



시스템 콜 및 모듈 프로그래밍

- 시스템 콜
- 인터럽트와 예외
- 모듈
- make 유틸리티

Section 01 시스템 콜

● 시스템 콜의 이해

- ▶ 사용자 모드에 있는 프로세스가 CPU, 디스크, 프린터 등의 하드웨어 장치와 상호 작용할 수 있도록 기능을 제공하는 일련의 인터페이스
- ▶ 소프트웨어 인터럽트를 통한 커널에 대한 서비스 요청
- ▶ 커널 내에 상주하는 저 수준 함수들로서 입출력과 같은 시스템 제어를 위한 기본적인 기능을 제공
- ▶ 사용자 프로세스가 커널의 기능을 이용하기 위해서는 시스템 콜을 사용해야 함
 - 예) 하드 디스크 내의 데이터를 읽기 위해서 `read()` 시스템 콜 호출
 - ✓ 커널은 `read()` 시스템 콜 인터페이스를 제공해야만 가능

Section 01 시스템 콜

- ▶ 사용자 프로세스는 직접적으로 혹은 라이브러리 함수를 이용해서 시스템 콜을 발생시킴
 - 라이브러리 함수
 - ✓ 다른 프로그램들과 링크되기 위하여 존재하는 하나 이상의 서브루틴이나 함수들이 저장된 파일들의 모음
 - ✓ 표준 C 언어 라이브러리에는 커널이 제공하는 시스템 호출을 사용할 수 있도록 하는 래퍼루틴(wrapper routine)을 포함
 - » open() 함수는 실제 시스템 콜 sys_open을 일으키는 래퍼 루틴
- ▶ 시스템 콜이 발생하면 사용자 모드에서 커널 모드로 전환되어 수행
- ▶ 장점
 - 프로그램 호환성 향상
 - ✓ 하드웨어 장치의 특성을 다루어야 하는 저수준 프로그래밍을 몰라도 되므로 프로그래밍이 쉬워짐
 - 시스템의 보안 향상
 - ✓ 요청을 처리하기 전에 인터페이스 수준에서 올바른 요청인지 검사할 수 있음

Section 01 시스템 콜

● 시스템 콜의 종류

- ▶ include/asm/unistd.h에 정의
- ▶ 각각의 시스템 콜을 구별하기 위해 고유 번호 할당
 - 예) open은 5번, read는 3번

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_
/*
 * This file contains the system call numbers.
 */

#define __NR_exit          1
#define __NR_fork          2
#define __NR_read          3
#define __NR_write          4
#define __NR_open          5
#define __NR_close          6
#define __NR_waitpid        7
#define __NR_creat          8
```

```
.....
#define __NR_sched_getaffinity 242
#define __NR_set_thread_area 243
#define __NR_get_thread_area 244
#define __NR_io_setup 245
#define __NR_io_destroy 246
#define __NR_io_getevents 247
#define __NR_io_submit 248
#define __NR_io_cancel 249
#define __NR_alloc_hugepages 250
#define __NR_free_hugepages 251
#define __NR_exit_group 252
```

Section 01 시스템 콜

▶ 시스템 콜 초기화

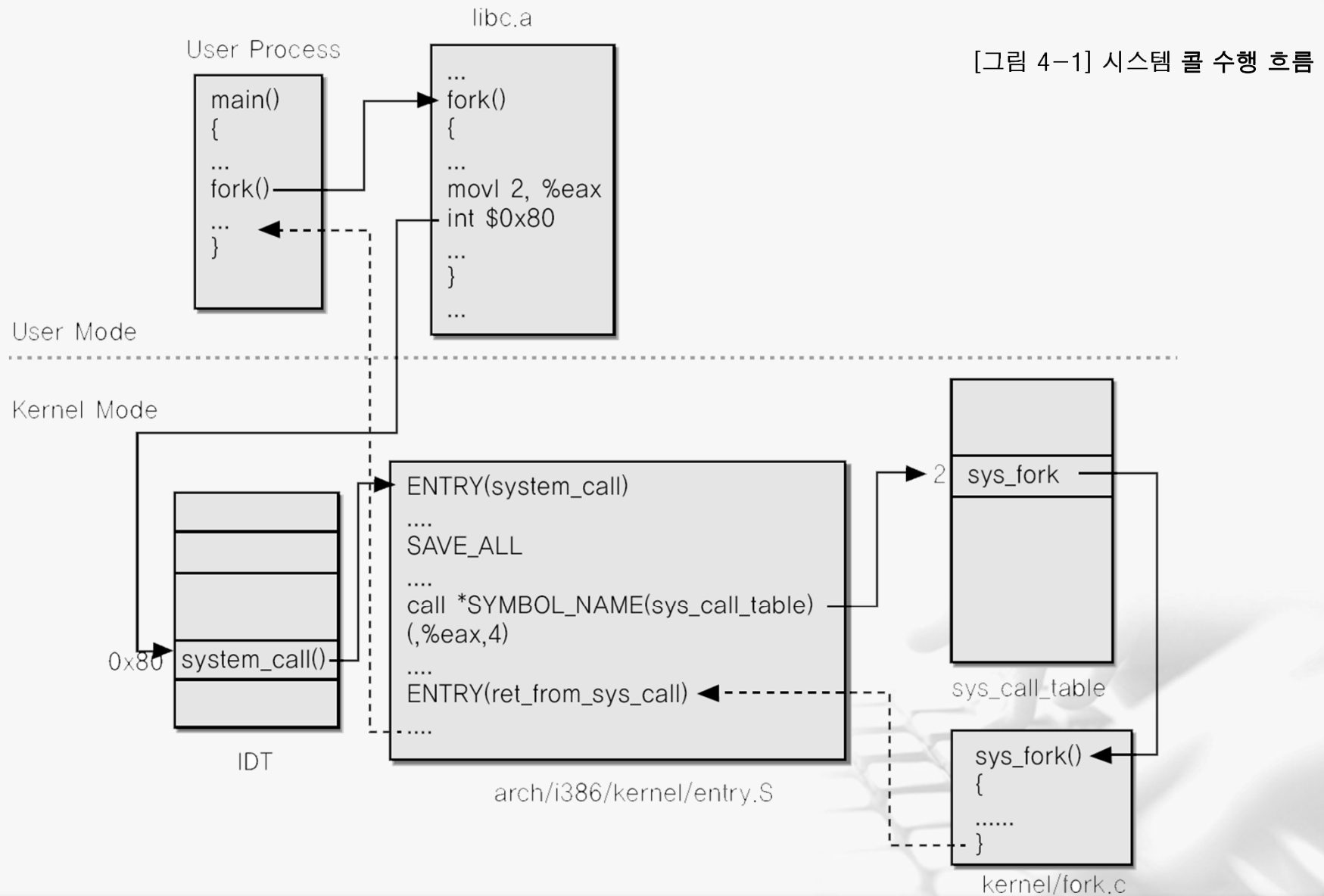
- 시스템 부팅 시 커널 초기화 과정에서 trap_init() 함수에서 수행
 - ✓ set_system_gate(SYSCALL_VECTOR, &system_call)를 수행
 - » 0x80 인터럽트를 위한 게이트 디스크립터(gate descriptor)를 설정
 - » SYSCALL_VECTOR : 0x80으로 정의된 상수
 - » system_call : 시스템 콜 처리를 위한 핸들러의 주소

● 시스템 콜의 처리 과정

▶ 커널

- 내부적으로 각 시스템 콜을 구분하기 위해 각 기능별로 번호를 할당해 둬
 - ✓ 각 번호에 해당하는 제어 루틴을 커널 내부에 정의
- 과정
 - ✓ 응용 프로그램에서 시스템 콜 호출
 - » 사용자 프로세스에서 시스템 콜을 발생시키면 C 라이브러리에서는 발생한 시스템 콜에 해당하는 번호를 특정 레지스터(리눅스에서는 eax)에 올린 후 특정 인터럽트(리눅스에서는 0x80 인터럽트)를 발생
 - ✓ 호출한 시스템 콜에 대응하는 기능 번호를 확인하고 관련된 서비스 루틴 호출
 - ✓ 서비스 루틴이 모두 수행
 - ✓ 응용 프로그램으로 복귀

Section 01 시스템 콜



Section 01 시스템 콜

- ▶ 사용자 프로세스에서 수행 중에 `fork()` 시스템 콜을 호출한다고 가정
 - `fork` 시스템 콜의 고유번호인 2를 `eax` 레지스터에 저장한 다음 `0x80` 인터럽트를 발생
 - 커널은 먼저 IDT에서 시스템 콜의 핸들러 함수인 `system_call()` 호출
 - ✓ `arch/i386/kernel/entry.S`
 - `system_call()`
 - ✓ 호출된 시스템 콜의 번호와 함께 모든 레지스터들을 스택에 저장
 - ✓ 바른 시스템 콜 번호인지 검사
 - ✓ 시스템 콜 테이블(`sys_call_table`)에 등록된 시스템 콜 번호에 해당하는 함수를 호출 (`sys_fork()` - 시스템 콜 서비스 루틴)
 - `ret_from_sys_call()` 함수 호출
 - 사용자 프로세스로 복귀

Section 01 시스템 콜

▶ system_call()

```
01 ENTRY(system_call)
02     pushl %eax                # save orig_eax
03     SAVE_ALL
04     GET_CURRENT(%ebx)
05     testb $0x02,tsk_ptrace(%ebx) # PT_TRACESYS
06     jne tracesys
07     cmpl $(NR_syscalls),%eax
08     jae badsys
09     call *SYMBOL_NAME(sys_call_table)(,%eax,4)
10     movl %eax,EAX(%esp)      # save the return value
```

- 2~3행 : 시스템 콜 번호와 함께 레지스터 값들을 커널 스택에 보관
- 4행 : ebx 레지스터에 현재 프로세스의 프로세스 디스크립터(task_struct) 주소를 저장
- 5~6행 : 프로세스의 EFLAG 값에 PF_TRACESYS가 설정되어 있으면 syscall_trace 함수를 호출

Section 01 시스템 콜

- 7~8행 : 시스템 콜 번호가 유효한 범위 내에 있는지 검사하고 범위를 벗어나면 에러 값과 함께 복귀
- 9행 : 시스템 콜 테이블(sys_call_table)에서 호출된 시스템 콜에 해당하는 엔트리 내의 sys_xxx 함수를 호출
- 10행 : 반환 값을 저장

▶ sys_call_table

```
ENTRY(sys_call_table)
.long SYMBOL_NAME(sys_ni_syscall)          /* 0 */
.long SYMBOL_NAME(sys_exit)
.long SYMBOL_NAME(sys_fork)
.long SYMBOL_NAME(sys_read)
.long SYMBOL_NAME(sys_write)
.long SYMBOL_NAME(sys_open)                /* 5 */
.long SYMBOL_NAME(sys_close)
.long SYMBOL_NAME(sys_waitpid)
.long SYMBOL_NAME(sys_creat)
.....
```

Section 01 시스템 콜

▶ ret_from_sys_call() 함수

- 재스케줄링의 필요 여부 확인
- 대기 중인 시그널 유무 확인
- 이전에 저장했던 CPU 레지스터들을 다시 불러들여 사용자 모드로 전환

```
ENTRY(ret_from_sys_call)
```

```
cli
```

```
cmpl $0, need_resched(%ebx)
```

```
jne reschedule
```

```
cmpl $0, sigpending(%ebx)
```

```
jne signal_return
```

```
restore_all:
```

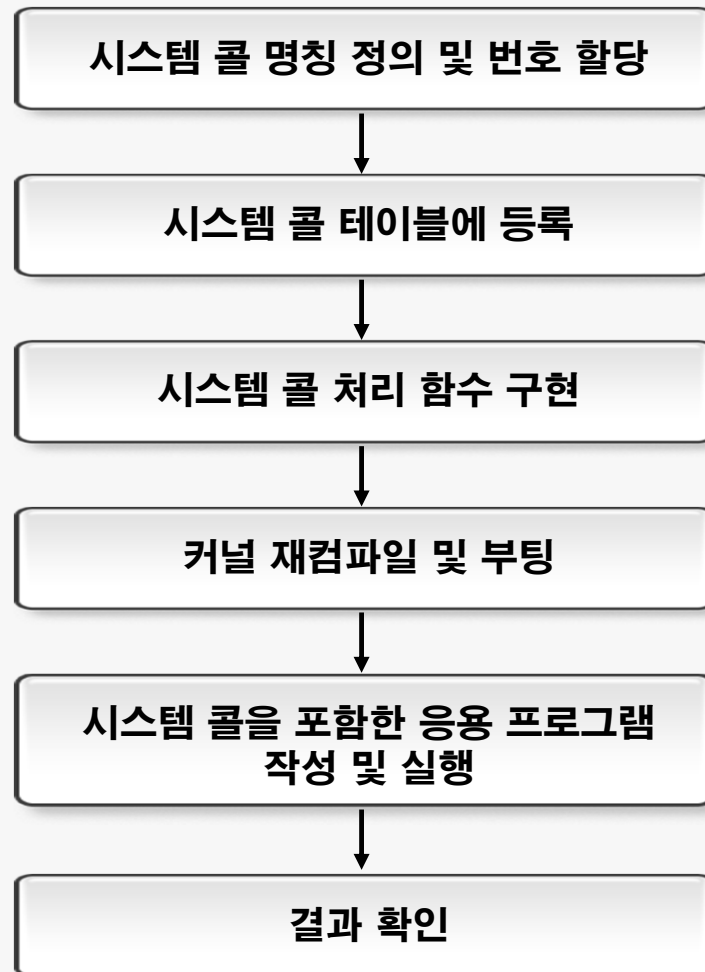
```
RESTORE_ALL
```

```
.....
```

```
.....
```

Section 01 시스템 콜

● 새로운 시스템 콜의 구현



[그림 4-2] 시스템 콜의 구현 및 테스트 순서

[실습 4-1] 간단한 문자열 출력 시스템 콜 구현

문제

시스템 콜을 호출하면 콘솔 상에 다음 문자열을 출력하시오.

"Hello new System Call!"

(1) 시스템 콜의 명칭 및 번호 할당

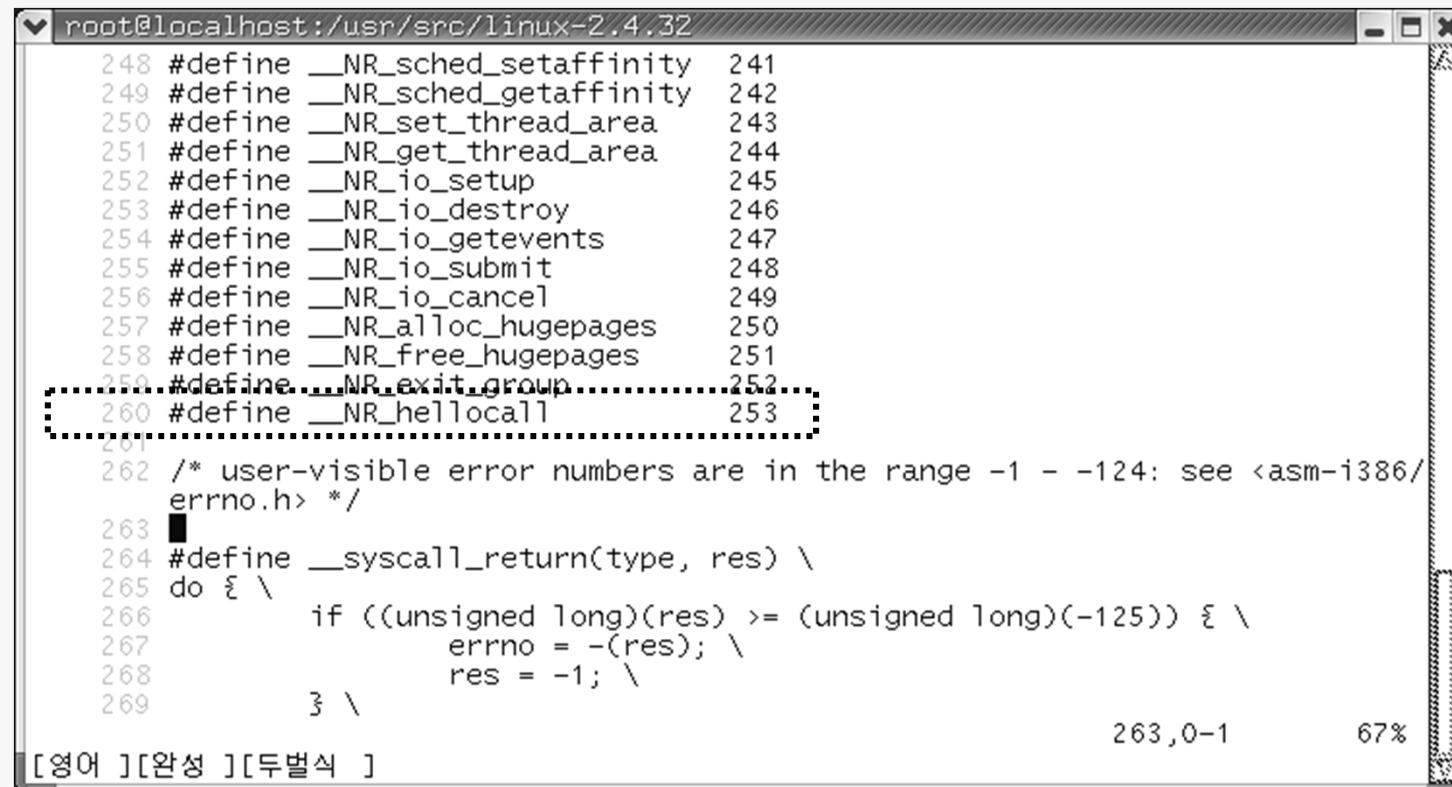
① 시스템 콜의 이름을 결정하고 기존에 사용하지 않는 번호 할당

- `hellocall` , 253

② `include/asm-i386/unistd.h`

- linux에서는 시스템 콜 번호 앞에 `__NR_` 접두어를 붙임
#define `__NR_hellocall` 253 을 추가

[실습 4-1] 간단한 문자열 출력 시스템 콜 구현



```
root@localhost:/usr/src/linux-2.4.32
248 #define __NR_sched_setaffinity 241
249 #define __NR_sched_getaffinity 242
250 #define __NR_set_thread_area 243
251 #define __NR_get_thread_area 244
252 #define __NR_io_setup 245
253 #define __NR_io_destroy 246
254 #define __NR_io_getevents 247
255 #define __NR_io_submit 248
256 #define __NR_io_cancel 249
257 #define __NR_alloc_hugepages 250
258 #define __NR_free_hugepages 251
259 #define __NR_exit_group 252
260 #define __NR_hellocall 253
261
262 /* user-visible error numbers are in the range -1 - -124: see <asm-i386/
errno.h> */
263
264 #define __syscall_return(type, res) \
265 do { \
266     if ((unsigned long)(res) >= (unsigned long)(-125)) { \
267         errno = -(res); \
268         res = -1; \
269     } \
270 } while(0)
```

[영어] [완성] [두벌식]

[그림 4-3] 시스템 콜 명칭 및 번호 할당

[실습 4-1] 간단한 문자열 출력 시스템 콜 구현

(2) 시스템 콜 테이블에 등록

- ① 시스템 콜이 호출되면 IDT를 거쳐 `system_call()` 함수가 수행되면서 참조되는 테이블
- ② `arch/i386/kernel/entry.S` 내에 존재
- ③ 253번째 항목 확인 및 수정
 - `.long SYMBOL_NAME(sys_ni_syscall)`
→ `.long SYMBOL_NAME(sys_hellocall)`
 - `sys_hellocall`
 - ✓ 시스템 콜이 발생하면 이를 처리할 함수
- ④ 주의
 - 시스템 콜의 번호와 시스템 콜 테이블의 위치가 동일하도록 할 것



[실습 4-1] 간단한 문자열 출력 시스템 콜 구현

```

root@localhost:/usr/src/linux-2.4.32
650      .long SYMBOL_NAME(sys_ni_syscall)      /* sys_set_thread_area */
651      /
652      .long SYMBOL_NAME(sys_ni_syscall)      /* sys_get_thread_area */
653      /
654      .long SYMBOL_NAME(sys_ni_syscall)      /* 245 sys_io_setup */
655      .long SYMBOL_NAME(sys_ni_syscall)      /* sys_io_destroy */
656      .long SYMBOL_NAME(sys_ni_syscall)      /* sys_io_getevents */
657      .long SYMBOL_NAME(sys_ni_syscall)      /* sys_io_submit */
658      .long SYMBOL_NAME(sys_ni_syscall)      /* sys_io_cancel */
659      .long SYMBOL_NAME(sys_ni_syscall)      /* 250 sys_alloc_hugepag
es */
660      .long SYMBOL_NAME(sys_ni_syscall)      /* sys_free_hugepages */
661      .long SYMBOL_NAME(sys_ni_syscall)      /* sys_exit_group */
662      .long SYMBOL_NAME(sys_helloall)        /* display string */
663      .long SYMBOL_NAME(sys_ni_syscall)      /* sys_epoll_create */
664      .long SYMBOL_NAME(sys_ni_syscall)      /* sys_epoll_ctl 255 */
665      .long SYMBOL_NAME(sys_ni_syscall)      /* sys_epoll_wait */
666      .long SYMBOL_NAME(sys_ni_syscall)      /* sys_remap_file_pages
*/
667      .long SYMBOL_NAME(sys_ni_syscall)      /* sys_set_tid_address */
668      /
669      .rept NR_syscalls-(.-sys_call_table)/4
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1
```

[그림 4-4] 시스템 콜 테이블 수정 화면

[실습 4-1] 간단한 문자열 출력 시스템 콜 구현

(3) 새로운 시스템 콜 처리 함수 구현

① 기존 파일의 일부분으로 추가하여 작성하거나 새로운 파일로 구현

② kernel/hellocall.c

- kernel 디렉토리 : 태스크와 관련된 함수들이 일반적으로 존재

```
1 /* hellocall.c */  
2 #include <linux/kernel.h>  
3  
4 asmlinkage int sys_hellocall()  
5 {  
6     printk( "Hello Linux System Call! \n" );  
7     return 0;  
8 }
```

[소스 4-1]

③ 시스템 콜 처리 함수의 타입은 정수형이며 정상적으로 수행을 마치면 0 값을 반환

④ asmlinkage

- C로 구현된 함수가 어셈블리 언어로 구현된 함수에서 호출될 때 사용하는 키워드

[실습 4-1] 간단한 문자열 출력 시스템 콜 구현

- ⑤ **printk** : 커널 수준에서 문자열을 터미널에 출력하는 커널 라이브러리 함수
 - **printf** : 사용자 수준에서 수행되는 표준 C 라이브러리 함수
 - **sys_helloall()** 은 커널 수준에서 수행되는 함수이므로 사용자 수준에서 수행되는 표준 C 라이브러리를 사용할 수 없다

(4) 커널 재컴파일 및 시스템 재부팅

- ① 구현한 시스템 콜 처리 함수가 동작하기 위해서는 커널 재컴파일 필요
- ② 재부팅
- ③ 해당 파일이 컴파일 시에 링크되도록 kernel/Make 파일 수정

```
obj-y = sched.o dma.o fork.o exec_domain.o panic.o printk.o \
module.o exit.o itimer.o info.o time.o softirq.o resource.o \
sysctl.o acct.o capability.o ptrace.o timer.o user.o \
signal.o sys.o kmod.o context.o helloall.o
```

- ④ **#make bzImage**
- ⑤ **/boot** 디렉토리로 커널 이미지 복사
- ⑥ **#make modules**(필요없음)
- ⑦ **#reboot**

[실습 4-1] 간단한 문자열 출력 시스템 콜 구현

(5) 응용 프로그램 작성 및 실행 확인

① 시스템 콜을 호출하는 명령을 포함한 응용 프로그램 작성

② app_hellocall.c

- 자신의 계정에서 작성

```
01 /* app_hellocall.c */
02 #include <linux/unistd.h>
03 #include <errno.h>
04 _syscall0(int, hellocall)
05
06 int main()
07 {
08     int i;
09     printf( "Before System Call \n");
10     i = hellocall();
11     printf( "After System Call \n");
12 }
```

[소스 4-2]

[실습 4-1] 간단한 문자열 출력 시스템 콜 구현

③ `_syscall0(int, hellocall)`

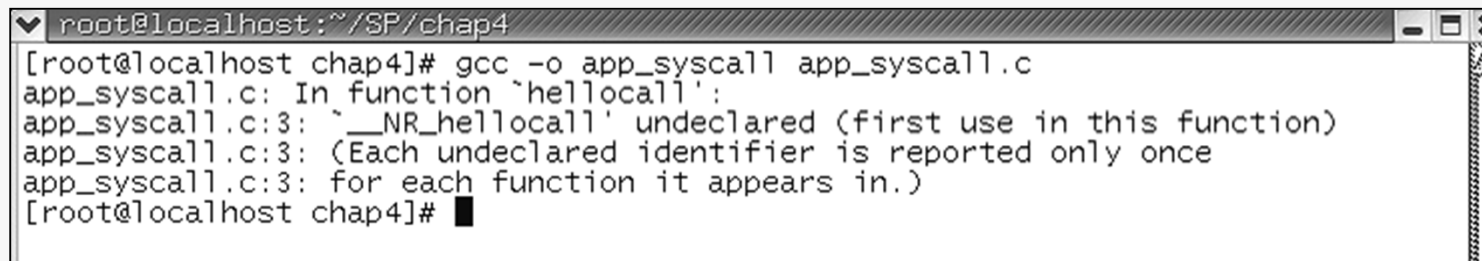
- `syscall0` - 인자가 없는 시스템 콜임을 의미
- `int` - 시스템 콜 처리 함수의 `type`을 정의
- `hellocall` - 시스템 콜의 이름

④ `i = hellocall()`

- 시스템 콜을 호출
- 만약 `hellocall`이 커널에 없는 경우 `i`에는 `-1` 값이 반환되며 정상적으로 처리되면 시스템 콜 처리 함수에서 `0`을 반환

(6) 컴파일

① `#gcc -o app_hellocall app_hellocall.c` ⇒ 에러 발생



```
root@localhost:~/SP/chap4
[root@localhost chap4]# gcc -o app_syscall app_syscall.c
app_syscall.c: In function 'hellocall':
app_syscall.c:3: '__NR_hellocall' undeclared (first use in this function)
app_syscall.c:3: (Each undeclared identifier is reported only once
app_syscall.c:3: for each function it appears in.)
[root@localhost chap4]#
```

[그림 4-5] 응용 프로그램 컴파일 수행 및 에러 메시지

[실습 4-1] 간단한 문자열 출력 시스템 콜 구현

② 오류 원인

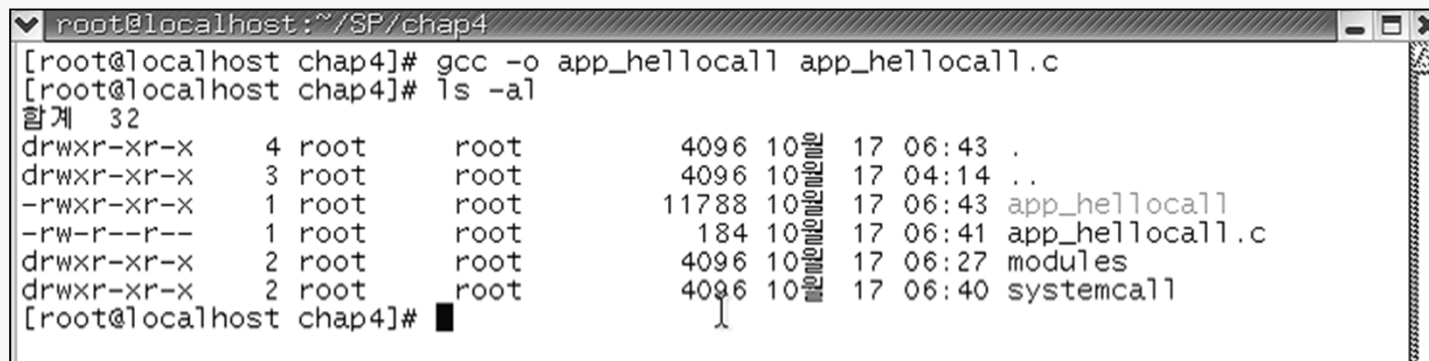
- 응용 프로그램에서 참조하는 unistd.h 파일을 linux/unistd.h로 정의했기 때문
- /usr/include/linux/unistd.h → /usr/include/asm/unistd.h
 - ✓ 보통 사용자 프로그램을 컴파일할 때 gcc 컴파일러가 참조하는 헤더 파일

③ 해결방법

- /usr/include/asm/unistd.h 파일도 커널에서 수정해 준 내용과 같이 수정
- gcc 컴파일 옵션에 커널 소스 내의 헤더 파일을 포함
 - ✓ #gcc -I /usr/src/linux-2.4.32/include -o app_helloall app_helloall.c

(7) 응용 프로그램 실행 및 결과 확인

① #./app_helloall



```
root@localhost: ~/SP/chap4
[root@localhost chap4]# gcc -o app_helloall app_helloall.c
[root@localhost chap4]# ls -al
합계 32
drwxr-xr-x  4 root    root      4096 10월 17 06:43 .
drwxr-xr-x  3 root    root      4096 10월 17 04:14 ..
-rwxr-xr-x  1 root    root     11788 10월 17 06:43 app_helloall
-rw-r--r--  1 root    root       184 10월 17 06:41 app_helloall.c
drwxr-xr-x  2 root    root      4096 10월 17 06:27 modules
drwxr-xr-x  2 root    root      4096 10월 17 06:40 systemcall
[root@localhost chap4]#
```

[그림 4-8] 컴파일 완료 및 실행 파일 생성 확인 화면

[실습 4-1] 간단한 문자열 출력 시스템 콜 구현

② cat /var/log/message 또는 텍스트 콘솔 상에서 다음과 같이 실행

```
[root@localhost chap4]# ls
app_hellocall app_hellocall.c modules syscall
[root@localhost chap4]# ./app_hellocall
Before System call
Hello Linux System Call!
After System call
[root@localhost chap4]# _
```

[그림 4-10] 텍스트 콘솔 상에서의 결과 확인 화면



Section 01 시스템 콜

● 인자가 있는 시스템 콜 구현

▶ [실습 4-1] 의 경우

- 사용자 프로그램에서 인자가 없이 단순히 시스템 콜만을 호출하고 출력도 커널 수준에서 수행하는 가장 단순한 시스템 콜

▶ 많은 시스템 콜에서 인자를 필요로 함

- 예) `open()` - 파일의 이름과 접근 권한 등
- 인자는 단순한 변수일 수도 있고 문자열이나 또는 구조체가 될 수 있음

▶ 어떻게 인자를 전달하는가?

▶ 커널 수준의 데이터를 어떻게 응용 프로그램으로 전달하는가?

- 커널이 사용하는 메모리 공간과 응용 프로그램이 사용하는 메모리 공간이 다르기 때문에 일반 응용 프로그램 사이에서 데이터를 주고 받는 형식으로는 처리 불가능

Section 01 시스템 콜

▶ syscall 매크로

- 인자의 개수만큼 인자를 넘겨주고 리턴 값을 정의
- `include/asm-i386/unistd.h`
- `_syscall0 ~ _syscall6`
- 사용 예)
 - ✓ 이름이 `addcall`이고 두 개의 정수 값을 더하여 결과 값을 돌려받는 시스템 콜을 구현한다고 가정
 - ✓ `_syscall3(int, addcall, int, x, int, y, int*, result)` 형식 사용
 - » `x, y`는 두 개의 정수 값에 해당하는 인자
 - » `result`는 결과 값을 돌려받기 위한 인자
 - ✓ 인자가 없는 [실습 4-1]의 `hellocall()`을 수정
→ `_syscall0` 매크로를 사용하여 `_syscall0(int, hellocall)`로 정의함.



Section 01 시스템 콜

▶ 커널 영역의 데이터를 사용자 영역으로 전달하는 방법

- `copy_to_user(to, from, n)`
 - ✓ 커널 영역 `from`을 사용자 영역 `to`로 `n` 크기만큼 복사
- `put_user(x, ptr)`
 - ✓ 커널 영역의 `x` 변수 값을 사용자 영역의 `ptr` 메모리 값에 대입
- `copy_from_user(to, from, n)`
 - ✓ 사용자 영역 `from`을 커널 영역 `to`로 `n` 크기만큼 복사
- `get_user(x, ptr)`
 - ✓ 사용자 영역의 `ptr` 메모리 값을 커널 영역의 `x` 변수 값에 대입

- 사용 예) `put_user(sum, result)`
 - ✓ `result`는 포인터 변수



[실습 4-2] 인자가 있는 시스템 콜의 구현

문제

[실습 4-1]을 참고하여 두 개의 정수 값을 더하여 결과 값을 반환하는 간단한 시스템콜을 작성해보자.

- 시스템 콜 번호는 254를 할당
- 시스템 콜의 이름은 `addcall`
- 시스템 콜 처리 함수는 [실습 4-1]의 `hellocall.c` 내에 작성하든지 별도의 파일 (예:`addsys.c`)로 작성



[실습 4-2] 인자가 있는 시스템 콜의 구현

(1) 시스템 콜 처리 함수 작성

```
/* addcall.c */  
#include <linux/kernel.h>  
#include <asm/uaccess.h>  
asmlinkage int sys_addcall(int x, int y, int *result){  
    int sum;  
    sum = x + y;  
    put_user( sum, result );  
    return (0);  
}
```

[소스 4-3]

① put_user(sum, result)

→ copy_to_user(result, &sum, sizeof(int)) 로 대체 가능

② verify_area() 함수를 추가하는 것이 바람직 함

- 사용자 공간인 result에 데이터를 쓰기 전에 메모리의 유효 검사 수행
- verify_area(type, addr, size)
→ verify_area(VERIFY_WRITE, result, sizeof(int))

[실습 4-2] 인자가 있는 시스템 콜의 구현

(2) 사용자 응용 프로그램 작성

```
01 /* myapp3.c */
02 #include <linux/unistd.h>
03 #include <errno.h>
04
05 _syscall3(int, addcall, int, x, int, y, int*, result)
06
07 int main(){
08     int i, x, y, add_result;
09     printf("input : ");
10     scanf("%d %d",&x, &y);
11     i = addcall(x, y, &add_result);
12     printf("Result : %d + %d = %d \n", x, y, add_result);
13 }
```

[소스 4-4]

- ① 인자가 3개이므로 _syscall3(int, addcall, int, x, int, y, int*, result)로 설정

[실습 4-2] 인자가 있는 시스템 콜의 구현

(3) 실행 결과 확인

```
# gcc -o app_addcall app_addcall.c
```

```
# ./app_addcall
```



```
root@localhost:~/SP/chap4
[root@localhost chap4]# gcc -o app_addcall app_addcall.c
[root@localhost chap4]# ./app_addcall
Input 2 integer value : 10 20
10 + 20 = 30
[root@localhost chap4]# ./app_addcall
Input 2 integer value : 5 4
5 + 4 = 9
[root@localhost chap4]#
```

[영어] [완성] [두벌식]

Section 02 인터럽트와 예외

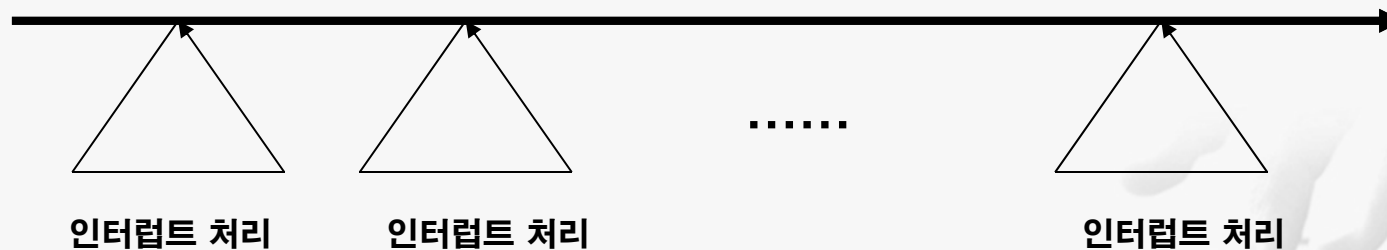
● 인터럽트에 대한 이해

- ▶ 어떤 일을 수행하는 도중에 끼어드는 사건(이벤트(event))
- ▶ 프로세스가 실행하는 명령어의 순서를 바꾸는 사건
- ▶ 하드웨어 인터럽트 vs 소프트웨어 인터럽트

- 예외

- ✓ divide by zero, 세그먼테이션 결함, 페이지 결함, 보호 결함, 잘못된 명령의 수행 등

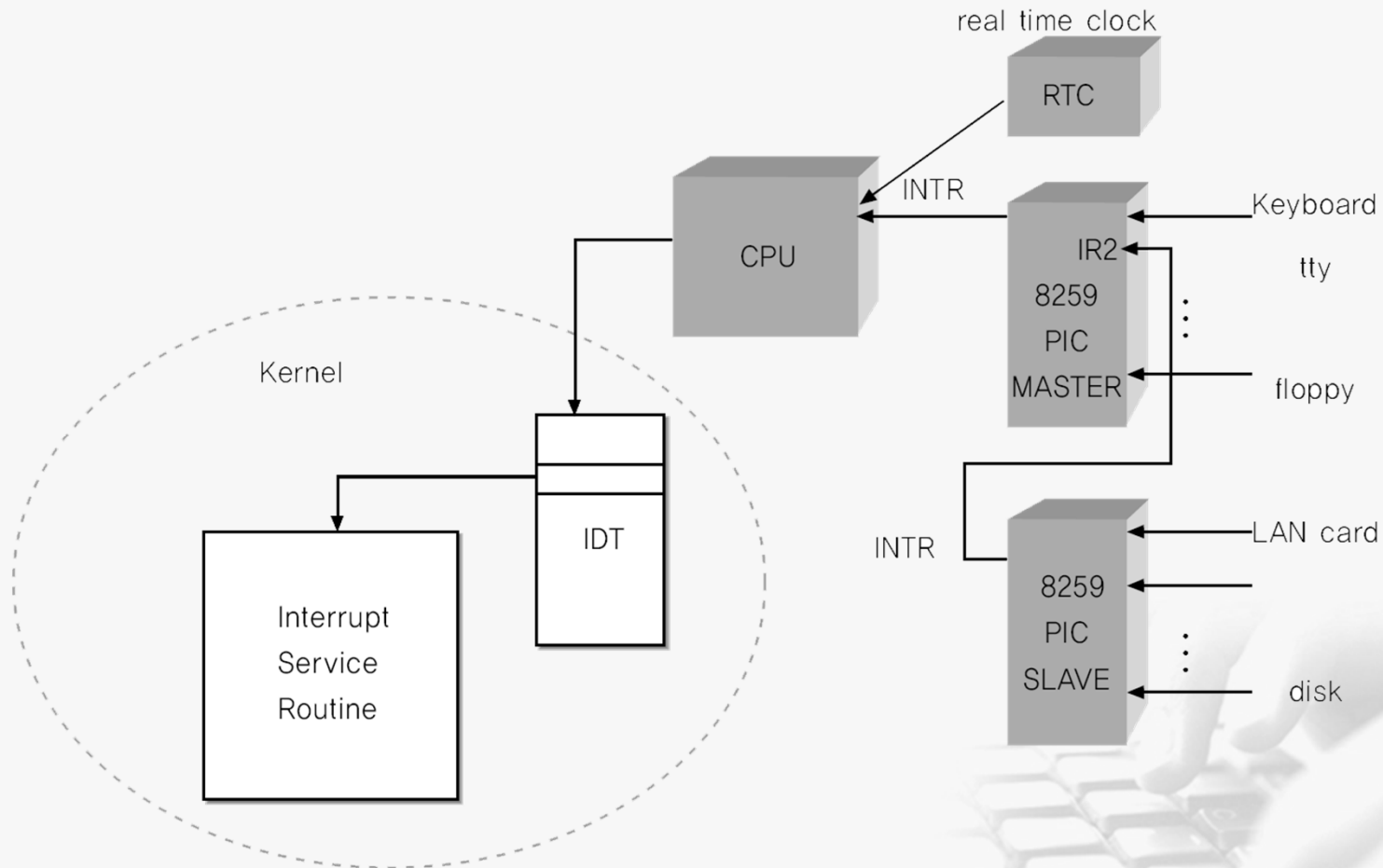
▶ 인터럽트 처리



[그림 4-12] 인터럽트 처리

Section 02 인터럽트와 예외

▶ 하드웨어 인터럽트



[그림 4-11] 하드웨어 인터럽트 처리

Section 02 인터럽트와 예외

● 미룰 수 있는 인터럽트

- ▶ 미룰 수 있는 인터럽트 처리는 뒤로 미루자는 개념
 - bottom-half
- ▶ 인터럽트는 빠른 시간 내에 처리되어야 함
 - 응답 시간의 향상
- ▶ 인터럽트가 발생하면 중요한 부분만 빨리 처리하고 뒤로 미뤄서 처리할 수 있는 부분은 미루자는 것
 - 예) 네트워크 카드가 수신한 패킷 처리
- ▶ SMP 환경을 위해 softirq와 tasklet 도입
- ▶ softirq
 - 같은 종류(type)의 softirq들이 여러 CPU에서 동시에 실행 가능
 - 동기화 문제 해결 필요, 재진입 가능한 코드 필요
 - 제한된 수의 softirq만을 사용
 - 성능을 높이기 위한 목적

Section 02 인터럽트와 예외

▶ tasklet

- 실행 시 동적으로 할당 가능
- 같은 종류의 tasklet이 여러 CPU에서 동시에 실행 불가능
- 다른 종류의 tasklet은 여러 CPU에서 동시에 실행 가능
- 동기화 문제 발생 안함
- 구현이 용이
- 대부분의 I/O 디바이스에서 사용하는 기법

▶ bottom-half

- 한 CPU에서 bottom-half를 실행하고 있으면
다른 CPU에서 다른 종류의 bottom-half도 실행 불가능
- 이전 버전과의 호환을 위해 존재
- 거의 사용 안 함
- 2.6 버전에서는 없어짐



Section 02 인터럽트와 예외

● IDT 테이블

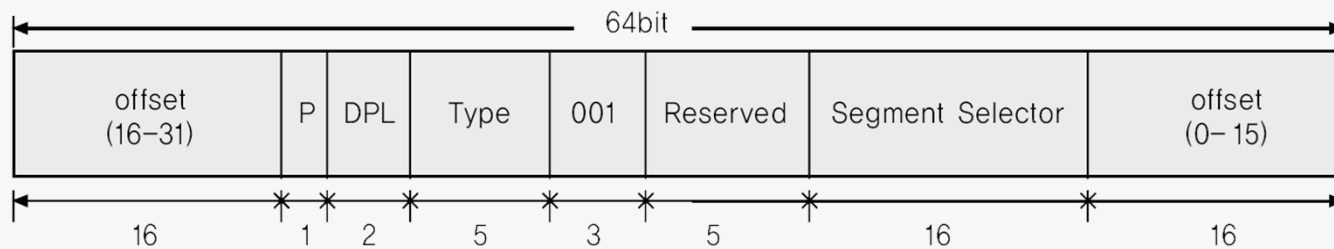
- ▶ 인터럽트를 처리 함수의 시작점 주소(핸들러 함수) 등록
 - 트랩과 시스템 콜도 마찬가지
- ▶ IDTR 레지스터가 가리킴
- ▶ `arch/i386/kernel/traps.c`
 - `idt_table` 구조체
 - `start_kernel()`의 `init_IRQ()` 함수에서 테이블 초기화
- ▶ 256개(0~0xff)의 엔트리
 - 각 엔트리는 8byte 크기의 디스크립터로 구성
 - ✓ $256\text{개} * 8\text{byte} = 2048\text{byte}$ 크기
 - 0~19(0x00~0x13) : 트랩
 - 32(0x20) ~ : 하드웨어 인터럽트
 - 128(0x80) : 시스템 콜



Section 02 인터럽트와 예외

● 디스크립터

- ▶ 태스크 게이트 디스크립터, 인터럽트 게이트 디스크립터, 트랩 게이트 디스크립터
 - 리눅스에서는 태스크 게이트 디스크립터를 사용 안함
 - 대신 시스템 게이트 사용(예: 시스템 콜과 같이 사용자 프로세스에서 사용)



- ✓ P : Present, 인터럽트를 수행하기 위해 세트
- ✓ type : 각 디스크립터 종류를 구분
- ✓ DPL : descriptor privilege level, 접근 권한 설정
 - » 인터럽트와 예외는 0을, 시스템 콜은 3을 사용
- ✓ offset : 인터럽트 핸들러 루틴의 주소 값
- ✓ Segment Selector : 인터럽트 핸들러가 위치한 코드 세그먼트

Section 02 인터럽트와 예외

● do_IRQ

- ▶ 인터럽트가 발생하면 실행하던 프로세스를 멈추고 커널 모드로 전환
→ do_IRQ 함수를 호출
- ▶ 인터럽트에 연관된 모든 인터럽트 서비스 루틴을 실행하기 위해 호출하는 함수
- ▶ 소스는 arch/i386/kernel/irq.c에 포함되어 있음

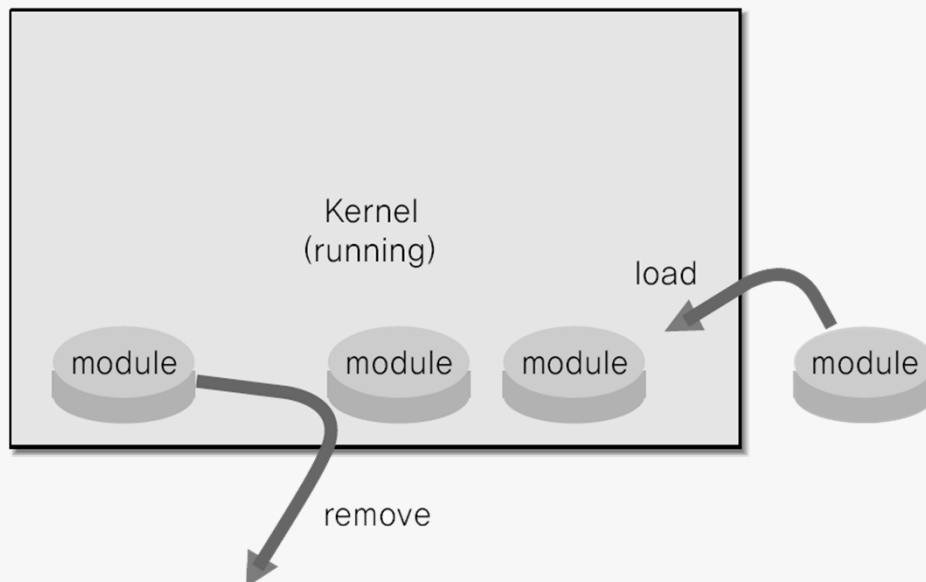
▶ 처리

- PIC(Programmable Interrupt Controller)에게 응답 신호를 송신
- 인터럽트를 처리하는 도중에 이 종류의 인터럽트(즉, 같은 IRQ번호)가 발생하는 것을 방지
- handle_IRQ_event() 함수를 호출
 - ✓ action->handler(irq, action->dev_id, regs)호출
 - » 실제 핸들러 함수를 호출하여 인터럽트 서비스 루틴을 실행
- softirq가 있다면 처리(do_softirq())도 수행
- 사용자 프로세스로 복귀

Section 03 모듈

● 모듈의 이해

- ▶ 커널 코드의 일부를 커널이 동작하는 상태에서 로드 또는 언로드 가능
- ▶ 커널 크기 최소화, 유연성 제공
- ▶ 각종 디바이스 드라이버를 사용할 때 유용
 - 마우스, 키보드, 사운드카드 드라이버는 종류가 다양하고 상황에 따라 사용하지 않을 수 있기 때문
 - ✓ 새로운 장치를 추가할 때마다 커널을 재컴파일한다면?
- ▶ 파일시스템, 통신 프로토콜 및 시스템 콜 등도 모듈로 구현 가능



[그림 4-14] 모듈의 동적 적재 및 제거

Section 03 모듈

● 특징

- ▶ main() 함수 없음
- ▶ 기본적으로 init_module() 함수와 cleanup_module() 함수를 포함
 - init_module() : 모듈이 커널에 적재될 때 호출되는 함수
 - cleanup_module() : 모듈이 커널에서 제거될 때 호출되는 함수
- ▶ 오브젝트(.o) 파일로 적재
- ▶ linux/module.h 헤더 파일

● 사용 명령

- ▶ insmod
 - 커널 오브젝트를 커널에 링크시키도록 도와주는 외부 유틸리티
- ▶ rmmod
 - 커널에서 모듈을 제거하는 명령
- ▶ lsmod
 - 커널에 적재된 모듈의 목록 출력
- ▶ 기타 : depmod, modprobe

Section 03 모듈

● 커널 모듈 추가 시

- ① 커널 모듈이 적재되면 오브젝트 파일의 내용이 커널 영역으로 복사
- ② `init_module()` 함수를 호출하여 적재된 커널 모듈 초기화
- ③ 커널 모듈의 초기화가 끝나면 커널 모듈 등록

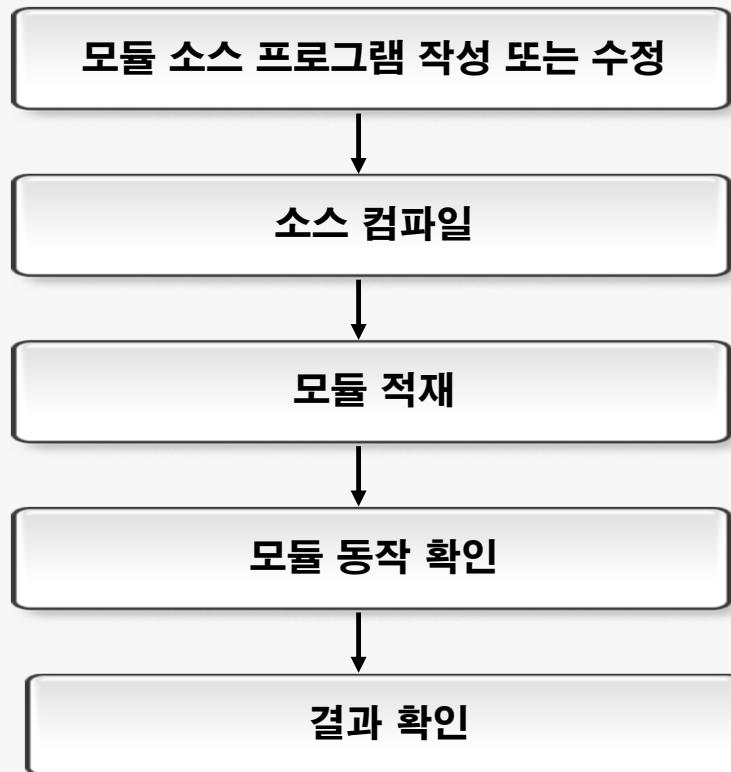
● 커널 모듈을 제거 시

- ① 커널 모듈이 제거되면 `cleanup_module()` 함수를 호출
- ② `init_module()` 함수에서 할당받은 자원을 반환
- ③ 커널 모듈의 등록 해제
- ④ 커널 모듈의 오브젝트 코드를 위해 할당했던 메모리를 반환



Section 03 모듈

● 모듈 프로그래밍 절차



[그림 4-15] 모듈 프로그래밍 절차

[실습 4-3] 모듈 로드/언로드 실습

문제

모듈을 커널에 올리고 내리는 과정을 보기 위해 간단한 문자열을 출력하는 실습이다.
적재와 제거가 동적으로 수행되는지 알아보자.

(1) 모듈 소스 코드 작성

```
01 /* hellomodule.c */
02
03 #define MODULE
04 #include <linux/module.h>
05 #include <linux/kernel.h>
06
07 int init_module() {
08     printk( "Module init \n" );
09     printk( "Hello Linux Module! \n" );
```

```
10     return 0;
11 }
12 void cleanup_module() {
13     printk( "Module Cleaned up \n" );
14 }
15 MODULE_LICENSE("GPL");
```

[소스 4-5]

[실습 4-3] 모듈 로드/언로드 실습

① 3행 : 프로그램의 내용이 커널 모듈임을 정의하는 문장

➤ 다른 헤더 파일들 앞에 선언해야 함

➤ 커널 버전 2.6에서는 #define MODULE 대신 #include <linux/init.h>를 사용

② 4행 : module.h에는 커널 모듈 프로그래밍에 필요한 정의, 함수 등이 선언

③ 8행~12행 : init_module() 함수는 insmod 명령으로 모듈이 적재될 때 호출

➤ 여기서는 콘솔 화면에 'Module init'과 'Hello Linux Module!' 두 문자열이 차례로 출력됨

④ 13행~15행 : cleanup_module() 함수는 rmmod 명령을 통해 호출됨

➤ 커널 모듈이 메모리에서 제거되며, 여기서는 콘솔 화면에 'Module Cleaned up' 문자열이 출력됨

⑤ 16행 : 모듈의 라이선스를 명시

➤ 리눅스 커널은 GPL 라이선스를 따름

➤ 이 부분이 명시되지 않으면 모듈 적재 시 경고 메시지가 출력됨

[실습 4-3] 모듈 로드/언로드 실습

(2) 모듈 컴파일

```
TARGET = hellomodule
INCLUDE = -isystem /usr/src/linux-2.4.32/include
CFLAGS = -O2 -D__KERNEL__ -DMODULE $(INCLUDE)
CC = gcc
$ {TARGET }.o: $ {TARGET }.c
clean :
    rm -rf $ {TARGET }.o
```

▶ Makefile 옵션

- -O2 : 어셈블리 매크로와 같은 함수 호출을 위해 사용
- -W -Wall : 컴파일러 경고 기능
- -isystem
 - ✓ 컴파일 대상이 되는 커널의 헤더를 사용하기 위한 옵션
 - ✓ gcc 컴파일러는 기본적으로 /usr/include/의 헤더 파일을 참조함
- -D__KERNEL__ -DMODULE
 - ✓ 헤더 파일에 이 코드가 커널 모드에서 동작하며 모듈이라는 것을 알리는 옵션
 - ✓ 단, 소스 코드에서 #define MODULE을 정의하였다면 필요 없음

[실습 4-3] 모듈 로드/언로드 실습

(3) 실행 결과 확인

```
root@localhost:~/SP/chap4/modules
[root@localhost modules]# ls
Makefile Makefile~ hellomodule.c temp
[root@localhost modules]# make
gcc -O2 -isystem /usr/src/linux-2.4.32/include -c -o hellomodule.o hellomodule
.c
[root@localhost modules]# ls
Makefile Makefile~ hellomodule.c hellomodule.o temp
[root@localhost modules]#
```

[영어] [완성] [두벌식]

[실습 4-3] 모듈 로드/언로드 실습

(4) 모듈 적재

- ① #insmod hellomodule.o로 생성된 오브젝트 파일을 커널에 로드

```
[root@localhost modules]# insmod hellomodule.o
Module Init
Hello Linux Module!
[root@localhost modules]# _
```

(5) 모듈 제거

- ① #rmmod hellomodule로 적재된 모듈을 제거
 - 확장자가 포함되지 않음에 유의

```
[root@localhost modules]# rmmod hellomodule
Module Cleaned up
[root@localhost modules]# _
```

[실습 4-4] 모듈을 이용한 시스템 콜의 구현

문제

시스템 콜을 일으킨 응용 프로그램(프로세스)과 이 프로세스의 부모 프로세스의 PID 값을 커널로부터 반환받아 출력해보자.

- `sys_getpid` : 시스템 콜을 호출한 프로세스에게 PID 값 반환
- `sys_getppid` : 호출한 프로세스의 부모 프로세스의 PID 값을 반환
- 이 두 값을 하나의 시스템 콜에서 반환받아 응용 프로그램에서 출력하도록 구현



[실습 4-4] 모듈을 이용한 시스템 콜의 구현

(1) 모듈 소스 코드 작성

[소스 4-7]

```
01 /* mygetpid.c */
02 #include <linux/kernel.h>
03 #include <linux/module.h>
04 #include <sys/syscall.h>
05 #include <asm/uaccess.h>
06
07 #define __NR_mygetpid 257
08
09 asmlinkage int (*saved_entry)(void);
10 extern void *sys_call_table[];
11
12 struct pid_ppid {
13     int pid;
14     int ppid;
15 };
16
17 asmlinkage int sys_mygetpid(struct
    pid_ppid *pp)
18 {
19
20     struct pid_ppid kpp;
21
22     kpp.pid = current->pid;
23     kpp.ppid = current->p_pptr->pid;
24     copy_to_user(pp, &kpp,
        sizeof(struct pid_ppid) );
25
26     return 0;
27 }
28
```

[실습 4-4] 모듈을 이용한 시스템 콜의 구현

(1) 모듈 소스 코드 작성(계속)

[소스 4-7]

```
29 int module_start()
30 {
31
32     saved_entry = sys_call_table[__NR_mygetpid];
33     sys_call_table[__NR_mygetpid] = sys_mygetpid;
34     return 0;
35 }
36
37 void module_end(void)
38 {
39     sys_call_table[__NR_mygetpid] = saved_entry;
40 }
41
42 module_init(module_start);
43 module_exit(module_end);
44
45 MODULE_LICENSE("GPL");
```

[실습 4-4] 모듈을 이용한 시스템 콜의 구현

① 7행 : 새로 작성되는 시스템 콜의 번호를 지정

➤ 여기서는 기존에 사용되지 않는 257번을 사용

② 9행 : saved_entry는 이 모듈에 제거되었을 때 기존의 시스템 콜 기능으로 복귀하기 위해 사용

➤ 이 실습에서는 이 시스템 콜 번호를 사용하는 시스템 콜이 존재하지 않으므로 필요 없을 수도 있지만 다른 시스템 콜을 래핑(wrapping)하는 경우에는 꼭 필요한 부분이다.

③ 10행 : 시스템 콜 테이블(sys_call_table)에 접근하기 위해 외부 변수 참조

➤ sys_call_table이 존재하는 메모리 위치를 직접 확인하고 싶다면 #cat /boot/System.map | grep sys_call_table 명령으로 확인

➤ 외부 변수를 참조하지 않고 시스템 콜 테이블의 위치를 직접 사용하여 모듈을 작성 가능

④ 12행~15행 : 프로세스의 pid 값과 부모 프로세스의 pid 값을 저장할 변수들로 구성된 구조체를 선언

[실습 4-4] 모듈을 이용한 시스템 콜의 구현

⑤ 17행~27행 : 시스템 콜 처리 함수 부분

- 17행 : 시스템 콜 처리 함수의 이름은 `sys_mygetpid`로 하며 구조체를 인자로 받는다.
- 22행 : 현재 프로세스(current)의 pid 값을 구조체 변수에 저장한다.
- 23행 : 현재 프로세스의 부모 프로세스의 pid 값을 저장한다.
- 24행 : 현재 커널 공간에 있는 구조체의 내용을 사용자 영역으로 복사하기 위해 `copy_to_user()` 함수를 사용
 - ➡ `put_user` 대신 배열이나 구조체 같은 값은 사용자 영역에 복사

⑥ 32행 : 현재 시스템에 설정되어 있는 257번의 시스템 콜 테이블을 9행의 `saved_entry`로 저장

- 모듈이 제거될 때 다시 원래의 시스템 콜 테이블로 설정해 두기 위함

⑦ 33행 : 새로운 시스템 콜을 테이블에 등록

⑧ 39행 : `rmmod` 명령으로 모듈이 제거될 때 새로운 시스템 콜의 적용을 해제하기 위해 원래의 값으로 복원

[실습 4-4] 모듈을 이용한 시스템 콜의 구현

⑨ 42행~43행

- insmod 명령을 실행했을 때 module_init() 함수를 사용
- 2.4 버전 이전 : init_module/cleanup_module 함수 사용
- 2.4 후반/2.6 버전: module_init()/module_exit() 매크로를 이용하여 모듈의 초기화와 제거 시에 원하는 함수 이름 사용 가능
- 모듈이 적재될 때 module_init() 함수가 호출되고, 다시 module_start() 호출
- [실습 4-3]과 같은 형식으로 바꾸어도 문제는 없음

(2) Makefile의 작성과 컴파일

```
TARGET = mygetpid
INCLUDE = -isystem /usr/src/linux-2.4.32/include
CFLAGS = -O2 -D__KERNEL__ -DMODULE $(INCLUDE)
CC = gcc
${TARGET}.o: ${TARGET}.c
clean :
    rm -rf ${TARGET}.o
```

[실습 4-4] 모듈을 이용한 시스템 콜의 구현

(3) 모듈 적재

```
# insmod mygetpid.o
```

(4) 사용자 응용 프로그램 작성

```
01 /*app_mygetpid.c */  
02 #include <linux/unistd.h>  
03 #include <errno.h>  
04  
05 struct pid_ppid {  
06     int pid;  
07     int ppid;  
08 };  
09  
10 _syscall1(int, mygetpid, struct pid_ppid*, pp)  
11
```

[소스 4-9]

[실습 4-4] 모듈을 이용한 시스템 콜의 구현

(4) 사용자 응용 프로그램 작성(계속)

```
12 int main()
13 {
14     int i, opid;
15     struct pid_ppid pp;
16
17     opid = getpid();
18     printf( "Original getpid System call PID = %d \n", opid );
19     i = mygetpid(&pp);
20     if ( i == 0 ) {
21         printf( "My getpid System call PID = %d, PPID = %d \n",
22             pp.pid, pp.ppid );
23     }
24     else {
25         printf( "Error \n" );
26     }
27 }
```

[소스 4-9]

[실습 4-4] 모듈을 이용한 시스템 콜의 구현

- ① 10행 : 호출한 시스템 콜이 하나의 구조체 타입을 인자를 받는다.
- ② 17행~18행 : 기존의 시스템에 존재하는 getpid() 함수를 사용하여 이 프로세스의 PID 값을 받아 출력
 - 소스를 수정하여 getppid() 함수를 통해 부모 프로세스의 pid를 확인할 것
- ③ 19행 : 우리가 작성한 새로운 시스템 콜을 호출
- ④ 20행~26행 : if 문을 통해 시스템 콜이 정상적으로 수행하면 자신의 PID 값과 부모 프로세스의 PID 값을 출력
 - 시스템 콜이 존재하지 않는 경우 에러 메시지를 출력
 - 모듈을 적재하지 않았거나 적재된 모듈을 제거한 후 응용 프로그램을 수행하면 "Error" 문자열을 출력할 것

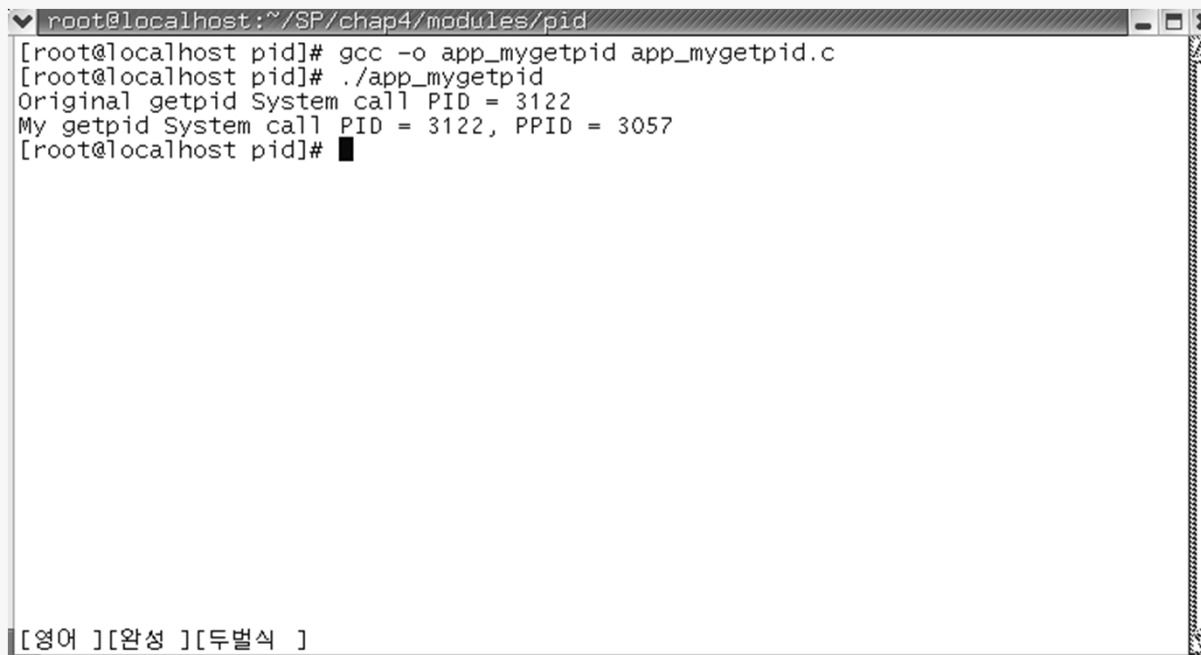
(5) 실행 결과 확인

```
# gcc -o app_addcall app_addcall.c  
# ./app_addcall
```

[실습 4-4] 모듈을 이용한 시스템 콜의 구현

(5) 실행 결과 확인

```
#gcc -o app_mygetpid app_mygetpid.c  
#./app_mygetpid
```



```
root@localhost:~/SP/chap4/modules/pid  
[root@localhost pid]# gcc -o app_mygetpid app_mygetpid.c  
[root@localhost pid]# ./app_mygetpid  
Original getpid System call PID = 3122  
My getpid System call PID = 3122, PPID = 3057  
[root@localhost pid]#
```

[영어] [완성] [두벌식]

(6) rmmod로 모듈을 해제하고 다시 응용 프로그램을 수행한 후 결과 확인

Section 04 make 유틸리티

● 필요성

- ▶ 소스 파일의 개수가 많고, 파일 간의 함수 참조가 있을 경우
 - 컴파일하지 않아도 될 것을 컴파일할 수 있음
 - 컴파일해야 할 것을 하지 못하는 경우 발생
- ⇒ make 프로그램을 사용하여 관계있는 것만 갱신

● make 명령의 기본 사용법

```
make [-f makefile]
```

● Makefile의 기본 구조

```
target [file name] : dependency [file names]
[tab]command
[tab]command
.....
```

Section 04 make 유틸리티

● 활용 예

- ▶ main.c, read.c, write.c 파일을 컴파일/링크하여 실행 파일을 얻기

```
test : main.o read.o write.o
        gcc -o test main.o read.o write.o
main.o : io.h main.c
        gcc -c main.c
read.o : io.h read.c
        gcc -c read.c
write.o : io.h write.c
        gcc -c write.c
```

- ▶ 레이블을 사용하여 작성한 Makefile의 예

```
test : main.o read.o write.o
        gcc -o test main.o read.o write.o
main.o : io.h main.c
        gcc -c main.c
read.o : io.h read.c
        gcc -c read.c
write.o : io.h write.c
        gcc -c write.c
```


Section 04 make 유틸리티

● 매크로

- ▶ 복잡하고 반복되는 일련의 코드를 단순하게 표현함
 - 매크로를 정의한 후 실제 사용 시에는 \$(매크로 이름)와 같이 기술

▶ 종류

- 사용자가 자유롭게 정의해 사용할 수 있는 매크로
- 시스템에서 미리 정한 매크로(Pre-defined Macro)

ASFLAGS = #as(어셈블러) 명령어의 옵션 세팅

AS = as

CFLAGS = #cc 혹은 gcc(컴파일러) 명령어의 옵션 세팅

CC = cc (=gcc)

CPPFLAGS = #g++(컴파일러) 명령어의 옵션 세팅

CXX = g++

LDFLAGS = #ld(로더) 명령어의 옵션 세팅

LD = ld

LFLAGS = #lex(Parser 생성기) 명령어의 옵션 세팅

MAKE_COMMAND = make

Section 04 make 유틸리티

[표 4-2] 내부 매크로 심볼

내부 매크로 심볼	기능
\$*	확장자가 없는 현재의 타겟 파일
\$@	현재의 타겟 파일
\$<	현재의 타겟 파일보다 더 최근에 갱신된 파일명
\$?	현재의 타겟 파일보다 더 최근에 갱신된 파일명
\$^	디폴트 소스 파일들(확장자가 .c인 파일들)



Thank you