Advanced Signal Handling
Signal Handlers and Reentrant Functions

- Signal handler may be called from within itself
- Can lead to inconsistent results
global list[10]  // will contain signal numbers in received order

handler(sig)

while (list[index] != 0)
    increment index

list[index] = sig
print “exiting handler”()

main()
handler for SIGUSR1 <- handler(sig)
handler for SIGUSR1 <- handler(sig)
handler for SIGTERM <- dump list; exit
while (1==1);
int list[10] = {0,0,0,0,0,0,0,0,0,0};

void handler(int sig) {
    int i = 0;
    while (list[i] != 0) { i++;
    list[i] = sig;
    write(1, "Outta here\n", 11);
}

void dump(int sig) {
    int i;
    for (i = 0; i < 10; i++) {
        printf("list[%d]=%d\n", i, list[i]);
    }
    _exit(0);
}

main() {
    signal(SIGUSR1, handler);
    signal(SIGUSR2, handler);
    signal(SIGTERM, dump);
    printf("Handlers installed\n");
    while (1 == 1);
}
[jmayo@asimov ~]$ ./notRentrant0 &
[1] 18432
[jmayo@asimov ~]$ Handlers installed

[jmayo@asimov ~]$ ps
       PID TTY          TIME CMD
18168 pts/0    00:00:00 tcsh
18432 pts/0    00:00:05 notRentrant0
18433 pts/0    00:00:00 ps
[jmayo@asimov ~]$ kill -USR1 18432
Outta here
[jmayo@asimov ~]$ kill -USR2 18432
Outta here
[jmayo@asimov ~]$ kill -TERM 18432
list[0]=10
list[1]=12
list[2]=0
list[3]=0
list[4]=0
list[5]=0
list[6]=0
list[7]=0
list[8]=0
list[9]=0

[1]  Done
[jmayo@asimov ~]$ ./notRentrant0
int list[10] = {0,0,0,0,0,0,0,0,0,0};

void handler(int sig){
    int i=0;
    while (list[i]!=0){i++;}
    sleep(15);list[i]=sig;
    write(1,"Outta here\n",11);
}

void dump(int sig){
    int i;
    for (i=0;i<10;i++){
        printf("list[%d]=%d\n",i,list[i]);
    }
    _exit(0);
}

main(){
    signal(SIGUSR1,handler);
    signal(SIGUSR2,handler);
    signal(SIGTERM,dump);
    printf("Handlers installed\n");
    while (1==1);
}
```
[jmayo@asimov ~]$ ./notReentrant &
[1] 18343
[jmayo@asimov ~]$ Handlers installed

[jmayo@asimov ~]$ ps
            PID   TTY      TIME CMD
  18168 pts/0  00:00:00 tcsh
  18343 pts/0  00:00:01 notReentrant
  18344 pts/0  00:00:00 ps
[jmayo@asimov ~]$ kill -USR1 18343
[jmayo@asimov ~]$ kill -USR2 18343
[jmayo@asimov ~]$ 
[jmayo@asimov ~]$ Outta here
Outta here

[jmayo@asimov ~]$ kill -TERM 18343
list[0]=10
list[1]=0
list[2]=0
list[3]=0
list[4]=0
list[5]=0
list[6]=0
list[7]=0
list[8]=0
list[9]=0

[1]   Done
[jmayo@asimov ~]$ 
```
Reentrant Functions

• A reentrant function can begin responding to one call, be interrupted by other calls, and complete them all with the same results as if the function had received and executed each call serially.

• POSIX.1 specifies functions that are guaranteed to be reentrant.
  – See “Advanced Programming in the UNIX Environment” or the POSIX standard
  – Handout includes partial listing
  – Notably: NOT malloc() and friends
fork(), exec()

- Signal handling options: catch, default, ignore
- On fork(), child inherits parent’s disposition
  - Address of handler in parent is meaningful
- On exec():
  - Caught signals restored to default
  - Other actions (default, ignore) carry over
fork()

**child?**

- **yes**
  - set SIGINT to ignore
  - printf pid
  - exec sleeper
  - install SIGTERM handler
  - (print TERM message)
  - pause()

- **no**
  - wait for child
  - exit

**pause():** process sleeps until signal

SIGINT at pause in child should be ignored
SIGTERM at pause in child should invoke handler
NAME

pause - wait for signal

SYNOPSIS

#include <unistd.h>
int pause(void);

DESCRIPTION

The pause library function causes the invoking process (or thread) to sleep until a signal is received that either terminates it or causes it to call a signal-catching function.

RETURN VALUE

The pause function only returns when a signal was caught and the signal-catching function returned. In this case pause returns -1, and errno is set to EINTR.

ERRORS

EINTR a signal was caught and the signal-catching function returned.
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>

main(int argc, char **argv){
    int cpid;
    int status;

    cpid=fork();
    if (cpid==0){
        signal(SIGINT,SIG_IGN);
        printf("Child pid <%d>\n",getpid());
        fflush(stdout);
        execlp("/home/csdept/jmayo/sleeper","sleeper");
        _exit(1);
    } else {
        wait(&status);
    }
}
```c
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

void handler()
{
    printf("Sleeper exits on SIGTERM\n");
    fflush(stdout);
}

main()
{
    signal(SIGTERM,handler);
    pause();
}
```
[jmayo@asimov ~]$ ./sigExec
Child pid <23894>
Sleeper exits on SIGTERM
[jmayo@asimov ~]$

[jmayo@asimov ~]$ kill -INT 23894
[jmayo@asimov ~]$ kill -TERM 23894
Blocking Signals

• Process has option to block a signal
• When blocked signal is generated:
  – If action is default or to catch, signal remains pending until
    • signal is unblocked
    • associated action set to ignore
  – Most systems deliver blocked signal only once
• Each process has signal mask for blocked signals
NAME
   sigaction, sigprocmask, sigpending, sigsuspend - POSIX signal handling functions

SYNOPSIS
   #include <signal.h>

   int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

   int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

   int sigpending(sigset_t *set);

   int sigsuspend(const sigset_t *mask);
The *sigaction* system call is used to change the action taken by a process on receipt of a specific signal.

.... OMMITTED …

The *sigprocmask* call is used to change the list of currently blocked signals. The behaviour of the call is dependent on the value of *how*, as follows.

**SIG_BLOCK**
- The set of blocked signals is the union of the current set and the set argument.

**SIG_UNBLOCK**
- The signals in set are removed from the current set of blocked signals. It is legal to attempt to unblock a signal which is not blocked.

**SIG_SETMASK**
- The set of blocked signals is set to the argument set.

If *oldset* is non-null, the previous value of the signal mask is stored in *oldset*. 
The **sigpending** call allows the examination of pending signals (ones which have been raised while blocked). The signal mask of pending signals is stored in set.

**RETURN VALUE**
The functions `sigaction`, `sigprocmask`, and `sigpending` return 0 on success and -1 on error. The function `sigsuspend` always returns -1, normally with the error EINTR.
handler(sig)

print number for received signal
exit
main()

associate handler(sig) w/ SIGINT
associate handler(sig) w/ SIGTERM

block SIGINT; print this status
sleep(60)
check for INT pending; print status

block SIGTERM; print this status
sleep(60)

print unblocking
unblock SIGINT, SIGTERM
sleep(60)
void handler(int receivedSig) {
    printf("Handler caught sig <%d>\n", receivedSig);
    fflush(stdout);
    _exit(1);
}

main(int argc, char **argv) {
    sigset_t termMask;
    sigset_t intMask;
    sigset_t oldMask;
    sigset_t pendingMask;

    //-- Install handlers for TERM, INT
    signal(SIGTERM, handler);
    signal(SIGINT, handler);
}
//-- Masks to block TERM, INT

sigemptyset(&termMask);
sigaddset(&termMask,SIGTERM);
sigemptyset(&intMask);
sigaddset(&intMask,SIGINT);

//-- Block INT. TERM still terminates.
sigprocmask(SIG_BLOCK,&intMask,&oldMask);
printf("SIGINT Blocked. Process <%d> sleeping.\n",
        getpid());
fflush(stdout);
sleep(60);

//-- If received INT, will print out here.

sigpending(&pendingMask);
if (sigismember(&pendingMask,SIGINT)){
    printf("SIGINT pending. Ready to block TERM.\n");
    fflush(stdout);
}

//--- Now both are blocked.

    sigprocmask(SIG_BLOCK,&termMask,NULL);
printf("SIGTERM blocked. Sleeping.\n"); fflush(stdout);
sleep(60);

printf("Unblocking.\n"); fflush(stdout);

//--- Unblock single signal
sigprocmask(SIG_UNBLOCK,&termMask,NULL);

//--- Return mask to original
sigprocmask(SIG_SETMASK,&oldMask,NULL);
sleep(60);
[jmayo@asimov ~]$ ./tstBlock
SIGINT Blocked, Process <4845> sleeping.
Handler caught sig <15>
[jmayo@asimov ~]$ 

[jmayo@asimov ~]$ kill -TERM 4845
[jmayo@asimov ~]$ 
[jmayo@asimov ~]$ ./tstBlock
SIGINT pending. Ready to block TERM.
SIGTERM blocked. Sleeping.
Unblocking.
Handler caught sig <2>
[jmayo@asimov ~]$ \\

[jmayo@asimov ~]$ kill -INT 4929
[jmayo@asimov ~]$ [ ]
/--- Now both are blocked.

    sigprocmask(SIG_BeecK,&termMask,NULL);
    printf("SIGTERM blocked. Sleeping.\n") ; fflush(stdout);

    sleep(60);

    printf("Unblocking.\n") ; fflush(stdout);

    //--- Unblock single signal
    sigprocmask(SIG_UNBECOCK,&termMask,NULL);

    //--- Return mask to original
    sigprocmask(SIG_SETMASK,&oldMask,NULL);

    sleep(60);
[jmayo@asimov ~]$ ./tstBlock
SIGINT Blocked. Process <5013> sleeping.
SIGTERM blocked. Sleeping.
Unblocking,
Handler caught sig <15>
[jmayo@asimov ~]$ []

[jmayo@asimov ~]$ kill -TERM 5013
[jmayo@asimov ~]$ ]
SIGTERM blocked. Sleeping.
Unblocking.
Handler caught sig <15>

kill -TERM 5096 ; kill -INT 5096
void handler(int receivedSig) {

    printf("Handler caught sig <%d>\n", receivedSig);
    fflush(stdout);
    // _exit(1);
}


//--- Now both are blocked.

sigprocmask(SIG_BLOCK, &termMask, NULL);
printf("SIGTERM blocked. Sleeping.\n"); fflush(stdout);
sleep(60);

printf("Unblocking.\n"); fflush(stdout);
[jmayo@asimov ~]$ gcc -o tstBlock2 tstBlock2.c
[jmayo@asimov ~]$ ./tstBlock2
SIGINT Blocked. Process <5395> sleeping.
SIGTERM blocked. Sleeping.
Unblocking.
Handler caught sig <15>
Handler caught sig <2>
[jmayo@asimov ~]$ 

[jmayo@asimov ~]$ kill -INT 5395 ; kill -TERM 5395
[jmayo@asimov ~]$