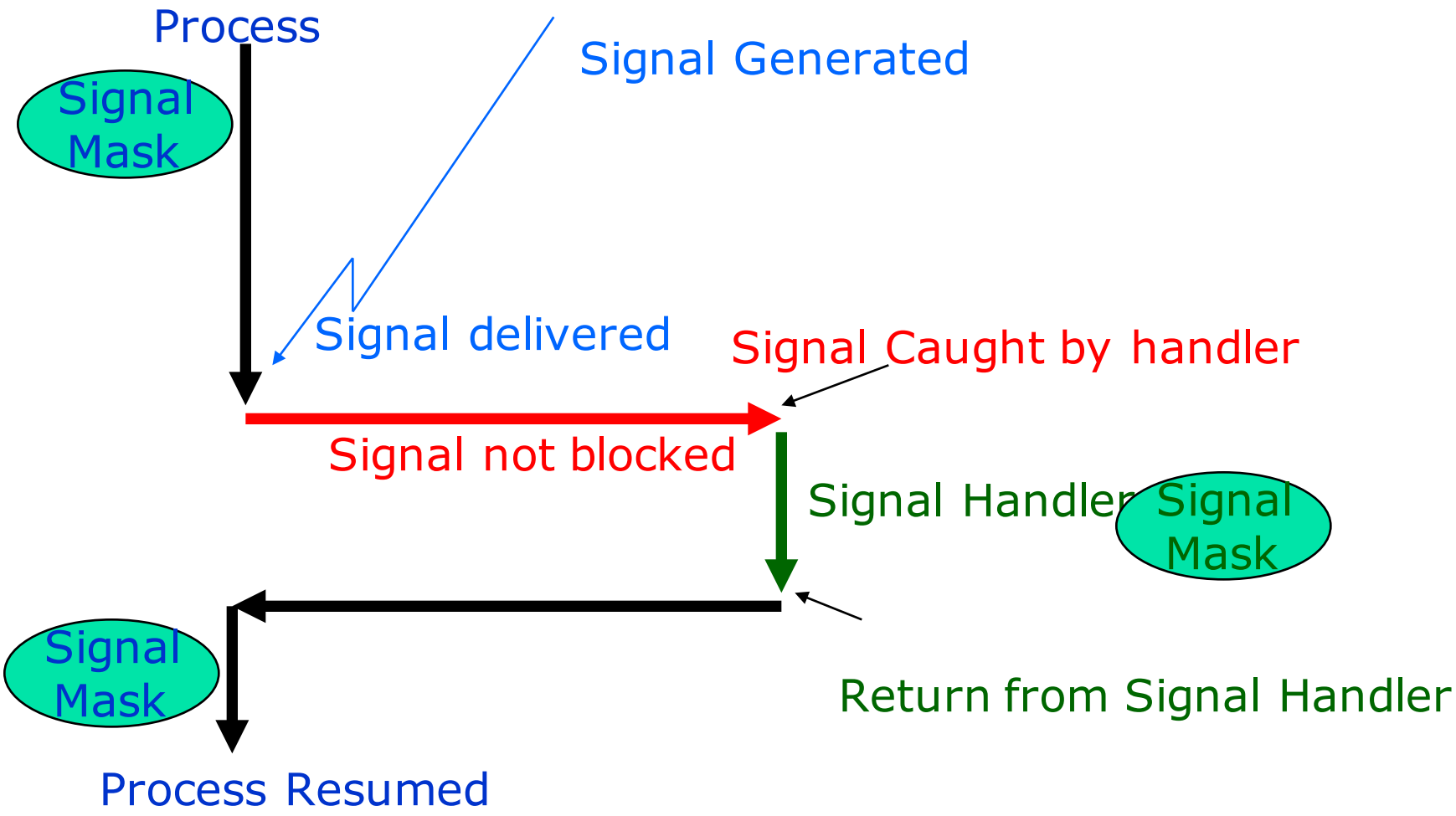
Introduction to signals

- Concept
- POSIX signals
- Generating signals
- Catching signals
- Signal sets
- Signal masks
- Waiting for signals

Signals

- A signal is a software notification to a process of an event.
- Needed to enable asynchronous events
- Examples of asynchronous events:
  - **Email message arrives** on my machine – mailing agent (user) process should retrieve
  - **Invalid memory access happens** – OS should inform scheduler to remove process from the processor
  - **Alarm clock goes on** – process which sets the alarm should catch it

How Signals Work



Examples of POSIX Signals

Signal	Description	default action
SIGABRT	process abort	implementation dependent
SIGALRM	**alarm clock**	**abnormal termination**
SIGBUS	access undefined part of memory object	implementation dependent
SIGCHLD	child terminated, stopped or continued	ignore
SIGILL	invalid hardware instruction	implementation dependent
SIGINT	**interactive attention signal (usually ctrl-C)**	**abnormal termination**
SIGKILL	**terminated (cannot be caught or ignored)**	**abnormal termination**

Examples of POSIX Signals

Signal	Description	default action
SIGSEGV	**Invalid memory reference**	**implementation dependent**
SIGSTOP	Execution stopped	stop
SIGTERM	**termination**	**Abnormal termination**
SIGTSTP	Terminal stop	stop
SIGTTIN	Background process attempting read	stop
SIGTTOU	Background process attempting write	stop
SIGURG	**High bandwidth data available on socket**	**ignore**
SIGUSR1	**User-defined signal 1**	**abnormal termination**

Generating Signals

- Signal has a symbolic name starting with SIG
- Signal names are defined in signal.h
- Users can generate signals (e.g., SIGUSR1)
- OS generates signals when certain errors occur (e.g., SIGSEGV – invalid memory reference)
- Specific calls generate signals such as alarm (e.g., SIGALRM)

Command Line Generates Signals

- You can send a signal to a process from the command line using **kill**
- kill -l will list the signals the system understands
- kill [-signal] pid will send a signal to a process.
  - The optional argument may be a name or a number (default is SIGTERM).
- To unconditionally kill a process, use:
  - kill -9 pid which is

    kill -SIGKILL pid.

Command Line Generates Signals

- CTRL-C is SIGINT (interactive attention signal
- CTRL-Z is SIGSTOP (execution stopped – cannot be ignored)
- CTRL-Y is SIGCONT (execution continued if stopped)
- CTRL-D is SIGQUIT (interactive termination: core dump)

Signal Masks

- Process can temporarily prevent signal from being delivered by *blocking* it.
- *Signal Mask* contains a set of signals currently blocked.
- **Important!** Blocking a signal is different from ignoring signal. Why?
- When a process blocks a signal, the OS does not deliver signal until the process unblocks the signal
    - A *blocked* signal is not delivered to a process until it is unblocked.
- When a process ignores signal, signal is delivered and the process handles it by throwing it away.

Signal Masks

SigMask

SigInt	SigQuit	SigKill	…	SigCont	SigAbrt
0	0	1	…	1	0

Signal SigInt Bit 2, Signal Sigkill Bit 9,
Signal SigChld Bit 20

A SIGSET is a collection of signals:
#000003 is SIGHUP + SIGINT

Manipulate signal mask

- Manipulate signal mask of a process
  - int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
    - *how*: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
    - set: New set
    - oset: Old set
  - See example2.c and example3.c

Manipulate signal sets

- #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);
```

Example: Blocking signals in a critical region

- For a critical region where you don't want certain signal to come, the program will look like:

```
sigprocmask(SIG_BLOCK, &newmask, &oldmask);
……. /* critical region */
sigprocmask(SIG_SETMASK, &oldmask, NULL);
```

Catching and Ignoring Signals - SIGACTION

- sigaction function allows the caller to examine or specify action associated with a specific signal
- Program installs *signal handler* by calling sigaction with the name of a user-written function.
- The function sigaction is used to specify what is to happen to a signal when it is delivered.

sigaction

- Supersedes the signal function
  - #include <signal.h>
  - int sigaction(int signo, const struct sigaction * act, struct sigaction *oact)
  - struct sigaction

```
{
  void (*sa_handler)(); /* signal handler */
  sigset_t sa_mask;     /*additional signal to block */
   int sa_flags;        /* options:  we will set it to 0 */
   sa_sigaction;          /* we will not use it */
};
```

- See example4.c

kill

- Send a signal to a process
  - #include <signal.h>
  - #include <sys/types.h>
  - int kill(pid_t pid, int signo);
  - See example5.c

Timers Generate SIGALRM Signals

- #include <unistd.h>
- unsigned alarm (unsigned seconds);
- alarm(20) creates SIGALRM to calling process after 20 real time seconds.
- Calls are not stacked
- alarm(0) cancels alarm
- See example6.c for the use of *alarm*

Impact of signals on system calls

- A system call may return prematurely
  - See example7.c
- How can we deal with this problem?
  - Check the return value of the system call and act accordingly

Waiting for Signals

- Signals provide method for waiting for event without busy waiting
- Busy waiting
  - Means continually using CPU cycles to test for occurrence of event
  - Means a program does this testing by checking the value of variable in loop
- More Efficient Approach
  - Suspend process until waited-for event occurs
- POSIX pause, sigsuspend, sigwait provide three mechanisms to suspend process until signal occurs
- pause() – too simple, and problematic if signals are delivered before pause (cannot unblock the signal and start pause() in atomic way)

sigsuspend

- sigsuspend function sets signal mask and suspends process until signal is caught by the process
- sigsuspend returns when signal handler of the caught signal returns
- #include <signal.h>
- int sigsupend(const sigset_t *sigmask);

Example: sigsuspend

- What's wrong?

```
sigfillset(&sigmost);
sigdelset(&sigmost, signum);
sigsuspend(&sigmost);
```

- signum is the only signal unblocked that can cause sigsuspend to return
- If the signal signum is delivered before the start of the code segment, the process still suspends itself and deadlocks if another signum is not generated

Example: sigsuspend (Correct Way to Wait for Single Signal)

```
static volatile sig_atomic_t sigreceived =0;
/*assume signal handler has been setup for signum and it sets sigreceived=1 */

sigset_t maskall, maskmost, maskold;
int signum = SIGUSR1;

sigfillset(&maskall);
sigfillset(&maskmost);
sigdelset(&maskmost, signum);
sigprocmask(SIG_SETMASK, &maskall, &maskold);
if (sigreceived == 0)
    sigsuspend(&maskmost);
sigprocmask(SIG_SETMASK, &maskold, NULL);
```

Use sigwait

- **This function is much cleaner and avoids races and errors !!!!**

- 1. First block all signals
- 2. Put the signals you want to wait for in sigset_t
- 3. Call sigwait
- 4. sigwait blocks the process until at least one of these signals is pending.
- 5. It removes one of the pending signals and gives you the corresponding signal number in the second parameter..
- 6. Do what you want: no signal handler needed.
- 7. It returns 0 on success and -1 on error with errno set.

```
#include <signal.h>
int sigwait(const sigset_t *restrict sigmask, int
*restrict signo);
```