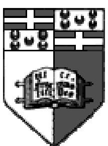
Signals

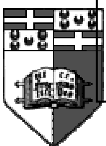
- Signals are software interrupts that give us a way to handle asynchronous events.
- Signals can be received by or sent to any existing process.
- It provides a flexible way to stop execution of a process or to tell it that something has happened, requiring its attention.
- All signals are integers which are *#define*'d as constants starting with the characters '**SIG**' inside the header <*signal.h*>.
- All signals received by a process have a default action associated with them which are: ignore, terminate, terminate with core or stop.



Signals (cont)

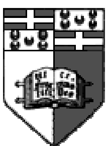
- Signals are of different types:
  - Kill process (SIGTERM, SIGABRT, SIGKILL, SIGQUIT, SIGSEGV)
  - Suspend process (SIGSTOP, SIGTSTP)
  - Resume a process (SIGCONT)
  - Notify ^C (SIGINT)
  - Memory violation (SIGSEGV)
  - Timer interval expiration (SIGALRM)
  - Child process termination (SIGCHLD)
  - User defined events (SIGUSR1,2)

SIGABRT	SIGIOT	SIGTSTP
SIGALRM	SIGKILL	SIGTTIN
SIGBUS	SIGPIPE	SIGTTOU
SIGCHLD	SIGPOLL	SIGURG
SIGCONT	SIGPROF	SIGUSR1
SIGEMT	SIGPWR	SIGUSR2
SIGFPE	SIGQUIT	SIGVTALRM
SIGHUP	SIGSEGV	SIGWINCH
SIGILL	SIGSTOP	SIGXCPU
SIGINFO	SIGSYS	SIGZFSZ
SIGINT	SIGTERM	
SIGIO	SIGTRAP	



Signals (cont)

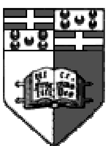
- Signals can occur due to one of the following events:
  - Certain terminal keys (ex. ^C)
  - Hardware exceptions
  - Software events
  - The *kill()* system call
- A process has one of three options on how to react to a signal arriving:
  - Ignore it
  - Perform the default action
  - Catch the signal using a user defined function



Signal Handling

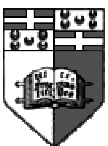
```
#include <signal.h>
void (*signal(int signo, void (*func)(int)) )(int);
                    returns pointer to previous signal
                    handler, SIG_ERR on error
or rather
          typedef void sigfunc(int);
          sigfunc *signal(int,sigfunc *);
```

- If we register a signal handler, when that signal is received, execution is temporarily suspended until the signal handler returns.
- *signal()* lets us register signal handlers for specific signals.
- *func* maybe one of :SIG_IGN, SIG_DFL, or a function that takes an integer and returns nothing.



Unix Signal Handling (cont)

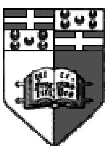
- SIGKILL and SIGSTOP should not be trapped and many systems do not let you trap them.
- On most systems, when a user defined signal handler executes, the signal handling disposition is reset and has to be set again.
- Most slow system calls on System V implementations, trap signals by returning an error value of –1 and setting *errno* to EINTR.
- One has to be careful about system calls which are interrupted by a signal and the action taken inside the signal handler. Only re-entrant functions can be used in the signal handler but the *errno* variable might still be corrupted.



Sending Signals

- From the command prompt one can send a signal to a process using:
  kill –<SIGNAL> <pid>
  default SIGNAL is SIGTERM
- *kill()* is used to send a signal to a specific process while *raise()* sends a signal to itself.
- A **superuser** can send a signal to any process.
- A process can send signals only to processes with the same real or effective UID.
- Signal **0** is a null signal to check existence of pid (*errno* = ESRCH).

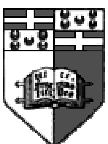
```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
     returns 0 if OK, -1 on error
```



pause()

```
#include <unistd.h>
int pause(void);
        returns –1 with errno= EINTR
```

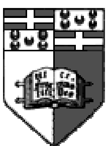
- *pause()* blocks the running process until a signal handler returns.
- Can be used as a sort of Inter Process Communication.



Unix *alarm()* and *abort()*

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
        returns previous value of timer if any
void abort(void);
        function never returns.
```

- *alarm()* returns immediately after setting a signal timer.
- *seconds* or more later, a signal (SIGALRM) will be sent to the calling process.
- This signal must be caught or ignored since the default action is to terminate the process.
- Previous timer is re-set and the previous value is returned.
- *seconds* equal to 0 also resets the timer.
- *abort()* sends the SIGABRT to the calling process.
- SIGABRT should not be ignored and signal handler should call *exit()* or *_exit()*



Exercises

- Implement your own *sleep(unsigned seconds)* system function.
- Write your own signal handler to catch the CTRL-C input.
- Implement the *raise()* system call.
- Use a slow re-entrant system call (ex. *read()*) and from another process bombard the process with signals. Make sure that the slow system call never fails.