen aynıdır. Aşağıdaki programda "ls" programı PATH      ↵
        çevre değişkeninde belirtilen dizinlerden birinde bulunacaktır. exec      ↵
        fonksiyonlarının p'li biçimleri eğer dyol ifadesinde hiç      ↵
4963 '/' karakteri yoksa prosesin çalışma dizinine hiç bakmazlar.      ↵
4964 -----*/
4965
4966 #include <stdio.h>
4967 #include <stdlib.h>
4968 #include <unistd.h>
4969 #include <sys/wait.h>
4970
4971 void exit_sys(const char *msg);
4972
4973 int main(void)
4974 {
4975     pid_t pid;
4976
4977     if ((pid = fork()) == -1)
4978         exit_sys("fork");
4979
4980     if (pid == 0 && execlp("ls", "ls", "-l", (char *)NULL) == -1)
4981         exit_sys("execl");
4982
4983     if (waitpid(pid, NULL, 0) == -1)
4984         exit_sys("waitpid");
4985
4986     return 0;
4987 }
4988
4989 void exit_sys(const char *msg)
```

```
4990 {
4991     perror(msg);
4992
4993     exit(EXIT_FAILURE);
4994 }
4995
4996 /
*-----*
-----*
4997     exec fonksiyonlarının p'li versiyonlarında "./prog" biminde yol ifadesi
        prosesin çalışma dizinindeki programı
4998     çalıştırma anlamına gelmektedir.
4999 -----*
-----*/
5000
5001 #include <stdio.h>
5002 #include <stdlib.h>
5003 #include <unistd.h>
5004 #include <sys/wait.h>
5005
5006 void exit_sys(const char *msg);
5007
5008 int main(void)
5009 {
5010     pid_t pid;
5011
5012     if ((pid = fork()) == -1)
5013         exit_sys("fork");
5014
5015     if (pid == 0 && execl("./mample", "mample", (char *)NULL) == -1)
5016         exit_sys("execl");
5017
5018     if (waitpid(pid, NULL, 0) == -1)
5019         exit_sys("waitpid");
5020
5021     return 0;
5022 }
5023
5024 void exit_sys(const char *msg)
5025 {
5026     perror(msg);
5027
5028     exit(EXIT_FAILURE);
5029 }
5030
5031 /
*-----*
-----*
5032     execvp fonksiyonu execv fonksiyonun p'li biçimidir. Shell programları
        genel olarak komut satırından girilen yol ifadelerini
5033     exec fonksiyonlarının p'li versiyonlarıyla çalışmaktadır. Bu nedenle
        biz prosesin çalışma dizini içerisindeki bir
5034     programı çalıştırabilmek için "./isim" biçiminde bir yol ifadesi
```

```
    kullanmak zorunda kalmaktayız. Shell programları güvenlik
5035    amacıyla exec fonksiyonlarının p'li versiyonlarını kullanmaktadır.
5036    -----
5037    -----
5038 #include <stdio.h>
5039 #include <stdlib.h>
5040 #include <unistd.h>
5041 #include <sys/wait.h>
5042
5043 void exit_sys(const char *msg);
5044
5045 int main(int argc, char *argv[])
5046 {
5047     pid_t pid;
5048
5049     if (argc == 1) {
5050         fprintf(stderr, "wrong number of arguments!..\n");
5051         exit(EXIT_FAILURE);
5052     }
5053
5054     if ((pid = fork()) == -1)
5055         exit_sys("fork");
5056
5057     if (pid == 0 && execvp(argv[1], &argv[1]) == -1)
5058         exit_sys("execl");
5059
5060     if (waitpid(pid, NULL, 0) == -1)
5061         exit_sys("waitpid");
5062
5063     return 0;
5064 }
5065
5066 void exit_sys(const char *msg)
5067 {
5068     perror(msg);
5069
5070     exit(EXIT_FAILURE);
5071 }
5072
5073 /
5074    *
5075    -----
5076    execle fonksiyonu exec fonksiyonlarının 'e' versiyonlarından biridir.    ↵
5077    exec fonksiyonlarının e'li versyonları ilgili programı,
5078    çalıştırırken çevre değişken listesini de değiştirmektedir.
5079    -----
5080    */
5081 /* sample.c */
```

```
5082 #include <unistd.h>
5083 #include <sys/wait.h>
5084
5085 void exit_sys(const char *msg);
5086
5087 int main(void)
5088 {
5089     pid_t pid;
5090     char *envs[] = {"city=istanbul", "name=ali", NULL};
5091
5092     if ((pid = fork()) == -1)
5093         exit_sys("fork");
5094
5095     if (pid == 0 && execle("./mample", "./mample", "ali", "veli", "selami", \
5096         (char *)NULL, envs) == -1)
5097         exit_sys("execl");
5098
5099     if (waitpid(pid, NULL, 0) == -1)
5100         exit_sys("waitpid");
5101
5102     return 0;
5103 }
5104 void exit_sys(const char *msg)
5105 {
5106     perror(msg);
5107
5108     exit(EXIT_FAILURE);
5109 }
5110 /* mample.c */
5111
5112 #include <stdio.h>
5113 #include <stdlib.h>
5114
5115 extern char **environ;
5116
5117
5118 int main(int argc, char *argv[])
5119 {
5120     int i;
5121
5122     printf("Command line arguments:");
5123
5124     for (i = 0; i < argc; ++i)
5125         puts(argv[i]);
5126
5127     printf("Environment variables:\n");
5128
5129     for (i = 0; environ[i] != NULL; ++i)
5130         puts(environ[i]);
5131
5132     return 0;
5133 }
```

```
5134
5135  /
5136      *-----*
5137      ----- execve fonksiyonu da execle ile benzerdir. Ancak bu fonksiyon komut
5138      ----- satırı argümanlarını tek tek değil dizi olarak alır.
5139  -----
5140  -----
5141  /* sample.c */
5142
5143  #include <stdio.h>
5144  #include <stdlib.h>
5145  #include <unistd.h>
5146  #include <sys/wait.h>
5147
5148  void exit_sys(const char *msg);
5149
5150  int main(void)
5151  {
5152      pid_t pid;
5153      char *args[] = {"./mample", "ali", "veli", "selami", NULL};
5154      if ((pid = fork()) == -1)
5155          exit_sys("fork");
5156
5157      if (pid == 0 && execve("./mample", args, envs) == -1)
5158          exit_sys("exec");
5159
5160      if (waitpid(pid, NULL, 0) == -1)
5161          exit_sys("waitpid");
5162
5163      return 0;
5164  }
5165
5166  void perror(const char *msg)
5167  {
5168      perror(msg);
5169
5170      exit(EXIT_FAILURE);
5171  }
5172
5173  /* mample.c */
5174
5175  #include <stdio.h>
5176  #include <stdlib.h>
5177
5178  extern char **environ;
5179
5180  int main(int argc, char *argv[])
5181  {
5182      int i;
```

```
5183     printf("Command line arguments:");
5184
5185     for (i = 0; i < argc; ++i)
5186         puts(argv[i]);
5187
5188     printf("Environment variables:\n");
5189
5190     for (i = 0; environ[i] != NULL; ++i)
5191         puts(environ[i]);
5192
5193
5194     return 0;
5195 }
5196
5197 /
*-----*
-----*
5198 fexecve fonksiyonu tamamen execve fonksiyonu gibidir. Tek farkı →
      çalıştırılacak dosyayı yol ifadesi ile almak yerine dosya →
5199 betimleyicisi yoluyla almasıdır. Yani çalıştırılacak dosya open →
      fonksiyonıyla O_RDONLY moda açılmışsa biz doğrudan fexecve →
5200 fonksiyonunu da kullanabiliriz.
5201 -----*/
5202
5203 /* sample.c */
5204
5205 #include <stdio.h>
5206 #include <stdlib.h>
5207 #include <fcntl.h>
5208 #include <unistd.h>
5209 #include <sys/wait.h>
5210
5211 void exit_sys(const char *msg);
5212
5213 int main(void)
5214 {
5215     pid_t pid;
5216     char *args[] = {"./mample", "ali", "veli", "selami", NULL};
5217     char *envs[] = {"city=istanbul", "name=ali", NULL};
5218     int fd;
5219
5220     if ((pid = fork()) == -1)
5221         exit_sys("fork");
5222
5223     if (pid == 0) {
5224         if ((fd = open("mample", O_RDONLY)) == -1)
5225             exit_sys("open");
5226         if (fexecve(fd, args, envs) == -1)
5227             exit_sys("execve");
5228         /* unreachable code */
5229     }
5230
```

```
5231     if (waitpid(pid, NULL, 0) == -1)
5232         exit_sys("waitpid");
5233
5234     return 0;
5235 }
5236
5237 void exit_sys(const char *msg)
5238 {
5239     perror(msg);
5240
5241     exit(EXIT_FAILURE);
5242 }
5243
5244 /* mample.c */
5245
5246 #include <stdio.h>
5247 #include <stdlib.h>
5248
5249 extern char **environ;
5250
5251 int main(int argc, char *argv[])
5252 {
5253     int i;
5254
5255     printf("Command line arguments:");
5256
5257     for (i = 0; i < argc; ++i)
5258         puts(argv[i]);
5259
5260     printf("Environment variables:\n");
5261
5262     for (i = 0; environ[i] != NULL; ++i)
5263         puts(environ[i]);
5264
5265     return 0;
5266 }
5267
5268 /
*-----*
----- Sistem fonksiyonlarının numara belirtilerek syscall isimli Linux
fonksiyonuyla çağrılmamasına örnek
-----*/
5271
5272 #include <stdio.h>
5273 #include <stdlib.h>
5274 #include <fcntl.h>
5275 #include <unistd.h>
5276 #include <sys/syscall.h>
5277 #include <sys/wait.h>
5278
5279 void exit_sys(const char *msg);
```

```
5280
5281     extern char **environ;
5282
5283 int main(void)
5284 {
5285     char *args[] = {"./bin/ls", "-l", NULL};
5286
5287     if (syscall(SYS_execve, "/bin/ls", args, environ) == -1)
5288         exit_sys("execve");
5289
5290     printf("Unreachable code!");
5291
5292     return 0;
5293 }
5294
5295 void exit_sys(const char *msg)
5296 {
5297     perror(msg);
5298
5299     exit(EXIT_FAILURE);
5300 }
5301
5302 /
*-----*
-----*
5303     chmod fonksiyonuyla dosyanın set user id, set group id ve sticky      ↗
      özelliklerinin set edilmesi (S_IFMT sembolik sabiti dosya türünü      ↗
      maskelemek için      ↗
5304     kullanılmaktadır. Bu durumda dosya erişim hakları için bu sembolik      ↗
      sabitin tersi ile & işlemi yapmak gereklidir.)      ↗
5305 -----*/
```

5306

```
5307 #include <stdio.h>
5308 #include <stdlib.h>
5309 #include <sys/stat.h>
5310
5311 void exit_sys(const char *msg);
5312
5313 int main(void)
5314 {
5315     struct stat finfo;
5316
5317     if (stat("mample", &finfo) == -1)
5318         exit_sys("stat");
5319
5320     if (chmod("mample", finfo.st_mode & ~S_IFMT|S_ISUID|S_ISGID|S_ISVTX) == -1)
5321         exit_sys("chmod");
5322
5323     return 0;
5324 }
```

```
5326 void exit_sys(const char *msg)
5327 {
5328     perror(msg);
5329
5330     exit(EXIT_FAILURE);
5331 }
5332
5333 /
*-----*
-----*
5334 Etkin kullanıcı id'si csd olan bir kabukta mample isimli kullanıcı      ↵
      id'si student olan bir programı çalıştırırsak prosesin      ↵
5335 etkin kullanıcı id'si yine csd olarak kalır. Fakat mample program      ↵
      dosyasının "set user id" özelliği set edilmişse bu durumda      ↵
5336 etkin kullanıcı id'si csd olan proses mample dosyasını exec yaparsa      ↵
      artık prosesin etkin kullanıcı id'si "student" olacaktır.      ↵
5337 Aşağıdaki mample.c programının program dosyasının set user id      ↵
      özelliğinin set edilmiş olduğunu düşünelim. student.txt dosyasının da      ↵
5338 erişim hakları rw-r--r-- biçiminde olsun. student.txt dosyasının      ↵
      kullanıcı id'si de student'tır. İşte bu durumda biz kim olursak olalım      ↵
5339 eğer mample dosyasını çalışma hakkına sahipsek bu mample student.txt      ↵
      dosyasını write modda açanilecektir.      ↵
5340 -----*/
5341
5342 /* mample.c */
5343
5344 #include <stdio.h>
5345 #include <stdlib.h>
5346 #include <fcntl.h>
5347 #include <unistd.h>
5348
5349 void exit_sys(const char *msg);
5350
5351 int main(void)
5352 {
5353     int fd;
5354
5355     if ((fd = open("student.txt", O_WRONLY)) == -1)
5356         exit_sys("open");
5357
5358     printf("success..\n");
5359
5360     return 0;
5361 }
5362
5363 void exit_sys(const char *msg)
5364 {
5365     perror(msg);
5366
5367     exit(EXIT_FAILURE);
5368 }
5369
```

```
5370  /
5371  *-----*
5371      Prosesin gerçek kullanıcı id'si getuid fonksiyonıyla, etkin kullanıcı      ↵
5371          id'si geteuid fonksiyonıyla, gerçek grup id'si
5372      getgid fonksiyonıyla etkin grup id'si de getegid fonksiyonıyla elde      ↵
5372          edilebilir. Saklı kullanıcı ve grup id'lerini elde eden
5373      bir POSIX fonksiyonu yoktur ancak Linux sistemlerinde bir fonksiyon      ↵
5373          vardır.
5374  -----*/
5375
5376 #include <stdio.h>
5377 #include <unistd.h>
5378
5379 int main(void)
5380 {
5381     uid_t ruid, euid;
5382     gid_t rgid, egid;
5383
5384     ruid = getuid();
5385     euid = geteuid();
5386
5387     rgid = getgid();
5388     egid = getegid();
5389
5390     printf("Real User Id: %lu\n", (unsigned long)ruid);
5391     printf("Effective User Id: %lu\n", (unsigned long)euid);
5392     printf("Real Group Id: %lu\n", (unsigned long)rgid);
5393     printf("Effective Group Id: %lu\n", (unsigned long)egid);
5394
5395     return 0;
5396 }
5397
5398 /*
5398 *-----*
5399      Aşağıdaki örnekte sample programı başka bir kullanıcıya ait olan ve set      ↵
5400          user id özelliği set edilmiş olan mample programını
5400          çalıştırmaktadır. Etkin kullanıcı id'sinin değiştiğine dikkat ediniz.      ↵
5400          (Denemeyi yapmadan önce mample programının sahipliğini
5401          değiştirip set user id özelliğini de set ediniz.)
5402  -----*/
5403
5404 /* sample.c */
5405
5406 #include <stdio.h>
5407 #include <stdlib.h>
5408 #include <unistd.h>
5409
5410 void exit_sys(const char *msg);
5411
```

```
5412 int main(void)
5413 {
5414     uid_t ruid, euid;
5415     gid_t rgid, egid;
5416
5417     ruid = getuid();
5418     euid = geteuid();
5419
5420     rgid = getgid();
5421     egid = getegid();
5422
5423     printf("Before exec:\n");
5424
5425     printf("Real User Id: %lu\n", (unsigned long)ruid);
5426     printf("Effective User Id: %lu\n", (unsigned long)euid);
5427     printf("Real Group Id: %lu\n", (unsigned long)rgid);
5428     printf("Effective Group Id: %lu\n\n", (unsigned long)egid);
5429
5430     if (execl("mample", "mample", (char *)NULL) == -1)
5431         exit_sys("execl");
5432
5433     /* unreachable code */
5434
5435     return 0;
5436 }
5437
5438 void exit_sys(const char *msg)
5439 {
5440     perror(msg);
5441
5442     exit(EXIT_FAILURE);
5443 }
5444
5445 /* mample.c */
5446
5447 #include <stdio.h>
5448 #include <unistd.h>
5449
5450 int main(void)
5451 {
5452     uid_t ruid, euid;
5453     gid_t rgid, egid;
5454
5455     ruid = getuid();
5456     euid = geteuid();
5457
5458     rgid = getgid();
5459     egid = getegid();
5460
5461     printf("Real User Id: %lu\n", (unsigned long)ruid);
5462     printf("Effective User Id: %lu\n", (unsigned long)euid);
5463     printf("Real Group Id: %lu\n", (unsigned long)rgid);
5464     printf("Effective Group Id: %lu\n", (unsigned long)egid);
```

```
5465     return 0;
5466 }
5467 }
5468 /
5469 *
-----*
5470     getresuid ve getresgid isimli fonksiyonlarla biz gerçek, etkin ve saklı id'leri alabiliriz. Ancak bu fonksiyon POSIX standartlarında yoktur. Linux ve BSD sistemlerinde bulunmaktadır. Bu fonksiyonları kullanmadan önce _GNU_SOURCE nitelik makrosunu dosyanın tepesine include ediniz. Ya da derleme işleminde -D_GNU_SOURCE komut satırı argümanını bulundurunuz
5471 -----*/
5472
5473 #define _GNU_SOURCE
5474
5475 #include <stdio.h>
5476 #include <stdlib.h>
5477 #include <unistd.h>
5478
5479 void exit_sys(const char *msg);
5480
5481 int main(void)
5482 {
5483     uid_t ruid, euid, ssuid;
5484     gid_t rgid, egid, ssgid;
5485
5486     if (getresuid(&ruid, &euid, &ssuid) == -1)
5487         exit_sys("getresuid");
5488
5489     if (getresgid(&rgid, &egid, &ssgid) == -1)
5490         exit_sys("getresgid");
5491
5492     printf("Real User Id = %lu, Effective User Id = %lu, Saved Set User Id = %lu\n",
5493           (unsigned long)ruid, (unsigned long)euid, (unsigned long)ssuid);
5494
5495     printf("Real Group Id = %lu, Effective Group Id = %lu, Saved Set Group Id = %lu\n",
5496           (unsigned long)rgid, (unsigned long)egid, (unsigned long)ssgid);
5497
5498     return 0;
5499 }
5500
5501 void exit_sys(const char *msg)
5502 {
5503     perror(msg);
5504
5505     exit(EXIT_FAILURE);
5506 }
5507
5508 }
```

```
5510  /
5511  *-----*
5512  Bir prosesin ek grup id'leri (supplementary group ids) getgroups isimli POSIX fonksiyonuyla elde edilebilir.
5513  -----
5514 #include <stdio.h>
5515 #include <stdlib.h>
5516 #include <unistd.h>
5517 #include <limits.h>
5518 #include <grp.h>
5519
5520 void exit_sys(const char *msg);
5521
5522 int main(void)
5523 {
5524     gid_t groups[NGROUPS_MAX + 1];
5525     int result, i;
5526     struct group *gr;
5527
5528     if ((result = getgroups(NGROUPS_MAX + 1, groups)) == -1)
5529         exit_sys("getgroups");
5530
5531     for (i = 0; i < result; ++i) {
5532         if ((gr = getgrgid(groups[i])) == NULL)
5533             exit_sys("getgrgid");
5534         printf("%s (%lu) ", gr->gr_name, (unsigned long)groups[i]);
5535     }
5536     printf("\n");
5537
5538     return 0;
5539 }
5540
5541 void exit_sys(const char *msg)
5542 {
5543     perror(msg);
5544
5545     exit(EXIT_FAILURE);
5546 }
5547
5548 /
5549 *-----*
5550
5551 setgroups isimli fonksiyonuyla prosesimizin ek group id'lerini set edebiliriz. Ancak setgroups fonksiyonu bir POSIX
5552 fonksiyonu değildir. Fakat yaygın Unix türevi sistemlerde (örneğin Linux)
5553 sistemlerinde) bu fonksiyon bulunmaktadır. Eğer proses root değilse ya da uygun yeterlilik sahip değilse fonksiyon başarısız olmaktadır. (Yani programı sudo ile çalıştırılmalısınız)
```

```
-----*/
5554
5555 #include <stdio.h>
5556 #include <stdlib.h>
5557 #include <unistd.h>
5558 #include <limits.h>
5559 #include <grp.h>
5560
5561 void exit_sys(const char *msg);
5562
5563 int main(void)
5564 {
5565     gid_t groups[NGROUPS_MAX + 1] = {115, 123, 127};
5566     int result, i;
5567     struct group *gr;
5568
5569     if ((result = setgroups(3, groups)) == -1)
5570         exit_sys("setgroups");
5571
5572     if ((result = getgroups(NGROUPS_MAX + 1, groups)) == -1)
5573         exit_sys("getgroups");
5574
5575     for (i = 0; i < result; ++i) {
5576         if ((gr = getgrgid(groups[i])) == NULL)
5577             exit_sys("getgrgid");
5578         printf("%s (%lu) ", gr->gr_name, (unsigned long)groups[i]);
5579     }
5580     printf("\n");
5581
5582     return 0;
5583 }
5584
5585 void exit_sys(const char *msg)
5586 {
5587     perror(msg);
5588
5589     exit(EXIT_FAILURE);
5590 }
5591
5592 /
*-----*
-----*
5593 Bu fonksiyon tipik olarak login programı tarafından kullanılır. →
      Fonksiyon parametresiyle aldığı kullanıcı ismine ilişkin →
5594 ek grupları /etc/passwd ve /etc/group dosyalarından elde eder ve →
      setgroups fonksiyonunu çağırarak proses için set eder. →
5595 initgroups bir POSIX fonksiyonu değildir. Linux ve BSD gibi sistemlerde →
      bulunmaktadır. Fonksiyon etkin kullanıcı id'si →
5596 root olmayan ve yeteneğe sahip olmayan prosesler tarafından →
      çağrılırsa başarısız olmaktadır. (Yani aşağıdaki programı →
5597 sudo ile çalıştırılmalısınız.) →
5598
-----*/
```

```
5599
5600 #include <stdio.h>
5601 #include <stdlib.h>
5602 #include <unistd.h>
5603 #include <limits.h>
5604 #include <grp.h>
5605
5606 void exit_sys(const char *msg);
5607
5608 int main(void)
5609 {
5610     gid_t groups[NGROUPS_MAX + 1];
5611     int result, i;
5612     struct group *gr;
5613
5614     if (initgroups("csd", getegid()) == -1)
5615         exit_sys("initgroups");
5616
5617     if ((result = getgroups(NGROUPS_MAX + 1, groups)) == -1)
5618         exit_sys("getgroups");
5619
5620     for (i = 0; i < result; ++i) {
5621         if ((gr = getgrgid(groups[i])) == NULL)
5622             exit_sys("getgrgid");
5623         printf("%s (%lu) ", gr->gr_name, (unsigned long)groups[i]);
5624     }
5625     printf("\n");
5626
5627     return 0;
5628 }
5629
5630 void exit_sys(const char *msg)
5631 {
5632     perror(msg);
5633
5634     exit(EXIT_FAILURE);
5635 }
5636
5637 /
*-----*
-----*
5638 İsimsiz boru (unnamed pipe) örneği. Bu örnekte proses önce boruyu sonra da alt prosesi yaratır. Üst proses boruya yazma yapar alt proses de borudan okuma yapmaktadır.
5639
5640 -----*/
5641
5642 #include <stdio.h>
5643 #include <stdlib.h>
5644 #include <unistd.h>
5645 #include <sys/wait.h>
5646
5647 void exit_sys(const char *msg);
```

```
5648
5649 int main(void)
5650 {
5651     int pfds[2];
5652     pid_t pid;
5653     ssize_t result;
5654     int i, val;
5655
5656     if (pipe(pfds) == -1)
5657         exit_sys("pipe");
5658
5659     if ((pid = fork()) == -1)
5660         exit_sys("fork");
5661
5662     if (pid != 0) { /* parent writes */
5663         close(pfds[0]);
5664         for (i = 0; i < 1000000; ++i)
5665             if (write(pfds[1], &i, sizeof(int)) == -1)
5666                 exit_sys("write");
5667         close(pfds[1]);
5668
5669         if (waitpid(pid, NULL, 0) == -1)
5670             exit_sys("waitpid");
5671     }
5672     else { /* child reads */
5673         close(pfds[1]);
5674         while ((result = read(pfds[0], &val, sizeof(int))) > 0) {
5675             printf("%d ", val);
5676             fflush(stdout);
5677         }
5678         printf("\n");
5679         if (result == -1)
5680             exit_sys("read");
5681         close(pfds[0]);
5682     }
5683
5684     return 0;
5685 }
5686
5687 void exit_sys(const char *msg)
5688 {
5689     perror(msg);
5690
5691     exit(EXIT_FAILURE);
5692 }
5693
5694 /
*-----*
-----*
5695 Kabuk üzerinde pipe işlemi nasıl yapılmaktadır? a | b biçimindeki bir ↗
      komutta kabuk önce isimsiz boruyu yaratır. Sonra a ↗
5696 programı için fork yapar. Henüz exec yapmadan alt prosesin 1 numaralı ↗
      betimleyicisini yazma amaçlı boruya yönlendirir. ↗
```

```
5697     Sonra exec yapar. Daha sonra yine b için fork yapar. henüz exec yapmadan →  
      alt proseste 0 numaralı betimleyiciyi okuma  
5698     amaçlı boruya yönlendirir. Sonra exec yapar. Bu işlemler sırasında →  
      gereksiz betimleyicilerin hepsi kapatılmaktadır. Aşağıdaki  
5699     örnekte kabuğun yaptığı gibi bir boru işlemi gerçekleştirilmektedir.  
5700 -----*/  
5701  
5702 #include <stdio.h>  
5703 #include <stdlib.h>  
5704 #include <string.h>  
5705 #include <unistd.h>  
5706 #include <sys/wait.h>  
5707  
5708 void exit_sys(const char *msg);  
5709  
5710 int main(int argc, char *argv[]) /* ./sample prog args ... "|" prog →  
      args ... */  
5711 {  
5712     int pfds[2];  
5713     pid_t pid1, pid2;  
5714     int i, pindex;  
5715  
5716     for (pindex = 0; pindex < argc; ++pindex)  
5717         if (!strcmp(argv[pindex], "|"))  
5718             break;  
5719     if (pindex == 0 || pindex == argc || pindex == argc - 1) {  
5720         fprintf(stderr, "invalid argument!\n");  
5721         exit(EXIT_FAILURE);  
5722     }  
5723     argv[pindex] = NULL;  
5724  
5725     if (pipe(pfds) == -1)  
5726         exit_sys("pipe");  
5727  
5728     if ((pid1 = fork()) == -1)  
5729         exit_sys("fork");  
5730     if (pid1 == 0) {  
5731         if (dup2(pfds[1], 1) == -1)  
5732             exit_sys("dup2");  
5733         close(pfds[0]);  
5734         close(pfds[1]);  
5735         if (execvp(argv[1], &argv[1]) == -1)  
5736             exit_sys("execv");  
5737  
5738         /* unreachable code */  
5739     }  
5740  
5741     if ((pid2 = fork()) == -1)  
5742         exit_sys("fork");  
5743     if (pid2 == 0) {  
5744         if (dup2(pfds[0], 0) == -1)  
5745             exit_sys("dup2");
```

```
5746         close(pfds[0]);
5747         close(pfds[1]);
5748         if (execvp(argv[pindex + 1], &argv[pindex + 1]) == -1)
5749             exit_sys("execv");
5750
5751     /* unreachable code */
5752 }
5753 close(pfds[0]);
5754 close(pfds[1]);
5755
5756 if (waitpid(pid1, NULL, 0) == -1)
5757     exit_sys("waitpid");
5758 if (waitpid(pid2, NULL, 0) == -1)
5759     exit_sys("waitpid");
5760
5761 return 0;
5762 }
5763
5764 void exit_sys(const char *msg)
5765 {
5766     perror(msg);
5767
5768     exit(EXIT_FAILURE);
5769 }
5770
5771 /
*-----*
-----*
5772 mkfifo komutuna benzer bir program. Programın -m, --mode ve -h, --help
      komut satırı argümanları vardır. Program belirtilen
      isimdeki isimli boruyu yaratır.
5773 -----
-----*/
5775
5776 #include <stdio.h>
5777 #include <stdlib.h>
5778 #include <unistd.h>
5779 #include <getopt.h>
5780 #include <sys/stat.h>
5781
5782 void exit_sys(const char *msg);
5783 int is_octal(const char *str);
5784
5785 int main(int argc, char *argv[])
5786 {
5787     int result, help_flag = 0, err_flag = 0;
5788     int mode_flag = 0;
5789     int i;
5790     mode_t mode, result_mode;
5791     mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP,
5792                       S_IROTH, S_IWOTH, S_IXOTH};
5793     char *mode_arg;
5794     struct option options[] = {
```

```
5794         {"mode", required_argument, NULL, 'm'},
5795         {"help", no_argument, NULL, 'h'},
5796         {0, 0, 0, 0}
5797     };
5798
5799     opterr = 0;
5800     while ((result = getopt_long(argc, argv, "m:h", options, NULL)) != -1) {
5801         switch (result) {
5802             case 'm':
5803                 mode_flag = 1;
5804                 mode_arg = optarg;
5805                 break;
5806             case 'h':
5807                 help_flag = 1;
5808                 break;
5809             case '?':
5810                 if (optopt != 0)
5811                     fprintf(stderr, "invalid switch: -%c\n", optopt);
5812                 else
5813                     fprintf(stderr, "invalid switch: %s\n", argv[optind - 1]); /* argv[optind - 1] dokümanı edilmemiş */
5814                 err_flag = 1;
5815             }
5816         }
5817         if (err_flag)
5818             exit(EXIT_FAILURE);
5819
5820         if (help_flag) {
5821             fprintf(stdout, "mymkfifo [-m |--mode <mode>] <file list>\n");
5822             exit(EXIT_SUCCESS);
5823         }
5824
5825         if (mode_flag) {
5826             if (!is_octal(mode_arg) || (mode = (mode_t)strtoul(mode_arg, NULL, 8)) > 0x777) {
5827                 fprintf(stderr, "invalid octal digits!..\n");
5828                 exit(EXIT_FAILURE);
5829             }
5830
5831             result_mode = 0;
5832             for (i = 8; i >= 0; --i) {
5833                 if (mode >> i & 1)
5834                     result_mode |= modes[8 - i];
5835             }
5836         }
5837         else
5838             result_mode = S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH;
5839
5840         umask(0);
5841         for (i = optind; i < argc; ++i)
5842             if (mkfifo(argv[i], result_mode) == -1) {
5843                 perror("mkfifo");
5844                 continue;
```

```
5845         }
5846
5847     return 0;
5848 }
5849
5850 int is_octal(const char *str)
5851 {
5852     int i;
5853
5854     for (i = 0; str[i] != '\0'; ++i)
5855         if (str[i] < '0' || str[i] > '7')
5856             return 0;
5857
5858     return 1;
5859 }
5860
5861 void exit_sys(const char *msg)
5862 {
5863     perror(msg);
5864
5865     exit(EXIT_FAILURE);
5866 }
5867
5868 /
*-----*
-----*
5869 Blokeli modda isimli borular open fonksiyonuyla O_RDONLY modunda      ↗
5870 açıldığında başka bir proses boruyu O_WRONLY ya da O_RDWR      ↗
5871 modunda açana kadar open blokede kalır. Benzer biçimde isimli borular      ↗
5872 open fonksiyonuyla O_WRONLY modunda açıldığında başka      ↗
5873 bir proses boruyu O_RDONLY ya da O_RDWR modunda açana kadar open blokede      ↗
5874 kalır. Ancak isimli boru O_RDWR modunda açılma      ↗
5875 çalışırsa bloke oluşmaz. Aşağıda isimli boru örneği verilmiştir.      ↗
5876 -----*/
5877
5878 /* named-pipe-proc1.c */
5879
5880 #include <stdio.h>
5881 #include <stdlib.h>
5882 #include <fcntl.h>
5883 #include <unistd.h>
5884
5885 void exit_sys(const char *msg);
5886
5887 int main(int argc, char *argv[])
5888 {
5889     int pipefd;
5890     int i;
5891
5892     if (argc != 2) {
5893         fprintf(stderr, "wrong number of arguments!..\n");
5894         exit(EXIT_FAILURE);
5895 }
```

```
5892     }
5893
5894     if ((pipefd = open(argv[1], O_WRONLY)) == -1)
5895         exit_sys("open");
5896
5897     for (i = 0; i < 1000000; ++i)
5898         if (write(pipefd, &i, sizeof(int)) == -1)
5899             exit_sys("write");
5900
5901     close(pipefd);
5902
5903     return 0;
5904 }
5905
5906 void exit_sys(const char *msg)
5907 {
5908     perror(msg);
5909
5910     exit(EXIT_FAILURE);
5911 }
5912
5913 /* named-pipe-proc2.c */
5914
5915 #include <stdio.h>
5916 #include <stdlib.h>
5917 #include <fcntl.h>
5918 #include <unistd.h>
5919
5920 void exit_sys(const char *msg);
5921
5922 int main(int argc, char *argv[])
5923 {
5924     int pipefd;
5925     int val;
5926     int result;
5927
5928     if (argc != 2) {
5929         fprintf(stderr, "wrong number of arguments!..\n");
5930         exit(EXIT_FAILURE);
5931     }
5932
5933     if ((pipefd = open(argv[1], O_RDONLY)) == -1)
5934         exit_sys("open");
5935
5936     while ((result = read(pipefd, &val, sizeof(int))) > 0)
5937         printf("%d ", val), fflush(stdout);
5938     printf("\n");
5939
5940     if (result == -1)
5941         exit_sys("read");
5942
5943     close(pipefd);
5944
```

```
5945     return 0;
5946 }
5947
5948 void exit_sys(const char *msg)
5949 {
5950     perror(msg);
5951
5952     exit(EXIT_FAILURE);
5953 }
5954
5955 /
*-----*
-----*
5956 Boru dosyasını O_NONBLOCK bayrağı ile blokesiz modda açtığımızda read    ↵
      fonksiyonu hiç bloke olmaz. read ile n byte okunmak
5957 istendüğünde eğer boruda az sayıda bilgi varsa read n byte okunana kadar ↵
      beklemez. Boruda olanı okur ve okuduğu byte sayısı
5958 ile geri döner. Eğer boruda 0 byte varsa read 0 ile geri dönmez -1 ile ↵
      geri döner yani başarısız olur. Ancak errno değeri
5959 EAGAIN biçiminde özel bir değere set edilir. Programcı da boruda hiçbir ↵
      şey yoksa arka plan işlemleri yapabilir. Benzer
5960 biçimde blokesiz modda write fonksiyonu da n byte'ın hepsi yazılına    ↵
      kadar blokede beklemez. Yazabildiği byte sayısını yazar
5961 yazabildiği byte sayısına geri döner. Boru tamamen doluya write    ↵
      başarısız olur -1 değerine geri döner ve errno EAGAIN
5962 değeri ile set edilir. Blokesiz modda open fonksiyonu da asla bloke    ↵
      olmaz. O_RDONLY ya da O_RDWR modunda boru açılmaya çalışılırsa
5963 open başarılı olur. open O_WRONLY modunda boru açılmak istenirse başka    ↵
      bir proses "read" modda boruyu açmamışsa başarız olur.
5964 -----*/
5965
5966 #include <stdio.h>
5967 #include <stdlib.h>
5968 #include <fcntl.h>
5969 #include <unistd.h>
5970 #include <errno.h>
5971
5972 void exit_sys(const char *msg);
5973
5974 int main(int argc, char *argv[])
5975 {
5976     int pipefd;
5977     int result;
5978     int i;
5979     char buf[11];
5980
5981     if (argc != 2) {
5982         fprintf(stderr, "wrong number of arguments!..\n");
5983         exit(EXIT_FAILURE);
5984     }
5985
5986     if ((pipefd = open(argv[1], O_RDONLY|O_NONBLOCK)) == -1)
```

```
5987         exit_sys("open");
5988
5989         sleep(1);
5990
5991     for (i = 0; i < 100; ++i) {
5992         if ((result = read(pipefd, buf, 10)) == -1) {
5993             if (errno == EAGAIN) {
5994                 printf("%d ", i), fflush(stdout);
5995                 sleep(1);
5996                 continue;
5997             }
5998
5999         }
6000         buf[result] = '\0';
6001         puts(buf);
6002         sleep(1);
6003     }
6004
6005
6006     printf("%d\n", result);
6007
6008
6009     close(pipefd);
6010
6011     return 0;
6012 }
6013
6014 void exit_sys(const char *msg)
6015 {
6016     perror(msg);
6017
6018     exit(EXIT_FAILURE);
6019 }
6020
6021 /
*-----*
-----*
6022     Nonblocking pipe örneği (önce nonblocking-pipe1.c programını      ↗
6023     çalıştırınız). Nonblocking işlemlerde mesgul döngü (busy loop)      ↗
6024     önemli bir problemdir. Bu problemi ortadan kaldırmak için select, poll      ↗
6025     gibi fonksiyonlar ve asenkron io yöntemleri kullanılır.      ↗
6026 -----*/*/
6027
6028 /* nonblocking-pipe1.c */
6029
6030 #include <stdio.h>
6031 #include <stdlib.h>
6032 #include <fcntl.h>
6033 #include <unistd.h>
6034 #include <errno.h>
6035
6036 void exit_sys(const char *msg);
```

```
6035
6036 int main(int argc, char *argv[])
6037 {
6038     int pipefd;
6039     int result;
6040     int val;
6041
6042     if (argc != 2) {
6043         fprintf(stderr, "wrong number of arguments!..\n");
6044         exit(EXIT_FAILURE);
6045     }
6046
6047     if ((pipefd = open(argv[1], O_RDONLY|O_NONBLOCK)) == -1)
6048         exit_sys("open");
6049     sleep(10);
6050     for (;;) {
6051         if ((result = read(pipefd, &val, sizeof(int))) == -1)
6052             if (errno == EAGAIN)
6053                 continue;
6054             else
6055                 exit_sys("read");
6056         if (result == 0)
6057             break;
6058         printf("%d ", val), fflush(stdout);
6059     }
6060     printf("\n");
6061
6062     close(pipefd);
6063
6064     return 0;
6065 }
6066
6067 void exit_sys(const char *msg)
6068 {
6069     perror(msg);
6070
6071     exit(EXIT_FAILURE);
6072 }
6073
6074 /* nonblocking-pipe2.c */
6075
6076 #include <stdio.h>
6077 #include <stdlib.h>
6078 #include <fcntl.h>
6079 #include <unistd.h>
6080 #include <errno.h>
6081
6082 void exit_sys(const char *msg);
6083
6084 int main(int argc, char *argv[])
6085 {
6086     int pipefd;
6087     int result;
```

```
6088     int val;
6089
6090     if (argc != 2) {
6091         fprintf(stderr, "wrong number of arguments!..\\n");
6092         exit(EXIT_FAILURE);
6093     }
6094
6095     if ((pipefd = open(argv[1], O_RDONLY|O_NONBLOCK)) == -1)
6096         exit_sys("open");
6097     sleep(10);
6098     for (;;) {
6099         if ((result = read(pipefd, &val, sizeof(int))) == -1)
6100             if (errno == EAGAIN)
6101                 continue;
6102             else
6103                 exit_sys("read");
6104         if (result == 0)
6105             break;
6106         printf("%d ", val), fflush(stdout);
6107     }
6108
6109     close(pipefd);
6110
6111     return 0;
6112 }
6113
6114 void exit_sys(const char *msg)
6115 {
6116     perror(msg);
6117
6118     exit(EXIT_FAILURE);
6119 }
6120
6121 /
*-----*
-----*
6122 Bir kabuk komutunu (dolayısıyla bir programı) çalıştırıp onun stdout      ↵
       dosyasına yazdıklarını ya da stdin dosyasından okuduklarını
6123 boruya yönlendirebiliriz. Bunun için popen ve pclose POSIX fonksiyonları ↵
       kullanılmaktadır.
6124 -----*/
```

6125

6126 #include <stdio.h>

6127 #include <stdlib.h>

6128

6129 void exit_sys(const char *msg);

6130

6131 int main(void)

6132 {

6133 FILE *f;

6134 int ch;

6135

```
6136     if ((f = popen("gcc -o mample mample.c", "r")) == NULL)
6137         exit_sys("popen");
6138
6139     while ((ch = fgetc(f)) != EOF)
6140         putchar(ch);
6141
6142     pclose(f);
6143
6144     return 0;
6145 }
6146
6147 void exit_sys(const char *msg)
6148 {
6149     perror(msg);
6150
6151     exit(EXIT_FAILURE);
6152 }
6153
6154 /
*-----*
-----*
6155     popen ve pclose fonksiyonlarının örnek bir gerçekleştirimi
6156 -----*/
6157
6158 #include <stdio.h>
6159 #include <stdlib.h>
6160 #include <fcntl.h>
6161 #include <unistd.h>
6162 #include <sys/wait.h>
6163
6164 void exit_sys(const char *msg);
6165
6166 static pid_t g_pid;
6167
6168 FILE *csd_popen(const char *command, const char *mode)
6169 {
6170     pid_t pid;
6171     int pfds[2];
6172     FILE *f;
6173
6174     if (mode[1] != '\0' || (mode[0] != 'r' && mode[0] != 'w'))
6175         return NULL;
6176
6177     if (pipe(pfds) == -1)
6178         return NULL;
6179
6180     if ((pid = fork()) == -1)
6181         return NULL;
6182
6183     if (pid == 0) {
6184         if (mode[0] == 'r') {
6185             if (dup2(pfds[1], 1) == -1)
```

```
6186             _exit(EXIT_FAILURE);
6187             close(pfds[0]);
6188             close(pfds[1]);
6189         }
6190     else {
6191         if (dup2(pfds[0], 0) == -1)
6192             _exit(EXIT_FAILURE);
6193         close(pfds[0]);
6194         close(pfds[1]);
6195     }
6196     if (execl("/bin/bash", "/bin/bash", "-c", command, (char *) NULL) == -1)
6197         _exit(EXIT_FAILURE);
6198 }
6199 g_pid = pid;
6200 if (mode[0] == 'r') {
6201     close(pfds[1]);
6202     f = fdopen(pfds[0], "r");
6203 }
6204 else {
6205     close(pfds[0]);
6206     f = fdopen(pfds[1], "w");
6207 }
6208
6209 return f;
6210 }
6211
6212 int csd_pclose(FILE *f)
6213 {
6214     int status;
6215
6216     if (waitpid(g_pid, &status, 0) == -1)
6217         return -1;
6218
6219     fclose(f);
6220
6221     return status;
6222 }
6223
6224 int main(void)
6225 {
6226     FILE *f;
6227     int ch;
6228
6229     if ((f = csd_popen("ls -l", "r")) == NULL) {
6230         fprintf(stderr, "csd_open failed\n");
6231         exit(EXIT_FAILURE);
6232     }
6233     while ((ch = fgetc(f)) != EOF)
6234         putchar(ch);
6235
6236     csd_pclose(f);
6237 }
```

```
6238     return 0;
6239 }
6240
6241 void exit_sys(const char *msg)
6242 {
6243     perror(msg);
6244
6245     exit(EXIT_FAILURE);
6246 }
6247
6248 /
*-----*
-----*
6249     Boru yoluyla client-server haberleşme örneği
6250 -----*/
6251 /* client.c */
6252
6253
6254 #include <stdio.h>
6255 #include <stdlib.h>
6256 #include <string.h>
6257 #include <ctype.h>
6258 #include <fcntl.h>
6259 #include <unistd.h>
6260 #include <signal.h>
6261
6262 /* Symbolic Constants */
6263
6264 #define SERVER_PIPE      "serverpipe"
6265 #define MAX_CMD_LEN       1024
6266 #define MAX_MSG_LEN        32768
6267 #define MAX_PIPE_PATH      1024
6268
6269 /* Type Declaration */
6270
6271 typedef struct tagCLIENT_MSG {
6272     int msglen;
6273     int client_id;
6274     char msg[MAX_MSG_LEN];
6275 } CLIENT_MSG;
6276
6277 typedef struct tagSERVER_MSG {
6278     int msglen;
6279     char msg[MAX_MSG_LEN];
6280 } SERVER_MSG;
6281
6282 typedef struct tagMSG_CONTENTS {
6283     char *msg_cmd;
6284     char *msg_param;
6285 } MSG_CONTENTS;
6286
6287 typedef struct tagMSG_PROC {
```

```
6288     const char *msg_cmd;
6289     int (*proc)(const char *msg_param);
6290 } MSG_PROC;
6291
6292 /* Function Prototypes */
6293
6294 void sigpipe_handler(int sno);
6295 int putmsg(const char *cmd);
6296 int get_server_msg(int fdp, SERVER_MSG *smsg);
6297 void parse_msg(char *msg, MSG_CONTENTS *msgc);
6298 void check_quit(char *cmd);
6299 int connect_to_server(void);
6300 int cmd_response_proc(const char *msg_param);
6301 int disconnect_accepted_proc(const char *msg_param);
6302 int invalid_command_proc(const char *msg_param);
6303 void clear_stdin(void);
6304 void exit_sys(const char *msg);
6305
6306 /* Global Data Definitions */
6307
6308 MSG_PROC g_msg_proc[] = {
6309     {"CMD_RESPONSE", cmd_response_proc},
6310     {"DISCONNECT_ACCEPTED", disconnect_accepted_proc},
6311     {"INVALID_COMMAND", invalid_command_proc},
6312     {NULL, NULL}
6313 };
6314
6315 int g_client_id;
6316 int g_fdps, g_fdpc;
6317
6318 /* Function Definitions */
6319
6320 int main(void)
6321 {
6322     char cmd[MAX_CMD_LEN];
6323     char *str;
6324     SERVER_MSG smsg;
6325     MSG_CONTENTS msgc;
6326     int i;
6327
6328     if (signal(SIGPIPE, sigpipe_handler) == SIG_ERR)
6329         exit_sys("signal");
6330
6331     if ((g_fdps = open(SERVER_PIPE, O_WRONLY)) == -1)
6332         exit_sys("open");
6333
6334     if (connect_to_server() == -1) {
6335         fprintf(stderr, "cannot connect to server! Try again...\n");
6336         exit(EXIT_FAILURE);
6337     }
6338
6339     for (;;) {
6340         printf("Client>");
```

```
6341         fflush(stdout);
6342         fgets(cmd, MAX_CMD_LEN, stdin);
6343         if ((str = strchr(cmd, '\n')) != NULL)
6344             *str = '\0';
6345
6346         check_quit(cmd);
6347
6348         if (putmsg(cmd) == -1)
6349             exit_sys("putmsg");
6350
6351         if (get_server_msg(g_fdpc, &smsg) == -1)
6352             exit_sys("get_client_msg");
6353
6354         parse_msg(smsg.msg, &msgc);
6355
6356         for (i = 0; g_msg_proc[i].msg_cmd != NULL; ++i)
6357             if (!strcmp(msgc.msg_cmd, g_msg_proc[i].msg_cmd)) {
6358                 if (g_msg_proc[i].proc(msgc.msg_param) == -1) {
6359                     fprintf(stderr, "command failed!\n");
6360                     exit(EXIT_FAILURE);
6361                 }
6362                 break;
6363             }
6364         if (g_msg_proc[i].msg_cmd == NULL) { /* command not found */
6365             fprintf(stderr, "Fatal Error: Unknown server message!\n");
6366             exit(EXIT_FAILURE);
6367         }
6368     }
6369
6370     return 0;
6371 }
6372
6373 void sigpipe_handler(int sno)
6374 {
6375     printf("server down, exiting...\n");
6376
6377     exit(EXIT_FAILURE);
6378 }
6379
6380 int putmsg(const char *cmd)
6381 {
6382     CLIENT_MSG cmsg;
6383     int i, k;
6384
6385     for (i = 0; isspace(cmd[i]); ++i)
6386         ;
6387     for (k = 0; !isspace(cmd[i]); ++i)
6388         cmsg.msg[k++] = cmd[i];
6389     cmsg.msg[k++] = ' ';
6390     for (; isspace(cmd[i]); ++i)
6391         ;
6392     for (; (cmsg.msg[k++] = cmd[i]) != '\0'; ++i)
6393         ;
```

```
6394     cmsg.msglen = (int)strlen(cmsg.msg);
6395     cmsg.client_id = g_client_id;
6396
6397     if (write(g_fdps, &cmsg, 2 * sizeof(int) + cmsg.msglen) == -1)
6398         return -1;
6399
6400     return 0;
6401 }
6402
6403 int get_server_msg(int fdp, SERVER_MSG *smsg)
6404 {
6405     if (read(fdp, &smsg->msglen, sizeof(int)) == -1)
6406         return -1;
6407
6408     if (read(fdp, smsg->msg, smsg->msglen) == -1)
6409         return -1;
6410
6411     smsg->msg[smsg->msglen] = '\0';
6412
6413     return 0;
6414 }
6415
6416 void parse_msg(char *msg, MSG_CONTENTS *msgc)
6417 {
6418     int i;
6419
6420     msgc->msg_cmd = msg;
6421
6422     for (i = 0; msg[i] != ' ' && msg[i] != '\0'; ++i)
6423         ;
6424     msg[i++] = '\0';
6425     msgc->msg_param = &msg[i];
6426 }
6427
6428 void check_quit(char *cmd)
6429 {
6430     int i, pos;
6431
6432     for (i = 0; isspace(cmd[i]); ++i)
6433         ;
6434     pos = i;
6435     for (; !isspace(cmd[i]) && cmd[i] != '\0'; ++i)
6436         ;
6437     if (!strncmp(&cmd[pos], "quit", pos - i))
6438         strcpy(cmd, "DISCONNECT_REQUEST");
6439 }
6440
6441 int connect_to_server(void)
6442 {
6443     char name[MAX_PIPE_PATH];
6444     char cmd[MAX_CMD_LEN];
6445     char *str;
6446     SERVER_MSG smsg;
```

```
6447     MSG_CONTENTS msgc;
6448     int response;
6449
6450     printf("Pipe name:");
6451     fgets(name, MAX_PIPE_PATH, stdin);
6452     if ((str = strchr(name, '\n')) != NULL)
6453         *str = '\0';
6454
6455     if (access(name, F_OK) == 0) {
6456         do {
6457             printf("Pipe already exists! Overwrite? (Y/N)");
6458             fflush(stdout);
6459             response = tolower(getchar());
6460             clear_stdin();
6461             if (response == 'y' && remove(name) == -1)
6462                 return -1;
6463         } while (response != 'y' && response != 'n');
6464         if (response == 'n')
6465             return -1;
6466     }
6467
6468     sprintf(cmd, "CONNECT %s", name);
6469
6470     if (putmsg(cmd) == -1)
6471         return -1;
6472
6473     while (access(name, F_OK) != 0)
6474         usleep(300);
6475
6476     if ((g_fdpc = open(name, O_RDONLY)) == -1)
6477         return -1;
6478
6479     if (get_server_msg(g_fdpc, &smsg) == -1)
6480         exit_sys("get_client_msg");
6481
6482     parse_msg(smsg.msg, &msgc);
6483
6484     if (strcmp(msgc.msg_cmd, "CONNECTED"))
6485         return -1;
6486
6487     g_client_id = (int)strtol(msgc.msg_param, NULL, 10);
6488
6489     printf("Connected server with '%d' id...\n", g_client_id);
6490
6491     return 0;
6492 }
6493
6494 int cmd_response_proc(const char *msg_param)
6495 {
6496     printf("%s\n", msg_param);
6497
6498     return 0;
6499 }
```

```
6500
6501 int disconnect_accepted_proc(const char *msg_param)
6502 {
6503     if (putmsg("DISCONNECT") == -1)
6504         exit_sys("putmsg");
6505
6506     exit(EXIT_SUCCESS);
6507
6508     return 0;
6509 }
6510
6511 int invalid_command_proc(const char *msg_param)
6512 {
6513     printf("invalid command: %s\n", msg_param);
6514
6515     return 0;
6516 }
6517
6518 void clear_stdin(void)
6519 {
6520     int ch;
6521
6522     while ((ch = getchar()) != '\n' && ch != EOF)
6523         ;
6524 }
6525
6526 void exit_sys(const char *msg)
6527 {
6528     perror(msg);
6529
6530     exit(EXIT_FAILURE);
6531 }
6532
6533 /* server.c */
6534
6535 #include <stdio.h>
6536 #include <stdlib.h>
6537 #include <string.h>
6538 #include <fcntl.h>
6539 #include <unistd.h>
6540 #include <errno.h>
6541 #include <sys/stat.h>
6542
6543 /* Symbolic Constants */
6544
6545 #define SERVER_PIPE          "serverpipe"
6546 #define MAX_MSG_LEN           32768
6547 #define MAX_PIPE_PATH          1024
6548 #define MAX_CLIENT             1024
6549
6550 /* Type Declaration */
6551
6552 typedef struct tagCLIENT_MSG {
```

```
6553     int msglen;
6554     int client_id;
6555     char msg[MAX_MSG_LEN];
6556 } CLIENT_MSG;
6557
6558 typedef struct tagSERVER_MSG {
6559     int msglen;
6560     char msg[MAX_MSG_LEN];
6561 } SERVER_MSG;
6562
6563 typedef struct tagMSG_CONTENTS {
6564     char *msg_cmd;
6565     char *msg_param;
6566 } MSG_CONTENTS;
6567
6568 typedef struct tagMSG_PROC {
6569     const char *msg_cmd;
6570     int (*proc)(int, const char *msg_param);
6571 } MSG_PROC;
6572
6573 typedef struct tagCLIENT_INFO {
6574     int fdp;
6575     char path[MAX_PIPE_PATH];
6576 } CLIENT_INFO;
6577
6578 /* Function Prototypes */
6579
6580 int get_client_msg(int fdp, CLIENT_MSG *cmsg);
6581 int putmsg(int client_id, const char *cmd);
6582 void parse_msg(char *msg, MSG_CONTENTS *msgc);
6583 void print_msg(const CLIENT_MSG *cmsg);
6584 int invalid_command(int client_id, const char *cmd);
6585 int connect_proc(int client_id, const char *msg_param);
6586 int disconnect_request_proc(int client_id, const char *msg_param);
6587 int disconnect_proc(int client_id, const char *msg_param);
6588 int cmd_proc(int client_id, const char *msg_param);
6589 void exit_sys(const char *msg);
6590
6591 /* Global Data Definitions */
6592
6593 MSG_PROC g_msg_proc[] = {
6594     {"CONNECT", connect_proc},
6595     {"DISCONNECT_REQUEST", disconnect_request_proc},
6596     {"DISCONNECT", disconnect_proc},
6597     {"CMD", cmd_proc},
6598     {NULL, NULL}
6599 };
6600
6601 CLIENT_INFO g_clients[MAX_CLIENT];
6602
6603 /* Function Definitions */
6604
6605 int main(void)
```

```
6606  {
6607      int fdp;
6608      CLIENT_MSG cmsg;
6609      MSG_CONTENTS msgc;
6610      int i;
6611
6612      if ((fdp = open(SERVER_PIPE, O_RDWR)) == -1)
6613          exit_sys("open");
6614
6615      for (;;) {
6616          if (get_client_msg(fdp, &cmsg) == -1)
6617              exit_sys("get_client_msg");
6618          print_msg(&cmsg);
6619          parse_msg(cmsg.msg, &msgc);
6620          for (i = 0; g_msg_proc[i].msg_cmd != NULL; ++i)
6621              if (!strcmp(msgc.msg_cmd, g_msg_proc[i].msg_cmd)) {
6622                  if (g_msg_proc[i].proc(cmsg.client_id, msgc.msg_param)) {
6623
6624                      }
6625                      break;
6626                  }
6627                  if (g_msg_proc[i].msg_cmd == NULL)
6628                      if (invalid_command(cmsg.client_id, msgc.msg_cmd) == -1)
6629                          continue;
6630          }
6631
6632          close(fdp);
6633
6634          return 0;
6635      }
6636
6637      int get_client_msg(int fdp, CLIENT_MSG *cmsg)
6638  {
6639          if (read(fdp, &cmsg->msglen, sizeof(int)) == -1)
6640              return -1;
6641
6642          if (read(fdp, &cmsg->client_id, sizeof(int)) == -1)
6643              return -1;
6644
6645          if (read(fdp, cmsg->msg, cmsg->msglen) == -1)
6646              return -1;
6647
6648          cmsg->msg[cmsg->msglen] = '\0';
6649
6650          return 0;
6651      }
6652
6653      int putmsg(int client_id, const char *cmd)
6654  {
6655          SERVER_MSG smsg;
6656          int fdp;
6657
6658          strcpy(smsg.msg, cmd);
```

```
6659     smsg.msglen = strlen(smsg.msg);
6660
6661     fdp = g_clients[client_id].fdp;
6662
6663     return write(fdp, &smsg, sizeof(int) + smsg.msglen) == -1 ? -1 : 0;
6664 }
6665
6666 void parse_msg(char *msg, MSG_CONTENTS *msgc)
6667 {
6668     int i;
6669
6670     msgc->msg_cmd = msg;
6671
6672     for (i = 0; msg[i] != ' ' && msg[i] != '\0'; ++i)
6673         ;
6674     msg[i++] = '\0';
6675     msgc->msg_param = &msg[i];
6676 }
6677
6678 void print_msg(const CLIENT_MSG *cmsg)
6679 {
6680     printf("Message from \"%s\": %s\n", cmsg->client_id ? g_clients[cmsg->client_id].path : "", cmsg->msg);
6681 }
6682
6683 int invalid_command(int client_id, const char *cmd)
6684 {
6685     char buf[MAX_MSG_LEN];
6686
6687     sprintf(buf, "INVALID_COMMAND %s", cmd);
6688     if (putmsg(client_id, buf) == -1)
6689         return -1;
6690
6691     return 0;
6692 }
6693
6694 int connect_proc(int client_id, const char *msg_param)
6695 {
6696     int fdp;
6697     char buf[MAX_MSG_LEN];
6698
6699     if (mkfifo(msg_param, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH) == -1) {
6700         printf("CONNECT message failed! Params = \"%s\"\n", msg_param);
6701         return -1;
6702     }
6703
6704     if ((fdp = open(msg_param, O_WRONLY)) == -1)
6705         exit_sys("open");
6706
6707     g_clients[fdp].fdp = fdp;
6708     strcpy(g_clients[fdp].path, msg_param);
6709
6710     sprintf(buf, "CONNECTED %d", fdp);
```

```
6711     if (putmsg(fdp, buf) == -1)
6712         exit_sys("putmsg");
6713
6714     return 0;
6715 }
6716
6717 int disconnect_request_proc(int client_id, const char *msg_param)
6718 {
6719     if (putmsg(client_id, "DISCONNECT_ACCEPTED") == -1)
6720         return -1;
6721
6722     return 0;
6723 }
6724
6725 int disconnect_proc(int client_id, const char *msg_param)
6726 {
6727     close(g_clients[client_id].fdp);
6728
6729     if (remove(g_clients[client_id].path) == -1)
6730         return -1;
6731
6732     return 0;
6733 }
6734
6735 int cmd_proc(int client_id, const char *msg_param)
6736 {
6737     FILE *f;
6738     char cmd[MAX_MSG_LEN] = "CMD_RESPONSE ";
6739     int i;
6740     int ch;
6741
6742     if ((f = popen(msg_param, "r")) == NULL) {
6743         printf("cannot execute shell command!..\n");
6744         return -1;
6745     }
6746
6747     for (i = 13; (ch = fgetc(f)) != EOF; ++i)
6748         cmd[i] = ch;
6749     cmd[i] = '\0';
6750
6751     if (putmsg(client_id, cmd) == -1)
6752         return -1;
6753
6754     return 0;
6755 }
6756
6757 void exit_sys(const char *msg)
6758 {
6759     perror(msg);
6760
6761     exit(EXIT_FAILURE);
6762 }
6763
```

```
6764  /
6765  *-----*
6766  ----- XSI Shared Memory: shmget, shmat ve shmdt fonksiyonlarının kullanımları
6767  -----*/
6768 /* proc1.c */
6769
6770 #include <stdio.h>
6771 #include <stdlib.h>
6772 #include <string.h>
6773 #include <sys/ipc.h>
6774 #include <sys/shm.h>
6775
6776 #define SHM_KEY      0x12345678
6777
6778 void exit_sys(const char *msg);
6779
6780 int main(void)
6781 {
6782     int shmid;
6783     void *addr;
6784     char *str;
6785
6786     if ((shmid = shmget(SHM_KEY, 8192, IPC_CREAT|0600)) == -1)
6787         exit_sys("shmget");
6788
6789     if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
6790         exit_sys("shmat");
6791
6792     str = (char *)addr;
6793
6794     strcpy(str, "This is a test....\n");
6795
6796     printf("press ENTER to continue...\n");
6797     getchar();
6798
6799     shmdt(addr);
6800
6801     return 0;
6802 }
6803
6804 void exit_sys(const char *msg)
6805 {
6806     perror(msg);
6807
6808     exit(EXIT_FAILURE);
6809 }
6810
6811
6812 /* proc2.c */
6813
```

```
6814 #include <stdio.h>
6815 #include <stdlib.h>
6816 #include <sys/ipc.h>
6817 #include <sys/shm.h>
6818
6819 #define SHM_KEY      0x12345678
6820
6821 void exit_sys(const char *msg);
6822
6823 int main(void)
6824 {
6825     int shmid;
6826     void *addr;
6827     char *str;
6828
6829     if ((shmid = shmget(SHM_KEY, 8192, IPC_CREAT|0600)) == -1)
6830         exit_sys("shmget");
6831
6832     if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
6833         exit_sys("shmat");
6834
6835     printf("press ENTER to continue..\n");
6836     getchar();
6837
6838     str = (char *)addr;
6839     puts(str);
6840
6841     shmdt(addr);
6842
6843     return 0;
6844 }
6845
6846 void exit_sys(const char *msg)
6847 {
6848     perror(msg);
6849
6850     exit(EXIT_FAILURE);
6851 }
6852
6853 /
*-----*-----*-----*
-----*-----*-----*
6854 Paylaşılan bellek alanı nesnesi (shared memory segment) onu hiçbir      ↗
proses kullanmıyor olsa bile sistem boot edilene kadar
6855 yaşamaya devam eder. Yani shmat ile yapılan işlem shmdt ile geri      ↗
alınmaktadır. Ancak shmget ile yapılan işlem eğer silinmezse
6856 sistem boor edilene kadar kalıcıdır. (Buna "kernel persistency"      ↗
denilmektedir.) Bu nesnesi silmek için shmctl fonksiyonunu
6857 IPC_RMID parametresiyle çağrırmak gereklidir. Silme işlemi için ipcrm isimli ↗
bir kabuk komutu da bulunmaktadır.
6858 -----*/
```

```
6860 if (shmctl(shmid, IPC_RMID, NULL) == -1)
6861     exit_sys("shmctl");
6862
6863 /
*-----*
6864 Paylaşılan bellek alanı için kernel tarafından oluşturulan shmid_ds      ↗
6865     yapısının bazı elemanları shmctl fonksiyonunun                    ↗
6866     IPC_STAT ve IPC_SET parametreleriyle değiştirilebilir            ↗
6867 -----*/
6868 #include <stdio.h>
6869 #include <stdlib.h>
6870 #include <sys/ipc.h>
6871 #include <sys/shm.h>
6872
6873 #define SHM_KEY      0x12345678
6874
6875 void exit_sys(const char *msg);
6876
6877 int main(void)
6878 {
6879     int shmid;
6880     struct shmid_ds ds;
6881
6882     if ((shmid = shmget(SHM_KEY, 8192, IPC_CREAT|0600)) == -1)
6883         exit_sys("shmget");
6884
6885     if (shmctl(shmid, IPC_STAT, &ds) == -1)
6886         exit_sys("shmctl");
6887
6888     ds.shm_perm.mode = 0666;
6889
6890     if (shmctl(shmid, IPC_SET, &ds) == -1)
6891         exit_sys("shmctl");
6892
6893     return 0;
6894 }
6895
6896
6897 void exit_sys(const char *msg)
6898 {
6899     perror(msg);
6900
6901     exit(EXIT_FAILURE);
6902 }
6903
6904 /
*-----*
6905 Paylaşılan bellek alanları yöntemi maalesef kendi içerisinde bir      ↗
6906     senkronizasyona sahip değildir. Bir prosesin bu alana            ↗
```

```
6906     sürekli bir şeyler yazıp diğerinin bunları okuması problemine "üretic,- →
       tüketici problemi (producer-consumer problem"
6907     denilmektedir. Üretici tüketici problemi "semaphore" denilen özel →
       senkronizasyon nesneleriyle çözülmektedir. Aşağıdaki
6908     örnekte bir senkronizasyon olmazsa okuyan prosesin aynı bilgiyi birden →
       fazla kez alabildiği ve bilgi kaçırıldığı gösterilmek
6909     istenmiştir.
6910     -----
6911     -----
6912 /* proc1. c */
6913
6914 #include <stdio.h>
6915 #include <stdlib.h>
6916 #include <string.h>
6917 #include <time.h>
6918 #include <unistd.h>
6919 #include <sys/ipc.h>
6920 #include <sys/shm.h>
6921
6922 #define SHM_KEY      0x12345678
6923
6924 struct SHARED_MEM_INFO {
6925     int val;
6926     /* ... */
6927 };
6928
6929 void exit_sys(const char *msg);
6930
6931 int main(void)
6932 {
6933     int shmid;
6934     void *addr;
6935     struct SHARED_MEM_INFO *smi;
6936
6937     srand(time(NULL));
6938
6939     if ((shmid = shmget(SHM_KEY, 8192, IPC_CREAT|0600)) == -1)
6940         exit_sys("shmget");
6941
6942     if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
6943         exit_sys("shmat");
6944
6945     smi = (struct SHARED_MEM_INFO *)addr;
6946
6947     for (int i = 0; i < 100; ++i) {
6948         smi->val = i;
6949         usleep(rand() % 300000);
6950     }
6951
6952     shmdt(addr);
6953
6954     if (shmctl(shmid, IPC_RMID, NULL) == -1)
```

```
6955         exit_sys("shmctl");
6956
6957     return 0;
6958 }
6959
6960 void exit_sys(const char *msg)
6961 {
6962     perror(msg);
6963
6964     exit(EXIT_FAILURE);
6965 }
6966
6967 /* proc2.c */
6968
6969 #include <stdio.h>
6970 #include <stdlib.h>
6971 #include <time.h>
6972 #include <unistd.h>
6973 #include <sys/ipc.h>
6974 #include <sys/shm.h>
6975
6976 #define SHM_KEY      0x12345678
6977
6978 struct SHARED_MEM_INFO {
6979     int val;
6980     /* ... */
6981 };
6982
6983 void exit_sys(const char *msg);
6984
6985 int main(void)
6986 {
6987     int shmid;
6988     void *addr;
6989     struct SHARED_MEM_INFO *smi;
6990
6991     srand(time(NULL));
6992
6993     if ((shmid = shmget(SHM_KEY, 8192, IPC_CREAT|0600)) == -1)
6994         exit_sys("shmget");
6995
6996     if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
6997         exit_sys("shmat");
6998
6999     smi = (struct SHARED_MEM_INFO *)addr;
7000
7001     for (;;) {
7002         printf("%d ", smi->val);
7003         fflush(stdout);
7004         usleep(rand() % 300000);
7005         if (smi->val == 99)
7006             break;
7007     }
```

```
7008     printf("\n");
7009
7010     shmdt(addr);
7011
7012     return 0;
7013 }
7014
7015 void exit_sys(const char *msg)
7016 {
7017     perror(msg);
7018
7019     exit(EXIT_FAILURE);
7020 }
7021
7022 /
*-----*
-----*
7023     Linux'a özgü biçimde shmat fonksiyonunun son parametresinde SHM_EXEC      ↵
    bayrağı kullanılabilir. Bu durumda paylaşan bellek alanına
7024     yerleştirilen kod çalıştırılabilir durumda olmaktadır.
7025 -----* /
7026
7027 #include <stdio.h>
7028 #include <stdlib.h>
7029 #include <string.h>
7030 #include <unistd.h>
7031 #include <sys/ipc.h>
7032 #include <sys/shm.h>
7033
7034 #define SHM_KEY      0x12345678
7035
7036 void exit_sys(const char *msg);
7037 void foo(void);
7038
7039 int main(void)
7040 {
7041     int shmid;
7042     void *addr;
7043     void (*pf)(void);
7044
7045     if ((shmid = shmget(SHM_KEY, 8192, IPC_CREAT|0777)) == -1)
7046         exit_sys("shmget");
7047
7048     if ((addr = shmat(shmid, NULL, SHM_EXEC)) == (void *)-1)
7049         exit_sys("shmat");
7050
7051     memcpy(addr, foo, 20);
7052     pf = (void (*)(void))addr;
7053     pf();
7054
7055     shmdt(addr);
7056
```

```
7057     if (shmctl(shmid, IPC_RMID, NULL) == -1)
7058         exit_sys("shmctl");
7059
7060     return 0;
7061 }
7062
7063 void foo(void)
7064 {
7065
7066 }
7067
7068 void exit_sys(const char *msg)
7069 {
7070     perror(msg);
7071
7072     exit(EXIT_FAILURE);
7073 }
7074
7075 /
*-----*
-----*
7076 System 5 IPC nesneleri anahtar hareketle id oluşturmaktadır. Anahtarlar key_t türündendir. Ancak anahtar numaraları okunabilir 7077 değildir. Bunun yerine bir dosyadan hareketle anahtar değeri uyduuran ftok isimli bir POSIX fonksiyonundan faydalanylabilmektedir. 7078 ftok dosyanın i-node numarası, dosyanın içerisinde bulunduğu aygıt 7079 numarası ve bizim verdiğimiz değeri kombine ederek bir anahtar 7080 uydurmaktadır. Böylece iki program aynı dosya isminden hareketle ortak paylaşım alanı oluşturabilmektedir.
7081 -----
-----*/
7082 /* proc1.c */
7083
7084 #include <stdio.h>
7085 #include <stdlib.h>
7086 #include <string.h>
7087 #include <unistd.h>
7088 #include <sys/ipc.h>
7089 #include <sys/shm.h>
7090
7091 void exit_sys(const char *msg);
7092
7093 int main(void)
7094 {
7095     int shmid;
7096     void *addr;
7097     char *str;
7098     key_t key;
7099
7100     if ((key = ftok("myfile", 123)) == -1)
7101         exit_sys("ftok");
7102 }
```

```
7103     if ((shmid = shmget(key, 8192, IPC_CREAT|0600)) == -1)
7104         exit_sys("shmget");
7105
7106     if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
7107         exit_sys("shmat");
7108
7109     str = (char *)addr;
7110
7111     strcpy(str, "this is a test...");
7112
7113     printf("Press ENTER to continue...\n");
7114     getchar();
7115
7116     shmdt(addr);
7117
7118     if (shmctl(shmid, IPC_RMID, NULL) == -1)
7119         exit_sys("shmctl");
7120
7121     return 0;
7122 }
7123
7124 void exit_sys(const char *msg)
7125 {
7126     perror(msg);
7127
7128     exit(EXIT_FAILURE);
7129 }
7130
7131 /* proc2.c */
7132
7133 #include <stdio.h>
7134 #include <stdlib.h>
7135 #include <unistd.h>
7136 #include <sys/ipc.h>
7137 #include <sys/shm.h>
7138
7139 void exit_sys(const char *msg);
7140
7141 int main(void)
7142 {
7143     int shmid;
7144     void *addr;
7145     char *str;
7146     key_t key;
7147
7148     if ((key = ftok("myfile", 123)) == -1)
7149         exit_sys("ftok");
7150
7151     if ((shmid = shmget(key, 8192, IPC_CREAT|0600)) == -1)
7152         exit_sys("shmget");
7153
7154     if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
7155         exit_sys("shmat");
```

```
7156
7157     str = (char *)addr;
7158
7159     puts(str);
7160
7161     shmdt(addr);
7162
7163     return 0;
7164 }
7165
7166 void exit_sys(const char *msg)
7167 {
7168     perror(msg);
7169
7170     exit(EXIT_FAILURE);
7171 }
7172
7173 /
*-----*
-----*
7174     POSIX paylaşılan bellek alanları shm_open, ftruncate, mmap, mmap ve      ↵
   shm_unlink fonksiyonları yardımıyla oluşturulmaktadır.
7175     shm_open fonksiyonuyla yaratılmış olan paylaşılan bellek alanı nesni   ↵
   sistem reboot edilene kadar ya da shm_unlink fonksiyonuyla
7176     silme yapılana kadar kalmaya devam etmektedir. mmap fonksiyonu UNIX      ↵
   turevi sistemlerde kullanılan genel amaçlı bir mapping
7177     fonksiyonudur.
7178 -----*/
```

7179 /* proc1.1 */

7180 #include <stdio.h>

7181 #include <stdlib.h>

7182 #include <string.h>

7183 #include <fcntl.h>

7184 #include <unistd.h>

7185 #include <sys/stat.h>

7186 #include <sys/mman.h>

7187

7188 #define SHM_PATH "/this_is_a_text"

7189

7190 void exit_sys(const char *msg);

7191

7192 int main(void)

7193 {

7194 int fdshm;

7195 void *addr;

7196 char *str;

7197

7198 if ((fdshm = shm_open(SHM_PATH, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR|S_IRGRP| ↵
 S_IROTH)) == -1)
7199 exit_sys("shm_open");

```
7202
7203     if (ftruncate(fdshm, 4096) == -1)
7204         exit_sys("ftruncate");
7205
7206     if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fdshm,    ↴
7207         0)) == MAP_FAILED)
7208         exit_sys("mmap");
7209
7210     str = (char *)addr;
7211     strcpy(str, "this is a test...");
7212
7213     printf("press ENTER to continue...\n");
7214     getchar();
7215
7216     munmap(addr, 4096);
7217
7218     close(fdshm);
7219
7220     if (shm_unlink(SHM_PATH) == -1)
7221         exit_sys("shm_unlink");
7222
7223     return 0;
7224 }
7225 void exit_sys(const char *msg)
7226 {
7227     perror(msg);
7228
7229     exit(EXIT_FAILURE);
7230 }
7231
7232 /* proc2.c */
7233
7234 #include <stdio.h>
7235 #include <stdlib.h>
7236 #include <fcntl.h>
7237 #include <sys/stat.h>
7238 #include <sys/mman.h>
7239
7240 #define SHM_PATH          "/this_is_a_text"
7241
7242 void exit_sys(const char *msg);
7243
7244 int main(void)
7245 {
7246     int fdshm;
7247     void *addr;
7248     char *str;
7249
7250     if ((fdshm = shm_open(SHM_PATH, O_RDWR, 0)) == -1)
7251         exit_sys("shm_open");
7252
7253     if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fdshm,    ↴
```

```
    0)) == MAP_FAILED)
7254     exit_sys("mmap");
7255
7256     str = (char *)addr;
7257
7258     puts(str);
7259
7260     munmap(addr, 4096);
7261
7262     close(fdshm);
7263
7264     return 0;
7265 }
7266
7267 void exit_sys(const char *msg)
7268 {
7269     perror(msg);
7270
7271     exit(EXIT_FAILURE);
7272 }
7273
7274 /
*-----*
-----*
7275 Bir dosyanın tamamının ya da belli bir kısmının otomatik biçimde belleğe ↵
    çekilip dosya fonksiyonları yerine göstericilerle
7276 bellek üzerinde dosya işlemleri yapmaya "bellek tabanlı dosyalar (memory ↵
    mapped files) denilmektedir". Bellek tabanlı dosyalar
7277 için open, mmap, munmap, close fonksionlarından faydalananır. Dosya önce ↵
    open fonksiyonuyla açılır sonra mmap fonksiyonu ile
7278 dosyanın tamamı ya da bir parçası belleğe çekilir.
7279 -----*/
```

7280

```
7281 #include <stdio.h>
7282 #include <stdlib.h>
7283 #include <unistd.h>
7284 #include <fcntl.h>
7285 #include <sys/mman.h>
```

7286

```
7287 void exit_sys(const char *msg);
```

7288

```
7289 int main(void)
7290 {
7291     int fd;
7292     void *addr;
7293     char *str;
7294     int len;
7295     int i;
7296
7297     if ((fd = open("test.txt", O_RDWR)) == -1)
7298         exit_sys("open");
7299 }
```

```
7300     len = (int)lseek(fd, 0, SEEK_END);
7301
7302     if ((addr = mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)) == ↪
7303         MAP_FAILED)
7304         exit_sys("mmap");
7305
7306     str = (char *)addr;
7307
7308     for (i = 0; i < len; ++i)
7309         putchar(str[i]);
7310     putchar('\n');
7311
7312     munmap(addr, len);
7313
7314     close(fd);
7315
7316     return 0;
7317 }
7318 void exit_sys(const char *msg)
7319 {
7320     perror(msg);
7321
7322     exit(EXIT_FAILURE);
7323 }
7324
7325 /
*-----*/  

-----  

7326 Bir dosya bellek tabanlı biçimde açıldığında bellek üzerinde dosya      ↪
    içeriği günellendığında bu değişiklikler hemen bu dosyayı      ↪
7327 open ile açıp read ile okuyan proseslerde gözükmür mü? POSIX      ↪
    stnadartlarına göre bunun bir garantisı yoktur. Bizim bunu garanti      ↪
7328 altına almamız için sync isimli POSIX fonksiyonunu MS_SYNC bayrak değeri      ↪
    ile çağrırmamız gereklidir. Ancak Linux işletim sistemi      ↪
7329 dosya sistemi gerçekleştirmesinden dolayı bu tür değişikliklerin başka      ↪
    prosesler tarafından hiç msync yapılmadan görülmemesine      ↪
7330 olanak vermektedir. (Yani Linux'ta aslında başka bir proses open      ↪
    fonksiyonuyla dışarıdan dosyayı açıp read ile okuduğu zaman      ↪
    çekirdek dosyanın ilgili parçasının o anda belleğe map edildiğini      ↪
    bildiği için zaten read fonksiyonu diske başvurmadan onu      ↪
7332 bellekten almaktadır.) Aşağıdaki programda dosya bellekte strcpy      ↪
    fonksiyonu ile güncellenmiştir. Bu güncellemedne sonra      ↪
7333 klavyeden ENTER tuşuna basılana kadar bekleme yapılmıştır. Bu noktada      ↪
    başka bir terminalden girek dosyayı cat ile açıp okumayı      ↪
7334 deneyiniz. Tabii biz bellek tabanlı dosyayı bellekte güncellediğimizde      ↪
    dosyanın diskteki mevcudiyeti güncellenmek zorunda değildir.
7335 Bunu garanti altına almak için msync fonksiyonu MS_SYNC bayrağı ile      ↪
    açırlmalıdır.  

7336 -----*/  

7337
7338 #include <stdio.h>
```

```
7339 #include <stdlib.h>
7340 #include <string.h>
7341 #include <unistd.h>
7342 #include <fcntl.h>
7343 #include <sys/mman.h>
7344
7345 void exit_sys(const char *msg);
7346
7347 int main(void)
7348 {
7349     int fd;
7350     void *addr;
7351     char *str;
7352     int len;
7353     int i;
7354
7355     if ((fd = open("test.txt", O_RDWR)) == -1)
7356         exit_sys("open");
7357
7358     len = (int)lseek(fd, 0, SEEK_END);
7359
7360     if ((addr = mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)) == ↪
7361         MAP_FAILED)
7362         exit_sys("mmap");
7363
7364     str = (char *)addr;
7365
7366     for (i = 0; i < len; ++i)
7367         putchar(str[i]);
7368     putchar('\n');
7369
7370     strcpy(str, "aaaaaaaaaaaaaaaaaa");
7371     printf("press ENTER to continue...\n");
7372     getchar();
7373
7374     munmap(addr, len);
7375
7376     close(fd);
7377
7378     return 0;
7379 }
7380
7381 void exit_sys(const char *msg)
7382 {
7383     perror(msg);
7384
7385     exit(EXIT_FAILURE);
7386 }
7387 /
*-----→
-----→
7388 Yukarıdaki işlemin tersine ilişkin örnek de aşağıdadır. Yani dosya →
```

```
    bellek tabanlı bir biçimde açılmış fakat o noktada ENTER      ↵
7389    tuşuna basılana kadar beklenmiştir. Dosyayı dışarıdan bir editör      ↵
        yardımıyla güncelleyiniz. Linux msync fonksiyonuna gereksinim      ↵
7390    duyulmadan bellekteki halini güncelleyecektir. (Tabii aslında yine      ↵
        editör güncellemeyi disk üzerinde değil zaten bellekteki      ↵
7391    map edilmiş alan üzerinde yapmaktadır.) Fakat POSIX standartlarında      ↵
        bunun garanti edilmesi için msync fonksiyonunun MS_INVALIDATE      ↵
7392    bayrağı ile çağrılması gerekmektedir.      ↵
7393  -----*/  

7394  
7395 #include <stdio.h>  
7396 #include <stdlib.h>  
7397 #include <string.h>  
7398 #include <unistd.h>  
7399 #include <fcntl.h>  
7400 #include <sys/mman.h>  
7401  
7402 void exit_sys(const char *msg);  
7403  
7404 int main(void)  
7405 {  
7406     int fd;  
7407     void *addr;  
7408     char *str;  
7409     int len;  
7410     int i;  
7411  
7412     if ((fd = open("test.txt", O_RDWR)) == -1)  
7413         exit_sys("open");  
7414  
7415     len = (int)lseek(fd, 0, SEEK_END);  
7416  
7417     if ((addr = mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)) == ↵
          MAP_FAILED)  
7418         exit_sys("mmap");  
7419  
7420     printf("press ENTER to continue...\n");  
7421     getchar();  
7422  
7423     str = (char *)addr;  
7424  
7425     for (i = 0; i < len; ++i)  
7426         putchar(str[i]);  
7427     putchar('\n');  
7428  
7429     munmap(addr, len);  
7430  
7431     close(fd);  
7432  
7433     return 0;  
7434 }  
7435
```

```
7436 void exit_sys(const char *msg)
7437 {
7438     perror(msg);
7439
7440     exit(EXIT_FAILURE);
7441 }
7442
7443 /
*-----*
-----*
7444 Her ne kadar Linux için gerek olmasa da taşınabilir bir program için      ↵
    POSIX standartlarında belirtiği gibi msync fonksiyonunun
7445 MS_SYNC ya da MS_INVALIDATE bayraklarıyla çağrılmaması gereklidir. MS_SYNC      ↵
    bellekten diskteki dosyaya, MS_INVALIDATE ise diskteki
7446 dosyadan belleğe tazeleme yapmaktadır.
7447 -----*/
```

```
7448
7449 #include <stdio.h>
7450 #include <stdlib.h>
7451 #include <string.h>
7452 #include <unistd.h>
7453 #include <fcntl.h>
7454 #include <sys/mman.h>
7455
7456 void exit_sys(const char *msg);
7457
7458 int main(void)
7459 {
7460     int fd;
7461     void *addr;
7462     char *str;
7463     int len;
7464     int i;
7465
7466     if ((fd = open("test.txt", O_RDWR)) == -1)
7467         exit_sys("open");
7468
7469     len = (int)lseek(fd, 0, SEEK_END);
7470
7471     if ((addr = mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)) == ↵
        MAP_FAILED)
7472         exit_sys("mmap");
7473
7474     printf("press ENTER to continue...\n");
7475     getchar(); getchar();
7476
7477     if (msync(addr, len, MS_INVALIDATE) == -1)
7478         exit_sys("msync");
7479
7480     str = (char *)addr;
7481
7482     for (i = 0; i < len; ++i)
```

```
7483     putchar(str[i]);
7484     putchar('\n');
7485
7486     strcpy(str, "xxxxxxxxxxxxxxxxxxxxxxxxxxxx");
7487
7488     printf("press ENTER to continue...\n");
7489     getchar();
7490
7491     if (msync(addr, len, MS_SYNC) == -1)
7492         exit_sys("msync");
7493
7494     munmap(addr, len);
7495
7496     close(fd);
7497
7498     return 0;
7499 }
7500
7501 void exit_sys(const char *msg)
7502 {
7503     perror(msg);
7504
7505     exit(EXIT_FAILURE);
7506 }
7507
7508 /
*-----*
```

7509 mmap fonksiyonunda MAP_PRIVATE yapılan değişikliklerin asıl dosyaya
yansıtılmayacağı gerektiğinde swap dosyasına yapılacağı
anlamına gelir. Bu copy on write mekanizmasıdır. MAP_SHARED yazılanların
asıl dosyaya aktarılacağı anlamına gelmektedir.

7511 Prosesler paylaşılan bellek alanlarını ya da dosyayı MAP_PRIVATE ile map
ederlerse birbirlerinin yazdıklarını göremezler.

7512 Bunun için MAP_SHARED gerekmektedir. Linux sistemlerinde POSIX'te
belirtilen bayraklardan çok daha fazlası vardır. Bu ekstra
bayrakların en önemlisi MAP_ANONYMOUS bayrağıdır. Buna "anonymous"
mapping denilmektedir. Bu bayrak MAP_PRIVATE ve MAP_SHARED
ile birlikte kullanılabilir. Anonymous mapping dosya ile ilgili olmadan
yani yalnızca sanal bellek alanında sayfa tabanlı tahsisat
yapmak için kullanılmaktadır. Dolayısıyla MAP_ANONYMOUS bayrağı
belirtildiğinde artık mmap fonksiyonunun son iki parametresi
(fd, offset) dikkate alınmamaktadır. MAP_ANONYMOUS|MAP_PRIVATE mapping
sanal bellek alanında dosyadan bağımsız tahsisat yapmak için
ancak alt prosese fork yapıldığında bu alanların farklı kopyalarının
kullanılması için tercih edilmektedir. MAP_ANONYMOUS|MAP_SHARED
ise fork işleminden sonra üst ve alt proseslerin aynı bellek bölgesini
görmesi için kullanılır. Aşağıdaki örnekte üst ve alt prosesler
bu yolla haberleşebilmektedir.

7521 -----*/

7522 #include <stdio.h>

```
7523 #include <stdlib.h>
7524 #include <string.h>
7525 #include <unistd.h>
7526 #include <sys/wait.h>
7527 #include <sys/mman.h>
7528
7529 void exit_sys(const char *msg);
7530
7531 int main(void)
7532 {
7533     void *addr;
7534     char *str;
7535     pid_t pid;
7536
7537     if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_ANONYMOUS|
7538                  MAP_SHARED, 0, 0)) == MAP_FAILED)
7539         exit_sys("mmap");
7540     str = (char *)addr;
7541
7542     if ((pid = fork()) == -1)
7543         exit_sys("fork");
7544
7545     if (pid != 0) { /* parent process */
7546         strcpy(str, "this is a test...");
7547         if (waitpid(pid, NULL, 0) == -1)
7548             exit_sys("waitpid");
7549     }
7550     else { /* child process */
7551         sleep(1);
7552         puts(str);
7553     }
7554
7555     munmap(addr, 4096);
7556
7557     return 0;
7558 }
7559
7560 void exit_sys(const char *msg)
7561 {
7562     perror(msg);
7563     exit(EXIT_FAILURE);
7564 }
7565
7566 /
*-----*
-----*
7567 Linux'ta POSIX shared memory nesneleri aslında /dev/shm dizininde birer dosya biçiminde görünmektedir. Yani bizim shm_open fonksiyonuyla kök dizinde bir isimle açtığımız dosya gerçekte /dev/shm dizininde yaratılmaktadır. /dev/shm dizini tmpfs denilen RAM tabanlı dosya sistemine ilişkindir. Yani /dev/shm dizini içerisindeki dosyalar aslında diskte değil RAM'da tutulmaktadır.
```

```
7570     Sistem reboot edilene kadar ya da shm_unlink fonksiyonu çağrılana kadar →  
          oarada durmaya devam ederler.  
7571 -----*/  
7572  
7573 csd@csd-vm:~/Study/Unix-Linux-SysProg$ ls /dev/shm -l  
7574 total 4  
7575 -rw-r--r-- 1 csd study 4096 Jun 11 21:47 this_is_a_text  
7576  
7577 /*-----*/  
7578     Sistem 5 mesaj kuyrukları msgget fonksiyonuyla yaratılır ya da açılır, →  
          msgsnd fonksiyonuyla kuyruğa mesaj gönderilir.  
7579     msgrcv fonksiyonu ile kuyruktan mesaj alınır. msgctl IPC_RMID kodu ile →  
          mesaj kuyruğu silinir. Mesajı gönderirken mesaj bilgisinin →  
7580     önünde onun türünü belirten long bir type alanı olmalıdır. Mesajın →  
          uzunluğuna bu alan dahil değildir. Mesaj alınırken de →  
7581     her zaman mesajın yerleştirileceği alanın önünde yine long bir type →  
          alanı olmalıdır. Mesajı alan taraf onun türünü de alır.  
7582     Böylece farklı prosesler aynı kuyruktan farklı type değerlerine ilişkin →  
          mesajları okuyabilirler. 0 type değeri kullanılmaz,  
7583     özel anlam ifade eder.  
7584 -----*/  
7585  
7586 /* proc1.c */  
7587  
7588 #include <stdio.h>  
7589 #include <stdlib.h>  
7590 #include <string.h>  
7591 #include <ctype.h>  
7592 #include <sys/ipc.h>  
7593 #include <sys/msg.h>  
7594  
7595 #define MAX_LEN      1024  
7596  
7597 void exit_sys(const char *msg);  
7598  
7599 typedef struct tagMSG {  
7600     long type;  
7601     char buf[MAX_LEN];  
7602 } MSG;  
7603  
7604 int main(void)  
7605 {  
7606     key_t key;  
7607     int msgid;  
7608     MSG msg;  
7609     char *str;  
7610     int ch;  
7611  
7612     if ((key = ftok("mymsg", 123)) == -1)
```

```
7613         exit_sys("ftok");
7614
7615     if ((msgid = msgget(key, IPC_CREAT|0666)) == -1)
7616         exit_sys("msgget");
7617
7618     for (;;) {
7619         printf("Bir type değeri ve yanına bir yazı giriniz:");
7620         scanf("%ld", &msg.type);
7621         while ((ch = getchar(), isspace(ch))
7622                 ;
7623             ungetc(ch, stdin);
7624             fgets(msg.buf, MAX_LEN, stdin));
7625             if ((str = strchr(msg.buf, '\n')) != NULL)
7626                 *str = '\0';
7627             if (msgsnd(msgid, &msg, strlen(msg.buf) + 1, 0) == -1)
7628                 exit_sys("msgsnd");
7629             if (!strcmp(msg.buf, "quit"))
7630                 break;
7631     }
7632
7633     if (msgctl(msgid, IPC_RMID, 0) == -1)
7634         exit_sys("msgctl");
7635
7636     return 0;
7637 }
7638
7639 void exit_sys(const char *msg)
7640 {
7641     perror(msg);
7642
7643     exit(EXIT_FAILURE);
7644 }
7645
7646 /* proc2.c */
7647
7648 #include <stdio.h>
7649 #include <stdlib.h>
7650 #include <string.h>
7651 #include <sys/ipc.h>
7652 #include <sys/msg.h>
7653
7654 #define MAX_LEN      1024
7655
7656 typedef struct tagMSG {
7657     long type;
7658     char buf[MAX_LEN];
7659 } MSG;
7660
7661 void exit_sys(const char *msg);
7662
7663 int main(void)
7664 {
7665     key_t key;
```

```
7666     int msgid;
7667     MSG msg;
7668
7669     if ((key = ftok("mymsg", 123)) == -1)
7670         exit_sys("ftok");
7671
7672     if ((msgid = msgget(key, 0666)) == -1)
7673         exit_sys("msgget");
7674
7675     for (;;) {
7676         if (msgrcv(msgid, &msg, MAX_LEN, 0, 0) == -1)
7677             exit_sys("msgrcv");
7678         printf("type: %ld, message: \"%s\"\n", msg.type, msg.buf);
7679         if (!strcmp(msg.buf, "quit"))
7680             break;
7681     }
7682
7683     return 0;
7684 }
7685
7686 void exit_sys(const char *msg)
7687 {
7688     perror(msg);
7689     exit(EXIT_FAILURE);
7690 }
7691
7692 /
7693 */
*-----*
-----*
7694 msgctl IPC_STAT koduya çekirdek tarafından oluşturulan msgqid_ds yapısı
    içerisindeki değerleri alabiliriz. Burada kuyrukta
7695 toplam kaç mesajın olduğu, kuyrukta toplam kaç byte'in bulunduğu,
    kuyrukta olabilecek maksimum byte sayısı gibi bilgiler de
7696 vardır.
7697 -----*/*
7698 /* msgctl.c */
7699
7700
7701 #include <stdio.h>
7702 #include <stdlib.h>
7703 #include <string.h>
7704 #include <getopt.h>
7705 #include <sys/ipc.h>
7706 #include <sys/msg.h>
7707
7708 void exit_sys(const char *msg);
7709
7710 /* ./msgctl [-q][Q] [id or key] */
7711
7712 int main(int argc, char *argv[])
7713 {
```

```
7714     int result;
7715     int q_flag = 0, Q_flag = 0;
7716     char *arg;
7717     long argval;
7718     int msgid;
7719     struct msqid_ds msqds;
7720
7721     opterr = 0;
7722     while ((result = getopt(argc, argv, "q:Q:")) != -1) {
7723         switch (result) {
7724             case 'q':
7725                 q_flag = 1;
7726                 arg = optarg;
7727                 break;
7728             case 'Q':
7729                 Q_flag = 1;
7730                 arg = optarg;
7731                 break;
7732             case '?':
7733                 if (optopt == 'q' || optopt == 'Q')
7734                     fprintf(stderr, "c switch without argument!..\n");
7735                 else
7736                     fprintf(stderr, "invalid switch: -%c\n", optopt);
7737                 exit(EXIT_FAILURE);
7738         }
7739     }
7740
7741     if (q_flag + Q_flag == 0) {
7742         fprintf(stderr, "neither -q nor -Q flag was given!\n");
7743         exit(EXIT_FAILURE);
7744     }
7745
7746     if (q_flag + Q_flag > 1) {
7747         fprintf(stderr, "both -q and -Q flag must not be given!\n");
7748         exit(EXIT_FAILURE);
7749     }
7750
7751     if (optind != argc) {
7752         fprintf(stderr, "too many arguments!..\n");
7753         exit(EXIT_FAILURE);
7754     }
7755
7756     argval = strtol(arg, NULL, 10);
7757
7758     if (Q_flag) {
7759         if ((msgid = msgget(argval, 0)) == -1)
7760             exit_sys("msgget");
7761     }
7762     else
7763         msgid = argval;
7764
7765     if (msgctl(msgid, IPC_STAT, &msqds) == -1)
7766         exit_sys("msgctl");
```

```
7767
7768     printf("Number of messages in queue: %lu\n", (unsigned long) ↵
7769         msqds.msg_qnum);
7770     printf("Maximum number of bytes allowed in queue: %lu\n", (unsigned ↵
7771         long)msqds.msg_qbytes);
7772     printf("Current number of bytes in queue: %lu\n", (unsigned long) ↵
7773         msqds._msg_cbytes);
7774
7775     return 0;
7776 }
7777 void exit_sys(const char *msg)
7778 {
7779     perror(msg);
7780 }
7781
7782 /
*-----*
-----  

7783     POSIX mesaj kuyrukları mq_open fonksiyonuyla yaratılır ya da açılır. ↵
7784     mq_send fonksiyonuyla kuyruğa mesaj bırakılır, mq_receive ↵
7785     fonksiyonuyla da kuyruktaki mesaj alınır. Kuyruğun mq_attr ile temsil ↵
7786     edilen dört özelliği vardır. Bu özellikler kuyruk yaratılırken ↵
7787     mq_open fonksiyonunda girilebilmektedir. Kuyruk özellikleri mq_getattr ↵
7788     fonksiyonuyla alınır. Eğer kuyruk yaratılırken bu ↵
7789     özellikler ilgili parametre NULL geçilerek belirtilmemişse default ↵
7790     özelliklerle kuyruk yaratılmaktadır. POSIX mesaj kuyrukları ↵
7791     tipki POSIX paylaşılan bellek alanları gibi "kernel persistent" ↵
7792     biçimindedir. Yani mq_unlink fonksiyonuyla silinmezlerse reboot ↵
7793     işlemine kadar yaşamaya devam ederler. mq_receive fonksiyonunda buffer ↵
7794     uzunluğunun en az mq_attr'de belirtilen uzunluk kadar ↵
7795     olması gerekmektedir. Kuyruklar da dosyalar gibi mq_close fonksiyonuyla ↵
7796     kapatılırlar. Linux işletim sistemi POSIX kuyruklarının ↵
7797     handle değerlerini dosya betimleyicisi biçiminde almaktadır. Ancak diğer ↵
7798     sistemlerde kuyruk handle değerlerinin dosya betimleyicisi ↵
7799     olması zorunlu değildir. POSIX mesaj kuyrukları da kuyruk için ayrılan ↵
7800     alan dolduysa mq_send fonksiyonunda, kuyrukta hiç mesaj yoksa ↵
7801     mq_receive fonksiyonunda bloke olurlar. Tabii O_NONBLOCK aşı bayrağı ile ↵
    nonblocking işlemler yapılabilir. Bu durumda mq_receive ↵
    ve mq_send fonksiyonları bloke olmazlar -1 değerine geri dönerler, errno ↵
    değişkeni de bu durumda EAGAIN değeriyle set edilmektedir.
```

```
7802 #include <string.h>
7803 #include <cctype.h>
7804 #include <sys/stat.h>
7805 #include <mqueue.h>
7806
7807 #define MSG_QUEUE_PATH      "/this_is_a_message_queue_last"
7808
7809 void exit_sys(const char *msg);
7810
7811 int main(void)
7812 {
7813     mqd_t mq;
7814     char *buf;
7815     struct mq_attr attr;
7816     int prio;
7817     int ch;
7818     char *str;
7819
7820     if ((mq = mq_open(MSG_QUEUE_PATH, O_CREAT|O_WRONLY, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH, NULL)) == -1)
7821         exit_sys("mq_open");
7822
7823     if (mq_getattr(mq, &attr) == -1)
7824         exit_sys("mq_getattr");
7825
7826     printf("Default attributes of the queue:\n");
7827     printf("Max msg: %ld\n", attr.mq_maxmsg);
7828     printf("Max msg size: %ld\n", attr.mq_msgsize);
7829     printf("Number of messages in queue: %ld\n", attr.mq_curmsgs);
7830     printf("-----\n");
7831
7832     if ((buf = (char *)malloc(attr.mq_msgsize)) == NULL)
7833         exit_sys("malloc");
7834
7835     for (;;) {
7836         printf("Bir öncelik derecesi ve yanına bir yazı giriniz:");
7837         scanf("%d", &prio);
7838         while ((ch = getchar()), isspace(ch))
7839             ;
7840         ungetc(ch, stdin);
7841         fgets(buf, MAX_LEN, stdin);
7842         if ((str = strchr(buf, '\n')) != NULL)
7843             *str = '\0';
7844
7845         if (mq_send(mq, buf, strlen(buf) + 1, prio))
7846             exit_sys("mq_send");
7847
7848         if (!strcmp(buf, "quit"))
7849             break;
7850     }
7851
7852     free(buf);
7853     mq_close(mq);
```

```
7854
7855     if (mq_unlink(MSG_QUEUE_PATH) == -1)
7856         exit_sys("mq_unlink");
7857
7858     return 0;
7859 }
7860
7861 void exit_sys(const char *msg)
7862 {
7863     perror(msg);
7864
7865     exit(EXIT_FAILURE);
7866 }
7867
7868 /* proc2.c */
7869
7870 #include <stdio.h>
7871 #include <stdlib.h>
7872 #include <string.h>
7873 #include <sys/stat.h>
7874 #include <mqueue.h>
7875
7876 #define MSG_QUEUE_PATH      "/this_is_a_message_queue_last"
7877
7878 void exit_sys(const char *msg);
7879
7880 int main(void)
7881 {
7882     mqd_t mq;
7883     char *buf;
7884     unsigned int prio;
7885     ssize_t result;
7886     struct mq_attr attr;
7887
7888     if ((mq = mq_open(MSG_QUEUE_PATH, O_RDONLY)) == -1)
7889         exit_sys("mq_open");
7890
7891     if (mq_getattr(mq, &attr) == -1)
7892         exit_sys("mq_getattr");
7893
7894     printf("Default attributes of the queue:\n");
7895     printf("Max msg: %ld\n", attr.mq_maxmsg);
7896     printf("Max msg size: %ld\n", attr.mq_msgsize);
7897     printf("Number of messages in queue: %ld\n", attr.mq_curmsgs);
7898     printf("-----\n");
7899
7900     if ((buf = (char *)malloc(attr.mq_msgsize)) == NULL)
7901         exit_sys("malloc");
7902
7903     for (;;) {
7904         if ((result = mq_receive(mq, buf, MAX_LEN, &prio)) == -1)
7905             exit_sys("mq_receive");
7906         printf("%ld bytes received at priority %u: \"%s\"\n", (long)result, ↵
```

```
        prio, buf);
7907     if (!strcmp(buf, "quit"))
7908         break;
7909     }
7910
7911     free(buf);
7912
7913     mq_close(mq);
7914
7915     return 0;
7916 }
7917
7918 void exit_sys(const char *msg)
7919 {
7920     perror(msg);
7921
7922     exit(EXIT_FAILURE);
7923 }
7924
7925 /
*-----*
-----*
7926     mq_open fonksiyonunda kuyruk özellikleri mq_attr parametresinde
7927     belirtilebilir. mq_open fonksiyonu bu yapıdaki yalnızca mq_maxmsg ve
7928     mq_msgsize
7929     elemanlarını dikkate almaktadır. Kuruk yaratıldıktan sonra ise yalnızca
7930     mq_setattr ile yapının mq_flags elemanı değiştirilebilmektedir.
7931     Ancak kuyruktaki maksimum mesaj sayısı mevcut Linux sistemlerinde
7932     default durumda en fazla 10 yapılmaktadır. Böylece biz mesaj
7933     büyüğünü değiştirip tampon alanımızı ona uygun oluşturabiliriz.
7934     Çekirdek tarafından izin verilen maksimum değerler
7935     /proc/sys/fs/mqueue dizinin içerisindeki dosyalarda belirtilemektedir.
7936     Etkin user id'si 0 olan prosesler bu dosyadaki limitlere takılmazlar.
7937
7938 -----*/
7939
7940 /* proc1.c */
7941
7942 #include <stdio.h>
7943 #include <stdlib.h>
7944 #include <string.h>
7945 #include <ctype.h>
7946 #include <sys/stat.h>
7947 #include <mqueue.h>
7948
7949 #define MSG_QUEUE_PATH      "/this_is_a_message_queue_last"
7950 #define MAX_MSG_LEN          1024
7951 #define MAX_MSG              10
7952
7953 void exit_sys(const char *msg);
7954
7955 int main(void)
7956 {
```

```
7950     mqd_t mq;
7951     char buf[MAX_MSG_LEN];
7952     struct mq_attr attr;
7953     int prio;
7954     int ch;
7955     char *str;
7956
7957     attr.mq_maxmsg = MAX_MSG;
7958     attr.mq_msgsize = MAX_MSG_LEN;
7959
7960     if ((mq = mq_open(MSG_QUEUE_PATH, O_CREAT|O_WRONLY, S_IRUSR|S_IWUSR|    ↵
7961           S_IRGRP|S_IROTH, &attr)) == -1)
7962         exit_sys("mq_open");
7963
7964     if (mq_getattr(mq, &attr) == -1)
7965         exit_sys("mq_getattr");
7966
7967     printf("Default attributes of the queue:\n");
7968     printf("Max msg: %ld\n", attr.mq_maxmsg);
7969     printf("Max msg size: %ld\n", attr.mq_msgsize);
7970     printf("Number of messages in queue: %ld\n", attr.mq_curmsgs);
7971     printf("-----\n");
7972
7973     for (;;) {
7974         printf("Bir öncelik derecesi ve yanına bir yazı giriniz:");
7975         scanf("%d", &prio);
7976         while ((ch = getchar()), isspace(ch))
7977             ;
7978         ungetc(ch, stdin);
7979         fgets(buf, MAX_MSG_LEN, stdin);
7980         if ((str = strchr(buf, '\n')) != NULL)
7981             *str = '\0';
7982
7983         if (mq_send(mq, buf, strlen(buf) + 1, prio))
7984             exit_sys("mq_send");
7985
7986         if (!strcmp(buf, "quit"))
7987             break;
7988     }
7989
7990     mq_close(mq);
7991
7992     if (mq_unlink(MSG_QUEUE_PATH) == -1)
7993         exit_sys("mq_unlink");
7994
7995     return 0;
7996 }
7997 void exit_sys(const char *msg)
7998 {
7999     perror(msg);
8000
8001     exit(EXIT_FAILURE);
```

```
8002  }
8003
8004 /* proc2.c */
8005
8006 #include <stdio.h>
8007 #include <stdlib.h>
8008 #include <string.h>
8009 #include <sys/stat.h>
8010 #include <mqueue.h>
8011
8012 #define MSG_QUEUE_PATH      "/this_is_a_message_queue_last"
8013 #define MAX_MSG_LEN          1024
8014
8015 void exit_sys(const char *msg);
8016
8017 int main(void)
8018 {
8019     mqd_t mq;
8020     char buf[MAX_MSG_LEN];
8021     unsigned int prio;
8022     ssize_t result;
8023     struct mq_attr attr;
8024
8025     if ((mq = mq_open(MSG_QUEUE_PATH, O_RDONLY)) == -1)
8026         exit_sys("mq_open");
8027
8028     if (mq_getattr(mq, &attr) == -1)
8029         exit_sys("mq_getattr");
8030
8031     printf("Default attributes of the queue:\n");
8032     printf("Max msg: %ld\n", attr.mq_maxmsg);
8033     printf("Max msg size: %ld\n", attr.mq_msgsize);
8034     printf("Number of messages in queue: %ld\n", attr.mq_curmsgs);
8035     printf("-----\n");
8036
8037     for (;;) {
8038         if ((result = mq_receive(mq, buf, MAX_MSG_LEN, &prio)) == -1)
8039             exit_sys("mq_receive");
8040         printf("%ld bytes received at priority %u: \"%s\"\n", (long)result, prio, buf);
8041         if (!strcmp(buf, "quit"))
8042             break;
8043     }
8044
8045     mq_close(mq);
8046
8047     return 0;
8048 }
8049
8050 void exit_sys(const char *msg)
8051 {
8052     perror(msg);
8053 }
```

```
8054     exit(EXIT_FAILURE);
8055 }
8056
8057 /
*-----*
-----
8058 Sistem Limitleri: POSIX standartlarında sisteme ilişkin çeşitli limit      ↵
8059 değerler <limits.h> dosyası içerisinde sembolik sabitler      ↵
8060 olarak bildirilmiştir. <limits.h> içerisinde sembolik sabitler çeşitli      ↵
8061 başlıklar altında gruplandırılmıştır. Bu başlıklar ve anlam      ↵
8062ları şöyledir:
8063
8064 Runtime Invariant Values (Possibly Indeterminate): Buradaki sembolik      ↵
8065 sabitlerin değeri ilgili sistemde değişmemektedir. Eğer      ↵
8066 buradaki değerler _POSIX_XXX ile belirtilen minimum değerlerden büyük fakat      ↵
8067 belirsiz (unspecified) ise bu sembolik sabitler      ↵
8068 <limits.h> içerisinde bulunmamak zorundadır. Bu durumda bunların değerlerini      ↵
8069 almak için sysconf fonksiyonu çağrılmalıdır. Fakat      ↵
8070 sysconf fonksiyonunun bu değerler için vereceği değerler aynıdır. Burada      ↵
8071 belirsiz demekle ilgili değerin bazı donanım ve sistem      ↵
8072 özelliklerine göre derleme zamanın belirlenemiyor olması anlatılmaktadır.
8073
8074 Pathname Variable Values: Buradaki sembolik sabit değerleri eğer      ↵
8075 _POSIX_XXX'ten büyük fakat dosya sisteme bağlı olarak değişiyorsa      ↵
8076 <limits.h> içerisinde bulunmamak zorundadır. Bu durumda bunların gerçek      ↵
8077 değerleri pathconf ya da fpathconf fonksiyonlarıyla alınmalıdır.
8078
8079 Runtime Increasable Values: Buradaki sembolik sabitlerin hepsi <limits.h>      ↵
8080 içerisinde define edilmek zorundadır. Fakat <limits.h> içerisinde      ↵
8081 define edilmiş değerler o sistemdeki minimum değerlerdir. Çünkü bu değerler      ↵
8082 çeşitli biçimlerde (örneğin setrlimit fonksiyonu ile)
8083 artırılmış olabilmektedir. O sistemdeki o andaki gerçek değerler yine      ↵
8084 sysconf fonksiyonuyla elde edilmelidir.
8085
8086 0 halde belli bir özellik eğer "Runtime Invariant Values" ya da "Pathname      ↵
8087 Variable Values" grububdaysa önce ifdef ile ilgili      ↵
8088 sembolik sabitin define edilmiş olup olmadığına bakılmalı eğer bu sembolik      ↵
8089 sabit define edilmişse o değeri almalı, define edilmemişse      ↵
8090 sysconf ya da pathconf/fpathconf çağrıması yapılmalıdır. "Runtime      ↵
8091 Increasable Values" lar için doğrudan o değer alınabilir ya da sysconf      ↵
8092 ile o andaki gerçek değerler alınabilir. Herhangi bir özelliği elde etmek      ↵
8093 için bir fonksiyon yazmak iyi bir teknik olabilir.
8094
8095 sysconf fonksiyonu parametre olarak _POSIX_XXX için _SC_XXX değerlerini      ↵
8096 kullanmaktadır. Fonksiyon başarısız olursa -1 değerine      ↵
8097 geri döner ve errno EINVAL olarak set edilir. Başarı durumunda ilgili limite      ↵
8098 geri döner. Değerin belirlenememesi (indeterminate)
8099 durumunda ise sysconf -1'e geri döner errno değerini değiştirmez. Genellikle      ↵
8100 programcılar değer belirlenememişse yüksek bir değer
8101 uydururlar.
8102 -----
8103 -----*/
8104
```

```
8086 #include <stdio.h>
8087 #include <stdlib.h>
8088 #include <unistd.h>
8089 #include <errno.h>
8090 #include <limits.h>
8091
8092 void exit_sys(const char *msg);
8093
8094 long child_max(void)
8095 {
8096     static long result = 0;
8097
8098 #define CHILD_MAX_INDETERMINATE_GUESS      4096
8099
8100 #ifdef CHILD_MAX
8101     result = CHILD_MAX;
8102 #else
8103     if (result == 0) {
8104         errno = 0;
8105         if ((result = sysconf(_SC_CHILD_MAX)) == -1 && errno == 0)
8106             result = CHILD_MAX_INDETERMINATE_GUESS;
8107     }
8108 #endif
8109
8110     return result;
8111 }
8112
8113 int main(void)
8114 {
8115     long cmax;
8116
8117     if ((cmax = child_max()) == -1)
8118         exit_sys("child_max");
8119
8120     printf("Child max = %ld\n", cmax);
8121
8122     return 0;
8123 }
8124
8125 void exit_sys(const char *msg)
8126 {
8127     perror(msg);
8128
8129     exit(EXIT_FAILURE);
8130 }
8131
8132 /
*-----*
-----*
8133 Bir dosyanın maksimum ismi ne olabilir? Eğer NAME_MAX sembolik sabiti      ↗
     <limits.h> içerisinde bildirilmişse onu almalıyız,
8134 bildirilmemişse pathconf ile asıl değeri elde etmeliyiz. pathconf bizden ↗
     ilgilendiğimiz dosya sisteme ilişkin
```

```
8135     bir dizin'in yol ifadesini de parametre olarak almaktadır. pathconf ve ↵
8136         fpathconf fonksiyonlarının ikinci parametreleri
8137     _POSIX_XXX için _PC_XXX biçimindeki sembolik sabitlerdir.
8138     -----
8139 #include <stdio.h>
8140 #include <stdlib.h>
8141 #include <unistd.h>
8142 #include <errno.h>
8143 #include <limits.h>
8144
8145 void exit_sys(const char *msg);
8146
8147 long name_max(const char *dirpath)
8148 {
8149     static long result = 0;
8150
8151 #define NAME_MAX_INDETERMINATE_GUESS      1024
8152
8153 #ifdef NAME_MAX
8154     result = NAME_MAX;
8155 #else
8156     if (result == 0) {
8157         errno = 0;
8158         if ((result = (pathconf(dirpath, _PC_NAME_MAX))) == -1 && errno == 0)
8159             result = NAME_MAX_INDETERMINATE_GUESS;
8160     }
8161 #endif
8162
8163     return result;
8164 }
8165
8166 int main(void)
8167 {
8168     long nmax;
8169
8170     if ((nmax = name_max(".")) == -1)
8171         exit_sys("name_max");
8172
8173     printf("Name max = %ld\n", nmax);
8174
8175     return 0;
8176 }
8177
8178 void exit_sys(const char *msg)
8179 {
8180     perror(msg);
8181
8182     exit(EXIT_FAILURE);
8183 }
8184
```

```
8185  /
8186  *-----*
8186      Bir yol ifadesini yerleştireceğimiz char türden dizinin dinamik tahsis  ↵
8186      edilmesi örneği (Stevens kitabında eski POSIX versiyon
8187     larını da dikkate almış. Fakat artık gereksiz olabilmektedir.)  ↵
8188  -----*/
8189
8190 #include <stdio.h>
8191 #include <stdlib.h>
8192 #include <unistd.h>
8193 #include <errno.h>
8194 #include <limits.h>
8195
8196 void exit_sys(const char *msg);
8197
8198 long path_max(void)
8199 {
8200     static long result = 0;
8201
8202 #define PATH_MAX_INDETERMINATE_GUESS      4096
8203
8204 #ifdef PATH_MAX
8205     result = PATH_MAX;
8206 #else
8207     if (result == 0) {
8208         errno = 0;
8209         if ((result = pathconf("/", _PC_PATH_MAX)) == -1 && errno == 0)
8210             result = PATH_MAX_INDETERMINATE_GUESS;
8211     }
8212 #endif
8213
8214     return result;
8215 }
8216
8217 int main(void)
8218 {
8219     long pmax;
8220     char *path;
8221
8222     if ((pmax = path_max()) == -1)
8223         exit_sys("path_max");
8224
8225     printf("Path max = %ld\n", pmax);
8226
8227     if ((path = (char *)malloc((size_t)(pmax))) == NULL)
8228         exit_sys("malloc");
8229
8230     free(path);
8231
8232     return 0;
8233 }
```

```
8234
8235 void exit_sys(const char *msg)
8236 {
8237     perror(msg);
8238
8239     exit(EXIT_FAILURE);
8240 }
8241
8242 /
*-----*/  

-----*
-----*
```

8243 Prosesin kaynak liitlerini elde etmek için getrlimit fonksiyonu
kullanılır. Bu fonksiyonun birinci parametresi hangi kaynağın
elde edileceğini belirtken RLIMIT_XXXX değeridir. İkinci parametresi ise
struct rlimit türünden bir yapının adresidir.

8244 Prosesin her kaynağının "soft limit" ve "hard limit" denilen iki eşik
değeri vardır. İşletim sistemi "soft limit" kontrolü yapmaktadır.

8245 Ancak programcı setrlimit fonksiyonu ile soft limiti hard limite kadar
yükselebilir. Hard limiti ise ancak etkin kullanıcı id'si
0 olan prosesler ya da ilgili yeteneğe (capability) sahip prosesler
yükselebilirler. Örneğin aşağıdaki programda prosesin açabileceği
maksimum dosya sayısının soft ve hard limitleri getrlimit fonksiyonuyla
elde edilip yazdırılmıştır. Buradan 1024 ve 4096 değerleri elde
edilmiştir.

8246 Yani biz prosesimizde ancak 1024 tane dosya açabiliriz. Ancak bunu
setrlimit ile ancak 4096'ya çıkartabiliriz.

```
8247 -----*/
8248
8249 #include <stdio.h>
8250 #include <stdlib.h>
8251 #include <unistd.h>
8252 #include <sys/resource.h>
8253
8254 void exit_sys(const char *msg);
8255
8256 int main(void)
8257 {
8258     struct rlimit rlim;
8259
8260     if (getrlimit(RLIMIT_NOFILE, &rlim) == -1)
8261         exit_sys("getrlimit");
8262
8263     printf("Soft limit: %lu\n", (unsigned long)rlim.rlim_cur);
8264     printf("Hard limit: %lu\n", (unsigned long)rlim.rlim_max);
8265
8266     return 0;
8267 }
8268
8269 void exit_sys(const char *msg)
8270 {
8271     perror(msg);
8272 }
```

```
8276     exit(EXIT_FAILURE);
8277 }
8278
8279 /
*-----*
-----*
8280     Biz bir proses limitini setrlimit fonksiyonu ile ancak hard limit kadar  -->
8281     yükseltebiliriz. Onun ötesine yükseltebilmemiz için  -->
8282     hard limiti yükseltmemiz gereklidir. İşte setrlimit fonksiyonu ile hard  -->
8283     limiti yükseltebilmemiz için bizim root prosesi olmamız  -->
8284     gereklidir. Aşağıdaki programda prosesin açabileceği dosya sayısı 1024'ten  -->
8285     hard limit olan 4096'ya yükseltilmiştir. Tabii proses
8286     setrlimit fonksiyonu ile kendi hard limitini düşürebilir.
8287 -----*/
8288
8289 #include <stdio.h>
8290 #include <stdlib.h>
8291 #include <fcntl.h>
8292 #include <unistd.h>
8293 #include <sys/resource.h>
8294
8295 void exit_sys(const char *msg);
8296
8297 int main(void)
8298 {
8299     struct rlimit rlim;
8300     int i;
8301     int fd;
8302
8303     if (getrlimit(RLIMIT_NOFILE, &rlim) == -1)
8304         exit_sys("getrlimit");
8305
8306     printf("Soft limit: %lu\n", (unsigned long)rlim.rlim_cur);
8307     printf("Hard limit: %lu\n", (unsigned long)rlim.rlim_max);
8308
8309     for (i = 0;; ++i) {
8310         if ((fd = open("sample.c", O_RDONLY)) == -1)
8311             break;
8312         printf("%d ", fd);
8313         fflush(stdout);
8314     }
8315     printf("\n");
8316
8317     rlim.rlim_cur = 4096;
8318     if (setrlimit(RLIMIT_NOFILE, &rlim) == -1)
8319         exit_sys("setrlimit");
8320
8321     for (i = 0;; ++i) {
8322         if ((fd = open("sample.c", O_RDONLY)) == -1)
```

```
8323         break;
8324     printf("%d ", fd);
8325     fflush(stdout);
8326 }
8327 printf("\n");
8328
8329     return 0;
8330 }
8331
8332 void exit_sys(const char *msg)
8333 {
8334     perror(msg);
8335
8336     exit(EXIT_FAILURE);
8337 }
8338
8339 /
*-----*
-----*
8340     Aşağıdaki örnekte prosesin açabileceği dosya sayısı hard limit ve soft    ↵
     limit yükseltilerek 8192'ye çıkartılmıştır.    ↵
8341     Tabii programın sudo ile çalıştırılması gereklidir.    ↵
8342 -----*/*
8343
8344 #include <stdio.h>
8345 #include <stdlib.h>
8346 #include <fcntl.h>
8347 #include <unistd.h>
8348 #include <sys/resource.h>
8349
8350 void exit_sys(const char *msg);
8351
8352 int main(void)
8353 {
8354     struct rlimit rlim;
8355     int i;
8356     int fd;
8357
8358     if (getrlimit(RLIMIT_NOFILE, &rlim) == -1)
8359         exit_sys("getrlimit");
8360
8361     printf("Soft limit: %lu\n", (unsigned long)rlim.rlim_cur);
8362     printf("Hard limit: %lu\n", (unsigned long)rlim.rlim_max);
8363
8364     rlim.rlim_cur = 8192;
8365     rlim.rlim_max = 8192;
8366     if (setrlimit(RLIMIT_NOFILE, &rlim) == -1)
8367         exit_sys("setrlimit");
8368
8369     for (i = 0;; ++i) {
8370         if ((fd = open("sample.c", O_RDONLY)) == -1)
8371             break;
```

```
8372         printf("%d ", fd);
8373         fflush(stdout);
8374     }
8375     printf("\n");
8376
8377     return 0;
8378 }
8379
8380 void exit_sys(const char *msg)
8381 {
8382     perror(msg);
8383
8384     exit(EXIT_FAILURE);
8385 }
8386
8387 /
*-----*
-----*
8388     komut satırındaki ulimit internal komutu -a ile shell prosesinin tüm
8389     soft limitlerini göstermektedir. Soft limitler -S ile hard
8390     limitler -H ile gösterilir. Aynı zamanda bu limitler değiştirile de
8391     bilmektedir. Prosesin tüm limitleri fork işlemi
8392     sırasında üst prosten alt prosese aktarılmaktadır. Yani biz örneğin
8393     kabuk üzerinde aşağıdaki gibi kabuğun açabilecegi
8394     dosya sayısının soft limitini değiştirebiliriz:
8395
8396     ulimit -S -n 4096
8397
8398     Şimdi artık kabuktan çalıştıracağımız programlar bu limiti
8399     kullanacaktır.
8400
8401     Proseslerin soft ve hard limitlerini kalıcı hale getirmek için Linux
8402     sistemlerinde /etc/security/limits.conf dosyası
8403     kullanılmaktadır. Bu dosyanın formatını ilgili docümanalardan
8404     inceleyiniz. Ancak bu dosyada spesifik bir kullanıcı için
8405     ve bir gruba dahil tüm kullanıcılar için limitler belirlenebilmektedir.
8406     Tabii bu belirlemeler sistem reboot edilince
8407     etki gösterir. Örneğin csd kullanıcısının proseslerinin açabileceği
8408     dosya syısını 8192 yapabilmek için şu satırları dosyaya
8409     eklemeliyiz:
8410
8411     csd    hard    nofile    8192
8412     csd    soft    nofile    8192
8413
8414 -----*/
8415
8416 /
*-----*
-----*
8417     POSIX thread fonksiyonlarına pthread kütüphanesi denilmektedir. Tüm
8418     thread fonksiyonları pthread_xxx biçiminde isimlendirilmiştir.
8419     pthread_create fonksiyonu thread'i yaratır ve çalıştırır. pthread_create
```

```
fonksiyonundan çıktıktan sonra artık prosesin
8411 iki bağımsız çizelgelenen akışı olacaktır. Thread fonksiyonlarının geri ↵
     dönüş değerleir ve parametreleri void * türünden
8412 olmak zorundadır. Thread fonksiyonlarının büyük bölümü int geri dönüş ↵
     değerine sahiptir. Bu fonksiyonlar başarı durumunda 0 değerine,
8413 başarısızlık durumunda errno değerinin kendisine geri dönerler. errno ↵
     değerini set etmezler.
8414 -----
8415 -----
8416 #include <stdio.h>
8417 #include <stdlib.h>
8418 #include <string.h>
8419 #include <unistd.h>
8420 #include <pthread.h>
8421
8422 void *thread_proc(void *param);
8423
8424 int main(void)
8425 {
8426     int result;
8427     pthread_t tid;
8428     int i;
8429
8430     if ((result = pthread_create(&tid, NULL, thread_proc, NULL)) != 0) {
8431         fprintf(stderr, "pthread_create: %s\n", strerror(result));
8432         exit(EXIT_FAILURE);
8433     }
8434
8435     for (int i = 0; i < 10; ++i) {
8436         printf("main thread\n");
8437         sleep(1);
8438     }
8439
8440     return 0;
8441 }
8442
8443 void *thread_proc(void *param)
8444 {
8445     for (int i = 0; i < 10; ++i) {
8446         printf("other thread\n");
8447         sleep(1);
8448     }
8449
8450     return NULL;
8451 }
8452
8453 /
*-----*
----- Thread fonksiyonları başarılı olduğunda hata mesajını ekrana yazdırıp ↵
     prosesi sonlandıran yardımcı bir exit_sys_thread ↵
     fonksiyonu kullanacağız.
```

```
8456 -----*/  
8457  
8458 #include <stdio.h>  
8459 #include <stdlib.h>  
8460 #include <string.h>  
8461 #include <unistd.h>  
8462 #include <pthread.h>  
8463  
8464 void exit_sys_thread(const char *msg, int err);  
8465 void *thread_proc(void *param);  
8466  
8467 int main(void)  
8468 {  
8469     int result;  
8470     pthread_t tid;  
8471     int i;  
8472  
8473     if ((result = pthread_create(&tid, NULL, thread_proc, NULL)) != 0)  
8474         exit_sys_thread("pthread_create", result);  
8475  
8476     for (int i = 0; i < 10; ++i) {  
8477         printf("main thread\n");  
8478         sleep(1);  
8479     }  
8480  
8481     return 0;  
8482 }  
8483  
8484 void exit_sys_thread(const char *msg, int err)  
8485 {  
8486     fprintf(stderr, "%s: %s\n", msg, strerror(err));  
8487     exit(EXIT_FAILURE);  
8488 }  
8489  
8490 void *thread_proc(void *param)  
8491 {  
8492     for (int i = 0; i < 10; ++i) {  
8493         printf("other thread\n");  
8494         sleep(1);  
8495     }  
8496  
8497     return NULL;  
8498 }  
8499  
8500 /-----*/  
8501 Thread fonksiyonuna geçirilecek argüman stack'teki bir nesnenin adresi  
8502 olmamalıdır. Argüman bir tamsayı ise void '*'a dönüştürülerek,  
8503 therad fonksiyonuna geçirilmeli oradan da yeniden tamsayı türüne  
8504 dönüştürülmelidir. Thread fonksiyonuna geçirilecek adresin  
8505 static ömürlü bir nesnenin adresi ya da heap'teki bir nesnenin adresi
```

```
olması gereklidir.
```

```
8504 -----* /-----  
8505  
8506 #include <stdio.h>  
8507 #include <stdlib.h>  
8508 #include <string.h>  
8509 #include <unistd.h>  
8510 #include <pthread.h>  
8511  
8512 void exit_sys_thread(const char *msg, int err);  
8513 void *thread_proc(void *param);  
8514  
8515 int main(void)  
8516 {  
8517     int result;  
8518     pthread_t tid;  
8519     int i;  
8520  
8521     if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)  
8522         exit_sys_thread("pthread_create", result);  
8523  
8524     for (int i = 0; i < 10; ++i) {  
8525         printf("main thread\n");  
8526         sleep(1);  
8527     }  
8528  
8529     return 0;  
8530 }  
8531  
8532 void exit_sys_thread(const char *msg, int err)  
8533 {  
8534     fprintf(stderr, "%s: %s\n", msg, strerror(err));  
8535     exit(EXIT_FAILURE);  
8536 }  
8537  
8538 void *thread_proc(void *param)  
8539 {  
8540     for (int i = 0; i < 10; ++i) {  
8541         printf("%s\n", (char *)param);  
8542         sleep(1);  
8543     }  
8544  
8545     return NULL;  
8546 }  
8547  
8548 /-----*-----  
8549  
8550 Eğer thread'e birden fazla parametre geçirilmek isteniyorsa bir yapı  
     bildirilmeli, bu yapı heap'te tahsis edilip, yapı nesnesinin adresi
```

```
geçirilmelidir.
```

```
8551 -----*/  
8552  
8553 #include <stdio.h>  
8554 #include <stdlib.h>  
8555 #include <string.h>  
8556 #include <unistd.h>  
8557 #include <pthread.h>  
8558  
8559 void exit_sys_thread(const char *msg, int err);  
8560 void *thread_proc(void *param);  
8561  
8562 struct THREAD_PARAM {  
8563     char name[64];  
8564     int val;  
8565 };  
8566  
8567 int main(void)  
8568 {  
8569     int result;  
8570     pthread_t tid;  
8571     int i;  
8572     struct THREAD_PARAM *thread_param;  
8573  
8574     if ((thread_param = (struct THREAD_PARAM *)malloc(sizeof(struct  
8575         THREAD_PARAM))) == NULL) {  
8576         fprintf(stderr, "cannot allocate memory!..\n");  
8577         exit(EXIT_FAILURE);  
8578     }  
8579     strcpy(thread_param->name, "Oter thread");  
8580     thread_param->val = 123;  
8581     if ((result = pthread_create(&tid, NULL, thread_proc, thread_param)) != 0)  
8582         exit_sys_thread("pthread_create", result);  
8583  
8584     for (int i = 0; i < 10; ++i) {  
8585         printf("main thread\n");  
8586         sleep(1);  
8587     }  
8588  
8589     return 0;  
8590 }  
8591  
8592 void exit_sys_thread(const char *msg, int err)  
8593 {  
8594     fprintf(stderr, "%s: %s\n", msg, strerror(err));  
8595     exit(EXIT_FAILURE);  
8596 }  
8597  
8598 void *thread_proc(void *param)  
8599 {
```

```
8600     struct THREAD_PARAM *thread_param = (struct THREAD_PARAM *)param;
8601
8602     for (int i = 0; i < 10; ++i) {
8603         printf("%s, %d\n", thread_param->name, thread_param->val);
8604         sleep(1);
8605     }
8606
8607     free(thread_param);
8608
8609     return NULL;
8610 }
8611
8612 /
*-----*
-----*
8613     main fonksiyonu bittiğinde exit fonksiyonuyla proses sonlandırıldığı
     için diğer thread'ler de exit sırasında yok edilmektedir.
8614 -----*/
8615
8616 #include <stdio.h>
8617 #include <stdlib.h>
8618 #include <string.h>
8619 #include <unistd.h>
8620 #include <pthread.h>
8621
8622 void exit_sys_thread(const char *msg, int err);
8623 void *thread_proc(void *param);
8624
8625 int main(void)
8626 {
8627     int result;
8628     pthread_t tid;
8629     int i;
8630
8631     if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
8632         exit_sys_thread("pthread_create", result);
8633
8634     for (int i = 0; i < 5; ++i) {
8635         printf("main thread: %d\n", i);
8636         sleep(1);
8637     }
8638
8639     return 0;
8640 }
8641
8642 void exit_sys_thread(const char *msg, int err)
8643 {
8644     fprintf(stderr, "%s: %s\n", msg, strerror(err));
8645     exit(EXIT_FAILURE);
8646 }
```

```
8648 void *thread_proc(void *param)
8649 {
8650     for (int i = 0; i < 10; ++i) {
8651         printf("%s: %d\n", (char *)param, i);
8652         sleep(1);
8653     }
8654
8655     return NULL;
8656 }
8657
8658 /
*-----*
-----*
8659     pthread_join fonksiyonu id'si ile verilen thread'in sonlanmasını bekler. ↵
     Onun exit kodunu alarak thread için ayrılan alanı yok eder.
8660 Thread'ler için de -prosesler kadar önemli olmasa da- hortlaklıktan ↵
     bahasedebiliriz. Yani normal olarak detached olmayan thread'leri
8661 pthread_join ile beklemeliyiz. pthread_join fonksiyonu proseslerde ↵
     kullandığımız wait fonksiyonlarına işlev olarak benzemektedir.
8662 Ancak pthread_join fonksiyonu herhangi bir thread akışı tarafından ↵
     prosesin herhangi bir thread'ini beklemek için kullanılabilir.
8663 -----*/ ↵
8664
8665 #include <stdio.h>
8666 #include <stdlib.h>
8667 #include <string.h>
8668 #include <unistd.h>
8669 #include <pthread.h>
8670
8671 void exit_sys_thread(const char *msg, int err);
8672 void *thread_proc(void *param);
8673
8674 int main(void)
8675 {
8676     int result;
8677     pthread_t tid;
8678     int i;
8679
8680     if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
8681         exit_sys_thread("pthread_create", result);
8682
8683     for (int i = 0; i < 5; ++i) {
8684         printf("main thread: %d\n", i);
8685         sleep(1);
8686     }
8687
8688     if ((result = pthread_join(tid, NULL)) != 0)
8689         exit_sys_thread("pthread_join", result);
8690
8691     return 0;
8692 }
```

```
8693
8694 void exit_sys_thread(const char *msg, int err)
8695 {
8696     fprintf(stderr, "%s: %s\n", msg, strerror(err));
8697     exit(EXIT_FAILURE);
8698 }
8699
8700 void *thread_proc(void *param)
8701 {
8702     for (int i = 0; i < 10; ++i) {
8703         printf("%s: %d\n", (char *)param, i);
8704         sleep(1);
8705     }
8706
8707     return NULL;
8708 }
8709
8710 /
*-----*
-----*
-----*
-----*
```

8711 thread'lerin exit kodlarının void * türünden olduğuna dikkat ediniz. →
 Eğer int bir exit kodu vermek istiyorsanız int değeri
8712 void * türüne dönüştürmelisiniz. (Yani int değeri sanki bir adresmiş
 gibi vermelisiniz.) →

```
-----*/
```

8714
8715 #include <stdio.h>
8716 #include <stdlib.h>
8717 #include <string.h>
8718 #include <unistd.h>
8719 #include <pthread.h>
8720
8721 void exit_sys_thread(const char *msg, int err);
8722 void *thread_proc(void *param);
8723
8724 int main(void)
8725 {
8726 int result;
8727 pthread_t tid;
8728 int i;
8729 void *retval;
8730
8731 if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
8732 exit_sys_thread("pthread_create", result);
8733
8734 for (int i = 0; i < 5; ++i) {
8735 printf("main thread: %d\n", i);
8736 sleep(1);
8737 }
8738
8739 if ((result = pthread_join(tid, &retval)) != 0)

```
8740         exit_sys_thread("pthread_join", result);
8741
8742     printf("Thread exited with %ld\n", (long)retval);
8743
8744     return 0;
8745 }
8746
8747 void exit_sys_thread(const char *msg, int err)
8748 {
8749     fprintf(stderr, "%s: %s\n", msg, strerror(err));
8750     exit(EXIT_FAILURE);
8751 }
8752
8753 void *thread_proc(void *param)
8754 {
8755     for (int i = 0; i < 10; ++i) {
8756         printf("%s: %d\n", (char *)param, i);
8757         sleep(1);
8758     }
8759
8760     return (void *)123;
8761 }
8762
8763 /
*-----*
-----*
8764 Bir thread'in exit kodunu almak istemiyorsak onu "joinable" durumdan    ↗
     çıkartıp "detach" duruma sokmamız gereklidir. Therad'i
8765 detach duruma sokmak için pthread_detach fonksiyonu kullanılmaktadır. Bu    ↗
     işlem thread yaratılırken thread attribute bilgisiyle de
8766 yapılmaktadır. Detach duruma sokulmuş bir thread sonlandığında      ↗
     işletim sistemi thread'in tüm kaynaklarını yok edecektir.
8767 Tabii detach bir thread pthread_join fonksiyonu ile beklenemez. Bu    ↗
     durumda pthread_join fonksiyonu başarısız olur.
8768 -----*/
```

8769
8770 #include <stdio.h>
8771 #include <stdlib.h>
8772 #include <string.h>
8773 #include <unistd.h>
8774 #include <pthread.h>
8775
8776 void exit_sys_thread(const char *msg, int err);
8777 void *thread_proc(void *param);
8778
8779 int main(void)
8780 {
8781 int result;
8782 pthread_t tid;
8783 int i;
8784 void *retval;
8785 }

```
8786     if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
8787         exit_sys_thread("pthread_create", result);
8788
8789     if ((result = pthread_detach(tid)) != 0)          /* thread detach */
8790         exit_sys_thread("pthread_detach", result);
8791
8792     for (int i = 0; i < 5; ++i) {
8793         printf("main thread: %d\n", i);
8794         sleep(1);
8795     }
8796
8797     if ((result = pthread_join(tid, &retval)) != 0)    /* detach duruma */
8798         exit_sys_thread("pthread_join", result);
8799
8800     printf("Thread exited with %ld\n", (long)retval);
8801
8802     return 0;
8803 }
8804
8805 void exit_sys_thread(const char *msg, int err)
8806 {
8807     fprintf(stderr, "%s: %s\n", msg, strerror(err));
8808     exit(EXIT_FAILURE);
8809 }
8810
8811 void *thread_proc(void *param)
8812 {
8813     for (int i = 0; i < 10; ++i) {
8814         printf("%s: %d\n", (char *)param, i);
8815         sleep(1);
8816     }
8817
8818     return (void *)123;
8819 }
8820
8821 /
-----*
-----*
8822     Prosesin herhangi bir thread'i prosesin herhangi bir thread'ini
8823     pthread_cancel fonksiyonuyla sonlandırabilir. Ancak thread'in
8824     sonlanabilmesi için özel bazı POSIX donksiyonlarının içerisine girmiş
8825     olması gereklidir. Bu POSIX fonksiyonlarına "thread cancellation points"
8826     denilmektedir.
-----*/
```

8827 #include <stdio.h>
8828 #include <stdlib.h>
8829 #include <string.h>
8830 #include <unistd.h>

```
8831 #include <pthread.h>
8832
8833 void exit_sys_thread(const char *msg, int err);
8834 void *thread_proc(void *param);
8835
8836 int main(void)
8837 {
8838     int result;
8839     pthread_t tid;
8840     int i;
8841
8842     if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
8843         exit_sys_thread("pthread_create", result);
8844
8845     printf("Press ENTER to cancel...\n");
8846     getchar();
8847
8848     if ((result = pthread_cancel(tid)) != 0)
8849         exit_sys_thread("pthread_cancel", result);
8850
8851     if ((result = pthread_join(tid, NULL)) != 0)
8852         exit_sys_thread("pthread_join", result);
8853
8854     printf("other thread cancelled!\n");
8855
8856     for (int i = 0; i < 10; ++i) {
8857         printf("main thread: %d\n", i);
8858         sleep(1);
8859     }
8860
8861     return 0;
8862 }
8863
8864 void exit_sys_thread(const char *msg, int err)
8865 {
8866     fprintf(stderr, "%s: %s\n", msg, strerror(err));
8867     exit(EXIT_FAILURE);
8868 }
8869
8870 void *thread_proc(void *param)
8871 {
8872     for (int i = 0; i < 10; ++i) {
8873         printf("%s: %d\n", (char *)param, i);
8874         sleep(1);
8875     }
8876
8877     return NULL;
8878 }
8879
8880 /
```

```
8881     Eğer pthread_cancel uygulanan thread cancellation point olan bir POSIX  ↵
        fonksiyonuna girmiyorsa sonlandırma yapılamaz.
8882 -----*/
8883
8884 #include <stdio.h>
8885 #include <stdlib.h>
8886 #include <string.h>
8887 #include <unistd.h>
8888 #include <pthread.h>
8889
8890 void exit_sys_thread(const char *msg, int err);
8891 void *thread_proc(void *param);
8892
8893 int main(void)
8894 {
8895     int result;
8896     pthread_t tid;
8897
8898     if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
8899         exit_sys_thread("pthread_create", result);
8900
8901     printf("Press ENTER to cancel...\n");
8902     getchar();
8903
8904     if ((result = pthread_cancel(tid)) != 0)
8905         exit_sys_thread("pthread_cancel", result);
8906
8907     if ((result = pthread_join(tid, NULL)) != 0)
8908         exit_sys_thread("pthread_join", result);
8909
8910     printf("other thread ends...\n");
8911
8912     return 0;
8913 }
8914
8915 void exit_sys_thread(const char *msg, int err)
8916 {
8917     fprintf(stderr, "%s: %s\n", msg, strerror(err));
8918     exit(EXIT_FAILURE);
8919 }
8920
8921 void *thread_proc(void *param)
8922 {
8923     long i;
8924
8925     for (i = 0; i < 10000000000; ++i)
8926         ;
8927
8928     return NULL;
8929 }
8930
```

```
8931 /  
8932 *-----  
8932     Eğer thread pthread_cancel ile sonlandırılmışsa pthread_join  
8932         fonksiyonundan thread exit kodu olarak PTHREAD_CANCELED  
8933     özel değeri elde edilir. (Bu değer void * türündendir.)  
8934 -----*/  
8935  
8936 #include <stdio.h>  
8937 #include <stdlib.h>  
8938 #include <string.h>  
8939 #include <unistd.h>  
8940 #include <pthread.h>  
8941  
8942 void exit_sys_thread(const char *msg, int err);  
8943 void *thread_proc(void *param);  
8944  
8945 int main(void)  
8946 {  
8947     int result;  
8948     pthread_t tid;  
8949     void *retval;  
8950  
8951     if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)  
8952         exit_sys_thread("pthread_create", result);  
8953  
8954     printf("Press ENTER to cancel...\n");  
8955     getchar();  
8956  
8957     if ((result = pthread_cancel(tid)) != 0)  
8958         exit_sys_thread("pthread_cancel", result);  
8959  
8960     if ((result = pthread_join(tid, &retval)) != 0)  
8961         exit_sys_thread("pthread_join", result);  
8962  
8963     if (retval == PTHREAD_CANCELED)  
8964         printf("other thread canceled!...\n");  
8965     else  
8966         printf("other thread finished normally...\n");  
8967  
8968     return 0;  
8969 }  
8970  
8971 void exit_sys_thread(const char *msg, int err)  
8972 {  
8973     fprintf(stderr, "%s: %s\n", msg, strerror(err));  
8974     exit(EXIT_FAILURE);  
8975 }  
8976  
8977 void *thread_proc(void *param)  
8978 {
```

```
8979     int i;
8980
8981     for (i = 0; i < 10; ++i) {
8982         printf("%s: %d\n", (char *)param, i);
8983         sleep(1);
8984     }
8985
8986     return NULL;
8987 }
8988
8989 /
*-----*
-----*
8990     Aşağıdaki örnekte bir döngü içerisinde 10 thread yaratılmış ve bu 10
8991         thread'in sonlanması beklenmiştir. Tüm thread'ler aynı
8992         fonksiyondan farklı parametreler alarak çalışmaya başlamaktadır.
8993 -----*/
8994 #include <stdio.h>
8995 #include <stdlib.h>
8996 #include <string.h>
8997 #include <unistd.h>
8998 #include <pthread.h>
8999
9000 #define MAX_THREAD      10
9001
9002 void exit_sys_thread(const char *msg, int err);
9003 void *thread_proc(void *param);
9004
9005 int main(void)
9006 {
9007     int result;
9008     pthread_t tids[MAX_THREAD];
9009     int i;
9010     char *arg;
9011
9012     for (i = 0; i < MAX_THREAD; ++i) {
9013         if ((arg = (char *)malloc(32)) == NULL) {
9014             fprintf(stderr, "cannot allocate memory!..\n");
9015             exit(EXIT_FAILURE);
9016         }
9017         sprintf(arg, "Thread-%d", (i + 1));
9018         if ((result = pthread_create(&tids[i], NULL, thread_proc, arg)) != 0)
9019             exit_sys_thread("pthread_create", result);
9020     }
9021
9022     for (i = 0; i < MAX_THREAD; ++i)
9023         if ((result = pthread_join(tids[i], NULL)) != 0)
9024             exit_sys_thread("pthread_join", result);
9025
9026     return 0;

```

```
9027 }
9028
9029 void exit_sys_thread(const char *msg, int err)
9030 {
9031     fprintf(stderr, "%s: %s\n", msg, strerror(err));
9032     exit(EXIT_FAILURE);
9033 }
9034
9035 void *thread_proc(void *param)
9036 {
9037     int i;
9038
9039     for (i = 0; i < 10; ++i) {
9040         printf("%s: %d\n", (char *)param, i);
9041         sleep(1);
9042     }
9043
9044     free(param);
9045
9046     return NULL;
9047 }
9048
9049 /
*-----*-----*-----*
```

9050 Thedad'in çeşitli özellikleri (attributes) vardır. Thread özellikleri
9051 pthead_attr_t türüyle temsil edilmiştir. Bu tür
9052 pek çok sistemde bir yapı belirtir. Özellik set etmek için önce
9053 pthead_attr_init fonksiyonu ile nesnenin initialize edilmesi gereklidir.

9054 Özellik nesnesinin elemanlarını set etmek için ise bir grup
9055 pthead_attr_setxxx isimli POSIX fonksiyonu bulunmaktadır.
9056 Örneğin thread'in detached mi yoksa joinable mi olacağını belirlemek
9057 için pthead_attr_setdetachstate fonksiyonu,
9058 thread'in stack uzunluğunu belirlemek için (Linux'ta default 8 MB)
9059 bu özellik nesnesinin pthead_destroy ile yok edilmesi gerekmektedir.
9060 (pthead_create fonksiyonu bu nesneyi kopya yoluyla kullanır.
9061 Yani pthead_destroy işlemi hemen pthead_create fonksiyonundan sonra da
9062 yapılabılır.

```
9063 -----*/
9064
9065 #include <stdio.h>
9066 #include <stdlib.h>
9067 #include <string.h>
9068 #include <unistd.h>
9069 #include <pthread.h>
```

```
9069 int main(void)
9070 {
9071     int result;
9072     pthread_t tid;
9073     pthread_attr_t attr;
9074
9075     if ((result = pthread_attr_init(&attr)) != 0)
9076         exit_sys_thread("pthread_attr_init", result);
9077
9078     if ((result = pthread_attr_setstacksize(&attr, 65536)) != 0)
9079         exit_sys_thread("pthread_attr_setstacksize", result);
9080
9081     if ((result = pthread_attr_setdetachstate(&attr,
9082                                              PTHREAD_CREATE_DETACHED)) != 0)
9083         exit_sys_thread("pthread_attr_setdetachstate", result); ↗
9084
9085     if ((result = pthread_create(&tid, &attr, thread_proc, "other
9086                                 thread")) != 0)
9087         exit_sys_thread("pthread_create", result); ↗
9088
9089     pthread_attr_destroy(&attr);
9090
9091     printf("Press ENTER to exit...\n");
9092     getchar();
9093
9094 }
9095 void exit_sys_thread(const char *msg, int err)
9096 {
9097     fprintf(stderr, "%s: %s\n", msg, strerror(err));
9098     exit(EXIT_FAILURE);
9099 }
9100
9101 void *thread_proc(void *param)
9102 {
9103     int i;
9104
9105     for (i = 0; i < 10; ++i) {
9106         printf("%s: %d\n", (char *)param, i);
9107         sleep(1);
9108     }
9109
9110     return NULL;
9111 }
9112
9113 / ↗
----- ↗
9114     pthread_self fonksiyonu hangi thread akışında çağrılırsa o thread'in
9115         kendi thread id'si elde edilir. ↗
-----*/ ↗
```

```
9116
9117
9118 #include <stdio.h>
9119 #include <stdlib.h>
9120 #include <string.h>
9121 #include <unistd.h>
9122 #include <pthread.h>
9123
9124 void exit_sys_thread(const char *msg, int err);
9125 void *thread_proc(void *param);
9126
9127 int main(void)
9128 {
9129     int result;
9130     pthread_t tid;
9131
9132     if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
9133         exit_sys_thread("pthread_create", result);
9134
9135     if ((result = pthread_join(tid, NULL)) != 0)
9136         exit_sys_thread("pthread_join", result);
9137
9138     return 0;
9139 }
9140
9141 void exit_sys_thread(const char *msg, int err)
9142 {
9143     fprintf(stderr, "%s: %s\n", msg, strerror(err));
9144     exit(EXIT_FAILURE);
9145 }
9146
9147 void *thread_proc(void *param)
9148 {
9149     pthread_t tid;
9150     int i;
9151
9152     tid = pthread_self();
9153
9154     pthread_cancel(tid);          /* pthread_exit daha normal */
9155
9156     for (i = 0; i < 10; ++i) {
9157         printf("%s: %d\n", (char *)param, i);
9158         sleep(1);
9159     }
9160
9161     return NULL;
9162 }
9163
9164 /
*-----*
-----*
9165     pthread_attr_t nesnesinin içerisindeki elemanlar pthread_attr_getxxx
```

```
fonksiyonlarıyla alınabilirler. Örneğin pthread_attr_getstacksize
9166    thread'in stack uzunluğunu almak için, pthread_attr_getdetachstate      ↵
        thread'in joinable olup olmadığını almak için kullanılır.
9167    Maalesef POSIX'te thread'in özellikleri elde edilememektedir. Ancak      ↵
        bunun için Linux'ta pthread_getattr_np fonksiyonu bulunmaktadır.
9168    Bu fonksiyon Linux'a özgüdür. (Fonksiyon isminin sonundaki _np      ↵
        "nonportable" dan geliyor.)
9169  -----
9170
9171 #define __GNU_SOURCE
9172
9173 #include <stdio.h>
9174 #include <stdlib.h>
9175 #include <string.h>
9176 #include <unistd.h>
9177 #include <pthread.h>
9178
9179 void exit_sys_thread(const char *msg, int err);
9180 void *thread_proc(void *param);
9181
9182 int main(void)
9183 {
9184     int result;
9185     pthread_t tid;
9186
9187     if ((result = pthread_create(&tid, NULL, thread_proc, "other thread")) != 0)
9188         exit_sys_thread("pthread_create", result);
9189
9190     if ((result = pthread_join(tid, NULL)) != 0)
9191         exit_sys_thread("pthread_join", result);
9192
9193     return 0;
9194 }
9195
9196 void exit_sys_thread(const char *msg, int err)
9197 {
9198     fprintf(stderr, "%s: %s\n", msg, strerror(err));
9199     exit(EXIT_FAILURE);
9200 }
9201
9202 void *thread_proc(void *param)
9203 {
9204     pthread_t tid;
9205     pthread_attr_t attr;
9206     int result;
9207     size_t stacksize;
9208
9209     tid = pthread_self();
9210
9211     if ((result = pthread_getattr_np(tid, &attr)) != 0)
9212         exit_sys_thread("pthread_getattr_np", result);
```

```
9213
9214     if ((result = pthread_attr_getstacksize(&attr, &stacksize)) != 0)
9215         exit_sys_thread("pthread_attr_getstacksize", result);
9216
9217     printf("Stack size: %lu\n", (unsigned long)stacksize);
9218
9219     if ((result = pthread_attr_destroy(&attr)) != 0)
9220         exit_sys_thread("pthread_attr_getstacksize", result);
9221
9222     return NULL;
9223 }
9224
9225 /
*-----*
-----*
9226     Mutex nesnesi ile kritik kod oluşturmak için önce pthread_mutex_t
9227     türünden global bir nesne tanımlanır. Daha sonra bu nesne
9228     PTHREAD_MUTEX_INITIALIZER makrosuyla ilkdeğer verilerek statik biçimde
9229     ya da pthread_mutex_init fonksiyonuyla dinamik biçimde
9230     ilkdeğerlenir. Kritik kod da pthread_mutex_lock ve pthread_mutex_unlock
9231     çağrıları arasına yerleştirilir. İşlemler bittikten sonra
9232     mutex nesnesi pthread_mutex_destroy fonksiyonuyla boşaltılmalıdır.
9233     pthread_mutex_destroy ile mutex nesnesini yok etmek için mutex
9234     nesnesinin locked durumda olmaması gereklidir. Aksi halde "tanımsız
9235     davranış (undefined behavior)" oluşacaktır.
9236 -----*/
9237
9238 #include <stdio.h>
9239 #include <stdlib.h>
9240 #include <string.h>
9241 #include <unistd.h>
9242 #include <pthread.h>
9243
9244 void exit_sys_thread(const char *msg, int err);
9245 void *thread_proc1(void *param);
9246 void *thread_proc2(void *param);
9247
9248 pthread_mutex_t g_mutex;
9249
9250 int g_count;
9251
9252 int main(void)
9253 {
9254     int result;
9255     pthread_t tid1, tid2;
9256
9257     if ((result = pthread_mutex_init(&g_mutex, NULL)) != 0)
9258         exit_sys_thread("pthread_mutex_init", result);
9259
9260     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
9261         exit_sys_thread("pthread_create", result);
```

```
9258     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
9259         exit_sys_thread("pthread_create", result);
9260
9261     if ((result = pthread_join(tid1, NULL)) != 0)
9262         exit_sys_thread("pthread_join", result);
9263
9264     if ((result = pthread_join(tid2, NULL)) != 0)
9265         exit_sys_thread("pthread_join", result);
9266
9267     printf("g_count = %d\n", g_count);
9268
9269     pthread_mutex_destroy(&g_mutex);
9270
9271     return 0;
9272 }
9273
9274 void exit_sys_thread(const char *msg, int err)
9275 {
9276     fprintf(stderr, "%s: %s\n", msg, strerror(err));
9277     exit(EXIT_FAILURE);
9278 }
9279
9280 void *thread_proc1(void *param)
9281 {
9282     int i;
9283     int result;
9284
9285     for (i = 0; i < 1000000; ++i) {
9286         if ((result = pthread_mutex_lock(&g_mutex)) != 0)
9287             exit_sys_thread("pthread_mutex_lock", result);
9288
9289         ++g_count; /* Critical section */
9290
9291         if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
9292             exit_sys_thread("pthread_mutex_lock", result);
9293     }
9294
9295     return NULL;
9296 }
9297
9298 void *thread_proc2(void *param)
9299 {
9300     int i;
9301     int result;
9302
9303     for (i = 0; i < 1000000; ++i) {
9304         if ((result = pthread_mutex_lock(&g_mutex)) != 0)
9305             exit_sys_thread("pthread_mutex_lock", result);
9306
9307         ++g_count; /* Critical section */
9308
9309         if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
9310             exit_sys_thread("pthread_mutex_lock", result);
```

```
9311     }
9312
9313     return NULL;
9314 }
9315
9316 /
*-----*
-----*
9317     Mutex nesnesi ile kritik kod oluşturma (mutex nesnesinin sahipliği
9318     pthread_mutex_unlock ile yalnızca onu almış thread tarafından
9319     bırakılabilir.) */
9320 #include <stdio.h>
9321 #include <stdlib.h>
9322 #include <string.h>
9323 #include <time.h>
9324 #include <unistd.h>
9325 #include <pthread.h>
9326
9327 void exit_sys_thread(const char *msg, int err);
9328 void *thread_proc1(void *param);
9329 void *thread_proc2(void *param);
9330 void do_something(const char *name);
9331
9332 pthread_mutex_t g_mutex = PTHREAD_MUTEX_INITIALIZER;
9333
9334 int main(void)
9335 {
9336     int result;
9337     pthread_t tid1, tid2;
9338
9339     srand(time(NULL));
9340
9341     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
9342         exit_sys_thread("pthread_create", result);
9343
9344     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
9345         exit_sys_thread("pthread_create", result);
9346
9347     if ((result = pthread_join(tid1, NULL)) != 0)
9348         exit_sys_thread("pthread_join", result);
9349
9350     if ((result = pthread_join(tid2, NULL)) != 0)
9351         exit_sys_thread("pthread_join", result);
9352
9353     printf("g_count = %d\n", g_count);
9354
9355     pthread_mutex_destroy(&g_mutex);
9356
9357     return 0;
9358 }
```

```
9359
9360 void exit_sys_thread(const char *msg, int err)
9361 {
9362     fprintf(stderr, "%s: %s\n", msg, strerror(err));
9363     exit(EXIT_FAILURE);
9364 }
9365
9366 void *thread_proc1(void *param)
9367 {
9368     int i;
9369
9370     for (i = 0; i < 10; ++i) {
9371         do_something("thread1");
9372         usleep(rand() % 30000);
9373     }
9374
9375     return NULL;
9376 }
9377
9378 void *thread_proc2(void *param)
9379 {
9380     int i;
9381
9382     for (i = 0; i < 10; ++i) {
9383         do_something("thread2");
9384         usleep(rand() % 30000);
9385     }
9386
9387     return NULL;
9388 }
9389
9390 void do_something(const char *name)
9391 {
9392     int result;
9393
9394     if ((result = pthread_mutex_lock(&g_mutex)) != 0)
9395         exit_sys_thread("pthread_mutex_lock", result);
9396
9397     printf("%s- Step 1\n", name);
9398     usleep(rand() % 30000);
9399     printf("%s- Step 2\n", name);
9400     usleep(rand() % 30000);
9401     printf("%s- Step 3\n", name);
9402     usleep(rand() % 30000);
9403     printf("%s- Step 4\n", name);
9404     usleep(rand() % 30000);
9405     printf("%s- Step 5\n", name);
9406     printf("-----\n");
9407
9408     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
9409         exit_sys_thread("pthread_mutex_unlock", result);
9410 }
9411
```

```
9412  /
9413      *-----*
9414      -----
9415      pthread_mutex_timedlock fonksiyonu en kötü olasılıkla kilit açılmamışsa →
9416          zaman aşımından dolayı blokeyi kaldırıbmaktadır.
9417      Ancak fonksiton zaman aşımı olarak epoch'tan itibaren gerçek zamanı →
9418          istemektedir. Yani örneğin biz 5 saniyelik bir zaman aşımını →
9419      belirteceksek önce epoch'tan itibaresn (01/01/1970) şimdiye kadar geçen →
9420          saniye sayısını bulmamız sonra buna 5 saniye eklememiz gereklidir.
9421      time fonksyonun epoch'tan geçen saniye sayısını bize verdiğini →
9422          anımsayınız. Eğer pthread_mutex_timedlock fonksiyonu zaman aşımından →
9423          dolayı sonlanmışsa error kodu ETIMEDOUT biçiminde elde edilmektedir. →
9424          Aşağıdaki programda birinci thread mutex'i kilitlenmiş ve 20 →
9425          saniye beklemiştir. İkinci thread ise 5 saniye kadar mutex kildi →
9426          açılmamışsa blokeyi sonlandırmaktadır.
9427
9428  -----
9429      -----
9430
9431  #include <stdio.h>
9432  #include <stdlib.h>
9433  #include <string.h>
9434  #include <time.h>
9435  #include <errno.h>
9436  #include <unistd.h>
9437  #include <pthread.h>
9438  #include <sys/time.h>
9439
9440  void exit_sys_thread(const char *msg, int err);
9441  void *thread_proc1(void *param);
9442  void *thread_proc2(void *param);
9443
9444  pthread_mutex_t g_mutex;
9445
9446  int main(void)
9447  {
9448      int result;
9449      pthread_t tid1, tid2;
9450
9451      if ((result = pthread_mutex_init(&g_mutex, NULL)) != 0)
9452          exit_sys_thread("pthread_mutex_init", result);
9453
9454      if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
9455          exit_sys_thread("pthread_create", result);
9456
9457      sleep(1);
9458
9459      if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
9460          exit_sys_thread("pthread_create", result);
9461
9462      if ((result = pthread_join(tid1, NULL)) != 0)
9463          exit_sys_thread("pthread_join", result);
9464
9465      if ((result = pthread_join(tid2, NULL)) != 0)
9466          exit_sys_thread("pthread_join", result);
9467
9468      if ((result = pthread_mutex_destroy(&g_mutex)) != 0)
9469          exit_sys_thread("pthread_mutex_destroy", result);
9470
9471      if ((result = pthread_exit(0)) != 0)
9472          exit_sys_thread("pthread_exit", result);
9473
9474  }
```

```
9456     if ((result = pthread_join(tid2, NULL)) != 0)
9457         exit_sys_thread("pthread_join", result);
9458
9459     pthread_mutex_destroy(&g_mutex);
9460
9461     return 0;
9462 }
9463
9464 void exit_sys_thread(const char *msg, int err)
9465 {
9466     fprintf(stderr, "%s: %s\n", msg, strerror(err));
9467     exit(EXIT_FAILURE);
9468 }
9469
9470 void *thread_proc1(void *param)
9471 {
9472     int result;
9473
9474     if ((result = pthread_mutex_lock(&g_mutex)) != 0)
9475         exit_sys_thread("pthread_mutex_lock", result);
9476
9477     sleep(20);
9478
9479     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
9480         exit_sys_thread("pthread_mutex_unlock", result);
9481
9482     printf("First thread unlocked mutex!\n");
9483
9484     return NULL;
9485 }
9486
9487 void *thread_proc2(void *param)
9488 {
9489     struct timespec ts;
9490     int result;
9491
9492     localtime(&ts.tv_sec);
9493
9494     ts.tv_sec += 5;
9495     ts.tv_nsec = 0;
9496
9497     if ((result = pthread_mutex_timedlock(&g_mutex, &ts)) != 0)
9498         if (result == ETIMEDOUT) {
9499             printf("Time out! Do something else!\n");
9500             return NULL;
9501         }
9502
9503     /* some code */
9504
9505     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
9506         exit_sys_thread("pthread_mutex_unlock", result);
9507
9508     return NULL;
```

```
9509 }
9510
9511 /
*-----*
-----*
9512     Mutex yaratıldığında recursive değildir. Yani bir thread mutex
9513     nesnesinin sahipliğini aldığında yeniden onu almaya çalışırsa
9514     kendi kendini kilitler. Aşağıda örnekte bu biçimde bir deadlock
9515     (kilitlenme) oluşacaktır.
9516 -----*/
9517
9518 #include <stdio.h>
9519 #include <stdlib.h>
9520 #include <string.h>
9521 #include <pthread.h>
9522
9523 void exit_sys_thread(const char *msg, int err);
9524 void *thread_proc1(void *param);
9525 void *thread_proc2(void *param);
9526 void foo(void);
9527 void bar(void);
9528
9529 pthread_mutex_t g_mutex;
9530
9531 int main(void)
9532 {
9533     int result;
9534     pthread_t tid1, tid2;
9535
9536     if ((result = pthread_mutex_init(&g_mutex, NULL)) != 0)
9537         exit_sys_thread("pthread_mutex_init", result);
9538
9539     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
9540         exit_sys_thread("pthread_create", result);
9541
9542     if ((result = pthread_join(tid1, NULL)) != 0)
9543         exit_sys_thread("pthread_join", result);
9544
9545     pthread_mutex_destroy(&g_mutex);
9546
9547     return 0;
9548 }
9549
9550 void exit_sys_thread(const char *msg, int err)
9551 {
9552     fprintf(stderr, "%s: %s\n", msg, strerror(err));
9553     exit(EXIT_FAILURE);
9554 }
9555
9556 void *thread_proc1(void *param)
```

```
9557     foo();  
9558  
9559     return NULL;  
9560 }  
9561  
9562 void foo(void)  
9563 {  
9564     int result;  
9565  
9566     if ((result = pthread_mutex_lock(&g_mutex)) != 0)  
9567         exit_sys_thread("pthread_mutex_lock", result);  
9568  
9569     printf("thread locked mutex first time!..\n");  
9570     bar();  
9571  
9572     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)  
9573         exit_sys_thread("pthread_mutex_unlock", result);  
9574 }  
9575  
9576 void bar(void)  
9577 {  
9578     int result;  
9579  
9580     if ((result = pthread_mutex_lock(&g_mutex)) != 0)  
9581         exit_sys_thread("pthread_mutex_lock", result);  
9582  
9583     printf("thread locked mutex second time!..\n");  
9584  
9585     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)  
9586         exit_sys_thread("pthread_mutex_unlock", result);  
9587 }  
9588  
9589 /  
*-----  
-----  
9590     Mutex özelliklerini set etmek için önce pthread_mutexattr_t türünden bir  
nesne tanımlanır. Sonra bu nesne pthread_mutexattr_init  
fonksiyonuyla ilkdeğerlenir. Daha sonra da pthread_mutexattr_setxxx  
fonksiyonlarıyla özellikler set edilir. Sonra da bu özellik nesnesi  
pthread_mutex_init fonksiyonuna parametre olarak geçirilir. Bu işlemden  
sonra artık bu özellik nesnesi pthread_mutexattr_destroy  
fonksiyonuyla yok edilebilir. pthread_mutexattr_settype mutex nesnesinin  
türünü set etmek için kullanılmaktadır. Burada  
PTHREAD_MUTEX_RECURSIVE  
tür olarak bir thread birden fazla kez aynı mutex nesnesinin sahipliğini  
almaya çalıştığında deadlock oluşmaması için kullanılmaktadır.  
Tabii recursive mutex'lerde ne kadar lock yapılmışsa nesneyi bırakmak  
için o sayıda unlock yapılması gerekmektedir. Burada type olarak  
PHREAD_MUTEX_ERRORCHECK girilirse mutex kendi kendini kilitlemez error  
koduyla geri döner. PHREAD_MUTEX_NORMAL kendi kendini kilitleyen  
mutex'tir.,  
9597     Mutex default durumda tür olarak PTHREAD_MUTEX_DEFAULT türüne sahiptir. POSIX standartları PTHREAD_MUTEX_DEFAULT türünün diğer üç türden
```

```
9598     bir tanesi olarak alınabileceğini söylemektedir. Linux sistemlerinde PTHREAD_MUTEX_DEFAULT türü PTHREAD_MUTEX_NORMAL biçiminde alınmıştır.
9599     Dolayısıyla Linux sistemlerinde mutex kendini kilitler durumdadır.
9600 -----
9601
9602 #include <stdio.h>
9603 #include <stdlib.h>
9604 #include <string.h>
9605 #include <unistd.h>
9606 #include <pthread.h>
9607
9608 void exit_sys_thread(const char *msg, int err);
9609 void *thread_proc1(void *param);
9610 void foo(void);
9611 void bar(void);
9612
9613 pthread_mutex_t g_mutex;
9614
9615 int main(void)
9616 {
9617     int result;
9618     pthread_t tid1;
9619     pthread_mutexattr_t mattr;
9620
9621     pthread_mutexattr_init(&mattr);
9622
9623     if ((result = pthread_mutexattr_settype(&mattr,
9624                                             PTHREAD_MUTEX_RECURSIVE)) != 0)
9625         exit_sys_thread("pthread_mutexattr_settype", result);
9626
9627     if ((result = pthread_mutex_init(&g_mutex, &mattr)) != 0)
9628         exit_sys_thread("pthread_mutex_init", result);
9629
9630     if ((result = pthread_mutexattr_destroy(&mattr)) != 0)
9631         exit_sys_thread("pthread_mutexattr_destroy", result);
9632
9633     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
9634         exit_sys_thread("pthread_create", result);
9635
9636     if ((result = pthread_join(tid1, NULL)) != 0)
9637         exit_sys_thread("pthread_join", result);
9638
9639     pthread_mutex_destroy(&g_mutex);
9640
9641     return 0;
9642 }
9643 void exit_sys_thread(const char *msg, int err)
9644 {
9645     fprintf(stderr, "%s: %s\n", msg, strerror(err));
9646     exit(EXIT_FAILURE);
9647 }
```

```
9648
9649 void *thread_proc1(void *param)
9650 {
9651     foo();
9652
9653     return NULL;
9654 }
9655
9656 void foo(void)
9657 {
9658     int result;
9659
9660     if ((result = pthread_mutex_lock(&g_mutex)) != 0)
9661         exit_sys_thread("pthread_mutex_lock", result);
9662
9663     printf("thread locked mutex first time!..\n");
9664     bar();
9665
9666     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
9667         exit_sys_thread("pthread_mutex_unlock", result);
9668 }
9669
9670 void bar(void)
9671 {
9672     int result;
9673
9674     if ((result = pthread_mutex_lock(&g_mutex)) != 0)
9675         exit_sys_thread("pthread_mutex_lock", result);
9676
9677     printf("thread locked mutex second time!..\n");
9678
9679     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
9680         exit_sys_thread("pthread_mutex_unlock", result);
9681 }
9682
9683 /
*-----*
-----*
9684     Mutex nesneleri farklı proseslerin thread'leri arasında da
9685         kullanılabilir. Bunun için mutex nesnesinin (yani pthread_mutex_t
9686             nesnesinin)
9687             paylaşılan bir bellek alanında yaratılmış olması gereklidir. Ancak nesnenin
9688                 paylaşılan bellek alanında yaratılmış olması yetmemektedir.
9689             Ayrıca mutex'in "shared" moda sokulması gereklidir. Bu işlem de
9690                 pthread_mutexattr_setpshared fonksiyonuyla yapılmaktadır. Bu
9691                     fonksiyonda
9692             parametre olarak PTHREAD_PROCESS_SHARED geçilmelidir.
9693
-----*/
9694 /* proc1.c */
9695
9696 #include <stdio.h>
```

```
9693 #include <stdlib.h>
9694 #include <string.h>
9695 #include <fcntl.h>
9696 #include <unistd.h>
9697 #include <pthread.h>
9698 #include <sys/mman.h>
9699
9700 #define SHM_PATH          "/shared_memory_mutex_test"
9701
9702 void exit_sys(const char *msg);
9703 void exit_sys_thread(const char *msg, int err);
9704
9705 int main(void)
9706 {
9707     int fdshm;
9708     void *addr;
9709     pthread_mutex_t *mutex;
9710     pthread_mutexattr_t mattr;
9711     int result;
9712
9713     if ((fdshm = shm_open(SHM_PATH, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR|S_IRGRP| ↪
9714         S_IROTH)) == -1)
9715         exit_sys("shm_open");
9716
9717     if (ftruncate(fdshm, 4096) == -1)
9718         exit_sys("ftruncate");
9719
9720     if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fdshm,    ↪
9721         0)) == MAP_FAILED)
9722         exit_sys("mmap");
9723
9724     mutex = (pthread_mutex_t *)addr;
9725
9726     pthread_mutexattr_init(&mattr);
9727     if ((result = pthread_mutexattr_setpshared(&mattr,
9728         PTHREAD_PROCESS_SHARED)) != 0)
9729         exit_sys_thread("pthread_mutexattr_setpshared", result);
9730
9731     if ((result = pthread_mutex_init(mutex, &mattr)) != 0)
9732         exit_sys_thread("pthread_mutex_init", result);
9733
9734     printf("Press ENTER to release mutex...\n");
9735     getchar();
9736
9737     if ((result = pthread_mutex_lock(mutex)) != 0)
9738         exit_sys_thread("pthread_mutex_lock", result);
9739
9740     sleep(2);
9741
9742     pthread_mutexattr_destroy(&mattr);
```

```
9743     munmap(addr, 4096);
9744
9745     close(fdshm);
9746
9747     if (shm_unlink(SHM_PATH) == -1)
9748         exit_sys("shm_unlink");
9749
9750     return 0;
9751 }
9752
9753 void exit_sys(const char *msg)
9754 {
9755     perror(msg);
9756
9757     exit(EXIT_FAILURE);
9758 }
9759
9760 void exit_sys_thread(const char *msg, int err)
9761 {
9762     fprintf(stderr, "%s: %s\n", msg, strerror(err));
9763     exit(EXIT_FAILURE);
9764 }
9765
9766
9767 /* proc2.c */
9768
9769 #include <stdio.h>
9770 #include <stdlib.h>
9771 #include <string.h>
9772 #include <fcntl.h>
9773 #include <unistd.h>
9774 #include <pthread.h>
9775 #include <sys/mman.h>
9776
9777 #define SHM_PATH      "/shaed_memory_mutex_test"
9778
9779 void exit_sys(const char *msg);
9780 void exit_sys_thread(const char *msg, int err);
9781
9782 int main(void)
9783 {
9784     int fdshm;
9785     void *addr;
9786     pthread_mutex_t *mutex;
9787     int result;
9788
9789     if ((fdshm = shm_open(SHM_PATH, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)) == -1)
9790         exit_sys("shm_open");
9791
9792     if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fdshm, 0)) == MAP_FAILED)
9793         exit_sys("mmap");
```

```
9794     mutex = (pthread_mutex_t *)addr;
9795
9796     printf("waiting for unclock...\n");
9797     if ((result = pthread_mutex_lock(mutex)) != 0)
9798         exit_sys_thread("pthread_mutex_lock", result);
9800
9801     printf("got mutex ownership!..\n");
9802
9803     if ((result = pthread_mutex_unlock(mutex)) != 0)
9804         exit_sys_thread("pthread_mutex_unlock", result);
9805
9806     munmap(addr, 4096);
9807
9808     close(fdshm);
9809
9810     return 0;
9811 }
9812
9813 void exit_sys(const char *msg)
9814 {
9815     perror(msg);
9816
9817     exit(EXIT_FAILURE);
9818 }
9819
9820 void exit_sys_thread(const char *msg, int err)
9821 {
9822     fprintf(stderr, "%s: %s\n", msg, strerror(err));
9823     exit(EXIT_FAILURE);
9824 }
9825
9826 /
*-----*-----*-----*
```

9827 Bir thread bir mutex'in sahipliğini aldıktan sonra sonlanırsa be olur? →
 İşte default durumda pthread_mutex_lock fonksiyonunda
9828 bekleyen thread'ler beklemeye devam ederler. Yeni thread'ler bu →
 fonksiyona girerlerse onlar da beklerler. Zaten bu durumdaki
9829 mutex'in açılması söz konusu değildir. Bu durumdaki mutex'ler "abandoned" →
 mutexes" denilmektedir. Fakat mutex "robust" denilen
9830 bir mutex özelliği de vardır. pthread_mutexattr_setrobust fonksiyonuyla →
 bu durum değiştirilebilir. Bu fonksiyonda PTHREAD_MUTEX_ROBUST
9831 parametresi girildiğinde artık mutex "robust" olur. Robust mutex'lere →
 sahip thread'ler sonlansa bile deadlock oluşmaz.
9832 pthread_mutex_lock fonksiyonları EOWNERDEAD hata koduyla geri dönerler. →
 Bu durumda mutex nesnesi unlcok yapılmırsa sorun oluşmaz. Ancak
9833 işlevini yerine getiremez durumdadır. Bu tür duurmdaki mutex'lere →
 bozulmuş (inconsistent) mutex denilmektedir. Bozulmuş mutex'leri eski →
 haline getirmek
9834 için pthread_mutex_consistent fonksiyonu çağrılmalıdır. Aşağıdaki bu →
 biçimde bir deadlock örneği verilmiştir.

----- →

```
-----*/
9836
9837 #include <stdio.h>
9838 #include <stdlib.h>
9839 #include <string.h>
9840 #include <unistd.h>
9841 #include <pthread.h>
9842
9843 void exit_sys_thread(const char *msg, int err);
9844 void *thread_proc1(void *param);
9845 void *thread_proc2(void *param);
9846
9847 pthread_mutex_t g_mutex;
9848
9849 int main(void)
9850 {
9851     int result;
9852     pthread_t tid1, tid2;
9853
9854     if ((result = pthread_mutex_init(&g_mutex, NULL)) != 0)
9855         exit_sys_thread("pthread_mutex_init", result);
9856
9857     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
9858         exit_sys_thread("pthread_create", result);
9859
9860     sleep(2);
9861
9862     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
9863         exit_sys_thread("pthread_create", result);
9864
9865     if ((result = pthread_join(tid1, NULL)) != 0)
9866         exit_sys_thread("pthread_join", result);
9867
9868     if ((result = pthread_join(tid2, NULL)) != 0)
9869         exit_sys_thread("pthread_join", result);
9870
9871
9872     pthread_mutex_destroy(&g_mutex);
9873
9874     return 0;
9875 }
9876
9877 void exit_sys_thread(const char *msg, int err)
9878 {
9879     fprintf(stderr, "%s: %s\n", msg, strerror(err));
9880     exit(EXIT_FAILURE);
9881 }
9882
9883 void *thread_proc1(void *param)
9884 {
9885     int result;
9886
9887     if ((result = pthread_mutex_lock(&g_mutex)) != 0)
```

```
9888     exit_sys_thread("pthread_mutex_lock", result);
9889
9890     printf("thread1 gets ownership and terminate!..\n");
9891     pthread_exit(NULL);
9892
9893     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
9894         exit_sys_thread("pthread_mutex_unlock", result);
9895
9896     return NULL;
9897 }
9898
9900 void *thread_proc2(void *param)
9901 {
9902     int result;
9903
9904     printf("second thread waits abandoned mutex (deadlock)... \n");
9905     if ((result = pthread_mutex_lock(&g_mutex)) != 0)
9906         exit_sys_thread("pthread_mutex_lock", result);
9907
9908     /* .... */
9909
9910     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
9911         exit_sys_thread("pthread_mutex_unlock", result);
9912
9913
9914     return NULL;
9915 }
9916
9917 /
*-----*
-----*
-----*
-----*
```

9918 Abandoned mutex'in yeniden kullanilabilir (consistent) duruma sokulmasi
9919 -----*

```
9920
9921 #include <stdio.h>
9922 #include <stdlib.h>
9923 #include <string.h>
9924 #include <unistd.h>
9925 #include <errno.h>
9926 #include <pthread.h>
9927
9928 void exit_sys_thread(const char *msg, int err);
9929 void *thread_proc1(void *param);
9930 void *thread_proc2(void *param);
9931
9932 pthread_mutex_t g_mutex;
9933
9934 int main(void)
9935 {
9936     int result;
9937     pthread_t tid1, tid2;
```

```
9938     pthread_mutexattr_t mattr;
9939
9940     pthread_mutexattr_init(&mattr);
9941
9942     if ((result = pthread_mutexattr_setrobust(&mattr,
9943                                              PTHREAD_MUTEX_ROBUST)) != 0)
9944         exit_sys_thread("pthread_mutexattr_settype", result);
9945
9946     if ((result = pthread_mutex_init(&g_mutex, &mattr)) != 0)
9947         exit_sys_thread("pthread_mutex_init", result);
9948
9949     if ((result = pthread_mutexattr_destroy(&mattr)) != 0)
9950         exit_sys_thread("pthread_mutexattr_destroy", result);
9951
9952     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
9953         exit_sys_thread("pthread_create", result);
9954
9955     sleep(2);
9956
9957     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
9958         exit_sys_thread("pthread_create", result);
9959
9960     if ((result = pthread_join(tid1, NULL)) != 0)
9961         exit_sys_thread("pthread_join", result);
9962
9963     if ((result = pthread_join(tid2, NULL)) != 0)
9964         exit_sys_thread("pthread_join", result);
9965
9966     pthread_mutex_destroy(&g_mutex);
9967
9968     return 0;
9969 }
9970
9971 void exit_sys_thread(const char *msg, int err)
9972 {
9973     fprintf(stderr, "%s: %s\n", msg, strerror(err));
9974     exit(EXIT_FAILURE);
9975 }
9976
9977 void *thread_proc1(void *param)
9978 {
9979     int result;
9980
9981     if ((result = pthread_mutex_lock(&g_mutex)) != 0)
9982         exit_sys_thread("pthread_mutex_lock", result);
9983
9984     printf("thread1 gets ownership and terminate!..\n");
9985
9986     pthread_exit(NULL);
9987
9988     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
9989         exit_sys_thread("pthread_mutex_unlock", result);
```

```
9990
9991     return NULL;
9992 }
9993
9994 void *thread_proc2(void *param)
9995 {
9996     int result;
9997
9998     printf("second thread waits abandoned mutex...\n");
9999     if ((result = pthread_mutex_lock(&g_mutex)) != 0 && result == EOWNERDEAD)
10000         if ((result = pthread_mutex_consistent(&g_mutex)) != 0)
10001             exit_sys_thread("pthread_mutex_consistent", result);
10002
10003     /* ... */
10004
10005     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
10006         exit_sys_thread("pthread_mutex_unlock", result);
10007
10008     printf("Ok\n");
10009
10010     return NULL;
10011 }
10012
10013 /
*-----*
-----*
10014 Durum değişkenleri (condition variables) mutex nesneleriyle birlikte
10015     kullanılan senkronizasyon nesneleridir. Genellikle
10016     bir koşul sağlanana kadar bir thread'i bloke ederek bekletmek için
10017     kullanılırlar. Durum değişkenlerinesneleri pthread_cond_t türüyle
10018     temsil edilmektedir. Bu nesne pthread_cond_init fonksiyonuyla dinamik
10019     olarak ya da PTHREAD_COND_INITIALIZER makrosuyla statik
10020     olarak ilkdeğerlenir. Bekleme işlemi pthread_cond_wait fonksiyonuyla
10021     durum değişken nesnesi ve mutex nesnesi verilerek sağlanmaktadır.
10022     Bekleme sırasında ilgili mutex kilitlenmiş olmalıdır. Bu nedenle bu
10023     fonksiyon pthread_mutex_lock ve pthread_mutex_unlock fonksiyonlarının
10024     arasında çağrıılır. pthread_cond_wait fonksiyonu çağrıldığında thread
10025     uykuya dalmadan önce atomik bir biçimde mutex nesnesinin
10026     kilitini açar. Thread'i uykudan uyandırmak için başka bir thread'in
10027     pthread_cond_signal fonksiyonunu çağrıması gereklidir. Bu durumda
10028     bekleyen thread atomik bir biçimde mutex'i yeniden kilitleyerek uyanır.
10029     Ancak maalesef pthread_cond_wait fonksiyonundan uyanmanın tek nedeni
10030     pthread_cond_wait uygulanması değildir. Sistemler bazen gereksiz
10031     uyandırmalar yapabilmektedir. Bu nedenle programcının uyanma
10032     sonrasında global bir durum değişkenine bakarak uyandırmanın
10033     gerçekliğinden emin olması gereklidir. Bu tipik olarak bir while
10034     döngüsüyle sağlanmaktadır:
10035
10036     while (global koşul değişkeni set edilmediği sürece)
10037         pthread?cond_wait(...);
10038
10039     Bu durumda tipik bekleme kalibi söyle olmalıdır:
```

```
10029     pthread_mutex_lock(&g_mutex);
10030
10031     while (g_flag == 0)
10032         pthread_cond_wait(&g_cond, &g_mutex);
10033
10034     pthread_mutex_unlock(&g_mutex);
10035
10036     pthread_cond_wait fonksiyonundan thread'in gereksiz uyandırılmasına →
10037     İngilizce "spurious wakeup" denilmektedir.
10038
10039     Diğer thread uyuyan thread'i uyandırmak için pthread_cond_signal →
10040     fonksiyonunu çağrımadan önce global flag değişkenini uygun biçimde set →
10041     etmelidir.
10042     Tabii bu işlem (zounlu olmasa da) yine mutex kontrolü içerisinde →
10043     yapılmalıdır.
10044
10045     pthread_mutex_lock(&g_mutex);
10046     g_flag = 1;
10047     pthread_mutex_unlock(&g_mutex);
10048     pthread_cond_signal(&g_cond);
10049
10050     pthread_cond_signal fonksiyonu kritik kod içerisinde de çağrılabılır. Bu →
10051     durumda uyandırma yapılır ancak uyandırılan thread mutex →
10052     kilitli olduğu için henüz tam uyanamaz. Bu durumda signal uygulayan →
10053     thread mutex'i bırakınca diğer thread tam olarak uyanır. Duerum →
10054     değişken nesnesi
10055     işlem bitince pthread_cond_destroy fonksiyonuyla geri bırakılmalıdır.
10056
10057 -----*/
10058
10059 #include <stdio.h>
10060 #include <stdlib.h>
10061 #include <string.h>
10062 #include <unistd.h>
10063 #include <pthread.h>
10064
10065 void exit_sys_thread(const char *msg, int err);
10066 void *thread_proc1(void *param);
10067 void *thread_proc2(void *param);
10068
10069 pthread_mutex_t g_mutex = PTHREAD_MUTEX_INITIALIZER;
10070 pthread_cond_t g_cond = PTHREAD_COND_INITIALIZER;
10071 int g_condition = 0;
10072
10073 int main(void)
10074 {
10075     int result;
10076     pthread_t tid1, tid2;
10077
10078     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
10079         exit_sys_thread("pthread_create", result);
```

```
10074
10075     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
10076         exit_sys_thread("pthread_create", result);
10077
10078     if ((result = pthread_join(tid1, NULL)) != 0)
10079         exit_sys_thread("pthread_join", result);
10080
10081     if ((result = pthread_join(tid2, NULL)) != 0)
10082         exit_sys_thread("pthread_join", result);
10083
10084     pthread_cond_destroy(&g_cond);
10085     pthread_mutex_destroy(&g_mutex);
10086
10087     return 0;
10088 }
10089
10090 void exit_sys_thread(const char *msg, int err)
10091 {
10092     fprintf(stderr, "%s: %s\n", msg, strerror(err));
10093     exit(EXIT_FAILURE);
10094 }
10095
10096 void *thread_proc1(void *param)
10097 {
10098     int result;
10099
10100    printf("thread1 is running...\n");
10101    sleep(5);
10102
10103    if ((result = pthread_mutex_lock(&g_mutex)) != 0)
10104        exit_sys_thread("pthread_mutex_lock", result);
10105
10106    g_condition = 1;
10107
10108    if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
10109        exit_sys_thread("pthread_mutex_unlock", result);
10110
10111    if ((result = pthread_cond_signal(&g_cond)) != 0)
10112        exit_sys_thread("pthread_cond_signal", result);
10113
10114    sleep(5);
10115
10116    return NULL;
10117 }
10118
10119 void *thread_proc2(void *param)
10120 {
10121     int result;
10122
10123     printf("thread2 waiting for the condition...\n");
10124
10125     if ((result = pthread_mutex_lock(&g_mutex)) != 0)
10126         exit_sys_thread("pthread_mutex_lock", result);
```

```
10127
10128     while (g_condition == 0)
10129         if ((result = pthread_cond_wait(&g_cond, &g_mutex)) != 0)
10130             exit_sys_thread("pthread_cond_wait", result);
10131
10132     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
10133         exit_sys_thread("pthread_mutex_unlock", result);
10134
10135     printf("condition granted...\n");
10136
10137     sleep(5);
10138
10139     return NULL;
10140 }
10141
10142 /
*-----*
```

10143 Eğer pthread_cond_wait fonksiyonunda aynı koşul değişken nesnesi ve aynı mutex eşliğinde uykuya dalmış birden fazla bekleyen thread varsa pthread_cond_signal bunlardan yalnızca bir tanesini uyandırmak ister. Diğerleri pthread_cond_wait fonksiyonunda uyumaya devam ederler. Ancak maalesef bazı gerçekleştirimler pthread_cond_signal uygulandığında tek bir thread'i uyandırmayı başaramayıp bunların hepsini uyandırabilmektedir. Mademki pthread_cond_signal fonksiyonunun anlamı uyuyanlardan yalnızca bir tanesini uyandırmaktır. Bu durumda programcı spurious wakeup için kendi önlemini almalıdır. Bu önlem tipik olarak ilgili global flag'in yeniden eski durumuna çekilmesi ile alınabilir.

```
10144-----*/
```

10150
10151 #include <stdio.h>
10152 #include <stdlib.h>
10153 #include <string.h>
10154 #include <unistd.h>
10155 #include <pthread.h>
10156
10157 void exit_sys_thread(const char *msg, int err);
10158 void *thread_proc1(void *param);
10159 void *thread_proc2(void *param);
10160 void *thread_proc3(void *param);
10161
10162 pthread_mutex_t g_mutex = PTHREAD_MUTEX_INITIALIZER;
10163 pthread_cond_t g_cond = PTHREAD_COND_INITIALIZER;
10164 int g_condition = 0;
10165
10166 int main(void)
10167 {
10168 int result;
10169 pthread_t tid1, tid2, tid3;

```
10171     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
10172         exit_sys_thread("pthread_create", result);
10173
10174     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
10175         exit_sys_thread("pthread_create", result);
10176
10177     if ((result = pthread_create(&tid3, NULL, thread_proc3, NULL)) != 0)
10178         exit_sys_thread("pthread_create", result);
10179
10180     if ((result = pthread_join(tid1, NULL)) != 0)
10181         exit_sys_thread("pthread_join", result);
10182
10183     if ((result = pthread_join(tid2, NULL)) != 0)
10184         exit_sys_thread("pthread_join", result);
10185
10186     if ((result = pthread_join(tid3, NULL)) != 0)
10187         exit_sys_thread("pthread_join", result);
10188
10189     pthread_cond_destroy(&g_cond);
10190     pthread_mutex_destroy(&g_mutex);
10191
10192     return 0;
10193 }
10194
10195 void exit_sys_thread(const char *msg, int err)
10196 {
10197     fprintf(stderr, "%s: %s\n", msg, strerror(err));
10198     exit(EXIT_FAILURE);
10199 }
10200
10201 void *thread_proc1(void *param)
10202 {
10203     int result;
10204
10205     printf("thread1 is running...\n");
10206
10207     printf("press ENTER to signal!..\n");
10208     getchar();
10209
10210     if ((result = pthread_mutex_lock(&g_mutex)) != 0)
10211         exit_sys_thread("pthread_mutex_lock", result);
10212
10213     g_condition = 1;
10214
10215     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
10216         exit_sys_thread("pthread_mutex_unlock", result);
10217
10218     if ((result = pthread_cond_signal(&g_cond)) != 0)
10219         exit_sys_thread("pthread_cond_signal", result);
10220
10221     sleep(5);
10222
10223     if ((result = pthread_mutex_lock(&g_mutex)) != 0)
```

```
10224     exit_sys_thread("pthread_mutex_lock", result);
10225
10226     g_condition = 1;
10227
10228     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
10229         exit_sys_thread("pthread_mutex_unlock", result);
10230
10231     if ((result = pthread_cond_signal(&g_cond)) != 0)
10232         exit_sys_thread("pthread_cond_signal", result);
10233
10234     return NULL;
10235 }
10236
10237 void *thread_proc2(void *param)
10238 {
10239     int result;
10240
10241     printf("thread2 waiting for the condition...\n");
10242
10243     if ((result = pthread_mutex_lock(&g_mutex)) != 0)
10244         exit_sys_thread("pthread_mutex_lock", result);
10245
10246     while (g_condition == 0)
10247         if ((result = pthread_cond_wait(&g_cond, &g_mutex)) != 0)
10248             exit_sys_thread("pthread_cond_wait", result);
10249
10250     g_condition = 0;
10251
10252     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
10253         exit_sys_thread("pthread_mutex_unlock", result);
10254
10255     printf("thread2 condition granted...\n");
10256
10257     return NULL;
10258 }
10259
10260 void *thread_proc3(void *param)
10261 {
10262     int result;
10263
10264     printf("thread3 waiting for the condition...\n");
10265
10266     if ((result = pthread_mutex_lock(&g_mutex)) != 0)
10267         exit_sys_thread("pthread_mutex_lock", result);
10268
10269     while (g_condition == 0)
10270         if ((result = pthread_cond_wait(&g_cond, &g_mutex)) != 0)
10271             exit_sys_thread("pthread_cond_wait", result);
10272
10273     g_condition = 0;
10274
10275     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
10276         exit_sys_thread("pthread_mutex_unlock", result);
```

```
10277     printf("thread3 condition granted...\n");
10278
10279     return NULL;
10280 }
10281 }
10282 /
10283 *
*-----*
10284
10285
10286
10287
10288 -----*/
10289
10290 #include <stdio.h>
10291 #include <stdlib.h>
10292 #include <string.h>
10293 #include <unistd.h>
10294 #include <pthread.h>
10295
10296 void exit_sys_thread(const char *msg, int err);
10297 void *thread_proc1(void *param);
10298 void *thread_proc2(void *param);
10299 void *thread_proc3(void *param);
10300
10301 pthread_mutex_t g_mutex = PTHREAD_MUTEX_INITIALIZER;
10302 pthread_cond_t g_cond = PTHREAD_COND_INITIALIZER;
10303 int g_condition = 0;
10304
10305 int main(void)
10306 {
10307     int result;
10308     pthread_t tid1, tid2, tid3;
10309
10310     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
10311         exit_sys_thread("pthread_create", result);
10312
10313     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
10314         exit_sys_thread("pthread_create", result);
10315
10316     if ((result = pthread_create(&tid3, NULL, thread_proc3, NULL)) != 0)
10317         exit_sys_thread("pthread_create", result);
10318
10319     if ((result = pthread_join(tid1, NULL)) != 0)
10320         exit_sys_thread("pthread_join", result);
10321
10322     if ((result = pthread_join(tid2, NULL)) != 0)
```

```
10323     exit_sys_thread("pthread_join", result);
10324
10325     if ((result = pthread_join(tid3, NULL)) != 0)
10326         exit_sys_thread("pthread_join", result);
10327
10328     pthread_cond_destroy(&g_cond);
10329     pthread_mutex_destroy(&g_mutex);
10330
10331     return 0;
10332 }
10333
10334 void exit_sys_thread(const char *msg, int err)
10335 {
10336     fprintf(stderr, "%s: %s\n", msg, strerror(err));
10337     exit(EXIT_FAILURE);
10338 }
10339
10340 void *thread_proc1(void *param)
10341 {
10342     int result;
10343
10344     printf("thread1 is running...\n");
10345
10346     printf("press ENTER to signal!..\n");
10347     getchar();
10348
10349     if ((result = pthread_mutex_lock(&g_mutex)) != 0)
10350         exit_sys_thread("pthread_mutex_lock", result);
10351
10352     g_condition = 1;
10353
10354     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
10355         exit_sys_thread("pthread_mutex_unlock", result);
10356
10357     if ((result = pthread_cond_broadcast(&g_cond)) != 0)
10358         exit_sys_thread("pthread_cond_broadcast", result);
10359
10360     return NULL;
10361 }
10362
10363 void *thread_proc2(void *param)
10364 {
10365     int result;
10366
10367     printf("thread2 waiting for the condition...\n");
10368
10369     if ((result = pthread_mutex_lock(&g_mutex)) != 0)
10370         exit_sys_thread("pthread_mutex_lock", result);
10371
10372     while (g_condition == 0)
10373         if ((result = pthread_cond_wait(&g_cond, &g_mutex)) != 0)
10374             exit_sys_thread("pthread_cond_wait", result);
10375
```

```
10376     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
10377         exit_sys_thread("pthread_mutex_unlock", result);
10378
10379     printf("thread2 condition granted...\n");
10380
10381     return NULL;
10382 }
10383
10384 void *thread_proc3(void *param)
10385 {
10386     int result;
10387
10388     printf("thread3 waiting for the condition...\n");
10389
10390     if ((result = pthread_mutex_lock(&g_mutex)) != 0)
10391         exit_sys_thread("pthread_mutex_lock", result);
10392
10393     while (g_condition == 0)
10394         if ((result = pthread_cond_wait(&g_cond, &g_mutex)) != 0)
10395             exit_sys_thread("pthread_cond_wait", result);
10396
10397     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
10398         exit_sys_thread("pthread_mutex_unlock", result);
10399
10400     printf("thread3 condition granted...\n");
10401
10402     return NULL;
10403 }
10404
10405 /
*-----*
```

10406 Üretici-Tüketicisi Problemi (Producer-Consumer Problem) gerçek hayatı en sık karşılaşılan thread senkronizasyon kalıbidir.

10407 Bu problemde üretici thread bir döngü içerisinde bir faaliye sonucunda bir değer elde eder. Bu değeri kendisi işlemez. Paylaşılan bir global değişikene yazar. Diğer thread yine bir döngü içerisinde o global değişkenin değeri alarak onu işler. Yani problemde thread'lerden biri değerleri elde etme işini diğer ise bunları işleme işini yapmaktadır. Değerleri elde eden thread'e üretici thread, değerleri alıp işleyen thread'e de tüketici thread denilmektedir. Şüphesiz tek bir değer değerleri elde edip kendisi işleyebilirdi.

10411 Ancak değerlerin elde edilmesi ve işlenmesi iki farklı thread'e yaptırıldığından hız kazancı sağlanmaktadır. Buradaki problemde söyle bir senkronizasyon sorunu vardır: Üretici thread elde ettiği değeri global değişikene yazdıktan sonra yeni bir değeri tüketici onu almadan global değişikene yerleştirmemelidir. Benzer biçimde tüketici thread de eski değeri ikinci kez alıp işlememelidir. Yani üretici thread ancak tüketici thread önceki değeri alırsa yeni değeri yerleştirmelidir. Benzer biçimde tüketici thread de üretici thread yeni bir değer yerleştirdiyse onu almalıdır. Aşağıda bir senkronizasyon

uygulanmadan yapılan bir simülasyonu görüyorsunuz.

```
10416 -----*  
10417  
10418 #include <stdio.h>  
10419 #include <stdlib.h>  
10420 #include <string.h>  
10421 #include <unistd.h>  
10422 #include <pthread.h>  
10423  
10424 void exit_sys_thread(const char *msg, int err);  
10425 void *thread_proc_producer(void *param);  
10426 void *thread_proc_consumer(void *param);  
10427  
10428 int g_shared;  
10429  
10430 int main(void)  
10431 {  
10432     int result;  
10433     pthread_t tid_producer, tid_consumer;  
10434  
10435     srand(time(NULL));  
10436  
10437     if ((result = pthread_create(&tid_producer, NULL, thread_proc_producer, ▷  
10438         NULL)) != 0)  
10439         exit_sys_thread("pthread_create", result);  
10440  
10441     if ((result = pthread_create(&tid_consumer, NULL, thread_proc_consumer, ▷  
10442         NULL)) != 0)  
10443         exit_sys_thread("pthread_create", result);  
10444  
10445     if ((result = pthread_join(tid_producer, NULL)) != 0)  
10446         exit_sys_thread("pthread_join", result);  
10447  
10448     return 0;  
10449 }  
10450  
10451 void exit_sys_thread(const char *msg, int err)  
10452 {  
10453     fprintf(stderr, "%s: %s\n", msg, strerror(err));  
10454     exit(EXIT_FAILURE);  
10455 }  
10456  
10457  
10458 void *thread_proc_producer(void *param)  
10459 {  
10460     unsigned int seedval;  
10461     int i;  
10462  
10463     seedval = (unsigned int)time(NULL) + 20000;  
10464 }
```

```
10465     i = 0;
10466     for (;;) {
10467         usleep(rand_r(&seedval) % 300000);
10468         g_shared = i;
10469         if (i == 99)
10470             break;
10471         ++i;
10472     }
10473
10474     return NULL;
10475 }
10476
10477 void *thread_proc_consumer(void *param)
10478 {
10479     unsigned int seedval;
10480     int val;
10481
10482     seedval = (unsigned int)time(NULL);
10483
10484     for (;;) {
10485         val = g_shared;
10486         usleep(rand_r(&seedval) % 300000);
10487         printf("%d ", val);
10488         fflush(stdout);
10489         if (val == 99)
10490             break;
10491     }
10492     printf("\n");
10493
10494     return NULL;
10495 }
10496
10497 /
*-----*
```

-----*

10498 Üretici-Tüketici problemi semaphore nesneleriyle ya da koşul değişken nesneleriyle çözülebilir. Aslında semaphore'lar 10499 bu konuda daha kolay bir kod oluşturmaktadır. Ancak çözüm yöntemi 10500 aynıdır. Çözümdeki temel fikir üreticinin tüketiyi uyandırması, tüketicinin de üreticiyi uyandırmasıdır. İki koşul değişkeni alınır. 10501 Koşulu belirten global bir flag de başlangıçta 0 ilkdeğerileyle tanımlanır. Üretici bu değişken 1 olduğu sürece, tüketici de 0 olduğu 10502 sürece bekleyecektir. İşin başında bu flag sıfır olduğuna göre tüketici bekler durumdadır. Fakat üretici uykuya dalmaz. Üretici 10503 paylaşılan alana değeri yerleştirir. Fakat flag değişkenini kendisi 1 yapar. Tüketiciyi pthread_cond_signal ile uyandırır. Tüketici 10504 uyandığında flag atrık 1 olduğu için durum değişkeninden geçer ancak üretici bu sefer flag 1 olduğu için beklemektedir. İşte tüketici 10505 paylaşılan alandan bilgiyi alır. flag'i yeniden 0 yapıp üreticiyi uyandırmak için pthread_cond_signal çağrısı yapar. Burada bir tateravallı gibi üretici tüketiciyi, tüketici de üreticiyi 10506 uyandırmaktadır.

```
-----*/
10507
10508 #include <stdio.h>
10509 #include <stdlib.h>
10510 #include <string.h>
10511 #include <unistd.h>
10512 #include <pthread.h>
10513
10514 void exit_sys_thread(const char *msg, int err);
10515 void *thread_proc_producer(void *param);
10516 void *thread_proc_consumer(void *param);
10517
10518 pthread_mutex_t g_mutex = PTHREAD_MUTEX_INITIALIZER;
10519 pthread_cond_t g_cond_producer = PTHREAD_COND_INITIALIZER;
10520 pthread_cond_t g_cond_consumer = PTHREAD_COND_INITIALIZER;
10521
10522 int g_shared;
10523 int g_flag;
10524
10525 int main(void)
10526 {
10527     int result;
10528     pthread_t tid_producer, tid_consumer;
10529
10530     if ((result = pthread_create(&tid_producer, NULL, thread_proc_producer, ▷
10531         NULL)) != 0)
10532         exit_sys_thread("pthread_create", result);
10533
10534     if ((result = pthread_create(&tid_consumer, NULL, thread_proc_consumer, ▷
10535         NULL)) != 0)
10536         exit_sys_thread("pthread_create", result);
10537
10538     if ((result = pthread_join(tid_producer, NULL)) != 0)
10539         exit_sys_thread("pthread_join", result);
10540
10541     if ((result = pthread_join(tid_consumer, NULL)) != 0)
10542         exit_sys_thread("pthread_join", result);
10543
10544     pthread_cond_destroy(&g_cond_producer);
10545     pthread_cond_destroy(&g_cond_consumer);
10546     pthread_mutex_destroy(&g_mutex);
10547
10548
10549 void exit_sys_thread(const char *msg, int err)
10550 {
10551     fprintf(stderr, "%s: %s\n", msg, strerror(err));
10552     exit(EXIT_FAILURE);
10553 }
10554
10555 void *thread_proc_producer(void *param)
10556 {
```

```
10557     unsigned int seedval;
10558     int result, i;
10559
10560     seedval = (unsigned int)time(NULL) + 20000;
10561
10562     i = 0;
10563     for (;;) {
10564         usleep(rand_r(&seedval) % 300000);
10565
10566         if ((result = pthread_mutex_lock(&g_mutex)) != 0)
10567             exit_sys_thread("pthread_mutex_lock", result);
10568
10569         while (g_flag == 1)
10570             if ((result = pthread_cond_wait(&g_cond_producer, &g_mutex)) != 0)
10571                 exit_sys_thread("pthread_cond_wait", result);
10572
10573         g_shared = i;
10574         g_flag = 1;
10575
10576         if ((result = pthread_cond_signal(&g_cond_consumer)) != 0)
10577             exit_sys_thread("pthread_cond_signal", result);
10578
10579         if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
10580             exit_sys_thread("pthread_mutex_unlock", result);
10581
10582         if (i == 99)
10583             break;
10584         ++i;
10585     }
10586
10587     return NULL;
10588 }
10589
10590 void *thread_proc_consumer(void *param)
10591 {
10592     unsigned int seedval;
10593     int result, val;
10594
10595     seedval = (unsigned int)time(NULL);
10596
10597     for (;;) {
10598         if ((result = pthread_mutex_lock(&g_mutex)) != 0)
10599             exit_sys_thread("pthread_mutex_lock", result);
10600
10601         while (g_flag == 0)
10602             if ((result = pthread_cond_wait(&g_cond_consumer,
10603                                             &g_mutex)) != 0)
10604                 exit_sys_thread("pthread_cond_wait", result);
10605
10606         val = g_shared;
10607         g_flag = 0;
```

```
10608
10609     if ((result = pthread_cond_signal(&g_cond_producer)) != 0)
10610         exit_sys_thread("pthread_cond_signal", result);
10611
10612     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
10613         exit_sys_thread("pthread_mutex_unlock", result);
10614
10615     usleep(rand_r(&seedval) % 300000);
10616     printf("%d ", val);
10617     fflush(stdout);
10618     if (val == 99)
10619         break;
10620 }
10621 printf("\n");
10622
10623 return NULL;
10624 }
10625
10626 /
*-----*
-----*
10627 Üretici-Tüketicisi probleminde ortada paylaşılan alan tek bir değişken
10628 yerine bir kuyruk sistemi olabilir. Dolayısıyla bu durumda
10629 toplam bloke miktarı azalacak ve performans yükselecektir. Aşağıdaki
10630 örnekte kuyruk sistemi index kaydırma yöntemiyle gerçekleştirilmişdir. ➔
10631
10632 #include <stdio.h>
10633 #include <stdlib.h>
10634 #include <string.h>
10635 #include <unistd.h>
10636 #include <pthread.h>
10637
10638 #define QSIZE      10
10639
10640 void exit_sys_thread(const char *msg, int err);
10641 void *thread_proc_producer(void *param);
10642 void *thread_proc_consumer(void *param);
10643
10644 pthread_mutex_t g_mutex = PTHREAD_MUTEX_INITIALIZER;
10645 pthread_cond_t g_cond_producer = PTHREAD_COND_INITIALIZER;
10646 pthread_cond_t g_cond_consumer = PTHREAD_COND_INITIALIZER;
10647
10648 int g_queue[QSIZE];
10649 int g_count;
10650 int g_head, g_tail;
10651
10652 int main(void)
10653 {
```

```
10654     int result;
10655     pthread_t tid_producer, tid_consumer;
10656
10657     if ((result = pthread_create(&tid_producer, NULL, thread_proc_producer, ↵
10658                               NULL)) != 0)
10659         exit_sys_thread("pthread_create", result);
10660
10661     if ((result = pthread_create(&tid_consumer, NULL, thread_proc_consumer, ↵
10662                               NULL)) != 0)
10663         exit_sys_thread("pthread_create", result);
10664
10665     if ((result = pthread_join(tid_producer, NULL)) != 0)
10666         exit_sys_thread("pthread_join", result);
10667
10668     if ((result = pthread_join(tid_consumer, NULL)) != 0)
10669         exit_sys_thread("pthread_join", result);
10670
10671     pthread_cond_destroy(&g_cond_producer);
10672     pthread_cond_destroy(&g_cond_consumer);
10673     pthread_mutex_destroy(&g_mutex);
10674 }
10675
10676 void exit_sys_thread(const char *msg, int err)
10677 {
10678     fprintf(stderr, "%s: %s\n", msg, strerror(err));
10679     exit(EXIT_FAILURE);
10680 }
10681
10682 void *thread_proc_producer(void *param)
10683 {
10684     unsigned int seedval;
10685     int result, i;
10686
10687     seedval = (unsigned int)time(NULL) + 20000;
10688
10689     i = 0;
10690     for (;;) {
10691         usleep(rand_r(&seedval) % 300000);
10692
10693         if ((result = pthread_mutex_lock(&g_mutex)) != 0)
10694             exit_sys_thread("pthread_mutex_lock", result);
10695
10696         while (g_count == QSIZE)
10697             if ((result = pthread_cond_wait(&g_cond_producer, &g_mutex)) != ↵
10698                 0)
10699                 exit_sys_thread("pthread_cond_wait", result);
10700
10701         g_queue[g_tail++] = i;
10702         g_tail %= QSIZE;
10703         ++g_count;
10704 }
```

```
10704     if ((result = pthread_cond_signal(&g_cond_consumer)) != 0)
10705         exit_sys_thread("pthread_cond_signal", result);
10706
10707     if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
10708         exit_sys_thread("pthread_mutex_unlock", result);
10709
10710     if (i == 99)
10711         break;
10712     ++i;
10713 }
10714
10715 return NULL;
10716 }
10717
10718 void *thread_proc_consumer(void *param)
10719 {
10720     unsigned int seedval;
10721     int result, val;
10722
10723     seedval = (unsigned int)time(NULL);
10724
10725     for (;;) {
10726         if ((result = pthread_mutex_lock(&g_mutex)) != 0)
10727             exit_sys_thread("pthread_mutex_lock", result);
10728
10729         while (g_count == 0)
10730             if ((result = pthread_cond_wait(&g_cond_consumer,
10731                                             &g_mutex)) != 0)
10732                 exit_sys_thread("pthread_cond_wait", result);
10733
10734         val = g_queue[g_head++];
10735         g_head %= QSIZE;
10736         --g_count;
10737
10738         if ((result = pthread_cond_signal(&g_cond_producer)) != 0)
10739             exit_sys_thread("pthread_cond_signal", result);
10740
10741         if ((result = pthread_mutex_unlock(&g_mutex)) != 0)
10742             exit_sys_thread("pthread_mutex_unlock", result);
10743
10744         usleep(rand_r(&seedval) % 300000);
10745         printf("%d ", val);
10746         fflush(stdout);
10747         if (val == 99)
10748             break;
10749     }
10750     printf("\n");
10751
10752 }
10753
10754 /-----  
*-----
```

```
-----  
10755 Durum değişkenleri için de attribute girilebilir. Attrbute oluşturmak →  
    içib pthread_condattr_t türünden bir nesne alınır.  
10756 Önce bu nesne pthread_condattr_init ile ilkdeğerlenir. SOnra →  
    pthread_condattr_setxxx fonksiyonlarıyla özellikler set edilir.  
10757 İşte bu attribute nesnesi durum değişken nesnesi pthread_cond_init →  
    fonksiyonuyla yaratılırken ona parametre olarak geçilmektedir.  
10758 Durum değişken nesnesi yaratıldıktan sonra pthread_condattr_destroy →  
    fonksiyonu ile attribute nesnesi yok edilir. Aslında toplamda  
10759 yalnızca bir tek özellik vardır. O da durum değişken nesnesinin →  
    proseslerarası kullanımı ile ilgilidir. pthread_condattr_setpshared →  
10760 fonlsiyonunda PTHREAD_PROCESS_SHARED parametresi kullanılırsa durum →  
    değişken nesnesi prosesler arasında kullanılabilmektedir.  
10761 Aşağıdaki örnekte bir paylaşilan bellek alanı oluşturulmuş. Sonra →  
    üretici-tüketicisi problemi için gereken tüm nesneler bu paylaşilan →  
10762 bellek alanında yaratılmıştır. Böylece iki farklı proses kendi →  
    aralarında üretici tüketici işlemlerini yapmaktadır.  
10763 -----*/  
10764 /* producer.c */  
10766  
10767 #include <stdio.h>  
10768 #include <stdlib.h>  
10769 #include <string.h>  
10770 #include <time.h>  
10771 #include <fcntl.h>  
10772 #include <sys/stat.h>  
10773 #include <unistd.h>  
10774 #include <pthread.h>  
10775 #include <sys/mman.h>  
10776  
10777 #define SHM_PATH          "/shared_memory_cond_test"  
10778 #define QSIZE             10  
10779  
10780 typedef struct {  
10781     pthread_mutex_t mutex;  
10782     pthread_cond_t cond_producer;  
10783     pthread_cond_t cond_consumer;  
10784     int queue[QSIZE];  
10785     int head;  
10786     int tail;  
10787     int count;  
10788 } PROD_CONS;  
10789  
10790 void exit_sys(const char *msg);  
10791 void exit_sys_thread(const char *msg, int err);  
10792  
10793 int main(void)  
10794 {  
10795     int fdshm;  
10796     void *addr;  
10797     pthread_mutexattr_t mattr;
```

```
10798     pthread_condattr_t cattr;
10799     PROD_CONS *prod_cons;
10800     int result;
10801     int i;
10802
10803     srand(time(NULL));
10804
10805     if ((fdshm = shm_open(SHM_PATH, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR|S_IRGRP| ↵
10806         S_IROTH)) == -1)
10807         exit_sys("shm_open");
10808
10809     if (ftruncate(fdshm, 4096) == -1)
10810         exit_sys("ftruncate");
10811
10812     if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fdshm,    ↵
10813         0)) == MAP_FAILED)
10814         exit_sys("mmap");
10815
10816     prod_cons = (PROD_CONS *)addr;
10817
10818     pthread_mutexattr_init(&mattr);
10819     if ((result = pthread_mutexattr_setpshared(&mattr,           ↵
10820         PTHREAD_PROCESS_SHARED)) != 0)
10821         exit_sys_thread("pthread_mutexattr_setpshared", result);
10822
10823     pthread_mutexattr_destroy(&mattr);
10824
10825     if ((result = pthread_condattr_init(&cattr)) != 0)
10826         exit_sys_thread("pthread_condattr_init", result);
10827
10828     if ((result = pthread_condattr_setpshared(&cattr,           ↵
10829         PTHREAD_PROCESS_SHARED )) != 0)
10830         exit_sys_thread("pthread_condattr_setpshared", result);
10831
10832     if ((result = pthread_cond_init(&prod_cons->cond_producer, &cattr)) !=  ↵
10833         0)
10834         exit_sys_thread("pthread_condinit", result);
10835
10836     if ((result = pthread_cond_init(&prod_cons->cond_consumer, &cattr)) !=  ↵
10837         0)
10838         exit_sys_thread("pthread_condinit", result);
10839
10840     i = 0;
10841     for (;;) {
10842         usleep(rand() % 300000);
10843
10844         if ((result = pthread_mutex_lock(&prod_cons->mutex)) != 0)
10845             exit_sys_thread("pthread_mutex_lock", result);
```

```
10845
10846     while (prod_cons->count == QSIZE)
10847         if ((result = pthread_cond_wait(&prod_cons->cond_producer,
10848                                         &prod_cons->mutex)) != 0)
10849             exit_sys_thread("pthread_cond_wait", result);
10849
10850     prod_cons->queue[prod_cons->tail++] = i;
10851     prod_cons->tail %= QSIZE;
10852     ++prod_cons->count;
10853
10854     if ((result = pthread_cond_signal(&prod_cons->cond_consumer)) != 0)
10855         exit_sys_thread("pthread_cond_signal", result);
10856
10857     if ((result = pthread_mutex_unlock(&prod_cons->mutex)) != 0)
10858         exit_sys_thread("pthread_mutex_unlock", result);
10859
10860     if (i == 99)
10861         break;
10862     ++i;
10863 }
10864
10865 sleep(5);
10866
10867 pthread_cond_destroy(&prod_cons->cond_producer);
10868 pthread_cond_destroy(&prod_cons->cond_consumer);
10869 pthread_mutex_destroy(&prod_cons->mutex);
10870 munmap(addr, 4096);
10871 close(fdshm);
10872 if (shm_unlink(SHM_PATH) == -1)
10873     exit_sys("shm_unlink");
10874
10875 return 0;
10876 }
10877
10878 void exit_sys_thread(const char *msg, int err)
10879 {
10880     fprintf(stderr, "%s: %s\n", msg, strerror(err));
10881     exit(EXIT_FAILURE);
10882 }
10883
10884 void exit_sys(const char *msg)
10885 {
10886     perror(msg);
10887
10888     exit(EXIT_FAILURE);
10889 }
10890
10891 /* consumer.c */
10892
10893 #include <stdio.h>
10894 #include <stdlib.h>
10895 #include <string.h>
10896 #include <time.h>
```

```
10897 #include <fcntl.h>
10898 #include <sys/stat.h>
10899 #include <unistd.h>
10900 #include <pthread.h>
10901 #include <sys/mman.h>
10902
10903 #define SHM_PATH      "/shared_memory_cond_test"
10904 #define QSIZE          10
10905
10906 typedef struct {
10907     pthread_mutex_t mutex;
10908     pthread_cond_t cond_producer;
10909     pthread_cond_t cond_consumer;
10910     int queue[QSIZE];
10911     int head;
10912     int tail;
10913     int count;
10914 } PROD_CONS;
10915
10916 void exit_sys(const char *msg);
10917 void exit_sys_thread(const char *msg, int err);
10918
10919 int main(void)
10920 {
10921     int fdshm;
10922     void *addr;
10923     PROD_CONS *prod_cons;
10924     int result;
10925     int val;
10926
10927     srand(time(NULL));
10928
10929     if ((fdshm = shm_open(SHM_PATH, O_RDWR, 0)) == -1)
10930         exit_sys("shm_open");
10931
10932     if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fdshm, 0)) == MAP_FAILED)
10933         exit_sys("mmap");
10934
10935     prod_cons = (PROD_CONS *)addr;
10936
10937     for (;;) {
10938         if ((result = pthread_mutex_lock(&prod_cons->mutex)) != 0)
10939             exit_sys_thread("pthread_mutex_lock", result);
10940
10941         while (prod_cons->count == 0)
10942             if ((result = pthread_cond_wait(&prod_cons->cond_consumer, &prod_cons->mutex)) != 0)
10943                 exit_sys_thread("pthread_cond_wait", result);
10944
10945         val = prod_cons->queue[prod_cons->head++];
10946         prod_cons->head %= QSIZE;
10947         --prod_cons->count;
```

```
10948
10949     if ((result = pthread_cond_signal(&prod_cons->cond_producer)) != 0)
10950         exit_sys_thread("pthread_cond_signal", result);
10951
10952     if ((result = pthread_mutex_unlock(&prod_cons->mutex)) != 0)
10953         exit_sys_thread("pthread_mutex_unlock", result);
10954
10955     usleep(rand() % 300000);
10956
10957     printf("%d ", val);
10958     fflush(stdout);
10959     if (val == 99)
10960         break;
10961 }
10962
10963 printf("\n");
10964
10965 close(fdshm);
10966
10967 return 0;
10968 }
10969
10970 void exit_sys_thread(const char *msg, int err)
10971 {
10972     fprintf(stderr, "%s: %s\n", msg, strerror(err));
10973     exit(EXIT_FAILURE);
10974 }
10975
10976 void exit_sys(const char *msg)
10977 {
10978     perror(msg);
10979
10980     exit(EXIT_FAILURE);
10981 }
10982
10983 /
*-----*-----*-----*
```

10984 barrier nesneleri n tane thread'in belli bir noktaya geldikten sonra
yollarına devam etmesi için düşünülmüştür. Barrier nesni
pthread_barrier_t türüyle temsil edilmiştir. Bu nesne
pthread_barrier_init fonksiyonuyla ilkdeğerlenir. Bu fonksiyonda bir
sayac değeri
10985 de belirtilmektedir. Bundan sonra bekleme işlemi için
pthread_barrier_wait fonksiyonu kullanılır. İşte sayac değeri n olmak
üzere
10986 n tane thread'e kadar her thread pthread_barrier_wait fonksiyonunda
bloke olarak bekler. En son n'inci thread de pthread_barrier_wait
10987 çağrısına geldiğinde uyuyan tüm thread'ler uyanarak yoluna devam
ederler. Bu nesne genellikle bir işin çeşitli parçalarını yapan
thread'lerin
10988 kendi işlerini bitirdikten sonra bekletilmeleri için kullanılmaktadır. Bu
tür uygulamalarda genellikle bu thread'lardan bir tanesi

```
10990     barrier çıkışında özel bir işlem yapar. Programcı bu thread'i kendisi →  
10991         belirleyebilir. Ancak barrier tasarımcıları bu işi kolaşlaştırmak →  
10991     için n thread'in n - 1 tanesini 0 ile yalnızca herhangi bir tanesini →  
10991         PTHREAD_BARRIER_SERIAL_THREAD değeri ile geri dönmektedir. →  
10991     Programcının →  
10992     bu değeri error değeri olarak yotumlamaması gereklidir. Aşağıdaki örnekte 4 →  
10992         farklı thread (main thread de dahil) bir barrier nesnesine ulaşana →  
10992         kadar →  
10993     bekletilmistiştir.  
10994 -----*/  
10995  
10996 #include <stdio.h>  
10997 #include <stdlib.h>  
10998 #include <string.h>  
10999 #include <unistd.h>  
11000 #include <pthread.h>  
11001  
11002 void exit_sys_thread(const char *msg, int err);  
11003 void *thread_proc1(void *param);  
11004 void *thread_proc2(void *param);  
11005 void *thread_proc3(void *param);  
11006  
11007 pthread_barrier_t g_barrier;  
11008  
11009 int main(void)  
11010 {  
11011     int result;  
11012     pthread_t tid1, tid2, tid3;  
11013  
11014     if ((result = pthread_barrier_init(&g_barrier, NULL, 4)) != 0)  
11015         exit_sys_thread("pthread_barrier_init", result);  
11016  
11017     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)  
11018         exit_sys_thread("pthread_create", result);  
11019  
11020     sleep(1);  
11021  
11022     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)  
11023         exit_sys_thread("pthread_create", result);  
11024  
11025     sleep(1);  
11026  
11027     if ((result = pthread_create(&tid3, NULL, thread_proc3, NULL)) != 0)  
11028         exit_sys_thread("pthread_create", result);  
11029  
11030     sleep(1);  
11031  
11032     printf("main threads enters barrier...\n");  
11033  
11034     if ((result = pthread_barrier_wait(&g_barrier)) != 0 && result != →  
11034         PTHREAD_BARRIER_SERIAL_THREAD)  
11035         exit_sys_thread("pthread_barrier_wait", result);
```

```
11036     printf("main thread exits barrier...\n");
11037
11038     if ((result = pthread_join(tid1, NULL)) != 0)
11039         exit_sys_thread("pthread_join", result);
11040
11041     if ((result = pthread_join(tid2, NULL)) != 0)
11042         exit_sys_thread("pthread_join", result);
11043
11044     if ((result = pthread_join(tid3, NULL)) != 0)
11045         exit_sys_thread("pthread_join", result);
11046
11047     pthread_barrier_destroy(&g_barrier);
11048
11049     return 0;
11050 }
11051
11052 void exit_sys_thread(const char *msg, int err)
11053 {
11054     fprintf(stderr, "%s: %s\n", msg, strerror(err));
11055     exit(EXIT_FAILURE);
11056 }
11057
11058 void *thread_proc1(void *param)
11059 {
11060     int result;
11061
11062     printf("thread1 enters barrier...\n");
11063
11064     if ((result = pthread_barrier_wait(&g_barrier)) != 0 && result != PTHREAD_BARRIER_SERIAL_THREAD)
11065         exit_sys_thread("pthread_barrier_wait", result);
11066
11067     printf("thread1 exits barrier...\n");
11068
11069     return NULL;
11070 }
11071
11072 void *thread_proc2(void *param)
11073 {
11074     int result;
11075
11076     printf("thread2 enters barrier...\n");
11077
11078     if ((result = pthread_barrier_wait(&g_barrier)) != 0 && result != PTHREAD_BARRIER_SERIAL_THREAD)
11079         exit_sys_thread("pthread_barrier_wait", result);
11080
11081     printf("thread2 exits barrier...\n");
11082
11083     return NULL;
11084 }
11085
11086 }
```

```
11087 void *thread_proc3(void *param)
11088 {
11089     int result;
11090
11091     printf("thread3 enters barrier...\n");
11092
11093     if ((result = pthread_barrier_wait(&g_barrier)) != 0 && result != PTHREAD_BARRIER_SERIAL_THREAD)
11094         exit_sys_thread("pthread_barrier_wait", result);
11095
11096     printf("thread3 exits barrier...\n");
11097
11098
11099     return NULL;
11100 }
11101
11102 /
*-----*
-----*
11103 Aşağıda bir barrier kullanım örneği verilmiştir. 50000000 rastgele      ↵
11104 değerlerden oluşan int türden bir dizi 10 oarcaya      ↵
11105 bölünmüş ve her parça bir thread tarafından qsort fonksiyonu ile sort      ↵
11106 edilmiştir. Sonra da bu 10 kendi aralarında sıralı olan      ↵
11107 parçalar birleştirilmiştir. 4 çekirdekli bir Linux sisteminde (sanal      ↵
11108 makine) toplam zaman 8.50 saniye civarındadır. Daha sonra      ↵
11109 aynı dizi tek bir thread'le (ana thread'le) yine qsort fonksiyonu      ↵
11110 kullanılarak sıraya dizilmiştir. Bu da 26.36 saniye civarında      ↵
11111 bir zaman almıştır. Görüldüğü gibi thread'li versiyonda yaklaşık 3 kat      ↵
11112 daha hızlı sonuç elde edilmiştir.
11108 -----*/
11109
11110 #include <stdio.h>
11111 #include <stdlib.h>
11112 #include <string.h>
11113 #include <time.h>
11114 #include <unistd.h>
11115 #include <pthread.h>
11116
11117 /* Thread'siz 50000000 için 11.30 saniye (100000000 için 43.1)*/
11118
11119 #define SIZE          50000000
11120 #define NTHREADS       10
11121
11122 void exit_sys_thread(const char *msg, int err);
11123 void *thread_proc(void *param);
11124 int comp(const void *pv1, const void *pv2);
11125 void merge(void);
11126 int check(void);
11127
11128 pthread_barrier_t g_barrier;
11129 int g_nums[SIZE];
11130 int g_snums[SIZE];
```

```
11131
11132 int main(void)
11133 {
11134     int result;
11135     int i;
11136     pthread_t tids[NTHREADS];
11137
11138     srand(time(NULL));
11139     for (i = 0; i < SIZE; ++i)
11140         g_nums[i] = rand();
11141
11142     if ((result = pthread_barrier_init(&g_barrier, NULL, NTHREADS)) != 0)
11143         exit_sys_thread("pthread_barrier_init", result);
11144
11145     for (i = 0; i < NTHREADS; ++i)
11146         if ((result = pthread_create(&tids[i], NULL, thread_proc, (void *) ↵
11147             i)) != 0)
11148             exit_sys_thread("pthread_create", result);
11149
11150     for (i = 0; i < NTHREADS; ++i)
11151         if ((result = pthread_join(tids[i], NULL)) != 0)
11152             exit_sys_thread("pthread_join", result);
11153
11154     pthread_barrier_destroy(&g_barrier);
11155
11156     printf(check() ? "Sorted\n" : "Not Sorted\n");
11157
11158     return 0;
11159 }
11160 void exit_sys_thread(const char *msg, int err)
11161 {
11162     fprintf(stderr, "%s: %s\n", msg, strerror(err));
11163     exit(EXIT_FAILURE);
11164 }
11165
11166 void *thread_proc(void *param)
11167 {
11168     int part = (int)param;
11169     int result;
11170
11171     qsort(g_snums + part * (SIZE / NTHREADS), SIZE / NTHREADS, sizeof(int), ↵
11172         comp);
11173
11174     result = pthread_barrier_wait(&g_barrier);
11175     if (result != 0 && result != PTHREAD_BARRIER_SERIAL_THREAD)
11176         exit_sys_thread("pthread_barrier_wait", result);
11177     if (result == PTHREAD_BARRIER_SERIAL_THREAD)
11178         merge();
11179
11180     /* other jobs... */
11181
11182     return NULL;
```

```
11182 }
11183
11184 int comp(const void *pv1, const void *pv2)
11185 {
11186     const int *pi1 = (const int *)pv1;
11187     const int *pi2 = (const int *)pv2;
11188
11189     return *pi1 - *pi2;
11190 }
11191
11192 void merge(void)
11193 {
11194     int indexes[NTHREADS];
11195     int min, min_index;
11196     int i, k;
11197     int partsize;
11198
11199     partsize = SIZE / NTHREADS;
11200
11201     for (i = 0; i < NTHREADS; ++i)
11202         indexes[i] = i * partsize;
11203
11204     for (i = 0; i < SIZE; ++i) {
11205         min = indexes[0];
11206         min_index = 0;
11207
11208         for (k = 1; k < NTHREADS; ++k)
11209             if (indexes[k] < (k + 1) * partsize && g_nums[indexes[k]] < min) {
11210                 {
11211                     min = g_nums[indexes[k]];
11212                     min_index = k;
11213                 }
11214             g_snums[i] = min;
11215             ++indexes[min_index];
11216         }
11217     }
11218     int check(void)
11219 {
11220     int i;
11221
11222     for (i = 0; i < SIZE - 1; ++i)
11223         if (g_snums[i] > g_snums[i + 1])
11224             return 0;
11225
11226     return 1;
11227 }
11228
11229 /
*-----
```

```
11231     sem_wait fonksiyonu eğer semaphore sayısı 0 ise blokeye yol açar. Eğer ↵
11232         semaphore sayacı 0'dan büyüğse bu fonksiyon sayacı 1 eksilterek ↵
11233         kritik koda girişe izin verir. sem_post ise semaphore sayacını 1 ↵
11234         artırmaktadır. Semaphore'un başlangıçtaki sayaco sem_init ya da ↵
11235         sem_open ↵
11236         fonksiyonlarında belirtilir. Bu değer kritik koda en fazla kaç thread ↵
11237         akışının girebileceğini belirtmektedir. Semaphore sayacı 1 ↵
11238         olan semaphore'lara "binary semaphore" denilmektedir. Binary ↵
11239         semaphore'lar mutex nesnelerine benzer. Fakat mutex nesnelerinin ↵
11240         thread temelinde ↵
11241         sahipliği vardır. Yani başka bir thread mutex'in sahipliğini bırakamaz. ↵
11242         Ancak semaphore'lar böyle değildir. Bu nedenle üretici-tüketicili ↵
11243         tarzı problemler semaphore nesneleri ile çok kolay çözülebilmiştir. ↵
11244         Semaphore nesneleri tipik olarak n tane kaynağın thread'lere ↵
11245         paylaşılması için tercih edilmelidir. Semaphore nesneleri kullanım ↵
11246         bittikten sonra sem_destroy fonksiyonuna yok edilmelidir. ↵
11247         Aşağıda bir binary semaphore örneği verilmiştir. ↵
11248
11249         -----
11250         -----
11251         -----
11252         -----
11253         -----
11254         -----
11255         -----
11256         -----
11257         -----
11258         int main(void)
11259     {
11260         int result;
11261         int i;
11262         pthread_t tid1, tid2;
11263
11264         if (sem_init(&g_sem, 0, 1) == -1)
11265             exit_sys("sem_init");
11266
11267         if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
11268             exit_sys_thread("pthread_create", result);
11269
11270         if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
11271             exit_sys_thread("pthread_create", result);
11272
11273         if ((result = pthread_join(tid1, NULL)) != 0)
```

```
11274         exit_sys_thread("pthread_join", result);
11275
11276     if ((result = pthread_join(tid2, NULL)) != 0)
11277         exit_sys_thread("pthread_join", result);
11278
11279     sem_destroy(&g_sem);
11280
11281     printf("%d\n", g_count);
11282
11283     return 0;
11284 }
11285
11286 void exit_sys(const char *msg)
11287 {
11288     perror(msg);
11289
11290     exit(EXIT_FAILURE);
11291 }
11292
11293 void exit_sys_thread(const char *msg, int err)
11294 {
11295     fprintf(stderr, "%s: %s\n", msg, strerror(err));
11296     exit(EXIT_FAILURE);
11297 }
11298
11299 void *thread_proc1(void *param)
11300 {
11301     int i;
11302
11303     for (i = 0; i < MAX; ++i) {
11304         if (sem_wait(&g_sem) == 1)
11305             exit_sys("sem_wait");
11306
11307         ++g_count;
11308
11309         if (sem_post(&g_sem) == 1)
11310             exit_sys("sem_post");
11311     }
11312
11313     return NULL;
11314 }
11315
11316 void *thread_proc2(void *param)
11317 {
11318     int i;
11319
11320     for (i = 0; i < MAX; ++i) {
11321         if (sem_wait(&g_sem) == 1)
11322             exit_sys("sem_wait");
11323
11324         ++g_count;
11325
11326         if (sem_post(&g_sem) == 1)
```

```
11327         exit_sys("sem_post");
11328     }
11329
11330     return NULL;
11331 }
11332
11333 /
*-----*
-----*
11334     Aşağıdaki örnekte global değişken yerine heap'te tahsis edilen bir alan
      thread'lere parametre yoluyla aktarılmıştır.
11335 -----*/
11336
11337 #include <stdio.h>
11338 #include <stdlib.h>
11339 #include <string.h>
11340 #include <pthread.h>
11341 #include <semaphore.h>
11342
11343 #define MAX      1000000
11344
11345 void exit_sys(const char *msg);
11346 void exit_sys_thread(const char *msg, int err);
11347 void *thread_proc1(void *param);
11348 void *thread_proc2(void *param);
11349
11350 typedef struct {
11351     sem_t sem;
11352     int count;
11353 } THREAD_PARAM;
11354
11355 int main(void)
11356 {
11357     int result;
11358     pthread_t tid1, tid2;
11359     THREAD_PARAM *tparam;
11360
11361     if ((tparam = (THREAD_PARAM *)malloc(sizeof(THREAD_PARAM))) == NULL) {
11362         fprintf(stderr, "cannot allocate memory!..\\n");
11363         exit(EXIT_FAILURE);
11364     }
11365
11366     tparam->count = 0;
11367
11368     if (sem_init(&tparam->sem, 0, 1) == -1)
11369         exit_sys("sem_init");
11370
11371     if ((result = pthread_create(&tid1, NULL, thread_proc1, tparam)) != 0)
11372         exit_sys_thread("pthread_create", result);
11373
11374     if ((result = pthread_create(&tid2, NULL, thread_proc2, tparam)) != 0)
11375         exit_sys_thread("pthread_create", result);
```

```
11376
11377     if ((result = pthread_join(tid1, NULL)) != 0)
11378         exit_sys_thread("pthread_join", result);
11379
11380     if ((result = pthread_join(tid2, NULL)) != 0)
11381         exit_sys_thread("pthread_join", result);
11382
11383     sem_destroy(&tparam->sem);
11384
11385     printf("%d\n", tparam->count);
11386
11387     free(tparam);
11388
11389     return 0;
11390 }
11391
11392 void exit_sys(const char *msg)
11393 {
11394     perror(msg);
11395
11396     exit(EXIT_FAILURE);
11397 }
11398
11399 void exit_sys_thread(const char *msg, int err)
11400 {
11401     fprintf(stderr, "%s: %s\n", msg, strerror(err));
11402     exit(EXIT_FAILURE);
11403 }
11404
11405 void *thread_proc1(void *param)
11406 {
11407     int i;
11408     THREAD_PARAM *tparam = (THREAD_PARAM *)param;
11409
11410     for (i = 0; i < MAX; ++i) {
11411         if (sem_wait(&tparam->sem) == -1)
11412             exit_sys("sem_wait");
11413
11414         ++tparam->count;
11415
11416         if (sem_post(&tparam->sem) == -1)
11417             exit_sys("sem_post");
11418     }
11419
11420     return NULL;
11421 }
11422
11423 void *thread_proc2(void *param)
11424 {
11425     int i;
11426     THREAD_PARAM *tparam = (THREAD_PARAM *)param;
11427
11428     for (i = 0; i < MAX; ++i) {
```

```
11429         if (sem_wait(&tparam->sem) == -1)
11430             exit_sys("sem_wait");
11431
11432         ++tparam->count;
11433
11434         if (sem_post(&tparam->sem) == -1)
11435             exit_sys("sem_post");
11436     }
11437
11438     return NULL;
11439 }
11440
11441 /
*-----*
-----*
11442     Üretici-Tüketici problemi semaphore nesneleri ile daha sade bir biçimde
11443         çözülebilir. Aşağıdaki örnekte QSIZE kadar bir kuyruk
11444         oluşturulmuştur. Üretici bu kuyruğa elde ettiği değerleri eklerken
11445             tüketici de kuyruktan değerleri almaktadır. İki semaphore
11446         kullanılmıştır. Üretici semaphore'unun sayacı başlangıçta kuyruk
11447             uzunluğu (QSIZE), tüketici semaphore'unun sayacı ise başlangıçta
11448             0 durumundadır. Üretici tüketicinin, tüketici de üreticinin semaphore
11449             sayaçlarını sem_post ile artırmaktadır. Kuyruktaki eleman
11450             sayısını tutmanın bir gereği yoktur. Üretici ve tüketici kuyrupta aynı
11451             bölge üzerinde çalışmadıklarından aynı anda kuyruğa erişimlerinde
11452             bir sorun olmayacağından emin olabiliriz.
11453
11454 -----*/
11455
11456 #include <stdio.h>
11457 #include <stdlib.h>
11458 #include <string.h>
11459 #include <unistd.h>
11460 #include <pthread.h>
11461 #include <semaphore.h>
11462
11463 #define QSIZE      10
11464
11465 void exit_sys(const char *msg);
11466 void exit_sys_thread(const char *msg, int err);
11467 void *thread_proc_producer(void *param);
11468 void *thread_proc_consumer(void *param);
11469
11470 sem_t g_sem_producer;
11471 sem_t g_sem_consumer;
11472
11473 int g_queue[QSIZE];
11474 int g_head, g_tail;
11475
11476 int main(void)
11477 {
11478     int result;
11479     pthread_t tid_producer, tid_consumer;
```

```
11474
11475     if ((result = sem_init(&g_sem_producer, 0, QSIZE)) != 0)
11476         exit_sys_thread("sem_init", result);
11477
11478     if ((result = sem_init(&g_sem_consumer, 0, 0)) != 0)
11479         exit_sys_thread("sem_init", result);
11480
11481     if ((result = pthread_create(&tid_producer, NULL, thread_proc_producer, ↴
11482                                     NULL)) != 0)
11483         exit_sys_thread("pthread_create", result);
11484
11485     if ((result = pthread_create(&tid_consumer, NULL, thread_proc_consumer, ↴
11486                                     NULL)) != 0)
11487         exit_sys_thread("pthread_create", result);
11488
11489     if ((result = pthread_join(tid_producer, NULL)) != 0)
11490         exit_sys_thread("pthread_join", result);
11491
11492     if ((result = pthread_join(tid_consumer, NULL)) != 0)
11493         exit_sys_thread("pthread_join", result);
11494
11495     sem_destroy(&g_sem_producer);
11496     sem_destroy(&g_sem_consumer);
11497
11498 }
11499 void exit_sys(const char *msg)
11500 {
11501     perror(msg);
11502
11503     exit(EXIT_FAILURE);
11504 }
11505
11506 void exit_sys_thread(const char *msg, int err)
11507 {
11508     fprintf(stderr, "%s: %s\n", msg, strerror(err));
11509     exit(EXIT_FAILURE);
11510 }
11511
11512 void *thread_proc_producer(void *param)
11513 {
11514     unsigned int seedval;
11515     int i;
11516
11517     seedval = (unsigned int)time(NULL) + 20000;
11518
11519     i = 0;
11520     for (;;) {
11521         usleep(rand_r(&seedval) % 300000);
11522
11523         if (sem_wait(&g_sem_producer) == -1)
11524             exit_sys("sem_wait");
```

```
11525
11526     g_queue[g_tail++] = i;
11527     g_tail %= QSIZE;
11528
11529     if (sem_post(&g_sem_consumer) == 1)
11530         exit_sys("sem_post");
11531
11532     if (i == 99)
11533         break;
11534     ++i;
11535 }
11536
11537     return NULL;
11538 }
11539
11540 void *thread_proc_consumer(void *param)
11541 {
11542     unsigned int seedval;
11543     int val;
11544
11545     seedval = (unsigned int)time(NULL);
11546
11547     for (;;) {
11548         if (sem_wait(&g_sem_consumer) == -1)
11549             exit_sys("sem_wait");
11550
11551         val = g_queue[g_head++];
11552         g_head %= QSIZE;
11553
11554         if (sem_post(&g_sem_producer) == -1)
11555             exit_sys("sem_wait");
11556
11557         usleep(rand_r(&seedval) % 300000);
11558         printf("%d ", val);
11559         fflush(stdout);
11560         if (val == 99)
11561             break;
11562     }
11563     printf("\n");
11564
11565     return NULL;
11566 }
11567
11568 /
*-----*-----*-----*
```

11569 POSIX semaphore nesneleri iki biçimde prosesler arasında kullanılır. ↗
11570 Birincisi semaphore nesnesini sem_init ile paylaşılan bellek alanında
pshared parametresi sıfır dışı geçilerek yaratmaktadır. İkincisi POSIX ↗
paylaşılan bellek alanları ya da mesaj kuyruklarında olduğu gibi
kök dizinde bir dosya ismi vererek sem_open fonksiyonuyla yaratmak ve ↗
açmaktadır. (Buna "isimli semaphore'lar" da denilmektedir.)
11572 sem_open fonksiyonuyla yaratılan ya da açılan semaphore nesneleri ↗

```
sem_close ile kapatılmalıdır (sem_destroy ile değil).
11573 Aşağıda üretici tüketici probleminin isimli semaphore'larla prosesler  ↵
     arasında geçekleştirimine ilişkin örnek bir kod bulunmaktadır.
11574 -----
-----*/
11575
11576 /* producer.c */
11577
11578 #include <stdio.h>
11579 #include <stdlib.h>
11580 #include <string.h>
11581 #include <time.h>
11582 #include <fcntl.h>
11583 #include <unistd.h>
11584 #include <sys/mman.h>
11585 #include <semaphore.h>
11586
11587 #define SHM_PATH          "/shared_memory_producer_consumer"
11588 #define SEM_PATH_PRODUCER "/semaphore_producer"
11589 #define SEM_PATH_CONSUMER "/semaphore_consumer"
11590
11591 #define QSIZE      10
11592
11593 void exit_sys(const char *msg);
11594 typedef struct {
11595     int queue[QSIZE];
11596     int head;
11597     int tail;
11598 } PROD_CONS;
11599
11600 int main(void)
11601 {
11602     int fdshm;
11603     void *addr;
11604     sem_t *sem_producer, *sem_consumer;
11605     PROD_CONS *prod_cons;
11606     int i;
11607
11608     srand(time(NULL));
11609
11610     if ((fdshm = shm_open(SHM_PATH, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR|S_IRGRP| ↵
11611         S_IROTH)) == -1)
11612         exit_sys("shm_open");
11613
11614     if (ftruncate(fdshm, 4096) == -1)
11615         exit_sys("ftruncate");
11616
11617     if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fdshm,    ↵
11618         0)) == MAP_FAILED)
11619         exit_sys("mmap");
11620
11621     prod_cons = (PROD_CONS *)addr;
```

```
11621     if ((sem_producer = sem_open(SEM_PATH_PRODUCER, O_CREAT|O_RDWR, S_IRUSR|  
11622             S_IWUSR|S_IRGRP|S_IROTH, QSIZE)) == NULL)  
11623         exit_sys("sem_open");  
11624     if ((sem_consumer = sem_open(SEM_PATH_CONSUMER, O_CREAT|O_RDWR, S_IRUSR|  
11625             S_IWUSR|S_IRGRP|S_IROTH, 0)) == NULL)  
11626         exit_sys("sem_open");  
11627     i = 0;  
11628     for (;;) {  
11629         usleep(rand() % 300000);  
11630         if (sem_wait(sem_producer) == -1)  
11631             exit_sys("sem_wait");  
11632         prod_cons->queue[prod_cons->tail++] = i;  
11633         prod_cons->tail %= QSIZE;  
11634  
11635         if (sem_post(sem_consumer) == -1)  
11636             exit_sys("sem_post");  
11637         if (i == 99)  
11638             break;  
11639         ++i;  
11640     }  
11641  
11642     munmap(addr, 4096);  
11643     close(fdshm);  
11644     if (shm_unlink(SHM_PATH) == -1)  
11645         exit_sys("shm_unlink");  
11646     sem_close(sem_producer);  
11647     sem_close(sem_consumer);  
11648  
11649     return 0;  
11650 }  
11651  
11652 void exit_sys(const char *msg)  
11653 {  
11654     perror(msg);  
11655  
11656     exit(EXIT_FAILURE);  
11657 }  
11658  
11659 /* consumer.c */  
11660  
11661 #include <stdio.h>  
11662 #include <stdlib.h>  
11663 #include <string.h>  
11664 #include <time.h>  
11665 #include <fcntl.h>  
11666 #include <unistd.h>  
11667 #include <sys/mman.h>  
11668 #include <semaphore.h>  
11669  
11670 #define SHM_PATH           "/shared_memory_producer_consumer"
```

```
11672 #define SEM_PATH_PRODUCER      "/semaphore_producer"
11673 #define SEM_PATH_CONSUMER       "/semaphore_consumer"
11674
11675 #define QSIZE          10
11676
11677 void exit_sys(const char *msg);
11678
11679 typedef struct {
11680     int queue[QSIZE];
11681     int head;
11682     int tail;
11683 } PROD_CONS;
11684
11685 int main(void)
11686 {
11687     int fdshm;
11688     void *addr;
11689     sem_t *sem_producer, *sem_consumer;
11690     PROD_CONS *prod_cons;
11691     int i;
11692
11693     srand(time(NULL));
11694
11695     if ((fdshm = shm_open(SHM_PATH, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)) == -1)
11696         exit_sys("shm_open");
11697
11698     if (ftruncate(fdshm, 4096) == -1)
11699         exit_sys("ftruncate");
11700
11701     if ((addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fdshm, 0)) == MAP_FAILED)
11702         exit_sys("mmap");
11703
11704     prod_cons = (PROD_CONS *)addr;
11705
11706     if ((sem_producer = sem_open(SEM_PATH_PRODUCER, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH, QSIZE)) == NULL)
11707         exit_sys("sem_open");
11708
11709     if ((sem_consumer = sem_open(SEM_PATH_CONSUMER, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH, 0)) == NULL)
11710         exit_sys("sem_open");
11711
11712     i = 0;
11713     for (;;) {
11714         usleep(rand() % 300000);
11715         if (sem_wait(sem_producer) == -1)
11716             exit_sys("sem_wait");
11717
11718         prod_cons->queue[prod_cons->tail++] = i;
11719         prod_cons->tail %= QSIZE;
11720 }
```

```
11721         if (sem_post(sem_consumer) == -1)
11722             exit_sys("sem_post");
11723         if (i == 99)
11724             break;
11725         ++i;
11726     }
11727
11728     munmap(addr, 4096);
11729     close(fdshm);
11730     if (shm_unlink(SHM_PATH) == -1)
11731         exit_sys("shm_unlink");
11732     sem_close(sem_producer);
11733     sem_close(sem_consumer);
11734
11735     return 0;
11736 }
11737
11738 void exit_sys(const char *msg)
11739 {
11740     perror(msg);
11741
11742     exit(EXIT_FAILURE);
11743 }
11744
11745 /
*-----*
```

11746 Sistem 5 semaphore'ları aslında 3 sistem IPC nesnesinin üçüncüsüdür. ↗
Genel kullanım biçimi sistm 5 paylaşılan bellek alanı ve
sistem 5 mesaj kuyruklarında olduğu gibidir. Semaphore kullanımı için
semget, semctl, semop isimli üç fonksiyondan faydalанılmaktadır.
11748 Sistem 5 semaphore'ları aynı prosesin thread'ler arasındaki
senkronizasyon için değil farklı prosesler arasındaki senkronizasyon
için
11749 kullanılmaktadır. (Zaten o zamanlar thread kavramı yoktu.) Arayüz biraz ↗
karışiktır. Bu karışıklığın en önemli nedeni bu fonksiyonların
11750 tek bir semaphore üzerinde değil bir grup semaphore üzerinde (semaphore ↗
set) işlem yapıyor olmasındandır. Ayrıca semaphore sayaç mekanizması ↗
da
11751 biraz karmaşık tasarlanmıştır. semget fonksiyonu yine anahtar ↗
verildiğinde id'nin elde edilmesi amacıyla kullanılır. Ancak semget
11752 fonksiyonu tek smaphore de♦♦♦♦♦il bir grup semaphore (semaphore set) ↗
yaratmaktadır. Semaphore kğmesinin kaç semaphore'dan oluşacağı
11753 sem get fonksiyonun ikinci parametresinde belirtilir.
11754
11755 int semget(key_t key, int nsems, int semflg);
11756
11757 Semahpore kümesi semget ile yaratıldıktan sonra küme içerisindeki ↗
semaphore'ların sayaçları set edilmelidir. Bunun için semctl
fonksiyonu
11758 kullanılır:
11759
11760

```
11761     int semctl(int semid, int semnum, int cmd, ...);
11762
11763     Bu fonksiyonun son parametresi (ellipsis yerindeki parametresi) →
11764         aşağıdaki gibi bir birlik (union) olmalıdır. Ancak bu birlik herhangi →
11765         bir başlık dosyasında bildirilmemiştir.
11766     Dolayısıyla programcı tarafından bildirilmelidir:
11767
11768     union semun {
11769         int val;
11770         struct semid_ds *buf;
11771         unsigned short  *array;
11772     } arg;
11773
11774     Fonksiyonun ikinci parametresi semaphore kümesi içerisindeki hangi →
11775         semaphore'un sayacı ile ilgili işlem yapılacağını belirtir.
11776     Kümedeki semaphore'ların numaraları 0'dan başlamaktadır. Üçüncü →
11777         parametre yapılacak işlemi belirtir. SETVAL semaphore sayacının set →
11778         edileceği →
11779         anlamına gelir. Bu durumda son parametredeki birliğin val elemanı set →
11780             edilecek semaphore sayaç değerini tutmalıdır. GETVAL benzer biçimde →
11781             ilgili semaphore'un sayaç değeri ile geri dönüleceği anlamına gelir. →
11782             SETALL tüm kümedeki semaphore'ların sayaç değerlerinin tek hamlede set →
11783             edileceği →
11784             anlamına gelmektedir. Bu durumda tüm sayaç değerleri short int bir →
11785                 diziye yazılmalı bu dizinin başlangıç adresi de birliğin array →
11786                 elemanında →
11787             bulunmalıdır. GETALL ise bunun tersini yapar. Tabii semctl'de cmd →
11788                 parametresi için IPC_RMID değeri de girilebilir. Bu da semaphore →
11789                 kümesinin →
11790             silineceği anlamına gelmektedir.
11791
11792     Sistem 5 semaphore'larında kritik kod semop fonksiyonuya →
11793         oluşturulmaktadır:
11794
11795     int semop(int semid, struct sembuf *sops, size_t nsops);
11796
11797     Bu fonksyon da aslında tek hamlede kümedeki nsops tane semaphore için →
11798         işlem yapabilmektedir. Bu işlemler fonksiyonun ikinci parametresindeki →
11799         struct sembuf içerisinde kodlanır. Bu yapı <sys/sem.h> başlık dosyasında →
11800             aşağıdaki gibi bildirilmiştir:
11801
11802     struct sembuf {
11803         unsigned short sem_num;
11804         short        sem_op;
11805         short        sem_flg;
11806     }
11807
11808     Buradaki sem_num elemanı semahore kümesindeki semaphore numarasını →
11809         belirtir. sem_flag parametresi 0 geçilebilir. Asıl önemli parametre →
11810             sem_op parametresidir.
11811     semop pozitif negatif ya da sıfır değerini alabilir. Kritik kodun →
11812         başında programcı sem_op değerini negatif yaparak (tipik olarak -1) →
11813             kritik koda giriş yapmaya çalışır. Kritik kodun sonunda da sem_op değeri →
```

```
pozitif yapılır (tipik olarak 1). Eğer semaphore sayacı n iken
programcı
11796 semop fonksiyounu sem_op değeri -k (k'nın mutlak değeri n'den büyük
olsun) ile çağırırsa bloke oluşur. Başka bir deyişle semaphore
sayacını 0'ın altına indirme
11797 girişimi blokeye yol açmaktadır. Bu blokeden kurtulmanın yolu. Semaphore
sayacını en azından 0'a çıkartacak bir semop işlemi yapmaktadır. sem_op
değeri pozitif ise
11798 bu değer semaphore sayacına toplanır. Örneğin semaphore sayacının değeri
1 olsun. Şimdi biz bu semaphor'u semop fonksiyonuyla sem_op değeri -1
olacak biçimde
11799 beklemek istersek bloke olmayız. Ancak sem_op değeri -2 olacak biçimde
beklemeye çalışırsak bloke oluşur. Bu durumda bizim blokeden
kurtulmamız için semaphore sayacının
11800 2'ye yükseltilmesi gerekmektedir. Semahore sayacına semval diyelim. Eğer
semop fonksiyonunda sem_op değeri negatifse ve bu değerin mutlak
değeri semval'dan küçük ise
11801 sayaç eksiltilir ve bloke oluşmaz. (Örneğin semval 2 ikin sem_op -1 ise
bloke oluşmaz semval 1 olur.) Ancak sem_op değeri negatif ve mutlak
değeri semval değerinden
11802 büyükse budurumda semaphore sayacı eksiltilemez ancak bloke oluşur. Bloke
semval değeri pozitif sem_op değerine erişince ortadan kalkar.
(Örneğin semval değeri
11803 2 olsun. Biz sem_op -3 ile beklersek semval 2'de kalır ancak bloke
olusur. Bu blokenin çözülmesi için semval değerinin 3 haline gelmesi
gerekektedir.)
11804
11805 Nihayet sem_op değeri 0 ise ancak semahore sayacı 0 olduğunda bloke
olusur. Bu seçenek pek kullanılmamaktadır.
11806
11807 Mademki sistem 5 semaphore'larının arayüzleri biraz karmaşıktır. O halde
sistem 5 semaphore'larını POSIZ semaphore'ları gibi kullanabilmek
için
11808 aşağıdaki gibi sarma (wrapper) fonksiyonlar yazılabılır
11809
11810 -----*/
```

```
11811 #include <stdio.h>
11812 #include <stdlib.h>
11813 #include <unistd.h>
11814 #include <sys/ipc.h>
11815 #include <sys/sem.h>
11816 #include "general.h"
11817
11818 int sem_create(int key, int mode);
11819 int sem_open(int key);
11820 int sem_init(int semid, int val);
11821 int sem_post(int semid);
11822 int sem_wait(int semid);
11823 int sem_destroy(int semid);
11824
11825 int sem_create(int key, int mode)
```

```
11827 {
11828     return semget(key, 1, IPC_CREAT|mode);
11829 }
11830
11831 int sem_open(int key)
11832 {
11833     return semget(key, 1, 0);
11834 }
11835
11836 int sem_init(int semid, int val)
11837 {
11838     union semun {
11839         int val;
11840         struct semid_ds *buf;
11841         unsigned short *array;
11842         struct seminfo *__buf;
11843     } su;
11844
11845     su.val = val;
11846
11847     return semctl(semid, 0, SETVAL, su);
11848 }
11849
11850 int sem_post(int semid)
11851 {
11852     struct sembuf sb;
11853
11854     sb.sem_num = 0;
11855     sb.sem_op = 1;
11856     sb.sem_flg = 0;
11857
11858     return semop(semid, &sb, 1);
11859 }
11860
11861 int sem_wait(int semid)
11862 {
11863     struct sembuf sb;
11864
11865     sb.sem_num = 0;
11866     sb.sem_op = -1;
11867     sb.sem_flg = 0;
11868
11869     return semop(semid, &sb, 1);
11870 }
11871
11872 int sem_destroy(int semid)
11873 {
11874     return semctl(semid, 0, IPC_RMID);
11875 }
11876
11877 /
```



```
11878     Sistem 5 Semaphore'ları ve Sistem 5 Paylaşılan Bellek Alanları İle      ↵
     Üretici Tüketicisi Probleminin Gerçekleştirilmesi
11879  -----
11880
11881 /* producer.c */
11882
11883 #include <stdio.h>
11884 #include <stdlib.h>
11885 #include <string.h>
11886 #include <time.h>
11887 #include <unistd.h>
11888 #include <sys/ipc.h>
11889 #include <sys/shm.h>
11890 #include <sys/sem.h>
11891
11892 #define QSIZE      10
11893
11894 typedef struct {
11895     int queue[QSIZE];
11896     int head;
11897     int tail;
11898 } PROD_CONS;
11899
11900 void exit_sys(const char *msg);
11901 int sem_create(int key, int mode);
11902 int sem_init(int semid, int val);
11903 int sem_post(int semid);
11904 int sem_wait(int semid);
11905 int sem_destroy(int semid);
11906
11907 int main(void)
11908 {
11909     key_t shmkey;
11910     int shmid;
11911     key_t prod_semkey, cons_semkey;
11912     int prod_semid, cons_semid;
11913     void *addr;
11914     PROD_CONS *prod_cons;
11915     int i;
11916
11917     srand(time(NULL));
11918
11919     if ((shmkey = ftok(".", 123)) == -1)
11920         exit_sys("ftok");
11921
11922     if ((shmid = shmget(shmkey, 4096, IPC_CREAT|0644)) == -1)
11923         exit_sys("shmget");
11924
11925     if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
11926         exit_sys("shmat");
11927
11928     prod_cons = (PROD_CONS *)addr;
```

```
11929
11930     if ((prod_semkey = ftok("producer", 123)) == -1)
11931         exit_sys("ftok");
11932
11933     if ((cons_semkey = ftok("consumer", 123)) == -1)
11934         exit_sys("ftok");
11935
11936     if ((prod_semid = sem_create(prod_semkey, 0644)) == -1)
11937         exit_sys("sem_create");
11938
11939     if ((cons_semid = sem_create(cons_semkey, 0644)) == -1)
11940         exit_sys("sem_create");
11941
11942     if (sem_init(prod_semid, QSIZE) == -1)
11943         exit_sys("sem_init");
11944
11945     if (sem_init(cons_semid, 0) == -1)
11946         exit_sys("sem_init");
11947
11948     i = 0;
11949     for (;;) {
11950         usleep(rand() % 300000);
11951         if (sem_wait(prod_semid) == -1)
11952             exit_sys("sem_wait");
11953
11954         prod_cons->queue[prod_cons->tail++] = i;
11955         prod_cons->tail %= QSIZE;
11956
11957         if (sem_post(cons_semid) == -1)
11958             exit_sys("sem_post");
11959         if (i == 99)
11960             break;
11961         ++i;
11962     }
11963
11964     printf("press ENTER to exit\n");
11965     getchar();
11966
11967     shmdt(addr);
11968
11969     if (shmctl(shmid, IPC_RMID, NULL) == -1)
11970         exit_sys("shmctl");
11971
11972     if (sem_destroy(prod_semid) == -1)
11973         exit_sys("sem_destroy");
11974
11975     if (sem_destroy(cons_semid) == -1)
11976         exit_sys("sem_destroy");
11977
11978     return 0;
11979 }
11980
11981 void exit_sys(const char *msg)
```

```
11982 {
11983     perror(msg);
11984
11985     exit(EXIT_FAILURE);
11986 }
11987
11988 int sem_create(int key, int mode)
11989 {
11990     return semget(key, 1, IPC_CREAT|mode);
11991 }
11992
11993
11994 int sem_init(int semid, int val)
11995 {
11996     union semun {
11997         int val;
11998         struct semid_ds *buf;
11999         unsigned short *array;
12000         struct seminfo *_buf;
12001     } su;
12002
12003     su.val = val;
12004
12005     return semctl(semid, 0, SETVAL, su);
12006 }
12007
12008 int sem_post(int semid)
12009 {
12010     struct sembuf sb;
12011
12012     sb.sem_num = 0;
12013     sb.sem_op = 1;
12014     sb.sem_flg = 0;
12015
12016     return semop(semid, &sb, 1);
12017 }
12018
12019 int sem_wait(int semid)
12020 {
12021     struct sembuf sb;
12022
12023     sb.sem_num = 0;
12024     sb.sem_op = -1;
12025     sb.sem_flg = 0;
12026
12027     return semop(semid, &sb, 1);
12028 }
12029
12030 int sem_destroy(int semid)
12031 {
12032     return semctl(semid, 0, IPC_RMID);
12033 }
12034
```

```
12035 /* consumer.c */
12036
12037 #include <stdio.h>
12038 #include <stdlib.h>
12039 #include <string.h>
12040 #include <time.h>
12041 #include <unistd.h>
12042 #include <sys/ipc.h>
12043 #include <sys/shm.h>
12044 #include <sys/sem.h>
12045
12046 #define QSIZE      10
12047
12048 typedef struct {
12049     int queue[QSIZE];
12050     int head;
12051     int tail;
12052 } PROD_CONS;
12053
12054 void exit_sys(const char *msg);
12055 int sem_open(int key);
12056 int sem_init(int semid, int val);
12057 int sem_post(int semid);
12058 int sem_wait(int semid);
12059 int sem_destroy(int semid);
12060
12061 int main(void)
12062 {
12063     key_t shmkey;
12064     int shmid;
12065     key_t prod_semkey, cons_semkey;
12066     int prod_semid, cons_semid;
12067     void *addr;
12068     PROD_CONS *prod_cons;
12069     int val;
12070
12071     srand(time(NULL));
12072
12073     if ((shmkey = ftok(".", 123)) == -1)
12074         exit_sys("ftok");
12075
12076     if ((shmid = shmget(shmkey, 4096, 0)) == -1)
12077         exit_sys("shmget");
12078
12079     if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
12080         exit_sys("shmat");
12081
12082     prod_cons = (PROD_CONS *)addr;
12083
12084     if ((prod_semkey = ftok("producer", 123)) == -1)
12085         exit_sys("ftok");
12086
12087     if ((cons_semkey = ftok("consumer", 123)) == -1)
```

```
12088     exit_sys("ftok");
12089
12090     if ((prod_semid = sem_open(prod_semkey)) == -1)
12091         exit_sys("sem_create");
12092
12093     if ((cons_semid = sem_open(cons_semkey)) == -1)
12094         exit_sys("sem_create");
12095
12096     for (;;) {
12097
12098         if (sem_wait(cons_semid) == -1)
12099             exit_sys("sem_wait");
12100
12101         val = prod_cons->queue[prod_cons->head++];
12102         prod_cons->head %= QSIZE;
12103
12104         if (sem_post(prod_semid) == -1)
12105             exit_sys("sem_post");
12106
12107         printf("%d ", val);
12108         fflush(stdout);
12109
12110         if (val == 99)
12111             break;
12112
12113         usleep(rand() % 300000);
12114     }
12115     printf("\n");
12116
12117     shmdt(addr);
12118
12119     return 0;
12120 }
12121
12122 void exit_sys(const char *msg)
12123 {
12124     perror(msg);
12125
12126     exit(EXIT_FAILURE);
12127 }
12128
12129 int sem_open(int key)
12130 {
12131     return semget(key, 1, 0);
12132 }
12133
12134 int sem_init(int semid, int val)
12135 {
12136     union semun {
12137         int val;
12138         struct semid_ds *buf;
12139         unsigned short *array;
12140         struct seminfo *__buf;
```

```
12141     } su;
12142
12143     su.val = val;
12144
12145     return semctl(semid, 0, SETVAL, su);
12146 }
12147
12148 int sem_post(int semid)
12149 {
12150     struct sembuf sb;
12151
12152     sb.sem_num = 0;
12153     sb.sem_op = 1;
12154     sb.sem_flg = 0;
12155
12156     return semop(semid, &sb, 1);
12157 }
12158
12159 int sem_wait(int semid)
12160 {
12161     struct sembuf sb;
12162
12163     sb.sem_num = 0;
12164     sb.sem_op = -1;
12165     sb.sem_flg = 0;
12166
12167     return semop(semid, &sb, 1);
12168 }
12169
12170 int sem_destroy(int semid)
12171 {
12172     return semctl(semid, 0, IPC_RMID);
12173 }
12174
12175 /
*-----*
```

12176 Yukarıdaki programın sarma fonksiyonları kullanılmadan yazılmış hali
12177 aşağıda verilmiştir. Aşağıdaki kodda tek bir semaphore kümlesi
12178 iki semaphore'u içerecek biçimde yaratılmıştır. Bunların sayaçlarına tek
hamlede semctl fonksiyonu ile SETALL komut kodu ile ilkdeğerleri
verilmiştir.

```
12179 -----*/
```

12180
12181 /* producer.c */
12182
12183 #include <stdio.h>
12184 #include <stdlib.h>
12185 #include <string.h>
12186 #include <time.h>
12187 #include <unistd.h>
12188 #include <sys/ipc.h>

```
12189 #include <sys/shm.h>
12190 #include <sys/sem.h>
12191
12192 #define QSIZE      10
12193
12194 typedef struct {
12195     int queue[QSIZE];
12196     int head;
12197     int tail;
12198 } PROD_CONS;
12199
12200 union semun {
12201     int val;
12202     struct semid_ds *buf;
12203     unsigned short *array;
12204     struct seminfo *_buf;
12205 };
12206
12207 void exit_sys(const char *msg);
12208
12209 int main(void)
12210 {
12211     key_t shmkey;
12212     int shmid;
12213     key_t semkey;
12214     int semid;
12215     void *addr;
12216     PROD_CONS *prod_cons;
12217     int i;
12218     union semun su;
12219     unsigned short semvals[] = {QSIZE, 0};
12220     struct sembuf sb;
12221
12222     srand(time(NULL));
12223
12224     if ((shmkey = ftok(".", 123)) == -1)
12225         exit_sys("ftok");
12226
12227     if ((shmid = shmget(shmkey, 4096, IPC_CREAT|0644)) == -1)
12228         exit_sys("shmget");
12229
12230     if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
12231         exit_sys("shmat");
12232
12233     prod_cons = (PROD_CONS *)addr;
12234
12235     if ((semkey = ftok(".", 123)) == -1)
12236         exit_sys("ftok");
12237
12238     if ((semid = semget(semkey, 2, IPC_CREAT|0644)) == -1)
12239         exit_sys("semget");
12240
12241     su.array = semvals;
```

```
12242     if (semctl(semid, 0, SETALL, su) == -1)
12243         exit_sys("semctl");
12244
12245     i = 0;
12246     for (;;) {
12247         usleep(rand() % 300000);
12248
12249         sb.sem_num = 0;
12250         sb.sem_op = -1;
12251         sb.sem_flg = 0;
12252
12253         if (semop(semid, &sb, 1) == -1)
12254             exit_sys("semop");
12255
12256         prod_cons->queue[prod_cons->tail++] = i;
12257         prod_cons->tail %= QSIZE;
12258
12259         sb.sem_num = 1;
12260         sb.sem_op = 1;
12261         sb.sem_flg = 0;
12262
12263         if (semop(semid, &sb, 1) == -1)
12264             exit_sys("semop");
12265
12266         if (i == 99)
12267             break;
12268         ++i;
12269     }
12270
12271     printf("press ENTER to exit\n");
12272     getchar();
12273
12274     shmdt(addr);
12275
12276     if (shmctl(shmid, IPC_RMID, NULL) == -1)
12277         exit_sys("shmctl");
12278
12279     if (semctl(semid, 0, IPC_RMID) == -1)
12280         exit_sys("semctl");
12281
12282     return 0;
12283 }
12284
12285 void exit_sys(const char *msg)
12286 {
12287     perror(msg);
12288
12289     exit(EXIT_FAILURE);
12290 }
12291
12292 /* consumer.c */
12293
12294 #include <stdio.h>
```

```
12295 #include <stdlib.h>
12296 #include <string.h>
12297 #include <time.h>
12298 #include <unistd.h>
12299 #include <sys/ipc.h>
12300 #include <sys/shm.h>
12301 #include <sys/sem.h>
12302
12303 #define QSIZE      10
12304
12305 typedef struct {
12306     int queue[QSIZE];
12307     int head;
12308     int tail;
12309 } PROD_CONS;
12310
12311 union semun {
12312     int val;
12313     struct semid_ds *buf;
12314     unsigned short  *array;
12315     struct seminfo   *__buf;
12316 };
12317
12318 void exit_sys(const char *msg);
12319
12320 int main(void)
12321 {
12322     key_t shmkey;
12323     int shmid;
12324     key_t semkey;
12325     int semid;
12326     void *addr;
12327     PROD_CONS *prod_cons;
12328     int val;
12329     struct sembuf sb;
12330
12331     srand(time(NULL));
12332
12333     if ((shmkey = ftok(".", 123)) == -1)
12334         exit_sys("ftok");
12335
12336     if ((shmid = shmget(shmkey, 4096, 0)) == -1)
12337         exit_sys("shmget");
12338
12339     if ((addr = shmat(shmid, NULL, 0)) == (void *)-1)
12340         exit_sys("shmat");
12341
12342     prod_cons = (PROD_CONS *)addr;
12343
12344     if ((semkey = ftok(".", 123)) == -1)
12345         exit_sys("ftok");
12346
12347     if ((semid = semget(semkey, 2, 0)) == -1)
```


okuma amaçlı kilitlediyse diğer thread'ler o thread kiliği açmadan
12395 yazma amaçlı kilitleyememektedir. Benzer biçimde bir thread kiliği yazma →
amaçlı kilitlediyse diğer thread'ler o thread kiliği açmadan
12396 kiliği okuma amaçlı kilitleyememektedir. Bu işlemler →
pthread_rwlock_rdlock ve pthread_rwlock_wrlock fonksiyonlarıyla →
yapılmaktadır.

Kilidin açılması için pthread_rwlock_unlock fonksiyonu kullanılır. →
Blokesiz işlemler için pthread_rwlock_tryrdlock ve →
12398 pthread_rwlocktrywrlock fonksiyonları da bulunmuştur. Yine bu →
fonksiyonların zaan aşımı pthread_rwlock_timedrdlock ve →
12399 pthread_rwlock_timed_wrlock isimli biçimleri de vardır. En sonunda nesne →
pthread_rwlock_destroy fonksiyonuyla yok edilir.

Bu nesnenin de özellikleri vardır. Ancak mevcut durumda tek bir özellik →
tanımlanmıştır bu da nesnenin prosesler arasında paylaşımı ile →
12401 ilgiliidir. Bu özelliği set etmek için önce pthread_rwlockattr_t türünden →
nesne alınır. Bu nesne pthread_rwlockattr_init fonksiyonuyla →
ilkdeğerlenir. Sonra nesneye pthread_rwlockattr_setpshared fonksiyonuyla →
prosesler arası paylaşım özelliği eklenir. Bu özellik nesnesiyle →
12403 asıl nesne yaratılır. Sonra da bu özellik nesnesi →
pthread_rwlockattr_destroy fonksiyonu ile yok edilir.

12404
12405 Aşağıdaki örnekte 4 thread yaratılmıştır. Bu thread'lerin 2 tanesi okuma →
amaçlı 2 tanesi de yazma amaçlı kritik koda girmek →
12406 istemektedir. Çıkan sonuca bakarak mekanizmayı gözden geçirebilirisiniz →
12407 -----*/
12408
12409 #include <stdio.h>
12410 #include <stdlib.h>
12411 #include <string.h>
12412 #include <time.h>
12413 #include <unistd.h>
12414 #include <pthread.h>
12415
12416 void exit_sys_thread(const char *msg, int err);
12417 void *thread_proc1(void *param);
12418 void *thread_proc2(void *param);
12419 void *thread_proc3(void *param);
12420 void *thread_proc4(void *param);
12421
12422 pthread_rwlock_t g_rwlock = PTHREAD_RWLOCK_INITIALIZER;
12423
12424 int main(void)
12425 {
12426 int result;
12427 pthread_t tid1, tid2, tid3, tid4;
12428
12429 if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
12430 exit_sys_thread("pthread_create", result);
12431
12432 if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
12433 exit_sys_thread("pthread_create", result);
12434 }

```
12435     if ((result = pthread_create(&tid3, NULL, thread_proc3, NULL)) != 0)
12436         exit_sys_thread("pthread_create", result);
12437
12438     if ((result = pthread_create(&tid4, NULL, thread_proc4, NULL)) != 0)
12439         exit_sys_thread("pthread_create", result);
12440
12441     if ((result = pthread_join(tid1, NULL)) != 0)
12442         exit_sys_thread("pthread_join", result);
12443
12444     if ((result = pthread_join(tid2, NULL)) != 0)
12445         exit_sys_thread("pthread_join", result);
12446
12447     if ((result = pthread_join(tid3, NULL)) != 0)
12448         exit_sys_thread("pthread_join", result);
12449
12450     if ((result = pthread_join(tid4, NULL)) != 0)
12451         exit_sys_thread("pthread_join", result);
12452
12453     pthread_rwlock_destroy(&g_rwlock);
12454
12455     return 0;
12456 }
12457
12458 void exit_sys_thread(const char *msg, int err)
12459 {
12460     fprintf(stderr, "%s: %s\n", msg, strerror(err));
12461     exit(EXIT_FAILURE);
12462 }
12463
12464 void *thread_proc1(void *param)
12465 {
12466     int i;
12467     int result;
12468     int seedval;
12469
12470     seedval = (unsigned int)time(NULL) + 20000;
12471
12472     for (i = 0; i < 10; ++i) {
12473         usleep(rand_r(&seedval) % 300000);
12474
12475         if ((result = pthread_rwlock_rdlock(&g_rwlock)) != 0)
12476             exit_sys_thread("pthread_rwlock_rdlock", result);
12477
12478         printf("thread1 ENTERS to critical section for READING...\n");
12479
12480         usleep(rand_r(&seedval) % 300000);
12481
12482         printf("thread1 EXITS from critical section...\n");
12483
12484         if ((result = pthread_rwlock_unlock(&g_rwlock)) != 0)
12485             exit_sys_thread("pthread_rwlock_rdlock", result);
12486     }
12487 }
```

```
12488     return NULL;
12489 }
12490
12491 void *thread_proc2(void *param)
12492 {
12493     int i;
12494     int result;
12495     int seedval;
12496
12497     seedval = (unsigned int)time(NULL) + 20000;
12498
12499     for (i = 0; i < 10; ++i) {
12500         usleep(rand_r(&seedval) % 300000);
12501
12502         if ((result = pthread_rwlock_rdlock(&g_rwlock)) != 0)
12503             exit_sys_thread("pthread_rwlock_rdlock", result);
12504
12505         printf("thread2 ENTERS to critical section for READING...\n");
12506
12507         usleep(rand_r(&seedval) % 300000);
12508
12509         printf("thread2 EXITS from critical section...\n");
12510
12511         if ((result = pthread_rwlock_unlock(&g_rwlock)) != 0)
12512             exit_sys_thread("pthread_rwlock_rdlock", result);
12513     }
12514
12515     return NULL;
12516 }
12517
12518 void *thread_proc3(void *param)
12519 {
12520     int i;
12521     int result;
12522     int seedval;
12523
12524     seedval = (unsigned int)time(NULL) + 20000;
12525
12526     for (i = 0; i < 10; ++i) {
12527         usleep(rand_r(&seedval) % 300000);
12528
12529         if ((result = pthread_rwlock_wrlock(&g_rwlock)) == -1)
12530             exit_sys_thread("pthread_rwlock_rdlock", result);
12531
12532         printf("thread3 ENTERS to critical section for WRITING...\n");
12533
12534         usleep(rand_r(&seedval) % 300000);
12535
12536         printf("thread3 EXITS from critical section...\n");
12537
12538         if ((result = pthread_rwlock_unlock(&g_rwlock)) == -1)
12539             exit_sys_thread("pthread_rwlock_rdlock", result);
12540     }
```

```
12541     return NULL;
12542 }
12543 }
12544
12545 void *thread_proc4(void *param)
12546 {
12547     int i;
12548     int result;
12549     int seedval;
12550
12551     seedval = (unsigned int)time(NULL) + 20000;
12552
12553     for (i = 0; i < 10; ++i) {
12554         usleep(rand_r(&seedval) % 300000);
12555
12556         if ((result = pthread_rwlock_wrlock(&g_rwlock)) == -1)
12557             exit_sys_thread("pthread_rwlock_rdlock", result);
12558
12559         printf("thread4 ENTERS to critical section for WRITING...\n");
12560
12561         usleep(rand_r(&seedval) % 300000);
12562
12563         printf("thread4 EXITS from critical section...\n");
12564
12565         if ((result = pthread_rwlock_unlock(&g_rwlock)) == -1)
12566             exit_sys_thread("pthread_rwlock_rdlock", result);
12567     }
12568
12569     return NULL;
12570 }
12571
12572 /
*-----*
```

-----*

12573 Spinlock (spin dönme anlamına geliyor) mesgul bir döngüde bloke olmadan kildin açılması için beklemeye denilmektedir. ↗

12574 Bazı uygulamalarda çok işlemcili ya da çekirdekli sistemlerde kildi elde tutan thread eğer bunu makul bir zamanda bırakacaksa ↗

12575 spinlock kullanımı performansı iyileştirmektedir. POSIX sistemlerinde spinlockpthread_spinlock_t isimşi, nesneye temsil edilmektedir. ↗

12576 Bu nesne pthread_spin_init isimli fonksiyonla ilkdeğerlenir. Krtik kod pthread_spin_lock ile pthread_spin_unlock çağrıları ↗

12577 arasında yerleştirilir. İşlem bitince spinlock pthread_spin_destroy fonksiyonuya yok edilmektedir. Bir thread pthread_spin_lock ↗

12578 açısından kildi alamazsa bloke olmadan bir döngü içerisinde işemcinin set/compare işlemini atomik bir biçimde yapan makine ↗

12579 komutlarıyla kildi sürekli yoklar (polling). O sırada kildi elde etmiş olan thread CPU'yu bırakırsa spinlock'ta bekleyen thread'ler ↗

12580 önemli ölçüde CPU zamanı harcarlar. Bu nedenle spinlock'ların yüksek öncelikli, thread'ler tarafından uygulanması daha uygun olmaktadır. ↗

12581 Aslında GNU libc Kütüphanesinde olduğu gibi pek çok POSIX kütüphanesinde kildi almaya çalışam mutex, semaphore, read/write lock gibi ↗

12582 nesneler kilit kapaklısa zaten bir süre spin işlemi yapıp ondan sonra ↗

```
bloke olmaktadır. Yani başka bir deyişle zaten diğer
12583     senkronizasyon nesneleri küçük spinlock görevi de yapmaktadır.
12584     -----
12585     -----
12586 #include <stdio.h>
12587 #include <stdlib.h>
12588 #include <string.h>
12589 #include <pthread.h>
12590
12591 void exit_sys_thread(const char *msg, int err);
12592 void *thread_proc1(void *param);
12593 void *thread_proc2(void *param);
12594
12595 pthread_spinlock_t g_spinlock;
12596 int g_count;
12597
12598 int main(void)
12599 {
12600     int result;
12601     pthread_t tid1, tid2;
12602
12603     if ((result = pthread_spin_init(&g_spinlock, 0)) != 0)
12604         exit_sys_thread("pthread_spin_init", result);
12605
12606     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
12607         exit_sys_thread("pthread_create", result);
12608
12609     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
12610         exit_sys_thread("pthread_create", result);
12611
12612     if ((result = pthread_join(tid1, NULL)) != 0)
12613         exit_sys_thread("pthread_join", result);
12614
12615     if ((result = pthread_join(tid2, NULL)) != 0)
12616         exit_sys_thread("pthread_join", result);
12617
12618     printf("%d\n", g_count);
12619
12620     pthread_spin_destroy(&g_spinlock);
12621
12622     return 0;
12623 }
12624
12625 void exit_sys_thread(const char *msg, int err)
12626 {
12627     fprintf(stderr, "%s: %s\n", msg, strerror(err));
12628     exit(EXIT_FAILURE);
12629 }
12630
12631 void *thread_proc1(void *param)
12632 {
12633     int result;
```

```
12634     int i;
12635
12636     for (i = 0; i < 1000000000; ++i) {
12637         if ((result = pthread_spin_lock(&g_spinlock)) != 0)
12638             exit_sys_thread("pthread_spin_lock", result);
12639
12640         ++g_count;
12641
12642         if ((result = pthread_spin_unlock(&g_spinlock)) != 0)
12643             exit_sys_thread("pthread_spin_unlock", result);
12644     }
12645
12646     return NULL;
12647 }
12648 }
12649
12650 void *thread_proc2(void *param)
12651 {
12652     int result;
12653     int i;
12654
12655     for (i = 0; i < 1000000000; ++i) {
12656         if ((result = pthread_spin_lock(&g_spinlock)) != 0)
12657             exit_sys_thread("pthread_spin_lock", result);
12658
12659         ++g_count;
12660
12661         if ((result = pthread_spin_unlock(&g_spinlock)) != 0)
12662             exit_sys_thread("pthread_spin_unlock", result);
12663     }
12664
12665     return NULL;
12666 }
12667
12668 /
*-----*
-----  
12669     Bazı basit işlemler için mutex kullanmak programı yavaşlatabilmektedir. →
    Örneğin global bir değişkenin 1 artırılması işleminde →
    artırım işlemi birden fazla thread tarafından yapılıyorsa kritik kod →
    işlemi uygulanmalıdır. Öte yandna bu kritik kod işlemi →
    zaman kaybına yol açmaktadır. Bu tür tek makine komutuyla yapılabilecek →
    bazı işlemlerin hiç mutex kullanmadan gerçekleştirilmeleri →
    mümkündür. Intel işlemcilerinde makine komutları bellek operandı →
    alabilmektedir. Makine komutları da atomik olduğu için →
    (yani makine komutu çalışırken threadler arası geçiş olmayacağı için) bu →
    işlemcilerde bellek operandı alan komutlar yoluyla →
    hiç mutex koruması olmadan basit bazı işlemler yapılabilemektedir. →
    Örneğin Intel işlemcilerinde INC mem makine komutu atomik →
    bir biçimde bellekteki değeri 1 artırabilmektedir. Ancak maalesef çok →
    işlemcili ya da çok çekirdekli sistemlerde bu tür →
    makine komutları farklı işlemci ya da çekirdeklerde aynı bellek bölgesi →
    üzerinde işlem yapılrken geçersiz değerlerin oluşmasına →
```

12677 yol açabilemktedir. Intel işlemcileri özellikle bellekteki operand →
dördün ya da sekizin katlarına hizalanmamışsa bu işlemi
12678 tek hamlede yapmaktadır. Intel işlemcilerinde bu tür işlemlerin diğer →
işlemcilerin müdahalelesine kapatılarak yapılabilmemesi için
12679 komutun başın LOCK önekinin getirilmesi gerekmektedir. ARM işlemcileri →
Load/Store tarzı çalışmaya sahiptir. Dolayısıyla bellek
12680 üzerinde işlem yapan komutlar ARM mimarisinde yoktur. ARM mimarisinde →
mecburen bellekteki nesne önce CPU yazmacına çekiliplorada
12681 işleme sokulduktan sonra yeniden belleğe aktarılmaktadır. Fakat yine de →
ARM işlemcilerinde bu tür basit işlemlerin mutex kontrolü
12682 olmadan döngüsel bir yapı ile düşük maliyetli gerçekleştirilmesi de →
mümkündür.

12683
12684 İşte gcc derleyicisi artırımı, eksiltme ve karşılaştırma gibi basit →
işlemelerin atomik bir biçimde yapılabilmesi için özel
12685 built-in (intrinsic) fonksiyonlar bulunmaktadır. Build-in fonksiyonlar →
gcc tarafından doğrudan tanınıp bir makro gibi bunlar
12686 için kısa kodlar üretilebilmektedir. gcc'deki atomik builtin →
fonksiyonlar __sync_xxx biçiminde isimlendirilmiştir. Bunlar builtin
12687 olduğu için prototip gereklilikleri yoktur. Ancak daha sonra gcc C+ →
+11'deki atomik semantığını uygulayabilmek için bu builtin
12688 fonksiyonları __atomic_xxx ismiyle yenilemiştir. Bu yeni versiyonlar →
ekstra "memory order" parametresi almaktadır.

12689
12690 Aşağıdak örnekte iki thread aynı global değişkeni hiç mutex ile kritik →
kod oluşturmadan atomik bir biçimde artırmaktadır.
12691 Bu işlem 1000000000 döngü için 25.9 saniye sürmüştür. Aynı programın →
mutex versiyonu ise 4 dakika 11 saniye sürmüştür.
12692 -----*/

```
12693  
12694 #include <stdio.h>  
12695 #include <stdlib.h>  
12696 #include <string.h>  
12697 #include <pthread.h>  
12698  
12699 #define SIZE 1000000000  
12700  
12701 void exit_sys_thread(const char *msg, int err);  
12702 void *thread_proc1(void *param);  
12703 void *thread_proc2(void *param);  
12704  
12705 int g_count;  
12706  
12707 int main(void)  
12708 {  
12709     int result;  
12710     pthread_t tid1, tid2;  
12711  
12712     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)  
12713         exit_sys_thread("pthread_create", result);  
12714  
12715     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
```

```
12716         exit_sys_thread("pthread_create", result);
12717
12718     if ((result = pthread_join(tid1, NULL)) != 0)
12719         exit_sys_thread("pthread_join", result);
12720
12721     if ((result = pthread_join(tid2, NULL)) != 0)
12722         exit_sys_thread("pthread_join", result);
12723
12724     printf("%d\n", g_count);
12725
12726     return 0;
12727 }
12728
12729 void exit_sys_thread(const char *msg, int err)
12730 {
12731     fprintf(stderr, "%s: %s\n", msg, strerror(err));
12732     exit(EXIT_FAILURE);
12733 }
12734
12735 void *thread_proc1(void *param)
12736 {
12737     int i;
12738
12739     for (i = 0; i < SIZE; ++i) {
12740         __sync_fetch_and_add(&g_count, 1);
12741     }
12742
12743     return NULL;
12744 }
12745
12746 void *thread_proc2(void *param)
12747 {
12748     int i;
12749
12750     for (i = 0; i < SIZE; ++i) {
12751         __sync_fetch_and_add(&g_count, 1);
12752     }
12753
12754     return NULL;
12755 }
12756
12757 /
*-----*
-----*
12758 Yukarıdaki programın __atomic_xxx builtin fonksiyonları ile eşdegeri
12759     aşağıdadır. __atomic_xxx fonksiyonlarının son
12760 parametresi olan "memory order" için __ATOMIC_SEQ_CST kullanılmıştır. Bu
12761     parametre __sync_xxx ile aynı semantiği sağlamaktadır.
12762 Buradaki memory order parametresi için ilgi dokümanlara
12763     başvurabilirsiniz.
12764 -----
-----*/
12765
```

```
12763 #include <stdio.h>
12764 #include <stdlib.h>
12765 #include <string.h>
12766 #include <pthread.h>
12767
12768 #define SIZE      1000000000
12769
12770 void exit_sys_thread(const char *msg, int err);
12771 void *thread_proc1(void *param);
12772 void *thread_proc2(void *param);
12773
12774 int g_count;
12775
12776 int main(void)
12777 {
12778     int result;
12779     pthread_t tid1, tid2;
12780
12781     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
12782         exit_sys_thread("pthread_create", result);
12783
12784     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
12785         exit_sys_thread("pthread_create", result);
12786
12787     if ((result = pthread_join(tid1, NULL)) != 0)
12788         exit_sys_thread("pthread_join", result);
12789
12790     if ((result = pthread_join(tid2, NULL)) != 0)
12791         exit_sys_thread("pthread_join", result);
12792
12793     printf("%d\n", g_count);
12794
12795     return 0;
12796 }
12797
12798 void exit_sys_thread(const char *msg, int err)
12799 {
12800     fprintf(stderr, "%s: %s\n", msg, strerror(err));
12801     exit(EXIT_FAILURE);
12802 }
12803
12804 void *thread_proc1(void *param)
12805 {
12806     int i;
12807
12808     for (i = 0; i < SIZE; ++i) {
12809         __atomic_add_fetch(&g_count, 1, __ATOMIC_SEQ_CST);
12810     }
12811
12812     return NULL;
12813 }
12814
12815 void *thread_proc2(void *param)
```

```
12816  {
12817      int i;
12818
12819      for (i = 0; i < SIZE; ++i) {
12820          __atomic_add_fetch(&g_count, 1, __ATOMIC_SEQ_CST);
12821     }
12822
12823     return NULL;
12824 }
12825
12826 /
*-----*
-----*
12827 C'nin 2011 revizyonunda (C11) dile _Atomic isminde bir tür niteleyicisi  -->
12828     (type qualifier) eklenmiştir. Bir nesne bu niteleyici ile
12829 bildirilirse +=, -=, |= gibi işlemler thread güvenli bir biçimde atomik  -->
12830     olarak gerçekleştirilmektedir. Aşağıdaki programda
12831 bu özellik kullanılmıştır. _Atomic anahtar sözcüğü C11 ile geldiği için  -->
12832     gcc derlemesinde -std=c11 ya da -std=c17 seçeneğini
12833 bulundurunuz.
12834 -----*/
12835
12836 #include <stdio.h>
12837 #include <stdlib.h>
12838 #include <string.h>
12839 #include <pthread.h>
12840
12841 #define SIZE      10000000
12842
12843 void exit_sys_thread(const char *msg, int err);
12844 void *thread_proc1(void *param);
12845 void *thread_proc2(void *param);
12846
12847 _Atomic int g_count;
12848
12849 int main(void)
12850 {
12851     int result;
12852     pthread_t tid1, tid2;
12853
12854     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
12855         exit_sys_thread("pthread_create", result);
12856
12857     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
12858         exit_sys_thread("pthread_create", result);
12859
12860     if ((result = pthread_join(tid1, NULL)) != 0)
12861         exit_sys_thread("pthread_join", result);
12862
12863     if ((result = pthread_join(tid2, NULL)) != 0)
12864         exit_sys_thread("pthread_join", result);
```

```
12863     printf("%d\n", g_count);
12864
12865     return 0;
12866 }
12867
12868 void exit_sys_thread(const char *msg, int err)
12869 {
12870     fprintf(stderr, "%s: %s\n", msg, strerror(err));
12871     exit(EXIT_FAILURE);
12872 }
12873
12874 void *thread_proc1(void *param)
12875 {
12876     int i;
12877
12878     for (i = 0; i < SIZE; ++i) {
12879         ++g_count;
12880     }
12881
12882     return NULL;
12883 }
12884
12885 void *thread_proc2(void *param)
12886 {
12887     int i;
12888
12889     for (i = 0; i < SIZE; ++i) {
12890         ++g_count;
12891     }
12892
12893     return NULL;
12894 }
12895
12896 /
*-----*
-----*
-----*
```

12897 C++'ta `<atomic>` başlık dosyası içerisindeki template atomic sınıfı
atomik işlemler yapmaktadır. Aşağıda bu sınıf kullanılarak
C++ örneği verilmiştir.

```
12900 -----*/
12901 #include <stdio.h>
12902 #include <stdlib.h>
12903 #include <string.h>
12904 #include <pthread.h>
12905 #include <atomic>
12906
12907 #define SIZE      10000000
12908
12909 void exit_sys_thread(const char *msg, int err);
12910 void *thread_proc1(void *param);
12911 void *thread_proc2(void *param);
```

```
12912
12913 using namespace std;
12914
12915 atomic<int> g_count;
12916
12917 int main(void)
12918 {
12919     int result;
12920     pthread_t tid1, tid2;
12921
12922     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
12923         exit_sys_thread("pthread_create", result);
12924
12925     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
12926         exit_sys_thread("pthread_create", result);
12927
12928     if ((result = pthread_join(tid1, NULL)) != 0)
12929         exit_sys_thread("pthread_join", result);
12930
12931     if ((result = pthread_join(tid2, NULL)) != 0)
12932         exit_sys_thread("pthread_join", result);
12933
12934     printf("%d\n", (int)g_count);
12935
12936     return 0;
12937 }
12938
12939 void exit_sys_thread(const char *msg, int err)
12940 {
12941     fprintf(stderr, "%s: %s\n", msg, strerror(err));
12942     exit(EXIT_FAILURE);
12943 }
12944
12945 void *thread_proc1(void *param)
12946 {
12947     int i;
12948
12949     for (i = 0; i < SIZE; ++i) {
12950         ++g_count;
12951     }
12952
12953     return NULL;
12954 }
12955
12956 void *thread_proc2(void *param)
12957 {
12958     int i;
12959
12960     for (i = 0; i < SIZE; ++i) {
12961         ++g_count;
12962     }
12963
12964     return NULL;
```

```
12965 }
12966
12967 /
*-----*
-----*
12968 C'nin stdio dosya fonksiyonları POSIZ sistemlerinde (fakat standart C'de ↵
     değil) zaten thread güvenlidir. Dolayısıyla iki ↵
12969 thread aynı dosyaya yazma ya da okuma yaparken programcının fwrite, ↵
     fread gibi fonksiyonları senkronize etmesine gerek yoktur. ↵
12970 Aşağıdaki programda iki thread aynı dosyaya fwrite fonksiyonlarıyla ↵
     0'dan SIZE'a kadar int değerleri yazmaktadır. Thread'lerde biri ↵
12971 tek sayıları yazarken diğerinin çift sayıları yazmaktadır. İşlemin sonunda ↵
     yazma işleminde bir sorun olup olmadığı test edilmiştir. ↵
12972
12973 Her ne kadar standart C'nin stdio fonksiyonları POSIX sistemlerinde ↵
     thread güvenli olsa da yine de birden fazla art arda çağrılarında ↵
12974 senkronizasyon sorunları ortaya çıkabilir. Örneğin iki thread fseek ↵
     uyguladıktan sonra fwrite işlemi yapsa thread'lerde biri ↵
12975 istemediği bir bölgeye yazma yapabilir. Bu durumda bit mutex koruması ↵
     düşünülebilir. Ancak POSIX sistemlerinde içsel olarak böyle ↵
12976 bir mutex kontrolü düşünülmüştür. flockfile ile bir stdio dosyasını ↵
     kilitlersek funlockfile uygulayana kadar başka bir thread ↵
12977 bu dosya üzerinde işlem yapmak istediği blokede beklemektedir. ↵
12978 -----*/
```

```
12979
12980 #include <stdio.h>
12981 #include <stdlib.h>
12982 #include <string.h>
12983 #include <pthread.h>
12984
12985 #define SIZE      1000000
12986
12987 FILE *g_f;
12988
12989 void exit_sys_thread(const char *msg, int err);
12990 void *thread_proc1(void *param);
12991 void *thread_proc2(void *param);
12992 int check(void);
12993
12994 int main(void)
12995 {
12996     int result;
12997     pthread_t tid1, tid2;
12998     int i;
12999
13000     if ((g_f = fopen("test.dat", "w+")) == NULL) {
13001         perror("fopen");
13002         exit(EXIT_FAILURE);
13003     }
13004
13005     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
13006         exit_sys_thread("pthread_create", result);
```

```
13007
13008     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
13009         exit_sys_thread("pthread_create", result);
13010
13011     if ((result = pthread_join(tid1, NULL)) != 0)
13012         exit_sys_thread("pthread_join", result);
13013
13014     if ((result = pthread_join(tid2, NULL)) != 0)
13015         exit_sys_thread("pthread_join", result);
13016
13017     printf(check() ? "success...\n" : "failed...\n");
13018
13019
13020     return 0;
13021 }
13022
13023 void exit_sys_thread(const char *msg, int err)
13024 {
13025     fprintf(stderr, "%s: %s\n", msg, strerror(err));
13026     exit(EXIT_FAILURE);
13027 }
13028
13029 void *thread_proc1(void *param)
13030 {
13031     int i;
13032
13033     for (i = 0; i < SIZE; i += 2)
13034         if (fwrite(&i, sizeof(int), 1, g_f) != 1) {
13035             perror("fwrite");
13036             exit(EXIT_FAILURE);
13037         }
13038
13039     return NULL;
13040 }
13041
13042 void *thread_proc2(void *param)
13043 {
13044     int i;
13045
13046     for (i = 1; i < SIZE; i += 2)
13047         if (fwrite(&i, sizeof(int), 1, g_f) != 1) {
13048             perror("fwrite");
13049             exit(EXIT_FAILURE);
13050         }
13051     return NULL;
13052 }
13053
13054
13055 int check(void)
13056 {
13057     char flags[SIZE] = {0};
13058     int val;
13059     int i;
```

```
13060
13061     rewind(g_f);
13062     while (fread(&val, sizeof(int), 1, g_f) == 1) {
13063         if (flags[val])
13064             return 0;
13065         flags[val] = 1;
13066     }
13067
13068     if (ferror(g_f)) {
13069         perror("fread");
13070         exit(EXIT_FAILURE);
13071     }
13072
13073     for (i = 0; i < SIZE; ++i)
13074         if (!flags[i])
13075             return 0;
13076
13077     return 1;
13078 }
13079
13080 /
*-----*
-----*
13081     Bir proses birtakım thread'ler yarattıktan sonra fork işlemi yaparsa alt proses her zaman tek bir thread ile çalışmaya devam
13082     eder. O da üst prosesin fork işleminin yapıldığı thread'tır.
13083 -----*/
13084
13085 #include <stdio.h>
13086 #include <stdlib.h>
13087 #include <string.h>
13088 #include <unistd.h>
13089 #include <sys/wait.h>
13090 #include <pthread.h>
13091
13092 void exit_sys(const char *msg);
13093 void exit_sys_thread(const char *msg, int err);
13094 void *thread_proc1(void *param);
13095 void *thread_proc2(void *param);
13096
13097 int main(void)
13098 {
13099     int result;
13100     pthread_t tid1, tid2;
13101     pid_t pid;
13102
13103     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
13104         exit_sys_thread("pthread_create", result);
13105
13106     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
13107         exit_sys_thread("pthread_create", result);
13108
```

```
13109     if ((pid = fork()) == -1)
13110         exit_sys("fork");
13111
13112     if (pid != 0)
13113         printf("parent process\n");
13114     else
13115     {
13116         printf("child process\n");
13117         pthread_exit(NULL);
13118     }
13119
13120     if ((result = pthread_join(tid1, NULL)) != 0)
13121         exit_sys_thread("pthread_join", result);
13122
13123     if ((result = pthread_join(tid2, NULL)) != 0)
13124         exit_sys_thread("pthread_join", result);
13125
13126     if (waitpid(pid, NULL, 0) == -1)
13127         exit_sys("waitpid");
13128
13129     return 0;
13130 }
13131
13132 void exit_sys(const char *msg)
13133 {
13134     perror(msg);
13135
13136     exit(EXIT_FAILURE);
13137 }
13138
13139 void exit_sys_thread(const char *msg, int err)
13140 {
13141     fprintf(stderr, "%s: %s\n", msg, strerror(err));
13142     exit(EXIT_FAILURE);
13143 }
13144
13145 void *thread_proc1(void *param)
13146 {
13147     int i;
13148
13149     for (i = 0; i < 20; ++i) {
13150         printf("thread-1: %d\n", i);
13151         sleep(1);
13152     }
13153
13154     return NULL;
13155 }
13156
13157 void *thread_proc2(void *param)
13158 {
13159     int i;
13160
13161     for (i = 0; i < 20; ++i) {
```

```
13162     printf("thread-2: %d\n", i);
13163     sleep(1);
13164 }
13165
13166     return NULL;
13167 }
13168
13169 /
*-----*
-----*
13170 Bir C programının exit fonksiyonuyla ya da main fonksiyonun bitmesiyle
sonlandığını biliyorsunuz. Ancak programın ana thread'i
pthread_exit fonksiyonuyla sonlandırılabilir. Bu durumda proses onun son
thread'i sonlandığında sonlandırılır.
13171
13172 Aşağıdaki programda fork işlemi ana thread'te değil başka bir thread'te
uygulanmıştır. Alt proses tek bir thread'le çalışmaya
13174 başlayacaktır. O da fork fonksiyonunu uygulayan thread'tir. Programın
çıktısını inceleyiniz.
13175 -----
-----*/
13176
13177 #include <stdio.h>
13178 #include <stdlib.h>
13179 #include <string.h>
13180 #include <unistd.h>
13181 #include <sys/wait.h>
13182 #include <pthread.h>
13183
13184 void exit_sys(const char *msg);
13185 void exit_sys_thread(const char *msg, int err);
13186 void *thread_proc1(void *param);
13187 void *thread_proc2(void *param);
13188
13189 int main(void)
13190 {
13191     int result;
13192     pthread_t tid1, tid2;
13193
13194     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
13195         exit_sys_thread("pthread_create", result);
13196
13197     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
13198         exit_sys_thread("pthread_create", result);
13199
13200     if ((result = pthread_join(tid1, NULL)) != 0)
13201         exit_sys_thread("pthread_join", result);
13202
13203     if ((result = pthread_join(tid2, NULL)) != 0)
13204         exit_sys_thread("pthread_join", result);
13205
13206     return 0;
13207 }
```

```
13208
13209 void exit_sys(const char *msg)
13210 {
13211     perror(msg);
13212
13213     exit(EXIT_FAILURE);
13214 }
13215
13216 void exit_sys_thread(const char *msg, int err)
13217 {
13218     fprintf(stderr, "%s: %s\n", msg, strerror(err));
13219     exit(EXIT_FAILURE);
13220 }
13221
13222 void *thread_proc1(void *param)
13223 {
13224     int i;
13225     pid_t pid;
13226
13227     for (i = 0; i < 20; ++i) {
13228         printf("thread-1: %d\n", i);
13229         if (i == 10) {
13230             if ((pid = fork()) == -1)
13231                 exit_sys("fork");
13232         }
13233         sleep(1);
13234     }
13235
13236     if (pid != 0 && waitpid(pid, NULL, 0) == -1)
13237         exit_sys("waitpid");
13238
13239     return NULL;
13240 }
13241
13242 void *thread_proc2(void *param)
13243 {
13244     int i;
13245
13246     for (i = 0; i < 20; ++i) {
13247         printf("thread-2: %d\n", i);
13248         sleep(1);
13249     }
13250
13251     return NULL;
13252 }
13253
13254 /
-----  

-----  

13255 Aslında çok thread'lu proseslerde fork işlemi organizasyonel bakımdan      ↗
      karmaşık birtakım sonuçlar doğurmaktadır.      ↗
13256 Bu nedenle çok thread'lu uygulamalarda ya fork yapılmamalı ya da fork      ↗
      yapılacaksa alt proste hemen exec uygulanmalıdır.
```

```
13257
13258      Bir proses birtakım thread'ler yarattıktan sonra herhangi bir thread'te ↵
13259          exec işlemi uygularsa programın bütün bellek alanı ↵
13260          boşaltılacağına göre exec yapılan prog tek bir thread'le çalışmasına ↵
13261          devam edecekt.r exec işlemi başarı durumunda exec işlemini uygulayan ↵
13262          thread dışında prosesin bütün thread'lerini otomatik olarak yok ↵
13263          etmektedir. Yani exec işlemi sonucunda exec edilen kod her zaman ↵
13264          tek bir thread'le çalışmaya başlar.
13265  -----
13266  -----
13267  -----
13268  -----
13269  -----
13270  -----
13271 void exit_sys(const char *msg);
13272 void exit_sys_thread(const char *msg, int err);
13273 void *thread_proc1(void *param);
13274 void *thread_proc2(void *param);
13275
13276 int main(void)
13277 {
13278     int result;
13279     pthread_t tid1, tid2;
13280
13281     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
13282         exit_sys_thread("pthread_create", result);
13283
13284     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
13285         exit_sys_thread("pthread_create", result);
13286
13287     if ((result = pthread_join(tid1, NULL)) != 0)
13288         exit_sys_thread("pthread_join", result);
13289
13290     if ((result = pthread_join(tid2, NULL)) != 0)
13291         exit_sys_thread("pthread_join", result);
13292
13293     return 0;
13294 }
13295
13296 void exit_sys(const char *msg)
13297 {
13298     perror(msg);
13299
13300     exit(EXIT_FAILURE);
13301 }
13302
13303 void exit_sys_thread(const char *msg, int err)
13304 {
13305     fprintf(stderr, "%s: %s\n", msg, strerror(err));
```

```
13306     exit(EXIT_FAILURE);
13307 }
13308
13309 void *thread_proc1(void *param)
13310 {
13311     int i;
13312     pid_t pid;
13313
13314     for (i = 0; i < 20; ++i) {
13315         printf("thread-1: %d\n", i);
13316         if (i == 10) {
13317             if ((pid = fork()) == -1)
13318                 exit_sys("fork");
13319             if (pid == 0)
13320                 if (execlp("ls", "ls", "-l", (char *)NULL) == -1)
13321                     exit_sys("execlp");
13322             /* unreachable code */
13323         }
13324         sleep(1);
13325     }
13326
13327     if (pid != 0 && waitpid(pid, NULL, 0) == -1)
13328         exit_sys("waitpid");
13329
13330     return NULL;
13331 }
13332
13333 void *thread_proc2(void *param)
13334 {
13335     int i;
13336
13337     for (i = 0; i < 20; ++i) {
13338         printf("thread-2: %d\n", i);
13339         sleep(1);
13340     }
13341
13342     return NULL;
13343 }
13344
13345 /
*-----*-----*-----*
```

13346 Thread'lı programların exec işlemi dışında fork yapması organizasyonel birtakım problemler doğurabilemektedir. Üst prosese birtakım senkronizasyon nesneleri varsa alt prosese bu senkronizasyon nesneleri akratıldığı için organizasyonel sorunlar oluşabilmektedir.

13347 Çünkü üst proses fork işlemi sırasına bazı senkronizasyon nesnelerini kilitlemiş olabilir.

13348 Bu senkronizasyon nesneleri kilitli bir biçimde alt prosese aktarıldığında alt prosese kilitlenmelere (deadlocks) yol açabilemektedir.

13349 Bu nedenle üst prosesteki bu senkronizasyon nesnelerinin kararlı bir durumda alt prosese aktarılabilmesi için pthread_atfork isimli bir

```
POSIX
13351 fonksiyonu düşünülmüştür. Bu fonksiyonun "prepare", "parent" ve "child" ↵
    biçiminde üç fonksiyon göstericisi parametresi vardır. prepare ile ↵
    belirtilen
13352 fonksiyon fork işleminin hemen öncesinde öncesinde üst proses tarafından ↵
    otomatik çalıştırılır. Programcının bu fonksiyonda tüm senkronizasyon ↵
    nesnelerini
13353 kilitlemesi uygun olur. parent isimli fonksiyon fork sonrasında üset ↵
    proses tarafından child isimli fonksiyon ise fork işlemi sonrasında ↵
    alt proses
13354 tarafından çalıştırılır. Her iki fonksiyonda da programcının bu ↵
    kilitlediği senkronizasyon nesnelerini açması beklenmektedir. Böylece ↵
    hiç olmazsa
13355 alt prosesde fork sonrasında bütün senkronizasyon nesnelerinin açık bir ↵
    biçimde olacağı garanti edilmiş olur. Aşağıdaki örnekte bu semantik ↵
    uygulanmıştır.
13356 -----
-----*/
13357
13358 #include <stdio.h>
13359 #include <stdlib.h>
13360 #include <string.h>
13361 #include <unistd.h>
13362 #include <sys/wait.h>
13363 #include <pthread.h>
13364
13365 void exit_sys(const char *msg);
13366 void exit_sys_thread(const char *msg, int err);
13367 void *thread_proc1(void *param);
13368 void *thread_proc2(void *param);
13369
13370 void prepare(void);
13371 void parent(void);
13372 void child(void);
13373
13374 pthread_mutex_t g_mutex1 = PTHREAD_MUTEX_INITIALIZER;
13375 pthread_mutex_t g_mutex2 = PTHREAD_MUTEX_INITIALIZER;
13376
13377 int main(void)
13378 {
13379     int result;
13380     pthread_t tid1, tid2;
13381
13382     if ((result = pthread_atfork(prepare, parent, child)) != 0)
13383         exit_sys_thread("pthread_atfork", result);
13384
13385     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
13386         exit_sys_thread("pthread_create", result);
13387
13388     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
13389         exit_sys_thread("pthread_create", result);
13390
13391     if ((result = pthread_join(tid1, NULL)) != 0)
```

```
13392         exit_sys_thread("pthread_join", result);
13393
13394     if ((result = pthread_join(tid2, NULL)) != 0)
13395         exit_sys_thread("pthread_join", result);
13396
13397     return 0;
13398 }
13399
13400 void exit_sys(const char *msg)
13401 {
13402     perror(msg);
13403
13404     exit(EXIT_FAILURE);
13405 }
13406
13407 void exit_sys_thread(const char *msg, int err)
13408 {
13409     fprintf(stderr, "%s: %s\n", msg, strerror(err));
13410     exit(EXIT_FAILURE);
13411 }
13412
13413 void *thread_proc1(void *param)
13414 {
13415     int i;
13416     pid_t pid;
13417
13418     for (i = 0; i < 20; ++i) {
13419         printf("thread-1: %d\n", i);
13420         if (i == 10) {
13421             if ((pid = fork()) == -1)
13422                 exit_sys("fork");
13423             /* unreachable code */
13424         }
13425         sleep(1);
13426     }
13427
13428     if (pid != 0 && waitpid(pid, NULL, 0) == -1)
13429         exit_sys("waitpid");
13430
13431     return NULL;
13432 }
13433
13434 void *thread_proc2(void *param)
13435 {
13436     int i;
13437
13438     for (i = 0; i < 20; ++i) {
13439         printf("thread-2: %d\n", i);
13440         sleep(1);
13441     }
13442
13443     return NULL;
13444 }
```

```
13445
13446 void prepare(void)
13447 {
13448     int result;
13449
13450     printf("prepared...\n");
13451
13452     if ((result = pthread_mutex_lock(&g_mutex1)) != 0)
13453         exit_sys_thread("pthread_mutex_lock", result);
13454
13455     if ((result = pthread_mutex_lock(&g_mutex2)) != 0)
13456         exit_sys_thread("pthread_mutex_lock", result);
13457 }
13458
13459 void parent(void)
13460 {
13461     int result;
13462
13463     printf("parent\n");
13464
13465     if ((result = pthread_mutex_unlock(&g_mutex1)) != 0)
13466         exit_sys_thread("pthread_mutex_unlock", result);
13467
13468     if ((result = pthread_mutex_unlock(&g_mutex2)) != 0)
13469         exit_sys_thread("pthread_mutex_lock", result);
13470 }
13471
13472 void child(void)
13473 {
13474     int result;
13475
13476     printf("child\n");
13477
13478     if ((result = pthread_mutex_unlock(&g_mutex1)) != 0)
13479         exit_sys_thread("pthread_mutex_unlock", result);
13480
13481     if ((result = pthread_mutex_unlock(&g_mutex2)) != 0)
13482         exit_sys_thread("pthread_mutex_lock", result);
13483 }
13484
13485 /
*-----*-----*-----*
```

13486 Global ya da static yerel nesne kullanan fonksiyonlar thread güvenli
değildir. Bunları prototiplerini değiştirmeden
13487 thread güvenli yapabilmek için thread'e özgü global değişken etkisinin
yaratılması gereklidir. İşte işletim sistemleri bu tür
13488 thread güvenli kodların yazılabilmesi için thread'e özgü global alanlar
oluşturmaktadır. Bu alanlara Windows sistemlerinde
13489 "Thread Local Storage", UNIX/Linux sistemlerinde "Thread Specific Data"
denilmektedir. Naîsl her thread'in bir stack'i varsa bir de
13490 TSD alanı vardır. Bu alan slotlardan oluşmaktadır. Slotların numaraları
vardır. Bu numaralar pthread_key_t türüyle temsil

13491 edilmiştir. Programcı önce `pthread_key_create` fonksiyonu ile bir slot ↵
 (ya da anahtar) yaratır. Slotlar `thread`'ler için ayrı ayrı ↵
13492 yaratılmamaktadır. Bir slot yaratıldığında tüm `thread`'ler için (daha ↵
 önce yaratılmış ve daha sonra yaratılacak olanlar da dahil olmak ↵
 üzere)
13493 yaratılmaktadır. Slotlara `void` bir adres yerleştirilip geri ↵
 alınabilmektedir. Bunun için `pthread_setspecific` ve ↵
 `pthread_getspecific`
13494 fonksiyonları kullanılmaktadır. Slot yaratıldığında içerisinde `NULL` ↵
 adres olduğu garanti edilmiştir. Programcı tipik olarak ↵
13495 `malloc` ile `heap`'te bir alan tahsis edip onun adresini slot'a yerleştirir. ↵
 Tabii tek bilgi aşağıdaki örnekte olduğu gibi
13496 bir adres gibi de doğrudan slotlara yerleştirilebilmektedir. Böylece her ↵
 `thread` aynı anahtarı kullanısa da aslında kendi `thread`'inin ↵
13497 slotuna erişir. En sonunda yaratılmış olan slot (anahtar) ↵
 `pthread_key_delete` fonksiyonu ile yok edilebilmektedir.
13498
13499 Aşağıdaki örnekte `rand` ve `srand` fonksiyonları TSD kullanılarak `thread` ↵
 güvenli hale getirilmiştir
13500 -----*/
13501
13502 #include <stdio.h>
13503 #include <stdlib.h>
13504 #include <string.h>
13505 #include <unistd.h>
13506 #include <pthread.h>
13507
13508 void exit_sys_thread(const char *msg, int err);
13509 void *thread_proc1(void *param);
13510 void *thread_proc2(void *param);
13511
13512 pthread_key_t g_key;
13513
13514 int myrand(void)
13515 {
13516 void *val;
13517 unsigned seed;
13518 int result;
13519
13520 if ((val = pthread_getspecific(g_key)) == NULL) {
13521 if ((result = pthread_setspecific(g_key, (void *)1)) != 0)
13522 exit_sys_thread("pthread_setspecific", result);
13523 seed = 1;
13524 }
13525 else
13526 seed = (unsigned) val;
13527
13528 seed = seed * 1103515245 + 12345;
13529 if ((result = pthread_setspecific(g_key, (void *)seed)) != 0)
13530 exit_sys_thread("pthread_setspecific", result);
13531
13532 return seed / 65536 % 32768;

```
13533 }
13534
13535 void myrand(unsigned seed)
13536 {
13537     int result;
13538
13539     if ((result = pthread_setspecific(g_key, (void *)seed)) != 0)
13540         exit_sys_thread("pthread_setspecific", result);
13541 }
13542
13543 int main(void)
13544 {
13545     int result;
13546     pthread_t tid1, tid2;
13547
13548     if ((result = pthread_key_create(&g_key, NULL)) != 0)
13549         exit_sys_thread("pthread_key_create", result);
13550
13551     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
13552         exit_sys_thread("pthread_create", result);
13553
13554     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
13555         exit_sys_thread("pthread_create", result);
13556
13557     if ((result = pthread_join(tid1, NULL)) != 0)
13558         exit_sys_thread("pthread_join", result);
13559
13560     if ((result = pthread_join(tid2, NULL)) != 0)
13561         exit_sys_thread("pthread_join", result);
13562
13563     if ((result = pthread_key_delete(g_key)) != 0)
13564         exit_sys_thread("pthread_key_delete", result);
13565
13566     return 0;
13567 }
13568
13569 void exit_sys_thread(const char *msg, int err)
13570 {
13571     fprintf(stderr, "%s: %s\n", msg, strerror(err));
13572     exit(EXIT_FAILURE);
13573 }
13574
13575 void *thread_proc1(void *param)
13576 {
13577     int i;
13578     int val;
13579
13580     for (i = 0; i < 10; ++i) {
13581         val = myrand();
13582         printf("Thread-1: %d\n", val % 100);
13583     }
13584     printf("\n");
13585 }
```

```
13586     return NULL;
13587 }
13588
13589 void *thread_proc2(void *param)
13590 {
13591     int i;
13592     int val;
13593
13594     for (i = 0; i < 10; ++i) {
13595         val = myrand();
13596         printf("Thread-2: %d\n", val % 100);
13597     }
13598     printf("\n");
13599
13600     return NULL;
13601 }
13602 /
*-----*
-----*
13604     Aşağıda getenv fonksiyonu TSD kullanılarak thread güvenli hale      ↵
      getirilmiştir. Read/Write lock nesnesi başka bir thread çevre      ↵
      değişkenlerini
13605     değiştirirken ya da çevre değişkenlerine ekleme yaparken bozulmayı      ↵
      englemek için kullanılmıştır. Tabii bunu değiştirecek thread de aynı      ↵
      nesneyle
13606     write amaçlı kiliti elde etmeye çalışmalıdır. Thread ilk kez slota      ↵
      yerleştirme yapacağı zaman bellek tahsis edip onun adresini slota      ↵
      yerleştirmiştir.
13607     Diğer çağrılarda artık bu tahsis ettiği alanı kullanmaktadır. Bu alanın      ↵
      otomatik free getirilmesi için pthread_key_create fonksiyonunda      ↵
      "destructor" parametresi
13608     kullanılmıştır. Destructor parametresiyle verilen fonksiyon      ↵
      pthread_key_delete tarafından değil thread sonlanırken çağrılmaktadır. ↵
13609     Ancak proses sonlanırken bu destructor fonksiyonları çağrılmaz.      ↵
      Destructor fonksiyonun çağrılmaması için slotta NULL dışında bir değerin      ↵
      bulunuyor olması gereklidir.
13610     (Yani destructor parametresi girildiği halde slota yerleştirme      ↵
      yapılmamışsa bu fonksiyon çağrılmamaktadır.) Destructor fonksiyonu      ↵
      thread pthread_cancel
13611     fonksiyonuyla başka bir thread tarafından sonlandırılırken de      ↵
      çağrılmaktadır. Eğer thread birden fazla slot için destructor      ↵
      fonksiyonuna sahipse
13612     bu durumda her slot için destructor fonksiyonu çağrılr. Ancak buların      ↵
      sırası hakkında bir belirlemede bulunulmamıştır.
13613 -----*/
```

13614
13615 #include <stdio.h>
13616 #include <stdlib.h>
13617 #include <string.h>
13618 #include <pthread.h>

```
13619
13620 #define MAX_ENV      4096
13621
13622 void exit_sys(const char *msg);
13623 void exit_sys_thread(const char *msg, int err);
13624 void *thread_proc1(void *param);
13625 void *thread_proc2(void *param);
13626 char *mygetenv(const char *env);
13627 void destructor(void *param);
13628
13629 extern char **environ;
13630 pthread_key_t g_envkey;
13631 pthread_rwlock_t g_rwlock = PTHREAD_RWLOCK_INITIALIZER;
13632
13633 char *mygetenv(const char *env)
13634 {
13635     char *pval = NULL;
13636     void *pv;
13637     int result;
13638     size_t len;
13639     int i;
13640
13641     if ((pval = (char *)pthread_getspecific(g_envkey)) == NULL) {
13642         if ((pv = malloc(MAX_ENV)) == NULL)
13643             return NULL;
13644         if ((result = pthread_setspecific(g_envkey, pv)) != 0)
13645             exit_sys_thread("pthread_setspecific", result);
13646         pval = (char *)pv;
13647     }
13648
13649     if ((result = pthread_rwlock_rdlock(&g_rwlock)) != 0)
13650         exit_sys_thread("pthread_rwlock_rdlock", result);
13651     for (i = 0; environ[i] != NULL; ++i) {
13652         len = strlen(environ[i]);
13653         if (!strncmp(env, environ[i], len)) {
13654             if (environ[i][len] != '=')
13655                 break;
13656             strcpy(pval, &environ[i][len + 1]);
13657             break;
13658         }
13659     }
13660     if ((result = pthread_rwlock_unlock(&g_rwlock)) != 0)
13661         exit_sys_thread("pthread_rwlock_unlock", result);
13662
13663     return pval;
13664 }
13665
13666 int main(void)
13667 {
13668     int result;
13669     pthread_t tid1, tid2;
13670
13671     if ((result = pthread_key_create(&g_envkey, destructor)) != 0)
```

```
13672     exit_sys_thread("pthread_key_create", result);
13673
13674     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
13675         exit_sys_thread("pthread_create", result);
13676
13677     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
13678         exit_sys_thread("pthread_create", result);
13679
13680     if ((result = pthread_join(tid1, NULL)) != 0)
13681         exit_sys_thread("pthread_join", result);
13682
13683     if ((result = pthread_join(tid2, NULL)) != 0)
13684         exit_sys_thread("pthread_join", result);
13685
13686     if ((result = pthread_key_delete(g_envkey)) != 0)
13687         exit_sys_thread("pthread_key_delete", result);
13688
13689
13690     return 0;
13691 }
13692
13693 void exit_sys(const char *msg)
13694 {
13695     perror(msg);
13696
13697     exit(EXIT_FAILURE);
13698 }
13699
13700 void exit_sys_thread(const char *msg, int err)
13701 {
13702     fprintf(stderr, "%s: %s\n", msg, strerror(err));
13703     exit(EXIT_FAILURE);
13704 }
13705
13706 void *thread_proc1(void *param)
13707 {
13708     char *val;
13709
13710     if ((val = mygetenv("PATH")) == NULL)
13711         exit_sys("mygetenv");
13712     puts(val);
13713
13714     return NULL;
13715 }
13716
13717 void *thread_proc2(void *param)
13718 {
13719     char *val;
13720
13721     if ((val = mygetenv("HOME")) == NULL)
13722         exit_sys("mygetenv");
13723     puts(val);
13724 }
```

```
13725     return NULL;
13726 }
13727
13728 void destructor(void *param)
13729 {
13730     free(param);
13731 }
13732
13733 /
*-----*
----- C11 ile birlikte ve C++11 ile birlikte C ve C++ dillerine thread_local isimli bir yer belirleyicisi de eklenmiştir.
13734 Bir global değişkeni ya da static yerel değişkeni biz bu belirleyici ile bildirdiğimizde derleyici onu thread specific alanda yaratır.
13735 Dolayısıyla bu değişken thread'e özgü global ya da static yerel değişken durumunda olur. Bu sayede biz Windows ve Linux farklı sistemlerde thread'e özgü alanları farklı kodlarla oluşturmak zorunda kalmayız.
13736
13737 Aşağıdaki kodu g++ derleyicisi ile -std=c++11 seçeneği ile derleyiniz.
13738
13739 -----*/
13740
13741 #include <stdio.h>
13742 #include <stdlib.h>
13743 #include <string.h>
13744 #include <unistd.h>
13745 #include <pthread.h>
13746
13747
13748 thread_local int g_i;
13749
13750 void exit_sys_thread(const char *msg, int err);
13751 void *thread_proc1(void *param);
13752 void *thread_proc2(void *param);
13753
13754 int main(void)
13755 {
13756     int result;
13757     pthread_t tid1, tid2;
13758
13759     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
13760         exit_sys_thread("pthread_create", result);
13761
13762     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
13763         exit_sys_thread("pthread_create", result);
13764
13765     if ((result = pthread_join(tid1, NULL)) != 0)
13766         exit_sys_thread("pthread_join", result);
13767
13768     if ((result = pthread_join(tid2, NULL)) != 0)
13769         exit_sys_thread("pthread_join", result);
13770
13771     return 0;
```

```
13772 }
13773
13774 void exit_sys_thread(const char *msg, int err)
13775 {
13776     fprintf(stderr, "%s: %s\n", msg, strerror(err));
13777     exit(EXIT_FAILURE);
13778 }
13779
13780 void *thread_proc1(void *param)
13781 {
13782     for (g_i = 0; g_i < 10; ++g_i) {
13783         printf("thread1: %d\n", g_i);
13784         sleep(1);
13785     }
13786
13787     return NULL;
13788 }
13789
13790 void *thread_proc2(void *param)
13791 {
13792     for (g_i = 0; g_i < 10; ++g_i) {
13793         printf("thread1: %d\n", g_i);
13794         sleep(1);
13795     }
13796
13797     return NULL;
13798 }
13799
13800 /
*-----*
-----*
13801 Birden fazla thread aynı kod üzerinde ilerlerken belli bir kod
13802 paraçasının yalnızca tek bir thread tarafından çalıştırılmasını
13803 isteyebiliriz. Özellikle TSD uygulamalarında bu tür isteklerle
13804 karşılaşılmaktadır. Bu işlemi pthread_once isimli fonksiyon yapar.
13805 Bu fonksiyona biz bir fonksiyon adresi veririz. Peç çok thread
13806 pthread_once fonksiyonunu görse de bu fonksiyon bizim verdığımız
13807 fonksiyonun
13808 yalnızca bir kez ilk thread akışı tarafından çağrılmasını sağlar.
13809
13810 Aşağıdaki örnekte iki thread foo fonksionunu çağrırmıştır. foo içerisinde
13811 iki thread de pthread_once çağrısına girmiştir. Ancak
13812 thread'lerden yalnızca bir tanesi pthread_once fonksiyonunda
13813 belirttiğimiz bar fonksiyonunu çağıracaktır.
13814 -----
13815 #include <stdio.h>
13816 #include <stdlib.h>
13817 #include <string.h>
13818 #include <pthread.h>
13819
13820 void exit_sys_thread(const char *msg, int err);
```

```
13816 void *thread_proc1(void *param);
13817 void *thread_proc2(void *param);
13818 void foo(void);
13819 void bar(void);
13820
13821 pthread_once_t g_once = PTHREAD_ONCE_INIT;
13822
13823 int main(void)
13824 {
13825     int result;
13826     pthread_t tid1, tid2;
13827
13828     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
13829         exit_sys_thread("pthread_create", result);
13830
13831     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
13832         exit_sys_thread("pthread_create", result);
13833
13834     if ((result = pthread_join(tid1, NULL)) != 0)
13835         exit_sys_thread("pthread_join", result);
13836
13837     if ((result = pthread_join(tid2, NULL)) != 0)
13838         exit_sys_thread("pthread_join", result);
13839
13840     return 0;
13841 }
13842
13843 void exit_sys(const char *msg)
13844 {
13845     perror(msg);
13846
13847     exit(EXIT_FAILURE);
13848 }
13849
13850 void exit_sys_thread(const char *msg, int err)
13851 {
13852     fprintf(stderr, "%s: %s\n", msg, strerror(err));
13853     exit(EXIT_FAILURE);
13854 }
13855
13856 void *thread_proc1(void *param)
13857 {
13858     foo();
13859
13860     return NULL;
13861 }
13862
13863 void *thread_proc2(void *param)
13864 {
13865     foo();
13866
13867     return NULL;
13868 }
```

```
13869
13870 void foo(void)
13871 {
13872     int result;
13873
13874     printf("foo called...\n");
13875
13876     if ((result = pthread_once(&g_once, bar)) != 0)
13877         exit_sys_thread("pthread_once", result);
13878 }
13879
13880 void bar(void)
13881 {
13882     printf("bar only called once...\n");
13883 }
13884
13885 /
*-----*
```

-----*

13886 ptherad_once fonksiyonu genellikle TSD kullanan programlarda karşımıza ➔
çıkmaktadır. Programcı TSD slotunu işin başında ➔
thread'leri yaratmadan pthread_key_create ile yaratabilir. Ancak bu ➔
durumda söz konusu thread güvenli fonksiyon hiç çağrılmazsa ➔
bu slot boşuna yaratılmış olacaktır. İşte bu durumu engellemek için ➔
thread güvenli hale getirilen fonksiyon ilk kez çağrıldığında ➔
TSD slotu yaratılır.

13890

13891 Aşağıdaki örnekte myrand ya da mysrand fonksiyonları ilk kez herhangi ➔
bir thread tarafından çağrıldığında TSD slotu pthread_once ➔
fonksiyonu sayesinde yaratılmıştır.

13892 -----*/

13893 -----*/

```
13894
13895 #include <stdio.h>
13896 #include <stdlib.h>
13897 #include <string.h>
13898 #include <unistd.h>
13899 #include <pthread.h>
13900
13901 void mysrand(unsigned seed);
13902 int myrand(void);
13903 void thread_once_proc(void);
13904
13905 void exit_sys_thread(const char *msg, int err);
13906 void *thread_proc1(void *param);
13907 void *thread_proc2(void *param);
13908
13909 pthread_key_t g_key;
13910 pthread_once_t g_once = PTHREAD_ONCE_INIT;
13911
13912 int myrand(void)
13913 {
13914     void *val;
```

```
13915     unsigned seed;
13916     int result;
13917
13918     if ((result = pthread_once(&g_once, thread_once_proc)) != 0)
13919         exit_sys_thread("pthread_once", result);
13920
13921     if ((val = pthread_getspecific(g_key)) == NULL) {
13922         if ((result = pthread_setspecific(g_key, (void *)1)) != 0)
13923             exit_sys_thread("pthread_setspecific", result);
13924         seed = 1;
13925     }
13926     else
13927         seed = (unsigned) val;
13928
13929     seed = seed * 1103515245 + 12345;
13930     if ((result = pthread_setspecific(g_key, (void *)seed)) != 0)
13931         exit_sys_thread("pthread_setspecific", result);
13932
13933     return seed / 65536 % 32768;
13934 }
13935
13936 void mysrand(unsigned seed)
13937 {
13938     int result;
13939
13940     if ((result = pthread_once(&g_once, thread_once_proc)) != 0)
13941         exit_sys_thread("pthread_once", result);
13942
13943     if ((result = pthread_setspecific(g_key, (void *)seed)) != 0)
13944         exit_sys_thread("pthread_setspecific", result);
13945 }
13946
13947 void thread_once_proc(void)
13948 {
13949     int result;
13950
13951     if ((result = pthread_key_create(&g_key, NULL)) != 0)
13952         exit_sys_thread("pthread_key_create", result);
13953 }
13954
13955 int main(void)
13956 {
13957     int result;
13958     pthread_t tid1, tid2;
13959
13960     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
13961         exit_sys_thread("pthread_create", result);
13962
13963     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
13964         exit_sys_thread("pthread_create", result);
13965
13966     if ((result = pthread_join(tid1, NULL)) != 0)
13967         exit_sys_thread("pthread_join", result);
```

```
13968
13969     if ((result = pthread_join(tid2, NULL)) != 0)
13970         exit_sys_thread("pthread_join", result);
13971
13972     return 0;
13973 }
13974
13975 void exit_sys_thread(const char *msg, int err)
13976 {
13977     fprintf(stderr, "%s: %s\n", msg, strerror(err));
13978     exit(EXIT_FAILURE);
13979 }
13980
13981 void *thread_proc1(void *param)
13982 {
13983     int i;
13984     int val;
13985
13986     for (i = 0; i < 10; ++i) {
13987         val = myrand();
13988         printf("Thread-1: %d\n", val % 100);
13989     }
13990     printf("\n");
13991
13992     return NULL;
13993 }
13994
13995 void *thread_proc2(void *param)
13996 {
13997     int i;
13998     int val;
13999
14000     for (i = 0; i < 10; ++i) {
14001         val = myrand();
14002         printf("Thread-2: %d\n", val % 100);
14003     }
14004     printf("\n");
14005
14006     return NULL;
14007 }
14008
14009 /
*-----*
-----*
14010 Aslında Linux işletim sisteminde proses ve thread yaratımı benzer
14011 biçimde yapılmaktadır. Thread'lerin de birer task_struct yapısı
14012 vardır.
14013 Bu task_struct içerisinde tüm prosesler ve prosesin thread'leri bağlı
14014 listelerle tutulmuştur. task_struct içerisindeki alanlar ve anlamları
14015 şöyledir:
14016
14017 - parent ve real_parent: Üst prosesin task_struct adresi
14018 - children: Prosesin alt proses listesindeki ilk alt prosesin
```

```
        task_struct adresi
14016    - sibling: Prosesin alt prosesleri dolaşılırken kullanılan sonraki      ↵
        kardeş prosesin task_struct adresi
14017    - thread_group: Aynı prosesin içerisindeki thread'lerin task_struct      ↵
        listesi. Her task_struct'ın thread_group göstericisi
14018    prosesin sonraki bir thread'inin task_struct yapısını gösterir.
14019    - group_leader: Prosesin ana thread'inin task_struct adresi
14020
14021    Linux işletim sistemi her zaman o anda çalışmakta olan kodun task_struct      ↵
        adresini biliyor durumdadır. Kernel içerisindeki current makrosu
14022    her zaman o anda çalışmakta olan thread'in task_struct adresini      ↵
        vermektedir. Örneğin bir thread read fonksiyonuyla bir dosyadna okuma      ↵
        yapacak olsa
14023    o thread'in task_struct yapısından hareketle prosesin dosya betimleyici      ↵
        tablosuna erişilmektedir. Her task_struct yapısının ayrı bir pid
14024    değeri vardır. Ancak POSIX'in getpid fonksiyonu çağrıldığında o anda      ↵
        çalışmakta olan thread'in task_struct pid'si verilmez. prosesin ana      ↵
        thread'inin
14025    pid'si verilir. Çünkü POSIX standartlarında thread'lerin pid'leri      ↵
        yoktur. Yalnızca proseslerin pid'leri vardır. Linux'a özgü gettid      ↵
        fonksiyonu
14026    aslında o thread'in pid'sini vermektedir. Sistemde yaratılabilen      ↵
        toplam task_struct sayısı bellidir. Bu sayı /proc/sys/kernel/threads-      ↵
        max
14027    dosyasında belirtilmektedir.
14028
14029    Linux'un sys_clone isimli sistem fonksiyonu aslında proses ve thread      ↵
        yaratımının yapıldığı en genel sistem fonksiyonudur.
14030    fork fonksiyonun özel bir clone çağırması olduğunu belirtelim. pthread      ↵
        kütüphanesi de sonuçta thread'i bu clone sistem fonksiyonuyla
14031    yaratmaktadır. clone sistem fonksiyonu bir kütüphane fonksiyonu olarak      ↵
        da (tabii ki POSIX fonksiyonu değil) Linux sistemlerinde      ↵
        bulunmaktadır.
14032
14033    Linux'un çizelgeleyici alt sistemi prosesleri çizelgelemez. Thread'leri      ↵
        çizelgeler. Dolayısıyla aslında çizelgeleyici task_struct'lardan      ↵
        hareketle
14034    context switch yapmaktadır.
14035
14036    -----
14037    -----
14038    /
14039    *
14040    -----
14041    -----
14042    POSIX standartlarına göre thread çizelgelemesi thread'lerin çizelgeleme      ↵
        politikalarına (schedul,ng policies) bağlıdır.
14043    POSIX 4 çizelgeleme politikası tanımlamıştır. Ancak işletim      ↵
        sistemlerinin daha fazla politikaya sahip olabileceği belirtilmiştir.
14044    SCHED_FIFO ve SCHED_RR çizelgeleme politikalarına "gerçek zamanlı (real      ↵
        time)" çizelgeleme politikaları denilmektedir. Thread'lerin
14045    default çizelgeleme politikaları SCHED_OTHER biçimindedir. SCHED_OTHER      ↵
        çizelgeleme politikası POSIX standartlarında tamamen işletim sistemini      ↵
```

yazarların ↗

14043 isteğine bırakılmıştır. Yani pratikte karşılaştığımız çizelgeleme ↗
politikası hep SCHED_OTHER biçimindedir. Bu da işletim sisteminden
14044 işletim sistemine farklılıklar göstermektedir.

14045 ↗

14046 POSIX standartlarında bloke olmamış thread'lerin bir run kuyruğunda ↗
bekledikleri varsayılmaktadır. Çizelgeleyici alt sistemin görevi ise
14047 buradan uygun thread'i seçip CPU'ya atamaktır. SCHED_RR ve SHED_FIFO ↗
çizelgeleme politikalarına sahip olan thread'ler her zaman SCHED_OTHER
14048 çizelgeleme politikasına sahip thread'lerden üstündür. Yani ↗
çizelgeleyici alt sistem her zaman SCHED_FIFO ya da SCHED_RR
polikasına sahip ↗

14049 thread'lere öncelik vermektedir. Başka bir deyişle bir SCHED_OTHER ↗
thread'inin çalışabilmesi için run kuyruğunda SCHED_FIFO ve SCHED_RR ↗
politikalarına
14050 sahip thread'lerin olmaması gereklidir.

14051 ↗

14052 SCHED_FIFO ve SCHED_RR politikalarına sahip thread'lerin birer statik ↗
öncelik derecesi vardır. Bu statik öncelik derecesi [1-99]
14053 arasında bir değere sahiptir. Sistem her zaman run kuyruğunda SCHED_FIFO ↗
ve SCHED_RR thread'lerinin en yüksek öncelikli olanını
14054 alarak CPU'ya verir. Ancak eğer aynı statik önceliğe sahip birden fazla ↗
SCHED_FIFO ve SCHED_RR prosesi varsa kuyrukta önce olan
14055 thread CPU'ya verilmektedir. Bu therad bir quanta süresi kadar ↗
çalıştırılır. Quanta bittiğinde eğer bu thread SCHED_FIFO politikasına
14056 sahip ise kuyruğun başına SCHED_RR politikasına sahipse kuyruğun sonuna ↗
yerleştirilir. Yeniden run kuyruğuna bakılarak en yüksek öncelikli
14057 kuyruğun önündeki thread alınarak CPU'ya atanır. İşlemler böyle devam ↗
ettirilir. Örneğin run kuruğunda şu thread'ler bulunuyor olsun:

14058 FIFO (50), OTHER, RR (50), RR(50) OTHER ↗

14060 ↗

14061 Burada en yüksek statik öncelikli kuyruğun önündeki thread ilk ↗
thread'tir. Sistem bunu CPU atar. Bir quanta çalıştırır.

14062 Quanta bitince kuyruğun başına yerleştirilir. Bu durumda yine onu CPU'ya ↗
atar. Yani bu durumda bu thread sonlanana kadar ya da bloke olup
14063 run kuyruğundan çıkışa kadar çalışacaktır. Şimdi bu thread'in bloke ↗
olduğunu düşünelim. Run kuyruğu şöyle olacaktır:

14064 ↗

14065 OTHER, RR(50), RR(50), OTHER ↗

14066 ↗

14067 Burada ikinci sırada bulunan RR (50) thread'i CPU'ya verilecek ve 1 ↗
quanta çalıştırılacaktır. Quanta bitince kuyruğun sonuna ↗
yerleştirileceği

14068 için artık üçüncü sıradaki RR(50) thread'i CPU'ya verilecektir. Böylece ↗
bu iki thread bloke olana kadar döngüsel çizelgeleme yöntemiyle
14069 çalıştırılacaktır. Görüldüğü gibi SCHED_OTHER thread'ler bunlar varken ↗
çalışma fırsatı bulamayacaktır.

14070 ↗

14071 Bir SCHED_RR ya da SCHED_FIFO thread çalışırken bloke çözülmüş olan daha ↗
yüksek öncelikli bir thread run kuyruğuna yerleştirilirse hemen
14072 context switch yapılarak çalışma bu thread'e verilmektedir. Zaten "real ↗
time" çizelgelemeden beklenen budur. (POSIX sistemleri real time ↗

sistemler değildir.)

14073

14074 Genel olarak SCHED_FIFO ya da SCHED_RR thread'lerin IO yoğun thread'ler olması uygundur. Aksi takirde diğer diğer thread'ler çalışmaya fırsat bulamayacaklardır.

14075

14076

14077 Birden fazla CPU ya da çekirdek olduğu durumda bu CPU'ya ya da çekirdeklerin aynı run kuyruğuna bakarak seçim yaptıkları düşünülmelidir. Örneğin:

14078

14079 FIFO (50), OTHER, RR (50), RR(50) OTHER

14080

14081 Bu durumda iki CPU varsa, sistem FIFO(50) thread'ini bunlardan birine RR (50) thread'ini diğerine atar. Diğer CPU'da RR(50) thread'leri döngüsel çizelgeleme yoluyla çalışacaktır.

14082

14083

14084 SCHED_OTHER politikasının işlevi POSIX standartlarında işletim sisteminin tanımlamasına bırakılmıştır. Ancak bu politikaların SCHED_FIFO ya da SCHED_RR

14085 ile özdeş olabilmesine de izin verilmiştir. Linux işletin sisteminde SCHED_OTHER thread'ler için çizelgeleme algoritması üç kere değiştirilmiştir.

14086

14087 Her ne kadar POSIX standartları SCHED_OTHER thread'ler konusunda belirlemeyi işletim sistemini yazanlara bırakmış olsa da

14088 bu thread'lerin bir dinamik önceliğinin olduğu ve bu önceliğin bu thread'lerin kullanacakları quanta sürelerini belirlemeye

14089 yaradığından bahsetmiştir. Gerçekten de Linux ve diğer POSIX sistemlerinde SCHED_OTHER politikasına sahip thread'lerin birer dinamik

14090 önceligi vardır. Bu dinamik önceliğe "nice değeri" de denilmektedir. Linux'ta SSCHED_OTHER thread'lerinin dinamik önceliği [1-40] arasındadır.

14091 Ancak yüksek öncelik düşük bir quanta süresi anlamına gelmektedir. Bu durumda en çok quanta kullanan dinamik öncelik 1'dir.

14092

14093 Linux sistemlerinde SCHED_OTHER politikasına sahip thread'lerin çizelgeleme algoritmaları üç kere değiştirilmiştir. İlk algoritma

14094 tipik olandır. Buna özel bir isim verilmemiştir. İkinci algoritma Linux 2.6 versiyonlarında kullanılmaya başlanmıştır. Buna

14095 "O(1) Çizelgelemesi" denilmektedir. Nihayet 2.6.23 ile birlikte şu anda kullanılmakta olan "CFS (completely Fair Scheduling)" algoritmasına geçilmiştir.

14096

14097

14098 SCHED_OTHER thread'lerin dinamik öncelikleri onların kaç mili saniye quanta süresi kullanacaklarını dolaylı olarak belirtmektedir. Yani

14099 sistemde n tane SCHED_OTHER thread var ise bunlar döngüsel çizelgelenmeye birlikte hepsi aynı sürede quanta kullanmak zorunda değildir.

14100 Buradakş detay Linux'un versiyonundan versiyonuna değişebilmektedir.

14101

14102 Genel olarak Linux çekirdeğininin pek çok vesiyonunda SCHED_OTHER thread'lerin kullanacakları quanta süreleri onların dinamik

```
        öncelikleri
14103    ile ilişkilendirilmiştir. Eun kuyruğundaki thread'lerin hepsinin quanta →
14104    süresi bitmeden yeniden doldurma yapılmamaktadır. Linux'un →
14105    pek çok versiyonunda timer kesmesi 10 ms'ye kurulmuştur. (Bilgisayarlar →
14106    hızlanınca artık 1 ms'ye ye kurmaya başladılar.) Her timer kesmesi →
14107    oluştuğunda task_struct içerisindeki quanta sayacı olan counter elemanı →
14108    1 eksilttilir. Bu eleman 0'a düşüğünde thread tüm quanta süresini →
14109    kullanmış olur.
14110    İşte run kuyruğundaki tüm thread'lerin coub-nter değerleri 0'a →
14111    düşüğünde yeniden bunlara dinamik öncelik temelinde yeni değerler →
14112    atanmaktadır.
14113    CFS algoritmasına kadar Linux çekirdekleri genel olarak run kuyruğunda →
14114    counter değeri en yüksek olan SCHED_OTHER thread'i CPU'ya atamaktadır. →
14115
14116    Böylece o zamana kadar az CPU kullanan thread'lere öncelik verilmiş →
14117    olmaktadır. Linux'un CFS sistemine kadar SCHED_OTHER therad'lerin →
14118    dinamik öncelikleri
14119    default durumda 20 idi. Bu da 200 ms. bir quanta süresine karşılık →
14120    gelmektedir. Mademki dinamik öncelik en fazla 1 olabilir. Bu durumda →
14121    (40 - 1 = 39)
14122    thread en fazla 390 ms. quanta süresine sahip olabilir.
14123
14124    -----
14125    -----
14126    ssched_setscheduler isimli POSIX fonksiyonu bir prosesin id değerini →
14127    alarak onun çizelgeleme politikasını ve statik ya da dinamik →
14128    önceliğini değiştirmekte kullanılır. Fonksiyon sturct sched_param →
14129    türünden bir yapı nesnesini de parametre olarak almaktadır.
14130    Bu yapının sched_priority isminde tek bir elemanı vardır. Bu eleman →
14131    SCHED_FIFO ve SCHED_RR için statik öceliği, SCHED_OTHER için →
14132    dinamik önceliği belirtir. POSIX standartlarına göre bir prosesin →
14133    çizelgeleme politikası bu fonksiyonla değiştirildiğinde prosesin →
14134    tüm thread'lerinin çizelgeleme politikalari değiştirilmiş olur. Ancak →
14135    Linux bunu desteklememektedir. Linux'ta bu fonksiyon çağrıldığında →
14136    prosesin yalnızca ana thread'inin çizelgeleme politikası değiştirilmiş →
14137    olur. Prosesin diğer thread'lerinin çizelgeleme politikası →
14138    Linux'ta bu fonksiyonun pid parametresi yerine gettid fonksiyonuyla elde →
14139    edilen değerin verilmesiyle yapılabilmektedir. Tabii prosesin →
14140    çizelgeleme politikasının değiştirilebilmesi ancak root proses için →
14141    mümkün değildir. Fonksiyonun pid parametresi yerine 0 da girilebilir.
14142    Bu durumda zaten getpid() anlaşılmaktadır.
14143
14144    -----
14145    -----
14146    #include <stdio.h>
14147    #include <stdlib.h>
14148    #include <unistd.h>
14149    #include <sched.h>
14150
```

```
14131 void exit_sys(const char *msg);
14132
14133 int main(void)
14134 {
14135     struct sched_param sparam;
14136
14137     sparam.sched_priority = 50;
14138
14139     if (sched_setscheduler(getpid(), SCHED_FIFO, &sparam) == -1)
14140         exit_sys("sched_setscheduler");
14141
14142     printf("Ok\n");
14143
14144     return 0;
14145 }
14146
14147 void exit_sys(const char *msg)
14148 {
14149     perror(msg);
14150
14151     exit(EXIT_FAILURE);
14152 }
14153
14154 /
*-----*
-----*
14155     Belli bir thread'in (ana thread de dahil olmak üzere) çizelgeleme      ↗
14156     politikası ve thread önceliği pthread_getschedparam POSIX fonksiyonu    ↗
14157     ile alınıp pthread_setschedparam POSIX fonksiyonuyla set edilebilir.    ↗
14158     Tabii bu işlemin yapılabilmesi için prosesin yine root önceliğinde    ↗
14159     olması gereklidir.                                                 ↗
-----*/
```

```
14159
14160 #include <stdio.h>
14161 #include <stdlib.h>
14162 #include <string.h>
14163 #include <unistd.h>
14164 #include <pthread.h>
14165
14166 void exit_sys_thread(const char *msg, int err);
14167 void *thread_proc(void *param);
14168
14169 int main(void)
14170 {
14171     int result;
14172     pthread_t tid;
14173     struct sched_param param;
14174
14175     if ((result = pthread_create(&tid, NULL, thread_proc, NULL)) != 0)
14176         exit_sys_thread("pthread_create", result);
14177
14178     param.sched_priority = 30;
```

```
14179
14180     if ((result = pthread_setschedparam(tid, SCHED_FIFO, &param)) != 0)
14181         exit_sys_thread("pthread_setschedparam", result);
14182
14183     if ((result = pthread_join(tid, NULL)) != 0)
14184         exit_sys_thread("pthread_join", result);
14185
14186     return 0;
14187 }
14188
14189 void exit_sys_thread(const char *msg, int err)
14190 {
14191     fprintf(stderr, "%s: %s\n", msg, strerror(err));
14192     exit(EXIT_FAILURE);
14193 }
14194
14195 void *thread_proc(void *param)
14196 {
14197     int i;
14198
14199     for (i = 0; i < 10; ++i) {
14200         printf("%d ", i);
14201         fflush(stdout);
14202         sleep(1);
14203     }
14204     printf("\n");
14205
14206     return NULL;
14207 }
14208
14209 /
*-----*
-----*
14210     gettid fonksiyonu belli bir süredir GNU kütüphanesinde bulunmamaktadır. ↗
14211     Bu Linux'ta bir sistem fonksiyonudur. Bu nedenle ↗
14212     bu fonksiyon syscall fonksiyonu ile çağrılabılır. Aşağıdaki örnekte ana ↗
14213     thread'in ve yaratılan thread'in task_struct pid değerleri ↗
14214     ekrana yazdırılmıştır. Tabii ana thread'in task_struct pid değeri ↗
14215     aslında getpid fonksiyonuyla da elde edilebilir. ↗
14216 -----*/
14217
14218 #include <stdio.h>
14219 #include <stdlib.h>
14220 #include <string.h>
14221 #include <sys/types.h>
14222 #include <unistd.h>
14223 #include <pthread.h>
14224 #include <sys/syscall.h>
```

```
14225
14226 void exit_sys_thread(const char *msg, int err);
14227 void *thread_proc(void *param);
14228
14229 int main(void)
14230 {
14231     int result;
14232     pthread_t tid;
14233     struct sched_param param;
14234     long task_pid;
14235
14236     if ((task_pid = syscall(SYS_gettid)) == -1) {           // quivalent getpid ↴
14237         perror("syscall");
14238         exit(EXIT_FAILURE);
14239     }
14240
14241     printf("Main thread task_struct pid: %ld\n", task_pid);
14242
14243     if ((result = pthread_create(&tid, NULL, thread_proc, NULL)) != 0)
14244         exit_sys_thread("pthread_create", result);
14245
14246     param.sched_priority = 30;
14247
14248     if ((result = pthread_setschedparam(tid, SCHED_FIFO, &param)) != 0)
14249         exit_sys_thread("pthread_setschedparam", result);
14250
14251     if ((result = pthread_join(tid, NULL)) != 0)
14252         exit_sys_thread("pthread_join", result);
14253
14254     return 0;
14255 }
14256
14257 void exit_sys_thread(const char *msg, int err)
14258 {
14259     fprintf(stderr, "%s: %s\n", msg, strerror(err));
14260     exit(EXIT_FAILURE);
14261 }
14262
14263 void *thread_proc(void *param)
14264 {
14265     int i;
14266     long task_pid;
14267
14268     if ((task_pid = syscall(SYS_gettid)) == -1) {
14269         perror("syscall");
14270         exit(EXIT_FAILURE);
14271     }
14272
14273     printf("Thread task_struct pid: %ld\n", task_pid);
14274
14275     for (i = 0; i < 100; ++i) {
14276         printf("%d ", i);
```

```
14277         fflush(stdout);
14278         sleep(1);
14279     }
14280     printf("\n");
14281
14282     return NULL;
14283 }
14284
14285 /
*-----*
```

14286 Proseslere ilişkin bilgiler bilindiği gibi ps komutuyla elde edilmektedir. ps komutunda -o parametresi istenilen sütunları ayarlamak için
14287 kullanılmaktadır. -T thread bilgilerini de verir. Örneğin:

14288

```
14289 ps -a -T -o pid,pri,tid,cmd,policy
```

14290

14291 Burada aynı kullanıcının tüm terminallerde çalışan prosesleri (-a) ve threadleri (-T) görüntülemiştir. Görüntülemede
14292 pid, pri (öncelik), tid (task_pid), cmd (komut), ve policy (çizelgeleme politikası) sütunları kullanılmıştır. Örneğin:

14293

| PID | PRI | TID | CMD | POL |
|-------|-----|-------|-----------------------------|-----|
| 13480 | 19 | 13480 | sudo ./sample | TS |
| 13482 | 19 | 13482 | ./sample | TS |
| 13482 | 70 | 13483 | ./sample | FF |
| 13566 | 19 | 13566 | ps -a -T -o pid,pri,tid,cmd | TS |

14294

14295

14296

14297

14298

14299

14300 Burada ./sample satırlarında pid değeri tid değerine eşit olan
satırda ./sample prosesin ana thread'ini diğer ise
14301 sonradan yaratılan thread'i belirtmektedir. PRI sütunu SCHED_OTHER prosesler için dinamik önceliği belirtmektedir.

14302 SCHED_FIFO ve SCHED_RR proseslerin statik öncelikleri makismum dinamik önceliğe (40) eklenerek gösterilmektedir. Örneğin

14303 13483 task_pid değerine sahip olan thread'in çizelgeleme politikası SCHED_FIFO biçimindedir. Tatik önceliği 30'dur.

14304

14305 pstree isimli kabul komutu prosesleri (ve onların thread'lerini) ve alt prosesleri bir ağaç biçiminde göstermektedir.

14306 Tipik olarak -p <pid> seçeneğiyse kullanılmaktadır.

14307

14308 Komut satırında bir prosesi belli bir çizelgeleme politikası ve statik/dinamik öncelikle çalıştırabilme için chrt komutu
14309 kullanılmaktadır. Komutta sırasıyla önce çizelgeleme politikası sonra öncelik sonra da çalıştırılacak komut belirtilir. Örneğin:

14310

```
14311 sudo chrt --fifo 30 ls
```

14312

14313 Burada ls programı SCHED_FIFO çizelgeleme politikasıyla 30 statik önceliğe sahip olacak biçimde çalıştırılmaktadır.

14314 Burada çalıştırılan program (örnekte ls) eğer thread yaratırsa bu thread'ler de SCHED_FIFO politikasına sahip olacaktır.

```
14315     Çünkü Linux'ta (ve POSIX standartlarında da böyle) thread'lerin      ↵
14316         çizelgeleme politikaları onu yaratan thread'ten alınmaktadır.
14317
14318     chrt komutu ile çalışmakta olan programların da çizelgeleme politikaları ↵
14319         değiştirilebilir. Bunun için program ismi yerine
14320         prosesin id'si belirtilmektedir. Örneğin:
14321
14322     sudo chrt -f -p 30 13482
14323 / *
14324
14325
14326
14327
14328
14329
14330 #include <stdio.h>
14331 #include <stdlib.h>
14332 #include <string.h>
14333 #include <sys/types.h>
14334 #include <unistd.h>
14335 #include <pthread.h>
14336
14337 void exit_sys_thread(const char *msg, int err);
14338 void *thread_proc(void *param);
14339
14340 int main(void)
14341 {
14342     int result;
14343     pthread_t tid;
14344     pthread_attr_t attr;
14345     struct sched_param param;
14346
14347     if ((result = pthread_attr_init(&attr)) != 0)
14348         exit_sys_thread("pthread_attr_init", result);
14349
14350     if ((result = pthread_attr_setschedpolicy(&attr, SCHED_FIFO)) != 0)
14351         exit_sys_thread("pthread_attr_setschedpolicy", result);
14352
14353     param.sched_priority = 30;
14354     if ((result = pthread_attr_setschedparam(&attr, &param)) != 0)
14355         exit_sys_thread("pthread_attr_setschedparam", result);
```

```

14356
14357     if ((result = pthread_attr_setinheritsched(&attr,
14358         PTHREAD_EXPLICIT_SCHED)) != 0)
14359         exit_sys_thread("pthread_attr_setinheritsched", result);
14360
14361     if ((result = pthread_create(&tid, &attr, thread_proc, NULL)) != 0)
14362         exit_sys_thread("pthread_create", result);
14363
14364     pthread_attr_destroy(&attr);
14365
14366     if ((result = pthread_join(tid, NULL)) != 0)
14367         exit_sys_thread("pthread_join", result);
14368
14369     return 0;
14370 }
14371 void exit_sys_thread(const char *msg, int err)
14372 {
14373     fprintf(stderr, "%s: %s\n", msg, strerror(err));
14374     exit(EXIT_FAILURE);
14375 }
14376
14377 void *thread_proc(void *param)
14378 {
14379     int i;
14380
14381     for (i = 0; ; ++i) {
14382         printf("%d ", i);
14383         fflush(stdout);
14384         sleep(1);
14385     }
14386     printf("\n");
14387
14388     return NULL;
14389 }
14390
14391 /
*-----*
-----*
14392 SCHED_OTHER politikasına sahip prosesin dinamik önceliği nice isimli      ↗
14393 POSIX fonksiyonuyla değiştirilebilmektedir. nice fonksiyonu      ↗
14394 mevcut dinamik önceliğe artırır ya da eksiltir yapar. Artırmak quantayı  ↗
14395 düşürmek eksiltmek quantayı yükseltmek anlamına gelmektedir.
14396 (Örneğin 1 nice değeri 10 ms. gibi bir etkiye yol açtığı      ↗
14397 varsayılabılır.) POSIX standartlarına göre nice fonksiyonu prosesin      ↗
14398 tüm thread'leri üzerinde etkili olmaktadır. Ancak Linux'ta yalnızca ana  ↗
14399 thread üzerinde etkili olur. Linux işletim sisteminde bir thread
14400 çizelgeleme politikasını ve öncelik derecesini onu yaratan thread'ten      ↗
14401 almaktadır. Yine nice ile dinamik önceliğin yükseltilmesi
14402 için prosesin root önceliğinde olması gerekmektedir.
14403 -----*/
14404
14405
14406
14407
14408
14409
14410
14411
14412
14413
14414
14415
14416
14417
14418
14419
14420
14421
14422
14423
14424
14425
14426
14427
14428
14429
14430
14431
14432
14433
14434
14435
14436
14437
14438
14439
14440
14441
14442
14443
14444
14445
14446
14447
14448
14449
14450
14451
14452
14453
14454
14455
14456
14457
14458
14459
14460
14461
14462
14463
14464
14465
14466
14467
14468
14469
14470
14471
14472
14473
14474
14475
14476
14477
14478
14479
14480
14481
14482
14483
14484
14485
14486
14487
14488
14489
14490
14491
14492
14493
14494
14495
14496
14497
14498
14499
14500
14501
14502
14503
14504
14505
14506
14507
14508
14509
14510
14511
14512
14513
14514
14515
14516
14517
14518
14519
14520
14521
14522
14523
14524
14525
14526
14527
14528
14529
14530
14531
14532
14533
14534
14535
14536
14537
14538
14539
14540
14541
14542
14543
14544
14545
14546
14547
14548
14549
14550
14551
14552
14553
14554
14555
14556
14557
14558
14559
14560
14561
14562
14563
14564
14565
14566
14567
14568
14569
14570
14571
14572
14573
14574
14575
14576
14577
14578
14579
14580
14581
14582
14583
14584
14585
14586
14587
14588
14589
14590
14591
14592
14593
14594
14595
14596
14597
14598
14599
14600
14601
14602
14603
14604
14605
14606
14607
14608
14609
14610
14611
14612
14613
14614
14615
14616
14617
14618
14619
14620
14621
14622
14623
14624
14625
14626
14627
14628
14629
14630
14631
14632
14633
14634
14635
14636
14637
14638
14639
14640
14641
14642
14643
14644
14645
14646
14647
14648
14649
14650
14651
14652
14653
14654
14655
14656
14657
14658
14659
14660
14661
14662
14663
14664
14665
14666
14667
14668
14669
14670
14671
14672
14673
14674
14675
14676
14677
14678
14679
14680
14681
14682
14683
14684
14685
14686
14687
14688
14689
14690
14691
14692
14693
14694
14695
14696
14697
14698
14699
14700
14701
14702
14703
14704
14705
14706
14707
14708
14709
14710
14711
14712
14713
14714
14715
14716
14717
14718
14719
14720
14721
14722
14723
14724
14725
14726
14727
14728
14729
14730
14731
14732
14733
14734
14735
14736
14737
14738
14739
14740
14741
14742
14743
14744
14745
14746
14747
14748
14749
14750
14751
14752
14753
14754
14755
14756
14757
14758
14759
14760
14761
14762
14763
14764
14765
14766
14767
14768
14769
14770
14771
14772
14773
14774
14775
14776
14777
14778
14779
14780
14781
14782
14783
14784
14785
14786
14787
14788
14789
14790
14791
14792
14793
14794
14795
14796
14797
14798
14799
14800
14801
14802
14803
14804
14805
14806
14807
14808
14809
14810
14811
14812
14813
14814
14815
14816
14817
14818
14819
14820
14821
14822
14823
14824
14825
14826
14827
14828
14829
14830
14831
14832
14833
14834
14835
14836
14837
14838
14839
14840
14841
14842
14843
14844
14845
14846
14847
14848
14849
14850
14851
14852
14853
14854
14855
14856
14857
14858
14859
14860
14861
14862
14863
14864
14865
14866
14867
14868
14869
14870
14871
14872
14873
14874
14875
14876
14877
14878
14879
14880
14881
14882
14883
14884
14885
14886
14887
14888
14889
14890
14891
14892
14893
14894
14895
14896
14897
14898
14899
14900
14901
14902
14903
14904
14905
14906
14907
14908
14909
14910
14911
14912
14913
14914
14915
14916
14917
14918
14919
14920
14921
14922
14923
14924
14925
14926
14927
14928
14929
14930
14931
14932
14933
14934
14935
14936
14937
14938
14939
14940
14941
14942
14943
14944
14945
14946
14947
14948
14949
14950
14951
14952
14953
14954
14955
14956
14957
14958
14959
14960
14961
14962
14963
14964
14965
14966
14967
14968
14969
14970
14971
14972
14973
14974
14975
14976
14977
14978
14979
14980
14981
14982
14983
14984
14985
14986
14987
14988
14989
14990
14991
14992
14993
14994
14995
14996
14997
14998
14999
149999

```

```
14400 #include <stdio.h>
14401 #include <stdlib.h>
14402 #include <string.h>
14403 #include <sys/types.h>
14404 #include <unistd.h>
14405 #include <pthread.h>
14406
14407 void exit_sys_thread(const char *msg, int err);
14408 void *thread_proc(void *param);
14409
14410 int main(void)
14411 {
14412     int result;
14413     pthread_t tid;
14414
14415     if ((result = pthread_create(&tid, NULL, thread_proc, NULL)) != 0)
14416         exit_sys_thread("pthread_create", result);
14417
14418     if (nice(-10) == -1) {
14419         perror("nice");
14420         exit(EXIT_FAILURE);
14421     }
14422
14423     if ((result = pthread_join(tid, NULL)) != 0)
14424         exit_sys_thread("pthread_join", result);
14425
14426     return 0;
14427 }
14428
14429 void exit_sys_thread(const char *msg, int err)
14430 {
14431     fprintf(stderr, "%s: %s\n", msg, strerror(err));
14432     exit(EXIT_FAILURE);
14433 }
14434
14435 void *thread_proc(void *param)
14436 {
14437     int i;
14438
14439     for (i = 0; ; ++i) {
14440         printf("%d ", i);
14441         fflush(stdout);
14442         sleep(1);
14443     }
14444     printf("\n");
14445
14446     return NULL;
14447 }
14448
14449 /
*-----→
-----→
14450 Komut satırından bir programı SCHED_OTHER olarak dinamik önceliği -20, →
```

```
        +19 arasında değiştirek çalıştırabilmek için
14451    nice komutu kullanılmaktadır. Örneğin:
14452
14453    sudo nice -n -10 ./sample
14454
14455    Ayrıca bir de renice isimli bir komut vardır. Ancak bu komut zaten      ↵
14456    çalışmakta olan bir prosesin nice değerini değiştirmek için
14457    kullanılır.
14458    -----
14459    /
14460    -----
14461    Linux sistemlerinde nice fonksiyonunun POSIX uyumlu olmadığını      ↵
14462    belirtmiştik. Bu fonksiyon Linuc sistemlerinde yalnızca      ↵
14463    ana thread'in dinamik önceliğini değiştiryordu. Pekiye Linux'ta      ↵
14464    herhangi bir SCHED_OTHER politikasına sahip thread'in      ↵
14465    dinamik önceliği nasıl değiştirilmektedir? Bunun için aslında      ↵
14466    ssched_setscheduler fonksiyonu gettid ile kullanılabilir.
14467    Ya da benzer biçimde pthread_setschedparam fonksiyonu da kullanılabilir. ↵
14468    Fakat sıfır bu iş için getpriority ve
14469    setpriority POSIX fonksiyonları da kullanılmaktadır.
14470
14471
14472
14473
14474
14475
14476
14477
14478
14479
14480
14481
14482
14483
14484
14485
14486
14487
14488
14489
14490
14491
14492
14493
```

```
14494 {
14495     fprintf(stderr, "%s: %s\n", msg, strerror(err));
14496     exit(EXIT_FAILURE);
14497 }
14498
14499 void *thread_proc(void *param)
14500 {
14501     int i;
14502     long task_pid;
14503
14504     if ((task_pid = syscall(SYS_gettid)) == -1) {
14505         perror("syscall");
14506         exit(EXIT_FAILURE);
14507     }
14508
14509     if (setpriority(PRI_O_PROCESS, task_pid, -10) == -1) {
14510         perror("setpriority");
14511         exit(EXIT_FAILURE);
14512     }
14513
14514     for (i = 0; ; ++i) {
14515         printf("%d ", i);
14516         fflush(stdout);
14517         sleep(1);
14518     }
14519     printf("\n");
14520
14521     return NULL;
14522 }
14523
14524 /
*-----*-----*-----*
-----*
-----*
```

14525 Belli thread'lerin belli CPU ya da çekirdeklere atanması paralel programlama uygulamalarında gerekebilmektedir.

14526 Bir thread'in hangi CPU ya da çekirdeklere atanabileceğinin "affinity mask" denilen bir özellikle belirlenmektedir. Default durumda thread'ler mevcut tüm CPU ya da çekirdeklere atanabilirler. Biz thread'leri farklı CPU ya da çekirdeklere atayarak onların aynı anda birlikte çalışmasını sağlayabiliyoruz.

14529

14530 Linux sistemlerinde affinity işlemleri için 4 fonksiyon kullanılmaktadır. `sched_getaffinity` bir prosesin (prosesin ana thread'inin)

14531 affinity değerini verir, `sched_setaffinity` fonksiyonu ise bunu set etmemizi sağlar. Aşağıdaki örnekte 2 çekirdekli bir sistemde prosesin ana thread'ının hangi CPU ya da çekirdeklere atanabileceklerini gösterilmiştir. Default durumda işletim sistemi thread'leri tüm CPU ya da çekirdeklere atayabilmektedir.

14534 -----*/

14535

14536 #define __GNU_SOURCE

```
14537
14538 #include <stdio.h>
14539 #include <stdlib.h>
14540 #include <unistd.h>
14541 #include <sched.h>
14542
14543 void exit_sys(const char *msg);
14544
14545 int main(void)
14546 {
14547     cpu_set_t set;
14548     int i;
14549
14550     if (sched_getaffinity(getpid(), sizeof(set), &set) == -1)
14551         exit_sys("sched_getaffinity");
14552
14553     for (i = 0; i < 2; ++i)
14554         printf("%d-CPU: %s\n", i, CPU_ISSET(i, &set) ? "YES" : "NO");
14555
14556     return 0;
14557 }
14558
14559 void exit_sys(const char *msg)
14560 {
14561     perror(msg);
14562
14563     exit(EXIT_FAILURE);
14564 }
14565
14566 /
*-----*-----*-----*
```

14567 sched_setaffinity fonksiyonu belli bir prosesin (prosesin ana
thread'inin) belli CPU ya da çekirdeklerde çalışmasını
sağlamak için kullanılmaktadır. Bunun için prosesin etkin kullanıcı
id'sinin affinity değişikliği yapılacak prosesin
gerçek ya da etkin kullanıcı id'si ile aynı olması ya da root önceliğine
sahip olması ya da CAP_SYS_NICE yeteneğine sahip
olması gerekmektedir. Biz bu fonksiyon ile bir prosesin belli bir
thread'inin affinity'sini de değiştirebiliriz. Tabii bunun için
ilgili thread'in task_struct pid değerinin getpid sistem fonksiyonuyla
elde edilmesi gerekmektedir.

14572

14573 Programı çalıştırıp aşağıdaki gibi ps komutuyla prosesin ana thread'inin
1 numaralı CPU'ya atanıp atanmadığını kontrol ediniz:

14574

14575 ps -a -o pid,psr,cmd

14576 -----*/

14577

14578 #define __GNU_SOURCE

14579

14580 #include <stdio.h>

```
14581 #include <stdlib.h>
14582 #include <unistd.h>
14583 #include <sched.h>
14584
14585 void exit_sys(const char *msg);
14586
14587 int main(void)
14588 {
14589     cpu_set_t set;
14590
14591     CPU_ZERO(&set);
14592     CPU_SET(1, &set);
14593
14594     if ((sched_setaffinity(getpid(), sizeof(set), &set)) == -1)
14595         exit_sys("sched_setaffinity");
14596
14597     for (;;)
14598         ;
14599
14600     return 0;
14601 }
14602
14603 void exit_sys(const char *msg)
14604 {
14605     perror(msg);
14606
14607     exit(EXIT_FAILURE);
14608 }
14609
14610 /
*-----*
-----*
14611     Belli bir thread'e affinity uygulamak için yukarıda belirtildiği gibi
14612     gettid ile sched_setaffinity fonksiyonu kullanılabilir.
14613     Ancak Linux'a özgü bir biçimde zaten pthread_setaffinity_np ve
14614     pthread_getaffinity_np isimli fonksiyonlar da bulunmaktadır.
14615     Bu fonksiyonlar doğrudan thread id'lerle çalışmaktadır. Aşağıdaki
14616     programda iki thread farklı CPU ya da çekirdeklerle atanarak bunlara
14617     CPU yoğun
14618     iş yaptırılmıştır.
14619 -----*/
14620
14621 #define _GNU_SOURCE
14622
14623 #include <stdio.h>
14624 #include <stdlib.h>
14625 #include <string.h>
14626 #include <sched.h>
14627 #include <pthread.h>
14628
14629 void exit_sys_thread(const char *msg, int err);
14630 void *thread_proc1(void *param);
```

```
14627 void *thread_proc2(void *param);
14628
14629 int main(void)
14630 {
14631     int result;
14632     pthread_t tid1, tid2;
14633     cpu_set_t set;
14634
14635     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
14636         exit_sys_thread("pthread_create", result);
14637
14638     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
14639         exit_sys_thread("pthread_create", result);
14640
14641     CPU_ZERO(&set);
14642     CPU_SET(0, &set);
14643
14644     if ((result = pthread_setaffinity_np(tid1, sizeof(set), &set)) != 0)
14645         exit_sys_thread("pthread_setaffinty", result);
14646
14647     CPU_ZERO(&set);
14648     CPU_SET(1, &set);
14649
14650     if ((result = pthread_setaffinity_np(tid1, sizeof(set), &set)) != 0)
14651         exit_sys_thread("pthread_setaffinty", result);
14652
14653     if ((result = pthread_join(tid1, NULL)) != 0)
14654         exit_sys_thread("pthread_join", result);
14655
14656     if ((result = pthread_join(tid2, NULL)) != 0)
14657         exit_sys_thread("pthread_join", result);
14658
14659     return 0;
14660 }
14661
14662 void exit_sys(const char *msg)
14663 {
14664     perror(msg);
14665
14666     exit(EXIT_FAILURE);
14667 }
14668
14669 void exit_sys_thread(const char *msg, int err)
14670 {
14671     fprintf(stderr, "%s: %s\n", msg, strerror(err));
14672     exit(EXIT_FAILURE);
14673 }
14674
14675 void *thread_proc1(void *param)
14676 {
14677     int i;
14678
14679     for (i = 0;i < 1000000000; ++i)
```

```
14680      ;
14681
14682     return NULL;
14683 }
14684
14685 void *thread_proc2(void *param)
14686 {
14687     int i;
14688
14689     for (i = 0;i < 1000000000; ++i)
14690         ;
14691
14692     return NULL;
14693 }
14694
14695 /
-----*-----*-----*
```

14696 Sinyal oluştuğunda çağrılacak fonksiyonu (signal handler) set etmek için iki POSIX fonksiyonu kullanılmaktadır: signal ve sigaction.

14697 signal fonksiyonu eskidir ve maalesef semantik konusunda problemleri vardır. Bu nedenle signal fonksiyonunu değişik sistemlerdeğişik biçimde gerçekleştirmiştir. sigaction fonksiyonunda bu semantik kusurlar ortadan kaldırılmıştır.

14699 Aşağıda SIGINT sinyali oluştuğunda (Klavyeden Ctrl + C tuşuna basıldığında SIGINT sinyali oluşmaktadır) çağrılacak fonksiyon set edilmiştir.

14700 -----*/

```
14703
14704 #include <stdio.h>
14705 #include <stdlib.h>
14706 #include <unistd.h>
14707 #include <signal.h>
14708
14709 void exit_sys(const char *msg);
14710 void sigint_handler(int sno);
14711
14712 int main(void)
14713 {
14714     int i;
14715
14716     if (signal(SIGINT, sigint_handler) == SIG_ERR)
14717         exit_sys("signal");
14718
14719     for (i = 0; i < 20; ++i) {
14720         printf("%d\n", i);
14721         sleep(1);
14722     }
14723
14724     return 0;
14725 }
```

```
14726
14727 void exit_sys(const char *msg)
14728 {
14729     perror(msg);
14730
14731     exit(EXIT_FAILURE);
14732 }
14733
14734 void sigint_handler(int sno)
14735 {
14736     printf("SIGINT handler running...\n");
14737 }
14738
14739 /
*-----*
-----*
14740     signal fonksiyonun ikinci parametresine SIG_DFL özel değeri geçilirse →
14741         sinyal fonksiyonu default duruma çekilir. Yani artık →
14742         sinyal oluştuğunda "default action" uygulanır. Bu parametreye SIG_IGN →
14743         özel değeri geçilirse sinyal işletim sistemi tarafından →
14744         görmeden gelinir (ignore edilir).
14745 -----*/ →
14744
14745 #include <stdio.h>
14746 #include <stdlib.h>
14747 #include <unistd.h>
14748 #include <signal.h>
14749
14750 void exit_sys(const char *msg);
14751
14752 int main(void)
14753 {
14754     int i;
14755
14756     if (signal(SIGINT, SIG_IGN) == SIG_ERR)
14757         exit_sys("signal");
14758
14759     for (i = 0; i < 20; ++i) {
14760         printf("%d\n", i);
14761         sleep(1);
14762     }
14763
14764     return 0;
14765 }
14766
14767 void exit_sys(const char *msg)
14768 {
14769     perror(msg);
14770
14771     exit(EXIT_FAILURE);
14772 }
14773
```

```
14774  /
14775      *-----*
14775      ----- kill isimli POSIX fonksiyonu (ismi yanlış uydurulmuştur) bir prosese ya →
14775          da proses grubuna sinyal göndermek için kullanılmaktadır.
14776      Eğer birinci parametresi olan pid "> 0" ise spesifik prosese sinyal →
14776          gönderir. Eğer bu parametre "= 0" ise kendi proses grubundaki
14777          tüm proseslere sinyal gönderir. Eğer bu parametre "< 0" ise abs(pid) →
14777          değerine sahip proses grubun tüm proseslerine sinyal gönderir.
14778      Eğer bu parametre "= -1" ise sinyal gönderebileceği tüm proseslere →
14778          sinyal gönderir.
14779
14780      kill fonksiyonu ile sinyal gönderebilmek için gönderen prosesin gerçek →
14780          ya da etkin kullanıcı id'sinin gönderilen prosesin gerçek ya da
14781          saved set kullanıcı id'sine eşit olması gereklidir. Tabii root prosesi her →
14781          prosese sinyal gönderebilir. (Ya da Linux'ta CAP_KILL yeteneğine
14782          sahip prosesler de tüm proseslere sinyal gönderebilirler.)
14783
14784      Aşağıda proc2 programı SIGUSR1 numaları sinyali proc1'e göndermek için →
14784          yazılmıştır. proc1'i önce çalıştırın. ps -a ile
14785          onun pid değerini bakıp o değerler proc2'yi çalıştırın.
14786  -----
14787
14788 /* proc1.c */
14789
14790 #include <stdio.h>
14791 #include <stdlib.h>
14792 #include <unistd.h>
14793 #include <signal.h>
14794
14795 void exit_sys(const char *msg);
14796 void sigusr1_handler(int sno);
14797
14798 int main(void)
14799 {
14800     int i;
14801
14802     if (signal(SIGUSR1, sigusr1_handler) == SIG_ERR)
14803         exit_sys("signal");
14804
14805     for (i = 0;; ++i) {
14806         printf("%d\n", i);
14807         sleep(1);
14808     }
14809
14810     return 0;
14811 }
14812
14813 void exit_sys(const char *msg)
14814 {
14815     perror(msg);
14816 }
```

```
14817     exit(EXIT_FAILURE);
14818 }
14819
14820 void sigusr1_handler(int sno)
14821 {
14822     printf("SIGUSR1 occurred...\n");
14823 }
14824
14825 /* proc2.c */
14826
14827 #include <stdio.h>
14828 #include <stdlib.h>
14829 #include <signal.h>
14830
14831 void exit_sys(const char *msg);
14832
14833 int main(int argc, char *argv[])
14834 {
14835     pid_t pid;
14836
14837     if (argc != 2) {
14838         fprintf(stderr, "wrong number of arguments!..\n");
14839         exit(EXIT_FAILURE);
14840     }
14841
14842     pid = (pid_t)strtol(argv[1], NULL, 10);
14843
14844     if (kill(pid, SIGUSR1) == -1)
14845         exit_sys("kill");
14846
14847     return 0;
14848 }
14849
14850 void exit_sys(const char *msg)
14851 {
14852     perror(msg);
14853
14854     exit(EXIT_FAILURE);
14855 }
14856
14857 /
*-----*-----*-----*
```

14858 Komut satırından da bir prosese sinyal göndermek için kill isimli komut kullanılmaktadır. kill komutunun basit kullanımı şöyledir: ➔
14859 kill -<numara> <process-id'ler>

14860
14861 Örneğin:
14862 kill -15 15711
14863
14864 Numara yerine sinyallerin sembolik sabit isimleri de kullanılabilir. ➔
14865
14866

Bunun için SIGXXX isimli sinyal -XXX biçiminde belirtilmelidir.

14867
14868 Örneğin:
14869
14870 kill -TERM 15711
14871
14872 Eğer kill komutunda sinyal numarası belirtilmezse -TERM (yani -15) ↗
 belirtilmiş gibi işlem görülür. Bu durumda bir prosesi garanti
14873 sonlandırmak için -KILL ya da -9 seçenekleri kullanılmalıdır.
14874
14875 -----*/
14876 /
*-----
14878 Bir prosesi sonlandırmak için sıklıkla iki sinyal kullanılmaktadır: ↗
 SIGTERM (15) ve SIGKILL (9). SIGTERM sinyali için
14879 sinyal fonksiyonu set edilebilir fakat SIGKILL için edilemez. Benzer ↗
 biçimde SIGTERM sinyali SIG_IGN ile ignore edilebilir ancak
14880 SIGKILL sinyali ignore edilemez. Yani SIGTERM ile sonlandırma garanti ↗
 değildir ancak SIGKILL ile
14881 sonlandırma garantidir. Örneğin:
14882
14883 kill -KILL 15711
14884
14885 Aşağıdaki programı bir terminalde çalıştırığ diğeri ile önce SIGTERM ↗
 sonra da SIGKILL sinyallarını göndermeyi deneyiniz
14886 -----*/
14887
14888 #include <stdio.h>
14889 #include <stdlib.h>
14890 #include <unistd.h>
14891 #include <signal.h>
14892
14893 void exit_sys(const char *msg);
14894 void sigterm_handler(int sno);
14895
14896 int main(void)
14897 {
14898 int i;
14899
14900 if (signal(SIGTERM, sigterm_handler) == SIG_ERR)
14901 exit_sys("signal");
14902
14903 for (i = 0;; ++i) {
14904 printf("%d\n", i);
14905 sleep(1);
14906 }
14907
14908 return 0;
14909 }

```
14910
14911 void exit_sys(const char *msg)
14912 {
14913     perror(msg);
14914
14915     exit(EXIT_FAILURE);
14916 }
14917
14918 void sigterm_handler(int sno)
14919 {
14920     printf("SIGTERM occurred...\n");
14921 }
14922
14923 /
*-----*
-----*
14924 signal fonksiyonu il sinyal set etmek biraz problemlidir. Problem bu →
14925     fonksiyonun davranışının UNIX türevi sistemlerde farklı →
14926 olabilirinden kaynaklanmaktadır. Eski UNIX sistemleri signal fonksiyonu →
14927     ile sinyal set edildiğinde sinyal fonksiyonu çalışırken →
14928 aynı sinyalin oluşmasına izin veriyordu. Böylece aynı sinyal fonksiyonu →
14929     iç içe çalışabiliyordu. Ancak BSD sistemleri sinyal oluştuğunda →
14930 sinyal fonksiyonu çalıştırılırken bu sinyali bloke edip bekletmektedir. →
14931 Ta ki sinyal fonksiyonu işini bitiren kadar. Böylece iç içe →
14932 geçme olmamaktadır. Yıne eski UNIX sistemlerinde signal fonksiyonu ile →
14933     set yapıldığında sinyal fonksiyonu çağrırlar çağrılmaz →
14934 o numaralı sinyal otomatik default'a çekiliyordu (yani set edilmemiş →
14935     duruma getiriliyordu). Ancak BSD sistemleri bunu yapmıyordu. →
14936 Yine bu fonksiyon ile set yapıldığında yavaş sistem fonksiyonlarının →
14937     otomatik restart edilip edilmeyeceği sistemler arasında →
14938 farklılık gösterebiliyordu.
```

14939 Linux işletim sisteminde signal ve sigaction isimli iki sistem →
14940 fonksiyonu vardır. signal sistem fonksiyonu Sistem 5 semantiği ile →
14941 sinyal fonksiyonunu set etmektedir. sigaction modern olandır. Fakat →
14942 glibc kütüphanesindeki signal fonksiyonu glibc 2.0'dan sonra →
14943 signal sistem fonksiyonunu değil sigaction sistem fonksiyonunu çağırarak →
14944 yazılmış durumdadır.

14945 Linux işletim sistemi signal isimli sistem fonksiyonunda eski UNIX →
14946 System-5 semantiğini uygulamaktadır. Yani:

14947 - Sinyal fonksiyonu çalıştırılırken sinyal default'a çekilir.

14948 - Aynı sinyal bloke edilmez. Sinyal fonksiyonu iç içe çalışabilir.

14949 - Yavaş sistem fonksiyonları restart edilmez.

14950 glibc 2.0 öncesinde Linux'taki signal POSIX fonksiyonu signal sistem →
14951 fonksiyonunu çağrıdığı için Sistem 5 semantiğini uyguluyordu. Fakat →
14952 glibc 2.0'dan sonra Linux'taki signal POSIX fonksiyonu sigaction →
14953 fonksiyonu çağrılarak BSD semantiğini uygulamaktadır. Yani:

14954 - Sinyal fonksiyonu çalışırken sinyal default'a çekilmez.

14955 - Aynı sinyal otomatik bloke edilir.

```
14948 - Yavaş sistem fonksiyonları otomatik restart edilir.  
14949  
14950 Maalesef sistem 5 semantiği default'a çekme yüzünden kusurlu →  
      tasarlanmıştır. Çünkü aşağıdaki gibi bir işlemde her zaman  
14951 program çökebilmektedir:  
14952  
14953 void signal_handler(int sno)  
14954 {  
14955     // burada yeniden sinyal gelirse proses sonlanır  
14956     signal(sno, signal_handler);  
14957     ....  
14958 }  
14959  
14960 signal fonksiyonu ile sinyal set etmeye UNIX dünyasında "unreliable" →  
      "signal" da denilmektedir.  
14961  
14962 -----*/  
14963  
14964 / -----*/  
-----  
14965 Sinyal oluştduğunda set çağrılmak üzere set edilen fonksiyonun (signal →  
      handler) int parametresi oluşan sinyalin numarasını  
14966 belirtmektedir. Böylece farklı sinyaller için aynı fonksiyon set →  
      edilebilir ve bu parametreye bakarak ayırtırma yapılabilir.  
14967 -----*/  
14968  
14969 #include <stdio.h>  
14970 #include <stdlib.h>  
14971 #include <unistd.h>  
14972 #include <signal.h>  
14973  
14974 void exit_sys(const char *msg);  
14975 void signal_handler(int sno);  
14976  
14977 int main(void)  
14978 {  
14979     int i;  
14980  
14981     if (signal(SIGUSR1, signal_handler) == SIG_ERR)  
14982         exit_sys("signal");  
14983  
14984     if (signal(SIGUSR2, signal_handler) == SIG_ERR)  
14985         exit_sys("signal");  
14986  
14987     for (i = 0;; ++i) {  
14988         printf("%d\n", i);  
14989         sleep(1);  
14990     }  
14991  
14992     return 0;
```

```
14993 }
14994
14995 void exit_sys(const char *msg)
14996 {
14997     perror(msg);
14998
14999     exit(EXIT_FAILURE);
15000 }
15001
15002 void signal_handler(int sno)
15003 {
15004     if (sno == SIGUSR1)
15005         printf("SIGUSR1\n");
15006     else if (SIGUSR2)
15007         printf("SIGUSR2\n");
15008 }
15009
15010 /
*-----*
```

15011 sigaction fonksiyonun semantığı her sistemde aynıdır. Dolayısıyla
signal fonksiyonu yerine bu fonksiyon tercih edilmelidir. ↗
15012 Bu fonksiyonda struct sigaction isimli bir yapı nesnesinin içi
doldurulur ve fonksiyona verilir. Bu yapının doldurulması gereken
elemanları şunlardır: ↗
15013
15014 sa_handler: Çağrılacak sinyal fonksiyonun adresi buraya yerleştirilir.
15015
15016 sa_mask: Bu eleman sigset_t türündendir. Sinyal fonksiyonu çalıştığı
sürece prosesin signal mask'ine burada belirtilen
15017 sinyaller eklenir. Fonksiyonun çalışması bittiğinde eklenmiş olan bu
sinyaller prosesin signal mask'inden çıkartılır. ↗
15018 Bu bir dizisi üzerinde işlem yapan sigemptyset, sigfillset, sigaddset,
sigdelset ve sigismember isimli fonksiyonlar bulunmaktadır: ↗
15019
15020 sa_flags: Bu elemana SA_XXXX biçiminde çeşitli bayraklar OR'lanarak
girilir. SA_RESETHAND sinyal fonksiyonu çalıştırıldığında
15021 sinyalin default'a çekileceğini belirtir. Bu flag set edilmezse sinyal
default'a çekilmemektedir. SA_NODEFER sinyal fonksiyonu
15022 çalıştığı sürece aynı numaralı sinyalin bloke edilmeyeceği anlamına
gelir. Bu flag belirtilmezse sinyal fonksiyonu çalışırken
15023 sa_mask dikkate alınmaksızın aynı numaralı sinyal bloke edilmektedir.
SA_RESTART yavaş sistem fonksiyonlarının yeniden otomatik
15024 başlatılacağı anlamına gelmektedir. ↗
15025
15026 Sistem 5 semantığı için sa_flags elemanın SA_NODEFER | SA_RESETHAND
15027 biçiminde olması gereklidir. BSD semantığı için ise
flags elemanı SA_RESTART biçiminde olması gereklidir. Bu durumda bu eleman
0'da tutulursa otomatik default'a çekme uygulanmaz,
15028 sinyal fonksiyonu çalıştığı sürece aynı numaralı sinyal bloke edilir ve
sistem fonksiyonları otomatik restart edilmez. Linux'taki
15029 signal POSIX fonksiyonu glibc 20'dan sonra sa_flags = SA_RESTART
biçimindedir ve BSD semantığını uygulamaktadır. ↗

```
15030
15031 -----*/  
15032
15033 #include <stdio.h>
15034 #include <stdlib.h>
15035 #include <unistd.h>
15036 #include <signal.h>
15037
15038 void exit_sys(const char *msg);
15039 void sigint_handler(int sno);
15040
15041 int main(void)
15042 {
15043     struct sigaction act;
15044     int i;
15045
15046     act.sa_handler = sigint_handler;
15047     sigemptyset(&act.sa_mask);
15048     act.sa_flags = 0;
15049
15050     if (sigaction(SIGINT, &act, NULL) == -1)
15051         exit_sys("sigaction");
15052
15053     for (i = 0;; ++i) {
15054         printf("%d\n", i);
15055         sleep(1);
15056     }
15057
15058     return 0;
15059 }
15060
15061 void exit_sys(const char *msg)
15062 {
15063     perror(msg);
15064
15065     exit(EXIT_FAILURE);
15066 }
15067
15068 void sigint_handler(int sno)
15069 {
15070     printf("sigint signal occurs...\n");
15071 }
15072
15073 /
-----*/  

-----  
15074 Her prosesin bloke edilmiş sinyalleri belirten bir signal mask'i vardır. Prosesin signal mask'i Linux sistemlerinde task_struct  
15075 içerisinde saklanmaktadır. Bloke edilen sinyal oluşursa işletim sistemi sinyali prosese teslim etmez (deliver etmez). Onu bekletir.  
15076 Bu beklemedeki bir sinyale "pending" durumda denilmektedir. Eğer proses bloke edilen sinyalin blokesini açarsa bu durumda
```

```
15077     işletim sistemi artık o pending durumda olan sinyali prosese teslim ➔
15078         eder. Prosesin signal mask'ine yeni sinyallerin yerleştirilmesi ➔
15079         ya da oradan sinyal çıkartılması sigprocmask isimli POSIX fonksiyonuyla ➔
15080             yapılmaktadır.
15081
15082     UNIX türevi sistemlerde klasik sinyallerde kuyruklama yoktur. Yani bloke ➔
15083         edilmiş bir sinyal birden fazla kez oluşursa sinyalin blokesi ➔
15084             çözüldüğünde yalnızca 1 kez bu sinyalden oluşur.
15085
15086     Aşağıdaki örnekte SIGINT sinyali önce prosesin signal mask'ine eklenerek ➔
15087         bloke edilmiş sonra da açılmıştır.
15088 -----
15089
15090
15091     #include <stdio.h>
15092     #include <stdlib.h>
15093     #include <unistd.h>
15094     #include <signal.h>
15095
15096     void exit_sys(const char *msg);
15097     void sigint_handler(int sno);
15098
15099     int main(void)
15100     {
15101         int i;
15102         sigset_t sset;
15103         struct sigaction act;
15104
15105         act.sa_handler = sigint_handler;
15106         sigemptyset(&act.sa_mask);
15107         act.sa_flags = 0;
15108
15109         if (sigaction(SIGINT, &act, NULL) == -1)
15110             exit_sys("sigaction");
15111
15112         sigemptyset(&sset);
15113         sigaddset(&sset, SIGINT);
15114
15115         if (sigprocmask(SIG_BLOCK, &sset, NULL) == -1)
15116             exit_sys("sigprocmask");
15117
15118         for (i = 0;; ++i) {
15119             printf("%d\n", i);
15120             if (i == 10)
15121                 if (sigprocmask(SIG_UNBLOCK, &sset, NULL) == -1)
15122                     exit_sys("sigprocmask");
15123             sleep(1);
15124         }
15125
15126         return 0;
15127     }
```

```
15124 void exit_sys(const char *msg)
15125 {
15126     perror(msg);
15127     exit(EXIT_FAILURE);
15128 }
15129
15130
15131 void sigint_handler(int sno)
15132 {
15133     printf("SIGINT occurred...\n");
15134 }
15135
15136 /
*-----*
-----*
15137     pause isimli POSIX fonksiyonu bir sinyal oluşana kadar blokede beklemeye
15138         yol açar. Eğer bir sinyal için sinyal fonksiyonu
15139         set edilmişse pause fonksiyonu sinyal fonksiyonu çalıştırıldıktan sonra
15140             geri dönmektedir. Fonksiyon her zaman -1 değerine geri
15141             döner ve errno değeri de her zaman EINTR olarak set edilir.
15142 -----*/
15143
15144 #include <stdio.h>
15145 #include <stdlib.h>
15146 #include <unistd.h>
15147 #include <signal.h>
15148
15149
15150 void exit_sys(const char *msg);
15151 void sigint_handler(int sno);
15152
15153 int main(void)
15154 {
15155     struct sigaction act;
15156
15157     act.sa_handler = sigint_handler;
15158     sigemptyset(&act.sa_mask);
15159     act.sa_flags = 0;
15160
15161     if (sigaction(SIGINT, &act, NULL) == -1)
15162         exit_sys("sigaction");
15163
15164     printf("waiting for a signal\n");
15165
15166     pause();
15167
15168     printf("ok\n");
15169
15170     return 0;
15171 }
```

```
15172     perror(msg);
15173
15174     exit(EXIT_FAILURE);
15175 }
15176
15177 void sigint_handler(int sno)
15178 {
15179     printf("SIGINT occurred...\n");
15180 }
15181
15182 /
*-----*
-----*
15183     Bazen programlar sonsuz döngüde sinyalleri bekleyerek işlevlerini
15184     gerçekleştirebilmektedir.
-----*/
15185
15186 #include <stdio.h>
15187 #include <stdlib.h>
15188 #include <unistd.h>
15189 #include <signal.h>
15190
15191 void exit_sys(const char *msg);
15192 void sigint_handler(int sno);
15193
15194 int main(void)
15195 {
15196     struct sigaction act;
15197
15198     act.sa_handler = sigint_handler;
15199     sigemptyset(&act.sa_mask);
15200     act.sa_flags = 0;
15201
15202     if (sigaction(SIGINT, &act, NULL) == -1)
15203         exit_sys("sigaction");
15204
15205     printf("waiting for a signal\n");
15206
15207     for (;;)
15208         pause();
15209
15210     return 0;
15211 }
15212
15213 void exit_sys(const char *msg)
15214 {
15215     perror(msg);
15216
15217     exit(EXIT_FAILURE);
15218 }
15219
15220 void sigint_handler(int sno)
```

```
15221 {
15222     printf("SIGINT occurred...\n");
15223 }
15224
15225 /
*-----*
-----*
15226     abort standart C fonksiyonu parametresizdir ve geri dönüş değerine sahip
15227     değildir. Bu fonksiyon anormal çıkışlar için düşünülmüştür.
15228 Bir programda çok ciddi birtakım sorunlar tespit edilmişse çıkış exit
15229     ile değil abort ile yapılmalıdır. abort SIGABRT isimli
15230     sinyalin oluşmasına yol açmaktadır.
15231 -----*/
15232 #include <stdio.h>
15233 #include <stdlib.h>
15234
15235 int main(void)
15236 {
15237     printf("program runs...\n");
15238     abort();
15239
15240     printf("unreachable code...\n");
15241
15242     return 0;
15243 }
15244
15245 /
*-----*
-----*
15246     abort fonksiyonu çağrıldığında SIGABRT sinyali "raise" edilmektedir.
15247     (Sinyalin raise edilmesi sonraki örneklerde ele alınmaktadır.)
15248 Bu durumda eğer sinyal fonksiyonu set edilmişse o fonksiyon çalıştırılır
15249     fonksiyonun çalışması bittiğinde proses sonlandırılır.
15250 SIGABRT sinyali her zaman core dosyasının oluşmasına yol açmaktadır.
15251
15252 Aşağıdaki örnekte abort fonksiyonu çağrılmında set edilen fonksiyonu
15253     çalışacak bu fonksiyon bittiğinde program sonlandırılacaktır.
15254 Yani abort'tan sonraki "ok" yazısı ekranda görülmeyecektir.
15255 -----*/
15256 #include <stdio.h>
15257 #include <stdlib.h>
15258 #include <unistd.h>
15259 #include <signal.h>
15260
15261 void exit_sys(const char *msg);
15262 void sigabrt_handler(int sno);
15263
15264 int main(void)
```

```
15263 {
15264     struct sigaction act;
15265
15266     act.sa_handler = sigabrt_handler;
15267     sigemptyset(&act.sa_mask);
15268     act.sa_flags = 0;
15269
15270     if (sigaction(SIGABRT, &act, NULL) == -1)
15271         exit_sys("sigaction");
15272
15273     printf("waiting for a signal\n");
15274
15275     abort();
15276
15277     printf("ok\n");
15278
15279     return 0;
15280 }
15281
15282 void exit_sys(const char *msg)
15283 {
15284     perror(msg);
15285
15286     exit(EXIT_FAILURE);
15287 }
15288
15289 void sigabrt_handler(int sno)
15290 {
15291     printf("SIGABRT occurred...\n");
15292 }
15293
15294 /
*-----*-----*-----*
```

15295 SIGABRT sinyali abort tarafından değil de kill gibi bir fonksiyon
tarafından gönderilirse sinyal fonksiyonu çalıştırınca sonra
program sonlanmaz. Çünkü programı raise işlemi sonrasında abort
fonksiyonu sonlandırmaktadır.

15296 Aşağıdaki programı çalıştırıp başka bir terminalden kill -ABRT <pid>
komutu ile SIGABRT sinyalini gönderiniz.

```
15297 -----*/
15298
15299 #include <stdio.h>
15300 #include <stdlib.h>
15301 #include <unistd.h>
15302 #include <signal.h>
15303
15304 void exit_sys(const char *msg);
15305 void sigabrt_handler(int sno);
15306
15307 int main(void)
```

```
15310 {
15311     struct sigaction act;
15312
15313     act.sa_handler = sigabrt_handler;
15314     sigemptyset(&act.sa_mask);
15315     act.sa_flags = 0;
15316
15317     if (sigaction(SIGABRT, &act, NULL) == -1)
15318         exit_sys("sigaction");
15319
15320     printf("waiting for a signal\n");
15321
15322     pause();
15323
15324     printf("ok\n");
15325
15326     return 0;
15327 }
15328
15329 void exit_sys(const char *msg)
15330 {
15331     perror(msg);
15332
15333     exit(EXIT_FAILURE);
15334 }
15335
15336 void sigabrt_handler(int sno)
15337 {
15338     printf("SIGABRT occurred...\n");
15339 }
15340
15341 /
*-----*-----*-----*
```

15342 C'de goto deyimi aynı fonksiyon içerisindeki atlamalarda kullanılır. Bir fonksiyondan başka bir fonksiyona atlamaya "nonlocal jump" ya da "long jump" denilmektedir. Ancak programcı herhangi bir yere long jump yapamaz. Ancak daha önce geçmiş olduğu bir noktaya long jump yapabilir. İşte setjmp fonksiyonu ile önce geri dönecek yer belirlenir. Sonra longjmp ile buraya geri dönlür. setjmp fonksiyonu ilk çağrıda 0 ile geri dönmektedir. longjmp yapıldığında akış yine setjmp'in içerisindeń çıkar.

15346 Fakat bu kez setjmp 0 ile değil longjmp'de belirtilen değerle geri dönmektedir. Nesne yönelimli programlama dillerindeki try-catch-throw mekanizmaları da aynı yöntemle derleyici tarafından sağlanmaktadır.

15349 -----*/

```
15350 #include <stdio.h>
15351 #include <stdlib.h>
15352 #include <setjmp.h>
15353
```

```
15354 void foo(void);
15355 void bar(void);
15356 jmp_buf g_jb;
15358
15359 int main(void)
15360 {
15361     int result;
15362
15363     printf("main begins...\n");
15364
15365     if ((result = setjmp(g_jb)) == 0)
15366         printf("first set...\n");
15367     else if (result == 1) {
15368         printf("return from longjmp...\n");
15369         exit(EXIT_SUCCESS);
15370     }
15371
15372     foo();
15373
15374     return 0;
15375 }
15376
15377 void foo(void)
15378 {
15379     printf("foo begins...\n");
15380     bar();
15381     printf("foo ends...\n");
15382 }
15383
15384 void bar(void)
15385 {
15386     printf("bar begins...\n");
15387     longjmp(g_jb, 1);
15388     printf("bar ends...\n");
15389 }
15390
15391 /
*-----*
-----*
15392     Eğer SIGABRT sinyali set edilen fonksiyonda longjmp yapılrsa abort      ↗
fonksiyonu prosesi sonlandırıramaz.                                     ↗
15393 -----*/*-----*/
```

15394
15395 #include <stdio.h>
15396 #include <stdlib.h>
15397 #include <setjmp.h>
15398 #include <unistd.h>
15399 #include <signal.h>
15400
15401 void exit_sys(const char *msg);
15402 void sigabrt_handler(int sno);

```
15403
15404 jmp_buf g_jb;
15405
15406 int main(void)
15407 {
15408     struct sigaction act;
15409     int result, i;
15410
15411     act.sa_handler = sigabrt_handler;
15412     sigemptyset(&act.sa_mask);
15413     act.sa_flags = 0;
15414
15415     if (sigaction(SIGABRT, &act, NULL) == -1)
15416         exit_sys("sigaction");
15417
15418     if ((result = setjmp(g_jb)) == 1)
15419         goto CONTINUE;
15420
15421     abort();
15422
15423 CONTINUE:
15424     for (i = 0; i < 10; ++i) {
15425         printf("%d ", i), fflush(stdout);
15426         sleep(1);
15427     }
15428     printf("\n");
15429
15430     return 0;
15431 }
15432
15433 void exit_sys(const char *msg)
15434 {
15435     perror(msg);
15436
15437     exit(EXIT_FAILURE);
15438 }
15439
15440 void sigabrt_handler(int sno)
15441 {
15442     printf("SIGABRT occurred...\n");
15443
15444     longjmp(g_jb, 1);
15445 }
15446
15447 /
*-----*
-----*
15448 raise POSIX fonksiyonu kendi prosesine sinyal göndermek için kullanılır. ➔
    Aslında raise(signo) ile kill(getpid(), signo) tamamen ➔
15449 eşdeğerdir. raise aynı zamanda standart bir C fonksiyonudur. raise ➔
    fonksiyonu ile sinyal gönderildiğinde fonksiyon geri dönmeden ➔
    kesinlikle ➔
15450 sinyal fonksiyonun çalışmış olacağı garanti edilmiştir. Aynı garanti ➔
```

kill fonksiyonu ile kendi prosesimize sinyal gönderirken de
verilmiştir.

15451 ----- ↵
15452 ----- */
15453 #include <stdio.h>
15454 #include <stdlib.h>
15455 #include <unistd.h>
15456 #include <signal.h>
15457
15458 void exit_sys(const char *msg);
15459 void sigusr1_handler(int sno);
15460
15461 int main(void)
15462 {
15463 struct sigaction act;
15464
15465 act.sa_handler = sigusr1_handler;
15466 sigemptyset(&act.sa_mask);
15467 act.sa_flags = 0;
15468
15469 if (sigaction(SIGUSR1, &act, NULL) == -1)
15470 exit_sys("sigaction");
15471
15472 printf("raise will be called...\n");
15473
15474 if (raise(SIGUSR1) == -1)
15475 exit_sys("raise");
15476
15477 printf("raise called...\n");
15478
15479 return 0;
15480 }
15481
15482 void exit_sys(const char *msg)
15483 {
15484 perror(msg);
15485
15486 exit(EXIT_FAILURE);
15487 }
15488
15489 void sigusr1_handler(int sno)
15490 {
15491 printf("SIGUSR1 occurred...\n");
15492 }
15493
15494 /
----- ↵
----- ↵
15495 alarm fonksiyonu belli bir saniye sonra kendi prosesine SIGALRM
15496 sinyalinin gönderilmesine yol açar. Eğer daha önce bir
15497 alarm set edilmişse o silinir yeni set edilmiş olur. Fonksiyonun
parametresi 0 olarak geçilirse eski alarm iptal edilmektedir.

```
15497     Fonksiyon bir önce set edilmiş alarm'daki kalan saniye sayısını      ↵
15498     vermektedir.
15499     -----
15500     #include <stdio.h>
15501     #include <stdlib.h>
15502     #include <unistd.h>
15503     #include <signal.h>
15504
15505     void exit_sys(const char *msg);
15506     void sigalrm_handler(int sno);
15507
15508     int main(void)
15509     {
15510         struct sigaction act;
15511
15512         act.sa_handler = sigalrm_handler;
15513         sigemptyset(&act.sa_mask);
15514         act.sa_flags = 0;
15515
15516         if (sigaction(SIGALRM, &act, NULL) == -1)
15517             exit_sys("sigaction");
15518
15519         alarm(10);
15520
15521         pause();
15522
15523         return 0;
15524     }
15525
15526     void exit_sys(const char *msg)
15527     {
15528         perror(msg);
15529
15530         exit(EXIT_FAILURE);
15531     }
15532
15533     void sigalrm_handler(int sno)
15534     {
15535         printf("SIGALRM occurred...\n");
15536     }
15537
15538 /
*-----*
-----*
15539     Programcılar genellikle periyodik timer oluşturmak için alarm sinyal      ↵
15540     fonksiyonunda yeniden alarm çağrıması yaparlar.
15541     -----
15542     #include <stdio.h>
15543     #include <stdlib.h>
```

```
15544 #include <unistd.h>
15545 #include <signal.h>
15546
15547 void exit_sys(const char *msg);
15548 void sigalarm_handler(int sno);
15549
15550 int main(void)
15551 {
15552     struct sigaction act;
15553
15554     act.sa_handler = sigalarm_handler;
15555     sigemptyset(&act.sa_mask);
15556     act.sa_flags = 0;
15557
15558     if (sigaction(SIGALRM, &act, NULL) == -1)
15559         exit_sys("sigaction");
15560
15561     alarm(1);
15562
15563     for(;;)
15564         pause();
15565
15566     return 0;
15567 }
15568
15569 void exit_sys(const char *msg)
15570 {
15571     perror(msg);
15572
15573     exit(EXIT_FAILURE);
15574 }
15575
15576 void sigalarm_handler(int sno)
15577 {
15578     printf("SIGALRM occurred...\n");
15579
15580     alarm(1);
15581 }
15582
15583 /
*-----*-----*-----*
```

15584 Bir sinyal için sinyal fonksiyonu (signal handler) set ettiğimizde o →
sinyal fonksiyonun içerisinde çağrıracagımız fonksiyonlara →
dikkat etmeliyiz. Sinyal fonksiyonlarının içerisinde ancak "sinyal →
güvenli (async-signal safe)" fonksiyonları çağrırmalıyız. →
POSIX standartlarında sinyal güvenli fonksiyonların neler olduğu →
listelenmiştir. Tabii bazı koşullar altında dikkat olmak koşuluyla →
sinyal fonksiyonlarının içerisinde sinyal güvenli olmayan fonksiyonları →
da çağrırlabiliriz.

15585 →
15586 →
15587 →
15588 →
15589 →

15590 sinyal güvenli
15590 değilse sorun oluşabilir. Thread güvenlilikle sinyal güvenlilik aynı →
15590 anlamda değildir. Bir fonksiyon şu kategorilerden
15591 birine girebilir:
15592
15593 1) Hem thread güvenli hem de sinyal güvenli
15594 2) Thread güvenli fakat sinyal güvenli değil
15595 3) Sinyal güvenli ama thread güvenli değil
15596 4) Thread güvenli de değil, sinyal güvenli de değil
15597
15598 Örneğin aşağıdaki fonksiyon thread güvenli olduğu halde sinyal güvenli →
15598 değildir:
15599
15600 `thread_local int g_i;`
15601
15602 `void foo(void)`
15603 {
15604 `g_i = 10;`
15605 ...
15606 `g_i = 20;`
15607 ...
15608 `g_i = 30;`
15609 }
15610
15611 Bazı POSIX fonksiyonlarının thread güvenli olduğu olduğu halde sinyal →
15611 güvenli olmadığına dikkat ediniz: Örneğin `malloc`, `calloc`,
15612 `stdio` fonksiyonları gibi...
15613 -----*/
15614 /
15615 /*-----

15616 Bir sinyal fonksiyonu içerisinde `errno`'yu değiştiren bir POSIX →
15616 fonksiyonu kullanılırsa sinyal kestiği fonksiyonun set ettiği `errno` →
15617 değeri bozulabilir. Bunun için eğer böyle bir fonksiyon çağrılacaksa →
15617 sinyal fonksiyonlarının başında `errno` değerini
15618 saklayıp sonunda yeniden set etmek iyi bir tekniktir. Örneğin:
15619
15620 `void signal_handler(int signo)`
15621 {
15622 `int temp = errno;`
15623 ...
15624 `errno = temp;`
15625 }
15626
15627 POSIX fonksiyonları başarı durumunda `errno` değerini 0 olarak set →
15627 etmezler. POSIX standartlarına göre `errno` hiçbir fonksiyon tarafından →
15627 0'a
15628 çekilmemektedir. POSIX standartları başarı durumunda `errno` değerinin →
15628 fonksiyon tarafından değiştirilebileceğini açıkça belirtmiştir.
15629 Ancak başarı durumunda `errno` değerini başka değerlere set eden POSIX →
15629 fonksiyonları vardır. Bu nedenle programcının `errno` değişkenini

```
15630     saklayıp geri yerleştirmesi uygun olur.  
15631     -----*/  
15632  
15633 /  
*-----  
15634 Alt proses sonlandığında işletim sistemi o prosesin üst prosesine      ↵  
    SIGCHLD sinyali göndermektedir. Bu sinyalin ismi AT&T  
15635 UNIX sistemlerinde eskiden SIGCLD biçimindeydi. Ancak POSIX BSD ismi      ↵  
    olan SIGCHLD ismini tercih etti. O zamanlar SIGCLD  
15636 sinyaliin semantiği ile SIGCHLD sinyalinin semantiği arasında da küçük      ↵  
    farklılıklar vardı. POSIX standartları bu farklılıklarını  
15637 "implementation dependent" hale getirmiştir.  
15638 -----*/  
15639  
15640 #include <stdio.h>  
15641 #include <stdlib.h>  
15642 #include <unistd.h>  
15643 #include <sys/wait.h>  
15644 #include <signal.h>  
15645  
15646 void exit_sys(const char *msg);  
15647 void sigchld_handler(int sno);  
15648  
15649 int main(void)  
15650 {  
15651     struct sigaction act;  
15652     pid_t pid;  
15653  
15654     act.sa_handler = sigchld_handler;  
15655     sigemptyset(&act.sa_mask);  
15656     act.sa_flags = 0;  
15657  
15658     if (sigaction(SIGCHLD, &act, NULL) == -1)  
15659         exit_sys("sigaction");  
15660  
15661     if ((pid = fork()) == -1)  
15662         exit_sys("fork");  
15663  
15664     if (pid == 0) {  
15665         sleep(5);  
15666         _exit(EXIT_FAILURE);  
15667     }  
15668  
15669     pause();  
15670  
15671     if (waitpid(pid, NULL, 0) == -1)  
15672         exit_sys("waitpid");  
15673  
15674     return 0;  
15675 }
```

```
15676
15677 void exit_sys(const char *msg)
15678 {
15679     perror(msg);
15680
15681     exit(EXIT_FAILURE);
15682 }
15683
15684 void sigchld_handler(int sno)
15685 {
15686     printf("SIGCHLD occurred...\n");
15687 }
15688
15689 /
*-----*
-----*
15690     SIGCHLD sinyali genellikle zombie proses oluşumunu otomatik engellemek
15691     için kullanılmaktadır. Yani tipik olarak programcı
15692     bu sinyale ilişkin sinyal fonksiyonu içerisinde wait fonksiyonlarını
15693     uygular.
15694 -----*/
15695
15696 #include <stdio.h>
15697 #include <stdlib.h>
15698 #include <unistd.h>
15699 #include <sys/wait.h>
15700 #include <signal.h>
15701
15702 void exit_sys(const char *msg);
15703 void sigchld_handler(int sno);
15704
15705 int main(void)
15706 {
15707     struct sigaction act;
15708     pid_t pid;
15709
15710     act.sa_handler = sigchld_handler;
15711     sigemptyset(&act.sa_mask);
15712     act.sa_flags = 0;
15713
15714     if (sigaction(SIGCHLD, &act, NULL) == -1)
15715         exit_sys("sigaction");
15716
15717     if ((pid = fork()) == -1)
15718         exit_sys("fork");
15719
15720     if (pid == 0) {
15721         sleep(5);
15722         _exit(EXIT_FAILURE);
15723     }
15724
15725 pause();
```

```
15724     return 0;
15725 }
15726 }
15727
15728 void exit_sys(const char *msg)
15729 {
15730     perror(msg);
15731
15732     exit(EXIT_FAILURE);
15733 }
15734
15735 void sigchld_handler(int sno)
15736 {
15737     if (waitpid(-1, NULL, 0) == -1)
15738         exit_sys("waitpid");
15739 }
15740
15741 /
*-----*
-----*
15742     Üst prosesin birden fazla alt proses yarattığı durumda bu alt prosesler
15743         birbirlerine yakın zamanlarda sonlanırsa sinyaller
15744         kuyruklanmadığından her alt proses için bir kez SIGCHLD sinyali
15745         oluşmayabilir. Bu durum da otomatik zombie engellemeye
15746         sekteye uğratabilir. Örneğin aşağıda 10 tane alt proses yaratılmıştır.
15747         Ancak sinyal fonksiyonu 10 kere çağrılmamaktadır.
15748 -----*/
15749
15750 #include <stdio.h>
15751 #include <stdlib.h>
15752 #include <unistd.h>
15753 #include <sys/wait.h>
15754 #include <signal.h>
15755
15756 void exit_sys(const char *msg);
15757 void sigchld_handler(int sno);
15758
15759 int main(void)
15760 {
15761     struct sigaction act;
15762     pid_t pids[10];
15763     int i;
15764
15765     act.sa_handler = sigchld_handler;
15766     sigemptyset(&act.sa_mask);
15767     act.sa_flags = 0;
15768
15769     if (sigaction(SIGCHLD, &act, NULL) == -1)
15770         exit_sys("sigaction");
15771
15772     for (i = 0; i < 10; ++i) {
```

```
15771     if ((pids[i] = fork()) == -1)
15772         exit_sys("fork");
15773     if (pids[i] == 0) {
15774         sleep(5);
15775         _exit(EXIT_FAILURE);
15776     }
15777 }
15778
15779 for (;;)
1580     pause();
1581
1582 return 0;
1583 }
1584
1585 void exit_sys(const char *msg)
1586 {
1587     perror(msg);
1588
1589     exit(EXIT_FAILURE);
1590 }
1591
1592 void sigchld_handler(int sno)
1593 {
1594     int temp = errno;
1595
1596     printf("SIGCHLD occurred...\n");
1597
1598     errno = temp;
1599 }
15800
15801 /
*-----*
-----*
-----*
-----*
```

15802 Yukarıdaki problem aşağıdaki gibi SIGCHLD sinyal fonksiyonunda bir döngü içerisinde wait fonksiyonları uygulanarak giderilebilir. Ekrana yazdırılan yazıların sayısına bakarak 10 tane olduğunu görünüz.

15803 -----*

```
15804 -----*/
15805
15806 #include <stdio.h>
15807 #include <stdlib.h>
15808 #include <unistd.h>
15809 #include <sys/wait.h>
15810 #include <errno.h>
15811 #include <signal.h>
15812
15813 void exit_sys(const char *msg);
15814 void sigchld_handler(int sno);
15815
15816 int main(void)
15817 {
15818     struct sigaction act;
```

```
15819     pid_t pids[10];
15820     int i;
15821
15822     act.sa_handler = sigchld_handler;
15823     sigemptyset(&act.sa_mask);
15824     act.sa_flags = 0;
15825
15826     if (sigaction(SIGCHLD, &act, NULL) == -1)
15827         exit_sys("sigaction");
15828
15829
15830     for (i = 0; i < 10; ++i) {
15831         if ((pids[i] = fork()) == -1)
15832             exit_sys("fork");
15833         if (pids[i] == 0) {
15834             sleep(5);
15835             _exit(EXIT_FAILURE);
15836         }
15837     }
15838
15839     for (;;)
15840         pause();
15841
15842     return 0;
15843 }
15844
15845 void exit_sys(const char *msg)
15846 {
15847     perror(msg);
15848
15849     exit(EXIT_FAILURE);
15850 }
15851
15852 void sigchld_handler(int sno)
15853 {
15854     int temp = errno;
15855
15856     while (waitpid(-1, NULL, WNOHANG) > 0)
15857         printf("prevented child to be zombie!..\n");
15858
15859     errno = temp;
15860 }
15861
15862 /
*-----*
-----*
15863 POSIX standartlarına göre otomatik zombie engellenmenin iki yolu daha    ↗
15864 vardır:
15865 1) SIGCHLD sinyali ignore'a çekilir (SIG_IGN)
15866 2) SIGCHLD sinyali için sigaction fonksiyonunda set yapılırken      ↗
15867     SA_NOCLDWAIT bayrağı da kullanılır.
```

```
15868     POSIX standartları SIGCHLD sinyali için SA_NOCLDWAIT bayrağı      ↵
15869         kullanıldığında sinyal fonksiyonun çağrılıp çağrılamayacağını    ↵
15870         işletim sisteminin isteğine bırakmıştır. Linux sistemleri sinyal    ↵
15871         fonksiyonunu çağrırmaktadır.
15872
15873     Her ne kadar SIGCHLD sinyalinin default eylemi (default action) "ignore" ↵
15874         olsa da burada "ignore'a çekmek" açıkça
15875     signal fonksiyonuyla ya da sigaction fonksiyonuyla SIG_IGN kullanmak    ↵
15876         anlamındadır.
15877
15878     Bu biçimde otomatik zombie engellendiğinde artık wait fonksiyonları      ↵
15879         kullanılmamalıdır. Eğer kullanılırsa bu fonksiyonlar -1'e
15880         geri dönerler ve errno da ECHILD olarak set edilir. Dolayısıyla    ↵
15881         aşağıdaki örnekte waitpid fonksiyonu otomatik zombie
15882         engellendiği için başarısızlıkla geri dönecektir.
15883
15884 -----*/
15885
15886 #include <stdio.h>
15887 #include <stdlib.h>
15888 #include <unistd.h>
15889 #include <sys/wait.h>
15890 #include <errno.h>
15891 #include <signal.h>
15892
15893 void exit_sys(const char *msg);
15894
15895 int main(void)
15896 {
15897     struct sigaction act;
15898     pid_t pids[10];
15899     int i;
15900
15901     act.sa_handler = SIG_IGN;
15902     sigemptyset(&act.sa_mask);
15903     act.sa_flags = 0;
15904
15905     if (sigaction(SIGCHLD, &act, NULL) == -1)
15906         exit_sys("sigaction");
15907
15908     /* signal(SIGCHLD, SIG_IGN); */
15909
15910     for (i = 0; i < 10; ++i) {
15911         if ((pids[i] = fork()) == -1)
15912             exit_sys("fork");
15913         if (pids[i] == 0) {
15914             _exit(EXIT_FAILURE);
15915         }
15916     }
15917
15918     printf("press ENTER to continue...\n");
15919     getchar();
15920 }
```

```
15914     if (waitpid(-1, NULL, 0) == -1)      /* waitpid will fail!.. */
15915         exit_sys("waitpid");
15916
15917     return 0;
15918 }
15919
15920 void exit_sys(const char *msg)
15921 {
15922     perror(msg);
15923
15924     exit(EXIT_FAILURE);
15925 }
15926
15927 /
*-----*
-----  
15928     Eğer sigaction fonksiyonunda SIGCHLD sinyali için hem fonksiyon set     ↗
15929     edilir hem de SA_NOCLDWAIT bayrağı kullanılırsa     ↗
15930     bu durumda otomatik zombie yine engellenmekle birlikte sinyal     ↗
15931     fonksiyonun çağrılmıştır (implementation dependent). Linux'ta çağrıma     ↗
15932     bırakılmıştır (implementation dependent). Linux'ta çağrıma     ↗
15933     yapılmaktadır. SA_NOCLDWAIT bayrağı yalnızca SIGCHLD sinyali     ↗
15934     için anlamlıdır.
15935 -----*/  
15936
15937 #include <stdio.h>
15938 #include <stdlib.h>
15939 #include <unistd.h>
15940 #include <sys/wait.h>
15941 #include <errno.h>
15942 #include <signal.h>
15943
15944 void exit_sys(const char *msg);
15945 void sigchld_handler(int sno);
15946
15947 int main(void)
15948 {
15949     struct sigaction act;
15950     pid_t pids[10];
15951     int i;
15952
15953     act.sa_handler = sigchld_handler;
15954     sigemptyset(&act.sa_mask);
15955     act.sa_flags = SA_NOCLDWAIT;
15956
15957     if (sigaction(SIGCHLD, &act, NULL) == -1)
15958         exit_sys("sigaction");
15959
15960     for (i = 0; i < 10; ++i) {
15961         if ((pids[i] = fork()) == -1)
15962             exit_sys("fork");
15963         if (pids[i] == 0) {
```

```
15961         sleep(1);
15962         _exit(EXIT_FAILURE);
15963     }
15964 }
15965
15966 for(;;)
15967     pause();
15968
15969 return 0;
15970 }
15971
15972 void exit_sys(const char *msg)
15973 {
15974     perror(msg);
15975
15976     exit(EXIT_FAILURE);
15977 }
15978
15979 void sigchld_handler(int sno)
15980 {
15981     int temp = errno;
15982
15983     printf("SIGCHLD occurred...\n"); /* wait should'nt be used! */
15984
15985     errno = temp;
15986 }
15987
15988 /
*-----*
```

15989 Bazı sistem fonksiyonları (read, write, sleep gibi) bazı kaynaklarla →
15990 çalışırken uzun süre bekleme yapabilmektedir. Çünkü →
15990 bu sistem fonksiyonları istenen şey gerçekleşene kadar çizelgeden →
15991 çakrtılıp blokeye yol açmaktadır. Peki bir proses →
15991 bu sistem fonksiyonlarında blokede beklerken bir sinyal oluştuğunda ne →
15992 olur? İşte işletim sistemi bu durumda sinyali hemen →
15992 teslim edebilmek için fonksiyonun yol açtığı blokenin bitmesini →
15992 beklemez. Sistem fonksiyonunu başarısızlıkla sonlandırır ve →
15993 hemen prosesi çizelgeye sokarak ona sinyali teslim eder. Bu tür sistem →
15993 fonksiyonları (yani bunları çağırılan POSIX fonksiyonları) →
15994 -1 değerine geri dönüp errno değerini EINTR olarak set etmektedir. Bu →
15994 durumda biz böylesi bir durumla karşılaşlığımızda →
15995 fonksiyonun yaptığı işte başarız olduğunu düşünmemeliyiz. Fonksiyonun →
15995 sinyal geldiğinden dolayı başarısız olduğunu ve →
15996 yeniden çağrırlırsa başarılı olabileceğini düşünmemeliyiz. O halde bu tür →
15996 sistem fonksiyonlarını çağırırken bizim eğer →
15997 başarısızlığın nedeni sinyal ise (EINTR) bu fonksiyonu yeniden →
15997 çağrılmamız gereklidir. Bu da maalesef programlama açısından sıkıntılıdır. →

15998
15999 Aşağıdaki programda alarm fonksiyonu dolayısıyla SIGALRM sinyali →
15999 gönderildiğinde scanf fonksiyonu (yani aslında onun çağrıdığı read →
fonksiyonu)

```
16000    başarılı olacaktır ve errno EINTR ile set edilecektir.  
16001  -----*/  
16002  
16003 #include <stdio.h>  
16004 #include <stdlib.h>  
16005 #include <unistd.h>  
16006 #include <errno.h>  
16007 #include <signal.h>  
16008  
16009 void exit_sys(const char *msg);  
16010 void sigalrm_handler(int sno);  
16011  
16012 int main(void)  
16013 {  
16014     struct sigaction act;  
16015     int val;  
16016  
16017     act.sa_handler = sigalrm_handler;  
16018     sigemptyset(&act.sa_mask);  
16019     act.sa_flags = 0;  
16020  
16021     if (sigaction(SIGALRM, &act, NULL) == -1)  
16022         exit_sys("sigaction");  
16023  
16024     alarm(1);  
16025  
16026     printf("Bir sayı giriniz:");  
16027     fflush(stdout);  
16028     scanf("%d", &val);  
16029     printf("%d\n", val * val);  
16030  
16031     return 0;  
16032 }  
16033  
16034 void exit_sys(const char *msg)  
16035 {  
16036     perror(msg);  
16037  
16038     exit(EXIT_FAILURE);  
16039 }  
16040  
16041 void sigalrm_handler(int sno)  
16042 {  
16043     /* do something */  
16044 }  
16045  
16046 /  
*-----  
16047 İşte bu tür durumlarda yöntemlerden birisi eğer fonksiyon sinyal  
yüzünden başarısız olmuşsa bir döngü içerisinde onu  
yeniden çağrılmaktır. Tabii eğer programcı kendi programında bir sinyal  
16048
```

```
    işliyorsa bu yola gitmelidir. Yoksa zaten işlenmeyen
16049    sinyaller için proses sonlandırılır. Bir sinyalin default eylemi      ↵
        (default action) eğer "ignore" ise bu durumda işletim
16050    sistemi yavaş sistem fonksiyonlarını başarısızlıkla sonlandırır.
16051    -----
16052    -----
16053 #include <stdio.h>
16054 #include <stdlib.h>
16055 #include <unistd.h>
16056 #include <errno.h>
16057 #include <signal.h>
16058
16059 void exit_sys(const char *msg);
16060 void sigalrm_handler(int sno);
16061
16062 int main(void)
16063 {
16064     struct sigaction act;
16065     int val;
16066
16067     act.sa_handler = sigalrm_handler;
16068     sigemptyset(&act.sa_mask);
16069     act.sa_flags = 0;
16070
16071     if (sigaction(SIGALRM, &act, NULL) == -1)
16072         exit_sys("sigaction");
16073
16074     alarm(1);
16075
16076     printf("Bir sayı giriniz:");
16077     fflush(stdout);
16078
16079     while (scanf("%d", &val) == -1 && errno == EINTR)
16080         ;
16081
16082     printf("%d\n", val * val);
16083
16084     return 0;
16085 }
16086
16087 void exit_sys(const char *msg)
16088 {
16089     perror(msg);
16090
16091     exit(EXIT_FAILURE);
16092 }
16093
16094 void sigalrm_handler(int sno)
16095 {
16096     /* do something */
16097 }
16098
```

```
16099  /
16100      *-----*
16100      Bu çok sıkıcı bir işelmdir. Bazı programcılar bunun için aşağıdaki gibi  -->
16100      bir makro kullanabilmektedir:
16101
16102      #define EINTR_LOOP(statement)    while ((statement) == -1 && errno == -1)
16103      -----*/
16104
16105      #include <stdio.h>
16106      #include <stdlib.h>
16107      #include <unistd.h>
16108      #include <errno.h>
16109      #include <signal.h>
16110
16111      #define EINTR_LOOP(statement)          while ((statement) == -1 && errno ==   -->
16111          EINTR)
16112
16113      void exit_sys(const char *msg);
16114      void sigalrm_handler(int sno);
16115
16116      int main(void)
16117  {
16118      struct sigaction act;
16119      int val;
16120
16121      act.sa_handler = sigalrm_handler;
16122      sigemptyset(&act.sa_mask);
16123      act.sa_flags = 0;
16124
16125      if (sigaction(SIGALRM, &act, NULL) == -1)
16126          exit_sys("sigaction");
16127
16128      alarm(1);
16129
16130      printf("Bir sayı giriniz:");
16131      fflush(stdout);
16132
16133      EINTR_LOOP(scanf("%d", &val));
16134      printf("%d\n", val * val);
16135
16136      return 0;
16137  }
16138
16139  void exit_sys(const char *msg)
16140  {
16141      perror(msg);
16142
16143      exit(EXIT_FAILURE);
16144  }
16145
16146  void sigalrm_handler(int sno)
```

```
16147 {
16148     /* do something */
16149 }
16150
16151 /
*-----*
-----+
16152     Bazı yavaş sistem fonksiyonlarının çekirdek tarafından otomatik restart →
16153         edilmesi için sigaction fonksiyonunda sa_flags →
16154             parametresi SA_RESTART girilmelidir. Bu durumda artık programcının →
16155                 sinyal nedeniyle sonlanmalar için kendisinin bir döngü oluşturmamasına →
16156                     gerek kalmaz.
16157 -----*/
16158 #include <stdio.h>
16159 #include <stdlib.h>
16160 #include <unistd.h>
16161 #include <errno.h>
16162 #include <signal.h>
16163 void exit_sys(const char *msg);
16164 void sigalrm_handler(int sno);
16165
16166 int main(void)
16167 {
16168     struct sigaction act;
16169     int val;
16170
16171     act.sa_handler = sigalrm_handler;
16172     sigemptyset(&act.sa_mask);
16173     act.sa_flags = SA_RESTART;
16174
16175     if (sigaction(SIGALRM, &act, NULL) == -1)
16176         exit_sys("sigaction");
16177
16178     alarm(1);
16179
16180     printf("Bir sayı giriniz:");
16181     fflush(stdout);
16182
16183     if (scanf("%d", &val) == -1)
16184         exit_sys("scanf");
16185
16186     printf("%d\n", val * val);
16187
16188     return 0;
16189 }
16190
16191 void exit_sys(const char *msg)
16192 {
16193     perror(msg);
16194 }
```

```
16195     exit(EXIT_FAILURE);
16196 }
16197
16198 void sigalrm_handler(int sno)
16199 {
16200     /* do something */
16201 }
16202
16203 /
*-----*
-----*
16204     Anımsanacağı gibi eski signal fonksiyonunda sinyal oluştuğunda yavaş
16205         fonksiyonlarında otomatik restart işlemi işletim sistemine
16206         bağlı olarak değişebiliyordu (implementation dependent). Linux
16207             sistemlerinde otomatik restart yapıldığını anımsayınız.
16208 -----*/
16209
16210 #include <stdio.h>
16211 #include <stdlib.h>
16212 #include <unistd.h>
16213 #include <errno.h>
16214 #include <signal.h>
16215
16216 void exit_sys(const char *msg);
16217 void sigalrm_handler(int sno);
16218
16219 int main(void)
16220 {
16221     int val;
16222
16223     if (signal(SIGALRM, sigalrm_handler) == SIG_ERR)
16224         exit_sys("signal");
16225
16226     alarm(1);
16227
16228     printf("Bir sayı giriniz:");
16229     fflush(stdout);
16230
16231     if (scanf("%d", &val) == -1)
16232         exit_sys("scanf");
16233
16234     printf("%d\n", val * val);
16235
16236     return 0;
16237 }
16238
16239 void exit_sys(const char *msg)
16240 {
16241     perror(msg);
16242     exit(EXIT_FAILURE);
16243 }
```

```
16243
16244 void sigalrm_handler(int sno)
16245 {
16246     /* do something */
16247 }
16248
16249 /
*-----*
-----*
16250     Biz sigaction fonksiyonunda flags parametresinde SA_RESTART girmiş olsak
16251         bile bazı yavaş sistem fonksiyonları doğası gereği
16252         restart edilememektedir. SA_RESTART bayrağı belirtildiğinde genel olarak
16253             POSIX standartlarında hangi fonksiyonların hangi
16254             koşullar altında restart edileceği belirtmemiştir. Örneğin sleep,
16255                 nanosleep gibi bekleme fonksiyonları restart işlemi
16256                 yapmazlar. Zaten bu işlemin bu fonksiyonlarda yapılması anlamlı
16257                     değildir.
16258         read (write vs.) fonksiyonu talep edildiği kadar bilgiyi henüz
16259             okuyamamışken sinyal oluşabilir. Bu durumda SA_RESTART bayrağı
16260             set edilmemiş olsa bile fonksiyon başarısız olmaz. Okuduğu kadar bilgiye
16261                 geri döner.
16262 -----
16263 -----
16264 #include <stdio.h>
16265 #include <stdlib.h>
16266 #include <unistd.h>
16267 #include <errno.h>
16268 #include <signal.h>
16269
16270 void exit_sys(const char *msg);
16271 void sigusr1_handler(int sno);
16272
16273 int main(void)
16274 {
16275     struct sigaction act;
16276
16277     act.sa_handler = sigusr1_handler;
16278     sigemptyset(&act.sa_mask);
16279     act.sa_flags = SA_RESTART;
16280
16281     if (sigaction(SIGUSR1, &act, NULL) == -1)
16282         exit_sys("sigaction");
16283
16284     if (sleep(60) != 0)
16285         if (errno == EINTR)
16286             printf("sleep finished due to signal...\n");
16287         else
16288             exit_sys("sleep");
16289     else
16290         printf("sleep finished normally...\n");
16291
16292     return 0;
```

```
16287 }
16288
16289 void exit_sys(const char *msg)
16290 {
16291     perror(msg);
16292
16293     exit(EXIT_FAILURE);
16294 }
16295
16296 void sigusr1_handler(int sno)
16297 {
16298     /* do something */
16299 }
16300
16301 /
*-----*
-----*
16302     open, read write gibi fonksiyonları bloke durumunda sinyal geldiğinde →
16303         eğer proses sonlanmazsa başarısız olmaktadır. →
16304         Aşağıdaki örnekte "myfifo" isimli bir boru dosyası açılmak istenmiştir. →
16305             Başka bir proses boruyu yazma modunda açana kadar →
16306             bloke olacakaktır. Bu duurmda başka bir terminalden SIGUSR1 sinyalini →
16307                 prosese göndererek open fonksiyonunun EINTR errno →
16308                 değeriyle başarısız olduğunu görünüz.
16309 -----*/
16310
16311 #include <stdio.h>
16312 #include <stdlib.h>
16313 #include <fcntl.h>
16314 #include <unistd.h>
16315 #include <errno.h>
16316 #include <signal.h>
16317
16318 void exit_sys(const char *msg);
16319 void sigusr1_handler(int sno);
16320
16321 int main(void)
16322 {
16323     struct sigaction act;
16324     int fd;
16325     ssize_t size;
16326     char buf[10];
16327
16328     act.sa_handler = sigusr1_handler;
16329     sigemptyset(&act.sa_mask);
16330     act.sa_flags = 0;
16331
16332     if (sigaction(SIGUSR1, &act, NULL) == -1)
16333         exit_sys("sigaction");
16334
16335     if ((fd = open("myfifo", O_RDONLY)) == -1)
16336         exit_sys("open");
```

```
16334
16335     printf("pipe opened successfully...\n");
16336
16337     if ((size = read(fd, buf, 10)) == -1)
16338         exit_sys("read");
16339
16340     printf("read read successfully\n");
16341
16342     close(fd);
16343
16344     return 0;
16345 }
16346
16347 void exit_sys(const char *msg)
16348 {
16349     perror(msg);
16350
16351     exit(EXIT_FAILURE);
16352 }
16353
16354 void sigusr1_handler(int sno)
16355 {
16356     /* do something */
16357 }
16358
16359 /
*-----*
```

16360 POSIX'e 90 yılların ortalarında "realtime extensions" başlığı altında "gerçek zamanlı (realtime)" sinyal kavramı da eklendi. ↗

16361 Gerçek zamanlı sinyalların numaraları [SIGRTMIN, SIGRTMAX] arasındadır. ↗

Bunlara ayrı isimler verilmemiştir. Gerçek zamanlı sinyallerin normal sinyallerden (ilk 32 sinyal numarası normal sinyaller için ayrılmıştır) farkları şunlardır:

16363 1) Gerçek zamanlı sinyaller kuyruklanmaktadır. Yani birden fazla gerçek zamanlı aynı sinyal oluştukunda kaç tane oluşmuş olduğu tutulur ve o sayıda prosese teslim edilir.

16364 2) Gerçek zamanlı sinyallerde bir bilgi de sinyale ilişirilebilmektedir. Bu bilgi ya int bir değer ya da bir gösterici olur.

16365 Gösterici kullanıldığında bu göstericinin gösterdiği yerin hedef proseste anlamlı olması gerekmektedir.

16366 3) Gerçek zamanlı sinyallerde bir öncelik ilişkisi (priority) vardır. Birden fazla farklı numaralı gerçek zamanlı sinyal bloke edildiği durumda bloke açılıncı bunların oluşma sırası küçük numaradan büyük numaraya göredir.

16367 Gerçek zamanlı sinyaller kill fonksiyonu ile değil sigqueue isimli fonksiyonla gönderilmemektedir. Eğer bu sinyaller kill ile gönderilirse kuyruklama yapılmış yapılmayacağı sistemden sisteme değişebilmektedir.

16368

16369

16370

16371

16372

16373

16374 Gerçek zamanlı sinyaller alınırken yine sigaction fonksiyonu kullanılmak →
zorundadır. Bu fonksiyonda sinyal fonksiyonu için
16375 artık sigaction yapısının sa_handler elemanına değil sa_sigaction →
elemanına atama yapılmalıdır. Tabii fonksiyonununu anlaması
16376 için flags parametresine de ayrıca SA_SIGINFO eklenmelidir. (Yani başka →
bir deyişle fonkisyon sa_flags parametresinde SA_SIGINFO
16377 değerini gördüğünde artık sinyal fonksiyonu için yapınının sa_sigaction →
elemanı bakar.)

16378

16379

16380 Aşağıdaki örnekte proc1 programı n tane gerçek zamanlı sinyal gönderir. →
proc2 ise bunları almaktadır. Programları çalıştırarak
16381 kuyruklamanın yapıldığına dikkat ediniz. sigqueue fonksiyonunda →
iliştirilen sinyal bilgisi siginfo_t yapısının si_value
16382 elemanından alınmaktadır.

16383

16384 sigqueue fonksiyonuyla set edilen inyal fonksiyonundaki siginfo_t →
yapısının diğer elemanlarını ilgili dokimanlardan inceleyiniz.
16385 (Örneğin burada sinyali gönderen proses id'si, gerçek kullanıcı id'si, →
sinyalin neden gönderildiği gibi bilgiler vardır.)

16386

16387 -----*/

16388

16389 /* proc1.c */

16390

16391 #include <stdio.h>

16392 #include <stdlib.h>

16393 #include <signal.h>

16394

16395 void exit_sys(const char *msg);

16396

16397 /* ./prog1 <realtime signal no> <process id> <count> */

16398

16399 int main(int argc, char *argv[])

16400 {

16401 int signo;

16402 pid_t pid;

16403 int count;

16404 int i;

16405 union sigval val;

16406

16407 if (argc != 4) {

16408 fprintf(stderr, "wrong number of arguments!..\n");

16409 exit(EXIT_FAILURE);

16410 }

16411

16412 signo = (int)strtol(argv[1], NULL, 10);

16413 pid = (pid_t)strtol(argv[2], NULL, 10);

16414 count = (int)strtol(argv[3], NULL, 10);

16415

16416 for (i = 0; i < count; ++i) {

16417 val.sival_int = i;

```
16418     if (sigqueue(pid, SIGRTMIN + signo, val) == -1)
16419         exit_sys("sigqueue");
16420     }
16421
16422     return 0;
16423 }
16424
16425 void exit_sys(const char *msg)
16426 {
16427     perror(msg);
16428
16429     exit(EXIT_FAILURE);
16430 }
16431
16432 void sigusr1_handler(int sno)
16433 {
16434     printf("sigusr1 occurred...\n");
16435 }
16436
16437 /* proc2.c */
16438
16439 #include <stdio.h>
16440 #include <stdlib.h>
16441 #include <unistd.h>
16442 #include <signal.h>
16443
16444 void exit_sys(const char *msg);
16445 void sigrt_handler(int signo, siginfo_t *info, void *context);
16446
16447 /* ./prog2 <realtime signal no> */
16448
16449 int main(int argc, char *argv[])
16450 {
16451     struct sigaction act;
16452     int signo;
16453
16454     if (argc != 2) {
16455         fprintf(stderr, "wrong number of arguments!..\n");
16456         exit(EXIT_FAILURE);
16457     }
16458
16459     signo = (int)strtol(argv[1], NULL, 10);
16460
16461     act.sa_sigaction = sigrt_handler;
16462     sigemptyset(&act.sa_mask);
16463     act.sa_flags = SA_SIGINFO;
16464
16465     if (sigaction(SIGRTMIN + signo, &act, NULL) == -1)
16466         exit_sys("sigaction");
16467
16468     for(;;)
16469         pause();
16470 }
```

```
16471     return 0;
16472 }
16473
16474 void exit_sys(const char *msg)
16475 {
16476     perror(msg);
16477
16478     exit(EXIT_FAILURE);
16479 }
16480
16481 void sigrt_handler(int signo, siginfo_t *info, void *context)
16482 {
16483     printf("SIGRTMIN + 0 occurred with %d code\n", info-
16484         >si_value.sival_int);
16485
16486 /
*-----*
-----*
16487 Sinyal konusu UNIX türevi sistemlerde 70'lerin başlarından beri vardı. ➔
16488 Oysa thread'ler 90'lارın ortalarında bu sistemlere ➔
16489 sokulmuştur. Yani aslında sinyal konusu thread'siz bir ortamda ortaya ➔
16490 çıkmış ve kullanılmaya başlanmıştır. Ancak thread'ler ➔
16491 eklendikten sonra bazı belirlemelerin de yapılması gerekmistiir. ➔
16492 Sinyaller default olarak kill ve sigqueue fonksiyonları tarafından ➔
16493 prosese gönderilirler. Proses bu sinyalleri herhangi bir thread akışı ➔
16494 tarafından işletebilir. Prosese gönderilen sinyalin hangi ➔
16495 thread tarafından işletileceği POSIX standartlarında belirsiz ➔
16496 bırakılmıştır.
16497
16498 Aşağıdaki programda 3 thread yaratılmıştır. Ana thread'le birlikte ➔
16499 toplam 4 thread vardır. Siz de başka bir terminalden ➔
16500 prosese kill komutıyla (kill fonksiyonuyla) SIGUSR1 sinyalini göndererek ➔
16501 bu sinyalin hangi thread tarafından ele alındığını bakınız.
16502
16503 -----*/
16504 #include <stdio.h>
16505 #include <stdlib.h>
16506 #include <string.h>
16507 #include <unistd.h>
16508 #include <pthread.h>
16509 #include <signal.h>
16510
16511 void exit_sys(const char *msg);
16512 void exit_sys_thread(const char *msg, int err);
16513 void *thread_proc1(void *param);
16514 void *thread_proc2(void *param);
16515 void *thread_proc3(void *param);
16516 void sigusr1_handler(int sno);
16517
16518 int main(void)
16519 {
```

```
16513     int result;
16514     pthread_t tid1, tid2, tid3;
16515     struct sigaction act;
16516
16517     act.sa_handler = sigusr1_handler;
16518     sigemptyset(&act.sa_mask);
16519     act.sa_flags = 0;
16520
16521     if (sigaction(SIGUSR1, &act, NULL) == -1)
16522         exit_sys("sigaction");
16523
16524     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
16525         exit_sys_thread("pthread_create", result);
16526
16527     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
16528         exit_sys_thread("pthread_create", result);
16529
16530     if ((result = pthread_create(&tid3, NULL, thread_proc3, NULL)) != 0)
16531         exit_sys_thread("pthread_create", result);
16532
16533     printf("main thread waiting at pause...\n");
16534     pause();
16535     printf("main thread resuming...\n");
16536
16537     if ((result = pthread_join(tid1, NULL)) != 0)
16538         exit_sys_thread("pthread_join", result);
16539
16540     if ((result = pthread_join(tid2, NULL)) != 0)
16541         exit_sys_thread("pthread_join", result);
16542
16543     if ((result = pthread_join(tid3, NULL)) != 0)
16544         exit_sys_thread("pthread_join", result);
16545
16546     return 0;
16547 }
16548
16549 void exit_sys(const char *msg)
16550 {
16551     perror(msg);
16552
16553     exit(EXIT_FAILURE);
16554 }
16555
16556 void exit_sys_thread(const char *msg, int err)
16557 {
16558     fprintf(stderr, "%s: %s\n", msg, strerror(err));
16559     exit(EXIT_FAILURE);
16560 }
16561
16562 void *thread_proc1(void *param)
16563 {
16564     printf("thread1 waiting at pause...\n");
16565     pause();
```

```
16566     printf("thread1 resuming...\n");
16567
16568     return NULL;
16569 }
16570
16571 void *thread_proc2(void *param)
16572 {
16573     printf("thread2 waiting at pause...\n");
16574     pause();
16575     printf("thread2 resuming...\n");
16576
16577     return NULL;
16578 }
16579
16580 void *thread_proc3(void *param)
16581 {
16582     printf("thread3 waiting at pause...\n");
16583     pause();
16584     printf("thread3 resuming...\n");
16585
16586     return NULL;
16587 }
16588
16589 void sigusr1_handler(int sno)
16590 {
16591     printf("SIGUSR1 occurred...\n");
16592 }
16593
16594 /
*-----*
-----  

16595     Sinyal set işlemi (signal disposition) thread'e özgü bir işlem değildir. ↵
    Prosesে özgü bir işlemidir. Yani biz falanca thread için ↵
16596     sinyal edemeyiz. Proses için edebiliriz. Ancak istersek prosesin hangi ↵
        thread'inin bir sinyali işleyeceğini belirleyebiliriz. ↵
16597     pthread_kill isimli fonksiyon prosesin belli bir thread'ine sinyal ↵
        gönderir. Yani kesinlikle sinyal fonksiyonu o thread tarafından ↵
16598     çalıştırılacaktır. Ancak pthread_kill ile başka bir proses başka bir ↵
        prosesin thread'ine sinyal gönderemez. Aynı proses kendi ↵
16599     thread'ine sinyal gönderebilir. (Animsanacağı gibi thread id'sini ↵
        belirten pthread_t değeri o proseste anlamlıdır. Sistem genelinde ↵
16600     tek değildir.) ↵
16601
16602     Aşağıdaki programda prosesin ana thread'i prosesin başka bir thread'ine ↵
        SIGUSR1 sinyalini göndermiştir. ↵
16603 -----*/  

16604
16605 #include <stdio.h>
16606 #include <stdlib.h>
16607 #include <string.h>
16608 #include <unistd.h>
16609 #include <pthread.h>
```

```
16610 #include <signal.h>
16611
16612 void exit_sys(const char *msg);
16613 void exit_sys_thread(const char *msg, int err);
16614 void *thread_proc1(void *param);
16615 void *thread_proc2(void *param);
16616 void *thread_proc3(void *param);
16617 void sigusr1_handler(int sno);
16618
16619 int main(void)
16620 {
16621     int result;
16622     pthread_t tid1, tid2, tid3;
16623     struct sigaction act;
16624
16625     act.sa_handler = sigusr1_handler;
16626     sigemptyset(&act.sa_mask);
16627     act.sa_flags = 0;
16628
16629     if (sigaction(SIGUSR1, &act, NULL) == -1)
16630         exit_sys("sigaction");
16631
16632     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
16633         exit_sys_thread("pthread_create", result);
16634
16635     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
16636         exit_sys_thread("pthread_create", result);
16637
16638     if ((result = pthread_create(&tid3, NULL, thread_proc3, NULL)) != 0)
16639         exit_sys_thread("pthread_create", result);
16640
16641     sleep(5);
16642
16643     if (pthread_kill(tid2, SIGUSR1) == -1)
16644         exit_sys("pthread_kill");
16645
16646     if ((result = pthread_join(tid1, NULL)) != 0)
16647         exit_sys_thread("pthread_join", result);
16648
16649     if ((result = pthread_join(tid2, NULL)) != 0)
16650         exit_sys_thread("pthread_join", result);
16651
16652     if ((result = pthread_join(tid3, NULL)) != 0)
16653         exit_sys_thread("pthread_join", result);
16654
16655     return 0;
16656 }
16657
16658 void exit_sys(const char *msg)
16659 {
16660     perror(msg);
16661
16662     exit(EXIT_FAILURE);
```

```
16663 }
16664
16665 void exit_sys_thread(const char *msg, int err)
16666 {
16667     fprintf(stderr, "%s: %s\n", msg, strerror(err));
16668     exit(EXIT_FAILURE);
16669 }
16670
16671 void *thread_proc1(void *param)
16672 {
16673     printf("thread1 waiting at pause...\n");
16674     pause();
16675     printf("thread1 resuming...\n");
16676
16677     return NULL;
16678 }
16679
16680 void *thread_proc2(void *param)
16681 {
16682     printf("thread2 waiting at pause...\n");
16683     pause();
16684     printf("thread2 resuming...\n");
16685
16686     return NULL;
16687 }
16688
16689 void *thread_proc3(void *param)
16690 {
16691     printf("thread3 waiting at pause...\n");
16692     pause();
16693     printf("thread3 resuming...\n");
16694
16695     return NULL;
16696 }
16697
16698 void sigusr1_handler(int sno)
16699 {
16700     printf("SIGUSR1 occurred...\n");
16701 }
16702
16703 /
*-----*-----*-----*
```

Başka bir prosesden başka bir prosesin thread'ine sinyal göndermek için POSIX standartlarında bir yöntem yoktur. Ancak Linux sistemlerinde bunun için tkill ve tgkill isimli sistem fonksiyonları bulunmaktadır. Fakat maalesef bu sistem fonksiyonlarını çağırın sarma kütüphane fonksiyonları (wrapper) bulunmamaktadır. tgkill sistem fonksiyonu tkill fonksiyonundan bir parametre daha fazladır. Bu fonksiyonda biz sinyal göndereceğimiz prosesin id değerini, sinyal göndereceğimiz thread'in task struct pid değerini ve sinyal numarasını veriririz. kill fonksiyonıyla (ya da komut satırından kill komutıyla) tid değeri verilerek sinyal gönderilmeye çalışılırsa

```
16709     maalesef bu durumda sinyal thread'e değil thread'in ilişkin olduğu →
16710     prosese gönderilmektedir.
16711
16712     Aşağıdaki örnekte tgkill.c programı thread'e sinyal göndermektedir. →
16713     sample.c programı ise üç thread oluşturup sinyal gelmesini →
16714     beklemektedir. Bu iki programı farklı terminalerde çalıştırınız. tgkill →
16715     ile sinyal göndermeden önce ps -a -L -o pid, tid, cmd →
16716     komutu ile yaratılmış olan thread'lerin task_struct pid değerlerini →
16717     görünüz. sample.c programı SIGUSR1 sinyalini işlemektedir.
16718     Bu sinyalin Linux sistemlerindeki numarası 10'dur.
16719     -----
16720
16721
16722
16723
16724
16725
16726
16727
16728
16729
16730
16731
16732
16733
16734
16735
16736
16737
16738
16739
16740
16741
16742
16743
16744
16745
16746
16747
16748
16749
16750
16751
16752
16753
16754
16755
16756
```

```
16757     exit(EXIT_FAILURE);
16758 }
16759
16760 /* sample.c */
16761
16762 #include <stdio.h>
16763 #include <stdlib.h>
16764 #include <string.h>
16765 #include <unistd.h>
16766 #include <pthread.h>
16767 #include <signal.h>
16768
16769 void exit_sys(const char *msg);
16770 void exit_sys_thread(const char *msg, int err);
16771 void *thread_proc1(void *param);
16772 void *thread_proc2(void *param);
16773 void *thread_proc3(void *param);
16774 void sigusr1_handler(int sno);
16775
16776 int main(void)
16777 {
16778     int result;
16779     pthread_t tid1, tid2, tid3;
16780     struct sigaction act;
16781
16782     act.sa_handler = sigusr1_handler;
16783     sigemptyset(&act.sa_mask);
16784     act.sa_flags = 0;
16785
16786     if (sigaction(SIGUSR1, &act, NULL) == -1)
16787         exit_sys("sigaction");
16788
16789     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
16790         exit_sys_thread("pthread_create", result);
16791
16792     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
16793         exit_sys_thread("pthread_create", result);
16794
16795     if ((result = pthread_create(&tid3, NULL, thread_proc3, NULL)) != 0)
16796         exit_sys_thread("pthread_create", result);
16797
16798     if ((result = pthread_join(tid1, NULL)) != 0)
16799         exit_sys_thread("pthread_join", result);
16800
16801     if ((result = pthread_join(tid2, NULL)) != 0)
16802         exit_sys_thread("pthread_join", result);
16803
16804     if ((result = pthread_join(tid3, NULL)) != 0)
16805         exit_sys_thread("pthread_join", result);
16806
16807     return 0;
16808 }
```

```
16810 void exit_sys(const char *msg)
16811 {
16812     perror(msg);
16813     exit(EXIT_FAILURE);
16814 }
16815
16816
16817 void exit_sys_thread(const char *msg, int err)
16818 {
16819     fprintf(stderr, "%s: %s\n", msg, strerror(err));
16820     exit(EXIT_FAILURE);
16821 }
16822
16823 void *thread_proc1(void *param)
16824 {
16825     printf("thread1 waiting at pause...\n");
16826     pause();
16827     printf("thread1 resuming...\n");
16828
16829     return NULL;
16830 }
16831
16832 void *thread_proc2(void *param)
16833 {
16834     printf("thread2 waiting at pause...\n");
16835     pause();
16836     printf("thread2 resuming...\n");
16837
16838     return NULL;
16839 }
16840
16841 void *thread_proc3(void *param)
16842 {
16843     printf("thread3 waiting at pause...\n");
16844     pause();
16845     printf("thread3 resuming...\n");
16846
16847     return NULL;
16848 }
16849
16850 void sigusr1_handler(int sno)
16851 {
16852     printf("SIGUSR1 occurred...\n");
16853 }
16854
16855 /
*-----*-----*-----*
```

16856 Bir thread'e gerçek zamanlı sinyal gönderebilmek için ise →
pthread_sigqueue fonksiyonu kullanılmaktadır. Tabii bu fonksiyon da →
bizeden pthread_t aldığı için ancak aynı proses içerisinde →
kullanılabilir. Başka bir prosten başka bir prosesin belli bir →
thread'ine →

```
16858     gerçek zamanlı sinyakl gönderebilmek için bir POSIX fonksiyonu yoktur. ↵
      Ancak Linux sistemlerinde rt_tgsigqueueinfo isimli
16859     sistem fonksiyonu ile bu yapılabilir. Fakat bu sistem fonksiyonunu      ↵
      çağrıran sarma bir Kütüphane fonksiyonu bulundurulmamıştır.
16860 -----
16861     -----
16862 #define _GNU_SOURCE
16863
16864 #include <stdio.h>
16865 #include <stdlib.h>
16866 #include <string.h>
16867 #include <unistd.h>
16868 #include <pthread.h>
16869 #include <signal.h>
16870
16871 void exit_sys(const char *msg);
16872 void exit_sys_thread(const char *msg, int err);
16873 void *thread_proc(void *param);
16874 void sigrt_handler(int signo, siginfo_t *info, void *context);
16875
16876 int main(void)
16877 {
16878     int result;
16879     pthread_t tid;
16880     struct sigaction act;
16881     union sigval val;
16882     int i;
16883
16884     act.sa_sigaction = sigrt_handler;
16885     sigemptyset(&act.sa_mask);
16886     act.sa_flags = SA_SIGINFO;
16887
16888     if (sigaction(SIGRTMIN, &act, NULL) == -1)
16889         exit_sys("sigaction");
16890
16891     if ((result = pthread_create(&tid, NULL, thread_proc, NULL)) != 0)
16892         exit_sys_thread("pthread_create", result);
16893
16894     sleep(1);
16895
16896     for (i = 0; i < 10; ++i) {
16897         val.sival_int = i;
16898         if ((result = pthread_sigqueue(tid, SIGRTMIN, val)) != 0)
16899             exit_sys_thread("pthread_sigqueue", result);
16900     }
16901
16902     sleep(1);
16903
16904     if ((result = pthread_cancel(tid)) == -1)
16905         exit_sys_thread("pthread_cancel", result);
16906
16907     if ((result = pthread_join(tid, NULL)) != 0)
```

```
16908         exit_sys_thread("pthread_join", result);
16909
16910     return 0;
16911 }
16912
16913 void exit_sys(const char *msg)
16914 {
16915     perror(msg);
16916
16917     exit(EXIT_FAILURE);
16918 }
16919
16920 void exit_sys_thread(const char *msg, int err)
16921 {
16922     fprintf(stderr, "%s: %s\n", msg, strerror(err));
16923
16924     exit(EXIT_FAILURE);
16925 }
16926
16927 void *thread_proc(void *param)
16928 {
16929     printf("thread1 waiting at pause...\n");
16930
16931     for (;;)
16932         pause();
16933
16934     return NULL;
16935 }
16936
16937 void sigrt_handler(int signo, siginfo_t *info, void *context)
16938 {
16939     printf("SIGRTMIN + 0 occurred with %d code\n", info-
16940             >si_value.sival_int);
16941
16942 /
16943 *-----*
-----
```

16943 POSIX sistemlerinde hem prosesin ana şalter görevinde olan bir sinyal
maskeleme kümesi (signal mask) hem de thread'lerin
16944 sinyal maskeleme kümesi vardır. Eğer sinyal kill fonksiyonuyla prosese
gönderilmişse prosesin sinyal maskeleme kümesi dikkate
16945 alınmaktadır. Eğer sinyal ptherad_kill ile belli bir thread'e gönderiliyorsa
bu durumda da thread'in maskeleme kümesi dikkate alınır.
16946 Thread'lerin maskeleme kümeleri aynı zamanda prosese gönderilen sinyalin
hangi thread tarafından işletileceği konusunda da dolaylı bir
16947 etkiye sahiptir. Şöyle ki: Eğer thread ilgili sinyal için bloke edilirse
prosese gönderilen bu sinyal artık bu thread tarafından işlenmez.
16948 Örneğin programcılar prosese gönderilen sinyalin belli bir thread tarafından
işlenmesini istiyorlarsa bu durumda tek bir therad'i sinyale
16949 açıp diğer thread'leri bu sinyale kapatabilmektedirler.
16950
16951 Aşağıdaki örnekte ikinci thread dışında tüm thread'ler SIGUSR1 sinyaline

```
16952 kapatılmıştır. Bu durumda diğer terminalden prosese SIGUSR1  
sinyali gönderildiğinde bunu iki numaralı thread çalıştıracaktır.  
16953 -----*/  
  
16954  
16955 #include <stdio.h>  
16956 #include <stdlib.h>  
16957 #include <string.h>  
16958 #include <unistd.h>  
16959 #include <pthread.h>  
16960 #include <signal.h>  
16961  
16962 void exit_sys(const char *msg);  
16963 void exit_sys_thread(const char *msg, int err);  
16964 void *thread_proc1(void *param);  
16965 void *thread_proc2(void *param);  
16966 void *thread_proc3(void *param);  
16967 void sigusr1_handler(int sno);  
16968  
16969 int main(void)  
16970 {  
16971     int result;  
16972     pthread_t tid1, tid2, tid3;  
16973     struct sigaction act;  
16974     sigset(SIG_BLOCK, &act);  
16975  
16976     act.sa_handler = sigusr1_handler;  
16977     sigemptyset(&act.sa_mask);  
16978     act.sa_flags = 0;  
16979  
16980     if (sigaction(SIGUSR1, &act, NULL) == -1)  
16981         exit_sys("sigaction");  
16982  
16983     if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)  
16984         exit_sys_thread("pthread_create", result);  
16985  
16986     if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)  
16987         exit_sys_thread("pthread_create", result);  
16988  
16989     if ((result = pthread_create(&tid3, NULL, thread_proc3, NULL)) != 0)  
16990         exit_sys_thread("pthread_create", result);  
16991  
16992     sigemptyset(&act);  
16993     sigadd(SIG_BLOCK, &act);  
16994  
16995     if ((result = pthread_sigmask(SIG_BLOCK, &act, NULL)) != 0)  
16996         exit_sys_thread("pthread_sigmask", result);  
16997  
16998     if ((result = pthread_join(tid1, NULL)) != 0)  
16999         exit_sys_thread("pthread_join", result);  
17000  
17001     if ((result = pthread_join(tid2, NULL)) != 0)  
17002         exit sys thread("pthread_join", result);
```

```
17003
17004     if ((result = pthread_join(tid3, NULL)) != 0)
17005         exit_sys_thread("pthread_join", result);
17006
17007     return 0;
17008 }
17009
17010 void exit_sys(const char *msg)
17011 {
17012     perror(msg);
17013
17014     exit(EXIT_FAILURE);
17015 }
17016
17017 void exit_sys_thread(const char *msg, int err)
17018 {
17019     fprintf(stderr, "%s: %s\n", msg, strerror(err));
17020     exit(EXIT_FAILURE);
17021 }
17022
17023 void *thread_proc1(void *param)
17024 {
17025     sigset_t sset;
17026     int result;
17027
17028     sigemptyset(&sset);
17029     sigaddset(&sset, SIGUSR1);
17030
17031     if ((result = pthread_sigmask(SIG_BLOCK, &sset, NULL)) != 0)
17032         exit_sys_thread("pthread_sigmask", result);
17033
17034     printf("thread1 waiting at pause...\n");
17035     pause();
17036     printf("thread1 resuming...\n");
17037
17038     return NULL;
17039 }
17040
17041 void *thread_proc2(void *param)
17042 {
17043     printf("thread2 waiting at pause...\n");
17044     pause();
17045     printf("thread2 resuming...\n");
17046
17047     return NULL;
17048 }
17049
17050 void *thread_proc3(void *param)
17051 {
17052     sigset_t sset;
17053     int result;
17054
17055     sigemptyset(&sset);
```

```
17056     sigaddset(&sset, SIGUSR1);
17057
17058     if ((result = pthread_sigmask(SIG_BLOCK, &sset, NULL)) != 0)
17059         exit_sys_thread("pthread_sigmask", result);
17060
17061     printf("thread3 waiting at pause...\n");
17062     pause();
17063     printf("thread3 resuming...\n");
17064
17065     return NULL;
17066 }
17067
17068 void sigusr1_handler(int sno)
17069 {
17070     printf("SIGUSR1 occurred...\n");
17071 }
17072 /
*-----*
-----*
17074 Nadiren programcı kritik birtakım işlemler yaparken sinyalleri bloke
      edip sonra açıp pause ile bekleyebilmektedir. ➔
17075
17076     siprocmask(<sinyalleri bloke et>);
17077
17078     <kritik kod>
17079
17080     siprocmask(<sinyalleri aç>);
17081 ---> Problem var!
17082     pause();
17083
17084 Programcı sinyalleri açıp pause beklemesini yapmak istediği sırada sinyal
      prosese gönderilirse henüz akış pause fonksiyona girmeden ➔
17085 sinyal teslim edilebilir. Daha sonra akış pause fonksiyonuna girdiğinde ➔
      buradan çıkışlamamış olabilir. Bu problem atomik bir biçimde
17086 sinyalleri açarak pause yapan bir fonksiyonla çözülebilir. İşte sigsuspend ➔
      fonksiyonu bunu yapmaktadır.
17087
17088 Yukarıda açıklanan temayı uygulayan aşağıdaki kodu inceleyiniz. Prosesin ➔
      sinyal maskelerini açıp pause ile değil sigsuspend
17089 ile beklenmelidir.
17090 -----
-----*/
```

17091
17092 #include <stdio.h>
17093 #include <stdlib.h>
17094 #include <string.h>
17095 #include <errno.h>
17096 #include <unistd.h>
17097 #include <signal.h>
17098
17099 void exit_sys(const char *msg);
17100 void sigusr1_handler(int sno);

```
17101
17102 int main(void)
17103 {
17104     struct sigaction act;
17105     sigset(SIG_BLOCK, &act.sa_mask);
17106     int i;
17107
17108     act.sa_handler = sigusr1_handler;
17109     sigemptyset(&act.sa_mask);
17110     act.sa_flags = 0;
17111
17112     if (sigaction(SIGUSR1, &act, NULL) == -1)
17113         exit_sys("sigaction");
17114
17115     sigfillset(&act.sa_mask);
17116     if (sigprocmask(SIG_BLOCK, &act.sa_mask, NULL) == -1)
17117         exit_sys("sigprocmask");
17118
17119     for (i = 0; i < 10; ++i) {
17120         sleep(1);
17121         printf("critical region running...\n");
17122     }
17123
17124     sigemptyset(&act.sa_mask);
17125     if (sigsuspend(&act.sa_mask) == -1 && errno != EINTR) /* pause yerine →
17126         sigsuspend kullanılmalı */
17127         exit_sys("sigsuspend");
17128
17129     sigfillset(&act.sa_mask);
17130     if (sigprocmask(SIG_UNBLOCK, &act.sa_mask, NULL) == -1)
17131         exit_sys("sigprocmask");
17132
17133     printf("process ends...\n");
17134
17135     return 0;
17136 }
17137 void exit_sys(const char *msg)
17138 {
17139     perror(msg);
17140
17141     exit(EXIT_FAILURE);
17142 }
17143
17144 void sigusr1_handler(int sno)
17145 {
17146     printf("SIGUSR1 occurred...\n");
17147 }
17148
17149 /
*-----→
-----→
17150     sigwait fonksiyonu programının belirlediği sinyallerden herhangi biri →
```

oluşana kadar bekleme yapar. Eğer programcının belirlemediği bir sinyal oluşursa ve sinyal fonksiyonu da set edilmişse fonksiyon çalıştırılır fakat bekleme devam eder. Eğer programcının belirlemediği bir sinyal oluşursa fakat bu sinyal için sinyal fonksiyonu set edilmemişse bu durumda default eylem gerçekleşir (muhtemelen prosesin sonlandırılması). Eğer sigwait'te beklerken programcının belirlediği sinyallerden biri oluşursa ve bu sinyal için sinyal fonksiyonu set edilmişse sinyal fonksiyonu çağrılmaz ancak bu sinyal için sinyal fonksiyonu set edilmemişse default eylem uygulanır (muhtemelen prosesin sonlandırılması). Eğer sigwait çağrıldığında zaten beklenen sinyallerin bazıları pending durumdaysa sigwait hiç bekleme yapmaz.

sigwait fonksiyonun sigwaitinfo isimli siginfo_t parametreli versiyonu da vardır. Bu fonksiyon yalnız oluşan sinyalin numarasını değil onun siginfo_t bilgilerini de vermektedir.

Aşağıdaki örnekte sigwait fonksiyonunda SIGUSR1 ve SIGUSR2 sinyalleri beklenmektedir. Başka bir sinyal oluşursa proses sonlandırılır. SIGUSR1 ve SIGUSR2 için sinyal fonksiyonu set edilmiştir. Bu durumda bu sinyaller oluşursa sinyal fonksiyonu çağrılmadan bekleme sonlandırılır. pause ve sisuspend fonksiyonlarının sinyal fonksiyonun çalışmasına yol açtığını anımsayınız.

```
-----*/  
17165  
17166 #include <stdio.h>  
17167 #include <stdlib.h>  
17168 #include <string.h>  
17169 #include <unistd.h>  
17170 #include <signal.h>  
17171  
17172 void exit_sys(const char *msg);  
17173 void sigusr1_handler(int sno);  
17174 void sigusr2_handler(int sno);  
17175  
17176 int main(void)  
17177 {  
17178     struct sigaction act;  
17179     sigset(SIGUSR1, &sigusr1_handler);  
17180     sigemptyset(&act.sa_mask);  
17181     act.sa_flags = 0;  
17182  
17183     if (sigaction(SIGUSR1, &act, NULL) == -1)  
17184         exit_sys("sigaction");  
17185  
17186     act.sa_handler = sigusr2_handler;
```

```
17191     if (sigaction(SIGUSR2, &act, NULL) == -1)
17192         exit_sys("sigaction");
17193
17194     sigemptyset(&sset);
17195     sigaddset(&sset, SIGUSR1);
17196
17197     printf("process waiting for SIGUSR1 signal...\n");
17198
17199     if ((result = sigwait(&sset, &signo)) != 0) {
17200         fprintf(stderr, "sigwait: %s\n", strerror(result));
17201         exit(EXIT_FAILURE);
17202     }
17203
17204     if (signo == SIGUSR1)
17205         printf("sigwait returns due to SIGUSR1\n");
17206     else
17207         printf("sigwait returns due to SIGINT\n");
17208
17209     printf("process ends...\n");
17210
17211     return 0;
17212 }
17213
17214 void exit_sys(const char *msg)
17215 {
17216     perror(msg);
17217
17218     exit(EXIT_FAILURE);
17219 }
17220
17221 void sigusr1_handler(int sno)
17222 {
17223     printf("SIGUSR1 occurred...\n");
17224 }
17225
17226 void sigusr2_handler(int sno)
17227 {
17228     printf("SIGUSR2 occurred...\n");
17229 }
17230
17231 /
*-----*
-----*
17232     sleep fonksiyonu parametresiyle belirtilen saniye kadar thread'i blokede
17233     bekletir. Fonksiyon zamandan dolayı sonlanırsa
17234     0 değerine sinyal dolayısıyla sonlanırsa kalan saniye değerine geri
17235     döner. sleep yeniden başlatılabilen (restartable) bir
17236     fonksiyon değildir.
17237
17238     Aşağıdaki örnekte SIGINT sinyali gelse bile sleep ile 10 saniye
17239     civarında bekleme yapılmıştır.
17240 -----*/
```

```
17238
17239 #include <stdio.h>
17240 #include <stdlib.h>
17241 #include <string.h>
17242 #include <unistd.h>
17243 #include <signal.h>
17244
17245 void exit_sys(const char *msg);
17246 void sigint_handler(int sno);
17247
17248 int main(void)
17249 {
17250     struct sigaction act;
17251     int t;
17252
17253     act.sa_handler = sigint_handler;
17254     sigemptyset(&act.sa_mask);
17255     act.sa_flags = 0;
17256
17257     if (sigaction(SIGINT, &act, NULL) == -1)
17258         exit_sys("sigaction");
17259
17260     printf("waiting 10 seconds even if SIGINT occurs...\n");
17261
17262     t = 10;
17263     while ((t = sleep(t)) != 0)
17264         ;
17265
17266     printf("process ends..\n");
17267
17268     return 0;
17269 }
17270
17271 void exit_sys(const char *msg)
17272 {
17273     perror(msg);
17274
17275     exit(EXIT_FAILURE);
17276 }
17277
17278 void sigint_handler(int sno)
17279 {
17280     printf("SIGINT occurred...\n");
17281 }
17282
17283 /
*-----*
-----*
17284     nanosleep nano saniye (saniyenin milyarda biri) çözünürlüğüne sahip bir POSIX bekleme fonksiyonudur. Fonksiyon timespec imzalı yapınesini parametre olarak alır. Sinyal oluşursa sonlanır ve kalan zamanı da yine bize timespec biçiminde verir.
17285     Her ne kadar bu fonksiyon nanosaniye çözünürlüğüne sahipse de işletim
```

sistemleri genellikle bu çözünürlüğü destekleyemektedir.

17287 Bu durumda bekleme tam istenen miktarda gerçekleşmez. Örneğin Linux →
17288 işletim sistemlerinde sleep kuyruğundaki thread'ler timer
17289 kesmesinde kontrol edilmektedir. Dolayısıyla genellikle bu çözünürlük 1 →
 milisaniyedir.

17289 -----*/

17290
17291 #include <stdio.h>
17292 #include <stdlib.h>
17293 #include <time.h>
17294
17295 void exit_sys(const char *msg);
17296
17297 int main(void)
17298 {
17299 struct timespec ts;
17300
17301 printf("waiting 3.5 seconds...\n");
17302 ts.tv_sec = 3;
17303 ts.tv_nsec = 500000000;
17304
17305 if (nanosleep(&ts, NULL) == -1)
17306 exit_sys("nanosleep");
17307
17308 printf("sleep finished..\n");
17309
17310 return 0;
17311 }
17312
17313 void exit_sys(const char *msg)
17314 {
17315 perror(msg);
17316
17317 exit(EXIT_FAILURE);
17318 }
17319
17320 / -----*/
17321 Ayrıca POSIX standartlarından 2008'de çıkarılmış olan mikrosaniye →
17322 çözünürlüğe sahip bir usleep fonksiyonu da vardır. Bu
17323 fonksiyon Linux sistemlerinde hala desteklenmektedir.
17323 -----*/

17324
17325 #include <stdio.h>
17326 #include <stdlib.h>
17327 #include <unistd.h>
17328
17329 void exit_sys(const char *msg);
17330
17331 int main(void)

```
17332 {
17333     printf("waiting 3.5 seconds even if SIGINT occurs...\n");
17334
17335     if (usleep(3500000) == -1)
17336         exit_sys("usleep");
17337
17338     printf("sleep finished..\n");
17339
17340     return 0;
17341 }
17342
17343 void exit_sys(const char *msg)
17344 {
17345     perror(msg);
17346
17347     exit(EXIT_FAILURE);
17348 }
17349
17350 /
*-----*
-----  
17351 Amacımız bekleme yapmak değil de zaman ölçmek ise ilk akla gelen POSIX
17352 fonksiyonu clock_gettime olmalıdır. Bu fonksiyonun
17353 birinci parametresi clockid_t türündendir. Hesabın yapılacağı saat
17354 cinsini belirtir. Bu parametre CLOCK_REALTIME biçiminde girilirse
17355 01/01/1970'ten geçen nano saniye sayısı bize verilir. Tabii biz buradan
17356 elde ettiğimiz mutlak zamanı C'nin klasik zaman fonksiyonlarına
17357 sokaup dönüştürmeler yapabiliriz. Saat ürünü olarak CLOCK_MONOTONIC değeri
17358 geçilirse belli bir zamandan itibaren geçen görelî zaman elde edilir.
17359 Linux sistemleri bu durumda boot zamanından itibaren geçen zamanı bize
17360 vermektedir.  
17361 -----*/  
17362 #include <stdio.h>
17363 #include <stdlib.h>
17364 #include <time.h>
17365
17366 void exit_sys(const char *msg);
17367
17368 int main(void)
17369 {
17370     struct timespec ts;
17371     struct tm *pt;
17372
17373     if (clock_gettime(CLOCK_REALTIME, &ts) == -1)
17374         exit_sys("clock_gettime");
17375
17376     printf("Seconds: %ld\n", (long)ts.tv_sec);
17377     printf("Nanoseconds: %ld\n", (long)ts.tv_nsec);
17378
17379     puts(ctime(&ts.tv_sec));
17380 }
```

```
17377     pt = localtime(&ts.tv_sec);
17378     printf("%02d/%02d/%04d %02d:%02d:%02d\n", pt->tm_mday, pt->tm_mon + 1,    ↵
17379             pt->tm_year + 1900, pt->tm_hour, pt->tm_min, pt->tm_sec);
17380     return 0;
17381 }
17382
17383 void exit_sys(const char *msg)
17384 {
17385     perror(msg);
17386
17387     exit(EXIT_FAILURE);
17388 }
17389
17390 /
*-----*
-----*
17391     Programın iki noktası arasında geçen gerçek zamanı taşınabilir bir      ↵
17392         biçimde ölçmek için ilk akla gelecek yöntem clock_gettime      ↵
17393         olmalıdır. Tabii benzer başka yöntemler de kullanılabilir.
-----*/
```

```
17394
17395 #include <stdio.h>
17396 #include <stdlib.h>
17397 #include <time.h>
17398
17399 void exit_sys(const char *msg);
17400
17401 int main(void)
17402 {
17403     struct timespec ts1, ts2;
17404     int i;
17405     double result;
17406
17407     if (clock_gettime(CLOCK_REALTIME, &ts1) == -1)
17408         exit_sys("clock_gettime");
17409
17410     for (i = 0; i < 1000000000; ++i)
17411         ;
17412
17413     if (clock_gettime(CLOCK_REALTIME, &ts2) == -1)
17414         exit_sys("clock_gettime");
17415
17416     result = (ts2.tv_sec - ts1.tv_sec) * 1000000000.0 + ts2.tv_nsec -      ↵
17417         ts1.tv_nsec;
17418     printf("Elapsed time (nonoseconds): %.0f\n", result);
17419     printf("Elapsed time (seconds): %f\n", result / 1000000000.0);
17420
17421     return 0;
17422 }
17423 void exit_sys(const char *msg)
```

```
17424 {
17425     perror(msg);
17426
17427     exit(EXIT_FAILURE);
17428 }
17429
17430 /
*-----*
-----*
17431     clock isimli standart C fonksiyonu bize zamanı clock_t türünden verir. ↵
    Ancak geçen zamanın saniye cinsine dönüştürülmesi ↵
17432 için CLOCKS_PER_SEC değerine bölünmesi gerekmektedir. Genellikle bu ↵
    fonksiyonlar işletim sisteminin timer kesmesi ile tick ↵
17433 sayısını verecek biçimde yazılmışlardır. Bu durumda zaman ölçmenin C ↵
    genelinde en taşınabilir yolu clock fonksiyonunu ↵
17434 kullanmak olabilir. Ancak genel olarak clock_gettime fonksiyonunun ↵
    çözünürlüğünün daha yüksek olduğu varsayılmalıdır.
17435 -----*/
17436
17437 #include <stdio.h>
17438 #include <stdlib.h>
17439 #include <time.h>
17440
17441 void exit_sys(const char *msg);
17442
17443 int main(void)
17444 {
17445     clock_t c1, c2;
17446     int i;
17447     double result;
17448
17449     c1 = clock();
17450
17451     for (i = 0; i < 1000000000; ++i)
17452         ;
17453
17454     c2 = clock();
17455
17456     result = (double)(c2 - c1) / CLOCKS_PER_SEC;
17457     printf("Elapsed time (seconds): %f\n", result);
17458
17459     return 0;
17460 }
17461
17462 void exit_sys(const char *msg)
17463 {
17464     perror(msg);
17465
17466     exit(EXIT_FAILURE);
17467 }
17468
17469 /
```

```
*-----  
17470     clock_gettime fonksiyonundaki duyarlılık clock_getres fonksiyonuyla elde  
17471         edilebilir. Pek çok işlemcide zamansal nano saniye  
17472         çözünürlük için özel makine komutları bulunmaktadır. Örneğin x64  
17473         işlemcide çalışan sanal makinede bu değer 1 nanosaniye çıkmıştır.  
17474     -----*/  
17475 #include <stdio.h>  
17476 #include <stdlib.h>  
17477 #include <time.h>  
17478 void exit_sys(const char *msg);  
17479  
17480 int main(void)  
17481 {  
17482     struct timespec ts;  
17483  
17484     if (clock_getres(CLOCK_REALTIME, &ts) == -1)  
17485         exit_sys("clock_getres");  
17486  
17487     printf("%ld second and %ld nano seconds:\n", (long)ts.tv_sec, (long)  
17488         ts.tv_nsec);  
17489  
17490     return 0;  
17491 }  
17492 void exit_sys(const char *msg)  
17493 {  
17494     perror(msg);  
17495  
17496     exit(EXIT_FAILURE);  
17497 }  
17498 /  
*-----  
17500     clock_gettime fonksiyonda clockid_t olarak CLOCK_PROCESS_CPUTIME_ID  
17501         geçilirse prosesin yalnızca CPU'da harcadığı zamanların  
17502         toplamı verilir. Bu gerçek zamanдан daha kısa olacaktır. eğer proses çok  
17503         thread'ten oluşuyorsa bu hesaba tüm thread'lerin CPU zamanları  
17504         toplanır.  
17505         Fakat clockid_t olarak CLOCK_THREAD_CPUTIME_ID verilirse bu da spesifik  
17506         bir thread'in (fonksiyonu çağrıran) CPU zamanını ölçmekte kullanılır.  
17507 -----*/  
17508 #include <stdio.h>  
17509 #include <stdlib.h>  
17510 #include <time.h>  
17511 #include <unistd.h>
```

```
17510 void exit_sys(const char *msg);
17511
17512 int main(void)
17513 {
17514     struct timespec ts1, ts2;
17515     int i;
17516     double result;
17517
17518     if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts1) == -1)
17519         exit_sys("clock_gettime");
17520
17521     for (i = 0; i < 10; ++i)
17522         sleep(1);
17523
17524     if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts2) == -1)
17525         exit_sys("clock_gettime");
17526
17527     result = (ts2.tv_sec - ts1.tv_sec) * 1000000000.0 + ts2.tv_nsec -
17528         ts1.tv_nsec;
17529     printf("Elapsed time (nonoseconds): %.0f\n", result);
17530     printf("Elapsed time (seconds): %f\n", result / 1000000000.0);
17531
17532     return 0;
17533 }
17534 void exit_sys(const char *msg)
17535 {
17536     perror(msg);
17537
17538     exit(EXIT_FAILURE);
17539 }
17540
17541 /
*-----*
-----*
17542     Başka bir prosese ilişkin clockid_t elde etmek mümkün değildir. Bu durumda o
17543     prosese ilişkin zamansal ölçümler yapılabilir.
17544     Bunun için clock_getcpuclockid POSIX fonksiyonu kullanılmaktadır. Eğer
17545     spesifik bir thread'e ilişkin clockid_t elde etmek için ise
17546     pthread_getcpuclockid fonksiyonu kullanılmaktadır.
17547 */
*-----*/
-----*
17548     İlgili thread'i duyarlıklı bir biçimde bekletmek için nanosleep
17549     fonksiyonu görmüştük. Her ne kadar POSIX standartlarından kaldırılmış
17550     olsa da Linux sistemlerinde desteklenen mikrosaniye duyarlıklı bir
     usleep fonksiyonu da vardı. Klasik sleep fonksiyonu ise eski bir
     fonksiyondu ve saniye duyarlılığına sahipti. İşte nanosleep
     fonksiyonunun clock_nanosleep isimli biraz daha gelişmiş bir biçimini
     vardır.
```

17551 Fonksiyonun bu versiyonu `clockid_t` aldığı için daha yeteneklidir. ↗
clock_nanosleep fonksiyonunun nanosleep fonksiyonundan en önemli ↗
avantajı ↗

17552 gerçek bekleme zamanını ölçebilmesidir. Eğer bekleme nanosleep ↗
fonksiyonuyla yapılrsa ve programda sinyal de kullanılıyorsa ↗
nanosleep sinyal ↗

17553 geldiğinde başarısızlıkla sonlandırılacak (`errno = EINTR`) programcı da ↗
kalan süreyi alarak yeniden nanosleep fonksiyonunu çağrıp beklemeye ↗
devam edecektir. ↗

17554 Ancak her sinyal geldiğinde sinyal fonksiyonun çalıştırılması zamanı ↗
ekstra bir zaman olarak bekleyemeye eklenecektir. ↗

17555

17556 Aşağıdaki örnekte programcı nanosleep kullanarak sinyalli bir ortamda 10 ↗
saniye beklemek istemiştir. Ancak SIGINT sinyali oluştuğunda ↗

17557 bu toplam bekleme gerçek zamana göre fazlalaşacaktır. ↗

17558 -----*/

17559

17560 `stdio.h>`

17561 `#include <stdlib.h>`

17562 `#include <time.h>`

17563 `#include <errno.h>`

17564 `#include <signal.h>`

17565

17566 `void exit_sys(const char *msg);`

17567 `void sigint_handler(int sno);`

17568

17569 `int main(void)`

17570 {

17571 `struct sigaction act;`

17572 `struct timespec ts, rm;`

17573 `struct timespec ts1, ts2;`

17574 `double result;`

17575

17576 `act.sa_handler = sigint_handler;`

17577 `sigemptyset(&act.sa_mask);`

17578 `act.sa_flags = 0;`

17579

17580 `if (sigaction(SIGINT, &act, NULL) == -1)`

17581 `exit_sys("sigaction");`

17582

17583 `printf("waiting 10 seconds...\n");`

17584 `ts.tv_sec = 10;`

17585 `ts.tv_nsec = 0;`

17586

17587 `if (clock_gettime(CLOCK_REALTIME, &ts1) == -1)`

17588 `exit_sys("clock_gettime");`

17589

17590 `while (nanosleep(&ts, &rm) == -1 && errno == EINTR)`

17591 `ts = rm;`

17592

17593 `if (clock_gettime(CLOCK_REALTIME, &ts2) == -1)`

17594 `exit_sys("clock_gettime");`

```
17595
17596     result = (ts2.tv_sec - ts1.tv_sec) * 1000000000.0 + ts2.tv_nsec -
17597         ts1.tv_nsec;
17598     printf("Elapsed time (nonoseconds): %.0f\n", result);
17599     printf("Elapsed time (seconds): %f\n", result / 1000000000.0);
17600
17601     return 0;
17602 }
17603 void exit_sys(const char *msg)
17604 {
17605     perror(msg);
17606
17607     exit(EXIT_FAILURE);
17608 }
17609
17610 void sigint_handler(int sno)
17611 {
17612     int i;
17613
17614     for (i = 0; i < 1000000000; ++i)
17615         ;
17616 }
17617
17618 /
*-----*
-----*
17619     clock_nanosleep fonksiyonun nanosleep fonksiyonundan en önemli avantajı
17620         sinyalli ortamda timer türü TIMER_ABSTIME alınarak
17621         umulan beklemeyi yapabilmesidir.
17622 -----*/
17623 #include <stdio.h>
17624 #include <stdlib.h>
17625 #include <time.h>
17626 #include <errno.h>
17627 #include <signal.h>
17628
17629 void exit_sys(const char *msg);
17630 void sigint_handler(int sno);
17631
17632 int main(void)
17633 {
17634     struct sigaction act;
17635     struct timespec ts;
17636     struct timespec ts1, ts2;
17637     double elapsed;
17638     int result;
17639
17640     act.sa_handler = sigint_handler;
17641     sigemptyset(&act.sa_mask);
17642     act.sa_flags = 0;
```

```
17643
17644     if (sigaction(SIGINT, &act, NULL) == -1)
17645         exit_sys("sigaction");
17646
17647     if (clock_gettime(CLOCK_REALTIME, &ts) == -1)
17648         exit_sys("clock_gettime");
17649
17650     ts.tv_sec += 10;
17651
17652     if (clock_gettime(CLOCK_REALTIME, &ts1) == -1)
17653         exit_sys("clock_gettime");
17654
17655     while ((result = clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &ts,
17656                                         NULL)) == EINTR)
17657         ;
17658
17659     if (clock_gettime(CLOCK_REALTIME, &ts2) == -1)
17660         exit_sys("clock_gettime");
17661
17662     elapsed = (ts2.tv_sec - ts1.tv_sec) * 1000000000.0 + ts2.tv_nsec -
17663             ts1.tv_nsec;
17664     printf("Elapsed time (nonoseconds): %.0f\n", elapsed);
17665     printf("Elapsed time (seconds): %f\n", elapsed / 1000000000.0);
17666
17667     return 0;
17668 }
17669 void exit_sys(const char *msg)
17670 {
17671     perror(msg);
17672     exit(EXIT_FAILURE);
17673 }
17674
17675 void sigint_handler(int sno)
17676 {
17677     int i;
17678
17679     for (i = 0; i < 1000000000; ++i)
17680         ;
17681 }
17682
17683 /
*-----*
-----*
17684 Periyodik bir biçimde iş yapılmasını sağlayan timer'lara İngilizce      ↵
17685 "interval timer" denilmektedir. POSIX sistemlerinde iki      ↵
interval timer mekanizması vardır. setitimer ile oluşturulan interval      ↵
timer'in kullanımı kolaydır. Ancak kullanımı zor olan      ↵
daha yetenekli interval timer mekanizması POSIX standartlarına eklendiği      ↵
için bu fonksiyon "obsolete" yapılmıştır. setitimer      ↵
fonksiyonunda interval timer için periyodik işleme başlamak için gereken      ↵
zaman ile periyot zamanı ayrı ayrı verilir. Yine interval timer'in      ↵
```

```
17688     türleri vardır. Her tür zaman dolduğunda farklı bir sinyalin oluşmasına →  
17689         yol açar. ITIMER_REAL timer'ı SIGALRM sinyaline yol açar ve  
17690         gerçek zamanlı timer'dır. Prosein 3 tüden tek farklı farklı tek interval →  
17691             timer'ları vardır. Fonksiyon ikinci kez çağrıldığında eski timer rest →  
17692             edilir.  
17693             Oradaki değerler programcıya verilebilmektedir. Aträca getitimer isimli →  
17694                 fonksiyon da mevcut interval timer'in timer değerlerini almak için  
17695                 kullanılabilmektedir.  
17696-----*/  
17695  
17696 #include <stdio.h>  
17697 #include <stdlib.h>  
17698 #include <sys/time.h>  
17699 #include <unistd.h>  
17700 #include <signal.h>  
17701  
17702 void exit_sys(const char *msg);  
17703 void sigalarm_handler(int sno);  
17704  
17705 int main(void)  
17706 {  
17707     struct itimerval itv;  
17708     struct sigaction act;  
17709  
17710     act.sa_handler = sigalarm_handler;  
17711     sigemptyset(&act.sa_mask);  
17712     act.sa_flags = 0;  
17713  
17714     if (sigaction(SIGALRM, &act, NULL) == -1)  
17715         exit_sys("sigaction");  
17716  
17717     itv.it_interval.tv_sec = 1;  
17718     itv.it_interval.tv_usec = 0;  
17719  
17720     itv.it_value.tv_sec = 5;  
17721     itv.it_value.tv_usec = 0;  
17722  
17723     if (setitimer(ITIMER_REAL, &itv, NULL) == -1)  
17724         exit_sys("setitimer");  
17725  
17726     for (;;)  
17727         pause();  
17728  
17729     return 0;  
17730 }  
17731  
17732 void exit_sys(const char *msg)  
17733 {  
17734     perror(msg);
```

```
17735
17736     exit(EXIT_FAILURE);
17737 }
17738
17739 void signal_handler(int sno)
17740 {
17741     static int i = 0;
17742
17743     printf("%d ", i);
17744     fflush(stdout);
17745
17746     if (i == 10) {
17747         printf("\n");
17748         exit(EXIT_SUCCESS);
17749     }
17750
17751     ++i;
17752 }
17753
17754 /
*-----
```

17755 Periyodik timer (interval timers) yaratmak için yukarıda setitimer fonksiyonunu kullanmıştık. Bu fonksiyon POSIX standartlarında artık "obsolete" ilan edilmiştir. Dolayısıyla gelecekte standartlardan kaldırılabilir. Bunun yerine POSIX'e daha yetenekli ama kullanılması daha zor olan bir grup yeni periyodik timer fonksiyonu eklenmiştir.

17756 Modern periyodik timer mekanizmasını kullanabilmek için önce timer'in timer_create fonksiyonu ile yaratılması gereklidir. timer_create fonksiyonu bizden sigevent türünden bir yapı nesnesinin adresini ister. O yapı nesnesini bizim tanımlayıp içini bizim doldurmamız gereklidir.

17757 sigevent yapısının sigev_notify elemanı periyodik timerde bizim nasıl haberdar edileceğimizi belirtir. Eğer bu elemana SIGEV_SIGNAL değeri girilirse biz sinyal yoluyla haberdar ediliriz. Ancak söz konusu sinyal SIGALRM olmak zorunda değildir. Söz konusu sinyalin numarası yapının sigev_signo elemanına girilmelidir. Eğer sigev_notify elemanı SIGEV_THREAD olarak girilirse işletim sistemi bir thread yaratıp bizim yapının sigev_notify_function elemanıyla belirttiğimiz fonksiyonunu bu thread akışıyla çağırarak bizi haberdar eder.

17758 Ancak işletim sisteminin tek bir thread'le mi bu işi yapacağı yoksa her çağrı için ayrı bir thread mi kullanacağı sistemden sisteme değişebilmektedir.

17759 Bu thread'in yaratılmasındaki thread özellikleri yapının sigev_notify_attributes elemanına girilebilir. Aynı zamanda istenirse oluşan sinyale bir değer de ilişkilendirilebilir. Bu değer sigev_value elemanına yerleştirilir ve sigaction ile siginfo_t parametreli sinyal fonksiyonu ile elde edilir.

17760 timer_create bir timer'i oluşturur ve onun id'sini fonksiyonun sigev_value timerid parametresine yerleştirir. Bu id sonraki fonksiyonlarda

```
17769 kullanılacaktır.
17770
17771 Timer yaratıldıkten sonra timer'ı kurma işlemi timer_settime fonksiyonıyla →
     yapılmaktadır. Burada yine ilk periyota kadar geçen zaman
17772 ve periyot zamanı belirtilmektedir.
17773
17774 Timer kullanıldıktan sonra timer_delete fonksiyonıyla silinir. Yaratılan →
     timer'lar fork işlemi sırasında alt prosese etkinliğini kaybetmektedir.
17775 Yine exec işlemi yapıldığında timer'lar yok edilmektedir.
17776
17777 Aşağıdaki örnekte sinyal yoluyla (SIGUSR1) haberdar edilme yöntemi →
     kullanılmıştır.
17778 -----
17779
17780 #include <stdio.h>
17781 #include <stdlib.h>
17782 #include <time.h>
17783 #include <setjmp.h>
17784 #include <unistd.h>
17785 #include <signal.h>
17786
17787 void exit_sys(const char *msg);
17788 void sigusr1_handler(int sno);
17789
17790 jmp_buf g_jb;
17791
17792 int main(void)
17793 {
17794     struct sigaction act;
17795     struct sigevent se;
17796     timer_t mytimer;
17797     struct itimerspec tspec;
17798
17799     act.sa_handler = sigusr1_handler;
17800     sigemptyset(&act.sa_mask);
17801     act.sa_flags = 0;
17802
17803     if (sigaction(SIGUSR1, &act, NULL) == -1)
17804         exit_sys("sigaction");
17805
17806     se.sigev_notify = SIGEV_SIGNAL;
17807     se.sigev_signo = SIGUSR1;
17808     se.sigev_value.sival_int = 0;
17809
17810     if (timer_create(CLOCK_REALTIME, &se, &mytimer) == -1)
17811         exit_sys("timer_create");
17812
17813     tspec.it_value.tv_sec = 5;
17814     tspec.it_value.tv_nsec = 0;
17815
17816     tspec.it_interval.tv_sec = 1;
17817     tspec.it_interval.tv_nsec = 0;
```

```
17818
17819     if (timer_settime(mytimer, 0, &tspec, NULL) == -1)
17820         exit_sys("timer_settime");
17821
17822     if (setjmp(g_jb) == 1) {
17823         if (timer_delete(mytimer) == -1)
17824             exit_sys("timer_delete");
17825         exit(EXIT_SUCCESS);
17826     }
17827
17828     for (;;)
17829         pause();
17830
17831     return 0;
17832 }
17833
17834 void exit_sys(const char *msg)
17835 {
17836     perror(msg);
17837
17838     exit(EXIT_FAILURE);
17839 }
17840
17841 void sigusr1_handler(int sno)
17842 {
17843     static int count = 0;
17844
17845     if (count == 10)
17846         longjmp(g_jb, 1);
17847
17848     printf("sigusr1 occurred...\n");
17849
17850     ++count;
17851 }
17852
17853 /
*-----*
-----*
17854     Eğer biz sinyal fonksiyonuna değer de aktarmak istiyorsak bu durumda
        siginfo_t * parametreli sinyal fonksiyonu set etmeliyiz.
17855 -----*/
17856
17857 #include <stdio.h>
17858 #include <stdlib.h>
17859 #include <time.h>
17860 #include <setjmp.h>
17861 #include <unistd.h>
17862 #include <signal.h>
17863
17864 void exit_sys(const char *msg);
17865 void sigusr1_handler(int sno, siginfo_t *info, void *ucont);
17866
```

```
17867 jmp_buf g_jb;
17868
17869 int main(void)
17870 {
17871     struct sigaction act;
17872     struct sigevent se;
17873     timer_t mytimer;
17874     struct itimerspec tspec;
17875
17876     act.sa_sigaction = sigusr1_handler;
17877     sigemptyset(&act.sa_mask);
17878     act.sa_flags = SA_SIGINFO;
17879
17880     if (sigaction(SIGUSR1, &act, NULL) == -1)
17881         exit_sys("sigaction");
17882
17883     se.sigev_notify = SIGEV_SIGNAL;
17884     se.sigev_signo = SIGUSR1;
17885     se.sigev_value.sival_int = 100;
17886
17887     if (timer_create(CLOCK_REALTIME, &se, &mytimer) == -1)
17888         exit_sys("timer_create");
17889
17890     tspec.it_value.tv_sec = 5;
17891     tspec.it_value.tv_nsec = 0;
17892
17893     tspec.it_interval.tv_sec = 1;
17894     tspec.it_interval.tv_nsec = 0;
17895
17896     if (timer_settime(mytimer, 0, &tspec, NULL) == -1)
17897         exit_sys("timer_settime");
17898
17899     if (setjmp(g_jb) == 1) {
17900         if (timer_delete(mytimer) == -1)
17901             exit_sys("timer_delete");
17902         exit(EXIT_SUCCESS);
17903     }
17904
17905     for (;;)
17906         pause();
17907
17908     return 0;
17909 }
17910
17911 void exit_sys(const char *msg)
17912 {
17913     perror(msg);
17914
17915     exit(EXIT_FAILURE);
17916 }
17917
17918 void sigusr1_handler(int sno, siginfo_t *info, void *ucontext)
17919 {
```

```
17920     static int count = 0;
17921
17922     if (count == 10)
17923         longjmp(g_jb, 1);
17924
17925     printf("sigusr1 occurred: %d\n", info->si_value.sival_int);
17926
17927     ++count;
17928 }
17929
17930 /
*-----*
-----*
17931     Periyot dolduğunda bir sinyalin oluşması yerine belirlenen bir
17932         fonksiyonun çağrılması da sağlanabilir. Bunun için yapının
17933         sigev_notify elemanı SIGEV_THREAD biçiminde girilmelidir. Çağrılacak
17934             fonksiyon da yapının sigev_notify_function elemanına girilir.
17935     İşletim sistemi kendisi bir therad yaratıp bizim girdiğimiz fonksiyonu o
17936         thread akışının çalıştırmasını sağlamaktadır.
17937     Linux bir kere thread yaratıp hep aynı thread'le sinyal fonksiyonu
17938         çağrılmaktadır.
17939 -----*/
17940
17941 #include <stdio.h>
17942 #include <stdlib.h>
17943 #include <time.h>
17944 #include <setjmp.h>
17945 #include <unistd.h>
17946 #include <signal.h>
17947
17948
17949 void exit_sys(const char *msg);
17950 void signal_notification_handler(union sigval val);
17951
17952 jmp_buf g_jb;
17953
17954 int main(void)
17955 {
17956     struct sigevent se;
17957     timer_t mytimer;
17958     struct itimerspec tspec;
17959
17960     se.sigev_notify = SIGEV_THREAD;
17961     se.sigev_notify_function = signal_notification_handler;
17962     se.sigev_notify_attributes = NULL;
17963     se.sigev_value.sival_int = 100;
17964
17965     if (timer_create(CLOCK_REALTIME, &se, &mytimer) == -1)
17966         exit_sys("timer_create");
17967
17968     tspec.it_value.tv_sec = 5;
17969     tspec.it_value.tv_nsec = 0;
17970
17971 }
```

```
17966     tspec.it_interval.tv_sec = 1;
17967     tspec.it_interval.tv_nsec = 0;
17968
17969     if (timer_settime(mytimer, 0, &tspec, NULL) == -1)
17970         exit_sys("timer_settime");
17971
17972     if (setjmp(g_jb) == 1) {
17973         if (timer_delete(mytimer) == -1)
17974             exit_sys("timer_delete");
17975         exit(EXIT_SUCCESS);
17976     }
17977
17978     for (;;)
17979         pause();
17980
17981     return 0;
17982 }
17983
17984 void exit_sys(const char *msg)
17985 {
17986     perror(msg);
17987
17988     exit(EXIT_FAILURE);
17989 }
17990
17991 void signal_notification_handler(union sigval val)
17992 {
17993     static int count = 0;
17994
17995     if (count == 10)
17996         longjmp(g_jb, 1);
17997
17998     printf("signal notification function called: %d\n", val.sival_int);
17999
18000     ++count;
18001 }
18002
18003 /
*-----*
-----*
18004     setitimer fonksiytonu ile her cins timer'dan yalnızca bir tane      ↗
18005     yaratılabilmekteydi. Oysa bu modern timer mekanizmasında      ↗
18006     istenildiği kadar çok timer yaratılabilir. Ayrıca haberdar edilme      ↗
18007     yönteminin sinyal yoluyla yapılması durumunda girilen sinyal      ↗
18008     numarası realtime bir sinyal numarası olsa bile kuyruklama      ↗
18009     yapılmamaktadır. Yani örneğin periyot kısa olabilir. Bu süre      ↗
18010     içerisinde
18011     sinyal fonksiyonu sonlanmamış olabilir. Ancak yalnızca 1 tane sinyal      ↗
18012     pending durumda bekleyecektir. Fakat programcı isterse      ↗
18013     aslında kaç periyodun geçmiş olduğunu timer_getoverrun isimli      ↗
18014     fonksiyonla istediği zaman alabilir. Bu fonksiyon aynı biçimde      ↗
18015     fonksiyon yoluyla haberdar edilmeye de kullanılır.      ↗
18016 -----*
```

```
-----*/
18011
18012 / *
*-----*
-----*
18013 Bir grup prosesin oluşturduğu grubu "proses grubu" denilmektedir. Proses →
    grubu kavramı bir grup prosese sinyal gönderebilmek için →
18014 uydurulmuştur. Gerçekten de kill sistem fonksiyonunun birinci →
    parametresi olan pid sıfırdan bir küçük br sayı olarak girilirse →
18015 abs(pid) numaralı proses grubuna sinyal gönderilmektedir. Bir sinyal bir →
    prosese grubuna gönderilirse o proses grubunun bütün üyeleri olan →
18016 proseslere gönderilmiş olur. kill fonksiyonun birinci parametresi 0 →
    girilirse bu durumda sinyal kill fonksiyonunu uygulayan prosesin →
18017 proses grubuna gönderilir. Yani proses kendi proses grubuna sinyali →
    göndermektedir.
18018
18019 Bir proses grubunun id'si vardır. Proses gruplarının id'si o proses →
    grubundaki bir prosesin proses id'si ile aynıdır. İşte →
18020 proses id'si proses grub id'sine eşit olan prosese o proses grubunun →
    "proses grup lideri (process group leader)" denilmektedir.
18021 Proses grup lideri genellikle proses grubunu yaratan prosestir. Alt →
    prosesin proses grubu onu yaratan üst prosesten alınmaktadır.
18022
18023 Bir prosesin ilişkin olduğu proses grubunun id'sini alabilmek için →
    getpgrp ya da getpgid POSIX fonksiyonları kullanılır.
18024
18025 pid_t getpgrp(void);
18026 pid_t getpgid(pid_t pid);
18027
18028 getpgid fonksiyonu herhangi bir prosesin proses grup id'sini almakta →
    kullanılabilir. Bu fonksiyonun parametresi 0 geçilirse →
18029 fonksiyonu çağrıran prosesin proses grub id'si alınmış olur. Yani →
    aşağıdaki çağrı eşdeğерdir:
18030
18031 pgid = getpgrp();
18032 pgid = getpgid();
18033
18034 Proses grup lideri sonlanmış olsa bile proses grubunun id'si aynı →
    biçimde kalmaya devam eder. Proses grubu gruptaki son prosesin →
    sonlanması →
18035 ya da grup değiştirmesiyle ömrünü tamamlamaktadır.
18036
18037 Bir programı kabul üzerinden çalıştığımızda kabuk yeni bir proses →
    grubu oluşturur ve çalıştırılan programa ilişkin prosesi →
18038 o proses grubunun proses grub lideri yapar. Aşağıdaki programı →
    çalıştırıp sonucunu inceleyiniz. Yapılan denemeden söyle bir sonuç →
    elde edilmiştir:
18039
18040 Parent process id: 9921
18041 Parent process id of the parent: 1894
18042 Parent process group id: 9921
18043 Child process id: 9922
18044 Parent process id of the child: 9921
```

```
18045     Child process group id: 9921
18046
18047 ----- */
18048
18049 #include <stdio.h>
18050 #include <stdlib.h>
18051 #include <unistd.h>
18052 #include <sys/wait.h>
18053
18054 void exit_sys(const char *msg);
18055
18056 int main(void)
18057 {
18058     pid_t pgid;
18059     pid_t pid;
18060
18061     if ((pid = fork()) == -1)
18062         exit_sys("fork");
18063
18064     if (pid != 0) { /* parent process */
18065         printf("Parent process id: %ld\n", (long)getpid());
18066         printf("Parent process id of the parent: %ld\n", (long)getppid());
18067         pgid = getpgrp();
18068         printf("Parent process group id: %ld\n", (long)pgid);
18069
18070         if (waitpid(pid, NULL, 0) == -1)
18071             exit_sys("waitpid");
18072     }
18073     else { /* child process */
18074         sleep(1);
18075         printf("Child process id: %ld\n", (long)getpid());
18076         printf("Parent process id of the child: %ld\n", (long)getppid());
18077         pgid = getpgrp();
18078         printf("Child process group id: %ld\n", (long)pgid);
18079     }
18080
18081     return 0;
18082 }
18083
18084 void exit_sys(const char *msg)
18085 {
18086     perror(msg);
18087
18088     exit(EXIT_FAILURE);
18089 }
18090
18091 /
*----- */
----- */
18092 Yeni bir proses grubu yaratmak ya da bir prosesin proses grubunu
18093      değiştirmek için setpgid POSIX fonksiyonu kullanılmaktadır.
```

```
18094     int setpgid(pid_t pid, pid_t pgid);
18095
18096     Fonskiyonun birinci parametresi proses grup id'si değiştirilecek prosesi ➔
18097     ikinci parametresi de hedef proses grup id'sini belirtmektedir.
18098     Eğer bu iki parametre aynı ise yeni bir proses grubu yaratılır ve bu ➔
18099     yeni grubun lideri de buradaki proses olur. Bir proses ➔
18100     (root olsa bile) ancak kendisinin ya da kendi alt proseslerinin proses ➔
18101     grup id'lerini değiştirebilir. Ancak üst proses alt proses ➔
18102     exec uyguladıktan sonra onun proses grup id'sini artık ➔
18103     değiştirememektedir. setpgid fonksiyonu ile proses kendisinin ya da ➔
18104     alt proseslerinin ➔
18105     proses grup id'lerini aynı "oturum (session)" içerisindeki bir proses ➔
18106     grup id'si olarak değiştirebilmektedir.
18107
18108     Örneğin kabuk çalıştırduğumuz programı yeni proses grubunun grup lideri ➔
18109     şöyle yapmaktadır: Kabuk önce fork yapar. Üst ya da ➔
18110     alt prosesde setpgid fonskiyonuyla yeni proses grubunu oluşturup alt ➔
18111     prosesin bu grubun lideri olmasını sağlar.
18112     -----
18113     -----
18114     -----
18115     -----
18116     -----
18117     -----
18118     -----
18119     -----
18120     -----
18121     -----
18122     -----
18123     -----
18124     -----
18125     -----
18126     -----
18127     -----
18128     -----
18129     -----
18130     -----
18131     -----
18132     -----
18133     -----
18134     -----
18135     -----
18136     -----
18137     -----
```

18104 ----- */
18105
18106 #include <stdio.h>
18107 #include <stdlib.h>
18108 #include <unistd.h>
18109 #include <sys/wait.h>
18110
18111 void exit_sys(const char *msg);
18112
18113 int main(void)
18114 {
18115 pid_t pgid;
18116 pid_t pid;
18117
18118 if ((pid = fork()) == -1)
18119 exit_sys("fork");
18120
18121 if (pid != 0) { /* parent process */
18122 printf("Parent process id: %ld\n", (long)getpid());
18123 printf("Parent process id of the parent: %ld\n", (long)getppid());
18124 pgid = getpgrp();
18125 printf("Parent process group id: %ld\n", (long)pgid);
18126
18127 if (waitpid(pid, NULL, 0) == -1)
18128 exit_sys("waitpid");
18129 }
18130 else { /* child process */
18131 sleep(1);
18132 if (setpgid(getpid(), getpid() /* 0 */) == -1)
18133 exit_sys("setpgid");
18134
18135 printf("Child process id: %ld\n", (long)getpid());
18136 printf("Parent process id of the child: %ld\n", (long)getppid());
18137 pgid = getpgrp();

```
18138         printf("Child process group id: %ld\n", (long)pgid);
18139     }
18140
18141     return 0;
18142 }
18143
18144 void exit_sys(const char *msg)
18145 {
18146     perror(msg);
18147
18148     exit(EXIT_FAILURE);
18149 }
18150
18151 /
*-----*
-----*
18152     Yukarıda da belirtildiği gibi kill POSIX fonksiyonuyla (ya da kill      ↵
18153         komutıyla) bir proses grubuna sinyal gönderildiğinde      ↵
18154         aslında proses grubundaki tüm proseslere sinyal gönderilmektedir. Bunu      ↵
18155         aşağıdaki programla test edebilirsiniz.
18156     Başka bir terminalden girip önce üst ve alt proseslerin id'lerini      ↵
18157         aşağıdaki komut ile elde ediniz:
18158
18159     ps -a -o pid,ppid,pgid,cmd
18160
18161     Sonra da proses grubuna kill komutıyla SIGUSR1 sinyalini gönderiniz:
18162
18163     kill -USR1 <proses grup id>
18164
18165     Proses grup id'yi eksi değer olarak yazmayı unutmayın.
18166 -----*/
18167
18168 #include <stdio.h>
18169 #include <stdlib.h>
18170 #include <unistd.h>
18171 #include <sys/wait.h>
18172 #include <signal.h>
18173
18174 void exit_sys(const char *msg);
18175 void sigusr1_handler(int sno);
18176
18177 int main(void)
18178 {
18179     pid_t pid;
18180     struct sigaction sa;
18181
18182     sa.sa_handler = sigusr1_handler;
18183     sigemptyset(&sa.sa_mask);
18184     sa.sa_flags = 0;
18185
18186     if (sigaction(SIGUSR1, &sa, NULL) == -1)
18187         exit_sys("sigaction");
```

```
18185
18186     if ((pid = fork()) == -1)
18187         exit_sys("fork");
18188
18189     if (pid != 0) {      /* parent process */
18190         printf("parent waitint at pause\n");
18191         pause();
18192         if (waitpid(pid, NULL, 0) == -1)
18193             exit_sys("waitpid");
18194     }
18195     else {      /* child process */
18196         printf("child waitint at pause\n");
18197         pause();
18198     }
18199
18200     return 0;
18201 }
18202
18203 void exit_sys(const char *msg)
18204 {
18205     perror(msg);
18206
18207     exit(EXIT_FAILURE);
18208 }
18209
18210 void sigusr1_handler(int sno)
18211 {
18212     printf("SIGUSR1 occurred in process %ld\n", (long)getpid());
18213 }
18214
18215 /
*-----*
-----*
-----*
```

18216 Kabuk ile birden fazla programı boru eşliğinde çalıştırıldığımızı düşünelim. Örneğin:

18217 cat | grep "xxx"

18218

18219 Burada kabuk cat ve grep için fork yapar. Bu iki proses kardeşir. Ancak aralarında göstük-altılık ilişkisi yoktur.

18220 Kabuk bu iki prosesi aynı grubu sokar ve birinci prosesi de (burada cat) bu grubun grup lideri yapar. Bu komutu bir terminalden çalıştırıp diğer bir terminalde aşağıdaki komutla durumu gözleleyiniz:

18221 ps -a -o pid,ppid,pgid,cmd

18222

18223 İşte bu durumda Ctrl+C ve Ctrl + Delete tuşları SIGINTR ve SIGQUIT sinyallarını bu proses grubuna yollayacak böylece buradaki iki proses de (eğer sinyali sinyali işlememişlerse) sonlanacaktır.

18224

18225 -----*/

18226

18227

18228

18229

18230

```
18231 / *
-----*
18232     Oturum (session) arka plan çalışmayı düzene sokmak için uydurulmuş bir →
18233         kavramdır. Bir oturum proses gruplarından oluşur.
18234     Oturumu oluşturan proses gruplarından yalnızca biri "ön plan →
18235         (foreground)", diğerlerinin hepsi "arka plan (background)" →
18236         gruplardır. İşte aslında klavye sinyallerini terminal sürücüsü (tty) →
18237         oturumun ön plan proses grubuna yollamaktadır.
18238
18239     Kabuk üzerinden bir komut yazıp sonuna & karakteri getirilirse bu →
18240         karakter "bu komutu arka planda çalıştır" anlamına gelir.
18241     Böylece kabul komutu wait ile beklemez. Yeniden prompt'a düşer. Ancak o →
18242         komut çalışmaya devam etmektedir. İşte kabuk & ile
18243         çalıştırılan prosesleri oturumun arka plan prosesi durumuna getirir. →
18244         Sonunda & olmadan çalıştırılan prosesler ise ön plan proses
18245         grubunu oluşturur. Aslında kabuk da ön plan proses grubunun →
18246         içerisindeindedir. Dolayısıyla kabuğun kendisi de aynı oturumdadır.
18247
18248     O halde durum özetle şöyledir: Aslında kabuk bir oturum yaratıp kendini →
18249         oturumun lideri yapmıştır. Sonra kabuk sonu & ile
18250         biten komutlar için yarattığı proses gruplarını oturumun arka plan proses →
18251         grupları yapar. Sonunda & olmayan komutları da
18252         oturumun ön plan proses grubu yapmaktadır. Terminal sürücüsü de oturumun →
18253         ön plan proses grubuna SIGINT ve SIGQUIT sinyallerini
18254         göndermektedir.
18255
18256     Oturumların da proses gruplarında olduğu gibi id'leri vardır. →
18257         Oturumların id'leri oturum içerisindeki bir proses grubunun liderinin →
18258         id'si ile aynıdır. Oturum terminal sürücüsüyle ilişkili bir kavram →
18259         olarak uydurulmuştur. Dolayısıyla oturumların bir
18260         "ilişkin olduğu terminal (controlling terminal)" vardır. Bu terminal →
18261         gerçek terminal ise tty terminallerinde biridir. Sahte (Pseudo)
18262         bir terminal ise pts terminallerinden biridir. Pencere yöneticilerinin →
18263         içerisinde açılan terminaller sahte terminallerdir. Ancak
18264         işlev olarak gerçek terminallerden bir farkları yoktur. Klavyeden Ctrl+C →
18265         ve Ctrl+Backspace tuşlarına basıldığında SIGINTR ve SIGQUIT
18266         sinyalleri bu terminal tarafından (aslında terminal sürücüsü tarafından) →
18267         oturumun ön plan proses grubuna gönderilmektedir.
18268
18269 -----*/
18270
18271 /
-----*
18272
18273     Bir prosesin ilişkin olduğu otum id'si (session id) getsid POSIX →
18274         fonksiyonuna alınmaktadır:
18275
18276     pid_t getsid(pid_t pid);
18277
18278     Aşağıdaki programda bir prosesin ve onun alt prosesinin id bilgileri →
18279         ekrana yazdırılmıştır. Üst ve alt proseslerin aynı session id'ye
```

```
18261     sahip olduğuna onun da bash'in session id'si olduğuna dikkat ediniz.  
18262 -----*/  
18263  
18264 #include <stdio.h>  
18265 #include <stdlib.h>  
18266 #include <unistd.h>  
18267 #include <sys/wait.h>  
18268  
18269 void exit_sys(const char *msg);  
18270  
18271 int main(void)  
18272 {  
18273     pid_t pgid;  
18274     pid_t pid;  
18275  
18276     if ((pid = fork()) == -1)  
18277         exit_sys("fork");  
18278  
18279     if (pid != 0) { /* parent process */  
18280         printf("Parent process id: %ld\n", (long)getpid());  
18281         printf("Parent process id of the parent: %ld\n", (long)getppid());  
18282         pgid = getpgrp();  
18283         printf("Parent process group id: %ld\n", (long)pgid);  
18284         printf("Parent process session id: %ld\n", (long)getsid(0));  
18285  
18286         if (waitpid(pid, NULL, 0) == -1)  
18287             exit_sys("waitpid");  
18288     }  
18289     else { /* child process */  
18290         sleep(1);  
18291         if (setpgid(getpid(), getpid() /* 0 */ ) == -1)  
18292             exit_sys("setpgid");  
18293  
18294         printf("Child process id: %ld\n", (long)getpid());  
18295         printf("Parent process id of the child: %ld\n", (long)getppid());  
18296         pgid = getpgrp();  
18297         printf("Child process group id: %ld\n", (long)pgid);  
18298         printf("Child process session id: %ld\n", (long)getsid(0));  
18299     }  
18300  
18301     return 0;  
18302 }  
18303  
18304 void exit_sys(const char *msg)  
18305 {  
18306     perror(msg);  
18307  
18308     exit(EXIT_FAILURE);  
18309 }  
18310  
18311 /-----*/
```

```
-----  
18312     Yeni bir oturum (session) yaratmak için setsid fonksiyonu      ↵  
18313         kullanılmaktadır:  
18314     pid_t setsid(void);  
18315  
18316     Fonksiyon şunları yapar:  
18317  
18318     - Yeni bir oturum oluşturur  
18319     - Bu oturum içerisinde yeni bir proses grubu oluşturur.  
18320     - Oluşturulan oturumun ve proses grubunun lideri fonksiyonu çağrıran    ↵  
18321         prosesidir.  
18322     setsid fonksiyonu tipik olarak kabuk programları tarafından işin başında    ↵  
18323         çağrılmaktadır. Böylece kabuk yeni bir oturumun hem lideri olur  
18324         hem de o oturum içerisinde yaratılmış olan bir proses grubunun lideri    ↵  
18325         olur. O halde bir komut uygulanmamış durumda kabuk ortamında bir    ↵  
18326         oturum  
18327         ve bir de proses grubu vardır. Kabuk bu ikisinin de lideri durumundadır.    ↵  
18328         Sonra kabukta sonu & ile bitmeyen bir komut çalıştırıldığında kabuk  
18329         bu komuta ilişkin proses için yeni bir proses grubu yaratır ve bu grubu    ↵  
18330         oturumun ön plan proses grubu yapar.  
18331  
18332 -----*/  
18333 #include <stdio.h>  
18334 #include <stdlib.h>  
18335 #include <unistd.h>  
18336  
18337 void exit_sys(const char *msg);  
18338  
18339 int main(void)  
18340 {  
18341     if (setsid() == -1) /* function possibly will fail! */  
18342         exit_sys("setsid");  
18343  
18344     return 0;  
18345 }  
18346  
18347 void exit_sys(const char *msg)  
18348 {  
18349     perror(msg);  
18350  
18351     exit(EXIT_FAILURE);  
18352 }  
18353 /
```

```
*-----  
18354     Eğer yeni bir oturum yaratılmak isteniyorsa programın nasıl  
18355     çalıştırılacağı bilinmediğine göre önce fork uygulayıp alt prosese  
18356     setsid uygulamak gereklidir.  
18357  
18358 #include <stdio.h>  
18359 #include <stdlib.h>  
18360 #include <unistd.h>  
18361  
18362 void exit_sys(const char *msg);  
18363  
18364 int main(void)  
18365 {  
18366     pid_t pid;  
18367  
18368     if ((pid = fork()) == -1)  
18369         exit_sys("fork");  
18370  
18371     if (pid != 0)  
18372         exit(EXIT_SUCCESS);  
18373  
18374     if (setsid() == -1)      /* function possibly will fail! */  
18375         exit_sys("setsid");  
18376  
18377     printf("Ok, i am session leader of the new session!\n");  
18378  
18379     return 0;  
18380 }  
18381  
18382 void exit_sys(const char *msg)  
18383 {  
18384     perror(msg);  
18385  
18386     exit(EXIT_FAILURE);  
18387 }  
18388  
18389 /  
*-----  
18390 Oturum lideri open fonksiyonuyla O_NOCTTY bayrağı kullanılmadan bir  
18391 terminal aygıt sürücüsünü açtığında artık o terminal  
oturumun ilişkin olduğu terminal (controlling terminal) durumuna gelir.  
Eğer terminal o anda başka bir oturumun terminali ise oradan  
kopartılmaktadır.  
18392 open işlemini yapan prosese ise "terminali kontrol eden proses  
(controlling process)" denilmektedir. Normal olarak kabuk programı  
(bash)  
18393 terminali kontrol eden proses (controlling process) durumundadır.  
Dosyaların betimleyicileri üst proseden alt prosese aktarıldığına  
göre
```

18394 bu terminal betimleyicisi her proses de gözükecektir. Buna "ilgili prosese ilişkin terminal (process controlling terminal)" denilmektedir.

18395 Anımsanacağı gibi aslında 0 numaralı betimleyici terminal aygit sürücüsünün (controlling terminal) O_RDONLY modunda açılmasıyla,

18396 1 numaralı betimleyici aynı aygit sürücünün O_WRONLY moduyla açılmasıyla ve stderr de 1 numaralı betimleyinin dup yapılmasıyla oluşturulmaktadır.

18397 Yani aslında 0, 1 ve 2 betimleyiciler aynı dosyaya ilişkindir. Buna terminal ya da bu bağlamda "controlling terminal" denilmektedir.

18398 -----*/

18399 /

18400 *

18401 Oturumdaki ön plan proses grubunun hangisi olduğu tcgetpgrp POSIX fonksiyonuyla elde edilebilir. Oturumun ön plan proses grubu da tcsetpgrp POSIX fonksiyonuna değiştirilebilir.

18402

```
18404 pid_t tcgetpgrp(int fildes);
18405 int tcsetpgrp(int fildes, pid_t pgid_id);
```

18406

18407 -----*/

18408 /

18409 *

18410 Oturumun arka plan bir plan prosesi prosesin ilişkin olduğu terminalden (controlling terminal) okuma yapmak isterse terminal sürücüsü o arka plan prosesin içinde bulunduğu proses grubuna SIGTTIN sinyalini göndermektedir.

18411 Bu sinyalin default durumu prosesin durdurulmasıdır. Bu biçimde durdurulmuş olan prosesler SIGCONT sinyali ile yeniden çalıştırılmak istenebilirler.

18412 Ancak yeniden okuma yapılrsa yine proses durdurulur. Bu tür prosesler kabuk üzerinden fg komutuyla ön plana çekilebilir. Bu durumda kabuk önce prosesin proses grubunu ön plan proses grubu yapar sonra da onu SOGCONT sinyali ile uyardırır.

18413

18414 Aşağıdaki programı komut satırında sonuna & gerirerek çalıştırınız.

18415 Program 10 saniye sonra stdin dosyasından okuma yapmaya

18416 çalışacak ve bu nedenden dolayı SIGTTIN sinyali gönderilerek

18417 durdurulacaktır. Prosesin durdurulmuş olduğunu ps -l komutu ile ya da

18418 ps -o stat,cmd komutuyla gözleyiniz

18419 -----*/

18420

```
18421 #include <stdio.h>
18422 #include <stdlib.h>
18423 #include <unistd.h>
```

18424

```
18425 void exit_sys(const char *msg);
18426
18427 int main(void)
18428 {
18429     sleep(10);
18430     fgets(s, 100, stdin);
18431     puts(s);
18432
18433     return 0;
18434 }
18435
18436 void exit_sys(const char *msg)
18437 {
18438     perror(msg);
18439
18440     exit(EXIT_FAILURE);
18441 }
18442
18443 /
*-----*
-----  
18444     Arka plan proses grubundaki proseses SIGTTIN sinyalini işleyebilir. Bu
18445         durumda eğer yeniden başlatılabilir olmayan bir sistem fonksiyonu
18446         içerisinde bulunuluyorsa bu sistem fonksiyonu EINTR hata koduya geri
18447         döner.  
18448     Aşağıda programı sonuna & getirerek çalıştırıp log dosyasını
18449         inceleyiniz.  
18450 -----*/
18451
18452 #include <stdio.h>
18453 #include <stdlib.h>
18454 #include <errno.h>
18455 #include <unistd.h>
18456 #include <signal.h>
18457
18458 void exit_sys(const char *msg);
18459 void sigttin_handler(int sno);
18460
18461 FILE *g_f;
18462
18463 int main(void)
18464 {
18465     char s[100];
18466     struct sigaction sa;
18467
18468     if ((g_f = fopen("log.txt", "w")) == NULL)
18469         exit_sys("fopen");
18470
18471     sa.sa_handler = sigttin_handler;
18472     sigemptyset(&sa.sa_mask);
18473     sa.sa_flags = 0;
```

```
18472
18473     if (sigaction(SIGTTIN, &sa, NULL) == -1)
18474         exit_sys("sigaction");
18475
18476     sleep(10);
18477     if (fgets(s, 100, stdin) == NULL && errno == EINTR)
18478         fprintf(g_f, "gets terminated by signal!..\\n");
18479     puts(s);
18480
18481     return 0;
18482 }
18483
18484 void exit_sys(const char *msg)
18485 {
18486     perror(msg);
18487
18488     exit(EXIT_FAILURE);
18489 }
18490
18491 void sigttin_handler(int sno)
18492 {
18493     fprintf(g_f, "SIGTTIN occurred!..\\n");
18494 }
18495
18496 /
*-----*
```

18497 Arka plan proses grubundaki bir prosesin ilişkin terminale (controlling terminal) bir şeyler yazmaya çalışması da uygun değildir.

18498 Bu durumda da terminal sürücüsü prosesin ilişkin olduğu arka plan proses grubuna SIGTTOU sinyalini göndermektedir. Bu sinyalin default eylemi yine prosesin durdurulmasıdır. Ancak bu sinyalin aygit sürücüsü tarafından arka plan proses grubuna gönderilmesi için Linux'ta terminalin TOSTOP modunda olması gereklidir. Eğer terminal bu modda değilse proses durdurulmaz ancak yazılanlar da kaybolur.

18501
18502 Aşağıdaki programı sonuna & koyarak çalıştırmayı deneyiniz
18503 -----*/

```
18504
18505 #include <stdio.h>
18506 #include <stdlib.h>
18507 #include <unistd.h>
18508
18509 int main(void)
18510 {
18511     sleep(10);
18512
18513     printf("test\\n");
18514
18515     sleep(10);
18516
```

```
18517     return 0;
18518 }
18519
18520 /
*-----*
-----*
18521     Bir prosesin durudulması çizelgeden çıkartılıp bloke edilmesi anlamına ➔
18522         gelir. Ancak bunun normal bir blokeden farkı vardır.
18523     Bloke olma bir nedene dayalıdır. O neden ortadan kalkınca (ya da ➔
18524         sağlanınca) bloke sonlanır. Ancak prosesin durudulması ➔
18525         herhangi bir olayı beklemek biçiminde bi bloke değildir. Durdurulmuş ➔
18526         prosesler SIGCONT sinyali ile yeniden normal yaşamlarına ➔
18527         dönerler. Prosesi durdurmak için ona SIGSTOP sinyaliin gönderilmesi ➔
18528         gereklidir. SIGSTOP sinyali için sinyal fonksiyonu set edilemez ve ➔
18529         bu sinyal ignore da edilemez. Bu anlamda SIGSTOP sinyali SIGKILL ➔
18530             sinyaline benzemektedir. SIGSTOP sinyaline benzer diğer bir sinyale de ➔
18531             SIGTSTP sinyali denilmektedir. Bu sinyal terminal sürücüsü tarafından ➔
18532                 klavyeden Ctrl + Z tuşuna basıldığında ön plan porses ➔
18533                 grubuna gönderilmektedir. Bu sinyalin de -ismi üzerinde default eylemi ➔
18534                 proseslerin durdurulmasıdır. Ancak SIGTSTP sinyali ➔
18535                 ignore edilebilir ya da bu sinyal için sinyal fonksiyonu set edilebilir. ➔
18536                 (SIGSTOP sinyaliin ignore edilemediğini ve bunun için ➔
18537                     bir sinyal fonksiyonu set edilemediğine dikkat ediniz.)
18538
18539
18540
18541
18542     Aşağıdaki programı çalıştırıp Ctrl + Z tuşlarına basmayı deneyiniz.
18543
18544 -----*/
18545
18546 #include <stdio.h>
18547 #include <stdlib.h>
18548 #include <errno.h>
18549 #include <unistd.h>
18550 #include <signal.h>
18551
18552 void exit_sys(const char *msg);
18553 void sigtstop_handler(int sno);
18554
18555 int main(void)
18556 {
18557     int i;
18558     struct sigaction sa;
18559
18560     sa.sa_handler = sigtstop_handler;
18561     sigemptyset(&sa.sa_mask);
18562     sa.sa_flags = 0;
18563
18564     if (sigaction(SIGTSTP, &sa, NULL) == -1)
18565         exit_sys("sigaction");
18566
18567     for (i = 0;; ++i) {
18568         printf("%d\n", i);
```

```
18559         sleep(1);
18560     }
18561
18562     return 0;
18563 }
18564
18565 void exit_sys(const char *msg)
18566 {
18567     perror(msg);
18568
18569     exit(EXIT_FAILURE);
18570 }
18571
18572 void sigtstop_handler(int sno)
18573 {
18574     printf("SIGTSTOP occurred!..\n");
18575 }
18576
18577 /
*-----*
-----*
18578     read ve write fonksiyonları POSIX standartlarına göre sistem genelinde ↗
18579     atomiktir. Yani örneğin iki proses dosyanın aynı bölgесine ↗
18580     write ile yazma yaparsa kesinlikle iç içe geçme olmaz. Ya onun ya da ↗
18581     diğerinin yazdıklar gözükür. Benzer biçimde bir proses ↗
18582     dosyanın belli bir bölümüne yazma yaparken aynı anda başka bir proses ↗
18583     aynı bölgeden okumak yapmak isterse okuyan bozuk bir şeyi okumaz. ↗
18584     Ya önceki durumu okur ya da diğerinin tüm yazdıklarını okur. ↗
18585
18586     Ancak VTYS (Veritabanı Yönetim Sistemleri) gibi programlar birden fazla ↗
18587     read ya da write ile ilişkili okuma ve yazma yapabilmektedir. ↗
18588     Farklı read ve write çağrıları tabii ki atomik değildir. Örneğin bir ↗
18589     proses data dosyasına bir yazdıktan sonra ona ilişkin bazı bilgileri ↗
18590     indeks dosyasına yazıyor olabilir. Başka bir proses de aynı işlemi ↗
18591     yapıyor olabilir. Burada iç içe geçme olabilir. ↗
18592     O halde bu gibi ayrik okuma ve yazmaları eğer gerekiyorsa senkronize ↗
18593     etmek gereklidir. Senkronizasyon için bir mutex kullanmak ↗
18594     iyi bir tek nik değildir. Çünkü bütünsel bir kilitlemeye yol açar. ↗
18595     Halbuki dosya işlemlerinde yalnızca çıkışan offset aralıkları için ↗
18596     kilitleme yapılrsa performs artar. İşte bu nedenle zamanla UNIX türevi ↗
18597     sistemlere "dosya kilitleme (file locking)" mekanizmaları eklenmiştir. ↗
18598
18599     Dosya kilitleme bütünsel olarak ya da offset temelinde bölgesel olarak ↗
18600     (kayıtsal olarak) yapılmaktadır. Genel olarak bütünsel kilitleme ↗
18601     çoğu kez kötü bir tekniktir.
18602 -----*/
18603
18604 /
*-----*
-----*
18605     Dosmayı bütünsel kilitlemek için flock fonksiyonu kullanılabilir. Bu ↗
```

fonksiyon bir POSIX fonksiyonu değildir. Ancak Linux dahil olmak üzere pek çok UNIX türevi sistemde bulunmaktadır.

int flock(int fd, int operation);

Fonksiyonun birinci parametresi kilitlenecek dosyaya ilişkin betimleyicidir. İkincisi parametresi kilitleme biçimidir. Bu biçim şunlardan birisi olabilir:

LOCK_SH: Okuma için kilidi alma
LOCK_EX: Yazma için kilidi alma
LOCK_UN: Kilidi bırakma

İstanirse bunlardan birinin yanısı sıra LOCK_NB bayrağı da eklenebilir. Tıpkı okuma yazma kilitlerinde olduğu gibi birbirleriyle çelişen kilitleme blokeye yol açmaktadır.

Bu biçimdeki kilitler alt prosese de fork işlemi sırasında geçirilmektedir. Eğer kilit hiç açılmazsa kilidin ilişkin olduğu dosya ilişkin son betimleyici kapatıldığında kilit otomatik olarak açılır.

Dosyayı bütünsel kilitlemek yerine proseslerarası çalışan bir senkronizasyon nesnesiyle benzer bir etki yaratılabilir. Ancak toplamda böyle bir senkronizasyon nesnesi oluşturmanın kłod maliyeti flock fonksiyonundan çok daha fazla olmaktadır.

Dosya hangi modda açılırsa açılsın flock ile kilitleme yapılmaktadır.

-----*/

/

Offset temelinde kilitlenmede fcntl fonksiyonu kullanılmaktadır. Anımsanacağı gibi bu fonksiyon açılmış dosyaların birtakım özelliklerini değiştirmek için de kullanılmaktaydı. Fonksiyonun prototipi şöyledir:

int fcntl(int fd, int cmd, ...);

Offset temelinde kilitleme için kilitlenecek dosyayı temsil eden bir dosya betimleyicisi kullanılır (Birinci parametre). Fonksiyonun parametresine F_SETLK, F_SETLKW ya da F_GETLK değerlerinden biri girilir. Üçüncü parametresine ise her zaman flock isimli bir yapı nesnesinin adresi yerleştirilmelidir. Bu yapı şöyledir:

struct flock {

```
18632     short l_type;
18633     short l_whence;
18634     off_t l_start;
18635     off_t l_len;
18636     pid_t l_pid;
18637 };
18638
18639 Bu yapı nesnesinin elemanları pid dışında programcı tarafından      ↗
     doldurulur. l_type elemanı kilitlmenin cinsini belirtir.
18640 Bu elemana F_RDLCK, F_WRLCK, F_UNLCK değerlerinden biri girilmelidir.    ↗
     Yapının l_whence elemanı offset için orijini belirtmektedir.
18641 Bu elemana da SEEK_SET, SEEK_CUR ya da SEEK_END değerleri girilmelidir.    ↗
     l_start kilitlenecek bölgenden başlangıç offset'ini
18642 l_len ise uzunluğunu belirtmektedir. l_pid F_GETLK komutu tarafından      ↗
     doldulmaktadır.
18643
18644 l_len değeri 0 ise bu l_start değerinden itibaren dosyanın sonuna ve      ↗
     ötesine kadar kilitleme anlamına gelmektedir. (Bu durumda
18645 örneğin lseek ile dosya göstericisi EOF ötesine konumlandırılsa bile      ↗
     kilit geçerli olmaktadır.)
18646
18647 F_SETLK blokesiz F_SETLKW blokeli kilitleme yapmaktadır. Kilitlenmek      ↗
     istenen alan daha önce kilitlenmiş olan başka
18648 alanları kapsıyor ya da kesişiyor olabilir. Bu durumda çelişme tüm      ↗
     kapsanan ve kesişen alanlar dikkate alınarak belirlenmektedir.
18649
18650 F_SETLK ile kilitlemek isteyen proses eğer bu alan başka bir proses      ↗
     tarafından çelişki yaratacak biçimde kilitlenmişse fcntl -1
18651 değerine geri döner ve errno EACCESS ya da EAGAIN değeriyle set edilir.    ↗
     F_SETLKW zaten bu durumda bloke olmaktadır.
18652 F_GETLK komutu için de programının flock nesnesini oluşturmuş olması      ↗
     gereklidir. Bu durumda fcntl bu alanın istege bağlı biçimde
18653 kilitlenip kilitlenmeyeceğini bize söyler. Yani bu durumda fcntl      ↗
     kilitleme yapmaz ama sanki yapacakmış gibi duruma bakar.
18654 Eğer çelişki yoksa fcntl yalnızca yapının type elemanını F_UNLCK haline    ↗
     getirir. Eğer çelişki varsa bu çelişkiye yol açan kilit
18655 bilgilerini yapı nesnesinin içerisine doldurur. Fakat o alan birden      ↗
     fazla proses tarafından farklı biçimde kilitlenmişse bu durumda
18656 fcntl bu kilitlerin herhangi birinin bilgisini bize verecektir.
18657
18658 fcntl ile offset temelindeki kilitler fork işlemi sırasında alt prosese    ↗
     aktarılmazlar. Yani üst prosesin kilitlenmiş olduğu alanlar
18659 alt proses tarafından da kilitlenmiş gibi olmazlar. exec işlemleri      ↗
     sırasında offset temelindeki kilitlemeler varlığını devam
     ettirmektedir.
18660 Bir proses sonlandığında ya da o i-node elemanına ilişkin tüm      ↗
     betimleyiciler kapatıldığında prosesin ilgili dosyadaki kilitleri      ↗
     serbest bırakılır.
18661
18662 Prosesin dosyaya F_WRLCK koyması için dosyanın yazma modunda,      ↗
     F_RDLCK koyması ise dosyanın okuma modunda
18663 açılması gereklidir.
18664
```

```
18665 Aşağıdaki program offset temelinde kayıt kilitlemeyi betimlemek için →  
18666     yazılımıdır. Program çalıştırıldığında bir komut satırına düşülecek  
18667     ve komut satırında şu komutlar verilecektir.  
18668     <fcntl command code> <lock type> <starting offset> <length>  
18669  
18670     Programı kilitlemeye kullanılacak dosyanın yol ifadesini komut satırı →  
18671     argümanı biçiminde vererek çalıştırınız. Örneğin:  
18672     ./rlock test.txt  
18673  
18674     Tabii vereceğiniz dosyanın boş olmaması gereklidir. Örnek komutlar →  
18675     söylenir:  
18676     Örneğin:  
18677  
18678     CSD (32567): F_SETLK F_WRLCK 0 64  
18679     CSD (32767): F_SETLK F_UNLCK 0 64  
18680  
18681     Command kodları şunlardır: F_SETLK, F_SETLKW, F_GETLK  
18682  
18683     lock türleri de şunlardır: F_RDLCK, F_WRLCK, F_UNLCK  
18684 -----*/  
18685  
18686 #include <stdio.h>  
18687 #include <stdlib.h>  
18688 #include <string.h>  
18689 #include <errno.h>  
18690 #include <fcntl.h>  
18691 #include <unistd.h>  
18692  
18693 #define MAX_CMDLINE      4096  
18694 #define MAX_ARGS          64  
18695  
18696 void parse_cmd(void);  
18697 int get_cmd(struct flock *fl);  
18698 void disp_flock(const struct flock *fl);  
18699 void exit_sys(const char *msg);  
18700  
18701 char g_cmd[MAX_CMDLINE];  
18702 int g_count;  
18703 char *g_args[MAX_ARGS];  
18704  
18705 int main(int argc, char *argv[])  
18706 {  
18707     int fd;  
18708     pid_t pid;  
18709     char *str;  
18710     struct flock fl;  
18711     int fcntl_cmd;  
18712  
18713     if (argc != 2) {
```

```
18714         fprintf(stderr, "wrong number of arguments!..\\n");
18715         exit(EXIT_FAILURE);
18716     }
18717
18718     pid = getpid();
18719
18720     if ((fd = open(argv[1], O_RDWR)) == -1)
18721         exit_sys("open");
18722
18723     for (;;) {
18724         printf("CSD (%ld)>", (long)pid), fflush(stdout);
18725         fgets(g_cmd, MAX_CMDLINE, stdin);
18726         if ((str = strchr(g_cmd, '\\n')) != NULL)
18727             *str = '\\0';
18728         parse_cmd();
18729         if (g_count == 0)
18730             continue;
18731         if (g_count == 1 && !strcmp(g_args[0], "quit"))
18732             break;
18733         if (g_count != 4) {
18734             printf("invalid command!\\n");
18735             continue;
18736         }
18737
18738         if ((fcntl_cmd = get_cmd(&f1)) == -1) {
18739             printf("invalid command!\\n");
18740             continue;
18741         }
18742
18743         if (fcntl(fd, fcntl_cmd, &f1) == -1)
18744             if (errno == EACCES || errno == EAGAIN)
18745                 printf("Locked failed!..\\n");
18746             else
18747                 exit_sys("fcntl");
18748             if (fcntl_cmd == F_GETLK)
18749                 disp_flock(&f1);
18750         }
18751
18752         close(fd);
18753
18754         return 0;
18755     }
18756
18757     void parse_cmd(void)
18758     {
18759         char *str;
18760
18761         g_count = 0;
18762         for (str = strtok(g_cmd, " \\t"); str != NULL; str = strtok(NULL, " \\t")) ↵
18763             g_args[g_count++] = str;
18764     }
18765
```

```
18766 int get_cmd(struct flock *fl)
18767 {
18768     int cmd, type;
18769
18770     if (!strcmp(g_args[0], "F_SETLK"))
18771         cmd = F_SETLK;
18772     else if (!strcmp(g_args[0], "F_SETLKW"))
18773         cmd = F_SETLKW;
18774     else if (!strcmp(g_args[0], "F_GETLK"))
18775         cmd = F_GETLK;
18776     else
18777         return -1;
18778
18779     if (!strcmp(g_args[1], "F_RDLCK"))
18780         type = F_RDLCK;
18781     else if (!strcmp(g_args[1], "F_WRLCK"))
18782         type = F_WRLCK;
18783     else if (!strcmp(g_args[1], "F_UNLCK"))
18784         type = F_UNLCK;
18785     else
18786         return -1;
18787
18788     fl->l_type = type;
18789     fl->l_whence = SEEK_SET;
18790     fl->l_start = (off_t)strtol(g_args[2], NULL, 10);
18791     fl->l_len = (off_t)strtol(g_args[3], NULL, 10);
18792
18793     return cmd;
18794 }
18795
18796 void disp_flock(const struct flock *fl)
18797 {
18798     switch (fl->l_type) {
18799         case F_RDLCK:
18800             printf("Read Lock\n");
18801             break;
18802         case F_WRLCK:
18803             printf("Write Lock\n");
18804             break;
18805         case F_UNLCK:
18806             printf("Unlocked (can be locked)\n");
18807     }
18808
18809     printf("Whence: %d\n", fl->l_whence);
18810     printf("Start: %ld\n", (long)fl->l_start);
18811     printf("Length: %ld\n", (long)fl->l_len);
18812     if (fl->l_type != F_UNLCK)
18813         printf("Process Id: %ld\n", (long)fl->l_pid);
18814 }
18815
18816 void exit_sys(const char *msg)
18817 {
18818     perror(msg);
```

```
18819     exit(EXIT_FAILURE);
18820 }
18821 }
18822 /
18823 *
-----*
18824     Eğer çok sayıda lock işlemi yapılacaksa fcntl çağrısını yapan bir satma →
fonksiyon kullanılabilir.
18825 -----*/
18826
18827 int setlock_wrapper(int fd, int type, int whence, off_t start, off_t len)
18828 {
18829     struct flock fl;
18830
18831     fl.l_type = type;
18832     fl.l_whence = whence;
18833     fl.l_start = start;
18834     fl.l_len = len;
18835
18836     return fcntl(fd, F_SETLK, &fl);
18837 }
18838
18839 int getlock_wrapper(int fd, int type, int whence, off_t start, off_t len)
18840 {
18841     if (fcntl(fd, F_GETLK, &fl) == -1)
18842         return -1;
18843
18844     return fl.l_type;
18845 }
18846
18847 /
18848 *
-----*
18849     Yukarıdaki sarma fonksiyonlarının bir benzeri POSIX standartlarında lockf →
(flock ile karıştırılmayınız) ismiyle bulunmaktadır.
18850
18851     int lockf(int fildes, int function, off_t size);
18852
18853     Bu fonksiyon her zaman dosya göstericisinin gösterdiği yerden itibaren →
size kadar byte'ı kilitler. size değeri negatif olabilir.
18854     İkinci parametre olan function şunlardan biri olabilir:
18855
18856     F_ULOCK
18857     F_LOCK
18858     F_TLOCK
18859     F_TEST
18860
18861     F_ULOCK "unlock" anlamındadır. F_LOCK kilitleme yapmaya çalışır. Eğer →
çelişkili bir durum varsa blokeye yol açar. F_TLOCK
18862     ise çelişki durumda bloke oluşturmayıp -1 ile geri döner. errno değeri →
```

de yine EACCES ya da EAGAIN değeri ile set edilir.

18863 F_TEST kilitleme yapmaz ancak durumu test eder. Eğer test çelişkili ise ↵ fonksiyon -1'e geri döüp errno değerini EACCES ya da

18864 EAGAIN değerine set etmektedir.

18865

18866 -----*/

18867 /

18868 *

18869

18870 Biz yukarıda "tavsiye niteliğinde (advisory)" kilitleme yaptık. Burada "tavsiye niteliğinde (advisory)" demekle şu anlatılmak

18871 istenmektedir: Bir proses fcntl (ya da lockf ya da flock ile) kilitleme yaptığında aslında başka bir proses isterse read ve write

18872 fonksiyonlarıyla istediği zaman okuma yazma yapabilmektedir. Yani read ve

18873 write fonksiyonları ilgili alanın kilitli olup olmadığına baktırmaktadır. Tabii programcı tüm proseslerin kodlarını kendisi yazdığı için (ya da bir protokol eşliğinde başkaları da yazmış olabilir)

18874 her zaman okuma yazma işlemlerinde öncelikle kilide bakçak biçimde bir strateji izler. İşte kilitlemenin diğer bir türüne de

18875 "zorlamalı kilitleme (mandatory locking)" denilmektedir. Artık bu kilitleme biçiminde başka bir proses kilitlenmiş alanı çelişki

18876 durumunda istede de read ve write ile okuyup yazamamaktadır.

18877

18878 Zorlamalı kilitleme POSIX standartlarının bulunmamaktadır. POSIZ standartları ↵ fcntl fonksiyonun açıklamasında bunun gerekliliklerini

18879 belirtmiştir. Bazı işletim sistemleri bunu hiç desteklememektedir. Linux 2.4 ↵ çekirdeği ile birlikte zorlamalı dosya kilitlemesini destekler

18880 hale gelmiştir. Zorlamalı kilitleme için aslında fcntl fonksiyonunda ek bir ↵ şey yapılmaz. Linux'ta zorlamalı kilitleme için şu koşulların sağlanmış

18881 olması gerekmektedir:

18882

18883 1) Dosyanın içinde bulunduğu dosya sisteminin -o mand ile mount edilmiş olması gereklidir. (remount -o mand,remount <device> <mount point>)

18884 2) İlgili dosyanın setg-group id bayrağı set edilip x hakkı varsa reset edilmelidir. (chmod <dosya> g+s g-x)

18885

18886 Zorlamalı kilitlemede çekirdek her read/write işleminde kilide denk gelinmiş mi diye kontroller yapmaktadır. Bu da çalışmayı yavaşlatmaktadır.

18887 Ayrıca zorlamalı kilitleme kötüyekullanıma da açıktır.

18888

18889 Aşağıda zorlamalı kilitlemeyi test etmek için iki program verilmiştir. Programlardan ilki belli bir dosyanın belli bir yerinden

18890 n byte okuma yazma yapar. Örneğin:

18891

18892 ./mlocktest test.txt r 0 64

18893 ./mlocktest test.txt w 30 50

18894

18895 Yazma sırasında hep 0'lar yazılır.

18896

18897 Diğer program yukarıdaki rlock.c programıdır.

```
18898
18899 Bu denemeyi yapmadan önce şu işlemleri uygulayınız:
18900
18901 sudo mount -o remount,mand /dev/sda2 /
18902 chmod g+s-x test.txt
18903
18904 Sonra programları farklı terminallerde dosya ismi vererek (burada test.txt) ➔
   çalıştırınız.
18905 -----
-----*/
18906
18907 /* mlocktest.c */
18908
18909 #include <stdio.h>
18910 #include <stdlib.h>
18911 #include <string.h>
18912 #include <fcntl.h>
18913 #include <unistd.h>
18914
18915 void exit_sys(const char *msg);
18916
18917 /* ./mlocktest <dosya ismi> <r/w> <offset> <uzunluk> */
18918
18919 int main(int argc, char *argv[])
18920 {
18921     int fd;
18922     int operation;
18923     off_t offset;
18924     off_t len;
18925     char *buf;
18926     ssize_t result;
18927
18928     if (argc != 5) {
18929         fprintf(stderr, "wrong number of arguments!..\n");
18930         exit(EXIT_FAILURE);
18931     }
18932
18933     if (strcmp(argv[2], "r") && strcmp(argv[2], "w")) {
18934         fprintf(stderr, "invalid operation!\n");
18935         exit(EXIT_FAILURE);
18936     }
18937
18938     offset = (off_t)strtol(argv[3], NULL, 10);
18939     len = (off_t)strtol(argv[4], NULL, 10);
18940
18941     if ((buf = (char *)calloc(len, 1)) == NULL) {
18942         fprintf(stderr, "cannot allocate memory!..\n");
18943         exit(EXIT_FAILURE);
18944     }
18945
18946     if ((fd = open(argv[1], argv[2][0] == 'r' ? O_RDONLY : O_WRONLY)) == -1)
18947         exit_sys("open");
18948
```

```
18949     lseek(fd, 0, offset);
18950     if (argv[2][0] == 'r') {
18951         if ((result = read(fd, buf, len)) == -1)
18952             exit_sys("read");
18953         printf("%ld bytes read\n", (long)result);
18954     }
18955     else {
18956         if ((result = write(fd, buf, len)) == -1)
18957             exit_sys("write");
18958         printf("%ld bytes written\n", (long)result);
18959     }
18960
18961     free(buf);
18962     close(fd);
18963
18964     return 0;
18965 }
18966
18967 void exit_sys(const char *msg)
18968 {
18969     perror(msg);
18970
18971     exit(EXIT_FAILURE);
18972 }
18973
18974 /* rlock.c */
18975
18976 #include <stdio.h>
18977 #include <stdlib.h>
18978 #include <string.h>
18979 #include <errno.h>
18980 #include <fcntl.h>
18981 #include <unistd.h>
18982
18983 #define MAX_CMDLINE      4096
18984 #define MAX_ARGS          64
18985
18986 void parse_cmd(void);
18987 int get_cmd(struct flock *fl);
18988 void disp_flock(const struct flock *fl);
18989 void exit_sys(const char *msg);
18990
18991 char g_cmd[MAX_CMDLINE];
18992 int g_count;
18993 char *g_args[MAX_ARGS];
18994
18995 int main(int argc, char *argv[])
18996 {
18997     int fd;
18998     pid_t pid;
18999     char *str;
19000     struct flock fl;
19001     int fcntl_cmd;
```

```
19002
19003     if (argc != 2) {
19004         fprintf(stderr, "wrong number of arguments!..\\n");
19005         exit(EXIT_FAILURE);
19006     }
19007
19008     pid = getpid();
19009
19010     if ((fd = open(argv[1], O_RDWR)) == -1)
19011         exit_sys("open");
19012
19013     for (;;) {
19014         printf("CSD (%ld)>", (long)pid), fflush(stdout);
19015         fgets(g_cmd, MAX_CMDLINE, stdin);
19016         if ((str = strchr(g_cmd, '\\n')) != NULL)
19017             *str = '\\0';
19018         parse_cmd();
19019         if (g_count == 0)
19020             continue;
19021         if (g_count == 1 && !strcmp(g_args[0], "quit"))
19022             break;
19023         if (g_count != 4) {
19024             printf("invalid command!\\n");
19025             continue;
19026         }
19027
19028         if ((fcntl_cmd = get_cmd(&f1)) == -1) {
19029             printf("invalid command!\\n");
19030             continue;
19031         }
19032
19033         if (fcntl(fd, fcntl_cmd, &f1) == -1)
19034             if (errno == EACCES || errno == EAGAIN)
19035                 printf("Locked failed!..\\n");
19036             else
19037                 exit_sys("fcntl");
19038             if (fcntl_cmd == F_GETLK)
19039                 disp_flock(&f1);
19040     }
19041
19042     close(fd);
19043
19044     return 0;
19045 }
19046
19047 void parse_cmd(void)
19048 {
19049     char *str;
19050
19051     g_count = 0;
19052     for (str = strtok(g_cmd, " \\t"); str != NULL; str = strtok(NULL, " \\t")) ↵
19053         g_args[g_count++] = str;
```

```
19054 }
19055
19056 int get_cmd(struct flock *fl)
19057 {
19058     int cmd, type;
19059
19060     if (!strcmp(g_args[0], "F_SETLK"))
19061         cmd = F_SETLK;
19062     else if (!strcmp(g_args[0], "F_SETLKW"))
19063         cmd = F_SETLKW;
19064     else if (!strcmp(g_args[0], "F_GETLK"))
19065         cmd = F_GETLK;
19066     else
19067         return -1;
19068
19069     if (!strcmp(g_args[1], "F_RDLCK"))
19070         type = F_RDLCK;
19071     else if (!strcmp(g_args[1], "F_WRLCK"))
19072         type = F_WRLCK;
19073     else if (!strcmp(g_args[1], "F_UNLCK"))
19074         type = F_UNLCK;
19075     else
19076         return -1;
19077
19078     fl->l_type = type;
19079     fl->l_whence = SEEK_SET;
19080     fl->l_start = (off_t)strtol(g_args[2], NULL, 10);
19081     fl->l_len = (off_t)strtol(g_args[3], NULL, 10);
19082
19083     return cmd;
19084 }
19085
19086 void disp_flock(const struct flock *fl)
19087 {
19088     switch (fl->l_type) {
19089         case F_RDLCK:
19090             printf("Read Lock\n");
19091             break;
19092         case F_WRLCK:
19093             printf("Write Lock\n");
19094             break;
19095         case F_UNLCK:
19096             printf("Unlocked (can be locked)\n");
19097     }
19098
19099     printf("Whence: %d\n", fl->l_whence);
19100     printf("Start: %ld\n", (long)fl->l_start);
19101     printf("Length: %ld\n", (long)fl->l_len);
19102     if (fl->l_type != F_UNLCK)
19103         printf("Process Id: %ld\n", (long)fl->l_pid);
19104 }
19105
19106 void exit_sys(const char *msg)
```

```
19107  {
19108      perror(msg);
19109
19110      exit(EXIT_FAILURE);
19111 }
19112
19113 int setlock_wrapper(int fd, int type, int whence, off_t start, off_t len)
19114 {
19115     struct flock fl;
19116
19117     fl.l_type = type;
19118     fl.l_whence = whence;
19119     fl.l_start = start;
19120     fl.l_len = len;
19121
19122     return fcntl(fd, F_SETLK, &fl);
19123 }
19124
19125 int getlock_wrapper(int fd, int type, int whence, off_t start, off_t len)
19126 {
19127     if (fcntl(fd, F_GETLK, &fl) == -1)
19128         return -1;
19129
19130     return fl.l_type;
19131 }
19132
19133 /
*-----*
-----*
19134     Linux sistemlerinde sistemdeki tüm kilitler /proc/locks isimli dosyay    ↗
     çekirdek tarafından yazılmaktadır. Örneğin:    ↗
19135
19136 1: POSIX ADVISORY WRITE 3515 08:02:393260 0 63
19137 2: POSIX ADVISORY READ 2339 08:02:786493 128 128
19138 3: POSIX ADVISORY READ 2339 08:02:786491 1073741826 1073742335
19139 4: FLOCK ADVISORY WRITE 770 00:17:614 0 EOF
19140
19141 Burada birinci sütun kildinin flock fonksiyonuyla mı yoksa fcntl    ↗
     fonksiyonuyla mı oluşturulduğunu belirtir. İkinci sütun
19142 kildinin isteğe bağlı mı yoksa zorlamalı mı olduğunu belirtmektedir. Üçüncü    ↗
     sütun kildinin yazmayı mı okumayı mı kilitlediğini
19143 belirtmektedir. Sonraki sütun kilitleyen prosesin id'sini sonraki sütun ise    ↗
     kilitlenmiş dosyanın dosya sistemindeki, aygit
19144 numarasını ve i-node numarasını belirtmektedir. Nihayet son sütun    ↗
     kilitlemein offset aralığını belirtmektedir.
19145 -----
*-----*/    ↗
-----*
19146
19147 /
*-----*
-----*
19148     POSIX sistemlerinde ileri IO işlemleri 4 bölüme ayrılarak incelenebilir:
19149
```

19150 1) Multiplexed IO: Bir grup betimleyici izlemeye alınır. Bu ↵
19151 betimleyicilerde ilgilenilen olay (read/write/error) yoksa blokede ↵
beklenir. Ta ki bu betimleyicilerden en az birinde ilgilenilen olay ↵
19152 gerçekleşene kadar. Multiplexed IO için select ve poll POSIX ↵
fonksiyonları kullanılmaktadır. Ancak Linux epoll isimli daha yetenekli ↵
bir fonksiyona da sahiptir.

19153
19154 2) Signal Driven IO: Burada belli betimleyiciler izlemeye alınır. Ancak ↵
19155 blokede beklenmez. Bu betimleyicilerde olay gerçekleştiğinde ↵
SIGIO isimli sinyal oluşur. Programcı da bu sinyal oluşturulduğunda blokeye ↵
maruz kalmadan read/write yapılabilir.

19156
19157 3) Asenkron IO: Burada read/write işlemleri başlatılır. Ancak bir bloke ↵
19158 oluşmaz. Arka planda çekirdek tarafından okuma ve yazma ↵
bir yandan yapılır. Ancak aktarım bittiğinde programcı haberdar edilir. ↵
Bunun signal driven IO'dan farkı şudur: Signal driven IO'da ↵
19159 aktarım yapılmamaktadır. Yalnızca okuma yazma yailrsa bloke ↵
olunmayacağı prosese söylenmektedir. Halbuki asenkron IO'da okuma ve ↵
yazma

19160 işlemi bloke çözüldüğünde arka planda gerçekleştirilmekte ve yalnızca ↵
işlemin bittiğ haber verilmektedir.

19161
19162 4) Scatter-Gather IO: Burada okuma birden fazla adrese yazma ise birden ↵
fazla adresten kaynağa yapılmaktadır.

19163 -----*/
19164
19165
19166 /
*-----

19167 Multiplexed IO blokeye yol açabilecek aynı anda birden fazla dosya ↵
betimleyici ile işlem yapmayı anlatan bir IO terimidir.
19168 Multiplexed IO için select, pselect, poll ve epoll fonksiyonları ↵
kullanılmaktadır.

19169
19170 select fonksiyonu ilk kez BSD sistemlerinde denenmiştir. Sonra POSIX ↵
standartlarına sokulmuştur. Fonksiyonun prototipi şöyledir:

19171
19172 int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set ↵
*errorfds, struct timeval *timeout);

19173
19174 Fonksiyonun birinci parametresi ilgilenilecek betimleyicilerin en büyük ↵
değerini almaktadır. En yüksek betimleyici numarası FD_SETSIZE ↵
19175 ile belirtilmiştir. Bu parametre doğrudan FD_SETSIZE (1024) olarak ↵
girilebilir. Ancak bu parametrenin uygun girilmesi bir hız kazancı ↵
19176 sağlayabilmektedir. Fonksiyon parametrelerindeki fd_set türü bir bit ↵
dizisi belirtmektedir. Bu bit dizisinin belli bitini 1 yapmak için ↵
19177 FD_SET, belli bitini 0 yapmak için FD_CLR, tüm bitlerini sıfır yapmak ↵
için FD_ZERO, belli bitini test etmek için ise FD_ISSET makroları ↵
19178 kullanılmaktadır. Biz fonksiyona üç tane fd_set nesnesi veririz. ↵
Bunlardan ilki "okuma", ikinci "yazma" üçüncü "error" kontrolü ↵
yaptırmak

19179 içindir. Fonksiyon son parametresi "zaman aşımı" belirtmektedir. Bu →
19180 parametreler için NULL adres girilebilir. Zaman aşımı için NULL adres →
girilirse zaman aşımının kullanılmayacağı anlaşılır. Eğer zaman aşımı →
için yapının her iki elemanı 0'da girilebilir. Budurumda select →
fonksiyonu →
19181 hemen testini yapar ve geri döner.
19182
19183 select fonksiyonu okuma takibi için tipik olarak şöyle kullanılır: →
 Progtamcı okuma izlemesi yapılacak betimleyicilerin numaralarına →
 ilişkin bitleri →
19184 bir fd_set içerisinde set eder. Bunu fonksyonun ikinci parametresine →
 verir. Fonksyon da yalnızca 1 olan bitlere ilişkin betimleyicileri →
izleyecktir. Programcı bu betimleyicilerden okuma yapılamayacak durumda →
 ise (yani okuma girişiminde bloke oluşacak durumda ise) select →
 blokede →
19185 akışın bekletir. En az bir betimleyicide okuma eylemi yapılabilecek →
 durumdaysa bloke çözülür. Fonksyon hangi betimleyicilerde okuma →
 işlemının yapılabileceğini →
19186 yine bizim ona verdığımız fd_set nesnesinin ilgili 1 yaparak bize →
 iletmektedir. Yani bizim fonksiyona verdığımız fd_set nesni →
 fonksiyon geri döndüğünde →
19187 artık bozmuş durumdadır yani hangi betimleyicilerin okumaya elvirişli →
 olduğunu gösterecek biçimde değiştirilmiş durumdadır. Buradaki aynı →
 çalışma →
19188 biçimi "yazma" ve "error" işlemi için de aynı biçimde söz konusudur. →
 Programcı aynı zamanda isterse üç ayrı fd_set nesnesini de fonksiyona →
 verebilir.
19189
19190
19191 select fonksiyon başarısızlık durumunda -1 değerine geri döner. Eğer →
 hiçbir betimleyicide olay gerçekleşmemiş ancak zaman aşımı dolmuşsa 0 →
 değerine geri döner.
19192 Eğer en az bir betimleyicide ilgili olay gerçekleşmişse toplam olay →
 gerçekleşen betimleyici sayısına geri döner. (Aynı betimleyici örneğin →
 hen okuma hem de yazma →
19193 için izleniyorsa ve bu betimleyicide hem okuma hem de yazma olayı →
 gerçekleşmişse bu değer 2 artırmaktadır.) →
19194
19195 select fonksyonun normal disk dosyaları için kullanılması anlamsızdır. →
 Uzun süre beklemeye yol açabilecek terminal gibi, boru gibi, soket →
 gibi aygıtlar için →
19196 kullanılmalıdır. Normal olarak select ile bekelenenek betimleyicilere →
 ilişkin kaynaklar "blokeli" modda açılmalıdır.
19197
19198 Aşağıdaki örnekte 0 numaralı betimleyici select ile izlenmiştir. Bu →
 betimleyici terminal sürücüne ilişkindir. Kullanıcı bir yazı girip →
 ENTER →
19199 tuşuna bastığında terminal sürücüsü select fonksyonun blokesini →
 çözecktir.
19200 -----*/ →
19201
19202 #include <stdio.h>
19203 #include <stdlib.h>

```
19204 #include <string.h>
19205 #include <unistd.h>
19206 #include <sys/select.h>
19207
19208 void exit_sys(const char *msg);
19209
19210 int main(int argc, char *argv[])
19211 {
19212     fd_set rset;
19213     int result;
19214     char buf[1024 + 1];
19215     ssize_t n;
19216
19217     FD_ZERO(&rset);
19218     FD_SET(0, &rset);
19219
19220     if ((result = select(1, &rset, NULL, NULL, NULL)) == -1)
19221         exit_sys("select");
19222
19223     printf("result = %d\n", result);
19224     if ((n = read(0, buf, 1024)) == -1)
19225         exit_sys("read");
19226     buf[n] = '\0';
19227     printf("%s", buf);
19228
19229     return 0;
19230 }
19231
19232 void exit_sys(const char *msg)
19233 {
19234     perror(msg);
19235
19236     exit(EXIT_FAILURE);
19237 }
19238
19239 /
*-----*
-----*
-----*
-----*
```

19240 Yukarıdaki programa 5 saniyelik bir zaman aşımı ekleyelim.

19241 -----*/

```
19242
19243 #include <stdio.h>
19244 #include <stdlib.h>
19245 #include <string.h>
19246 #include <unistd.h>
19247 #include <sys/select.h>
19248
19249 void exit_sys(const char *msg);
19250
19251 int main(int argc, char *argv[])
19252 {
19253     fd_set rset;
```

```
19254     int result;
19255     char buf[1024 + 1];
19256     ssize_t n;
19257     struct timeval tv;
19258
19259     FD_ZERO(&rset);
19260     FD_SET(0, &rset);
19261
19262     tv.tv_sec = 5;
19263     tv.tv_usec = 0;
19264
19265     if ((result = select(1, &rset, NULL, NULL, &tv)) == -1)
19266         exit_sys("select");
19267
19268     if (result == 0)
19269         printf("Timeout!\n");
19270     else {
19271         printf("result = %d\n", result);
19272         if ((n = read(0, buf, 1024)) == -1)
19273             exit_sys("read");
19274         buf[n] = '\0';
19275         printf("%s", buf);
19276     }
19277
19278     return 0;
19279 }
19280
19281 void exit_sys(const char *msg)
19282 {
19283     perror(msg);
19284
19285     exit(EXIT_FAILURE);
19286 }
19287
19288 /
*-----*
```

19289 POSIX standartları zaman aşımı için verdığımız timeval yapı nesnesinin fonksiyon tarafından güncellenip güncellenmeyeceğini
19290 isteğe bağlı olarak sisteme bırakmıştır. Linux'ta select sonlanmadan önce bu yapıya kalan zaman miktarı yerleştirilir.

19291 -----*/

```
19292
19293 #include <stdio.h>
19294 #include <stdlib.h>
19295 #include <string.h>
19296 #include <unistd.h>
19297 #include <sys/select.h>
19298
19299 void exit_sys(const char *msg);
19300
19301 int main(int argc, char *argv[])
```

```
19302 {
19303     fd_set rset;
19304     int result;
19305     char buf[1024 + 1];
19306     ssize_t n;
19307     struct timeval tv;
19308
19309     FD_ZERO(&rset);
19310     FD_SET(0, &rset);
19311
19312     tv.tv_sec = 10;
19313     tv.tv_usec = 0;
19314
19315     if ((result = select(1, &rset, NULL, NULL, &tv)) == -1)
19316         exit_sys("select");
19317
19318     if (result == 0)
19319         printf("Timeout!\n");
19320     else {
19321         printf("result = %d\n", result);
19322         if ((n = read(0, buf, 1024)) == -1)
19323             exit_sys("read");
19324         buf[n] = '\0';
19325         printf("%s", buf);
19326         printf("Remaining time: %ld.%03ld\n", (long)tv.tv_sec, (long)
19327             tv.tv_usec / 1000);
19328     }
19329     return 0;
19330 }
19331
19332 void exit_sys(const char *msg)
19333 {
19334     perror(msg);
19335
19336     exit(EXIT_FAILURE);
19337 }
19338
19339 /
*-----*
-----*
19340     select ile birden fazla betimleyiciyi izlerken bloke çözüldüğünde
19341         programcının hangi betimleyiciler dolayısıyla blokenin
19342             çözüldüğünü belirlemesi gereklidir. Bunun için FD_ISSET makrosu ile izlenen
19343                 tüm betimleyiciler kontrol edilmelidir. Aslında
19344                 bu kontrol yapılrken 0'dan FD_SETSIZE değerine kadar bile bir döngü
19345                     kullanılabilir. Naıl olsa izlenmeyen tüm betimleyicilerin
19346                     bitleri 0 olacaktır. Tabii programcı döngüyü kısalmak için eğer
19347                         betimleyicilerin numaralarını bir yerde saklamışsa yalnızca onu da
19348                             sorgulayabilir.
19349
19350     Aşağıdaki birden fazla isimli borundan okuma yapmaya çalışan bir örnek
19351         bulunmaktadır. Boruların isimleri komut satırı
```

```
19346     argümanlarıyla verilmektedir. Bu isimli borular açılıp bunların      ↵
19347         betimleyicileri bir dizide saklanmıştır. select'in blokesi      ↵
19348         çözüldüğünde hangi borularda okuma eyleminin yapılabileceği FD_ISSET le ↵
19349             kontrol edilmiştir. Yazan taraf boruyu kapattığında      ↵
19350             bu da bir okuma eylemi gibi select'in blokesini çözer. Ancak bu durumda ↵
19351             borudan 0 byte okunacaktır. Aşağıdaki uygulama için      ↵
19352             önce isimli borular yaratıp farklı terminallerden bu boruları cat ile      ↵
19353             aşağıdaki gibi açınız.  

19354     cat > boru ismi  

19355 -----*/  

19356  

19357 #include <stdio.h>  

19358 #include <stdlib.h>  

19359 #include <string.h>  

19360 #include <fcntl.h>  

19361 #include <unistd.h>  

19362 #include <sys/select.h>  

19363  

19364 void exit_sys(const char *msg);  

19365  

19366 int main(int argc, char *argv[])
19367 {
19368     int fds[MAX_ARGS];
19369     int i;
19370     fd_set rset, rset_o;
19371     char buf[BUFFER_SIZE + 1];
19372     ssize_t result;
19373     int maxfds, nfds;
19374
19375     if (argc == 1) {
19376         fprintf(stderr, "too few arguments!..\n");
19377         exit(EXIT_FAILURE);
19378     }
19379
19380     printf("waiting at open...\n");
19381
19382     maxfds = 0;
19383     nfds = 0;
19384     FD_ZERO(&rset_o);
19385     for (i = 0; i < argc - 1; ++i) {
19386         if ((fds[i] = open(argv[i + 1], O_RDONLY)) == -1)
19387             exit_sys("open");
19388         FD_SET(fds[i], &rset_o);
19389         if (fds[i] > maxfds)
19390             maxfds = fds[i];
19391         ++nfds;
19392     }
19393     printf("waiting at select...\n");
```

```
19394     for (;;) {
19395         rset = rset_o;
19396         if (select(maxfds + 1, &rset, NULL, NULL, NULL) == -1)
19397             exit_sys("select");
19398         for (i = 0; i < argc - 1; ++i)
19399             if (FD_ISSET(fds[i], &rset)) {
19400                 if ((result = read(fds[i], buf, BUFFER_SIZE)) == -1)
19401                     exit_sys("read");
19402                 if (!result) {
19403                     printf("pipe closing...\n");
19404                     FD_CLR(fds[i], &rset_o);
19405                     close(fds[i]);
19406                     --nfds;
19407                     if (!nfds)
19408                         goto EXIT;
19409                 }
19410
19411                 buf[result] = '\0';
19412                 printf("%s", buf);
19413             }
19414         }
19415     EXIT:
19416         return 0;
19417     }
19418
19419     void exit_sys(const char *msg)
19420     {
19421         perror(msg);
19422         exit(EXIT_FAILURE);
19423     }
19424
19425     /
19426     *
-----*-----*-----*
-----*-----*-----*
```

19427 Aşağıdaki örnekte ise boruların betimleyicileri bir dizide toplanmıştır.

```
19428 -----*/
```

19429

```
19430 #include <stdio.h>
19431 #include <stdlib.h>
19432 #include <string.h>
19433 #include <fcntl.h>
19434 #include <unistd.h>
19435 #include <sys/select.h>
19436
19437 #define MAX_ARGS      32
19438 #define BUFFER_SIZE    1024
19439
19440 void exit_sys(const char *msg);
19441
19442 int main(int argc, char *argv[])
```

```
19443 {
19444     int fd;
19445     fd_set rset, rset_o;
19446     char buf[BUFFER_SIZE + 1];
19447     ssize_t result;
19448     int maxfds, nfds;
19449     int i;
19450
19451     if (argc == 1) {
19452         fprintf(stderr, "too few arguments!..\n");
19453         exit(EXIT_FAILURE);
19454     }
19455
19456     printf("waiting at open...\n");
19457
19458     maxfds = 0;
19459     nfds = 0;
19460     FD_ZERO(&rset_o);
19461     for (i = 0; i < argc - 1; ++i) {
19462         if ((fd = open(argv[i + 1], O_RDONLY)) == -1)
19463             exit_sys("open");
19464         FD_SET(fd, &rset_o);
19465         if (fd > maxfds)
19466             maxfds = fd;
19467         ++nfds;
19468     }
19469     printf("waiting at select...\n");
19470     for (;;) {
19471         rset = rset_o;
19472         if (select(maxfds + 1, &rset, NULL, NULL, NULL) == -1)
19473             exit_sys("select");
19474         for (i = 0; i <= maxfds; ++i)
19475             if (FD_ISSET(i, &rset)) {
19476                 if ((result = read(i, buf, BUFFER_SIZE)) == -1)
19477                     exit_sys("read");
19478                 if (!result) {
19479                     printf("pipe closing...\n");
19480                     FD_CLR(i, &rset_o);
19481                     close(i);
19482                     --nfds;
19483                     if (!nfds)
19484                         goto EXIT;
19485                 }
19486
19487                 buf[result] = '\0';
19488                 printf("%s", buf);
19489             }
19490     }
19491     EXIT:
19492     return 0;
19493 }
19494
19495 void exit_sys(const char *msg)
```

```
19496  {
19497      perror(msg);
19498
19499      exit(EXIT_FAILURE);
19500  }
19501
19502  /
*-----*
-----*
19503 pselect fonksiyonu select fonksiyonun biraz daha gelişmiş bir versiyonudur. ↵
    Prototipi şöyledir:
19504
19505 int pselect(int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds, ↵
    const struct timespec *timeout, const sigset_t *restrict sigmask);
19506
19507 Fonksiyonun select fonksiyonundan yalnızca üç farkı vardır. Diğer bütün ↵
    davranışları aynıdır.
19508
19509 1) Zaman aşımı için timeval yapısı değil timespec yapısı kullanılmıştır. Bu ↵
        da nanosaniye çözünürlük anlamına gelmektedir.
19510 2) Fonksiyon bir "signal set" parametresine sahiptir. Biz istersek fonksiyon ↵
        çalışana kadar belli sinyallerin bloke edilmesini sağlayabiliriz.
19511 Tabii bu parametreyi NULL da geçebiliriz. Fonksiyon sonlandığında otomatik ↵
        olarak bu sinyaller prosesin sinyal mask kümesinden çıkarılmaktadır.
19512 3) pselect fonksiyonunun zaman aşımı parametresi const biçimdedir. Yani ↵
        fonksiyon tarafından günellenmemektedir.
19513
19514 select ve pselect fonksiyonları eğer bloke edilmemişse ilgili sinyal ↵
        oluştuğunda -1 ile geri döner ve errno EINTR ile set edilir.
19515 Bu fonksiyonlar "restartable" yapılamazlar.
19516
19517 -----*/
19518
19519 /
*-----*
-----*
19520 poll fonksiyonu amaç bakımından select fonksiyonuna çok benzemektedir. ↵
    select ve poll aynı işi yapan farklı arayüzler biçiminde
19521 düşünülebilir. Eskiden select BSD sistemlerinde, poll ise AT&T UNIX ↵
        sistemlerinde kullanılıyordu. Tabii uzun süredir bu iki
19522 fonksiyon da POSIX standartlarında bulunmaktadır.
19523
19524 int poll(struct pollfd fds[], nfds_t nfds, int timeout);
19525
19526 poll fonksiyonu ilgilenilen betimleyicileri ve olayları tek tek bir yapı ↵
        dizisi biçiminde bizden ister. Yani biz bir yapı dizisi
19527 oluşturup onun içini doldururuz. Sonra bu yapı dizisinin adresini poll ↵
        fonksiyonuna veriririz. poll fonksiyonu da select fonksiyonunda
19528 olduğu gibi bu olayları blokede izler. Bu betimleyicilerden herhangi ↵
        birinde bir olay olduğunda blokeyi çözer. Programcı da girdiği
19529 diziyi kontrol ederek hangi olayların olduğunu anlayıp uygun işlemleri ↵
        yapar. Fonksiyonun timeout parametresi -1 girilirse zaman aşımı
```

```
19530     ortadan kaldırılmaktadır. Bu parametre 0 girilirse fonksiyon  
19531         betimleyicilerin durumlarına hemen bak****p geri döner. Sıfır  
19532         dışı değeri  
19533     milisaniye cinsinden zaman aşımı belirtmektedir. poll fonksiyonu  
19534         başarısızlık durumunda -1 değerine, zaman aşımından dolayı  
19535         sonlanmalarda 0  
19536     değerine ve normal sonlanmalarda ise olay gerçekleşen betimleyici  
19537     sayısına geri dönmektedir.  
19538 ------ */  
19539  
19540 #include <stdio.h>  
19541 #include <stdlib.h>  
19542 #include <unistd.h>  
19543 #include <poll.h>  
19544  
19545 void exit_sys(const char *msg);  
19546  
19547 int main(int argc, char *argv[]) {  
19548     struct pollfd pfds[1];  
19549     int result;  
19550     char buf[1024 + 1];  
19551     ssize_t n;  
19552  
19553     pfds[0].fd = 0;  
19554     pfds[0].events = POLLIN;  
19555  
19556     if ((result = poll(pfds, 1, -1)) == -1)  
19557         exit_sys("poll");  
19558  
19559     printf("%d event(s) occurred\n", result);  
19560  
19561     if (pfds[0].revents & POLLIN) {  
19562         if ((n = read(0, buf, 1024)) == -1)  
19563             exit_sys("read");  
19564         buf[n] = '\0';  
19565         printf("%s", buf);  
19566     }  
19567  
19568     return 0;  
19569 }  
19570  
19571 void exit_sys(const char *msg)  
19572 {
```

```
19574     perror(msg);
19575
19576     exit(EXIT_FAILURE);
19577 }
19578
19579 /
*-----*
-----*
19580     Aşağıdaki programda birden fazla isimli boru ile poll fonksiyonu
19581         kullanılarak multiplexed IO işlemi yapılmıştır.
19582     Programı çalıştırırken komut satırı argümanı olarak boru isimlerini
19583         giriniz. Girdiğiniz boruları başka terminallerden
19584         cat > boru ismi komutu ile açınız.
19585     */
19586
19587 #include <stdio.h>
19588 #include <stdlib.h>
19589 #include <fcntl.h>
19590 #include <unistd.h>
19591 #include <poll.h>
19592
19593 #define MAX_ARGS      32
19594 #define BUFFER_SIZE    1024
19595
19596 void exit_sys(const char *msg);
19597 {
19598     int i;
19599     int fd;
19600     char buf[BUFFER_SIZE + 1];
19601     int result;
19602     ssize_t n;
19603     int nfds;
19604     struct pollfd pfds[MAX_ARGS];
19605
19606     if (argc == 1) {
19607         fprintf(stderr, "too few arguments!..\n");
19608         exit(EXIT_FAILURE);
19609     }
19610
19611     printf("waiting at open...\n");
19612
19613     nfds = 0;
19614     for (i = 0; i < argc - 1; ++i) {
19615         if ((fd = open(argv[i + 1], O_RDONLY)) == -1)
19616             exit_sys("open");
19617         pfds[i].fd = fd;
19618         pfds[i].events = POLLIN;
19619         ++nfds;
19620     }
19621
```

```
19622     printf("waiting at poll...\n");
19623     for (;;) {
19624         if ((result = poll(pfds, nfds, -1)) == -1)
19625             exit_sys("poll");
19626         for (i = 0; i < nfds; ++i) {
19627             if (pfds[i].revents & POLLIN) {
19628                 if ((n = read(pfds[i].fd, buf, BUFFER_SIZE)) == -1)
19629                     exit_sys("read");
19630                 buf[n] = '\0';
19631                 printf("%s", buf);
19632             }
19633             else if (pfds[i].revents & POLLHUP) {
19634                 close(pfds[i].fd);
19635                 --nfds;
19636                 if (!nfds)
19637                     goto EXIT;
19638             }
19639         }
19640     }
19641 }
19642 EXIT:
19643     return 0;
19644 }
19645
19646 void exit_sys(const char *msg)
19647 {
19648     perror(msg);
19649
19650     exit(EXIT_FAILURE);
19651 }
19652
19653 /
*-----*
-----*
19654     select fonksiyonunun sinyal blokesi yapan pselect biçiminde bir      ↗
19655         versiyonu vardı. İşte poll fonksiyonun da Linux sistemlerinde      ↗
19656         sinyal blokesi yapan ppoll isimli bir versiyonu vardır. Ancak pselect      ↗
19657         POSIX standartlarında bulunduğu halde ppoll bulunmamaktadır.      ↗
19658         ppoll Linux'a özgüdür:
19659
19660     int ppoll(struct pollfd *fds, nfds_t nfds, struct timespec *timeout_ts, ↗
19661         const sigset_t *sigmask);
19662
*-----*/
19663
-----*
19664     select ve poll fonksiyonları bazı sistemlerde yüksek performansla      ↗
19665         çalışabilmektedir. Ancak Linux'ta bu fonksiyonların izlediği      ↗
19666         betimleyici sayısı arttıkça fonksiyonlar önemli bir yavaşlama içerisinde      ↗
19667         girmektedir. Yani maalesef Linux sistemlerinde select ve poll      ↗
19668         fonksiyonları iyi biçimde ölçeklendirilmemiştir. İşte Linux'ta daha iyi      ↗
```

ölçeklendirilmiş epoll isimli bir sistem fonksiyonu bulundurulmuştur.

19665 Yüksek performans isteyen server programlar Linux'ta epoll sistemini ↗
tercih etmektedir. Tabii epoll POSIX standartlarında mevcut değildir.

19666 Yalnızca Linux sistemlerinde bulunmaktadır. epoll sistemi söyle ↗
kullanılmaktadır:

19667

19668 1) Programcı önce epoll_create isimli fonksiyonla bir betimleyici elde ↗
eder. Bu etimleyicinin IO olaylarının izleneceği betimleyici ile
19669 bir ilgisi yoktur. Bu betimleyici diğer fonksiyonlara bir handle gibi ↗
geçirilmektedir:

19670

19671 int epoll_create(int size);

19672

19673 Fonksiyonun parametresi kaç betimleyiinin izlenileceğine yönelik bir ip ↗
ucu değeri alır. Programcı burada verdiği değerden daha fazla
19674 betimleyiciyi izleyebilir. Daha sonra bu parametre tasarımcıları ↗
rahatsız etmiş ve epoll_create isimli fonksiyonla kaldırılmıştır.

19675

19676 int epoll_create1(int flags);

19677

19678 Buradaki flags şimdilik yalnızca FD_CLOEXEC değerini ya da 0 değerini ↗
alabilmektedir. Fonksiyonların geri dönüş değeri
19679 başarı durumunda handle görevind eolan bir betimleyicidir.

19680

19681 2) Artık programcı izleyeceği betimleyicileri epoll sistemine epoll_ctl ↗
fonksiyonuyla ekler. Örneğin programcı 5 boru betimleyicisini
19682 izleyecekse bu 5 betimleyici için de ayrı ayrı epoll_ctl çağrısı ↗
yapmalıdır.

19683

19684 int epoll_ctl(int efd, int op, int fd, struct epoll_event *event);

19685

19686 Fonksiyonun birinci parametresi epoll_create ya da epoll_create1 ↗
fonksiyonundan elde edilen handle değeridir. İkinci parametre
19687 EPOLL_CTL_ADD, EPOLL_CTL_MOD, EPOLL_CTL_DEL değerlerinden birini alır. ↗
EPOLL_CTL_ADD ekleme için, EPOLL_CTL_DEL çıkarma için ve
19688 EPOLL_CTL_MOD mevcut eklenmiş betimleyicide izleme değişikliği yapmak ↗
için kullanılmaktadır. Üçüncü parametre izlenecek betimleyiciyi
19689 belirtir. Son parametre izlenecek olayı belirtmektedir. sturct ↗
epoll_event yapısı söyle bildirilmiştir:

19690

19691 struct epoll_event {

19692 uint32_t events;

19693 epoll_data_t data;

19694 };

19695

19696 Yapının events elemanı tipki poll fonksiyonunda olduğu gibi EPOLLIN, ↗
EPOLLOUT değerlerini almaktadır. Geri döndürülen
19697 olay da yine EPOLLIN, EPOLLOUT, EPOLLERR ve EPOLLHUP gibi olaylardır. ↗
Yapının data elemanı aslında çekirdek tarafından saklanıp
19698 epoll_wait fonksiyonu yoluyla bize geri verilmektedir. Bu eleman bir ↗
birlik biçiminde bildirilmiştir:

19699

19700 typedef union epoll_data {

```
19701     void      *ptr;
19702     int       fd;
19703     uint32_t   u32;
19704     uint64_t   u64;
19705 } epoll_data_t;

19706
19707 select ve poll fonksiyonları "düzey tetiklemeli (level triggered)"     ↵
    çalışmaktadır. epoll fonksiyonu da default durumda düzey tetiklemeli
19708 çalışır. Ancak events parametresine bit or işlemi ile EPOLLET eklenirse ↵
    o betimleyici için "kenar tetiklemeli (edge triggered)" mod
    kullanılır.
19709 Düzey tetiklemeli mod demek (select, poll daki durum ve epoll'daki
    default durum) bir okuma ya da yazma olayı açılıp bloke çözüldüğünde
19710 programcı eğer okuma ya da yazma yapmayı yeniden bu fonksiyonları
    çağrırsa bekleme yapılmayacak demektir. Yani örneğin biz select ya
    poll ile
19711 stdin dosyasını izliyorsak ve klavyeden bir giriş yapıldıysa bu
    fonksiyonlar blokeyi çözer. Fakat biz read ile okuma yapmazsak ve
    yeniden select ve poll
19712 fonksiyonlarını çağrırsak artık bloke oluşmaz. Halbuki kenar
    tetiklemeli modda biz okuma yapmasak bile yeni okuma eylemi oluşana
    kadar yine
19713 blokede kalırız.

19714
19715 3) Asıl bekleme ve izleme işlemi epoll_wait fonksiyonu tarafından
    yapılmaktadır. Bu fonksiyon poll select ve poll fonksiyonu gibi
19716 bloke oluşturur ve arka planda betimleyicileri izler.
19717
19718     int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int ↵
        timeout);
19719
19720 Fonksiyonun birinci parametresi epoll_create ya da epoll_create1
    fonksiyonundan elde edilmiş olan handle değeridir. İkinci
19721 parametre oluşan olayların depolanacağı yapı dizisinin adresidir. Biz bu
    yapının events elemanından oluşan olayın ne olduğunu
19722 anlarız. Yapının data elemanı epoll_ctl sırasında verdığımız değeri
    belirtir. Bizim en azından epoll_ctl fonksiyonund ilgili
19723 betimleyiciyi bu data elemanında girmiş olmamız gereklidir. Fonksiyonun
    üçüncü parametresi ikinci parametresiyle belirtilen dizinin
19724 uzunluğudur. (Normal olarak bu dizinin eklenmiş olan betimleyici sayısı
    kadar olması gereklidir. Ancak buradaki değer toplam izlenecek
    betimleyici
19725 asyisinden az olabilir. Bu parametre tek en fazla hamlede kaç
    betimleyici hakkında bilgi verileceğini belirtmektedir.) Son
    parametre yine milisaniye cinsinden
19726 zaman aşımını belirtir. -1 değeri zaman aşımının kullanılmayacağını, 0
    değeri hemen bakıp çıkışاقığını belirtmektedir. Fonksiyon
19727 başarı durumunda diziye doldurduğu eleman sayısı ile başarısızlık
    durumda -1 ile geri döner. Fonksiyon 0 değeri ile geri dönerse
19728 sonlanmanın zaman aşımından dolayı olduğu anlaşılır.
19729
19730 Sistemin kapatılması için epoll_create ya da epoll_create1 ile elde
    edilen betimleyici kapatılabilir.
```

```
19731 -----*/  
19732  
19733 #include <stdio.h>  
19734 #include <stdlib.h>  
19735 #include <unistd.h>  
19736 #include <sys/epoll.h>  
19737  
19738 void exit_sys(const char *msg);  
19739  
19740 int main(int argc, char *argv[])  
19741 {  
19742     int epfd;  
19743     struct epoll_event epe;  
19744     struct epoll_event epe_out[1];  
19745     int result;  
19746     char buf[1024 + 1];  
19747     ssize_t n;  
19748  
19749     if ((epfd = epoll_create(1)) == -1)  
19750         exit_sys("epoll_create");  
19751  
19752     epe.events = EPOLLIN;  
19753     epe.data.fd = STDIN_FILENO;  
19754     if (epoll_ctl(epfd, EPOLL_CTL_ADD, STDIN_FILENO, &epe) == -1)  
19755         exit_sys("epoll_ctl");  
19756  
19757     printf("waiting stdin...\n");  
19758  
19759     if ((result = epoll_wait(epfd, epe_out, 1, -1)) == -1)  
19760         exit_sys("epoll_wait");  
19761  
19762     if (epe_out[0].events & EPOLLIN) {  
19763         if ((n = read(epe_out[0].data.fd, buf, 1024)) == -1)  
19764             exit_sys("read");  
19765         buf[n] = '\0';  
19766         printf("%d event(s) occurred...\n", result);  
19767         printf("%s", buf);  
19768     }  
19769  
19770     return 0;  
19771 }  
19772  
19773 void exit_sys(const char *msg)  
19774 {  
19775     perror(msg);  
19776  
19777     exit(EXIT_FAILURE);  
19778 }  
19779  
19780 /-----*/  
-----*/
```

```
19781     epoll sisteminde izlemek istediğimiz betimleyicileri ekledikten sonra    ↵
19782         bunları çıkarmamız gerekmek. Bu betimleyicilere
19783         ilişkin dosyalar kapatıldığında zaten ilgili betimleyici izlemeden    ↵
19784             otomatik olarak çıkartılmaktadır.
19785
19786     Aşağıdaki örnekte yine bu kez epoll sistemi ile isimli borularla    ↵
19787         multiplexed io uygulaması yapılmıştır. Yine bu programı
19788         komut satırı argümanı olarak isimli boruların yolu fadelerini vererek    ↵
19789             çalıştırınız. Diğer terminallerden boruları yazma
19790         modunda cat > borusu ismi biçiminde açınız.
19791
19792     -----
19793     -----
19794
19795     #include <stdio.h>
19796     #include <stdlib.h>
19797     #include <fcntl.h>
19798     #include <unistd.h>
19799     #include <sys/epoll.h>
19800
19801     #define MAX_ARGS          32
19802     #define BUFFER_SIZE        1024
19803
19804     void exit_sys(const char *msg);
19805
19806     int main(int argc, char *argv[])
19807     {
19808         int epfd;
19809         struct epoll_event epe;
19810         struct epoll_event *epe_outs;
19811         int fd;
19812         char buf[BUFFER_SIZE + 1];
19813         int result;
19814         ssize_t n;
19815         int nfds;
19816         int i;
19817
19818         if (argc == 1) {
19819             fprintf(stderr, "too few arguments!..\n");
19820             exit(EXIT_FAILURE);
19821         }
19822
19823         if ((epfd = epoll_create(2)) == -1)
19824             exit_sys("epoll_create");
19825
19826         printf("waiting at open...\n");
19827
19828         nfds = 0;
19829         for (i = 0; i < argc - 1; ++i) {
19830             if ((fd = open(argv[i + 1], O_RDONLY)) == -1)
19831                 exit_sys("open");
19832             epe.events = EPOLLIN;
19833             epe.data.fd = fd;
19834             if (epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &epe) == -1)
```

```
19829         exit_sys("epoll_ctl");
19830         ++nfds;
19831     }
19832
19833     if ((epe_outs = (struct epoll_event *)malloc(nfds * sizeof(struct
19834         epoll_event))) == NULL)
19835         exit_sys("malloc");
19836
19837     printf("waiting at epoll...\n");
19838     for (;;) {
19839         if ((result = epoll_wait(epfd, epe_outs, nfd, -1)) == -1)
19840             exit_sys("epoll_wait");
19841         for (i = 0; i < nfd; ++i) {
19842             if (epe_outs[i].events & EPOLLIN) {
19843                 if ((n = read(epe_outs[i].data.fd, buf, BUFFER_SIZE)) == -1)
19844                     exit_sys("read");
19845                 buf[n] = '\0';
19846                 printf("%s", buf);
19847             }
19848             else if (epe_outs[i].events & EPOLLHUP) {
19849                 close(epe_outs[i].data.fd);
19850                 --nfd;
19851                 if (!nfd)
19852                     goto EXIT;
19853             }
19854         }
19855     EXIT:
19856     free(epe_outs);
19857
19858     return 0;
19859 }
19860
19861 void exit_sys(const char *msg)
19862 {
19863     perror(msg);
19864
19865     exit(EXIT_FAILURE);
19866 }
19867
19868 /
*-----*
-----*
-----*
```

19869 Linux sistemlerinde epoll özel bir ihtiyamla hazırlanmıştır. Bu nedenle epoll sistemi Linux'ta select ve poll fonksiyonlarından oldukça hızlı çalışmaktadır. Pekiyi neden? Çünkü Linux çekirdeği dosya nesnesinin içerisinde bu işlem için alan ayırmıştır.

19870 Biz bir betimleyiciyi izleme listesine epoll_ctl ile eklediğimizde çekirdek hemen onu ilgili bağlı listelere eklemektedir.

19871 Sonra bu betimleyicide olay gerçekleştiğinde zaten bu betimleyiciyi zaten gerçekleşen olay listesine almaktadır. Yani sonuçta aslında betimleyicilerin çekirdek tarafından izlenmesine, gözden geçirilmesine gerek kalmamaktadır. İzlenen betimleyicilerin artması

durumunda
19874 zaman kaybı "The Linux Programming Interface" kitabında 1365'inci ↵
sayfada verilmiştir:
19875
19876 Number of descriptors monitored (N) poll() CPU time (seconds) ↵
select() CPU time (seconds) epoll CPU time (seconds)
19877 10 0.61 ↵
0.73 0.41
19878 100 2.9 ↵
3.0 0.42
19879 1000 35 ↵
35 0.53
19880 10000 990 ↵
930 0.66
19881 -----*/
19882 /
19883 *-----

19884 Signal Driven IO işleminde belli bir betimleyicide olay oluşupunda ↵
SIGIO isimli sinyal oluşturulmaktadır. Böylece programcı
19885 bu oluştuğunda okuma/yazma işlemini yapabilmektedir. Ancak bu model ↵
Pgüncel POSIZ standartlarında bulunmamaktadır. Linux bu
19886 modeli desteklemektedir. Bunun için sırasıyla yapılmalıdır:
19887
19888 1) Betimleyici open fonksiyonuyla açılır.
19889 2) SIGIO sinyali için sinyal fonksiyonu set edilir.
19890 3) Betimleyici üzerinde F_SETOWN komut koduyla fcntl uygulanır. Üçüncü ↵
parametreye sinyalin gönderileceği prosesin
19891 ya da proses grubunun id'si girilmelidir. (Tabii genellikle programcı ↵
kendi prosesinin id'sini girer.)
19892 4) Betimleyici blokesiz moda sokulur ve aynı zamanda O_SYNC bayrağı da ↵
set edilir. Bu işlem fcntl fonksiyonunda F_SETFL komut
19893 koduyla yapılmaktadır.
19894
19895 fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK | O_ASYNC);
19896
19897 O_ASYNC bayrağı POSIX standartlarında bulunmamaktadır.
19898
19899 Bu yöntemde ilgilenilen olay (yani read mı write mı) gizlice open ↵
fonksiyonundaki açış modunda belirtilmektedir. Yani örneğin
19900 biz open fonksiyonunda dosyayı O_RDONLY modunda açarsak yalnızca okuma ↵
ilgilendigimiz, O_WRONLY modunda açarsak yalnızca yazma ile ↵
ilgilendigimiz,
19901 O_RDWR modunda açarsak hem okuma hem de yazma ile ilgilendigimiz sonucu ↵
çıkar.
19902
19903 5) Artık normal akışa devam eder. İlgilenilen olay gerçekleştiğinde ↵
sinyal oluşturulmaktadır.
19904
19905 Signal driven IO "kenar tetiklemeli (edge triggered)" bir yöntemdir. ↵
Blokesiz okuma/yazma yapılabilecek bir durumda (örneğin okuma

```
19906     durumunda boruya okunacak birşeyler gelmesi gibi) sinyal oluşur. Ancak ↵
19907         okuma/yazma yapılmasa bile yeni bir benzer durum oluştuğunda ↵
19908     yeniden sinyal oluşur. (Örneğin boruya bilgi geldiğinde sinyal oluşur. ↵
19909         Biz borudan okuma yapmasak bile yeniden boruya bilgi gelirse ↵
19910     yine sinyal oluşur. Halbuki select, poll böyle değildir. Anımsanacağı ↵
19911         gibi epoll'da bu durum ayarlanabilemktedir.) ↵
19912
19913     Aşağıdaki programda yine borular komut satırı argümanlarıyla ↵
19914         verilmektedir. Borular üzerinde okuma işlemi söz konusu olduğunda ↵
19915     SIGIO sinyali oluşacaktır. Okuma işlemi sinyal içerisinde değil dışarıda ↵
19916         yapılmıştır. Ancak sinyal fonksiyonunda bir bayrak set ↵
19917         edilmiştir. Yazan taraf boruyu kapattığında da yine SIGIO sinyali ↵
19918         oluşmaktadır. Tabii bu durumda yine read fonksiyonu blokeye ↵
19919         yol açmadan 0 ile geri dönecektir. Örnekte bekleme işleminin sigprocmask ↵
19920         ve sigsuspend ile yapıldığına dikkat ediniz. Bu tür ↵
19921         durumlarda pause kullanmak -puase öncesinde son bir sinyal gelirse- ↵
19922         sorunlara yol açma potansiyelindedir. ↵
19923
19924     Aslında istenirse sinyal fonksiyonu içerisinde hangi betimleyicide olay ↵
19925         olduğu anlaşılabilir. Ama bunun için sigaction ↵
19926         fonksiyonda flags parametresinin SA_SIGINFO biçiminde geçip ↵
19927         siginfo_t parametreli sinyal fonksiyonun set edilmesi sağlanmalıdır. ↵
19928     Bu siginfot yapısında Linux'ta (POSIX'te yok) si_fd elemanı SIGIO ↵
19929         oluşmasına yol açan dosya betimleyicisini bulundurmaktadır. ↵
19930
19931     -----
19932     -----
19933     -----
19934     -----
19935     -----
19936     -----
19937     -----
19938     -----
19939     -----
19940     -----
19941     -----
19942     -----
19943     -----
19944     -----
19945     -----
19946     -----*/
```

```
19921 #include <stdio.h>
19922 #include <stdlib.h>
19923 #include <errno.h>
19924 #include <fcntl.h>
19925 #include <unistd.h>
19926 #include <signal.h>
19927
19928 #define MAX_ARGS      32
19929 #define BUFFER_SIZE    1024
19930
19931 void sigio_handler(int sno);
19932 void exit_sys(const char *msg);
19933
19934 int g_flag;
19935
19936 int main(int argc, char *argv[])
19937 {
19938     int nfds, nfds_open;
19939     int fds[MAX_ARGS];
19940     struct sigaction sa;
19941     sigset(SIGPOLLIN, &sigio_handler);
19942     sigset(SIGPOLLERR, &exit_sys);
19943     int i;
19944     ssize_t n;
19945     char buf[BUFFER_SIZE];
19946 }
```

```
19947     if (argc == 1) {
19948         fprintf(stderr, "too few arguments!..\n");
19949         exit(EXIT_FAILURE);
19950     }
19951
19952     sa.sa_handler = sigio_handler;
19953     sigemptyset(&sa.sa_mask);
19954     sa.sa_flags = SA_RESTART;
19955
19956     if (sigaction(SIGIO, &sa, NULL) == -1)
19957         exit_sys("sigaction");
19958
19959     nfds = 0;
19960     for (i = 0; i < argc - 1; ++i) {
19961         if ((fds[i] = open(argv[i + 1], O_RDONLY)) == -1)
19962             exit_sys("open");
19963         if (fcntl(fds[i], F_SETOWN, getpid()) == -1)
19964             exit_sys("fcntl");
19965         if (fcntl(fds[i], F_SETFL, fcntl(fds[i], F_GETFL) | O_ASYNC |
19966             O_NONBLOCK) == -1)
19967             exit_sys("fcntl");
19968         ++nfds;
19969     }
19970
19971     sigemptyset(&sm);
19972     sigaddset(&sm, SIGIO);
19973     if (sigprocmask(SIG_BLOCK, &sm, &osm) == -1)
19974         exit_sys("sigprocmask");
19975
19976     nfds_open = nfds;
19977     printf("waiting at sigsuspend...\n");
19978     for (;;) {
19979         if (g_flag) {
19980             for (i = 0; i < nfds; ++i) {
19981                 if (fds[i] == -1)
19982                     continue;
19983                 if ((n = read(fds[i], buf, BUFFER_SIZE)) == -1) {
19984                     if (errno == EAGAIN)
19985                         continue;
19986                     exit_sys("read");
19987                 }
19988                 if (n == 0) {
19989                     close(fds[i]);
19990                     fds[i] = -1;
19991                     --nfds_open;
19992                     if (nfds_open == 0)
19993                         goto EXIT;
19994                     continue;
19995                 }
19996                 buf[n] = '\0';
19997                 printf("%s", buf);
19998             }
19999 }
```

```
19999
20000     g_flag = 0;
20001 }
20002 }
20003
20004 EXIT:
20005     return 0;
20006 }
20007
20008 void sigio_handler(int sno)
20009 {
20010     g_flag = 1;
20011 }
20012
20013 void exit_sys(const char *msg)
20014 {
20015     perror(msg);
20016
20017     exit(EXIT_FAILURE);
20018 }
20019
20020 /
*-----*-----*
```

20021 İleri Modellerinden biri de "Asenkron IO Modelidir". Bu modelde okuma/ yazma gibi işlemler başlatılır ancak akış devam eder. ↗
20022 İşlemler bittiğinde durum programcıya bir sinyal ya da fonksiyon çağrıSİ ile bildirilir. İşlemler tipik olarak şöyle ↗
20023 yürütülmektedir.

20024

20025 1) Önce struct aiocb isimli bir yapı türünden nesne tanımlayıp içinin doldurulması gereklidir. Yapı şu biçimdedir: ↗

```
20026
20027 struct aiocb {
20028     int          aio_fildes;
20029     off_t        aio_offset;
20030     volatile void *aio_buf;
20031     size_t       aio_nbytes;
20032     int          aio_reqprio;
20033     struct sigevent aio_sigevent;
20034     int          aio_lio_opcode;
20035 };
20036
```

20037 Yapının aio_fildes elemanına okuma/yazma yapılmak istenen dosyaya ilişkin dosya betimleyicisi yerleştirilir. Asenkron okuma/yazma ↗
20038 işlemleri dosya göstericisinin gösterdiği yerden itibaren yapılmamaktadır. Okuma/yazmanın dosyanın neresinden yapılacak yapının ↗

20039 aio_offset elemanında belirtilir. (Eğer yazma durumu söz konusuya ve dosya O_APPEND modda açıldıysa bu durumda aio_offset elemanın değerini dikkate alınmaz. Her yazılıan dosyaya eklenir.) Yapının aio_buf elemanı transfer yapılacak bellek adresini belirtir. Bu adresteKI dizinin işlem sonlanana kadar taşıyor durumda olması gerekmektedir. ↗

20040

20041

Yapının aio_nbytes elemanı okunacak ya da yazılacak byte miktarını belirtmektedir. Tabii burada belirtilen byte miktarı aslında aio_buf dizisinin uzunluğunu belirtmektedir. Yoksa bildirimde bulunulacak byte sayısını belirtmez. Yani örneğin asenkron biçimde bir borudan 100 byte okumak istesek bize "işlem bitti" bildirimini 100 byte okuduktan sonra gelmek zorunda değildir. Daha az miktarda okuma olayı gerçekleşmişse de "işlem bitti bildirimini" yapılır. Tabii hiçbir zaman burada belirtlen byte miktarından fazla okuma yazma yapılmayacaktır. Yapının aio_repio elemanı ise okuma/yazma için bir öncelik derecesi belirtmektedir. Yani bu değer yapılacak transferin önceliğine ilişkin bir ip ucu belirtir. Ancak işletim sisteminin bu ipucunu kullanıp kullanmayacağı isteğe bağlı bırakılmıştır. Bu eleman 0 geçilebilir. Yapının aio_sigevent elemanı işlem bittiğinde yapılacak bildirim hakkında bilgileri barındırmaktadır. Bu sigevent yapısını daha önce görmüştük. Şöyledi:

```
20049
20050     struct sigevent {
20051         int           sigev_notify;
20052         int           sigev_signo;
20053         union sigval sigev_value;
20054         void          (*sigev_notify_function) (union sigval);
20055         void          *sigev_notify_attributes;
20056     };
20057
20058     Bu yapının sigev_notify elemanı bildirimin türünü belirtir. Bu tür SIGEV_NONE, SIGEV_SIGNAL, SIGEV_THREAD biçiminde olabilir.
20059     Yapının sigev_signo elemanı ise eğer sinyal yoluyla bildirimde bulunulacaksa sinyalin numrasını belirtmektedir. Yapının sigev_value elemanı sinyal fonksiyonuna ya da athread fonksiyonuna gönderilecek kulanıcı tanımlı bilgiyi temsil etmektedir. Yapının sigev_notify_function elemanı eğer bildirim thread yoluyla yapılacaksa işletim sistemi tarafından yaratılan thread'in çağrıracığı fonksiyonu belirtmektedir. Yapının sigev_notify_attributes elemanı ise yaratılacak thread'in özelliklerini belirtir. Bu parametre NULL geçilebilir.
```

20063
20064 2) Şimdi okuma ya da yazma olayını aio_read ya da aio_write fonksiyonuyla başlatmak gereklidir. Artık akış bu fonksiyonlarda bloke olmayacağı fakat işlem bitince bize bildirimde bulunulacaktır.

```
20066
20067     int aio_read(struct aiocb *aiocbp);
20068     int aio_write(struct aiocb *aiocbp);
```

20069
20070 Fonksiyonlar başarı durumunda 0 başarısızlık durumunda -1 değerine geri dönmektedir. İşlemlerin devam ettiğine yani henüz sonlanmadığına dikkat ediniz. Bu fonksiyonlara verdığımız aiocb yapılarının işlem tamamlanana kadar yaşıyor olması gereklidir. Yani fonksiyon bizim verdiğimiz aicb yapısını çalışırken kullanıyor olabilir.

```
20073
20074     3) Anımsanacağı gibi biz aiocb yapısının aio_nbytes elemanına maksimum
```

```
okuma/yazma miktarını vermiştık. Halbuki bundan daha az okuma/yazma
20075 yapılması mümkün değildir. Pekiyi bize bildirimde bulunulduğunda ne kadar ➔
miktarda bilginin okunmuş ya da yazılmış olduğunu nasıl anlayacağımız?
20076 İşte bunun için aio_result isimli fonksiyon kullanılmaktadır:
20077
20078     ssize_t aio_return(struct aiocb *aiocbp);
20079
20080 Fonksiyon başarı durumunda transfer edilen byte sayısına başarısızlık ➔
durumunda -1'e geri dönmektedir. Eğer bildirim gelmeden
20081 bu fonksion çağrılırsa geri dönüş değeri anlamlı olmayabilir. aio_read ➔
ve aio_write fonksiyonları sinyal güvenli değildir ancak
20082 aio_Return ve aio_error fonksiyonları sinyal güvenlidir.
20083
20084 Aşağıdaki programda stdin dosyasından asenkron bir biçimde okuma ➔
yapımıdır. Her aio_read için okuma yeniden başlatılmış ve
20085 bildirim olarak SIGUSR1 sinyali seçilmiştir. Ana program beklemeyi pause ➔
ile değil sigprocmask ve sigsuspend fonksiyonları ile
20086 yapmaktadır. Sinyal fonksiyonunda yalnızca bir bayrak set edilmiş asıl ➔
işlem dışında yapılmıştır. Sinyal fonksiyonun içerisinde
20087 aio_return yapılabilir (abii bunun için aiocb yapı nesnesinin global ➔
alınması gereklidir) ancak maalesef işlemin devam ettirilmesi
20088 için aio_read yapılamaz. Çünkü aio_read sinyal güvenli değildir. (Tabii ➔
bildirim olarak sinyal yerine thread yöntemi kullanılırsa
20089 artık thread fonksiyonun içerisinde yeniden aio_read işlemi ➔
yapılabilmektedir.)
20090
20091 -----
20092 -----*/
20093 #include <stdio.h>
20094 #include <stdlib.h>
20095 #include <string.h>
20096 #include <signal.h>
20097 #include <aio.h>
20098
20099 void sigusr1_handler(int sno);
20100 void exit_sys(const char *msg);
20101
20102 char g_buf[1024 + 1];
20103 int g_flag;
20104
20105 int main(void)
20106 {
20107     struct aiocb cb;
20108     struct sigaction sa;
20109     ssize_t size;
20110     sigset_t sm, osm;
20111
20112     sa.sa_handler = sigusr1_handler;
20113     sigemptyset(&sa.sa_mask);
20114     sa.sa_flags = 0;
20115
20116     if (sigaction(SIGUSR1, &sa, NULL) == -1)
```

```
20117     exit_sys("siagaction");
20118
20119     cb.io_fildes = 0;          /* stdin */
20120     cb.io_offset = 0;          /* stdin ve pipe için 0 vermek gereklidir */
20121     cb.io_nbytes = 1024;
20122     cb.io_buf = g_buf;
20123     cb.io_reqprio = 0;
20124     cb.io_sigevent.sigev_notify = SIGEV_SIGNAL;
20125     cb.io_sigevent.sigev_signo = SIGUSR1;
20126     cb.io_sigevent.sigev_value.sival_int = 0;
20127
20128     sigemptyset(&sm);
20129     sigaddset(&sm, SIGUSR1);
20130
20131     if (sigprocmask(SIG_BLOCK, &sm, &osm) == -1)
20132         exit_sys("sigprocmask");
20133
20134     if (aio_read(&cb) == -1)
20135         exit_sys("aio_read");
20136
20137     for (;;) {
20138         sigsuspend(&osm);
20139
20140         if (g_flag) {
20141             if ((size = aio_return(&cb)) == -1)
20142                 exit_sys("aio_return");
20143             g_buf[size] = '\0';
20144             if (!strcmp(g_buf, "quit\n"))
20145                 break;
20146             printf("%s", g_buf);
20147             g_flag = 0;
20148             if (aio_read(&cb) == -1)
20149                 exit_sys("aio_read");
20150         }
20151     }
20152
20153     return 0;
20154 }
20155
20156 void sigusr1_handler(int sno)
20157 {
20158     g_flag = 1;
20159 }
20160
20161 void exit_sys(const char *msg)
20162 {
20163     perror(msg);
20164
20165     exit(EXIT_FAILURE);
20166 }
20167
20168 /
```

----- →

```
-----  
20169 Yukarıdaki örnekte bekleme işlemi bir senkronizasyon sorunu oluşmasın →  
20170 diye sigprocmask ve sigsuspend fonksiyonları yardımıyla →  
20171 yapılmıştır. Aslında bunun yerine istenirse aio_suspend isimli →  
20172 fonksiyondan da faydalanylabilir.  
20173  
20174 int aio_suspend(const struct aiocb *const list[], int nent, const struct →  
20175 timespec *timeout);  
20176 Fonksiyonun birinci parametresi beklenecek asenkron olayara ilişkin aiocb →  
20177 yapılarının bulunduğu dizinin adresini almaktadır.  
20178 Yani fonksiyon aslında birden fazla olayı bekleyebilmektedir. İkinci →  
20179 parametre birinci parametredeki dizinin uzunluğunu belirtir.  
20180 Son parametre zaman aşımı belirtmektedir. NULL geçilirse zaman aşımı →  
20181 kullanılmaz. Fonksiyon başarı durumunda 0, başarısızlık durumunda →  
20182 (yani zaman aşımı ya da sinyal oluşma durumunda) -1'e geri dönmektedir. →  
20183 Geri dönüş değerinin kontrol edilmesine genellikle gerek olmaz.  
20184 aio_suspend fonksiyonu eğer başlatılan io olayı sona ermişse hiç bloke →  
20185 oluşturmadmaktadır. (Halbuki örneğin pause fonksiyonu yalnızca sinyal →  
20186 oluştduğunda geri dönmektedir.)  
20187  
20188 -----*/  
20189  
20190 #include <stdio.h>  
20191 #include <stdlib.h>  
20192 #include <string.h>  
20193 #include <signal.h>  
20194 #include <aio.h>  
20195  
20196 void sigusr1_handler(int sno);  
20197 void exit_sys(const char *msg);  
20198  
20199 char g_buf[1024 + 1];  
20200 int g_flag;  
20201  
20202 int main(void)  
20203 {  
20204     struct aiocb cb;  
20205     const struct aiocb *pcb;  
20206     struct sigaction sa;  
20207     ssize_t size;  
20208  
20209     sa.sa_handler = sigusr1_handler;  
20210     sigemptyset(&sa.sa_mask);  
20211     sa.sa_flags = 0;  
20212  
20213     if (sigaction(SIGUSR1, &sa, NULL) == -1)  
20214         exit_sys("sigaction");  
20215  
20216     cb.io_fildes = 0;      /* stdin */
```

```
20211     cb.aio_offset = 0;           /* stdin ve pipe için 0 vermek gereklidir */
20212     cb.aio_nbytes = 1024;
20213     cb.aio_buf = g_buf;
20214     cb.aio_reqprio = 0;
20215     cb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
20216     cb.aio_sigevent.sigev_signo = SIGUSR1;
20217     cb.aio_sigevent.sigev_value.sival_int = 0;
20218
20219     if (aio_read(&cb) == -1)
20220         exit_sys("aio_read");
20221
20222     pcb = &cb;
20223     for (;;) {
20224         aio_suspend(&pcb, 1, NULL);
20225
20226         if (g_flag) {
20227             if ((size = aio_return(&cb)) == -1)
20228                 exit_sys("aio_return");
20229             g_buf[size] = '\0';
20230             if (!strcmp(g_buf, "quit\n"))
20231                 break;
20232             printf("%s", g_buf);
20233             g_flag = 0;
20234             if (aio_read(&cb) == -1)
20235                 exit_sys("aio_read");
20236         }
20237     }
20238
20239     return 0;
20240 }
20241
20242 void sigusr1_handler(int sno)
20243 {
20244     g_flag = 1;
20245 }
20246
20247 void exit_sys(const char *msg)
20248 {
20249     perror(msg);
20250
20251     exit(EXIT_FAILURE);
20252 }
20253
20254 /
*-----*
-----*
20255     aio_read ya da aio_write fonksiyonlarında belirtilen işlem bittiğinde      ↗
20256     istenirse sigevent yapısı yoluyla bildirim biçimini      ↗
20257     SIGEV_THREAD seçilebilir. Bu durumda işlem bittiğinde işletim sistemi      ↗
20258     tarafından bir thread yaratılacak ve o thread akışı tarafından      ↗
20259     belirlenen fonksiyon çağrılmacaktır. İşlemlerin devam ettirilmesi bu      ↗
20260     thread fonksiyonu tarafından yapılabilir.
```

```
20259     Aşağıdaki örnekte program komut argümanlarıyla fifo dosyalarının yol      ↵
20260         ifadelerini almaktadır. Bunları asenkron io yöntemiyle okumaktadır.      ↵
20261     Her io olayı bittiğinde belirlenen fonksiyon çağrılmış ve okunan      ↵
20262         bilgiler ekrana yazdırılmıştır. Bu programda struct aiocb yapısının      ↵
20263         ve transfer alanının (buffer'in) bir yapıda tutulduğuna dikkat ediniz.      ↵
20264         Bu teknik bu tür durumlarda sık kullanılmaktadır.      ↵
20265     Çünkü bazen (özellikle soket uygulamalarında) programcı tampon alanda      ↵
20266         biriktirme yapıp birikmiş olan bilgiyi işleme sokmak isteyebilir.      ↵
20267     -----
20268         -----
20269         -----
20270         -----
20271         -----
20272         -----
20273         -----
20274     void read_completion_proc(union sigval val);
20275     void exit_sys(const char *msg);
20276
20277     typedef struct {
20278         struct aiocb cb;
20279         char buf[BUFFER_SIZE + 1];
20280     } IOCB_BUF;
20281
20282     int main(int argc, char *argv[])
20283     {
20284         int nfds;
20285         int fds[MAX_ARGS];
20286         IOCB_BUF *cbufs[MAX_ARGS];
20287         int i;
20288
20289         if (argc == 1) {
20290             fprintf(stderr, "too few arguments!..\\n");
20291             exit(EXIT_FAILURE);
20292         }
20293
20294         printf("waiting for pipe to be opened...\\n");
20295         nfds = 0;
20296         for (i = 0; i < argc - 1; ++i) {
20297             if ((fds[i] = open(argv[i + 1], O_RDONLY)) == -1)
20298                 exit_sys("open");
20299
20300             if ((cbufs[i] = (IOCB_BUF *)calloc(1, sizeof(IOCB_BUF))) == NULL)
20301                 exit_sys("calloc");
20302
20303             cbufs[i]->cb.aio_fildes = fds[i];
20304             cbufs[i]->cb.aio_offset = 0;
20305             cbufs[i]->cb.aio_buf = cbufs[i]->buf;
20306             cbufs[i]->cb.aio_nbytes = BUFFER_SIZE;
```

```
20307         cbbufs[i]->cb.aio_reqprio = 0;
20308         cbbufs[i]->cb.aio_sigevent.sigev_notify = SIGEV_THREAD;
20309         cbbufs[i]->cb.aio_sigevent.sigev_value.sival_ptr = cbbufs[i];
20310         cbbufs[i]->cb.aio_sigevent.sigev_notify_function =
20311             read_completion_proc;
20312
20313     if (aio_read(&cbbufs[i]->cb) == -1)
20314         exit_sys("aio_read");
20315
20316     ++nfdss;
20317 }
20318
20319 printf("waiting at getchar...\n");
20320 getchar();
20321
20322 for (i = 0; i < nfdss; ++i) {
20323     close(cbbufs[i]->cb.aio_fildes);
20324     free(cbbufs[i]);
20325 }
20326
20327 return 0;
20328 }
20329 void read_completion_proc(union sigval val)
20330 {
20331     IOCB_BUF *cbbuf = (IOCB_BUF *)val.sival_ptr;
20332     ssize_t n;
20333
20334     if ((n = aio_return(&cbbuf->cb)) == -1)
20335         exit_sys("aio_return");
20336
20337     if (n == 0) {
20338         printf("closing pipe...\n");
20339         return;
20340     }
20341
20342     cbbuf->buf[n] = '\0';
20343     printf("%s", cbbuf->buf);
20344
20345     if (aio_read(&cbbuf->cb) == -1)
20346         exit_sys("aio_read");
20347 }
20348
20349 void exit_sys(const char *msg)
20350 {
20351     perror(msg);
20352
20353     exit(EXIT_FAILURE);
20354 }
20355
20356 /
*-----*-----*
```

```
20357     Yukarıdaki program şöyle de düzenlenebilirdi
20358  -----
20359
20360 #include <stdio.h>
20361 #include <stdlib.h>
20362 #include <fcntl.h>
20363 #include <unistd.h>
20364 #include <aio.h>
20365
20366 #define MAX_ARGS      32
20367 #define BUFFER_SIZE    102
20368
20369 void read_completion_proc(union sigval val);
20370 void exit_sys(const char *msg);
20371
20372 typedef struct {
20373     struct aiocb cb;
20374     char buf[BUFFER_SIZE + 1];
20375 } IOCB_BUF;
20376
20377 int main(int argc, char *argv[])
20378 {
20379     int fds[MAX_ARGS];
20380     IOCB_BUF *cbbuf;
20381     int i;
20382
20383     if (argc == 1) {
20384         fprintf(stderr, "too few arguments!..\n");
20385         exit(EXIT_FAILURE);
20386     }
20387
20388     printf("waiting for pipe to be opened...\n");
20389     for (i = 0; i < argc - 1; ++i) {
20390         if ((fds[i] = open(argv[i + 1], O_RDONLY)) == -1)
20391             exit_sys("open");
20392
20393         if ((cbbuf = (IOCB_BUF *)calloc(1, sizeof(IOCB_BUF))) == NULL)
20394             exit_sys("malloc");
20395
20396         cbbuf->cb.aio_fildes = fds[i];
20397         cbbuf->cb.aio_offset = 0;
20398         cbbuf->cb.aio_buf = cbbuf->buf;
20399         cbbuf->cb.aio_nbytes = BUFFER_SIZE;
20400         cbbuf->cb.aio_reqprio = 0;
20401         cbbuf->cb.sigevent.sigev_notify = SIGEV_THREAD;
20402         cbbuf->cb.sigevent.sigev_value.sival_ptr = cbbuf;
20403         cbbuf->cb.sigevent.sigev_notify_function = read_completion_proc;
20404
20405         if (aio_read(&cbbuf->cb) == -1)
20406             exit_sys("aio_read");
20407     }
20408 }
```

```
20409     printf("waiting at getchar...\n");
20410
20411     getchar();
20412
20413     return 0;
20414 }
20415
20416 void read_completion_proc(union sigval val)
20417 {
20418     IOCB_BUF *cbbuf = (IOCB_BUF *)val.sival_ptr;
20419     ssize_t n;
20420
20421     if ((n = aio_return(&cbbuf->cb)) == -1)
20422         exit_sys("aio_return");
20423
20424     if (n == 0) {
20425         close(cbbuf->cb.aio_fildes);
20426         free(cbbuf);
20427         return;
20428     }
20429
20430     cbbuf->buf[n] = '\0';
20431     printf("%s", cbbuf->buf);
20432
20433     if (aio_read(&cbbuf->cb) == -1)
20434         exit_sys("aio_read");
20435 }
20436
20437 void exit_sys(const char *msg)
20438 {
20439     perror(msg);
20440
20441     exit(EXIT_FAILURE);
20442 }
20443
20444 /
*-----*-----*-----*
-----*-----*-----*
```

20445 Bu tür uygulamalarda (aslında multiplexed io da böyle) programcı önce
biriktirme yapıp sonra belli bir tampon dolduğunda
20446 biriktirdiklerini işleme sokmak isteyebilir. Bu tür isteklerle özellikle
soket uygulamalarda çok karşılaşılmaktadır.

20447 Aşağıda bu biçimde biriktirerek işleme sokmaya yönelik bir örnek
verilmiştir

```
-----*/
```

20449
20450 #include <stdio.h>
20451 #include <stdlib.h>
20452 #include <fcntl.h>
20453 #include <unistd.h>
20454 #include <aio.h>
20455

```
20456 #define MAX_ARGS          32
20457 #define MSG_SIZE           10
20458
20459 void read_completion_proc(union sigval val);
20460 void exit_sys(const char *msg);
20461
20462 typedef struct {
20463     struct aiocb cb;
20464     char buf[MSG_SIZE + 1];
20465     size_t index;
20466     size_t left;
20467 } IOCB_BUF;
20468
20469 int main(int argc, char *argv[])
20470 {
20471     int fds[MAX_ARGS];
20472     IOCB_BUF *cbbuf;
20473     int i;
20474
20475     if (argc == 1) {
20476         fprintf(stderr, "too few arguments!..\n");
20477         exit(EXIT_FAILURE);
20478     }
20479
20480     printf("waiting for pipe to be opened...\n");
20481     for (i = 0; i < argc - 1; ++i) {
20482         if ((fds[i] = open(argv[i + 1], O_RDONLY)) == -1)
20483             exit_sys("open");
20484
20485         if ((cbbuf = (IOCB_BUF *)calloc(1, sizeof(IOCB_BUF))) == NULL)
20486             exit_sys("malloc");
20487
20488         cbbuf->index = 0;
20489         cbbuf->left = MSG_SIZE;
20490         cbbuf->cb.io_fildes = fds[i];
20491         cbbuf->cb.io_offset = 0;
20492         cbbuf->cb.io_buf = cbbuf->buf;
20493         cbbuf->cb.io_nbytes = MSG_SIZE;
20494         cbbuf->cb.io_reqprio = 0;
20495         cbbuf->cb.sigevent.sigev_notify = SIGEV_THREAD;
20496         cbbuf->cb.sigevent.sigev_value.sival_ptr = cbbuf;
20497         cbbuf->cb.sigevent.sigev_notify_function = read_completion_proc;
20498
20499         if (aio_read(&cbbuf->cb) == -1)
20500             exit_sys("aio_read");
20501     }
20502
20503     printf("waiting at getchar...\n");
20504
20505     getchar();
20506
20507     return 0;
20508 }
```

```
20509
20510 void read_completion_proc(union sigval val)
20511 {
20512     IOCB_BUF *cbbuf = (IOCB_BUF *)val.sival_ptr;
20513     ssize_t n;
20514
20515     if ((n = aio_return(&cbbuf->cb)) == -1)
20516         exit_sys("aio_return");
20517
20518     if (n == 0) {
20519         close(cbbuf->cb.aio_fildes);
20520         free(cbbuf);
20521         return;
20522     }
20523
20524     cbbuf->index += n;
20525     cbbuf->left -= n;
20526
20527     if (cbbuf->left == 0) {
20528         cbbuf->buf[MSG_SIZE] = '\0';
20529         printf("Buffer filled: %s\n", cbbuf->buf);
20530         cbbuf->index = 0;
20531         cbbuf->left = MSG_SIZE;
20532     }
20533
20534     cbbuf->cb.aio_nbytes = cbbuf->left;
20535     cbbuf->cb.aio_buf = cbbuf->buf + cbbuf->index;
20536     if (aio_read(&cbbuf->cb) == -1)
20537         exit_sys("aio_read");
20538 }
20539
20540 void exit_sys(const char *msg)
20541 {
20542     perror(msg);
20543
20544     exit(EXIT_FAILURE);
20545 }
20546
20547 /
-----*-----*-----*-----*
```

20548 aio_error isimli fonksiyon herhangi bir durumda başlatılan işlemin akibeti konusunda bilgi almak için kullanılabilir.

20549

20550 int aio_error(const struct aiocb *aiocbp);

20551

20552 Fonksiyonun geri dönüş değeri bu asenkron işlemin o anda ne durumda olduğu hakkında bize bilgi verir. Eğer fonksiyon

20553 EINPROGRESS değerine geri dönerse işlemin hala devam ettiği anlamını çıkar. Geri dönüş değeri ECANCELED ise bu durumda

20554 işlem aio_cancel fonksiyonuyla iptal edilmiştir. Fonksiyon errno değerini set etmez. Geri dönüş değeri diğer pozitif değerlerden birisi ise hata ile ilgili başka bir errno değerini belirtir. Fonksiyon işlem

```
başarılı bir biçimde işlem bitmişse 0
20556 değerine geri döner.
20557 -----
20558 -----
20559 / -----
20560     aio_cancel fonksiyonu ise başlatılmış olan bir işlemi iptal etmek için
20561     kullanılmaktadır.
20562
20563     int aio_cancel(int fd, struct aiocb *aiocbp);
20564
20565     Fonksiyonun birinci parametresi iptal edilecek betimleyiciyi belirtir. Eğer
20566     iocb NULL geçilirse bu betimleyiciye ilişkin
20567     bütün asenkron işlemler iptal edilmektedir.
20568
20569     Fonksiyon AIO_CANCELED değerine geri dönerse iptal başarılıdır.
20570     AIO_NOTCANCELED değerine geri dönerse işlem aktif biçimde
20571     devam etmekte olduğu için iptal başarısızdır. AIO_ALLDONE değeri ise
20572     işlemin zaten bittiğini belirtir. Fonksiyon başarısızlık
20573     durumunda -1 değerine geri döner.
20574 -----
20575 / -----
20576
20577     Pek çok uygulamada değişik adreslerdeki bilgilerin peşi sıraya dosyaya
20578     yazılması ya da ters olarak okunması gerekebilmektedir.
20579     Örneğin bir kaydı temsil eden aşağıdaki üç bilginin birbiri ardına
20580     dosyaya yazılmak istendiğini düşünelim:
20581
20582     int record_len;
20583     char record[RECORD_SIZE];
20584     int record_no;
20585
20586     Bu bilgilerin dosyaya yazılması için normal olarak üç ayrı write işlemi
20587     yapmak gereklidir:
20588
20589     if (write(fd, &record_len, sizeof(int)) != sizeof(int)) {
20590         ...
20591     }
20592
20593     if (write(fd, record, RECORD_SIZE) != RECORD_SIZE) {
20594         ...
20595     }
20596
20597     if (write(fd, &record_no, sizeof(int)) != sizeof(int)) {
20598         ...
20599     }
```

```
20595     üç write işlemi göreli bir zaman kaybı oluşturabilmektedir. Tabii zaman →  
20596         kaybı uygulamaların ancak çok azında önem oluşturur.  
20597     Buradaki zaman kaybının en önemli nedeni her write çağrısının kernel →  
20598         mode'a geçiş yapmasıdır. Eğer bu zaman kaybını aşağı çekmek  
20599         istiyorsak ilk gelen yöntem önce bu bilgileri başka bir tampona →  
20600             kopyalayıp tek bir write işlemi yapmaktadır:  
20601     char buf[BUFSIZE];  
20602     memcpy(buf, &recordlen, sizeof(int));  
20603     memcpy(buf + sizeof(int), record, BUFSIZE);  
20604     memcpy(buf + sizeof(int) + BUFSIZE, &record_no, sizeof(int));  
20605     if (write(fd, buf, 2 * sizeof(int) + BUFSIZE) != 2 * sizeof(int) + →  
20606         BUFSIZE) {  
20607         ....  
20608     }  
20609     Bu işlem üç ayrı write işlemine göre oldukça hızlidır. İşte readv ve →  
20610         writev isimli fonksiyonlar farklı adreslerdeki bilgileri  
20611         yukarıdakine benzer biçimde dosyaya yazıp dosyadan okumaktadır. Bu →  
20612         işlemlere İngilizce "scatter/gather IO" denilmektedir. readv ve  
20613         writev fonksiyonlarının prototipleri şöyledir:  
20614     ssize_t readv(int fildes, const struct iovec *iov, int iovcnt);  
20615     ssize_t writev(int fildes, const struct iovec *iov, int iovcnt);  
20616     Fonksiyonların birinci parametreleri okuma ya da yazma işlemin →  
20617         yapılabacağı dosya betimleyicisini, ikinci parametreleri kullanılacak  
20618         tampon uzunluğun belirtildiği yapı dizisinin adresini, üçüncü →  
20619         parametresi de bu dizinin uzunluğunu belirtir. Programcı struct iovec  
20620         türünden bir yapı dizisi oluşturmuş onun içini doldurmalıdır. Geri →  
20621         dönüş değeri başarısızlık durumunda -1, diğer durumlarda okunan  
20622         yazılan  
20623         byte miktarıdır. Okuma ve yazma işlemleri tek parça haline atomik →  
20624         biçimde yapılmaktadır. iovec yapısı şöyle bildirilmiştir:  
20625     struct iovec {  
20626         void *iov_base;  
20627         size_t iov_len;  
20628     };  
20629     Aşağıdaki programda üç ayrı adresteği yazılar peşi sıra tek bir writev →  
20630         çağrı ile dosyaya yazdırılmıştır.  
20631     -----*/  
20632     #include <stdio.h>  
20633     #include <stdlib.h>  
20634     #include <string.h>
```

```
20635 #include <sys/uio.h>
20636
20637 #define BUFFER_SIZE      10
20638
20639 void exit_sys(const char *msg);
20640
20641 int main(void)
20642 {
20643     int fd;
20644     char *buf1[BUFFER_SIZE];
20645     char *buf2[BUFFER_SIZE];
20646     char *buf3[BUFFER_SIZE];
20647     struct iovec vec[3];
20648
20649     if ((fd = open("test.dat", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|
20650           S_IRGRP|S_IROTH)) == -1)
20651         exit_sys("open");
20652
20653     memset(buf1, 'a', BUFFER_SIZE);
20654     memset(buf2, 'b', BUFFER_SIZE);
20655     memset(buf3, 'c', BUFFER_SIZE);
20656
20657     vec[0].iov_base = buf1;
20658     vec[0].iov_len = BUFFER_SIZE;
20659
20660     vec[1].iov_base = buf2;
20661     vec[1].iov_len = BUFFER_SIZE;
20662
20663     vec[2].iov_base = buf3;
20664     vec[2].iov_len = BUFFER_SIZE;
20665
20666     if (writev(fd, vec, 3) == -1)
20667         exit_sys("writev");
20668
20669     close(fd);
20670
20671     return 0;
20672 }
20673 void exit_sys(const char *msg)
20674 {
20675     perror(msg);
20676
20677     exit(EXIT_FAILURE);
20678 }
20679
20680 /
*-----*
-----*
20681 Aşağıdaki programda da readv kullanılarak yukarıdaki işlemin tersi
20682 yapılmıştır
-----*/
```

```
20683
20684 #include <stdio.h>
20685 #include <stdlib.h>
20686 #include <string.h>
20687 #include <fcntl.h>
20688 #include <unistd.h>
20689 #include <sys/uio.h>
20690
20691 #define BUFFER_SIZE      10
20692
20693 void exit_sys(const char *msg);
20694
20695 int main(void)
20696 {
20697     int fd;
20698     char *buf1[BUFFER_SIZE];
20699     char *buf2[BUFFER_SIZE];
20700     char *buf3[BUFFER_SIZE];
20701     struct iovec vec[3];
20702
20703     if ((fd = open("test.dat", O_RDONLY)) == -1)
20704         exit_sys("open");
20705
20706     vec[0].iov_base = buf1;
20707     vec[0].iov_len = BUFFER_SIZE;
20708
20709     vec[1].iov_base = buf2;
20710     vec[1].iov_len = BUFFER_SIZE;
20711
20712     vec[2].iov_base = buf3;
20713     vec[2].iov_len = BUFFER_SIZE;
20714
20715     if (readv(fd, vec, 3) == -1)
20716         exit_sys("writev");
20717
20718     write(1, buf1, BUFFER_SIZE);
20719     write(1, buf2, BUFFER_SIZE);
20720     write(1, buf3, BUFFER_SIZE);
20721
20722     close(fd);
20723
20724     return 0;
20725 }
20726
20727 void exit_sys(const char *msg)
20728 {
20729     perror(msg);
20730
20731     exit(EXIT_FAILURE);
20732 }
20733
20734 /-----*-----*-----*-----*-----*
```

20735 Arka planda sessiz sedasız çalışan bir kullanıcı arayüzü olmayan, ↵
kullanıcılarla etkileşmeyen programlara Windows dünyasında
20736 "service" UNIX/Linux dünyasında "daemon" denilmektedir. Servisler ya da ↵
daemon'lar tipik olarak boot sırasında çalışmaya başlatılırlar
20737 ve yine tipik olarak makine reboots edilene kadar çalışmaya devam ↵
ederler. Tabii böyle bir sorumluluk yoktur. Servis ya da "daemon"
20738 kernel mod bir kavram değildir. Yani servisler ve daemon'lar genellikle ↵
"user mode"da yazılırlar. UNIX/Linux dünyasında geleneksel
20739 olarak "daemon"lar xxxxdd biçiminde sonuna 'd' harfi getirilerek ↵
isimlendirilmektedir. Çekirdeğe ilişkin bazı thread'ler de servis
20740 benzeri işlemler yaptıkları için bunlar da çoğu kez sonu 'd' ile bitecek ↵
ancak başı da 'k' ile başlayacak biçimde isimlendirilmiştir.
20741 Bu kernel daemon'ların bizim şu andaki konumuz olan daemon'larla hiçbir ↵
ilgisi yoktur. Yalnızca işlev bakımından bir benzerlikleri vardır.
20742 UNIX/Linux dünyasında daemon dendiğinde akla tabii ki "server" ↵
programları gelir. Örneğin ftp server programının ismi "ftpd" ↵
biçiminde
20743 olabilir. Ya da örneğin HTTP server programının ismi "httpd" biçiminde ↵
olabilir.
20744
20745 daemon'lar genellikle arka planda önemli işlemler yaptıkları için "root" ↵
(process id 0) hakkıyla (yani sudo ile) çalıştırılırlar. Daemon ↵
programları
20746 pek çok modern UNIX/Linux sisteminde init paketleri içerisindeki özel ↵
utility'ler tarafından başlatılmış, sürdürülüp sonlandırılmaktadır.
20747 Yani ilgili dağıtımın bu daemon'ları idare etmek için özel komutları ↵
bulunabilmektedir. Linux sistemlerinde init prosesi ve diğer ↵
proseslerin
20748 kodları ve boot süreci ile ilgili utility'ler "init paketleri" denilen ↵
ve farklı proje grupları tarafından oluşturulmuştur. Ve tipik olarak
20749 bugüne kadar yaygın 3 init paketi kullanılmıştır:
20750
20751 1) sysvinit: Klasik System5'teki işlevleri yapan init paketi. Linux uzun ↵
bir süre bu paketi kullanmıştır.
20752 2) Upstart: 2006 yılında oluşturulmuştur ve 2010'ların ortalarına kadar ↵
(bazı dağıtımlarda hala) kullanılmaya devam edilmiştir.
20753 3) systemd: 2010 yılında oluşturulmuştur ve son yıllarda pek çok Linux ↵
dağıtımında kullanılmaya başlanmıştır.
20754
20755 Br daemon programının yazılması tipik olarak şu aşamalardan geçilerek ↵
yapılmaktadır:
20756
20757 1) Daemon programları bir dosya açmak istediklerinde tam olarak ↵
belirlenen haklarla bunu yapmalıdır. Bu nedenle bu proseslerin
20758 umask değerlerinin 0 yapılması uygun olur.
20759
20760 2) Bir prosesin daemon etkisi yaratması için terminalle bir ↵
bağlantısının kalmaması gereklidir. Bu maalesef 0, 1, 2 numaralı terminal
20761 betimleyicilerinin kapatılmasıyla sağlanamaz. Bunu sağlananın en temel ↵
yolu setsid fonksiyonunu çağrılmaktır. Anımsanacağı gibi
20762 setsid fonksiyonu yeni bir oturum (session) ve yeni bir proses grubu ↵
oluşturup lgili prosesi bu proses grubunun ve oturumun lideri

20763 yapmaktadır. Ayrıca setsid fonksiyonu prosesin terminal ilişkisini (controlling terminal) ortadan kaldırmaktadır. Ancak setsid uygulayabilmek →
20764 için prosesin herhangi bir prosesin grup lideri olmaması gereklidir. Aksi takdirde fonksiyon başarısız olmaktadır. Anımsanacağı gibi →
20765 kabuk programları çalıştırıldıkları programlar için bir proses grubu yaratıp o programı da proses grub lideri yapıyordu. İşte proses grub →
20766 lideri olmaktan kurtulmak için bir kez fork yapıp üst prosesi sonlandırabiliriz. Aynı zamanda bu işlem kabuk programının hemen komut →

20767 satırına yeniden düşmesine yol açacaktır. O halde 2'inci aşamadı fork yapılip üst proses sonlandırılmalıdır. Bilindiği gibi teeminale bağlı →
20768 programlar kabukta çalıştırıldıklarında yeniden komut satırına düşük bile kabuk programları kapatıldığında bu programlar →
sonlandırılmaktadır.
20769 Yani örneğin biz terminal ilişkisini kesmezsek bir kez fork yapıp alt proses arka planda çalışır gibi olur ancak bu durumda terminal kapatıldığında o alt proses de sanlandırılır.
20770
20771 3) Artık alt proses setsid fonksiyonunu uygulayarak yeni bir oturum →
yaratır ve terminal ilişkisini keser. Terminal ilişkisinin kesilmesi →
20772 ile artık terminal kapatılsa bile programımız çalışmaya devam eder. Tabii setsid ile terminal bağlantısının kesilmiş olması programın →
20773 terminale bir şey yazamayacağı anlamına gelmez. Hala 0, 1, 2 numaralı betimleyiciler açıkta. Terminal açık olduğu sürece oraya yazma →
20774 yapılabilir.
20775
20776 4) Daemon programların çalışma dizinlerinin (current working directory) →
sağlam bir dizin olması tavsiye edilir. Aksi takdirde
20777 o dizin silinirse arka plan bu programların çalışmaları bozulur. Bu nedenle daemon programların çoğu kök dizini (silinemeyeceği için) →
20778 çalışma dizini yapmaktadır. Tabii bu zorunlu değildir. Bunun yerine varlığı garanti edilmiş olan herhangi bir dizin de çalışma →
20779 dizini yapılabilir.
20780
20781 5) Daemon programın o ana kadar açılmış olan tüm betimleyicileri →
kapatması uygun olur. Örneğin 0, 1, 2 numaralı betimleyiciler →
20782 ilgili terminale ilişkindir ve artık o terminal kapatılmış ya da →
kapatılacak olabilir. Program kendini daemon yaptığı →
20783 sırada açmış olduğu diğer dosyaları da kapatmalıdır. Bunu sağlamanın basit bir yolu prosesin toplam dosya betimleyici tablosunun →
20784 uzunluğunu elde edip her bir betimleyici için close işlemi uygulamaktır. Çünkü maalesef biz açık betimleyicileri pratik bir →
20785 biçimde tespit edememekteyiz. Zaten kapalı bir betimleyiciye close →
uygulanırsa close başarısız olur ancak program çökmez.
20786 Prosesin toplam betimleyici sayısı sysconf çağrılarında _SC_OPEN_MAX →
argümanıyla ya da getrlimit fonksiyonunda RLIMIT_NOFILE →
20787 argümanıyla elde edilebilir. İki fonksiyon da aynı değeri vermektedir.
20788
20789 6) Zorunlu olmamakla birlikte ilk üç betimleyiciyi /dev/null aygıtına →
yönlendirmek iyi bir fikirdir. Çünkü bir biçimde bazı →
20790 fonksiyonlar bu betimleyicileri kullanıyor olabilirler. Anımsanacağı gibi /dev/null aygıtına yazılanlar kaybolmaktadır. Bu aygıtta

```
20791     okume yapılmak istendiğinde ise EOF etkisi oluşur.
20792
20793     Pekiyi daemon'lar ne yaparlar? İşte daemon2lar arka planda genellikle →
20794         sürekli bir biçimde birtakım işler yapmaktadır. Bu anlamda →
20795         en tipik daemon örnekleri "server" programlardır. Örneğin http server →
20796             aslında httpd isimli bir daemon'dan ibarettir. Bunun gibi →
20797             UNIX/Linux sistemlerinde genellikle boot zamanında devreye giren onlarca →
20798                 daemon program vardır. Örneğin belli amanlarda belli işlerin →
20799                 yapılması için kullanılan cron utility'si aslında bir daemon olarak →
20800                     çalışmaktadır.
20801 -----
20802 -----*/
20803 #include <stdio.h>
20804 #include <stdlib.h>
20805 #include <fcntl.h>
20806 #include <sys/stat.h>
20807 #include <unistd.h>
20808
20809 void make_daemon(void)
20810 {
20811     pid_t pid;
20812     int maxfd;
20813     int fd;
20814     int i;
20815
20816     umask(0);
20817
20818     if ((pid = fork()) == -1) {
20819         perror("fork");
20820         exit(EXIT_FAILURE);
20821     }
20822
20823     if (pid != 0)
20824         exit(EXIT_SUCCESS);
20825
20826     if (setsid() == -1) {
20827         perror("setsid");
20828         exit(EXIT_FAILURE);
20829     }
20830
20831     if (chdir("/") == -1)
20832         exit(EXIT_FAILURE);
20833
20834     maxfd = sysconf(_SC_OPEN_MAX);
20835     for (i = 0; i < maxfd; ++i)
20836         close(i);
20837
```

```
20838     if ((fd = open("/dev/null", O_RDWR)) == -1)
20839         exit(EXIT_FAILURE);
20840
20841     if (dup(fd) == -1 || dup(fd) == -1)
20842         exit(EXIT_FAILURE);
20843 }
20844
20845 int main(void)
20846 {
20847     make_daemon();
20848
20849     /* daemon kodu, muhtemelen bir server program */
20850
20851
20852     return 0;
20853 }
20854
20855 /
*-----*
```

20856

20857 Pekiyi mademki daemon programlarının terminal ilişkileri yoktur, o zaman bu programlar dış dünyaya nasıl bildirimde bulunacaklardır? ↗

20858 İlk akla gelen yöntem önceden belirlenmiş bazı dosyalara yazmak olabilir. Ancak her daemon'ın kendi belirlediği dosyalara yazması ↗

20859 karışık bir durum oluşturabilmektedir. Bazı daemon'lar çok uzun süre çalışırlar bu da onların yazdıkları log'ların çok büyümESİNE yol açar. ↗

20860 UNIX türəvi sistemlerde daha genel ve merkezi bir log mekazisması düşünülmüştür. Bu mekanizma sayesinde farklı daemon'lar aynı log dosyasına ↗

20861 bildirimleri yazarlar. Böylece hem daha güvenli hem daha makul bir loglama sistemi oluşturulmuş olur. ↗

20862

20863 Merkezi loglama için üç temel POSIX fonksiyonu kullanılmaktadır: ↗

openlog, syslog ve closelog fonksiyonları. Log oluşturmadan önce ↗

20864 openlog fonksiyonu çağrılrak bazı belirlemeler yapılır. Fonksiyonun prototipi şöyledir: ↗

20865

```
20866 void openlog(const char *ident, int logopt, int facility);
```

20867

20868 Fonksiyonun birinci parametresi log mesajlarına görüntülenecek program ismini belirtmektedir. Genellikle programcılar bu parametre için ↗

20869 program ismini argüman olarak verirler. Linux sistemlerinde bu ↗

20870 parameytre NULL geçilebilmektedir. Bu durumda sanki bu parametre için ↗

program ismi yazılmış gibi işlem yapılır. Ancak POSIX standartlarında ↗

NULL geçme durumu belirtmemiştir. İkinci parametre aşağıdaki ↗

20871 sembolik sabitlerin bit or işlemine sokulmasıyla oluşturulabilir: ↗

20872

```
20873 LOG_PID
20874 LOG_CONS
20875 LOG_NDELAY
20876 LOG_ODELAY
```

20877 LOG_NOWAIT
20878
20879 Burada LOG_PID log mesajında prosesin proses id'sinin de
bulundurulacağını belirtir. LOG_CONS log mesajlarının aynı zamanda
default console (/dev/console) ➔
20880 da yazılacağını belirtmektedir. Bu parametre için argüman 0 olarak da
girilebilir. Fonksiyonun üçüncü parametresi log mesajını yollayan
prosesin kim olduğu ➔
20881 hakkında temel bir bilgi vermek için düşünülmüştür. Bu parametre
LOG_USER olarak girilebilir. LOG_USER bir user proses tarafından bu
loglamanın yapıldığını ➔
20882 belirtmektedir. LOG_KERN mesajın kernel tarafından gönderildiğini
belirtir. LOG_DAEMON mesajın bir sistem daemon programı tarafından
gönderildiğini ➔
20883 belirtmektedir. LOG_LOCAL0'dan LOG_LOCAL7'ye akadarki sembolik sabitler
özel log kaynaklarını belirtmektedir. Fonksiyon başarısız
olamamaktadır. bu nednele ➔
20884 geri dönüş değeri void yapılmıştır. Bu parametre de 0 geçilebilir. Bu
durumda LOLOCAL0 anlaşıılır. ➔
20885
20886 syslog fonksiyonu asıl log işlemini yapan fonksiyondur. Aslında syslog
için işin başında openlog çağrısının yapılmasına da gerek yoktur. Bu
durumda ➔
20887 default belirlemeler kullanılmaktadır. Fonksiyonun parametrik yapısı
şöyledir: ➔
20888
20889 void syslog(int priority, const char *format, ...);
20890
20891 Fonksiyonun birinci parametresi mesajın öncelik derecesini (yani
önemini) belirtir. Diğer paraetreler tamamen printf fonksiyonundaki
gibidir. ➔
20892 Öncelik değerlerleir şunlardır:
20893
20894 LOG_EMERG
20895 LOG_ALERT
20896 LOG_CRIT
20897 LOG_ERR
20898 LOG_WARNING
20899 LOG_NOTICE
20900 LOG_INFO
20901 LOG_DEBUG
20902
20903 En çok kullanılanlar error mesajları için LOG_ERR, uyarı mesajları için ➔
LOG_WARNING ve genel bilgilendirme mesajları için LOG_INFO
20904 değerleridir. syslog için openlog çağrılmak zorunda olmadığından syslog ➔
fonksiyonun birinci parametresi ile istenirse openlog fonksiyonunun
üçüncü parametresi kombine edilebilir.
20905
20906
20907 Nihayet eğer proses birmeden log sistemi prosese kapatılmak isteniyorsa ➔
closelog fonksiyonu çağrılmalıdır:
20908
20909 void closelog(void);
20910

```
20911     Aşağıda log fonksiyonlarının kullanımına ilişkin bir örnek verilmiştir
20912
20913 ----- */
20914
20915 #include <stdio.h>
20916 #include <stdlib.h>
20917 #include <syslog.h>
20918
20919 void exit_sys(const char *msg);
20920
20921 int main(void)
20922 {
20923     int i;
20924
20925     openlog("sample", LOG_PID, LOG_LOCAL0);
20926
20927     for (i = 0; i < 10; ++i)
20928         syslog(LOG_INFO, "This is a test: %d\n", i);
20929
20930     closelog();
20931
20932     return 0;
20933 }
20934
20935 void exit_sys(const char *msg)
20936 {
20937     perror(msg);
20938
20939     exit(EXIT_FAILURE);
20940 }
20941
20942 /
----- */
20943 Tabii log sistemi aslında tipik olarak daemon programları ve aygit
20944 sürücüler tarafından kullanılmaktadır. Aşağıda
20945 bu sistemi kullanan bir daemon örneği görüyorsunuz. Bu program
20946 çalışırken tail /var/log/syslog komutu ile syslog isimli
20947 log dosyasının sonuna bakabilirsiniz.
20948 ----- */
20949
20950 #include <stdio.h>
20951 #include <stdlib.h>
20952 #include <fcntl.h>
20953 #include <sys/stat.h>
20954 #include <unistd.h>
20955 #include <syslog.h>
20956
20957 void make_daemon(void)
20958 {
20959     pid_t pid;
```

```
20958     int maxfd;
20959     int fd;
20960     int i;
20961
20962     umask(0);
20963
20964     if ((pid = fork()) == -1) {
20965         perror("fork");
20966         exit(EXIT_FAILURE);
20967     }
20968
20969     if (pid != 0)
20970         exit(EXIT_SUCCESS);
20971
20972     if (setsid() == -1) {
20973         perror("setsid");
20974         exit(EXIT_FAILURE);
20975     }
20976
20977     if (chdir("/") == -1)
20978         exit(EXIT_FAILURE);
20979
20980     maxfd = sysconf(_SC_OPEN_MAX);
20981     for (i = 0; i < maxfd; ++i)
20982         close(i);
20983
20984     if ((fd = open("/dev/null", O_RDWR)) == -1)
20985         exit(EXIT_FAILURE);
20986
20987     if (dup(fd) == -1 || dup(fd) == -1)
20988         exit(EXIT_FAILURE);
20989 }
20990
20991 int main(void)
20992 {
20993     int i;
20994
20995     make_daemon();
20996
20997     openlog("sampled", LOG_PID, LOG_LOCAL0);
20998
20999     for (i = 0; i < 100; ++i) {
21000         syslog(LOG_INFO, "daemon running: %d\n", i);
21001         sleep(1);
21002     }
21003
21004     closelog();
21005
21006     return 0;
21007 }
21008
21009 /
```



21010 Pekiyi syslog log mesajlarını nereye yazmaktadır? Aslına log mesajlarını →
syslog fonksiyonun kendisi yazmaz. Log mesajlarının yazdırılması →
21011 için özel bir daemon prosesten faydalانılmaktadır. Bu proses eskiden →
syslogd ismindeydi, daha sonra bunun biraz daha gelişmiş versiyonu →
olan →
21012 rsyslogd prosesi kullanılmaya başlandı. Kursun yapıldığı zamanlarda →
Linux sistemlerinde yaygın olarak rsyslogd isimli daemon →
kullanılmaktadır.

21013 →
21014 syslogd ya da rsyslogd aslında diğer proseslerden proseslerarası →
haberleşmeyle yazdırılacak log mesajlarını almaktadır. Tipik olarak →
syslog fonksiyonu →
21015 /dev/log isimli soket dosyasını (datagram UNIX domain soket) kullanarak →
mesajları syslogd ya da rsyslogd daemon'ına göndermektedir. (Yani →
aslında syslogd ya da →
21016 rsyslogd datagram server görevi de yapmaktadır). Kernel'in kendisi de →
(örneğin printk fonksiyonunda) aslında neticede bu syslogd ya da →
rsyslogd daemon'ına →
21017 /dev/log soketi yoluyla datagram mesaj göndererek log mesajlarını →
yazdırmaktadır. Uzak bağlantı söz konusu olduğunda syslogd ya da →
rsyslogd datagram mesajları →
21018 UDP/IP 514 numaralı porttan almaktadır. İşte aslında log mesajlarının →
hangi dosyalara yazılacağına syslogd ya da rsyslogd daemon'ları karar →
vermektedir.

21019 Bu daemon'lar çalışmaya başladıklarında default durumda /etc/syslog.conf →
ya da /etc/rsyslog.conf dosyalarına bakmaktadır. İşte bu daemon'ların →
hangi dosyalara →
21020 yazacağı bu konfigürasyon dosyalarında sistem yönetici tarafından →
belirlenebilmektedir. Ancak bu dosyada da belirleme yapılmamışsa →
default olarak →
21021 pek çok mesaj grubu (error, warning, info) /var/log/syslog dosyasına →
yazılmaktadır. O halde programcı bu mesajlar için bu dosyaya →
başvurmalıdır.

21022 →
21023 Log dosyalarını incelemek için pek çok utility bulunmaktadır. Örneğin →
lnav, glogg ksystemlog gibi.

21024 →
21025 -----*/ →
21026 →
21027 /* -----*

21028 Yukarıda da belirtildiği gibi pek çok daemon aslında sistem boot →
edilirken çalıştırılmakta ve sistem kapatılana kadar çalışır durumda →
kalmaktadır.

21029 Fakat bazı daemon'lar ise gerektiğinde çalıştırılıp, gerekmemiğinde →
durdurulabilmektedir. İşte UNIX/Linux sistemlerinde bu çalıştırma, →
durdurma gibi falliyetler →
21030 için daha yüksek seviyeli araçlar bulundurulmaktadır. Bu araçlar init →
prosesinin paketinde yer alırlar. Tarihsel süreç içerisinde Linux →
sistemlerinde →

21031 boot işleminden ve servis işlemlerinden sorumlu üç önemli paket ➔
geliştirilmiştir:

21032

21033 systemVinit (klasik)

21034 upstart

21035 systemd

21036

21037 Kursun yapıldığı zaman diliminde ağırlıklı biçimde systemd paketi ➔
kullanılmaktadır. Sisteminizde hangi init paketinin kullanıldığını ➔
anlamak için

21038 birkaç yol söz konusu olabilir:

21039

21040 ls -l /sbin/init

21041 cat /proc/1/status

21042

21043 systemd init paketinin servis yönetici programı systemctl isimli ➔
programdır. Bu program yoluyla daemon işlemleri yapabilmek için ➔
öncelikle

21044 bir .service uzantılı dosyasının oluşturulması gereklidir. Bu programın ➔
çalıştırabileceği servislere "unit" denilmektedir. .service uzantılı ➔
dosyada da unit bildirimleri bulunur. Sonra bu dosyanın /etc/systemd/ ➔
system dizinine kopyalanması gerekmektedir. Ondan sonra asıl daemon ➔
programının da

21045 /usr/bin içeresine çekilmesi uygundur. (Tabii burada bazı seçenekler söz ➔
konusudur. Ancak biz tipik durumları ele alıyoruz. Ayrıntılı bilgi ➔
için systemd

21046 dokümantasyonuna bakınız.) Artık şu komutlar uygulanarak servis ➔
yönetilebilir:

21047

21048 systemctl start <daemon ismi> (daemon'ı çalıştırır)

21049 systemctl stop <daemon ismi> (daemon'ı durdurur)

21050 systemctl restart <daemon ismi> (durdurup yeniden başlatır)

21051 systemctl show <daemon ismi> (daemon'ın durumunu gösterir)

21052 systemctl enable <service dosyasının ismi> (boot zamanında devreye ➔
sokmak için)

21053 systemctl disable <service dosyasının ismi> (boot zamanında devreden ➔
çıkarmak için)

21054

21055 systemctl için tipik bir .service uzantılı dosyanın içeriği şöyledir:

21056

21057 [Unit]

21058 Description=Example Unit

21059 [Service]

21060 Type=forking

21061 ExecStart=/usr/bin/sampled

21062 [Install]

21063 WantedBy=multi-user.target

21064

21065 Daemon'lar işleme başlamadan önce çeşitli parametrik bilgileri bazı ➔
konfigürasyon dosyalarından (genellikle bunların uzantıları .conf ➔
olur) okuyabilmektedir.

21066 Sistem yöneticisi de bu dosyalarda değişiklik yapıp daemon'ı restart ➔
edebilmektedir.

21068
21069 -----*/
21070
21071 / -----*

21072 Farklı makinelerin prosesleri arasında haberleşme (yani ağ altında haberleşme) daha çetrefil bir haberleşme biçimidir.
21073 Çünkü burada ilgili işletim sisteminin dışında pek çok belirlemelerin önceden yapılmış olması gereklidir. İşte ağ haberleşmesinde önceden belirlenmiş kurallar topluluğuna "protokol" denilmektedir. Ağ haberleşmesi için tarihsel süreç içerisinde pek çok protokol gerçekletirilmiştir. Bunların bazıları büyük şirketlerin kontrolü altındadır. Ancak açık bir protokol olan IP protokol ailesi günümüzde hemen her zaman tercih edilen protokol ailesidir.

21077
21078 Protokol ailesi (protocol family) denildiğinde birbirleriyle ilişkili bir grup protokol anlaşılır. Ailenin pek çok protokülü başka protokollerin üzerine konumlandırılmıştır. Böylece protokol aileleri katmanlı (layered) bir yapıya sahip olmuştur.

21080 Üst seviye bir protokol alt seviye protokolün zaten var olduğu fikriyle o alt seviye protokol kullanılarak oluşturulur.

21081
21082 ISO Bir protokol ailesinde katmanlı olarak hangi tarzda protokollerin bulundurulabileceği yönelik OSI (Open System Interconnection) isimli bir referans dokümanı oluşturmuştur. Bunlara OSI katmanları denilmektedir. OSI'nin 7 katmanı vardır. Aşağıdaki yukarıya bunlar şöyledir: Phsical Layer, Data Link Layer, Network Layer, Transport Layer, Session Layer, Presentation Layer, Application Layer.

21085 En aşağı seviyeli elektriksel tanımlamaların yapıldığı katmana "fiziksel katman" denilmektedir. (Örneğin kabloların, konnektörlerin özellikleri vs.)

21086 Veri Bağlantı Katmanı artık bilgisayarlar arasında fiziksel bir adreslemenin yapıldığı ve bilgilerin paketlere ayrılarak gönderip alındığı bir ortam tanımlarlar. Örneğin bugün kullandığımız Ethernet kartları MAC adresi denilen dünya genelinde tek olan adresler yoluyla paket paket (packet switching)

21088 bilginin gönderip alınmasını sağlar. Bu yüzden Ethernet protokolü "veri bağlantı katmanına ilişkindir." Ağ Katmanı (Netword layer) artık "inernetworking" yapmak

21089 için gerekli kuralları tanımlar. "Internetworking" terimi network'lerden oluşan network'ler anlamına gelir. Tipik olarak internetworking yapmak için

21090 yerel ağlar (local area networks) "router" denilen aygıtlarla birbirine bağlanmaktadır. Ağ katmanında artık fiziksel bir adresleme değil, mantıksal adresleme

21091 sistemi kullanılmaktadır. Ayrıca bilgilerin paketlere ayrılarak router'lardan dolasılıp hedefe varması için rotalama mekanizması da bu katmanda tanımlanmaktadır.

-
- 21092 Yani elimizde yalnızca network katmanı varsa bi yalnızca ↗
"internetworking" ortamında belli bir kaynaktan hedefe bir paket ↗
yollayabiliriz. Transport katmanları
- 21093 network katmanlarının üzerindedir. Transpot katmanında artık kaynak ile ↗
hedef arasında bir bağlantı oluşturulabilmekte ve veri aktarımı daha ↗
güvenli
- 21094 olarak yapılmaktadır. Aynı zamanda transport kavramı "multiplex" bir ↗
kaynak hedef yapısı da oluşturmaktadır. Bu sayede hedefe bilgiler ↗
oradaki spesifik bir
- 21095 programa gönderilebilmektedir. Oturum katmanı pek çok ailedе yoktur. ↗
Görevi oturum açma kapama gibi yüksek seviyeli bazı belirlemeleri ↗
yapmaktadır. Presentation
- 21096 katmanı verilerin sıkıştırılması, şifrelenmesi gibi tanımlamalar ↗
icermektedir. Nihayet bu protokolü kullanan bütün programlar aslında ↗
uygulama katmanını oluşturmaktadır.
- 21097 Yani ağ ortamında haberleşen her program zaten kendi içerisinde açık ya ↗
da gizli bir protokol oluşturmuş durumdadır.
- 21098
- 21099 IP ailesi neden bu kadar popüler olmuştur? Bunun en büyük nedeni 1983 ↗
yılında hepimizin katıldığı Internet'in (I'nin büyük yazıldığına ↗
dikkat ediniz)
- 21100 bu aileyi kullanmaya başlamasındadır. Böylece IP ailesini kullanarak ↗
yazdığımız programlar hem aynı bilgisayarda hem yerel ağımızdaki ↗
bilgisayarlarda hem de
- 21101 Internet'te çalışabilmektedir. Aynı zamanda IP ailesinin açık bir (yani ↗
bir şirketin malı değil) protokol olması da cazibeyi çok artırmıştır.
- 21102
- 21103 IP ailesi 70'li yıllarda Vint Cerf ve Bob Kahn tarafından ↗
geliştirilmiştir. IP ismi Internet Protocol'den gelmektedir. Burada ↗
internet "internetworking"
- 21104 anlamında kullanılmıştır.
- 21105
- 21106 Bugün hepimizin bağlandığı büyük ağa da "Internet" denilmektedir. Bu ağ ↗
ilk kez 1969 yılında Amerika'da Amerikan Savunma Bakanlığının bir ↗
soğuk savaş
- 21107 projesi biçiminde başlatıldı. O zamana kadar yalnızca yerel ağlar vardı. ↗
1969 yılında ilk kez bir "WAN (Wide Area Network)" oluşturuldu.
- 21108 Bu proje Amerikan savunma bakanlığının DARPA isimli araştırma kurumu ↗
tarafından başlatılmıştır ve ARPA.NET ismi verilmiştir. Daha bu ağa ↗
Amerika'daki
- 21109 çeşitli devlet kurumları ve üniversiteler katıldı. Sonra ağ Avrupa'ya ↗
sıçradı. 1983 yılında bu ağ NCP protokolünden IP protokol ailesine ↗
geçiş yaptı.
- 21110 Bundan sonra artık APRA.NET ismi yerine "Internet" ismi kullanılmaya ↗
başlandı.
- 21111
- 21112 IP protokol ailesi 4 katmanlı bir ailedir. Fiziksel ve Veri Bağlantı ↗
Katmanı bir arada düşünülebilir. Bugün bunlar Ethernet ve Wireless ↗
protokoller
- 21113 biçiminde pratikte kullanılmaktadır. IP ailesinin ağ katmanı aileye ↗
ismini veren IP protokolünden oluşmaktadır. Transport katmanı ise TCP ↗
ve UDP
- 21114 protokollerinden oluşur. Nihayet TCP üzerine oturtulmuş olan HTTP, ↗

TELNET, SSH, POP3, IMAP gibi pek çok protokol ailenin uygulama katmanını oluşturmaktadır.

21115 IP protokolü tek başına kullanılırsa ancak bir paket gönderip alma işini yapar. Bu nedenle bu protokolün tek başına kullanılması çok seyrekтир.

21116 Uygulamada genellikle trasport katmanına ilişkin TCP ve UDP protokollerini kullanılmaktadır.

21117 TCP "stream tabanlı", UDP "datagram (paket) tabanlı" bir protokoldür. Stream tabanlı demek tamamen boru haberleşmesinde olduğu gibi gönderen tarafın bilgilerinin bir kuyruk sistemi eşliğinde alıcıda organize edilmesi ve alıcının istediği kadar byte'ı parça parça okuyabilmesi demektir. Datagram tabanlı demek tamamen mesaj kuyruklarında olduğu gibi bilginin paket paket iletilmesi demektir. Yani datagram haberleşmede alıcı taraf gönderen tarafın tüm paketini tek hamlede almak zorundadır.

21118 21119 21120 21121 21122 21123 21124 21125 21126 21127 21128 21129 21130 21131 21132 21133 21134 21135 21136 21137 21138 21139 21140 TCP bağlantılı (connection-oriented) UDP bağlantısız (connectionless) bir protokoldür. Buradaki bağlantı IP paketleriyle yapılan mantıksal bir bağlantıdır. Bağlantı sırasında gönderici ve alıcı birbirlerini tanır ve haberleşme boyunca konuşabilirler. Oysa UDP'de alıcı onun hiç bilmemiği bir kullanıcidan bilgi alabilmektedir. UDP'de gönderen ve alan arasında bir kayıt turulmaz.

TCP güvenilir (reliable) UDP (güvenilir) olmayan (unreliable) bir protokoldür. TCP'de mantıksal bir bağlantı oluşturulduğu için yolda kaybolan paketlerin telafi edilmesi mümkündür. Alıcı taraf gönderenin bilgilerini eksiksiz ve bozulmadan aldığıni bilir.

IP protokol ailesinde ağa bağlı olan birimlere "host" denilmektedir. Host bir bilgisayar olmak zorunda değildir. İşte bu protokolde her host'un mantıksal bir adresi vardır. Bu adres IP adresi denilmektedir. IP adresi IPV4'te 4 byte uzunlukta, IPV6'da 16 byte uzunluktadır. Ancak bir host'ta farklı programlar farklı host'lara haberleşiyor olabilir. İşte aynı host'a gönderilen IP paketlerinin o host'ta ayrıştırılması için "protokol port numarası" diye içsel bir numara uydurulmuştur. Port numarası bir şirketin içerisinde çalışanların dahili numarası gibi düşünülebilir. Port numaraları IPV4'te 2 byte'la, IPV6'da 4 byte'la ifade edilmektedir. İlk 1024 port numarası IP ailesinin uygulama katmanındaki protokoller için ayrılmıştır. Bunlara "well known ports" denilmektedir. Bu nedenle programcılar port numaralarını 1024'ten büyük olacak biçimde almaları gereklidir. Bu durumda TCP ve UDP'de bilgiler belirli bir IP adresindeki host'un belirli bir portuna gönderilir. Gönderilen bu bilgiler de o portla ilgilenen programlar tarafından alınmaktadır.

21141 IP haberleşmesi (yani paketlerin, oluşturulması, gönderilmesi alınması vs.) işletim sistemlerinin çekirdekleri tarafından yapılmaktadır.

21142 Tabii User mod programlar için sistem çağrılarını yapana API fonksiyonlarına ve kütüphanelerine gereksinim vardır. İşte bunların en yaygın

21143 kullandıkları "soket kütüphanesi" denilen kütüphanedir. Bu kütüphane ilk kez 1983 yılında 4.2BSD'de gerçekleştirilmiştir ve pek çok UNIX türevi

21144 sistem bu kütüphaneyi aynı biçimde benimsemiştir. Microsoft'un Windows sistemleri de bu API kütüphanesini desteklemektedir. Bu kütüphaneye "Winsock"

21145 ya da "WSA (Windows Socket API)" denilmektedir. Microsoft'un Winsock kütüphanesinde hem BSD fonksiyonları orijinal haliyle bulunmakta hem de

21146 başı WSAXXX ile başlayan Wibndows'a özgü fonksiyonlar bulunmaktadır.

21147 -----*/

21148 / -----*

21149 -----*

21150 Bir TCP/Ip uygulmasında server ve client olmak üzere iki ayrı program yazılır. Server program şu fonksiyonlar çağrılarak olulturulmaktadır:

21151 socket, bind, listen, accept, read/write/recv/send, shutdown, close

21152 socket fonksiyonu bir handle alanı yaratır ve bize bir dosya betimleyicisi verir. Biz diğer fonksiyonlarda soket dediğimiz bu betimleyiciyi kullanırız.

21153 int socket(int domain, int type, int protocol);

21154 Fonksiyonun birinci parametresi kullanılacak protokol ailesini belirtir. Bu parametre AF_XXX biçimindeki sembolik sabitlerden biri

21155 olarak girilir. IPV4 için bu parametreye AF_INET, IPV6 için AF_INET6 girilmelidir. UNIX protokolü için AF_UNIX kullanılır. İkinci

21156 parametre kullanılacak protokolün stream mi datagram mi ya da başka bir türden mi olduğunu belirtir. Stream soketler için SOCK_STREAM,

21157 datagram soketler için SOCK_DGRAM kullanılmalıdır. Başka soket türleri de vardır. Üçüncü parametre transport katmanındaki protokolü belirtmektedir.

21158 Ancak zaten ikinci parametreden transport protokolü anlaşılıyorsa üçüncü parametre 0 geçilebilir. Örneğin IP ailesinde üçüncü parametreye

21159 gerek duyulmamaktadır. Çünkü SOCK_STREAM zaten TCP'yi SOCK_DGRAM ise UDP'yi anlatmaktadır. Fakat yine de bu parametreye istenirse IP ailesi

21160 için IPPROTO_TCP ya da IPPROTO_UDP girilebilir. (Bu sembolik sabitler <netinet/in.h> içerisindeindedir.) Fonksiyon başarılıysa soket betimleyicisine

21161 başarısızsa -1 değerine geri döner.

21162 Server program soketi yarattıktan sonra onu bağlamalıdır (bind etmelidir). bind işlemi sırasında server'in hangi portu dinleyeceği

21163 ve hangi network arayüzünden (kartından) gelen bağlantı isteklerini

21170 kabul edecek bir yapı oluşturur.

21171 int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

21172 Fonksiyonun birinci parametresi yaratılmış olan soket betimleyicisini alır. İkinci parametre her ne kadar sockaddr isimli yapı türündense de aslında her protokol için ayrı bir yapı adresini almaktadır. Yani sockaddr yapısı genellikle (void gösterici gibi) temsil etmek için kullanılmıştır. IPV4 için kullanılacak yapı sockaddr_in IPV6 için, sockaddr_in6 ve örneğin Unix domain soketler için ise

21173 sockaddr_un biçiminde olmalıdır. Üçüncü parametre ikinci parametredeki yapıının uzunluğu olarak girilmelidir.

21174

21175

21176

21177

21178 sockaddr_in yapısı şöyledir:

21179

21180 struct sockaddr_in {

21181 sa_family_t sin_family;

21182 in_port_t sin_port;

21183 struct in_addr sin_addr;

21184 };

21185

21186 Yapının sin_family elemanına protokol ailesini belirten AF_XXX değeri girilmelidir. Bu eleman tipik olarak short biçimde bildirilmiştir.

21187 Yapının sin_port elemanı in_port_t elemanı türündendir ve bu tür uint16_t olarak typedef edilmiştir. Bu eleman server'in dinleyeceği port numarasını belirtir. Yapının sin_addr elemanı IP numarası belirten bir elemandır. Bu eleman in_addr isimli bir yapı türündendir.

21188

21189 Bu yapı da şöyle bildirilmiştir:

21190

21191 struct in_addr {

21192 in_addr_t s_addr;

21193 };

21194

21195 in_addr_t 4 byte'lık işaretsız tamsayı türünü (uint32_t) belirtmektedir. Böylece s_addr 4 byte'lık IP adresini temsil eder.

21196

21197 IP ailesinde tüm sayısal değerler BIG ENDIAN formatıyla belirtilmek zorundadır. Bu ailedede "network byte ordering" denildiğinde BIG ENDIAN anlaşılır. Oysa makinelerin belli bir bölümü (örneğin INTEL ve default ARM) LITTLE ENDIAN kullanmaktadır. İşte elimzdeki makinenin endianlığı ne olursa olsun onu BIG ENDIAN'a dönüştüren htons (host to network byte ordering short) ve htonl (host to network byte ordering long) isimli bir iki fonksiyon vardır. Bu işlemlerin tersini yapan ntohs ve ntohl fonksiyonları da bulunmaktadır. IP adresi olarak INADDR_ANY özel bir değerdir ve "tüm network kartlarından gelen bağlantı isteklerini kabul et" anlamına gelir. Bu durumda sockaddr_in yapısı tipik olarak şöyle doldurulabilir:

21198

21199

21200

21201

21202

21203

21204 struct sockaddr_in sinaddr;

21205

21206 sinaddr.sin_family = AF_INET;

```
21207     sinaddr.sin_port = htons(SERVER_PORT);
21208     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
21209
21210     server bind işleminden sonra soketi aktif dinleme konumuna sokmak için →
21211         listen fonksiyonunu çağrırmalıdır. Fonksiyonun prototipi şöyledir:
21212
21213     int listen(int socket, int backlog);
21214
21215     Fonksiyonun birinci parametresi soketin handle değeri, ikinci →
21216         parametresi kuyruk uzunluğunu belirtir. listen işlemi blokeye →
21217         yol açmamaktadır. İşletim sistemi listen işleminden sonra ilgili porta →
21218         gelen bağlantı isteklerini uygulama için bir kuyruk sisteminde →
21219         biriktirir. accept fonksiyonu bu kuyruğa bakmaktadır. Kuyruk uzunluğunu →
21220         yüksek tutmak meeşgul server'larda bağlantı isteklerinin →
21221         kaçırılmamasını →
21222         sağlanabilir. Linux'ta default durumda verilebilecek en yüksek değer →
21223         128'dir. Ancak /proc/sys/net/core/somaxconn dosyasındaki →
21224         değer yükseltilebilir. Fonksiyon başarı durumunda 0 değerine →
21225         başarısızlık durumunda -1 değerine geri döner. Bu fonksiyon işletim →
21226         sisteminin "firewall mekanizması" tarafından denetlenmektedir.
21227
21228     Nihayet asıl bağlantı accept fonksiyonuyla sağlanmaktadır. accept →
21229         fonksiyonu bağlantı kuyruğuna bakar. Eğer orada bir bağlantı isteği →
21230         varsa →
21231         onu alır ve hemen geri döner. Eğer oarada bir bağlantı isteği yoksa →
21232         default durumda blokede bekler. Fonksiyonun prototipi şöyledir:
21233
21234     int accept(int socket, struct sockaddr *address, socklen_t →
21235         *address_len);
21236
21237     Fonksiyonun birinci parametresi soketin dosya betimleyicisini →
21238         almaktadır. İkinci parametre bağlanılan client'a ilişkin bilgilerin →
21239         yerleştirileceği yapının adresini almaktadır. Tabii bu yapı yine →
21240         protokole göre değişebilen bir yapıdır. Örneğin IPV4 için bu yapı →
21241         yine sockaddr_in olmalıdır. Bu yapının içinden biz bağlanılan client'in →
21242         ip adresini, kaynak port numarasını alabiliriz.
21243
21244     Bu bilgiler yine BIG ENDIAN formattadır. accpry fonksiyonu başarı →
21245         durumunda client ile konuşma içinde kullanılacak soket değerine →
21246         (soket dosya betimleyicisine) başarısızlık durumunda 0 değerine geri →
21247         dönmektedir.
21248
21249     Server programın listen ve accept işlemlerini yapmak için kullandığı →
21250         sokete "pasif soket" ya da "dinleme soketi" denilmektedir.
21251     Bu pasif soket başka bir amaçla kullanılamaz. Client'larla konuşmak için →
21252         accept fonksiyonun verdiği soketler kullanılır.
21253
21254     Aşağıda accept işlemine kadar tipik bir server örneği verilmiştir. Bu →
21255         program dinlenecek prot numarasını komut satırı argümanıyla almaktadır.
21256
21257     -----
21258     -----
21259 #include <stdio.h>
```

```
21240 #include <stdlib.h>
21241 #include <unistd.h>
21242 #include <sys/socket.h>
21243 #include <netinet/in.h>
21244 #include <arpa/inet.h>
21245
21246 void exit_sys(const char *msg);
21247
21248 int main(int argc, char *argv[])
21249 {
21250     int sock, sock_client;
21251     struct sockaddr_in sinaddr, sinaddr_client;
21252     socklen_t sinaddr_len;
21253     in_port_t port;
21254
21255     if (argc != 2) {
21256         fprintf(stderr, "wrong number of arguments!..\n");
21257         exit(EXIT_FAILURE);
21258     }
21259
21260     port = (in_port_t)strtoul(argv[1], NULL, 10);
21261
21262     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
21263         exit_sys("socket");
21264
21265     sinaddr.sin_family = AF_INET;
21266     sinaddr.sin_port = htons(port);
21267     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
21268
21269     if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
21270         exit_sys("bind");
21271
21272     if (listen(sock, 8) == -1)
21273         exit_sys("listen");
21274
21275
21276     printf("Waiting for connection...\n");
21277
21278     sinaddr_len = sizeof(sinaddr_client);
21279     if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,
21280                               &sinaddr_len)) == -1)
21281         exit_sys("accept");
21282
21283     printf("Connected: %s : %u\n", inet_ntoa(sinaddr_client.sin_addr),
21284           (unsigned)ntohs(sinaddr_client.sin_port));
21285
21286     /* other stuff */
21287
21288     return 0;
21289 }
21290 void exit_sys(const char *msg)
21291 {
```

```
21291     perror(msg);
21292
21293     exit(EXIT_FAILURE);
21294 }
21295
21296 /
*-----  
-----  

21297 Client program tipik olarak şu aşamalardan geçerek yazilmaktadır:
21298     socket, bind (optional), gethostbyname (optional), connect,
21299     read/write/recv/send, shutdown, close.
21300
21301 Client program yine socket fonksiyonuyla bir soket yaratır. İsterse bu
21302     soketi bind eder. Client bind işlemi yapmak zorunda değildir.
21303 Ancak client'in server'a belirli bir kaynak porttan bağlanması
21304     isteniyorsa client bu kaynak portu belirlemek için bind işlemi
21305     yapmalıdır.
21306 Eğer client bind işlemi yapmazsa işletim sistemi belli bir aralıkta boş
21307     bir kaynak port numarasını tahsis eder. Client program server'in
21308 ip adresini ve port numarasını bilerek ona bağlanacaktır. Ancak IP
21309     adresleri akılda zor tutulduğu için IP adreslerine isimler karşı
21310     düşürülmüştür.
21311 Pratikte daha çok bu isimler kullanılmaktadır. connect fonksiyonun
21312     prototipi şöyledir:
21313
21314 int connect(int socket, const struct sockaddr *address, socklen_t
21315     address_len);
21316 Fonksiyonun birinci parametresi soket betimleyicisi alır. İkinci
21317     parametre yine IPV4 için sockaddr_in IPV6 için sockaddr_in6 türünden
21318 bir yapının adresini alır. Böylece IPV4 için programcı bir sockaddr_in
21319     yapısı alıp bağlanacağı server'in ip adresini ve port numarasını
21320 bu yapıya yerleştirir. Fonksiyonun son parametresi ikinci parametredeki
21321     nesnenin byte uzunluğunu almaktadır. Fonksiyon başarı durumunda 0,
21322     başarısızlık durumunda -1 değerine geri dönmektedir.
21323
21324 IP adreslerinin sockaddr_in yapısına BIG ENDIAN (network byte ordering)
21325     biçiminde girilmesi gerektiğini anımsayınız. Bu durumda örneğin
21326 a.b.c.d biçimindeki bir IP adresi htonl(a << 24 | b << 16 | c << 8 | d)
21327     biçiminde oluşturulabilir. Ancak bunun yerine bu dönüşümü yapana
21328 inet_addr isimli bir fonksiyon bulundurulmuştur. Bu fonksiyon "a.b.c.d"
21329     biçiminde yazılı olarak verilen IP adresini parse ederek BIG ENDIAN
21330 formata dönüştürmektedir. Eğer noktalı formdaki (dotted decimal form) IP
21331     adresi yazılış girilmişse fonksiyon INADDR_NONE değerine geri
21332     dönüktedir.
21333 Bu işlemin tersini yapan inet_ntoa isimli bir fonksiyon vardır. Bu iki
21334     fonksiyon IPV4'te kullanılabilmektedir. Daha sonraları IPV6'da da
21335     kullanılabilecek
21336     biçimde yeni fonksiyonlar inet_pton ve inet_aton fonksiyonlarıdır.
21337
21338 Genellikle client programlar server'in IP adresini ya da host ismini
21339     (domain ismini) alarak çalışacak biçimde yazılırlar. İşte programcının
```

21322 girilen değerin öncelikle "noktalı desimal formda (dotted decimal form)" \Rightarrow
olup olmadığını kontrol etmesi gereklidir. Zaten inet_addr fonksiyonu bu \Rightarrow
kontrolü yapabilmektedir. O halde girilen yazı önceler inet_addr fonksiyonuna \Rightarrow
sokulmalı eğer oradan INADDR_NONE değeri elde ediliyorsa artık girilen \Rightarrow
yazının IP adresi olmayacağı anlaşılır. İşte host isimlerinin IP adreslerine dönüştürülmesi \Rightarrow
için IP protokol ailesinde "DNS protokolü" denilen özel bir protokol \Rightarrow
bulunmaktadır.

21325 İsimlerin IP adres karşılıkları "Domain Name Server" denilen özel \Rightarrow
server'larda tutulur. Client program bu server'lara danışarak ismini IP \Rightarrow
numarasına dönüştürür. Bu işlem için IPV4'te gethostbyname isimli fonksiyon \Rightarrow
kullanılıyordu. Daha sonra IPV6'yi da kapsayacak biçimde getnameinfo \Rightarrow
ve getaddrinfo fonksiyonları oluşturdu. gethostbyname fonksiyonun prototipi şöyledir:

21328
21329 struct hostent *gethostbyname(const char *name);
21330
21331 Fonksiyon parametre olarak host ismini alır, ve hostent isimli bir yapı \Rightarrow
adresine geri döner. hostent yapısı şöyledir:

21332
21333 struct hostent {
21334 char *h_name;
21335 char **h_aliases;
21336 int h_addrtype;
21337 int h_length;
21338 char **h_addr_list;
21339 };
21340
21341 Bit host ismine karşı birden fazla IP adresi olabileceği gibi, bir IP \Rightarrow
adresine karşı da birden fazla host ismi olabilmektedir. Yapının
21342 h_addr_list elemanı her biri 4 char'dan oluşan dört elemanlı dizilerin \Rightarrow
adreslerini tutmaktadır. Bunlar host ismine karşı gelen IP \Rightarrow
adresleridir.
21343 Bu gösterici dizisinin sonunda NULL adres vardır.
21344
21345 Aşağıdaki programda anlatılan yere kadar bir client program iskeleti \Rightarrow
verilmiştir. Program komut satırı argümanı olarak server'in
21346 ip adresini (ya da host ismini) ve port numarasını alır ve server'a TCP \Rightarrow
ile bağlanır.

21347
21348 -----*/
21349
21350 #include <stdio.h>
21351 #include <stdlib.h>
21352 #include <string.h>
21353 #include <unistd.h>
21354 #include <sys/socket.h>
21355 #include <netinet/in.h>
21356 #include <arpa/inet.h>

```
21357 #include <netdb.h>
21358
21359 void exit_sys(const char *msg);
21360
21361 int main(int argc, char *argv[])
21362 {
21363     int sock;
21364     struct sockaddr_in sinaddr;
21365     struct hostent *hent;
21366     in_port_t port;
21367
21368     if (argc != 3) {
21369         fprintf(stderr, "wrong number of arguments!..\\n");
21370         exit(EXIT_FAILURE);
21371     }
21372
21373     port = (in_port_t)strtol(argv[2], NULL, 10);
21374
21375     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
21376         exit_sys("socket");
21377
21378     sinaddr.sin_family = AF_INET;
21379     sinaddr.sin_port = htons(port);
21380
21381     if ((sinaddr.sin_addr.s_addr = inet_addr(argv[1])) == INADDR_NONE) {
21382         if ((hent = gethostbyname(argv[1])) == NULL)
21383             exit_sys("gethostbyname");
21384         memcpy(&sinaddr.sin_addr.s_addr, hent->h_addr_list[0], hent-
21385             >h_length);
21386     }
21387
21388     if (connect(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
21389         exit_sys("connect");
21390
21391     printf("Connected...\\n");
21392
21393     /* other stuff */
21394
21395 }
21396
21397 void exit_sys(const char *msg)
21398 {
21399     perror(msg);
21400
21401     exit(EXIT_FAILURE);
21402 }
21403
21404 /
*-----*
-----*
21405 Client taraf soketi yarattıktan sonra kaynak port numarasının belli bir
değerde olmasını sağlayabilir. Bunun için client
```

```
21406     tarafın da bind işlemi uygulaması gereklidir. Aşağıdaki örnekte client →
21407     taraf 5051 numaralı port'a bind işlemi yapmıştır.
21408     -----
21409     -----*/
21410
21411 #include <stdio.h>
21412 #include <stdlib.h>
21413 #include <string.h>
21414 #include <unistd.h>
21415 #include <sys/socket.h>
21416 #include <netinet/in.h>
21417 #include <arpa/inet.h>
21418 #include <netdb.h>
21419
21420 void exit_sys(const char *msg);
21421
21422 int main(int argc, char *argv[])
21423 {
21424     int sock;
21425     struct sockaddr_in sinaddr;
21426     struct hostent *hent;
21427     in_port_t port;
21428
21429     if (argc != 3) {
21430         fprintf(stderr, "wrong number of arguments!..\n");
21431         exit(EXIT_FAILURE);
21432     }
21433
21434     port = (in_port_t)strtol(argv[2], NULL, 10);
21435
21436     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
21437         exit_sys("socket");
21438
21439     sinaddr.sin_family = AF_INET;
21440     sinaddr.sin_port = htons(5051);
21441     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
21442
21443     if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
21444         exit_sys("bind");
21445
21446     sinaddr.sin_family = AF_INET;
21447     sinaddr.sin_port = htons(port);
21448
21449     if ((sinaddr.sin_addr.s_addr = inet_addr(argv[1])) == INADDR_NONE) {
21450         if ((hent = gethostbyname(argv[1])) == NULL)
21451             exit_sys("gethostbyname");
21452         memcpy(&sinaddr.sin_addr.s_addr, hent->h_addr_list[0], hent-
21453             >h_length);
21454     }
21455
21456     if (connect(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
21457         exit_sys("connect");
21458
21459 }
```

```
21456     printf("Connected...\n");
21457
21458     /* other stuff */
21459
21460
21461     return 0;
21462 }
21463
21464 void exit_sys(const char *msg)
21465 {
21466     perror(msg);
21467
21468     exit(EXIT_FAILURE);
21469 }
21470
21471 /
*-----*
```

21472 Bağlantı sağlandıktan sonra artık iki taraf da birbirlerine bilgi
gönderip alabilirler (full duplex). Bilgi gönderip alma
aslında bir tamponlama (bufffering) mekanizmasıyla
gerçekleştirilmektedir. Her iki tarafta da işletim sistemi "gönderme
tamponu (send buffer)"
ve "alma tamponu (receive buffer)" isminde iki tampon bulundurur. Biz
bir soketle karşı tarafa bilgi göndermek istediğimizde aslında
göndermek
istediğimiz bilgiler önce kendi bilgisayarımızın gönderme taponuna (send
buffer) yazılmaktadır. Bu gönderme tamponuna yazılan bilgiler
işletim sistemi tarafından TCP ve dolayısıyla IP paketlerine
dönüştürülp uygun bir zamanda gerçekten gönderilmektedir. Benzer
biçimde
aslında bilgisayarımıza gelen paketler kesme (interrupt) mekanizması
yoluyla işletim sistemi tarafından alma tamponuna yerleştirilmektedir.

21473 21474 21475 21476 21477 21478 21479 21480 21481 21482 21483 21484 21485 21486 21487 21488

21472 Bağlantı sağlandıktan sonra artık iki taraf da birbirlerine bilgi
gönderip alabilirler (full duplex). Bilgi gönderip alma
aslında bir tamponlama (bufffering) mekanizmasıyla
gerçekleştirilmektedir. Her iki tarafta da işletim sistemi "gönderme
tamponu (send buffer)"
ve "alma tamponu (receive buffer)" isminde iki tampon bulundurur. Biz
bir soketle karşı tarafa bilgi göndermek istediğimizde aslında
göndermek
istediğimiz bilgiler önce kendi bilgisayarımızın gönderme taponuna (send
buffer) yazılmaktadır. Bu gönderme tamponuna yazılan bilgiler
işletim sistemi tarafından TCP ve dolayısıyla IP paketlerine
dönüştürülp uygun bir zamanda gerçekten gönderilmektedir. Benzer
biçimde
aslında bilgisayarımıza gelen paketler kesme (interrupt) mekanizması
yoluyla işletim sistemi tarafından alma tamponuna yerleştirilmektedir.

Biz soketten okuma yapmak istediğimizde bu tampona yerleştirilmiş
olanları okuruz. Pekiyi alma tamponu dolarsa ve biz hiç okuma
yapmazsak
ne olur? İşte TCP protokolü (ama IP değil) akış kontrolüne (flow
control) sahiptir. Akış kontrolü tampon taşmasını engellemek için iki
tarafın
konuşarak birbirlerini gereğiğinde durdurması anlamına gelmektedir.
TCP'de soket arayüzü kullanarak bilgi göndermek için write ya da send
fonksiyonları kullanılmaktadır. send fonksiyonunun write
fonksiyonundan
fazla bir flags parametresi vardır.
ssize_t write(int fd, const char *buf, size_t size);
ssize_t send(int fd, const void *buffer, size_t ssize, int flags);
Soketten bilgi okumak için ise read ya da recv fonksiyonları
kullanılmaktadır. Yine aslında recv fonksiyonu read fonksiyonundan
farklı

```
21489     olarak bir flags parametresine sahiptir:  
21490  
21491     ssize_t read(int fd, void *buf, size_t count);  
21492     ssize_t recv(int fd, void *buf, size_t len, int flags);  
21493  
21494     send fonksiyonunun flags parametresi 0 geçilirse, recv fonksiyonunun da →  
21495         flags parametresi 0 geçilirse bunların write ve read'ten  
21496         hiçbir farklılığı kalmamaktadır.  
21497  
21498         write ya da send fonksiyonu ile bilgi gönderilirken aslında bilgi →  
21499             "gönderme tamponuna yazılıp" hemen bu fonksiyonlar geri dönerler. →  
21500             write ya da send  
21501             fonksiyonları POSIX standartlarına göre normal olarak tüm bilgi →  
21502                 gönderme tamponuna yazılıana kadar blokeye yol açmaktadır. Örneğin biz →  
21503                 bu fonksiyonlarla  
21504             100 byte göndermek isteyelim ancak gönderme tamponunda 90 byte'lık yer →  
21505                 kalmış olsun. Bu durumda write ya da send fonksiyonları bu 100 →  
21506                 byte'in tamamı yazılıana  
21507             kadar blokeye yol açmaktadır. (send fonksiyonunun bu davranışını Windows →  
21508                 sistemlerinde değişiklik gösterebilir. Bu sistemlerde send →  
21509                 fonksiyonu bilginin tamamı tampona aktarılana kadar blokeye yol açmak →  
21510                 zorunda değildir. Bu sistemlerde send fonksiyonu yazıldığı kadar →  
21511                 byte'i tampona  
21512             yazıp yazıldığı byte sayısına geri dönebilmektedir.)  
21513  
21514             read ya da recv fonksiyonları "alma tamponuna" bilmektedir. Bu tampon →  
21515                 tamamen boş ise bunlar blokeli modda (default durum) en az 1 byte →  
21516                 okuyana  
21517             kadar beklerler. Ancak eğer tamponda en az 1 byte'lük bilgi varsa bu →  
21518                 fonksiyonlar blokeye yol açmazlar okuyabildikleri kadar byte'ı →  
21519                 okuyarak okuyabildikleri  
21520             byte sayısına geri dönerle.  
21521  
21522             TCP/IP haberleşmede önemli bir durum vardır: Bir tarafın tek bir write/ →  
21523                 send ile gönderdiği gilgiyi diğer taraf tek bir read/recv ile →  
21524                 okuyamayabilir.  
21525             Çünkü write/send yapan taraf bu bilgileri önce gönderme tamponuna →  
21526                 yerleştirir. Buradaki bilgiler işletim sistemi tarafından farklı IP →  
21527                 paketleri ile  
21528             iletilebilir. Bunun sonucunda alıcı taraf bunları parça parça alabilir.  
21529             -----*/  
21530  
21531             /* server.c */  
21532  
21533             #include <stdio.h>  
21534             #include <stdlib.h>  
21535             #include <string.h>  
21536             #include <unistd.h>  
21537             #include <sys/socket.h>  
21538             #include <netinet/in.h>  
21539             #include <arpa/inet.h>  
21540  
21541
```

```
21522 #define BUFFER_SIZE      1024
21523
21524 char *revstr(char *str);
21525 void exit_sys(const char *msg);
21526
21527 int main(int argc, char *argv[])
21528 {
21529     int sock, sock_client;
21530     struct sockaddr_in sinaddr, sinaddr_client;
21531     socklen_t sinaddr_len;
21532     in_port_t port;
21533     ssize_t result;
21534     char buf[BUFFER_SIZE + 1];
21535
21536     if (argc != 2) {
21537         fprintf(stderr, "wrong number of arguments!..\n");
21538         exit(EXIT_FAILURE);
21539     }
21540
21541     port = (in_port_t)strtoul(argv[1], NULL, 10);
21542
21543     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
21544         exit_sys("socket");
21545
21546     sinaddr.sin_family = AF_INET;
21547     sinaddr.sin_port = htons(port);
21548     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
21549
21550     if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
21551         exit_sys("bind");
21552
21553     if (listen(sock, 8) == -1)
21554         exit_sys("listen");
21555
21556     printf("Waiting for connection...\n");
21557
21558     sinaddr_len = sizeof(sinaddr_client);
21559     if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,
21560                               &sinaddr_len)) == -1)
21561         exit_sys("accept");
21562
21563     printf("Connected: %s : %u\n", inet_ntoa(sinaddr_client.sin_addr),
21564           (unsigned)ntohs(sinaddr_client.sin_port));
21565
21566     for (;;) {
21567         if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
21568             exit_sys("recv");
21569         if (result == 0)
21570             break;
21571         buf[result] = '\0';
21572         if (!strcmp(buf, "quit"))
21573             break;
21574         printf("%ld bytes received from %s (%u): %s\n", (long)result,
```

```
                inet_ntoa(sinaddr_client.sin_addr),
21573            (unsigned)ntohs(sinaddr_client.sin_port), buf);
21574        revstr(buf);
21575        if (send(sock_client, buf, strlen(buf), 0) == -1)
21576            exit_sys("send");
21577    }
21578
21579    /* Other stuff */
21580
21581    return 0;
21582 }
21583
21584 char *revstr(char *str)
21585 {
21586     size_t i, k;
21587     char temp;
21588
21589     for (i = 0; str[i] != '\0'; ++i)
21590         ;
21591
21592     for (--i, k = 0; k < i; ++k, --i) {
21593         temp = str[k];
21594         str[k] = str[i];
21595         str[i] = temp;
21596     }
21597
21598     return str;
21599 }
21600
21601 void exit_sys(const char *msg)
21602 {
21603     perror(msg);
21604
21605     exit(EXIT_FAILURE);
21606 }
21607
21608 /* client.c */
21609
21610 #include <stdio.h>
21611 #include <stdlib.h>
21612 #include <string.h>
21613 #include <unistd.h>
21614 #include <sys/socket.h>
21615 #include <netinet/in.h>
21616 #include <arpa/inet.h>
21617 #include <netdb.h>
21618
21619 #define BUFFER_SIZE      1024
21620
21621 void exit_sys(const char *msg);
21622
21623 int main(int argc, char *argv[])
21624 {
```

```
21625     int sock;
21626     struct sockaddr_in sinaddr;
21627     struct hostent *hent;
21628     in_port_t port;
21629     char buf[BUFFER_SIZE];
21630     char *str;
21631     ssize_t result;
21632
21633     if (argc != 3) {
21634         fprintf(stderr, "wrong number of arguments!..\n");
21635         exit(EXIT_FAILURE);
21636     }
21637
21638     port = (in_port_t)strtol(argv[2], NULL, 10);
21639
21640     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
21641         exit_sys("socket");
21642
21643     sinaddr.sin_family = AF_INET;
21644     sinaddr.sin_port = htons(port);
21645
21646     if ((sinaddr.sin_addr.s_addr = inet_addr(argv[1])) == INADDR_NONE) {
21647         if ((hent = gethostbyname(argv[1])) == NULL)
21648             exit_sys("gethostbyname");
21649         memcpy(&sinaddr.sin_addr.s_addr, hent->h_addr_list[0], hent-
21650               >h_length);
21651     }
21652
21653     if (connect(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
21654         exit_sys("connect");
21655
21656     printf("Connected...\n");
21657
21658     for (;;) {
21659         printf("Yazi giriniz:");
21660         fgets(buf, BUFFER_SIZE, stdin);
21661         if ((str = strchr(buf, '\n')) != NULL)
21662             *str = '\0';
21663         if ((send(sock, buf, strlen(buf), 0)) == -1)
21664             exit_sys("send");
21665         if (!strcmp(buf, "quit"))
21666             break;
21667
21668         if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == -1)
21669             exit_sys("recv");
21670         if (result == 0)
21671             break;
21672         buf[result] = '\0';
21673         printf("%ld bytes received from %s (%u): %s\n", (long)result,
21674               inet_ntoa(sinaddr.sin_addr),
21675               (unsigned)ntohs(sinaddr.sin_port), buf);
21676     }
21677
```

```
21676     /* Other stuff */
21677
21678
21679     return 0;
21680 }
21681
21682 void exit_sys(const char *msg)
21683 {
21684     perror(msg);
21685
21686     exit(EXIT_FAILURE);
21687 }
21688
21689 /
*-----*
-----
```

21690 Haberleşmenin sonunda TCP soketi nasıl kapatılmalıdır? Mademli soketler ↗
 UNIX/Linux sistemlerinde birer dosya betimleyicisi gibidir
21691 o halde soketi kapatma işlemi close ile yapılmaktadır. Tabii yine close ↗
 işlemi yapılmazsa işletim sistemi proses normal ya da
21692 sinyal gibi nedenlerle sonlandığında otomatik close işleminş yapar. ↗
 Soket betimleyicileri de dup işlemeye sokulabilir. Bu durumda
21693 close işlemi soket nesnesinin yok edileceği anlamına gelmez. Benzer ↗
 biçimde fork işlemi sırasında da betimleyicilerin
21694 çiftlendiğine dikkat ediniz.

21695

21696 Aktif soketlerin doğrudan close ile kapatılması iyi bir teknik değildir. ↗
 Bu soketler önce shutdown ile haberleşmeden kesilmeli sonra close
21697 işlemi uygulanmalıdır. Bu biçimde soketlerin kapatılmasına İngilizce ↗
 "graceful close (zarif kapatma)" denilmektedir. Pekiyi shutdown
21698 fonksiyonu ne yapmaktadır ve neden gerekmektedir? close işlemi ile bir ↗
 soket kapatıldığında işletim sistemi sokete ilişkin tüm veri ↗
 yapılarını

21699 ve bağlantı bilgilerini siler. Örneğin biz karşı tarafa send ile bir şey ↗
 gönderdikten hemen sonraki satırda close yaparsak artık
21700 send ile gönderdiklerimizin karşı tarafa ulaşacağının hiçbir garantisı ↗
 yoktur. Çünkü anımsanacağı gibi send aslında "gonderme tamponuna"
21701 bilgiyi yazıp geri dönmektedir. Hemen arkasından close işlemi ↗
 uygulandığında artık bu sokete ilişkin gönderme ve alma tamponları da
21702 yok edileceğinden tamponda gönderilmeyi bekleyen bilgiler hiç ↗
 gönderilmeyeblecektir. İşte shutdown fonksiyonun üç işlevi vardır:

21703

21704 1) Haberleşmeyi TCP çerçevesinde el sıkışarak sonlandırmak.
21705 2) Göndeme tamponuna yazılan bilgilerin gönderildiğine emin olmak.
21706 3) Okuma ya da yazma işlemini sonlandırıp diğer işleme devam edebilmek.

21707

21708 shutdown fonksiyonun prototipi şöyledir:

21709

21710 int shutdown(int socket, int how);

21711

21712 Fonksiyonun birinci parametresi sonlandırılacak soketin betimleyicisini, ↗
 ikinci parametresi biçimini belirtmektedir İkinci parametre
21713 sunlardan biri olarak girilebilir:

```
21714
21715      SHUT_RD: Bu işlemden sonra artık soketten okuma yapılamaz. Fakat sokete ↵
21716          yazma yapılabilir. Bu seçenek pek kullanılmamaktadır.
21717      SHUR_WR: Burada artık shutdown daha önce gönderme tamponuna yazılmış ↵
21718          olan byte'ların gönderilmesine kadar bloke oluşturulabilir.
21719      Bu işlemden sonra artık sokete yazma yapılamaz ancak okuma işlemi devam ↵
21720          ettirilebilir.
21721      SHUT_RDWR: En çok kullanılan seçenektır. Burada da artık shutdown daha ↵
21722          önce gönderme tamponuna yazılmış olan byte'ların gönderilmesine ↵
21723          kadar bloke oluşturulabilir. Artık bundan sonra soketten okuma ya da yazma ↵
21724          yapılamamaktadır. shutdown başarı durumunda 0 değerine, başarısızlık ↵
21725          durumunda -1
21726          değerine geri dönmektedir.
21727
21728      0 halde aktif bir soketin kapatılması tipik olarak şöyle yapılmaktadır:
21729
21730      shutdown(sock, SHUT_RDWR);
21731      close(sock);
21732
21733      Karşı taraf (peer) soketi shutdown ile SHUT_WR ya da SHUT_RDWR ile ↵
21734          sonlandırmışsa artık biz o soketten okuma yaptığımızda read ya da ↵
21735          recv fonksiyonları 0 ile geri döner. Benzer biçimde karşı taraf doğrudan ↵
21736          soketi close ile katapmışsa yine biz recv işleminden 0 elde ederiz.
21737
21738      Karşı tarafın soketi kapatıp kapatmadığı tipik olarak recv fonksiyonunda ↵
21739          anlaşılabilmektektir. Ancak karşı taraf soketi kapatıktan sonra biz ↵
21740          sokete
21741
21742      write ya da send ile birşeyler yazmak istersek default durumda UNIX/ ↵
21743          Linux sistemlerinde SIGPIPE sinyali oluşturmaktadır. Programcı send ↵
21744          fonksiyonun
21745
21746      flags parametresine MSG_NOSIGNAL değerini girerse bu durumda send ↵
21747          başarısız olmakta ve errno EPIPE değeri ile set edilmektedir.
21748
21749      Karşı taraf soketi kapatmamış ancak bağlantı kopmuş olabilir. Bu durumda ↵
21750          send ve recv fonksiyonları -1 ile geri döner.
21751
21752      Yukarıdaki programlara zarif kapatma özelliğini de ekleyebiliriz.
21753  -----
21754  -----
21755  -----
21756  -----
21757  -----
21758  -----
21759  -----
21760  -----
21761  -----
21762  -----
21763  -----
21764  -----
21765  -----
21766  -----
21767  -----
21768  -----
21769  -----
21770  -----
21771  -----
21772  -----
21773  -----
21774  -----
21775  -----
21776  -----
21777  -----
21778  -----
21779  -----
21780  -----
21781  -----
21782  -----
21783  -----
21784  -----
21785  -----
21786  -----
21787  -----
21788  -----
21789  -----
21790  -----
21791  -----
21792  -----
21793  -----
21794  -----
21795  -----
21796  -----
21797  -----
21798  -----
21799  -----
21800  -----
21801  -----
21802  -----
21803  -----
21804  -----
21805  -----
21806  -----
21807  -----
21808  -----
21809  -----
21810  -----
21811  -----
21812  -----
21813  -----
21814  -----
21815  -----
21816  -----
21817  -----
21818  -----
21819  -----
21820  -----
21821  -----
21822  -----
21823  -----
21824  -----
21825  -----
21826  -----
21827  -----
21828  -----
21829  -----
21830  -----
21831  -----
21832  -----
21833  -----
21834  -----
21835  -----
21836  -----
21837  -----
21838  -----
21839  -----
21840  -----
21841  -----
21842  -----
21843  -----
21844  -----
21845  -----
21846  -----
21847  -----
21848  -----
21849  -----
21850  -----
21851  -----
21852  -----
21853  -----
21854  -----
21855  -----
21856  -----
21857  -----
21858  -----
21859  -----
21860  -----
21861  -----
21862  -----
21863  -----
21864  -----
21865  -----
21866  -----
21867  -----
21868  -----
21869  -----
21870  -----
21871  -----
21872  -----
21873  -----
21874  -----
21875  -----
21876  -----
21877  -----
21878  -----
21879  -----
21880  -----
21881  -----
21882  -----
21883  -----
21884  -----
21885  -----
21886  -----
21887  -----
21888  -----
21889  -----
21890  -----
21891  -----
21892  -----
21893  -----
21894  -----
21895  -----
21896  -----
21897  -----
21898  -----
21899  -----
21900  -----
21901  -----
21902  -----
21903  -----
21904  -----
21905  -----
21906  -----
21907  -----
21908  -----
21909  -----
21910  -----
21911  -----
21912  -----
21913  -----
21914  -----
21915  -----
21916  -----
21917  -----
21918  -----
21919  -----
21920  -----
21921  -----
21922  -----
21923  -----
21924  -----
21925  -----
21926  -----
21927  -----
21928  -----
21929  -----
21930  -----
21931  -----
21932  -----
21933  -----
21934  -----
21935  -----
21936  -----
21937  -----
21938  -----
21939  -----
21940  -----
21941  -----
21942  -----
21943  -----
21944  -----
21945  -----
21946  -----
21947  -----
21948  -----
21949  -----
21950  -----
21951  -----
21952  -----
21953  -----
21954  -----
21955  -----
21956  -----
21957  -----
21958  -----
21959  -----
21960  -----
21961  -----
21962  -----
21963  -----
21964  -----
21965  -----
21966  -----
21967  -----
21968  -----
21969  -----
21970  -----
21971  -----
21972  -----
21973  -----
21974  -----
21975  -----
21976  -----
21977  -----
21978  -----
21979  -----
21980  -----
21981  -----
21982  -----
21983  -----
21984  -----
21985  -----
21986  -----
21987  -----
21988  -----
21989  -----
21990  -----
21991  -----
21992  -----
21993  -----
21994  -----
21995  -----
21996  -----
21997  -----
21998  -----
21999  -----
21999  -----*/
```

```
21752 int main(int argc, char *argv[])
21753 {
21754     int sock, sock_client;
21755     struct sockaddr_in sinaddr, sinaddr_client;
21756     socklen_t sinaddr_len;
21757     in_port_t port;
21758     ssize_t result;
21759     char buf[BUFFER_SIZE + 1];
21760
21761     if (argc != 2) {
21762         fprintf(stderr, "wrong number of arguments!..\n");
21763         exit(EXIT_FAILURE);
21764     }
21765
21766     port = (in_port_t)strtoul(argv[1], NULL, 10);
21767
21768     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
21769         exit_sys("socket");
21770
21771     sinaddr.sin_family = AF_INET;
21772     sinaddr.sin_port = htons(port);
21773     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
21774
21775     if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
21776         exit_sys("bind");
21777
21778     if (listen(sock, 8) == -1)
21779         exit_sys("listen");
21780
21781     printf("Waiting for connection...\n");
21782
21783     sinaddr_len = sizeof(sinaddr_client);
21784     if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,    ↴
21785         &sinaddr_len)) == -1)
21786         exit_sys("accept");
21787
21788     printf("Connected: %s : %u\n", inet_ntoa(sinaddr_client.sin_addr),    ↴
21789         (unsigned)ntohs(sinaddr_client.sin_port));
21790
21791     for (;;) {
21792         if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
21793             exit_sys("recv");
21794         if (result == 0)
21795             break;
21796         buf[result] = '\0';
21797         if (!strcmp(buf, "quit"))
21798             break;
21799         printf("%ld bytes received from %s (%u): %s\n", (long)result,
21800               inet_ntoa(sinaddr_client.sin_addr),
21801               (unsigned)ntohs(sinaddr_client.sin_port), buf);
21802         revstr(buf);
21803         if (send(sock_client, buf, strlen(buf), 0) == -1)
21804             exit_sys("send");
```

```
21802     }
21803
21804     shutdown(sock_client, SHUT_RDWR);
21805     close(sock_client);
21806     close(sock);
21807
21808     return 0;
21809 }
21810
21811 char *revstr(char *str)
21812 {
21813     size_t i, k;
21814     char temp;
21815
21816     for (i = 0; str[i] != '\0'; ++i)
21817         ;
21818
21819     for (--i, k = 0; k < i; ++k, --i) {
21820         temp = str[k];
21821         str[k] = str[i];
21822         str[i] = temp;
21823     }
21824
21825     return str;
21826 }
21827
21828 void exit_sys(const char *msg)
21829 {
21830     perror(msg);
21831
21832     exit(EXIT_FAILURE);
21833 }
21834
21835 /* client.c */
21836
21837 #include <stdio.h>
21838 #include <stdlib.h>
21839 #include <string.h>
21840 #include <unistd.h>
21841 #include <sys/socket.h>
21842 #include <netinet/in.h>
21843 #include <arpa/inet.h>
21844 #include <netdb.h>
21845
21846 #define BUFFER_SIZE      1024
21847
21848 void exit_sys(const char *msg);
21849
21850 int main(int argc, char *argv[])
21851 {
21852     int sock;
21853     struct sockaddr_in sinaddr;
21854     struct hostent *hent;
```

```
21855     in_port_t port;
21856     char buf[BUFFER_SIZE];
21857     char *str;
21858     ssize_t result;
21859
21860     if (argc != 3) {
21861         fprintf(stderr, "wrong number of arguments!..\n");
21862         exit(EXIT_FAILURE);
21863     }
21864
21865     port = (in_port_t)strtol(argv[2], NULL, 10);
21866
21867     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
21868         exit_sys("socket");
21869
21870     sinaddr.sin_family = AF_INET;
21871     sinaddr.sin_port = htons(port);
21872
21873     if ((sinaddr.sin_addr.s_addr = inet_addr(argv[1])) == INADDR_NONE) {
21874         if ((hent = gethostbyname(argv[1])) == NULL)
21875             exit_sys("gethostbyname");
21876         memcpy(&sinaddr.sin_addr.s_addr, hent->h_addr_list[0], hent-
21877               >h_length);
21878     }
21879
21880     if (connect(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
21881         exit_sys("connect");
21882
21883     printf("Connected...\n");
21884
21885     for (;;) {
21886         printf("Yazi giriniz:");
21887         fgets(buf, BUFFER_SIZE, stdin);
21888         if ((str = strchr(buf, '\n')) != NULL)
21889             *str = '\0';
21890         if ((send(sock, buf, strlen(buf), 0)) == -1)
21891             exit_sys("send");
21892         if (!strcmp(buf, "quit"))
21893             break;
21894
21895         if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == -1)
21896             exit_sys("recv");
21897         if (result == 0)
21898             break;
21899         buf[result] = '\0';
22000         printf("%ld bytes received from %s (%u): %s\n", (long)result,
22001               inet_ntoa(sinaddr.sin_addr),
22002               (unsigned)ntohs(sinaddr.sin_port), buf);
22003     }
22004
22005     shutdown(sock, SHUT_RDWR);
22006     close(sock);
22007
```

```
21906     return 0;
21907 }
21908
21909 void exit_sys(const char *msg)
21910 {
21911     perror(msg);
21912
21913     exit(EXIT_FAILURE);
21914 }
21915
21916 /
*-----*
```

21917 Windows sistemlerindeki soket kütüphanesine "Winsock" denilmektedir. Şu anda bu kütüphanenin 2'inci versiyonu kullanılmaktadır.

21918 Winsok fonksiyonları "UNIX/Linux uyumlu" fonksiyonlar ve Windows'a özgü fonksiyonlar olmak üzere iki biçimde kullanılabilmektedir.

21919 Ancak Winsock'un UNIX/Linux uyumlu fonksiyonlarında da birtakım değişiklikler söz konusudur. Bir UNIX/Linux ortamında yazılmış soket uygulamasının Windows sistemlerine aktarılması için şu düzeltmelerin yapılması gereklidir:

21921 1) POSIX'in soket sistemine ilişkin tüm başlık dosyaları kaldırılır. Onun yerine <winsok2.h> dosyası include edilir.

21922 2) xxx_t'li typedef türleri silinir ve onların yerine (dokümanlara da bakabilirsiniz) int, short, unsigned int, unsigned short türleri kullanılır.

21923 3) Windows'ta soket sisteminin ilklendirilmesi için WSASStartup fonksiyonu işin başında çağrıılır ve işin sonunda da bu işlem WSACleanup fonksiyonuyla geri alınır.

21924 4) Windows'ta dosya betimleyicisi kavramı yoktur. (Onun yerine "handle" kavramı vardır.) Dolayısıyla soket türü de int değil, SOCKET isimli bir typedef türüdür.

21925 5) shutdown fonksiyonun ikinci parametresi SD_RECEIVE, SD_SEND ve SD_BOTH biçimindedir.

21926 6) close fonksiyonu yerine closesocket ile soket kapatılır.

21927 7) Windows'ta soket fonksiyonları başarısızlık durumunda -1 değerine geri dönmeler. socket fonksiyonu başarısızlık durumunda INVALID_SOCKET değerine, diğerleri ise SOCKET_ERROR değerine geri dönmektedir.

21928 8) Visual Studiko IDE'sinde default durumda "deprecated" durumlar "error"e yükseltilmiştir. Bunlar için bir sembolik sabit define edilebilmektedir. Ancak

21929 proje ayarlarından "sdl check" disable edilebilir. Benzer biçimde proje ayarlarından "Unicode" "not set" yapılmalıdır.

21930 9) Projelin linker ayarlarından Input/Additional Dependencies edit alanına Winsock kütüphanesi olan "Ws2_32.lib" import kütüphanesi eklenir.

21931 Yukarıdaki programların Winsock'a dönüştürülmüş biçimleri şyledir:

21932 -----*/

```
21940
21941 /* server.c */
21942
21943 #include <stdio.h>
21944 #include <stdlib.h>
21945 #include <string.h>
21946 #include <WinSock2.h>
21947
21948 #define BUFFER_SIZE      1024
21949
21950 char* revstr(char* str);
21951 void exit_sys(LPCSTR lpszMsg, int status, DWORD dwLastError);
21952
21953 int main(int argc, char* argv[])
21954 {
21955     SOCKET sock, sock_client;
21956     struct sockaddr_in sinaddr, sinaddr_client;
21957     int sinaddr_len;
21958     unsigned short port;
21959     int result;
21960     char buf[BUFFER_SIZE + 1];
21961     WSADATA wsadata;
21962
21963     if ((result = WSAStartup(MAKEWORD(2, 2), &wsadata)) != 0)
21964         exit_sys("WSAStartup", EXIT_FAILURE, result);
21965
21966     if (argc != 2) {
21967         fprintf(stderr, "wrong number of arguments!..\n");
21968         exit(EXIT_FAILURE);
21969     }
21970
21971     port = (unsigned short)strtoul(argv[1], NULL, 10);
21972
21973     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET)
21974         exit_sys("socket", EXIT_FAILURE, WSAGetLastError());
21975
21976     sinaddr.sin_family = AF_INET;
21977     sinaddr.sin_port = htons(port);
21978     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
21979
21980     if (bind(sock, (struct sockaddr*)&sinaddr, sizeof(sinaddr)) == SOCKET_ERROR)
21981         exit_sys("bind", EXIT_FAILURE, WSAGetLastError());
21982
21983     if (listen(sock, 8) == -1)
21984         exit_sys("listen", EXIT_FAILURE, WSAGetLastError());
21985
21986     printf("Waiting for connection...\n");
21987
21988     sinaddr_len = sizeof(sinaddr_client);
21989     if ((sock_client = accept(sock, (struct sockaddr*)&sinaddr_client,
21990                               &sinaddr_len)) == SOCKET_ERROR)
21991         exit_sys("accept", EXIT_FAILURE, WSAGetLastError());
```

```
21991
21992     printf("Connected: %s : %u\n", inet_ntoa(sinaddr_client.sin_addr),
21993             (unsigned)ntohs(sinaddr_client.sin_port));
21994     for (;;) {
21995         if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
21996             exit_sys("recv", EXIT_FAILURE, WSAGetLastError());
21997         if (result == 0)
21998             break;
21999         buf[result] = '\0';
22000         if (!strcmp(buf, "quit"))
22001             break;
22002         printf("%ld bytes received from %s (%u): %s\n", (long)result,
22003                inet_ntoa(sinaddr_client.sin_addr),
22004                (unsigned)ntohs(sinaddr_client.sin_port), buf);
22005         revstr(buf);
22006         if (send(sock_client, buf, strlen(buf), 0) == -1)
22007             exit_sys("send", EXIT_FAILURE, WSAGetLastError());
22008     }
22009     shutdown(sock_client, SD_BOTH);
22010     closesocket(sock_client);
22011     closesocket(sock);
22012
22013     WSACleanup();
22014
22015     return 0;
22016 }
22017
22018 char *revstr(char* str)
22019 {
22020     size_t i, k;
22021     char temp;
22022
22023     for (i = 0; str[i] != '\0'; ++i)
22024         ;
22025
22026     for (--i, k = 0; k < i; ++k, --i) {
22027         temp = str[k];
22028         str[k] = str[i];
22029         str[i] = temp;
22030     }
22031
22032     return str;
22033 }
22034
22035 void exit_sys(LPCSTR lpszMsg, int status, DWORD dwLastError)
22036 {
22037     LPTSTR lpszErr;
22038
22039     if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
22040                       FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
22041                       MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0,
```

```
        NULL)) {
22041     fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
22042     LocalFree(lpszErr);
22043 }
22044
22045     exit(status);
22046 }
22047
22048 /* client.c */
22049
22050 #include <stdio.h>
22051 #include <stdlib.h>
22052 #include <string.h>
22053 #include <WinSock2.h>
22054
22055 #define BUFFER_SIZE      1024
22056
22057 void exit_sys(LPCSTR lpszMsg, int status, DWORD dwLastError);
22058
22059 int main(int argc, char* argv[])
22060 {
22061     SOCKET sock;
22062     WSADATA wsadata;
22063     struct sockaddr_in sinaddr;
22064     struct hostent* hent;
22065     unsigned short port;
22066     char buf[BUFFER_SIZE];
22067     char *str;
22068     int result;
22069
22070     if (argc != 3) {
22071         fprintf(stderr, "wrong number of arguments!..\n");
22072         exit(EXIT_FAILURE);
22073     }
22074
22075     port = (unsigned short)strtol(argv[2], NULL, 10);
22076
22077     if ((result = WSAStartup(MAKEWORD(2, 2), &wsadata)) != 0)
22078         exit_sys("WSAStartup", EXIT_FAILURE, result);
22079
22080     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == INVALID_SOCKET)
22081         exit_sys("socket", EXIT_FAILURE, WSAGetLastError());
22082
22083     sinaddr.sin_family = AF_INET;
22084     sinaddr.sin_port = htons(port);
22085
22086     if ((sinaddr.sin_addr.s_addr = inet_addr(argv[1])) == INADDR_NONE) {
22087         if ((hent = gethostbyname(argv[1])) == NULL)
22088             exit_sys("gethostbyname", EXIT_FAILURE, WSAGetLastError());
22089         memcpy(&sinaddr.sin_addr.s_addr, hent->h_addr_list[0], hent-
22090             >h_length);
22091     }
22091
```

```
22092     if (connect(sock, (struct sockaddr*)&sinaddr, sizeof(sinaddr)) == SOCKET_ERROR)
22093         exit_sys("connect", EXIT_FAILURE, WSAGetLastError());
22094     printf("Connected...\n");
22095
22096     for (;;) {
22097         printf("Yazi giriniz:");
22098         fgets(buf, BUFFER_SIZE, stdin);
22099         if ((str = strchr(buf, '\n')) != NULL)
22100             *str = '\0';
22101         if ((send(sock, buf, strlen(buf), 0)) == SOCKET_ERROR)
22102             exit_sys("send", EXIT_FAILURE, WSAGetLastError());
22103         if (!strcmp(buf, "quit"))
22104             break;
22105
22106         if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == SOCKET_ERROR)
22107             exit_sys("recv", EXIT_FAILURE, WSAGetLastError());
22108         if (result == 0)
22109             break;
22110         buf[result] = '\0';
22111         printf("%ld bytes received from %s (%u): %s\n", (long)result,
22112               inet_ntoa(sinaddr.sin_addr),
22113               (unsigned)ntohs(sinaddr.sin_port), buf);
22114     }
22115
22116     shutdown(sock, SD_BOTH);
22117     closesocket(sock);
22118
22119     WSACleanup();
22120
22121     return 0;
22122 }
22123
22124 void exit_sys(LPCSTR lpszMsg, int status, DWORD dwLastError)
22125 {
22126     LPTSTR lpszErr;
22127
22128     if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
22129                     FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
22130                     MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0,
22131                     NULL)) {
22132         fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
22133         LocalFree(lpszErr);
22134     }
22135     exit(status);
22136 }
22137 /
*-----*
-----*
22138     Daha önce write/send ve read/recv fonksiyonlarının davranışları hakkında
```

temel bazı şeyler söylemişтик. Şimdi biraz ayırtılalım.

22139 POSIX standartlarına göre send fonksiyonu blokeli modda bloke garantisi ↗
vermektedir. Yani örneğin gönderme tamponunda 10 byte'lık boş bir ↗
alan varsa fakat biz write/send ile 100 byte göndermeye çalışıyorsak ↗
blokeli modda send bu 100 byte'in tamamını tampona yazana kadar ↗
blokede bekler. Böylece send genel olarak bizim yazmak istediğimiz byte ↗
miktarına geri dönmektedir. Ancak bu davranış eskiden tam böyle ↗
değildi.

22142 Windows sistemlerinde de tam böyle değildir. Windows sistemlerinde send ↗
fonksiyonu eğer gönderme tamponu tüm gönderilecek bilgiyi alamayacak ↗
biçimde doluya send blokeye yol açmayıabilmektedir. Böylece Windows ↗
sistemlerinde send talep edilenden daha düşük bir değerle ↗
geri dönebilmektedir. Blokesin modda POSIX standartlarına göre send ↗
fonksiyonu yine kısmi yazma izin vermemektedir. Örneğin gönderme ↗
tamponunda yine 10 byte boşluk olsun. Biz de blokesiz modda 100 byte ↗
göndermek isteyelim. Bu durumda send tüm bilgileri tampona ↗
yazamayacaksız

22145 hiçbirini yazmaz ve -1 ile geri döner. errno EAGAIN değeri ile set ↗
edilir.

22147
22148 read/recv fonksyonları daha önceden belirtildiği gibi blokeli modda ↗
eğer alma tamponu tamamen boş ise blokeye yol açmaktadır. Alma ↗
tamponunda en az 1 byte varsa read/recv bloke oluşturmadan okur ve ↗
geri döner. Blokesiz modda write/recv tampon boşsa bile blokeye ↗
yol açmamaktadır. Bu durumda bu fonksyonlar -1 ile geri dönerler ve ↗
errno EAGAIN değeriyle set edilir.

22151
22152 Yukarıda da belirtildiği gibi karşı taraf shutdown uygulamışsa ya da ↗
soketi kapatmışsa read/recv 0 ile geri döner. Başka diğer ↗
hatalarda (örneğin bağlantı kopması) bu fonksyonlar başarısız olup -1 ↗
ile geri dönerler.

22154 -----*/
22155 /
22156 */-----

22157 Zamanla bazı klasik soket fonksyonları yerine onların işlevini ↗
yapabilecek daha yetenekli fonksiyonlar oluşturulmuştur.

22158 Eski fonksiyonlar IPV4 zamanlarında tasarlanmıştır. IPV6 için bunların ↗
bazlarının farklı isimli biçimleri devreye sokuldu.

22159 En sonunda hem IPV4'te hem de IPV6 da çalışacak daha yetenekli ↗
fonksiyonlar tasarlandı. Programlarınızda bunları kullanmanız ↗
artık daha uygun olabilir.

22161
22162 inet_nto fonksiyonu bilindiği gibi noktalı desimal formattaki IP ↗
adresini 4 byte'lık IPV4 formuna dönüştürüyordu.

22163 İşte bu fonksyonun inet_ptoa isimli IPV6'yu da kapsayan gelişmiş ↗
biçimi vardır:

22164
22165
22166 const char *inet_ntop(int af, const void *src, char *dst, socklen_t ↗
size);

```
22167
22168    Fonksiyonun birinci parametresi AF_INET (IPV4) ya da AF_INET& (IPV6)     ↵
22169        olarak girilmelidir. İkinci parametre nümerik IPV4 ya da IPv6     ↵
22170        adresinin bulunduğu adresi belirtmektedir. Fonksiyon yazışal noktalı     ↵
22171        desimal formatı üçüncü parametresiyle belirtilen     ↵
22172        adrese yerleştirir. Son parametre üçüncü parametredeki dizinin     ↵
22173        uzunluğunu belirtir. Bu parametre INET_ADDRSTRLEN ya da     ↵
22174            INET6_ADDRSTRLEN
22175        biçiminde girilebilir. Fonksiyon başarı durumunda üçüncü parametreyle     ↵
22176            belirtilen adrese başarısızlık durumunda NULLL adrese     ↵
22177        geri döner.Örneğin:
22178
22179        char ntohsbuf[INET_ADDRSTRLEN];
22180        ...
22181        printf("Connected: %s : %u\n", inet_ntop(AF_INET,
22182                    &sinaddr_client.sin_addr, ntohsbuf, INET_ADDRSTRLEN), (unsigned)ntohs
22183                    (sinaddr_client.sin_port));
22184
22185        inet_ntoa işleminin tersinin inet_addr ile yapıldığını belirtmiştık.     ↵
22186        İşte inet_addr fonksiyonunun yerine hem IPV4 hem de IPV6 ielk çalışan     ↵
22187        inet_ptoa fonksiyonu kullanılabilmektedir:
22188
22189        int inet_pton(int af, const char *src, void *restrict dst);
22190
22191        Fonksiyonun birinci parametresi yine AF_INET ya da AF_INET6 biçiminde     ↵
22192        geçilir. İkinci parametre noktalı desimal formatın bulunduğu     ↵
22193        yazının adresini üçüncü parametre ise nümerik adresin yerleştirileceği     ↵
22194        adresi almaktadır. Fonksiyon başarı durumunda 1 değerine, başarısızlık     ↵
22195        durumundurumunda 0 ya da -1 değerine geri döner. Başarısızlık birinci     ↵
22196        parametreden kaynaklanıyorsa -1, ikinci parametreden yanaklıyorsa 0)     ↵
22197        Bu durumda örneğin client programda inet_addr yerine inet_ptoa     ↵
22198        fonksiyonunu şöyle çağırabilirdik:
22199
22200        if (inet_pton(AF_INET, argv[1], &sinaddr.sin_addr.s_addr) == 0) {
22201            ...
22202
22203            IPV6 ile birlikte yeni gelen diğer bir fonksiyon da getaddrinfo isimli     ↵
22204            fonksiyondur. Bu fonksiyonasında inet_addr ve gethosbyname     ↵
22205            fonksiyonlarının IPV6'yi da içerecek biçimde genişletilmiş bir     ↵
22206            biçimidir. Yani getaddrinfo hem noktalı desimal formatı nümerik adrese     ↵
22207            dönüsürür hem de eğer isim söz konusuysa DNS işlemi yaparak ilgili     ↵
22208            host'un IP adresini elde eder. Maalesef fonksiyon biraz karışık     ↵
22209            tasarlannmıştır.
22210            Fonksiyonun prototipi şöyledir:
22211
22212
22213        int getaddrinfo(const char *node, const char *service, const struct     ↵
22214            addrinfo *hints, struct addrinfo **res);
22215
22216        Fonksiyonun birinci parametres, "noktalı desimal formatlı IP adresi" ya     ↵
22217        da "host ismi"dir. İkinci parametre NULL geçilebilir ya da
```

22202 buraya port numarası girilebilir. Ancak bu port numarası girileceğse ↵
yazışal girilmelidir. Fonksiyon kendisi bu port numarasını ↵
22203 htons yaparak Big Endian formata dönüştürüp bize verecektir. Bu ↵
parametre IP ailesinin uygulama katmanına ilişkin spesifik bir ↵
protokol ↵
22204 ismi olarak da girilebilmektedir (Örneğin "http" gibi). Eğer bu ↵
parametre NULL girilirse bize port olarak 0 verilir. Fonksiyonun ↵
Üçüncü ↵
22205 parametresi nasıl bir adres istediğimizi anlatan filtreleme ↵
seçeneklerini belirtir. Bu parametre addrinfo isimli bir yapı ↵
türündendir. ↵
22206 Bu yapının yalnızca ilk dört elemanı programcı tarafından ↵
girilebilmektedir. Ancak POSIX standartları bu yapının elemanlarının ↵
sıfırlanmasını ↵
22207 öngörmektedir (sıfırlanmak normal tirler için bit 0, göstericiler için ↵
NULL adres). Yapı şöyledir: ↵
22208
22209 **struct addrinfo {**
22210 int ai_flags;
22211 int ai_family;
22212 int ai_socktype;
22213 int ai_protocol;
22214 socklen_t ai_addrlen;
22215 **struct sockaddr ***ai_addr;
22216 char *ai_canonname;
22217 **struct addrinfo ***ai_next;
22218 **};**
22219
22220 Yapının ai_flags elemanı pek çok bayrak değeri alabilmektedir. Bu değer ↵
0 olarak da geçilebilir. Yapının ai_family elemanı AF_INET girilirse ↵
22221 host'a ilişkin IPV4 adresleri, AF_INET6 girilirse IPV6 adresleri ↵
AF_UNSPEC girilirse hem IPV4 hem de IPV6 adresleri elde edilir. ↵
22222 Yapının ai_socktype elemanı 0 girilebilir ya da SOCK_STREAM veya ↵
SOCK_DGRAM girilebilir. Fonksiyonun ayrıntılı açıklaması için ↵
dokümanlara ↵
22223 başvurunuz. ↵
22224
22225 getaddrinfo fonksiyonun son parametresine bir bağlı listenin ilk ↵
elemanını gösteren adres yerleştirilmektedir. Buradaki bağlı listenin ↵
22226 bağ elemanı struct addrinfo yapısının ai_next elemanıdır. Bu bağlı ↵
listenin boşaltımı freeaddrinfo fonksiyonu tarafından yapılmaktadır. ↵
22227
22228 getaddrinfo fonksiyonu başarı durumunda 0 değerine başarısızlık ↵
durumunda error koduna geri döner. Bu error kodları klasik errno ↵
değerlerinden ↵
22229 farklı olduğu için strerror fonksiyonuyla değil gai_strerror ↵
fonksiyonuyla yazıya dönüştürülmelidir. ↵
22230
22231 getaddrinfo fonksiyonun tersini yapan getnameinfo isminde bir fonksiyon ↵
da sonraları soket kütüphanesine eklenmiştir. getnameinfo aslında ↵
inet_ntop, ↵
22232 getserbyname (biz göremedik) fonksiyonlarının bileşimi gibidir. Biz ↵
aşağıdaki örnekte bu fonksiyonu kullanmayacağız. ↵

```
22233
22234     int getnameinfo(const struct sockaddr *addr, socklen_t addrlen, char      ↵
22235             *host, socklen_t hostlen, char *serv, socklen_t servlen, int flags);
22236     Fonksiyonun birinci parametresi sockaddr_in ya da sockaddr_in6           ↵
22237             yapısıdır. İkinci parametre birinci parametredeki yapının uzunluğudur. ↵
22238
22239     Fonksiyonun sonraki dört parametresi sırasıyla noktal hostun yazışal      ↵
22240             temsilin yerleştirileceği dizinin asresi ve uzunluğu, port numarasına      ↵
22241             ilişkin
22242     yazının (servis ismi) yerleştirileceği dizinin adresi ve uzunluğudur.      ↵
22243     Son parametre 0 geçilebilir. Maksimum host ismi NI_MAXHOST ile
22244     maksimum servis ismi ise NI_MAXSERV ile belirtilmiştir.
22245
22246     Aşağıda daha önce yazmış olduğumuz server'ın bu yeni fonksiyonlarla      ↵
22247             yazılmış halini görüyorsunuz.
22248
22249     -----
22250
22251
22252
22253
22254
22255
22256
22257
22258
22259
22260
22261
22262
22263
22264
22265
22266
22267
22268
22269
22270
22271
22272
22273
22274
22275
22276
22277
```

-----*/

```
22243
22244 /* server.c */
22245
22246 #include <stdio.h>
22247 #include <stdlib.h>
22248 #include <string.h>
22249 #include <unistd.h>
22250 #include <sys/socket.h>
22251 #include <netinet/in.h>
22252 #include <arpa/inet.h>
22253
22254 #define BUFFER_SIZE      1024
22255
22256 char *revstr(char *str);
22257 void exit_sys(const char *msg);
22258
22259 int main(int argc, char *argv[])
22260 {
22261     int sock, sock_client;
22262     struct sockaddr_in sinaddr, sinaddr_client;
22263     socklen_t sinaddr_len;
22264     char ntopbuf[INET_ADDRSTRLEN];
22265     in_port_t port;
22266     ssize_t result;
22267     char buf[BUFFER_SIZE + 1];
22268
22269     if (argc != 2) {
22270         fprintf(stderr, "wrong number of arguments!..\n");
22271         exit(EXIT_FAILURE);
22272     }
22273
22274     port = (in_port_t)strtoul(argv[1], NULL, 10);
22275
22276     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
22277         exit_sys("socket");
```

```
22278
22279     sinaddr.sin_family = AF_INET;
22280     sinaddr.sin_port = htons(port);
22281     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
22282
22283     if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
22284         exit_sys("bind");
22285
22286     if (listen(sock, 8) == -1)
22287         exit_sys("listen");
22288
22289     printf("Waiting for connection...\n");
22290
22291     sinaddr_len = sizeof(sinaddr_client);
22292     if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,
22293                               &sinaddr_len)) == -1)
22294         exit_sys("accept");
22295
22296     printf("Connected: %s : %u\n", inet_ntop(AF_INET,
22297                                               &sinaddr_client.sin_addr, ntopbuf, INET_ADDRSTRLEN), (unsigned)ntohs
22298                                               (sinaddr_client.sin_port));
22299
22300     for (;;) {
22301         if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
22302             exit_sys("recv");
22303         if (result == 0)
22304             break;
22305         buf[result] = '\0';
22306         if (!strcmp(buf, "quit"))
22307             break;
22308         printf("%ld bytes received from %s (%u): %s\n", (long)result,
22309               ntopbuf, (unsigned)ntohs(sinaddr_client.sin_port), buf);
22310         revstr(buf);
22311         if (send(sock_client, buf, strlen(buf), 0) == -1)
22312             exit_sys("send");
22313     }
22314
22315     return 0;
22316 }
22317
22318 char *revstr(char *str)
22319 {
22320     size_t i, k;
22321     char temp;
22322
22323     for (i = 0; str[i] != '\0'; ++i)
22324         ;
22325
22326     for (--i, k = 0; k < i; ++k, --i) {
```

```
22327         temp = str[k];
22328         str[k] = str[i];
22329         str[i] = temp;
22330     }
22331
22332     return str;
22333 }
22334
22335 void exit_sys(const char *msg)
22336 {
22337     perror(msg);
22338
22339     exit(EXIT_FAILURE);
22340 }
22341
22342 /* client.c */
22343
22344 #include <stdio.h>
22345 #include <stdlib.h>
22346 #include <string.h>
22347 #include <unistd.h>
22348 #include <sys/socket.h>
22349 #include <netinet/in.h>
22350 #include <arpa/inet.h>
22351 #include <netdb.h>
22352
22353 #define BUFFER_SIZE      1024
22354
22355 void exit_sys(const char *msg);
22356
22357 int main(int argc, char *argv[])
22358 {
22359     int sock;
22360     struct addrinfo *ai, *ri;
22361     struct addrinfo hints = {0};
22362     char buf[BUFFER_SIZE];
22363     char *str;
22364     ssize_t result;
22365     int sresult;
22366
22367     if (argc != 3) {
22368         fprintf(stderr, "wrong number of arguments!..\n");
22369         exit(EXIT_FAILURE);
22370     }
22371
22372     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
22373         exit_sys("socket");
22374
22375     hints.ai_family = AF_INET;
22376     hints.ai_socktype = SOCK_STREAM;
22377
22378     if ((sresult = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
22379         fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(sresult));
```

```
22380         exit(EXIT_FAILURE);
22381     }
22382
22383     for (ri = ai; ri != NULL; ri = ri->ai_next)
22384         if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
22385             break;
22386
22387     if (ri == NULL)
22388         exit_sys("connect");
22389
22390     freeaddrinfo(ai);
22391
22392     printf("Connected...\n");
22393
22394     for (;;) {
22395         printf("Yazi giriniz:");
22396         fgets(buf, BUFFER_SIZE, stdin);
22397         if ((str = strchr(buf, '\n')) != NULL)
22398             *str = '\0';
22399         if ((send(sock, buf, strlen(buf), 0)) == -1)
22400             exit_sys("send");
22401         if (!strcmp(buf, "quit"))
22402             break;
22403
22404         if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == -1)
22405             exit_sys("recv");
22406         if (result == 0)
22407             break;
22408         buf[result] = '\0';
22409         printf("%ld bytes received from %s (%s): %s\n", (long)result, argv[1], argv[2], buf);
22410     }
22411
22412     shutdown(sock, SHUT_RDWR);
22413     close(sock);
22414
22415     return 0;
22416 }
22417
22418 void exit_sys(const char *msg)
22419 {
22420     perror(msg);
22421
22422     exit(EXIT_FAILURE);
22423 }
22424
22425 /
*-----  
-----  
22426 Tipki borularda olduğu gibi çok client'li TCP server programları birden fazla client'tan gelen bilgileri okuyabilmelidir.  
22427 Budurumda server programın organizasyonu client programlara göre daha zor olmaktadır. Buradaki server problemi yine söyledir:  
-----  
-----
```

```
22428     Server bir client için recv yaparken eğer o soketin alma tamponu boşsa →  
22429         bloke olur. Bu durumda maalesef diğer client'lardan  
22430         gelmiş olan bilgileri okuyamaz. İşte bu tür durumlarda tıpkı daha önce →  
22431         borular için yaptığımız "ileri io modelleri" kullanılmalıdır.  
22432  
22433     Şüphesiz en basit çok client'lı organizasyon fork organizasyonudur. →  
22434         Şöyle ki bu organizasyonda server her client bağlantısı  
22435         sağlandığında fork yapar, böylece her client ile aslında ayrı bir proses →  
22436         konuşmuş olur. Şüphesiz her client bağlantısında bir  
22437         prosesin oluşturulması verimli bir yöntem değildir. Bu öntemi uygularken →  
22438         üst prosesin otomatik zombie engellemeye yapması gereklidir.  
22439     Çünkü üst proses hep accept işleminde bloke bekleyeceğine göre alt →  
22440         prosesi zombie'likten kurtaramaz.  
22441  
22442     Aşağıda böyle bir fork modeli örneği verilmiştir  
22443     -----*/  
22444  
22445     /* server.c */  
22446  
22447     #include <stdio.h>  
22448     #include <stdlib.h>  
22449     #include <string.h>  
22450     #include <unistd.h>  
22451     #include <sys/socket.h>  
22452     #include <netinet/in.h>  
22453     #include <arpa/inet.h>  
22454  
22455     #define BUFFER_SIZE      1024  
22456  
22457     char *revstr(char *str);  
22458     void exit_sys(const char *msg);  
22459  
22460     int main(int argc, char *argv[]) {  
22461         int sock, sock_client;  
22462         struct sockaddr_in sinaddr, sinaddr_client;  
22463         socklen_t sinaddr_len;  
22464         char ntopbuf[INET_ADDRSTRLEN];  
22465         in_port_t port;  
22466         ssize_t result;  
22467         char buf[BUFFER_SIZE + 1];  
22468         pid_t pid;  
22469  
22470         if (argc != 2) {  
22471             fprintf(stderr, "wrong number of arguments!..\\n");  
22472             exit(EXIT_FAILURE);  
22473         }  
22474         port = (in_port_t)strtoul(argv[1], NULL, 10);  
22475  
22476         if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)  
22477             exit_sys("socket");
```

```
22474
22475     sinaddr.sin_family = AF_INET;
22476     sinaddr.sin_port = htons(port);
22477     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
22478
22479     if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
22480         exit_sys("bind");
22481
22482     if (listen(sock, 8) == -1)
22483         exit_sys("listen");
22484
22485     printf("Waiting for connection...\n");
22486
22487     for (;;) {
22488         sinaddr_len = sizeof(sinaddr_client);
22489         if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client, &
22490             &sinaddr_len)) == -1)
22491             exit_sys("accept");
22492
22493         printf("Connected: %s : %u\n", inet_ntop(AF_INET,
22494             &sinaddr_client.sin_addr, ntopbuf, INET_ADDRSTRLEN), (unsigned)    ↵
22495             ntohs(sinaddr_client.sin_port));
22496
22497         if ((pid = fork()) == -1)
22498             exit_sys("fork");
22499         if (pid == 0) {
22500             for(;;) {
22501                 if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
22502                     exit_sys("recv");
22503                 if (result == 0)
22504                     break;
22505                 buf[result] = '\0';
22506                 if (!strcmp(buf, "quit"))
22507                     break;
22508                 printf("%ld bytes received from %s (%u): %s\n", (long)    ↵
22509                     result, ntopbuf, (unsigned)ntohs
22510                         (sinaddr_client.sin_port), buf);
22511                 revstr(buf);
22512                 if (send(sock_client, buf, strlen(buf), 0) == -1)
22513                     exit_sys("send");
22514             }
22515             shutdown(sock_client, SHUT_RDWR);
22516             close(sock_client);
22517             exit(EXIT_SUCCESS);
22518         }
22519     }
22520
22521     char *revstr(char *str)
```

```
22522 {
22523     size_t i, k;
22524     char temp;
22525
22526     for (i = 0; str[i] != '\0'; ++i)
22527         ;
22528
22529     for (--i, k = 0; k < i; ++k, --i) {
22530         temp = str[k];
22531         str[k] = str[i];
22532         str[i] = temp;
22533     }
22534
22535     return str;
22536 }
22537
22538 void exit_sys(const char *msg)
22539 {
22540     perror(msg);
22541
22542     exit(EXIT_FAILURE);
22543 }
22544
22545 /* client.c */
22546
22547 #include <stdio.h>
22548 #include <stdlib.h>
22549 #include <string.h>
22550 #include <unistd.h>
22551 #include <sys/socket.h>
22552 #include <netinet/in.h>
22553 #include <arpa/inet.h>
22554 #include <netdb.h>
22555
22556 #define BUFFER_SIZE      1024
22557
22558 void exit_sys(const char *msg);
22559
22560 int main(int argc, char *argv[])
22561 {
22562     int sock;
22563     struct addrinfo *ai, *ri;
22564     struct addrinfo hints = {0};
22565     char buf[BUFFER_SIZE];
22566     char *str;
22567     ssize_t result;
22568     int sresult;
22569
22570     if (argc != 3) {
22571         fprintf(stderr, "wrong number of arguments!..\\n");
22572         exit(EXIT_FAILURE);
22573     }
22574 }
```

```
22575     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
22576         exit_sys("socket");
22577
22578     hints.ai_family = AF_INET;
22579     hints.ai_socktype = SOCK_STREAM;
22580
22581     if ((sresult = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
22582         fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(sresult));
22583         exit(EXIT_FAILURE);
22584     }
22585
22586     for (ri = ai; ri != NULL; ri = ri->ai_next)
22587         if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
22588             break;
22589
22590     if (ri == NULL)
22591         exit_sys("connect");
22592
22593     freeaddrinfo(ai);
22594
22595     printf("Connected...\n");
22596
22597     for (;;) {
22598         printf("Yazi giriniz:");
22599         fgets(buf, BUFFER_SIZE, stdin);
22600         if ((str = strchr(buf, '\n')) != NULL)
22601             *str = '\0';
22602         if ((send(sock, buf, strlen(buf), 0)) == -1)
22603             exit_sys("send");
22604         if (!strcmp(buf, "quit"))
22605             break;
22606
22607         if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == -1)
22608             exit_sys("recv");
22609         if (result == 0)
22610             break;
22611         buf[result] = '\0';
22612         printf("%ld bytes received from %s (%s): %s\n", (long)result, argv
22613             [1], argv[2], buf);
22614     }
22615
22616     shutdown(sock, SHUT_RDWR);
22617     close(sock);
22618
22619     return 0;
22620 }
22621 void exit_sys(const char *msg)
22622 {
22623     perror(msg);
22624
22625     exit(EXIT_FAILURE);
22626 }
```

```
22627
22628  /
22629      *-----*
22630      -----
22631      Yukarıdaki açıkladığımız fork modeli en basit model olmakla birlikte en →
22632      verimsiz modeldir. Her client bağlantısında →
22633      bir prosesin yaratılması client sayısı arttıkça sistemde yük oluşturur. →
22634      Tabii fork modeli az client için uygulanabilecek bir →
22635      modeldir. Ancak burada server client'lara müdahale edemez. Çünkü her →
22636      server her client için ayrı proses yaratmıştır.
22637      İşte fork modeli yerine thread modeli her bakımından daha etkindir. Bu →
22638      modelde server her client bağlantısında bir proses değil thread →
22639      yaratır.
22640
22641      Aşağıda thread modeline ilişkin örnek görülmektedir.
22642
22643
22644
22645
22646
22647
22648  typedef struct {
22649      int sock;
22650      struct sockaddr_in sinaddr;
22651      char ntopbuf[INET_ADDRSTRLEN];
22652  } CLIENT_INFO;
22653
22654  #define BUFFER_SIZE      1024
22655
22656  void *client_proc(void *param);
22657  char *revstr(char *str);
22658  void exit_sys(const char *msg);
22659
22660  int main(int argc, char *argv[])
22661  {
22662      int sock, sock_client;
22663      struct sockaddr_in sinaddr, sinaddr_client;
22664      socklen_t sinaddr_len;
22665      in_port_t port;
22666      ssize_t result;
22667      CLIENT_INFO *ci;
22668      pthread_t tid;
22669
22670      if (argc != 2) {
22671          fprintf(stderr, "wrong number of arguments!..\n");
```

```
22672         exit(EXIT_FAILURE);
22673     }
22674
22675     port = (in_port_t)strtoul(argv[1], NULL, 10);
22676
22677     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
22678         exit_sys("socket");
22679
22680     sinaddr.sin_family = AF_INET;
22681     sinaddr.sin_port = htons(port);
22682     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
22683
22684     if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
22685         exit_sys("bind");
22686
22687     if (listen(sock, 8) == -1)
22688         exit_sys("listen");
22689
22690     printf("Waiting for connection...\n");
22691
22692     for (;;) {
22693         sinaddr_len = sizeof(sinaddr_client);
22694         if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client, &
22695             &sinaddr_len)) == -1)
22696             exit_sys("accept");
22697
22698         if ((ci = (CLIENT_INFO *)malloc(sizeof(CLIENT_INFO))) == NULL)
22699             exit_sys("malloc");
22700
22701         ci->sock = sock_client;
22702         ci->sinaddr = sinaddr_client;
22703         inet_ntop(AF_INET, &sinaddr_client.sin_addr, ci->ntopbuf,           ↴
22704             INET_ADDRSTRLEN);
22705
22706         printf("Connected: %s : %u\n", ci->ntopbuf, (unsigned)ntohs
22707             (sinaddr_client.sin_port));
22708
22709         if ((result = pthread_create(&tid, NULL, client_proc, ci)) != 0) {
22710             fprintf(stderr, "pthread_create: %s\n", strerror(result));
22711             exit(EXIT_FAILURE);
22712         }
22713     }
22714 }
22715
22716     close(sock);
22717
22718     return 0;
22719 }
22720
22721 void *client_proc(void *param)
```

```
22722 {  
22723     CLIENT_INFO *ci = (CLIENT_INFO *)param;  
22724     char buf[BUFFER_SIZE + 1];  
22725     ssize_t result;  
22726  
22727     for (;;) {  
22728         if ((result = recv(ci->sock, buf, BUFFER_SIZE, 0)) == -1)  
22729             exit_sys("recv");  
22730         if (result == 0)  
22731             break;  
22732         buf[result] = '\0';  
22733         if (!strcmp(buf, "quit"))  
22734             break;  
22735         printf("%ld bytes received from %s (%u): %s\n", (long)result, ci-    ↩  
              >ntopbuf, (unsigned)ntohs(ci->sinaddr.sin_port), buf);  
22736         revstr(buf);  
22737         if (send(ci->sock, buf, strlen(buf), 0) == -1)  
22738             exit_sys("send");  
22739     }  
22740  
22741     shutdown(ci->sock, SHUT_RDWR);  
22742     close(ci->sock);  
22743  
22744     free(ci);  
22745  
22746     return NULL;  
22747 }  
22748  
22749 char *revstr(char *str)  
22750 {  
22751     size_t i, k;  
22752     char temp;  
22753  
22754     for (i = 0; str[i] != '\0'; ++i)  
22755         ;  
22756  
22757     for (--i, k = 0; k < i; ++k, --i) {  
22758         temp = str[k];  
22759         str[k] = str[i];  
22760         str[i] = temp;  
22761     }  
22762  
22763     return str;  
22764 }  
22765  
22766 void exit_sys(const char *msg)  
22767 {  
22768     perror(msg);  
22769  
22770     exit(EXIT_FAILURE);  
22771 }  
22772  
22773 /* client.c */
```

```
22774
22775 /* server.c */
22776
22777 #include <stdio.h>
22778 #include <stdlib.h>
22779 #include <string.h>
22780 #include <unistd.h>
22781 #include <sys/socket.h>
22782 #include <netinet/in.h>
22783 #include <arpa/inet.h>
22784
22785 #define BUFFER_SIZE      1024
22786
22787 char *revstr(char *str);
22788 void exit_sys(const char *msg);
22789
22790 int main(int argc, char *argv[])
22791 {
22792     int sock, sock_client;
22793     struct sockaddr_in sinaddr, sinaddr_client;
22794     socklen_t sinaddr_len;
22795     char ntopbuf[INET_ADDRSTRLEN];
22796     in_port_t port;
22797     ssize_t result;
22798     char buf[BUFFER_SIZE + 1];
22799
22800     if (argc != 2) {
22801         fprintf(stderr, "wrong number of arguments!..\n");
22802         exit(EXIT_FAILURE);
22803     }
22804
22805     port = (in_port_t)strtoul(argv[1], NULL, 10);
22806
22807     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
22808         exit_sys("socket");
22809
22810     sinaddr.sin_family = AF_INET;
22811     sinaddr.sin_port = htons(port);
22812     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
22813
22814     if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
22815         exit_sys("bind");
22816
22817     if (listen(sock, 8) == -1)
22818         exit_sys("listen");
22819
22820     printf("Waiting for connection...\n");
22821
22822     sinaddr_len = sizeof(sinaddr_client);
22823     if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,
22824                               &sinaddr_len)) == -1)
22825         exit_sys("accept");
```

```
22826     printf("Connected: %s : %u\n", inet_ntop(AF_INET, &sinaddr_client,
22827                                         ntopbuf, INET_ADDRSTRLEN), (unsigned)ntohs(sinaddr_client.sin_port)); ↵
22828
22829     for (;;) {
22830         if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
22831             exit_sys("recv");
22832         if (result == 0)
22833             break;
22834         buf[result] = '\0';
22835         if (!strcmp(buf, "quit"))
22836             break;
22837         printf("Connected: %s : %u\n", inet_ntop(AF_INET, &sinaddr_client,
22838                                         ntopbuf, INET_ADDRSTRLEN), (unsigned)ntohs
22839                                         (sinaddr_client.sin_port));
22840         revstr(buf);
22841         if (send(sock_client, buf, strlen(buf), 0) == -1)
22842             exit_sys("send");
22843     }
22844
22845     shutdown(sock_client, SHUT_RDWR);
22846     close(sock_client);
22847     close(sock);
22848
22849     return 0;
22850 }
22851
22852     size_t i, k;
22853     char temp;
22854
22855     for (i = 0; str[i] != '\0'; ++i)
22856         ;
22857
22858     for (--i, k = 0; k < i; ++k, --i) {
22859         temp = str[k];
22860         str[k] = str[i];
22861         str[i] = temp;
22862     }
22863
22864     return str;
22865 }
22866 void exit_sys(const char *msg)
22867 {
22868     perror(msg);
22869
22870     exit(EXIT_FAILURE);
22871 }
22872
22873
22874 /
```

```
*-----  
22875    Çok client'lı server soketler için en çok kullanılan modellerden biri  
22876        "select modeli"dir. Bu model hem UNIX/Linux hem de  
22877        Windows sistemlerine kullanılabilmektedir. Bu model şöyle özetlenebilir:  
22878  
22879    1) Önce select fonksiyonunun için "read set" parametresi için içib os  
22880        bir fd_set nesnesi yaratılır.  
22881    2) Dinleme soketi (pasif soket) yaratılır, bind işlemi uygulanır ve bu  
22882        soket "read set" içerisinde FD_SET ile eklenir.  
22883    Artık dinleme soketinde bir bağlantı isteği olduğunda sanki bir okuma  
22884        durumu olmuş gibi select blokesi çözülecektir.  
22885    Eğer select'in blokesi dinleme soketi yüzünden çözülmüşse o noktada  
22886        bizin accept işlemini yapmamız gereklidir.  
22887    3) accept işlemiyle bir client ile bağlantı sağlandığında bu client'in  
22888        soketi de "read set" içerisinde eklenir. Böylece artık  
22889        "read set" içerisinde hem dinleme soketi hem de client soketleri  
22890        bulunur.  
22891    4) Program bir döngü içerisinde select fonksiyonun çağrılması biçiminde  
22892        oluşturulur. select'in blokesi çözüldüğünde hangi sokette  
22893        okuma eyleminin olduğunu bakılır. Eğer dinleme soketinde okuma eylemi  
22894        olmuşsa bu noktada yukarıda da belirtildiği gibi accept  
22895        yapılmalıdır,  
22896        eğer bir client sokette okuma eylemi olmuşsa o client soketten read/  
22897        recv yapılır.  
22898    5) Bir client close ya da shutdown uyguladığında bu da select için bir  
22899        okuma eylemi olmuş gibi ele alınır. Bu durumda programcı read/recv  
22900        yapacak ve buradan eğer 0 değeri elde ederse karşı tarafın soketi  
22901        kapattionı anlayacaktır. Artık programcının bu noktada ilgili client  
22902        soketi kapatıp  
22903        onu "read set" kümesinden çıkartması gereklidir.  
22904  
22905    -----*/  
22906  
22907    /* server.c */  
22908  
22909  
22910    #include <stdio.h>  
22911    #include <stdlib.h>  
22912    #include <string.h>  
22913    #include <unistd.h>  
22914    #include <sys/socket.h>  
22915    #include <netinet/in.h>  
22916    #include <arpa/inet.h>  
22917    #include <sys/select.h>  
22918  
22919    #define BUFFER_SIZE      1024  
22920  
22921    char *revstr(char *str);  
22922    void exit_sys(const char *msg);  
22923  
22924    int main(int argc, char *argv[])
```

```
22911 {  
22912     int sock, sock_client;  
22913     struct sockaddr_in sinaddr, sinaddr_client;  
22914     socklen_t sinaddr_len;  
22915     char ntopbuf[INET_ADDRSTRLEN];  
22916     in_port_t port;  
22917     ssize_t result;  
22918     char buf[BUFFER_SIZE + 1];  
22919     fd_set rset, tset;  
22920     int maxfd;  
22921     int i;  
22922  
22923     if (argc != 2) {  
22924         fprintf(stderr, "wrong number of arguments!..\\n");  
22925         exit(EXIT_FAILURE);  
22926     }  
22927  
22928     port = (in_port_t)strtoul(argv[1], NULL, 10);  
22929     FD_ZERO(&rset);  
22930  
22931     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)  
22932         exit_sys("socket");  
22933  
22934     sinaddr.sin_family = AF_INET;  
22935     sinaddr.sin_port = htons(port);  
22936     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);  
22937  
22938     if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)  
22939         exit_sys("bind");  
22940  
22941     FD_SET(sock, &rset);  
22942     maxfd = sock;  
22943  
22944     if (listen(sock, 8) == -1)  
22945         exit_sys("listen");  
22946  
22947     printf("Waiting for connection...\\n");  
22948  
22949     for (;;) {  
22950         tset = rset;  
22951         if (select(maxfd + 1, &tset, NULL, NULL, NULL) == -1)  
22952             exit_sys("select");  
22953         for (i = 0; i <= maxfd; ++i)  
22954             if (FD_ISSET(i, &tset)) {  
22955                 if (i == sock) {  
22956                     sinaddr_len = sizeof(sinaddr_client);  
22957                     if ((sock_client = accept(sock, (struct sockaddr *)  
22958                         &sinaddr_client, &sinaddr_len)) == -1)  
22959                         exit_sys("accept");  
22960  
22961                         inet_ntop(AF_INET, &sinaddr_client.sin_addr, ntopbuf,  
22962                             INET_ADDRSTRLEN);  
22963                         printf("Connected: %s : %u\\n", ntopbuf, (unsigned)ntohs
```

```
                                (sinaddr_client.sin_port));
22962                         FD_SET(sock_client, &rset);
22963                         if (sock_client > maxfd)
22964                             maxfd = sock_client;
22965                         continue;
22966                     }
22967
22968                     if ((result = recv(i, buf, BUFFER_SIZE, 0)) == -1)
22969                         exit_sys("recv");
22970                     if (result > 0) {
22971                         buf[result] = '\0';
22972                         sinaddr_len = sizeof(sinaddr_client);
22973                         if (getpeername(i, (struct sockaddr *)&sinaddr_client,    ↪
22974                                         &sinaddr_len) == -1)
22975                             exit_sys("getpeername");
22976
22977                         inet_ntop(AF_INET, &sinaddr_client.sin_addr, ntopbuf,    ↪
22978                                         INET_ADDRSTRLEN);
22979                         printf("%ld bytes received from %s (%u): %s\n", (long)    ↪
22980                                         result, ntopbuf, (unsigned)ntohs
22981                                         (sinaddr_client.sin_port), buf);
22982                         revstr(buf);
22983                         if (send(i, buf, strlen(buf), 0) == -1)
22984                             exit_sys("send");
22985                     }
22986                     else {
22987                         shutdown(i, SHUT_RDWR);
22988                         close(i);
22989                         FD_CLR(i, &rset);
22990                     }
22991
22992                     close(sock);
22993             }
22994
22995         char *revstr(char *str)
22996     {
22997         size_t i, k;
22998         char temp;
22999
23000         for (i = 0; str[i] != '\0'; ++i)
23001             ;
23002
23003         for (--i, k = 0; k < i; ++k, --i) {
23004             temp = str[k];
23005             str[k] = str[i];
23006             str[i] = temp;
23007         }
23008
23009         return str;
```

```
23010 }
23011
23012 void exit_sys(const char *msg)
23013 {
23014     perror(msg);
23015
23016     exit(EXIT_FAILURE);
23017 }
23018
23019 /* client.c */
23020
23021 #include <stdio.h>
23022 #include <stdlib.h>
23023 #include <string.h>
23024 #include <unistd.h>
23025 #include <sys/socket.h>
23026 #include <netinet/in.h>
23027 #include <arpa/inet.h>
23028 #include <netdb.h>
23029
23030 #define BUFFER_SIZE      1024
23031
23032 void exit_sys(const char *msg);
23033
23034 int main(int argc, char *argv[])
23035 {
23036     int sock;
23037     struct addrinfo *ai, *ri;
23038     struct addrinfo hints = {0};
23039     char buf[BUFFER_SIZE];
23040     char *str;
23041     ssize_t result;
23042     int sresult;
23043
23044     if (argc != 3) {
23045         fprintf(stderr, "wrong number of arguments!..\n");
23046         exit(EXIT_FAILURE);
23047     }
23048
23049     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
23050         exit_sys("socket");
23051
23052     hints.ai_family = AF_INET;
23053     hints.ai_socktype = SOCK_STREAM;
23054
23055     if ((sresult = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
23056         fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(sresult));
23057         exit(EXIT_FAILURE);
23058     }
23059
23060     for (ri = ai; ri != NULL; ri = ri->ai_next)
23061         if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
23062             break;
```

```
23063
23064     if (ri == NULL)
23065         exit_sys("connect");
23066
23067     freeaddrinfo(ai);
23068
23069     printf("Connected...\n");
23070
23071     for (;;) {
23072         printf("Yazi giriniz:");
23073         fgets(buf, BUFFER_SIZE, stdin);
23074         if ((str = strchr(buf, '\n')) != NULL)
23075             *str = '\0';
23076         if (!strcmp(buf, "quit"))
23077             break;
23078
23079         if ((send(sock, buf, strlen(buf), 0)) == -1)
23080             exit_sys("send");
23081
23082         if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == -1)
23083             exit_sys("recv");
23084         if (result == 0)
23085             break;
23086         buf[result] = '\0';
23087         printf("%ld bytes received from %s (%s): %s\n", (long)result, argv[1], argv[2], buf);
23088     }
23089
23090     shutdown(sock, SHUT_RDWR);
23091     close(sock);
23092
23093     return 0;
23094 }
23095
23096 void exit_sys(const char *msg)
23097 {
23098     perror(msg);
23099
23100     exit(EXIT_FAILURE);
23101 }
23102
23103 /
*-----*
```

23104 Yukarıdaki programda daha önce görmedğimiz getpeername isimli bir
fonksiyon kullanılmıştır. Bu fonksiyon bağlı bir soketi
parametre olarak alır ve karşı tarafın ip adresini ve port numarasını
bize sockaddr_in ya da sockaddr_in6 biçiminde verir.
23105 Tabii aslında server bağlantısını yaptığımda karşı tarafın bilgisini zaten
accept fonksiyonunda almaktadır. B bilgi saklanarak
23106 kullanılabilir. Ancak bu bilgi saklanmamışsa istenildiği zaman
getpeername fonksiyonuyla alınabilemektedir. Fonksiyonun
23107 prototipi şöyledir:

```
23109
23110     int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
23111
23112     Fonksiyonun birinci parametresi soket betimleyicisidir. İkinci parametre ↵
23113     duruma göre bilgilerin yerleştirileceği ssockaddr_in ya da ↵
23114     sockaddr_in6 yapı nesnesinin adresini alır. Son parametre ikinci ↵
23115     parametredeki yapının uzunluğudur. Eğer buraya az bir uzunluk ↵
23116     girilirse ↵
23117     kırma yapılır ve gerçek uzunluk verdiğimiz adresteki nesneye ↵
23118     yerleştirilir. fonksiyon başarı durumunda 0 değerine başarısızlık ↵
23119     değerine geri dönmektedir.
23120
23121     getpprname fonksiyonunun ters işlemini getsockname fonksiyonu ↵
23122     yapmaktadır. Bu fonksiyon kendi bağlı soketimizin ip adresini ve ↵
23123     port numarasını elde etmek için kullanılır. Genellikle b1 fonksiyona ↵
23124     gereksinim duyulmamaktadır.
23125
23126     -----
23127     -----
23128     -----
23129     -----
23130     -----
23131
23132     1) Önce poll fonksiyonu için struct pollfd türünden uygun genişlikte bir ↵
23133     dizi yaratılır. (Tabii programcı isterse dinamik dizi de ↵
23134     kullanabilir)
23135     2) Dinleme soketi (pasif soket) yaratılır, bind işlemi uygulanır ve bu ↵
23136     soket, POLLIN koduya diziye eklenir. Şimdi dizinin sktif eleman ↵
23137     1'dir.
23138     Artık dinleme soketinde bir bağlantı isteği olduğunda sanki POLLIN ↵
23139     işlemi sağlanmış gibi select blokesi çözülecektir. poll fonksiyonunda ↵
23140     dinleme soketine
23141     bağlantı isteği geldiğinde POLLIN olayı gerçekleşir. Eğer poll ↵
23142     fonksiyonunun blokesi dinleme soketi yüzünden çözülmüşse o noktada ↵
23143     bizin accept işlemini
23144     yapmamız gereklidir.
23145     3) accept işlemiyle bir client ile bağlantı sağlandığında bu client'in ↵
```

soketi ve POLLIN olayı da dizinin sonuna eklenir. Böylece dizinin aktif eleman sayısı

23138 gittikçe artacaktır. Dizinin içerisinde hem dinleme soketinin hem de client soketlerin olduğuna dikkat ediniz.

23139 4) Program bir döngü içerisinde poll fonksiyonun çağrılması biçiminde oluşturulur. poll'un blokesi çözüldüğünde hangi sokette

23140 POLLIN eyleminin olduğunu bakılır. Eğer dinleme soketinde POLLIN eylemi olmuşsa bu noktada yukarıda da belirtildiği gibi accept yapılmalıdır,

23141 eğer bir client sokette POLLIN eylemi olmuşsa o client soketten read/recv yapılır.

23142 5) Bir client close ya da shutdown uyguladığında bu da poll için bir POLLIN eylemi oluşturur. Bu durumda programcı read/recv yapacak ve buradan eğer 0 değerini elde ederse karşı tarafın soketi kapattığını anlayacaktır. Artık programcının bu noktada ilgili client soketi kapatıp onu diziden çıkartması gereklidir.

23145

23146 Aşağıda bir poll server modeli görülmektedir.

23147 -----*/

```
23148 /* server.c */
23150
23151 #include <stdio.h>
23152 #include <stdlib.h>
23153 #include <string.h>
23154 #include <unistd.h>
23155 #include <sys/socket.h>
23156 #include <netinet/in.h>
23157 #include <arpa/inet.h>
23158 #include <poll.h>
23159
23160 #define BUFFER_SIZE      1024
23161 #define MAX_CLIENT        1024
23162
23163 char *revstr(char *str);
23164 void exit_sys(const char *msg);
23165
23166 int main(int argc, char *argv[])
23167 {
23168     int sock, sock_client;
23169     struct sockaddr_in sinaddr, sinaddr_client;
23170     socklen_t sinaddr_len;
23171     char ntopbuf[INET_ADDRSTRLEN];
23172     in_port_t port;
23173     ssize_t result;
23174     char buf[BUFFER_SIZE + 1];
23175     struct pollfd pfds[MAX_CLIENT];
23176     int npfds;
23177     int i;
23178
23179     if (argc != 2) {
```



```
23230         if (getpeername(pfds[i].fd, (struct sockaddr *)&sinaddr_client, &sinaddr_len) == -1)
23231             exit_sys("getpeername");
23232         inet_ntop(AF_INET, &sinaddr_client.sin_addr, ntopbuf, INET_ADDRSTRLEN);
23233         printf("%ld bytes received from %s (%u): %s\n", (long) result, ntopbuf, (unsigned)ntohs
23234             (sinaddr_client.sin_port), buf);
23235         revstr(buf);
23236         if (send(pfds[i].fd, buf, strlen(buf), 0) == -1)
23237             exit_sys("send");
23238     } else {
23239         shutdown(pfds[i].fd, SHUT_RDWR);
23240         close(pfds[i].fd);
23241         pfds[i] = pfds[npfds - 1];
23242         --npfds;
23243     }
23244 }
23245 }
23246
23247     close(sock);
23248
23249     return 0;
23250 }
23251
23252 char *revstr(char *str)
23253 {
23254     size_t i, k;
23255     char temp;
23256
23257     for (i = 0; str[i] != '\0'; ++i)
23258         ;
23259
23260     for (--i, k = 0; k < i; ++k, --i) {
23261         temp = str[k];
23262         str[k] = str[i];
23263         str[i] = temp;
23264     }
23265
23266     return str;
23267 }
23268
23269 void exit_sys(const char *msg)
23270 {
23271     perror(msg);
23272
23273     exit(EXIT_FAILURE);
23274 }
23275
23276 /* client.c */
23277
23278 #include <stdio.h>
```

```
23279 #include <stdlib.h>
23280 #include <string.h>
23281 #include <unistd.h>
23282 #include <sys/socket.h>
23283 #include <netinet/in.h>
23284 #include <arpa/inet.h>
23285 #include <netdb.h>
23286
23287 #define BUFFER_SIZE          1024
23288
23289 void exit_sys(const char *msg);
23290
23291 int main(int argc, char *argv[])
23292 {
23293     int sock;
23294     struct addrinfo *ai, *ri;
23295     struct addrinfo hints = {0};
23296     char buf[BUFFER_SIZE];
23297     char *str;
23298     ssize_t result;
23299     int sresult;
23300
23301     if (argc != 3) {
23302         fprintf(stderr, "wrong number of arguments!..\n");
23303         exit(EXIT_FAILURE);
23304     }
23305
23306     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
23307         exit_sys("socket");
23308
23309     hints.ai_family = AF_INET;
23310     hints.ai_socktype = SOCK_STREAM;
23311
23312     if ((sresult = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
23313         fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(sresult));
23314         exit(EXIT_FAILURE);
23315     }
23316
23317     for (ri = ai; ri != NULL; ri = ri->ai_next)
23318         if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
23319             break;
23320
23321     if (ri == NULL)
23322         exit_sys("connect");
23323
23324     freeaddrinfo(ai);
23325
23326     printf("Connected...\n");
23327
23328     for (;;) {
23329         printf("Yazi giriniz:");
23330         fgets(buf, BUFFER_SIZE, stdin);
23331         if ((str = strchr(buf, '\n')) != NULL)
```

```
23332         *str = '\0';
23333         if (!strcmp(buf, "quit"))
23334             break;
23335
23336         if ((send(sock, buf, strlen(buf), 0)) == -1)
23337             exit_sys("send");
23338
23339         if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == -1)
23340             exit_sys("recv");
23341         if (result == 0)
23342             break;
23343         buf[result] = '\0';
23344         printf("%ld bytes received from %s (%s): %s\n", (long)result, argv    ↴
23345             [1], argv[2], buf);
23346     }
23347     shutdown(sock, SHUT_RDWR);
23348     close(sock);
23349
23350     return 0;
23351 }
23352
23353 void exit_sys(const char *msg)
23354 {
23355     perror(msg);
23356
23357     exit(EXIT_FAILURE);
23358 }
23359
23360 /
*-----*
```

23361 23362 Anımsanacağı gibi poll yönteminin Linux sistemlerinde epoll isimli bir ↴
benzeri vardı. Linux sistemlerinde en iyi performansı
23363 bu epoll modeli vermektedir. Bu modelde önce epoll_create ile bir epoll ↴
nesnesi oluşturulur. Sonra dinleme soketi epoll_ctl fonksiyonu ile
23364 ilgi listesine eklenir. Tıpkı poll modelinde olduğu gibi dinleme ↴
soketine bir bağlantı isteği geldiğinde, herhangi bir sokete
23365 okunacak bir bilgi geldiğinde ya da bir soket shutdown ya da close ↴
fonksiyonuyla kapatıldığında EPOLLIN olayı gerçekleşmekte
23366 İşte programcı döngü içerisinde epoll_wait fonksiyonu ile beklemek. ↴
Sonra hangi sokette olay gerçekleşmişse gerektiğini yapar.
23367
23368 epoll modeli Linux'a özgürdür diğer UNIX türevi sistemlerde yoktur.
23369
23370 Aşağıda epoll modelini uygulayan bir "level triggered" server örneği ↴
verilmiştir.
23371
23372 -----*/

23373
23374 /* server.c */

```
23375
23376 #include <stdio.h>
23377 #include <stdlib.h>
23378 #include <string.h>
23379 #include <unistd.h>
23380 #include <sys/socket.h>
23381 #include <netinet/in.h>
23382 #include <arpa/inet.h>
23383 #include <sys/epoll.h>
23384
23385 #define BUFFER_SIZE      1024
23386 #define MAX_CLIENT        1024
23387
23388 char *revstr(char *str);
23389 void exit_sys(const char *msg);
23390
23391 int main(int argc, char *argv[])
23392 {
23393     int sock, sock_client;
23394     int epfd;
23395     struct sockaddr_in sinaddr, sinaddr_client;
23396     socklen_t sinaddr_len;
23397     char ntopbuf[INET_ADDRSTRLEN];
23398     in_port_t port;
23399     ssize_t result;
23400     char buf[BUFFER_SIZE + 1];
23401     int nepes;
23402     int rnepes;
23403     int i;
23404     struct epoll_event epe;
23405     struct epoll_event epes[MAX_CLIENT];
23406
23407     if (argc != 2) {
23408         fprintf(stderr, "wrong number of arguments!..\n");
23409         exit(EXIT_FAILURE);
23410     }
23411
23412     port = (in_port_t)strtoul(argv[1], NULL, 10);
23413
23414     if ((epfd = epoll_create(1024)) == -1)
23415         exit_sys("epoll_create");
23416
23417     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
23418         exit_sys("socket");
23419
23420     sinaddr.sin_family = AF_INET;
23421     sinaddr.sin_port = htons(port);
23422     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
23423
23424     if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
23425         exit_sys("bind");
23426
23427     nepes = 1;
```



```
23471             revstr(buf);
23472             if (send(epes[i].data.fd, buf, strlen(buf), 0) == -1)
23473                 exit_sys("send");
23474         }
23475     else {
23476         shutdown(epes[i].data.fd, SHUT_RDWR);
23477         close(epes[i].data.fd);
23478
23479         --nepes;
23480         printf("Client connection ended, total descriptors
23481             interested: %d\n", nepes);
23482     }
23483 }
23484
23485 close(sock);
23486 close(epfd);
23487
23488 return 0;
23489 }
23490
23491 char *revstr(char *str)
23492 {
23493     size_t i, k;
23494     char temp;
23495
23496     for (i = 0; str[i] != '\0'; ++i)
23497         ;
23498
23499     for (--i, k = 0; k < i; ++k, --i) {
23500         temp = str[k];
23501         str[k] = str[i];
23502         str[i] = temp;
23503     }
23504
23505     return str;
23506 }
23507
23508 void exit_sys(const char *msg)
23509 {
23510     perror(msg);
23511
23512     exit(EXIT_FAILURE);
23513 }
23514
23515 /
*-----*
-----*
```

modda biz gelen bilgilerin hepsini okumalıyız. Aksi takdirde
23519 sürekli bloke çözülecektir ve bu da sonsuz döngü oluşturabilecektir. ↗
Ancak bazı uygulamalarda gelen bilginin hepsinin okunması ↗
istenmeyebilir.

23520 Örneğin programcı gelen bilgi belli bir büyülüğe ulaşınca onu kendi ↗
tamponuna taşımak isteyebilir. Bu durumda düzey tetiklemeli çalışma ↗
uygulanamaz.

23521 Kenar tetiklemeli modda "yeni bilginin gelip gelmediğine" bakılmaktadır. ↗
Yani network tamponunda yeni bilginin olmasına değil yeni bir ↗
gelmesiyle bloke çözülmektedir. (Başka bir deyişle bir epoll_wait ↗
işleminden sonra ancak yeni bir bilgi gelmişse tetikleme oluşur.)

23523

23524 Düzey tetiklemeli IO modellerinde bir bilgi geldiğinde o bilgi okunup ↗
işleme sokulurken başka bir bilginin gelmesi durumunda

23525 bu yeni gelen bilgi ikinci turda ancak ele alınabilmektedir. Buradaki ↗
zaman kaybını engellemek için soketi blokesiz moda sokup döngü ↗
içerisinde

23526 okuma yapmak uygun bir yöntem olabilmektedir. Örneğin:

23527

23528 nepes = epoll(.....);

23529

23530 for (i = 0; i < nepes; ++i) {

23531 if (epes[i].events & EPOLLIN) {

23532 for (;;) {

23533 result = recv(....);

23534 if (result == -1) {

23535 if (errno = EAGAIN)

23536 break;

23537 exit_sys("recv");

23538 }

23539

23540 }

23541 }

23542 }

23543

23544 Bu yöntemde sokete bilgi gelmişse ele alınırken yeni bir bilginin ↗
gelmesi durumunda bu yeni gelen bilgi ikinci tura bırakılmadan

23545 hemen işlenebilmektedir. Bazı performans kritik durumlarda bu yöntem ↗
tercih edilebilmektedir. Ancak bu yöntem için de soketin blokesiz moda ↗

23546 sokulması gereklidir. Soket accept ile edildiğinde default blokeli ↗
moddadır. Soketi blokesiz moda sokmak fcntl fonksiyonuyla yapılabilir. ↗

23547 Ancak nonblocking olarak döngüsel okumalarda sorunlardan biri belli bir ↗
client'tan çok fazla ve sürekli bilgi geldiğinde bu döngüde çok fazla ↗
zaman

23548 geçirilmesi ve bunun sonucunda da diğer soketlerle ilgilenilememesi ↗
olmaktadır. Programcı böyle bir sorunu belli bir limit

23549 koyarak çözebilir.

23550

23551 Kenar tetiklemeli modda eğer programcı her yeni tetiklemede önceden ↗
gelmiş olan tüm bilgileri okumayı garanti etmek istiyorsa ya blokeli ↗
modda read/recv fonksiyonunda talep ettiğinden daha az değerin okunmuş ↗

```
olmasına bakmalı ya da yukarıdaki gibi blokesiz modda bir döngü
23553  içerisinde EAGAIN durumu oluşana kadar read/recv uygulamalıdır.
23554
23555  Epoll yönteminde genel olarak kenar tetiklemeli modun düzey tetiklemeli →
         moddan daha verimli olduğu görülmüştür. Kenar tetiklemeli modda
23556  yeni bir bilgi geldiğinde tetiklemenin yapılması çok sayıda betimleyici →
         ile çalışılırken etkin artışına yol açmaktadır.() Bunun en önemli
23557  nedenlerinden biri çekirdek kodlarının dha az betimleyiciye bakıyor →
         olmasındandır.)
```

23558

23559 Yukarıdaki programı mantıksal olarak kenar tetiklemeli ve döngüsel →
 biçimde aşağıdaki gibi sokabiliriz.

23560

23561 -----*/

```
23562
23563 /* server.c */
23564
23565 #include <stdio.h>
23566 #include <stdlib.h>
23567 #include <string.h>
23568 #include <errno.h>
23569 #include <fcntl.h>
23570 #include <unistd.h>
23571 #include <sys/socket.h>
23572 #include <netinet/in.h>
23573 #include <arpa/inet.h>
23574 #include <sys/epoll.h>
23575
23576 #define BUFFER_SIZE      1024
23577 #define MAX_CLIENT        1024
23578
23579 char *revstr(char *str);
23580 void exit_sys(const char *msg);
23581
23582 int main(int argc, char *argv[])
23583 {
23584     int sock, sock_client;
23585     int epfd;
23586     struct sockaddr_in sinaddr, sinaddr_client;
23587     socklen_t sinaddr_len;
23588     char ntopbuf[INET_ADDRSTRLEN];
23589     in_port_t port;
23590     ssize_t result;
23591     char buf[BUFFER_SIZE + 1];
23592     int nepes;
23593     int rnepes;
23594     int i;
23595     struct epoll_event epe;
23596     struct epoll_event epes[MAX_CLIENT];
23597
23598     if (argc != 2) {
```

```
23600     fprintf(stderr, "wrong number of arguments!..\n");
23601     exit(EXIT_FAILURE);
23602 }
23603
23604     port = (in_port_t)strtoul(argv[1], NULL, 10);
23605
23606     if ((epfd = epoll_create(1024)) == -1)
23607         exit_sys("epoll_create");
23608
23609     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
23610         exit_sys("socket");
23611
23612     sinaddr.sin_family = AF_INET;
23613     sinaddr.sin_port = htons(port);
23614     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
23615
23616     if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
23617         exit_sys("bind");
23618
23619     nepes = 1;
23620     epe.events = EPOLLIN|EPOLLET;
23621     epe.data.fd = sock;
23622     if (epoll_ctl(epfd, EPOLL_CTL_ADD, sock, &epe) == -1)
23623         exit_sys("epoll_ctl");
23624
23625     if (listen(sock, 8) == -1)
23626         exit_sys("listen");
23627
23628     printf("Waiting for connection...\n");
23629
23630     for (;;) {
23631         if ((rnpes = epoll_wait(epfd, epes, nepes, -1)) == -1)
23632             exit_sys("epoll");
23633         for (i = 0; i < rnpes; ++i)
23634             if (epes[i].events & EPOLLIN) {
23635                 if (epes[i].data.fd == sock) {
23636                     sinaddr_len = sizeof(sinaddr_client);
23637                     if ((sock_client = accept(sock, (struct sockaddr *)    ↪
23638                     &sinaddr_client, &sinaddr_len)) == -1)
23639                         exit_sys("accept");
23640                     if (fcntl(sock_client, F_SETFL, fcntl(sock_client,    ↪
23641                         F_GETFL) | O_NONBLOCK) == -1)
23642                         exit_sys("fcntl");
23643
23644                     inet_ntop(AF_INET, &sinaddr_client.sin_addr, ntopbuf,    ↪
23645                         INET_ADDRSTRLEN);
23646
23647                     epe.events = EPOLLIN|EPOLLET;
23648                     epe.data.fd = sock_client;
23649                     if (epoll_ctl(epfd, EPOLL_CTL_ADD, sock_client, &epe) == ↪
23650                         -1)
23651                         exit_sys("epoll_ctl");
23652                     ++nepes;
23653                 }
```

```
23649
23650         printf("New client Connected: %s : %u, total descriptor ↵
23651             interested: %d\n", ntohs(epes[i].data.sin_port), nepes);
23652         continue;
23653     }
23654
23655     for (;;) {
23656         if ((result = recv(epes[i].data.fd, buf, BUFFER_SIZE, 0)) == -1) {
23657             if (errno == EAGAIN)
23658                 break;
23659             exit_sys("recv");
23660         }
23661
23662         if (result > 0) {
23663             buf[result] = '\0';
23664             sinaddr_len = sizeof(sinaddr_client);
23665
23666             if (getpeername(epes[i].data.fd, (struct sockaddr *) ↵
23667             &sinaddr_client, &sinaddr_len) == -1)
23668                 exit_sys("getpeername");
23669             inet_ntop(AF_INET, &sinaddr_client.sin_addr,
23670             ntohs(epes[i].data.sin_port), INET_ADDRSTRLEN);
23671             printf("%ld bytes received from %s (%u): %s\n",
23672             (long)result, ntohs(epes[i].data.sin_port), buf);
23673             revstr(buf);
23674             if (send(epes[i].data.fd, buf, strlen(buf), 0) == -1)
23675                 exit_sys("send");
23676         }
23677         else {
23678             shutdown(epes[i].data.fd, SHUT_RDWR);
23679             close(epes[i].data.fd);
23680
23681             --nepes;
23682             printf("Client connection ended, total descriptors ↵
23683             interested: %d\n", nepes);
23684             break;
23685         }
23686     }
23687     close(sock);
23688     close(epfd);
23689
23690     return 0;
23691 }
23692 char *revstr(char *str)
```

```
23693 {
23694     size_t i, k;
23695     char temp;
23696
23697     for (i = 0; str[i] != '\0'; ++i)
23698         ;
23699
23700     for (--i, k = 0; k < i; ++k, --i) {
23701         temp = str[k];
23702         str[k] = str[i];
23703         str[i] = temp;
23704     }
23705
23706     return str;
23707 }
23708
23709 void exit_sys(const char *msg)
23710 {
23711     perror(msg);
23712
23713     exit(EXIT_FAILURE);
23714 }
23715
23716 /
*-----*
```

-----*

23717 Nihayet soket uygulamalarında asenkron IO modeli de uygulanabilmektedir. ↗
Ancak bu modelim soketlerde taşınabilirlik bakımından bazı
23718 sorunları vardır. POSIX standartları bu modelin soketlerde kullanımı ↗
konusunda gerekli olan bazı gri noktaları açıklamamıştır.
23719 Dolayısıyla asenkron model soketlerde kullanılırken taşınabilirlik ↗
sorunları oluşabilmektedir. Linux sistemlerinde bu model epoll ↗
modelinden
23720 daha yavaş olma eğilimindedir.

23721

23722 Bu modelde daha önce yaptığımız gibi her betimleyici için bir AIO Kontrol ↗
Blok oluşturulur ve sonra okuma işlemi aio_read ile başlatılır.
23723 İşlem bittiğinde sistem bize bir sinyal ya da thread yoluyla bunu ↗
bildirmektedir. Biz de orada yeniden aio_read yaparak işlemi devam ↗
ettiririz.

23724

23725 Bu yöntemde dinleme soketi üzerinde asenkron işlem yapılamamaktadır. ↗
Yani yalnızca accept ile elde edilen soket üzerinde asenkron işlemler
23726 yapılmaktadır.

23727 -----*/

23728
23729 /* server.c */
23730
23731 #include <stdio.h>
23732 #include <stdlib.h>
23733 #include <string.h>
23734 #include <unistd.h>

```
23735 #include <sys/socket.h>
23736 #include <netinet/in.h>
23737 #include <arpa/inet.h>
23738 #include <aio.h>
23739
23740 #define BUFFER_SIZE      1024
23741 #define MAX_CLIENT       1024
23742
23743 typedef struct {
23744     struct aiocb cb;
23745     char buf[BUFFER_SIZE + 1];
23746     char peer_name[INET_ADDRSTRLEN];
23747     in_port_t peer_port;
23748 } IOCB_BUF;
23749
23750 void read_completion_proc(union sigval val);
23751 char *revstr(char *str);
23752 void exit_sys(const char *msg);
23753
23754 int main(int argc, char *argv[])
23755 {
23756     int sock, sock_client;
23757     struct sockaddr_in sinaddr, sinaddr_client;
23758     socklen_t sinaddr_len;
23759     in_port_t port;
23760     IOCB_BUF *aiobuf;
23761
23762     if (argc != 2) {
23763         fprintf(stderr, "wrong number of arguments!..\n");
23764         exit(EXIT_FAILURE);
23765     }
23766
23767     port = (in_port_t)strtoul(argv[1], NULL, 10);
23768
23769     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
23770         exit_sys("socket");
23771
23772     sinaddr.sin_family = AF_INET;
23773     sinaddr.sin_port = htons(port);
23774     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
23775
23776     if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
23777         exit_sys("bind");
23778
23779     if (listen(sock, 8) == -1)
23780         exit_sys("listen");
23781
23782     printf("Waiting for connection...\n");
23783
23784     for (;;) {
23785         if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client, ↴
23786             &sinaddr_len)) == -1)
23787             exit_sys("accept");
```

```
23787
23788     if ((aiobuf = (IOCB_BUF *)calloc(1, sizeof(IOCB_BUF))) == NULL)
23789         exit_sys("calloc");
23790
23791     inet_ntop(AF_INET, &sinaddr_client.sin_addr, aiobuf->peer_name,
23792                 INET_ADDRSTRLEN);
23793     aiobuf->peer_port = htons(sinaddr_client.sin_port);
23794
23795     printf("New client Connected: %s : %u\n", aiobuf->peer_name,
23796           (unsigned)aiobuf->peer_port);
23797
23798     aiobuf->cb.aio_fildes = sock_client;
23799     aiobuf->cb.aio_offset = 0;
23800     aiobuf->cb.aio_buf = aiobuf->buf;
23801     aiobuf->cb.aio_nbytes = BUFFER_SIZE;
23802     aiobuf->cb.aio_reqprio = 0;
23803     aiobuf->cb.aio_sigevent.sigev_notify = SIGEV_THREAD;
23804     aiobuf->cb.aio_sigevent.sigev_value.sival_ptr = aiobuf;
23805     aiobuf->cb.aio_sigevent.sigev_notify_function =
23806             read_completion_proc;
23807
23808     if (aio_read(&aiobuf->cb) == -1)
23809         exit_sys("aio_read");
23810
23811     close(sock);
23812
23813     return 0;
23814 }
23815 void read_completion_proc(union sigval val)
23816 {
23817     IOCB_BUF *aiobuf = (IOCB_BUF *)val.sival_ptr;
23818     ssize_t n;
23819
23820     if ((n = aio_return(&aiobuf->cb)) == -1)
23821         exit_sys("aio_return");
23822
23823     if (n == 0) {
23824         shutdown(aiobuf->cb.aio_fildes, SHUT_RDWR);
23825         close(aiobuf->cb.aio_fildes);
23826         free(aiobuf);
23827         return;
23828     }
23829
23830     aiobuf->buf[n] = '\0';
23831     printf("%ld bytes received from %s (%u): %s\n", (long)n, aiobuf-
23832           >peer_name, (unsigned)aiobuf->peer_port, (char *)aiobuf->cb.aio_buf);
23833     revstr(aiobuf->buf);
23834     if (send(aiobuf->cb.aio_fildes, (void *)aiobuf->cb.aio_buf, strlen
23835           (aiobuf->buf), 0) == -1)
23836         exit_sys("send");
```

```
23835     if (aio_read(&aiobuf->cb) == -1)
23836         exit_sys("aio_read");
23837     }
23838
23839     char *revstr(char *str)
23840     {
23841         size_t i, k;
23842         char temp;
23843
23844         for (i = 0; str[i] != '\0'; ++i)
23845             ;
23846
23847         for (--i, k = 0; k < i; ++k, --i) {
23848             temp = str[k];
23849             str[k] = str[i];
23850             str[i] = temp;
23851         }
23852
23853         return str;
23854     }
23855
23856     void exit_sys(const char *msg)
23857     {
23858         perror(msg);
23859
23860         exit(EXIT_FAILURE);
23861     }
23862
23863 /* client.c */
23864
23865 #include <stdio.h>
23866 #include <stdlib.h>
23867 #include <string.h>
23868 #include <unistd.h>
23869 #include <sys/socket.h>
23870 #include <netinet/in.h>
23871 #include <arpa/inet.h>
23872 #include <netdb.h>
23873
23874 #define BUFFER_SIZE          1024
23875
23876 void exit_sys(const char *msg);
23877
23878 int main(int argc, char *argv[])
23879 {
23880     int sock;
23881     struct addrinfo *ai, *ri;
23882     struct addrinfo hints = {0};
23883     char buf[BUFFER_SIZE];
23884     char *str;
23885     ssize_t result;
23886     int sresult;
23887 }
```

```
23888     if (argc != 3) {
23889         fprintf(stderr, "wrong number of arguments!..\\n");
2390         exit(EXIT_FAILURE);
2391     }
2392
2393     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
2394         exit_sys("socket");
2395
2396     hints.ai_family = AF_INET;
2397     hints.ai_socktype = SOCK_STREAM;
2398
2399     if ((sresult = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
2400         fprintf(stderr, "getaddrinfo: %s\\n", gai_strerror(sresult));
2401         exit(EXIT_FAILURE);
2402     }
2403
2404     for (ri = ai; ri != NULL; ri = ri->ai_next)
2405         if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
2406             break;
2407
2408     if (ri == NULL)
2409         exit_sys("connect");
2410
2411     freeaddrinfo(ai);
2412
2413     printf("Connected...\\n");
2414
2415     for (;;) {
2416         printf("Yazi giriniz:");
2417         fgets(buf, BUFFER_SIZE, stdin);
2418         if ((str = strchr(buf, '\\n')) != NULL)
2419             *str = '\\0';
2420         if (!strcmp(buf, "quit"))
2421             break;
2422
2423         if ((send(sock, buf, strlen(buf), 0)) == -1)
2424             exit_sys("send");
2425
2426         if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == -1)
2427             exit_sys("recv");
2428         if (result == 0)
2429             break;
2430         buf[result] = '\\0';
2431         printf("%ld bytes received from %s (%s): %s\\n", (long)result, argv    ↵
2432               [1], argv[2], buf);
2433     }
2434
2435     shutdown(sock, SHUT_RDWR);
2436     close(sock);
2437
2438     return 0;
2439 }
```

```
23940 void exit_sys(const char *msg)
23941 {
23942     perror(msg);
23943
23944     exit(EXIT_FAILURE);
23945 }
23946
23947 /
*-----*
-----*
23948 UDP (User Datagram Protocol) bağlantılı olmayan bir protokoldür.      ↵
23949 Dolayısıyla bir taraf bir tarafa hiç bağlanmadan onun IP      ↵
23950 adresini ve port numarasını bilerek UDP paketlerini gönderebilir.      ↵
23951 Gönderen taraf alan tarafın paketi alıp olmadığını bilmez.      ↵
23952 Bir akış kontrolü yoktur. Dolayısıyla alan taraf bilgi kaçıracabilir. UDP ↵
23953 tabii ki TCP'ye göre daha hızlıdır. Zaten TCP bir bakıma      ↵
23954 UDP'nin organize edilmiş bağlantılı biçimidir.      ↵
23955 UDP bağlantılı olmadığı için burada "client" ve "server" terimleri tam      ↵
23956 oturmamaktadır. Ancak yine de genellikle bilgiyi gönderen tarafa      ↵
23957 "client" alan tarafa "server" denilmektedir. UDP özellikle periyodik      ↵
23958 kısa bilgilerin gönderildiği ve alındığı durumlarda tercih      ↵
23959 edilmektedir.      ↵
23960 UDP haberleşmesinde bilgi alan tarafın (server) bilgi kaçırlabilmesi söz      ↵
23961 konusu olabileceğinden dolayı böyle kaçırımlarda sistemde önemli bir      ↵
23962 aksamanın olmaması gereklidir. Örneğin birtakım makineler belli      ↵
23963 periyotlarda server'a "ben çalışıyorum" demek için periyodik UDP      ↵
23964 paketleri yollayabilir.      ↵
23965 Server da hangi makinenin çalıştığını anlayabilir. Bir araba simülatörü      ↵
23966 arabanın durumunu UDP paketleriyle dış dünyaya verebilir. Ya da      ↵
23967 örneğin      ↵
23968 bir görüntü aktarımı UDP paketleriyle yapılabilir. Bir UDP paketi 64K      ↵
23969 gibi bir sınıra sahiptir. Büyük verilerin UDP ile gönderilmesi için      ↵
23970 programcının      ↵
23971 paketlere kendisinin manuel numara vermesi gerekebilir. Zaten TCP      ↵
23972 protokolü bu şekilde bir numaralandırmayı kendi içerisinde      ↵
23973 yapmaktadır. UDP haberleşmesinin      ↵
23974 önemli bir farkı da "broadcasting" işlemidir. Yerel ağıda belli bir      ↵
23975 makine tüm host'lara UDP paketini gönderebilir. TCP'de böyle bir      ↵
23976 broadcasting mekanizması      ↵
23977 yoktur.      ↵
23978
23979 UDP server programın tasarıımı şöyledir:
23980
23981 1) Server socket fonksiyonuyla soketi SOCK_DGRAM parametresiyle yaratır.
23982 2) Soketi bind fonksiyonuyla bağlar.
23983 3) recvfrom fonksiyonuyla gelen paketleri alır ve işler
23984 4) Haberleşme bitince soketi close ile kapatır.
23985
23986 UDP Client Programın tasarıımı da şöyledir:
23987
23988 1) Client socket fonksiyonuyla soketi SOCK_DGRAM parametresiyle yaratır.
23989 2) Client isteği bağlı olarak soketi bind fonksiyonuyla bağlayabilir.
```

23974 3) Client server'ın host isminden hareketle server'ın IP adresini →
 gethostbyname ya da getaddrinfo fonksiyonuyla elde edebilir.

23975 4) Client sendto fonksiyonuyla UDP paketlerini gönderir.

23976 5) Haberleşme birince client close fonksiyonuyla soketi kapatır.

23977

23978 recvfrom fonksiyonu UDP paketini okumak için kullanılmaktadır. →
 Fonksiyonun prototipi şöyledir:

23979

23980 ssize_t recvfrom(int socket, void *buffer, size_t length, int flags, →
 struct sockaddr *address, socklen_t *address_len);

23981

23982 Fonksiyonun birinci parametresi bind edilen soketi belirtir. ikinci →
 parametre alınacak bilginin yerleştirileceğin adresi belirtir. Üçüncü →
 parametre

23983 ikinci parametredeki dizinin uzunluğunu belirtir. Eğer buradaki değer →
 değer UDP paketindeki gönderilmiş olan byte sayısından daha az ise →
 kırıplarak

23984 diziye yerleştirme yapmaktadır. flgas parametresi birkaç seçenek →
 sahiptir. 0 girilebilir. fonksionun dördüncü parametresi UDP paketin →
 gönderen tarafın

23985 IP adresinin ve port numarasının yerleştirileceği sockaddr_in yapısının →
 adresini alır. Son parametre ise bu yapının uzunluğunu tutan int →
 nesnenin adresini almaktadır.

23986 Eğer bu değer küçük girilirse fonksiyon asıl değeri buraya →
 yerleştirmektedir. Fonksiyon başarı durumunda UDP paketindeki byte →
 sayısına, başarısızlık durumunda
 -1 değerine geri dönmektedir.

23987

23988 sendto fonksiyonu da şöyledir:

23989

23990

23991 ssize_t sendto(int socket, const void *message, size_t length, int →
 flags, const struct sockaddr *dest_addr, socklen_t dest_len);

23992

23993 Fonksiyonun parametreleri recfrom da olduğu gibidir. Ancak buffer yönü →
 terstir. Fonksiyon blokeli modda UDP paketinin tamamı network →
 tamponuna

23994 yazılına kadar blokeye yol açmaktadır. Başarı durumunda network →
 tamponuna yazılan byte sayısına başarısızlık durumunda -1'e geri →
 dönmektedir.

23995

23996

23997 Aşağıda örnek bir UDP-Server ve Client_program verilmiştir.

23998 -----*/

23999

24000 /* udp-server.c */

24001

24002 #include <stdio.h>

24003 #include <stdlib.h>

24004 #include <string.h>

24005 #include <unistd.h>

24006 #include <sys/socket.h>

24007 #include <netinet/in.h>

```
24008 #include <arpa/inet.h>
24009
24010 #define BUFFER_SIZE      4096
24011
24012 char *revstr(char *str);
24013 void exit_sys(const char *msg);
24014
24015 int main(int argc, char *argv[])
24016 {
24017     int sock;
24018     struct sockaddr_in sinaddr, sinaddr_client;
24019     socklen_t sinaddr_len;
24020     char ntopbuf[INET_ADDRSTRLEN];
24021     in_port_t port;
24022     ssize_t result;
24023     char buf[BUFFER_SIZE + 1];
24024
24025     if (argc != 2) {
24026         fprintf(stderr, "wrong number of arguments!..\n");
24027         exit(EXIT_FAILURE);
24028     }
24029
24030     port = (in_port_t)strtoul(argv[1], NULL, 10);
24031
24032     if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
24033         exit_sys("socket");
24034
24035     sinaddr.sin_family = AF_INET;
24036     sinaddr.sin_port = htons(port);
24037     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
24038
24039     if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
24040         exit_sys("bind");
24041
24042     printf("Waiting for client data...\n");
24043
24044     for (;;) {
24045         sinaddr_len = sizeof(sinaddr_client);
24046         if ((result = recvfrom(sock, buf, BUFFER_SIZE, 0, (struct sockaddr *) &sinaddr_client, &sinaddr_len)) == -1)
24047             exit_sys("recvfrom");
24048
24049         buf[result] = '\0';
24050         inet_ntop(AF_INET, &sinaddr_client.sin_addr, ntopbuf,
24051                   INET_ADDRSTRLEN);
24052         printf("%ld bytes received from %s (%u): %s\n", (long)result,
24053               ntopbuf, (unsigned)ntohs(sinaddr_client.sin_port), buf);
24054
24055         revstr(buf);
24056         if (sendto(sock, buf, strlen(buf), 0, (struct sockaddr *) &sinaddr_client, sizeof(struct sockaddr)) == -1)
24057             exit_sys("sendto");
24058     }
```

```
24057     close(sock);
24058
24059     return 0;
24060 }
24061 }
24062
24063 char *revstr(char *str)
24064 {
24065     size_t i, k;
24066     char temp;
24067
24068     for (i = 0; str[i] != '\0'; ++i)
24069         ;
24070
24071     for (--i, k = 0; k < i; ++k, --i) {
24072         temp = str[k];
24073         str[k] = str[i];
24074         str[i] = temp;
24075     }
24076
24077     return str;
24078 }
24079
24080 void exit_sys(const char *msg)
24081 {
24082     perror(msg);
24083
24084     exit(EXIT_FAILURE);
24085 }
24086
24087 /* udp-client.c */
24088
24089 #include <stdio.h>
24090 #include <stdlib.h>
24091 #include <string.h>
24092 #include <unistd.h>
24093 #include <sys/socket.h>
24094 #include <netinet/in.h>
24095 #include <arpa/inet.h>
24096 #include <netdb.h>
24097
24098 #define BUFFER_SIZE          4096
24099
24100 void exit_sys(const char *msg);
24101
24102 int main(int argc, char *argv[])
24103 {
24104     int sock;
24105     struct addrinfo *ai;
24106     struct addrinfo hints = {0};
24107     struct sockaddr_in sinaddr;
24108     int sinaddr_len;
24109     char buf[BUFFER_SIZE];
```

```
24110     char ntopbuf[INET_ADDRSTRLEN];
24111     char *str;
24112     ssize_t result;
24113     int sresult;
24114
24115     if (argc != 3) {
24116         fprintf(stderr, "wrong number of arguments!..\n");
24117         exit(EXIT_FAILURE);
24118     }
24119
24120     if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
24121         exit_sys("socket");
24122
24123     hints.ai_family = AF_INET;
24124     hints.ai_socktype = SOCK_DGRAM;
24125
24126     if ((sresult = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
24127         fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(sresult));
24128         exit(EXIT_FAILURE);
24129     }
24130
24131     for (;;) {
24132         printf("Yazi giriniz:");
24133         fgets(buf, BUFFER_SIZE, stdin);
24134         if ((str = strchr(buf, '\n')) != NULL)
24135             *str = '\0';
24136         if (!strcmp(buf, "quit"))
24137             break;
24138
24139         if (sendto(sock, buf, strlen(buf), 0, ai->ai_addr, sizeof(struct
24140             sockaddr)) == -1)
24141             exit_sys("sendto");
24142
24143         sinaddr_len = sizeof(sinaddr);
24144         if ((result = recvfrom(sock, buf, BUFFER_SIZE, 0, (struct sockaddr
24145             *)&sinaddr, &sinaddr_len)) == -1)
24146             exit_sys("recvfrom");
24147
24148         buf[result] = '\0';
24149         inet_ntop(AF_INET, &sinaddr.sin_addr, ntopbuf, INET_ADDRSTRLEN);
24150         printf("%ld bytes received from %s (%u): %s\n", (long)result,
24151             ntopbuf, (unsigned)ntohs(sinaddr.sin_port), buf);
24152     }
24153
24154     return 0;
24155 }
24156
24157 void exit_sys(const char *msg)
24158 {
24159     perror(msg);
```

```
24160
24161     exit(EXIT_FAILURE);
24162 }
24163
24164 /
*-----*
----- Server uygulamalarında server'in bir client'tan gelen isteği yerine      ↵
      getirmesi bir zaman kaybı oluşturabilmektektir. Server bir      ↵
      client ile uğraşırken diğer client'ların istekleri mecburen bekletilir.      ↵
      İşte bu durumu en aza indirmek için select, poll, epoll      ↵
      modellerinde server bir client'in isteğini bir thread ile yerine      ↵
      getirebilir. Böylece birden fazla client'a aynı anda hizmet      ↵
      verebilecektir.
24165     Örneğin select modelinde bu işlem şöyle yapılabilir:
24166
24167     for (;;) {
24168         select(...);
24169         if (<belli bir sokete bilgi geldi>) {
24170             <thread oluştur ve işlemi thread'e devret>
24171         }
24172     }
24173
24174     Tabii burada küçük bir işlem için yeni bir thread'in yaratılıp yok      ↵
      edilmesi etkin bir yöntem değildir. Çünkü bilindiği gibi      ↵
      thread'lerin yaratılıp yok edilmeleri de dikkate değer bir zaman kaybı      ↵
      oluşturmaktadır. İşte bu tür durumlarda "thread havuzları (thread      ↵
      pools)"      ↵
24175     kullanılabilir. Thread havuzlarında zaten belli bir miktar thread      ↵
      yaratılmış ve bekler durumda (suspend durumda) tutulmaktadır. Böylece      ↵
      client'tan gelen      ↵
24176     isteğin bu thread'lerden biri tarafından gerçekleştirilmesi uygun olur.      ↵
      Tabii havuzda ne kadar thread'in bekletileceği neselesi vardır. Eğer      ↵
      genel amaçlı
24177     bir thread havuzu yazılacaksa havuzda bekletilecek thread dinamik bir      ↵
      biçimde belirlenebilir. (Yani çok fazla gereksinim olduğunda havuzu      ↵
      büyütmek,
24178     gereksinim azaldığında küçültmek gibi.) Daha önceden de belirttiğimiz      ↵
      gibi UNIX/Linux sistemlerinde standart bir thread havuzu mekanizması      ↵
      yoktur.
24179     Halbuki örneğin Windows sistemlerinde işletim sistemi tarafından sunulan      ↵
      threda havuz mekanizması bulunmaktadır. Bu durumda UNIX/Linux      ↵
      sistemlerinde
24180     başkaları tarafından yazılmış thread havuzları kullanılabilir ya da      ↵
      uygulamaya yönelik basit bir havuz mekanizması oluşturulabilir.
24181
24182 -----*/
24183
24184 /-----*
----- recv ve send fonksiyonlarının read ve write fonksiyonlarından tek farkı      ↵
```

24190 flags parametresidir. Biz yukarıdaki örneklerde bu parametreyi 0
24190 geçtiğ. Dolayısıyla yukarıdaki örneklerde kullandığımız recv ve send
24190 fonksiyonlarının read ve write fonksiyonlarından hiçbir farkı yoktur.
24191 Pekiyi bu flag değerleri neler olabilir? İşte POSIX standartlarında recv
24191 fonksiyonundaki flag değerleri şunlardan biri ya da birden fazlası
24191 olabilir:
24192
24193 MSG_PEEK: Bu seçenekte bilgi reive tamponundan alınır ama oradan
24193 silinemez. (Belki de programcı mesaj uygunsa okumak istemektedir.)
24194 MSG_OOB: Outof-band data (urgent data) denilen okumalar için
24194 kullanılmaktadır.
24195 MSG_WAITALL: Bu seçenekte n byte okunmak istendiğinde bu n byte kesin
24195 olarak okunana kadar blokede beklenir. Fakat bu durumda bir sinyal
24195 geldiğinde
24196 yine recv -1 ile geri döner ve errno EINTR ile set edilir.
24197
24198 send fonksiyonundaki POSIX bayrakları da şunlardır:
24199
24200 MSG_EOR: Soket türü SOCK_SEQPACKET ise kaydı sonlandırmakta kullanılır.
24201 MSG_OOB: Outof-band data gönderimi için kullanılmaktadır.
24202 MSG_NOSIGNAL: Normal olarak send ya da write işlemi yapılırken karşı
24202 taraf soketi kapatmışsa bu fonksiyonların çağrıldığı tarafta
24203 SIGPIPE sinyali oluşmaktadır. Bu bayrak kullanılırsa SIGPIPE sinyali
24203 oluşmaz send -1 ile geri döner ve errno EPIPE değeri ile set edilir.
24204
24205 Linux POSIX'in bayraklarından daha fazlasını bulundurmaktadır. Örneğin
24205 recv ve send işleminde MSG_DONTWAIT bir çağrımlık "nonblocking" etki
24206 yatırmaktadır. Yani recv sırasında network tamponunda hiç bilgi yoksa
24206 recv bloke olmaz, -1 ile geri döner errno EAGAIN değeri ile set
24206 edilir.
24207 send işlemi sırasında da network tamponu dolu ise send bloke olmaz -1
24207 ile geri döner errno EAGAIN değeri ile set edilir.
24208
24209 -----*/
24210
24211 /*-----
24212 Client server uygulamaların çoğu client server'a isteğini yazışal
24212 biçimde iletmektedir. Fakat istek binary biçimde de
24213 iletilebilir. bu durumda server'ın soketten belli bir miktar
24213 okuayabilecek biçimde orhganize edilmesi gereklidir. Bazen client
24214 önce server'a okunacak mesajın uzunluğunu gönderir. Sonra da mesajın
24214 kendisi gönderir. Bu yöntem text ya da binary tabanlı
24214 yapılabilmektedir.
24215 Bu durumda programının belli bir miktar okuyana kadar biriktirme
24215 yapması gereklidir. Garantili n byte okumak için recv fonksiyonu
24215 MSG_WAITALL
24216 bayrağı ile kullanılabilir. Ancak bu da sıkıntılıdır. Çünkü select,
24216 poll, ya da asenkron io modellerinde n byte okuyana kadar bloke
24216 olusturmak
24217 tasarıma terstir. Bu tür durumlarda daha önce borularda yaptığımız gibi

```
        "index, left" mekanizmasını kullanabiliriz.  
24218  
24219     Aşağıda her defasında BUFFER_SIZE (10) byte bilginin hepsi okunduğunda →  
          onları işleme soken bir epoll modelli server örneği  
24220     verilmiştir.  
24221  
24222 -----*/  
24223  
24224 /* server.c */  
24225  
24226 #include <stdio.h>  
24227 #include <stdlib.h>  
24228 #include <string.h>  
24229 #include <unistd.h>  
24230 #include <sys/socket.h>  
24231 #include <netinet/in.h>  
24232 #include <arpa/inet.h>  
24233 #include <sys/epoll.h>  
24234  
24235 #define BUFFER_SIZE      10  
24236 #define MAX_CLIENT       1024  
24237  
24238 typedef struct {  
24239     int fd;  
24240     char buf[BUFFER_SIZE + 1];  
24241     ssize_t index;  
24242     ssize_t left;  
24243 } CLIENT_INFO;  
24244  
24245 char *revstr(char *str);  
24246 void exit_sys(const char *msg);  
24247  
24248 int main(int argc, char *argv[])  
24249 {  
24250     int sock, sock_client;  
24251     int epfd;  
24252     struct sockaddr_in sinaddr, sinaddr_client;  
24253     socklen_t sinaddr_len;  
24254     char ntopbuf[INET_ADDRSTRLEN];  
24255     in_port_t port;  
24256     ssize_t result;  
24257     int nepes;  
24258     int rnpes;  
24259     int i;  
24260     struct epoll_event epe;  
24261     struct epoll_event epes[MAX_CLIENT];  
24262     CLIENT_INFO *ci;  
24263  
24264     if (argc != 2) {  
24265         fprintf(stderr, "wrong number of arguments!..\\n");  
24266         exit(EXIT_FAILURE);  
24267     }
```

```
24268
24269     port = (in_port_t)strtoul(argv[1], NULL, 10);
24270
24271     if ((epfd = epoll_create(1024)) == -1)
24272         exit_sys("epoll_create");
24273
24274     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
24275         exit_sys("socket");
24276
24277     sinaddr.sin_family = AF_INET;
24278     sinaddr.sin_port = htons(port);
24279     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
24280
24281     if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
24282         exit_sys("bind");
24283
24284     nepes = 1;
24285     epe.events = EPOLLIN;
24286     epe.data.fd = sock;
24287     if (epoll_ctl(epfd, EPOLL_CTL_ADD, sock, &epe) == -1)
24288         exit_sys("epoll_ctl");
24289
24290     if (listen(sock, 8) == -1)
24291         exit_sys("listen");
24292
24293     printf("Waiting for connection...\n");
24294
24295     for (;;) {
24296         if ((rnpes = epoll_wait(epfd, epes, nepes, -1)) == -1)
24297             exit_sys("epoll");
24298         for (i = 0; i < rnpes; ++i)
24299             if (epes[i].events & EPOLLIN) {
24300                 if (epes[i].data.fd == sock) {
24301                     sinaddr_len = sizeof(sinaddr_client);
24302                     if ((sock_client = accept(sock, (struct sockaddr *)    ↪
24303                     &sinaddr_client, &sinaddr_len)) == -1)
24304                         exit_sys("accept");
24305
24306                     inet_ntop(AF_INET, &sinaddr_client.sin_addr, ntopbuf,    ↪
24307                     INET_ADDRSTRLEN);
24308
24309                     if ((ci = (CLIENT_INFO *)malloc(sizeof(CLIENT_INFO))) == ↪
24310                         NULL)
24311                         exit_sys("malloc");
24312
24313                     ci->fd = sock_client;
24314                     ci->index = 0;
24315                     ci->left = BUFFER_SIZE;
24316                     epe.events = EPOLLIN;
24317                     epe.data.ptr = ci;
24318
24319                     if (epoll_ctl(epfd, EPOLL_CTL_ADD, sock_client, &epe) == ↪
```

```
-1)
24318         exit_sys("epoll_ctl");
24319         ++nepes;
24320
24321         printf("New client Connected: %s : %u, total descriptor ↵
24322             interested: %d\n", ntohs(sinaddr_client.sin_port), nepes);
24323         continue;
24324     }
24325
24326     ci = (CLIENT_INFO *)epes[i].data.ptr;
24327     if ((result = recv(ci->fd, ci->buf + ci->index, ci->left,
24328         0)) == -1)
24329         exit_sys("recv");
24330     if (result > 0) {
24331         ci->index += result;
24332         ci->left -= result;
24333         if (ci->left == 0) {
24334             ci->buf[ci->index] = '\0';
24335             sinaddr_len = sizeof(sinaddr_client);
24336
24337             if (getpeername(ci->fd, (struct sockaddr *)
24338                 &sinaddr_client, &sinaddr_len) == -1)
24339                 exit_sys("getpeername");
24340             inet_ntop(AF_INET, &sinaddr_client.sin_addr,
24341                 ntohsbuf, INET_ADDRSTRLEN);
24342             printf("%ld bytes received from %s (%u): %s\n",
24343                 (long)ci->index, ntohs(sinaddr_client.sin_port),
24344                 ci->buf);
24345             ci->index = 0;
24346             ci->left = BUFFER_SIZE;
24347         }
24348     } else {
24349         shutdown(ci->fd, SHUT_RDWR);
24350         close(ci->fd);
24351         --nepes;
24352         free(ci);
24353         printf("Client connection ended, total descriptors ↵
24354             interested: %d\n", nepes);
24355     }
24356 }
24357
24358     return 0;
24359 }
24360 void exit_sys(const char *msg)
24361 {
```

```
24362     perror(msg);
24363
24364     exit(EXIT_FAILURE);
24365 }
24366
24367 /* client.c */
24368
24369 #include <stdio.h>
24370 #include <stdlib.h>
24371 #include <string.h>
24372 #include <unistd.h>
24373 #include <sys/socket.h>
24374 #include <netinet/in.h>
24375 #include <arpa/inet.h>
24376 #include <netdb.h>
24377
24378 #define BUFFER_SIZE      1024
24379
24380 void exit_sys(const char *msg);
24381
24382 int main(int argc, char *argv[])
24383 {
24384     int sock;
24385     struct addrinfo *ai, *ri;
24386     struct addrinfo hints = {0};
24387     char buf[BUFFER_SIZE];
24388     char *str;
24389     int result;
24390
24391     if (argc != 3) {
24392         fprintf(stderr, "wrong number of arguments!..\n");
24393         exit(EXIT_FAILURE);
24394     }
24395
24396     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
24397         exit_sys("socket");
24398
24399     hints.ai_family = AF_INET;
24400     hints.ai_socktype = SOCK_STREAM;
24401
24402     if ((result = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
24403         fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
24404         exit(EXIT_FAILURE);
24405     }
24406
24407     for (ri = ai; ri != NULL; ri = ri->ai_next)
24408         if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
24409             break;
24410
24411     if (ri == NULL)
24412         exit_sys("connect");
24413
24414     freeaddrinfo(ai);
```

```
24415
24416     printf("Connected...\n");
24417
24418     for (;;) {
24419         printf("Yazi giriniz:");
24420         fgets(buf, BUFFER_SIZE, stdin);
24421         if ((str = strchr(buf, '\n')) != NULL)
24422             *str = '\0';
24423         if (!strcmp(buf, "quit"))
24424             break;
24425
24426         if ((send(sock, buf, strlen(buf), 0)) == -1)
24427             exit_sys("send");
24428     }
24429
24430     shutdown(sock, SHUT_RDWR);
24431     close(sock);
24432
24433     return 0;
24434 }
24435
24436 void exit_sys(const char *msg)
24437 {
24438     perror(msg);
24439
24440     exit(EXIT_FAILURE);
24441 }
24442
24443 /
-----*-----*-----*
-----*-----*-----*
24444     Aşağıda biriktirmeli işlem yapan server'ın select ile gerçekleştirimi
          görülmektedir.
24445 -----*-----*-----*/
24446
24447 /* server.c */
24448
24449 #include <stdio.h>
24450 #include <stdlib.h>
24451 #include <string.h>
24452 #include <unistd.h>
24453 #include <sys/socket.h>
24454 #include <netinet/in.h>
24455 #include <arpa/inet.h>
24456 #include <sys/select.h>
24457
24458 #define BUFFER_SIZE      10
24459 #define MAX_CLIENT       1024
24460
24461 char *revstr(char *str);
24462 void exit_sys(const char *msg);
24463
```

```
24464     typedef struct {
24465         int fd;
24466         char buf[BUFFER_SIZE + 1];
24467         ssize_t index;
24468         ssize_t left;
24469     } CLIENT_INFO;
24470
24471     int main(int argc, char *argv[])
24472     {
24473         int sock, sock_client;
24474         struct sockaddr_in sinaddr, sinaddr_client;
24475         socklen_t sinaddr_len;
24476         char ntopbuf[INET_ADDRSTRLEN];
24477         in_port_t port;
24478         ssize_t result;
24479         fd_set rset, tset;
24480         CLIENT_INFO clients[MAX_CLIENT];
24481         int maxfd;
24482         int nclients = 0;
24483         int i;
24484
24485         if (argc != 2) {
24486             fprintf(stderr, "wrong number of arguments!..\n");
24487             exit(EXIT_FAILURE);
24488         }
24489
24490         port = (in_port_t)strtoul(argv[1], NULL, 10);
24491         FD_ZERO(&rset);
24492
24493         if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
24494             exit_sys("socket");
24495
24496         sinaddr.sin_family = AF_INET;
24497         sinaddr.sin_port = htons(port);
24498         sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
24499
24500         if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
24501             exit_sys("bind");
24502
24503         FD_SET(sock, &rset);
24504         maxfd = sock;
24505
24506         if (listen(sock, 8) == -1)
24507             exit_sys("listen");
24508
24509         printf("Waiting for connection...\n");
24510
24511         for (;;) {
24512             tset = rset;
24513             if (select(maxfd + 1, &tset, NULL, NULL, NULL) == -1)
24514                 exit_sys("select");
24515
24516             if (FD_ISSET(sock, &tset)) {
```

```
24517         sinaddr_len = sizeof(sinaddr_client);
24518         if ((sock_client = accept(sock, (struct sockaddr *)    ↵
24519             &sinaddr_client, &sinaddr_len)) == -1)
24520             exit_sys("accept");
24521
24522         clients[nclients].fd = sock_client;
24523         clients[nclients].index = 0;
24524         clients[nclients].left = BUFFER_SIZE;
24525         ++nclients;
24526
24527         inet_ntop(AF_INET, &sinaddr_client.sin_addr, ntopbuf,    ↵
24528             INET_ADDRSTRLEN);
24529         printf("Connected: %s : %u\n", ntopbuf, (unsigned)ntohs    ↵
24530             (sinaddr_client.sin_port));
24531         FD_SET(sock_client, &rset);
24532         if (sock_client > maxfd)
24533             maxfd = sock_client;
24534     }
24535
24536     for (i = 0; i < nclients; ++i)
24537         if (FD_ISSET(clients[i].fd, &tset)) {
24538             if ((result = recv(clients[i].fd, clients[i].buf + clients    ↵
24539                 [i].index, clients[i].left, 0)) == -1)
24540                 exit_sys("recv");
24541             if (result > 0) {
24542                 clients[i].index += result;
24543                 clients[i].left -= result;
24544                 if (clients[i].left == 0) {
24545                     clients[i].buf[clients[i].index] = '\0';
24546                     sinaddr_len = sizeof(sinaddr_client);
24547                     if (getpeername(clients[i].fd, (struct sockaddr *)    ↵
24548                         &sinaddr_client, &sinaddr_len) == -1)
24549                         exit_sys("getpeername"));
24550
24551                 inet_ntop(AF_INET, &sinaddr_client.sin_addr,    ↵
24552                     ntopbuf, INET_ADDRSTRLEN);
24553                 printf("%ld bytes received from %s (%u): %s\n",    ↵
24554                     (long)clients[i].index, ntopbuf, (unsigned)ntohs    ↵
24555                     (sinaddr_client.sin_port), clients[i].buf);
24556
24557                 clients[i].index = 0;
24558                 clients[i].left = BUFFER_SIZE;
24559             }
24560         }
24561     }
```

```
24562     }
24563
24564     close(sock);
24565
24566     return 0;
24567 }
24568
24569 void exit_sys(const char *msg)
24570 {
24571     perror(msg);
24572
24573     exit(EXIT_FAILURE);
24574 }
24575
24576 /* client.c */
24577
24578 #include <stdio.h>
24579 #include <stdlib.h>
24580 #include <string.h>
24581 #include <unistd.h>
24582 #include <sys/socket.h>
24583 #include <netinet/in.h>
24584 #include <arpa/inet.h>
24585 #include <netdb.h>
24586
24587 #define BUFFER_SIZE      1024
24588
24589 void exit_sys(const char *msg);
24590
24591 int main(int argc, char *argv[])
24592 {
24593     int sock;
24594     struct addrinfo *ai, *ri;
24595     struct addrinfo hints = {0};
24596     char buf[BUFFER_SIZE];
24597     char *str;
24598     int result;
24599
24600     if (argc != 3) {
24601         fprintf(stderr, "wrong number of arguments!..\n");
24602         exit(EXIT_FAILURE);
24603     }
24604
24605     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
24606         exit_sys("socket");
24607
24608     hints.ai_family = AF_INET;
24609     hints.ai_socktype = SOCK_STREAM;
24610
24611     if ((result = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
24612         fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
24613         exit(EXIT_FAILURE);
24614     }
```

```
24615
24616     for (ri = ai; ri != NULL; ri = ri->ai_next)
24617         if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
24618             break;
24619
24620     if (ri == NULL)
24621         exit_sys("connect");
24622
24623     freeaddrinfo(ai);
24624
24625     printf("Connected...\n");
24626
24627     for (;;) {
24628         printf("Yazi giriniz:");
24629         fgets(buf, BUFFER_SIZE, stdin);
24630         if ((str = strchr(buf, '\n')) != NULL)
24631             *str = '\0';
24632         if (!strcmp(buf, "quit"))
24633             break;
24634
24635         if ((send(sock, buf, strlen(buf), 0)) == -1)
24636             exit_sys("send");
24637     }
24638
24639     shutdown(sock, SHUT_RDWR);
24640     close(sock);
24641
24642     return 0;
24643 }
24644
24645 void exit_sys(const char *msg)
24646 {
24647     perror(msg);
24648
24649     exit(EXIT_FAILURE);
24650 }
24651
24652 /
*-----*-----*-----*
```

24653 Bir tarafın diğer tarafa bir dosyanın içeriğini göndermesi (dosya aktarımı) sık rastlanan durumlardandır. Bunun şüphesiz klasik gerçekleştirim

24654 biçim dosyadan blok blok okuma yapıp onları soketten yollamak ve benzer biçimde bir döngü içerisinde recv fonksiyonuyla alarak dosyaya yazmak olabilir.

24655 Linux sistemlerinde (başka bazı sistemlerde de var) bu işlem için sendfile isimli bir sistem fonksiyonu bulundurulmuştur. Fonksiyonu prototipi şöyledir:

```
24656 ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

24658 Fonskiyonun birinci parametresi soket betimleyicisini almaktadır. İkinci

```
parametre normal bir dosya betimleyicisidir. Fonksiyonun üçüncü →
parametresi →
24660 aktarımın başlatılacağı offset değerini belirtmektedir. Bu parametre →
NULL geçilirse aktarım ikinci parametreyle belirtilen dosyanın →
başından itibaren yapılır. →
24661 Fonksiyon sonlandıktan sonra dosya göstericisinin yeni değeri bu adrese →
yerleştirilmektedir. Bu parametre NULL da geçilebilir. Bu durumda →
kopyalama ikinci parametreye →
24662 belirtilen dosyanın başından itibaren yapılır. Eğer bu offset →
parametresi NULL geçilmediyse ikinci parametreyle belirtilen dosyaya →
ilişkin dosya göstericisinin →
24663 değeri değiştirilmez ama bu parametre NULL geçilmişse bu dosyaya ilişkin →
dosya göstericisinin degeir değiştirilir. Son parametre kaç byte →
transfer edileceğidir. →
24664 Fonksiyon başarı durumunda aktarılan byte sayına başarısızlık durumunda →
-1 değerine geri döner. Bu fonksiyon sayesinde hiç user düzeyinde →
tampon kullanmadan tek bir hamlede →
24665 bir dosyanın içeriğini soketten karşı tarafa gönderebiliriz. Fonksiyon →
başarı durumunda tampona yazılmış olan byte sayısıyla başarısızlık →
durumunda -1 değeri ile geri döner →
24666 →
24667 sendfile fonksiyonun POSIX standartlarında bulunmadığına Linux'a özgü →
olduğu dikkat ediniz. →
24668 →
24669 sendfile sistem fonksiyonu kernel 2.6.33'e kadar yalnızca dosyadan →
sokete kopyalama yapmak için kullanılıyordu. Bu versiyondan sonra →
fonksiyon herhangi iki betimleyici →
24670 arasında çalışabilir hale getirilmiştir. copy_file_range fonksiyonu ise →
halen yalnızca normal dosyalar için çalışabilmektedir. →
24671 →
24672 -----*/ →
24673 →
24674 / →
-----*/ →
----- →
24675 TCP/IP client server uygulamaların önemli bir bölümünde mesajlaşma text →
tabanlı yapılmaktadır. Yani client server'a isteğini →
24676 bir yazı göndererek iletir, server da client'a yanıtı bir yazı biçiminde →
verir. Gerçekten de IP ailesinin TELNET, SSH, HTTP, →
24677 POP3, IMAP gibi protokoller hep yazışal işlem yapmaktadır. Bir satır →
bilginin gönderilmesinde hiçbir problem yoktur. Ancak bir →
24678 satırlık bilginin etkin bir biçimde alınabilmesi için bir algoritma →
kullanmak gereklidir. Çünkü karakterlerin tek tek read ya da →
24679 recv ile okunması yavaş bir seçenektedir. Aşağıda her çağrıda →
soketten bir satır okuyan örnek bir fonksiyon verilmiştir. →
24680 -----*/ →
24681 →
24682 int sock_readline(int sock, char *buf, size_t len) →
24683 { →
24684     char *bufx = buf; →
24685     static char *bp;
```

```
24686     static ssize_t count = 0;
24687     static char b[2048];
24688
24689     if (len <= 2)
24690         return -1;
24691
24692     while (--len > 0) {
24693         if (--count <= 0) {
24694             if ((count = recv(sock, b, sizeof(b), 0)) == -1)
24695                 return -1;
24696             if (count == 0)
24697                 return 0;
24698             bp = b;
24699         }
24700         *buf++ = *bp++;
24701         if (buf[-1] == '\n' && buf[-2] == '\r') {
24702             *buf = '\0';
24703             break;
24704         }
24705     }
24706
24707     return buf - bufx;
24708 }
24709
24710 /
*-----*
```

24711 IP ailesinin uygulama katmanındaki Telnet, SSH, HTTP, POP3, SMTP gibi protokoller yukarıda belirtildiği gibi hep yazışal işlem yapmaktadır. ↗
24712 Örneğin POP3 (Post Office Version 3) protokolü kabaca şöyledir:

24713

24714 1) Client server'a (POP3 server) 110 numaralı porttan (POP3 için well known port) bağlanır. Bağlanma başarılı ise ↗
24715 Server Client'a "+OK <mesaj>\r\n" biçiminde bir satır yollar.

24716

24717 2) Bundan sonra client "USER <user name>\r\n" komutu ile mail adresini ↗
24718 iletir. (Örneğin test@csystem.org). Server eğer bu işlem başarılıysa "+OK\r\n" yazısını gönderir.

24719

24720 3) Client bundan sonra "PASS <password>\r\n" yazısını server'a ↗
24721 göndererek password'ü iletmiş olur. Eğer password başarılı ise server "+OK Logged in.\r\n" yazısını balarısız ise "-ERR [AUTH] Authentication ↗
24722 failed.\r\n" yazısını göndermektedir.

24723

24724 4) Şimdi client posta kutusundaki postalar hakkında "LIST\r\n" komutu ↗
24725 ile bilgileri elde eder. Bu komuta karşı server birden fazla
24726 satırдан oluşan şöyle bir vermektedir:
24727
24728 1 8088
24729 2 1568
24730 3 3962
24731 .

24731 Son satırın '.' ile bittiğine dikkat ediniz.

24732

24733 5) Client herhangi bir postayı "RETR <no>\r\n" ile elde edebilir. Buna →
karşı server e-postanın içeriğini birden fazla satır biçiminde →
yollayacaktır.

24734 E-posta düz text ya da genellikle HTML olarak gönderilmiş olabilir. e- →
posta'nın içeriğinde MIME Type zaten belirtilemktedir. Bu konuda POP3 →
ve MIME
dokğanlarına başvurabilirsiniz.

24735

24736

24737 Bu protokolde server'ın olumlu yanıtlarında "+OK" olumlusuz →
yanıtlarında "-ERR" biçiminde satırlar gönderilmemektedir.

24738

24739 6) Nihayet client işini bitirdikten sonra server'a "QUIT\t\n" komutunu →
gönderir. Server'da soketi kapatır. Client da sketin kapatılmış →
olduğunu anlar
ve işini sonlandırır.

24740

24741

24742 Aşağıda POP3 işlemi yapan TELNET benzeri bir program örneği verilmiştir. →
Programda bir thread şürekli server'dan gelen satırları blokeli →
biçimde okumaktadır.

24743

24744 -----*/

24745

24746 /* pop3client.c */

24747

24748 #include <stdio.h>

24749 #include <stdlib.h>

24750 #include <string.h>

24751 #include <unistd.h>

24752 #include <errno.h>

24753 #include <pthread.h>

24754 #include <signal.h>

24755 #include <sys/socket.h>

24756 #include <netinet/in.h>

24757 #include <arpa/inet.h>

24758 #include <netdb.h>

24759

24760 #define BUFFER_SIZE 1024

24761

24762 void *thread_proc(void *param);

24763 int sock_readline(int sock, char *buf, size_t len);

24764 void sigusr1_handler(int sno);

24765 void exit_sys(const char *msg);

24766

24767 typedef struct {

24768 int sock;

24769 pthread_t tid;

24770 } THREAD_PARAM;

24771

24772 int main(int argc, char *argv[])

24773 {

```
24774     int sock;
24775     struct addrinfo *ai, *ri;
24776     struct addrinfo hints = {0};
24777     char buf[BUFFER_SIZE + 1];
24778     char *str;
24779     int result;
24780     pthread_t tid;
24781     struct sigaction sa;
24782     THREAD_PARAM *tp;
24783
24784     if (argc != 3) {
24785         fprintf(stderr, "wrong number of arguments!..\n");
24786         exit(EXIT_FAILURE);
24787     }
24788
24789     sa.sa_handler = sigusr1_handler;
24790     sa.sa_flags = 0;
24791     sigemptyset(&sa.sa_mask);
24792
24793     if (sigaction(SIGUSR1, &sa, NULL) == -1)
24794         exit_sys("sigaction");
24795
24796     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
24797         exit_sys("socket");
24798
24799     hints.ai_family = AF_INET;
24800     hints.ai_socktype = SOCK_STREAM;
24801
24802     if ((result = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
24803         fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
24804         exit(EXIT_FAILURE);
24805     }
24806
24807     for (ri = ai; ri != NULL; ri = ri->ai_next)
24808         if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
24809             break;
24810
24811     if (ri == NULL)
24812         exit_sys("connect");
24813
24814     freeaddrinfo(ai);
24815
24816     if ((tp = malloc(sizeof(THREAD_PARAM))) == NULL)
24817         exit_sys("malloc");
24818     tp->sock = sock;
24819     tp->tid = pthread_self();
24820     if ((result = pthread_create(&tid, NULL, thread_proc, tp) != 0)) {
24821         fprintf(stderr, "pthread_create: %s\n", strerror(result));
24822         exit(EXIT_FAILURE);
24823     }
24824
24825     for (;;) {
24826         if (fgets(buf, BUFFER_SIZE, stdin) == NULL)
```

```
24827         break;
24828     if (errno == EINTR)
24829         break;
24830     if ((str = strchr(buf, '\n')) != NULL)
24831         *str = '\0';
24832     if (!strcmp(buf, "quit"))
24833         break;
24834
24835     strcat(buf, "\r\n");
24836
24837     if (send(sock, buf, strlen(buf), 0) == -1)
24838         exit_sys("send");
24839 }
24840
24841 shutdown(sock, SHUT_RDWR);
24842 close(sock);
24843
24844 return 0;
24845 }
24846
24847 void *thread_proc(void *param)
24848 {
24849     char buf[BUFFER_SIZE + 1];
24850     THREAD_PARAM *tp = (THREAD_PARAM *)param;
24851     int result;
24852
24853     for (;;) {
24854         if ((result = sock_readline(tp->sock, buf, BUFFER_SIZE)) == -1)
24855             exit_sys("sock_readline");
24856         if (!result)
24857             break;
24858
24859         printf("%s", buf);
24860     }
24861
24862     if ((result = pthread_kill(tp->tid, SIGUSR1)) != 0) {
24863         fprintf(stderr, "pthread_create: %s\n", strerror(result));
24864         exit(EXIT_FAILURE);
24865     }
24866
24867     return NULL;
24868 }
24869
24870 int sock_readline(int sock, char *buf, size_t len)
24871 {
24872     char *bufx = buf;
24873     static char *bp;
24874     static ssize_t count = 0;
24875     static char b[2048];
24876
24877     if (len <= 2)
24878         return -1;
24879 }
```

```
24880     while (--len > 0) {
24881         if (--count <= 0) {
24882             if ((count = recv(sock, b, sizeof(b), 0)) == -1)
24883                 return -1;
24884             if (count == 0)
24885                 return 0;
24886             bp = b;
24887         }
24888         *buf++ = *bp++;
24889         if (buf[-1] == '\n' && buf[-2] == '\r') {
24890             *buf = '\0';
24891             break;
24892         }
24893     }
24894
24895     return buf - bufx;
24896 }
24897
24898 void sigusr1_handler(int sno)
24899 {
24900 }
24901
24902 void exit_sys(const char *msg)
24903 {
24904     perror(msg);
24905
24906     exit(EXIT_FAILURE);
24907 }
24908
24909 /
*-----*
```

-----*

24910 Aslında POP3'te olduğu gibi pek çok protokol her komuta en az 1 satır yanıt vermektedir. Eğer server komuta karşı bir satırdan daha fazla bir yanıt veriyorsa genellikle bu ilk satırda ya da son satırda açığa vurulmaktadır. Örneğin POP3 protokolünde LIST komutu birden fazla satırlık bilgi gönderebilmektedir. Ancak son satır özellikle protokolde '..' biçiminde tutulmuştur. Böylece client programı yazan programcı bu '..' satırını gördüğünde satır okuma işlemini bitirilebilmektedir. Benzer biçimde POP3 protokolünde RETR komutunun birinci satırında e-postanın kaç byte'lık bir bilgi içeriği belirtilmektedir. Böylece POP3 client programını yazan programcı bu sayıyı alarak tam o kadar byte okuyabilir.

24914 Diğer diğer protokollerdeki mantıkta aslında çok benzerdir.

24915
24916
24917 Aşağıdaki program yukarıdaki programın thread'sız dolayısıyla daha uygun bir versiyondur. Bu programda POP3 komutlarındaki yanıta da bakılarak porttan uygun miktarda okumalar yapılmıştır.

24919 -----*/

24920
24921 #include <stdio.h>

```
24922 #include <stdlib.h>
24923 #include <cctype.h>
24924 #include <string.h>
24925 #include <unistd.h>
24926 #include <errno.h>
24927 #include <sys/socket.h>
24928 #include <netinet/in.h>
24929 #include <arpa/inet.h>
24930 #include <netdb.h>
24931
24932 #define BUFFER_SIZE      1024
24933
24934 void proc_list(int sock);
24935 void proc_retr(int sock);
24936 void getcmd(const char *buf, char *cmd);
24937 int sock_readline(int sock, char *buf, size_t len);
24938 void exit_sys(const char *msg);
24939
24940 int main(int argc, char *argv[])
24941 {
24942     int sock;
24943     struct addrinfo *ai, *ri;
24944     struct addrinfo hints = {0};
24945     char bufsend[BUFFER_SIZE + 1];
24946     char bufrecv[BUFFER_SIZE + 1];
24947     char *str;
24948     ssize_t result;
24949     char cmd[BUFFER_SIZE];
24950
24951     if (argc != 3) {
24952         fprintf(stderr, "wrong number of arguments!..\\n");
24953         exit(EXIT_FAILURE);
24954     }
24955
24956     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
24957         exit_sys("socket");
24958
24959     hints.ai_family = AF_INET;
24960     hints.ai_socktype = SOCK_STREAM;
24961
24962     if ((result = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
24963         fprintf(stderr, "getaddrinfo: %s\\n", gai_strerror(result));
24964         exit(EXIT_FAILURE);
24965     }
24966
24967     for (ri = ai; ri != NULL; ri = ri->ai_next)
24968         if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
24969             break;
24970
24971     if (ri == NULL)
24972         exit_sys("connect");
24973
24974     freeaddrinfo(ai);
```

```
24975
24976     if (sock_readline(sock, bufrecv, BUFFER_SIZE) == -1)
24977         exit_sys("sock_readline");
24978     printf("%s", bufrecv);
24979
24980     for (;;) {
24981         printf(">");
24982         fflush(stdout);
24983         if (fgets(bufsend, BUFFER_SIZE, stdin) == NULL)
24984             break;
24985         if ((str = strchr(bufsend, '\n')) != NULL)
24986             *str = '\0';
24987
24988         strcat(bufsend, "\r\n");
24989
24990         if (send(sock, bufsend, strlen(bufsend), 0) == -1)
24991             exit_sys("send");
24992
24993         getcmd(bufsend, cmd);
24994         if (!strcmp(cmd, "LIST"))
24995             proc_list(sock);
24996         else if (!strcmp(cmd, "RETR"))
24997             proc_retr(sock);
24998         else {
24999             if ((result = sock_readline(sock, bufrecv, BUFFER_SIZE)) == -1)
25000                 exit_sys("sock_readline");
25001             if (result == 0)
25002                 break;
25003             printf("%s", bufrecv);
25004             if (!strcmp(cmd, "QUIT"))
25005                 break;
25006         }
25007     }
25008
25009     shutdown(sock, SHUT_RDWR);
25010     close(sock);
25011
25012     return 0;
25013 }
25014
25015 void proc_list(int sock)
25016 {
25017     ssize_t result;
25018     char bufrecv[BUFFER_SIZE];
25019
25020     do {
25021         if ((result = sock_readline(sock, bufrecv, BUFFER_SIZE)) == -1)
25022             exit_sys("sock_readline");
25023         if (result == 0)
25024             break;
25025         printf("%s", bufrecv);
25026     } while (*bufrecv != '.');
25027 }
```

```
25028
25029 void proc_retr(int sock)
25030 {
25031     ssize_t result;
25032     char bufrecv[BUFFER_SIZE + 1];
25033     ssize_t n;
25034     int i, ch;
25035
25036     for (i = 0;; ++i) {
25037         if ((result = recv(sock, &ch, 1, 0)) == -1)
25038             exit_sys("sock_readline");
25039         if (result == 0)
25040             return;
25041         if ((bufrecv[i] = ch) == '\n')
25042             break;
25043     }
25044     bufrecv[i] = '\0';
25045
25046     printf("%s", bufrecv);
25047     n = (ssize_t)strtol(bufrecv + 3, NULL, 10);
25048     printf("%ld\n", n);
25049
25050     while (n > 0) {
25051         if ((result = recv(sock, bufrecv, BUFFER_SIZE, 0)) == -1)
25052             exit_sys("recv");
25053         if (result == 0)
25054             break;
25055         bufrecv[result] = '\0';
25056         printf("%s", bufrecv);
25057         fflush(stdout);
25058         n -= result;
25059     }
25060 }
25061
25062 void getcmd(const char *buf, char *cmd)
25063 {
25064     int i;
25065
25066     for (i = 0; buf[i] != '\0' && !isspace(buf[i]); ++i)
25067         cmd[i] = buf[i];
25068     cmd[i] = '\0';
25069 }
25070
25071 int sock_readline(int sock, char *buf, size_t len)
25072 {
25073     char *bufx = buf;
25074     static char *bp;
25075     static ssize_t count = 0;
25076     static char b[2048];
25077
25078     if (len <= 2)
25079         return -1;
25080 }
```

```
25081     while (--len > 0) {
25082         if (--count <= 0) {
25083             if ((count = recv(sock, b, sizeof(b), 0)) == -1)
25084                 return -1;
25085             if (count == 0)
25086                 return 0;
25087             bp = b;
25088         }
25089         *buf++ = *bp++;
25090         if (buf[-1] == '\n' && buf[-2] == '\r') {
25091             *buf = '\0';
25092             break;
25093         }
25094     }
25095
25096     return buf - bufx;
25097 }
25098
25099 void exit_sys(const char *msg)
25100 {
25101     perror(msg);
25102
25103     exit(EXIT_FAILURE);
25104 }
25105
25106 /
*-----
```

25107 Soketler yaratıldıktan sonra onların bazı özellikleri setsockopt isimli ↗
fonksiyonla değiştirilebilir ve getsockopt isimli fonksiyonla
da alınabilir. setsockopt fonksiyonunun prototipi şyledir:

25108 int setsockopt(int socket, int level, int option_name, const void ↗
*option_value, socklen_t option_len);

25109 Fonksiyonun birinci parametresi özelliği değiştirecek soketi belirtir. ↗
ikinci parametresi değişimin hangi düzeyde yapılacağını belirten bir
sembolik sabit olarak girilir. Soket düzeyi için her zaman SOL_SOCKET ↗
girilmelidir. Üçüncü parametre hangi özelliğin değiştirileceğini ↗
belirtmektedir.

25110 Dördüncü parametre değiştirecek özelliğin değeri bulduğu nesnenin ↗
adresini almaktadır. Son parametre dördüncü parametredeki nesnenin ↗
uzunluğunu belirtmektedir.

25111 Fonksiyon başarı durumunda 0, başarısızlık durumunda -1 değerine geri ↗
döner.

25112 Soket seçeneğini elde etmek için de getsockopt fonksiyonu ↗
kullanılmaktadır:

25113 int getsockopt(int socket, int level, int option_name, void *restrict ↗
option_value, socklen_t *restrict option_len);

25114 Parametreler setsockopt'ta olduğu gibidir. Yalnızca dördüncü ↗

parametrenin yönü değişiktir.

25122
25123 Tipik soket seçenekleri (üçüncü parametre) şunlardan biri olabilir:
25124 SO_ACCEPTCONN
25125 SO_BROADCAST
25126 SO_DEBUG
25127 SO_DONTROUTE
25128 SO_ERROR
25129 SO_KEEPALIVE
25130 SO_LINGER
25131 SO_OOBINLINE
25132 SO_RCVBUF
25133 SO_RCVLOWAT
25134 SO_RCVTIMEO
25135 SO_REUSEADDR
25136 SO_SNDBUF
25137 SO SNDLOWAT
25138 SO_SNDTIMEO
25139 SO_TYPE
25140
25141 Burada bizim için şimdilik önemli olan birkaç seçenek vardır: ↗
SO_BROADCAST, SO_OOBINLINE, SO_SNDBUF, SO_RECVBUF, SO_REUSEADDR.
25142
25143 Örneğin:
25144
25145
25146 int buflen;
25147 int optsize = sizeof(int)
25148 ...
25149 if (getsockopt(sock_client, SOL_SOCKET, SO_RCVBUF, &buflen, &optsize) == ↗
-1)
25150 exit_sys("getsockopt");
25151
25152 -----*/ ↗
25153
25154 /* -----*/ ↗
25155 SO_REUSEADDR seçeneği belli bir port için bind işlemi yapmış bir ↗
server'ın sonlanması sonucunda bu server'ın yeniden çalıştırılıp
25156 aynı portu bind edebilmesi için kullanılmaktadır. Bir portu bind eden ↗
server, bir client ile bağlandıktan sonra çökerse, ya da herhangi
25157 bir biçimde sonanırsa işletim sistemleri o portun yeniden belli bir ↗
sure bind edilmesini engellemektedir. Bunun nedeni eski çalışan server ↗
ile
25158 yeni çalışacak olan server'ın göndereceği ve alacağı paketlerin ↗
karışabilme olasılığıdır. Eski paketlerin ağda maksimum bir geçerlilik ↗
süresi vardır.
25159 İşletim sistemi de bunun iki katı kadar bir süre (2 dakika civarı, neden ↗
iki katı olduğu protokolün aşağı seviyeli çalışması ile ilgilidir) bu ↗
protun yeniden
25160 bind edilmesini engellemektedir. İşte eğer bu soket seçeneği kullanılırsa ↗

```
        artık sonlanan ya da çöken bir server hemen yeniden      ↵
        çalıştırıldığında
25161    bind işlemi sırasında "Address already in use" biçiminde bir hata ile      ↵
        karşılaşılacak olacaktır. Örneğin:
25162
25163    int sockapt = 1;
25164    ...
25165    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &sockopt, sizeof      ↵
        (sockopt)) == -1)
25166        exit_sys("setsockopt");
25167
25168    SO_REUSEADDR seçeneğini set etmek için int bir nesne alıp onun içerisinde      ↵
        sıfır dışı bir değer yerleştirdi, onun adresini setsockopt      ↵
        fonksiyonunun dördüncü
25169    parametresine girmek gereklidir. Bu nesneye 0 girip fonksiyonu çağrırsak      ↵
        bu özelliği kapatmış oluruz. Fonksiyonun dördüncü parametresine adresi      ↵
        girilecek nesnenin
25170    seçeneğe göre değişebileceğine dikkat ediniz.
25171
25172    SO_REUSEADDR bayrağı daha önce başka bir program tarafından bind edilmiş      ↵
        soketin ikinci kez diğer bir program tarafından bind edilmesi için      ↵
        kullanılmamaktadır.
25173    Eğer böyle bir ihtiyaç varsa (nadiren olabilir) Linux'ta (fakat POSIX'te      ↵
        değil) SO_REUSEPORT soket seçeneği kullanılmalıdır. Bu soket seçeneği      ↵
        benzer biçimde
25174    Windows sistemlerinde SO_EXCLUSIVEADDRUSE biçimindedir.
25175
25176    Aşağıdaki server programını client ile bağlandıktan sonra      ↵
25177    Ctrl+C ile sonlandırın. Sonra yeniden çalışmaya çalışınız.      ↵
        SO_REUSEADDR seçeneği kullanıldığından dolayı bir sorun ile      ↵
        karşılaşılacak olacaktır.
25178    Daha sonra server programdan o kısmı silerek yeniden denemeyi yapınız.
25179  -----
-----*/
```

```
25180
25181 /* server.c */
25182
25183 #include <stdio.h>
25184 #include <stdlib.h>
25185 #include <string.h>
25186 #include <unistd.h>
25187 #include <sys/socket.h>
25188 #include <netinet/in.h>
25189 #include <arpa/inet.h>
25190
25191 #define BUFFER_SIZE      1024
25192
25193 void exit_sys(const char *msg);
25194
25195 int main(int argc, char *argv[])
25196 {
25197     int sock, sock_client;
25198     struct sockaddr_in sinaddr, sinaddr_client;
```

```
25199     socklen_t sinaddr_len;
25200     char ntopbuf[INET_ADDRSTRLEN];
25201     in_port_t port;
25202     ssize_t result;
25203     char buf[BUFFER_SIZE + 1];
25204     intsockopt = 1;
25205
25206     if (argc != 2) {
25207         fprintf(stderr, "wrong number of arguments!..\n");
25208         exit(EXIT_FAILURE);
25209     }
25210
25211     port = (in_port_t)strtoul(argv[1], NULL, 10);
25212
25213     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
25214         exit_sys("socket");
25215
25216     if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &sockopt, sizeof
25217             (sockopt)) == -1)      ↵
25218         exit_sys("setsockopt");
25219
25220     sinaddr.sin_family = AF_INET;
25221     sinaddr.sin_port = htons(port);
25222     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
25223
25224     if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
25225         exit_sys("bind");
25226
25227     if (listen(sock, 8) == -1)
25228         exit_sys("listen");
25229
25230     printf("Waiting for connection...\n");
25231
25232     sinaddr_len = sizeof(sinaddr_client);
25233     if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,
25234             &sinaddr_len)) == -1)      ↵
25235         exit_sys("accept");
25236
25237     for (;;) {
25238         if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
25239             exit_sys("recv");
25240         if (result == 0)
25241             break;
25242         buf[result] = '\0';
25243         if (!strcmp(buf, "quit"))
25244             break;
25245         printf("%ld bytes received from %s (%u): %s\n", (long)result,
25246               ntopbuf, (unsigned)ntohs(sinaddr_client.sin_port), buf);
25247     }
```

```
25247
25248     shutdown(sock_client, SHUT_RDWR);
25249     close(sock_client);
25250     close(sock);
25251
25252     return 0;
25253 }
25254
25255 void exit_sys(const char *msg)
25256 {
25257     perror(msg);
25258
25259     exit(EXIT_FAILURE);
25260 }
25261
25262 /* client.c */
25263
25264 #include <stdio.h>
25265 #include <stdlib.h>
25266 #include <string.h>
25267 #include <unistd.h>
25268 #include <sys/socket.h>
25269 #include <netinet/in.h>
25270 #include <arpa/inet.h>
25271 #include <netdb.h>
25272
25273 #define BUFFER_SIZE      1024
25274
25275 void exit_sys(const char *msg);
25276
25277 int main(int argc, char *argv[])
25278 {
25279     int sock;
25280     struct addrinfo *ai, *ri;
25281     struct addrinfo hints = {0};
25282     char buf[BUFFER_SIZE];
25283     char *str;
25284     int result;
25285
25286     if (argc != 3) {
25287         fprintf(stderr, "wrong number of arguments!..\n");
25288         exit(EXIT_FAILURE);
25289     }
25290
25291     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
25292         exit_sys("socket");
25293
25294     hints.ai_family = AF_INET;
25295     hints.ai_socktype = SOCK_STREAM;
25296
25297     if ((result = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
25298         fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
25299         exit(EXIT_FAILURE);
```

```
25300     }
25301
25302     for (ri = ai; ri != NULL; ri = ri->ai_next)
25303         if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
25304             break;
25305
25306     if (ri == NULL)
25307         exit_sys("connect");
25308
25309     freeaddrinfo(ai);
25310
25311     printf("Connected...\n");
25312
25313     for (;;) {
25314         printf("Yazi giriniz:");
25315         fgets(buf, BUFFER_SIZE, stdin);
25316         if ((str = strchr(buf, '\n')) != NULL)
25317             *str = '\0';
25318         if (!strcmp(buf, "quit"))
25319             break;
25320
25321         if ((send(sock, buf, strlen(buf), 0)) == -1)
25322             exit_sys("send");
25323     }
25324
25325     shutdown(sock, SHUT_RDWR);
25326     close(sock);
25327
25328     return 0;
25329 }
25330
25331 void exit_sys(const char *msg)
25332 {
25333     perror(msg);
25334
25335     exit(EXIT_FAILURE);
25336 }
25337
25338 /
-----  

-----  

25339     OOB (Out-Of_Band) Data bazı stream protokollerinde olan bir özellikleştir. →
25340     Örneğin TCP protokolü bunu 1 byte olarak desteklemektedir. →
25341     OOB Verisine TCP'de "Acil (Urgent)" veri de denilmektedir. Bunun amacı →
25342     OOB verisinin normal stream sırasında değil daha önce gönderilenlerin →
25343     -eğer hedef host'ta henüz ele alınmamışlarsa- önünde ele →
25344     alınabilmesidir. Yani biz TCP'de birtakım verileri gönderdikten sonra →
25345     OOB verisini gönderirsek →
25346     bu veri önce göndermiş olduğumuzdan daha önde işleme sokulabilir. →
25347     Böylece uygulamalarda önce gönderilen birtakım işlemlerin iptal →
25348     edilmesi gibi →
25349     gerekçelerle kullanılabilmektedir.
```

25345 OOB verisini gönderebilmek için send fonksiyonunun flags parametresine ↵
MSG_OOB girmek gereklidir. Tabii TCP yalnızca 1 byte uzunlığında
25346 OOB verisinin gönderilmesine izin vermektedir. Bu durumda eğer send ile ↵
birden fazla byte MSG_OOB bayrağı ile gönderilmek istenirse ↵
gönderilenlerin
25347 yalnızca son byte'ı OOB olarak gönderilir.

25348

25349 Normal olarak OOB verisi recv fonksiyonunda MSG_OOB bayrağı ile alınır. ↵
Ancak bu bayrak kullanılarak recv çağrılarında eğer bir OOB verisi
25350 sırada yoksa recv başarısız olmaktadır. recv fonksiyonun MSG_OOB ↵
bayraklı çağrısında başarılı olabilmesi için o anda bir OOB verisinin ↵
gelmiş olması gereklidir.

25351 Pekiyi OOB verisinin geldiğini nasıl anlarız? İşte tipik yöntem SIGURG ↵
sinyaliin kullanılmasıdır. Çünkü sokete bir OOB verisi geldiğinde ↵
işletim sistemi

25352 SIGURG sinyali oluşturabilmektedir. Bu sinyalin default durumu IGNORE ↵
biçimindedir. (Yani eğer set edilmemişse sanki sinyal olmuşmamış gibi ↵
davranılır.)

25353 Fakat default olarak OOB verisi geldiğinde sinyal oluşmamaktadır. Bunu ↵
mungkin hale getirmek için soket üzerinde fcntl fonksiyonu ile F_SETOWN ↵
komut kodunu

25354 kullanarak set işlemi yapmak gereklidir. fcntl fonksiyonun son parametresi ↵
bu durumda sinyalin gönderileceği prosesin id değeri olarak ↵
girilmelidir.

25355 Eğer bu parametre negatif bir proses grup id'si oalrak girilirse bu ↵
durumda işletim sistemi tüm proses grubuna sinyal göndermektedir. ↵
Örneğin:

25356

```
25357 if (fcntl(sock_client, F_SETOWN, getpid()) == -1)  
25358     exit_sys("fcntl");
```

25359

25360 Aşağıdaki server programda bir OOB verisi geldiğinde SIGURG sinyali ↵
oluşturulmaktadır. Bu sinyalin içerisinde recv fonksiyonu MSG_OOB ↵
bayrağı

25361 ile çağrılmıştır. OOB verisinin okunması için MSG_OOB bayrağı gereklidir, ↵
ancak OOB verisinin olmadığı bir duurmda bu bayrak kullanılırsa recv ↵
başarısız olur.

25362 O halde SIGURG sinyali geldiğinde recv MSG_OOB ile çağrılmalıdır. Bu ↵
durumda TCP'de her zaman yalnızca 1 byte okunabilemektedir. Ayrıca ↵
server programda

25363 SIGURG sinyali set edilirken sigaction yapısının flags parametresinin ↵
SA_RESTART biçimind egeçildiğine dikkat ediniz. Bu recv üzerinde ↵
beklerken olusabilecek

25364 SIGURG sinyalinden sonra recv'in otomatik yeniden başlatılması için ↵
kullanılmıştır.

25365

25366 Aşağıdaki client programda başı "u" harfi ile başlayan yazılar MSG_OOB ↵
bayrağı kullanılarak gönderilmiştir. Bu durumda yalnızca son byte'ın ↵
OOB verisi

25367 olacağını anımsayınız. OOB verileri her zaman mümkünse diğer verilerden ↵
önce ele alınmaktadır.

25368

25369 ----- ↵

```
-----*/
25370
25371 /* oobserver.c */
25372
25373 #include <stdio.h>
25374 #include <stdlib.h>
25375 #include <string.h>
25376 #include <fcntl.h>
25377 #include <unistd.h>
25378 #include <signal.h>
25379 #include <sys/socket.h>
25380 #include <netinet/in.h>
25381 #include <arpa/inet.h>
25382
25383 #define BUFFER_SIZE      1024
25384
25385 void sigurg_handler(int sno);
25386 void exit_sys(const char *msg);
25387
25388 int sock_client;
25389
25390 int main(int argc, char *argv[])
25391 {
25392     int sock;
25393     struct sockaddr_in sinaddr, sinaddr_client;
25394     socklen_t sinaddr_len;
25395     char ntopbuf[INET_ADDRSTRLEN];
25396     in_port_t port;
25397     ssize_t result;
25398     char buf[BUFFER_SIZE + 1];
25399     struct sigaction sa;
25400
25401     if (argc != 2) {
25402         fprintf(stderr, "wrong number of arguments!..\n");
25403         exit(EXIT_FAILURE);
25404     }
25405
25406     sa.sa_handler = sigurg_handler;
25407     sa.sa_flags = SA_RESTART;
25408     sigemptyset(&sa.sa_mask);
25409
25410     if (sigaction(SIGURG, &sa, NULL) == -1)
25411         exit_sys("sigaction");
25412
25413     port = (in_port_t)strtoul(argv[1], NULL, 10);
25414
25415     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
25416         exit_sys("socket");
25417
25418     sinaddr.sin_family = AF_INET;
25419     sinaddr.sin_port = htons(port);
25420     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
25421
```

```
25422     if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
25423         exit_sys("bind");
25424
25425     if (listen(sock, 8) == -1)
25426         exit_sys("listen");
25427
25428     printf("Waiting for connection...\n");
25429
25430     sinaddr_len = sizeof(sinaddr_client);
25431     if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,
25432                               &sinaddr_len)) == -1)
25433         exit_sys("accept");
25434
25435     if (fcntl(sock_client, F_SETOWN, getpid()) == -1)
25436         exit_sys("fcntl");
25437
25438     printf("Connected: %s : %u\n", inet_ntop(AF_INET, &sinaddr_client,
25439                                               ntopbuf, INET_ADDRSTRLEN), (unsigned)ntohs(sinaddr_client.sin_port)); ↵
25440
25441     for (;;) {
25442         if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
25443             exit_sys("recv");
25444         if (result == 0)
25445             break;
25446         buf[result] = '\0';
25447         if (!strcmp(buf, "quit"))
25448             break;
25449         printf("%ld bytes received from %s (%u): %s\n", (long)result,
25450               ntopbuf, (unsigned)ntohs(sinaddr_client.sin_port), buf);
25451     }
25452
25453     shutdown(sock_client, SHUT_RDWR);
25454     close(sock_client);
25455     close(sock);
25456
25457     return 0;
25458 }
25459
25460 void sigurg_handler(int sno)
25461 {
25462     char oob;
25463
25464     if (recv(sock_client, &oob, 1, MSG_OOB) == -1)
25465         exit_sys("recv");
25466
25467     printf("OOB Data received: %c\n", oob);
25468 }
25469
25470 void exit_sys(const char *msg)
25471 {
25472     perror(msg);
```

```
25471     exit(EXIT_FAILURE);
25472 }
25473
25474 /* oobclient.c */
25475
25476 #include <stdio.h>
25477 #include <stdlib.h>
25478 #include <string.h>
25479 #include <unistd.h>
25480 #include <sys/socket.h>
25481 #include <netinet/in.h>
25482 #include <arpa/inet.h>
25483 #include <netdb.h>
25484
25485 #define BUFFER_SIZE      1024
25486
25487 void exit_sys(const char *msg);
25488
25489 int main(int argc, char *argv[])
25490 {
25491     int sock;
25492     struct addrinfo *ai, *ri;
25493     struct addrinfo hints = {0};
25494     char buf[BUFFER_SIZE];
25495     char *str;
25496     int result;
25497
25498     if (argc != 3) {
25499         fprintf(stderr, "wrong number of arguments!..\n");
25500         exit(EXIT_FAILURE);
25501     }
25502
25503     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
25504         exit_sys("socket");
25505
25506     hints.ai_family = AF_INET;
25507     hints.ai_socktype = SOCK_STREAM;
25508
25509     if ((result = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {
25510         fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
25511         exit(EXIT_FAILURE);
25512     }
25513
25514     for (ri = ai; ri != NULL; ri = ri->ai_next)
25515         if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)
25516             break;
25517
25518     if (ri == NULL)
25519         exit_sys("connect");
25520
25521     freeaddrinfo(ai);
25522
25523     printf("Connected...\n");
```

```
25524
25525     for (;;) {
25526         printf("Yazi giriniz:");
25527         fgets(buf, BUFFER_SIZE, stdin);
25528         if ((str = strchr(buf, '\n')) != NULL)
25529             *str = '\0';
25530         if (!strcmp(buf, "quit"))
25531             break;
25532
25533         if ((send(sock, buf, strlen(buf), buf[0] == 'u' ? MSG_OOB : 0)) == -1)
25534             exit_sys("send");
25535     }
25536
25537     shutdown(sock, SHUT_RDWR);
25538     close(sock);
25539
25540     return 0;
25541 }
25542
25543 void exit_sys(const char *msg)
25544 {
25545     perror(msg);
25546
25547     exit(EXIT_FAILURE);
25548 }
25549
25550 /
*-----*
-----+
25551     OOB verisi geldiğinde select fonksiyonu "sokette bir işlem oluştu"
25552     biçiminde geri dönmektedir. Ancak oluşan işlem "exceptional"
25553     kabul edilmektedir. Yani select fonksiyonunda dördüncü parametredeki
25554     bayraklara set işlemi yapmamız gereklidir. Benzer biçimde OOB geldiğinde
25555     poll fonksiyonu da POLLPRI bayrağını set etmektedir.
25556 -----*/
25557
25558 /* oobserver.c */
25559
25560 #include <stdio.h>
25561 #include <stdlib.h>
25562 #include <string.h>
25563 #include <fcntl.h>
25564 #include <unistd.h>
25565 #include <sys/socket.h>
25566 #include <netinet/in.h>
25567 #include <arpa/inet.h>
25568 #include <sys/select.h>
25569
25570 #define BUFFER_SIZE      1024
```

```
25571
25572 void exit_sys(const char *msg);
25573
25574 int main(int argc, char *argv[])
25575 {
25576     int sock, sock_client;
25577     struct sockaddr_in sinaddr, sinaddr_client;
25578     socklen_t sinaddr_len;
25579     char ntopbuf[INET_ADDRSTRLEN];
25580     in_port_t port;
25581     ssize_t result;
25582     char buf[BUFFER_SIZE + 1];
25583     fd_set rset, trset, eset, teset;
25584     char oob;
25585
25586     if (argc != 2) {
25587         fprintf(stderr, "wrong number of arguments!..\\n");
25588         exit(EXIT_FAILURE);
25589     }
25590
25591     port = (in_port_t)strtoul(argv[1], NULL, 10);
25592
25593     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
25594         exit_sys("socket");
25595
25596     sinaddr.sin_family = AF_INET;
25597     sinaddr.sin_port = htons(port);
25598     sinaddr.sin_addr.s_addr = htonl(INADDR_ANY);
25599
25600     if (bind(sock, (struct sockaddr *)&sinaddr, sizeof(sinaddr)) == -1)
25601         exit_sys("bind");
25602
25603     if (listen(sock, 8) == -1)
25604         exit_sys("listen");
25605
25606     printf("Waiting for connection...\\n");
25607
25608     sinaddr_len = sizeof(sinaddr_client);
25609     if ((sock_client = accept(sock, (struct sockaddr *)&sinaddr_client,
25610         &sinaddr_len)) == -1)
25611         exit_sys("accept");
25612
25613     printf("Connected: %s : %u\\n", inet_ntop(AF_INET, &sinaddr_client,
25614         ntopbuf, INET_ADDRSTRLEN), (unsigned)ntohs(sinaddr_client.sin_port)); ↵
25615
25616     FD_ZERO(&eset);
25617     FD_ZERO(&teset);
25618
25619     FD_SET(sock_client, &rset);
25620     FD_SET(sock_client, &eset);
25621
25622     for (;;) {
```

```
25621     trset = rset;
25622     teset = eset;
25623     if (select(sock_client + 1, &trset, NULL, &teset, NULL) == -1)
25624         exit_sys("select");
25625
25626     if (FD_ISSET(sock_client, &trset)) {
25627         if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
25628             exit_sys("recv");
25629         if (result == 0)
25630             break;
25631         buf[result] = '\0';
25632         if (!strcmp(buf, "quit"))
25633             break;
25634         printf("%ld bytes received from %s (%u): %s\n", (long)result,
25635               ntohs(sinaddr_client.sin_port), buf);
25636     }
25637     if (FD_ISSET(sock_client, &teset)) {
25638         if ((result = recv(sock_client, &oob, 1, MSG_OOB)) == -1)
25639             exit_sys("recv");
25640         printf("OOB Data received: %c\n", oob);
25641     }
25642
25643     shutdown(sock_client, SHUT_RDWR);
25644     close(sock_client);
25645     close(sock);
25646
25647     return 0;
25648 }
25649
25650 void exit_sys(const char *msg)
25651 {
25652     perror(msg);
25653
25654     exit(EXIT_FAILURE);
25655 }
25656
25657 /* oobclient.c */
25658
25659 #include <stdio.h>
25660 #include <stdlib.h>
25661 #include <string.h>
25662 #include <unistd.h>
25663 #include <sys/socket.h>
25664 #include <netinet/in.h>
25665 #include <arpa/inet.h>
25666 #include <netdb.h>
25667
25668 #define BUFFER_SIZE      1024
25669
25670 void exit_sys(const char *msg);
25671
25672 int main(int argc, char *argv[])
```

```
25673 {  
25674     int sock;  
25675     struct addrinfo *ai, *ri;  
25676     struct addrinfo hints = {0};  
25677     char buf[BUFFER_SIZE];  
25678     char *str;  
25679     int result;  
25680  
25681     if (argc != 3) {  
25682         fprintf(stderr, "wrong number of arguments!..\n");  
25683         exit(EXIT_FAILURE);  
25684     }  
25685  
25686     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)  
25687         exit_sys("socket");  
25688  
25689     hints.ai_family = AF_INET;  
25690     hints.ai_socktype = SOCK_STREAM;  
25691  
25692     if ((result = getaddrinfo(argv[1], argv[2], &hints, &ai)) != 0) {  
25693         fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));  
25694         exit(EXIT_FAILURE);  
25695     }  
25696  
25697     for (ri = ai; ri != NULL; ri = ri->ai_next)  
25698         if (connect(sock, ri->ai_addr, ri->ai_addrlen) != -1)  
25699             break;  
25700  
25701     if (ri == NULL)  
25702         exit_sys("connect");  
25703  
25704     freeaddrinfo(ai);  
25705  
25706     printf("Connected...\n");  
25707  
25708     for (;;) {  
25709         printf("Yazi giriniz:");  
25710         fgets(buf, BUFFER_SIZE, stdin);  
25711         if ((str = strchr(buf, '\n')) != NULL)  
25712             *str = '\0';  
25713         if (!strcmp(buf, "quit"))  
25714             break;  
25715  
25716         if ((send(sock, buf, strlen(buf), buf[0] == 'u' ? MSG_OOB : 0)) == -1)  
25717             exit_sys("send");  
25718     }  
25719  
25720     shutdown(sock, SHUT_RDWR);  
25721     close(sock);  
25722  
25723     return 0;  
25724 }
```

```
25725
25726 void exit_sys(const char *msg)
25727 {
25728     perror(msg);
25729
25730     exit(EXIT_FAILURE);
25731 }
25732
25733 /
*-----*
-----*
25734     UNIX Domain soketler aynı makinenin prosesleri arasında haberleşmeler
25735         için kullanılabilmektedir. Bu bakımından bu soketler
25736         IP ailesini kullanan soketlerden farklıdır. Unix domain soket
25737         yaratabilmek için socket fonksiyonun birinci parametresi
25738         AF_UNIX geçilmelidir. Unix domain soketlerin TCP/IP ya da UDP/IP
25739         soketlerle bir ilgisi yoktur. Bu soketler UNIX/Linux sistemlerinde
25740         oldukça etkin bir biçimde gerçekleştirilmektedir. Dolayısıyla aynı
25741         makinenin prosesleri arasında haberleşmede borulara, mesaj
25742         kuyruklarına,
25743         paylaşılan bellek alanlarına bir seçenek olarak kullanılabilmektedir.
25744         Hatta bazı UNIX türevi sistemlerde (ama Linux'ta böyle değil) aslında
25745         çekirdek tarafından önce bu protokol gerçekleştiriliip daha sonra boru
25746         mekanizması bu protokol kullanılarak gerçekleştirilmektedir. Böylece
25747         örneğin aynı makinedeki
25748         iki prosesin haberleşmesi için UNIX domain soketler TCP/IP ve UDP/IP
25749         soketlerine göre çok daha hızlı çalışmaktadır. Aynı makine üzerinde
25750         çok client'lı uygulamalar için
25751         UNIX domain soketler boru haberleşmesine göre organizasyonel avantaj
25752         bakımından tercih edilebilmektedir.
25753
25754     UNIX domain soketlerin kullanımı en çok boru kullanımına benzemektedir.
25755         Ancak Unix domain soketlerin borulara olan bir üstünlüğü "full duplex"
25756         haberleşme sunmasıdır.
25757     Bilindiği gibi borular "half duplex" bir haberleşme sunmaktadır. Ancak
25758         genel olarak boru haberleşmeleri UNIX domain soket haberleşmelerine göre
25759         daha hızlı olma eğilimindedir.
25760
25761     UNIX domain soketler kullanım olarak daha önce görmüş olduğumuz TCP/IP
25762         ve UDP/IP protokollerine çok benzemektedir. Yani işlemler
25763         sanki TCP/IP ya da UDP/IP client server program yazılıyormuş gibi
25764         yapılır. Başka bir deyişle UNIX domain soketlerinde client ve
25765         server programlarının genel yazım adımları TCP/IP ve UDP/IP ile aynıdır.
25766
25767     UNIX domain soketlerde client'in server'a bağlanması için gereken adres
25768         bir dosya ismi yani yol ifadesi biçimindedir. Kullanılacak yapı
25769         sockaddr_in değil sockaddr_un yapısıdır. Bu yapı en azından şu
25770         elemanlara sahip olmak zorundadır:
25771
25772     struct sockaddr_un {
25773         sa_family_t sun_family;
25774         char sun_path[108];
25775     };
25776 
```

25757
25758 Yapının sun_family elemanı AF_UNIX biçiminde girilmeli, sun_path ↗
 elemanı da soketi temsil eden dosyanın yol ifadesi biçiminde ↗
 girilmelidir. Burada yol ifadesi
25759 verilen dosya bind işlemi tarafından yaratılmaktadır. Yaratılan bu ↗
 dosyanın türü "(s)ocket" biçiminde görüntülenmektedir. Eğer bu dosya ↗
 zaten varsa bind fonksiyonu
25760 başarısız olur. Dolayısıyla bu dosyanın varsa silinmesi gerekmektedir. ↗
 0 halde client ve server programların işin başında bir isim altında ↗
 anlaşmaları gerekmektedir. sockaddr_un yapısı kullanılmadan önce ↗
 sıfırlanmalıdır.
25761 bind tarafından yaratılan bu soket dosyaları normal bir dosya ↗
 değildir. Yani open fonksiyonuyla açılamamaktadır.
25762
25763 Aşağıda stream tabanlı örnek bir UNIX domain uygulaması verilmiştir. Bu ↗
 örnekte client yalnızca bilgi göndermeye server ise bilgiyi alıp ↗
 yazdırmaktadır.
25764
25765 -----*/
25766
25767 /* uds-server.c */
25768
25769 /* server.c */
25770
25771 #include <stdio.h>
25772 #include <stdlib.h>
25773 #include <string.h>
25774 #include <errno.h>
25775 #include <unistd.h>
25776 #include <sys/socket.h>
25777 #include <sys/un.h>
25778
25779 #define BUFFER_SIZE 1024
25780
25781 char *revstr(char *str);
25782 void exit_sys(const char *msg);
25783
25784 int main(int argc, char *argv[])
25785 {
25786 int sock, sock_client;
25787 socklen_t sunaddr_len;
25788 struct sockaddr_un sunaddr, sunaddr_client;
25789 ssize_t result;
25790 char buf[BUFFER_SIZE + 1];
25791
25792 if (argc != 2) {
25793 fprintf(stderr, "wrong number of arguments!..\n");
25794 exit(EXIT_FAILURE);
25795 }
25796
25797 if ((sock = socket(AF_UNIX, SOCK_STREAM, 0)) == -1)
25798 exit_sys("socket");

```
25799
25800     memset(&sunaddr, 0, sizeof(sunaddr));
25801     sunaddr.sun_family = AF_UNIX;
25802     strcpy(sunaddr.sun_path, argv[1]);
25803
25804     if (remove(argv[1]) == -1 && errno != ENOENT)
25805         exit_sys("remove");
25806
25807     if (bind(sock, (struct sockaddr *)&sunaddr, sizeof(sunaddr)) == -1)
25808         exit_sys("bind");
25809
25810     if (listen(sock, 8) == -1)
25811         exit_sys("listen");
25812
25813     printf("Waiting for connection...\n");
25814
25815     sunaddr_len = sizeof(sunaddr_client);
25816     if ((sock_client = accept(sock, (struct sockaddr *)&sunaddr_client,    ↴
25817         &sunaddr_len)) == -1)
25818         exit_sys("accept");
25819
25820     printf("Connected: %s\n", sunaddr_client.sun_path);
25821
25822     for(;;) {
25823         if ((result = recv(sock_client, buf, BUFFER_SIZE, 0)) == -1)
25824             exit_sys("recv");
25825         if (result == 0)
25826             break;
25827         buf[result] = '\0';
25828         if (!strcmp(buf, "quit"))
25829             break;
25830         printf("%ld bytes received: %s\n", (long)result, buf);
25831
25832         revstr(buf);
25833         if (send(sock_client, buf, strlen(buf), 0) == -1)
25834             exit_sys("send");
25835     }
25836
25837     close(sock_client);
25838     close(sock);
25839
25840     return 0;
25841 }
25842 char *revstr(char *str)
25843 {
25844     size_t i, k;
25845     char temp;
25846
25847     for (i = 0; str[i] != '\0'; ++i)
25848         ;
25849
25850     for (--i, k = 0; k < i; ++k, --i) {
```

```
25851         temp = str[k];
25852         str[k] = str[i];
25853         str[i] = temp;
25854     }
25855
25856     return str;
25857 }
25858
25859 void exit_sys(const char *msg)
25860 {
25861     perror(msg);
25862
25863     exit(EXIT_FAILURE);
25864 }
25865
25866 /* uds-client.c */
25867
25868 #include <stdio.h>
25869 #include <stdlib.h>
25870 #include <string.h>
25871 #include <unistd.h>
25872 #include <sys/socket.h>
25873 #include <sys/un.h>
25874
25875 #define BUFFER_SIZE      1024
25876
25877 void exit_sys(const char *msg);
25878
25879 int main(int argc, char *argv[])
25880 {
25881     int sock;
25882     struct sockaddr_un sunaddr;
25883     char buf[BUFFER_SIZE];
25884     char *str;
25885     ssize_t result;
25886     int sresult;
25887
25888     if (argc != 2) {
25889         fprintf(stderr, "wrong number of arguments!..\n");
25890         exit(EXIT_FAILURE);
25891     }
25892
25893     if ((sock = socket(AF_UNIX, SOCK_STREAM, 0)) == -1)
25894         exit_sys("socket");
25895
25896     memset(&sunaddr, 0, sizeof(sunaddr));
25897     sunaddr.sun_family = AF_UNIX;
25898     strcpy(sunaddr.sun_path, argv[1]);
25899
25900     if (connect(sock, (struct sockaddr *)&sunaddr, sizeof(sunaddr)) == -1)
25901         exit_sys("socket");
25902
25903     for (;;) {
```

```
25904     printf("Yazi giriniz:");
25905     fgets(buf, BUFFER_SIZE, stdin);
25906     if ((str = strchr(buf, '\n')) != NULL)
25907         *str = '\0';
25908     if ((send(sock, buf, strlen(buf), 0)) == -1)
25909         exit_sys("send");
25910     if (!strcmp(buf, "quit"))
25911         break;
25912
25913     if ((result = recv(sock, buf, BUFFER_SIZE, 0)) == -1)
25914         exit_sys("recv");
25915     if (result == 0)
25916         break;
25917     buf[result] = '\0';
25918     printf("%ld bytes received: %s\n", (long)result, buf);
25919 }
25920
25921     close(sock);
25922
25923     return 0;
25924 }
25925
25926 void exit_sys(const char *msg)
25927 {
25928     perror(msg);
25929
25930     exit(EXIT_FAILURE);
25931 }
25932
25933 /
*-----*
-----*
25934 Stream tabanlı UNIX domain soketlerde server accept uyguladığında      ↗
    client'a ilişkin sockaddr_un yapısında ne almaktadır?      ↗
25935 Aslında bu protokolde bir port kavramı olmadığına göre server'ın      ↗
    bağlantından bir bilgi elde etmeyecektir. Fakat yine de      ↗
25936 client program da bind uygulayıp ondan sonra sokete bağlanabilir. Bu      ↗
    durumda server client bağlantısından sonra sockaddr_un      ↗
    yapısından client'ın bind ettiği soket dosyanın yol ifadesini elde eder. ↗
25937
25938
25939     Multi client UNIX domain soket uygulamalarında tamamen TCP/IP'de yapmış ↗
        olduğumuz server modellerinin aynıları kullanılabilmektedir.
25940 -----*/      ↗
25941
25942 /
*-----*
-----*
25943     UNIX domain soketlerde datagram haberleşme de yapılabilir. Bu haberleşme ↗
        mesaj kuyruklarına bir seçenektedir. UNIX domain      ↗
25944 soket datagram haberleşmede gönderilen datagram'ların aynı sırada      ↗
        alınması garanti edilmiştir. Yani gönderim UDP/IP'de olduğu gibi      ↗
```

```
25945     güvensiz değil güvenlidir.  
25946  
25947     Aşağıda UNIX domain soketler kullanılarak bir datagram haberleşme örneği →  
         verilmektedir. Burada server hiç bağlantı sağlamadan herhangi bir  
25948     client'tan paketi alır, oradaki yazıyı ters çevirip ona geri gönderir. →  
         Hem client hem de server ayrı ayrı iki dosya ismi ile bind işlemi →  
         paymaktadır.  
25949     Server program komut satırı argümanı olarak kendi bind edeceği soket →  
         dosyasının yol ifadesini, client program ise hem kendi bind edeceği →  
         soket dosyasının →  
25950     yol ifadesini hem de server soketin yol ifadesini almaktadır.  
25951 -----*/  
25952  
25953 /* uds-dg-server.c */  
25954  
25955 #include <stdio.h>  
25956 #include <stdlib.h>  
25957 #include <string.h>  
25958 #include <errno.h>  
25959 #include <unistd.h>  
25960 #include <sys/socket.h>  
25961 #include <sys/un.h>  
25962  
25963 #define BUFFER_SIZE      4096  
25964  
25965 char *revstr(char *str);  
25966 void exit_sys(const char *msg);  
25967  
25968 int main(int argc, char *argv[])  
25969 {  
25970     int sock;  
25971     struct sockaddr_un sunaddr, sunaddr_client;  
25972     socklen_t sunaddr_len;  
25973     ssize_t result;  
25974     char buf[BUFFER_SIZE + 1];  
25975  
25976     if (argc != 2) {  
25977         fprintf(stderr, "wrong number of arguments!..\n");  
25978         exit(EXIT_FAILURE);  
25979     }  
25980  
25981     if ((sock = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1)  
25982         exit_sys("socket");  
25983  
25984     memset(&sunaddr, 0, sizeof(sunaddr));  
25985     sunaddr.sun_family = AF_UNIX;  
25986     strcpy(sunaddr.sun_path, argv[1]);  
25987  
25988     if (remove(argv[1]) == -1 && errno != ENOENT)  
25989         exit_sys("remove");  
25990  
25991     if (bind(sock, (struct sockaddr *)&sunaddr, sizeof(sunaddr)) == -1)
```

```
25992     exit_sys("bind");
25993
25994     printf("Waiting for client data...\n");
25995
25996     for (;;) {
25997         sunaddr_len = sizeof(sunaddr_client);
25998         if ((result = recvfrom(sock, buf, BUFFER_SIZE, 0, (struct sockaddr *) &sunaddr_client, &sunaddr_len)) == -1)
25999             exit_sys("recvfrom");
26000
26001         buf[result] = '\0';
26002         printf("%ld bytes received from %s: %s\n", (long)result,
26003                sunaddr_client.sun_path, buf);
26004
26005         revstr(buf);
26006         if (sendto(sock, buf, strlen(buf), 0, (struct sockaddr *) &sunaddr_client, sizeof(struct sockaddr)) == -1)
26007             exit_sys("sendto");
26008     }
26009     close(sock);
26010
26011     return 0;
26012 }
26013
26014 char *revstr(char *str)
26015 {
26016     size_t i, k;
26017     char temp;
26018
26019     for (i = 0; str[i] != '\0'; ++i)
26020         ;
26021
26022     for (--i, k = 0; k < i; ++k, --i) {
26023         temp = str[k];
26024         str[k] = str[i];
26025         str[i] = temp;
26026     }
26027
26028     return str;
26029 }
26030
26031 void exit_sys(const char *msg)
26032 {
26033     perror(msg);
26034
26035     exit(EXIT_FAILURE);
26036 }
26037
26038 /* uds-dg-client.c */
26039
26040 #include <stdio.h>
26041 #include <stdlib.h>
```

```
26042 #include <string.h>
26043 #include <errno.h>
26044 #include <unistd.h>
26045 #include <sys/socket.h>
26046 #include <sys/un.h>
26047
26048 #define BUFFER_SIZE      4096
26049
26050 void exit_sys(const char *msg);
26051
26052 int main(int argc, char *argv[])
26053 {
26054     int sock;
26055     struct sockaddr_un sunaddr, sunaddr_server;
26056     socklen_t sunaddr_len;
26057     char buf[BUFFER_SIZE];
26058     char *str;
26059     ssize_t result;
26060
26061     if (argc != 3) {
26062         fprintf(stderr, "wrong number of arguments!..\n");
26063         exit(EXIT_FAILURE);
26064     }
26065
26066     if ((sock = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1)
26067         exit_sys("socket");
26068
26069     memset(&sunaddr, 0, sizeof(sunaddr));
26070     sunaddr.sun_family = AF_UNIX;
26071     strcpy(sunaddr.sun_path, argv[1]);
26072
26073     if (remove(argv[1]) == -1 && errno != ENOENT)
26074         exit_sys("remove");
26075
26076     if (bind(sock, (struct sockaddr *)&sunaddr, sizeof(sunaddr)) == -1)
26077         exit_sys("bind");
26078
26079     memset(&sunaddr_server, 0, sizeof(sunaddr));
26080     sunaddr_server.sun_family = AF_UNIX;
26081     strcpy(sunaddr_server.sun_path, argv[2]);
26082
26083     for (;;) {
26084         printf("Yazi giriniz:");
26085         fgets(buf, BUFFER_SIZE, stdin);
26086         if ((str = strchr(buf, '\n')) != NULL)
26087             *str = '\0';
26088         if (!strcmp(buf, "quit"))
26089             break;
26090
26091         if (sendto(sock, buf, strlen(buf), 0, (struct sockaddr *)
26092             &sunaddr_server, sizeof(sunaddr_server)) == -1) ↗
26092             exit_sys("sendto");
26093
```

```
26094     sunaddr_len = sizeof(sunaddr_server);
26095     if ((result = recvfrom(sock, buf, BUFFER_SIZE, 0, (struct sockaddr * )&sunaddr_server, &sunaddr_len)) == -1)
26096         exit_sys("recvfrom");
26097
26098     buf[result] = '\0';
26099     printf("%ld bytes received from %s: %s\n", (long)result,
26100           sunaddr_server.sun_path, buf);
26101 }
26102 close(sock);
26103
26104 return 0;
26105 }
26106
26107 void exit_sys(const char *msg)
26108 {
26109     perror(msg);
26110
26111     exit(EXIT_FAILURE);
26112 }
26113
26114 /
*-----*
```

26115 UNIX domain soketler "isimsiz boru haberleşmesine" benzer biçimde de
26116 kullanılabilmektedir. Anımsanacağı gibi isimsiz boru hanerleşmesinde
26117 yalnızca üst alt prosesler arası haberleşme yapılmaktaydı. pipe
fonksiyonu bize iki betimleyici veriyordu. Biz de
26118 fork işlemi ile bu betimleyicileri alt prosese geçiriyorduk. Burada da
benzer bir model uygulanmaktadır.

26119 İsimsiz (unbound) soket yaratımı socketpair isimli fonksiyonuya
26120 yapılmaktadır. Fonksiyonun prototipi şöyledir:

```
26121 int socketpair(int domain, int type, int protocol, int sv[2]);
```

26122 Fonksiyonun birinci parametresi protokol ailesinin ismini alır. Her ne
26123 kadar fonksiyon genel olsa da pek çok işletim sistemi
26124 bu fonksiyonu yalnızca UNIX domain soketler için gerçekleştirmektedir.
(Gerçekten de üst ve alt prosesler arasında UNIX domain soketler
26125 varken örneğin TCP/IP soketleriyle haberleşmenin zarardan başka bir
faydası olmayacağı.) Fonksiyonun ikinci parametresi kullanılacak
26126 soketin türünü belirtir. Bu parametre yine SOCK_STREAM ya da SOCK_DGRAM
biiminde girilmelidir. Üçüncü parametre kulahılıacak transport
26127 katmanını belirtmektedir. Bu parametre 0 olarak geçilebilir. Son
parametre bir çift soket betimleyicisinin yerleştirileceği iki
elemanlı

26128 int türden dizinin başlangıç adresini almaktadır. Fonksiyon başarı
durumunda 0 değerine, başarısızlık durumunda -1 değerine geri
26129 dönmektedir.

26130 socketpair fonksiyonu SOCK_STREAM soketler için zaten bağlantı sağlanmış

iki soketi bize vermektedir. Yani bu fonksiyon çağrıldıktan sonra listen, accept, connect gibi fonksiyonların çağrılması gereksizdir.

Pekiyi isimsiz borularla socketpair fonksiyonuyla oluşturulan isimsiz UNIX domain soketler arasında ne fark vardır?

Aslında bu iki kullanım benzer etkilere sahiptir. Ancak en önemli farklılık UNIX domain soketlerin çift yönlü (full duplex) bir haberleşme sağlamasıdır. Ayrıca isimsiz mesaj kuyrukları olmadığına dikkat ediniz. Halbuki isimsiz UNIX domain soketler sanka isimsiz mesaj kuyrukları gibi de kullanılabilmektedir.

Aşağıdaki programda tipki isimsiz boru haberleşmesinde olduğu gibi üst ve alt prosesler birbirileri arasında isimsiz UNIX domain soketler yoluyla haberleşmekte dir. Buradaki soketlerin çift yönlü haberleşmeye olanak verdiği anımsayınız.

```
-----*/  
26141  
26142 /* uds-socketpair.c */  
26143  
26144 #include <stdio.h>  
26145 #include <stdlib.h>  
26146 #include <string.h>  
26147 #include <unistd.h>  
26148 #include <sys/wait.h>  
26149 #include <sys/socket.h>  
26150  
26151 #define BUFFER_SIZE 1024  
26152  
26153 char *revstr(char *str);  
26154 void exit_sys(const char *msg);  
26155  
26156 int main(void)  
26157 {  
26158     int socks[2];  
26159     char buf[BUFFER_SIZE + 1];  
26160     char *str;  
26161     ssize_t result;  
26162     pid_t pid;  
26163  
26164     if (socketpair(AF_UNIX, SOCK_STREAM, 0, socks) == -1)  
26165         exit_sys("socketpair");  
26166  
26167     if ((pid = fork()) == -1)  
26168         exit_sys("fork");  
26169  
26170     if (pid != 0) { /* parent */  
26171         close(socks[1]);  
26172  
26173         for (;;) {  
26174             if ((result = recv(socks[0], buf, BUFFER_SIZE, 0)) == -1)  
26175                 exit_sys("recv");  
26176             if (result == 0)
```

```
26177         break;
26178         buf[result] = '\0';
26179         if (!strcmp(buf, "quit"))
26180             break;
26181         revstr(buf);
26182         if (send(socks[0], buf, strlen(buf), 0) == -1)
26183             exit_sys("send");
26184     }
26185
26186     if (waitpid(pid, NULL, 0) == -1)
26187         exit_sys("waitpid");
26188
26189     close(socks[0]);
26190
26191     exit(EXIT_SUCCESS);
26192 }
26193 else { /* child */
26194     close(socks[0]);
26195     for (;;) {
26196         printf("Yazi giriniz:");
26197         fgets(buf, BUFFER_SIZE, stdin);
26198         if ((str = strchr(buf, '\n')) != NULL)
26199             *str = '\0';
26200         if ((send(socks[1], buf, strlen(buf), 0)) == -1)
26201             exit_sys("send");
26202         if (!strcmp(buf, "quit"))
26203             break;
26204         if ((result = recv(socks[1], buf, BUFFER_SIZE, 0)) == -1)
26205             exit_sys("recv");
26206         if (result == 0)
26207             break;
26208         buf[result] = '\0';
26209         printf("%ld bytes received: %s\n", (long)result, buf);
26210     }
26211     close(socks[1]);
26212     exit(EXIT_SUCCESS);
26213 }
26214
26215     return 0;
26216 }
26217
26218 char *revstr(char *str)
26219 {
26220     size_t i, k;
26221     char temp;
26222
26223     for (i = 0; str[i] != '\0'; ++i)
26224         ;
26225
26226     for (--i, k = 0; k < i; ++k, --i) {
26227         temp = str[k];
26228         str[k] = str[i];
26229         str[i] = temp;
```

```
26230     }
26231
26232     return str;
26233 }
26234
26235 void exit_sys(const char *msg)
26236 {
26237     perror(msg);
26238
26239     exit(EXIT_FAILURE);
26240 }
26241
26242 /
*-----  
-----  
26243     TCP Protokolünün Seviyeli Çalışma Biçimi
26244
26245     TCP protokolü RFC 793 dokümanlarında (https://tools.ietf.org/html/rfc793) tanımlanmıştır. Sonradan protokole bazı revizyonlar ve eklemeler yapılmıştır.
26246
26247     Paket tabanlı protokollerin hepsinde gönderilip alınan veriler paket biçimindedir (yani bir grup byte biçimindedir). Bu paketlerin "başlık (header) ve veri (data)" kismı vardır. Örneğin Ethernet paketinin başlık ve data kısmı, IP paketinin başlık ve data kısmı TCP paketinin başlık ve data kısmı bulunmaktadır. Öte yandan TCP protokolü aslında IP protokolünün üzerine konumlandırılmıştır. Yani aslında
26248
26249     TCP Paketleri IP paketleri gibi gönderilip alınmaktadır. İşin gerçeği bu paketler bilgisayarımıza Ethernet paketi olarak gelmektedir.
26250
26251     Paketlerin başlık kısımlarında önemli "meta data" bilgileri bulunmaktadır. O halde aslında bizim network kartımıza bilgiler sanki ethernet paketiymiş gibi gelmektedir.
26252
26253     O zaman network kartımıza gelen paketin bir ethernet başlığı ve bir de ethernet data kısmı olmalıdır. İşte aslında IP paketi de Ethernet paketinin data kısmında
26254
26255     konuşlandırılır. TCP paketi de aslında IP paketinin data kısmında konuşlandırmaktadır. O halde aslında bize gelen ethernet paketinin data kısmında ip paketi,
26256
26257     IP Başlığı
26258         TCP Başlığı
26259             TCP Data'sı
26260
26261
26262     Örneğin biz send fonksiyonuyla "ankara" yazısını gönderiyor olsak bu "ankara" yazısını oluşturan byte'lar aslında TCP Data'sındadır.
26263
```

26264 Ether Başlığı
26265 IP Başlığı
26266 TCP Başlığı
26267 "Ankara"
26268
26269 Tabii IP paketleri aslında yalnızca bilgisayarımıza gelirken Ethernet paketi olarak gelir. Dışarda rotalanırken Ethernet paketi söz konusu değildir. ↗
26270
26271 TCP Başlığı şöyledir:
26272
26273 Source Port | Destination Port
26274 Sequence Number
26275 Acknowledge Number
26276 Data Offset | 000 | Flags | Window Size
26277 Checksum | Urgent Pointer
26278 Options
26279
26280 Burada her satır 32 bit yani 4 byte yer kaplamaktadır. TCP başlığı 20 byte'tan 60 byte'a kadar değişen uzunlukta olabilir. Başlıktaki Data Offset ↗
26281 TCP data'sının hangi offsetten başladığını dolayısıyla TCP başlığının uzunuunu belirtir. Options 4 bit uzunluğundadır. Buradak değer 4 ile çarpılmalıdır. ↗
26282 Böylece Data Offset kısmında en az 5 (toplam 20 byte) en fazla 15 (toplam 60 byte) değeri bulunabilir. Bu başlıkta kaynak ve hedef IP adreslerinin ve ↗
26283 TCP data kısmının uzunluğunun bulunmadığına dikkat ediniz. Çünkü bu bilgiler zaten IP başlığında doğrudan ya da dolaylı biçimde bulunmaktadır. ↗
26284 TCP paketi her zaman IP paketinin data kısmında kodlanır.
26285
26286 Başlıktaki Flags alanı 6 bitten oluşmaktadır. Her bir bir özelliği temsil eder. Buradaki belli bitler set edildiğinde başlıktaki belli alanlar da ↗
26287 anlamlı hale getirilmektedir. Buradaki bitler şöyledir: URG, ACK, PSH, RST, SYN, FIN. Flags alanındaki bir'den fazla bit 1 olabilir. Bir TCP paketi (TCP segment) ↗
26288 yalnızca başlık içerebilir. Yani hiç data içermeyebilir.
26289
26290 TCP protokolünde o anda iki tarafında bulunduğu bir durum vardır. Taraflar belli eylemler sonucunda durumdan duruma geçiş yaparlar. Bu nedenle TCP'nin çalışması ↗
26291 bir "sonlu durum makinesi (finite state machine)" biçiminde ele alınıp açıklanabilir. Henüz bağlantı yoksa iki taraf da CLOSED denilen durumdadır. ↗
26292
26293 TCP bağlantısının kurulması için client ile server bazı paketleri gönderip almaktadır. Buna el sıkışma (hand shaking) denilmektedir. TCP'de bağlantı kurulması ↗
26294 için yapılan el sıkışma 3'lü (three way) ya da 4'lü (four way) olabilir. Uygulamada 3'lü el sıkışma kullanılmaktadır. Normal olarak bağlantılarının kurulabilmesi ↗

26295 için iki tarafın da birbirlerine SYN biti set edilmiş data kısmı olmayan →
 TCP paketi (20 byte) gönderip karşı taraftan ACK biti set edilmiş →
 (data'sı olmayan)

26296 TCP paketi alması gereklidir. Bunun iki yolu olabilir:

26297

26298 Client Server
26299 CLOSED CLOSED
26300 ----- SYN ----->
26301 SYN-SENT SYN-RECEIVED
26302 <----- ACK -----
26303 <----- SYN -----
26304 ESTABLISHED
26305 ----- ACK ----->
26306 ESTABLISHED
26307

26308 Burada 4 paket kullanıldığı için buna 4'lü el sıkışma denilmektedir. →
 Ancak pratikte 3'lü el sıkışma kullanılmaktadır:

26309

26310 Client Server
26311 CLOSED CLOSED
26312 ----- SYN ----->
26313 SYN-SENT SYN-RECEIVED
26314 <-- SYN + ACK ----
26315 ESTABLISHED
26316 ----- ACK ----->
26317 ESTABLISHED
26318

26319 Bağlantı kopartılması için iki tarafın da birbirlerine FIN biti set →
 edilmiş paketler gönderip ACK biti set edilmiş paketleri alması →
 gerekir.

26320 Bağlantının kopartılması da tipik olarak 3'lü ya da 4'lü el sıkışma →
 yoluyla yapılmaktadır. Bağlantının kopartılması talebini herhangi bir →
 taraf başlatabilir.

26321 3'lü el sıkışma ile bağlantının kopartılması şöyle
26322 yapılmaktadır

26323

26324 Peer-1 Peer-2

26325

26326 ESTABLISHED ESTABLISHED
26327 ----- FIN ----->
26328 FIN-WAIT-1 CLOSE_WAIT
26329 <-- FIN + ACK ----
26330 FIN-WAIT-2 LAST-ACK
26331 ----- ACK ----->
26332 TIME-WAIT CLOSED
26333 CLOSED

26334

26335 Burada özetle bir taraf önce karşı tarafa FIN paketi yollamıştır. Karşı →
 taraf buna ACK+FIN ile karşılık vermiştir. Diğer taraf da son olarak →
 karşı tarafa ACK

26336 yollamıştır. Ancak bağlantıyı koprmak isteyen taraf bu ACK yollama →
 işinden sonra MSL (Maximum Segment Life) denilen bir zaman aralığının →
 iki katı kadar

-
- 26337 beklemektedir (Tipik olarak 2 dakika). Eğer bu taraf beklemedne hemen CLOSED duruma geçseydi şöyle bir sorun oluşabilirdi: Bu taraf hemen yeniden programı çalıştırıp ↗
26338 karşı tarafa paket yollayıp sanki eski bağlantıyı devam ettirme gibi bozuk bir durum söz konusu olabilirdi. Halbuki diğer taraf zaten bu son ACK paketini almamışsa ↗
26339 belli zaman sonra CLOSED duruma geçmektedir.
- 26340
- 26341 Eğer shutdown işlemi ile sonlandırma yapılrsa sonlandırma 4'lü el sıkışmayla gerçekleşir. Pratikte daha çok bu durumla karşılaşılmaktadır. 4'lü el sıkışmaya ↗
26342 "yarı kapama (half close)" el sıkışması da denmektedir. Yarı kapama bir tarafın artık send yapmayıcağını ancak receive yapabileceğini belirtir. Bu durumda bağlantı devam edebilir. Ta ki karşı taraf da bu half close işlemini yapana kadar. ↗
26343
- 26344
- 26345 Peer-1 Peer-2
- 26346
- 26347 ESTABLISHED ESTABLISHED
- 26348 ----- FIN ----->
- 26349 FIN-WAIT-1 CLOSE-WAIT
- 26350 <-- ACK -----
- 26351 CLOSING
- 26352
- 26353 Artık Peer-1 data gönderemez ama alabilir. Peer-2 ise alamaz ama gönderebilir. ↗
26354
- 26355 <-- FIN -----
- 26356 TIME_WAIT
- 26357 ---- ACK ----->
- 26358 TIME_WAIT
- 26359 CLOSED CLOSED
- 26360
- 26361 TCP'de akış kontrolü için "acknowledgement" yani alındı bildirimini kullanılmaktadır. Bir taraf bir tarafa birşeyler gönderdiği zaman karşı taraf bunu ↗
26362 aldığıni Flags kısmındaki ACK biti set edilmiş bir paket ile bildirir. ACK paketleri boş olabilir ya da dolu olabilir. Yani karşı taraf bir bilgiyi ↗
26363 gönderirken de aynı anda daha önce almış olduğu bilgiler için ACK yollayabilir. TCP'de kümülatif bir "acknowledgement" sistemi kullanılmaktadır. ↗
26364 Gönderilen her byte'in bir başlangıç sıra numarası vardır. Buna "sequence number" denilmektedir. Sequence number TCP başlığında 32 bitlik bir alandır. ↗
26365 Bu alandaki sayı son noktasına gelirse yeniden başa dönmektedir. Sequence number bağlantı kurulduğunda sıfırdan başlamaz, rastgele bir değerden başlatılmaktadır. ↗
26366 Örneğin belli bir anda bir tarafın sequence number değeri 1552 olsun. Şimdi bu taraf karşı tarafa 300 byte göndersin. Artık bu sequence number 1852 olacaktır. ↗
26367 Yani bir sonraki gönderimde bu taraf sequence number olarak 1852 kullanacaktır. Sequence number her gönderimde bulundurulmak zorundadır. ↗

- 26393 gönderici taraf en fazla 3K kadar daha bilgi gönderebilir. Tam bu sırada \Rightarrow
iki ayrı 1K için kümülatif olarak ACK geldiğini düşünelim. Şimdi \Rightarrow
pencere genişliği gönderen taraf \Rightarrow
- 26394 için ne olmalıdır? İşte bu durum alıcı tarafın ACK gönderirken aslında \Rightarrow
ACK gönderdiği bilgileri tamponunda eritmiş olup olmadığına bağlıdır. \Rightarrow
Pekala alıcı taraf \Rightarrow
- 26395 ACK gönderdiği halde henüz oradaki proses recv yapmadığı için bu \Rightarrow
bilgileri 8K'lık tamponunda hala bekletiyor olabilir. Bu durumda \Rightarrow
gönderici tarafın pencere genişliğinin \Rightarrow
- 26396 5K'ya düşürülmesi gereklidir. İşte TCP'de her ACK sırasında yeni pencere \Rightarrow
genişliği de aslında karşı tarafa gönderilmektedir. O halde aslında \Rightarrow
pencere genişliği her ACK \Rightarrow
- 26397 bildiriminde dinamik olarak ayarlanmaktadır. Tabii TCP çift taraflı \Rightarrow
olduğuna göre her iki tarafın ayrı sequence number, acknowledgement \Rightarrow
number ve pencere genişlikleri \Rightarrow
- 26398 vardır. Ayrıca ACK paketlerinin illa da daha önce alınmış olan bir \Rightarrow
paketen alındığını bildirmek amaçlı kullanılmayabileceğini de \Rightarrow
belirtelim. Bir taraf pencere genişliğini \Rightarrow
- 26399 ayarlamak için de ACK gönderebilir. Tabii bu durumda "acknowledgement \Rightarrow
number" yeni aynı değerde olmalıdır. \Rightarrow
- 26400
- 26401 Pekiyi sequence number, pencere genişlikleri nasıl iki tarafa \Rightarrow
bildirilmektedir. İşte bağlantı kurulurken client taraf SYN paketi \Rightarrow
içerisinde \Rightarrow
- 26402 kendi başlangıç sequence number'ını karşı tarafa iletmektedir. Server de \Rightarrow
bağlantıyı kabul ederken yine SYN (ya da SYN + ACK) paketinde kendi \Rightarrow
sequence number'ını \Rightarrow
- 26403 karşı tarafa bildirmektedir. Aslında SYN paketi yalnızca başlık \Rightarrow
icermemektedir. Data içermemektedir. Ancak TCP protokolüne göre özel bir \Rightarrow
durum olarak SYN paketlerinde \Rightarrow
- 26404 sanki tek bir byte varmış gibi acknowledgement number alınmaktadır. \Rightarrow
Pencere genişliği de aslında ilk kez bağlantı yapılarkenki ACK \Rightarrow
paketlerinde belirtilmektedir. \Rightarrow
- 26405 Yukarıda belirtildiği gibi pencere genişliği her ACK paketinde \Rightarrow
bildirilir. \Rightarrow
- 26406
- 26407 TCP/IP stack gerçekleştirimleri ACK stratejisi için bazı yöntemler \Rightarrow
uygulamaktadır. Örneğin eğer gönderilecek paket varsa bununla birlikte \Rightarrow
ACK paketinin \Rightarrow
- 26408 gönderilmesi, ACK'ların iki paket biriktirildikten sonra gönderilmesi \Rightarrow
gibi. Bunun için "TCP IP Protocol Suite 466'inci sayfaya \Rightarrow
başvurabilirsiniz." \Rightarrow
- 26409
- 26410 TCP paketindeki önemli Flag'lerden birisi de "RST" bitidir. Buna "reset \Rightarrow
isteği" denilmektedir. Bir taraf RST bayrağı set edilmiş paket alırsa \Rightarrow
artık karşı tarafın \Rightarrow
- 26411 "abnormal" bir biçimde bağlantıyı kopartıp yeniden bağlanma talep ettiği \Rightarrow
anlaşılır. Normal sonlanma el sıkışarak başarılı bir biçimde \Rightarrow
yapılırken RST işlemi anormal \Rightarrow
- 26412 sonlanmaları temsil eder. Örneğin soket kütüphanelerinde hiç shutdown \Rightarrow
yapmadan soket close edilirse close eden taraf karşı tarafa RST paketi \Rightarrow
göndermektedir. \Rightarrow
- 26413 Halbuki önce shutdown yapılrsa el sıkışmalı sonlanma gerçekleştirilir. \Rightarrow

O halde her zaman aktif soketler shutdown yapıldıktan sonra close edilmelidir.

26414
26415 -----*/
26416
26417 /
*-----

26418 UDP protokolü aslında saf IP protokolüne çok benzerdir. UDP'yi IP'den ayıran iki önemli farklılık şudur:
26419
26420 1) UDP'nin port numarası kavramına sahip olması
26421 2) UDP'nin hata için bir checksum kontrolü uygulayamamasıdır.
26422
26423 Bir UDP paketi yine aslında IP paketinin data kısmında bulunmaktadır. IPV4 Formatı şöyledir:
26424
26425 Source Port | Destination Port
26426 Total Length | Checksum
26427 Data
26428
26429 Burada UDP paketinin toplam uzunluğunun bulunması aslında gereksizdir. Çünkü uzunluk TCP'de olduğu gibi aslında IP paketinin başlığına bakılarak
26430 tespit edilebilmektedir. Ancak hesaplama kolaylı oluşturmak için bu uzunluk UDP başlığında ayrıca bulundurulmuştur. Aslında checksum UDP paketlerinde
26431 bulunmak zorunda değildir. Eğer gönderici checksum kontrolü istemiyorsa burayı 0 bitleriyle doldurur. (Eğer zaten checksum 0 ise burayı 1 bitleriyle
26432 doldurmaktadır.) Alan taraf checksum hatasıyla karşılaşırsa TCP'de olduğu gibi paketi yeniden talep etmez. Yalnızca onu atar.
26433 -----*/
26434
26435 /
*-----

26436 İçerisinde derlenmiş bir biçimde fonksiyonların bulunduğu dosyalara kütüphane (library) denilmektedir. Kütüphaneler yalnızca fonksiyon değil
26437 global nesneler de içerebilirler. Kütüphaneler "statik" ve "dinamik" olmak üzere ikiye ayrılmaktadır. Statik kütüphane dosyalarının uzantıları
26438 UNIX/Linux sistemlerinde ".a", Windows sistemlerinde ".lib" biçimindedir. Kütüphanelerin nasıl oluşturulacağı ve nasıl kullanılacağı iki ayrı alt konusu
26439 içermektedir.
26440
26441 Statik kütüphaneler aslında "object modülleri (yani .o dosyalarını)" tutan bir kap gibidir. Statik kütüphaneler aslında object modüllerden oluşmaktadır.

26442 Statik kütüphanelere link aşamasında linker tarafından bakılır. Bir →
program statik kütüphane dosyasından bir çağrıma yaptıysa (ya da o →
kütüphaneden bir global değişkeni kullandıysa)

26443 linker o static kütüphane içerisinde ilgili fonksiyonun bulunduğu →
object modülü link aşamasında static kütüphane dosyasından çekerek →
çalıştırılabilir dosyaya

26444 yazar. (Yani static kütüphanedne bir tek fonksiyon çağrırsak bile aslında →
o fonksiyonun bulunduğu object modülün tamamı çalıştırılabilen →
dosyaya yazılmaktadır.)

26445 Statik kütüphaneleri kullanan programlar artık o static kütüphaneler →
olmadan çalıştırılabilirler.

26446

26447 Statik kütüphanelerin şu dezavantajları vardır:

26448

26449 - Kütüphaneyi kullanan farklı programlar aynı fonksiyonun (onun →
bulunduğu object modülün) bir kopyasını çalıştırılabilir dosya →
icerisinde bulundururlar.

26450 Yani örneğin printf fonksiyonu static kütüphanede ise her printf →
kullanan C programı aslında printf fonksiyonun bir kopyasını da →
barındırıyor durumda olur.

26451 Bu da disk hacmini azaltır. Programların gereksinim duyacağı sanal →
bellek miktarını artırrır.

26452

26453 - Statik kütüphane kullanan programlar dinamik kütüphane kullanan →
programlara göre kimi zaman daha geç yüklenebilirler.

26454

26455 - Statik kütüphanede bir değişiklik yapıldığında onu kullanan →
programların yeniden link edilmesi gereklir.

26456

26457

26458 Statik kütüphane dosyalarının şu avantajları vardır:

26459

26460 - Kolay konuşlandırılabilirler.

26461

26462 - Kullanımı kolaydır ve build işlemini sadelştirirler.

26463

26464 UNIX/Linux sistemlerinde static kütüphane dosyaları ar isimli utility →
programla oluşturulur. Tipik kullanım şöyledir:

26465 ar r libsample.a a.o b.o c.o

26466

26467 r seçeneği (yanında - olmadığına dikkat ediniz) ilgili object modüllerin →
kütüphaneye yerleştirilmesini sağlar. kütüphane yoksa ayrıca kütüphane →
dosyası da

26468 yararlımadır. t seçeneği kütüphane içerisindeki object modüllerin →
listesini almakta kullanılır. Örneğin:

26469 ar t libsample.a

26470

26471 d seçeneği kütüphaneden bir object modülü silmekte, x seçeneği onu .o →
dosyası biçiminde save etmekte ve m seçeneği yeni versiyonu eski →
versiyonla değiştirmekte

26472 kullanılır. O halde a.c dosyasının içindekileri static kütüphaneye →
kullanılır.

eklemek şöyle yapılmalıdır:

26475
26476 gcc -c a.c
26477 ar r libsample.a a.o
26478
26479 Statik kütüphaneler link aşamasında linker için komut satırında
 belirtilmelidir. Örneğin: [»](#)
26480
26481 gcc -o app app.c libsample.a
26482
26483 gcc .a uzantılı dosyaları linker'a pass etmektedir.
26484
26485 Komut satırında kütüphane dosyalarının komut satırı argümanlarının
 sonunda belirtilmesi uygundur. Çünkü gcc programı kütüphane
 dosyalarının
26486 solundaki dosyalar link edilirken ilgili kütüphane dosyasını bu işleme [»](#)
 dahil ederler. [»](#)
26487
26488 Şüphesiz statik kütüphane kullanmak yerine aslında object modülleri de [»](#)
 doğrudan link işlemeye sokabiliriz. Örneğin:
26489
26490 gcc -o sample sample.c a.o b.o
26491
26492 Çok sayıda object modül söz konusu olduğunda bu işlemin zorlaşacağına [»](#)
 dikkat ediniz. Yani object modüller dosyalara benzetilirse statik [»](#)
 kütüphane dosyaları
26493 dizinler gibi düşünülebilir.
26494
26495 Derleme işlemi sırasında kütüphane dosyası -l<isim> biçiminde de [»](#)
 belirtilebilir. Bu durumda arama sırasında "lib" öneki aramaya dahil [»](#)
 edilmektedir.
26496 Yani örneğin:
26497
26498 gcc -o sample sample.c -lsample
26499
26500 İşleminde aslında libsample.a (ya da libsample.so) dosyaları [»](#)
 aranmaktadır. Arama işlemi sırasıyla bazı dizinlerde yapılmaktadır. [»](#)
 Örneğin /lib dizini,
26501 /usr/lib dizini gibi dizinlere bakılmaktadır. Ancak bulunulan dizine [»](#)
 bakılmamaktadır. Belli bir dizine bakılması isteniyorsa -L seçeneği [»](#)
 ile dizin eklenebilir.
26502 Örneğin:
26503
26504 gcc -o sample sample.c -lsample -L.
26505
26506 Buradaki '.' çalışma dizinini temsil etmektedir.
26507
26508 Bir statik kütüphane başka bir statik kütüphaneye bağımlı olabilir. [»](#)
 Örneğin biz liby.a Kütüphanesindeki kodda libx.a kütüphanesindeki [»](#)
 fonksiyonları kullanmış olabiliriz.
26509 Bu durumda liby.a Kütüphanesini kullanan program libx.a kütüphanesini de [»](#)
 komut satırında belirtmek zorundadır. Örneğin:
26510

26511 gcc -o sample sample.c libx.a liby.a
26512
26513 -----
26514 */
26515 /
*-----
26516 Dinamik kütüphane dosyalarının UNIX/Linux sistemlerinde uzantıları
26517 ".so" (so shared object'ten kısaltma), Windows sistemlerinde ise
26518 .dll (Dynamic Link Library) biçimindedir.
26519 Bir dinamik kütüphaneden bir fonksiyon çağrıldığında linker statik
26520 kütüphanede olduğu gibi gidip fonksiyonun kodunu çalıştırılabilen
26521 dosyaya yazmaz.
26522 Bunun yerine çalıştırılabilen dosyaya çağrılan fonksiyonun hangi dinamik
26523 kütüphanede olduğu bilgisini yazar. Çalıştırılabilen dosyayı yükleyen
26524 işletim sistemi
26525 o dosyanın çalışması için gerekli olan dinamik kütüphaneleri
26526 çalıştırılabilen dosyayla birlikte bütünsel olarak sanal bellek
26527 alanına yüklemektedir. Böylece birtakım
26528 ayarlamalar yapıldıktan sonra artık çağrılan fonksiyon için gerçekten o
26529 anda sanal belleğe yüklü olan dinamik kütüphane kodlarına
26530 gidilmektedir.
26531
26532 Dinamik kütüphane kullanımının avantajları şunlardır:
26533
26534 - Çalıştırılabilen dosyalar fonksiyon kodlarını içermezler. Dolayısıyla
26535 önemli bir disk hacmi kazanılmış olur.
26536
26537 - Dinamik kütüphaneler birden fazla program tarafından tekrar tekrar
26538 yüklenmeden kullanılabilirler. Yani işletim sistemi arka planda
26539 aslında
26540 aynı dinamik kütüphaneyi kullanan programlarda bu kütüphaneyi tekrar
26541 tekrar fiziksel belleğe yüklememektedir. Bu da statik kütüphanelere
26542 göre önemli bir avantaj
26543 oluşturmaktadır. Bu durum eğer dinamik kütüphane daha önce yüklenmişse
26544 programın daha hızlı yüklemesine de yol açabilmektedir.
26545
26546 - Dinamik kütüphaneleri kullanan programlar bu dinamik kütüphanelerdeki
26547 değişikliklerden etkilenmezler. Yani biz dinamik kütüphanenin yeni bir
26548 versiyonunu
26549 oluşturduğumuzda bunu kullanan programları derlemek ya da link etmek
26550 zorunda kalmayız.
26551
26552 Dinamik kütüphaneler proseslerin sanal bellek alanlarının farklı
26553 yerlerine yüklenebilirler. Bunların yüklenme adreslerinin değişik
26554 olabilmesi "relocation"
26555 denilen sorunu ortaya çıkarmaktadır. "Relocation" çalıştırılabilir ya da
26556 dinamik kütüphane dosyalarının sanal bellekte herhangi bir yere
26557 yüklenebilmesi
26558 anlamına gelir. Ancak relocation işlemi basit bir işlem değildir. Çünkü
26559 bu dosyalar derlenirken derleyici onların sanal belleğin neresine

yükleneceğini bilmemektedir.

26538 Relocation işlemi için temelde iki teknik kullanılmaktadır:

26539

26540 1) Relocation Tablosu Tekniği

26541 2) Konumdan Bağımsız Kod (Position Independent Code) Tekniği

26542

26543 Windows işletim sistemi "relocation tablosu" tekniğini kullanmaktadır. ↵
Burada dosyada "relocation" bilgilerinin bulunduğu bir alan vardır.

26544 Bu alanda adresleri düzeltilecek offset'ler dosyan başından itibaren tek ↵
tek belirtilmektedir. Yükleyici DLL'i uygun yere yükledikten sonra

26545 bu relocation tablosundaki offsetleri tek tek düzeltir. Böylece kod ↵
yükleniği yerde çalışır hale getirilir. UNIX/Linux sistemlerinde ise ↵
ağırlıklı olarak

26546 "Konumdan Bağımsız Kod" Tekniği kullanılmaktadır. Burada üretilen kodlar ↵
zaten hep görelî adresler içerir. Böylece üretilen bu kodlar zaten ↵
görelî adresler

26547 içeriği için nereye yüklenirse çalışabilir.

26548

26549 Relocation Tablosu yönteminde "relocation" işlemi belli bir zaman ↵
almaktadır. Bu da programın yüklenme zamanını uzatabilmektedir. Fakat ↵
dinamik kütüphane

26550 "reloaction" yapıldığında çalışma hızlidır. Konumdan Bağımsız Kod ↵
teknlığında "reloaction" zamanı minimize edilmiştir. Fakat görelî ↵
adreslerle çalışan kodlar

26551 toplamda daha yavaş olma eğilimindedir.

26552

26553 UNIX/Linux sistemlerinde bri dinamik oluşturma işlemi şöyle yapılır:

26554

26555 1) Önce dinamik kütüphaneye yerleştirilecek object modüllerin -fPIC ↵
seçeneği ile Konumdan Bağımsız Kod teknliği kullanılarak derlenmesi ↵
gerekşir

26556 2) Link aşamasında -shared seçeneğini kullanılması gereklidir. -shared ↵
kullanılmazsa dinamik kütüphane değil normal çalıştırılabilir dosya ↵
oluşturulur.

26557

26558 gcc -fPIC a.c b.c c.c

26559 gcc -o libsample.so -shared a.o b.o c.o

26560

26561 Dinamik kütüphanelere daha sonra dosya eklenip çıkartılamaz. Onların her ↵
defasında yeniden oluşturulması gerekmektedir. Yukarıdaki işlem tek ↵
hamlede

26562 şöyle de yapılabilir:

26563

26564 gcc -o libsample.so -shared -fPIC a.c b.c c.c

26565

26566 Dinamik kütüphane kullanan bir program link edilirken kullanılan dinamik ↵
kütüphanenin komut satırında belirtilmesi gereklidir. Örneğin:

26567

26568 gcc -o sample sample.c libsample.so

26569

26570 Tabii bu işlem yine -l seçeneği ile de yapılabilirdi:

26571

26572 gcc -o sample sample.c -lsample -L.

26573
26574 -----*/
26575
26576 / -----*

26577 Default durumda gcc (ve tabii clang) derleyicileri standart C
fonksiyonlarının ve POSIX fonksiyonlarını (libc kütüphanesi)
26578 dinamik kütüphaneden alarak kullanır. Ancak programcı isterse -static
seçeneği ile statik link işlemi de yapabilir. Bu durumda
26579 bu fonksiyonlar statik kütüphanelerden alınarak çalıştırılabilen
dosyalara yazılacaktır. Örneğin:
26580
26581 gcc -o sample -static sample.c
26582
26583 Tabii bu biçimde statik link işlemi yapıldığında çalıştırılabilen
dosyanın boyutu çok büyüyecektir.
26584
26585 Bir programın kullandığı dinamik kütüphaneler ldd isimli utility
program ile basit bir biçimde görüntülenebilir.
26586 Örneğin:
26587
26588 csd@csd-vm:~/Study/Unix-Linux-SysProg/static-lib\$ ldd sample
26589 linux-vdso.so.1 (0x00007fff38162000)
26590 libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f7ec0b5c000)
26591 /lib64/ld-linux-x86-64.so.2 (0x00007f7ec114f000)
26592
26593 -----*/
26594
26595 / -----*

26596 Pekiyi bizim programımız örneğin libsample.so isimli bir dinamik
kütüphaneden çağrı yapıyor olsun. Bu libsample.so dosyasının
26597 program çalıştırılırken nerede bulundurulması gereklidir? İşte program
çalıştırılırken ilgili dinamik kütüphane dosyasının özel bazı
dizinlerde
26598 bulunuyor olması gereklidir. Yükleme sırasında hangi dizinlere bakıldığı
"man ld.so" sayfasından elde edilebilir. Bu dokümanda belirli sırada
26599 tek tek hangi dizinlere bakıldığı belirtilmiştir. Ayrıntılar bir yana
bırakılırsa kabaca LD_LIBRARY_PATH isimli çevre değişkeninde
26600 belirtilen dizinlere (bu çevre değişkeni de PATH çevre değişkeni gibi
':ler ile ayrılmış dizinler oluşmaktadır. '.' dizini yine "o anki
çalışma dizini"
26601 anlamına gelmektedir.) ve /lib, /usr/lib dizinlerine bakılmaktadır. O
halde basit bir seçenek bizim dinamik kütüphanemizi /lib ya da /usr/
lib
26602 dizinlerinin birinin içine çekmek ya da LD_LIBRARY_PATH çevre
değişkenine dinamik kütüphanemizin bulunduğu dizini eklemektedir. /lib
dizini işletim sisteminin
26603 boot edilebilmesi için gereken birtakım araçların da kullandığı daha

temel bir kütüphane dizinidir. Uygulama programcılarının kütüphanelerini buraya değil /usr/lib içerisinde yerlestirmesi daha uygun olur.

26605 Dinamik kütüphanelerin aranacağı yer aslında doğrudan çalıştırılabilen dosyanın içerisinde (dynamic isimli bölüme (section)) da yazılabilir. Bunun için -rpath <yol ifadesi> linker seçeneği kullanılmalıdır. Buradaki yol ifadesinin mutlak olması gereklidir. Ancak eskiden bunun için DT_RPATH isimli tag kullanılırken daha sonra bu tag "deprecated" yapılmış DT_RUNPATG tag'ı kullanılmaya başlanmıştır. Bu seçeneği linker'a geçirebilmek için gcc'de -Wl seçeneğini kullanmak gereklidir. -Wl seçeneği bitişik yazılan virgülü alanlardan oluşmaktadır. gcc bını ld linler'ına virgüller yerine boşluklar koyarak geçirmektedir. Örneğin:

```
26610 gcc -o sample -Wl,-rpath,/home/csd/Study/Unix-Linux-SysProg/static-  
26611 lib, sample.c libsample.so
```

26612 Burada ELF formatının DT_RUNPATH tag'ına yerleştirme yapmaktadır. Eğer DT_RPATH tag'ına yerleştirme yapmak isteniyorsa ayrıca --disable-new-dtags seçeneğinin de girilmesi gerekmektedir. Ancak burada küçük bir pürüz vardır. Kurulum sırasında dinamik kütüphane ya başka dizine yerleştirilirse ne olacaktır? İşte -rpath seçeneğinde '\$ORIGIN' program dosyasının bulunduğu dizin anlamına gelir. Örneğin:

```
26616 gcc -o sample -Wl,-rpath,'$ORIGIN'/. sample.c libsample.so
```

26618 Burada artık neresi olursa olsun dinamik kütüphane çalıştırılan programla aynı dizinde bulunursa sorun oluşmayacaktır. Peki DT_RPATH tag'ı neden deprecated yapılmıştır?

26620 DT_RUNPATH tag'ı eklenmiştir? İşte aslında arama sırası bakımından DT_RPATH tag'ının en yukarıda olması (daha sağlığı LD_LIBRARY_PATH'in yukarısında olması)

26621 yanlış bir tasarımdır. Geriye doğru uyumu koruyarak bu yanlış tasarım telafi edilmiştir.

26622 Dinamik kütüphanenin aranması sırasında /lib ve /usr/lib dizilerine bakılmadan önce özel bir dosyaya da bakılmaktadır. Bu dosya /etc/ld.so.cache isimli dosyadır.

26624 /etc/ld.so.cache dosyası aslında binary bir dosyadır. Hızlı arama yapılabilen sözlük tarzı ilgili dosyanın hangi dizinde olduğunu gösteren bir yapıdadır.

26625 Yani bu dosta .so dosyalarının hangi dizinlerde olduğunu belirten binary bir dosyadır. Peki bu dosyanın içerisinde hangi so dosyaları vardır? Aslında bu dosyanın içerisinde /lib ve /usr/lib dizinindeki so dosyalarının hepsi vardır. Ama programcı isterse kendi dosyalarını da bu cache dosyasının içerisinde yerlestirebilir.

26627 Peki neden böyle bir cache dosyasına geeksinim duyulmuştur? Dinamik

kütüphaneler yüklenirken /lib ve /usr/lib dizinlerinin taranması
göreli olarak uzun zaman

26628 almaktadır. Bu da programın yüklenme süresini uzatabilmektedir. Halbuki bu dizinlere bakılmadan önce bu cache dosyasına bakılırsa ilgili dosyanın olup olmadığı varsa

26629 nerede olduğu çok daha hızlı bir biçimde elde edilebilmektedir. Burada dikkat edilmesi gereken nokta bu cache dosyasına /lib ve /usr/lib dizinlerinden daha önce bakıldığı

26630 ve bu dizinlerin içeriğinin de zaten bu cache dosyasının içerisinde olduğunu. O halde aslında /lib ve /usr/lib dizinlerinde arama çok nadir olarak yapılır.

26631

26632 Pekiyi /etc/ld.so.cache dosyasına biz nasıl bir dosya ekleriz? Aslında programcı bunu dolaylı olarak yapar. Şöyle ki: /sbin/ldconfig isimli bir program vardır.

26633 Bu program /etc/ld.so.conf isimli bir text dosyaya bakar. Bu dosya dizinlerden oluşmaktadır. Bu dizinlerin içindeki so dosyalarını /etc/ld.so.cache dosyasına

26634 ekler. Şimdi de /etc/ld.so.conf dosyasının içeriği şöyledir:

26635

26636 include /etc/ld.so.conf.d/*.conf

26637

26638 Bu ifade /etc/ld.so.conf.d dizinindeki tüm .conf uzantılı dosyaların bu işleme dahil edileceğini belirtir.

26639

26640 Biz ldconfig programını çalıştırduğumda bu program /lib, /usr/lib ve /etc/ld.so.conf (dolayısıyla /etc/ld.so.conf.d dizinindeki .conf dosyaları)na bakarak

26641 /etc/ld.so.cache dosyasını yeniden oluşturmaktadır. O halde bizim bu cache'e ekleme yapmak için tek yapacağımız şey /etc/ld.so.conf.d dizinindeki bir .conf

26642 dosyasına yeni bir satır olarak bir dizinin yol ifadesini girmektir. (.conf dosyaları her satırda bir dizinin yol ifadesinden oluşmaktadır) Tabii programcı

26643 isterse bu dizine yeni bir .conf dosyası da ekleyebilir. İşte programcı bu işlemi yaptıktan sonra /sbin/ldconfig programını çalıştırınca artık onun eklediği

26644 dizinin içerisindeki so dosyaları da /etc/ld.so.cache dosyasının içerisine eklenmiş olacaktır. Daha açık bir anlatımla programcı bu cache dosyasına ekleme işini

26645 adım adım şöyle yapar:

26646

26647 1) Önce so dosyasını bir dizine yerleştirir.

26648 2) Bu dizinin ismini /etc/ld.so.conf.d dizinindeki bir dosyanın sonuna ekler.

26649 3) /sbin/ldconfig programını çalıştırır.

26650

26651 Programcı /etc/ld.so.conf.d dizinindeki herhangi bir dosyaya değil de -f seçeneği ile kendi belirdiği bir dosyaya da ilgili dizini yazabilmektedir. ldconfig her çalıştırıldığında sıfırdan yeniden cache dosyasını oluşturmaktadır.

26653

26654 Programcı /lib ya da /usr/lib dizinine bir so dosyası eklediğinde

ldconfig programını çalıştırması zorunlu olmasa da iyi bir tekniktir. ↗
Çünkü o dosya da cache
26655 dosyasına yazılacak ve daha hızlı bulunacaktır.

26656
26657 -----*/
26658 /
26659 *-----
26660 UNIX/Linux sistemlerinde dinamik kütüphane dosyalarına istege bağlı
olarak birer versiyon numarası verilmektedir. Bu versiyon numarası
dosya
26661 isminin bir parçası durumundadır. Linux sistemlerine izlenen tipik
numaralandırma (convention) şöyledir:
26662
26663 <dosya ismi>.so.<majör numara>.<minör yüksek numara>.<minör alçak
numara>
26664
26665 Örneğin:
26666
26667 libsample.so.2.4.6
26668
26669 Majör numara büyük değişiklikleri, minör numaralar ise küçük
değişiklikleri anlatmaktadır. Majör numara değişimse yeni dinamik
kütüphane
26670 eskisiyle uyumlu değildir. Burada "uyumlu değildir" lafi eski dinamik
kullanan programların yenisini kullanamayacağı anlamına gelmektedir.
26671 Çünkü muhtemelen bu yeni versiyonda fonksiyonların isimlerinde,
parametrik yapılarında değişiklikler söz konusu olmuş olabilir. Bazı
fonksiyonlar silinmiş olabilir.
26672 Fakat majör numarası aynı ancak minör numaraları farklı olan
kütüphaneler birbirleriyle uyumludur. Yani alçak minör numarayı
kullanan program yüksek minör
26673 numarayı kullanırsa sorun olmayacağıdır. Bu durumda yüksek minör numaralı
kütüphanede hiçbir fonksiyonun ismi, parametrik yapısı değişmemiştir
ve hiçbir fonksiyon
26674 silinmemiştir. Örneğin fonksiyonlar daha hızlı çalışacak biçimde
optimizasyonlar yapılmış olabilir. Ya da örneğin yeni birtakım
fonksiyonlar da eklenmiş
26675 olabilir.
26676
26677 Linux sistemlerinde bir konvansiyon olarak bir kütüphanenin üç ismi
bulunmaktadır: Gerçek ismi (real name), so ismi (so name) ve linker
ismi (linker name).
26678 Kütüphanenin majör ve çift minör versiyonlu ismine gerçek ismi
denilmektedir. Örneğin:
26679
26680 libsample.so.2.4.6
26681
26682 so ismi ise yalnızca majör numara içerisinde ismidir. Örneğin yukarıdaki
gerçek ismin so ismi şöyledir:
26683

26684 libsample.so.2
26685
26686 Linker ismi ismi ise hiç versiyon numarası içermeyen ismidir. Örneğin ↗
 yukarıdaki kütüphanelerin linker ismi ise şöyledir:
26687
26688 libsample.so
26689
26690 Tabii aslında kütüphaneden bir tane vardır. O da gerçek isimli ↗
 kütüphanedir. so ismi gerçek isme sembolik link, linker ismi de so ↗
 ismine sembolik
26691 link yapıılır.
26692
26693 linker ismi ---> so ismi ---> gerçek ismi
26694
26695 Örneğin:
26696
26697 gcc -o libsample.so.1.0.0 -shared -fPIC a.c b.c c.c (gerçek isimli ↗
 kütüphane dosyası oluşturuldu)
26698 ln -s libsample.so.1.0.0 libsample.so.1 (so ismi ↗
 oluşturuldu)
26699 ln -s libsample.so.1 libsample.so (linker ismi ↗
 oluşturuldu)
26700
26701 lrwxrwxrwx 1 csd study 14 Nov 26 22:32 libsample.so -> libsample.so.1
26702 lrwxrwxrwx 1 csd study 18 Nov 26 22:31 libsample.so.1 -> ↗
 libsample.so.1.0.0
26703 -rw xr-xr-x 1 csd study 7552 Nov 26 22:30 libsample.so.1.0.
26704
26705 Dinamik kütüphanelerin linker isimleri link aşamasında kullanılan ↗
 isimlerdir. Bu sayede link işlemini yapan programcıların daha az tuşa ↗
 basarak
26706 genel bir isim kullanması sağlanmıştır. Bu durumda örneğin biz libsample ↗
 isimli kütüphane kullanan programı link etmek istersek şyle ↗
 yapabiliriz:
26707
26708 gcc -o sample sample.c libsample.so
26709
26710 Ya da şöyledir yapabiliriz:
26711
26712 gcc -o sample sample.c -lsample -L.
26713
26714 Dinamik kütüphanelerin so isimleri aslında kütüphanenin tüm uyumlu ↗
 (compatible) versiyonlarının ortak ismi gibi düşünülebilir. ↗
 Kütüphanelerin SO isimleri aslında
26715 aynı zamanda kütüphanelerin gerçek isimlerine ilişkin kütüphane ↗
 dosyalarının içerisinde SONAME tag'ıyla yazılmaktadır. Yani dinamik ↗
 kütüphane dosyalarının içerisinde
26716 SONAME isimli bir alan vardır. Bu SONAME alanı link işlemi sırasında - ↗
 sonmame seneği ile doldurulur. Dİnamik kütüphaneyi oluşturan programcı ↗
 genel olarak bu SONAME
26717 alanına kütüphanenin soname'ini yazmalıdır. Örneğin:
26718
26719 gcc -o libsample.so.1.0.0 -Wl,-soname,libsample.so.1 -shared a.o b.o c.o ↗

26720
26721 Tabii programcı kütüphane dosyasında bu SONAME alanına hiç bir şey yazmayabilir. Örneğin: ↗
26722
26723 gcc -o libsample.so.1.0.0 -shared a.o b.o c.o
26724
26725 Bir kütüphane dosyasının SONAME alanı readelf -d seçeneği ile görüntülenebilir. (-d seçeneği ".dynamic" isimli bölünmü görüntülemektedir. SONAME alanı bu bölüm (section) içerisindeindir.) Örneğin: ↗
26726
26727
26728 readelf -d libsample.so.1.0.0 | grep SONAME
26729
26730 Aynı işlemi objdump utility'si ile de şöyle yapabiliriz:
26731
26732 objdump -x libsample.so.1.0.0 | grep SONAME
26733
26734 Dinamik kütüphanelerin so isimleri yükleme sırasında gerçek yüklenerek dosyayı belirtmektedir. Yani örneğin libsample.so.1.0.0 dinamik kütüphane dosyasının SONAME alanında libsample.so.1 ismi yazıyorsa aslında bu kütüphane yerine yükleyici libsample.so.1 isimli kütüphaneyi yükleyecektir. Tabii mademki
26735 dosyasının SONAME alanında libsample.so.1 ismi yazıyorsa aslında bu kütüphane yerine yükleyici libsample.so.1 isimli kütüphaneyi yükleyecektir. Tabii mademki
26736 libsample.so.1 dosyası libsample.so.1.0.0 dosyasına sembolik link yapılmıştır. Bu durumda yükleyici yine libsample.so.1.0.0 dosyasını yükler. Şimdi, bu işlemleri yeniden sırasıyla en baştan yapalım: ↗
26737
26738
26739 gcc -c -fPIC a.c b.c c.c
26740 gcc -o libsample.so.1.0.0 -shared -Wl,-soname,libsample.so.1 a.o b.o c.o
26741 ln -s libsample.so.1.0.0 libsample.so.1
26742 ln -s libsample.so.1 libsample.so
26743
26744 Şimdi de kütüphaneyi kullanan sample isimli programı derleyip link edelim: ↗
26745
26746 gcc -o sample sample.c libsample.so
26747
26748 Şimdi ldd utility'si ile sample programının kullandığı dinamik kütüphanelere bakalım: ↗
26749
26750 linux-vdso.so.1 (0x00007ffd809b6000)
26751 libsample.so.1 => ./libsample.so.1 (0x00007ff0f491d000)
26752 libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff0f452c000)
26753 /lib64/ld-linux-x86-64.so.2 (0x00007ff0f4d21000)
26754
26755 Görüldüğü gibi aslında sample programı libsample.so sembolik linkinin izlediği libsample.so.1.0.0 dosyasını değil bu dosyanın SONAME alanında yazılı olan ↗
26756 libsample.so.1 dosyasını yüklemek istemektedir. Tabii neticede kütüphanelerin so isimleri gerçek isimlerine sembolik link yapıldığına göre aslında yine yükleyici ↗

26757 libsample.so.1.0.0 dinamik kütüphanesini yükleyecektir.

26758

26759 Pekiyi çalıştırılabilen bir dosyanın hangi dinamik kütüphaneleri kullanıldığı nerede yazmaktadır? İşte aslında bu bilgi çalıştırılabilen dosyanın içersinde

26760 .dynamic isimli bölümde DT_NEEDED taglarıyla belirtilir. ldd programı da aslında bu tag'lara bakmaktadır. Biz bu tagı yine istersek readelf -d ile görüntüleyebiliriz.

26761 Örneğin:

26762

26763 readelf -d sample | grep NEEDED

26764 0x0000000000000001 (NEEDED) Shared library: [libsample.so.1]

26765 0x0000000000000001 (NEEDED) Shared library: [libc.so.6]

26766

26767 İşte aslında dinamik kütüphane kullanan program link edilirken linker kütüphane dosyasının SONAME tag alanına bakıp orada yazan dosyayı çalıştırılabilen dosyanın

26768 NEEDED alanına yazmaktadır. Yükleyici aslında her zaman bu NEEDED alanındaki dinamik kütüphaneleri yüklemeye çalışmaktadır.

26769

26770 Pekiyi kütüphanenin SONAME alanına bir şey yazılmazsa ne olur? İşte bu durumda linker link işlemi sırasında belirtilen dinamik kütüphane dosyasını çalıştırılabilen dosyanın

26771 NEEDED alanına yerleştirir. Örneğin:

26772

26773 gcc -c -fPIC a.c b.c c.c

26774 gcc -o libsample.so.1.0.0 a.o b.o c.o

26775 ln -s libsample.so.1.0.0 libsample.so.1

26776 in -s libsample.so.1 libsample.so

26777

26778 gcc -o sample sample.c libsample.so

26779

26780 Burada sample programının NEEDED tag'ı şöyle oluşturulacaktır:

26781

26782 readelf -d sample | grep NEEDED

26783 x0000000000000001 (NEEDED) Shared library: [libsample.so]

26784 0x0000000000000001 (NEEDED) Shared library: [libc.so.6]

26785

26786 Tabii sonuç olarak oluşturulan semblik linlerden dolayı yine yükleyici libsample.so.1.0.0 dosyasını yükleyecektir.

26787

26788 Peki bu durumda so isminin anlamı nedir? İşte eğer biz asıl isme ilişkin dosyanın so ismine yerleştirme yaparsak çalıştırılabilen dosya o isimdeki dosyası yüklemek

26789 isteyecektir. Biz de sembolik linkleri değiştirerek onun başka bir minor versiyonu yüklemesini sağlayabiliriz. Yani örneğin yukarıdaki uygulamada sample programı aslında

26790 sample.so.1 dosyasını yüklemek istemektedir. Normalde de bu dosya sample.so.1.0.0 dosyasını göstermektedir. Şimdi biz bu dosyanın ileride örneğin sample.so.1.0.1

26791 dosyasını göstermesini sağlayabiliriz. Artık sample programı çalıştırıldığında bu yeni versiyonu yüklemeye çalışacaktır.

26792

26793 so ismine ilişkin sembolik link çıkartma ve /etc/ld.so.cache dosyasının güncellenmesi işlemi ldconfig tarafından otomatik yapılmaktadır. Yani aslında biz kütüphanenin gerçek isimli dosyasını
26794 /lib ya da /usr/lib içerisinde yerleştirip ldconfig programını çalıştırıldığımızda bu program zaten so ismine ilişkin sembolik linki de oluşturmaktadır.

26795
26796 Özetle Dinamik Kütüphane kullanırken şu konvansiyona uymak iyi bir tekniktir:

26797 - Kütüphane ismini lib ile başlatarak vermek
26798 - Kütüphane ismine mjör ve minör numara vermek
26799 - Gerçek isimli Kütüphane dosyasını oluştururken so ismi olarak -Wl,-soname seçeneği ile kütüphanenin so ismini yazmak
26800 - Kütüphane için linker ismi ve so ismini sembolik link biçiminde oluşturmak
26801 - Kütüphane paylaşılacaksa onu /usr/lib dizinine yerleştirmek ve ldconfig programı çalıştırarak /etc/ld.so.cache dosyasının güncellenmesini sağlamak. Bu işlem eğer
26802 kütüphaneyi /usr/lib içersine yerleştirmiyorsak yapılmayabilir.
26803
26804
26805 -----*/
26806
26807 /*-----
26808 Dinamik Kütüphane dosyaları program çalıştırıldıktan sonra çalışma zamanı sırasında belli bir noktada da yüklenebilir. Bu işlemin bazı avantajları
26809 şunlar olabilmektedir:
26810
26811 - Program daha hızlı yüklenebilir.
26812 - Programın sanal bellek alanı gereksiz bir biçimde doldurulmayabilir. (Örneğin nadiren çalışacak bir fonksiyon dinamik kürüphanede olabilir. Bu durumda o dinamik
26813 kütüphanenin işin başında yüklenmesi gereksi bir yükleme zamanı ve bellek israfına yol açabilir.)
26814
26815 Dinamik kürüphanelerin dinamik yüklenmesi için dlopen, dlsym, dlerror ve dlclose fonksiyonlarıyla yapılmaktadır. Bu fonksiyonlarda libdl kütüphanesi
26816 içerisindeindedir. Dolayısıyla link işlemi için -ldl seçeneğinin bulundurulması gereklidir. dlopen fonksiyonun prototipi şöyledir:
26817
26818 void *dlopen(const char *filename, int flag);
26819
26820 Fonksiyonun birinci parametresi yüklenecek dinamik kütüphanenin yol ifadesini, ikinci parametresi seçenek belirten bayrakları almaktadır. Fonksiyon başarı durumunda
26821 kütüphaneyi temsil eden bir handle değerine başarısızlık durumunda NULL adresе geri dönmektedir. Başarısızlık durumunda fonksiyon errno değişkenini set etmez.

26822 Başarısızlığa ilkin yazı doğrudan dLError fonksiyonuya elde →
edilmektedir:

26823
26824 char *dLError(void);
26825
26826 dlopen fonksiyonun birinci parametresindeki dinamik kütüphane isminde →
eğer hiç / karakteri yoksa bu durumda kütüphanenin aranması daha önce →
ele aldığımız prosedüre göre yapılmaktadır. Eğer dosya isminde en az bir / karakteri →
varsı dosya yalnızca bu mutlak ya da görelı yol ifadesinde →
aranmaktadır.

26827
26828 Dinamik yükleme sırasında yüklenecek kütüphanenin SONAME alanında →
yazılan isme hiç bakılmamaktadır. (Bu SONAME alanındaki isim yalnızca →
link aşamasında kullanılır)

26829
26830 Kütüphanein adres alanından boşaltılması ise dlclose fonksiyonıyla →
yapılmaktadır:

26831
26832 int dlclose(void *handle);
26833
26834 Fonksiyon başarı durumda 0, başarısızlık durumunda 0 dışı bir değere geri →
dönmektedir. Aynı kütüphane ikinci kez yüklenebilir. Bu durumda →
gerçek bir yükleme yapılmaz.

26835 Ancak yüklenen sayıda close işleminin yapılması gerekmektedir.

26836
26837 Kütüphanein içerisindeki fonksiyonlar ya da global nesneler adresleri →
elde eiderek kullanılırlar. Bunların adreslerini elde edebilmek için →
dlsym isimli

26838 Fonksiyon kullanılmaktadır:

26839
26840 void *dlsym(void *handle, const char *symbol);
26841
26842 Fonksiyon başarı durumunda ilgili sembolün adresine başarısızlık →
durumunda NULL adrese geri döner.

26843
26844 Aşağıda bir dinamik kütüphane dinamik olarak yüklenmiş ve oradan bir →
fonksiyon ve data adresi alınarak kullanılmıştır.

26845
26846 -----*/

26847
26848 #include <stdio.h>
26849 #include <stdlib.h>
26850 #include <dlfcn.h>

26851
26852 int main(void)
26853 {
26854 void *dl;
26855 int (*padd)(int ,int);
26856 int result;
26857 int *pi;
26858
26859 if ((dl = dlopen("libsample.so", RTLD_NOW)) == NULL) {

```
26860     fprintf(stderr, "dlopen: %s\n", dlerror());
26861     exit(EXIT_FAILURE);
26862 }
26863
26864 if ((*void **)&add = dlsym(dl, "add")) == NULL) {
26865     fprintf(stderr, "dlsym: %s\n", dlerror());
26866     exit(EXIT_FAILURE);
26867 }
26868
26869 result = add(10, 20);
26870 printf("%d\n", result);
26871
26872 if ((pi = (int *)dlsym(dl, "x")) == NULL) {
26873     fprintf(stderr, "dlsym: %s\n", dlerror());
26874     exit(EXIT_FAILURE);
26875 }
26876 printf("%d\n", *pi);
26877
26878 dlclose(dl);
26879
26880 return 0;
26881 }
26882
26883 /
*-----*
```

-----*

26884 Dinamik kütüphane `dlopen` fonksiyonuyla yüklenirken global değişkenlerin ve fonksiyonların nihai yükleme adresleri bu `dlopen` işlemi sırasında hesaplanabilir ya da onlar kullanıldıklarında hesaplanabilir. İki arasında kullanıcı açısından bir fark olmamakla birlikte tüm sembollerin adreslerinin yükleme sırasında hesaplanması bazen yükleme işlemini (eğer çok simbol varsa) uzatabilmektedir. Bu durumu ayarlamak için `dlopen` fonksiyonun flags parametresi kullanılır. Bu parametre `RTLD_NOW` olarak girilirse tüm sembollerin adresleri `dlopen` sırasında, `RTLD_LAZY` girilirse kullanıldığı noktada hesaplanmaktadır. İki biçim arasında çoğu kez programcı için bir farklılık oluşmamaktadır. Ancak aşağıdaki örnekte bu iki biçimin ne anlama geldiği gösterilmektedir.

26889 Aşağıdaki örnekte `libmample.so` kütüphanesindeki `foo` fonksiyonu gerçekten olmayan bir `bar` fonksiyonunu çağrırmıştır. Bu fonksiyonun gerçekten olmadığı `foo` fonksiyonun simbol çözümlemesi yapıldığında anlaşılmacaktır. İşte eğer bu kütüphaneyi kullanan `sample.c` programı kütüphaneyi `RTLD_NOW` ile yüklerse tüm semboller o anda çözülmeye çalışılacağından dolayı `bar`'ın bulunmuyor olması hatası da `dlopen` sırasında oluşacaktır. Eğer kütüphane `RTLD_LAZY` ile yüklenirse bu durumda simbol çözümlemesi `foo`'nun kullanıldığı noktası (yani `dlsym` fonksiyonunda) gerçekleşecektir. Dolayısıyla hata da o o

noktada oluşacaktır.

```
26895 -----*  
26896  
26897 /* mample.c */  
26898  
26899 #include <stdio.h>  
26900  
26901 void bar(void);  
26902  
26903 void foo(void)  
26904 {  
26905     bar();  
26906 }  
26907  
26908 /* sample.c */  
26909  
26910 #include <stdio.h>  
26911 #include <stdlib.h>  
26912 #include <dlsfcn.h>  
26913  
26914 int main(void)  
26915 {  
26916     void *dl;  
26917     void (*pf)(void);  
26918  
26919     if ((dl = dlopen("libmample.so", RTLD_LAZY)) == NULL) { /* RTLD_NOW ile de deneyiniz */  
26920         fprintf(stderr, "dlopen: %s\n", dlerror());  
26921         exit(EXIT_FAILURE);  
26922     }  
26923  
26924     printf("Ok\n");  
26925  
26926     if ((*void **)&pf = dlsym(dl, "foo")) == NULL) {  
26927         fprintf(stderr, "dlsym: %s\n", dlerror());  
26928         exit(EXIT_FAILURE);  
26929     }  
26930  
26931     pf();  
26932  
26933     dlclose(dl);  
26934  
26935     return 0;  
26936 }  
26937  
26938 /-----*
```

Bazen bir dinamik kütüphane içerisindeki sembollerin o dinamik kütüphaneyi kullanan kodlar tarafından kullanılması istenmeyebilir. Örneğin dinamik kütüphanede bar isimli bir fonksiyon vardır. Bu fonksiyon bu dinamik kütüphanenin kendi içerisindeki başka fonksiyonlar

tarafından ↗
26941 kullanılıyor olabilir. Ancak bu fonksiyonun dinamik kütüphanenin ↗
 dışından kullanılması istenmeyebilir. (Bunun çeşitli nedenleri ↗
 olabilir. Örneğin ↗
26942 kapsülleme sağlamak için, dışarıdaki simbol çakışmalarını ortadan ↗
 kaldırmak için vs.) İşte bunu sağlamak amacıyla gcc'ye özgü ↗
 __attribute__((...)) ↗
26943 eklentisinden faydalılmaktadır. __attribute__((...)) eklentisi pek çok ↗
 seçeneğe sahip platform spesifik bazı işlevlere yol açmaktadır. Bu ↗
 eklentinin ↗
26944 seçeneklerini gcc dokümanlarından elde edbilirsiniz. Bizim bu amaçla ↗
 kullanacağımız __attribute__((...)) seçeneği "visibility" isimli ↗
 seçenektir. ↗
26945
26946 Aşağıdaki örnekte bar fonksiyonu foo fonksiyonu tarafından ↗
 kullanılmaktadır. Ancak kütüphanenin dışından bu fonksiyonun ↗
 kullanılması istenmemiştir. ↗
26947
26948 Burada fonksiyon özelliğinin (yani __attribute__ sentaksının) fonksiyon ↗
 isminin hemen sola getirildiğine ve çift parantez kullanıldığına ↗
 dikkat ediniz. ↗
26949 Bırada kullanılan özellik "visibility" isimli özelliktir ve değeri ↗
 "hidden" biçimde verilmiştir. ↗
26950
26951 Default durumda dinamik kütüphanedeki bütün global semboller dışarıdan ↗
 kullanılabilmektektir. Buradaki __attribute__((visibility("hidden"))) ↗
 özellikinin ↗
26952 fonksiyonu static yapmakla aynı şey olmadığına dikkat ediniz. Fonksiyon ↗
 static yapılrsa o dinamik dinamik kütüphanedeki diğer modüller ↗
 tarafından da kullanılamamaktadır. ↗
26953
26954 -----*/ ↗
26955
26956 /* mample.c */
26957
26958 #include <stdio.h>
26959
26960 void bar(void);
26961
26962 void foo(void)
26963 {
26964 printf("foo\n");
26965 bar();
26966 }
26967
26968 void __attribute__((visibility("hidden"))) bar(void)
26969 {
26970 printf("bar\n");
26971 }
26972
26973 /
 -----*/ ↗

```
-----  
26974     Bir dinamik kütüphane normal olarak ya da dinamik olarak yükleniğinde ↵  
26975         birtakım ilk işlerin yapılması gerekebilir. (Örneğin ↵  
26976         kütüphane thread güvenli olma iddiasındadır ve birtakım senkronizasyon ↵  
26977             nesnelerinin ve thread'e özgü alanların yaratılması gerekebilir.) ↵  
26978     Bunun için __attribute__((constructor)) fonksiyon özelliği ↵  
26979         kullanılmaktadır. Benzer biçimde dinamik kütüphane programın adres ↵  
26980             alanından ↵  
26981     boşaltılırken de birtakım son işlemler için __attribute__((destructor)) ↵  
26982         ile belirtilen fonksiyon çağrılmaktadır. (Aslında bu constructor ve ↵  
26983         destructor fonksiyonları normal programlarda da kullanılabilir. Bu durumda ilgili ↵  
26984         fonksiyonlar main fonksiyonundan önce ve main fonksiyonundan sonra ↵  
26985             çağrılmaktadır.) ↵  
26986 -----*/  
26983 /* mample.c */  
26984  
26985  
26986 #include <stdio.h>  
26987  
26988 void __attribute__((constructor)) dynamic_init()  
26989 {  
26990     printf("dinamik kütüphane kullanılıyor..\n");  
26991 }  
26992  
26993 void __attribute__((destructor)) dynamic_exit()  
26994 {  
26995     printf("dinamik kütüphane boşaltılıyor...\n");  
26996 }  
26997  
26998 void foo(void)  
26999 {  
27000     printf("foo\n");  
27001 }  
27002  
27003 /* sample.c */  
27004  
27005 #include <stdio.h>  
27006 #include <stdlib.h>  
27007 #include <dlfcn.h>  
27008  
27009 int main(void)  
27010 {  
27011     void *dl;  
27012  
27013     if ((dl = dlopen("libmample.so", RTLD_NOW)) == NULL) {  
27014         fprintf(stderr, "dlopen: %s\n", dlerror());  
27015         exit(EXIT_FAILURE);
```

```
27016     }
27017
27018     printf("ok\n");
27019
27020     if ((dl = dlopen("libmample.so", RTLD_NOW)) == NULL) {
27021         fprintf(stderr, "dlopen: %s\n", dlerror());
27022         exit(EXIT_FAILURE);
27023     }
27024
27025     dlclose(dl);
27026
27027     return 0;
27028 }
27029
27030 / *
-----*
27031     O anda makinemizdeki işletim sistemiındaki bilgi uname komutuyla
27032     elde edilebilir. Bu komut -r ile kullanılırsa o makinede yüklü olan
27033     kernek versiyonu elde edilmektedir. Örneğin:
27034
27035     uname -r
27036     4.15.0-20-generic
27037 -----*/
27038 /
-----*
27039     Kernel'in bir parçası gibi işlev gören, herhangi bir koruma engeline
27040     takılmayan, kernel modda çalışan özel olarak hazırlanmış modüllere
27041     (yani kod parçalarına) Linux dünyasında "çekirdek modülleri (kernel
27042     modules)" denilmektedir. Çekirdek modülleri eğer kesme gibi bazı
27043     mekanizmaları
27044     kullanıyorsa ve bir donanım aygitini yönetme iddiasındaysa bunlara özel
27045     olarak "aygit sürücülerleri (device drivers)" denilmektedir.
27046
27047     Genellikle bir Linux sistemini yüklediğimizde zaten kernel modüllerini
27048     ve aygit sürücülerini oluşturabilmek için gereken kütüphaneler ve başlık
27049     dosyaları
27050     zaten yüklü biçimde bulunmaktadır. Tabii programcı kernel kodlarını da
27051     kendi makinesine indirmek isteyebilir. Bunun için aşağıdaki komut
27052     kullanılabilir:
27053
27054     sudo apt-get install linux-source
27055
27056     Eğer sisteminizde Linux'un kaynak kodları yüklü ise bu kaynak kodlar /usr/src
27057     dizininde bulunmaktadır. Bu dizindeki linux-headers-$(uname -r)
27058     dizini kaynak kodlar yüklü olmasa bile bulunan bir dizindir ve bu dizin
27059     çekirdek modülleri ve aygit sürücülerin "build edilmeleri" için
27060     gereken başlık dosyalarını
27061     barındırmaktadır. Benzer biçimde /lib/modules isimli dizinde $(uname -r)
```

isimli bir dizin vardır. Bu dizin çekerek modüllerinin build edilmesi için gereken bazı kodları içингereken bazı kodları
ve Kütüphaneleri bulundurmaktadır.

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

Bir kernel modülünde biz user mod için yazılmış kodları kullanamayız. Çünkü orası bir dünyadır. Kernnel modüllerinde biz yalnızca kernel içerisindeki bazı fonksiyonları kullanabiliriz. Bunlara "kernel tarafından export edilmiş fonksiyonlar" denilmektedir.
"Kernel tarafından export edilmiş fonksiyon kavramıyla "sistem fonksiyonu" kavramının bir ilgisi yoktur. Sistem fonksiyonları user moddan çağrılmak üzere tasarlanmış ayrı bir grup fonksiyondur. Oysa kernek tarafından export edilmiş fonksiyonlar user moddan çağrılamazlar. Yalnızca kernel modellerinden çağrılabılırler. Buradan çıkan sonuç şudur: Bir kernel modül yazılırken ancak kernel'in export ettiği fonksiyon ve datalar kullanılabilir. Kernel kaynak kodları çok büyktür buradaki kısıtlı sayıda fonksiyon export edilmiştir. Benzer biçimde programcının oluşturduğu bir kernel modül içerisindeki belli fonksiyonları da programcı export edebilir. Bu duurmda bu fonksiyonlar da başka kernel modüllerinden kullanılabilirler.

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

Mademki kernel modüller işletim sisteminin kernel kodlarındaki fonksiyon ve dataları kullanabiliyorlar o zaman kernel modüller o anda çalışılan kernel'in yapısına da bağlı durumdadırlar. İşletim sistemlerinde "aygı sürücü yazmak" ya da "kernel modül yazmak" biçiminde genel bir konu yoktur.
Her işletim sisteminin kernel modül ve aygit sürücü mimarisi diğerlerinden farklıdır. Dolayısıyla bu konu spesifik bir işletim sistemi için geçerli olabilecek oldukça platform bağımlı bir konudur. Hatta işletim sistemlerinde bazı versiyonlarda genel aygit sürücü mimarisi bile değiştirilebilmektedir. Dolayısıyla eski aygit sürücülerini yeni versiyonlarda çalışmamakta yenileri de eski versiyonlarda çalışmamaktadır. İşletim sistemlerinin yeni versiyonlarında kernel mimarisi değiştirilmiş olabilmektedir. Dolayısıyla daha farklı kernel fonksiyonları eskilerinin yerlerini almış olabilir. Yeni birtakım başka fonksiyonlar export edilmiş olabilir.

```
-----*/
27073
27074 /
27075 *-----*
-----*
27075 Bir kernel modülünü derlemek ve link etmek maalesef sanıldığından daha →
27076 zordur. Herne kadar kernel modüller de ELF object dosları →
27076 iseler de bunlarda özel bazı bölmeler (sections) bulunmaktadır. →
27077 Dolayısıyla bu modillerin derlenmesinde özel gcc seçenekleri devreye →
27077 sokulur.
27078 Bunların link edilmeleri de bazı kütüphane dosyalarının devreye →
27078 sokulmasıyla yapılır. Dolayısıyla bir kernel modülün "build edilmesi" →
27078 biraz
27078 ayrıntılı bilgi gerektirmektedir. İşte kernel tasarımcıları bu sıkıcı →
27078 işlemleri kolaylaştırmak için özel "make dosyaları" düzenlemişlerdir. →
27078 Programcı
27079 bu make dosyalarından faydalananarak build işlemini çok daha kolay →
27079 yapabilmektedir.
27080
27081 Kernel modüller için build işlemini yapan örnek bir Makefile aşağıdaki →
27081 gibi olabilir. Burada önce /lib/modules/$(uname -r)/builf dizinindeki →
27081 Makefile
27082 çalıştırılmış onadan sonra çalışma bu dosyadan devam ettirilmiştir. →
27082 Özetle b Make dosyası "helloworld.c" isimli dosyanın derlenerek kernel →
27082 modül biçimind e link
27083 edilmesini sağlamaktadır. Kernel modül birden fazla kaynak dosyadan →
27083 oluşturulabilir. Bu durumda obj-m += a.o b.o c.o... biçiminde tüm →
27083 dosyalar belirtilir. Ya da bunlar
27084 teker teker şöyle de belirtilebilir:
27085
27086 obj-m += a.o
27087 obj-m += b.o
27088 obj-m += c.o
27089 ...
27090 -----*/
27091
27092 # Makelfile
27093
27094 obj-m += helloworld.o
27095
27096 all:
27097     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
27098 clean:
27099     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
27100
27101 /
27102 *-----*
-----*
27102 Tabii aslında make dosyası parametrik biçimde de oluşturabilmektedir. →
27102 Tabii bu durumda make programı çalıştırılıken bu parametrenin değeri →
27102 de
27103 belirtilmelidir. Örneğin:
```

```
27104
27105     make file=helloworld
27106
27107 -----*/  
27108
27109 # Makelfile
27110
27111 obj-m += $(file).o
27112
27113 all:
27114     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
27115 clean:
27116     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
27117
27118 /
*-----*/  
27119     Tipik bir "helloworld" modülü şöyle oluşturulabilir. Bu modülü  
yukarıdaki make dosyası ile aşağıdaki gibi build yapmalısınız:  
27120
27121     make file=helloworld
27122
27123     Kernel modüllerin yüklenmesi için insmod isimli program kullanılmaktadır. Tabii bu program sudo ile çalıştırılmalıdır. Örneğin:  
27124
27125     sudo insmod helloworld.ko
27126
27127     Kernel modüller istenildiği zaman rmmod isimli programla  
boşaltılabilirler. Örneğin:  
27128
27129     sudo rmmod helloworld.ko
27130
27131 -----*/
27132
27133 /* helloworld.c */
27134
27135 #include <linux/module.h>
27136 #include <linux/kernel.h>
27137
27138 int init_module(void)
27139 {
27140     printk(KERN_INFO "Hello World...\n");
27141
27142     return 0;
27143 }
27144
27145 void cleanup_module(void)
27146 {
27147     printk(KERN_INFO "Goodbye World...\n");
27148 }
27149
```

```
27150 # Makelfile
27151
27152 obj-m += $(file).o
27153
27154 all:
27155     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
27156 clean:
27157     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
27158
27159 /
*-----  
-----  
27160 En basit bir kernel modülde aşağıdaki iki temel dosya include  
edilmelidir:  
27161
27162 #include <linux/module.h>
27163 #include <linux/kernel.h>
27164
27165 Bu iki dosya /lib/modules/$(uname -r)/build/include dizini içerisindedir. (Yani libc ve POSIX kütüphanelerinin başlık dosyalarının bulunduğu /usr/include içerisinde değildir.) Yukarıda açıklanan Make dosyası include dosyalarının bu zidinde aranmasını sağlamaktadır. Bir kernel modül yüklenliğinde
27166 kernel modül içerisinde belirlenmiş olan bir fonksiyon çağrılır (bu "constructor" gibi düşünülebilir.) Default çağrılabilecek fonksiyonun ismi init_module biçimindedir. Bu fonksiyonun geri dönüş değeri int türdendir ve parametresi yoktur. Fonksiyon başarı durumunda 0 değerine başarısızlık
27167 durumunda negatif hata koduna geri dönmelidir. Bu fonksiyon başarısızlıkla geri dönerse modülün yüklenmesinden vazgeçilmektedir. Benzer biçimde bir modül kernel alanından boşaltıldığında de yine bir fonksiyon çağrılmaktadır. (Bu fonksiyon "destructor" gibi düşünülebilir.) Default çağrılabilecek fonksiyonun ismi cleanup_module biçimindedir.
27168 Bu fonksiyonun geri dönüş değeri ve parametresi void biçimdedir.
27169
27170 Kernel modüller típkí deamon'lar gibi ekrana değil log dosyalarına ileri yazarlar. Bunun için kernel içindeki printk isimli fonksiyon kullanılmaktadır.
27171 printk fonksiyonun genel kullanımı printf gibidir. Default durumda yeni kernel'larda bu fonksiyon mesajların /var/log/syslog dosyasına yazdırılması sağlanmaktadır.
27172 printk fonksiyonun prototipi <linux/kernel.h> dosyası içerisindeindedir.
27173
27174 printk fonksiyonun örnek kullanımı şöyledir:
27175
27176 printk(KERN_INFO "This is test\n");
27177
27178 Mesajın solundaki KERN_XXX biçimindeki makrolar aslında bir string açımı yapmaktadır. Dolayısıyla yan yana iki string birleştirildiği için mesaj yazısının başında
27179 küçük bir alan bulunur. Bu alan (yani bu makro) mesajın türünü ve aciliyetini belirtmektedir. Tipik KERN_XXX makroları şunlardır:
```

```
27183
27184 KERN_EMERG
27185 KERN_ALERT
27186 KERN_CRIT
27187 KERN_ERR
27188 KERN_WARN
27189 KERN_NOTICE
27190 KERN_INFO
27191 KERN_DEBUG
27192
27193 Bu makroların tipik biçimini söyledirir:
27194
27195 #define KERN_EMERG    "<0>"
27196 #define KERN_ALERT    "<1>"
27197 #define KERN_CRIT    "<2>"
27198 #define KERN_ERR     "<3>"
27199 #define KERN_WARNING  "<4>"
27200 #define KERN_NOTICE   "<5>"
27201 #define KERN_INFO    "<6>"
27202 #define KERN_DEBUG   "<7>"
27203
27204 Aslında KERN_XXX makroları ile printk fonksiyonunu kullanmak yerine pr_XXX →
     makrolarında kullanılabilir. Şöyledir ki:
27205
27206 printk(KERN_INFO "Hello World...\n");
27207
27208 ile
27209
27210 pr_info("Hello World...\n");
27211
27212 tamamen eşdeğerdir. Aşağıdaki makrolar bulunmaktadır:
27213
27214 pr_emerg
27215 pr_alert
27216 pr_crit
27217 pr_err
27218 pr_warning
27219 pr_notice
27220 pr_info
27221 pr_debug
27222
27223
27224 printk fonksiyonun yazdıklarını /var/log/syslog dosyasına bakarak →
     görebiliriz. Örneğin:
27225
27226 tail /var/log/syslog
27227
27228 Ya da dmesg programı ile de aynı bilgi elde edilebilir.
27229
27230 Belli bir anda yüklenmiş olan modüller /proc/modules dosyasından elde →
     edilebilir. Aslında lsmod isimli bir program daha vardır. lsmod
27231 zaten /proc/modules dosyasını okuyarak sonu basitleştirmektedir.
27232 ----- →
```

```
-----*/
27233 /
27234 /*
27235 *-----*
27236 -----*
27237 Aslında init_module ve cleanup_module fonksiyonlarının ismi
27238 değiştirilebilir. Fakat bunun için bildirimde bulunmak gereklidir.
27239 Bildirimde bulunmak için ise module_init(...) ve module_exit(...)
27240 makroları kullanılmaktadır. Bu makrolar kaynak kodun herhangi bir
27241 yerine
27242 yazılabilir. Ancak makro içerisinde belirtilen fonksiyonların
27243 bildirimlerinin bu makroların yerleştiği yere kadar yapılmış olması
27244 gerekmektedir.
27245 Bu makrolar tipik olarak kaynak kodun sonuna yerleştirilmektedir.
27246 -----
27247 */
27248 /* helloworld.c */
27249
27250 #include <linux/module.h>
27251 #include <linux/kernel.h>
27252
27253 int helloworld_init(void)
27254 {
27255     printk(KERN_INFO "Hello World...\n");
27256     return 0;
27257 }
27258
27259 void helloworld_exit(void)
27260 {
27261     printk(KERN_INFO "Goodbye World...\n");
27262 }
27263
27264 module_init(helloworld_init);
27265 module_exit(helloworld_exit)
27266
27267 # Makelfile
27268
27269 obj-m += $(file).o
27270
27271 all:
27272     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
27273 clean:
27274     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
27275
27276 /
27277 *-----*
27278 -----*
27279 Genellikle kernel modül içerisindeki global değişkenlerin ve
27280 fonksiyonlarının "internal linkage" yapılması tercih edilmektedir.
27281 -----*
```

```
27273
27274 /* helloworld.c */
27275
27276 #include <linux/module.h>
27277 #include <linux/kernel.h>
27278
27279 static int helloworld_init(void)
27280 {
27281     printk(KERN_INFO "Hello World...\n");
27282
27283     return 0;
27284 }
27285
27286 static void helloworld_exit(void)
27287 {
27288     printk(KERN_INFO "Goodbye World...\n");
27289 }
27290
27291 module_init(helloworld_init);
27292 module_exit(helloworld_exit);
27293
27294
27295 # Makelfile
27296
27297 obj-m += $(file).o
27298
27299 all:
27300     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
27301 clean:
27302     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
27303
27304 /
*-----*
-----*
27305     Kernel modüllerde başlangıç ve bitiş fonksiyonlarında fonksiyon      ↗
27306         isimlerinin soluna __init ve __exit makroları getirilebilmektedir.      ↗
27307     Bu makrolar <linux/init.h> dosyası içerisindeindedir. __init ilgili      ↗
27308         fonksiyonu özel bir ELF bölümüne (section) yerleştirir. Modül      ↗
27309         yüklenikten sonra
27310         bu bölüm kernel alanından atılmaktadır. __exit makrosu ise kernel'in      ↗
27311         içine gömülü modüllerde fonksiyonun dikkate alınmayacağı      ↗
27312         (dolayısıyla
27313         hiç yüklenmeyeceğini) belirtir. Ancak sonradan yüklemelerde bu makronun      ↗
27314         bir etkisi yoktur.
27315 -----*/
```

```
27317 {
27318     printk(KERN_INFO "Hello World...\n");
27319
27320     return 0;
27321 }
27322
27323 static void __exit helloworld_exit(void)
27324 {
27325     printk(KERN_INFO "Goodbye World...\n");
27326 }
27327
27328 module_init(helloworld_init);
27329 module_exit(helloworld_exit);
27330
27331 # Makelfile
27332
27333 obj-m += $(file).o
27334
27335 all:
27336     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
27337 clean:
27338     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
27339
27340 /
*-----*
-----*
27341 insmod ile yüklediğimiz her modül için /sys/modules dizinin içerisinde     ↵
      ismi modül ismiyle aynı olan bir dizin yaratılmaktadır.             ↵
27342 /proc/modules dosyası ile bu dizini karıştırmayınız. /proc/modules       ↵
      dosyasının satırları yüklü olan modüllerin isimlerini tutar.           ↵
27343 Modüllere ilkinin asıl önemli bilgiler kenel tarafından /sys/modules     ↵
      dizininde tutulmaktadır.                                              ↵
27344 -----*/
27345
27346 /
*-----*
-----*
27347 Modüllere parametre geçirebiliriz. Parametre geçirme işlemi insmod ile     ↵
      modül yüklenirken komut satırında modül isminden sonra             ↵
      değişken=değer çiftleriyle yapılmaktadır. Örneğin:                 ↵
27348
27349 sudo insmod ./helloworld number=10 msg=\"This is a test\""
27350         values=10,20,30,40,50
27351
27352 Kernel modüllere bu biçimde değer aktarmak için module_param ve        ↵
      module_param_array isimli makrolar kullanılır. Makronun üç parametresi ↵
27353 vardır:
27354
27355 module_param(name, type, perm)
27356
27357 name parametresi ilgili değişkenin ismini belirtmektedir. Bu ismin komut ↵
```

satırındaki argüman ismiyle aynı olması gereklidir. type ilgili parametrenin türünü belirtir. Bu tür şunlardan biri olabilir: int, long, short, uint, ulong, ushort, charp, bool, invbool. perm ise /sys/modules/<modül ismi> dizininde yaratılacak olan parameters dizininin erişim haklarını belirtir. Bu makrolar global alanda herhangi bir yere yerleştirilebilir.

27361

27362 module_param_array makrosu da şöyledir:

27363

```
27364 define module_param_array(name, type, nump, perm)
```

27365

27366 Makronun birinci ve ikinci parametreleri yine değişken ismi ve türüdür Tabii buradaki değişken isminin bir dizi ismi olarak girilmesi gerekmektedir.

27367 Üçüncü parametre toplam kaç değerin modüle dizi biçiminde aktarıldığını belirten değişkenin adresini (ismini değil) alır. Son parametre yine erişim haklarını belirtmektedir.

27368

27369 Aşağıdaki örnekte üç parametre komut satırından kernel module'ye geçirilmiştir. Komut satırındaki isimlerle programın içerisindeki değişken isimlerinin aynı olması gerekmektedir. Yazıların geçirilmesinde iki tırnaklar kullanılır. Ancak kabuk programının söz konusu yazıyı tek bir parametre olarak ele alabilmesi için yazının dışarıdan da tırnaklanması gerekmektedir. Kabuktaki iki tırnak ile tek tırnak arasında yalnızca küçük bir farklılık vardır. Dizi geçirirken yanlışlıkla boşluk karakteri kullanılmamalıdır.

27370

27371

27372

27373

27374

27375 Aşağıdaki programı şöyle bir örnekle çalıştırabilirsiniz:

27376

```
27377 sudo insmod helloworld.ko count=100 msg='this is a test' values=10,20,30,40,50
```

27378

27379 -----*/

27380

```
27381 /* helloworld.c */
```

27382

```
27383 #include <linux/module.h>
```

```
27384 #include <linux/kernel.h>
```

```
27385 #include <linux/moduleparam.h>
```

27386

```
27387 static int count;
```

```
27388 static char *msg;
```

```
27389 static int values[5];
```

```
27390 static int size;
```

27391

```
27392 module_param(count, int, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
```

```
27393 module_param(msg, charp, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
```

```
27394 module_param_array(values, int, &size, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
```

27395

```
27396 static int __init helloworld_init(void)
```

27397 {

```
27398     int i;
27399
27400     printk(KERN_INFO "Hello World...\n");
27401     printk(KERN_INFO "count = %d, msg = %s\n", count, msg);
27402
27403     printk(KERN_INFO "values:\n");
27404     for (i = 0; i < size; ++i)
27405         printk(KERN_INFO "%d\n", values[i]);
27406
27407     return 0;
27408 }
27409
27410 static void __exit helloworld_exit(void)
27411 {
27412     printk(KERN_INFO "Goodbye World...\n");
27413 }
27414
27415 module_init(helloworld_init);
27416 module_exit(helloworld_exit);
27417
27418 # Makelfile
27419
27420 obj-m += $(file).o
27421
27422 all:
27423     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
27424 clean:
27425     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
27426
27427 /
*-----*
-----*
27428     Her kernel modül parametresi /sys/modules/<modül_ismi>/parameters      ↵
27429         dizinin içerisinde bir dosyayla temsil edilir. Örneğin yukarıdaki      ↵
27430         kodda biz bu dizin içerisinde count, msg ve values isimli üç dosya      ↵
27431         görürüz. Zatem module_param ve module_param_array makrolarındaki      ↵
27432         erişim hakları bu dosyaların erişim haklarıdır.
27433 -----
*-----*/
27434     errno değişkeni aslında libc kütüphanesinin (standart C ve POSIX      ↵
27435         kütüphanesi) içerisinde tanımlanmış bir değişkendir.      ↵
27436     Kernel modda yani kernel'ın içerisinde errno isimli bir değişken yoktur.      ↵
27437         Bu nedenle kerneldaki fonksiyonlar POSIX fonksiyonları gibi      ↵
27438         başarısızlık durumunda -1 ile geri dönmezler. Başarısızlık durumunda      ↵
27439         negatif errno değeri ile geri dönerler. Örneğin open POSIX      ↵
27440         fonksiyonu sys_open kernel fonksiyonunu çağrıduğunda onun negatif bir      ↵
27441         değerler geri dönüp dönmediğine bakar. Eğer sys_open      ↵
27442         negatif errno değeriyle geri dönerse bu durumda bu değerin pozitiflisini      ↵
27443         errno değişkenine yerleştirip open -1 ile geri dönmektedir.
```

27439 Kernel modül yazan programcılarında bu konvansiyona uyması iyi bir tekniktir. Örneğin:

27440 if (işler ters gitti)
27441 return -EXXXX;

27443

27444 POSIX arayüzünde adrese geri dönen fonksiyonlar genel olarak başarısızlık durumunda NULL adrese geri dönmemektedir. Oysa kernel kodlarında

27445 adrese geri dönen fonksiyonlar başarısız olduklarında yine sanki bir adresmiş gibi negatif erro değerine geri dönmemektedir.

27446

27447 Kernel kodlarındaki ERR_PTR isimli makro bir tamsayı değeri alıp onu adres türüne dönüştürmektedir. Bu nedenle adrese geri dönen fonksiyonlarda şöylesi kodlar görülebilir:

27449

27450 void *foo(void)
27451 {
27452 ...
27453 if (işler ters gitti)
27454 return ERR_PTR(-EXXXX);
27455 ...
27456 }

27457

27458 Bu işlemin tersi de PTR_ERR makrosuyla yapılmaktadır. Yani PTR_ERR makrosu bir adresi alıp onu tamsayıya dönüştürmektedir. Bu durumda

27459 kernel kodlarında adrese geri dönen fonksiyonların başarısızlığını aşağıdaki gibi kodlarla kontrol edilmektedir:

27460

27461 void *ptr;
27462
27463 ptr = foo();
27464 if (PTR_ERR(ptr) < 0)
27465 return PTR_ERR(ptr)

27466

27467 Kernel modül programlarının da buradaki konvansiyona uygun kod yazması iyi bir tekniktir.

27468

27469 Linux çekirdeğindeki EXXX sembolik sabitleri POSIX arayüzündeki EXXX sabitleriyle aynı değerdedir.

27470 -----*/

27471 /

27472 *

27473 -----

27473 Linux'ta bir kernel modül artık user mod'tan kullanılabilir hale getirildiye buna "aygit sürücü (device driver)" denir.

27474 Aygit sürücüler open fonksiyonuyla bir dosya gibi açılırlar. Bu açma işleminden bir dosya betimleyicisi elde edilir. Bu dosya betimleyicisi read, write, lseek, close gibi fonksiyonlarda kullanılabilir. Aygit sürücülere ilişkin dosya betimleyicileri bu fonksiyonlarla kullanıldığındá aygit sürücü içerisindeki belirlenen

27477 bazı fonksiyonlar çağrılmaktadır. Yani tersten gidersek biz ↵
27478 örneğin read fonksiyonu çağrıldığında aygit sürücümüzdeki belli bir ↵
fonksiyonun çalışmasını sağlayabiliriz. Böylece aygit sürücü ile ↵
27478 user mod arasında veri transferleri yine sanki dosyamış gibi read, ↵
write gibi fonksiyonlarla yapılır. Pekiyi user moddan aygit ↵
sürücümüzdeki ↵
27479 herhangi bir fonksiyonu çağrılabilir miyiz? Yanıt evet. Bunun için ioctl ↵
isimli POSIX fonksiyonu kullanılmaktadır. Aygit sürücü içerisinde ↵
27480 fonksiyonlara birer kod numarası atanır. Sonra ioctl fonksiyonunda bu ↵
kod numarası belirtilir. Böylece akış user mod'tan kernel moda ↵
27481 geçerek belirlenen fonksiyonu kernel modda çalıştıracaktır.

27482

27483 Pekiyi aygit sürücülerini açmak için open fonksiyonunda yol ifadesi olarak ↵
(yani dosya ismi olarak) ne verilecektir? İşte aygit sürücüler ↵
27484 dosya sisteminde bir dizin girişile temsil edilir. O dizin girişini open ↵
ile açıldığında aslında o dizin girişinin temsil ettiği aygit sürücü ↵
27485 açılmış olur. İşte aygit sürücülerini temsil eden dizin girişlerine "aygit ↵
dosyaları (device files)" denilmektedir.

27486

27487 Pekiyi bir aygit dosyası nasıl bir aygit sürücüyü temsil eder hale ↵
getirilmektedir ve nasıl yaratılmaktadır? İşte her aygit sürücünün ↵
27488 majör ve minör numaraları vardır. Aynı zamanda aygit dosyalarının da ↵
majör ve minör numaraları vardır. Bir aygit sürücünün majör ve ↵
27489 minör numarası bir aygit dosyasının majör ve minör numarasıyla aynıysa ↵
bu durumda o aygit dosyası o aygit sürücüyü temsil eder.

27490 Aygit dosyaları özel dosyalardır. Bir dosyanın aygit dosyası olup ↵
olmadığı ls -l komutunda dosya türü olarak 'c' (karakter aygit ↵
sürücüsü)

27491 ya da 'b' (blok aygit sürücüsü) ile temsil edilir. Anımsanacağı gibi ↵
dosya bilgileri stat, fstat, lstat fonksiyonlarıyla elde edilmektedir.

27492 İşte struct stat yapısının dev_t türünden st_rdev elemanı eğer dosya bir ↵
aygit dosyasısa dosyanın majör ve minör numaralarını belirtir.

27493 Tabii biz <sys/stat.h> dosyasındaki S_ISCHR ve S_ISBLK makrolarıyla bunu ↵
öğrenebiliriz.

27494

27495 O halde şimdi bizim bir aygit dosyasını nasıl oluşturacağımızı ve aygit ↵
sürücüye nasıl majör ve minör numara atayacağımızı bilmemiz ↵
27496 gerekir.

27497

27498 -----*/

27499 /

27500 *

27501 Aygit dosyaları mknod isimli POSIX fonksiyonuyla (bu fonksiyon sistem ↵
fonksiyounu çağrılmaktadır) ya da komut satırından mknod ↵
27502 komutuyla (bu komut da mknod fonksiyonu ile işlemini yapmaktadır) ↵
yaratılabilir. mknod fonksiyonun prototipi şöyledir:

27503 int mknod(const char *pathname, mode_t mode, dev_t dev);

27505 Fonksiyonun birinci parametresi yaratılacak aygit dosyasının yol ↵

ifadesini, ikinci parametresi erişim haklarını ve üçüncü parametresi de ↵
27507 aygit dosyasının majör ve minör numarasını belirtmektedir. Aygit ↵
dosyasının majör ve minör numaraları user mod programda makedev ↵
makrosuyla
27508 tek bir dev_t nesnesi haline getirilebilmektedir. Bir dev_t nesnesinin ↵
icerisindeki numaralardan majör olani major makrosuyla, minör olani ↵
27509 ise minor makrosuyla elde edebiliriz.
27510
27511 dev_t makedev(unsigned int maj, unsigned int min);
27512 unsigned int major(dev_t dev);
27513 unsigned int minor(dev_t dev);
27514
27515 Ancak kernel modda bu makrolar yerine aşağıdakiler kullanılmaktadır:
27516
27517 MKDEV(major, minor)
27518 MAJOR(dev)
27519 MINOR(dev)
27520
27521 Linux'ta son versiyonlar da dikkate alındığında dev_t 32 bitlik ↵
işaretsiz bir tamsayı türündendir. Bu 32 bitin yüksek anlamlı 10 biti ↵
majör numarayı,
27522 düşük anlamlı 20 biti ise minör numarayı temsil etmektedir. Ancak ↵
programci bu varsayımlarla kodunu düzenlememeli yukarıda belirtilen ↵
makroları
27523 kullanmalıdır.
27524
27525 mknod fonksyonun ikinci parametresindeki erişim haklarına ayrıca eğer ↵
karakter aygit sürücü dosyası yaratılmak isteniyorse S_IFCHR, blok ↵
27526 aygit sürücü dosyası yaratılmak isteniyorsa S_IFBLK bayraklarını eklemek ↵
gerekmektedir. Tabii mknod fonksyonun başarılı olabilmesi için
27527 ilkgili prosesin proses id'sinin 0 olması gereklidir.
27528
27529 Aşağıdaki aygit dosyası yaratan örnek bir program görülmektedir.
27530
27531 -----*/
27532
27533 /* mymknod.c */
27534
27535 #include <stdio.h>
27536 #include <stdlib.h>
27537 #include <string.h>
27538 #include <sys/stat.h>
27539 #include <sys/sysmacros.h>
27540
27541 int is_octal(const char *str);
27542 void exit_sys(const char *msg);
27543
27544 /* mymknod <erisim hakları> <aygit cinsi> <majör numara> <minör numara> ↵
<dosya ismi> */
27545
27546 int main(int argc, char *argv[])

```
27547 {
27548     int i;
27549     int mode;
27550     mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, ↵
27551         S_IROTH, S_IWOTH, S_IXOTH};
27552     mode_t result_mode;
27553     unsigned major, minor;
27554
27555     if (argc != 6) {
27556         fprintf(stderr, "wrong number of arguments!..\n");
27557         exit(EXIT_FAILURE);
27558     }
27559
27560     if (!is_octal(argv[1]) || (mode = (mode_t)strtoul(argv[1], NULL, 8)) > ↵
27561         0x777) {
27562         fprintf(stderr, "invalid octal digits!..\n");
27563         exit(EXIT_FAILURE);
27564     }
27565
27566     major = (unsigned)strtoul(argv[3], NULL, 10);
27567     minor = (unsigned)strtoul(argv[4], NULL, 10);
27568
27569     result_mode = 0;
27570     for (i = 8; i >= 0; --i)
27571         if (mode >> i & 1)
27572             result_mode |= modes[8 - i];
27573
27574     if (strcmp(argv[2], "c") == 0)
27575         result_mode |= S_IFCHR;
27576     else if (strcmp(argv[2], "b") == 0)
27577         result_mode |= S_IFBLK;
27578     else {
27579         fprintf(stderr, "invalid device type!..\n");
27580         exit(EXIT_FAILURE);
27581     }
27582
27583     umask(0);
27584
27585     if (mknod(argv[5], result_mode, makedev(major, minor)) == -1)
27586         exit_sys("mkdir");
27587
27588     return 0;
27589 }
27590
27591 int is_octal(const char *str)
27592 {
27593     int i;
27594
27595     for (i = 0; str[i] != '\0'; ++i)
27596         if (str[i] < '0' || str[i] > '7')
27597             return 0;
27598
27599     return 1;
```

```
27598 }
27599
27600 void exit_sys(const char *msg)
27601 {
27602     perror(msg);
27603
27604     exit(EXIT_FAILURE);
27605 }
27606
27607 # Makelfile
27608
27609 obj-m += $(file).o
27610
27611 all:
27612     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
27613 clean:
27614     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
27615
27616 /
27617 -----*-----*-----*-----*
27618 -----*-----*-----*-----*
27619 -----*-----*-----*-----*
27620 -----*-----*-----*-----*
27621 -----*-----*-----*-----*
27622 -----*-----*-----*-----*
27623 -----*-----*-----*-----*
27624 -----*-----*-----*-----*
```

Aslında `mknod` fonksiyonunu kullanmak yerine doğrudan `mknod` komutuyla da kabuk üzerinden aygıt dosyası yaratılabilir. Komutun genel biçimi şöyledir:

```
27620     sudo mknod <dosya ismi> <c ya da b> <majör numara> <minör numara>
```

Örneğin:

```
27621     sudo mknod devfile c 20 30
```

Bir kernel modülün karakter aygıt sürücüsü haline getirilebilmesi için öncelikle bir aygıt numarasıyla temsil edilip çekirdeğe kaydettirilmesi gerekmektedir. Bu işlem tipik olarak `register_chrdev_region` isimli fonksiyonla yapılır. Fonksiyonun prototipi şöyledir:

```
27622     int register_chrdev_region(dev_t from, unsigned count, const char *name)
```

Fonksiyonun birinci parametresi aygıt sürücünün majör ve minör numaralarının belirtildiği `dev_t` türünden nesneyi belirtir. Bu genellikle `MKDEV` makrosuyla oluşturulmaktadır. İkinci parametre ilk parametrede belirtilen minör numaradan itibaren kaç minör numaranın kaydettirileceğini belirtmektedir. Örneğin biz `majör= 20, minör=0'dan itibaren 5 minör numarayı` kaydettirebiliriz. Fonksiyonun son parametresi `proc` ve `sys`

dosyası
sistemlerindeki aygit sürücünün ismini belirtir. Kernel modüllerinin isimleri kernel modül dosyasından gelmekte dir. Ancak karakter aygit sürücülerinin isimlerini biz istedigimiz gibi veririz. Her aygit sürücü bir kernel modül biçiminde yazilmak zorundadır. Fonksiyon basari durumunda 0 değerine başarısızlık durumunda negatif errno değerine geri döner.

register_chrdev_region fonksiyonu modülün init fonksiyonunda register ettirilen majör ve minör numaralar unregister_chrdev_region fonksiyonuyla modülün exit fonksiyonlarında geri bırakılmalıdır. Aksi halde modül kernel alanından rmmod komutuyla atılsa bile bu aygit numaraları tahsis edilmiş bir biçimde kalmaya devam etmektedir. unregister_chrdev_region fonksiyonun prototipi şöyledir:

```
void unregister_chrdev_region (dev_t from, unsigned count);
```

Fonksiyonun birinci parametresi aygit sürücünün register ettirilmiş olan majör ve minör numarasını, ikinci parametresi ise yine o noktadan başlayan kaç minör numaranın unregister ettirileceğidir.

Bir aygit sürücü register-chrdev-region fonksiyonuyla majör ve minör numarayı register ettirdiğinde artık /proc/devices dosyasında bu aygit sürücü için bir satır yaratılmaktadır. Aygit sürücü unregister_chrdev_region fonksiyonuyla yok edildiğinde /proc/devices dosyasındaki satır silinmektedir.

Bir kernel modülü yazarken o modülle ilgili önemli bazı belirlemeler "modül makroları" denilen MODULE_XXXX biçimindeki makrolarla yapılmaktadır. Her ne kadar bu modül makrolarının bulundurulması zorunlu değilse de şiddetle tavsiye edilmektedir. En önemli üç makronun tipik kullanımı şöyledir:

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kaan Aslan");
MODULE_DESCRIPTION("General Character Device Driver");
```

Modül lisansı herhangi bir open source lisans olabilir. Tipik olarak "GPL" tercih edilmektedir. MODULE_AUTHOR makrosu ile modülün yazarı belirtilir.

MODULE_DESCRIPTION modülün ne iş yaprigina yönelik kısa bir başlık yazısı içermektedir.

Bu makrolar global alanda herhangi bir yere yerleştirilebilmektedir.

27670 -----*/
27671
27672 /
*-----
27673 Bir modülü bir aygıt numarasıyla ilişkilendirdikten sonra artık ona
gerçek anlamda bir karakter aygıt sürücü kimliği kazandırmak
gerekmektedir. Bu işlem sstruct cdev isimli bir yapının için
doldurularak sisteme eklenmesi (yerleştirilmesi) ile yapılır. Linux
çekirdeği
27675 tüm kernel modülleri ve aygıt sürücülerini çeşitli veri yapılarıyla
tutmaktadır. Aygıt sürücü yazan programcılar kernel'in bu
organizasyonunu
27676 bilmek zorunda değiller. Ancak bazı işlemleri tam gerektiği gibi yapmak
zorundalar.
27677
27678 cdev yapısı aşağıdaki gibi bir yapıdır:
27679
27680 struct cdev {
27681 struct kobject kobj;
27682 struct module *owner;
27683 const struct file_operations *ops;
27684 struct list_head list;
27685 dev_t dev;
27686 unsigned int count;
27687 };
27688
27689 Bu türden bir yapı nesnesi programcı tarafından global olarak
tanımlanabilir ya da alloc_cdev isimli kernel fonksiyonuyla kernel'in
27690 heap sistemi (slab allocator) kullanılarak dinamik bir biçimde tahsis
edilebilir. Eğer bu yapı nesni programcı tarafından global bir
biçimde
27691 tanımlanacaksa yapının elemanlarına ilkdeğer vermek için cdev_init
fonksiyonu çağrılmalıdır. Eğer cdev yapısı cdev_alloc fonksiyonuyla
dinamik bir biçimde
27692 tahsis edileceğse cdev_init yapılmaz çünkü zaten cdev_alloc bu işlemi de
yapmaktadır. Fakat yine de programının bu kez manuel olarak bu
yapının bazı
27693 elemanlarına değer ataması gereklidir. Bu iki yoldan biriyle oluşturulmuş
olan cdev yapısının en sonunda cdev_add isimli fonksiyonla kernel veri
yapılarına
27694 yerleştirilmeleri gereklidir. Tabii aygıt sürücü boşaltılırken bu
yerleştirme işlemi cdev_del fonksiyonuyla geri alınmalıdır.
27695
27696 cdev_init fonksiyonun parametrik yapısı şöyledir:
27697
27698 void cdev_init(struct cdev *cdev, const struct file_operations *fops);
27699
27700 Fonksiyonun birinci parametresi ilkdeğer verilecek global cdev nesnenin
adresini alır. İkinci parametre ise file_operations türünden
27701 bir yapı nesnesinin adresi almaktadır. file_operations isimli yapı
birtakım fonksiyon adreslerinden oluşmaktadır. Yani yapının tüm

elemanları

27702 birer fonksiyon göstericisidir. Bu yapı user moddaki program tarafından ↵
ilgili aygit dosyası açılıp çeşitli işlemler yapıldığında çağrılacak

27703 fonksiyonların adreslerini tutmaktadır. Yani örneğin user moddaki ↵
program open, close, read, write yaptığında çağrılacak ↵
fonksiyonlarını burada belirtiriz.

27704 file_operations yapısı büyük bir yapıdır:

27705

```
27706 struct file_operations {  
27707     struct module *owner;  
27708     loff_t (*llseek) (struct file *, loff_t, int);  
27709     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);  
27710     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);  
27711     ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);  
27712     ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);  
27713     int (*iopoll)(struct kiocb *kiocb, bool spin);  
27714     int (*iterate) (struct file *, struct dir_context *);  
27715     int (*iterate_shared) (struct file *, struct dir_context *);  
27716     __poll_t (*poll) (struct file *, struct poll_table_struct *);  
27717     long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);  
27718     long (*compat_ioctl) (struct file *, unsigned int, unsigned long);  
27719     int (*mmap) (struct file *, struct vm_area_struct *);  
27720     unsigned long mmap_supported_flags;  
27721     int (*open) (struct inode *, struct file *);  
27722     int (*flush) (struct file *, fl_owner_t id);  
27723     int (*release) (struct inode *, struct file *);  
27724     int (*fsync) (struct file *, loff_t, loff_t, int datasync);  
27725     int (*fasync) (int, struct file *, int);  
27726     int (*lock) (struct file *, int, struct file_lock *);  
27727     ssize_t (*sendpage) (struct file *, struct page *, int, size_t,  
27728         loff_t *, int);  
27729     unsigned long (*get_unmapped_area)(struct file *, unsigned long,  
27730         unsigned long, unsigned long, unsigned long);  
27731     int (*check_flags)(int);  
27732     int (*flock) (struct file *, int, struct file_lock *);  
27733     ssize_t (*splice_write)(struct pipe_inode_info *, struct file *,  
27734         loff_t *, size_t, unsigned int);  
27735     ssize_t (*splice_read)(struct file *, loff_t *, struct  
27736         pipe_inode_info *, size_t, unsigned int);  
27737     int (*setlease)(struct file *, long, struct file_lock **, void **);  
27738     long (*fallocate)(struct file *file, int mode, loff_t offset,  
27739         loff_t len);  
27740     void (*show_fdinfo)(struct seq_file *m, struct file *f);  
27741 #ifndef CONFIG_MMU  
27742     unsigned (*mmap_capabilities)(struct file *);  
27743 #endif  
27744     ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,  
27745         loff_t, size_t, unsigned int);  
27746     loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,  
27747         struct file *file_out, loff_t pos_out,  
27748         loff_t len, unsigned int remap_flags);  
27749     int (*fadvise)(struct file *, loff_t, loff_t, int);
```

```
27746     };
27747
27748     Bu yapının bazı elemanlarına atama yapabiliriz. Bunun için gcc      ↵
27749     eklentileri kullanılabilir. (Bu eklentiler C99 ile birlikte C'ye      ↵
27750     eklenmiştir.) Örneğin:
27751
27752     int generic_open(struct inode *inodep, struct file *filp);
27753     int generic_release(struct inode *inodep, struct file *filp);
27754
27755     struct file_operations g_file_ops = {
27756         .owner = THIS_MODULE,
27757         .open = generic_open,
27758         .release = generic_release
27759     };
27760
27761     Yapının owner elemanına THIS_MODULE atamasının yapılması iyi bir      ↵
27762     tekniktir.
27763
27764     cdev yapısı cdev_alloc fonksiyonuyla dinamik bir biçimde de yahsis      ↵
27765     edilebilir:
27766
27767     struct cdev *cdev_alloc(void);
27768
27769     Fonksiyon başarı durumunda cdev yapısının adresine başarısılık durumunda      ↵
27770     NULL adrese geri dönmektedir. Yukarıda da belirtildiği gibi cdev      ↵
27771     yapısı
27772     cdev_alloc ile tahsis edilmişse cdev_init yapılmasına gerek yoktur.      ↵
27773     Ancak bu durumda programının manuel olarak owner, ops elemanlarına
27774     değer ataması uygun olur. Örneğin:
27775
27776     struct cdev *g_cdev;
27777     ...
27778     if ((gcdev = cdev_alloc()) == NULL) {
27779         printk(KERN_INFO "Cannot allocate cdev!..\n");
27780         return -ENOMEM;
27781     }
27782     g_cdev->owner = THIS_MODULE;
27783     g_cdev->ops = &g_file_ops;
27784
27785     cdev yapı nesnesi başarılı bir biçimde oluşturulduktan sonra artık bu      ↵
27786     yapının kernel içerisinde yerleştirilmesi gereklidir. Bu da cdev_add      ↵
27787     fonksiyonuyla
27788     yapılmaktadır:
27789
27790     int cdev_add(struct cdev *devp, dev_t dev, unsigned count);
27791
27792     Fonksiyonun birinci parametresi cdev türünden yapı nesnesinin adresini      ↵
27793     almaktadır. İkinci parametre aygit sürücünün majör ve minör numarasını      ↵
27794     belirtmektedir.
27795     Üçüncü paraetres ise ilgili minör numaradan kaç minör numaranın      ↵
27796     kullanılacağı belirtir. Fonksiyon başarı durumunda sıfır değerine
27797     başarısızlık durumunda negatif
27798     errno değerine geri döner.
```

```
27786
27787      Aygit sürücü boşaltılırken cdev_add ile yapılan işlemin geri alınması     ↵
27788          gereklidir. Bu da cdev_del fonksiyonuyla yapılmaktadır. (cdev_alloc     ↵
27789          işlemi için
27790          bunu free hale getiren ayrı bir fonksiyon yoktur. cdev_del sırasında     ↵
27791          eğer bu yapı dinamik tahsis edilmişse free işlemi yapılmaktadır.)
27792
27793      void cdev_del(struct cdev *devp);
27794
27795      Fonksiyon parametre olarak cdev yapısının adresini almaktadır.
27796
27797      Buradaki önemli bir nokta şudur: cdev_add fonksiyonu cdev nesnesinin     ↵
27798          içini kernel'daki uygun veri yapısına kopyalamamaktadır. Bizzat bu     ↵
27799          nesnenin adresini
27800      kullanmaktadır. Bu cdev nesnesinin yaşıyor olması gereklidir. Bu da cdev     ↵
27801          nesnesinin ve file_operations nesnesinin global biçimde tanımlanması     ↵
27802          gereklidir.
27803
27804      Aşağıda bu işlemlerin yapıldığı örnek bir karakter aygit sürücüsü     ↵
27805          verilmiştir. Bu aygit sürücü majör=25, minör=0 aygitini     ↵
27806          kullanmaktadır.
27807
27808      Dolayısıyla aşağıdaki programın testi için şöyle bir aygit dosyasının     ↵
27809          yaratılması gereklidir:
27810
27811      sudo mknod generic -m=666 c 25 0
27812
27813  -----
27814  -----
27815  -----
27816  -----
27817  -----
27818  -----
27819  -----
27820  -----
27821  -----
27822  -----
27823  -----
27824  -----
27825  -----
27826  -----
27827  -----*/
```

```
27804 /* generic-char-driver.c */
27805
27806 #include <linux/module.h>
27807 #include <linux/kernel.h>
27808 #include <linux/fs.h>
27809 #include <linux/cdev.h>
27810
27811 MODULE_LICENSE("GPL");
27812 MODULE_DESCRIPTION("General Character Device Driver");
27813 MODULE_AUTHOR("Kaan Aslan");
27814
27815 #define DEV_MAJOR    25
27816 #define DEV_MINOR    0
27817
27818 static int generic_open(struct inode *inodep, struct file *filp);
27819 static int generic_release(struct inode *inodep, struct file *filp);
27820
27821 static dev_t g_dev = MKDEV(DEV_MAJOR, DEV_MINOR);
27822 static struct cdev g_cdev;
27823 static struct file_operations g_file_ops = {
27824     .owner = THIS_MODULE,
27825     .open = generic_open,
27826     .release = generic_release,
27827 };
```

```
27828
27829 static int __init generic_init(void)
27830 {
27831     int result;
27832
27833     if ((result = register_chrdev_region(g_dev, 1, "generic-char-driver")) < 0) {
27834         printk(KERN_INFO "Cannot register driver!...\n");
27835         return result;
27836     }
27837
27838     cdev_init(&g_cdev, &g_file_ops);
27839
27840     if ((result = cdev_add(&g_cdev, g_dev, 1)) != 0) {
27841         unregister_chrdev_region(g_dev, 1);
27842         printk(KERN_INFO "Cannot add character device driver!...\n");
27843         return result;
27844     }
27845
27846     printk(KERN_INFO "Success...\n");
27847
27848     return 0;
27849 }
27850
27851 static void __exit generic_exit(void)
27852 {
27853     cdev_del(&g_cdev);
27854     unregister_chrdev_region(g_dev, 1);
27855
27856     printk(KERN_INFO "Goodbye...\n");
27857 }
27858
27859 static int generic_open(struct inode *inodep, struct file *filp)
27860 {
27861     printk(KERN_INFO "Generic device opened!..\n");
27862
27863     return 0;
27864 }
27865
27866 static int generic_release(struct inode *inodep, struct file *filp)
27867 {
27868     printk(KERN_INFO "Generic device released!..\n");
27869
27870     return 0;
27871 }
27872
27873 module_init(generic_init);
27874 module_exit(generic_exit);
27875
27876 # Makelfile
27877
27878 obj-m += $(file).o
27879
```

```
27880 all:
27881     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
27882 clean:
27883     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
27884
27885 /* sample.c */
27886
27887 #include <stdio.h>
27888 #include <stdlib.h>
27889 #include <fcntl.h>
27890 #include <unistd.h>
27891
27892 void exit_sys(const char *msg);
27893
27894 int main(void)
27895 {
27896     int fd;
27897
27898     if ((fd = open("generic", O_RDWR)) == -1)
27899         exit_sys("open");
27900
27901     close(fd);
27902
27903     return 0;
27904 }
27905
27906 void exit_sys(const char *msg)
27907 {
27908     perror(msg);
27909
27910     exit(EXIT_FAILURE);
27911 }
27912
27913 /
*-----*
-----*
27914 Yukarıdaki programı cdev nesnesini alloc_cdev fonksiyonuyla dinamik bir
biçimde tahsis ederek de şöyle yazabiliriz:
27915 -----*/
27916
27917 /* generic-char-driver.c */
27918
27919 #include <linux/module.h>
27920 #include <linux/kernel.h>
27921 #include <linux/fs.h>
27922 #include <linux/cdev.h>
27923
27924 MODULE_LICENSE("GPL");
27925 MODULE_DESCRIPTION("General Character Device Driver");
27926 MODULE_AUTHOR("Kaan Aslan");
27927
27928 #define DEV_MAJOR      25
```

```
27929 #define DEV_MINOR      0
27930
27931 static int generic_open(struct inode *inodep, struct file *filp);
27932 static int generic_release(struct inode *inodep, struct file *filp);
27933
27934 static dev_t g_dev = MKDEV(DEV_MAJOR, DEV_MINOR);
27935 static struct cdev *g_cdev;
27936 static struct file_operations g_file_ops = {
27937     .owner = THIS_MODULE,
27938     .open = generic_open,
27939     .release = generic_release,
27940 };
27941
27942 static int __init generic_init(void)
27943 {
27944     int result;
27945
27946     if ((result = register_chrdev_region(g_dev, 1, "generic-char-driver")) < 0) {
27947         printk(KERN_INFO "Cannot register driver!...\n");
27948         return result;
27949     }
27950
27951     if ((g_cdev = cdev_alloc()) == NULL) {
27952         printk(KERN_INFO "Cannot allocate cdev!..\n");
27953         return -ENOMEM;
27954     }
27955
27956     g_cdev->owner = THIS_MODULE;
27957     g_cdev->ops = &g_file_ops;
27958
27959     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
27960         unregister_chrdev_region(g_dev, 1);
27961         printk(KERN_INFO "Cannot add character device driver!...\n");
27962         return result;
27963     }
27964
27965     printk(KERN_INFO "Success...\n");
27966
27967     return 0;
27968 }
27969
27970 static void __exit generic_exit(void)
27971 {
27972     cdev_del(g_cdev);
27973     unregister_chrdev_region(g_dev, 1);
27974
27975     printk(KERN_INFO "Goodbye...\n");
27976 }
27977
27978 static int generic_open(struct inode *inodep, struct file *filp)
27979 {
27980     printk(KERN_INFO "Generic device opened!..\n");
```

```
27981     return 0;
27982 }
27983 }
27984
27985 static int generic_release(struct inode *inodep, struct file *filp)
27986 {
27987     printk(KERN_INFO "Generic device released!..\n");
27988
27989     return 0;
27990 }
27991
27992 module_init(generic_init);
27993 module_exit(generic_exit);
27994
27995 # Makelfile
27996
27997 obj-m += $(file).o
27998
27999 all:
28000     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
28001 clean:
28002     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
28003
28004 /* sample.c */
28005
28006 #include <stdio.h>
28007 #include <stdlib.h>
28008 #include <fcntl.h>
28009 #include <unistd.h>
28010
28011 void exit_sys(const char *msg);
28012
28013 int main(void)
28014 {
28015     int fd;
28016
28017     if ((fd = open("generic", O_RDWR)) == -1)
28018         exit_sys("open");
28019
28020     close(fd);
28021
28022     return 0;
28023 }
28024
28025 void exit_sys(const char *msg)
28026 {
28027     perror(msg);
28028
28029     exit(EXIT_FAILURE);
28030 }
28031
28032 /
*----- →
```

28033 Kernel kodları ya da aygit sürücü kodları çoğu zaman kernel alanı ile ↵
user alanı arasında veri tarsferi yapmak isterler.

28034 Örneğin sys_read fonksiyonu kernel alanında elde ettiği bilgileri user ↵
alanındaki programcının verdiği adrese kopyalar.

28035 Benzer biçimde sys_write fonksiyonu da bunun tersini yapmaktadır. Kernel ↵
alanı ile user alanı arasında birkaç nedenden dolayı

28036 memcpy fonksiyonuyla transfer yapılamamaktadır. Bu tür transferlerde ↵
kernel mod programcılarının user alanındaki adresin geçerliliğini

28037 kontrol etmesi gereklidir. Aksi takdirde kernel modda geçersiz bir alana ↵
kopyalama yapmak sistemin çökmesine yol açabilmektedir. Ayrıca

28038 user alanına ilişkin prosesin sayfa tablosunun bazı bölmeleri o anda ↵
bellekte olmayabilir (yani swap out yapılmış olabilir). Böyle bir ↵
durumda

28039 işleme devam etmek kernel tasarımları açısından sorun olmaktadır. Eğer ↵
bu bir durum varsa kernel kodlarının önce sayfa tablosunu RAM'e ↵
geri yükleyip

28040 işlemine devam etmesi uygun olmaktadır.

28041

28042 İşte yukarıda açıklanan bazı nedenlerden dolayı kernel alanı ile user ↵
alanı arasında kopyalama işlemi için özel kernel fonksiyonları ↵
kullanılmaktadır.

28043 En temel iki fonksiyon copy_to_user ve copy_from_user fonksiyonlarıdır. ↵
Bu fonksiyonların prototipleri <linux/uaccess.h> içerisinde yer almaktadır:

28044

28045 unsigned long copy_to_user(void *to, const void *from, unsigned len);

28046 unsigned long copy_from_user(void *to, const void *from, unsigned len);

28047

28048 Fonksiyonların birinci parametreleri kopyalamanın yapılacak hedef ↵
adreslerdir. Yani copy_to_user için birinci parametre user alanındaki ↵
adres,

28049 copy_from_user için birinci parametre kernel alanındaki adresdir. İkinci ↵
parametre kaynak adresi belirtmektedir. Bu kaynak adres copy_to_user ↵
için kernel

28050 alanındaki adres, copy_from_user için user alanındaki adresdir. Son ↵
parametre transfer edilecek byte sayısını belirtmektedir. Fonksiyonlar ↵
başarılı durumunda 0

28051 değerine, başarısızlık durumunda transfer edilemeyen byte sayısına geri ↵
dönerler. Kernel mod programcılının bu fonksiyonlar başarısızken bunu ↵
çağırın fonsiyonlarını

28052 -EFAULT (Bad address) ile geri döndürmesi uygun olur.

28053

28054 Bazen user alanındaki adresin zaten geçerliliği sınanmıştır. Bu durumda ↵
yeniden geçerlilik sınaması yapmadan yukarıdaki işlemleri yapan

28055 __copy_to_user ve __copy_from_user fonksiyonları kullanılabilir. Bu ↵
fonksiyonların parametrik yapıları aynıdır.

28056

28057 Bazı durumlarda programcı 1 byte, 2 byte, 4 byte, 8 byte'lık verileri ↵
kopyalamak isteyebilir. Bu küçük miktarlarda verilen kopyalaması

28058 için daha hızlı çalışan özel iki makro vardır: put_user ve get_user. ↵
Makroların parametrik yapısı şöyledir:

28059

28060 put_user(x, ptr)

```
28061     get_user(x, ptr)
28062
28063     Burada x aktarılacak nesneyi belirtir. (Adresini programcı almaz, makro ↵
28064         içinde bu işlem yapılmaktadır.) ptr ise transfer adresini belirtir. ↵
28065         Aktarım ikinci
28066     parametrede belirtilen adresin türünün uzunluğu kadar yapılmaktadır. ↵
28067         Yine makrolar başarı durumunda 0 başarısızlık durumunda negatif hata ↵
28068         koduna geri dönmektedir.
28069     Bu makroların da geçerlilik kontrolü yapmayan __put_user ve __get_user ↵
28070         isimli versiyonları vardır. Örneğin biz 4 byte'lık int bir x ↵
28071         nesnesinin içerisindeki bilgiyi
28072     pi user adresine kopyalamak isteyelim. Bu işlemi şöye yaparız:
28073
28074     int x;
28075     int *puser;
28076     ...
28077     put_user(x, puser)
28078
28079     Nihayet user alanındaki adresin geçerliliği de access_ok isimli makroula ↵
28080         sorgulanabilmektedir. Makro şöyledir:
28081
28082     access_ok(type,addr,size)
28083
28084     Buradaki type geçerliliğin türünü anlatmaktadır. Okuma geçerliliği için ↵
28085         bu parametre VERIFY_READ, yazma geçerliliği için VERIFY_WRITE ve hem ↵
28086         okuma hem de yazma
28087     geçerliliği için VERIFY_READ|VERIFY_WRITE biçiminde olmalıdır. İkinci ↵
28088         parametre geçerliliği sınanacak adresi ve üçüncü parametre de o ↵
28089         adresinden başlayan alanın
28090     uzunluğunu belirtmektedir. Fonksiyon başarı durumunda sıfır dışı bir ↵
28091         değere başarısızlık durumunda sırr değerine geri dönmektedir.
28092
28093     -----
28094     -----
28095     Aygit sürücümüz için read ve write fonksiyonları aşağıdaki parametrik ↵
28096         yapıya uygun olacak biminde file_operations yapısına ↵
28097         yerleştirilmelidir:
28098
28099     static ssize_t generic_read(struct file *filp, char *buf, size_t size, ↵
28100         loff_t *off);
28101     static ssize_t generic_write(struct file *filp, const char *buf, size_t ↵
28102         size, loff_t *off);
28103
28104     static struct file_operations g_file_ops = {
28105         .owner = THIS_MODULE,
28106         .open = generic_open,
28107         .release = generic_release,
28108         .read = generic_read,
28109         .write = generic_write,
```

```
28095     };
28096
28097     read ve write fonksiyonlarının birinci parametresi açılmış dosyaya      ↵
28098         ilişkin struct file nesnesinin adresini belirtir. Linux dosya      ↵
28099         sisteminde
28100     üç önemli yapı vardır: struct file, struct dentry ve struct inode. file ↵
28101         yapısının bir elemanı denrty yapısını, dentry yapısının da bir elemanı ↵
28102         inode yapısını
28103     göstermektedir. Yani biz dentry ve inode yapılarının elemanlarına      ↵
28104         aslında struct file nesnesi yoluyla erişiriz. Daha önceden de      ↵
28105         anımsanacağı gibi struct file
28106     nesnesi içerisinde dosya göstericisinin konumu, dosyanın erişim hakları, ↵
28107         referans sayacının değeri, dosyanın açış modu ve başka birtakım      ↵
28108         bilgiler de vardır. read ve
28109     write fonksiyonlarının ikinci parametresi user alanındaki transfer      ↵
28110         adresini belirtir. Üçüncü parametreler okunacak ya da yazılacak byte      ↵
28111         miktarını belirtmektedir.
28112     Son parametre dosya göstericisinin konumunu belirtir. (Aslında bu      ↵
28113         parametre file yapısı içerisindeki f_pos lemanın adresidir. Ancak      ↵
28114         kolay erişim için ayrı bir
28115     parametre olarak geçirilmiştir.) Fonksiyon başarı durumunda transfer      ↵
28116         edilen byte sayısına başarısızlık durumunda negatif errno değerine      ↵
28117         geri dönmektedir.
28118
28119     Aşağıdaki örnekte aygit sürücü için read fonksiyonu yazılmıştır. Bu      ↵
28120         fonksiyon aslında g_buf isimli dizinin içini dosya gibi vermektedir.
28121     -----
28122     -----
28123
28124     /* generic-char-driver.c */
28125
28126     #include <linux/module.h>
28127     #include <linux/kernel.h>
28128     #include <linux/fs.h>
28129     #include <linux/cdev.h>
28130     #include <linux/uaccess.h>
28131
28132     MODULE_LICENSE("GPL");
28133     MODULE_DESCRIPTION("General Character Device Driver");
28134     MODULE_AUTHOR("Kaan Aslan");
28135
28136     #define DEV_MAJOR    25
28137     #define DEV_MINOR    0
28138
28139     static int generic_open(struct inode *inodep, struct file *filp);
28140     static int generic_release(struct inode *inodep, struct file *filp);
28141     static ssize_t generic_read(struct file *filp, char *buf, size_t size,      ↵
28142         loff_t *off);
28143
28144     static dev_t g_dev = MKDEV(DEV_MAJOR, DEV_MINOR);
28145     static struct cdev *g_cdev;
28146     static struct file_operations g_file_ops = {
```

```
28131     .owner = THIS_MODULE,
28132     .open = generic_open,
28133     .release = generic_release,
28134     .read = generic_read,
28135 };
28136
28137 char g_buf[] = "abcdefghijklmnopqrstuvwxyz";
28138
28139 static int __init generic_init(void)
28140 {
28141     int result;
28142
28143     if ((result = register_chrdev_region(g_dev, 1, "generic-char-driver")) < 0) {
28144         printk(KERN_INFO "Cannot register driver!...\n");
28145         return result;
28146     }
28147
28148     if ((g_cdev = cdev_alloc()) == NULL) {
28149         printk(KERN_INFO "Cannot allocate cdev!..\n");
28150         return -ENOMEM;
28151     }
28152
28153     g_cdev->owner = THIS_MODULE;
28154     g_cdev->ops = &g_file_ops;
28155
28156     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
28157         unregister_chrdev_region(g_dev, 1);
28158         printk(KERN_INFO "Cannot add character device driver!...\n");
28159         return result;
28160     }
28161
28162     printk(KERN_INFO "Success...\n");
28163
28164     return 0;
28165 }
28166
28167 static void __exit generic_exit(void)
28168 {
28169     cdev_del(g_cdev);
28170     unregister_chrdev_region(g_dev, 1);
28171
28172     printk(KERN_INFO "Goodbye...\n");
28173 }
28174
28175 static int generic_open(struct inode *inodep, struct file *filp)
28176 {
28177     printk(KERN_INFO "Generic device opened!..\n");
28178
28179     return 0;
28180 }
28181
28182 static int generic_release(struct inode *inodep, struct file *filp)
```

```
28183 {
28184     printk(KERN_INFO "Generic device released!..\n");
28185
28186     return 0;
28187 }
28188
28189 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
28190                             loff_t *off)             ↵
28190 {
28191     size_t asize;
28192
28193     asize = size + *off > strlen(g_buf) ? strlen(g_buf) - *off : size;
28194
28195     if (copy_to_user(buf, g_buf + *off, asize) != 0)
28196         return -EFAULT;
28197
28198     *off += asize;
28199
28200     printk(KERN_INFO "Reading...\n");
28201
28202     return asize;
28203 }
28204
28205 module_init(generic_init);
28206 module_exit(generic_exit);
28207
28208 /* sample.c */
28209
28210 #include <stdio.h>
28211 #include <stdlib.h>
28212 #include <fcntl.h>
28213 #include <unistd.h>
28214
28215 void exit_sys(const char *msg);
28216
28217 int main(void)
28218 {
28219     int fd;
28220     char buf[100];
28221     ssize_t result;
28222
28223     if ((fd = open("generic", O_RDWR)) == -1)
28224         exit_sys("open");
28225
28226     while ((result = read(fd, buf, 3)) > 0) {
28227         buf[result] = '\0';
28228         printf("Read from %ld bytes: %s\n", (long)result, buf);
28229     }
28230
28231     if (result < 0)
28232         exit_sys("read");
28233
28234     close(fd);
```

```
28235
28236     return 0;
28237 }
28238
28239 void exit_sys(const char *msg)
28240 {
28241     perror(msg);
28242
28243     exit(EXIT_FAILURE);
28244 }
28245
28246 # Makelfile
28247
28248 obj-m += $(file).o
28249
28250 all:
28251     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
28252 clean:
28253     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
28254
28255 /
*-----*
-----*
28256     Aygit sürücü için write fonksiyonu da tamamen read fonksiyonuna benzer
28257     biçimde yazılmaktadır. write fonksiyonu içerisinde
28258     biz user moddaki bilgiyi copy_from_user ya da get_user fonksiyonlarıyla
28259     alırız. Yine write fonksiyonu da bir sorun çıktığında
28260     -EFAULT değeri ile başarılı sonlanmada yazılan (kernel alanına yazılan)
28261     byte miktarı ile geri dönmeliidir. Aşağıda erite
28262     işlemeye bir örnek verilmiştir.
28263 -----*/
28264 /* generic-char-driver.c */
28265
28266 #include <linux/module.h>
28267 #include <linux/kernel.h>
28268 #include <linux/fs.h>
28269 #include <linux/cdev.h>
28270 #include <linux/uaccess.h>
28271
28272 MODULE_LICENSE("GPL");
28273 MODULE_DESCRIPTION("General Character Device Driver");
28274 MODULE_AUTHOR("Kaan Aslan");
28275
28276 #define DEV_MAJOR    25
28277 #define DEV_MINOR    0
28278
28279 static int generic_open(struct inode *inodep, struct file *filp);
28280 static int generic_release(struct inode *inodep, struct file *filp);
28281 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
```

```
    loff_t *off);
28282 static ssize_t generic_write(struct file *filp, const char *buf, size_t      ↵
    size, loff_t *off);
28283
28284 static dev_t g_dev = MKDEV(DEV_MAJOR, DEV_MINOR);
28285 static struct cdev *g_cdev;
28286 static struct file_operations g_file_ops = {
28287     .owner = THIS_MODULE,
28288     .open = generic_open,
28289     .release = generic_release,
28290     .read = generic_read,
28291     .write = generic_write,
28292 };
28293
28294 char g_buf[BUFFER_SIZE];
28295
28296 static int __init generic_init(void)
28297 {
28298     int result;
28299
28300     if ((result = register_chrdev_region(g_dev, 1, "generic-char-driver")) < ↵
28301         0) {
28301         printk(KERN_INFO "Cannot register driver!...\n");
28302         return result;
28303     }
28304
28305     if ((g_cdev = cdev_alloc()) == NULL) {
28306         printk(KERN_INFO "Cannot allocate cdev!..\n");
28307         return -ENOMEM;
28308     }
28309
28310     g_cdev->owner = THIS_MODULE;
28311     g_cdev->ops = &g_file_ops;
28312
28313     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
28314         unregister_chrdev_region(g_dev, 1);
28315         printk(KERN_INFO "Cannot add character device driver!...\n");
28316         return result;
28317     }
28318
28319     printk(KERN_INFO "Success...\n");
28320
28321     return 0;
28322 }
28323
28324 static void __exit generic_exit(void)
28325 {
28326     cdev_del(g_cdev);
28327     unregister_chrdev_region(g_dev, 1);
28328
28329     printk(KERN_INFO "Goodbye...\\n");
28330 }
28331
```

```
28332 static int generic_open(struct inode *inodep, struct file *filp)
28333 {
28334     printk(KERN_INFO "Generic device opened!..\n");
28335
28336     return 0;
28337 }
28338
28339 static int generic_release(struct inode *inodep, struct file *filp)
28340 {
28341     printk(KERN_INFO "Generic device released!..\n");
28342
28343     return 0;
28344 }
28345
28346 static ssize_t generic_read(struct file *filp, char *buf, size_t size,      ↵
28347     loff_t *off)
28348 {
28349     size_t asize;
28350
28351     asize = size + *off > BUFFER_SIZE ? BUFFER_SIZE - *off : size;
28352
28353     if (copy_to_user(buf, g_buf + *off, asize) != 0)
28354         return -EFAULT;
28355
28356     *off += asize;
28357
28358     printk(KERN_INFO "Reading...\\n");
28359
28360     return asize;
28361 }
28362
28363 static ssize_t generic_write(struct file *filp, const char *buf, size_t      ↵
28364     size, loff_t *off)
28365 {
28366     size_t asize;
28367
28368     asize = size + *off > BUFFER_SIZE ? BUFFER_SIZE - *off : size;
28369     if (copy_from_user(g_buf + *off, buf, asize) != 0)
28370         return -EFAULT;
28371
28372     *off += asize;
28373
28374     printk(KERN_INFO "Writing...\\n");
28375
28376 module_init(generic_init);
28377 module_exit(generic_exit);
28378
28379 # Makelfile
28380
28381 obj-m += $(file).o
```

```
28383
28384 all:
28385     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
28386 clean:
28387     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
28388
28389 /* sample.c */
28390
28391 #include <stdio.h>
28392 #include <stdlib.h>
28393 #include <string.h>
28394 #include <fcntl.h>
28395 #include <unistd.h>
28396
28397 void exit_sys(const char *msg);
28398
28399 int main(void)
28400 {
28401     int fd;
28402     char wbuf[100] = "this is a test";
28403     char rbuf[100];
28404     ssize_t result;
28405
28406     if ((fd = open("generic", O_WRONLY)) == -1)
28407         exit_sys("open");
28408
28409     if ((result = write(fd, wbuf, strlen(wbuf))) == -1)
28410         exit_sys("write");
28411
28412     printf("%ld bytes written...\n", (long)result);
28413
28414     close(fd);
28415
28416     if ((fd = open("generic", O_RDONLY)) == -1)
28417         exit_sys("open");
28418
28419     if ((result = read(fd, rbuf, 10)) == -1)
28420         exit_sys("read");
28421     rbuf[result] = '\0';
28422
28423     printf("%ld bytes read: %s\n", result, rbuf);
28424
28425     close(fd);
28426
28427     return 0;
28428 }
28429
28430 void exit_sys(const char *msg)
28431 {
28432     perror(msg);
28433
28434     exit(EXIT_FAILURE);
28435 }
```

```
28436
28437  /
28438  *-----*
28439  ----- User moddan aygit dosyası betimleyicisi ile lseek işlemi yapıldığında
28440  ----- aygit sürücünün file_operations yapısı içerisinde yerleştirilen
28441  llseek fonksiyonu çağrılmaktadır. Fonksiyonun parametrik yapısı
28442  şöyledir:
28443
28444  static loff_t generic_llseek(struct file *filp, loff_t off, int whence);
28445
28446  Fonksiyonun birinci parametresi dosya nesnesini, ikinci parametresi
28447  konumlandırılacak istenen offset'i üçüncü parametresi ise
28448  konumlandırmanın nereye göre yapılacağını belirtir. Bu fonksiyonu
28449  gerçekleştirirken programcı file yapısı içerisindeki f_pos
28450  elemanını güncellemelidir. Tipik olarak programcı whence parametresini
28451  switch içerisinde alır. Hedeflene offset'i hesaplar ve en sonunda
28452  file yapsının f_pos elemanına bu hedeflenen offset'i yerleştirir.
28453  Hedeflenen offset uygun değilse fonksiyon tipik olarak -EINVAL
28454  değeriyle
28455  geri döndürülür. Eğer konumlandırma offset'i başarılı ise fonksiyon
28456  dosya göstericisinin yenri değerine geri dönmelidir.
28457
28458  Aşağıda llseek fonksiyonun gerçekleştirimesine ilişkin bir örnek
28459  verilmiştir.
28460  -----
28461  -----
28462  -----
28463  -----
28464  #include <linux/module.h>
28465  #include <linux/kernel.h>
28466  #include <linux/fs.h>
28467  #include <linux/cdev.h>
28468  #include <linux/uaccess.h>
28469  MODULE_LICENSE("GPL");
28470  MODULE_DESCRIPTION("General Character Device Driver");
28471  MODULE_AUTHOR("Kaan Aslan");
28472
28473  #define DEV_MAJOR    25
28474  #define DEV_MINOR     0
28475
28476  #define BUFFER_SIZE    128
28477
28478  static int generic_open(struct inode *inodep, struct file *filp);
28479  static int generic_release(struct inode *inodep, struct file *filp);
28480  static ssize_t generic_read(struct file *filp, char *buf, size_t size,
28481  loff_t *off);
28482  static ssize_t generic_write(struct file *filp, const char *buf, size_t
28483  size, loff_t *off);
28484  static loff_t generic_llseek(struct file *filp, loff_t off, int whence);
```

```
28474
28475 static dev_t g_dev = MKDEV(DEV_MAJOR, DEV_MINOR);
28476 static struct cdev *g_cdev;
28477 static struct file_operations g_file_ops = {
28478     .owner = THIS_MODULE,
28479     .open = generic_open,
28480     .release = generic_release,
28481     .read = generic_read,
28482     .write = generic_write,
28483     .llseek = generic_llseek,
28484 };
28485
28486 char g_buf[BUFFER_SIZE];
28487
28488 static int __init generic_init(void)
28489 {
28490     int result;
28491
28492     if ((result = register_chrdev_region(g_dev, 1, "generic-char-driver")) < 0) {
28493         printk(KERN_INFO "Cannot register driver!...\n");
28494         return result;
28495     }
28496
28497     if ((g_cdev = cdev_alloc()) == NULL) {
28498         printk(KERN_INFO "Cannot allocate cdev!..\n");
28499         return -ENOMEM;
28500     }
28501
28502     g_cdev->owner = THIS_MODULE;
28503     g_cdev->ops = &g_file_ops;
28504
28505     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
28506         unregister_chrdev_region(g_dev, 1);
28507         printk(KERN_INFO "Cannot add character device driver!...\n");
28508         return result;
28509     }
28510
28511     printk(KERN_INFO "Success...\n");
28512
28513     return 0;
28514 }
28515
28516 static void __exit generic_exit(void)
28517 {
28518     cdev_del(g_cdev);
28519     unregister_chrdev_region(g_dev, 1);
28520
28521     printk(KERN_INFO "Goodbye...\n");
28522 }
28523
28524 static int generic_open(struct inode *inodep, struct file *filp)
28525 {
```

```
28526     printk(KERN_INFO "Generic device opened!..\n");
28527
28528     return 0;
28529 }
28530
28531 static int generic_release(struct inode *inodep, struct file *filp)
28532 {
28533     printk(KERN_INFO "Generic device released!..\n");
28534
28535     return 0;
28536 }
28537
28538 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
28539                             loff_t *off)      ↵
28539 {
28540     size_t asize;
28541
28542     asize = size + *off > BUFFER_SIZE ? BUFFER_SIZE - *off : size;
28543
28544     if (copy_to_user(buf, g_buf + *off, asize) != 0)
28545         return -EFAULT;
28546
28547     *off += asize;
28548
28549     printk(KERN_INFO "Reading...\\n");
28550
28551     return asize;
28552 }
28553
28554 static ssize_t generic_write(struct file *filp, const char *buf, size_t      ↵
28555                     size, loff_t *off)
28555 {
28556     size_t asize;
28557
28558     asize = size + *off > BUFFER_SIZE ? BUFFER_SIZE - *off : size;
28559     if (copy_from_user(g_buf + *off, buf, asize) != 0)
28560         return -EFAULT;
28561
28562     *off += asize;
28563
28564     printk(KERN_INFO "Writing...\\n");
28565
28566     return asize;
28567 }
28568
28569 static loff_t generic_llseek(struct file *filp, loff_t off, int whence)
28570 {
28571     loff_t toff;
28572
28573     switch (whence) {
28574         case SEEK_SET:
28575             toff = off;
28576             break;
```

```
28577     case SEEK_CUR:
28578         toff = filp->f_pos + off;
28579         break;
28580     case SEEK_END:
28581         toff = BUFFER_SIZE + off;
28582         break;
28583     default:
28584         return -EINVAL;
28585     }
28586
28587     if (toff > BUFFER_SIZE || toff < 0)
28588         return -EINVAL;
28589
28590     filp->f_pos = toff;
28591
28592     printk(KERN_INFO "file pointer positioning...\n");
28593
28594     return toff;
28595 }
28596
28597 module_init(generic_init);
28598 module_exit(generic_exit);
28599
28600 # Makelfile
28601
28602     obj-m += $(file).o
28603
28604 all:
28605     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
28606 clean:
28607     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
28608
28609 /* sample.c */
28610
28611 #include <stdio.h>
28612 #include <stdlib.h>
28613 #include <string.h>
28614 #include <fcntl.h>
28615 #include <unistd.h>
28616
28617 void exit_sys(const char *msg);
28618
28619 int main(void)
28620 {
28621     int fd;
28622     char wbuf[100] = "0123456789012345678901234567890";
28623     char rbuf[100];
28624     ssize_t result;
28625
28626     if ((fd = open("generic", O_RDWR)) == -1)
28627         exit_sys("open");
28628
28629     if ((result = write(fd, wbuf, strlen(wbuf))) == -1)
```

```
28630     exit_sys("write");
28631
28632     printf("%ld bytes written...\n", (long)result);
28633
28634     if (lseek(fd, 0, SEEK_SET) == -1)
28635         exit_sys("lseek");
28636
28637     if ((result = read(fd, rbuf, 10)) == -1)
28638         exit_sys("read");
28639     rbuf[result] = '\0';
28640
28641     printf("%ld bytes read: %s\n", result, rbuf);
28642
28643     if (lseek(fd, -5, SEEK_CUR) == -1)
28644         exit_sys("lseek");
28645
28646     if ((result = read(fd, rbuf, 10)) == -1)
28647         exit_sys("read");
28648     rbuf[result] = '\0';
28649
28650     printf("%ld bytes read: %s\n", result, rbuf);
28651
28652     close(fd);
28653
28654     return 0;
28655 }
28656
28657 void exit_sys(const char *msg)
28658 {
28659     perror(msg);
28660
28661     exit(EXIT_FAILURE);
28662 }
28663
28664 /
*-----*
```

28665 Biz şimdiye kadarki örneklerimizde aygit sürücümüzün majör ve minör numarasını batan belirledik. Bunun en önemli sakıncası zaten o numaralı bir aygit sürücünün yüklü olarak bulunuyor olmasıdır. Bu durumda aygit sürücümüz yüklenmeyecektir. Aslında daha doğru bir strateji tersten gitmektedir. Yani önce aygit sürücümüz içerisinde biz boş bir aygit numarasını bulup onu kullanmalıyız. Sonra user moddan bu numaralı bir aygit dosyası yaratmalıyız.

28669 Boş bir aygit numarasını bize veren alloc_chrdev_region isimli bir kernel fonksiyonu vardır. Fonksiyonun parametrik yapısı şöyledir:

28671 int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name);

28673

28674 Fonksiyonun birinci parametresi boş aygit numarasının yerleştirileceği →
dev_t nesnesinin adresini alır. İkinci ve üçüncü parametreler →
başlangıç

28675 minör numarası ve onun sayısıdır. Son parametre ise aygit rüscünün / →
proc/devices dosyasında ve /sys/dev dizininde görüntülenecek ismini →

28676 belirtir. alloc_chrdev_region fonksiyonu zaten register_chrdev_region →
fonksiyonun yaptığına da yapmaktadır. Dolayısıyla bu iki fonksiyondan →
yalnızca biri kullanılmalıdır. Fonksiyon başarı durumunda 0 başarısızlık →
durumunda negatif errno değerine geri döner.

28678

28679 Aygit sürücümüzde alloc_chrdev_region fonksiyonu ile boş bir majör →
numara numaranın bulunup augit sürücümüzün register ettirildiğini →
düşünelim.

28680 Pekiyi biz bu numaraya nasıl bilip bu numaraya uygun aygit dosyası →
yaratacağız? İşte bunun genellikle izlenen yöntem /proc/devices →
dosyasına bakıp

28681 oradan majör numarayı alıp aygit dosyasını yaratmaktadır. Tabii bu manuel →
olarak yapılabılır ancak bir shell script ile otomatize de edilebilir. →

28682 Aşağıdaki load isimli script bu işlemi yapmaktadır.

28683

```
28684 #!/bin/bash
28685
28686 module=$1
28687 mode=666
28688
28689 /sbin/insmod ./${module}.ko ${@:2} || exit 1
28690 major=$(awk "\$2 == \"\$module\" {print \$1}" /proc/devices)
28691 rm -f $module
28692 mknod $module c $major 0
28693 chmod $mode $module
28694
28695 Artık biz bu load script'i ile aygit sürücümüzü yükleyip aygit →  
dosyasımızı yaratacağız. Bu script'i load ismiyle yazıp aşağıdaki gibi →  
x hakkı vermelisiniz:

28696



```
28697 chmod +x load
28698
28699 Çalıştırmayı komut satırı argümanı vererek aşağıdaki gibi yapmalısınız:
28700
28701 ./load generic-char-driver
28702
28703 Aşağıdaki örnekte majör numara alloc_chrdev_region fonksiyonıyla elde →
edilmiştir. Yüklemeyi load scripti ile yapınız.

28704

28705 -----*/

28706


```
28707 /* generic-char-driver.c */
28708
28709 #include <linux/module.h>
28710 #include <linux/kernel.h>
28711 #include <linux/fs.h>
```


```


```

```
28712 #include <linux/cdev.h>
28713 #include <linux/uaccess.h>
28714
28715 MODULE_LICENSE("GPL");
28716 MODULE_DESCRIPTION("General Character Device Driver");
28717 MODULE_AUTHOR("Kaan Aslan");
28718
28719 #define BUFFER_SIZE      128
28720
28721 static int generic_open(struct inode *inodep, struct file *filp);
28722 static int generic_release(struct inode *inodep, struct file *filp);
28723 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
28724                             loff_t *off); ↵
28724 static ssize_t generic_write(struct file *filp, const char *buf, size_t size,
28725                             loff_t *off); ↵
28725 static loff_t generic_llseek(struct file *filp, loff_t off, int whence); ↵
28726
28727 static dev_t g_dev;
28728 static struct cdev *g_cdev;
28729 static struct file_operations g_file_ops = {
28730     .owner = THIS_MODULE,
28731     .open = generic_open,
28732     .release = generic_release,
28733     .read = generic_read,
28734     .write = generic_write,
28735     .llseek = generic_llseek,
28736 };
28737
28738 char g_buf[BUFFER_SIZE];
28739
28740 static int __init generic_init(void)
28741 {
28742     int result;
28743
28744     if ((result = alloc_chrdev_region(&g_dev, 0, 1, "generic-char-driver")) < 0) {
28745         printk(KERN_INFO "Cannot alloc char driver!...\n");
28746         return result;
28747     }
28748
28749     if ((g_cdev = cdev_alloc()) == NULL) {
28750         printk(KERN_INFO "Cannot allocate cdev!..\n");
28751         return -ENOMEM;
28752     }
28753
28754     g_cdev->owner = THIS_MODULE;
28755     g_cdev->ops = &g_file_ops;
28756
28757     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
28758         unregister_chrdev_region(g_dev, 1);
28759         printk(KERN_INFO "Cannot add character device driver!...\n");
28760         return result;
28761     }
```

```
28762
28763     printk(KERN_INFO "Module initialized with %d:%d device number...\n",      ↵
28764             MAJOR(g_dev), MINOR(g_dev));
28765
28766 }
28767
28768 static void __exit generic_exit(void)
28769 {
28770     cdev_del(g_cdev);
28771     unregister_chrdev_region(g_dev, 1);
28772
28773     printk(KERN_INFO "Goodbye...\n");
28774 }
28775
28776 static int generic_open(struct inode *inodep, struct file *filp)
28777 {
28778     printk(KERN_INFO "Generic device opened!..\n");
28779
28780     return 0;
28781 }
28782
28783 static int generic_release(struct inode *inodep, struct file *filp)
28784 {
28785     printk(KERN_INFO "Generic device released!..\n");
28786
28787     return 0;
28788 }
28789
28790 static ssize_t generic_read(struct file *filp, char *buf, size_t size,      ↵
28791     loff_t *off)
28792 {
28793     size_t asize;
28794
28795     asize = size + *off > BUFFER_SIZE ? BUFFER_SIZE - *off : size;
28796
28797     if (copy_to_user(buf, g_buf + *off, asize) != 0)
28798         return -EFAULT;
28799
28800     *off += asize;
28801
28802     printk(KERN_INFO "Reading...\n");
28803
28804     return asize;
28805 }
28806
28807 static ssize_t generic_write(struct file *filp, const char *buf, size_t      ↵
28808     size, loff_t *off)
28809 {
28810     size_t asize;
28811
28812     asize = size + *off > BUFFER_SIZE ? BUFFER_SIZE - *off : size;
28813     if (copy_from_user(g_buf + *off, buf, asize) != 0)
```

```
28812         return -EFAULT;
28813
28814     *off += asize;
28815
28816     printk(KERN_INFO "Writing...\n");
28817
28818     return asize;
28819 }
28820
28821 static loff_t generic_llseek(struct file *filp, loff_t off, int whence)
28822 {
28823     loff_t toff;
28824
28825     switch (whence) {
28826         case SEEK_SET:
28827             toff = off;
28828             break;
28829         case SEEK_CUR:
28830             toff = filp->f_pos + off;
28831             break;
28832         case 2:
28833             toff = BUFFER_SIZE + off;
28834         default:
28835             return -EINVAL;
28836     }
28837
28838     if (toff > BUFFER_SIZE || toff < 0)
28839         return -EINVAL;
28840
28841     filp->f_pos = toff;
28842
28843     printk(KERN_INFO "file pointer positioning...\n");
28844
28845     return toff;
28846 }
28847
28848 module_init(generic_init);
28849 module_exit(generic_exit);
28850
28851 # Makelfile
28852
28853 obj-m += $(file).o
28854
28855 all:
28856     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
28857 clean:
28858     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
28859
28860
28861 /* sample.c */
28862
28863 #include <stdio.h>
28864 #include <stdlib.h>
```

```
28865 #include <string.h>
28866 #include <fcntl.h>
28867 #include <unistd.h>
28868
28869 void exit_sys(const char *msg);
28870
28871 int main(void)
28872 {
28873     int fd;
28874     char wbuf[100] = "0123456789012345678901234567890";
28875     char rbuf[100];
28876     ssize_t result;
28877
28878     if ((fd = open("generic-char-driver", O_RDWR)) == -1)
28879         exit_sys("open");
28880
28881     if ((result = write(fd, wbuf, strlen(wbuf))) == -1)
28882         exit_sys("write");
28883
28884     printf("%ld bytes written...\n", (long)result);
28885
28886     if (lseek(fd, 0, SEEK_SET) == -1)
28887         exit_sys("lseek");
28888
28889     if ((result = read(fd, rbuf, 10)) == -1)
28890         exit_sys("read");
28891     rbuf[result] = '\0';
28892
28893     printf("%ld bytes read: %s\n", result, rbuf);
28894
28895     if (lseek(fd, -5, SEEK_CUR) == -1)
28896         exit_sys("lseek");
28897
28898     if ((result = read(fd, rbuf, 10)) == -1)
28899         exit_sys("read");
28900     rbuf[result] = '\0';
28901
28902     printf("%ld bytes read: %s\n", result, rbuf);
28903
28904     close(fd);
28905
28906     return 0;
28907 }
28908
28909 void exit_sys(const char *msg)
28910 {
28911     perror(msg);
28912
28913     exit(EXIT_FAILURE);
28914 }
28915
28916 /* load */
28917
```

```
28918 #!/bin/bash
28919
28920 module=$1
28921 mode=666
28922
28923 /sbin/insmod ./${module}.ko ${@:2} || exit 1
28924 major=$(awk "\$2 == \"\$module\" {print \$1}" /proc/devices)
28925 rm -f $module
28926 mknod $module c $major 0
28927 chmod $mode $module
28928
28929 /
*-----*
-----*
28930 Aşağıda gelinen noktaya kadar görülmüş olan konulara kullanılarak      ↗
      yazılmış basit bir boru örneği verilmiştir.
28931 Bu boru örneğinde bir proses boruyu yazma modunda açar ve write      ↗
      fonksiyonuyla yazdıkları aygit sürücü içerisindeki bir kuyruğa
28932 yazılır. Diğer proses de read fonksiyonuyla bu kuyruktan okuma yapar. Bu ↗
      gerçekleştirimin özellekleri söyledir:
28933
28934 1) write fonksiyonu borudaki boş alan miktarından daha fazla bilgi      ↗
      yazılmasına çalışılırsa blokeye yol açmaz, -ENOMEM ile geri döner.
28935 2) read fonksiyonu boruda hiçbir bilgi yoksa blokeye yol açmaz 0 ile      ↗
      geri döner. Ancak read diğer durumda okuyabildiği kadar
28936 byte sayısını okuyup onunla geri döner.
28937 3) Aygit sürücünün read/write fonksiyonlarında hiçbir senkronizasyon      ↗
      uygulanmamıştır. Dolayısıyla eşzamanlı işlemlerde tanımsız davranış
28938 ortaya çıkabilir.
28939 4) İki proses de boruyu kapatsa bile boru silinmemektedir.
28940
28941 Aygit sürücüyü önce build edip sonra aşağıdaki gibi yükleyiniz:
28942
28943 make file=pipe-dirver
28944 sudo ./load pipe-driver
28945 -----
-----*/
28946
28947 /* pipe-driver.c */
28948
28949 #include <linux/module.h>
28950 #include <linux/kernel.h>
28951 #include <linux/fs.h>
28952 #include <linux/cdev.h>
28953 #include <linux/uaccess.h>
28954
28955 MODULE_LICENSE("GPL");
28956 MODULE_DESCRIPTION("General Character Device Driver");
28957 MODULE_AUTHOR("Kaan Aslan");
28958
28959 #define MIN(a, b) ((a) < (b) ? (a) : (b))
28960
28961 #define PIPE_SIZE 4096
```

```
28962
28963 static int generic_open(struct inode *inodep, struct file *filp);
28964 static int generic_release(struct inode *inodep, struct file *filp);
28965 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
28966                           loff_t *off);                                ↵
28967 static ssize_t generic_write(struct file *filp, const char *buf, size_t
28968                             size, loff_t *off);                            ↵
28969
28970 static dev_t g_dev;
28971 static struct cdev *g_cdev;
28972 static struct file_operations g_file_ops = {
28973     .owner = THIS_MODULE,
28974     .open = generic_open,
28975     .release = generic_release,
28976     .read = generic_read,
28977     .write = generic_write,
28978 };
28979
28980 static char g_pipebuf[PIPE_SIZE];
28981 static size_t g_head = 0, g_tail = 0;
28982 static size_t g_count = 0;
28983
28984 static int __init generic_init(void)
28985 {
28986     int result;
28987
28988     if ((result = alloc_chrdev_region(&g_dev, 0, 1, "pipe-driver")) < 0) {
28989         printk(KERN_INFO "Cannot alloc char driver!...\n");
28990         return result;
28991     }
28992
28993     if ((g_cdev = cdev_alloc()) == NULL) {
28994         printk(KERN_INFO "Cannot allocate cdev!..\n");
28995         return -ENOMEM;
28996     }
28997
28998     g_cdev->owner = THIS_MODULE;
28999     g_cdev->ops = &g_file_ops;
29000
29001     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
29002         unregister_chrdev_region(g_dev, 1);
29003         printk(KERN_INFO "Cannot add character device driver!...\n");
29004         return result;
29005     }
29006
29007     printk(KERN_INFO "Pipe driver initialized with %d:%d device number...
29008             \n", MAJOR(g_dev), MINOR(g_dev));
29009
29010     return 0;
29011 }
```

```
29012     cdev_del(g_cdev);
29013     unregister_chrdev_region(g_dev, 1);
29014
29015     printk(KERN_INFO "Goodbye...\n");
29016 }
29017
29018 static int generic_open(struct inode *inodep, struct file *filp)
29019 {
29020     printk(KERN_INFO "Pipe device opened!..\n");
29021
29022     return 0;
29023 }
29024
29025 static int generic_release(struct inode *inodep, struct file *filp)
29026 {
29027     printk(KERN_INFO "Pipe device released!..\n");
29028
29029     return 0;
29030 }
29031
29032 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
29033                             loff_t *off)      ↵
29033 {
29034     size_t len;
29035
29036     size = MIN(size, g_count);
29037
29038     if (size == 0)
29039         return 0;
29040
29041     if (g_head >= g_tail)
29042         len = MIN(size, PIPE_SIZE - g_head);
29043     else
29044         len = size;
29045
29046     if (copy_to_user(buf, g_pipebuf + g_head, len) != 0)
29047         return -EFAULT;
29048     if (size > len)
29049         if (copy_to_user(buf + len, g_pipebuf, size - len) != 0)
29050             return -EFAULT;
29051
29052     g_head = (g_head + size) % PIPE_SIZE;
29053     g_count -= size;
29054
29055     return size;
29056 }
29057
29058 static ssize_t generic_write(struct file *filp, const char *buf, size_t      ↵
29059                               size, loff_t *off)
29059 {
29060     size_t len;
29061
29062     if (size > PIPE_SIZE - g_count)
```

```
29063         return -ENOMEM;
29064
29065     if (g_tail >= g_head)
29066         len = MIN(size, PIPE_SIZE - g_tail);
29067     else
29068         len = size;
29069
29070     if (copy_from_user(g_pipebuf + g_tail, buf, len) != 0)
29071         return -EFAULT;
29072
29073     if (size > len)
29074         if (copy_from_user(g_pipebuf, buf + len, size - len) != 0)
29075             return -EFAULT;
29076
29077     g_tail = (g_tail + size) % PIPE_SIZE;
29078     g_count += size;
29079
29080     return size;
29081 }
29082
29083 module_init(generic_init);
29084 module_exit(generic_exit);
29085
29086 # Makelfile
29087
29088 obj-m += $(file).o
29089
29090 all:
29091     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
29092 clean:
29093     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
29094
29095 /* pipeproc1.c */
29096
29097 #include <stdio.h>
29098 #include <stdlib.h>
29099 #include <string.h>
29100 #include <fcntl.h>
29101 #include <unistd.h>
29102
29103 #define PIPE_SIZE      4096
29104
29105 void exit_sys(const char *msg);
29106
29107 int main(void)
29108 {
29109     int fd;
29110     char buf[PIPE_SIZE];
29111     char *str;
29112     size_t len;
29113
29114     if ((fd = open("pipe-driver", O_WRONLY)) == -1)
29115         exit_sys("open");
```

```
29116
29117     for (;;) {
29118         printf("Enter text:");
29119         fflush(stdout);
29120         fgets(buf, PIPE_SIZE, stdin);
29121         if ((str = strchr(buf, '\n')) != NULL)
29122             *str = '\0';
29123         if (!strcmp(buf, "quit"))
29124             break;
29125         len = strlen(buf);
29126         if (write(fd, buf, len) == -1) {
29127             fprintf(stderr, "Cannot write pipe! Maybe pipe is full!\n");
29128             continue;
29129         }
29130         printf("%lu bytes written...\n", (unsigned long)len);
29132     }
29133
29134     close(fd);
29135
29136     return 0;
29137 }
29138
29139 void exit_sys(const char *msg)
29140 {
29141     perror(msg);
29142
29143     exit(EXIT_FAILURE);
29144 }
29145
29146 /* pipeproc2.c */
29147
29148 #include <stdio.h>
29149 #include <stdlib.h>
29150 #include <string.h>
29151 #include <fcntl.h>
29152 #include <unistd.h>
29153
29154 #define PIPE_SIZE      4096
29155
29156 void exit_sys(const char *msg);
29157
29158 int main(void)
29159 {
29160     int fd;
29161     char buf[PIPE_SIZE + 1];
29162     int len;
29163     ssize_t result;
29164
29165     if ((fd = open("pipe-driver", O_RDONLY)) == -1)
29166         exit_sys("open");
29167
29168     for (;;) {
```

```
29169     printf("How many bytes to read? ");
29170     fflush(stdout);
29171     scanf("%d", &len);
29172     if (!len)
29173         break;
29174     if ((result = read(fd, buf, len)) == -1)
29175         exit_sys("read");
29176     buf[result] = '\0';
29177
29178     printf("%ld bytes read: \"%s\"\n", (long)result, buf);
29179 }
29180
29181     close(fd);
29182
29183     return 0;
29184 }
29185
29186 void exit_sys(const char *msg)
29187 {
29188     perror(msg);
29189
29190     exit(EXIT_FAILURE);
29191 }
29192
29193 #!/bin/bash
29194
29195 module=$1
29196 mode=666
29197
29198 /sbin/insmod ./${module}.ko ${@:2} || exit 1
29199 major=$(awk "$2 == \"$module\" {print $1}" /proc/devices)
29200 rm -f $module
29201 mknod $module c $major 0
29202 chmod $mode $module
29203
29204 /
*-----*
-----*
29205     Kernel modda da senkronizasyon nesneleri vardır. Aygit sürücülerde bu
        senkronizasyon nesneleri mecburen çok yoğun kullanılmaktadır. ↗
29206     Çünkü aynı anda aygit sürücü kodları değişik prosesler tarafından
        kullanıldığından buradaki değerler bozulabilir. Yukarıdaki boru
        örneğinde ↗
29207     iki proses aynı anda read/write yaparsa kuyruğu oluşturan veri yapıları
        bozulacaktır. ↗
29208
29209     Biz kernel modda user moddaki senkronizasyon nesnelerini kullanamayız. ↗
        User modda kullandığımız bütün o senkronizasyon nesnelerinin
29210     birer kernel mod karşılıkları vardır. Aslında pthread kütüphanesi de
        arka planda bu mekanizmaları kullanmaktadır. ↗
29211 -----*/
29212
```

```
29213 /  
29214 *-----  
29214 Kernel modda mutex mekanizması 2.6 kernel'ında eklenmiştir. Bu mutex  
29214     mekanizması user moddaki mutex mekanizmasına çok benzemektedir.  
29215 Yine kernel moddaki mutex'in thread temelinde sahliği vardır. Thread'e  
29215     bloke edip sleep kuyruklarında bekletebilmektedir. Kernel mod mutex  
29216 mekanizmasının tipik çalışması şöyledir:  
29217  
29218 1) lock işlemi sırasında maliyersiz compare/jump komutlarıyla mutex'in  
29218     kilitli olup olmadığına bakılır.  
29219 2) mutex kilitliyse biraz spin işlemi yapılır (diğer bir işlemcideki  
29219     proses kilitlenmişse boşuna bloke olmamak için)  
29220 3) spin işleminden sonuç elde eidlemese bloke oluşturulur.  
29221  
29222 Kernel moddaki mutex'ler tipik olarak şöyle kullanılmaktadır:  
29223  
29224 1) mutex DEFINE_MUTEX(name) makrosuyla ya da mutex_init fonksiyonuyla  
29224     ilkdeğerlenir. Örneğin:  
29225  
29226     DEFINE_MUTEX(g_name);  
29227  
29228     ile aşağıdaki işlem işlevsel olarak eşdeğrdir:  
29229  
29230     struct mutex g_mutex;  
29231     ...  
29232     mutex_init(&g_mutex);  
29233  
29234 2) Mutex'i kilitlemek için mutex_lock fonksiyonu kullanılır:  
29235  
29236     void mutex_lock(struct mutex *lock);  
29237  
29238 Mutex'in kilitli olup olmadığı ise mutex_trylock fonksiyonuyla kontrol  
29238     edilebilir:  
29239  
29240     int mutex_trylock(struct mutex *lock);  
29241  
29242 Sinyal geldiğinde uykudaki thread'in uyandırılması için  
29242     mutex_lock_interruptible fonksiyonu kullanılmaktadır:  
29243  
29244     int mutex_lock_interruptible(struct mutex *lock);  
29245  
29246 Fonksiyon başarı normal sonlanmada 0 değerine, sinyal dolayıla  
29246     sonlandığında -EINTR değerine geri dönmektedir.  
29247  
29248 3) Mutex'i unlock etmek için mutex_unlock fonksiyonu kullanılmaktadır:  
29249  
29250     void mutex_unlock(struct mutex *lock);  
29251  
29252 Aşağıdaki örnekte yukarıdaki boru sürücüsü daha güvenli olacak biçimde  
29252     mutex nesneleriyle senkronize edilmiştir.  
29253 -----  
29254 -----
```

```
-----*/
29255
29256 /* pipe-driver.c */
29257
29258 #include <linux/module.h>
29259 #include <linux/kernel.h>
29260 #include <linux/fs.h>
29261 #include <linux/cdev.h>
29262 #include <linux/uaccess.h>
29263 #include <linux/mutex.h>
29264
29265 MODULE_LICENSE("GPL");
29266 MODULE_DESCRIPTION("General Character Device Driver");
29267 MODULE_AUTHOR("Kaan Aslan");
29268
29269 #define MIN(a, b) ((a) < (b) ? (a) : (b))
29270
29271 #define PIPE_SIZE 4096
29272
29273 static int generic_open(struct inode *inodep, struct file *filp);
29274 static int generic_release(struct inode *inodep, struct file *filp);
29275 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
29276                             loff_t *off); ↗
29276 static ssize_t generic_write(struct file *filp, const char *buf, size_t size,
29277                             loff_t *off); ↗
29277
29278 static dev_t g_dev;
29279 static struct cdev *g_cdev;
29280 static struct file_operations g_file_ops = {
29281     .owner = THIS_MODULE,
29282     .open = generic_open,
29283     .release = generic_release,
29284     .read = generic_read,
29285     .write = generic_write,
29286 };
29287
29288 static char g_pipebuf[PIPE_SIZE];
29289 static size_t g_head = 0, g_tail = 0;
29290 static size_t g_count = 0;
29291 static DEFINE_MUTEX(g_mutex);
29292
29293 static int __init generic_init(void)
29294 {
29295     int result;
29296
29297     if ((result = alloc_chrdev_region(&g_dev, 0, 1, "pipe-driver")) < 0) {
29298         printk(KERN_INFO "Cannot alloc char driver!...\n");
29299         return result;
29300     }
29301
29302     if ((g_cdev = cdev_alloc()) == NULL) {
29303         printk(KERN_INFO "Cannot allocate cdev!..\n");
29304         return -ENOMEM;
```

```
29305     }
29306
29307     g_cdev->owner = THIS_MODULE;
29308     g_cdev->ops = &g_file_ops;
29309
29310    if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
29311        unregister_chrdev_region(g_dev, 1);
29312        printk(KERN_INFO "Cannot add character device driver!...\n");
29313        return result;
29314    }
29315
29316    printk(KERN_INFO "Pipe driver initialized with %d:%d device number...
29317          \n", MAJOR(g_dev), MINOR(g_dev));
29318
29319    return 0;
29320}
29321 static void __exit generic_exit(void)
29322 {
29323     cdev_del(g_cdev);
29324     unregister_chrdev_region(g_dev, 1);
29325
29326     printk(KERN_INFO "Goodbye...\n");
29327 }
29328
29329 static int generic_open(struct inode *inodep, struct file *filp)
29330 {
29331     printk(KERN_INFO "Pipe device opened!...\n");
29332
29333     return 0;
29334 }
29335
29336 static int generic_release(struct inode *inodep, struct file *filp)
29337 {
29338     printk(KERN_INFO "Pipe device released!...\n");
29339
29340     return 0;
29341 }
29342
29343 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
29344                             loff_t *off)
29345 {
29346     size_t len;
29347     ssize_t result;
29348
29349     if (mutex_lock_interruptible(&g_mutex))
29350         return -ERESTARTSYS;
29351
29352     size = MIN(size, g_count);
29353
29354     if (size == 0) {
29355         result = 0;
29356         goto EXIT;
```

```
29356     }
29357
29358     if (g_head >= g_tail)
29359         len = MIN(size, PIPE_SIZE - g_head);
29360     else
29361         len = size;
29362
29363     if (copy_to_user(buf, g_pipebuf + g_head, len) != 0) {
29364         result = -EFAULT;
29365         goto EXIT;
29366     }
29367
29368     if (size > len)
29369         if (copy_to_user(buf + len, g_pipebuf, size - len) != 0) {
29370             result = -EFAULT;
29371             goto EXIT;
29372         }
29373
29374     g_head = (g_head + size) % PIPE_SIZE;
29375     g_count -= size;
29376
29377     result = size;
29378 EXIT:
29379     mutex_unlock(&g_mutex);
29380
29381     return result;
29382 }
29383
29384 static ssize_t generic_write(struct file *filp, const char *buf, size_t      ↴
29385     size, loff_t *off)
29386 {
29387     size_t len;
29388     ssize_t result;
29389
29390     if (mutex_lock_interruptible(&g_mutex))
29391         return -ERESTARTSYS;
29392
29393     if (size > PIPE_SIZE - g_count)    {
29394         result = -ENOMEM;
29395         goto EXIT;
29396     }
29397
29398     if (g_tail >= g_head)
29399         len = MIN(size, PIPE_SIZE - g_tail);
29400     else
29401         len = size;
29402
29403     if (copy_from_user(g_pipebuf + g_tail, buf, len) != 0) {
29404         result = -EFAULT;
29405         goto EXIT;
29406     }
29407     if (size > len)
```

```
29408     if (copy_from_user(g_pipebuf, buf + len, size - len) != 0) {
29409         result = -EFAULT;
29410         goto EXIT;
29411     }
29412
29413     g_tail = (g_tail + size ) % PIPE_SIZE;
29414     g_count += size;
29415
29416     result = size;
29417
29418     mutex_unlock(&g_mutex);
29419 EXIT:
29420     return result;
29421 }
29422
29423 module_init(generic_init);
29424 module_exit(generic_exit);
29425
29426 # Makelfile
29427
29428 obj-m += $(file).o
29429
29430 all:
29431     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
29432 clean:
29433     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
29434
29435 /* pipeproc1.c */
29436
29437 #include <stdio.h>
29438 #include <stdlib.h>
29439 #include <string.h>
29440 #include <fcntl.h>
29441 #include <unistd.h>
29442
29443 #define PIPE_SIZE      4096
29444
29445 void exit_sys(const char *msg);
29446
29447 int main(void)
29448 {
29449     int fd;
29450     char buf[PIPE_SIZE];
29451     char *str;
29452     size_t len;
29453
29454     if ((fd = open("pipe-driver", O_WRONLY)) == -1)
29455         exit_sys("open");
29456
29457     for (;;) {
29458         printf("Enter text:");
29459         fflush(stdout);
29460         fgets(buf, PIPE_SIZE, stdin);
```

```
29461     if ((str = strchr(buf, '\n')) != NULL)
29462         *str = '\0';
29463     if (!strcmp(buf, "quit"))
29464         break;
29465     len = strlen(buf);
29466     if (write(fd, buf, len) == -1) {
29467         fprintf(stderr, "Cannot write pipe! Maybe pipe is full!\n");
29468         continue;
29469     }
29470
29471     printf("%lu bytes written...\n", (unsigned long)len);
29472 }
29473
29474     close(fd);
29475
29476     return 0;
29477 }
29478
29479 void exit_sys(const char *msg)
29480 {
29481     perror(msg);
29482
29483     exit(EXIT_FAILURE);
29484 }
29485
29486 /* pipeproc2.c */
29487
29488 #include <stdio.h>
29489 #include <stdlib.h>
29490 #include <string.h>
29491 #include <fcntl.h>
29492 #include <unistd.h>
29493
29494 #define PIPE_SIZE      4096
29495
29496 void exit_sys(const char *msg);
29497
29498 int main(void)
29499 {
29500     int fd;
29501     char buf[PIPE_SIZE + 1];
29502     int len;
29503     ssize_t result;
29504
29505     if ((fd = open("pipe-driver", O_RDONLY)) == -1)
29506         exit_sys("open");
29507
29508     for (;;) {
29509         printf("How many bytes to read? ");
29510         fflush(stdout);
29511         scanf("%d", &len);
29512         if (!len)
29513             break;
```

```
29514     if ((result = read(fd, buf, len)) == -1)
29515         exit_sys("read");
29516     buf[result] = '\0';
29517
29518     printf("%ld bytes read: \"%s\"\n", (long)result, buf);
29519 }
29520
29521     close(fd);
29522
29523     return 0;
29524 }
29525
29526 void exit_sys(const char *msg)
29527 {
29528     perror(msg);
29529
29530     exit(EXIT_FAILURE);
29531 }
29532
29533 #!/bin/bash
29534
29535 module=$1
29536 mode=666
29537
29538 /sbin/insmod ./${module}.ko ${@:2} || exit 1
29539 major=$(awk "$2 == \"$module\" {print $1}" /proc/devices)
29540 rm -f $module
29541 mknod $module c $major 0
29542 chmod $mode $module
29543
29544 /
*-----*-----*-----*
```

29545 Kernel semaphore nesneleri de tamamen user modda kullandığımız semaphore nesneleri gibidir. Bunların sayacı vardır. Sayaç 0'dan büyükse semaphore açık durumdadır. Kritik koda girildiğinde sayaç eksiltilir. Sayaç 0 olduğunda thread bloke edilir. Yine bloke işleminde biraz spin işlemi yapılip sonra bloke uygulanmaktadır. Semaphore nesneleri şöyle kullanılmaktadır:

29549 1) Semaphore nesnesi struct semaphore isimli bir yapıyla temsil edilmiştir. Bir semaphore nesnesi DEFINE_SEMAPHORE(name) makrosuyla aşağıdaki gibi oluşturulabilir.

29551

29552 DEFINE_SEMAPHORE(g_sem);

29553

29554 Bu biçimde yaratılan semaphore nesnesinin sayaç değeri başlangıçta 1'dir. Semaphore nesneleri sema_init fonksiyonuyla da yaratılabilmektedir:

29555

29556 struct semaphore g_sem;

29557 ...
29558 sema_init(&g_sem, 1);

29559
29560 Fonksiyonun ikinci parametresi başlangıç sayaç numarasıdır.
29561
29562 2) Kritik kod down ve up fonksiyonları arasına alınır. down →
fonksiyonları sayacı bir eksilterek kritik koda giriş yapar. up →
fonksiyonu ise
29563 sayacı bir artırmaktadır. Fonksiyonların prototipleri şöyledir:
29564
29565 void down(struct semaphore *sem);
29566 int down_interruptible(struct semaphore *sem);
29567 int down_killable(struct semaphore *sem);
29568 int down_trylock(struct semaphore *sem);
29569 int down_timeout(struct semaphore *sem, long jiffies);
29570 void up(struct semaphore *sem);
29571
29572 Yine down_interruptible sinyal geldiğinde sonlanabilen bir fonksiyondur. →
down_killable yalnızca SIGKILL sinyali geldiğinde sonlanabilir.
29573 down fonksiyonu sinyal gelse de sonlanmaz. down_trylock yine nesnenin →
açık olup olmadığına bakmaktadır. down_timeout en kötü olasılıkla →
belli miktar
29574 jiffy zamanı kadar blokeye yol açar.
29575
29576 interruptible down fonksiyonları normal sonlanmada 0 değerine, sinyal →
yoluyla sonlanmada -ERESTARTSYS değeri ile geri dönmektedir.
29577 Normal uygulama eğer bu fonksiyonlar -ERESTARTSYS ile geri dönerse aygit →
sürücüdeki fonksiyonu da aynı değerle geri döndürmektedir. Zaten VFS
29578 bu -ERESTARTSYS geri dönüş değerini aldığında asıl sistem fonksiyonunu - →
EINTR değeri ile geri döndürmektedir. Bu da tabii POSIX →
fonksiyonlarının
29579 başarısız olup errno değerinin EINTR biçiminde set edilmesine yol →
açmaktadır.
29580
29581 Yukarıdaki boru örneğinde biz mutex nesnesi yerine binary semaphore →
nesnesi de kullanabilirdik.
29582
29583 -----*/
29584
29585 /* pipe-driver.c */
29586
29587 #include <linux/module.h>
29588 #include <linux/kernel.h>
29589 #include <linux/fs.h>
29590 #include <linux/cdev.h>
29591 #include <linux/uaccess.h>
29592 #include <linux/semafor.h>
29593
29594 MODULE_LICENSE("GPL");
29595 MODULE_DESCRIPTION("General Character Device Driver");
29596 MODULE_AUTHOR("Kaan Aslan");
29597
29598 #define MIN(a, b) ((a) < (b) ? (a) : (b))
29599

```
29600 #define PIPE_SIZE      4096
29601
29602 static int generic_open(struct inode *inodep, struct file *filp);
29603 static int generic_release(struct inode *inodep, struct file *filp);
29604 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
29605                             loff_t *off);
29606 static ssize_t generic_write(struct file *filp, const char *buf, size_t
29607                             size, loff_t *off);
29608
29609 static dev_t g_dev;
29610 static struct cdev *g_cdev;
29611 static struct file_operations g_file_ops = {
29612     .owner = THIS_MODULE,
29613     .open = generic_open,
29614     .release = generic_release,
29615     .read = generic_read,
29616     .write = generic_write,
29617 };
29618
29619 static char g_pipebuf[PIPE_SIZE];
29620 static size_t g_head = 0, g_tail = 0;
29621 static size_t g_count = 0;
29622 static DEFINE_SEMAPHORE(g_sem);
29623
29624 static int __init generic_init(void)
29625 {
29626     int result;
29627
29628     if ((result = alloc_chrdev_region(&g_dev, 0, 1, "pipe-driver")) < 0) {
29629         printk(KERN_INFO "Cannot alloc char driver!...\n");
29630         return result;
29631     }
29632
29633     if ((g_cdev = cdev_alloc()) == NULL) {
29634         printk(KERN_INFO "Cannot allocate cdev!..\n");
29635         return -ENOMEM;
29636     }
29637
29638     g_cdev->owner = THIS_MODULE;
29639     g_cdev->ops = &g_file_ops;
29640
29641     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
29642         unregister_chrdev_region(g_dev, 1);
29643         printk(KERN_INFO "Cannot add character device driver!...\n");
29644         return result;
29645     }
29646
29647     printk(KERN_INFO "Pipe driver initialized with %d:%d device number...
29648             \n", MAJOR(g_dev), MINOR(g_dev));
29649 }
```

```
29650 static void __exit generic_exit(void)
29651 {
29652     cdev_del(g_cdev);
29653     unregister_chrdev_region(g_dev, 1);
29654
29655     printk(KERN_INFO "Goodbye...\n");
29656 }
29657
29658 static int generic_open(struct inode *inodep, struct file *filp)
29659 {
29660     printk(KERN_INFO "Pipe device opened!..\n");
29661
29662     return 0;
29663 }
29664
29665 static int generic_release(struct inode *inodep, struct file *filp)
29666 {
29667     printk(KERN_INFO "Pipe device released!..\n");
29668
29669     return 0;
29670 }
29671
29672 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
29673                             loff_t *off)
29674 {
29675     size_t len;
29676     ssize_t result;
29677
29678     if (down_interruptible(&g_sem))
29679         return -ERESTARTSYS;
29680
29681     size = MIN(size, g_count);
29682
29683     if (size == 0) {
29684         result = 0;
29685         goto EXIT;
29686     }
29687
29688     if (g_head >= g_tail)
29689         len = MIN(size, PIPE_SIZE - g_head);
29690     else
29691         len = size;
29692
29693     if (copy_to_user(buf, g_pipebuf + g_head, len) != 0) {
29694         result = -EFAULT;
29695         goto EXIT;
29696     }
29697
29698     if (size > len)
29699         if (copy_to_user(buf + len, g_pipebuf, size - len) != 0) {
29700             result = -EFAULT;
29701             goto EXIT;
29702         }
```

```
29702
29703     g_head = (g_head + size ) % PIPE_SIZE;
29704     g_count -= size;
29705
29706     result = size;
29707 EXIT:
29708     up(&g_sem);
29709
29710     return result;
29711 }
29712
29713 static ssize_t generic_write(struct file *filp, const char *buf, size_t      ↵
29714     size, loff_t *off)
29715 {
29716     size_t len;
29717     ssize_t result;
29718
29719     if (down_interruptible(&g_sem))
29720         return -ERESTARTSYS;
29721
29722     if (size > PIPE_SIZE - g_count)    {
29723         result = -ENOMEM;
29724         goto EXIT;
29725     }
29726
29727     if (g_tail >= g_head)
29728         len = MIN(size, PIPE_SIZE - g_tail);
29729     else
29730         len = size;
29731
29732     if (copy_from_user(g_pipebuf + g_tail, buf, len) != 0) {
29733         result = -EFAULT;
29734         goto EXIT;
29735     }
29736
29737     if (copy_from_user(g_pipebuf, buf + len, size - len) != 0) {
29738         result = -EFAULT;
29739         goto EXIT;
29740     }
29741
29742     g_tail = (g_tail + size ) % PIPE_SIZE;
29743     g_count += size;
29744
29745     result = size;
29746
29747     up(&g_sem);
29748 EXIT:
29749     return result;
29750 }
29751
29752 module_init(generic_init);
29753 module_exit(generic_exit);
```

```
29754 # Makelfile
29755 #include <sys/types.h>
29756 #include <sys/stat.h>
29757 #include <fcntl.h>
29758 #include <errno.h>
29759 #include <stropts.h>
29760 #include <stropts.h>
29761 #include <stropts.h>
29762 #include <stropts.h>
29763 #include <stropts.h>
29764 /* pipeproc1.c */
29765
29766 #include <stdio.h>
29767 #include <stdlib.h>
29768 #include <string.h>
29769 #include <fcntl.h>
29770 #include <unistd.h>
29771
29772 #define PIPE_SIZE      4096
29773
29774 void exit_sys(const char *msg);
29775
29776 int main(void)
29777 {
29778     int fd;
29779     char buf[PIPE_SIZE];
29780     char *str;
29781     size_t len;
29782
29783     if ((fd = open("pipe-driver", O_WRONLY)) == -1)
29784         exit_sys("open");
29785
29786     for (;;) {
29787         printf("Enter text:");
29788         fflush(stdout);
29789         fgets(buf, PIPE_SIZE, stdin);
29790         if ((str = strchr(buf, '\n')) != NULL)
29791             *str = '\0';
29792         if (!strcmp(buf, "quit"))
29793             break;
29794         len = strlen(buf);
29795         if (write(fd, buf, len) == -1) {
29796             fprintf(stderr, "Cannot write pipe! Maybe pipe is full!\n");
29797             continue;
29798         }
29799
29800         printf("%lu bytes written...\n", (unsigned long)len);
29801     }
29802
29803     close(fd);
29804
29805     return 0;
29806 }
```

```
29807
29808 void exit_sys(const char *msg)
29809 {
29810     perror(msg);
29811
29812     exit(EXIT_FAILURE);
29813 }
29814
29815 /* pipeproc2.c */
29816
29817 #include <stdio.h>
29818 #include <stdlib.h>
29819 #include <string.h>
29820 #include <fcntl.h>
29821 #include <unistd.h>
29822
29823 #define PIPE_SIZE      4096
29824
29825 void exit_sys(const char *msg);
29826
29827 int main(void)
29828 {
29829     int fd;
29830     char buf[PIPE_SIZE + 1];
29831     int len;
29832     ssize_t result;
29833
29834     if ((fd = open("pipe-driver", O_RDONLY)) == -1)
29835         exit_sys("open");
29836
29837     for (;;) {
29838         printf("How many bytes to read? ");
29839         fflush(stdout);
29840         scanf("%d", &len);
29841         if (!len)
29842             break;
29843         if ((result = read(fd, buf, len)) == -1)
29844             exit_sys("read");
29845         buf[result] = '\0';
29846
29847         printf("%ld bytes read: \"%s\"\n", (long)result, buf);
29848     }
29849
29850     close(fd);
29851
29852     return 0;
29853 }
29854
29855 void exit_sys(const char *msg)
29856 {
29857     perror(msg);
29858     exit(EXIT_FAILURE);
29859 }
```

```
29860 }
29861
29862 #!/bin/bash
29863
29864 module=$1
29865 mode=666
29866
29867 /sbin/insmod ./${module}.ko ${@:2} || exit 1
29868 major=$(awk "$2 == \"$module\" {print \$1}" /proc/devices)
29869 rm -f $module
29870 mknod $module c $major 0
29871 chmod $mode $module
29872
29873 /
*-----*
-----*
29874 Tipki thread'lerde olduğu gibi aygit sürücülerde de basit atama,
29875 artırma, eksiltme gibi işlemlerin atomic yapılması sağlayan
29876 özel fonksiyonlar vardır. Aslında bu işlemler thread konusunda görmüş
29877 olduğumuz gcc'nin built-in atomic fonksiyonlarıyla
29878 yaplabilir. Ancak çekirdek içerisindeki fonksiyonların kullanılması uyum
29879 bakımından daha uygundur. Bu fonksiyonların hepsi
29880 nesneyi atomic_t türü biçiminde istemektedir. Bu aslında içerisinde
29881 yalnızca int bir nesne olan bir yapı türüdür. Bu yapı nesnesinin
29882 içerisindeki değeri alan atomic_read isimli bir fonksiyon da vardır.
29883 Atomic fonksiyonlarının bazıları şunlardır:
29884
29885 atomic_set
29886 atomic_add
29887 atomic_sub
29888 atomic_inc
29889 atomic_dec
29890 ....
29891 -----*/
29892
29893
29894 Bir thread nasıl bloke olmaktadır? Aslında bloke olma demek kabaca ilgi
29895 thread'i (yani task_struct nesnesini) Run kuyruğundan
29896 çıkartarak bir wait kuyruğuna yerleştirmek demektir. Pekiyi wait
29897 kuyrukları nerededir ve biz bir thread'i wait kuyruklarına nasıl
29898 yerleştirip
29899 yeniden run kuyruğuna atayabiliriz?
29900
29901 İşte aslında wait kuyrukları istenildiği kadar çok olabilir. Her aygit
29902 sürücü kendi wait kuyruğunu yaratabilir. Kernel içeisinde
29903 wait kuyruğu yaratan mekanizmalar vardır. Bir thread'i (yani task_struct
29904 nesnesini) run kuyruğundan wait kuyruğuna, wait kuyruğundan run
29905 kuyruğuna yerlestiren hazır fonksiyonlar bulunmaktadır.
29906
29907
```

29898 Wait kuyrukları `wait_queue_head_t` isimli yapıyla temsil edilmektedir. ↗
Bir wait kuyruğu `DECLARE_WAIT_QUEUE_HEAD(name)` makrosuyla ↗
oluşturulabilir:

29899
29900 `DECLARE_WAIT_QUEUE_HEAD(g_wq);`

29901
29902 Ya da nesne tanımlanıp `init_waitqueue_head` fonksiyonuyla da ↗
ilkdeğerlenebilir:

29903
29904 `wait_queue_head_t g_wq;`
29905 `...`
29906 `init_waitqueue_head(&g_wq);`

29907
29908 Bir `thread`'i (yani `task_struct` nesnesini) `run` kuyruğundan çıkartıp ↗
istenilen wait kuyruğuna yerleştirme işlemi `event_wait` makrolarıyla ↗
gerçekleştirilmektedir. Temel `wait_event` makroları şunlardır:

29910
29911 `wait_event(wq_head, condition);`
29912 `wait_event_interruptible(wq_head, condition);`
29913 `wait_event_killable(wq_head, condition);`
29914 `wait_event_timeout(wq_head, condition, timeout);`
29915 `wait_event_interruptible_timeout(wq_head, condition, timeout);`
29916 `wait_event_interruptible_exclusive(wq_head, condition);`

29917
29918 `wait_event` fonksiyonu "uninterruptible" biçimdedir. Yani o andaki `thread` ↗
akışını `wait` kuyruğuna yerleştirir. Dolayısıyla `run` kuyruğundan ↗
çıkartır.

29919 Fakat sinyal oluştduğunda `thread` uykudan uyanmaz. ↗
`wait_queue_interruptible` makrosu ise aynı işlemi "interruptible" ↗
olarak yapmaktadır. Yani sinyal ↗

29920 geldiğinde `thread` uyandırılır. `wait_event_killable` makrosu yalnızca ↗
`SIGKILL` sinyali için `thread`'i uyandırır. `wait_event_timeout` ise belli ↗
bir jiffy değerini ↗

29921 zaman aşımı olarak kullanır. Makrolardaki `condition` (koşul) parametresi ↗
aslında `bool` bir ifade biçiminde oluşturulmakadır. Bu ifade ya sıfır ↗
olur ya da sıfır dışı olur. Bu koşul ifadesi "yanık kalmak için bir" ↗
koşul" belirtmektedir. Ancak bu koşul uyandırma koşulu değildir. ↗
Uyanık kalma koşuludur.

29922 Thread uyandırılır, sonra uyanmış olan `thread` bu koşula bakar. Koşul ↗
sağlanmıyorsa (`condition == 0` ise) yeniden uyutulur. Tabii aslında ↗
`thread` henüz ↗

29923 uyutulmadan da bu koşula bakılmaktadır. Bu koşul zaten sağlanıyorsa ↗
`thread` hiç uyutulmamaktadır.

29924
29925
29926 `wait_event_interruptible_exclusive` (bunun `interruptible` olmayan biçimini ↗
yoktur) fonksiyonu Linux çekirdeklerine 2.6'ının belli sürümünden ↗
sonra sokulmuştur.

29927 Yine bu fonksiyonla birlikte aşağıda ele alınan `wake_up_xxxx_nr` ↗
fonksiyonları da eklenmiştir. Bir prosesin `exclusive` olarak `wait` ↗
kuyruğuna yerleştirilmes

29928 onlardan belli sayıda olanların uyandırılabilmesini sağlamaktadır.

29929
29930 O halde `wait_event` makroları o andaki `thread`'i çizelgeden (yani `run` ↗

29931 kuyruğundan) çıkartıp wait kuyruğuna yerleştirdikten sonra "task
switch" işlemini ↗
29932 yapar. Task switch işlemi sonrasında artık run kuyruğundaki yeni bir ↗
thread çalışır.
29933 Wait kuyruğundaki thread (yani task_struct nesnesi) yeniden run ↗
kuyruğuna wake_up makrolarıyla sokulmaktadır. wait_event ↗
makrolarındaki koşula
29934 wake_up makroları bakmaz. wake_up makroları yalnızca thread'i wait ↗
kuyruklarından run kuyruğuna taşımaktadır. Koşula uyandırılmış ↗
thread'in kendisi
29935 bilmektedir. Eğer koşul sağlanmıyorsa yeniden uyutulmaktadır. (Zaten ↗
belli bir koşula bakarak uyandırma gerçekleştiririm olarak mümkün ↗
değildir.)
29936 Durum böyle olsa da aslında sonuçta koşulu sağlamayan thread'ler yeniden ↗
uyutulduklarına göre gerçek anlamda uyandırılmış olmamaktadır. ↗
Kullanılan
29937 wake_up makroları şunlardır:
29938
29939 wake_up(wq_head);
29940 wake_up_nr(wq_head, nr);
29941 wake_up_all(wq_head);
29942 wake_up_interruptible(wq_head);
29943 wake_up_interruptible_nr(wq_head, nr);
29944 wake_up_interruptible_all(wq_head);
29945
29946 Buradaki sonu xxx_interruptible ile biten isimdeki makrolar "sinyalle ↗
kesilebilir" olanları uyandırma işini yapar. Sonu xxx_nr ile biten ↗
makrolar ise
29947 wait kuyruğundaki belki sayıda exclusive bekleyen (yani ↗
wait_event_interruptible_exclusive ile bekleyen) thread'i ↗
uyandırmaktadır. wake_up_all makrosu
29948 wait kuyruğundaki tüm thread'leri uyandırır. wake_up ve ↗
wake_up_interruptible makroları kuyruktaki tek bir exclusive thread'i ↗
uyandırmaktadır.
29949 wake_up makrolarının "interruptible" olanları yalnızca "interruptible" ↗
olarak wait kuyruğunda bekleyenleri (yani wait_event_interruptible ile ↗
bekleyenleri) uyandırmaktadır.
29950 Eskiden prosesi uyandıran wake_up fonksiyonlarının yalnızca wait ↗
kuyruğundaki tüm prosesleri uyandıran biçimleri vardı. Sonra belki ↗
sayıda exclusive bekleyen prosesi uyandıran
29951 wake_up fonksiyonları da çekirdeğe eklenmiştir. Belli sayıda exclusive ↗
prosesi uayndıramın makrolar exclusive olmayan prosesleri de ↗
uyandırmaktadır.
29952 Programcı wake_up makrolarına wait_queue_head_t nesnelerinin adreslerini ↗
geçirmelidir. (Halbuki wait_event makrolarına
29953 programcı nesnelerin kendisini geçirmektedir. Adresi makrolar ↗
almaktadır.)
29954
29955 -----*/ ↗
29956 / ↗
29957 / ↗

```
*-----  
-----  
29958 Aşağıdaki örnekte aygıt sürücünün read fonksiyonunda proses      ↵  
    wait_event_interruptible fonksiyonu ile bloke edilerek yaratılmış olan ↵  
    wait  
29959 kuyruğuna aktarılmıştır. write fonksiyonu ise prosesi wait kuyruğundan ↵  
    yeniden wake_up fonksiyonuyla run kuyruğuna aktarmaktadır.  
29960 -----*/  
29961  
29962 /* generic-char-driver.c */  
29963  
29964 #include <linux/module.h>  
29965 #include <linux/kernel.h>  
29966 #include <linux/fs.h>  
29967 #include <linux/cdev.h>  
29968 #include <linux/wait.h>  
29969  
29970 MODULE_LICENSE("GPL");  
29971 MODULE_DESCRIPTION("General Character Device Driver");  
29972 MODULE_AUTHOR("Kaan Aslan");  
29973  
29974 static int generic_open(struct inode *inodep, struct file *filp);  
29975 static int generic_release(struct inode *inodep, struct file *filp);  
29976 static ssize_t generic_read(struct file *filp, char *buf, size_t size,  
     loff_t *off);  
29977 static ssize_t generic_write(struct file *filp, const char *buf, size_t  
     size, loff_t *off);  
29978  
29979 static dev_t g_dev;  
29980 static struct cdev *g_cdev;  
29981 static struct file_operations g_file_ops = {  
29982     .owner = THIS_MODULE,  
29983     .open = generic_open,  
29984     .release = generic_release,  
29985     .read = generic_read,  
29986     .write = generic_write,  
29987 };  
29988 static DECLARE_WAIT_QUEUE_HEAD(g_wq);  
29989 static int g_cond;  
29990  
29991 static int __init generic_init(void)  
29992 {  
29993     int result;  
29994  
29995     if ((result = alloc_chrdev_region(&g_dev, 0, 1, "generic-char-driver")) ↵  
         < 0) {  
29996         printk(KERN_INFO "Cannot alloc char driver!...\n");  
29997         return result;  
29998     }  
29999  
30000     if ((g_cdev = cdev_alloc()) == NULL) {  
30001         printk(KERN_INFO "Cannot allocate cdev!..\n");
```

```
30002         return -ENOMEM;
30003     }
30004
30005     g_cdev->owner = THIS_MODULE;
30006     g_cdev->ops = &g_file_ops;
30007
30008     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
30009         unregister_chrdev_region(g_dev, 1);
30010         printk(KERN_INFO "Cannot add character device driver!...\n");
30011         return result;
30012     }
30013
30014     printk(KERN_INFO "Module initialized with %d:%d device number...\\n",
30015           MAJOR(g_dev), MINOR(g_dev));
30016
30017     return 0;
30018 }
30019 static void __exit generic_exit(void)
30020 {
30021     cdev_del(g_cdev);
30022     unregister_chrdev_region(g_dev, 1);
30023
30024     printk(KERN_INFO "Goodbye...\\n");
30025 }
30026
30027 static int generic_open(struct inode *inodep, struct file *filp)
30028 {
30029     printk(KERN_INFO "Pipe device opened!..\\n");
30030
30031     return 0;
30032 }
30033
30034 static int generic_release(struct inode *inodep, struct file *filp)
30035 {
30036     printk(KERN_INFO "Pipe device released!..\\n");
30037
30038     return 0;
30039 }
30040
30041 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
30042                             loff_t *off)
30043 {
30044     printk(KERN_INFO "Process sleeping!..\\n");
30045
30046     g_cond = 0;
30047     if (wait_event_interruptible(g_wq, g_cond != 0))
30048         return -ERESTARTSYS;
30049
30050     printk(KERN_INFO "Process waking up!..\\n");
30051
30052     return 0;
30053 }
```

```
30053
30054 static ssize_t generic_write(struct file *filp, const char *buf, size_t      ↵
      size, loff_t *off)
30055 {
30056     g_cond = 1;
30057     wake_up_all(&g_wq);
30058
30059     return 0;
30060 }
30061
30062 module_init(generic_init);
30063 module_exit(generic_exit);
30064
30065 module_init(generic_init);
30066 module_exit(generic_exit);
30067
30068 # Makelfile
30069
30070 obj-m += $(file).o
30071
30072 all:
30073     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
30074 clean:
30075     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
30076
30077 /* ddproc.1 */
30078
30079 #include <stdio.h>
30080 #include <stdlib.h>
30081 #include <fcntl.h>
30082 #include <unistd.h>
30083
30084 void exit_sys(const char *msg);
30085
30086 int main(void)
30087 {
30088     int fd;
30089     char ch;
30090
30091     if ((fd = open("generic-char-driver", O_RDONLY)) == -1)
30092         exit_sys("open");
30093
30094     printf("read call begins...\n");
30095
30096     if (read(fd, &ch, 1) == -1)
30097         exit_sys("read");
30098
30099     printf("read call ends...\n");
30100
30101     close(fd);
30102
30103     return 0;
30104 }
```

```
30105
30106 void exit_sys(const char *msg)
30107 {
30108     perror(msg);
30109
30110     exit(EXIT_FAILURE);
30111 }
30112
30113 /* ddproc2.c */
30114
30115 #include <stdio.h>
30116 #include <stdlib.h>
30117 #include <fcntl.h>
30118 #include <unistd.h>
30119
30120 void exit_sys(const char *msg);
30121
30122 int main(void)
30123 {
30124     int fd;
30125
30126     if ((fd = open("generic-char-driver", O_WRONLY)) == -1)
30127         exit_sys("open");
30128
30129     printf("Press ENTER to write!..\n");
30130     getchar();
30131
30132     if (write(fd, "a", 1) == -1)
30133         exit_sys("read");
30134
30135     close(fd);
30136
30137     return 0;
30138 }
30139
30140 void exit_sys(const char *msg)
30141 {
30142     perror(msg);
30143
30144     exit(EXIT_FAILURE);
30145 }
30146
30147#!/bin/bash
30148
30149 module=$1
30150 mode=666
30151
30152 /sbin/insmod ./${module}.ko ${@:2} || exit 1
30153 major=$(awk "\$2 == \"\$module\" {print \$1}" /proc/devices)
30154 rm -f $module
30155 mknod $module c $major 0
30156 chmod $mode $module
30157
```

```
30158  /
30159      *-----*
30159      Aşağıdai kodda wait kuyruğunda bekleyen proseslerden yalnızca 1 tanesini
30159      wake_up makrosuyla çıkartan kod verilmiştir.
30160  -----*/
30161
30162 /* generic-char-driver.c */
30163
30164 #include <linux/module.h>
30165 #include <linux/kernel.h>
30166 #include <linux/fs.h>
30167 #include <linux/cdev.h>
30168 #include <linux/wait.h>
30169
30170 MODULE_LICENSE("GPL");
30171 MODULE_DESCRIPTION("General Character Device Driver");
30172 MODULE_AUTHOR("Kaan Aslan");
30173
30174 static int generic_open(struct inode *inodep, struct file *filp);
30175 static int generic_release(struct inode *inodep, struct file *filp);
30176 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
30176     loff_t *off);
30177 static ssize_t generic_write(struct file *filp, const char *buf, size_t
30177     size, loff_t *off);
30178
30179 static dev_t g_dev;
30180 static struct cdev *g_cdev;
30181 static struct file_operations g_file_ops = {
30182     .owner = THIS_MODULE,
30183     .open = generic_open,
30184     .release = generic_release,
30185     .read = generic_read,
30186     .write = generic_write,
30187 };
30188 static DECLARE_WAIT_QUEUE_HEAD(g_wq);
30189 static int g_cond;
30190
30191 static int __init generic_init(void)
30192 {
30193     int result;
30194
30195     if ((result = alloc_chrdev_region(&g_dev, 0, 1, "generic-char-driver")) < 0) {
30196         printk(KERN_INFO "Cannot alloc char driver!...\n");
30197         return result;
30198     }
30199
30200     if ((g_cdev = cdev_alloc()) == NULL) {
30201         printk(KERN_INFO "Cannot allocate cdev!..\n");
30202         return -ENOMEM;
30203     }
```

```
30204
30205     g_cdev->owner = THIS_MODULE;
30206     g_cdev->ops = &g_file_ops;
30207
30208     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
30209         unregister_chrdev_region(g_dev, 1);
30210         printk(KERN_INFO "Cannot add character device driver!...\n");
30211         return result;
30212     }
30213
30214     printk(KERN_INFO "Module initialized with %d:%d device number...\n",
30215           MAJOR(g_dev), MINOR(g_dev));
30216
30217     return 0;
30218 }
30219 static void __exit generic_exit(void)
30220 {
30221     cdev_del(g_cdev);
30222     unregister_chrdev_region(g_dev, 1);
30223
30224     printk(KERN_INFO "Goodbye...\n");
30225 }
30226
30227 static int generic_open(struct inode *inodep, struct file *filp)
30228 {
30229     printk(KERN_INFO "Generic device opened!..\n");
30230
30231     return 0;
30232 }
30233
30234 static int generic_release(struct inode *inodep, struct file *filp)
30235 {
30236     printk(KERN_INFO "Generic device released!..\n");
30237
30238     return 0;
30239 }
30240
30241 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
30242                             loff_t *off)
30243 {
30244     printk(KERN_INFO "Process sleeping!..\n");
30245
30246     g_cond = 0;
30247     if (wait_event_interruptible_exclusive(g_wq, g_cond != 0))
30248         return -ERESTARTSYS;
30249
30250     printk(KERN_INFO "Process waking up!..\n");
30251
30252     return 0;
30253 }
30254 static ssize_t generic_write(struct file *filp, const char *buf, size_t
```

```
    size, loff_t *off)
30255 {
30256     g_cond = 1;
30257     wake_up(&g_wq);
30258
30259     return 0;
30260 }
30261
30262 module_init(generic_init);
30263 module_exit(generic_exit);
30264
30265 /
*-----*
-----*
30266     Şimdi de boru örneğimizi gerçek bir pipe haline getirelim. Yani      ↵
30267     özellikleri katalım:
30268
30269     1) Okuma yapan prosesler eğer boruda okunacak hiçbir byte yoksa blokede ↵
30270         beklemelidir. (Tabii birden fazla proses bloke olmuşsa boruya
30271         bilgi geldiğinde hangisinin bu bilgiyi alacağıının garantisı yoktur.)
30272
30273     2) Yazma işlemi sırasında eğer boruda yazılanların hepsini alacak kadar ↵
30274         boş yer yoksa yazan proses bloke olmalıdır.
30275
30276     Aşağıdaki örnekte blokenin çözülmesi sırasında while dönüşüne dikkat      ↵
30277     ediniz:
30278
30279     while (g_count == 0) {
30280         up(&g_sem);
30281         if (wait_event_interruptible(g_wqread, g_count > 0))
30282             return -ERESTARTSYS;
30283         if (down_interruptible(&g_sem))
30284             return -ERESTARTSYS;
30285     }
30286
30287     Burada birden fazla proses uykudan uyandırılabilir. Ancak bunlardan biri ↵
30288     semaphore'dan geçecektir. Bu durumda semaphore'dan geçen proses
30289     her şeyi okuyabileceği için uyanmış olup semaphore'da kalan prosesin      ↵
30290     semaphore'dan çıktığında yeniden koşula bakması gereklidir. Bu nedenle      ↵
30291     yukarıdaki kodda
30292     mecburen if deyīmi yerine while deyīmi kullanılmıştır. Aynı durum write ↵
30293     işleminde de söz konusudur. Ayrıca bu örnekte uyandırma işlemi tüm      ↵
30294     kuyruktaki processlerin
30295     wake_up_all ile uyandırılması biçiminde gerçekleştirilmektedir. Çünkü      ↵
30296     birden fazla proses boruda kalanları okuyabilmelidir.
30297
30298 -----*/
```

```
30294 #include <linux/fs.h>
30295 #include <linux/cdev.h>
30296 #include <linux/uaccess.h>
30297 #include <linux/semaphore.h>
30298
30299 MODULE_LICENSE("GPL");
30300 MODULE_DESCRIPTION("General Character Device Driver");
30301 MODULE_AUTHOR("Kaan Aslan");
30302
30303 #define MIN(a, b) ((a) < (b) ? (a) : (b))
30304
30305 #define PIPE_SIZE 10
30306
30307 static int generic_open(struct inode *inodep, struct file *filp);
30308 static int generic_release(struct inode *inodep, struct file *filp);
30309 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
30310                             loff_t *off); ↵
30310 static ssize_t generic_write(struct file *filp, const char *buf, size_t size,
30311                             loff_t *off); ↵
30311
30312 static dev_t g_dev;
30313 static struct cdev *g_cdev;
30314 static struct file_operations g_file_ops = {
30315     .owner = THIS_MODULE,
30316     .open = generic_open,
30317     .release = generic_release,
30318     .read = generic_read,
30319     .write = generic_write,
30320 };
30321
30322 static char g_pipebuf[PIPE_SIZE];
30323 static size_t g_head = 0, g_tail = 0;
30324 static size_t g_count = 0;
30325 static DEFINE_SEMAPHORE(g_sem);
30326 static DECLARE_WAIT_QUEUE_HEAD(g_wqread);
30327 static DECLARE_WAIT_QUEUE_HEAD(g_wqwrite);
30328
30329 static int __init generic_init(void)
30330 {
30331     int result;
30332
30333     if ((result = alloc_chrdev_region(&g_dev, 0, 1, "pipe-driver")) < 0) {
30334         printk(KERN_INFO "Cannot alloc char driver!...\n");
30335         return result;
30336     }
30337
30338     if ((g_cdev = cdev_alloc()) == NULL) {
30339         printk(KERN_INFO "Cannot allocate cdev!..\n");
30340         return -ENOMEM;
30341     }
30342
30343     g_cdev->owner = THIS_MODULE;
30344     g_cdev->ops = &g_file_ops;
```

```
30345
30346     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
30347         unregister_chrdev_region(g_dev, 1);
30348         printk(KERN_INFO "Cannot add character device driver!...\\n");
30349         return result;
30350     }
30351
30352     printk(KERN_INFO "Pipe driver initialized with %d:%d device number... ↵
30353             \\n", MAJOR(g_dev), MINOR(g_dev));
30354
30355     return 0;
30356 }
30357 static void __exit generic_exit(void)
30358 {
30359     cdev_del(g_cdev);
30360     unregister_chrdev_region(g_dev, 1);
30361
30362     printk(KERN_INFO "Goodbye...\\n");
30363 }
30364
30365 static int generic_open(struct inode *inodep, struct file *filp)
30366 {
30367     printk(KERN_INFO "Pipe device opened!..\\n");
30368
30369     return 0;
30370 }
30371
30372 static int generic_release(struct inode *inodep, struct file *filp)
30373 {
30374     printk(KERN_INFO "Pipe device released!..\\n");
30375
30376     return 0;
30377 }
30378
30379 static ssize_t generic_read(struct file *filp, char *buf, size_t size, ↵
30380     loff_t *off)
30381 {
30382     size_t len;
30383     ssize_t result;
30384
30385     if (down_interruptible(&g_sem))
30386         return -ERESTARTSYS;
30387
30388     while (g_count == 0) {
30389         up(&g_sem);
30390         if (wait_event_interruptible(g_wqread, g_count > 0))
30391             return -ERESTARTSYS;
30392         if (down_interruptible(&g_sem))
30393             return -ERESTARTSYS;
30394     }
30395
30396     size = MIN(size, g_count);
```

```
30396
30397     if (g_head >= g_tail)
30398         len = MIN(size, PIPE_SIZE - g_head);
30399     else
30400         len = size;
30401
30402     if (copy_to_user(buf, g_pipebuf + g_head, len) != 0) {
30403         result = -EFAULT;
30404         goto EXIT;
30405     }
30406
30407     if (size > len)
30408         if (copy_to_user(buf + len, g_pipebuf, size - len) != 0) {
30409             result = -EFAULT;
30410             goto EXIT;
30411         }
30412
30413     g_head = (g_head + size) % PIPE_SIZE;
30414     g_count -= size;
30415
30416     result = size;
30417
30418     wake_up_all(&g_wqwrite);
30419 EXIT:
30420     up(&g_sem);
30421
30422     return result;
30423 }
30424
30425 static ssize_t generic_write(struct file *filp, const char *buf, size_t      ↵
30426     size, loff_t *off)
30427 {
30428     size_t len;
30429     ssize_t result;
30430
30431     if (down_interruptible(&g_sem))
30432         return -ERESTARTSYS;
30433
30434     while (PIPE_SIZE - g_count < size) {
30435         up(&g_sem);
30436         if (wait_event_interruptible(g_wqwrite, PIPE_SIZE - g_count >=      ↵
30437             size))
30438             return -ERESTARTSYS;
30439
30440         if (down_interruptible(&g_sem))
30441             return -ERESTARTSYS;
30442     }
30443
30444     if (g_tail >= g_head)
30445         len = MIN(size, PIPE_SIZE - g_tail);
30446     else
30447         len = size;
```

```
30447     if (copy_from_user(g_pipebuf + g_tail, buf, len) != 0) {
30448         result = -EFAULT;
30449         goto EXIT;
30450     }
30451
30452     if (size > len)
30453         if (copy_from_user(g_pipebuf, buf + len, size - len) != 0) {
30454             result = -EFAULT;
30455             goto EXIT;
30456         }
30457
30458     g_tail = (g_tail + size ) % PIPE_SIZE;
30459     g_count += size;
30460
30461     result = size;
30462
30463     wake_up_all(&g_wqread);
30464
30465 EXIT:
30466     up(&g_sem);
30467
30468     return result;
30469 }
30470
30471 module_init(generic_init);
30472 module_exit(generic_exit);
30473
30474 obj-m += $(file).o
30475
30476 all:
30477     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
30478 clean:
30479     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
30480
30481 /* pipeproc1.c */
30482
30483 #include <stdio.h>
30484 #include <stdlib.h>
30485 #include <string.h>
30486 #include <fcntl.h>
30487 #include <unistd.h>
30488
30489 #define PIPE_SIZE      4096
30490
30491 void exit_sys(const char *msg);
30492
30493 int main(void)
30494 {
30495     int fd;
30496     char buf[PIPE_SIZE];
30497     char *str;
30498     size_t len;
30499 }
```

```
30500     if ((fd = open("pipe-driver", O_WRONLY)) == -1)
30501         exit_sys("open");
30502
30503     for (;;) {
30504         printf("Enter text:");
30505         fflush(stdout);
30506         fgets(buf, PIPE_SIZE, stdin);
30507         if ((str = strchr(buf, '\n')) != NULL)
30508             *str = '\0';
30509         if (!strcmp(buf, "quit"))
30510             break;
30511         len = strlen(buf);
30512         if (write(fd, buf, len) == -1)
30513             exit_sys("write");
30514
30515         printf("%lu bytes written...\n", (unsigned long)len);
30516     }
30517
30518     close(fd);
30519
30520     return 0;
30521 }
30522
30523 void exit_sys(const char *msg)
30524 {
30525     perror(msg);
30526
30527     exit(EXIT_FAILURE);
30528 }
30529
30530 /* pipeproc2.c */
30531
30532 #include <stdio.h>
30533 #include <stdlib.h>
30534 #include <string.h>
30535 #include <fcntl.h>
30536 #include <unistd.h>
30537
30538 #define PIPE_SIZE      4096
30539
30540 void exit_sys(const char *msg);
30541
30542 int main(void)
30543 {
30544     int fd;
30545     char buf[PIPE_SIZE + 1];
30546     int len;
30547     ssize_t result;
30548
30549     if ((fd = open("pipe-driver", O_RDONLY)) == -1)
30550         exit_sys("open");
30551
30552     for (;;) {
```

```
30553     printf("How many bytes to read? ");
30554     fflush(stdout);
30555     scanf("%d", &len);
30556     if (!len)
30557         break;
30558     if ((result = read(fd, buf, len)) == -1)
30559         exit_sys("read");
30560     buf[result] = '\0';
30561
30562     printf("%ld bytes read: \"%s\"\n", (long)result, buf);
30563 }
30564
30565     close(fd);
30566
30567     return 0;
30568 }
30569
30570 void exit_sys(const char *msg)
30571 {
30572     perror(msg);
30573
30574     exit(EXIT_FAILURE);
30575 }
30576
30577 #!/bin/bash
30578
30579 module=$1
30580 mode=666
30581
30582 /sbin/insmod ./${module}.ko ${@:2} || exit 1
30583 major=$(awk "\$2 == \"\$module\" {print \$1}" /proc/devices)
30584 rm -f $module
30585 mknod $module c $major 0
30586 chmod $mode $module
30587
30588 /
*-----*-----*-----*
```

30589 Aslında wait_event fonksiyonları export edilmiş birkaç fonksiyon çağrılarak yazılmıştır. Dolayısıyla wait_event fonksiyonlarını çağrırmak

30590 yerine programcı daha aşağı seviyeli (zaten wait_event fonksiyonlarının çağrılmış olduğu) fonksiyonları çağrılabılır. Yani bu işlemi

30591 daha aşağı seviyede manuel yapabilir. MProsesin manuel olarak wait kuyruğuna alınması prepare_to_wait ve prepare_to_wait_exclusive isimli fonksiyonlar

30592 tarafından yapılmaktadır:

30593

30594 void prepare_to_wait(struct wait_queue_head *wq_head, struct

30595 wait_queue_entry *wq_entry, int state);

30595 void prepare_to_wait_exclusive(struct wait_queue_head *wq_head, struct

30596 wait_queue_entry *wq_entry, int state);

30597 Bu fonksiyonlardaki state parametreleri TASK_UNINTERRUPTIBLE ya da →
TASK_INTERRUPTIBLE olabilmektedir. Aslında wait kuyrukları struct →
wait_queue_entry nesnelerinden →

30598 oluşmaktadır. Dolayısıyla bu fonksiyonlar da aslında wait kuruğuna bir →
wait_queue_entry nesnesi eklemektedir. Yani programcının bunun için →
yeni bir struct →

30599 wait_queue_entry nesnesi oluşturulması gerekmektedir. →
prepare_to_wait_exclusive exclusive uyuma için kullanılmaktadır. →

30600

30601 Bir struct wait_queue_entry nesnesi şöyle oluşturulabilir: →

30602

30603 DEFINE_WAIT(entry)

30604

30605 Ya da açıkça tanımlanıp init_wait makrosuyle ilkdeğerlenebilir. Örneğin: →

30606

30607 struct wait_queue_entry entry;

30608 ...

30609 init_wait(&entry);

30610

30611 prepare_to_wait ve prepare_to_wait_exclusive fonksiyonları sunları →
yapmaktadır: →

30612

30613 1) Prosesi run kuyruğundan çıkartıp wait kuyruğuna yerleştirir. (Run →
kuruğun organizasyonu ve bu işlemin gerçek ayrıntıları biraz →
karmaşıktır.) →

30614 2) Prosesin durum bilgisini (task state) state parametresiye belirtilen →
duruma çeker. →

30615 3) prepare_to_wait fonksiyonu kuyruk elemanını exclusive olmaktan →
çıkartırken, prepare_to_wait_exclusive onu exclusive yapar. →

30616

30617 Tabii biz prepare_wait ya da prepare_to_wait_event fonksiyonlarını →
çağırdıktan sonra bir biçimde koşul durumuna bakmalıyız. Eğer koşul →
sağlanmışsa →

30618 hiç prossi uykuya daldırmadan hemen wait kuyruğundan çıkarmalıyız. Eğer →
koşul sağlanmamışsa gerçekten artık schedule fonksiyonuya →
"prosesarası geçiş"

30619 yapmalıyız. Biz schedle fonksiyonunu çağrıdıktan sonra artık →
uyandırılana kadar bir daha çizelgelenmeyez. Başka bir akış bizi →
uyandırınca →

30620 biz schedule fonksiyonun içinden çıkararak yoluma devam ederiz. (Prosesin →
bloke olup dondurulması aslında schedule fonksiyonu içerisinde →
yapılmaktadır.) →

30621 Dolayısıyla proses wake_up fonksiyonlarıyla uyandırıldığından aslında →
schedule içerisindeki akış devam eder ve schedule fonksiyonu geri →
döner.) →

30622

30623 Yukarıda da belirtildiği gibi koşul sağlanmışsa bu uyuma işleminden →
vazgeçmek gereklidir. Vazgeçme işlemi için finish_wait fonksiyonu →
çağrılır: →

30624

30625 void finish_wait(struct wait_queue_head *wq_head, struct →
wait_queue_entry *wq_entry); →

30626

```
30627     Bu fonksiyon zaten proses wake_up fonksiyonları tarafından wait      ↵
            kuyruğundan çıkartılmışsa çıkartma işlemini yapmamaktadır. Bu durumda ↵
            manuel  
30628     uyuma şöyle yapılabilir.  
30629  
30630     DEFINE_WAIT(entry);  
30631  
30632     prepare_to_wait(&g_wq, &entry, TASK_UNINTERRUPTIBLE);  
30633     if (!condition)  
30634         schedule();  
30635     finish_wait(&entry);  
30636  
30637     Tabii eğer INTERRUPTIBLE olarak uygunuyorsa schedule fonksiyonundan      ↵
            çıktılığında sinyal dolayısıyla da çıkışmış olabilir. Bunu anlamak      ↵
            için  
30638     signal_pending isimli fonksiyon çağrılır. Bu fonksiyon sıfır dışı bir      ↵
            değerle geri dönmüşse uyandırma işleminin sinyal yoluyla yapıldığı      ↵
            anlaşılır.  
30639     Bu durumda tabii aygit sürücü fonksiyonu -ERESTARTSYS ile geri      ↵
            döndürülmelidir. Örneğin:  
30640  
30641     DEFINE_WAIT(entry);  
30642  
30643     prepare_to_wait(&g_wq, &entry, TASK_INTERRUPTIBLE);  
30644     if (!condition)  
30645         schedule();  
30646     if (signal_pending(current))  
30647         return -ERESTARTSYS;  
30648     finish_wait(&entry);  
30649  
30650     wake_up makroları da şunları yapmaktadır:  
30651  
30652     1) Wait kuyruğundaki prosesleri çıkartarak run kuyruğuna yerleştirir.  
30653     2) Prosesin durumunu TASK_RUNNING haline getirir.  
30654  
30655     Aşağıda boruörneğinde manuel uykuya dalma işlemi uygulamıştır.  
30656  
30657 -----*/  
30658  
30659 /* pipe-driver.c */  
30660  
30661 #include <linux/module.h>  
30662 #include <linux/kernel.h>  
30663 #include <linux/fs.h>  
30664 #include <linux/cdev.h>  
30665 #include <linux/uaccess.h>  
30666 #include <linux/semaphore.h>  
30667 #include <linux/sched/signal.h>  
30668  
30669 MODULE_LICENSE("GPL");  
30670 MODULE_DESCRIPTION("General Character Device Driver");  
30671 MODULE_AUTHOR("Kaan Aslan");
```

```
30672
30673 #define MIN(a, b)      ((a) < (b) ? (a) : (b))
30674
30675 #define PIPE_SIZE      10
30676
30677 static int generic_open(struct inode *inodep, struct file *filp);
30678 static int generic_release(struct inode *inodep, struct file *filp);
30679 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
30680                           loff_t *off);                                     ↵
30680 static ssize_t generic_write(struct file *filp, const char *buf, size_t    ↵
30681                           size, loff_t *off);
30681
30682 static dev_t g_dev;
30683 static struct cdev *g_cdev;
30684 static struct file_operations g_file_ops = {
30685     .owner = THIS_MODULE,
30686     .open = generic_open,
30687     .release = generic_release,
30688     .read = generic_read,
30689     .write = generic_write,
30690 };
30691
30692 static char g_pipebuf[PIPE_SIZE];
30693 static size_t g_head = 0, g_tail = 0;
30694 static size_t g_count = 0;
30695 static DEFINE_SEMAPHORE(g_sem);
30696 static DECLARE_WAIT_QUEUE_HEAD(g_wqread);
30697 static DECLARE_WAIT_QUEUE_HEAD(g_wqwrite);
30698
30699 static int __init generic_init(void)
30700 {
30701     int result;
30702
30703     if ((result = alloc_chrdev_region(&g_dev, 0, 1, "pipe-driver")) < 0) {
30704         printk(KERN_INFO "Cannot alloc char driver!...\n");
30705         return result;
30706     }
30707
30708     if ((g_cdev = cdev_alloc()) == NULL) {
30709         printk(KERN_INFO "Cannot allocate cdev!..\n");
30710         return -ENOMEM;
30711     }
30712
30713     g_cdev->owner = THIS_MODULE;
30714     g_cdev->ops = &g_file_ops;
30715
30716     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
30717         unregister_chrdev_region(g_dev, 1);
30718         printk(KERN_INFO "Cannot add character device driver!...\n");
30719         return result;
30720     }
30721
30722     printk(KERN_INFO "Pipe driver initialized with %d:%d device number..."    ↵
```

```
30723     \n", MAJOR(g_dev), MINOR(g_dev));
30724     return 0;
30725 }
30726
30727 static void __exit generic_exit(void)
30728 {
30729     cdev_del(g_cdev);
30730     unregister_chrdev_region(g_dev, 1);
30731
30732     printk(KERN_INFO "Goodbye...\n");
30733 }
30734
30735 static int generic_open(struct inode *inodep, struct file *filp)
30736 {
30737     printk(KERN_INFO "Pipe device opened!..\n");
30738
30739     return 0;
30740 }
30741
30742 static int generic_release(struct inode *inodep, struct file *filp)
30743 {
30744     printk(KERN_INFO "Pipe device released!..\n");
30745
30746     return 0;
30747 }
30748
30749 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
30750                             loff_t *off)      ↵
30751 {
30752     size_t len;
30753     ssize_t result;
30754     DEFINE_WAIT(entry);
30755
30756     if (down_interruptible(&g_sem))
30757         return -ERESTARTSYS;
30758
30759     while (g_count == 0) {
30760         up(&g_sem);
30761
30762         prepare_to_wait(&g_wqread, &entry, TASK_INTERRUPTIBLE);
30763         if (g_count == 0)
30764             schedule();
30765         if (signal_pending(current))
30766             return -ERESTARTSYS;
30767         finish_wait(&g_wqread, &entry);
30768
30769         if (down_interruptible(&g_sem))
30770             return -ERESTARTSYS;
30771     }
30772     size = MIN(size, g_count);
30773 }
```

```
30774     if (g_head >= g_tail)
30775         len = MIN(size, PIPE_SIZE - g_head);
30776     else
30777         len = size;
30778
30779     if (copy_to_user(buf, g_pipebuf + g_head, len) != 0) {
30780         result = -EFAULT;
30781         goto EXIT;
30782     }
30783
30784     if (size > len)
30785         if (copy_to_user(buf + len, g_pipebuf, size - len) != 0) {
30786             result = -EFAULT;
30787             goto EXIT;
30788         }
30789
30790     g_head = (g_head + size) % PIPE_SIZE;
30791     g_count -= size;
30792
30793     result = size;
30794
30795     wake_up_all(&g_wqwrite);
30796 EXIT:
30797     up(&g_sem);
30798
30799     return result;
30800 }
30801
30802 static ssize_t generic_write(struct file *filp, const char *buf, size_t      ↴
30803     size, loff_t *off)
30804 {
30805     size_t len;
30806     ssize_t result;
30807     DEFINE_WAIT(entry);
30808
30809     if (down_interruptible(&g_sem))
30810         return -ERESTARTSYS;
30811
30812     while (PIPE_SIZE - g_count < size) {
30813         up(&g_sem);
30814
30815         prepare_to_wait(&g_wqwrite, &entry, TASK_INTERRUPTIBLE);
30816         if (PIPE_SIZE - g_count < size)
30817             schedule();
30818         if (signal_pending(current))
30819             return -ERESTARTSYS;
30820         finish_wait(&g_wqwrite, &entry);
30821
30822         if (down_interruptible(&g_sem))
30823             return -ERESTARTSYS;
30824     }
30825
30826     if (g_tail >= g_head)
```

```
30826         len = MIN(size, PIPE_SIZE - g_tail);
30827     else
30828         len = size;
30829
30830     if (copy_from_user(g_pipebuf + g_tail, buf, len) != 0) {
30831         result = -EFAULT;
30832         goto EXIT;
30833     }
30834
30835     if (size > len)
30836         if (copy_from_user(g_pipebuf, buf + len, size - len) != 0) {
30837             result = -EFAULT;
30838             goto EXIT;
30839         }
30840
30841     g_tail = (g_tail + size) % PIPE_SIZE;
30842     g_count += size;
30843
30844     result = size;
30845
30846     wake_up_all(&g_wqread);
30847
30848 EXIT:
30849     up(&g_sem);
30850
30851     return result;
30852 }
30853
30854 module_init(generic_init);
30855 module_exit(generic_exit);
30856
30857 obj-m += $(file).o
30858
30859 all:
30860     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
30861 clean:
30862     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
30863
30864 /* pipeproc1.c */
30865
30866 #include <stdio.h>
30867 #include <stdlib.h>
30868 #include <string.h>
30869 #include <fcntl.h>
30870 #include <unistd.h>
30871
30872 #define PIPE_SIZE      4096
30873
30874 void exit_sys(const char *msg);
30875
30876 int main(void)
30877 {
30878     int fd;
```

```
30879     char buf[PIPE_SIZE];
30880     char *str;
30881     size_t len;
30882
30883     if ((fd = open("pipe-driver", O_WRONLY)) == -1)
30884         exit_sys("open");
30885
30886     for (;;) {
30887         printf("Enter text:");
30888         fflush(stdout);
30889         fgets(buf, PIPE_SIZE, stdin);
30890         if ((str = strchr(buf, '\n')) != NULL)
30891             *str = '\0';
30892         if (!strcmp(buf, "quit"))
30893             break;
30894         len = strlen(buf);
30895         if (write(fd, buf, len) == -1)
30896             exit_sys("write");
30897
30898         printf("%lu bytes written...\n", (unsigned long)len);
30899     }
30900
30901     close(fd);
30902
30903     return 0;
30904 }
30905
30906 void exit_sys(const char *msg)
30907 {
30908     perror(msg);
30909
30910     exit(EXIT_FAILURE);
30911 }
30912
30913 /* pipeproc2.c */
30914
30915 #include <stdio.h>
30916 #include <stdlib.h>
30917 #include <string.h>
30918 #include <fcntl.h>
30919 #include <unistd.h>
30920
30921 #define PIPE_SIZE      4096
30922
30923 void exit_sys(const char *msg);
30924
30925 int main(void)
30926 {
30927     int fd;
30928     char buf[PIPE_SIZE + 1];
30929     int len;
30930     ssize_t result;
30931 }
```

```
30932     if ((fd = open("pipe-driver", O_RDONLY)) == -1)
30933         exit_sys("open");
30934
30935     for (;;) {
30936         printf("How many bytes to read? ");
30937         fflush(stdout);
30938         scanf("%d", &len);
30939         if (!len)
30940             break;
30941         if ((result = read(fd, buf, len)) == -1)
30942             exit_sys("read");
30943         buf[result] = '\0';
30944
30945         printf("%ld bytes read: \"%s\"\n", (long)result, buf);
30946     }
30947
30948     close(fd);
30949
30950     return 0;
30951 }
30952
30953 void exit_sys(const char *msg)
30954 {
30955     perror(msg);
30956
30957     exit(EXIT_FAILURE);
30958 }
30959
30960 #!/bin/bash
30961
30962 module=$1
30963 mode=666
30964
30965 /sbin/insmod ./${module}.ko ${@:2} || exit 1
30966 major=$(awk "\$2 == \"\$module\" {print \$1}" /proc/devices)
30967 rm -f $module
30968 mknod $module c $major 0
30969 chmod $mode $module
30970
30971 /
*-----*
-----*
-----*
```

30972 Aygit sürücülerdeki okuma ve yazma işlemlerine "nonblocking" desteği
verilebilir. open fonksiyonundaki açış modu dosya nesnesini
belirten struct file yapısının f_flags elemanına kopyalanmaktadır.
Dolayısıyla aygit sürücüyü yazan programcı aygit sürücünün read
ve write fonksiyonlarında aygitin "nonblocking" modda açılıp
açılmadığını şmyle anlayabilir:

30975
30976 if (filp->f_flags & O_NONBLOCK) {
30977 /* open fonksiyonunda aygit O_NONBLOCK bayrağı ile açılmış
30978 }
30979

```
30980     Aşağıdaki örnekte boru işlemine "nonblocking" okuma yazma özelliği de ↵
            eklenmiştir. Anımsanacağı gibi borularla blokesiz okuma işlemi ↵
            sırasında ↵
30981     eğer boruda hiçbir bilgi yoksa read fonksiyonu -1 ile geri döner ve ↵
            errno EAGAIN değeri ile set edilir. Benzer biçimde yazma sırasında da ↵
30982     eğer boruda yazılmak istenenlerin hepsini alacak kadar yer yoksa write ↵
            boruya hiçbir şey yazmaz ve -1 değerine geri döner. errno EAGAIN ↵
30983     değeri ile set edilir. ↵
30984     ↵
30985 -----*/ ↵
30986 ↵
30987 #include <linux/module.h> ↵
30988 #include <linux/kernel.h> ↵
30989 #include <linux/fs.h> ↵
30990 #include <linux/cdev.h> ↵
30991 #include <linux/uaccess.h> ↵
30992 #include <linux/semaphore.h> ↵
30993 ↵
30994 MODULE_LICENSE("GPL"); ↵
30995 MODULE_DESCRIPTION("General Character Device Driver"); ↵
30996 MODULE_AUTHOR("Kaan Aslan"); ↵
30997 ↵
30998 #define MIN(a, b) ((a) < (b) ? (a) : (b)) ↵
30999 ↵
31000 #define PIPE_SIZE 10 ↵
31001 ↵
31002 static int generic_open(struct inode *inodep, struct file *filp); ↵
31003 static int generic_release(struct inode *inodep, struct file *filp); ↵
31004 static ssize_t generic_read(struct file *filp, char *buf, size_t size, ↵
            loff_t *off); ↵
31005 static ssize_t generic_write(struct file *filp, const char *buf, size_t ↵
            size, loff_t *off); ↵
31006 ↵
31007 static dev_t g_dev; ↵
31008 static struct cdev *g_cdev; ↵
31009 static struct file_operations g_file_ops = { ↵
31010     .owner = THIS_MODULE, ↵
31011     .open = generic_open, ↵
31012     .release = generic_release, ↵
31013     .read = generic_read, ↵
31014     .write = generic_write, ↵
31015 }; ↵
31016 ↵
31017 static char g_pipebuf[PIPE_SIZE]; ↵
31018 static size_t g_head = 0, g_tail = 0; ↵
31019 static size_t g_count = 0; ↵
31020 static DEFINE_SEMAPHORE(g_sem); ↵
31021 static DECLARE_WAIT_QUEUE_HEAD(g_wqread); ↵
31022 static DECLARE_WAIT_QUEUE_HEAD(g_wqwrite); ↵
31023 ↵
31024 static int __init generic_init(void) ↵
31025 {
```

```
31026     int result;
31027
31028     if ((result = alloc_chrdev_region(&g_dev, 0, 1, "pipe-driver")) < 0) {
31029         printk(KERN_INFO "Cannot alloc char driver!...\n");
31030         return result;
31031     }
31032
31033     if ((g_cdev = cdev_alloc()) == NULL) {
31034         printk(KERN_INFO "Cannot allocate cdev!..\n");
31035         return -ENOMEM;
31036     }
31037
31038     g_cdev->owner = THIS_MODULE;
31039     g_cdev->ops = &g_file_ops;
31040
31041     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
31042         unregister_chrdev_region(g_dev, 1);
31043         printk(KERN_INFO "Cannot add character device driver!...\n");
31044         return result;
31045     }
31046
31047     printk(KERN_INFO "Pipe driver initialized with %d:%d device number... ↵
31048           \n", MAJOR(g_dev), MINOR(g_dev));
31049     return 0;
31050 }
31051
31052 static void __exit generic_exit(void)
31053 {
31054     cdev_del(g_cdev);
31055     unregister_chrdev_region(g_dev, 1);
31056
31057     printk(KERN_INFO "Goodbye...\n");
31058 }
31059
31060 static int generic_open(struct inode *inodep, struct file *filp)
31061 {
31062     printk(KERN_INFO "Pipe device opened!..\n");
31063
31064     return 0;
31065 }
31066
31067 static int generic_release(struct inode *inodep, struct file *filp)
31068 {
31069     printk(KERN_INFO "Pipe device released!..\n");
31070
31071     return 0;
31072 }
31073
31074 static ssize_t generic_read(struct file *filp, char *buf, size_t size, ↵
31075   loff_t *off)
31076 {
31077     size_t len;
```

```
31077     ssize_t result;
31078
31079     if (down_interruptible(&g_sem))
31080         return -ERESTARTSYS;
31081
31082     while (g_count == 0) {
31083         up(&g_sem);
31084
31085         if (filp->f_flags & O_NONBLOCK)
31086             return -EAGAIN;
31087
31088         if (wait_event_interruptible(g_wqread, g_count > 0))
31089             return -ERESTARTSYS;
31090         if (down_interruptible(&g_sem))
31091             return -ERESTARTSYS;
31092     }
31093
31094     size = MIN(size, g_count);
31095
31096     if (g_head >= g_tail)
31097         len = MIN(size, PIPE_SIZE - g_head);
31098     else
31099         len = size;
31100
31101     if (copy_to_user(buf, g_pipebuf + g_head, len) != 0) {
31102         result = -EFAULT;
31103         goto EXIT;
31104     }
31105
31106     if (size > len)
31107         if (copy_to_user(buf + len, g_pipebuf, size - len) != 0) {
31108             result = -EFAULT;
31109             goto EXIT;
31110         }
31111
31112     g_head = (g_head + size) % PIPE_SIZE;
31113     g_count -= size;
31114
31115     result = size;
31116
31117     wake_up_all(&g_wqwrite);
31118 EXIT:
31119     up(&g_sem);
31120
31121     return result;
31122 }
31123
31124 static ssize_t generic_write(struct file *filp, const char *buf, size_t      ↵
31125     size, loff_t *off)
31126 {
31127     size_t len;
31128     ssize_t result;
```

```
31129     if (down_interruptible(&g_sem))
31130         return -ERESTARTSYS;
31131
31132     while (PIPE_SIZE - g_count < size) {
31133         up(&g_sem);
31134
31135         if (filp->f_flags & O_NONBLOCK)
31136             return -EAGAIN;
31137         if (wait_event_interruptible(g_wqwrite, PIPE_SIZE - g_count >=
31138             size))
31139             return -ERESTARTSYS;
31140
31141         if (down_interruptible(&g_sem))
31142             return -ERESTARTSYS;
31143     }
31144
31145     if (g_tail >= g_head)
31146         len = MIN(size, PIPE_SIZE - g_tail);
31147     else
31148         len = size;
31149
31150     if (copy_from_user(g_pipebuf + g_tail, buf, len) != 0) {
31151         result = -EFAULT;
31152         goto EXIT;
31153     }
31154
31155     if (size > len)
31156         if (copy_from_user(g_pipebuf, buf + len, size - len) != 0) {
31157             result = -EFAULT;
31158             goto EXIT;
31159         }
31160
31161     g_tail = (g_tail + size) % PIPE_SIZE;
31162     g_count += size;
31163
31164     result = size;
31165
31166     wake_up_all(&g_wqread);
31167
31168     EXIT:
31169     up(&g_sem);
31170
31171     return result;
31172 }
31173 module_init(generic_init);
31174 module_exit(generic_exit);
31175
31176 obj-m += $(file).o
31177
31178 all:
31179     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
31180 clean:
```

```
31181     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
31182
31183 /* pipeproc1.c */
31184
31185 #include <stdio.h>
31186 #include <stdlib.h>
31187 #include <string.h>
31188 #include <errno.h>
31189 #include <fcntl.h>
31190 #include <unistd.h>
31191
31192 #define PIPE_SIZE      4096
31193
31194 void exit_sys(const char *msg);
31195
31196 int main(void)
31197 {
31198     int fd;
31199     char buf[PIPE_SIZE];
31200     char *str;
31201     size_t len;
31202
31203     if ((fd = open("pipe-driver", O_WRONLY|O_NONBLOCK)) == -1)
31204         exit_sys("open");
31205
31206     for (;;) {
31207         printf("Enter text:");
31208         fflush(stdout);
31209         fgets(buf, PIPE_SIZE, stdin);
31210         if ((str = strchr(buf, '\n')) != NULL)
31211             *str = '\0';
31212         if (!strcmp(buf, "quit"))
31213             break;
31214         len = strlen(buf);
31215         if (write(fd, buf, len) == -1) {
31216             if (errno == EAGAIN) {
31217                 printf("Nonblocking write returns -1 with errno EAGAIN...\n");
31218                 continue;
31219             }
31220             else
31221                 exit_sys("write");
31222         }
31223
31224         printf("%lu bytes written...\n", (unsigned long)len);
31225     }
31226
31227     close(fd);
31228
31229     return 0;
31230 }
31231
31232 void exit_sys(const char *msg)
```

```
31233 {
31234     perror(msg);
31235
31236     exit(EXIT_FAILURE);
31237 }
31238
31239 /* pipeproc2.c */
31240
31241 #include <stdio.h>
31242 #include <stdlib.h>
31243 #include <string.h>
31244 #include <errno.h>
31245 #include <fcntl.h>
31246 #include <unistd.h>
31247
31248 #define PIPE_SIZE      4096
31249
31250 void exit_sys(const char *msg);
31251
31252 int main(void)
31253 {
31254     int fd;
31255     char buf[PIPE_SIZE + 1];
31256     int len;
31257     ssize_t result;
31258
31259     if ((fd = open("pipe-driver", O_RDONLY|O_NONBLOCK)) == -1)
31260         exit_sys("open");
31261
31262     for (;;) {
31263         printf("How many bytes to read? ");
31264         fflush(stdout);
31265         scanf("%d", &len);
31266         if (!len)
31267             break;
31268         if ((result = read(fd, buf, len)) == -1) {
31269             if (errno == EAGAIN) {
31270                 printf("Nonblocking read return -1 with errno EAGAIN...\n");
31271                 continue;
31272             }
31273             else
31274                 exit_sys("read");
31275         }
31276
31277         buf[result] = '\0';
31278
31279         printf("%ld bytes read: \"%s\"\n", (long)result, buf);
31280     }
31281
31282     close(fd);
31283
31284     return 0;
31285 }
```

```
31286
31287 void exit_sys(const char *msg)
31288 {
31289     perror(msg);
31290
31291     exit(EXIT_FAILURE);
31292 }
31293
31294 #!/bin/bash
31295
31296 module=$1
31297 mode=666
31298
31299 /sbin/insmod ./${module}.ko ${@:2} || exit 1
31300 major=$(awk "\$2 == \"\$module\" {print \$1}" /proc/devices)
31301 rm -f $module
31302 mknod $module c $major 0
31303 chmod $mode $module
31304
31305 /
*-----*
-----*
31306      32 Bit linux sistemlerinde proseslerin sanal bellek alanları 3GB User,    ↵
            1GB Kernel olmak üzere 2 bölüme ayrılmıştır. 64 bir Linux
31307      sistemlerinde ise yalnızca sanal bellek alanının 256 TB'sı           ↵
            kullanılmaktadır. 128 TB User alanı için 128 TB Kernel alanı için    ↵
            kullanılmaktadır.
31308      Çekirdeğin bulunduğu sayfa tabloları prosesler arası geçişte sabit    ↵
            kalır. Dolayısıyla o andaki proses ne olursa olsun her zaman çekirdek    ↵
            aynı
31309      sanal adresteñdir. Tabii çekirdek fiziksel RAM'in de istediği yerine    ↵
            erişebilmelidir. Bunu için Linux'ta çekirdeğin sanal bellekte    ↵
            başlangıç yerinden
31310      itibaren fiziksel RAM sayfa tablosunda haritalandırılmıştır. Yani        ↵
            örneğin 32 bit Linuz sistemlerinde çekirdek alanı 0xC0000000'dan (3GB)    ↵
            başlar.
31311      biz bu sistemlerde kernel moddayken (örneğin aygit sürücü içerisindeyken)    ↵
            0xC0000000 adresine eriştiğimizde aslında fiziksel RAM'in 0'inci    ↵
            adresine erişmiş oluruz.)
31312      Böylece 32 bit Linux sistemlerinde kernel moddaki kodlar fiziksel RAM'de    ↵
            ptr adresine erişmek istediklerinde aslında erişimi 0xC0000000 + ptr    ↵
            biçiminde yaparlar.
31313      Ancak 32 bit Linux sistemleri tüm fiziksel RAM'i bu biçimde    ↵
            haritalandırmamaktadır. (Bunun nedeni çekirdeğin kendisinin de sanal    ↵
            bellekte beli bir yer kaplamasıdır.)
31314      32 Bit Linux sistemleri 0xC0000000'dan itibaren ancak fiziksel RAM'in    ↵
            896 MB'ını bu biçimde haritalandırılabilmektedir. 32 Linux sistemlerinde    ↵
            RAM'in ilk 896 MB'dır
31315      "normal memory zone" denilmektedir. 32 bit Linux sistemleri 896 MB'ın    ↵
            yukarısındaki RAM'e sayfat tablosunu değiştirerek biraz daha fazla    ↵
            çabayla erişmektedir.
31316      32 Bit Linux sistemlerinde 896 MB'nin yukarısındaki fiziksel RAM'e "high    ↵
            memory zone" denilmektedir.
```

31317
31318 64 Bit Linux sistemlerinde çekirdeğin sanal bellek alanı 128 TB olduğu →
 için bu 128' TB'nin başından itibaren tüm fiziksel RAM →
 haritalandırılmıştır. Dolayısıyla →
31319 64 bit Linux sistemlerinde "normal memory zone" tüm fiziksel RAM'i →
 kapsamaktadır. 64 bit Linuz sistemlerinde "high memory zone" yoktur. →
31320
31321
31322 Biz kernel modda kod yazarken belli bir fiziksel adrese erişmek istersek →
 onun sanal adresini bulmamız gereklidir. Bu işin manuel yapılması yerine →
 bunun için →
31323 __va isimli makro kullanılmaktadır. Biz bu makroya bir fiziksel adres →
 veririz o da bize o fiziksel adrese erişmek için gereken sanal adresi →
 verir. Benzer biçimde →
31324 bir sanal adresin fiziksel RAM karşılığını bulmak için de __pa makrosu →
 kullanılmaktadır. Biz bu makroya sanal adresi veririz o da bize o →
 sanal adresin aslında →
31325 RAM'deki hangi fiziksel adres olduğunu verir.
31326
31327 Kernel modda RAM'in her yerine erişebildiğimize ve bu konuda bizi →
 engelleyen hiçbir mekanizmanın olmadığına dikkat ediniz.
31328
31329 Linux çekirdeği için RAM temel olarak 3 bölgeye (zone) ayrılmıştır:
31330
31331 ZONE_DMA
31332 ZONE_NORMAL
31333 ZONE_HIGHMEM
31334
31335 ZONE_DMA ilgili sistemde disk ile RAM arasında transfer yapan DMA'nın →
 erişebildiği RAM alanıdır. Bazı sistemlerde DMA tüm fiziksel RAM'in her →
 yerine transfer yapamamaktadır.
31336 ZONE_NORMAL doğrudan çekirdeğin sayfa tablosu yoluyla haritalandırıldığı →
 fiziksel bellek bölgesidir. Intel 32 bit Linuz sistemlerinde bu bölge ilk →
 896 MB'dır.
31337 ZONE_HIGHMEM ise çekirdeğin doğrudan haritalanmadığı sayfa tablosu →
 değişimiyle erişebildiği fiziksel RAM alanıdır. 32 Bit Intel Linuz →
 sistemlerinde 896 MB'nin
31338 yukarısındaki fiziksel RAM'dir. 64 bit Intel işlemcilerinde ZONE_NORMAL tüm →
 fiziksel RAM'i kapsar. Bu sistemlerde ZONE_HIGHMEM yoktur.
31339 -----*/
31340
31341 /
 *-----

31342 malloc gibi fonksiyonların uyguladığı klasik tahsisat algoritması "boş →
 alan bağlı listesi" denilen yöntemdir. Bu yöntemde yalnızca →
31343 boş alanların kayıtları bir bağlı listede tutulur. Dolayısıyla malloc →
 gibi bir fonksiyon bu bağlı listede uygun bir elemanı bulmaktadır. →
 free fonksiyonu da →
31344 tahsis edilmiş olan alanı bağlı listeye eklemektedir. Ancak bu klasik →
 yöntem çekirdek heap sistemi için çok yavaş kalmaktadır. Bu nedenle →
 çekirdeğin heap sistemi →

31345 için hızlı çalışan tahsisat algoritmaları kullanılmaktadır. BSD ve Linux →
 sistemleri "dilimli tahsis sistemi (slab allocator)" denilen bir →
 algoritmik yöntemi →
31346 kullanmaktadır.
31347
31348 İşte eğer her yapı için ayrı ve eşit uzunlukta boş bloklardan oluşan →
 heap kullanılırsa bağlı listede arama işlemi elimine edilebilmektedir. →
 Dielimli tahsisat
31349 sistemi çekirdek içerisindeki her veri yapısı için (örneğin task_struct, →
 struct file, struct inode gibi) eşit uzunluklu bloklardan oluşan ayrı →
 bir heap oluşturmaktadır.
31350 Çekirdek genel amaçlı (yani kernel tarafından uzunluğunu bilinmeyen) →
 alanlar için dilimli tahsisat heap'leri de oluşturmuştur. Bu genel →
 dilim sistemlerinin blok uzunlukları
31351 şöyledir: 32, 64, 96, 128, 192, 256 512, 1024, 2048, 4096, 8192, 16384, →
 32768, 65536,
31352
31353 İşte kmalloc isimli genel amaçlı çekirdektahsisat fonksiyonu aslında bu →
 dilim sistemlerinin bir tanesinden eşit uzunluklu dilimlerden bize →
 tahsisat yapmaktadır.
31354 Örneğin biz kmalloc fonksiyonu ile 100 byte tahsis etmek isteyelim. →
 Aslında kmalloc 128'lik dilimlerin bulunduğu heap'ten bize blok verir. →
 Yani aslında bu durumda
31355 kmalloc bizim talep ettiğimizden daha büyük bir alanı bize vermektedir. →
 Benzer biçimde kfree fonksiyonu da hangi dilim heap'inden tahsisat →
 yapılmışsa o alanı
31356 o dilim sistemine iade etmektedir. Dilimli tahsisat sisteminden amaç →
 sabit zamanlı dinamik tahsisat işlemini yapılmasıdır. Yani bu →
 tahsisat sistemi ile bellek
31357 tahsis edilirken bir döngü kullanılmamaktadır.
31358
31359 void *kmalloc (size_t size, int flags);
31360 void kfree (const void *objp);
31361
31362 kmalloc fonksiyonunun birinci parametresi tahsis edilecek byte sayısını →
 belirtir. İkincisi parametresi tahsis edilecek alan ve biçim hakkında →
31363 bayrakları içermektedir. Bu ikinci parametre çeşitli sembolik →
 sabitlerden oluşturulmaktadır. Burada önemli birkaç byrak şunlardır:
31364
31365 GFP_KERNEL: Kernel alanı içerisinde normal tahsisat. En sık bu bayrak →
 kullanılmaktadır. Burada RAM doluya swap işlemi yapılabilir. Bu işlem →
 sırásında
31366 akış kernel modda wait kuruklarında bekletilebilir. Tahsisat ZONE_NORMAL →
 alanından yapılmaktadır.
31367
31368 GFP_NOWAIT: GFP_KERNEL gibidir. Ancak hazırda bellek yoksa proses uykuya →
 yatılmaz. Fonksiyon başarısız olur.
31369
31370 GFP_HIGHUSER: 32 bit sistemlerde ZONE_HIHMEM alanından tahssat yapar.
31371
31372 GFP_DMA: İlgili sistemde DMA'nın erişebildiği RAM alanından tahsisat →
 yapar.
31373

```
31374     kfree fonksiyonun parametresi daha önce tahsis edilmiş olan bloğun →  
     adresidir.  
31375  
31376     Aşağıdaki örnekte borunun kullandığı değişkenler bir yapı içerisinde →  
     toplanmış ve o yapı türünden kmalloc ile dinamik tahsisat yapılmıştır. →  
31377     Aygit sürücü yok edilirken alan kfree fonksiyonuyla boşaltılmıştır.  
31378  
31379 -----*/  
31380  
31381 #include <linux/module.h>  
31382 #include <linux/kernel.h>  
31383 #include <linux/fs.h>  
31384 #include <linux/cdev.h>  
31385 #include <linux/uaccess.h>  
31386 #include <linux/semaphore.h>  
31387 #include <linux/slab.h>  
31388  
31389 MODULE_LICENSE("GPL");  
31390 MODULE_DESCRIPTION("General Character Device Driver");  
31391 MODULE_AUTHOR("Kaan Aslan");  
31392  
31393 #define MIN(a, b) ((a) < (b) ? (a) : (b))  
31394  
31395 #define PIPE_SIZE 10  
31396  
31397 static int generic_open(struct inode *inodep, struct file *filp);  
31398 static int generic_release(struct inode *inodep, struct file *filp);  
31399 static ssize_t generic_read(struct file *filp, char *buf, size_t size,  
     loff_t *off);  
31400 static ssize_t generic_write(struct file *filp, const char *buf, size_t →  
     size, loff_t *off);  
31401  
31402 static dev_t g_dev;  
31403 static struct file_operations g_file_ops = {  
31404     .owner = THIS_MODULE,  
31405     .open = generic_open,  
31406     .release = generic_release,  
31407     .read = generic_read,  
31408     .write = generic_write,  
31409 };  
31410  
31411 struct pipe_dev {  
31412     char pipebuf[PIPE_SIZE];  
31413     size_t head;  
31414     size_t tail;  
31415     size_t count;  
31416     struct semaphore sem;  
31417     wait_queue_head_t wqread;  
31418     wait_queue_head_t wqwrite;  
31419     struct cdev cdev;  
31420 };
```

```
31421
31422 static struct pipe_dev *g_pdev;
31423
31424 static int __init generic_init(void)
31425 {
31426     int result;
31427
31428     if ((result = alloc_chrdev_region(&g_dev, 0, 1, "pipe-driver")) < 0) {
31429         printk(KERN_INFO "Cannot alloc char driver!...\n");
31430         return result;
31431     }
31432
31433     if ((g_pdev = (struct pipe_dev *)kmalloc(sizeof(struct pipe_dev), GFP_KERNEL)) == NULL) {
31434         printk(KERN_INFO "Cannot allocate memory!...\n");
31435         return -ENOMEM;
31436     }
31437
31438     g_pdev->head = g_pdev->tail = g_pdev->count = 0;
31439     init_waitqueue_head(&g_pdev->wqread);
31440     init_waitqueue_head(&g_pdev->wqwrite);
31441     sema_init(&g_pdev->sem, 1);
31442
31443     cdev_init(&g_pdev->cdev, &g_file_ops);
31444
31445     if ((result = cdev_add(&g_pdev->cdev, g_dev, 1)) != 0) {
31446         kfree(g_pdev);
31447         unregister_chrdev_region(g_dev, 1);
31448         printk(KERN_INFO "Cannot add character device driver!...\n");
31449         return result;
31450     }
31451
31452     printk(KERN_INFO "Pipe driver initialized with %d:%d device number...
31453             \n", MAJOR(g_dev), MINOR(g_dev));
31454
31455     return 0;
31456 }
31457
31458 static void __exit generic_exit(void)
31459 {
31460     cdev_del(&g_pdev->cdev);
31461     unregister_chrdev_region(g_dev, 1);
31462     kfree(g_pdev);
31463
31464     printk(KERN_INFO "Goodbye...\n");
31465 }
31466
31467 static int generic_open(struct inode *inodep, struct file *filp)
31468 {
31469     printk(KERN_INFO "Pipe device opened!..\n");
31470
31471     return 0;
31472 }
```

```
31472
31473 static int generic_release(struct inode *inodep, struct file *filp)
31474 {
31475     printk(KERN_INFO "Pipe device released!..\n");
31476
31477     return 0;
31478 }
31479
31480 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
31481                             loff_t *off)      ↵
31481 {
31482     size_t len;
31483     ssize_t result;
31484
31485     if (down_interruptible(&g_pdev->sem))
31486         return -ERESTARTSYS;
31487
31488     while (g_pdev->count == 0) {
31489         up(&g_pdev->sem);
31490
31491         if (filp->f_flags & O_NONBLOCK)
31492             return -EAGAIN;
31493
31494         if (wait_event_interruptible(g_pdev->wqread, g_pdev->count > 0))
31495             return -ERESTARTSYS;
31496         if (down_interruptible(&g_pdev->sem))
31497             return -ERESTARTSYS;
31498     }
31499
31500     size = MIN(size, g_pdev->count);
31501
31502     if (g_pdev->head >= g_pdev->tail)
31503         len = MIN(size, PIPE_SIZE - g_pdev->head);
31504     else
31505         len = size;
31506
31507     if (copy_to_user(buf, g_pdev->pipebuf + g_pdev->head, len) != 0) {
31508         result = -EFAULT;
31509         goto EXIT;
31510     }
31511
31512     if (size > len)
31513         if (copy_to_user(buf + len, g_pdev->pipebuf, size - len) != 0) {
31514             result = -EFAULT;
31515             goto EXIT;
31516         }
31517
31518     g_pdev->head = (g_pdev->head + size) % PIPE_SIZE;
31519     g_pdev->count -= size;
31520
31521     result = size;
31522
31523     wake_up_all(&g_pdev->wqwrite);
```

```
31524 EXIT:  
31525     up(&g_pdev->sem);  
31526  
31527     return result;  
31528 }  
31529  
31530 static ssize_t generic_write(struct file *filp, const char *buf, size_t      ↵  
      size, loff_t *off)  
31531 {  
31532     size_t len;  
31533     ssize_t result;  
31534  
31535     if (down_interruptible(&g_pdev->sem))  
31536         return -ERESTARTSYS;  
31537  
31538     while (PIPE_SIZE - g_pdev->count < size) {  
31539         up(&g_pdev->sem);  
31540  
31541         if (filp->f_flags & O_NONBLOCK)  
31542             return -EAGAIN;  
31543         if (wait_event_interruptible(g_pdev->wqwrite, PIPE_SIZE - g_pdev-      ↵  
            >count >= size))  
31544             return -ERESTARTSYS;  
31545  
31546         if (down_interruptible(&g_pdev->sem))  
31547             return -ERESTARTSYS;  
31548     }  
31549  
31550     if (g_pdev->tail >= g_pdev->head)  
31551         len = MIN(size, PIPE_SIZE - g_pdev->tail);  
31552     else  
31553         len = size;  
31554  
31555     if (copy_from_user(g_pdev->pipebuf + g_pdev->tail, buf, len) != 0) {  
31556         result = -EFAULT;  
31557         goto EXIT;  
31558     }  
31559  
31560     if (size > len)  
31561         if (copy_from_user(g_pdev->pipebuf, buf + len, size - len) != 0) {  
31562             result = -EFAULT;  
31563             goto EXIT;  
31564         }  
31565  
31566     g_pdev->tail = (g_pdev->tail + size) % PIPE_SIZE;  
31567     g_pdev->count += size;  
31568  
31569     result = size;  
31570  
31571     wake_up_all(&g_pdev->wqread);  
31572  
31573 EXIT:  
31574     up(&g_pdev->sem);
```

```
31575     return result;
31576 }
31578 module_init(generic_init);
31580 module_exit(generic_exit);
31581
31582 obj-m += $(file).o
31583
31584 all:
31585     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
31586 clean:
31587     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
31588
31589 /* pipeproc1.c */
31590
31591 #include <stdio.h>
31592 #include <stdlib.h>
31593 #include <string.h>
31594 #include <errno.h>
31595 #include <fcntl.h>
31596 #include <unistd.h>
31597
31598 #define PIPE_SIZE      4096
31599
31600 void exit_sys(const char *msg);
31601
31602 int main(void)
31603 {
31604     int fd;
31605     char buf[PIPE_SIZE];
31606     char *str;
31607     size_t len;
31608
31609     if ((fd = open("pipe-driver", O_WRONLY)) == -1)
31610         exit_sys("open");
31611
31612     for (;;) {
31613         printf("Enter text:");
31614         fflush(stdout);
31615         fgets(buf, PIPE_SIZE, stdin);
31616         if ((str = strchr(buf, '\n')) != NULL)
31617             *str = '\0';
31618         if (!strcmp(buf, "quit"))
31619             break;
31620         len = strlen(buf);
31621         if (write(fd, buf, len) == -1) {
31622             if (errno == EAGAIN) {
31623                 printf("Nonblocking write returns -1 with errno EAGAIN... ↵
31624                         \n");
31625                 continue;
31626             }
31627             else
```

```
31627         exit_sys("write") ;
31628     }
31629
31630     printf("%lu bytes written...\n", (unsigned long)len);
31631 }
31632
31633     close(fd);
31634
31635     return 0;
31636 }
31637
31638 void exit_sys(const char *msg)
31639 {
31640     perror(msg);
31641
31642     exit(EXIT_FAILURE);
31643 }
31644
31645 /* pipeproc1.c */
31646
31647 #include <stdio.h>
31648 #include <stdlib.h>
31649 #include <string.h>
31650 #include <errno.h>
31651 #include <fcntl.h>
31652 #include <unistd.h>
31653
31654 #define PIPE_SIZE      4096
31655
31656 void exit_sys(const char *msg);
31657
31658 int main(void)
31659 {
31660     int fd;
31661     char buf[PIPE_SIZE];
31662     char *str;
31663     size_t len;
31664
31665     if ((fd = open("pipe-driver", O_WRONLY)) == -1)
31666         exit_sys("open");
31667
31668     for (;;) {
31669         printf("Enter text:");
31670         fflush(stdout);
31671         fgets(buf, PIPE_SIZE, stdin);
31672         if ((str = strchr(buf, '\n')) != NULL)
31673             *str = '\0';
31674         if (!strcmp(buf, "quit"))
31675             break;
31676         len = strlen(buf);
31677         if (write(fd, buf, len) == -1) {
31678             if (errno == EAGAIN) {
31679                 printf("Nonblocking write returns -1 with errno EAGAIN... ↵
```

```
                                \n");
31680                         continue;
31681                     }
31682                     else
31683                         exit_sys("write") ;
31684                     }
31685
31686                     printf("%lu bytes written...\n", (unsigned long)len);
31687                 }
31688
31689                 close(fd);
31690
31691             return 0;
31692         }
31693
31694     void exit_sys(const char *msg)
31695     {
31696         perror(msg);
31697
31698         exit(EXIT_FAILURE);
31699     }
31700
31701 /* pipeproc1.c */
31702
31703 #include <stdio.h>
31704 #include <stdlib.h>
31705 #include <string.h>
31706 #include <errno.h>
31707 #include <fcntl.h>
31708 #include <unistd.h>
31709
31710 #define PIPE_SIZE      4096
31711
31712 void exit_sys(const char *msg);
31713
31714 int main(void)
31715 {
31716     int fd;
31717     char buf[PIPE_SIZE];
31718     char *str;
31719     size_t len;
31720
31721     if ((fd = open("pipe-driver", O_WRONLY)) == -1)
31722         exit_sys("open");
31723
31724     for (;;) {
31725         printf("Enter text:");
31726         fflush(stdout);
31727         fgets(buf, PIPE_SIZE, stdin);
31728         if ((str = strchr(buf, '\n')) != NULL)
31729             *str = '\0';
31730         if (!strcmp(buf, "quit"))
31731             break;
```

```
31732     len = strlen(buf);
31733     if (write(fd, buf, len) == -1) {
31734         if (errno == EAGAIN) {
31735             printf("Nonblocking write returns -1 with errno EAGAIN...\n");
31736             continue;
31737         }
31738         else
31739             exit_sys("write");
31740     }
31741
31742     printf("%lu bytes written...\n", (unsigned long)len);
31743 }
31744
31745     close(fd);
31746
31747     return 0;
31748 }
31749
31750 void exit_sys(const char *msg)
31751 {
31752     perror(msg);
31753
31754     exit(EXIT_FAILURE);
31755 }
31756
31757 #!/bin/bash
31758
31759 module=$1
31760 mode=666
31761
31762 /sbin/insmod ./${module}.ko ${@:2} || exit 1
31763 major=$(awk "\$2 == \"\$module\" {print \$1}" /proc/devices)
31764 rm -f $module
31765 mknod $module c $major 0
31766 chmod $mode $module
31767
31768 /
*-----*
-----
```

31769 Aygit sürücünün majör ve minör numaraları ne anlam ifade etmektedir?
Majör numara aygit sürücünün türünü belirtir.

31770 Minör numara ise aynı türden aygit sürücülerin farklı örneklerini
(instance'larını) belirtmektedir. Örneğin biz yukarıdaki pipe-driver

31771 aygit sürücümüzün tek bir boruyu değil on farklı boruyu idare etmesini
isteyebiliriz. Bu durumda aygit sürücümüzün bir tane majör numarası,
10 tane minör numarası

31772 olacaktır. Aygit sürücülerin majör numaraları aynı ise bunların kodları
aynidakır. O aynı kod birden fazla aygit sürücü için işlev görmektedir.

31773
31774 Birden fazla minör numara kullanan aygit sürücülerini yazarken dikkatli
olmak gereklidir. Çünkü tek bir kod birden fazla aynı türden bağımsız
aygıtları

31775 idare edecektir.

31776

31777 Birden fazla minör numara üzerinde çalışacak aygit sürücülerini tipik olarak şöyle yazılmalıdır: ↗

31778

31779 1) Programcının majör ve minör numaraları tahsis etmesi gereklidir. Majör numara alloc_chrdev_region fonksiyonuyla dinamik olarak belirlenebilir. ↗ ↗

31780 Bu fonksiyon aynı zamanda belli bir minör numaradan başlayarak n tane minör numarayı da tahsis edebilmektedir. Örneğin: ↗

31781

```
31782 if ((result = alloc_chrdev_region(&g_dev, 0, 10, "pipe-driver")) < 0) {  
31783     printk(KERN_INFO "Cannot alloc char driver!...\n");  
31784     return result;  
31785 }
```

31786

31787 Burada 0'inci minör numaradan 10 tane minör numara için aygit tahsisatı yapılmıştır. ↗

31788

31789 2) Her aygit bir yapıyla temsil edilmeli ve N tane aygit için N elemanlı bir yapı dizisi oluşturulmalıdır. Yukarıdaki örnekte olduğu gibi struct cdev nesnesi bu yapının içinde bulundurulmalıdır. ↗

31790

31791

31792 3) N tane minör numaralı aygit için cdev_add fonksiyonuyla aygitlar çekirdeğe eklenmelidir. ↗

31793

31794 4) Aygit sürücünün open fonksiyonunda programcının inode yapısının i_cdev elemanından hareketle cdev nesnesinin içinde bulunduğu yapı nesnesinin başlangıç adresini container_of makrosuyla bulmalı ve yapı adresini struct file yapısının private_data elemanına yerleştirmelidir. Böylece aygit sürücünün read ve write fonksiyonları ilgili minör numaraya ilişkin aygitın yapı nesnesine erişebilecektir. ↗

31795

31796

31797

31798 5) Aygit sürücünün read ve write fonksiyonları yazılırç

31799

31800 6) release (close) işleminde yapılacak birtakım son işlemler varsa yapılır. ↗

31801

31802 7) Aygit sürücünün exit fonksiyonunda yine tüm minör numaralar için cdev_del ve unregister_chrdev_region işlemi yapılır. ↗

31803

31804 Tabii birden fazla minör numara için çalışacak aygit sürücülerin birden fazla aygit dosyası yaratması gereklidir. Bu da onları yüklemek için kullandığımız load scriptinde değişiklik yapmayı gereklidir. ↗

31805 O halde biz N tane minör numaraya ilişkin aygit dosyası yaratacak biçimde yeni bir "loadmulti" isimli script yazalım: ↗

31806

```
31807 #!/bin/bash  
31808  
31809 module=$2
```

```
31811     mode=666
31812
31813     /sbin/insmod ./${module}.ko ${@:3} || exit 1
31814     major=$(awk "\$2 == \"\$module\" {print \$1}" /proc/devices)
31815
31816     for ((i = 0; i < $1; ++i))
31817     do
31818         rm -f ${module}${i}
31819         mknod ${module}${i} c $major $i
31820         chmod $mode ${module}${i}
31821     done
31822
31823     Aşağıdaki örnekte boru aygıt sürücüsü MAX_DEVICE kadar minör numarayı ↵
31824             (yani farklı boruları) destekleyecek duruma getirilmiştir.
31825 -----
31826 -----*/
31827 #include <linux/module.h>
31828 #include <linux/kernel.h>
31829 #include <linux/fs.h>
31830 #include <linux/cdev.h>
31831 #include <linux/uaccess.h>
31832 #include <linux/semaphore.h>
31833
31834 MODULE_LICENSE("GPL");
31835 MODULE_DESCRIPTION("General Character Device Driver");
31836 MODULE_AUTHOR("Kaan Aslan");
31837
31838 #define MIN(a, b)      ((a) < (b) ? (a) : (b))
31839
31840 #define MAX_DEVICE      10
31841 #define PIPE_SIZE        4096
31842
31843 static int generic_open(struct inode *inodep, struct file *filp);
31844 static int generic_release(struct inode *inodep, struct file *filp);
31845 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
31846                             loff_t *off); ↵
31846 static ssize_t generic_write(struct file *filp, const char *buf, size_t
31847                             size, loff_t *off); ↵
31847
31848 static dev_t g_dev;
31849 static struct file_operations g_file_ops = {
31850     .owner = THIS_MODULE,
31851     .open = generic_open,
31852     .release = generic_release,
31853     .read = generic_read,
31854     .write = generic_write,
31855 };
31856
31857 struct pipe_dev {
31858     char pipebuf[PIPE_SIZE];
31859     size_t head;
```

```
31860     size_t tail;
31861     size_t count;
31862     struct semaphore sem;
31863     wait_queue_head_t wqread;
31864     wait_queue_head_t wqwrite;
31865     struct cdev cdev;
31866 };
31867
31868 static struct pipe_dev g_devs[MAX_DEVICE];
31869
31870 static int __init generic_init(void)
31871 {
31872     int result;
31873     int i, k;
31874     dev_t dev;
31875
31876     if ((result = alloc_chrdev_region(&g_dev, 0, MAX_DEVICE, "pipe-driver")) < 0) {
31877         printk(KERN_INFO "Cannot alloc char driver!...\n");
31878         return result;
31879     }
31880
31881     for (i = 0; i < MAX_DEVICE; ++i) {
31882         g_devs[i].head = g_devs[i].tail = g_devs[i].count = 0;
31883         init_waitqueue_head(&g_devs[i].wqread);
31884         init_waitqueue_head(&g_devs[i].wqwrite);
31885         sema_init(&g_devs[i].sem, 1);
31886
31887         cdev_init(&g_devs[i].cdev, &g_file_ops);
31888         dev = MKDEV(MAJOR(g_dev), i);
31889         if ((result = cdev_add(&g_devs[i].cdev, dev, 1)) != 0) {
31890             for (k = 0; k < i; ++k)
31891                 cdev_del(&g_devs[i].cdev);
31892             unregister_chrdev_region(g_dev, MAX_DEVICE);
31893             return result;
31894         }
31895     }
31896
31897     printk(KERN_INFO "Pipe driver initialized with %d:%d + %d device
31898           numbers...\n", MAJOR(g_dev), MINOR(g_dev), MAX_DEVICE);    ↵
31899
31900     return 0;
31901 }
31902
31903 static void __exit generic_exit(void)
31904 {
31905     int i;
31906
31907     for (i = 0; i < MAX_DEVICE; ++i)
31908         cdev_del(&g_devs[i].cdev);
31909
31910     unregister_chrdev_region(g_dev, MAX_DEVICE);
```

```
31911     printk(KERN_INFO "Goodbye...\n");
31912 }
31913 }
31914
31915 static int generic_open(struct inode *inodep, struct file *filp)
31916 {
31917     struct pipe_dev *pdev = container_of(inodep->i_cdev, struct pipe_dev, cdev);
31918
31919     filp->private_data = pdev;
31920
31921     printk(KERN_INFO "Pipe device opened!..\n");
31922
31923     return 0;
31924 }
31925
31926 static int generic_release(struct inode *inodep, struct file *filp)
31927 {
31928     printk(KERN_INFO "Pipe device released!..\n");
31929
31930     return 0;
31931 }
31932
31933 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
31934                             loff_t *off)
31935 {
31936     size_t len;
31937     ssize_t result;
31938     struct pipe_dev *pdev = (struct pipe_dev *)filp->private_data;
31939
31940     if (down_interruptible(&pdev->sem))
31941         return -ERESTARTSYS;
31942
31943     while (pdev->count == 0) {
31944         up(&pdev->sem);
31945
31946         if (filp->f_flags & O_NONBLOCK)
31947             return -EAGAIN;
31948
31949         if (wait_event_interruptible(pdev->wqread, pdev->count > 0))
31950             return -ERESTARTSYS;
31951         if (down_interruptible(&pdev->sem))
31952             return -ERESTARTSYS;
31953
31954     size = MIN(size, pdev->count);
31955
31956     if (pdev->head >= pdev->tail)
31957         len = MIN(size, PIPE_SIZE - pdev->head);
31958     else
31959         len = size;
31960
31961     if (copy_to_user(buf, pdev->pipebuf + pdev->head, len) != 0) {
```

```
31962         result = -EFAULT;
31963         goto EXIT;
31964     }
31965
31966     if (size > len)
31967         if (copy_to_user(buf + len, pdev->pipebuf, size - len) != 0) {
31968             result = -EFAULT;
31969             goto EXIT;
31970         }
31971
31972     pdev->head = (pdev->head + size) % PIPE_SIZE;
31973     pdev->count -= size;
31974
31975     result = size;
31976
31977     wake_up_all(&pdev->wqwrite);
31978 EXIT:
31979     up(&pdev->sem);
31980
31981     return result;
31982 }
31983
31984 static ssize_t generic_write(struct file *filp, const char *buf, size_t      ↴
31985     size, loff_t *off)
31986 {
31987     size_t len;
31988     ssize_t result;
31989     struct pipe_dev *pdev = (struct pipe_dev *)filp->private_data;
31990
31991     if (down_interruptible(&pdev->sem))
31992         return -ERESTARTSYS;
31993
31994     while (PIPE_SIZE - pdev->count < size) {
31995         up(&pdev->sem);
31996
31997         if (filp->f_flags & O_NONBLOCK)
31998             return -EAGAIN;
31999
32000         if (wait_event_interruptible(pdev->wqwrite, PIPE_SIZE - pdev->count      ↴
32001             >= size))
32002             return -ERESTARTSYS;
32003
32004     }
32005
32006     if (pdev->tail >= pdev->head)
32007         len = MIN(size, PIPE_SIZE - pdev->tail);
32008     else
32009         len = size;
32010
32011     if (copy_from_user(pdev->pipebuf + pdev->tail, buf, len) != 0) {
32012         result = -EFAULT;
```

```
32013     goto EXIT;
32014 }
32015
32016     if (size > len)
32017         if (copy_from_user(pdev->pipebuf, buf + len, size - len) != 0) {
32018             result = -EFAULT;
32019             goto EXIT;
32020         }
32021
32022     pdev->tail = (pdev->tail + size) % PIPE_SIZE;
32023     pdev->count += size;
32024
32025     result = size;
32026
32027     wake_up_all(&pdev->wqread);
32028
32029 EXIT:
32030     up(&pdev->sem);
32031
32032     return result;
32033 }
32034
32035 module_init(generic_init);
32036 module_exit(generic_exit);
32037
32038 obj-m += $(file).o
32039
32040 all:
32041     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
32042 clean:
32043     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
32044
32045 /* pipeproc1.c */
32046
32047 #include <stdio.h>
32048 #include <stdlib.h>
32049 #include <string.h>
32050 #include <errno.h>
32051 #include <fcntl.h>
32052 #include <unistd.h>
32053
32054 #define PIPE_SIZE      4096
32055
32056 void exit_sys(const char *msg);
32057
32058 int main(int argc, char *argv[])
32059 {
32060     int fd;
32061     char buf[PIPE_SIZE];
32062     char *str;
32063     size_t len;
32064
32065     if (argc != 2) {
```

```
32066         fprintf(stderr, "wrong number of arguments!..\n");
32067         exit(EXIT_FAILURE);
32068     }
32069
32070     if ((fd = open(argv[1], O_WRONLY)) == -1)
32071         exit_sys("open");
32072
32073     for (;;) {
32074         printf("Enter text:");
32075         fflush(stdout);
32076         fgets(buf, PIPE_SIZE, stdin);
32077         if ((str = strchr(buf, '\n')) != NULL)
32078             *str = '\0';
32079         if (!strcmp(buf, "quit"))
32080             break;
32081         len = strlen(buf);
32082         if (write(fd, buf, len) == -1) {
32083             if (errno == EAGAIN) {
32084                 printf("Nonblocking write returns -1 with errno EAGAIN... \n");
32085                 continue;
32086             }
32087             else
32088                 exit_sys("write");
32089         }
32090
32091         printf("%lu bytes written...\n", (unsigned long)len);
32092     }
32093
32094     close(fd);
32095
32096     return 0;
32097 }
32098
32099 void exit_sys(const char *msg)
32100 {
32101     perror(msg);
32102
32103     exit(EXIT_FAILURE);
32104 }
32105
32106 /* pipeproc2.c */
32107
32108 #include <stdio.h>
32109 #include <stdlib.h>
32110 #include <string.h>
32111 #include <errno.h>
32112 #include <fcntl.h>
32113 #include <unistd.h>
32114
32115 #define PIPE_SIZE      4096
32116
32117 void exit_sys(const char *msg);
```

```
32118
32119 int main(int argc, char *argv[])
32120 {
32121     int fd;
32122     char buf[PIPE_SIZE + 1];
32123     int len;
32124     ssize_t result;
32125
32126     if (argc != 2) {
32127         fprintf(stderr, "wrong number of arguments!..\n");
32128         exit(EXIT_FAILURE);
32129     }
32130
32131     if ((fd = open(argv[1], O_RDONLY)) == -1)
32132         exit_sys("open");
32133
32134     for (;;) {
32135         printf("How many bytes to read? ");
32136         fflush(stdout);
32137         scanf("%d", &len);
32138         if (!len)
32139             break;
32140         if ((result = read(fd, buf, len)) == -1) {
32141             if (errno == EAGAIN) {
32142                 printf("Nonblocking read return -1 with errno EAGAIN...\n");
32143                 continue;
32144             }
32145             else
32146                 exit_sys("read");
32147         }
32148
32149         buf[result] = '\0';
32150
32151         printf("%ld bytes read: \"%s\"\n", (long)result, buf);
32152     }
32153
32154     close(fd);
32155
32156     return 0;
32157 }
32158
32159 void exit_sys(const char *msg)
32160 {
32161     perror(msg);
32162
32163     exit(EXIT_FAILURE);
32164 }
32165
32166 #!/bin/bash
32167
32168 module=$2
32169 mode=666
32170
```

```
32171 /sbin/insmod ./${module}.ko ${@:3} || exit 1
32172 major=$(awk "\$2 == \"\$module\" {print \$1}" /proc/devices)
32173
32174 for ((i = 0; i < $1; ++i))
32175 do
32176     rm -f ${module}${i}
32177     mknod ${module}${i} c $major $i
32178     chmod $mode ${module}${i}
32179 done
32180
32181 /
*-----*
-----*
32182 Birden fazla minör numara kullanılırken bu minör numaralar için yapı →
nesneleri kmalloc ile de dinamik bir biçimde tahsis edilebilir. →
32183 Bu durum tipik olarak modül parametresi ile aygit sayısının dışarıdan →
verildiği uygulamalarda gerekebilmektedir. →
32184
32185 Aşağıda örnekte aygit sürücünün kaç minör numarayı destekleyeceği aygit →
sürücü yüklenirken ndevice modül parametresiyle belirlenmektedir. →
32186 Eğer bu parametre girilmezse default değer 10 olarak alınmaktadır. Bu →
aygit sürücüyü söyle yükleyebilirsiniz:
32187
32188 ./loadmulti 2 pipe-driver ndevice=2
32189 -----*/
32190
32191 #include <linux/module.h>
32192 #include <linux/kernel.h>
32193 #include <linux/fs.h>
32194 #include <linux/cdev.h>
32195 #include <linux/uaccess.h>
32196 #include <linux/semaphore.h>
32197 #include <linux/slab.h>
32198
32199 MODULE_LICENSE("GPL");
32200 MODULE_DESCRIPTION("General Character Device Driver");
32201 MODULE_AUTHOR("Kaan Aslan");
32202
32203 #define MIN(a, b) ((a) < (b) ? (a) : (b))
32204
32205 #define DEF_NDEVICE 10
32206 #define PIPE_SIZE 4096
32207
32208 static int generic_open(struct inode *inodep, struct file *filp);
32209 static int generic_release(struct inode *inodep, struct file *filp);
32210 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
loff_t *off);
32211 static ssize_t generic_write(struct file *filp, const char *buf, size_t
size, loff_t *off);
32212
32213 static dev_t g_dev;
32214 static struct file_operations g_file_ops = {
```

```
32215     .owner = THIS_MODULE,
32216     .open = generic_open,
32217     .release = generic_release,
32218     .read = generic_read,
32219     .write = generic_write,
32220 };
32221
32222 struct pipe_dev {
32223     char pipebuf[PIPE_SIZE];
32224     size_t head;
32225     size_t tail;
32226     size_t count;
32227     struct semaphore sem;
32228     wait_queue_head_t wqread;
32229     wait_queue_head_t wqwrite;
32230     struct cdev cdev;
32231 };
32232
32233 static int ndevice = DEF_NDEVICE;
32234 static struct pipe_dev *g_devs;
32235
32236 module_param(ndevice, int, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
32237
32238 static int __init generic_init(void)
32239 {
32240     int result;
32241     int i, k;
32242     dev_t dev;
32243
32244     if ((result = alloc_chrdev_region(&g_dev, 0, ndevice, "pipe-driver")) < 0) {
32245         printk(KERN_INFO "Cannot alloc char driver!...\n");
32246         return result;
32247     }
32248
32249     if ((g_devs = (struct pipe_dev *)kmalloc(sizeof(struct pipe_dev) * ndevice, GFP_KERNEL)) == NULL) {
32250         unregister_chrdev_region(g_dev, ndevice);
32251         printk(KERN_INFO "Cannot allocate memory!..\n");
32252         return -ENOMEM;
32253     }
32254
32255     for (i = 0; i < ndevice; ++i) {
32256         g_devs[i].head = g_devs[i].tail = g_devs[i].count = 0;
32257         init_waitqueue_head(&g_devs[i].wqread);
32258         init_waitqueue_head(&g_devs[i].wqwrite);
32259         sema_init(&g_devs[i].sem, 1);
32260
32261         cdev_init(&g_devs[i].cdev, &g_file_ops);
32262         dev = MKDEV(MAJOR(g_dev), i);
32263         if ((result = cdev_add(&g_devs[i].cdev, dev, 1)) != 0) {
32264             for (k = 0; k < i; ++k)
32265                 cdev_del(&g_devs[i].cdev);
```

```
32266         unregister_chrdev_region(g_dev, ndevice);
32267         kfree(g_devs);
32268         return result;
32269     }
32270 }
32271
32272     printk(KERN_INFO "Pipe driver initialized with %d:%d + %d device
32273             numbers...\n", MAJOR(g_dev), MINOR(g_dev), ndevice); ↵
32274
32275     return 0;
32276 }
32277 static void __exit generic_exit(void)
32278 {
32279     int i;
32280
32281     for (i = 0; i < ndevice; ++i)
32282         cdev_del(&g_devs[i].cdev);
32283
32284     unregister_chrdev_region(g_dev, ndevice);
32285     kfree(g_devs);
32286
32287     printk(KERN_INFO "Goodbye...\n");
32288 }
32289
32290 static int generic_open(struct inode *inodep, struct file *filp)
32291 {
32292     struct pipe_dev *pdev = container_of(inodep->i_cdev, struct pipe_dev,
32293                                         cdev);
32294
32295     filp->private_data = pdev;
32296
32297     printk(KERN_INFO "Pipe device opened!..\n");
32298
32299     return 0;
32300 }
32301
32302 static int generic_release(struct inode *inodep, struct file *filp)
32303 {
32304     printk(KERN_INFO "Pipe device released!..\n");
32305
32306     return 0;
32307 }
32308 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
32309                             loff_t *off)
32310 {
32311     size_t len;
32312     ssize_t result;
32313     struct pipe_dev *pdev = (struct pipe_dev *)filp->private_data;
32314
32315     if (down_interruptible(&pdev->sem))
32316         return -ERESTARTSYS;
```

```
32316
32317     while (pdev->count == 0) {
32318         up(&pdev->sem);
32319
32320         if (filp->f_flags & O_NONBLOCK)
32321             return -EAGAIN;
32322
32323         if (wait_event_interruptible(pdev->wqread, pdev->count > 0))
32324             return -ERESTARTSYS;
32325         if (down_interruptible(&pdev->sem))
32326             return -ERESTARTSYS;
32327     }
32328
32329     size = MIN(size, pdev->count);
32330
32331     if (pdev->head >= pdev->tail)
32332         len = MIN(size, PIPE_SIZE - pdev->head);
32333     else
32334         len = size;
32335
32336     if (copy_to_user(buf, pdev->pipebuf + pdev->head, len) != 0) {
32337         result = -EFAULT;
32338         goto EXIT;
32339     }
32340
32341     if (size > len)
32342         if (copy_to_user(buf + len, pdev->pipebuf, size - len) != 0) {
32343             result = -EFAULT;
32344             goto EXIT;
32345         }
32346
32347     pdev->head = (pdev->head + size) % PIPE_SIZE;
32348     pdev->count -= size;
32349
32350     result = size;
32351
32352     wake_up_all(&pdev->wqwrite);
32353 EXIT:
32354     up(&pdev->sem);
32355
32356     return result;
32357 }
32358
32359 static ssize_t generic_write(struct file *filp, const char *buf, size_t      ↵
32360     size, loff_t *off)
32361 {
32362     size_t len;
32363     ssize_t result;
32364     struct pipe_dev *pdev = (struct pipe_dev *)filp->private_data;
32365
32366     if (down_interruptible(&pdev->sem))
32367         return -ERESTARTSYS;
32368 }
```

```
32368     while (PIPE_SIZE - pdev->count < size) {
32369         up(&pdev->sem);
32370
32371         if (filp->f_flags & O_NONBLOCK)
32372             return -EAGAIN;
32373
32374         if (wait_event_interruptible(pdev->wqwrite, PIPE_SIZE - pdev->count >=
32375                                     size))
32376             return -ERESTARTSYS;
32377
32378         if (down_interruptible(&pdev->sem))
32379             return -ERESTARTSYS;
32380     }
32381
32382     if (pdev->tail >= pdev->head)
32383         len = MIN(size, PIPE_SIZE - pdev->tail);
32384     else
32385         len = size;
32386
32387     if (copy_from_user(pdev->pipebuf + pdev->tail, buf, len) != 0) {
32388         result = -EFAULT;
32389         goto EXIT;
32390     }
32391
32392     if (size > len)
32393         if (copy_from_user(pdev->pipebuf, buf + len, size - len) != 0) {
32394             result = -EFAULT;
32395             goto EXIT;
32396         }
32397
32398     pdev->tail = (pdev->tail + size) % PIPE_SIZE;
32399     pdev->count += size;
32400
32401     result = size;
32402
32403     wake_up_all(&pdev->wqread);
32404
32405     EXIT:
32406         up(&pdev->sem);
32407
32408     return result;
32409 }
32410 module_init(generic_init);
32411 module_exit(generic_exit);
32412
32413 obj-m += $(file).o
32414
32415 all:
32416     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
32417 clean:
32418     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
32419
```

```
32420 /* pipeproc1.c */
32421
32422 #include <stdio.h>
32423 #include <stdlib.h>
32424 #include <string.h>
32425 #include <errno.h>
32426 #include <fcntl.h>
32427 #include <unistd.h>
32428
32429 #define PIPE_SIZE      4096
32430
32431 void exit_sys(const char *msg);
32432
32433 int main(int argc, char *argv[])
32434 {
32435     int fd;
32436     char buf[PIPE_SIZE];
32437     char *str;
32438     size_t len;
32439
32440     if (argc != 2) {
32441         fprintf(stderr, "wrong number of arguments!..\\n");
32442         exit(EXIT_FAILURE);
32443     }
32444
32445     if ((fd = open(argv[1], O_WRONLY)) == -1)
32446         exit_sys("open");
32447
32448     for (;;) {
32449         printf("Enter text:");
32450         fflush(stdout);
32451         fgets(buf, PIPE_SIZE, stdin);
32452         if ((str = strchr(buf, '\\n')) != NULL)
32453             *str = '\\0';
32454         if (!strcmp(buf, "quit"))
32455             break;
32456         len = strlen(buf);
32457         if (write(fd, buf, len) == -1) {
32458             if (errno == EAGAIN) {
32459                 printf("Nonblocking write returns -1 with errno EAGAIN...    ↵
32460                     \\n");
32461                 continue;
32462             }
32463             exit_sys("write");
32464         }
32465
32466         printf("%lu bytes written...\\n", (unsigned long)len);
32467     }
32468
32469     close(fd);
32470
32471     return 0;
```

```
32472 }
32473
32474 void exit_sys(const char *msg)
32475 {
32476     perror(msg);
32477
32478     exit(EXIT_FAILURE);
32479 }
32480
32481 /* pipeproc2.c */
32482
32483 #include <stdio.h>
32484 #include <stdlib.h>
32485 #include <string.h>
32486 #include <errno.h>
32487 #include <fcntl.h>
32488 #include <unistd.h>
32489
32490 #define PIPE_SIZE      4096
32491
32492 void exit_sys(const char *msg);
32493
32494 int main(int argc, char *argv[])
32495 {
32496     int fd;
32497     char buf[PIPE_SIZE + 1];
32498     int len;
32499     ssize_t result;
32500
32501     if (argc != 2) {
32502         fprintf(stderr, "wrong number of arguments!..\n");
32503         exit(EXIT_FAILURE);
32504     }
32505
32506     if ((fd = open(argv[1], O_RDONLY)) == -1)
32507         exit_sys("open");
32508
32509     for (;;) {
32510         printf("How many bytes to read? ");
32511         fflush(stdout);
32512         scanf("%d", &len);
32513         if (!len)
32514             break;
32515         if ((result = read(fd, buf, len)) == -1) {
32516             if (errno == EAGAIN) {
32517                 printf("Nonblocking read return -1 with errno EAGAIN...\n");
32518                 continue;
32519             }
32520             else
32521                 exit_sys("read");
32522         }
32523
32524     buf[result] = '\0';
```

```
32525
32526     printf("%ld bytes read: \"%s\"\n", (long)result, buf);
32527 }
32528 close(fd);
32529 return 0;
32530 }
32531
32532 }
32533 void exit_sys(const char *msg)
32534 {
32535     perror(msg);
32536     exit(EXIT_FAILURE);
32537 }
32538
32539 }
32540
32541 #!/bin/bash
32542
32543 module=$2
32544 mode=666
32545
32546 /sbin/insmod ./${module}.ko ${@:3} || exit 1
32547 major=$(awk "\$2 == \"\$module\" {print \$1}" /proc/devices)
32548
32549 for ((i = 0; i < $1; ++i))
32550 do
32551     rm -f ${module}${i}
32552     mknod ${module}${i} c $major $i
32553     chmod $mode ${module}${i}
32554 done
32555
32556 /
*-----*-----*-----*
```

32557 Aygit sürücüden bilgi okumak için read fonksiyonun, aygit sürücüye bilgi göndermek için ise write fonksiyonun kullanıldığını gördük.

32558 Ancak bazen aygit sürücüye başka komutların da gönderilmesi gerekebilir. Benzer biçimde aygit sürücüden bazı bilgilerin de okunması gerekebilir. Bazen hiç bilgi okumadan ve gönderilmeden aygit sürücüye bazı şeylerin yaptırılması da istenebilir. Bu işlemler read ve write fonksiyonlarıyla yapılmaya çalışılırsa bu read ve write fonksiyonlarının işlevleri farklı olduğu için bu işlem hiç kolay olmaz.

32561 Örneğin yukarıdaki pipe-driverörneğinde biz aygit sürücüden kullandığı FIFO alanın uzunluğunu isteyebiliriz. Ya da bu alanın boyutunu değiştirmek isteyebiliriz. Bu işlemleri read ve write fonksiyonlarıyla yapmaya çalışıksak aygit sürücümüz sanki boruyu temsil eden kuyruktañ okuma yazma yapmak istediğimizi sanacaktır. Tabii bu tür amaçlarla read ve write fonksiyonları yine kullanılabilir. Ancak bu tür tasarımların gerçekleştirimiñleri çok zor olabilmektedir.

32566

32567 İşte aygit sürücüye komut gönderip ondan bilgi almak için genel amaçlı

32568 ioctl siminde özel bir fonksiyon bulunmaktadır. Linux sistemlerinde
32569 ioctl POSIX fonksiyonu sys_ioctl sistem fonksiyonunu çağrırmaktadır. ↗
32570 ioctl fonksiyonunun parametrik yapısı şöyledir:
32571
32572 int ioctl(int fd, unsigned long request, ...);
32573 Fonksiyonun birinci parametresi aygit sürücüye ilişkin dosya
32574 betimleyicisini belirtir. İkinci parametre ileride açıklanacak olan
32575 komut kodudur.
32576 Programcı aygit sürücüsünde farklı komut kodları (yani numaralar)
32577 oluşturur. Sonra bu numaraları switch içerisinde sokarak hangi numara
32578 istekte
32579 bulunulmuşsa onu yapar. ioctl fonksiyonu iki parametreyle ya da üç
32580 parametreyle kullanılmaktadır. Eğer bir veri transferi söz konusu
32581 değilse
32582 ioctl iki argümanla çağrırlar. Ancak bir veri transferi söz konusu ise
32583 ioctl üç argümanla çağrılmalıdır. Bu durumda üçüncü argüman transfer
32584 adres olmalıdır.
32585 Tabii aslında bu üçüncü parametrenin veri transferi ile ilgili olması
32586 dlayısıyla bir adres belirtmesi zorunlu değildir.
32587
32588 Fonksiyon başarı durumunda 0 değerine, başarısızlık durumunda -1 değerine
32589 geri döner. errno uygun biçimde set edilmektedir.
32590
32591 User moddan bir program aygit sürücü için ioctl fonksyonunu
32592 çağrırdığında proses use r moddan kernel moda geçer ve aygit sürücüdeki
32593 file_operations yapısının unlocked_ioctl elemanında belirtilen fonksiyon
32594 çağrırlar. Bu fonksiyonun parametrik yapısı söyle olmalıdır:
32595
32596 long generic_ioctl(struct file *filp, unsigned int cmd, unsigned long
32597 arg);
32598
32599 Bu fonksiyon başarı durumunda 0 değerine başarısızlık durumunda negatif
32600 hata koduna geri dönmelidir. Fakat bazen programcı doğrudan iletilecek
32601 değeri
32602 geri dönüş değerini biçiminde oluşturabilir. Bu durumda geri dönüş değeri
32603 pozitif değer olabilir.
32604
32605 ioctl işleminde fonksiyonun ikinci parametresi olan kontrol kodu belli
32606 bir biçimde göre (konvansiyona göre) oluşturulmaktadır. Bu biçimde göre
32607 kontrol kodu toplam 32 bitlik dört parçadan oluşturulmalıdır. Bu dört
32608 parça istenirse _IOC isimli makro ile birleştirilebilir. Bu makro şu
32609 parametrelere sahiptir:
32610
32611 _IOC(dir, type, nr, size)
32612
32613 dir (direction): Bu iki bitlik bir alandır. Buradaki kullanılacak
32614 sembolik sabitler _IOC_NONE, _IOC_READ, _IOC_WRITE
32615 ve _IOC_READ|_IOC_WRITE biçimindedir. ([30, 31] bitleri)
32616
32617 type: Bu aygit sürücüyü yazan programcı tarafından uydurulması gereken 1
32618

```
byte'lık bir değeridir. Buna "magic number" da denilmektedir. ([8,    ↵
15] bitleri)
32599
32600     nr: Programcı tarafından kontrol koduna verilen sıra numarasıdır.      ↵
            Genellikle aygıt sürücü programcılar 0'dan başlayarak her koda bir      ↵
            numara
32601     vermektedir. ([0, 7] bitleri)
32602
32603     size: Bu alan kaç byte'lık bir transferin yapılacağını belirtmektedir.    ↵
            ([16:29] bitleri)
32604
32605     Örnek bir komut numarası şöyle oluşturulabilir:
32606
32607     #define PIPE_MAGIC          'x'
32608     #define IOC_PIPE_GETBUFSIZE _IOC(_IOC_READ, PIPE_MAGIC, 0, 4)
32609
32610     Aslında _IOC makrosundan daha kolay kullanılabilen aşağıdaki makrolar da    ↵
            oluşturulmuştur:
32611
32612     #ifndef __KERNEL__
32613         #define _IOC_TYPECHECK(t) (sizeof(t))
32614     #endif
32615
32616     #define _IO(type, nr)        _IOC(_IOC_NONE, (type), (nr), 0)
32617     #define _IOR(type, nr, size)  _IOC(_IOC_READ, (type), (nr), (_IOC_TYPECHECK    ↵
            (size)))
32618     #define _IOW(type, nr, size)  _IOC(_IOC_WRITE, (type), (nr), (_IOC_TYPECHECK    ↵
            (size)))
32619     #define _IOWR(type, nr, size) _IOC(_IOC_READ|_IOC_WRITE, (type), (nr),      ↵
            (_IOC_TYPECHECK(size)))
32620
32621     Bu makrolarda artık uzunluk byte olarak değil tür olarak      ↵
            belirtilmelidir. Makrolar bu türleri sizof operatörüne kendisi      ↵
            sokmaktadır. Görüldüğü
32622     gibi _IO makrosu veri transferinin söz konusu olmadığı durumda      ↵
            kullanılır. _IOR aygıt sürücüden okuma yapıldığı durumda, _IOW aygıt      ↵
            sürücüye yazma
32623     yapıldığı durumda, _IOWR ise aygıt sürücüden hem okuma hem de yazma      ↵
            yapıldığı durumlarda kullanılmaktadır. Örneğin:
32624
32625     #define PIPE_MAGIC          'x'
32626     #define IOC_PIPE_GETBUFSIZE _IOR(PIPE_MAGIC, 0, int)
32627
32628     ioctl için kontrol kodları hem aygıt sürücünün içerisindeki hem de user      ↵
            moddan kullanılacağına göre user moddan kullanım için bir başlık      ↵
            dosyasının
32629     oluşturulması uygun olur. Örneğin:
32630
32631     #ifndef PIPE_DRIVER_H_
32632     #define PIPE_DRIVER_H_
32633
32634     #include <sys/ioctl.h>
32635
```

```
32636 #define PIPE_MAGIC          'x'
32637 #define IOC_PIPE_GETBUFSIZE _IOR(PIPE_MAGIC, 0, int)
32638
32639 #endif
32640
32641 Aygit sürücüdeki ioctl fonksiyonunu yazarken iki noktaya dikkat etmek →
32642     gereklidir:
32643
32644 1) ioctl fonksiyonun üçüncü parametresi unsigned long türden olmasına →
32645     karşın aslında genellikle user mod programcısı buraya bir nesnenin →
32646     adresini geçirmektedir. Dolayısıyla bu transfer adresine aktarım →
32647     gerekmektedir. Bunun için copy_to_user, copy_from_user, put_user, →
32648     get_user
32649 gibi adresin geçerliliğini sorguladıktan sonra transfers yapan →
32650     fonksiyonlar kullanılabilir.
32651
32652 -----
32653 -----*/
32654 #include <linux/module.h>
32655 #include <linux/kernel.h>
32656 #include <linux/fs.h>
32657 #include <linux/cdev.h>
32658 #include <linux/uaccess.h>
32659 #include <linux/semaphore.h>
32660 #include <linux/slab.h>
32661
32662 MODULE_LICENSE("GPL");
32663 MODULE_DESCRIPTION("General Character Device Driver");
32664 MODULE_AUTHOR("Kaan Aslan");
32665
32666 #define MIN(a, b)      ((a) < (b) ? (a) : (b))
32667
32668 #define DEF_NDEVICE    10
32669 #define PIPE_SIZE       4096
32670
32671 #define PIPE_MAGIC          'x'
32672 #define IOC_PIPE_GETBUFSIZE _IOR(PIPE_MAGIC, 0, int)
32673
32674 static int generic_open(struct inode *inodep, struct file *filp);
32675 static int generic_release(struct inode *inodep, struct file *filp);
32676 static ssize_t generic_read(struct file *filp, char *buf, size_t size, →
32677     loff_t *off);
```

```
32677 static ssize_t generic_write(struct file *filp, const char *buf, size_t      ↵
32678     size, loff_t *off);
32679 long generic_ioctl(struct file *filp, unsigned int cmd, unsigned long arg);
32680
32681 static dev_t g_dev;
32682 static struct file_operations g_file_ops = {
32683     .owner = THIS_MODULE,
32684     .open = generic_open,
32685     .release = generic_release,
32686     .read = generic_read,
32687     .write = generic_write,
32688     .unlocked_ioctl = generic_ioctl,
32689 };
32690
32691 struct pipe_dev {
32692     char pipebuf[PIPE_SIZE];
32693     size_t head;
32694     size_t tail;
32695     size_t count;
32696     struct semaphore sem;
32697     wait_queue_head_t wqread;
32698     wait_queue_head_t wqwrite;
32699     struct cdev cdev;
32700 };
32701
32702 static int ndevice = DEF_NDEVICE;
32703 static struct pipe_dev *g_devs;
32704
32705 module_param(ndevice, int, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
32706
32707 static int __init generic_init(void)
32708 {
32709     int result;
32710     int i, k;
32711     dev_t dev;
32712
32713     if ((result = alloc_chrdev_region(&g_dev, 0, ndevice, "pipe-driver")) < ↵
32714         0) {
32715         printk(KERN_INFO "Cannot alloc char driver!...\n");
32716         return result;
32717     }
32718     if ((g_devs = (struct pipe_dev *)kmalloc(sizeof(struct pipe_dev) *      ↵
32719         ndevice, GFP_KERNEL)) == NULL) {
32720         unregister_chrdev_region(g_dev, ndevice);
32721         printk(KERN_INFO "Cannot allocate memory!..\n");
32722         return -ENOMEM;
32723     }
32724     for (i = 0; i < ndevice; ++i) {
32725         g_devs[i].head = g_devs[i].tail = g_devs[i].count = 0;
32726         init_waitqueue_head(&g_devs[i].wqread);
```

```
32727     init_waitqueue_head(&g_devs[i].wqwrite);
32728     sema_init(&g_devs[i].sem, 1);
32729
32730     cdev_init(&g_devs[i].cdev, &g_file_ops);
32731     dev = MKDEV(MAJOR(g_dev), i);
32732     if ((result = cdev_add(&g_devs[i].cdev, dev, 1)) != 0) {
32733         for (k = 0; k < i; ++k)
32734             cdev_del(&g_devs[i].cdev);
32735         unregister_chrdev_region(g_dev, ndevice);
32736         kfree(g_devs);
32737         return result;
32738     }
32739 }
32740
32741     printk(KERN_INFO "Pipe driver initialized with %d:%d + %d device numbers...\n", MAJOR(g_dev), MINOR(g_dev), ndevice); ↵
32742
32743     return 0;
32744 }
32745
32746 static void __exit generic_exit(void)
32747 {
32748     int i;
32749
32750     for (i = 0; i < ndevice; ++i)
32751         cdev_del(&g_devs[i].cdev);
32752
32753     unregister_chrdev_region(g_dev, ndevice);
32754     kfree(g_devs);
32755
32756     printk(KERN_INFO "Goodbye...\n");
32757 }
32758
32759 static int generic_open(struct inode *inodep, struct file *filp)
32760 {
32761     struct pipe_dev *pdev = container_of(inodep->i_cdev, struct pipe_dev, cdev);
32762
32763     filp->private_data = pdev;
32764
32765     printk(KERN_INFO "Pipe device opened!..\n");
32766
32767     return 0;
32768 }
32769
32770 static int generic_release(struct inode *inodep, struct file *filp)
32771 {
32772     printk(KERN_INFO "Pipe device released!..\n");
32773
32774     return 0;
32775 }
32776
32777 static ssize_t generic_read(struct file *filp, char *buf, size_t size, ↵
```

```
    loff_t *off)
32778 {
32779     size_t len;
3280     ssize_t result;
3281     struct pipe_dev *pdev = (struct pipe_dev *)filp->private_data;
3282
3283     if (down_interruptible(&pdev->sem))
3284         return -ERESTARTSYS;
3285
3286     while (pdev->count == 0) {
3287         up(&pdev->sem);
3288
3289         if (filp->f_flags & O_NONBLOCK)
3290             return -EAGAIN;
3291
3292         if (wait_event_interruptible(pdev->wqread, pdev->count > 0))
3293             return -ERESTARTSYS;
3294         if (down_interruptible(&pdev->sem))
3295             return -ERESTARTSYS;
3296     }
3297
3298     size = MIN(size, pdev->count);
3299
3300     if (pdev->head >= pdev->tail)
3301         len = MIN(size, PIPE_SIZE - pdev->head);
3302     else
3303         len = size;
3304
3305     if (copy_to_user(buf, pdev->pipebuf + pdev->head, len) != 0) {
3306         result = -EFAULT;
3307         goto EXIT;
3308     }
3309
3310     if (size > len)
3311         if (copy_to_user(buf + len, pdev->pipebuf, size - len) != 0) {
3312             result = -EFAULT;
3313             goto EXIT;
3314         }
3315
3316     pdev->head = (pdev->head + size) % PIPE_SIZE;
3317     pdev->count -= size;
3318
3319     result = size;
3320
3321     wake_up_all(&pdev->wqwrite);
3322 EXIT:
3323     up(&pdev->sem);
3324
3325     return result;
3326 }
3327
3328 static ssize_t generic_write(struct file *filp, const char *buf, size_t
3329     size, loff_t *off)
```

```
32829 {  
3290     size_t len;  
3291     ssize_t result;  
3292     struct pipe_dev *pdev = (struct pipe_dev *)filp->private_data;  
3293  
3294     if (down_interruptible(&pdev->sem))  
3295         return -ERESTARTSYS;  
3296  
3297     while (PIPE_SIZE - pdev->count < size) {  
3298         up(&pdev->sem);  
3299  
3300         if (filp->f_flags & O_NONBLOCK)  
3301             return -EAGAIN;  
3302  
3303         if (wait_event_interruptible(pdev->wqwrite, PIPE_SIZE - pdev->count ↗  
3304             >= size))  
3305             return -ERESTARTSYS;  
3306  
3307         if (down_interruptible(&pdev->sem))  
3308             return -ERESTARTSYS;  
3309     }  
3310  
3311     if (pdev->tail >= pdev->head)  
3312         len = MIN(size, PIPE_SIZE - pdev->tail);  
3313     else  
3314         len = size;  
3315  
3316     if (copy_from_user(pdev->pipebuf + pdev->tail, buf, len) != 0) {  
3317         result = -EFAULT;  
3318         goto EXIT;  
3319     }  
3320  
3321     if (size > len)  
3322         if (copy_from_user(pdev->pipebuf, buf + len, size - len) != 0) {  
3323             result = -EFAULT;  
3324             goto EXIT;  
3325         }  
3326  
3327     pdev->tail = (pdev->tail + size) % PIPE_SIZE;  
3328     pdev->count += size;  
3329  
3330     result = size;  
3331  
3332     wake_up_all(&pdev->wqread);  
3333  
3334 EXIT:  
3335     up(&pdev->sem);  
3336  
3337     return result;  
3338 }  
3339  
3340 long generic_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)  
3341 {
```

```
32881     int *pi = (int *)arg;
32882
32883     switch (cmd) {
32884         case IOC_PIPE_GETBUFSIZE:
32885             if (put_user(PIPE_BUF, pi) != 0)
32886                 return -EFAULT;
32887             break;
32888         default:
32889             return -ENOTTY;
32890     }
32891
32892     return 0;
32893 }
32894
32895 module_init(generic_init);
32896 module_exit(generic_exit);
32897
32898 obj-m += $(file).o
32899
32900 all:
32901     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
32902 clean:
32903     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
32904
32905 /* pipe-driver.h */
32906
32907 #ifndef PIPE_DRIVER_H_
32908 #define PIPE_DRIVER_H_
32909
32910 #include <sys/ioctl.h>
32911
32912 /* Driver Control Codes */
32913
32914 #define PIPE_MAGIC      'x'
32915 #define IOC_PIPE_GETBUFSIZE    _IOR(PIPE_MAGIC, 0, int)
32916
32917 #endif
32918
32919 /* pipeproc-test.c */
32920
32921 #include <stdio.h>
32922 #include <stdlib.h>
32923 #include <string.h>
32924 #include <fcntl.h>
32925 #include <unistd.h>
32926 #include <sys/ioctl.h>
32927 #include "pipe-driver.h"
32928
32929 #define PIPE_SIZE      4096
32930
32931 void exit_sys(const char *msg);
32932
32933 int main(void)
```

```
32934 {
32935     int fd;
32936     int size;
32937
32938     if ((fd = open("pipe-driver0", O_WRONLY)) == -1)
32939         exit_sys("open");
32940
32941     if (ioctl(fd, IOC PIPE _GETBUFSIZE, &size) == -1)
32942         exit_sys("ioctl");
32943
32944     printf("Pipe buffer size: %d\n", size);
32945
32946     close(fd);
32947
32948     return 0;
32949 }
32950
32951 void exit_sys(const char *msg)
32952 {
32953     perror(msg);
32954
32955     exit(EXIT_FAILURE);
32956 }
32957
32958 #!/bin/bash
32959
32960 module=$2
32961 mode=666
32962
32963 /sbin/insmod ./${module}.ko ${@:3} || exit 1
32964 major=$(awk "\$2 == \"\$module\" {print \$1}" /proc/devices)
32965
32966 for ((i = 0; i < $1; ++i))
32967 do
32968     rm -f ${module}$i
32969     mknod ${module}$i c $major $i
32970     chmod $mode ${module}$i
32971 done
32972
32973 /
*-----*
-----*
-----*
```

32974 Aşağıdaki örnekte birkaç ioctl kodu işlenmiştir. Borudan atmadan byte →
okumak için özel peek'li ioctl fonksiyonları yazılmıştır.

32975 ioctl fonksiyonun tek bir parametresi olabilir. Birden fazla parametre →
bir yapı biçiminde fonksiyona verilmelidir. Yukarıda da belirtildiği →
gibi

32976 ioctl fonksiyonun geri dönüş değeri başarısızlık durumunda negatif hata →
kodu, başarı durumunda tipik olarak 0'dır. Ancak başarı durumunda 0 →
olması

32977 mutlak bir zorunluluk değildir. Programcılar işlerini uzatmamak için →
başarı durumunda anlamlı başka bir değerle de geri dönenmektedir.

32978 Aşağıda örnekte peek için iki ayrı ioctl fonksiyonu bulundurulmuştur. →

```
    IOC PIPE_PEEK fonksiyonunda okunan byte miktarı ioctl fonksiyonun  
32979    geri dönüş değeri biçiminde bize verilmektedir. Halbuki      ↵  
    IOC PIPE_PEEK_BIDIREC fonksiyonunda okunan byte miktarı yapının      ↵  
    yeniden size elemanına  
32980    yerleştirilmiştir.  
32981  -----*/  
32982  
32983 /* pipe-driver.c */  
32984  
32985 #include <linux/module.h>  
32986 #include <linux/kernel.h>  
32987 #include <linux/fs.h>  
32988 #include <linux/cdev.h>  
32989 #include <linux/uaccess.h>  
32990 #include <linux/semaphore.h>  
32991 #include <linux/slab.h>  
32992 #include <linux/atomic.h>  
32993  
32994 MODULE_LICENSE("GPL");  
32995 MODULE_DESCRIPTION("General Character Device Driver");  
32996 MODULE_AUTHOR("Kaan Aslan");  
32997  
32998 #define MIN(a, b) ((a) < (b) ? (a) : (b))  
32999  
33000 #define DEF_NDEVICE 10  
33001 #define PIPE_SIZE 4096  
33002  
33003 #define PIPE_MAGIC 'x'  
33004 #define IOC_PIPE_GETBUFSIZE _IOR(PIPE_MAGIC, 0, int)  
33005 #define IOC_PIPE_GETMAXDEVICE _IOR(PIPE_MAGIC, 1, int)  
33006 #define IOC_PIPE_GETOPENCOUNT _IOR(PIPE_MAGIC, 2, int)  
33007 #define IOC_PIPE_PEEK _IOC(_IOC_READ|_IOC_WRITE, PIPE_MAGIC, 3,      ↵  
    4096)  
33008 #define IOC_PIPE_PEEK_BIDIREC _IOC(_IOC_READ|_IOC_WRITE, PIPE_MAGIC, 4,      ↵  
    4096)  
33009  
33010 static int generic_open(struct inode *inodep, struct file *filp);  
33011 static int generic_release(struct inode *inodep, struct file *filp);  
33012 static ssize_t generic_read(struct file *filp, char *buf, size_t size,      ↵  
    loff_t *off);  
33013 static ssize_t generic_write(struct file *filp, const char *buf, size_t      ↵  
    size, loff_t *off);  
33014 static long generic_ioctl(struct file *filp, unsigned int cmd, unsigned long      ↵  
    arg);  
33015  
33016 static dev_t g_dev;  
33017 static struct file_operations g_file_ops = {  
33018     .owner = THIS_MODULE,  
33019     .open = generic_open,  
33020     .release = generic_release,  
33021     .read = generic_read,  
33022     .write = generic_write,
```

```
33023     .unlocked_ioctl = generic_ioctl,
33024 };
33025
33026 struct pipe_dev {
33027     char pipebuf[PIPE_SIZE];
33028     size_t head;
33029     size_t tail;
33030     size_t count;
33031     struct semaphore sem;
33032     wait_queue_head_t wqread;
33033     wait_queue_head_t wqwrite;
33034     struct cdev cdev;
33035 };
33036
33037 struct pipe_peek {
33038     int size;
33039     char *buf;
33040 };
33041
33042 static int read_peek(struct file *filp, const struct pipe_peek *pp);
33043
33044 static int ndevice = DEF_NDEVICE;
33045 static struct pipe_dev *g_devs;
33046 static atomic_t g_nopens;
33047
33048 module_param(ndevice, int, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
33049
33050 static int __init generic_init(void)
33051 {
33052     int result;
33053     int i, k;
33054     dev_t dev;
33055
33056     if ((result = alloc_chrdev_region(&g_dev, 0, ndevice, "pipe-driver")) < 0)
33057     {
33058         printk(KERN_INFO "Cannot alloc char driver!...\n");
33059         return result;
33060     }
33061
33062     if ((g_devs = (struct pipe_dev *)kmalloc(sizeof(struct pipe_dev) *
33063                                             ndevice, GFP_KERNEL)) == NULL) {
33064         unregister_chrdev_region(g_dev, ndevice);
33065         printk(KERN_INFO "Cannot allocate memory!..\n");
33066         return -ENOMEM;
33067     }
33068
33069     for (i = 0; i < ndevice; ++i) {
33070         g_devs[i].head = g_devs[i].tail = g_devs[i].count = 0;
33071         init_waitqueue_head(&g_devs[i].wqread);
33072         init_waitqueue_head(&g_devs[i].wqwrite);
33073         sema_init(&g_devs[i].sem, 1);
33074
33075         cdev_init(&g_devs[i].cdev, &g_file_ops);
```

```
33074         dev = MKDEV(MAJOR(g_dev), i);
33075         if ((result = cdev_add(&g_devs[i].cdev, dev, 1)) != 0) {
33076             for (k = 0; k < i; ++k)
33077                 cdev_del(&g_devs[i].cdev);
33078             unregister_chrdev_region(g_dev, ndevice);
33079             kfree(g_devs);
33080             return result;
33081         }
33082     }
33083
33084     printk(KERN_INFO "Pipe driver initialized with %d:%d + %d device
33085             numbers...\n", MAJOR(g_dev), MINOR(g_dev), ndevice); ↵
33086
33087     return 0;
33088 }
33089 static void __exit generic_exit(void)
33090 {
33091     int i;
33092
33093     for (i = 0; i < ndevice; ++i)
33094         cdev_del(&g_devs[i].cdev);
33095
33096     unregister_chrdev_region(g_dev, ndevice);
33097     kfree(g_devs);
33098
33099     printk(KERN_INFO "Goodbye...\n");
33100 }
33101
33102 static int generic_open(struct inode *inodep, struct file *filp)
33103 {
33104     struct pipe_dev *pdev = container_of(inodep->i_cdev, struct pipe_dev,
33105             cdev);
33106
33107     filp->private_data = pdev;
33108     atomic_inc(&g_nopens);
33109
33110     printk(KERN_INFO "Pipe device opened!..\n");
33111
33112     return 0;
33113 }
33114 static int generic_release(struct inode *inodep, struct file *filp)
33115 {
33116     printk(KERN_INFO "Pipe device released!..\n");
33117
33118     atomic_dec(&g_nopens);
33119
33120     return 0;
33121 }
33122
33123 static ssize_t generic_read(struct file *filp, char *buf, size_t size,
33124             loff_t *off) ↵
```

```
33124  {
33125      size_t len;
33126      ssize_t result;
33127      struct pipe_dev *pdev = (struct pipe_dev *)filp->private_data;
33128
33129      if (down_interruptible(&pdev->sem))
33130          return -ERESTARTSYS;
33131
33132      while (pdev->count == 0) {
33133          up(&pdev->sem);
33134
33135          if (filp->f_flags & O_NONBLOCK)
33136              return -EAGAIN;
33137
33138          if (wait_event_interruptible(pdev->wqread, pdev->count > 0))
33139              return -ERESTARTSYS;
33140          if (down_interruptible(&pdev->sem))
33141              return -ERESTARTSYS;
33142      }
33143
33144      size = MIN(size, pdev->count);
33145
33146      if (pdev->head >= pdev->tail)
33147          len = MIN(size, PIPE_SIZE - pdev->head);
33148      else
33149          len = size;
33150
33151      if (copy_to_user(buf, pdev->pipebuf + pdev->head, len) != 0) {
33152          result = -EFAULT;
33153          goto EXIT;
33154      }
33155
33156      if (size > len)
33157          if (copy_to_user(buf + len, pdev->pipebuf, size - len) != 0) {
33158              result = -EFAULT;
33159              goto EXIT;
33160          }
33161
33162      pdev->head = (pdev->head + size) % PIPE_SIZE;
33163      pdev->count -= size;
33164
33165      result = size;
33166
33167      wake_up_all(&pdev->wqwrite);
33168 EXIT:
33169      up(&pdev->sem);
33170
33171      return result;
33172  }
33173
33174 static ssize_t generic_write(struct file *filp, const char *buf, size_t      ↵
33175     size, loff_t *off)
33176  {
```

```
33176     size_t len;
33177     ssize_t result;
33178     struct pipe_dev *pdev = (struct pipe_dev *)filp->private_data;
33179
33180     if (down_interruptible(&pdev->sem))
33181         return -ERESTARTSYS;
33182
33183     while (PIPE_SIZE - pdev->count < size) {
33184         up(&pdev->sem);
33185
33186         if (filp->f_flags & O_NONBLOCK)
33187             return -EAGAIN;
33188
33189         if (wait_event_interruptible(pdev->wqwrite, PIPE_SIZE - pdev->count >= size))
33190             return -ERESTARTSYS;
33191
33192         if (down_interruptible(&pdev->sem))
33193             return -ERESTARTSYS;
33194     }
33195
33196     if (pdev->tail >= pdev->head)
33197         len = MIN(size, PIPE_SIZE - pdev->tail);
33198     else
33199         len = size;
33200
33201     if (copy_from_user(pdev->pipebuf + pdev->tail, buf, len) != 0) {
33202         result = -EFAULT;
33203         goto EXIT;
33204     }
33205
33206     if (size > len)
33207         if (copy_from_user(pdev->pipebuf, buf + len, size - len) != 0) {
33208             result = -EFAULT;
33209             goto EXIT;
33210         }
33211
33212     pdev->tail = (pdev->tail + size) % PIPE_SIZE;
33213     pdev->count += size;
33214
33215     result = size;
33216
33217     wake_up_all(&pdev->wqread);
33218
33219 EXIT:
33220     up(&pdev->sem);
33221
33222     return result;
33223 }
33224
33225 static long generic_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
33226 {
```

```
33227     int *pi;
33228     struct pipe_peek peek;
33229     int result;
33230
33231     switch (cmd) {
33232         case IOC_PIPE_GETBUFSIZE:
33233             pi = (int *)arg;
33234             if (put_user(PIPE_BUF, pi) != 0)
33235                 return -EFAULT;
33236             break;
33237         case IOC_PIPE_GETMAXDEVICE:
33238             pi = (int *)arg;
33239             if (put_user(ndevice, pi) != 0)
33240                 return -EFAULT;
33241             break;
33242         case IOC_PIPE_GETOPENCOUNT:
33243             pi = (int *)arg;
33244             if (put_user(atomic_read(&g_nopens), pi) != 0)
33245                 return -EFAULT;
33246             break;
33247
33248         case IOC_PIPE_PEEK:
33249             if (copy_from_user(&peek, (void *)arg, sizeof(struct
33250                                         pipe_peek)) != 0)
33251                 return -EFAULT;
33252             return read_peek(filp, &peek);
33253
33254         case IOC_PIPE_PEEK_BIDIREC:
33255             if (copy_from_user(&peek, (void *)arg, sizeof(struct
33256                                         pipe_peek)) != 0)
33257                 return -EFAULT;
33258             result = read_peek(filp, &peek);
33259             peek.size = result;
33260             if (copy_to_user((void *)arg, &peek, sizeof(struct pipe_peek)) != 0)
33261                 return -EFAULT;
33262             break;
33263
33264     default:
33265         return -ENOTTY;
33266     }
33267 }
33268
33269 static int read_peek(struct file *filp, const struct pipe_peek *pp)
33270 {
33271     size_t len;
33272     ssize_t result;
33273     ssize_t size;
33274     struct pipe_dev *pdev = (struct pipe_dev *)filp->private_data;
33275
33276     if (down_interruptible(&pdev->sem))
```

```
33277         return -ERESTARTSYS;
33278
33279     if (pdev->count == 0) {
33280         result = 0;
33281         goto EXIT;
33282     }
33283
33284     size = MIN(pp->size, pdev->count);
33285
33286     if (pdev->head >= pdev->tail)
33287         len = MIN(size, PIPE_SIZE - pdev->head);
33288     else
33289         len = size;
33290
33291     if (copy_to_user(pp->buf, pdev->pipebuf + pdev->head, len) != 0) {
33292         result = -EFAULT;
33293         goto EXIT;
33294     }
33295
33296     if (size > len)
33297         if (copy_to_user(pp->buf + len, pdev->pipebuf, size - len) != 0) {
33298             result = -EFAULT;
33299             goto EXIT;
33300         }
33301
33302     result = size;
33303 EXIT:
33304     up(&pdev->sem);
33305
33306     return result;
33307 }
33308
33309 module_init(generic_init);
33310 module_exit(generic_exit);
33311
33312 obj-m += $(file).o
33313
33314 all:
33315     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
33316 clean:
33317     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
33318
33319 /* pipe-driver.h */
33320
33321 #ifndef PIPE_DRIVER_H_
33322 #define PIPE_DRIVER_H_
33323
33324 #include <sys/ioctl.h>
33325
33326 /* Type declarations */
33327
33328 struct pipe_peek {
33329     int size;
```

```
33330     char *buf;
33331 };
33332
33333 /* Driver Control Codes */
33334
33335 #define PIPE_MAGIC      'x'
33336 #define IOC_PIPE_GETBUFSIZE    _IOR(PIPE_MAGIC, 0, int)
33337 #define IOC_PIPE_GETMAXDEVICE   _IOR(PIPE_MAGIC, 1, int)
33338 #define IOC_PIPE_GETOPENCOUNT    _IOR(PIPE_MAGIC, 2, int)
33339 #define IOC_PIPE_PEEK          _IOC(_IOC_READ|_IOC_WRITE, PIPE_MAGIC, 3, 4096)
33340 #define IOC_PIPE_PEEK_BIDIREC   _IOC(_IOC_READ|_IOC_WRITE, PIPE_MAGIC, 4, 4096)
33341
33342 #endif
33343 /* pipeproc-test.c */
33344
33345 #include <stdio.h>
33346 #include <stdlib.h>
33347 #include <string.h>
33348 #include <fcntl.h>
33349 #include <unistd.h>
33350 #include <sys/ioctl.h>
33351 #include "pipe-driver.h"
33352
33353 #define PIPE_SIZE      4096
33354
33355 void exit_sys(const char *msg);
33356
33357 int main(void)
33358 {
33359     int fd;
33360     int size, ndevice, nopens;
33361     struct pipe_peek pp;
33362     char buf[11];
33363     int result;
33364
33365     if ((fd = open("pipe-driver0", O_WRONLY)) == -1)
33366         exit_sys("open");
33367
33368     if (ioctl(fd, IOC_PIPE_GETBUFSIZE, &size) == -1)
33369         exit_sys("ioctl");
33370
33371     printf("Pipe buffer size: %d\n", size);
33372
33373     if (ioctl(fd, IOC_PIPE_GETMAXDEVICE, &ndevice) == -1)
33374         exit_sys("ioctl");
33375
33376     printf("Maximum device count: %d\n", ndevice);
33377
33378     if (ioctl(fd, IOC_PIPE_GETOPENCOUNT, &nopens) == -1)
33379         exit_sys("ioctl");
33380
```

```
33381     printf("Number of opens: %d\n", nopens);
33382
33383     pp.size = 10;
33384     pp.buf = buf;
33385
33386     if ((result = ioctl(fd, IOC_PIPE_PEEK, &pp)) == -1)
33387         exit_sys("ioctl");
33388
33389     buf[result] = '\0';
33390     printf("%d bytes poke: %s\n", result, buf);
33391
33392     if ((result = ioctl(fd, IOC_PIPE_PEEK, &pp)) == -1)
33393         exit_sys("ioctl");
33394
33395     buf[pp.size] = '\0';
33396     printf("%d bytes poke: %s\n", pp.size, buf);
33397
33398     getchar();
33399
33400     close(fd);
33401
33402     return 0;
33403 }
33404
33405 void exit_sys(const char *msg)
33406 {
33407     perror(msg);
33408
33409     exit(EXIT_FAILURE);
33410 }
33411
33412 #!/bin/bash
33413
33414 module=$2
33415 mode=666
33416
33417 /sbin/insmod ./${module}.ko ${@:3} || exit 1
33418 major=$(awk "$2 == \"$module\" {print $1}" /proc/devices)
33419
33420 for ((i = 0; i < $1; ++i))
33421 do
33422     rm -f ${module}$i
33423     mknod ${module}$i c $major $i
33424     chmo
33425
33426 /
*-----*-----*-----*
-----*
-----*
```

33427 Anımsanacağı gibi proc dosya sistemi disk tabanlı bir dosya sistemi →
33428 değildir. Kernel çalışması sırasında dış dümyaya bilgi vermek →
33429 için bazen de davranışını dış dünyadan gelen verilerle değiştirebilmek →
33429 için proc dosya sistemini kullanmaktadır. Daha sonra proc →
33429 gibi sys isimli dosya sistemi de Linuc sistemlerine eklenmiştir.

33430
33431 proc dosya sistemi aslında yalnızca kernel tarafından değil aygit →
sürücüler tarafından da kullanılabilir. Ancak bu dosya →
sisteminin
33432 içerisinde user moddan dosyalar ya da dizinler yaratılamamaktadır.
33433
33434 proc dosya sisteminin kernel ve aygit sürücüler tarafından →
kullanılmasına ilişkin fonksiyonlar birkaç kere değişik kernel →
versiyonlarında değiştirilmiştir.
33435 Dolayısıyla eski kernel'larda çalışan kodlar yeni kernel'larda →
derlenmeyecektir. Biz burada en yeni fonksiyonları ele alacağız.
33436
33437 proc dosya sisteminde bir dosya yaratılmak için proc_create isimli →
fonksiyon kullanılmaktadır.
33438
33439 struct proc_dir_entry *proc_create(const char *name, umode_t mode, →
struct proc_dir_entry *parent, const struct proc_ops *proc_ops);
33440
33441 Fonksiyonun birinci parametresi yaratılacak dosyanın ismini belirtir. →
İkinci parametresi erişim haklarını belirtmektedir. Bu parametre 0 →
geçilirse
33442 default erişim hakları kullanılır. Üçüncü parametre dosyanın hangi →
dizinde yaratılacağını belirtmektedir. Bu parametre NULL geçilirse →
dosya ana
33443 /proc dizini içerisinde yaratılır. Son parametre proc dosya sistemindeki →
ilgi dosyaya yazma ve okuma yaplığında çalıştırılacak fonksiyonları →
belirtir.
33444 Aslında birkaç sene önceki kernel'larda bu fonksiyonun son parametresi →
struct proc_ops biçiminde değil, struct file_operations biçimindeydi. →
Dolayısıyla
33445 kernelinizde hangi fonksiyonun bulunuyor olduğuna dikkat ediniz. Kursun →
yapıldığı sistemde bu fonksiyonun son parametresi struct →
file_operations biçimindedir.
33446 Fonksiyon başarı durumunda yaratılan dosyanın bilgilerini içeren →
proc_dir_entry türünden bir yapı nesnesinin adresiyle, başarısızlık →
durumunda NULL adresle
33447 geri dönmektedir.
33448
33449 proc dosya sisteminde yaratılan dosya remove_proc_entry fonksiyonuyla →
silinmektedir.
33450
33451 void remove_proc_entry(const char *name, struct proc_dir_entry *parent);
33452
33453 Aşağıdaki örnekte modülün init fonksiyonunda proc dosya sisteminin →
kökünde "generic-char-driver" isimli bir dosya yaratılmıştır. Modülün →
exit fonksiyonunda
33454 bu dosya remove_proc_entry fonksiyonuyla silinmiştir.
33455
33456 -----*/
33457
33458 /* generic-char-driver.c */
33459

```
33460 #include <linux/module.h>
33461 #include <linux/kernel.h>
33462 #include <linux/fs.h>
33463 #include <linux/cdev.h>
33464 #include <linux/proc_fs.h>
33465
33466 MODULE_LICENSE("GPL");
33467 MODULE_DESCRIPTION("General Character Device Driver");
33468 MODULE_AUTHOR("Kaan Aslan");
33469
33470 static dev_t g_dev;
33471 static struct cdev *g_cdev;
33472 static struct file_operations g_file_ops = {
33473     .owner = THIS_MODULE,
33474 };
33475
33476 static ssize_t proc_read(struct file *filp, char *buf, size_t size, loff_t *off);
33477
33478 static struct file_operations g_proc_ops = {
33479     .owner = THIS_MODULE,
33480     .read = proc_read,
33481 };
33482
33483 static int __init generic_init(void)
33484 {
33485     int result;
33486
33487     if ((result = alloc_chrdev_region(&g_dev, 0, 1, "generic-char-driver")) < 0) {
33488         printk(KERN_INFO "Cannot alloc char driver!...\n");
33489         return result;
33490     }
33491
33492     if ((g_cdev = cdev_alloc()) == NULL) {
33493         printk(KERN_INFO "Cannot allocate cdev!..\n");
33494         return -ENOMEM;
33495     }
33496
33497     g_cdev->owner = THIS_MODULE;
33498     g_cdev->ops = &g_file_ops;
33499
33500     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
33501         unregister_chrdev_region(g_dev, 1);
33502         printk(KERN_INFO "Cannot add character device driver!...\n");
33503         return result;
33504     }
33505
33506     if (proc_create("generic-char-driver", S_IRUSR|S_IRGRP|S_IROTH, NULL, &g_proc_ops) == NULL) {
33507         unregister_chrdev_region(g_dev, 1);
33508         cdev_del(g_cdev);
33509         printk(KERN_INFO "Cannot create proc file!...\n");
```

```
33510         return -ENOMEM;
33511     }
33512
33513     printk(KERN_INFO "Module initialized with %d:%d device number...\n",
33514           MAJOR(g_dev), MINOR(g_dev));
33515
33516     return 0;
33517 }
33518 static void __exit generic_exit(void)
33519 {
33520     cdev_del(g_cdev);
33521     unregister_chrdev_region(g_dev, 1);
33522     remove_proc_entry("generic-char-driver", NULL);
33523
33524     printk(KERN_INFO "Goodbye...\n");
33525 }
33526
33527 static ssize_t proc_read(struct file *filp, char *buf, size_t size, loff_t
33528 *off)
33529 {
33530     return 0;
33531 }
33532 module_init(generic_init);
33533 module_exit(generic_exit);
33534
33535 obj-m += $(file).o
33536
33537 all:
33538     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
33539 clean:
33540     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
33541
33542 #!/bin/bash
33543
33544 module=$1
33545 mode=666
33546
33547 /sbin/insmod ./${module}.ko ${@:2} || exit 1
33548 major=$(awk "\$2 == \"\$module\" {print \$1}" /proc/devices)
33549 rm -f $module
33550 mknod $module c $major 0
33551 chmod $mode $module
33552
33553 /
33554     *
33555     -----
33556     proc dosya sisteminde yaratılmış olan dosyadan okuma ve yazma yapılması
33557 -----*/
33558 /* generic-char-driver.c */
```

```
33558
33559 #include <linux/module.h>
33560 #include <linux/kernel.h>
33561 #include <linux/fs.h>
33562 #include <linux/cdev.h>
33563 #include <linux/uaccess.h>
33564 #include <linux/proc_fs.h>
33565
33566 #define BUF_SIZE          4096
33567
33568 MODULE_LICENSE("GPL");
33569 MODULE_DESCRIPTION("General Character Device Driver");
33570 MODULE_AUTHOR("Kaan Aslan");
33571
33572 static dev_t g_dev;
33573 static struct cdev *g_cdev;
33574 static struct file_operations g_file_ops = {
33575     .owner = THIS_MODULE,
33576 };
33577
33578 static ssize_t proc_read(struct file *filp, char *buf, size_t size, loff_t  ↵
 *off);
33579 static ssize_t proc_write(struct file *filp, const char *buf, size_t size,   ↵
 loff_t *off);
33580 static loff_t  proc_llseek(struct file *filp, loff_t off, int whence);
33581
33582 static struct file_operations g_proc_ops = {
33583     .owner = THIS_MODULE,
33584     .read = proc_read,
33585     .write = proc_write,
33586     .llseek = proc_llseek,
33587 };
33588
33589 static char g_text[BUF_SIZE] = "this is a test\n";
33590
33591 static int __init generic_init(void)
33592 {
33593     int result;
33594
33595     if ((result = alloc_chrdev_region(&g_dev, 0, 1, "generic-char-driver"))  ↵
 < 0) {
33596         printk(KERN_INFO "Cannot alloc char driver!...\n");
33597         return result;
33598     }
33599
33600     if ((g_cdev = cdev_alloc()) == NULL) {
33601         printk(KERN_INFO "Cannot allocate cdev!..\n");
33602         return -ENOMEM;
33603     }
33604
33605     g_cdev->owner = THIS_MODULE;
33606     g_cdev->ops = &g_file_ops;
33607 }
```

```
33608     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
33609         unregister_chrdev_region(g_dev, 1);
33610         printk(KERN_INFO "Cannot add character device driver!...\n");
33611         return result;
33612     }
33613
33614     if (proc_create("generic-char-driver", S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH, NULL,
33615                     &g_proc_ops) == NULL) {
33616         unregister_chrdev_region(g_dev, 1);
33617         cdev_del(g_cdev);
33618         printk(KERN_INFO "Cannot create proc file!...\n");
33619         return -ENOMEM;
33620     }
33621     printk(KERN_INFO "Module initialized with %d:%d device number...\n",
33622           MAJOR(g_dev), MINOR(g_dev));
33623
33624 }
33625
33626 static void __exit generic_exit(void)
33627 {
33628     cdev_del(g_cdev);
33629     unregister_chrdev_region(g_dev, 1);
33630     remove_proc_entry("generic-char-driver", NULL);
33631
33632     printk(KERN_INFO "Goodbye...\n");
33633 }
33634
33635 static ssize_t proc_read(struct file *filp, char *buf, size_t size, loff_t *off)
33636 {
33637     ssize_t left, n;
33638
33639     left = strlen(g_text) - *off;
33640     n = left < size ? left : size;
33641
33642     if (n != 0) {
33643         if (copy_to_user(buf, g_text, n) != 0)
33644             return -EFAULT;
33645         *off += n;
33646     }
33647
33648     return n;
33649 }
33650
33651 static ssize_t proc_write(struct file *filp, const char *buf, size_t size, loff_t *off)
33652 {
33653     ssize_t left, n;
33654
33655     if (filp->f_flags & O_APPEND)
33656         *off = strlen(g_text);
```

```
33657
33658     left = BUF_SIZE - *off;
33659     n = left < size ? left : size;
33660     if (n != 0) {
33661         if (copy_from_user(g_text + *off, buf, n) != 0)
33662             return -EFAULT;
33663         *off += n;
33664     }
33665
33666     return n;
33667 }
33668
33669 static loff_t  proc_llseek(struct file *filp, loff_t off, int whence)
33670 {
33671     loff_t toff;
33672
33673     switch (whence) {
33674         case SEEK_SET:
33675             toff = off;
33676             break;
33677         case SEEK_CUR:
33678             toff = filp->f_pos + off;
33679             break;
33680         case SEEK_END:
33681             toff = (loff_t)strlen(g_text) - off;
33682             break;
33683         default:
33684             return -EINVAL;
33685     }
33686
33687     if (toff > BUF_SIZE || toff < 0)
33688         return -EINVAL;
33689
33690     filp->f_pos = toff;
33691
33692     return toff;
33693 }
33694
33695
33696 module_init(generic_init);
33697 module_exit(generic_exit);
33698
33699 obj-m += $(file).o
33700
33701 all:
33702     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
33703 clean:
33704     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
33705
33706 #!/bin/bash
33707
33708 module=$1
33709 mode=666
```

```
33710
33711 /sbin/insmod ./module.ko ${@:2} || exit 1
33712 major=$(awk "\$2 == \"\$module\" {print \$1}" /proc/devices)
33713 rm -f $module
33714 mknod $module c $major 0
33715 chmod $mode $module
33716
33717 /* sample.c */
33718
33719 #include <stdio.h>
33720 #include <stdlib.h>
33721 #include <string.h>
33722 #include <fcntl.h>
33723 #include <unistd.h>
33724
33725 void exit_sys(const char *msg);
33726
33727 int main(void)
33728 {
33729     int fd;
33730     ssize_t n;
33731
33732     if ((fd = open("/proc/generic-char-driver", O_RDWR|O_APPEND)) == -1)
33733         exit_sys("open");
33734
33735     n = write(fd, "xxx", 3);
33736     if (n == -1)
33737         exit_sys("write");
33738
33739     printf("%ld bytes written...\n", (long)n);
33740
33741     close(fd);
33742
33743     return 0;
33744 }
33745
33746 void exit_sys(const char *msg)
33747 {
33748     perror(msg);
33749
33750     exit(EXIT_FAILURE);
33751 }
33752
33753 /
*-----*
-----*
33754 Biz yukarıdaki örneklerde dosyayı proc dizininin kökünde yarattık. ↗
33755 İstersek proc dizininde bir dizin yaratıp dosyalarımızı ↗
33756 o dizinin içerisinde de yaratbilirdik. Bunun için proc_mkdir fonksiyonu ↗
33757 kullanılmaktadır:
33758
33759 struct proc_dir_entry *proc_mkdir(const char *name, struct ↗
33760     proc_dir_entry *parent);
```

```
33758
33759     Fonksiyonun birinci parametresi yaratılacak dizin'in simini, ikinci      ↵
            parametresi dizinin hanfi dizin içerisinde yaratılacağını belirtir. Bu ↵
            parametre NULL
33760     geçilirse dizin proc dizininin kökünde yaratılır. Buradan aldığımız geri ↵
            dönüş değerini proc_create fonksiyonun parent parent parametresinde
33761     kullanırsak ilgili dosyamızı bu dizinde yaratmış oluruz. Örneğin:
33762
33763     struct proc_dir_entry *pdir;
33764
33765     pdir = proc_mkdir("generic-char-driver", NULL);
33766     proc_create("info", 0, pdir, &g_proc_ops);
33767
33768     Dizinlerin silinmesi yine remove_proc_entry fonksiyonuyla      ↵
            yapılabilmektedir. Dizin içerisindeki dosyaları silerken      ↵
            remove_proc_entry fonksiyonund ayine
33769     dosyanın hangi dizin içerisinde olduğu belirtilmelidir. Aslında bütün      ↵
            silme işlemleri proc_remove fonksiyonuyla da yapılabilmektedir. Bu      ↵
            fonksiyon
33770     parametre olarak proc_create ya da proc_mkdir fonksiyonun verdiği geri      ↵
            dönüş değerini alır.
33771
33772     Aşağıda generic-char-driver isimli aygit sürücü proc dizinin altında      ↵
            "generic-char-driver" isimli bir dizin yaratmıştır. Bu dizinin      ↵
            içerisinde de
33773     "info" isimli bir dosya yaratmıştır.
33774
33775 -----*/
```

```
33776
33777 /* generic-char-driver.c */
33778
33779 #include <linux/module.h>
33780 #include <linux/kernel.h>
33781 #include <linux/fs.h>
33782 #include <linux/cdev.h>
33783 #include <linux/uaccess.h>
33784 #include <linux/proc_fs.h>
33785
33786 #define BUF_SIZE          4096
33787
33788 MODULE_LICENSE("GPL");
33789 MODULE_DESCRIPTION("General Character Device Driver");
33790 MODULE_AUTHOR("Kaan Aslan");
33791
33792 static dev_t g_dev;
33793 static struct cdev *g_cdev;
33794 static struct file_operations g_file_ops = {
33795     .owner = THIS_MODULE,
33796 };
33797
33798 static ssize_t proc_read(struct file *filp, char *buf, size_t size, loff_t    ↵
            *off);
```

```
33799 static ssize_t proc_write(struct file *filp, const char *buf, size_t size,    ↵
   loff_t *off);
33800 static loff_t  proc_llseek(struct file *filp, loff_t off, int whence);
33801
33802 static struct file_operations g_proc_ops = {
33803     .owner = THIS_MODULE,
33804     .read = proc_read,
33805     .write = proc_write,
33806     .llseek = proc_llseek,
33807 };
33808
33809 static struct proc_dir_entry *g_pdir;
33810 static char g_text[BUF_SIZE] = "this is a test\n";
33811
33812 static int __init generic_init(void)
33813 {
33814     int result;
33815
33816     if ((result = alloc_chrdev_region(&g_dev, 0, 1, "generic-char-driver")) < 0) {
33817         printk(KERN_INFO "Cannot alloc char driver!...\n");
33818         return result;
33819     }
33820
33821     if ((g_cdev = cdev_alloc()) == NULL) {
33822         printk(KERN_INFO "Cannot allocate cdev!..\n");
33823         return -ENOMEM;
33824     }
33825
33826     g_cdev->owner = THIS_MODULE;
33827     g_cdev->ops = &g_file_ops;
33828
33829     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
33830         unregister_chrdev_region(g_dev, 1);
33831         printk(KERN_INFO "Cannot add character device driver!...\n");
33832         return result;
33833     }
33834
33835     if ((g_pdir = proc_mkdir("generic-char-driver", NULL)) == NULL) {
33836         unregister_chrdev_region(g_dev, 1);
33837         cdev_del(g_cdev);
33838         printk(KERN_INFO "Cannot create proc directory!...\n");
33839         return -ENOMEM;
33840     }
33841     if (proc_create("info", S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH, g_pdir,
33842                     &g_proc_ops) == NULL) {
33843         remove_proc_entry("generic-char-driver", NULL);
33844         unregister_chrdev_region(g_dev, 1);
33845         cdev_del(g_cdev);
33846         printk(KERN_INFO "Cannot create proc file!...\n");
33847         return -ENOMEM;
33848 }
```

```
33849     printk(KERN_INFO "Module initialized with %d:%d device number...\n", ↵
            MAJOR(g_dev), MINOR(g_dev));
33850
33851     return 0;
33852 }
33853
33854 static void __exit generic_exit(void)
33855 {
33856     cdev_del(g_cdev);
33857     unregister_chrdev_region(g_dev, 1);
33858     remove_proc_entry("generic-char-driver", NULL);
33859     remove_proc_entry("info", g_pdir);
33860     printk(KERN_INFO "Goodbye...\n");
33861 }
33862
33863 static ssize_t proc_read(struct file *filp, char *buf, size_t size, loff_t ↵
    *off)
33864 {
33865     ssize_t left, n;
33866
33867     left = strlen(g_text) - *off;
33868     n = left < size ? left : size;
33869
33870     if (n != 0) {
33871         if (copy_to_user(buf, g_text, n) != 0)
33872             return -EFAULT;
33873         *off += n;
33874     }
33875
33876     return n;
33877 }
33878
33879 static ssize_t proc_write(struct file *filp, const char *buf, size_t size, ↵
    loff_t *off)
33880 {
33881     ssize_t left, n;
33882
33883     if (filp->f_flags & O_APPEND)
33884         *off = strlen(g_text);
33885
33886     left = BUF_SIZE - *off;
33887     n = left < size ? left : size;
33888     if (n != 0) {
33889         if (copy_from_user(g_text + *off, buf, n) != 0)
33890             return -EFAULT;
33891         *off += n;
33892     }
33893
33894     return n;
33895 }
33896
33897 static loff_t proc_llseek(struct file *filp, loff_t off, int whence)
```

```
33899     loff_t toff;
33900
33901     switch (whence) {
33902         case SEEK_SET:
33903             toff = off;
33904             break;
33905         case SEEK_CUR:
33906             toff = filp->f_pos + off;
33907             break;
33908         case SEEK_END:
33909             toff = (loff_t)strlen(g_text) - off;
33910             break;
33911         default:
33912             return -EINVAL;
33913     }
33914
33915     if (toff > BUF_SIZE || toff < 0)
33916         return -EINVAL;
33917
33918     filp->f_pos = toff;
33919
33920     return toff;
33921 }
33922
33923 module_init(generic_init);
33924 module_exit(generic_exit);
33925
33926 obj-m += $(file).o
33927
33928 all:
33929     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
33930 clean:
33931     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
33932
33933 #!/bin/bash
33934
33935 module=$1
33936 mode=666
33937
33938 /sbin/insmod ./${module}.ko ${@:2} || exit 1
33939 major=$(awk "\$2 == \"\$module\" {print \$1}" /proc/devices)
33940 rm -f $module
33941 mknod $module c $major 0
33942 chmod $mode $module
33943
33944 /
*-----*-----*-----*
-----*
-----*
-----*
```

33945 Kesmeler (interrupts) konusunda bugün kullandığımız PC mimarisindeki
kesme mekanizmasının donanımsal tarafını ele alacağız. ↗

33946 Eskiden tek CPU'lu makineler kullanıyordu. Bugün ağırlıklı olarak birden ↗
fazla çekirdeğe sahip işlemcileri kullanıyoruz.

33947 Bu çekirdeklerin her birinin içerisinde o çekirdeğe özgü periyodik kesme ↗


```
33977
33978 #define BUF_SIZE          4096
33979
33980 MODULE_LICENSE("GPL");
33981 MODULE_DESCRIPTION("General Character Device Driver");
33982 MODULE_AUTHOR("Kaan Aslan");
33983
33984 static dev_t g_dev;
33985 static struct cdev *g_cdev;
33986 static struct file_operations g_file_ops = {
33987     .owner = THIS_MODULE,
33988 };
33989
33990 static ssize_t proc_read(struct file *filp, char *buf, size_t size, loff_t  ↪
33991             *off);
33992 static struct file_operations g_proc_ops = {
33993     .owner = THIS_MODULE,
33994     .read = proc_read,
33995 };
33996
33997 static int __init generic_init(void)
33998 {
33999     int result;
34000
34001     if ((result = alloc_chrdev_region(&g_dev, 0, 1, "generic-char-driver"))  ↪
34002         < 0) {
34003         printk(KERN_INFO "Cannot alloc char driver!...\\n");
34004         return result;
34005     }
34006
34007     if ((g_cdev = cdev_alloc()) == NULL) {
34008         printk(KERN_INFO "Cannot allocate cdev!..\\n");
34009         return -ENOMEM;
34010     }
34011
34012     g_cdev->owner = THIS_MODULE;
34013     g_cdev->ops = &g_file_ops;
34014
34015     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
34016         unregister_chrdev_region(g_dev, 1);
34017         printk(KERN_INFO "Cannot add character device driver!...\\n");
34018         return result;
34019     }
34020
34021     if (proc_create("generic-char-driver", S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH,  ↪
34022                 NULL, &g_proc_ops) == NULL) {
34023         cdev_del(g_cdev);
34024         unregister_chrdev_region(g_dev, 1);
34025         printk(KERN_INFO "Cannot create proc file!...\\n");
34026         return -ENOMEM;
34027     }
34028 }
```

```
34027     printk(KERN_INFO "Module initialized with %d:%d device number...\n",      ↵
            MAJOR(g_dev), MINOR(g_dev));
34028
34029     return 0;
34030 }
34031
34032 static void __exit generic_exit(void)
34033 {
34034     cdev_del(g_cdev);
34035     unregister_chrdev_region(g_dev, 1);
34036     remove_proc_entry("generic-char-driver", NULL);
34037     printk(KERN_INFO "Goodbye...\n");
34038 }
34039
34040 static ssize_t proc_read(struct file *filp, char *buf, size_t size, loff_t    ↵
                           *off)
34041 {
34042     static char sbuf[32];
34043     ssize_t left, n;
34044
34045     sprintf(sbuf, "%lu\n", jiffies);
34046
34047     left = strlen(sbuf) - *off;
34048     n = left < size ? left : size;
34049     if (n != 0) {
34050         if (copy_to_user(buf, sbuf + *off, n) != 0)
34051             return -EFAULT;
34052         *off += n;
34053     }
34054
34055     return n;
34056 }
34057
34058 module_init(generic_init);
34059 module_exit(generic_exit);
34060
34061 obj-m += $(file).o
34062
34063 all:
34064     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
34065 clean:
34066     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
34067
34068 #!/bin/bash
34069
34070 module=$1
34071 mode=666
34072
34073 /sbin/insmod ./${module}.ko ${@:2} || exit 1
34074 major=$(awk "\$2 == \"\$module\" {print \$1}" /proc/devices)
34075 rm -f $module
34076 mknod $module c $major 0
34077 chmod $mode $module
```

```
34078
34079  /
34080      *-----*
34081      -----*
34082      -----*
34083      -----*
34084      -----*
34085      -----*
34086      -----*
34087      -----*
34088      -----*
34089      -----*
34090      -----*
34091      -----*
34092      -----*
34093      -----*
34094      -----*
34095      -----*
34096      -----*
34097      -----*
34098      -----*
34099      -----*
34100 static ssize_t proc_read(struct file *filp, char *buf, size_t size, loff_t
34101           *off)
34102 {
34103     static char sbuf[1024];
34104     loff_t left, n;
34105     long int net_jiffies;
34106
34107     net_jiffies = (long)jiffies - (long)g_prev_jiffies;
34108
34109     sprintf(sbuf, "Son okumadan geçen zaman: %10ld\n", net_jiffies);
34110
34111     left = (loff_t)strlen(sbuf) - *off;
34112     printk(KERN_INFO "Left: %lld, size=%lu\n", left, size);
34113     n = left < size ? left : size;
34114
34115     if (n != 0) {
34116         if (copy_to_user(buf, sbuf + *off, n) != 0)
34117             return -EFAULT;
34118         *off += n;
```

```
34118     }
34119
34120     g_prev_jiffies = jiffies;
34121
34122     return (ssize_t)n;
34123 }
34124
34125 /
*-----*
-----*
34126     Aygit sürücü içerisinde bazen belli bir süre beklemek gerekebilir. ↵
      Bekleme işlemi kısa ise mesgul bir döngüde yapılabilir.
34127     Ancak beklenenek süre uzun ise mesgul döngü uygun bir yöntem olmaz. Bu ↵
      durumda daha önceden de görmüş olduğumuz wait kuyruklarında
34128     bekleme uygun olabilir.
34129
34130     Uzun beklemelerin aşağıdaki gibi yapılması uygun değildir:
34131
34132     while (time_before(jiffies, jiffies_target))
34133         schedule();
34134
34135     Buradaki temel sorun CPU'nun çok mesgul edilmesidir. Her ne kadar aygit ↵
      sürücü hemen schedule fonksiyonu ile CPU'yu bırakıyor olsa da
34136     context switch önemli bir zaman kaybına (throughput düşmesine) neden ↵
      olmaktadır. Bu nedenle uzun beklemelerin gerçekten zaman aşımılı wait ↵
      kuyruklarında
34137     yapılması gereklidir.
34138
34139     Uzun beklemeler için bir wait kuyruğu oluşturulup wait_event_timeout ya ↵
      da wait_event_interruptible_timeout fonksiyonlarıyla koşul
34140     0 yapılarak gerçekleştirilebilir. Ancak bunun için bir wait kuyruğunu oluşturulması gereklidir. Bu işlemi zaten kendi içerisinde yapan özel ↵
      fonksiyonlar vardır.
34141
34142     schedule_timeout fonksiyonu belli bir jiffy zamanı geçene kadar thread'i ↵
      önceden oluşturulmuş bir wait kuyruğunda bekletir.
34143
34144     signed long schedule_timeout(signed long timeout);
34145
34146     Fonksiyon parametre olarak beklenenek jiffy değerini alır. Eğer sinyal ↵
      dolayısıyla fonksiyon sonlanırsa kalan jiffy sayısına, eğer zaman ↵
      aşımının dolması nedeniyle
34147     fonksiyon sonlanısa 0 değerine geri döner. Fonksiyon başarısız ↵
      olmamaktadır. Fonksiyonu kullanmadan önce prosesin durum bilgisini ↵
      set_current_state isimli
34148     fonksiyonla değiştirmek gereklidir. Değiştirilecek durum
      TASK_UNINTERRUPTIBLE ya da TASK_INTERRUPTIBLE olabilir. Bu işlem ↵
      yapılmazsa bekleme gerçekleşmemektedir.
34149     Örneğin:
34150
34151     set_current_state(TASK_INTERRUPTIBLE);
34152     schedule_timeout(jiffies + 5 * HZ);
34153
```

34154 Uzun beklemeyi kendi içerisinde yapan (yani `schedule_timeout`)
fonksiyonunu kullanarak yapan üç yardımcı fonksiyon da vardır:

34155

34156 `void msleep(unsigned int msecs);`

34157 `unsigned long msleep_interruptible(unsigned int msecs);`

34158 `void ssleep(unsigned int secs);`

34159 `void ssleep_interruptible(unsigned int secs);`

34160

34161 Aşağıdaki örnekte `/proc/generic` dosyasından okuma yapılmaya
çalışıldığında 5 saniye kernel modda thread bekletilecektir.

34162 Burada yalnızca `proc_read` fonksiyonu verilmiştir. Yukarıdaki programı bu
fonksiyonu değiştirerek kullanabilirsiniz.

34163

34164 -----*/

34165

34166 `static ssize_t proc_read(struct file *filp, char *buf, size_t size, loff_t *off)`

34167 {

34168 set_current_state(TASK_INTERRUPTIBLE);

34169 schedule_timeout(5 * HZ);

34170

34171 /* `ssleep_interruptible(5)` */

34172

34173 return 0;

34174 }

34175

34176 /-----*/

34177 milisaniye, mikrosaniye ve nano saniye değerlerini jiffies değerine
dönüştüren üç fonksiyon bulunmaktadır:

34178

34179 `unsigned long msecs_to_jiffies(const unsigned int m);`

34180 `unsigned long usecs_to_jiffies(const unsigned int m);`

34181 `unsigned long usecs_to_jiffies(const unsigned int m);`

34182

34183 Bu işlemin tersini yapan da üç fonksiyon vardır:

34184

34185 `unsigned int jiffies_to_msecs(const unsigned long j);`

34186 `unsigned int jiffies_to_usecs(const unsigned long j);`

34187 `unsigned int jiffies_to_nsecs(const unsigned long j);`

34188

34189 Bu fonksiyonlar o andaki aktif HZ değerini dikkate almaktadır.

34190

34191 Ayrıca jiffies değerini saniye ve nano saniye biçiminde ayırip bize
struct `timespec64` biçiminde bir yapı nesnesi olarak veren

34192 `jiffies_to_timespec64` isimli bir fonksiyon da vardır. Bunun tersi
`timespec64_to_jiffies` fonksiyonuyla yapılmaktadır.

34193

34194

34195 Örneğin aygıt sürücü içerisinde 5 saniye bekleme işlemi aşağıdaki gibi
de yapılabilirdi

```
34196 -----*/  
34197  
34198 static ssize_t proc_read(struct file *filp, char *buf, size_t size, loff_t *off)  
34199 {  
34200     set_current_state(TASK_INTERRUPTIBLE);  
34201     schedule_timeout(msecs_to_jiffies(5000));  
34202  
34203     return 0;  
34204 }  
34205  
34206 /-----*/  
34207     Aygit sürücü içerisinde kısa beklemeler gerebilmektedir. Çünkü bazı  
donanım aygıtlarının programlanabilmesi için bazı beklemelere  
gereksinim  
34208     duyulabilmektedir. Kısa beklemeler megul döngü yoluyla yani hiç sleep  
yapılmadan sağlanmaktadır. Ayrıca kısa bekleme yapan fonksiyonlar  
atomiktir.  
34209     Atomiklikten kastedilen şey preemption işleminin kapatılmasıdır. Yani  
kısa bekleme yapan fonksiyonlar context switch işlemini o işlemci için  
kapatırlar.  
34210     Bu sırada thread'ler arası geçiş söz konusu olmamaktadır. Ancak donanım  
kesmeleri bu süre içerisinde oluşabilmektedir.  
34211  
34212     Kısa süreli döngü içerisinde bekleme yapan fonksiyonlar şunlardır:  
34213  
34214     void ndelay(unsigned int nsecs);  
34215     void udelay(unsigned int usecs);  
34216     void mdelay(unsigned int msecs);  
34217  
34218 -----*/  
34219  
34220 /-----*/  
34221     Linux çekirdeklerine belli versionyondan sonra bir timer mekanizması da  
eklenmiştir. Bu sayede aygit sürücü programcısı belli bir zaman sonra  
belirlediği bir fonksiyonun çağrılmasını saplayabilmektedir. Bu  
mekanizmaya "kernel timer" mekanizması denilmektedir. Maalesf kernel  
timer mekanizması da  
34223     birkaç kere arayüz olarak değiştirilmiştir. Belli zaman sonra çağrılacak  
fonksiyonun bir proses adına çalışmadığına dikkat etmek gereklidir. Yani  
belirlenen fonksiyon  
34224     çapıldığında biz current değişkeni ile o andaki prosese erişemeyiz. O  
anda çalışan prosesin user alanına kopyalamalar yapamayız. Çünkü bu  
fonksiyon timer tick kesmeleri  
34225     tarafından çağrılmaktadır. Son Linux çekirdeklerindeki kernel timer  
kullanımı şöyledir:  
34226
```

```
34227    1) struct timer_list türünden bir yapı nesnesi statik düzeyde tanımlanır →  
        ve bu yapı nesnesine ilkdeğeri verilir. İlkdeğer verme işlemi →  
        DEFINE_TIMER makrosuyla  
34228    yapılabılır.  
34229  
34230    #define DEFINE_TIMER(_name, _function)  
34231  
34232    ya da timer_setup fonksiyonuyla yapılabilmektedir:  
34233  
34234    #define timer_setup(timer, callback, flags)  
34235  
34236    Makronun birinci parametresi timer nesnesinin adresini almaktadır. →  
        İkinci parametresi çağrılacak fonksiyonun adresidir. flags parametresi →  
        0 geçilebilir.  
34237  
34238    2) Tanımlanan struct timer_list nesnesi add_timer fonksiyonu ile bir →  
        bağlı listeye yerleştirilir.  
34239  
34240    void add_timer(struct timer_list *timer);  
34241  
34242    3) Daha sonra ne zaman fonksiyonun çağrılacağını anlatmak için modtimer →  
        fonksiyonu kullanılır.  
34243  
34244    int mod_timer(struct timer_list *timer, unsigned long expires);  
34245  
34246    Buradaki expiry parametresi jiffy türündendir. Bu parametre hedef jiffy →  
        değerini içermeliir. (Yani jiffies + gecikme jiffy değeri)  
34247  
34248    4) Timer nesnesinin silinmesi için del_timer fonksiyonu →  
        kullanılmaktadır:  
34249  
34250    int del_timer(struct timer_list * timer);  
34251  
34252    Normal olarak belirlenen fonksiyon yalnızca 1 kez çağrılmaktadır. Ancak →  
        bu fonksiyonun içerisinde yeniden mod_timer ile yeniden çağrıma →  
        sağlanabilmektedir.  
34253  
34254    Çağrılması istenen fonksiyonun parametrik yapısı şöyle olmalıdır:  
34255  
34256    void call_back_func(struct timer_list *tl);  
34257  
34258    Bu fonksiyon add_timer sırasında verdığımız struct timer_list adresi →  
        parametre yapılarda çağrılmaktadır.  
34259  
34260    Aşağıda kernel timer'larının kullanımına ilişkin bir örnek →  
        görülmektedir. Bu örnekte kernel timer /proc/generic-char-driver →  
        dosyasından  
34261    okuma yapıldığında yaratılıp periyodik bir biçimde çağrılmaktadır. 5 →  
        çağrımdan sonra yaratılan timer yok edilmiştir.  
34262  
34263    -----*/  
34264
```

```
34265 /* generic-char-driver.c */
34266
34267 #include <linux/module.h>
34268 #include <linux/kernel.h>
34269 #include <linux/fs.h>
34270 #include <linux/cdev.h>
34271 #include <linux/uaccess.h>
34272 #include <linux/proc_fs.h>
34273 #include <linux/sched.h>
34274
34275 #define BUF_SIZE          4096
34276
34277 MODULE_LICENSE("GPL");
34278 MODULE_DESCRIPTION("General Character Device Driver");
34279 MODULE_AUTHOR("Kaan Aslan");
34280
34281 static dev_t g_dev;
34282 static struct cdev *g_cdev;
34283 static struct file_operations g_file_ops = {
34284     .owner = THIS_MODULE,
34285 };
34286 static long int g_prev_jiffies;
34287
34288 static ssize_t proc_read(struct file *filp, char *buf, size_t size, loff_t  ↪
34289 *off);
34290
34291 static struct file_operations g_proc_ops = {
34292     .owner = THIS_MODULE,
34293     .read = proc_read,
34294 };
34295
34296 static int __init generic_init(void)
34297 {
34298     int result;
34299
34300     if ((result = alloc_chrdev_region(&g_dev, 0, 1, "generic-char-driver"))  ↪
34301         < 0) {
34302         printk(KERN_INFO "Cannot alloc char driver!...\n");
34303         return result;
34304     }
34305
34306     if ((g_cdev = cdev_alloc()) == NULL) {
34307         printk(KERN_INFO "Cannot allocate cdev!..\n");
34308         return -ENOMEM;
34309     }
34310
34311     g_cdev->owner = THIS_MODULE;
34312     g_cdev->ops = &g_file_ops;
34313
34314     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
34315         unregister_chrdev_region(g_dev, 1);
34316         printk(KERN_INFO "Cannot add character device driver!...\n");
34317         return result;
```

```
34316     }
34317
34318     if (proc_create("generic-char-driver", S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH, ↵
34319         NULL, &g_proc_ops) == NULL) {
34320         cdev_del(g_cdev);
34321         unregister_chrdev_region(g_dev, 1);
34322         printk(KERN_INFO "Cannot create proc file!...\n");
34323         return -ENOMEM;
34324     }
34325     g_prev_jiffies = jiffies;
34326
34327     printk(KERN_INFO "Module initialized with %d:%d device number...\\n", ↵
34328           MAJOR(g_dev), MINOR(g_dev));
34329
34330     return 0;
34331 }
34332 static void __exit generic_exit(void)
34333 {
34334     cdev_del(g_cdev);
34335     unregister_chrdev_region(g_dev, 1);
34336     remove_proc_entry("generic-char-driver", NULL);
34337     printk(KERN_INFO "Goodbye...\\n");
34338 }
34339
34340 static void timer_callback_func(struct timer_list *tl)
34341 {
34342     static int count = 0;
34343
34344     printk(KERN_INFO "timer callback...\\n");
34345
34346     mod_timer(tl, jiffies + msecs_to_jiffies(5000));
34347
34348     if (count == 5) {
34349         del_timer(tl);
34350         count = 0;
34351     }
34352     else
34353         ++count;
34354 }
34355
34356 static ssize_t proc_read(struct file *filp, char *buf, size_t size, loff_t ↵
34357 *off)
34358 {
34359     static struct timer_list tl;
34360
34361     timer_setup(&tl, timer_callback_func, 0);
34362     add_timer(&tl);
34363     mod_timer(&tl, jiffies + msecs_to_jiffies(5000));
34364
34365     return 0;
34366 }
```

```
34366
34367 module_init(generic_init);
34368 module_exit(generic_exit);
34369
34370 obj-m += $(file).o
34371
34372 all:
34373     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
34374 clean:
34375     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
34376
34377 #!/bin/bash
34378
34379 module=$1
34380 mode=666
34381
34382 /sbin/insmod ./${module}.ko ${@:2} || exit 1
34383 major=$(awk "\$2 == \"\$module\" {print \$1}" /proc/devices)
34384 rm -f $module
34385 mknod $module c $major 0
34386 chmod $mode $module
34387
34388 /
*-----→
-----→
34389 Önceki konularda da UNIX/Linux sistemlerinde kernel modda çalışan →
    işletim sistemine ait thread'ler olduğundan bahsetmiştik. Bu →
    thread'ler
34390 run kuyruğunda bulunan uykuya dalabilen işletim sisteminin programı gibi →
    çalışan thread'leridir. Çeşitli işlemlerdne sorumludurlar. Linux →
34391 işletim sisteminde bunların da isimleri genellikle "user mode deamon"lar →
    gibi sonu 'd' ile bitmektedir ve yine bunlar →
34392 isimleri genellikle 'k' harfi ile başlarlar. Örneğin "kupdate", →
    "kswapd", "keventd" gibi.
34393
34394 İşte aygit sürücüler de isterlerse arka planda kernel modda bir proses →
    gibi çalışan thread'ler yaratırabilirler. Ancak bu thread'ler
34395 bir proses ile ilişkisiz çalıştırılmaktadır. Bu nedenle bunlar →
    içerisinde current değişkeni kullanılamaz ve copy_to_user ya da →
    copy_from_user
34396 gibi fonksiyonlar kullanılmaz.
34397
34398 Aygit sürücüdeki kodlarımız genellikle bir olay olduğunda (örneğin kesme →
    gibi), ya da user moddan çağrıldığında (read, write, ioctl) gibi →
34399 çalıştırılmaktadır. Ancak kernel thread'ler aygit sürücüye sanki bir →
    programmış gibi kernel modda sürekli çalışma imkanı verirler.
34400
34401 Kernel thread'ler şöyle kullanılırlar:
34402
34403 1) Önce kernel thread aygit sürücü içerisinde yaratılır. Yaratılma →
    modülün init işleminde yapılabileceği gibi aygit sürücü ilk kez →
    açıldığında
34404 open fonksiyonunda ya da belli bir süre sonra belli bir fonksiyonda da →
```

yapılabilmekteydi. Kernel thread'ler kthread_create fonksiyonıyla
yaratılabilirler:

```
34405
34406     struct task_struct *kthread_create (int (*threadfn)(void *data), void    ↵
34407             *data, const char  *namefmt);
34408
34409     Fonksiyon başarı durumunda yaratılan thread'in task_struct adresine,    ↵
34410         başarısızlık durumunda negatif error değerine geri dönmektedir.
34411     Adrese geri dönen diğer kernel fonksiyonlarında olduğu gibi başarısızlık ↵
34412         IS_ERR makrosuyla test edilmelidir. Eğer fonksiyon başarısız olmuşsa
34413         error kodu PTR_ERR makrosuyle elde edilmelidir. Örneğin:
34414
34415         struct task_struct *ts;
34416
34417         ts = kthread_create(...);
34418         if (IS_ERR(ts)) {
34419             printk(KERN_ERROR "cannot create kernel thread!..")
34420             return PTR_ERR(ts);
34421
34422     Kernel threda bu fonksiyonla yaratıldıktan sonra hemen çalışmaz. Onu    ↵
34423         çalıştırmak için wake_process fonksiyonun çağrılması gereklidir:
34424
34425         int wake_up_process(struct task_struct *tsk);
34426
34427         Fonksiyon başarı durumunda 0 değerine başarısızlık durumunda negatif    ↵
34428             error koduna geri dönmektedir.
34429
34430         Aslında yukarıdaki işlemi tek hamlede yapan kthread_run isimli bir    ↵
34431             fonksiyon da vardır:
34432
34433         struct task_struct *kthread_run(int (*threadfn)(void *data), void *data, ↵
34434             const char  *namefmt);
34435
34436         2) Kernel thread kthread_stop fonksiyonuyla herhangi bir zaman ya da    ↵
34437             aygit sürücü silinirken yok edilebilir:
34438
34439         int kthread_stop(struct task_struct *ts);
34440
34441         Fonksiyon thread sonlanana kadar blokeye yol açar. Fonksiyon thread    ↵
34442             fonksiyonunun exit koduyla (yani thread fonksiyonunun geri dönüş    ↵
34443                 değeri ile)
34444         geri döner. Genellikle programcılar başarı için 0 değerini    ↵
34445             kullanmaktadır. Burada önemli nokta kthread_stop fonksiyonunun kernel    ↵
34446                 thread'i zorla
34447         sonlandırmadığıdır. Kerle thread'in sonlanması zorla yapılmaz.    ↵
34448             kthread_stop fonksiyonu bir bayrağı set eder. Kernel thread de bu    ↵
34449                 bayrak set edilmiş mi
34450         diye bir döngü içerisinde bakar. Eğer baurak set edilmişse kendini    ↵
34451             sonlandırır. Kernel thread'in bu bayrağa bakması kthread_should_stop
34452             fonksiyonuyla yapılmaktadır.
34453
34454         bool kthread_should_stop(void);
```

```
34441
34442      Fonksiyon eğer flag set edilmişse sıfır dışı bir değere ser edilmediyse →
34443          0 değerine geri dönmektedir. Tipik olarak kernel threda fonksiyonu
34444      aşağıdaki gibi bir döngüde yaşamını geçirir:
34445
34446      while (!kthread_should_stop()) {
34447          ...
34448      }
34449
34450      Aşağıdaki örnekte modül initialize edilirken kernel thread yaratılmış, →
34451          modül yok edilirken kthread_stop ile
34452      kernel thread'in sonlanması beklenmiştir. kernel thread içerisinde →
34453          msleep fonksiyonu ile 1 saniyelik beklemeler yapılmıştır.
34454  -----
34455
34456
34457
34458
34459
34460
34461
34462
34463
34464
34465
34466
34467
34468
34469
34470
34471
34472
34473
34474
34475
34476
34477
34478
34479
34480
34481
34482
34483
34484
34485
34486
34487
34488
```

```
34489         printk(KERN_INFO "Cannot allocate cdev!..\n");
34490         return -ENOMEM;
34491     }
34492
34493     g_cdev->owner = THIS_MODULE;
34494     g_cdev->ops = &g_file_ops;
34495
34496     if ((result = cdev_add(g_cdev, g_dev, 1)) != 0) {
34497         unregister_chrdev_region(g_dev, 1);
34498         printk(KERN_INFO "Cannot add character device driver!...\n");
34499         return result;
34500     }
34501
34502     g_ts = kthread_run(generic_kernel_thread, NULL, "generic-kernel-      ↵
34503         thread");
34504     if (IS_ERR(g_ts)) {
34505         printk(KERN_INFO "cannot create kernel thread!..");
34506         return PTR_ERR(g_ts);
34507     }
34508     printk(KERN_INFO "Module initialized with %d:%d device number...\\n",      ↵
34509         MAJOR(g_dev), MINOR(g_dev));
34510     return 0;
34511 }
34512
34513 static void __exit generic_exit(void)
34514 {
34515     kthread_stop(g_ts);
34516     cdev_del(g_cdev);
34517     unregister_chrdev_region(g_dev, 1);
34518     remove_proc_entry("generic-char-driver", NULL);
34519     printk(KERN_INFO "Goodbye...\\n");
34520 }
34521
34522 static int generic_kernel_thread(void *data)
34523 {
34524     static int count = 0;
34525
34526     while (!kthread_should_stop()) {
34527         printk(KERN_INFO "Kernel thread running: %d\\n", count);
34528         ++count;
34529         msleep(1000);
34530     }
34531
34532     return 0;
34533 }
34534
34535 module_init(generic_init);
34536 module_exit(generic_exit);
34537
34538 obj-m += $(file).o
34539
```

```
34540 all:
34541     make -C /lib/modules/$(shell uname -r)/build M=${PWD} modules
34542 clean:
34543     make -C /lib/modules/$(shell uname -r)/build M=${PWD} clean
34544
34545 #!/bin/bash
34546
34547 module=$1
34548 mode=666
34549
34550 /sbin/insmod ./${module}.ko ${@:2} || exit 1
34551 major=$(awk "\$2 == \"\$module\" {print \$1}" /proc/devices)
34552 rm -f $module
34553 mknod $module c $major 0
34554 chmod $mode $module
34555
34556 /
*-----*-----*-----*
```

34557 Kesme işlemcinin çalıştırılmakta olduğu koda ara vererek başka bir kodu çalıştırması sürecidir. Kesme oluştuğunda çalıştırılan koda 34558 "kesme kodu (interrupt handler)" denilmektedir. Kesmeler üç kısma ayrılabılır. Kesme denildiğinde akla "donanım kesmeleri (hardware interrupts)" 34559 gelir. Donanım kesmeleri işlemcilerin INT ucunun elektriksel olarak dış bir aygıt tarafından uyarılmasıyla oluşturulur. Böylece aygıtlar birtakım 34560 olayları bildirmek için kesme oluşturabilmektedir. "Yazılım kesmeleri (software interrupts)" her işlemcide yoktur. Bunlar program kodu içerisindeinde 34561 özel kesme oluşturan makine kodları ile (örneğin Intel'deki INT makine komutu gibi) oluşturulurlar. "İçsel kesmeler (internal interrupts)" ise 34562 işlemci tarafından bazı müşkülerle karşılaşıldığında oluşturulmaktadır. Örneğin işlemci sanal adresi fiziksel adrese dönüştürürken syfa tablosunda 34563 sanal sayfaya karşı bir fiziksel sayfanın eşleştirilmediğini gördüğünde "page fault" denilen içsel kesmeyi oluşturmaktadır. Ancak kesme denildiğinde 34564 default olarak donanım kesmeleri akla gelmektedir.

34565

34566 Donanım kesmesi oluşturan elektronik birimlerin hepsi doğrudan CPU'nın INT ucuna bağlanmamaktadır. Genellikle bu işe aracılık eden daha akıllı 34567 bir işlemci kullanılır. Bu işlemcilere "kesme denetleyecileri (interrupt controller)" denilmektedir. Bazı mimarilerde kesme denetleyicisi işlemciin 34568 içerisinde taşınmıştır. Bazı mimarilerde dışında ayrı bir entegre devre olarak bulunur. Bazı mimarilerde her iki durum da söz konusudur. Kesme 34569 şu faydaları vardır:

34570

34571 1) Birden fazla donanım biriminin aynı anda kesme oluşturulması durumunda

bunları sıraya dizebilme
34572 2) Birden fazla donanım biriminin aynı anda kesme oluşturulması durumunda →
bunlara öncelik verebilme
34573 3) Belli birimlerden gelen kesme isteklerini disable edebilme, ya da →
tümden tüm uçlardan gelen istekleri disable edebilme
34574 4) Multicore sistemlerde kesmenin belli bir core'da →
çalıştırılabilmesini sağlama
34575
34576 Bugün kullandığımız PC'lerde (laptop ve notebook'lar da dahil olmak →
üzere) eskiden kesme denetleyicisi olarak bir tane 8259 (PIC) devresi →
vardı.
34577 Bunun 8 girişi bulunuyordu. Yani bu denetleyicinin uçları 8 ayrı donanım →
birimine bağlanabiliyordu. Sonra bunun sayısı ikiye çıkartıldı.
34578 Ancak bir uç ikisini bağlamak için kullanıldığından toplam 15 uç elde →
edildi. Bu uçlara IRQ (Interrupt Request) denildi ve bunlara numaralar →
verildi (IRQ-0, IRQ-1, ...) Ancak bu 8259 denetleyicisinin zamanla →
yetersizliği görüldü. Özellikle birden fazla çekirdek ile çalışmalar →
başlayınca
34580 bu denetleyici yetersiz kalmıştır. Çünkü bu denetleyici tek bir →
çekirdeğe bağlanabilmektedir. İşte daha sonraları 8259 yerine ismine →
34581 IOAPIC (82801) denilen daha gelişmiş bir kesme denetleyicisi →
kullanılmaya başlanmıştır. Bu yeni kesme denetleyicisinin 24 ucu →
vardır. Aynı zamanda bu
34582 kesme denetleyicisi birden fazla çekirdeğin bulunduğu ortamda →
çekirdeklerin istenen bir tanesinde kesme oluşturabilmektedir. Kesme →
denetleyicilerinin
34583 bazı uçlarına zaten üretimden bazı birimler bağlanmıştır. Ancak bunların →
bazıları boşadır. Genişleme yuvalarına takılan kartlar bu boştaki →
kesme uçlarını
34584 kullanabilmektedir. Bugün PC'lerde kullanılan modern genişleme →
yuvalarına PCI ve PCI-X denilmektedir.
34585
34586 Bugün Pentium ve esdeger AMD işlemcilerinin içeisinde de ismine "Local →
APIC" denilen bir kesme denetleyicisi vardır. Bu local APIC iki uca →
sahiptir.
34587 Aynı zamanda bir timer devresi de bulundurmaktadır. Bu local APIC →
icерисинде timer devresi o işlemcide jiffy oluşturulmasında ve →
context switch
34588 yapılmasında kullanılmaktadır. Local APIC'in en önemli özelliklerinden →
birisi "data bus" yoluyla kesme alabilmesidir. Bu özellik sayesinde →
hiç işlemcinin
34589 INT uyarılmadan çok fazla sayıda kesme sanki belleğe bir değer →
yazıyorum gizbi oluşturulabilmektedir. Gerçekten de bugün PCI →
slotlara takılan bazı
34590 kartlar kesmeleri bu biçimde oluşturmaktadır. Bu tekniğe "Message →
Signaled Interrupt (MSI)" denilmektedir.
34591
34592 0 halde bugünkü durum şöyledir:
34593
34594 - Bazı donanım birimleri built-in biçimde IOAPIC'in uçlarına bağlı →
durumdadır. Bu u.lar eskiye uyumu korumak için 8259'un uçlarıyla →
aynıdır.
- Bazı PCI kartlar slot üzerindeki 4 IRQ hattından (INTA, INTB, INTC, →

INTD) birini kullanarak kesme oluşturmaktadır. Bu hatlar IOAPIC'in bazı uçlarına bağlıdır.

- Bazı PCI kartlar ise doğurduan MSI biçiminde IOAPIC'i pass geçerek memory işlemleriyle ilgili çekirdekte kesme oluşturabilmektedir.

Bir aygit sürücü programcısı mademki birtakım kartlar için onu işler hale getiren temel yazılımları da yazma iddiasındadır. O halde o kartın kullanacağı kesme kodu yazabilмелidir. Tabii işletim sisteminin aygit sürücü mimarisinde bu işlemler de özel kernel fonksiyonlarıyla yapılır. Yani kesme kodu yazmanın belli bir kuralı vardır. Eskiden ve hala bazı kesmeler birden fazla donanım tarafından ortak kullanılabilir.

Örneğin kesme denetleyicisinin belli bir ucunu birden fazla donanım uyarabilir. Bu durumda işletim sisteminin bir kesme oluştuğunda birden fazla kesme kodunu kuyruğa alıp çalıştırabilmesi beklenir. Böylece aygit sürücüyü yazan programcı kesmenin kendi kartından gelip gelmediğini anlamak zorundadır. Eğer kesme kendi kartından gelmemişse programcı kesme kodundan hemen çıkar, kesme kendi kartından gelmişse onu işler.

Pekiyi çok çekirdekli bilgisayar sistemlerinde oluşan bir kesme hangi çekirdek tarafından işlenecektir? Bugün kullanılan IOAPIC devreleri bu bakımından

şu özelliklere sahiptir:

- 34609 1) Kesme IOAPIC tarafından donanım birimi tarafından istenilen bir çekirdekte oluşturulabilir.
- 34610 2) IOAPIC en az yüklü çekirdeğe karar vererek kesmenin orada oluşturulmasını sağlayabilir.
- 34611 3) IOAPIC döngüsel bir biçimde çekirdeklerle kesme oluşturabilmektedir.

IOAPIC'in en az yüklü işlemciyi bilmesi mümkün değildir. Onu işletim sistemi bilebilir. İşte işlemcilerin Local APIC'leri içerisinde özel bazı yazmaçlar vardır. Aslında IOAPIC bu yazmaçtaki değerlere bakıp en düşüğünü seçmektedir. Bu değerleri de işletim sistemi set eder. İşletim sisteminin yaptığı bu faaliyete "kesme dengeleme (IRQ balancing)" denilmektedir. Linux sistemlerinde bir süredir kesme dengeleyicisi bir kernel thread (irqbalance) olarak çalıştırılmaktadır. Böylece Linux sistemlerinde aslında donanım kesmeleri her defasında farklı çekirdeklerde çalıştırışıyor olabilir.

Pek çok CPU ailesinde donanım kesmelerinin teorik maksimum bir limiti vardır. Örneğin Intel mimarisinde toplam kesme sayısı 256'yı geçememektedir.

34619 Bu mimaride her kesmenin bir numarası vardır. IRQ numarası ile kesme numarasının bir ilgisi yoktur. Biz örneğin PIC ya da IOAPIC'i programlayarak belli bir kesmenin

34620 belli bir IRQ için belli numaralı bir kesmenin oluşmasını →
sağlayabiliriz. Örneğin timer (IRQ-0) için 8 numaralı kesmenin →
çalışmasını sağlayabiliriz.

34621 Pekiyi bir IRQ oluşturulduğunda çekirdek kaç numaralı kesme kodunun →
çalıştırılacağını nereden anlamaktadır? İşte PIC ya da IOAPIC CPU'nun →
INT ucunu uyararak

34622 kesme oluşturuken Data Bus'in ilk 8 ucundan kesme numarasını da CPU'ya →
bildirmektedir.

34623

34624 -----*/

34625

34626 / -----*

34627 Bir aygit sürücüsü bir kesme oluştuguunda kendi fonksiyonunun →
çağrılmasını istiyorsa onu request_irq isimli kernel fonksiyonuyla →
register ettirmelidir.

34628 static int request_irq(unsigned int irq, irq_handler_t handler, unsigned →
long flags, const char *name, void *dev_id);

34629

34630 Fonksiyonun birinci parametresi IRQ numarasıdır. İkinci parametresi IRQ →
oluştuguunda çağrılacak fonksiyonu belirtmektedir. Bu fonksiyonun →
geri dönüş değeri irqreturn_t üründen (bu bir tamsayı türündür) →
parametreleri de sırasıyla int ve void * türündendir. Örneğin:

34631

34632 irqreturn_t my_irq_handler(int irq, void *dev_id)

34633 {

34634

34635 }

34636

34637 Fonksiyonun üçüncü parametresi bazı bayraklardan oluşur. Bu bayrak 0 →
geçilebilir ya da örneğin IRQF_SHARED geçilebilir. Diğer seçenekler →
için dokümanlara başvurabilirsiniz.

34638 IRQF_SHARED aynı kesmenin birden fazla aygit sürücü tarafından →
kullanılabileceği anlamına gelmektedir. (Tabii biz ilk register →
ettiren değilsek daha önce register ettirenlerin

34639 bu bayrağı kullanmış olması gereklidir. Aksi halde biz de bu bayrağı →
kullanamayız.) Fonksiyonun dördüncü parametresi /proc/interrupts →
dosyasına görüntülenecek ismi belirtir. Son parametre

34640 sistem genelinde tek olan bir nesnenin adresi olarak girilmelidir. Aygit →
sürücü programcılar bu parametreye tipik olarak aygit yapısını ya da →
çağrılacak foksiyonu girerler. irq_request fonksiyonu

34641 başarı durumunda 0 değerine başarısızlık durumunda negatif hata değerine →
geri dönmektedir.

34642

34643

34644 Bir kesme kodu request_irq fonksiyonuyla register ettirilmişse bunun →
geri alınması free_irq fonksiyonuyla yapılmaktadır:

34645

34646

34647 const void *free_irq(unsigned int irq, void *dev_id);

34651
34652 Fonksiyonun birinci parametresi silinecekirq numarasını, ikinci →
parametresi irq_reuest fonksiyonuna girilen son parametreyi belirtir. →
Fonksiyon başarı durumunda aygit ismine,
34653 başarısızlık durumunda NULL adrese geri dönmektedir.

34654 -----*/

34655
34656 / →

34657 Kesme mekanizmasının tipik örneklerinden biri klavye kullanımıdır. PC →
klavyesinde bir tuşa basıldığında klavye içerisindeki
34658 işlemci (keyboard encoder - Intel 8048) basılan ya da çekilen tuşun →
klavyedeki sıra numarasını (buna "scan code" denilmektedir) dış →
dünyaya
34659 seri bir biçimde kodlamaktadır. Bu bilgi bilgisayardaki klavye →
denetleyicisine (Intel 8042) gelir. Klavye denetleyicisi PIC ya da →
APIC'in 1 numaralı
34660 ucuna bağlıdır ve bu uçtan IRQ1 kesmesini oluşturur. Ama basılan ve →
çekilen tuşun scan kodunu kendi içerisinde saklar. Böylece tuşa →
basıldığında ve parmek tuştan
34661 çekildiğinde işletim sistemini IRQ1 kesme kodu çalışmaktadır. Bu kodda →
klavye denetleyicisinden basılan ya da çekilen kodun scan kodu →
alınarak bir yazılımsal kuyruk sistemine
34662 yerleştirilmektedir. stdin terminal sürücüsü aslında bu kuyuktan →
okumayı yapmaktadır.

34663
34664 Klavyedeki tuşların üzerinde yazan harflerin hiçbir önemi yoktur. Yani →
İngilizce klavye ile Türkçe klavye aynı tuşlar için aynı scan kodu →
gondermektedir.
34665 Basılan tuşun hangi tuş olduğu aslında dil ayarlarına bakılarak işletim →
sistemi tarafından anlamlandırılmaktadır. Klavye ile bilgisayar →
arasındaki iletişim
34666 tek yönlü değil çift yönlüdür. Yani Klavye işlemcisi de isterse klavye →
içerisindeki işlemciye komut gönderebilmektedir. Aslında klavye →
üzerindeki ışıkların yakılması da
34667 klavyenin içerisinde tuşa basılınca yapılmamaktadır. Işıklı tuşlara →
basıldığında gönderilen scan kod klavye denetleyicisi tarafından →
alınır ve kesme kodu yeniden klavye işlemcisine
34668 ışığı yak komutunu gönderir. Örneğin bilgisayardaki işletim sistemi →
çökmuşse bu ışıklar yanmayacaktır.

34669 -----*/

34670
34671 / →

34672
34673 -----*/

34674
34675 / →

```
*-
34676
34677   */
34678
34679   /
34680
34681   */
34682
34683   /
34684
34685   */
34686
34687   /
34688
34689   */
34690
34691   /
34692
34693   */
34694
34695   /
34696
34697   */
34698
34699   /
34700
34701   */
34702
34703   /
34704
34705   */
```

```
-----*/  
34706 /  
*-----  
-----*/  
34708  
34709 -----*/  
-----*/  
34710  
34711 /  
*-----  
-----*/  
34712  
34713 -----*/  
-----*/  
34714  
34715 /  
*-----  
-----*/  
34716  
34717 -----*/  
-----*/  
34718  
34719 /  
*-----  
-----*/  
34720  
34721 -----*/  
-----*/  
34722  
34723 /  
*-----  
-----*/  
34724  
34725 -----*/  
-----*/  
34726  
34727 /  
*-----  
-----*/  
34728  
34729 -----*/  
-----*/  
34730  
34731 /  
*-----  
-----*/  
34732  
34733 -----*/  
-----*/  
34734  
34735 /  
*-----
```

```
34736
34737     */
34738
34739     /
34740
34741     */
34742
34743     /
34744
34745     */
34746
34747     /
34748
34749     */
34750
34751     /
34752
34753     */
34754
34755     /
34756
34757     */
34758
34759     /
34760
34761     */
34762
34763     /
34764
34765     */
```

```
34766
34767 /
34768
34769 -----
34770
34771 /
34772
34773 -----
34774
34775 /
34776
34777 -----
34778
34779 /
34780
34781 -----
34782
34783 /
34784
34785 -----
34786
34787 /
34788
34789 -----
34790
34791 /
34792
34793 -----
34794
34795 /
```

```
34796
34797 -----*/>
34798 -----*/
34799 /-----*/>
34800 -----*/
34801 /-----*/>
34802 -----*/
34803 /-----*/>
34804 -----*/
34805 /-----*/>
34806 -----*/
34807 /-----*/>
34808 -----*/
34809 /-----*/>
34810 -----*/
34811 /-----*/>
34812 -----*/
34813 /-----*/>
34814 -----*/
34815 /-----*/>
34816 -----*/
34817 /-----*/>
34818 -----*/
34819 /-----*/>
34820 -----*/
34821 /-----*/>
34822 -----*/
34823 /-----*/>
34824 -----*/
34825 /-----*/>
```

```
34827 / *
-----*
34828 -----
34829 ----- */
34830 /
-----*
34831 / *
-----*
34832 -----
34833 ----- */
34834 /
-----*
34835 / *
-----*
34836 -----
34837 ----- */
34838 /
-----*
34839 / *
-----*
34840 -----
34841 ----- */
34842 /
-----*
34843 / *
-----*
34844 -----
34845 ----- */
34846 /
-----*
34847 / *
-----*
34848 -----
34849 ----- */
34850 /
-----*
34851 / *
-----*
34852 -----
34853 ----- */
34854 /
-----*
34855 / *
-----*
34856 -----
```

```
34857 -----*/  
34858 /  
34859 *-----  
34860  
34861 -----*/  
34862 /  
34863 *-----  
34864  
34865 -----*/  
34866 /  
34867 *-----  
34868  
34869 -----*/  
34870 /  
34871 *-----  
34872  
34873 -----*/  
34874 /  
34875 *-----  
34876  
34877 -----*/  
34878 /  
34879 *-----  
34880  
34881 -----*/  
34882 /  
34883 *-----  
34884  
34885 -----*/  
34886 /  
34887 */
```

```
*-
34888
34889   *-
34890   -/
34891   /
34892   *-
34893   -/
34894   -/
34895   /
34896   *-
34897   -/
34898   -/
34899   /
34900   *-
34901   -/
34902
34903
34904
```