Karadeniz Teknik Üniversitesi
Mühendislik Fakültesi
Bilgisayar Mühendisliği Bölümü
Bilgisayar Sistemleri Laboratuarı

İŞLETİM SİSTEMİ ÇEVRELERİNDE SÜREÇ GÖZETLEME

Süreç gözetleme bir sistemin servisleri üzerindeki yerel ve harici saldırıların tespiti, veri kayıplarının önlenmesi ve program davranışlarını belirleme gibi alanlarda kullanılır. Bu föyde Unix işletim sistemi çevreleri için bir süreç gözetleme uygulamasının nasıl geliştirileceği anlatılmıştır.

1. Süreç Gözetleme Mekanizmaları

Unix işletim sistemlerinde iki çeşit süreç gözetleme mekanizması vardır; **`ptrace`** sistem çağrısı ve **`/proc`** dosya sistemi. Bu mekanizmalar, bir sürecin icrasının başlangıcından sonuna kadar gerçekleştirdiği aktivitelerin gözetlenebilmesine izin verir. Gözetlenen süreç adım adım koşturulabilir, bellek alanı okunup değiştirilebilir ve sistem tarafından tutulan süreç tablosuna erişilebilir.

Süreç gözetleme mekanizmaları, bir sürecin diğer bir sürecin sistem kaynaklarını kullanımını kolay bir şekilde takip edebilmesini sağlar. Bu nedenle, bir süreç gözetleme işlemi en az iki süreç içerir; gözetleyen ve gözetlenen süreç. Örneğin, bir hata ayıklayıcı (debugger) süreç gözetlenecek bir süreç oluşturur ve **`ptrace`** yada **`/proc`** mekanizması ile sürecin durma noktalarını (breakpoints) setler yada resetler ve adres alanındaki veriyi okur. Gözetlenen süreçler bazen yeni süreçler (child processes) oluştururlar. Bu süreçlerin gözetlenmesi, onların herbiri için yeni gözetleyen süreçler üreterek yapılabileceği gibi sadece bir gözetleyen süreç ile de yapılabilir.

Süreç gözetleme mekanizmaları birbirinin alternatifi gibi görünse de, **`ptrace`** ile karşılaştırıldığında, **`/proc`** mekanizması daha gelişmiş özelliklerle donatılmıştır:

- Süreç gözetleme daha hızlıdır ve performansı çok az etkiler,
- Süreçler eşzamanlı olarak gözetlenebilir,
- Çalışır durumda olan süreçleri gözetlemek mümkündür,
- Sistem çağrıları bireysel olarak gözetlenebilir.

Birçok Unix programı (`gdb`, `ps`, `top` ve `truss` gibi) süreçleri kontrol etmek ve onlara ilişkin bilgi toplamak için `/proc` dosya sistemini kullanır.

2. `/proc` Dosya Sistemi

Süreç dosya sistemi, işletim sistemindeki her sürecin adres alanına erişimi sağlar. Sistemde oluşturulan bütün süreçler */proc* dizini altında temsili dosyalar şeklinde tutulurlar. */proc* dizinindeki herbir dosya, ilgili sürecin ID'si ile isimlendirilmiştir ve herbir dosyanın sahibi sürecin gerçek kullanıcı (real user) ID'si ile belirlenir. Dosyayı açma izinleri genel dosya sistemi izinlerinden daha kısıtlıdır.

Standart sistem çağrısı arayüzü */proc* dosyalarına erişim için kullanılır: `open()`, `lseek()`, *`read`*`()`, *`write`*`()` ve *`ioctl`*`()`. Örneğin, `lseek()` çağrısı ile sürecin adres alanına erişilir ve `read()` yada `write()` çağrıları ile de bu alandan veri okunur yada yazılır. Kontrol işlemleri `ioctl()` çağrıları ile yapılır. Bir süreç için doğal kontrol noktası, icra kontrolünün işletim sistemi çekirdeğine verildiği ve geri alındığı noktadır. Diğer bir ifadeyle, gözetlenen sürecin sistem çağrısı yaptığı ve bu çağrıdan geri dönüldüğü noktalarda onun adres alanına erişilir. Adres alanındaki okuma ve yazma işlemleri sadece gözetlenen sürecin icrası durdurulduktan sonra gerçekleştirilebilir. Gözetlenen sürecin durdurulması yada icrasının başlatılması işletim sisteminin kontrolündedir.

Bir süreç gözetlenmeye başlanmadan önce onun hangi sistem aktiviteleri (sistem çağrıları, sinyaller, bazı donanım hataları) ile ilgileniliyor ise bu aktiviteler işletim sistemine gözetlemeyi yapan süreç tarafından bildirilir. Gözetlenen süreç bu aktivitelerden birini gerçekleştireceği zaman işletim sistemi tarafından durdurulur ve gözetleyen sürece haber verilir. İşletim sistemi bir süreci, sistem çağrısı argümanlarını almadan önce ve çağrının geri döndürdüğü değerleri veri alanlarına (işlemci kaydedicileri yada adres alanları) yazdıktan sonra durdurur. Bu durma noktaları, sistem çağrısı argümanlarını ve geri dönüş değerlerini öğrenmeye fırsat verir.

3. Sistem Çağrısı Arayüzü

Kullanıcı programlarının işletim sistemi ile etkileşimi genellikle kütüphane arayüzü içerisinden (fonksiyon çağrıları ile) yapılır. Ancak, C dilinde yazılan programlarda sistem çağrılarını da kullanmak mümkündür. C programları içindeki sistem çağrıları sıradan fonksiyon çağrıları gibi görünürler. Sistem çağrısı arayüzü işletim sisteminin bir parçasıdır ve herbir sistem çağrısına bir çekirdek yordamı karşılık gelir.

Yandaki şekilde (Şekil 1) C dilinde yazılan programların işletim sistemi ile etkileşimi gösterilmiştir. Bu programlar, sistem çağrısı arayüzünü hem doğrudan hem de kütüphaneler içerisinden kullanabilirler. Örneğin, bir kullanıcıya

```
char *mesg = "Unix sistemlerinde süreçler
sadece fork() çağrısı ile oluşturulabilir.";
```

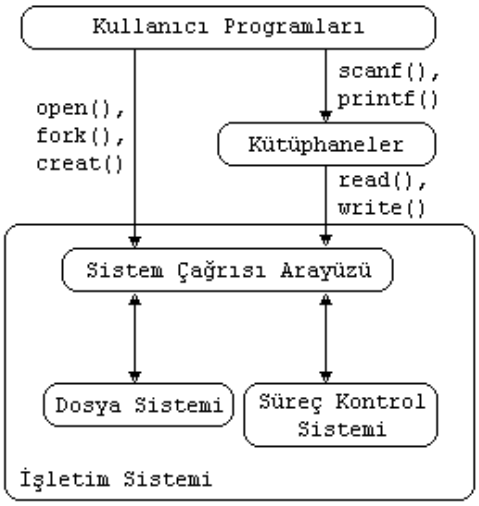
mesajı, bir kütüphane fonksiyonu ile

```
printf("%s\n", mesg);
```

yada sistem çağrılarını kullanarak

```
write(1, mesg, 69);
write(1, "\n", 1);
```

biçiminde gösterilebilir. Dolayısıyla, `printf` fonksiyonu `write` çağrılarının bir dizisinden oluşur. Benzer olarak, `scanf` fonksiyonu da `read` çağrılarının bir dizisini içerir.



Şekil 1: Program icra bileşenleri ve mimarisi

Sistem çağrıları, ilgili oldukları sistem kaynaklarına göre 4 gruba ayrılabilir; süreç yönetimi çağrıları (`execve`, `fork`, `pipe`), bellek yönetimi çağrıları (`mmap`, `brk`, `sync`), dosya sistemi çağrıları (`open`, `chdir`, `stat`) ve diğer çağrılar (`ioctl`, `time`, `sysconfig`). *Süreç yönetimi çağrıları* bir programın görüntüsünü çalışan bir sürece sarma (`execve`), süreç oluşturma (`fork`) ve süreçler arası haberleşmeyi sağlama (`pipe`) gibi işlevlere sahiptirler. *Bellek yönetimi çağrıları* bir dosyanın görüntüsünü diskten belleğe alma (`mmap`), sürece tahsis edilen belleğin boyutunu kontrol etme (`brk`) ve süper bloğu güncelleme gibi işlemleri yerine getirirler. *Dosya sistemi çağrıları* dosyaları erişim için açma (`open`), dizinler arası hareket etme (`chdir`) ve dosya yada dizinler hakkında bilgi alma (`stat`) gibi işlemleri gerçekleştirirler. Diğer sistem çağrılarının işlevleri birbirlerinden oldukça farkeder.

4. `truss` Programı ile Süreçleri Gözetleme

Unix programlarının hangi sistem çağrılarını yaptıkları, bu programları icra eden süreçler gözetlenerek belirlenebilir. Süreç gözetlemenin tipik bir örneği `truss` programının kullanımıdır. Bu program, girilen bir komutun icra edilirken gerçekleştirdiği bütün sistem aktivitelerini gösterir. Aşağıda, `date` komutu icra edilirken `truss` komutunun yakaladığı sistem çağrılarının birkaçı gösterilmiştir.

```
$ truss date
execve("/usr/bin/date", 0x08047D40, 0x08047D48)  argc = 1
sysconfig(_CONFIG_PAGESIZE)                      = 4096
mmap(0x00000000, 140, PROT_READ, MAP_SHARED, 3, 0) = 0xDDBB0000
open("/usr/lib/libc.so.1", O_RDONLY)             = 3
resolvepath("/usr/lib/libc.so.1", "/usr/lib/libc.so.1", 1023) = 18
close(3)                                         = 0
brk(0x080626E0)                                  = 0
time()                                           = 1089118722
open("/usr/share/lib/zoneinfo/Turkey", O_RDONLY) = 3
read(3, " T Z i f\0\0\0\0\0\0\0".., 993)         = 993
close(3)                                         = 0
ioctl(1, TCGETA, 0x08047BE0)                     = 0
fstat64(1, 0x08047B50)                           = 0
Tue Jul  6 15:58:42 EEST 2004
write(1, " T u e   J u l     6   1".., 30)       = 30
_exit(0)
```

Yukarıdaki çıktıdan görüldüğü gibi, bir komutun icrası `execve` çağrısı ile başlar ve `_exit` çağrısı ile sona erer. Bu iki çağrı arasında bulunan diğer çağrılar, komutların yada programların sistem kaynaklarını hangi ölçüde kullandıklarını gösterirler. Bir komutun yaptığı sistem çağrılarının sayısı, onun kaynak kodundaki görünür ifadelerle sınırlı değildir. Kütüphane dosyalarının açılıp okunması ve komutu koşacak sürecin yönetimi gibi gereksinmeleri karşılamak için gerekli bazı kod parçaları da komutun derlenmesi esnasında icra edilebilir koda dahil edilir. Bu kod parçaları birçok sistem çağrısının yapılmasına ihtiyaç duyar.

5. Süreç Gözetleme Programının Geliştirilmesi

Bu bölümde `truss`'a benzer bir programın (`tracer`) yazılımı anlatılmıştır. Programın ana kaynak kodu aşağıda verilmiştir. Kaynak kod içerisinden sistem çağrıları için hata kontrolü yapan ifadeler çıkarılmıştır.

usage prosedürü `tracer` programının komut satırından nasıl kullanılacağını gösterir. `tracer` programına bir komut yada çalışmakta olan bir sürecin `PID`'si argüman olarak verilebilir.

```
void usage(void)
{
  printf("%s\n%s\n",
         "usage: tracer -p <pid>",
         "       tracer <command> [args]");
  exit(1);
}
```

initTraceFlags prosedürü gözetlenen süreç tarafından yapılan hangi sistem çağrılarının işletim sistemi çekirdeğine girişte ve çıkışta durdurulmasının istendiğini işletim sistemine bildirmek için kullanılır. Prosedür içinde sistemde mevcut bütün çağrıların çekirdeğe girişte ve çıkışta durdurulması istenmiştir.

```
void initTraceFlags(int proc_fd) {
  int sysnum;
  sysset_t traced_calls;

  premptyset (&traced_calls);

  for (sysnum = 0; sysnum < 254; sysnum++)
    praddset (&traced_calls, sysnum);     /* All system calls to trace */

  ioctl(proc_fd, PIOCSENTRY, &traced_calls);
  ioctl(proc_fd, PIOCSEXIT, &traced_calls);
}
```

Komut satırından girilen komut gözetlenen süreç tarafından icra edilmeye başlanır. İcranın başlangıcı gözetleyen sürecin gerekli bildirimleri işletim sistemine yapmasından hemen sonra olmalıdır.

```
char **command;
void handler_cmd() {
  execvp (command[0], command);
}
```

Gözetleyen süreç (`tracer`) icraya başlar ve komut satırından girilen veriyi inceleyerek icra edilecek komutu yada çalışmakta olan sürecin PID'sini belirler. `tracer` programına herhangi bir argüman geçilmemiş ise programın kullanımına ilişkin bilgi gösterir.

```
main (int argc, char *argv[]) {
  char pid_s[15], sysname[50];
  prstatus_t status;
  struct syscall *sc;
  int c, i, fd, sysnum, pid=0, child_pid=0;

  while ((c = getopt(argc, argv, "p:o")) != -1) {
    switch (c) {
    case 'p':               /* Specified pid */
      pid = atoi(optarg);   /* optarg = 1234 for "a.out -p 1234". */
      break;
    default:
      usage();
    }
```

```
    }

    argc -= optind; argv += optind;
    if ((pid == 0 && argc == 0) || (pid != 0 && argc != 0))
      usage();
```

Gözetlenen süreç oluşturulur ve bu süreç içinde, gözetleyen süreç (parent process) sistem çağrısı bildirimlerini yapıncaya kadar beklenir. Bildirimlerin yapıldığı gözetleyen süreçten alınan `SIGCHLD` sinyali ile anlaşılır ve `handle_cmd` prosedürü ile komut icra edilir.

```
    if (pid == 0) {
      child_pid = 1;
      if ((pid = fork()) == 0) {
        command = argv;
        signal(SIGCHLD, handler_cmd);
        for(;;) ;
      }
    }
```

Komut satırından `PID`'si girilen yada girilen bir komutu icra etmek için oluşturulan süreç ile ilgili olarak işletim sistemine bildirimlerde bulunulur. Bu bildirimlerden sonra, gözetlenecek süreç yeni oluşturulmuş ise ona `SIGCHLD` sinyali gönderilerek icrasını başlatması istenir.

```
    sprintf (pid_s, "/proc/%d", pid);
    fd = open (pid_s, O_RDWR | O_EXCL);
    initTraceFlags(fd);
    if (child_pid)
      kill(pid, SIGCHLD);
```

Bir süreç durma eylemi ile karşılaşıldığı zaman yapılan sistem çağrısının adı, argümanları yada geri dönüş değerleri gösterilir ve sürecin icrası durduğu noktadan itibaren yeniden başlatılır.

```
    while (1) {
      ioctl (fd, PIOCWSTOP, &status);
      sysnum = status.pr_syscall;
      sprintf(sysname, "%s", syscallnames[sysnum]);
      sc = getSyscall(sysname);
      if (sc) {
        if (status.pr_why == PR_SYSENTRY) {
          printf("%s(", sc->name);
          for (i = 0; i < sc->nargs; i++)
            printf("%s, ", getArg(fd, sc->args_type[i], status.pr_sysarg[i]));
          if (sc->nargs > 0)
            printf("\b\b");
          printf(")");

          if (status.pr_what == SYS_exit)
            break;
        }
        else {
          if (sc->ret_type == Ptr)
            printf(" = 0x%x\n", status.pr_reg[R_R0]);
          else
            printf(" = %d\n", status.pr_reg[R_R0]);
        }
      }

      ioctl (fd, PIOCRUN, 0);
    }

    ioctl (fd, PIOCRUN, 0);
  }
```

Bu programda kullanılan `syscallnames` dizisinin elemanları sistem çağrılarının isimlerinden oluşur. Bu dizi içerisinde herbir elemanın bulunduğu index numarası, Unix sistemi tarafından o sistem çağrısına tahsis edilen numaraya karşılık gelir. Örneğin, `open` sistem çağrısı `5` sayısı ile temsil edilir.

```
  char *syscallnames[] = {
        "syscall", "exit", "fork", "read", "write", "open", "close", "wait",
        "creat", "link", "unlink", "exec", "chdir", "time", "mknod", "chmod",
        "chown", "brk", "stat", "lseek", "getpid", "mount", ... }
```

getSyscall ve **getArg** kullanıcı tanımlı fonksiyonlardır. **getSyscall**, *sysname* ile belirtilen sistem çağrısının yapısını aşağıda birkaç elemanı verilen `syscalls` dizisinden alır. Sistem çağrılarının yapısı birbirinden farklı olduğundan dolayı, ilgilenilen herbir sistem çağrısı ile ilgili bilgi bu dizi içinde tutulur. **getArg** fonksiyonu ise sistem çağrısının argüman değerlerini öğrenir.

```
struct syscall
syscalls[] = {
        { "mmap",    Ptr,  6,  { Hex, Int, Hex, Hex, Int, Quad }},
        { "open64",  Int,  3,  { String, Int, Octal }},
        { "open",    Int,  3,  { String, Int, Octal }},
        { "execve",  Int,  3,  { String, Hex, Hex }},
        { "ioctl",   Int,  3,  { Int, Ioctl, Hex }},
        { "lseek",   Int,  3,  { Int, Quad, Int }},
        { "link",    Int,  2,  { String, String }},
        { "rename",  Int,  2,  { String, String }},
        { "read",    Int,  3,  { Int, Ptr, Int }},
        { "write",   Int,  3,  { Int, Ptr, Int }},
        { "stat",    Int,  2,  { String, Ptr }},
        { "creat",   Int,  2,  { String, Hex }},
        { "chmod",   Int,  2,  { String, Hex }},
        { "unlink",  Int,  1,  { String }},
        { "chdir",   Int,  1,  { String }},
        { "close",   Int,  1,  { Int }},
        { "exit",    None, 1,  { Int }},
        { "fork",    Int,  0,  { }},
        { 0,         0,    0,  { 0 }}
};
```

Bu dizinin herbir elemanı ilgili sistem çağrısının ismini, geri dönüş değerini, argümanlarının sayısını ve bu argümanların (`types` veri tipi tanımlanan) tiplerini içerir. Örneğin, `open` çağrısının geri döndürdüğü değer `Int` tipinden, argüman sayısı 3 ve argümanlarının tipleri sırasıyla `String, Int, Octal`'dır. Dolayısıyla yukarıda verilen dizinin elemanları aşağıdaki veri tipine sahiptir.

```
  typedef enum {None=1, Hex, Octal, Int, String, Ptr, Stat, Ioctl, Quad} types;

  struct syscall {
      char   *name;              /* çağrının ismi */
      types  ret_type;           /* geri dönüş değeri */
      int    nargs;              /* argümanların sayısı */
      types  args_type[10];      /* herbir argümanın tipi */
  };
```

Yukarıda geliştirilen program `truss`'dan biraz farklı çalışır. Dinamik olarak oluşturulan süreçler `tracer` programı tarafından gözetlenmezler. Bu çeşit süreçlerin gözetlenmesi ya her biri için yeni bir gözetleyici süreç üreterek yada `poll` sistem çağrısı kullanılarak tek bir gözetleyici süreç ile yapılabilir.

```
int waitForAProcessToStopOrDie(struct pollfd *pollfds, int n_fds, int *pindex)
{
  int i, n;

  while (1) {
    n = poll(pollfds, n_fds, INFTIM);
    if (n > 0) {
      /* Check for an event of interest (STOPPED, DIED) */
      for (i = 0; i < n_fds; i++) {
        if (pollfds[i].revents & POLLPRI) {       /* A process has stopped */
          *pindex = i;
          return 0;
        }
        else if (pollfds[i].revents & POLLHUP) {  /* A process has died */
          *pindex = i;
```

```
          return 1;
        }
      }
    }
  }
  return -1;
}
```

Yukarıdaki prosedürde `poll` çağrısı bir süreç durdurulduğu zaman geri döner ve sonra hangi sürecin niçin durduğu belirlenir.

6. Deney Hazırlığı

1. Deneyin uygulaması `ktuce.ktu.edu.tr` adresli Unix makinesine PuTTY programı ile uzaktan bağlanılarak yapılacaktır. Bu nedenle Unix sistemlerindeki kullanıcı hesapları kavramı ile `http://ktuce.ktu.edu.tr/programs` adresinden indirilebilen PuTTY programını öğreniniz.
2. Unix sistemine bağlanılarak `truss` programının kullanımı öğreniniz.
   `$> man truss`
3. Unix sistem çağrıları ve bu çağrıların kütüphane fonksiyonlarıyla ilişkilerini inceleyiniz. Sistem çağrılarının bir listesi `/usr/include/sys/syscall.h` dosyasında yer almaktadır.
4. Sistem komutları ile shell komutları (built-in komutları) arasındaki farkı kavrayınız.
5. Süreç gözetlemede kullanılan `/proc` dosya sistemini araştırınız ve sistemde çalışan bütün süreçlerin temsili birer dizin olarak tutulduğu `/proc` dizinini inceleyiniz.
   `$> man –s 4 proc`
6. `/proc` dosya sisteminde bulunan `premptyset`, `praddset` gibi fonksiyonları ve ***`pstatus`*** veri yapısını inceleyiniz.

7. Deney Tasarımı ve Uygulaması

1. Süreç gözetleyiciler/izleyiciler bir süreç ile işletim sistemi arasına yerleştirilen ileri düzeyde yetkilerle donatılmış sistem programlarıdır. Bir süreç gözetleme programının (`truss` gibi) hangi işlevleri karşılaması gerektiğini belirleyiniz ve bu işlevleri karşılayacak mimarisel yapıyı bileşenleri ile birlikte tasarlayınız.
2. Bu mimarisel yapının Şekil 1'deki mimariye nasıl entegre edileceğini gösteriniz.
3. PuTTY programı ile iki terminal penceresi (T1 ve T2 terminalleri) açın ve her iki terminal içerisinden de `ktuce.ktu.edu.tr` adresine sahip Unix makinesindeki `labs` isimli kullanıcı hesabına bağlanın.
4. T1 terminalinden `truss` programı ile birkaç Unix komutunun (`ls`, `pwd`, `date` gibi) icrasını gözetleyin. `truss` programının gösterdiği listenin sonuna doğru yer alan sistem çağrıları (`write` gibi) koşulan komutun ana işlevini temsil eder.
   `$> truss ls`
5. `truss` programı ve koşulan Unix komutunun üreteceği çıktılar aynı PuTTY terminali üzerinde görünecektir. Bunlar arasından Unix komutunun ürettiği çıktı satırlarını belirlemeye çalışın.
6. `truss` programının çıktısı T1 terminali içinde görünürken, Unix komutunun çıktısı T2 terminaline yönlendirilebilir. Bunun için T2 terminalinde
   `$> tty`
   komutu ile terminal ismini öğrenin ve T1 içinde bu isme yönlendirme yapın.
   `$> truss ls > T2_terminalinin_ismi`
7. T2 teminalinde bu terminali kontrol eden komut yorumlayıcının (shell, SH2 süreci) PID'sini belirleyin.
   `$> echo $$`
8. T1 terminali içinde `truss` programını SH2 sürecini gözetleyecek şekilde çalıştırın.
   `$> truss –p` *SH2'nin_PID'si*
9. T2 terminalinden Unix komutları girilirken ve icra edilirken T1 terminalinde listelenen sistem çağrılarını ve verilerini değerlendirin.
10. `truss` programı normal kullanımda sadece ana süreci gözetler, çocuk süreçler ile ilgilenmez. Bu nedenle T1 içinde `truss` programının icrasını `Ctrl-C` ile durdurarak çocuk süreçleri de gözetleyecek şekilde yeniden çalıştırın.
    `$> truss –f –p` *SH2'nin_PID'si*
11. Adım 7'yi tekrarlayın ve farkı gözlemleyin; sistem çağrıları süreçlerin PID'leri ile etiketlenmiştir.
12. T2 içinde daha karmaşık (birçok süreç oluşturan) komutlar icra edin ve T1 içinde listelenen sistem çağrılarını inceleyin. `fork` ve `pipe` çağrılarının sayısına dikkat edin.
    `$> ls -al | egrep "\.c"`

```
$> ls -al | sort | wc
$> date & sleep 5
```

Komutların ürettiği sistem çağrılarının hepsi T1 üzerinde görülemiyorsa, `truss` komutunu aşağıdaki gibi çalıştırın ve üretilen çıktının bir dosyaya yazılmasını sağlayın. Sonra `truss` komutunu durdurarak `pico` ile dosyanın içeriğini inceleyin.

```
$> truss -o dosya_ismi -f -p SH2'nin_PID'si
```

13. Şimdi `truss` programının basit bir sürümü olan `tracer` programı incelenecektir. Bu programa ait dosyaları aşağıdaki komutlar ile kendi home dizininiz altına kopyalayın ve çalışma dizinini değiştirin.
```
$> cp -r /tracer ./
$> cd tracer
```
14. `tracer` kaynak kodunu inceleyerek mimarisel bileşenlerin nasıl kodlandığını belirleyiniz.
15. `tracer` programı 4 kaynak dosyadan oluşur: `tracer.c`, `handler.c`, `handler.h` ve `syscallnames.c`. Bu dosyaların içeriğini görmek için `more` yada `cat` komutlarını ve değiştirmek için `pico` komutunu kullanın.
```
$> more/pico tracer.c
```
16. `syscallnames.c` dosyasında bulunan ***`syscallnames`*** dizisindeki çağrıların sırası ile sistemde yer alan `/usr/include/sys/syscall.h` dosyasındaki çağrı sırasını inceleyin.
17. `handler.h` dosyasında tanımlanan ***`types`*** sıralı tipini ve ***`syscall`*** veri yapısını inceleyin.
18. `handler.c` dosyasında verilen ***`syscalls`*** dizisini ve ***`getSyscall`*** ile ***`getArg`*** fonksiyonlarını inceleyin. ***`syscalls`*** dizisi bazı sistem çağrılarının tanımlamalarını içerir.
19. `tracer.c` dosyasındaki `tracer` programının ana bloğunu (`while` ifadesi) inceleyin.
20. `tracer` programını inşa etmek için `make` komutunu kullanın ve bazı Unix komutlarını bu program ile gözetleyin.
```
$> make
$> tracer df -k
```
21. `tracer` programı bir komutun yaptığı bütün sistem çağrılarını gözetlemez. Gözetlenecek her bir sistem çağrısı için `handler.c` içindeki ***`syscalls`*** dizisine yeni bir tanımlama eklenmelidir. `time` ve `getrlimit` çağrılarının tanımlamalarını aşağıdaki gibi ekleyin.

Önce `man` komutu ile bu çağrıların yapısını (parametre ve geri dönüş tipleri) belirleyin.
```
$> man -s 2 time          → time_t time(time_t *tloc);
$> man -s 2 getrlimit     → int getrlimit(int resource, struct rlimit *rlp);
```
Sonra aşağıdaki satırları `pico` komutu yardımıyla syscalls dizisine yerleştirin.
```
{ "time", Int, 1, { Ptr }}
{ "getrlimit", Int, 2, { Int,  Ptr }}
```
22. `make` komutu ile `tracer` programını yeniden inşa ederek `ls` komutunu gözetleyin ve `time` ile `getrlimit` çağrılarının yakalanıp yakalanmadığını kontrol edin.
```
$> make
$> tracer ls
```
23. Kütüphane fonksiyonları ile küçük bir C programı (`prog.c`) yazınız ve bu programı `tracer` ile gözetleyiniz.
```
$> gcc -o prog prog.c
$> tracer prog
```
24. Sistem çağrılarını kullanarak küçük bir C programı (`prog2.c`) yazınız ve bu programı `tracer` ile gözetleyiniz.
```
$> gcc -o prog2 prog2.c
$> tracer prog2
```
25. `truss` ile olduğu gibi `tracer` programı ile de bir shell sürecini gözetlemeyi deneyin.
```
$> tracer -p Shell_PID
```

8. Deney Soruları

1. `truss` programı girilen komut ile nasıl ilgilenir? Bütün komut icralarının `execve` sistem çağrısı ile başlatıldığına dikkat ediniz.
2. Bir shell süreci `truss` programı ile gözetlenmeye başlandığında `read` çağrısı niçin gösterilir? Yukarıdaki Adım 6'yi dikkate alınız.
3. Bir Unix shell programı sistem komutlarının icrasını nasıl başlatır? Yukarıdaki Adım 7'de T1 içinde görünen `fork` çağrısını dikkate alınız.
4. T2 terminalinde sistem komutları icra edilirken T1 terminalinde `execve` çağrısı niçin görünmez? Yukarıdaki Adım 7'yi dikkate alınız.
5. T2 terminalinde shell komutları (örneğin, `echo`, `pwd`, `cd` gibi) icra edilirken T1 terminalinde `fork` çağrısı niçin görünmez?

6. `syscallnames.c` ile `/usr/include/sys/syscall.h` dosyaları içindeki sistem çağrıları sırasının aynı olmasının nedenini açıklayınız. Unix sistemlerinde çağrıların birer numara ile temsil edildiğine dikkat ediniz.
7. `handler.h` dosyasındaki *`types`* sıralı tipi niçin tanımlanmıştır?
8. `tracer` programında gözetleyen ve gözetlenen süreçler arasında `kill-signal` mekanizması niçin kullanılmıştır?
9. `tracer` programı ile `handler.c` içinde tanımlaması verilmeyen sistem çağrıları niçin gözetlenemez?
10. `tracer.c` dosyasında yer alan aşağıdaki `if` bloğuna niçin ihtiyaç duyulur? Bu bloğun olmaması `tracer` programının çalışmasını nasıl etkiler?

```
if (status.pr_what == SYS_exit) {
  printf("\n");
  break;
}
```

11. Aynı yapıya sahip `getdents` ve `getdents64` sistem çağrılarını gözetleyebilmek için handler.c içindeki syscalls dizisine hangi veri eklenmelidir? `getdents64` çağrısı `ls` komutu tarafından dizin içeriğini okumak için kullanılmaktadır.

9. Deney Raporu

1. Süreç gözetlemenin nasıl yapıldığını kısaca anlatınız.
2. Gözetleyici programını bileşenleri ile tasarlayınız.
3. Şekil 1'deki mimariyi gözetleyici programını da kapsayacak şekilde geliştiriniz.
4. `truss` ile `tracer` arasındaki en önemli işlev farklılıklarını listeleyiniz.
5. Deney esnasında koşulan komutlara karşı yakalanan sistem çağrılarını yorumlayınız.
6. Deney sorularını kısaca cevaplandırınız.