Asynchronously handling signals

- We have used `pause`(2) to **wait for** a signal.
- Signals can occur **at any time**, and interrupt processing *at any point*. Even, *e.g.*, during a `malloc`.
  - What should happen if a **system call** is interrupted...
  - ...or a function that modifies **global variables**...
  - ...or the signal **handler** itself?

Solutions *signal handling is messy*

- A function is called **reentrant** pr **async-signal-safe**, if it can safely be called from an interrupt handler *while* itself is being interrupted by a signal.
  - *Only* use reentrant functions in signal handlers!
  - A list is in `signal`(7): No `printf`, no `malloc`, ...
- You might even need to maintain & restore a copy of `errno`!
- When interrupted by a signal, **system calls** and **library functions** shall fail with `EINTR`, or restart automatically.
- A program may **block signals** during critical sections, *e.g.*, signal handlers.

Restart system calls — and other flags

```
1 struct sigaction {
2     void (*sa_handler)(int);
3     int sa_flags;
4
5     /* still incomplete */
6 };
```

- The `sa_flags` member further tunes the behavior of signal handling.
- It is the bitwise OR of zero or more of the following:

`SA_NOCLDWAIT` If used with the `SIGCHLD` signal, do not transform children into **zombies** when they terminate. See `wait`(2), *Notes*!

`SA_RESETHAND`* Trigger the handler **only once**, then reset handler.

`SA_RESTART`* Some system calls or library functions may be **restarted** instead of failing with `EINTR`. See `signal`(7) for further details.

- `wait`(2) is restarted if `SA_RESTART` is set.
- `pause`(2) is never restarted, otherwise it would never return.

`SA_SIGINFO` *cf.* page 288.

... more flags are described in `sigaction`(2).

Note Marked* flags are not available with `gcc -std=c99`.

Blocking signals

- By default, a signal is blocked while a signal of **the same type** (*i.e.*, the same signal number) is handled.
- Signals occurring while they are blocked, become **pending**.
  - Pending signals are delivered **when unblocked**.
  - **Only one** signal of each type may be pending, so it's rather a **flag**.

(There's an exception for the signals above 32)

```
$ ./signal0
pid is 13789
^Ccaught signal 2          /* Repeat C-c... */
^C^C^C^Ccaught signal 2          /* ...fast */
```

- The signal handler has a delay of 1s built in.
- **Only two** signals are caught!

- What if **other signals** arrive while inside the handler?

```
$ kill -USR1 13789 & kill -INT 13789      # try this
```

- We need means to block more signals inside a handler.

The signal mask

```
1 #include <signal.h>
2
3 int sigemptyset(sigset_t *set);          int sigaddset(sigset_t *set, int sig);
4 int sigfillset(sigset_t *set);           int sigdelset(sigset_t *set, int sig);
5
6 int sigprocmask(int how, const sigset_t *set, sigset_t *old);
```

- `sigemptyset`(3) **initializes** an empty set of signals, `sigfillset`(3) a set containing all signals.
- `sigaddset`(3) adds a `sig`nal to a set, `sigdelset`(3) removes it.
- `sigprocmask`(2) sets/queries the set of currently blocked signals:
  - If not `NULL`, the `set` pointed to is used.
  - If not `NULL`, the old set is passed back via `old`.
  - `how` specifies how `set` is used:

    `SIG_BLOCK` Add the `set` to currently blocked signals.
    `SIG_UNBLOCK` Remove the `set` from currently blocked signals.
    `SIG_SETMASK` Block exactly the signals in the `set`.

Example

```c
#define _POSIX_C_SOURCE 199309L

#include <signal.h>
#include <unistd.h>

int main(void)
{
    sigset_t ss;

    sigfillset(&ss);
    sigprocmask(SIG_BLOCK, &ss, NULL);

    pause();
    return 0;
}
```

Questions

- What does this program (not) do?
- To block additional signals in a signal handler, why can we not use `sigprocmask`?

The signal mask

- A signal we want to block might be delivered to the handler **just before masking** is completed!

Solution

- The `struct sigaction` has a member `sa_mask` to specify signals that should be blocked:

```
1 struct sigaction {
2     void (*sa_handler)(int);
3     int sa_flags;
4     sigset_t sa_mask;  /* Additional set of signals to block inside handler */
5
6     /* still incomplete */
7 };
8
```

Example: Implement `sleep`, 1st try

```
1  void wakeup(int s)
2  {
3      (void)s;  /* nothing to do, just mark variable as used */
4  }
5
6  unsigned int sleep1(unsigned int n)
7  {
8      struct sigaction sa;
9
10     memset(&sa, 0, sizeof(sa));
11     sa.sa_handler = wakeup;
12
13     if (sigaction(SIGALRM, &sa, NULL) != 0)
14         return n;            /* not slept at all */
15
16     alarm(n);                /* start the timer */
17     pause();                 /* wait for any signal */
18     return alarm(0);         /* turn off timer, return remaining time */
19 }
```

Question What flaws does this code have?

```c
unsigned int sleep1(unsigned int n)                    /* same code again */
{
    struct sigaction sa;

    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = wakeup;

    if (sigaction(SIGALRM, &sa, NULL) != 0)
        return n;              /* not slept at all */

    alarm(n);                  /* start the timer */
    pause();                   /* wait for any signal */
    return alarm(0);           /* turn off timer, return remaining time */
}
```

- What if there already was a handler for `SIGALRM`, ...
- ...if `SIGALRM` was blocked, or...
- ...if an alarm goes off between the calls to `sigaction` and `pause`?

pause with modified mask

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

- sigsuspend(2) waits for any signal in the set.
  - This is like pause(2) with a **temporarily modified** signal mask.

Motivation Why do we need this? Blocking is not enough.

```
sigset_t newmask, oldmask;

sigemptyset(&newmask);                    /* block SIGALRM and save current signal mask */
sigaddset(&newmask, SIGALRM);
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) err(1, "sigprocmask");

 /* critical region of code: sigaction set up here */

 /* reset signal mask, which should unblock SIGALRM */
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) err(1, "sigprocmask");
 /* window is open */
pause();  /* wait for signal to occur */
 /* continue processing */
```

A more robust implementation of `sleep`

```
1  unsigned int sleep2(unsigned int n)
2  {
3      struct sigaction sa, old_sa;
4      memset(&sa, 0, sizeof(sa));
5      sa.sa_handler = wakeup;
6
7      sigset_t newmask, oldmask;          /* save current mask, block SIGALRM */
8      sigemptyset(&newmask);
9      sigaddset(&newmask, SIGALRM);
10     sigprocmask(SIG_BLOCK, &newmask, &oldmask);
11
12     sigaction(SIGALRM, &sa, &old_sa);      /* missing: handle error, cleanup */
13     alarm(n);
14
15     sigset_t suspmask = oldmask;           /* wait for any signal... */
16     sigdelset(&suspmask, SIGALRM);         /* ...SIGALRM is not blocked... */
17     sigsuspend(&suspmask);                 /* ...only while suspending... */
18                                            /* ...then it is blocked again */
19     unsigned int remain = alarm(0);
20     sigaction(SIGALRM, &old_sa, NULL);  /* reset previous action */
21     sigprocmask(SIG_SETMASK, &oldmask, NULL);  /* reset signal mask */
22     return remain;
23 }
```

Getting more information about the signal

```
#define _POSIX_C_SOURCE 199309L
struct sigaction {
    void (*sa_handler)(int);
    int sa_flags;
    sigset_t sa_mask;  /* Additional set of signals to block inside handler */
    void (*sa_sigaction)(int, siginfo_t *, void *);
};
```

- An **alternative** signal handler can be installed via the `sa_sigaction` member.
- Only assign to **either** `sa_sigaction`, **or** `sa_handler`. Never to both!
- Set `SA_SIGINFO` in `sa_flags`, to indicate the use of `sa_sigaction`.
- **Information** will be passed to `sa_sigaction` in a struct pointed to by `siginfo_t`.
- The `void` pointer can pass information about the context where the signal occurred. Mostly unused.

Signal information

```
siginfo_t {
    int      si_signo;     /* Signal number */
    int      si_code;      /* Signal code */
    pid_t    si_pid;       /* Sending process ID */
    uid_t    si_uid;       /* Real user ID of sending process */
    int      si_status;    /* Exit value or signal */
    int      si_int;       /* value passed by sender, cf. page 291 */
    void    *si_ptr;       /* value passed by sender, cf. page 291 */
     /* many more, cf. sigaction(2) */
}
```

`si_signo` is the signal number.

`si_code` indicates, together with `si_signo`, how the signal was sent, and how the other fields need to be interpreted.

Example `si_code == SI_USER` — sent by `kill`(2).

`si_pid` Sending process ID. Only with `SI_USER`.

`si_uid` Real user ID of sending process. Only with `SI_USER`.

- Some of the members may share storage ⇒ Obey `si_code`.

```c
void action(int s, siginfo_t *si, void *v)
{
    (void)v;     /* not used */

    if (si->si_code == SI_USER)
        printf("got signal %d via kill: pid %d, uid %d\n",
           s, si->si_pid, si->si_uid);
    else
        printf("got signal %d\n", s);
}

int main(void)
{
    printf("pid is %d\n", getpid());

    struct sigaction sa;
    sigemptyset(&sa.sa_mask);
    sa.sa_sigaction = action;
    sa.sa_flags = SA_SIGINFO;

    sigaction(SIGINT, &sa, NULL);  /* install handler for SIGINT */

    while (1) pause();  /* sleep until signal arrives */
    return 0;
}
```

Passing information to the handler

```
1 #define _POSIX_C_SOURCE 199309L
2 #include <signal.h>
3
4 int sigqueue(pid_t pid, int s, const union sigval value);
5
6 union sigval {
7     int   sival_int;
8     void *sival_ptr;
9 };
```

- `sigqueue`(2) sends signal `s` to process `pid`.
- It sets `si_code` to `SI_QUEUE`, and fills `si_int` and `si_ptr` with `sival_int` and `sival_ptr` respectively.
  ⇒ You can pass either a pointer, or an integer.

From the shell

`/bin/kill [-s sig] [-q val] pid...` Allows to send a value in the `si_int` field. This is **not** the shell builtin, *cf.* `kill`(1)

```c
void action(int s, siginfo_t *si, void *v)
{
    (void)v;

    if (si->si_code == SI_USER)
        printf("got signal %d via kill: pid %d, uid %d\n",
           s, si->si_pid, si->si_uid);

    else if (si->si_code == SI_QUEUE)
        printf("queued signal %d: pid %d, uid %d, val %d\n",
           s, si->si_pid, si->si_uid, si->si_int);

    else
        printf("got signal %d\n", s);
}
```

- Sending a signal with a value:

```
$ ./signal1
pid is 10799
queued signal 2: pid 10806, uid 1000, val 666
```

```
$ /bin/kill -INT -q 666 10799
```

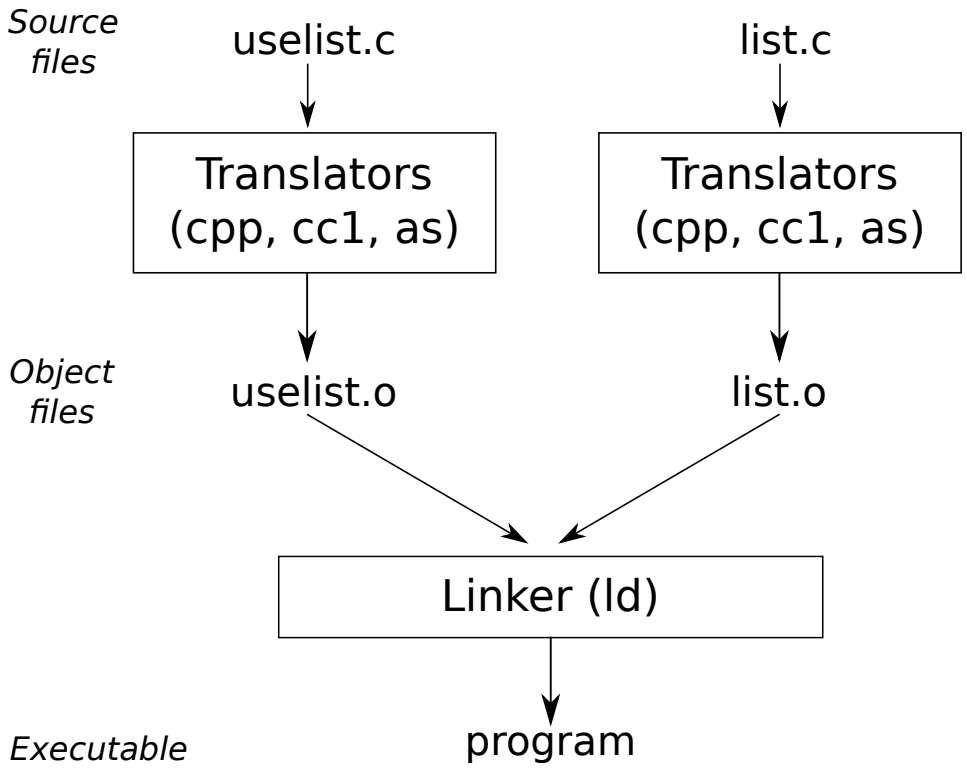
12
Linking

Some of the examples in this chapter are taken from the excellent book

- Randal E. Bryant, David O'Hallaron. *Computer Systems, A Programmer's Perspective*. 2003, Pearson Education, Prentice Hall. ISBN 0-13-178456-0.

12.1 Introduction

- We have already seen how separate compilation works (*cf.* page 183).
- The compiler driver `gcc`(1) employs a bunch of different tools for this task:
  - preprocessor `cpp`(1) — removes comments, applies macros.
  - compiler `cc1` — compiles into assembler code.
  - assembler `as`(1) — translates into binary object file.
  - linker `ld`(1) — **links together the compiled object files**.



- We'll have a closer look at linking now...

Object files

Object files contain chunks of data, (almost) ready to be copied to memory for execution.

- program code, *i.e.*, CPU instructions compiled from your program, and
- constant data (*e.g.*, string literals),

There are three kinds of object files:

- **Executable** object files can be executed directly, *cf.* page 305.
  - Generated by the linker, not by the compiler!
- **Relocatable** object files can be **linked** with other relocatable object files, to form an executable.
  - Symbols may change their position (*cf.* page 313), hence the name.
- **Shared** object files are relocatable object files that can be loaded into memory at runtime, and be shared amongst processes (*cf.* page 329).

The functions, global variables, and `static` variables defined in an object file, can be referred to by name: The **symbols**.

Linker Symbols

Relocatable object files come with a **symbol table**, that lists all the symbols an object file exposes.

- **Global** symbols are defined in the object file, and may be referenced from other object files.
- **External** symbols are referenced by the object file, but not defined. *I.e.*, the definition must be provided in another object file.
- **Local** symbols are defined and referenced only from within the object file.

Note *Local symbols* have nothing to do with function-local variables in a C-program. Unless `static`, they are never visible in the symbol table. (Compare debugger symbols.)

Example

```
1  extern int buf[];
2  int *bufp0 = &buf[0];
3  int *bufp1;
4
5  void swap(void)
6  {
7      int temp;
8      static int count = 42;
9
10     bufp1 = &buf[1];
11     temp = *bufp0;
12     *bufp0 = *bufp1;
13     *bufp1 = temp;
14
15     count++;
16 }
```

```
1  $ pk-cc -c swap.c
2  $ readelf -s swap.o                    # cf. readelf(1)
3  Symbol table '.symtab' contains 18 entries:
4     Num: Size Type    Bind   Ndx Name
5  #...
6       5:    4 OBJECT  LOCAL    3 count.1597
7  #...
8      14:    8 OBJECT  GLOBAL   3 bufp0
9      15:    0 NOTYPE  GLOBAL UND buf
10     16:    8 OBJECT  GLOBAL COM bufp1
11     17:   74 FUNC    GLOBAL   1 swap
```

(some lines and columns have been removed)

- The local symbol `count` (has its name extended to avoid name clashes) uses 4 bytes, and will be stored in section 3 (Section? *cf.* page 309)
- Object `bufp0` uses 8 bytes in section 3, function `swap` uses 74B in section 1.
- `buf` is `UND`efined, *i.e.*, referenced by this module, but we have no idea where it will be in the compiled program.
- `COM`mon objects, like `bufp1` are uninitialized and not yet allocated