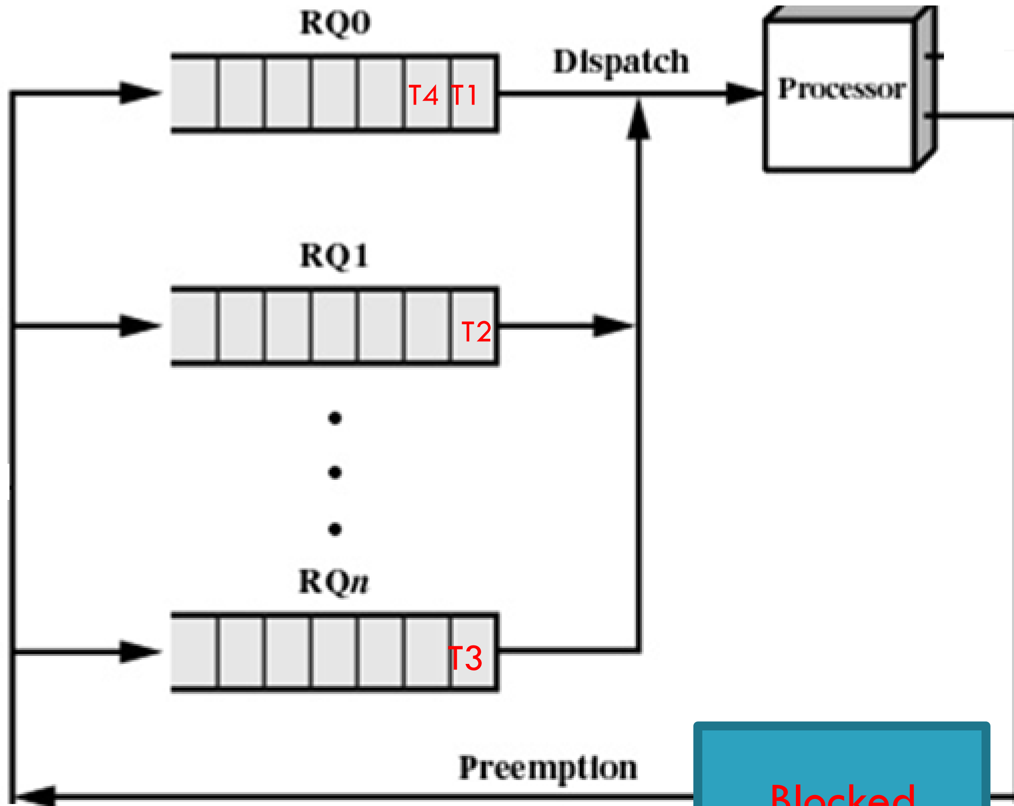
CT420 REAL-TIME SYSTEMS

POSIX - SIGNALS

Dr. Michael Schukat

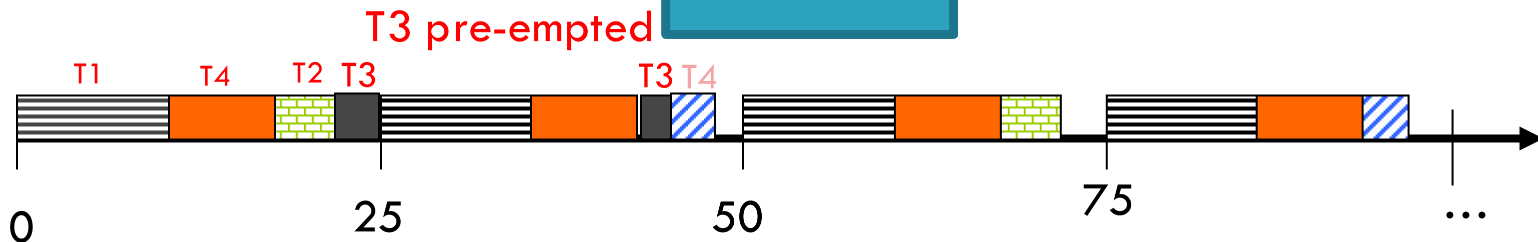
Ollscoil na Gaillimhe
University of Galway

Recap: Task Invocation using Timer

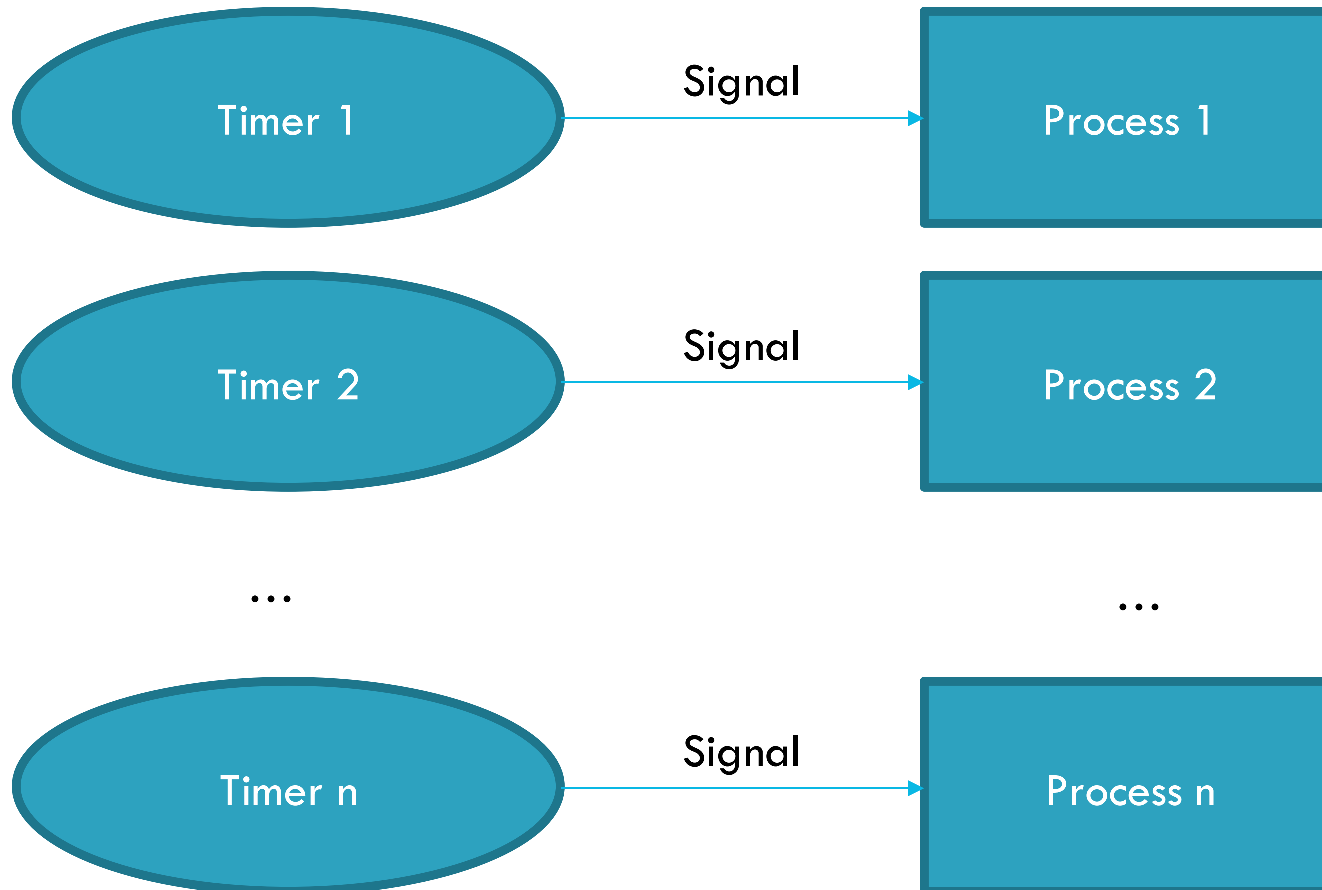



```
Process Tx:
int main() {
// Initialise process
// Setup timer x to notify Tx
// about begin of every cycle, e.g.
// T1: 25ms; T2 = 50ms; T3 = 100ms
while (1) {
  do_something();
  block_until_timer_signal();
}
}
```

T3 pre-empted



Timers, Processes and Signals



POSIX Signals

- Signals are an integral part of Unix/POSIX
- They are the software equivalent of an interrupt
- Signals are used for
  - Exception handling (e.g., division by zero)
    - Termed **synchronously-generated** as occur in response to something the process itself does
  - **Asynchronous** event occurrence notification
    - Asynchronous as happens external to process execution
    - E.g., Timer expiration, I/O completion,
    - CTRL-C (process terminate)
  - (Rudimentary) mechanism for inter-process communication (one option)

POSIX Signal Terminology

- ACTION or DISPOSITION for signal
  - 1. Ignore
  - 2. Catch → write a handler function or
  - 3. Default action → usually terminate process
- Signal is GENERATED for a process (or sent to a process) when the event that causes the signal occurs
- Signal is DELIVERED to a process when action for a signal is taken
  - In interim between GENERATION and DELIVERY, signal is said to be PENDING

POSIX Signal Terminology

- Process can BLOCK delivery of a signal → signal remains PENDING until unblocked
- Signals <u>may</u> be blocked
  - To ensure that critical sections of code are not interrupted
  - Signal can then be unblocked when out of critical section
- Signal mask defines set of signals currently BLOCKED from DELIVERY to that process.
  - E.g., with 1 bit / signal, signal is blocked if bit is 'ON'
- Some OS-generated critical signals cannot be blocked (e.g. process termination)

Signal Masks

- Data structure that contains 1 bit per signal, e.g. unsigned 32-bit int (*val* in macro) and 4 signals

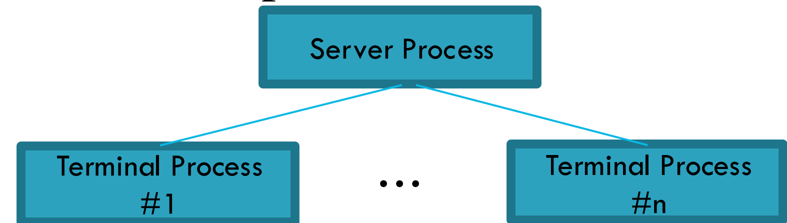
```
#define SIGNAL1        1
#define SIGNAL2        2
#define SIGNAL3        4
#define SIGNAL4        8
#define SET(val, signal)        val |= signal
#define TEST(val, signal)       ((val & signal) != 0)
```

Masking Signals

```
/* define a new mask set */
sigset_t mask_set;
/* first clear the set (i.e. make it contain no signal numbers) */
sigemptyset(&mask_set);
/* Add the TSTP and INT signals to our mask set */
sigaddset(&mask_set, SIGTSTP);
sigaddset(&mask_set, SIGINT);
/* Remove the TSTP signal from the set. */
sigdelset(&mask_set, SIGTSTP);
/* Check if the INT signal is defined in our set */
if (sigismember(&mask_set, SIGINT)
    printf("signal INT is in our set\n");
else
    printf("signal INT is not in our set\n");
/* finally, let's make the set contain ALL signals available on our system */
sigfillset(&mask_set);
```

Signal Terminal-Server Example (Server Code, single Process)

- Idea: server process shuts down a number of terminal processes (children), it has previously created:




```
#define SIG_GO_AWAY SIGUSR1
void shutdown_server(void)
{
  printf("Shutting down server\n");
  // Now kill all children with signal to process group
  kill(0,SIG_GO_AWAY); // Notify terminal processes
}
```

- Send signal `SIG_GO_AWAY` using `kill()` from server to all terminals
  - 1st arg 0 ➔ all processes in process group signalled
- `SIG_GO_AWAY is alias for` **`SIGUSR1`**, one of 2 signals available to programmers with POSIX.1

FYI: Process Group

- In a POSIX-conformant OS, a process group denotes a collection of one or more processes
- It is used to control the distribution of a signal; when a signal is directed to a process group, the signal is delivered to each process that is a member of the group
- The system call **setsid()** is used to create a new single (new) process group, with the current process as the **process group leader**
- Process groups are identified by a positive integer, the process group ID, which is the process identifier of the process that is (or was) the process group leader
- Process groups need not necessarily have leaders, although they always begin with one
- While POSIX prohibits the change of the process group ID of a session leader, the system call **setpgid()** sets the process group ID of a process, thereby typically joining the process to an existing process group

Signal Terminal-Server Example: **Terminal Code** (potentially multiple Processes)

```
#define SIG_GO_AWAY SIGUSR1
//signal handler
void terminate_normally(int signo) {
exit(0);
}
main() {
   struct sigaction sa;
   sa.sa_handler=terminate_normally; // Signal handler
   sigemptyset(&sa.sa_mask);// Set of signals to be blocked
                            // during execution of handler

   sa.sa_flags=0; // Later
   if(sigaction(SIG_GO_AWAY, &sa, NULL)) {
      perror("sigaction");
      exit(1);
   }
   while(1){
     // Carry out normal terminal duties, e.g. wait on user input
     …
   }
}
```

struct sigaction

- `struct` used to set all the details of what your process should do when a particular signal arrives
- Used with `sigaction` signal function (identical names !?)

```
struct sigaction{
  void (*sa_handler)();
  sigset_t sa_mask;
  int sa_flags;
  void(*sa_sigaction)(int, siginfo_t *, void *);
};
```

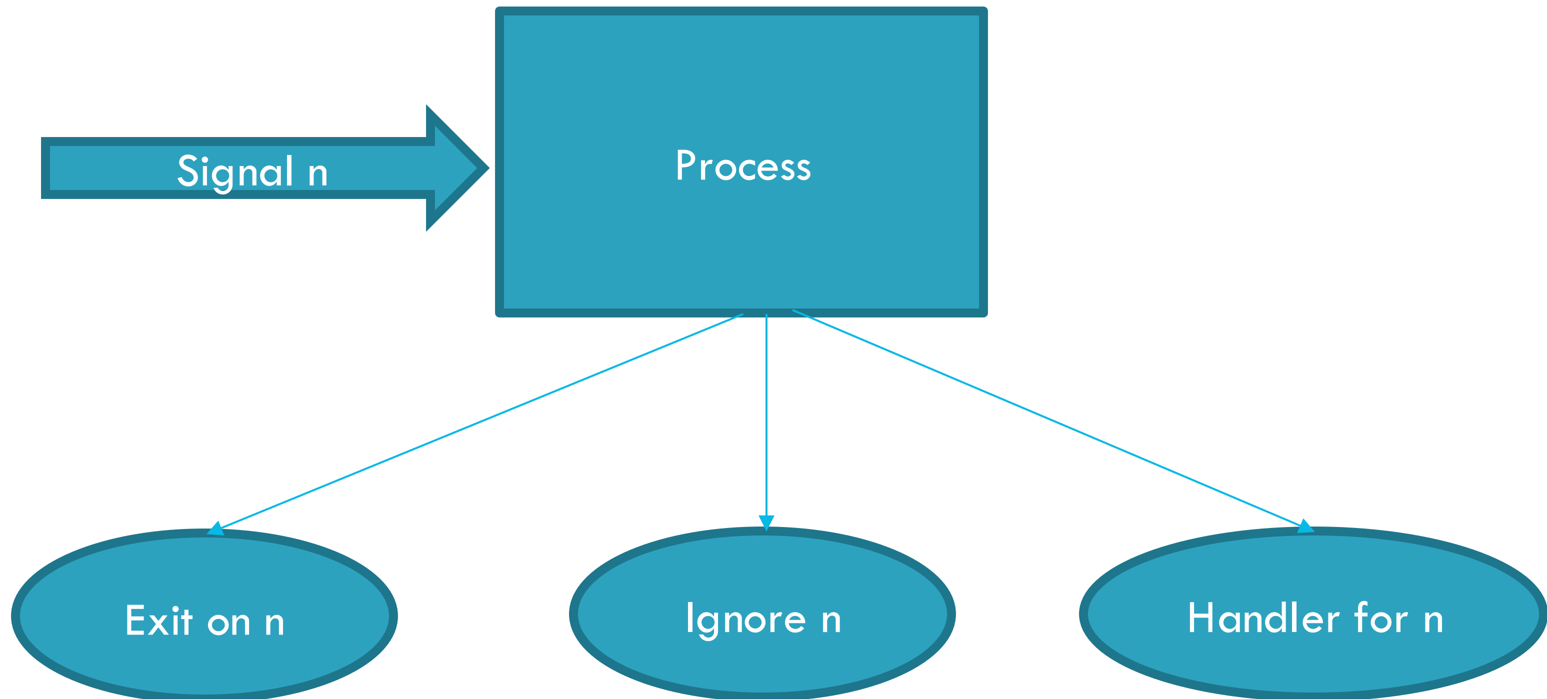
POSIX.4 (later!)

- 1st member can be `SIG_DFL` (default action), `SIG_IGN` (ignore) or take a pointer to a function `sa_handler` (used for POSIX.1 signals)

Signal Terminal-Server Example

- Child (terminal) sets up a signal handler using `sigaction` signal function
  - 3 arguments specify
    - signal to wait for
    - struct sigaction
    - old sigaction
  - When signal delivered from server, terminal is terminated gracefully using function `exit()`

Signals and Process Behaviour

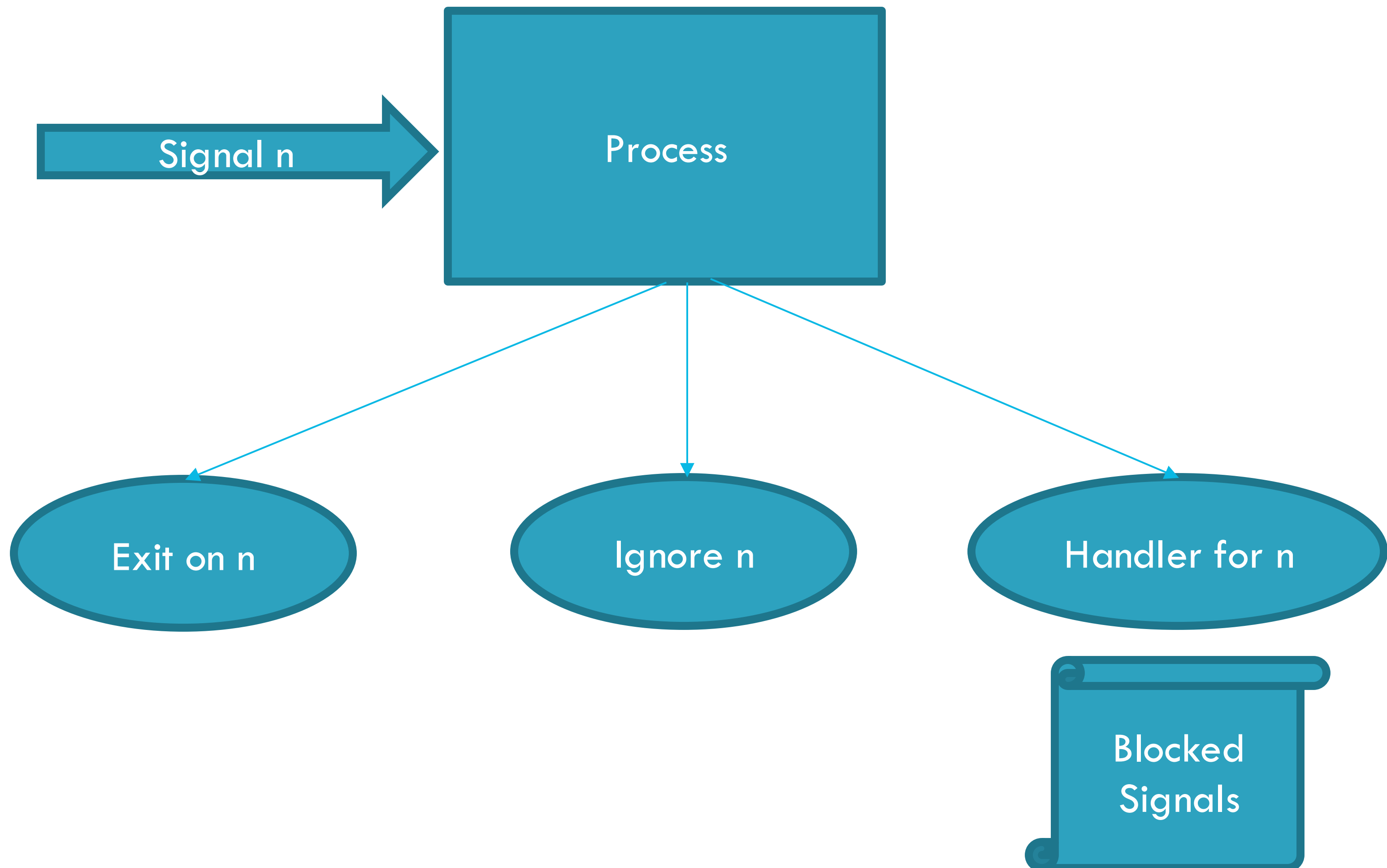


`struct sigaction`

- `sa_mask` used to define set of signals to be blocked from delivery **while handler is executing**
  - Overall mask in operation =
    - mask in effect for process (e.g., inherited) + signal being delivered + signals specified in `sa_mask`
    - Signal being delivered included to avoid 2nd occurrence whilst handling 1st
  - Note : Some signals eg. `SIGKILL, SIGSTOP` cannot be blocked

- 3rd member is `sa_flags` .. See POSIX.4
- 4th member is `sa_sigaction`
  - Similar to `sa_handler` but used for queued POSIX.4 signals

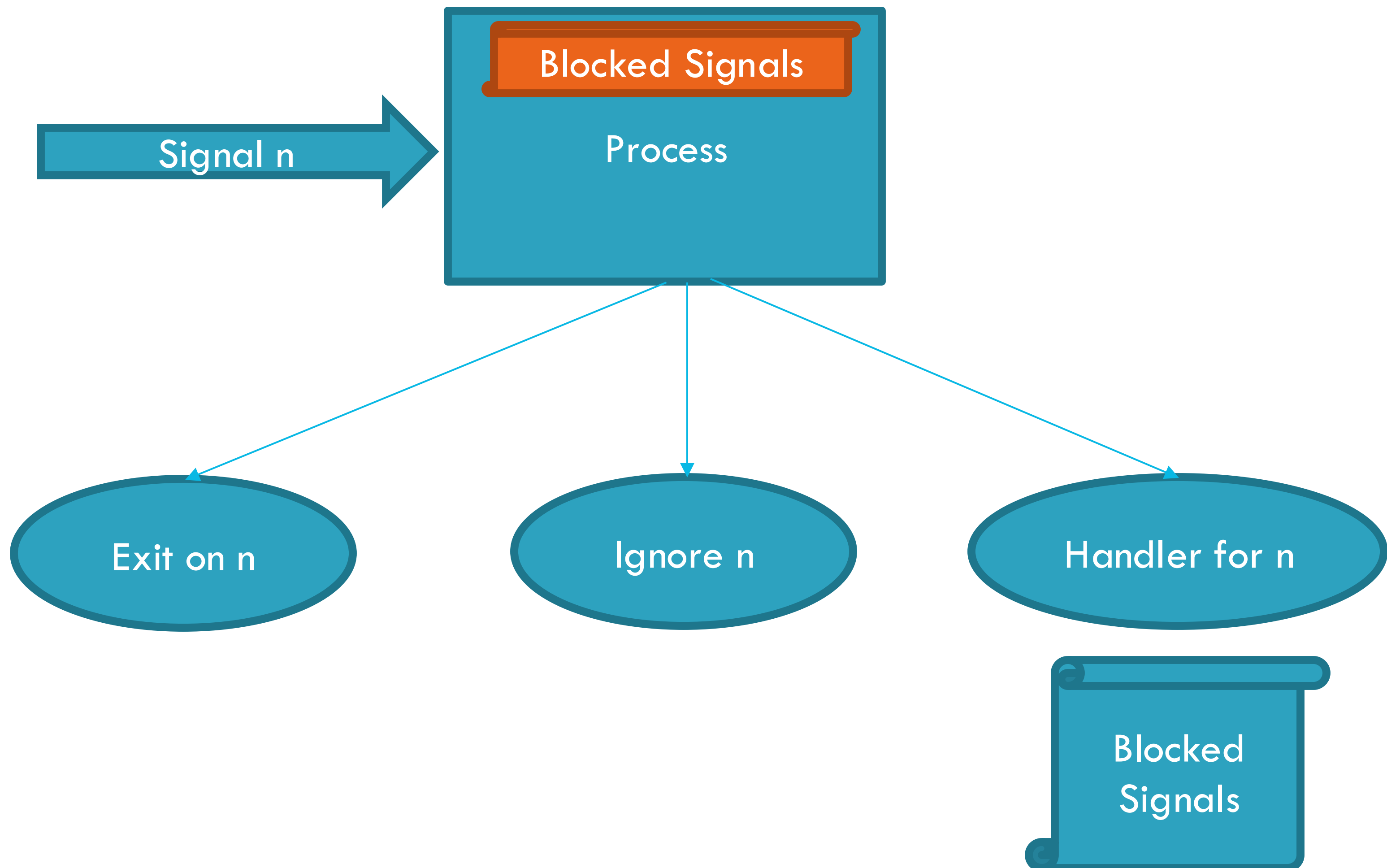
Signals and Process Behaviour



Signal Mask

- Each POSIX process has an associated signal mask
  - Signals which will be blocked (held pending) if they are generated until unblocked
    - Will be delivered once unblocked
    - `Note: sa_mask` sets mask **while handler is executed**
  - What about setting mask in program code?
    - **`sigprocmask(1st arg, &newest,&oldest)`**
      - 1st arg can be `SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK`
      - Newest is set of signals (type sigset_t) that you are adding for blocking/unblocking from mask or for setting mask
    - Use `sigemptyset(),sigfillset()` etc. to modify signal sets

Signals and Process Behaviour



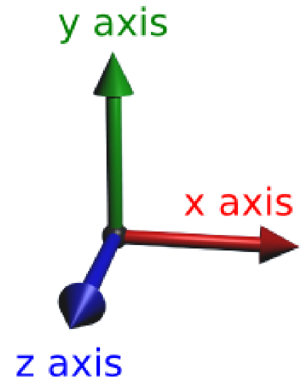
Case Study

- A flying Mars robot (let's call it Ingenuity) https://www.youtube.com/watch?v=NHMIgQ5RAI8

  has a build-in gyroscope connected to the CPU via some interface
  - A device for measuring the robot's orientation and angular velocity
- The robot's orientation is controlled by a process
- The process reads the gyroscope every 50ms using a signal handler invoked by a timer signal (signal A)
- However, during the landing phase the gyro is also read every time that one of the robot's legs touches the ground
  - To make sure that the robot is parallel to the flat ground and doesn't topple (or damage it blades…)
- Here the asynchronous signal B ("robot touches ground") calls a second signal handler that reads the gyroscope
- The gyro can only be accessed by one handler at a time

Case Study

- Signal handler A checks the entire orientation of the drone (every 50 ms), while signal handler B checks only for horizontal alignment (parallel to the ground, i.e. X and Z axes), a rotation around the Y axis doesn matter
- **Why must each handler mutually block the other handler?**




```
SignalHandlerA()
{
   // Get drone orientation in
   // space relative to X, Y, and
   // z axis:
   // I.e., send command:
   // “Get X-Y-Z orientation”
   // ...
   // Read X, Y and Z
}
```

```
SignalHandlerB()
{
   // Get drone orientation in
   // space relative to X and Y
   // axis only:
   // I.e., send command:
   // “Get X-Z orientation”
   // ...
   // Read X and Z
}
```

Case Study

- Correct Sequence:

- Incorrect Sequence:

Recap: Signal Terminal-Server Example: Terminal Code (potentially multiple Processes)

```
#define SIG_GO_AWAY SIGUSR1
//signal handler
void terminate_normally(int signo) {
exit(0);
}
main(int argc, char **argv) {
   struct sigaction sa;
   sa.sa_handler=terminate_normally; // Signal handler
   sigemptyset(&sa.sa_mask);// Set of signals to be blocked
                            // during execution of handler

   sa.sa_flags=0; // Later
   if(sigaction(SIG_GO_AWAY, &sa, NULL)) {
   perror("sigaction");
   exit(1);
   }
   while(1){
   // Carry out normal terminal duties, e.g. wait on user input
   …
   }
}
```

Option: Return last signal handler

sigsuspend( )

- Terminal code
  - Infinite `while(1)` loop
    - Wasting CPU cycles
  - Useful to be able to put terminal to sleep and wait for something to happen
    - e.g. signal from server to indicate it has completed some work
  - Need to make sure that signal cannot arrive before process is put to sleep, i.e.

```
while(sig_received == false)
    pause(); //waiting for a signal
```

    - Could get scenario where sig_received is TRUE just after above check but before process is put to sleep
      - Waiting for signal that has just previously arrived
      - ➔ sleep forever!
    - Need to block signal until process is put to sleep
  - `sigsuspend(&signal_mask)` facilitates this

`sigsuspend( )`

- Installs `signal_mask` as process mask AND puts process to sleep in atomic operation
  - Keep signal blocked until process put to sleep
  - `sigsuspend()` unblocks and sleeps atomically
- Halts execution until unblocked signal (e.g. not set in signal_mask) arrives
- Process woken up and signal handler called
- When signal handler returns, `sigsuspend()` returns and original signal mask is set for process

Example: sigsuspend( )

More complex server / terminal

```
#define SIG_GO_AWAY SIGUSR1 // as before
// 2nd signal
#define SIG_QUERY_COMPLETE SIGUSR2

//2nd signal handler
void query_has_completed(int signo){
…
}
void terminate_normally(int signo){
exit(0);
}
```

Example: sigsuspend( )

```
main(int argc, char **argv){
    struct sigaction sa, sa2;
    sigset_t wait_for_these;
    sa2.sa_handler=query_has_completed;
    sigemptyset(&sa2.sa_mask);
    sa2_sa_flags=0;
    if(sigaction(SIG_QUERY_COMPLETE, &sa2, NULL)) {
            perror("sigaction");
            exit(1);
    }
    sigemptyset(&wait_for_these);
    sigaddset(&wait_for_these, SIG_QUERY_COMPLETE);
    sigprocmask(SIG_BLOCK, &wait_for_these, NULL);
   // SIG_QUERY_COMPLETE now blocked ..reset signal set
    sigemptyset(&wait_for_these);
    //other signal handling code for SIG_GO_AWAY
    while(1){
            …
            sigsuspend(&wait_for_these);//unblock signal and sleep
    }
}
```

Example: sigsuspend( )

- Here: Used to protect critical code section from SIGINT:

```
sigset_t newmask,oldmask,waitmask;
//set up signal handler for SIGINT via sigaction etc.
sigemptyset(&waitmask);
sigaddset(&waitmask,SIGUSR1);
sigemptyset(&newmask);
sigaddset(&newmask,SIGINT);
sigprocmask(SIG_BLOCK, &newmask, &oldmask);
// Enter critical section.. SIGINT blocked
// …
// Leave critical section
sigsuspend(&waitmask); //process sleeps, SIGINT
   unblocked,SIGUSR1 blocked
//When SIGINT arrives, signal handler called and
   sigsuspend() returns, restores mask to that prior i.e.
   SIGINT now blocked, SIGUSR1 now unblocked
//Now reset old mask .. Both  unblocked
sigprocmask(SIG_SETMASK,&oldmask, NULL);
//continue
```

POSIX.4 Signals

- Addresses some of limitations of POSIX.1
  - More signals
    - SIGRTMIN to SIGRTMAX (minimum 8)
      - Specified in RTSIG_MAX
      - Decreasing priority in order of delivery if more than 1 pending
  - Real-time signals are delivered in a guaranteed order
  - Can queue signals → can see if more than one has occurred during a signal blocked period
    - POSIX.1 does not queue signals
    - Implemented via `sa_flags` member of `sigaction struct`
      - Set `SA_SIGINFO` bit in `sa_flags`

struct sigaction revisited

- This structure is used to set all the details of what your process should do when a particular signal arrives
- Used with `sigaction` signal function

```
struct sigaction{
  void (*sa_handler)();
  sigset_t sa_mask;
  int sa_flags;
  void(*sa_sigaction)(int, siginfo_t *, void *);
};
```

- 1st member can be `SIG_DFL` (default action), `SIG_IGN` (ignore) or take a pointer to a function `sa_handler` (used for POSIX.1 signals)

POSIX.4 Signals

- Separate signal handler method **for queued signals**
  - Recall 4th member of `sigaction struct`
    - `*sa_sigaction: pointer to sig handler function`
  - Signal handler has 3 arguments

```
void handler(int signum,siginfo_t *data, void *extra)
```

    - Recall POSIX.1 signal handler has 1 argument
  - Set `SA_SIGINFO` **bit in** `sa_flags` **in sigaction() to select new handler over POSIX.1 handler**
  - `data` **is structure with various member fields**
    - signal number, signal value, cause of signal (e.g., timer)
- **Queued signals can deliver more data**

siginfo_t

```
typedef struct {
...
int si_signo; // Signal id as before
int si_code;  // Who sent signal?
              //See slide “Constants for si_code”
union sigval si_value; // See also next slides
...
} siginfo_t;
union sigval {
...
int sival_int;
void *sival_ptr;
...
) sigval;
```

Unions in C

```c
#include <stdio.h>
#include <stdlib.h>
main() {
    union {
    float y;
    char x;
    } e;

    e.y = 23.5;
    printf("value is %f\n", e.y);
    e.x = 5;
    printf("value is %d\n", e.x);
    printf("value is %f\n", e.y);
    exit(EXIT_SUCCESS);
}
```

Program output:

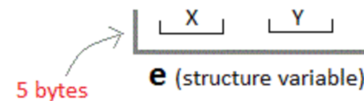
value is 23.5

value is 5

value is 327394.343

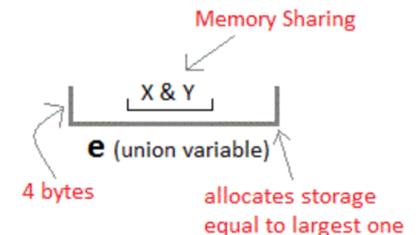
Structure

```c
struct Emp
{
 char X ;    // size 1 byte
 float Y ;   // size 4 byte
}e ;
```



Unions

```c
union Emp
{
 char X ;
 float Y ;
}e ;
```



Unions in C: How to specify Data Type stored

```
#include <stdio.h>
#include <stdlib.h>
/* code for types in union */
#define FLOAT_TYPE 1
#define CHAR_TYPE 2
#define INT_TYPE 3
struct var_type {
    int type_in_union;
    union {
            float un_float;
            char un_char;
            int un_int;
            } vt_un;
    } var_type;
```

Constants for siginfo_t->si_code

- **SI_QUEUE**
  Signal was sent by sigqueue() (next slide)
- **SI_TIMER**
  Signal was generated by expiration of a timer set by timer_settimer() (as seen before)
- SI_ASYNCIO
  Signal was generated by completion of an asynchronous I/O request (not important for us)

sigqueue()

- **int sigqueue(pid_t pid, int sig, const union sigval value);**
- The sigqueue() function sends a signal to a process or a group of processes that *pid* specifies along with the value specified by *value*.
- The signal to be sent is specified by *sig*

Example: Server Code

```
#define SIG_QUERY_COMPLETE SIGRTMIN

void
send_reply(request_t r)
{
    union sigval sval;

    /* Send a notification to the terminal */
    sval.sival_ptr = r->r_params;
    if (sigqueue(r->r_requestor, SIG_QUERY_COMPLETE, sval) < 0)
        perror("sigqueue");
}
```

Example: Client Code (I)

```
#define SIG_QUERY_COMPLETE SIGRTMIN

void
query_has_completed(int signo, siginfo_t *info, void *ignored)
{
    /* Deal with query completion.  Query identifier could
     * be stored as integer or pointer in info. */

    void *ptr_val = info->si_value.sival_ptr;
    int int_val = info->si_value.sival_int;

    printf("Val %08x completed\n", int_val);
    return;
}

main(int argc, char **argv)
{
    struct sigaction sa;

    sa.sa_handler = terminate_normally;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
```

Example: Client Code (II)

```
    if (sigaction(SIG_GO_AWAY, &sa, NULL)) {
        perror("sigaction");
        exit(1);
    }

    sa.sa_sigaction = query_has_completed;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO;    /* This is a queued signal */
    if (sigaction(SIG_QUERY_COMPLETE, &sa, NULL)) {
        perror("sigaction");
        exit(1);
    }
...
```

`struct sigevent`

- Server-Terminal example
  - POSIX.1 signals delivered via `kill( )`
- POSIX.4 signals can be generated by:
  - `sigqueue()` … similar to `kill()` in above example
    - Facilitates extra data required .. `signal value`
  - Timer expiration
  - Completion of asynch I/O
  - Message queues (not covered here)
- Last 2 scenarios
  - A process can generate signals including data payload via `sigqueue()`
  - **Asynchronous events (e.g. timer) use `sigevent`**

struct sigevent

- struct sigevent {
  int sigev_notify; // **must be SiGEV_SIGNALS**
  int sigev_signo; // **SIGRTMIN to SIGRTMAX**
  union sigval sigev_value; // Value for RT signal
  …
  };
- union sigval {
  int sival_int; /* Integer value */
  void *sival_ptr; /* Pointer value */
  }

Example

- Interval Timer example

```
timer_t created_timer;
sigevent se;
// Init se
…
i = timer_create(CLOCK_REALTIME, &se ,  &created_timer);
struct itimerspec new,old;
new.it_value.tv_sec=1;
new.it_value.tv_nsec=0;
new.it_interval.tv_sec=0;
new.it_interval.tv_nsec=100000;
i=timer_settime(created_timer, 0,&new, &old)
..
i=timer_delete(created_timer);
```

`signals & timers:` Summary

- Need to create & configure timer settings
  - `timer_create(), timer_settime()`
  - `struct sigevent`
    - Details of signal to be sent upon timer expiration
- Need to set up signal handler
  - `sigaction()` to describe what signal to wait for and what to do when it arrives
- Avoid resource wasting via polling
  - `sigsuspend()` to put process to sleep and wait for signal
  - Implement signal blocking correctly