Overview

Today: Posix signals

- The content of this lecture is partially covered by
  - Book: Programming for The Real World, Bill O. Gallmeister, O’Reilly&Associates, Inc.

Posix.1 Signals

- Signals are an integral part of multitasking in the UNIX/POSIX environment. Signals are used for many purposes, including:
  1. Exception handling (bad pointer accesses, divide by zero, etc.)
  2. Process notification of asynchronous event occurrence (I/O completion, timer expiration, etc.)
  3. Process termination in abnormal circumstances
  4. Interprocess communication

- Signals are similar to the notion of hardware interrupts. However, they are managed and delivered by the Operating System.

Posix Signals

- A POSIX signal is the software equivalent of an interrupt or exception occurrence. When a process receives a signal, it means that something happened which requires the process’ s attention.

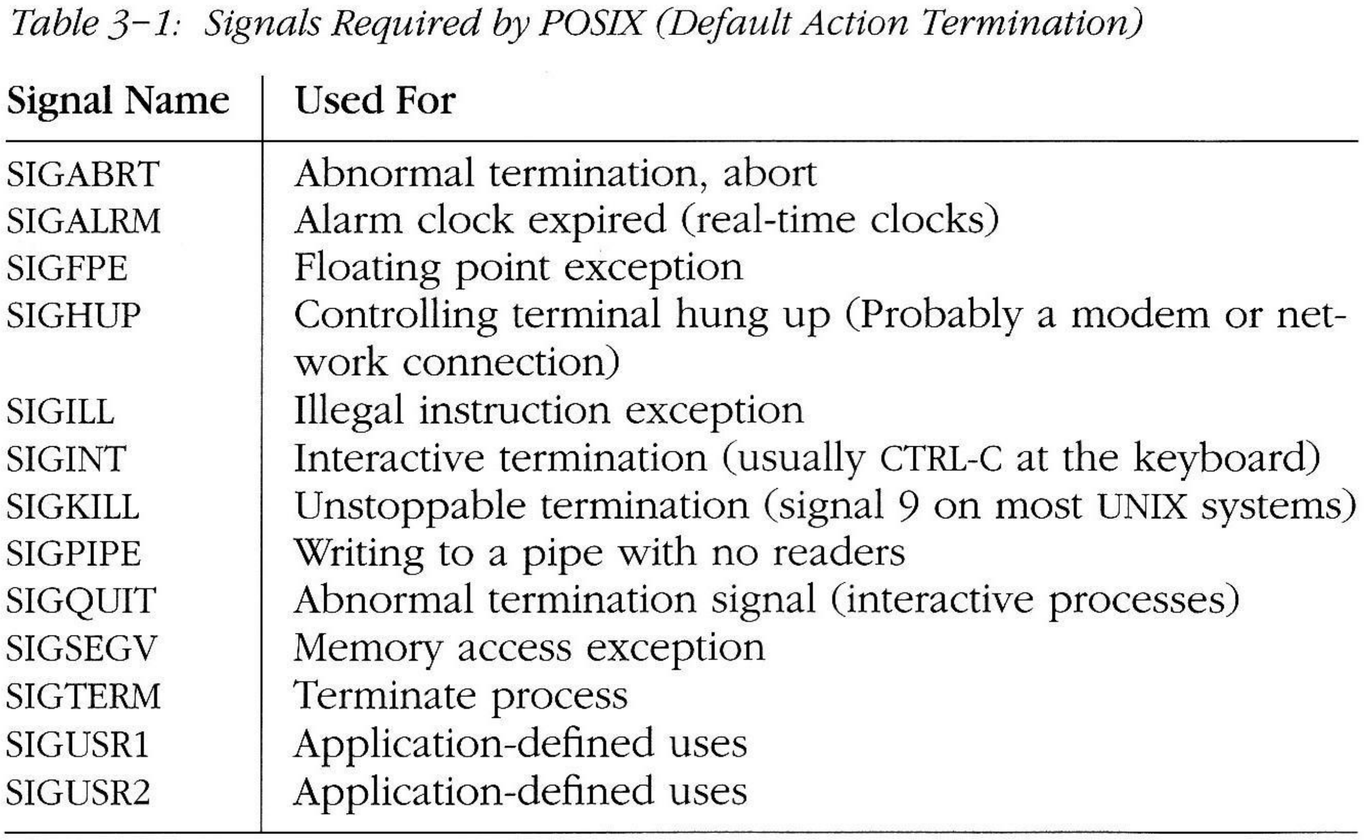


Table 3–1: Signals Required by POSIX (Default Action Termination)

Signal Name	Used For
SIGABRT	Abnormal termination, abort
SIGALRM	Alarm clock expired (real-time clocks)
SIGFPE	Floating point exception
SIGHUP	Controlling terminal hung up (Probably a modem or network connection)
SIGILL	Illegal instruction exception
SIGINT	Interactive termination (usually CTRL-C at the keyboard)
SIGKILL	Unstoppable termination (signal 9 on most UNIX systems)
SIGPIPE	Writing to a pipe with no readers
SIGQUIT	Abnormal termination signal (interactive processes)
SIGSEGV	Memory access exception
SIGTERM	Terminate process
SIGUSR1	Application-defined uses
SIGUSR2	Application-defined uses

Dealing with signals

- There are different ways in which you can deal with a signal:
  1. You can block a signal for a while, and get to it (by unblocking it) later. Blocking signals is a temporary measure.
  2. You can ignore the signal, in which case it is as if the signal never arrived.
  3. You can handle the signal by executing a default action to deal with the signal (the default action often is to kill the process receiving the signal)
  4. You can handle the signal by setting up a function to be called whenever a signal with a particular number arrives (**asynchronous handling**).
  5. You can handle the signal by blocking it at first, and later by calling within the process sigwait(&signal_set, &signal); (**synchronous handling**).

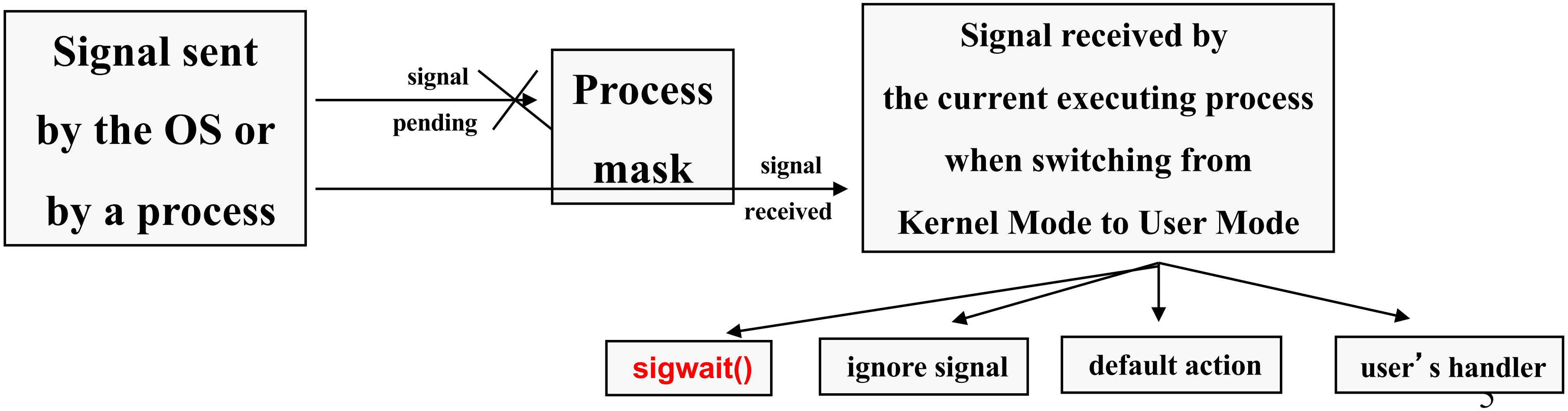
the set of signals to await
is passed in signal_set

- There are two spare signals available to user applications: **SIGUSR1** and **SIGUSR2**. Any application can use them as it wants.

Steps of Signal's Delivery and Handling

- Event of **sending a signal**:
  1. The OS updates the descriptor of the destination process to represent that a new signal has been sent.
  2. Signals that have been sent but not yet received are called **pending signals**. At any time, only one pending signal of a given type may exist for a process; additional pending signals of the same type to the same process are not queued but simply discarded.

- Event of **receiving a signal**:
  1. If the sent signal is blocked by the process mask, the process will not receive the signal until it removes the block: the signal remains pending.
  2. When switching from Kernel Mode to User Mode, check whether a signal has been sent to the current executing process (this happens at every timer interrupt). Unblocked signals are received.
  3. Determine whether ignore the signal, execute a default action, execute user's signal handler, or call sigwait() explicitly.



Signal sent by the OS or by a process

signal pending

Process mask

signal received

Signal received by the current executing process when switching from Kernel Mode to User Mode

sigwait()

ignore signal

default action

user's handler

Setting the Process Mask

- The collection of signals that are currently blocked is called the *signal mask*. Each process has its own signal mask. When you create a new process, it inherits its parent's mask. You can block or unblock signals with total flexibility by modifying the signal mask

- Function: int **sigprocmask** *(int how, const sigset_t *set, sigset_t *oldset)*
  - The sigprocmask function is used to examine or change the calling process's signal mask. The ***how*** argument determines how the signal mask is changed, and must be one of the following values:
  - **SIG_BLOCK**
    - Block the signals in set---add them to the existing mask. In other words, the new mask is the union of the existing mask and *set*.
  - **SIG_UNBLOCK**
    - Unblock the signals in *set*---remove them from the existing mask.
  - **SIG_SETMASK**
    - Use *set* for the mask; ignore the previous value of the mask.
  - The last argument, *oldset*, is used to return information about the old process signal mask

Dealing with signals

- How can we set/change the process's signal mask?

```
sigset_t  newset, oldset;
int i;
i = sigprocmask(SIG_BLOCK, &newset, &oldset);
```

The signals in the argument newset are those that are added to the process's signal mask

```
i = sigprocmask(SIG_UNBLOCK, &newset, &oldset);
```

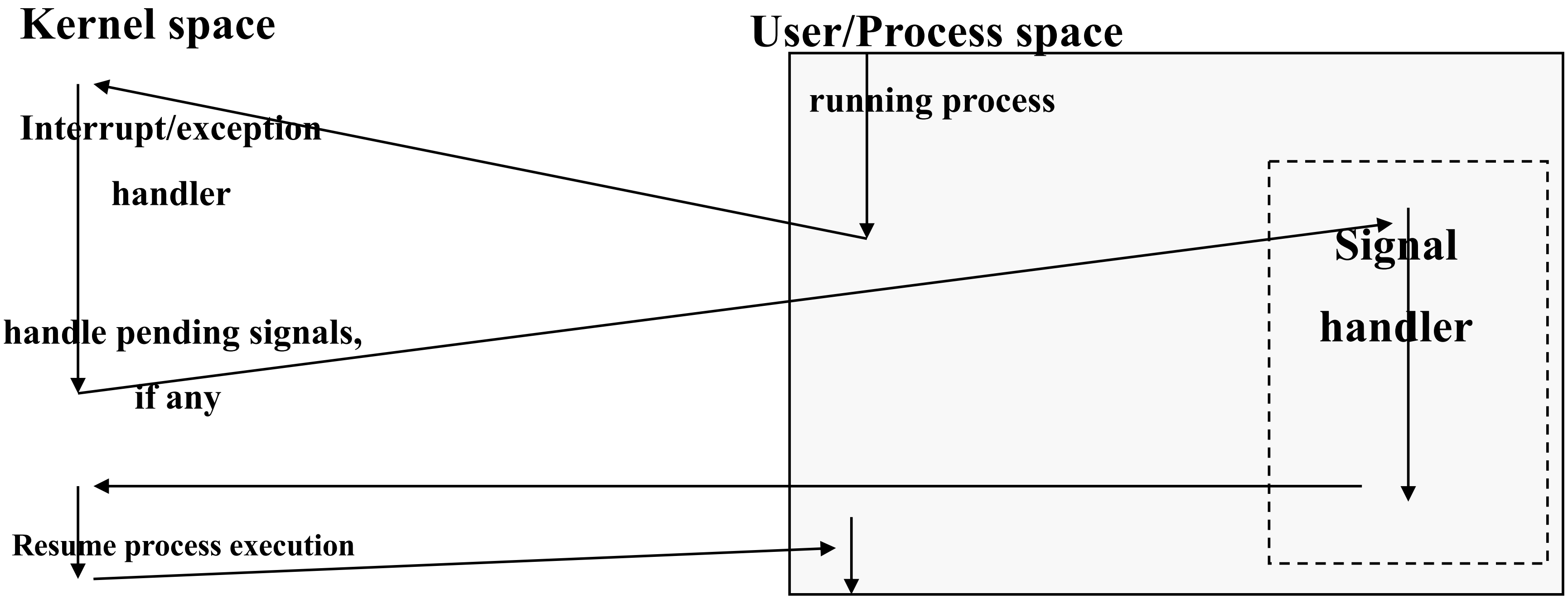
The signals in the argument newset are those that are subtracted to the process's signal mask

```
i = sigprocmask(SIG_SETMASK, &newset, &oldset);
```

The signals in the argument newset are those that become the process's signal mask

Appendix: signal handlers execute in user space

- In the Linux kernel, each process descriptor has a field where to store the signals mask and each pending signal.



- A signal handler can interact with the regular execution flow of the corresponding process by simply sharing global variables: **the regular execution flow and signal handler execute in the same memory space**.

Appendix: installing signal handlers (simplified technique)

signal() function is a simplified interface to the more general **sigaction** approach.

SYNOPSIS

```
#include <signal.h>

typedef void (*sig_t) (int);

sig_t signal(int sig, sig_t func);
```

DESCRIPTION

The sig argument specifies which signal to handle. The func procedure allows a user to choose the action upon receipt of a signal. To set the default action for the signal, func should be SIG_DFL. A SIG_DFL resets the default action. To ignore the signal, func should be SIG_IGN. This will cause subsequent instances of the signal to be ignored and pending instances to be discarded.

RETURN VALUES

The previous action is returned on a successful call. Otherwise, SIG_ERR is returned and the global variable errno is set to indicate the error.

Sending Signals to a process

The **kill** function is used to send a POSIX signal

kill -- send signal to a process

SYNOPSIS

```
    #include <signal.h>

    int kill(pid_t pid, int sig);
```

DESCRIPTION

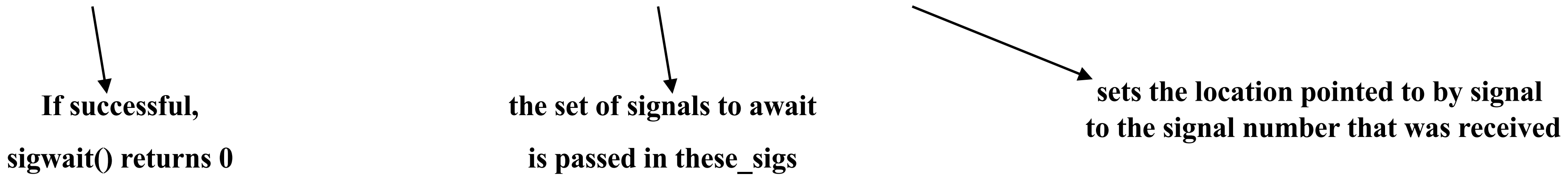
The **kill**() function sends the signal specified by sig to pid, a process or a group of processes. For a process to have permission to send a signal to a process designated by pid, the user ID of the receiving process must match that of the sending process or the user must have appropriate privileges (such as the user is the super-user).

Synchronous signal-waiting

- To synchronously wait for a signal, **sigwait** can be used.

- **sigwait** performs the wait for signals, but it does not call the signal handler for the signal that arrives. It just tells you which signal arrived.

- int **sigwait** (const sigset_t *these_sigs, int *signal);



If successful,

sigwait() returns 0

the set of signals to await

is passed in these_sigs

sets the location pointed to by signal
to the signal number that was received

- **Quiz: should the process normally block (mask) the signals to be waited by sigwait?**

- **Quiz: what does happen if an unblocked signal is delivered before executing sigwait?**

Synchronous signal-waiting

- int **sigwait** (const sigset_t *these_sigs, int *signal);

- **Quiz: should the process normally block (mask) the signals to be waited by sigwait?**
- **Answer: When using sigwait, the process mask should normally block the signals that sigwait wants to wait for. Sigwait will alter the process mask during execution, and then it will restore the original mask after it returns.**

- **Quiz:what does happen if an unblocked signal is delivered before executing sigwait?**
- **Answer: If the expected signals do arrive before calling sigwait and are not blocked by the process mask, then they are handled by the handler functions, not passed to sigwait. In conclusions, the old-style handlers take precedence.**

- If you don’t want to experience an arbitrarily long waiting time, **sigtimedwait** is equivalent to sigwait but it has a timeout!

An example with SIGALRM and sigwait

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
#include "errno.h"

extern int errno;


void timestamp( char *str ) {
   time_t t;

   time(&t);
   printf("The time %s is %s\n", str, ctime(&t));
}
```

```
int main( int argc, char *argv[] ) {

   int result = -1;
   sigset_t waitset;
   int  signum;


   sigemptyset( &waitset );
   sigaddset( &waitset, SIGALRM );

   sigprocmask(SIG_BLOCK, &waitset, NULL);

   alarm(2);

   timestamp( "before sigwait" );

   result = sigwait( &waitset, &signum );

   if( result == 0 )
      printf( "sigwait returned for signal %d\n", signum );
   else {
      printf( "sigwait returned error number %d\n", errno );
      perror( "sigwait() function failed\n" );
   }

   timestamp( "after sigwait" );
}
```

An example with SIGALRM and sigwait

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
#include "errno.h"

extern int errno;


void timestamp( char *str ) {
   time_t t;

   time(&t);
   printf("The time %s is %s\n", str, ctime(&t));
}
```

```
mcaccamo@versilia:~/cs431_f16_teaching/code/lecture7$ ./signal

The time before sigwait is Tue Sep 13 21:05:54 2016

sigwait returned for signal 14
The time after sigwait is Tue Sep 13 21:05:56 2016

mcaccamo@versilia:~/cs431_f16_teaching/code/lecture7$
```

```
int main( int argc, char *argv[] ) {

   int result = -1;
   sigset_t waitset;
   int  signum;


   sigemptyset( &waitset );
   sigaddset( &waitset, SIGALRM );

   sigprocmask(SIG_BLOCK, &waitset, NULL);

   alarm(2);

   timestamp( "before sigwait" );

   result = sigwait( &waitset, &signum );

   if( result == 0 )
      printf( "sigwait returned for signal %d\n", signum );
   else {
      printf( "sigwait returned error number %d\n", errno );
      perror( "sigwait() function failed\n" );
   }

   timestamp( "after sigwait" );
}
```

Appendix:
An example with SIGALRM and sigwait

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
#include "errno.h"

extern int errno;

void catcher(int sig) {
   printf("Signal catcher called for signal %d\n", sig);
}


void timestamp( char *str ) {
   time_t t;

   time(&t);
   printf("The time %s is %s\n", str, ctime(&t));
}
```

```
int main( int argc, char *argv[] ) {

   int result = -1;
   sigset_t waitset;
   int  signum;


   sigemptyset( &waitset );
   sigaddset( &waitset, SIGALRM );

   sigprocmask(SIG_BLOCK, &waitset, NULL);

   signal(SIGALRM, catcher);


   alarm(10);

   timestamp( "before sigwait" );

   //while(1);

   result = sigwait( &waitset, &signum );

   if( result == 0 )
      printf( "sigwait returned for signal %d\n", signum );
   else {
      printf( "sigwait returned error number %d\n", errno );
      perror( "sigwait() function failed\n" );
   }

   timestamp( "after sigwait" );
}
```

comment it out if
you want to use the catcher
function for signal delivery

Appendix:
An example with SIGALRM and sigwait

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
#include "errno.h"

extern int errno;

void catcher(int sig) {
   printf("Signal catcher called for signal %d\n", sig);
}


void timestamp( char *str ) {
   time_t t;

   time(&t);
   printf("The time %s is %s\n", str, ctime(&t));
}
```

```
Marcos-MBP:POSIX_code mcaccamo$ ./signals_handler
The time before sigwait is Wed Jun 10 13:22:40 2020


sigwait returned for signal 14
The time after sigwait is Wed Jun 10 13:22:50 2020

Marcos-MBP:POSIX_code mcaccamo$
```

```
int main( int argc, char *argv[] ) {

   int result = -1;
   sigset_t waitset;
   int  signum;


   sigemptyset( &waitset );
   sigaddset( &waitset, SIGALRM );

   sigprocmask(SIG_BLOCK, &waitset, NULL);

   signal(SIGALRM, catcher);


   alarm(10);

   timestamp( "before sigwait" );

   //while(1);

   result = sigwait( &waitset, &signum );

   if( result == 0 )
      printf( "sigwait returned for signal %d\n", signum );
   else {
      printf( "sigwait returned error number %d\n", errno );
      perror( "sigwait() function failed\n" );
   }

   timestamp( "after sigwait" );
}
```

comment it out if
you want to use the catcher
function for signal delivery