Real-time Systems D0003E

Lecture 13: More on inter-process communication (Burns & Wellings ch. 8, 11 & 10.6)

Overview
- Posix signals
- Posix timers
- Timers in cygwin (lab environment)
- Event-loop pattern
- Semaphores

Overview
- Posix signals
- Posix timers
- Timers in cygwin (lab environment)
- Event-loop pattern
- Semaphores

POSIX signals
- A group of threads executing in a protected virtual address space forms the POSIX notion of a process
- A process can thus be thought of as a virtual computer
- The POSIX signal concept is analogous to a virtual interrupt to such a virtual computer
- Signals predate threads and shared memory communication in Unix, although they are now standardized by POSIX
- The default handling of most signals is to kill the receiving process, but user-defined handler can be installed to override this behavior

Some predefined signals
- SIGINT "interrupt" (Control-C)
- SIGKILL illegal instruction (internal)
- SIGALRM timer alarm
- SIGCHLD child process terminated
- SIGTSTP terminal stop (Control-Z)
- SIGUSR1, SIGUSR2
  - user defined
- SIGRTMIN, SIGRTMAX
  - user defined (with extra data payload)

Handling signals
To install a handler for signal X:
```c
#include <signal.h>

void handler(int signo){
   /* Your signal handling code here */
}

struct sigaction act;
act.sa_handler = handler;
... = 0; // extras, normally 0
sigemptyset( &act.sa_mask ); // signals to block while running handler

sigaction( X, &act, NULL );
```

A signal can be sent to a process by executing
```c
kill(pid, sig);
```
where pid is process number (0 means self)
- or by
  ```c
  or by
dividing by zero (equals kill(0, SIGFPE));
  ```
  addressing outside your address space (SIGSEGV)
  ```c
  etc...
  ```
- This actually mimics the interrupt sources on most real architectures:
  - external hardware signals
  - internal hardware misuse

Some predefined signals
- SIGINT "interrupt" (Control-C)
- SIGKILL illegal instruction (internal)
- SIGALRM timer alarm
- SIGCHLD child process terminated
- SIGTSTP terminal stop (Control-Z)
- SIGUSR1, SIGUSR2
  - user defined
- SIGRTMIN, SIGRTMAX
  - user defined (with extra data payload)
Masking a signal

To temporarily mask (disable) signal handling:
```
sigset_t sigs; sigemptyset(&sigs); sigaddset( &sigs, SIGINT );
sigprocmask( SIG_BLOCK, &sigs, NULL );
```

To unmask (enables handling of pending signals):
```
sigprocmask( SIG_UNBLOCK, &sigs, NULL );
```

[C.f. the macros that enable or disable interrupts for various devices on the AVR]

Synchronizing with signals

To wait for the handling of a signal:
```
sigset_t ignore;             // signals to ignore
sigemptyset( &ignore );
sigaddset( &ignore, SIGINT );
sigsuspend( &ignore );   // return after handler has run once
```

Note: sigsuspend stops execution until a signal not mentioned in the argument has arrived and the corresponding handler has run, just as if it were implemented as a busy-wait loop

Alternatively

To wait for a signal that doesn’t have a handler:
```
sigset_t accept; int received; sigemptyset( &accept );
sigaddset( &accept, SIGALRM );
sigaddset( &accept, ... );
sigprocmask( SIG_BLOCK, &accept, NULL );
sigwait( &accept, &received );
```

That is, sigwait stops execution until one of the signals in its first argument has arrived. Note that these signals must be blocked when sigwait is called (so that a handler won’t interfere)

POSIX timers

○ OS abstraction built on top of the system ticks signalled by the clock hardware

○ To create a new timer abstraction:
```
#include <time.h>
timer_t tmr;
struct sigevent ev;
SIGEV_SIGNAL( &ev, signo );
timer_create( CLOCK_REALTIME, &ev, &tmr );
```

These commands amount to “buying” a new alarm clock, aligning it with real time, and configuring it with a certain alarm signal

Setting the alarm time

A timer is armed by supplying a single alarm time and an optional repetition period.

The alarm time can be relative (to the time of the call), or absolute.

To set a timer to ring in one second, and thereafter every 50 milliseconds, write
```
struct itimerspec spec;
spec.it_value.tv_sec = 1;
spec.it_value.tv_nsec = 0;
spec.it_interval.tv_sec = 0;
spec.it_interval.tv_nsec = 50000000;
timer_settime( tmr, 0, &spec, NULL );
```

How alarms should be delivered
```
SIGEV_SIGNAL( &ev, signo ) – doesn’t have to be SIGALRM
```

To set a timer to ring just once, set the `it_interval` to 0
```
spec.it_value.tv_sec = 1;
spec.it_value.tv_nsec = 0;
spec.it_interval.tv_sec = 0;
spec.it_interval.tv_nsec = 0;
timer_settime( tmr, 0, &spec, NULL );
```

To temporarily disable a timer, set the `it_value` to 0
```
spec.it_value.tv_sec = 1;
spec.it_value.tv_nsec = 0;
spec.it_interval.tv_sec = 0;
spec.it_interval.tv_nsec = 0;
timer_settime( tmr, 0, &spec, NULL );
```

More timer settings
A cyclic POSIX thread

```c
void *proc (void *arg) {
timer_t tmr;
sigset_t accept;
struct sigevent ev;
int sig_received;
struct itimerspec spec = {{0,50000000}, {0,50000000}};
SIGEV_SIGNAL( &ev, SIGALRM );
timer_create( CLOCK_REALTIME, &ev, &tmr );
timer_settime( tmr, 0, &spec, NULL );
sigemptyset( &accept );
sigaddset( &accept, SIGALRM );
sigprocmask( SIG_BLOCK, &accept, NULL );
while (1) {
sigwait( &accept, &sig_received );
// do periodic action
}
}
```

N.B.: signals are global within a POSIX process and not tied to a specific thread. This means that for this example to work, no other thread should use this signal (SIGALRM) for any other purpose.

C.f. the periodic reactive object

```c
int m( Class *self, int arg ) {
// do periodic action
AFTER( MILLISEC(500), self, m, exp );
}
```

Apart from the obvious differences in size, there's another, more fundamental, difference in structure as well:

### Reactive object:

- Reactive object:
- **POSIX thread:**

Cygwin timers

As it happens, Cygwin - the PC side lab programming environment - doesn't support `timer_create()` and `timer_settime()` (so much for POSIX compatibility!)

Instead, a predefined interval timer for each process is offered, in the older style of BSD Unix

This timer always generates the signal `SIGALRM`, and is configured via the system call `setitimer()`

Actually, one such interval timer can be configured for each thread, but since they all use SIGALRM for output, using multiple interval timers in a process is a little tricky and not recommended.

A cyclic POSIX thread (2)

```c
void *fun( void *arg ) {
sigset_t accept;
struct itimerval x;
x.it_value.tv_sec = 0;
x.it_value.tv_usec = 1000;
x.it_interval.tv_sec = 0;
x.it_interval.tv_usec = 1000;
setitimer( ITIMER_REAL, &x, NULL );
sigemptyset( &accept );
sigaddset( &accept, SIGALRM );
sigprocmask( SIG_BLOCK, &accept, NULL );
while (1) {
    int sig_received;
sigwait( &accept, &sig_received );
    // do periodic action
}
```

Handling multiple events

```c
int m2( Class *self, int x ) {
// update self
}
```

How do we tell a POSIX thread to wait for a timer signal and be prepared to update its state at the same time?

The event-loop pattern

A standard pattern for encoding reactions in terms of a continuous (POSIX) thread:

```c
void *fun( void *arg ) {
    while (1) {
        INITIALIZE;
        // PAGE
        REACT;
    }
}
```

The "parking" operation is at the core of this pattern - it causes the thread to stop execution indefinitely.

Implicit side-condition: neither INITIALIZE nor REACT must call any parking operation.
Parking operations (def)

- In principle: any operation implemented by busy-waiting on an external state change
- In practice: an operation provided by the underlying operating system that behaves as if it were implemented by busy-waiting on an external state change
- Core questions regarding a parking op: what events does it wait for? Just one? Many? Is it extensible?
- Example: `sigwait` waits for a definable set of POSIX signals (but not for keyboard entry, for example)

Example: `getchar` waits for data from one input source only (the keyboard), but not for any signals.

The multi-event-loop pattern

Ideally we would like a parking operation that waits for exactly those events a thread is interested in:

```c
void *fun( void *arg ) { INITIALIZE; while (1) { x = PARK; switch (x) { case 0: REACT0; break; case 1: REACT1; break; ... case n: REACTn; break; }}}
```

Unfortunately, a truly generic parking op doesn’t exist...

POSIX select()

- A way to wait for input on multiple files (keyboard, pipes, network sockets, etc) simultaneously.
- Returns info on which file descriptor data arrived first.
- Also supports an optional timeout value, equivalent to a multiple wait that includes a delay.

Syntax:

```c
int select( int nfds, fd_set *r, fd_set *w, fd_set *e, struct timeval *t )
```

Sets:

- Can be read
- Can be written
- Have pending exceptions

Original process synchronization device (due to Dijkstra 1965)

- Supports two operations: wait and signal (signal is called post in POSIX)
- wait: if value > 0: decrease; if = 0: block!
- Signal: increase value
- The general semaphore is counting: i.e., it remembers the number of signal calls, and allows the same number of wait calls to succeed without stopping
- A counting semaphore must be initialized with its starting value (the number of initially allowed wait calls)

Semaphore vs. mutex

- A mutex is a semaphore that is:
  - Binary (only distinguishes between the values 0 and 1)
  - Automatically initialized to 1
  - Restricted so that signal/post can only be called by the thread that has successfully called wait
- Mutexes use the name `lock` for wait, and `unlock` for signal

A semaphore Bounded Buffer

```c
#include <semaphore.h>
sem_t mut;
sem_t space, items;
int head = 0, tail = 0;
T buf[SIZE];
...
sem_init( &mut, 0, 1 );
sem_init( &items, 0, 0 );
sem_init( &space, 0, SIZE );
```

Notice the lock/unlock pattern
Semaphores for parking

- General pattern:
  - One thread produces tokens by means of the signal call.
  - Another thread consumes tokens by means of the wait call, and parks if none are available.
  - The initial number of tokens can be any number ≥ 0.
- This parking pattern is often referred to as conditional synchronization.
- Notice: the producer and the consumer must agree on what the tokens mean - i.e., number of items in a buffer, the fact that a variable is not empty, the fact that one of two variables is not empty, etc.

Generic parking

- In general, a generic parking mechanism cannot be implemented under POSIX.
- Instead, the established coding style is to use separate threads for each kind of event, and rely on shared memory and semaphores to synchronize these threads in turn.
- Some operating systems like QNX offer a message mechanism as an alternative, but even here the parking operation is not fully extensible to handle signals, etc.
- Reactive objects as we know them can therefore not be directly supported in POSIX.
- Neither can our direct interrupt handling, and the notion of baselines and deadlines...
- But of course: POSIX programs are much more portable!