```
#include <signal.h>
#include <sys/time.h>

/* signal handlers */
rtalrm() {
  printf("Got a SIGALRM\n");
}
valrm() {
  printf("Got a SIGVTALRM\n");
}
palrm() {
  printf("Got a SIGPROF\n");
}

struct itimerval rtimer,vtimer,ptimer;

int x;
main() {
  /* set up signal handlers */
  (*signal(SIGALRM,rtalrm))();
  (*signal(SIGVTALRM,valrm))();
  (*signal(SIGPROF,palrm))();

  /* set up timers */

  rtimer.it_value.tv_sec=10;
  rtimer.it_value.tv_usec=0;
  vtimer.it_value.tv_sec=2;
  vtimer.it_value.tv_usec=0;
  ptimer.it_value.tv_sec=1;
  ptimer.it_value.tv_usec=500000;

  rtimer.it_interval.tv_sec=10;
  rtimer.it_interval.tv_usec=0;
  vtimer.it_interval.tv_sec=2;
  vtimer.it_interval.tv_usec=0;
  ptimer.it_interval.tv_sec=1;
  ptimer.it_interval.tv_usec=500000;

  setitimer(0,&rtimer);
  setitimer(1,&vtimer);
  setitimer(2,&ptimer);

  while(1) {
    x=500%21;
    x=x*2;
    x=x/3;
  }
}
```

```
/* Last modified on Mon Nov 11 12:22:50 1985 by mcjones */

#include <varargs.h>

exit(retcode) int retcode; {
  flush();
  _exit(retcode);
  }

#define STDOUT 1

char outBuf[132];
int outBufPos = 0;

putchar(c) char c; {
  outBuf[outBufPos] = c;
  outBufPos++;
  if(c=='\n' || outBufPos==sizeof outBuf) {
    write(STDOUT, outBuf, outBufPos);
    outBufPos = 0;
    }
  }

flush() {
  if(outBufPos>0) {
    write(STDOUT, outBuf, outBufPos);
    outBufPos = 0;
    }
  }

printf(va_alist) va_dcl {
  va_list ap;
  char *fmt, c, *s, sc;
  int i, x;
  va_start(ap);
  fmt = va_arg(ap, char *);
  while((c = *fmt)!=0) {
    fmt++;
    if(c!='%') putchar(c);
    else {
      if((c = *fmt++)==0) break;
      switch(c) {
        case 's':
          s = va_arg(ap, char *);
          while((sc = *s++)!=0) putchar(sc);
          break;
        case 'd':
          i = va_arg(ap, int);
          if(i<0) {putchar('-'); i = -i;}
          printu(i, 10);
          break;
        case 'x':
          x = va_arg(ap, int);
          printu(x, 16);
          break;
        default: putchar('%'); putchar(c);
        }
      }
    }
  }

printu(u, r) unsigned int u, r; {
  if(u==0) putchar('0');
  else printupos(u, r);
  }
```

```
char digit[] = "0123456789abcdef";

printupos(u, r) unsigned int u, r; {
  unsigned int d;
  d = u % r;
  u = u / r;
  if(u!=0) printupos(u, r);
  putchar(digit[d]);
  }
```

```
#define BANNER "uxb (last modified on Thu Sep 11 15:46:15 1986 by mcjones)\n\n"

/*********/
/* uxb.c */
/*********/

/* A diagnostic for Ux{,Pro,As,Time}. */

#define FALSE 0

#include <errno.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <setjmp.h>

char *sbrk();

#define INITPID 1 /* init process id */
#define SUPER 0 /* super user id */
#define LASTSIG SIGPROF /* ignore SIGPROF+1 ... NSIG */

main(argc, argv, envp) int argc; char **argv, **envp; {
  printf("\n");
  printf(BANNER);
  TestSignalState();
  EstablishBackstop();
  TestProcessCreation(0); /* using fork */
  TestProcessCreation(1); /* using vfork */
  TestUserID();
  TestProcessGroups();
  TestSignalAction();
  EstablishBackstop();
  TestRUsage();
  TestExec();
  TestVM();
  TestTime();
  TestHostIdentity();
  printf("uxb finished\n");
  exit(0);
  }

Assert(b, s) int b; char *s; {
  if(!b) {
    printf("*** Assertion failed: %s\n", s);
/*
    if(debug) CallDebugger();
    else exit(1);
*/
    }
  }

AssertOk(cc) int cc; {
  Assert(cc==0, "expected 0 result");
  }

extern int errno;

AssertErr(cc, e) int cc, e; {
  Assert(cc==-1, "expected result -1");
  Assert(errno==e, "unexpected errno");
  }
```

```
EqVec(v1, v2) struct sigvec *v1, *v2; {
  return v1->sv_handler==v2->sv_handler
    && v1->sv_mask==v2->sv_mask
    && v1->sv_onstack==v2->sv_onstack;
  }

int sigHandled, codeHandled;
struct sigcontext scpHandled;

HandleTest(sig, code, scp)
  int sig, code; struct sigcontext *scp; {
  sigHandled = sig; codeHandled = code; scpHandled = *scp; /* struct assign */
  }

struct sigvec vecDfl = {SIG_DFL, 0, 0};
struct sigvec vecIgn = {SIG_IGN, 0, 0};
struct sigvec vecHan = {HandleTest, 0, 0};

TestSignalState() {
  int cc, sig;
  unsigned int mask, omask, emask;
  struct sigstack ss, oss;
  struct sigvec ovec;
  char stack; /* JUST A MARKER FOR NOW */

  printf("  TestSignalState\n");

  /* Check mask initialization. */
  printf("    Try sigsetmask\n");
  omask = sigsetmask(0);
  Assert(omask==0, "unexpected mask value");

  /* Try setting nonzero mask. */
  for(sig = 1; sig<=LASTSIG; sig++) {
    mask = 1<<(sig-1);
    if(sig!=SIGKILL && sig!=SIGSTOP && sig!=SIGCONT) emask = mask;
    else emask = 0;
    omask = sigsetmask(mask);
    Assert(omask>=0, "unexpected error");
    omask = sigsetmask(0);
    Assert(omask==emask, "unexpected mask");
    }

  /* Try ORing into mask with sigblock. */
  printf("    Try sigblock\n");  emask = 0;
  for(sig = 1; sig<=LASTSIG; sig++) {
    mask = 1<<(sig-1);
    if(sig!=SIGKILL && sig!=SIGSTOP && sig!=SIGCONT) emask |= mask;
    omask = sigblock(mask);
    Assert(omask>=0, "unexpected error");
    omask = sigblock(0);
    Assert(omask>=0, "unexpected error");
    Assert(omask==emask, "unexpected mask");
    }
  /* Restore zero mask. */
  omask = sigsetmask(0);
  Assert(omask>=0, "unexpected mask from ");

  /* Check signal stack context initialization. */
  printf("    Try sigstack\n");
  cc = sigstack(0, &oss);
  AssertOk(cc);
  Assert(oss.ss_onstack==0 && oss.ss_sp==(char *)0,
    "unexpected signal stack context");
```

```
  /* Try setting signal stack. */
  ss.ss_sp = &stack; ss.ss_onstack = 0;
  cc = sigstack(&ss, 0);
  AssertOk(cc);
  cc = sigstack(0, &oss);
  Assert(oss.ss_sp==&stack && oss.ss_onstack==0,
    "unexpected signal stack context");

  printf("    Try sigvec\n");

  for(sig = 1; sig<=LASTSIG; sig++) {
    cc = sigvec(sig, 0, &ovec);

    /* Can't even query current state for SIGKILL or SIGSTOP. */
    if(sig!=SIGKILL && sig!=SIGSTOP) {
      AssertOk(cc);
if(cc!=0) printf("unexpected %d error from sigvec(%d, 0, &ovec)\n",
errno, sig);
      Assert(EqVec(&ovec, &vecDfl), "unexpected vector");
if(!EqVec(&ovec, &vecDfl))
printf("unexpected vector from sigvec(%d, 0, &ovec)\n", sig);
      }
    else {
      AssertErr(cc, EINVAL);
if(cc!=-1) printf("expected error from sigvec(%d, 0, &ovec)\n", sig);
      }

    /* Try ignoring. */
    cc = sigvec(sig, &vecIgn, 0);
    if(sig!=SIGKILL && sig!=SIGSTOP && sig!=SIGCONT) {
      AssertOk(cc);
if(cc!=0) printf("unexpected %d error from sigvec(%d, &vecIgn, 0)\n",
errno, sig);
      cc = sigvec(sig, 0, &ovec);
      AssertOk(cc);
      Assert(EqVec(&ovec, &vecIgn), "unexpected vector");
if(cc!=0) printf("unexpected %d error from sigvec(%d, 0, &ovec)\n",
errno, sig);
      }
    else {
      AssertErr(cc, EINVAL);
if(cc!=-1) printf("expected error from sigvec(%d, &vecIgn, 0)\n", sig);
      }

    /* Try setting handler. */
    cc = sigvec(sig, &vecHan, 0);
    if(sig!=SIGKILL && sig!=SIGSTOP) {
      AssertOk(cc);
if(cc!=0) printf("unexpected %d error from sigvec(%d, &vecHan, 0)\n",
errno, sig);
      cc = sigvec(sig, 0, &ovec);
      AssertOk(cc);
      Assert(EqVec(&ovec, &vecHan), "unexpected vector");
if(cc!=0) printf("unexpected %d error from sigvec(%d, 0, &ovec)\n",
errno, sig);

      /* Try setting handler with nonzero mask, onstack. */

      }
    else {
      AssertErr(cc, EINVAL);
if(cc!=-1) printf("expected error from sigvec(%d, &vecHan, 0)\n", sig);
      }

    /* Try restoring default. */
    cc = sigvec(sig, &vecDfl, 0);
    if(sig!=SIGKILL && sig!=SIGSTOP) {
      AssertOk(cc);
if(cc!=0) printf("unexpected %d error from sigvec(%d, &vecDfl, 0)\n",
errno, sig);
      cc = sigvec(sig, 0, &ovec);
      AssertOk(cc);
      Assert(EqVec(&ovec, &vecDfl), "unexpected vector");
if(cc!=0) printf("unexpected %d error from sigvec(%d, 0, &ovec)\n",
errno, sig);
      }
    else {
      AssertErr(cc, EINVAL);
if(cc!=-1) printf("expected error from sigvec(%d, &vecDfl, 0)\n", sig);
      }
    }
  } /* TestSignalState */

EstablishBackstop() {
  int sig;
  for(sig = SIGHUP; sig <= SIGPROF; sig++)
    if(sig!=SIGKILL && sig!=SIGSTOP)
      sigvec(sig, &vecDfl, 0);
  sigsetmask(0);
  }

AssertBackstop() { /* check for handler set above */
  int sig; unsigned int omask; struct sigvec ovec;
  for(sig = SIGHUP; sig <= SIGPROF; sig++) {
    sigvec(sig, 0, &ovec);
    if(sig!=SIGKILL && sig!=SIGSTOP)
      Assert(EqVec(&ovec, &vecDfl), "unexpected handler");
    }
  omask = sigblock(0);
  Assert(omask==0, "mask not equal to 0");
  }

int ppid, selfpid; /* initialized in TestProcessCreation */

TestProcessCreation(useVfork) int useVfork; {

  int pid, childpid[3], i;
  union wait ws;

  printf("  TestProcessCreation using ");
  printf(useVfork ? "vfork\n" : "fork\n");

  /* Assertion: process ids are constant, unique. */
  selfpid = getpid();
  pid = getpid();
  Assert(pid==selfpid, "getpid() nonconstant");
  ppid = getppid();
  Assert(INITPID<ppid, "ppid<=INITPID");
  Assert(ppid!=selfpid, "pid=ppid");

  /* *** Also: whenever forking, check selfpid!=childpid[i]!=childpid[j]. */

  /* Try waiting with no children. */
  pid = wait(&ws);
  AssertErr(pid, ECHILD);
  pid = wait3(&ws, WNOHANG, 0);
  AssertErr(pid, ECHILD);

  /* Fork some children. */
  for(i = 0; i<3; i++) {
```

```
    pid = useVfork ? vfork() : fork();
    Assert(pid>=0, "error forking");
    if(pid>0) { /* parent */
/*
printf("child[%d] = %d\n", i, pid);
*/
      childpid[i] = pid;
      }
    else { /* child */
      pid = getppid();
      Assert(pid==selfpid, "unexpected ppid in child");
      pid = getpid();
      Assert(selfpid!=pid, "child pid equals parent pid");
      AssertBackstop();

      /* Should also check signal state, etc. */

      exit(10+i);
      }
    }

  /* Wait for them to complete. */
  for(i = 0; i<3; i++) {
    pid = wait(&ws);
/*
printf("i: %d, pid: %d, ws.w_retcode: %d, childpid[ws.w_retcode-10]: %d\n",
  i, pid, ws.w_retcode, childpid[ws.w_retcode-10]);
*/
    Assert(pid>=0 && ws.w_stopval!=WSTOPPED && ws.w_termsig==0
      && 10<=ws.w_retcode && ws.w_retcode<13 && childpid[ws.w_retcode-10]==pid,
      "unexpected status");
    }

  /* Check that they are all gone. */
  pid = wait(&ws);
  AssertErr(pid, ECHILD);

  } /* TestProcessCreation */

RunAsChild(proc) int (*proc)(); {

  /* NOTE: Uses vfork. */

  int cc;
  union wait ws;
  cc = vfork();
  Assert(cc>=0, "RunAsChild: error forking");
  if (cc > 0) { /* parent */
    Assert(selfpid<cc, "RunAsChild: fork()<=getpid()");
    wait(&ws);
    Assert(ws.w_T.w_Termsig==0 && ws.w_T.w_Retcode==0,
      "RunAsChild: unexpected wait status");
    }
  else { /* child */
    (*proc)();
    exit(0);
    }
  }

int rSrc, eSrc, rTarg, eTarg; /* real/effective source/target */

TrySetreuidCase() {
  int cc, rAct, eAct;

  /* Try to establish source state. */
  cc = setreuid(rSrc, eSrc);
/*
printf("    Current ruid: %d euid: %d...\n", getuid(), geteuid(),
  getuid(), geteuid());
printf("    Establish ruid: %d euid: %d; cc: %d errno: %d\n",
  rSrc, eSrc, cc, errno);
*/
  AssertOk(cc);
  Assert(rSrc==getuid() && eSrc==geteuid(),
    "couldn't establish desired source state for setreuid");

  cc = setreuid(rTarg, eTarg);
  rAct = getuid(); eAct = geteuid();

  if((rTarg==-1 || rTarg==rSrc || rTarg==eSrc || eSrc==SUPER)
      && (eTarg==-1 || eTarg==eSrc || eTarg==rSrc || eSrc==SUPER)) {

    /* Should be allowed. */
    AssertOk(cc);

    /* New value is requested value, except -1 means same as old value. */
    Assert(rTarg == -1 && rAct == rSrc || rTarg != -1 && rAct == rTarg,
      "unexpected real user id");
    Assert(eTarg == -1 && eAct == eSrc || eTarg != -1 && eAct == eTarg,
      "unexpected effective user id");
    }
  else {
    AssertErr(cc, EPERM);

    /* in case of an error, the user id is not changed */
    Assert(rAct == rSrc && eAct == eSrc, "unexpected user id change");
    }
  }

/* The following should be user ids actually appearing in /etc/passwd
   for this to work under Taos. */
#define M 316
#define N 119
#define R 323
#define S 64

struct Pair {int r, e};
struct Pair src[5] = { {0, 0}, {M, 0}, {0, M}, {M, M}, {M, N} };
#define NTARG 6
int targ[NTARG] = { -1, 0, M, N, R, S };

TestUserID() {
  int iSrc, iRTarg, iETarg;

  printf("  TestUserID\n");
  if(geteuid()!=SUPER) {
    printf("    (must be run by super user)\n");
    return;
    }
  for (iSrc = 0; iSrc < 5; iSrc++) {
    rSrc = src[iSrc].r; eSrc = src[iSrc].e;
    for (iRTarg = 0; iRTarg < NTARG; iRTarg++) {
      rTarg = targ[iRTarg];
      for (iETarg = 0; iETarg < NTARG; iETarg++) {
        eTarg = targ[iETarg];
        RunAsChild(TrySetreuidCase);
        }
      }
    }
  }
```

```
int selfpgrp; /* initialized in TestProcessGroups */

TestProcessGroups() { /* get/set */
  int cc, opgrp, pgrp, pgrp2;

  printf("  TestProcessGroups\n");

  selfpgrp = getpgrp(selfpid);

  cc = setpgrp(40000, 0);

  /* Try setting our own process group. */
  for(pgrp = 0; pgrp<100; pgrp++) {
    cc = setpgrp(selfpid, pgrp);
    AssertOk(cc);
    pgrp2 = getpgrp(selfpid);
    Assert(pgrp2==pgrp, "unexpected pgrp");
    }

  /* Try setting our own process group, using zero pid. */
  for(pgrp = 0; pgrp<100; pgrp++) {
    cc = setpgrp(0, pgrp);
    AssertOk(cc);
    pgrp2 = getpgrp(selfpid);
    Assert(pgrp2==pgrp, "unexpected pgrp");
    pgrp2 = getpgrp(0);
    Assert(pgrp2==pgrp, "unexpected pgrp");
    }

  setpgrp(selfpid, selfpgrp);

  } /* TestProcessGroups */

jmp_buf envFPD;

HandleFPD(sig, code, scp)
  int sig, code; struct sigcontext *scp; {
  sigHandled = sig;
  longjmp(envFPD, 1);
  }

struct sigvec vecFPD = {HandleFPD, 0, 0};

TestSignalAction() { /* including wait3(,WNOHANG,) */
  int cc, sig, pid, pid2;
  unsigned int mask;
  struct sigvec ovec;
  union wait ws;
  char *p;

  printf("  TestSignalAction\n");

  printf("    Try signalling self\n");

  for(sig = 1; sig<=LASTSIG; sig++)
    if(sig!=SIGKILL && sig!=SIGSTOP && sig!=SIGCONT) {
      mask = 1<<(sig-1);
      sigvec(sig, 0, &ovec);

      /* Not masked. */
      sigsetmask(0);
      cc = sigvec(sig, &vecHan, 0);
      AssertOk(cc);
if(cc!=0) printf("unexpected %d error from sigvec(%d, %vecHan, 0)\n",
errno, sig);
      sigHandled = 0;
      cc = kill(selfpid, sig);
      AssertOk(cc);
if(cc!=0) printf("unexpected %d error from unmasked kill(selfpid, %d)\n",
errno, sig);
      Assert(sigHandled!=0, "missing signal");
if(sigHandled==0) printf("expected to handle kill(selfpid, %d)\n", sig);
      Assert(sigHandled==sig, "wrong signal");
if(sigHandled!=sig) printf("handled %d after kill(selfpid, %d\n",
sigHandled, sig);

      /* Masked. */
      sigsetmask(mask);
      sigHandled = 0;
      cc = kill(selfpid, sig);
      AssertOk(cc);
if(cc!=0) printf("unexpected %d error from masked kill(selfpid, %d)\n",
errno, sig);
      Assert(sigHandled==0, "signal not masked");
if(sigHandled!=0) printf("handled %d after masked kill(selfpid, %d)\n",
sigHandled, sig);
      sigsetmask(0);
      Assert(sigHandled==sig, "wrong or missing signal");
if(sigHandled!=sig) printf("unexpected %d signal after unmasking kill of %d\n",
sigHandled, sig);

      /* Pausing. */
      sigsetmask(mask);
      sigHandled = 0;
      cc = kill(selfpid, sig);
      AssertOk(cc);
if(cc!=0) printf("unexpected %d error from kill(selfpid, %d) before pause\n",
errno, sig);
      Assert(sigHandled==0, "signal not masked");
if(sigHandled!=0) printf("unexpected %d signal before pausing for %d\n",
sigHandled, sig);
      cc = sigpause(0);
      AssertErr(cc, EINTR);
if(cc!=-1) printf("unexpected result %d from sigpause(0) with blocked %d\n",
cc, sig);
if(cc==-1 && errno!=EINTR)
printf("unexpected errno %d from sigpause(0) with blocked %d\n", errno, sig);
      Assert(sigHandled==sig, "wrong or missing signal");
if(sigHandled!=sig)
printf("unexpected signal %d during sigpause(0) with blocked %d\n",
sigHandled, sig);

      /* To own process group (via kill with zero pid). */
      sigsetmask(0);
      cc = sigvec(sig, &vecHan, 0);
      AssertOk(cc);
if(cc!=0) printf("unexpected %d error from sigvec(%d, %vecHan, 0)\n",
errno, sig);
      sigHandled = 0;
      cc = kill(0, sig);
      AssertOk(cc);
if(cc!=0) printf("unexpected %d error from unmasked kill(0, %d)\n",
errno, sig);
      Assert(sigHandled!=0, "missing signal");
if(sigHandled==0) printf("expected to handle kill(0, %d)\n", sig);
      Assert(sigHandled==sig, "wrong signal");
if(sigHandled!=sig) printf("handled %d after kill(0, %d\n)",
sigHandled, sig);
```

```
	/* To own process group (via killpg). */
	sigsetmask(0);
	cc = sigvec(sig, &vecHan, 0);
	AssertOk(cc);
if(cc!=0) printf("unexpected %d error from sigvec(%d, %vecHan, 0)\n",
errno, sig);
	sigHandled = 0;
	cc = killpg(selfpgrp, sig);
	AssertOk(cc);
if(cc!=0) printf("unexpected %d error from unmasked killpg(selfpgrp, %d)\n",
errno, sig);
	Assert(sigHandled!=0, "missing signal");
if(sigHandled==0) printf("expected to handle killpg(selfpgrp, %d)\n", sig);
	Assert(sigHandled==sig, "wrong signal");
if(sigHandled!=sig) printf("handled %d after killpg(selfpgrp, %d\n)",
sigHandled, sig);

	/* To own process group (via killpg with zero pgrp). */
	sigsetmask(0);
	cc = sigvec(sig, &vecHan, 0);
	AssertOk(cc);
if(cc!=0) printf("unexpected %d error from sigvec(%d, %vecHan, 0)\n",
errno, sig);
	sigHandled = 0;
	cc = killpg(0, sig);
	AssertOk(cc);
if(cc!=0) printf("unexpected %d error from unmasked killpg(0, %d)\n",
errno, sig);
	Assert(sigHandled!=0, "missing signal");
if(sigHandled==0) printf("expected to handle killpg(0, %d)\n", sig);
	Assert(sigHandled==sig, "wrong signal");
if(sigHandled!=sig) printf("handled %d after killpg(0, %d\n)",
sigHandled, sig);

	/* Restore original handler. */
	sigvec(sig, &ovec, 0);
	}

  /* Try catching trap which sets VAX First Part Done flag. */
  sigsetmask(0);
  sigHandled = 0;
  cc = sigvec(SIGSEGV, &vecFPD, 0);
  AssertOk(cc);
  p = sbrk(0);
  switch(setjmp(envFPD)) {
    case 0: bcopy(0, p, 32768); Assert(FALSE, "bcopy didn't trap");
    case 1: ;
    }
  Assert(sigHandled!=0, "missing signal");
if(sigHandled==0) printf("expected to handle SIGSEGV\n");
  Assert(sigHandled==SIGSEGV, "wrong signal");
if(sigHandled!=0 && sigHandled!=SIGSEGV)
  printf("handled %d after address violation\n", sigHandled);
  sigvec(SIGSEGV, &ovec, 0);

  printf("    Try signalling child\n");

  for(sig = SIGHUP; sig<=SIGPROF; sig++)
    if(sig!=SIGKILL && sig!=SIGSTOP && sig!=SIGCONT) {

      /* Mask signal to be raised. */
      mask = 1<<(sig-1);
      sigsetmask(mask);
      sigHandled = 0;
```

```
      pid = fork();
      Assert(pid>=0, "error forking");
      if(pid>0) { /* parent */
        cc = kill(pid, sig);
        AssertOk(cc);
        pid2 = wait(&ws);
        Assert(pid2==pid && WIFEXITED(ws) && ws.w_retcode==0,
          "unexpected status");
        }
      else { /* child */
        cc = sigvec(sig, &vecHan, 0);
        AssertOk(cc);
        Assert(sigHandled==0, "signal not masked");
        cc = sigpause(0);
        AssertErr(cc, EINTR);
        Assert(sigHandled==sig, "wrong or missing signal");
        exit(0);
        }
      }

  /* Try killing process group including children. */

  /* Try forking child with posted masked signal. */

  printf("    Try terminating child\n");

  for(sig = 1; sig<=LASTSIG; sig++)
    if(sig==SIGHUP || sig==SIGINT || sig==SIGKILL || sig==SIGPIPE
      || sig==SIGALRM || sig==SIGTERM || 24<=sig) {

      /* Make sure default (termination) in force. */
      sigvec(sig, &vecDfl, 0);
      sigsetmask(0);

      pid = fork();
      Assert(pid>=0, "error forking");
      if(pid>0) { /* parent */
        cc = kill(pid, sig);
        AssertOk(cc);
        pid2 = wait(&ws);
        Assert(pid2==pid && WIFSIGNALED(ws) && ws.w_termsig==sig,
          "unexpected status");
        }
      else { /* child */
        cc = sigpause(0);
        Assert(FALSE, "child not terminated");
        exit(0);
        }
      }

  printf("    Try restartable kernel call\n");

  sigHandled = 0;
  sigvec(SIGCHLD, &vecIgn, 0); /* don't want to see these now */
  sigvec(SIGTERM, &vecHan, 0);
  sigsetmask(0);
  pid = fork();
  Assert(pid>=0, "error forking");
  if(pid>0) { /* parent */
    pid2 = wait(&ws);
    Assert(pid2==pid && WIFEXITED(ws) && ws.w_retcode==0, "unexpected status");
    Assert(sigHandled==SIGTERM, "missing SIGTERM");
if(sigHandled!=SIGTERM) printf("sigHandled: %d\n", sigHandled);
    pid2 = wait(&ws);
    AssertErr(pid2, ECHILD);
```

```
    sigvec(SIGCHLD, &vecDfl, 0);
    sigvec(SIGTERM, &vecDfl, 0);
    }
  else { /* child */
    kill(getppid(), SIGTERM);
    exit(0);
    }

  printf("    Try stopping child\n");

  for(sig = 1; sig<=LASTSIG; sig++)
    if(sig==SIGSTOP || sig==SIGTSTP || sig==SIGTTIN || sig==SIGTTOU) {
/*
printf("signal: %d\n", sig);
*/
      /* Make sure default (stop) in force for sig; catch SIGCHLD (in
      parent) and SIGCONT (in child). */
      sigvec(sig, &vecDfl, 0);
      sigHandled = 0;
      sigvec(SIGCHLD, &vecHan, 0);
      sigvec(SIGCONT, &vecHan, 0);
      sigsetmask(0);

      pid = fork();
      Assert(pid>=0, "error forking");
      if(pid>0) { /* parent */

        /* The following sleep seems to be necessary: without it, the
        child sometimes pauses forever, as if the SIGCONT is lost. */
        sleep(1);

        /* Stop the child. */
        cc = kill(pid, sig);
if(cc!=0) printf("kill returned %d; errno: %d\n", cc, errno);
        AssertOk(cc);

        /* Wait for it to stop. */
        pid2 = wait3(&ws, WUNTRACED, 0);
        Assert(pid2==pid && WIFSTOPPED(ws) && ws.w_stopsig==sig,
          "unexpected status waiting for child to stop");
        Assert(sigHandled==SIGCHLD,
          "expected SIGCHLD in parent for stop");

        /* Check that it is only reported once. */
        pid2 = wait3(&ws, WUNTRACED | WNOHANG, 0);
        Assert(pid2==0, "wait3 reported stopped child twice");

        /* Start the child again and wait for it to exit. */
        sigHandled = 0;
        cc = kill(pid, SIGCONT);
        AssertOk(cc);
        pid2 = wait(&ws);
        Assert(pid2==pid && WIFEXITED(ws) && ws.w_retcode==0,
          "unexpected status waiting for child to exit");
        Assert(sigHandled==SIGCHLD,
         "expected SIGCHLD in parent for exit");
        }
      else { /* child */
        cc = sigpause(0);
        AssertErr(cc, EINTR);
        Assert(sigHandled==SIGCONT, "child expected SIGCONT");
        exit(0);
        }
      }
```

```
  /* Trying sending to a process group. */

  /* Try broadcasting. */

  /* Try sending an invalid signal number. */
  cc = kill(selfpid, NSIG+1);
  AssertErr(cc, EINVAL);
if (cc!=-1) printf("expected error from kill(selfpid, NSIG+1)\n");

  /* Try sending a signal to a nonexistent process. */
  cc = kill(-2, SIGTERM);
  AssertErr(cc, ESRCH);

  /* Try sending with wrong effective user id. */

  } /* TestSignalAction */

TestRUsage() {
  int cc;
  struct rusage rusage;
  printf("  TestRUsage\n");

  /* Try invalid 'who'. */
  cc = getrusage(1,  &rusage);
  AssertErr(cc,  EINVAL);

  } /* TestRUsage */

TestExec() {
  printf("  TestExec\n");
  } /* TestExec */

jmp_buf envPro;

HandleProbe(sig, code, scp)
  int sig, code; struct sigcontext *scp; {
  longjmp(envPro, 1);
  }

struct sigvec vecPro = {HandleProbe, 0, 0};

char *Probe() {
  struct sigvec ovec;
  char *p, c;
  sigvec(SIGSEGV, &vecPro, &ovec);
  sigHandled = 0;
  switch(setjmp(envPro)) {
    case 0:
      for(p = (char *)0; ; p += 512) {
/*
printf("    Probing %x\n", p);
*/
        c = *p; /* eventually get a SIGSEGV */
        }
    case 1:
      sigvec(SIGSEGV, &ovec, 0);
      return p;
    }
  }

TestVM() { /* sbreak, pagesize, rlimit */
  int cc, pagesize;
  char *brk1, *brk2;

  printf("  TestVM\n");
```