



---

# Signal Handling and Threads

Lecture 16

Klara Nahrstedt

# CS241 Administrative

- Read Chapter 8 in R&R and Chapter 13.5 in R&R
- SMP4 due on Monday, 2/26
- Homework 1 will be posted this evening, 2/23
- **Deadline for Homework 1 is Friday, March 2, at 5pm**
  - Homework 1 is a preparation for Midterm
  - Homework 1 is an individual effort – NO WORKING IN GROUPS/PAIRS!!
- **MIDTERM – MONDAY, March 5, 11am**

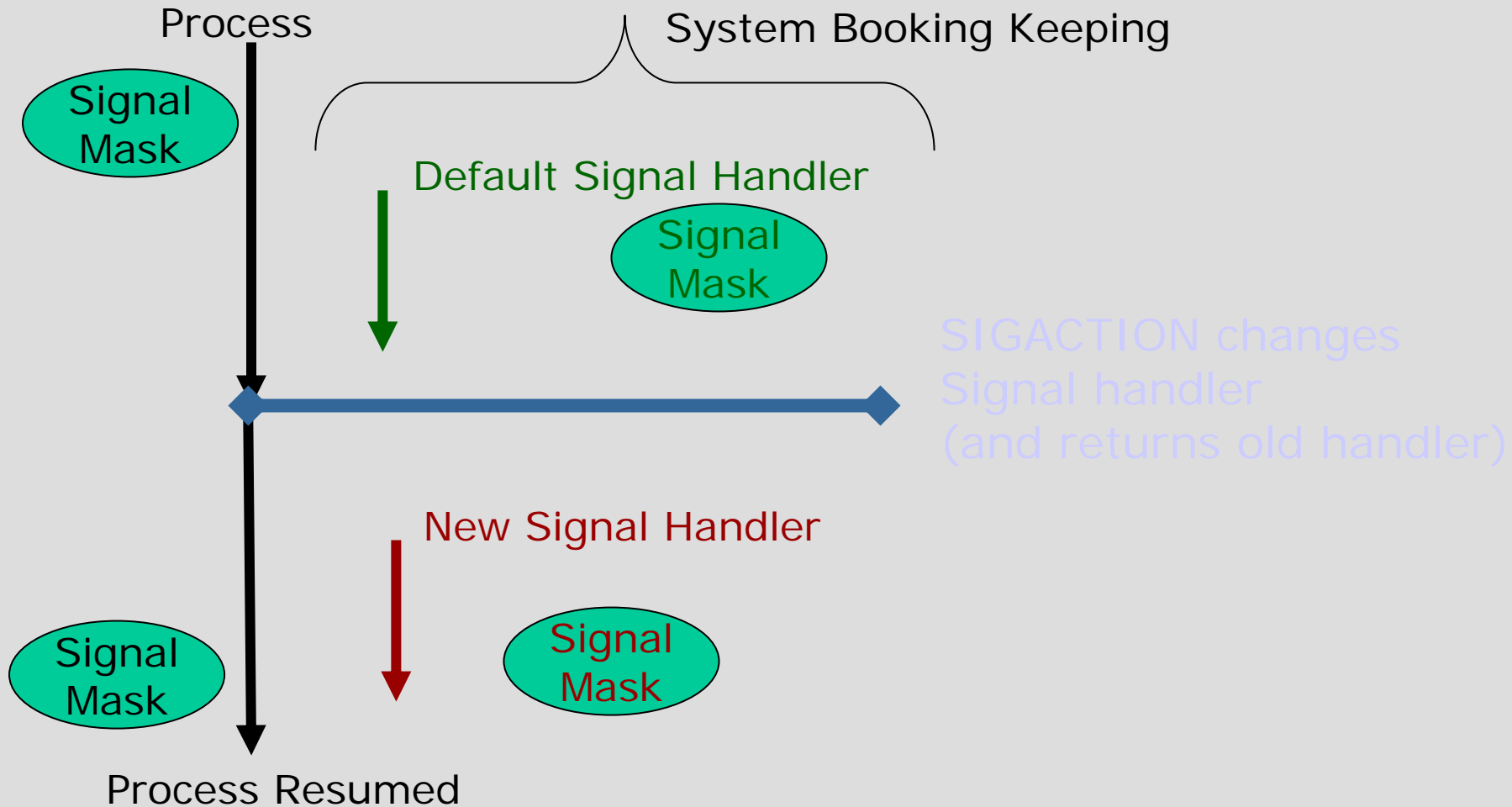
# Outline

- Use signal masks and handlers
- Signals and threads

# 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



```
#include <signal.h>
```

```
int sigaction(int signo, const struct  
sigaction *act,  
struct sigaction *oact);
```

```
struct sigaction {  
    void (*sa_handler)(int); /* SIG_DFL,  
    SIG_IGN or pointer to function */  
    sigset_t sa_mask; /* additional signals to be  
    blocked during execution of handler */  
    int sa_flags; /* special flags and options */  
    void(*sa_sigaction)(int, siginfo_t *,  
void *); /* real-time handler */ };
```

1. Either **act** or **oact** may be `NULL`.
2. If **SA\_SIGINFO** flag of **sa\_flags** field is clear, **sa\_handler** specifies the action to be taken.

# Example: Set up Signal Handler for SIGINT

```
struct sigaction newact;
```

```
newact.sa_handler = mysighand; /*set new handler*/
```

```
newact.sa_flags = 0; /* no special options */
```

```
if ((sigemptyset(&newact,sa_mask) == -1) ||
```

```
    (sigaction(SIGINT, &newact, NULL) == -1))
```

```
perror("Failed to install SIGINT signal handler");
```

# Set up Signal Handler that Catches SIGINT Generated by Ctrl-C

```
void catchctrlc(int signo) {  
    char handmsg[] = "I found Ctrl-C\n";  
    int msglen = sizeof(handmsg);  
  
    write(STDERR_FILENO, handmsg, msglen);  
}  
...  
struct sigaction act;  
act.sa_handler = catchctrlc;  
act.sa_flags = 0;  
if ((sigemptyset(&act.sa_mask) == -1) ||  
    (sigaction(SIGINT, &act, NULL) == -1))  
    perror("Failed to set SIGINT to handle Ctrl-C");
```

Note: write is async-signal safe – meaning it can be called inside a signal handler. Not so for printf or strlen.

# 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 sigsuspend(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);
```

# Signal Handling & Threads

(Ch13.5 p473-475)

<b>type</b>	<b>Delivery action</b>
asynchronous	Delivered to some thread that has it unblocked
synchronous	Delivered to thread that caused it
directed	Delivered to the identified thread

# Directing Signal to Particular Thread

```
#include <signal.h>
```

```
#include <pthread.h>
```

```
int pthread_kill(pthread_t thread, int sig);
```

```
-----
```

If successful `pthread_kill` returns 0.

```
If (pthread_kill(pthread_self(), SIGKILL))  
    fprintf(stderr, "failed to commit suicide"\n");
```

**Example: Thread kills itself and the process it is in because SIGKILL cannot be caught, blocked, or ignored.**

**Not always the case for some signals**

# Masking Signals for Threads

- Signal handlers are **process-wide**
- Each thread has its **own signal mask** for process wide signals – only use sigprocmask for initialization before creating threads
- Thread can
  - Examine its signal mask
  - Set its signal mask
- sigprocmask function **should not be used** when the process has multiple threads

# Masking Signals for Threads

```
#include <pthread.h>
#include <signal.h>
int pthread_sigmask(int how,
                    const sigset_t *restrict set,
                    sigset_t *restrict pset);
```

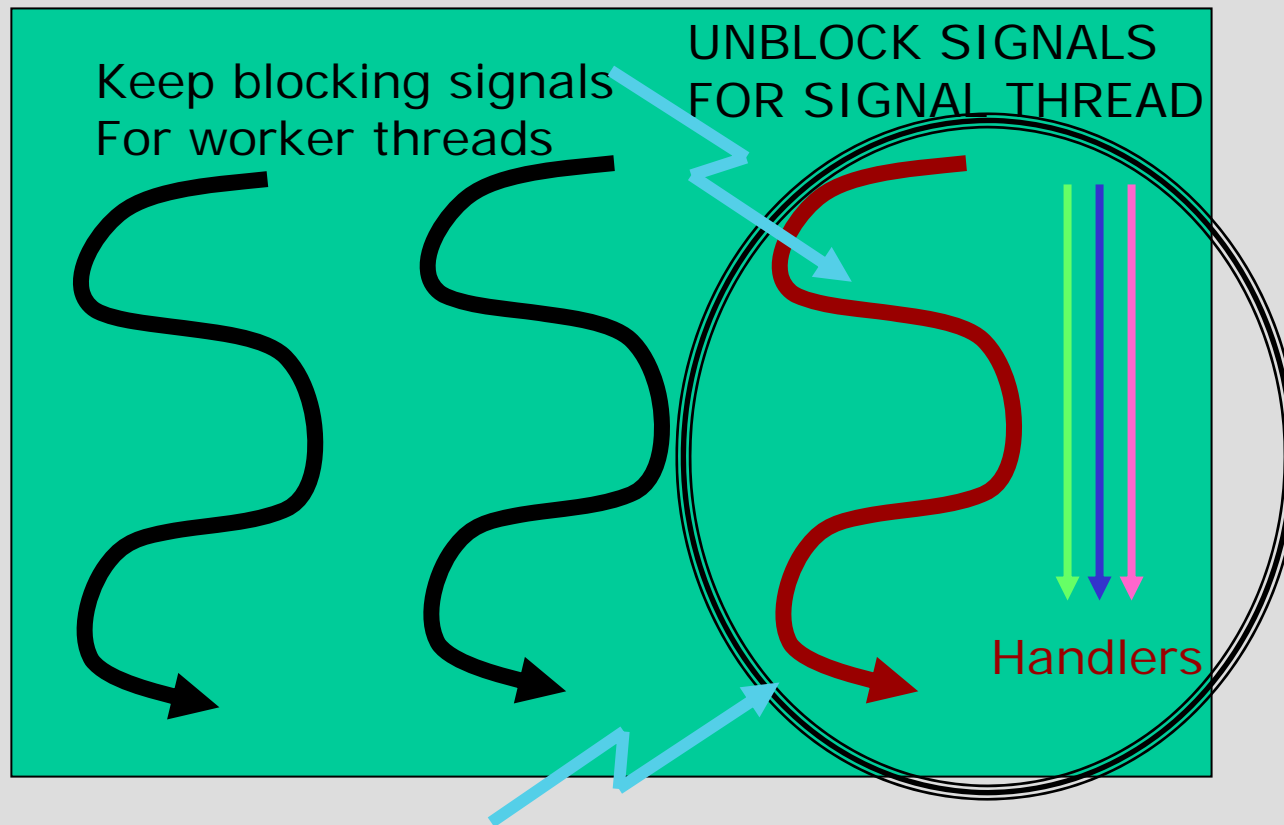
If 'how' is SIG\_SETMASK, then thread's signal mask is replaced by 'set'

If 'how' is SIG\_BLOCK, then additional signals in 'set' are blocked by the thread

If 'how' is SIG\_UNBLOCK, then any of blocked signals in 'set' are removed from the thread's current signal mask

# Threads

PROCESS BLOCKS ALL SIGNALS



# Dedicating Threads for Signal Handling

- Signal handlers are **process-wide** and installed with `sigaction` as in single-threaded processes
- Distinction between **process-wide signal handlers** and **thread-specific signal masks** is important!!!
- Remember:
  - When a signal is caught, the signal that caused the event is automatically blocked on entry to the signal handler
  - With multi-threaded application, nothing prevents another signal of the same type from being delivered to another thread that has the signal unblocked
- **A recommended strategy for dealing with signals in multi-threaded processes is to dedicate particular threads to signal handling**

# Dedicating Threads for Signal Handling

- ❑ Dedicate particular threads to handle signals
- ❑ Main thread blocks all signals before creating any child threads
- ❑ Signal mask is **inherited** by children
- ❑ Thread dedicated to signal handling executes `sigwait` on specific signals to be handled or uses `pthread_sigmask` to unblock signal

# Summary

- Signals – asynchronous events
- Generating signals
- Programming signals
- Signal masks
- Signal handlers
- Capturing and ignoring signals
- Waiting for signals