Signal Basics

- This section of information is from

  Kay A. Robbins and Steven Robbins "Practical Unix Programming - A guide to concurrency, communication, and multithreading", 1996, Prentice Hall PTR

- A signal is **generated** when the event that causes the signal occurs

- A signal is **delivered** when the process takes action based on the signal
  - The process must be running on a processor at the time action is taken
    - there may be considerable time between signal generation and signal delivery
  - The lifetime of a signal is the interval between its generation and its delivery

- A signal is **pending** if it has been generated but not yet delivered

- A process can **catch, block,** or **ignore** a signal
  - A process catches a signal if it executes a **signal handler** when the signal is delivered
  - A process maintains a signal mask that contains a list of signals to be blocked
    - if a pending signal is blocked, it is delivered when the process unblocks that signal

Signal Basics (cont)

- A process also maintains a list of signals that are to be ignored
  - ignored_signals are delivered and thrown away

- Try "signal -l", which gives a list of symbolic signal names for the system
  - Unfortunately, Minix does not implement this, but Cygwin does

- To send a signal to a process, use a kill system call

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

  - if pid is negative, it sends the signal to the process group with group ID |pid|
  - if pid is 0, it sends a signal to member of the caller's process group

- Two special characters from the keyboard send the SIGINT and SIGQUIT signals
  - "stty -a" reports, among others, the settings of the signal generating character

Signal Basics (cont)

- The "alarm" function causes a SIGALARM signal to be sent to the calling process after a specified number of seconds have elapsed.

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds);
```

  - if alarm is called before the previous one expires, the alarm is reset to the new value and the call returns the number of seconds remaining on the alarm before the call reset the value
  - if alarm with called with 0 for seconds, a previous alarm request is canceled

- Execute the “alarm.c”

```
#include <unistd.h>

void main(void) {
   alarm(5);
   while(1);
}
```


The Signal Mask and Signal Sets

- An instance of sigset_t is used to specify signals to be masked (blocked) and ignored
- There are following functions to manipulate sigset_t variables

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember (const sigset_t *set, int signo);
```


The Signal Mask and Signal Sets (cont)

- A process can examine or modify its process signal mask with "sigprocmask"

  ```
  #include <signal.h>
  int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
  ```

  - The "how" takes one of the following values:
    SIG_BLOCK : add a collection of signals to those currently blocked
    SIG_UNBLOCK : delete a collection of signals from those currently blocked
    SIG_SETMASK : set the collection of signals being blocked
  - "*oset" is a sigset_t variable that holds the set of signals that were blocked before the call to sigprocmask
  - The call returns 0 if the call was successful. It returns -1 and sets "errno" if an error occurs

Sigaction System Call and Sigaction Struct

- The "sigaction" function installs signal handlers for a process.
- "struct sigaction" holds the handler information

```
struct sigaction {
  void (*sa_handler)(); // SIG_DEF, SIG_IGN, or pointer to a signal handler
  sigset_t sa_mask;  // ADDITIONAL signals to be blocked during execution of the
       handler
  int sa_flags;  // special flags and options
 };
```

  - A signal handler is a function that takes one argument (signal number to be delivered) and returns void
  - Most signal handlers do not use the argument, but is it possible to set the same function to handle several different signals.

- The sigaction function has the following signature

```
#include <signal.h>
int sigaction (int signo, const struct sigaction *act, struct sigaction *oact);
```


Sigaction System Call and Sigaction Struct

- Compile sigint_handle.c by "cc -D_POSIX_SOURE sigint_handle.c" and execute it
- Modify sigint_handle.c to capture SIGALARM 5 seconds after the program starts

```
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

int sig_captured = 0;

void catch_ctrl_c(int signo) {
    sig_captured = 1;
    printf("In Handler : I captured ^c
          (signal number: %d)\n", signo);
}
```

```
void main(void) {
   struct sigaction act;
   act.sa_handler = catch_ctrl_c;
   sigemptyset(&act.sa_mask);
   act.sa_flags = 0;
   if (sigaction(SIGINT, &act, NULL) < 0) {
      printf("I could not set a signal handler
            for SIGINT\n");
      exit(-1);
   };

   while (!sig_captured);
   printf("In Main : Yes, you captured SIGINT\n");
}
```


Pause and Sigsuspend System Calls

- One reason for using signals is to avoid busy waiting.
- Posix provides two functions that allow a process to suspend itself until a signal occurs: **pause** and **sigsuspend**

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

  - The pause call suspends the calling process until a signal that is not being ignored is delivered to the process
  - If a signal is caught by the process, the pause returns after the signal handler returns
  - The pause function always returns -1

Pause and Sigsuspend System Calls (cont)

- Consider the following code:

```
#include <unistd.h>
int signal_received = 0;

while (signal_received == 0) pause();
// pause return when any signal is delivered.
// If the signal was not the right one, the program keeps waiting in the loop
```

- Question: What will happen if the signal is delivered between the test of signal_received and pause ?

Pause and Sigsuspend System Calls (cont)

- A workable solution is to test the value of *signal_received* while the signal is blocked, but the signal must be unblocked at the same time as the pause is executed.
- The sigsuspend system call provides a method of doing this

```
#include <signal.h>
int sigsuspend(const sigset_t *sigmask);
```

  - The sigsuspend, with in one system call, sets the signal mask to **sigmask* and suspends the process until a (unmasked) signal is delivered to the process
  - After sigsuspend returns, the signal mask is restored to the state it had before the sigsuspend.

Pause and Sigsuspend System Calls (cont)

- A typical way to use sigsuspend is explained in sigsuspend_handle.c

```
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

int sig_captured = 0;

void catch_ctrl_c(int signo) {
  sig_captured = 1;
  printf("In Handler : I captured ^c
        (signal number: %d)\n", signo);
}

void main(void) {
   struct sigaction act;
   sigset_t original_mask;
   sigset_t before_suspend_mask;
   sigset_t in_suspend_mask;

/*  can have some code above this line  */
   sigprocmask(SIG_SETMASK, NULL, &original_mask);
   sigprocmask(SIG_SETMASK, NULL, &before_suspend_mask);
   sigprocmask(SIG_SETMASK, NULL, &in_suspend_mask);
   sigaddset(&before_suspend_mask, SIGINT);
   sigdelset(&in_suspend_mask, SIGINT);
   sigprocmask(SIG_SETMASK, &before_suspend_mask, NULL);

   act.sa_handler = catch_ctrl_c;
   sigemptyset(&act.sa_mask);
   act.sa_flags = 0;
   if (sigaction(SIGINT, &act, NULL) < 0) {
      printf("I could not set a signal handler for SIGINT\n");
      exit(-1);
   };

   printf("Now, the user can type ^C\n");
   while (!sig_captured)
      sigsuspend(&in_suspend_mask);
   sigprocmask(SIG_SETMASK, &original_mask, NULL);
   printf("In Main : Yes, you captured SIGINT\n");
}
```


Synchronous Alarm (watchdog timer)

- Asynchronous Alarm:
  - a signal may arrive (all user processes) or
  - a watchdog (only tasks can ? since the function must be in the same address space as CLOCK; this feature is used to implement Synchronous Alarm, see below) may be activated without any relation to what part of a process is currently executing

- Synchronous Alarm: A signal is delivered as a message, and thus can be received only when the recipient has executed “RECEIVE” (only when the recipient expects it)
  - Only system processes can receive synchronous alarm (notification message)
  - To implement synchronous alarm, a watchdog timer is used (the “s_alarm_timer” field in the “priv” table (the “timer_t” type)_
    - System processes ask the SYSTEM task to execute do_setalarm (10628, called from 7137 and 10254)
    - the watchdog function (tmr_func in struct timer (1328)) is “cause_alarm” (10239) which issues “lock_notify” (10270)

Synchronous Alarm (watchdog timer) (cont)

- Examples of synchronous message
  - Each time the hard disk driver sends a command to the disk controller, it sets a wakeup call in case the command fails completely
  - The floppy disk driver uses a timer to wait for the disk motor to get up to speed and to shut down the motor if no activity occurs for a while
  - Some printers with a movable print head can print at 120 characters/sec (8.3msec/character) but cannot return the print head to the left margin in 8.3 msec, so the terminal driver delays after typing a carriage return

- Note: this section is pretty confusing since I took the explanation from the text.
  - In the previous versions of Minix, all device drivers and Kernel shared the address space. Therefore, watchdog was used extensively.
  - However, in Minix 3, each system process (device driver, server) runs in its own address space, and therefore, a watchdog function cannot be used easily.
  - Therefore, system processes use synchronous messages
  - But the explanation in the text seems to be based on the previous implementations of Minix