Signals

Signals are used to inform a process that a certain event have occurred.

Signals can be generated by:

- The keyboard
- Errors in a process (for example an incorrect memory reference)
- Asynchronous events (for example a timer)
- The system call *kill* can be used to send all signals.
- Every time a process returns from a system call, it is tested if a signal has arrived.

Default action for most signals is to terminate the process.

A process can use the system call *sigaction* to define an alternative action for a certain signal:

- Signals can be ignored.
- A user written signal handler can be called.

Signals (cont.)

The kill signal (number 9) cannot be catched by a signal handler or ignored.

If the kernel has lightweight processes, signal handling is more complicated. The following options exist:

- Deliver the signal to the thread that caused it.
- Deliver the signal to all threads in the process.
- Deliver the signal to certain threads in the process.
- Deliver the signal to a specific thread.

Asynchronous signals (e.g. CTRL C) is handled by process groups in UNIX/Linux and are sent to all relevant threads/processes.

A thread (process) can use the system call *sigprocmask*, to block certain signals.

Lightweight processes (threads)

- Normal processes have an own address space. Process switching therefore requires change of address space and takes relatively long time (magnitude 1000 instructions)
- In certain implementations, a process (task) contain several threads (lightweight processes)
- All threads within a task has the same address space, which means that they share global variables and it exists no memory protection between them.
- Threads are scheduled according to same principle as processes but since only registers, pc, and stack pointer need to be changed, switching of threads is a lot quicker.

Advantages with threads

Reasons to have threads:

- Many program languages (e.g. Ada and Java) have concurrent processes in the language. These processes cannot be implemented as separate UNIX processes since process switching takes to long time.
- If Ada processes are implemented as threads, they can execute in parallel on a multiprocessor.
- Server processes need to await several events at the same time. With the aid of threads, one can program such processes with clean sequential code and blocking system calls.

User level or kernel level threads?

Threads can exist both in user programs and in the operating system.

The mapping between user level and kernel level can be done in different ways:

- many-to -one
- one-to-one
- many-to-many

User implemented threads

An alternative to kernel implemented threads is to implement them in user code as a library.

Advantages with user implemented threads:

- Do not need changes to the operating system.
- Faster process switching, as a trap to the operating system is eliminated.
- Different applications can use different scheduling algorithms.

User implemented threads (cont.)

Disadvantages with user implemented threads:

- If a thread makes a blocking system call, all threads in the task will stop. This is unacceptable but unavoidable if blocking system calls is the only alternative (which is common). It can be solved in a clumsy way if there is a separate system call to test if read will block.
- Implementation of preemptive scheduling (with signals) is usually inefficient. Non-preemptive scheduling means that a looping thread will stop all other threads in same task.
- Parallel execution of threads in a multiprocessor is not possible.

Thread libraries

There exist a few libraries for programming with threads in C.

Pthreads Posix standard that now exists on most UNIX/Linux systems.

Win32_threads C interface for threads in Win32 API. Similar functionality as pthreads, but different syntax.

Kernel level lightweight processes

The support for lightweight processes is not standardized in UNIX/Linux kernels.

Older UNIX systems did not have lightweight processes.

FreeBSD can create lightweight processes within an existing process with the system call *kse_create*. Processes that share resources with their parent can be created with *rfork*.

Linux cannot create threads within an existing process, but can create processes with almost arbitrary resource sharing with the system call *clone*.

Parameters to clone:

CLONE_FS File system information is shared.
CLONE_VM Memory is shared.
CLONE_SIGHAND Signal handlers are shared.
CLONE_FILES Open files are shared.