Signals & Pipes

Emmanuel Fleury
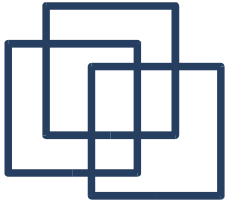
B1-201

fleury@cs.aau.dk
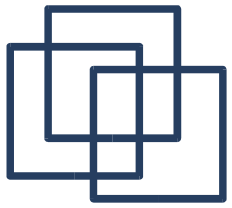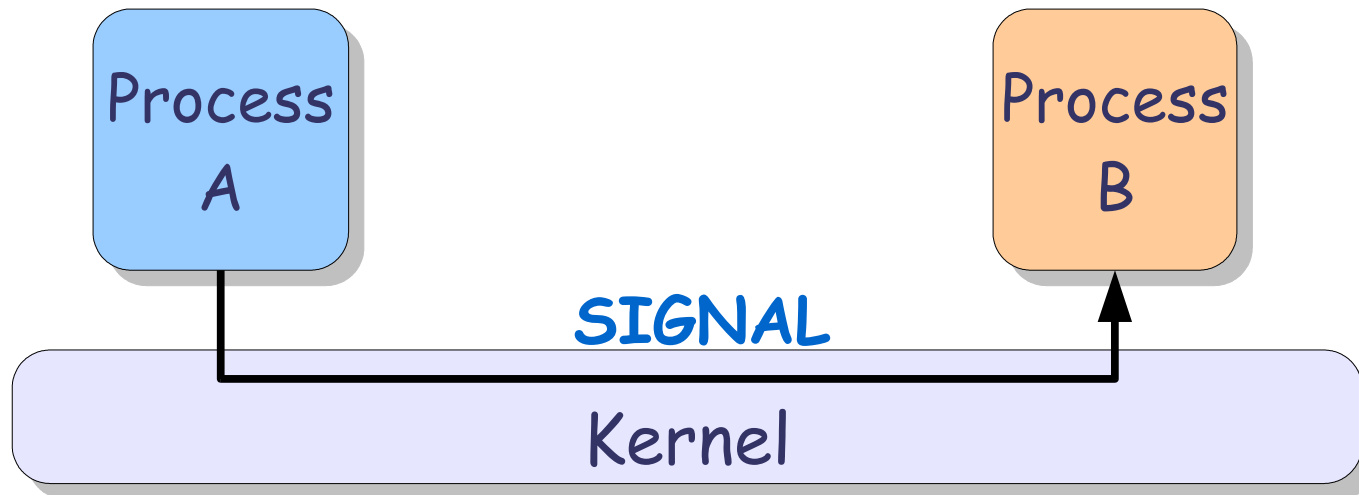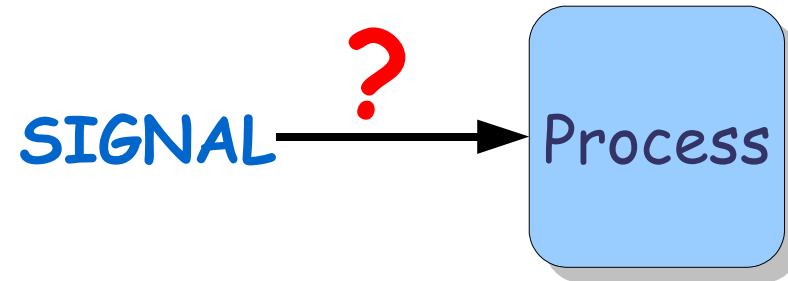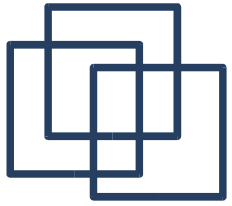
Signals

What is a Signal ?

Most primitive form of inter-process communication

- A signal is:

  - Sent by a process

  - Received by another process

  - Carried out by the kernel

What can we do with it ?

SIGNAL $\longrightarrow$ **?** Process

- When receiving a signal, a process can:

  - Perform the default behaviour of this signal

  - Intercept the signal and perform some custom behaviour or ignore it.

- Default behaviours are:

  - **Ignore**: Ignored by the process

  - **Stop**: Stop the process

  - **Exit**: Terminate the process

  - **Core**: Terminate the process and dump a core
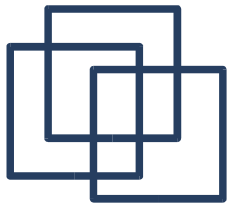
POSIX Signals

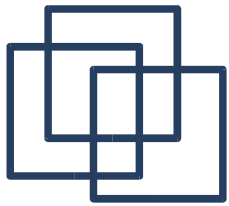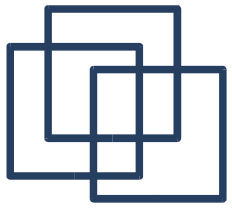| Name	Code	Dflt	Comments
SIGHUP	1	Exit	Hangup
SIGQUIT	3	Core	Terminal quit
SIGTRAP	5	Core	Trace trap
SIGKILL	9	Exit	Kill (can't be caught or ignored)
SIGUSR1	10	Exit	User defined signal 1
SIGUSR2	12	Exit	User defined signal 2
SIGPIPE	13	Exit	Broken pipe (write on a pipe with no reader)
SIGALRM	14	Exit	Notify the end of a timer
SIGCHLD	17	Ignore	Child process has exited or stopped
SIGCONT	18	Restart	Continue execution if stopped
SIGSTOP	19	Stop	Stop execution (can't be caught or ignored)
SIGTSTP	20	Stop	Terminal stop signal
SIGTTIN	21	Stop	Background process trying to read from TTY
SIGTTOUT	22	Stop	Background process trying to write from TTY

ANSI Signals

All the signals used to report program failures

Name	Code	Dflt	Comments
SIGINT	2	Exit	Terminal interrupt (Ctrl-C)
SIGILL	4	Core	Illegal instruction
SIGFPE	8	Core	Floating point exception
SIGSEGV	11	Core	Segmentation Fault (invalid memory access)
SIGTERM	15	Exit	Termination
SIGSTKFLT	16	Exit	Stack Fault

BSD/System V Signals

| Name	Code	Dflt	Comments
SIGABRT	6	Core	Abort the process
SIGBUS	7	Core	Bus error
SIGURG	23	Ignore	Urgent condition on socket
SIGXCPU	24	Core	CPU limit exceeded
SIGXFSZ	25	Core	File size limit exceeded
SIGVTALRM	Exit	Exit	Virtual alarm clock
SIGPROF	Exit	Exit	Profiling alarm clock
SIGWINCH	28	Ignore	Windows size change
SIGIO	29	Exit	Pollable event (SIGPOLL)
SIGPWR	30	Ignore	Power failure restart
SIGSYS	31	Core	Bad argument to routine
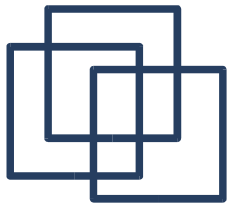
Real-time Signals

Linux supports 32 real-time signals as defined in the POSIX.4 real-time extensions

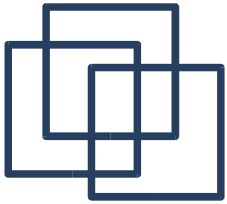| Name	Code	Dflt	Comments
SIGRTMIN+N	33-48	Ignore	N in [0,15]
SIGRTMAX-N	49-64	Ignore	N in [0,15]

Note: These signals have no predefined meaning.

Example: SIGRTMIN+7, SIGRTMAX-7
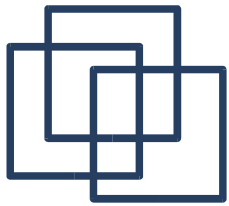
Available Signals (kill -l)

```
[fleury@hermes]$ kill -l
 1) SIGHUP         2) SIGINT         3) SIGQUIT        4) SIGILL
 5) SIGTRAP        6) SIGABRT        7) SIGBUS         8) SIGFPE
 9) SIGKILL       10) SIGUSR1       11) SIGSEGV       12) SIGUSR2
13) SIGPIPE       14) SIGALRM       15) SIGTERM       17) SIGCHLD
18) SIGCONT       19) SIGSTOP       20) SIGTSTP       21) SIGTTIN
22) SIGTTOU       23) SIGURG        24) SIGXCPU       25) SIGXFSZ
26) SIGVTALRM     27) SIGPROF       28) SIGWINCH      29) SIGIO
30) SIGPWR        31) SIGSYS        33) SIGRTMIN      34) SIGRTMIN+1
35) SIGRTMIN+2   36) SIGRTMIN+3    37) SIGRTMIN+4    38) SIGRTMIN+5
39) SIGRTMIN+6   40) SIGRTMIN+7    41) SIGRTMIN+8    42) SIGRTMIN+9
43) SIGRTMIN+10  44) SIGRTMIN+11   45) SIGRTMIN+12   46) SIGRTMIN+13
47) SIGRTMIN+14  48) SIGRTMIN+15   49) SIGRTMAX-15   50) SIGRTMAX-14
51) SIGRTMAX-13  52) SIGRTMAX-12   53) SIGRTMAX-11   54) SIGRTMAX-10
55) SIGRTMAX-9   56) SIGRTMAX-8    57) SIGRTMAX-7    58) SIGRTMAX-6
59) SIGRTMAX-5   60) SIGRTMAX-4    61) SIGRTMAX-3    62) SIGRTMAX-2
63) SIGRTMAX-1   64) SIGRTMAX
[fleury@hermes]$ kill -l SIGKILL
9
[fleury@hermes]$ kill -l 9
KILL
```
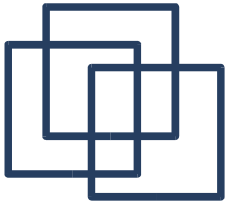
SIGKILL

Note: This signal can't be caught nor ignored !!!
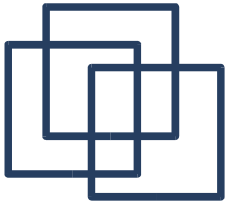Except if the process is in the state "uninterruptible sleep".

Uninterruptible sleep ('D'): The process has requested an operation from the kernel and this operation has been suspended for some reason(s). Usually the operation is waiting for some external events, e.g., an interrupt from the hard-drive when an I/O operation is completed. If something wrong happen with the hardware or the kernel data-structures (due to some bug in the kernel). There is no way to kill a process stuck in the kernel ('D') state. You should either reboot or just kill the parent of this process (if this is init, then you have a problem).
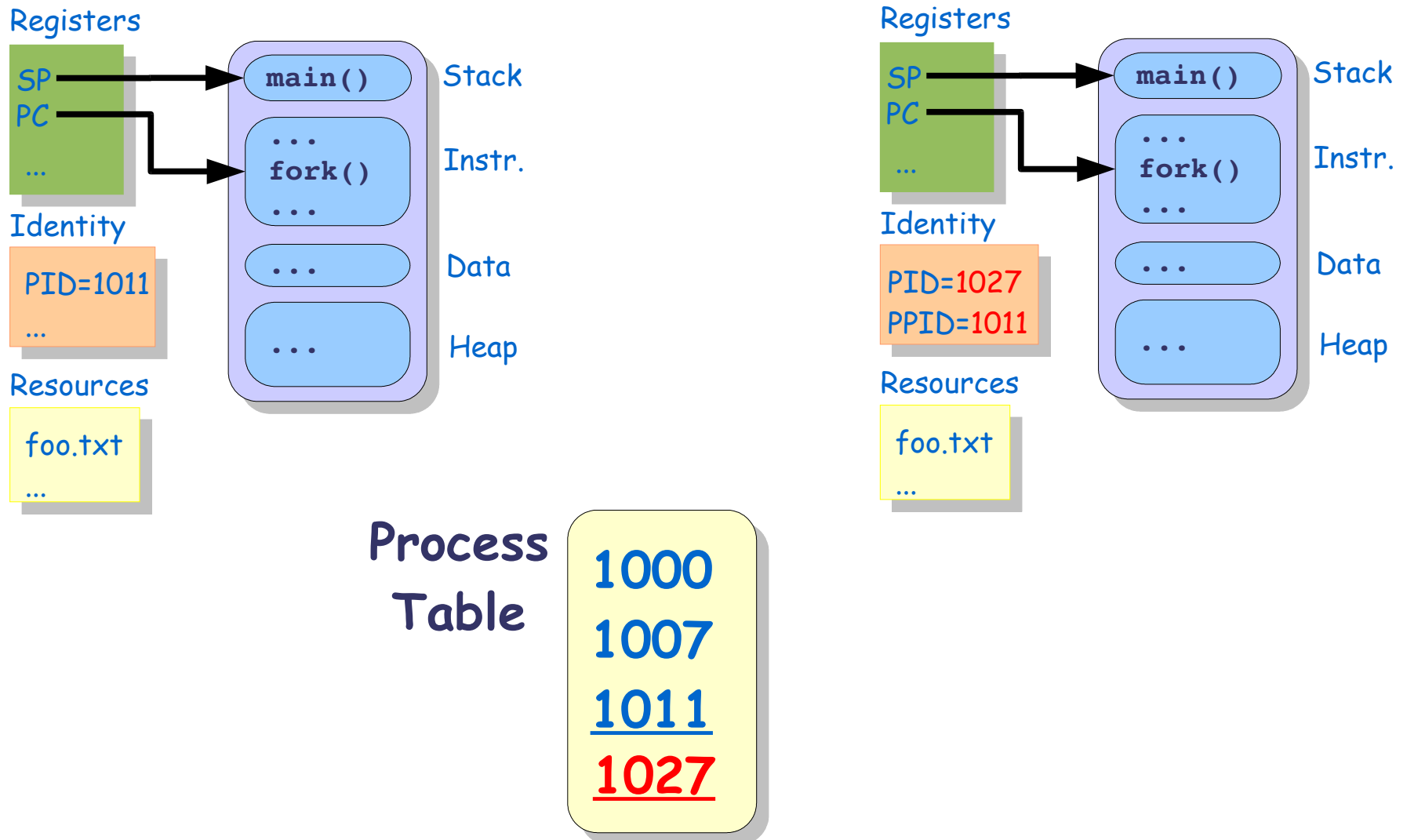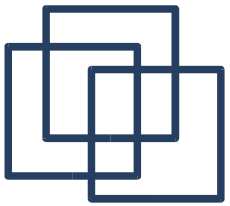
SIGSEGV

Program

SIGSEGV

Kernel

Program
Memory

?

Memory
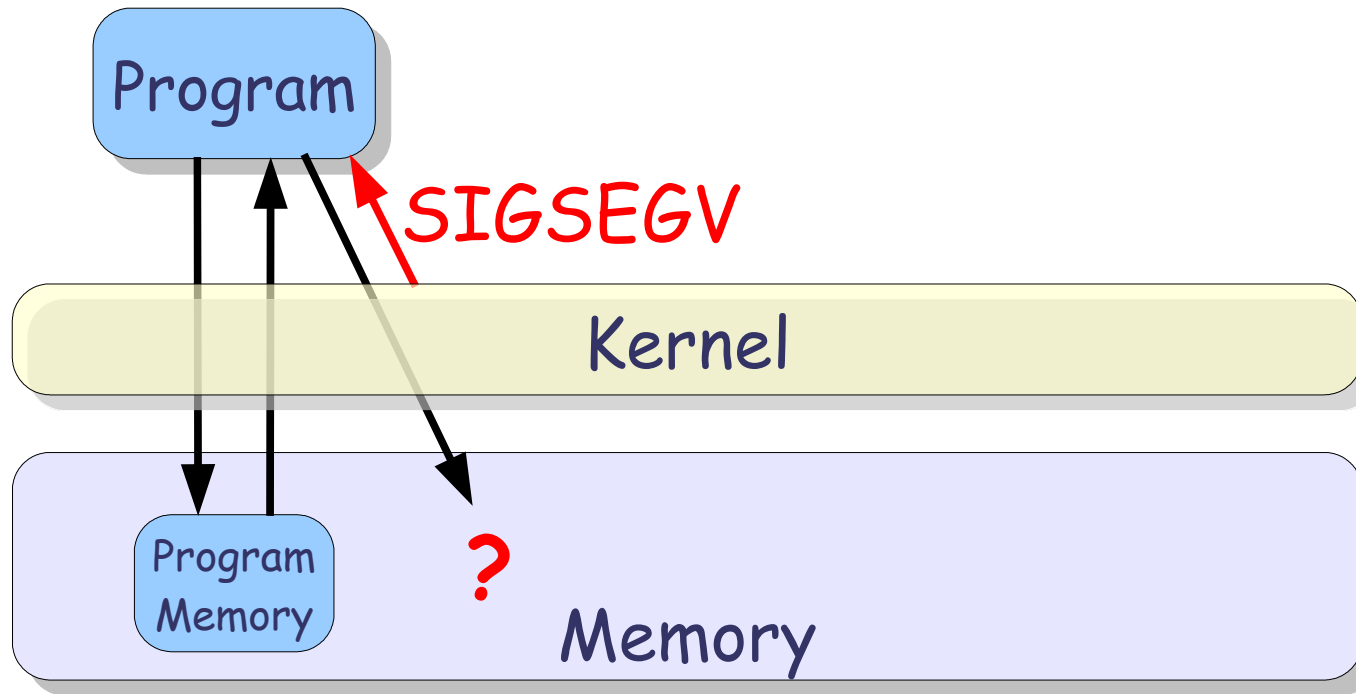
Note: This signal can be caught but this is dangerous. Typically a debugger need to handle this signal.

Registers

SP
PC
...

main() — Stack

... fork() ... — Instr.

... — Data

... — Heap

Identity

PID=1011
...

Resources

foo.txt
...

Registers

SP
PC
...

main() — Stack

... fork() ... — Instr.

... — Data

... — Heap

Identity

PID=1027
PPID=1011

Resources

foo.txt
...

Process Table

1000
1007
1011
1027

SIGCHLD

SIGCHLD

Registers

SP

PC

...

main() — Stack

... **continue** ... — Instr.

... — Data

... — Heap

Retrieve the return code

SIGCHLD

Identity

PID=1011

...

Resources

foo.txt

...

Registers

SP

PC

...

main() — Stack

... **exit()** ... — Instr.

... — Data

... — Heap

Identity

PID=1027

PPID=1011

Resources

foo.txt

...

Process Table

1000
1007
1011
1027

Sending a Signal (kill)

- **pid**:

  - **>0**: Sent to the specified PID

  ```
  #include <signal.h>
  int kill(pid, signal)
  ```

  - **0**: Sent to all processes with the same group ID than the sender, and for which the sender has permission to send a signal.

  - **-1**: Sent to all processes for which the sender has permission to send a signal.

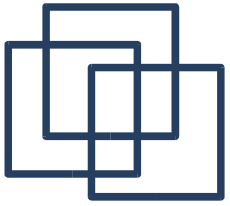 - **<-1**: Sent to all processes with group ID -pid, and for which the sender has permission to send a signal.

- **signal**:

  - **>0**: Send the corresponding signal

  - **0**: Send no signal (might be used to check the pid)

- Return values:

  - **0** if the message is successfully sent.

  - **-1** if the signal is invalid, the permissions are wrong or no such PID can be found.
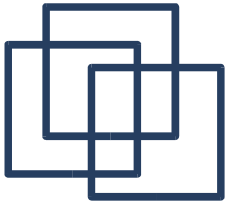
Sending a Signal

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
#include <signal.h>
#include <sys/types.h>

int main() {
  pid_t pid;
  int status;
  switch(pid = fork())
    {
    case -1: /* Failure */
      perror("signal_child");
      exit(1);
    case 0: /* Child code */
      while(1);
      exit(0);
    default: /* Parent Code */
      if (kill(pid, SIGTERM)) { /* SIGINT, SIGKILL, SIGTERM, 65, SIGCHLD */
        perror("signal_child");
        exit(1);
      }
      printf("The process %i returned a status: %i\n", wait(&status), status);
      exit(0);
    }
}
```

```
[fleury@hermes]$ ./int_child
The process 5333 returned a status: 2
[fleury@hermes]$ ./kill_child
The process 5340 returned a status: 9
[fleury@hermes]$ ./term_child
The process 5347 returned a status: 15
[fleury@hermes]$ ./65_child
send_signal: Invalid argument
[fleury@hermes]$ ./chld_child
```
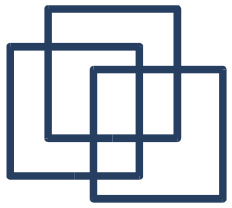
Sending a Signal (raise)

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
#include <signal.h>

int main() {
  int status;
  switch(fork())
    {
    case -1: /* Failure */
      perror("signal_child");
      exit(1);
    case 0: /* Child code */
      while(1);
      exit(0);
    default: /* Parent Code */
      if (raise(SIGKILL)) {
        perror("signal_child");
        exit(1);
      }
    printf("The process %i returned a status: %i\n", wait(&status), status);
    exit(0);
    }
}
```

```
[fleury@hermes]$ ./kill_child
^C
[fleury@hermes]$ ps ax | grep kill_child
 5198 R      ./kill_child
 5200 R+    grep kill_child
[fleury@hermes]$ kill -9 5198
```

Note: raise(signal)=kill(getpid(),signal)
Useful for setting alarms (see later).

Handling a Signal (System V)

```
#include <signal.h>
sighandler_t signal(signum, handler);
```

- **signum**: The signal to intercept.

- **handler**: Set the new handler when receiving the signal to either:

    - Address of the new procedure to handle the message

    - **SIG_IGN**: Ignore the signal

    - **SIG_DFL**: Default behaviour

- Return values:

    - **Address of the previous handler:** On success

    - **SIG_ERR**: On error
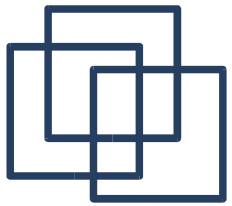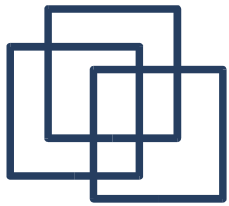
Handling a Signal (System V)

Note: `signal()` is conforming to ANSI C, not to POSIX ! The semantics might change from one Unix to another (Linux/Solaris). It's safer to use sigaction() (see later).

```
[fleury@hermes]$ ./signal_handler
Have a nice day !!!
[fleury@hermes]$ ./signal_ignore
^Z
[4]+  Stopped                   ./signal_child
[fleury@hermes]$ ps ax | grep signal_ignore
 5256 pts/4     T        0:04 ./signal_ignore
 5258 pts/4     R+       0:00 grep signal_ignore
[fleury@hermes]$ kill -9 5256
[4]+  Killed                    ./signal_ignore
[fleury@hermes]$ ./signal_default
^C
[fleury@hermes]$
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void handler(int signum) {
  printf("Have a nice day !!!\n");
  exit(0);
}

int main() {
  signal(SIGINT, handler); /* handler, SIG_IGN, SIG_DFL */
  while(1);
  exit(0);
}
```

Handling several Signals (System V)

```c
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

static int n=1;

void handler(int signum) {
  signal(SIGALRM, handler);
  switch(signum) {
  case SIGINT:
    printf("Goodbye!\n");
    exit(0);
  case SIGALRM:
    printf("Alarm number %i\n", n++);
    return;
  }
}

int main() {
  unsigned int i;
  signal(SIGINT,  handler);
  signal(SIGALRM, handler);

  while(1) {
    for(i=0; i<100; i++)
      raise(SIGALRM);
  }
  exit(0);
}
```
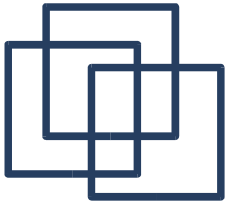
Note: `signal(SIGALRM, handler)` is protecting this piece of code when an alarm occur during its execution.
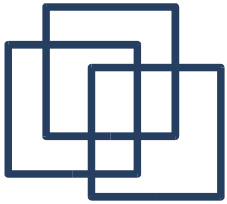
```
[fleury@hermes]$ ./signals_handler
...
Alarm number 32618
Alarm number 32619
Alarm number 32620
Alarm number 32621
Alarm number 32622
Alarm number 32623
Alarm number 32624
Alarm number 32625
Alarm number 32626
Alarm numbAlarm number 32832
Goodbye!
```

Alarm & Pause

```
#include <unistd.h>
unsigned int alarm(time);
int pause();
```

- **alarm()**:

    - time: The number of seconds to wait before raising the signal.

    - Return the **time left** if an alarm was already set, **0** otherwise.

- **pause()**:

    - The process is sleeping until any signal wake it up.

    - Return **-1** on errors, **0** otherwise.

Alarm & Pause

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

static int n=1;

void handler(int signum) {
  signal(SIGALRM, handler);
  switch(signum) {
  case SIGINT:
    printf("Goodbye!\n");
    exit(0);
  case SIGALRM:
    printf("Alarm number %i\n", n++);
    return;
  }
}

int main() {
  signal(SIGINT,  handler);
  signal(SIGALRM, handler);

  while(1) {
    alarm(5);
    pause();
  }
  exit(0);
}
```
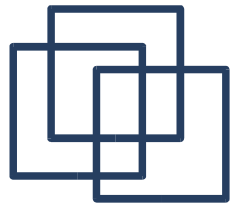
```
[fleury@hermes]$ ./signal_alarm
Alarm number 1
Alarm number 2
Alarm number 3
Alarm number 4
Alarm number 5
Alarm number 6
Alarm number 7
Goodbye!
[fleury@hermes]$
```
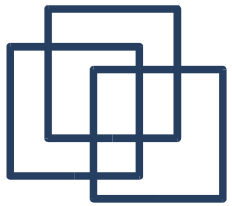
Handling a Signal (POSIX)

`signal()` is extremely simple to use,
but …

Problems with System V Signals:

- Not reliable (can be lost)

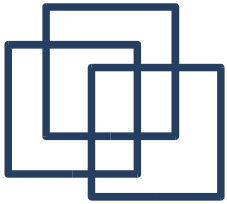- Semantics differ from BSD signals

- POSIX is using BSD semantics

Solution: Using `sigaction()` !

The Sigaction Structure

```
struct sigaction {
 void (*sa_handler)(int);
 void (*sa_sigaction)(int, siginfo_t *, void *);
 sigset_t sa_mask;
 int sa_flags;
}
```
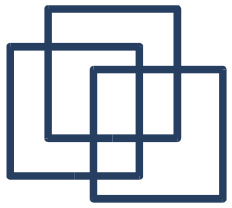
- **sa_handler**:
  Can be either SIG_DFL, SIG_IGN or a pointer to the new handler.

- **sa_sigaction**: Another handler with a more precise handling. Receive as arguments the signal number, and then a pointer to the siginfo_t and a pointer to the ucontext_t of the signal.

- **sa_mask**: Set of signals to block

- **sa_flags**: A set of options (see next slide).

sa_flags

Specifies a set of flags changing the behaviour of the signal handling process. It is formed by a bitwise OR of:

- `SA_NOCLDSTOP`: If signum is SIGCHLD, do not receive notification when child processes stop or resume.

- `SA_NOCLDWAIT`: If signum is SIGCHLD, do not transform children into zombies when they terminate.

- `SA_RESETHAND`: Restore the signal action to the default state once the signal handler has been called.

- `SA_ONSTACK`: Call the signal handler on an alternate signal stack provided by sigaltstack(). If an alternate stack isn't available, default stack is used.

- `SA_NODEFER`: Don't prevent the signal from being received from within its own signal handler.

- `SA_SIGINFO`: The signal handler takes 3 arguments, not one. In this case, sa_sigaction should be set instead of sa_handler.
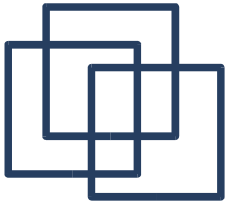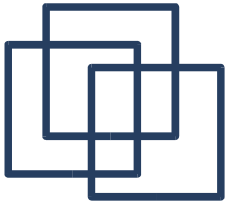
Handling Sets of Signals

#include <signal.h>

- struct sigset_t: A Set of signals

- int sigemptyset(set): Create an empty set of signals

- int sigfillset(set): Create a set of all the available signals

- int sigaddset(set, signum): Add signum to the set

- int sigdelset(set, signum): Remove signum from the set

- int sigismember(set, signum):
  Return 1 if signum is in set, 0 if not and -1 if an error occur

sigaction()

```
sigaction(int signum,
          const struct sigaction *act,
          struct sigaction *oldact)
```

- signum: Code of the signal to intercept

- act: Reference to the new signal handler
  (can also be SIG_DFL or SIG_IGN)

- oldact: Reference to the old signal handler
  (NULL if we don't want to have it)

sigaction()

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

static int n=1;
struct sigaction action;

int main() {
 /* Setting the sigaction struct */
 action.sa_handler = handler;
 action.sa_flags = 0;
 sigemptyset(&action.sa_mask);
 sigaction(SIGINT, &action, NULL);
 sigaction(SIGALRM, &action, NULL);

 /* Core of the program */
 while(1) {
   alarm(5);
   pause();
 }
 exit(0);
}
```
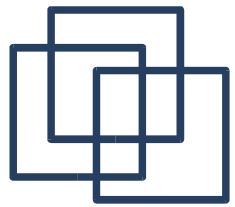
```c
/* Signal handler */
void handler(int signum) {
  switch(signum) {
  case SIGINT:
   printf("Goodbye!\n");
   exit(0);
  case SIGALRM:
   printf("Alarm number %i\n",n++);
   return;
  }
}
```

```
[fleury@hermes]$ ./sigaction
Alarm number 1
Alarm number 2
Alarm number 3
Alarm number 4
Alarm number 5
Alarm number 6
Goodbye!
[fleury@hermes]$
```

Handling a Signal (POSIX)

In the POSIX (BSD) model signals are either:

- **Pending:**

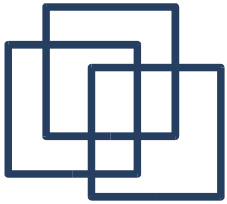 Not yet taken into account by the receiving process

- **Delivered:**

  Taken into account by the receiving process

- **Blocked or Masked:**

  The process want to stop the delivery of some signals

We need extra functions to handle the different states of a signal !

Sigaction Functions

- `sigprocmask(int how,sigset_t *set,sigset_t *oldset)`

  - `how`: Define the behaviour of this function

    - `SIG_BLOCK`: Add `set` to the blocked signals

    - `SIG_UNBLOCK`: Release `set` from blocked signals

    - `SIG_SETMASK`: The set of blocked signals is replaced by `set`

  - `set`: A set of signals

  - `oldset`: If not-null the old value of the signal mask is stored here

- `sigpending(sigset_t *set)`
  The signal mask of pending signals is stored in `set`

- `sigsuspend(sigset_t *mask)`
  Wait for signals which are NOT stored in `mask` (see `wait()`)
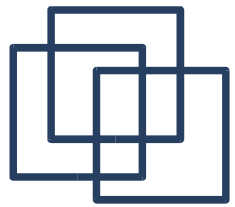
sigprocmask() & sigpending()

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

int sig;
sigset_t set, pending;

int main() {
  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  sigaddset(&set, SIGQUIT);
  sigprocmask(SIG_SETMASK, &set, NULL);

  sleep(15);

  /* Display the pending signals */
  sigpending(&pending);
  printf("Pending signals: ");

  for (sig=1; sig < NSIG; sig++)
    if (sigismember(&pending, sig))
      printf("%i ", sig);
  printf("\n");

  sleep(15);
```

```
[fleury@hermes]$ ./sigprocmask
^C^C^Z
[4]+  Stopped      ./sigprocmask
[fleury@hermes]$ fg
./sigprocmask
Pending signals: 2
Releasing blocked signals !

[fleury@hermes]$ ./sigprocmask
Pending signals:
Releasing blocked signals !
Exit normally !
[fleury@hermes]$
```

```c
  /* Releasing signals */
  sigemptyset(&set);
  printf("Releasing blocked signals !\n");
  sigprocmask(SIG_SETMASK, &set, NULL);

  printf("Exit normally !\n");

  exit(0);
}
```
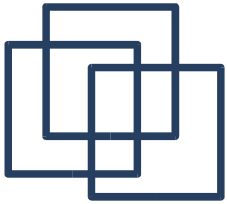
sigsuspend()

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void handler(int signum) {
 printf("The handler received an alarm !\n");
}

int main() {
    struct sigaction action;
    sigset_t block;

    sigfillset(&block);
    sigdelset(&block, SIGALRM);

    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    action.sa_handler = handler;
    sigaction(SIGALRM, &action, NULL);

    printf("Starting sigsuspend()\n");
    alarm(10);
    sigsuspend(&block);
    printf("After sigsuspend()\n");

    exit(0);
}
```
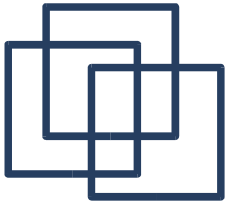
```
[fleury@hermes]$ ./sigsuspend
Starting sigsuspend()
The handler received an alarm !
After sigsuspend()
[fleury@hermes]$ ./sigsuspend
Starting sigsuspend()
The handler received an alarm !
^C^Z
[fleury@hermes]$ ./sigsuspend
Starting sigsuspend()
The handler received an alarm !
^Z^C
[fleury@hermes]$ ./sigsuspend
Starting sigsuspend()
The handler received an alarm !
^Z
[5]+  Stopped        ./sigsuspend
[fleury@hermes]$ fg
./sigsuspend
After sigsuspend()
[fleury@hermes]$
```
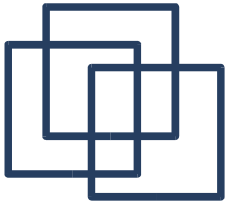
siginfo_t Structure

```
siginfo_t {
    int       si_signo;  /* Signal number */
    int       si_errno;  /* An errno value */
    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 */
    clock_t   si_utime;  /* User time consumed */
    clock_t   si_stime;  /* System time consumed */
    sigval_t  si_value;  /* Signal value */
    int       si_int;    /* POSIX.1b signal */
    void *    si_ptr;    /* POSIX.1b signal */
    void *    si_addr;   /* Memory location which caused fault */
    int       si_band;   /* Band event */
    int       si_fd;     /* File descriptor */
}
```
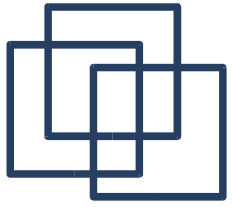
Sending a Signal (kill)

- kill -signum PID

- kill -signum -1

- kill -l [signum|signame]

Catching a Signal (trap)

```
[fleury@hermes]$ ./catching.sh
This prints before the "trap" --
even though the script sees the "trap" first.

Variable Listing --- a = 39 b = 36
```

```sh
#!/bin/sh
# Hunting variables with a trap.

trap 'echo Variable Listing --- a = $a  b = $b' EXIT
#  EXIT is the name of the signal generated upon exit from a script.
#
#  The command specified by the "trap" doesn't execute until
#+ the appropriate signal is sent.

echo "This prints before the \"trap\" --"
echo "even though the script sees the \"trap\" first."
echo

a=39
b=36
```
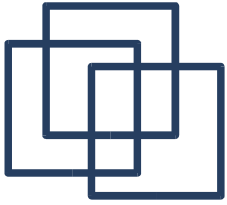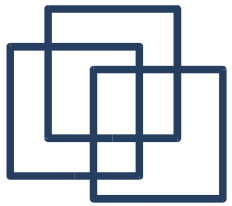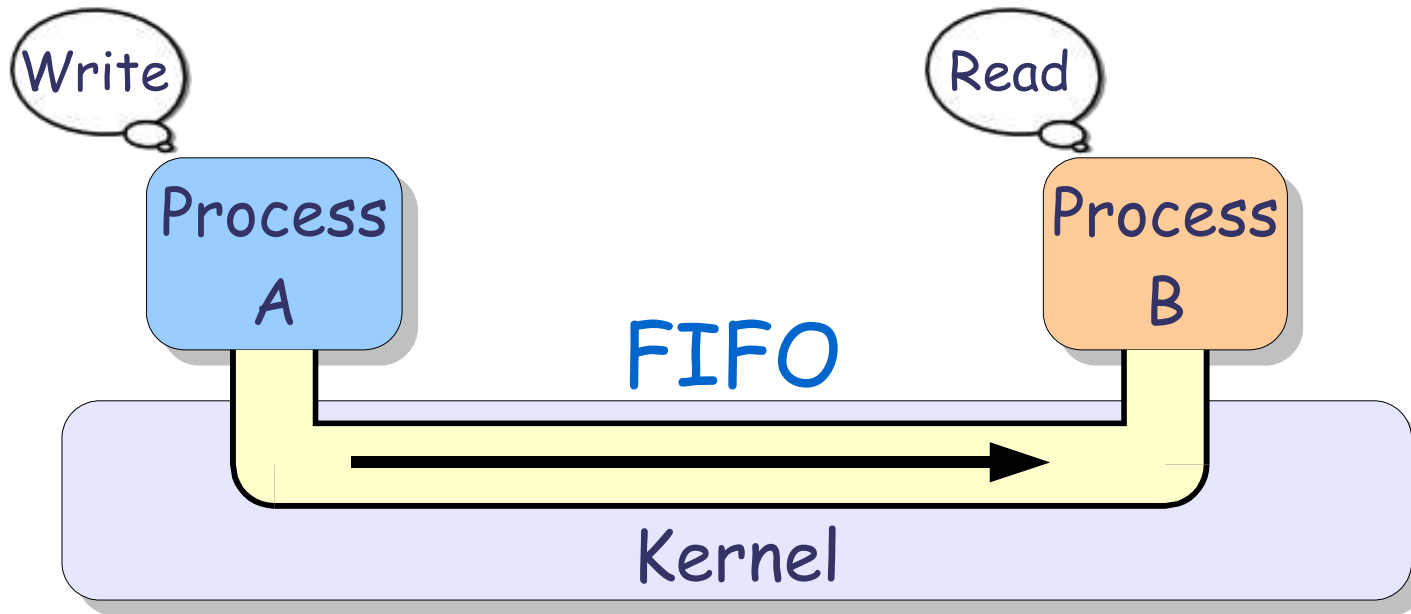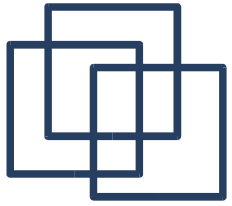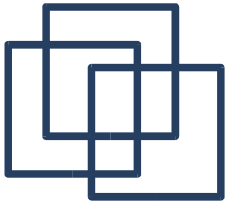
Pipes

Communication by Pipes

- Unidirectional Inter-process communication mechanism (similar to file descriptors)

- FIFO (First In, First Out)

- Streaming (reading destroy the data)

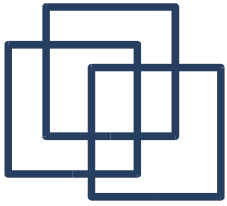- Limited Capacity (overflow is possible)

Two Types of Pipes

- Unnamed Pipes (one inode)

  - Creation/Destruction: pipe()/close()

  - Read/Write: read()/write()

  - Pipe Status: fstat()

- Named Pipes (a name in the file-system and an inode)

  - Creation/Destruction: mkfifo()/unlink()

  - Opening (in read or write): open()

  - Read/Write: read()/write()

  - Pipe Status: fstat()

pipe()

```
#include <unistd.h>
int pipe(int p[2]);
```

- p[2]: An array of file descriptors where:

  – p[0]: File descriptor to read from the pipe

  – p[1]: File descriptor to write to the pipe

- Return: 0 (success) or -1 (failure) and set errno.

pipe()

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  int nchar, p[2];
  char buffer[128], message[] = "abcdef";

  if (pipe(p)) /* Creating the pipe */
    perror("pipes");

  switch(fork()) {
  case -1:
    perror("pipes");
    exit(1);

  case 0: /* The child write to the pipe */
    close(p[0]);
    nchar = write(p[1], message, sizeof(message));
    printf("I wrote %i octets to my parent\n", nchar);
    exit(0);

  default: /* The parent read from the pipe */
    close(p[1]);
    nchar = read(p[0], buffer, sizeof(buffer));
    printf("I read %i octets from my child\n", nchar);
    printf("The message is \"%s\"\n", buffer);
    exit(0);
  }
}
```
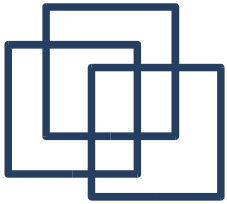
```
[fleury@hermes]$ ./pipes
I wrote 7 octets to my parent
I read 7 octets from my child
The message is "abcdef"
[fleury@hermes]$
```

40

pipe()

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  int nchar, p[2];
  char buffer[128], message[] = "abcdef";

  if (pipe(p)) /* Creating the pipe */
    perror("pipes");

  switch(fork()) {
  case -1:
    perror("pipes");
    exit(1);
  case 0: /* The child write to the pipe */
    close(p[0]);
    nchar = write(p[1], message, sizeof(message));
    sleep(1);
    nchar = write(p[1], message, sizeof(message));
    printf("I wrote %i octets to my parent\n", nchar);
    exit(0);
  default: /* The parent read from the pipe */
    close(p[1]);
    nchar = read(p[0], buffer, sizeof(buffer));
    printf("I read %i octets from my child\n", nchar);
    printf("The message is \"%s\"\n", buffer);
    exit(0);
  }
}
```
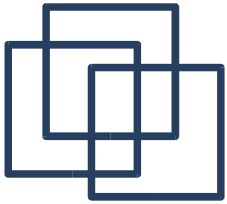
```
[fleury@hermes]$ ./pipes
I wrote 7 octets to my parent
I read 7 octets from my child
The message is "abcdef"
[fleury@hermes]$
```

Note: What if we do several writes ?

pipe()

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  int nchar, p[2];
  char buffer[128], message[] = "abcdef";

  if (pipe(p)) /* Creating the pipe */
    perror("pipes");

  switch(fork()) {
  case -1:
    perror("pipes");
    exit(1);
  case 0: /* The child write to the pipe */
    close(p[0]);
    nchar = write(p[1], message, sizeof(message));
    sleep(1);
    nchar = write(p[1], message, sizeof(message));
    printf("I wrote %i octets to my parent\n", nchar);
    exit(0);
  default: /* The parent read from the pipe */
    close(p[1]);
    while (nchar = read(p[0], buffer, sizeof(buffer))>0)
      printf("I read %i octets from my child\n", nchar);
    printf("The last message is \"%s\"\n", buffer);
    exit(0);
  }
}
```
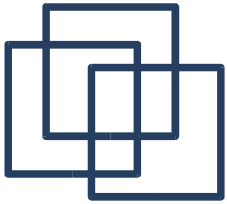
```
[fleury@hermes]$ ./pipes
I read 7 octets from my child
I wrote 14 octets to my parent
I read 7 octets from my child
The last message is abcdef
[fleury@hermes]$
```

Note: We have to read the pipe until the process writing in it dies.

42

pipe()

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  int nchar, p[2];
  char buffer[128], message[] = "abcdef";

  if (pipe(p)) /* Creating the pipe */
    perror("pipes");

  switch(fork()) {
  case -1:
    perror("pipes");
    exit(1);
  case 0: /* The child write to the pipe */
    close(p[0]);
    nchar = write(p[1], message, sizeof(message));
    sleep(1);
    nchar = write(p[1], message, sizeof(message));
    printf("I wrote %i octets to my parent\n", nchar);
    exit(0);
  default: /* The parent read from the pipe */
    /* close(p[1]); */
    while (nchar = read(p[0], buffer, sizeof(buffer))>0)
      printf("I read %i octets from my child\n", nchar);
    printf("The last message is \"%s\"\n", buffer);
    exit(0);
  }
}
```
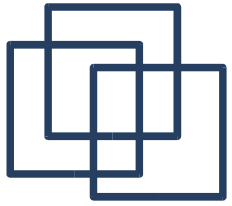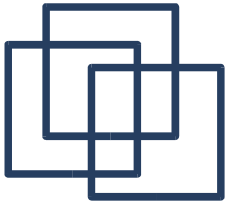
```
[fleury@hermes]$ ./pipes
I read 7 octets from my child
I wrote 14 octets to my parent
I read 7 octets from my child
^C
[fleury@hermes]$
```

Note: Closing an end of the pipe is crucial. If not, it might cause some deadlocks.
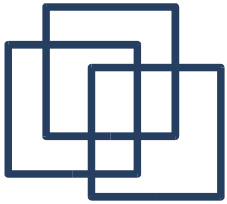
43

Named Pipes

- **Problems with unnamed pipes:**

  - They can be shared only by processes which are from the same parent (where the pipe(s) has been defined

  - Once they are closed, they are lost

- **Named Pipes:**

  - They provide a name in the file-system to hook on

  - Any process, knowing this name, can read/write to it (similar to network sockets)

mkfifo()

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

- **pathname**: The name of the pipe in the file system

- **mode**: Read/Write/Execute mode

- **Return**: 0 (success) or -1 (failure) and set errno

mkfifo (shell)

```
[fleury@hermes]$ mkfifo myfifo
[fleury@hermes]$ echo "fooooooooooooooo" > myfifo &
[2] 7500
[fleury@hermes]$ cat myfifo
fooooooooooooooo
[2]+  Done                    echo "fooooooooooooooo" >myfifo
[fleury@hermes]$ echo "fooooooooooooooo" > myfifo &
[2] 7534
[fleury@hermes]$ echo "baaaaaaaaaaaaar" > myfifo &
[3] 7536
[fleury@hermes]$ ls -lh myfifo
prw-r--r--  1 fleury fleury 0 2005-04-12 09:26 myfifo
[fleury@hermes]$ cat myfifo
baaaaaaaaaaaaar
fooooooooooooooo
[2]-  Done                    echo "fooooooooooooooo" >myfifo
[3]+  Done                    echo "baaaaaaaaaaaaar" >myfifo
[fleury@hermes]$
```
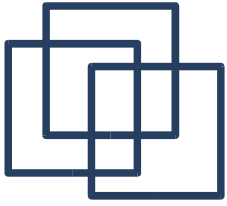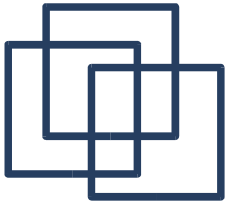
Questions ?
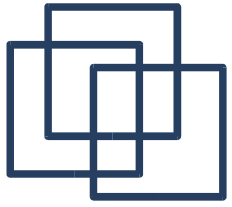
Next Week

- Inter-Process Communication:

  - Queues

  - Semaphores

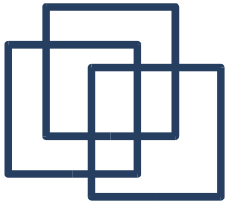 - Shared Memory Segments

- Network Sockets

POSIX.1

Core Services

(incorporates Standard ANSI C)

- Process Creation & Control
- Signals
- Floating Point Exceptions
- Segmentation Violations
- Illegal Instructions

- Timers
- File and Directory Operations
- Pipes
- C Library (Standard C)
- I/O Port Interface & Control

POSIX.1b

Real-time Extensions

- Priority Scheduling

- Real-Time Signals

- Clocks and Timers

- Semaphores

- Message Passing

- Shared Memory

- Asynchronous & Synchronous I/O

- Memory Locking
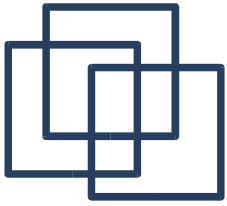
Thread Extensions

- Thread Creation, Control, & Clean-up

- Thread Scheduling

- Thread Synchronization

- Signal Handling