COMP3511 Operating Systems

Topic 4: Multithreaded Programming

Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

Acknowledgment: The lecture notes are based on various sources on the Internet

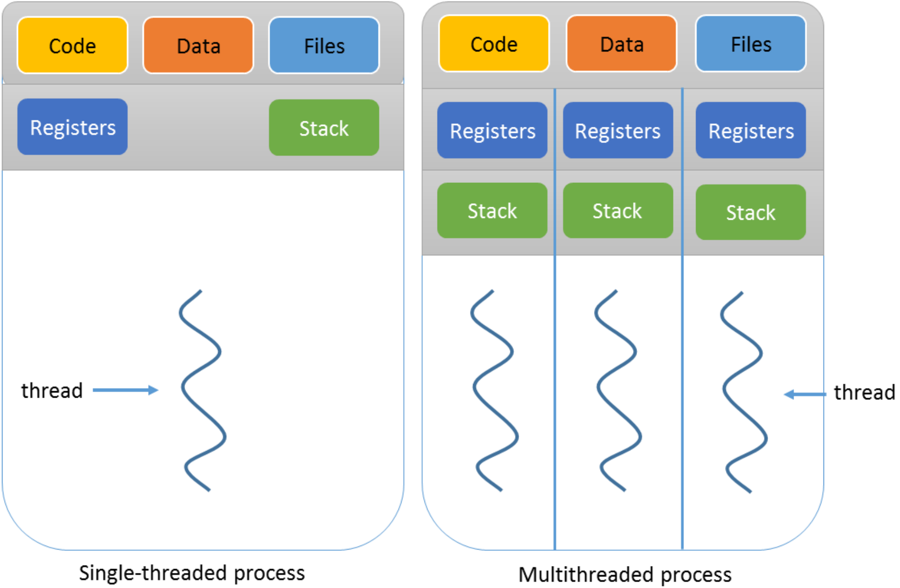
Threads

Threads

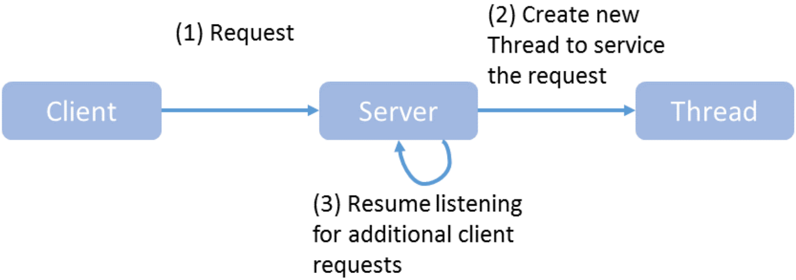
A thread is short for a thread of execution. A thread is a portion of code that may be executed independently of the main program.

- Most modern applications and/or programs are multithreaded
- Threads run within an application or a process
- Multiple tasks with the application can be implemented by separate threads, e.g., a word processor may have separate threads:
  - Render graphics
  - Respond to keystrokes form user
  - Spell/grammar checking
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Single and Multithreaded Processes



Multithreaded Server Architecture



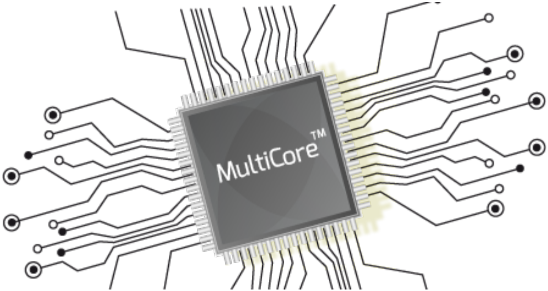
- A single application may be required to perform several similar tasks. For example a busy web server may process thousands of web requests concurrently. Creating one process for each client request is cumbersome (resource-intensive) and time-consuming
- A single application may need to do multiple tasks. For example, a web browser (client) needs to display images or text (one thread) while another thread retrieves data from the network

Benefits

- Responsiveness
  - May allow continued execution if part of process is blocked, especially important for user interface
- Resource sharing
  - Threads within a process share resources of the process by default, easier than shared memory or message passing that must be explicitly arranged by the programmer
- Economy
  - Thread creation is much cheaper than process creation, thread switching also has much lower overhead than context switching (switching to a different process)
- Scalability
  - A process can take advantage of multiprocessor architectures by running multiple threads of the process simultaneously on different processors (CPUs)

Multicore Programming

- Multicore or multiprocessor systems put pressure on programmers to make better use of the multiple computing cores
- Programming challenges in multicore systems include:
  - Identifying tasks: To divide applications into separate, concurrent tasks
  - Balance: Tasks perform equal work of equal value
  - Data splitting
  - Data dependency
  - Testing and debugging



Parallelism vs. Concurrency

- Parallelism implies a system can perform more than one task simultaneously
- Concurrency supports more than one task making progress
  - Single processor / core, scheduler providing concurrency
- Types of parallelism
  - Data parallelism - distributes subsets of the same data across multiple cores, same operation on each
  - Task parallelism - distributing threads across cores, each thread performing unique operation

Poppy Cat

Parallelism vs. Concurrency (Cont'd)

- Concurrent execution on single-core system:

Single core	T_1	T_2	T_3	T_4	T_1	T_2	T_3	T_4	T_1	...

time $\longrightarrow$

- Parallelism on a multicore system

Core 1	T_1	T_3	T_1	T_3	T_1	...
Core 2	T_2	T_4	T_2	T_4	T_2	...

time $\longrightarrow$

Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- S denotes percentage of serial portion and N denotes number of processing cores

$$speedup \leq \frac{1}{S + \frac{1-S}{N}}$$

  - Example: What is the speedup if an application is 75% parallel and 25% serial, moving from 1 to 2 cores?

$$speedup \leq \frac{1}{0.25 + \frac{1-0.25}{2}} = 1.6$$

  - What if N approaches infinity?

$$speedup \leq \frac{1}{S + \frac{1-S}{\infty}} = 1/S$$

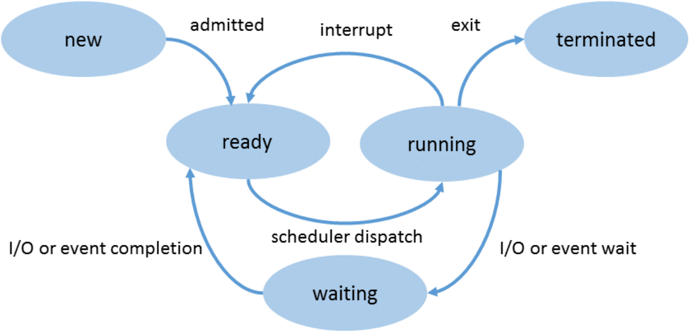
  - What if N is 1, i.e. single core?

- Serial portion of an application has disproportionate effect on performance gained by adding additional cores

Thread State

- Each thread has a Thread Control Block (TCB)
  - Execution information: CPU registers, program counter, pointer to stack
  - Scheduling information: State (more later), priority, CPU time
  - Accounting information
  - Various pointers (for implementing scheduling queues)
- OS keeps track of TCBs in protected memory
  - Array / linked list, ...
- Information shared by all threads in process / address space
  - Contents of memory (global variables, heap)
  - I/O state (file system, network connections, etc.)
- State "private" to each thread
  - Kept in TCB
  - CPU registers (including program counter)
  - Execution stack
    - Parameters, temporary variables
    - Keep program counters while called procedures are executing

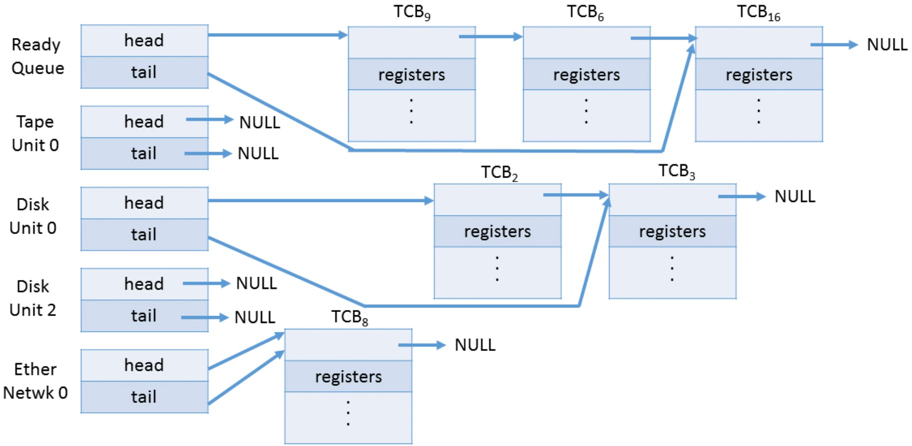
Life Cycle of a Thread



- As a thread executes, it changes state
  - new: The thread is being created
  - ready: The thread is waiting to run
  - running: Instructions are being executed
  - waiting: Thread waiting for some event to occur
  - terminated: The thread has finished execution
- "Active" threads are represented by their TCBs
  - TCBs organized into queues based on their states

Ready Queue and Various I/O Device Queues

- Thread not running → TCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy



Examples of Multithreaded Programs

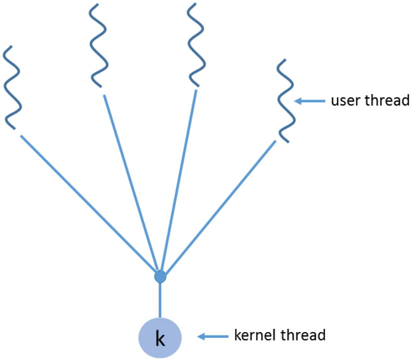
- Embedded systems
  - Elevators, Planes, Medical Systems, Wristwatches
- Most modern OS kernels
  - Internally concurrent to deal with concurrent requests by multiple users
  - But no protection needed within kernel
- Database servers
  - Access to shared data by many concurrent users
  - Also background utility programming must be done
- Network servers
  - Concurrent requests from network
  - Again, single programs, multiple concurrent operations
  - File server, Web server, the airline reservation systems
- Parallel programming (More than one physical CPU)
  - Split program into multiple threads for parallelism

User Threads and Kernel Threads

- Support for threads may be provided at either the user level, for user threads, or by the kernel, for kernel threads
- User threads - management done by user-level threads library without kernel support. Three primary thread libraries
  - POSIX Pthreads
  - Win32 threads
  - Java threads
- Kernel threads - supported by the kernel. Virtually all general-purpose operating systems support kernel threads, including
  - Windows
  - Solaris
  - Linux
  - Mac OS X
- Ultimately, a relationship must exist between user threads and kernel threads
  - Many-to-One
  - One-to-One
  - Many-to-Many

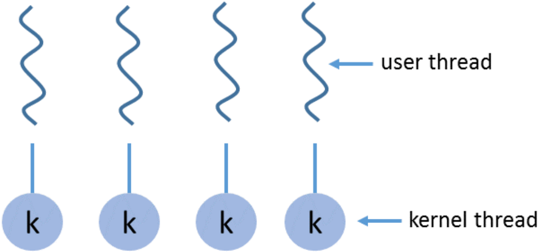
Many-to-One

- Many user-level threads mapped to a single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on a multicore system because only one may be in kernel at a time
- Few systems currently use this model. For examples:
  - Solaris Green Threads
  - GNU Portable Threads



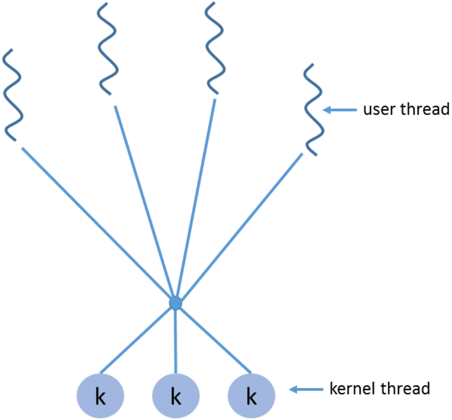
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples:
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later



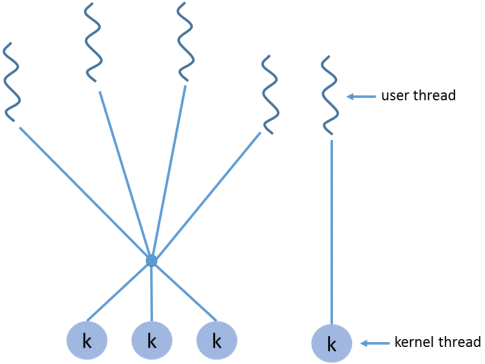
Many-to-Many

- Allows many user-level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Examples
  - Solaris prior to version 9
  - Windows NT/2000 with the ThreadFiber package



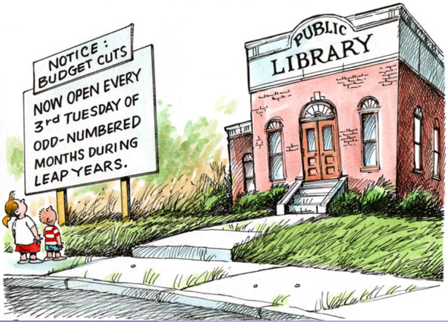
Two-Level Model

- Similar to many-to-many, except that it also allows a user thread to be bound to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Implementing in two primary ways
  - Library entirely in user space with no kernel support. This means that invoking a function in the library results in a local function call in user space, and not a system call
  - Kernel-level library supported directly by the OS
- Three main thread libraries are in use today:
  - POSIX Pthreads
  - Windows
  - Java



POSIX Pthreads

- Create a thread

```
int pthread_create(pthread_t* thread,
                   const pthread_attr_t* attr,
                   void* (*start_routine)(void*),
                   void* arg)
```

  - thread - returns the thread id
  - attr - set to NULL if default thread attributes are used
  - start - pointer to function to be threaded
  - arg - pointer to argument of function
  - return 0 if the thread creation is successful

- Waits for the thread to terminate

```
int pthread_join(pthread_t th, void** thread_return)
```

  - th - thread suspended until the thread identified by th terminates
  - thread_return - if thread_return is not NULL, the return value of th is stored in the location pointed to by thread_return
  - return 0 if the thread join is successful

- Terminates a thread

```
void pthread_exit(void* retval)
```

  - retval - return value of pthread_exit()

POSIX Pthreads: Example

```c
/* threadexample.c */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* print(void *ptr) { printf("%s \n", (char*)ptr); }

int main() {
  pthread_t t1, t2;
  const char* msg1 = "Thread 1";
  const char* msg2 = "Thread 2";
  int ret1, ret2;

  ret1 = pthread_create(&t1, NULL, print, (void*)msg1);                    // create a thread
  if(ret1) { fprintf(stderr, "Error, pthread_create(): %d\n",ret1); exit(1); }
  ret2 = pthread_create(&t2, NULL, print, (void*)msg2);                    // create a thread
  if(ret2) { fprintf(stderr, "Error, pthread_create(): %d\n",ret2); exit(1); }

  printf("pthread_create() for thread 1: %d\n", ret1);
  printf("pthread_create() for thread 2: %d\n", ret2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  exit(0);
}
```

POSIX Pthreads: Example (Cont'd)

- Command to compile the program:

```
gcc -pthread threadexample.c
```

- Output:

```
pthread_create() for thread 1: 0
pthread_create() for thread 2: 0
Thread 1
Thread 2
```

Threading Issues - Semantics of fork() and exec() in Multithreaded Environment

- Does fork() duplicate only the calling thread or all threads?
  - Some UNIX have two versions of fork, one version of fork() duplicates all threads and the other duplicates only the thread that invoked the fork() system call
  - If exec() is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process
- exec() usually works as normal
  - Replace the running process including all threads

Signal in UNIX

Signals

Signals are in UNIX systems to notify a process that a particular event has occurred

- A signal may be received either synchronously or asynchronously, depending on the source of and the reason for the event being signaled
- All signals follow the same pattern
  - Signal is generated by the occurrence of a particular event
  - Signal is delivered to a process
  - Once delivered, the signal must be handled
- A signal may be handled by one of the two possible handlers
  - A default signal handler
  - A user-defined signal handler
- Every signal has default handler that kernel runs when handling signal
  - User-defined signal handler can override the default handler
  - For single-threaded, signal delivered to process

UNIX / Linux Signal Handling

- Prototype of a signal handling function

```
void <signal handler func name>(int sig)
```

- To get the signal handler function registered to the kernel, the signal handler function pointer is passed as second argument to the 'signal' function. The prototype of the signal function is

```
void (*signal(int signo, void (*func )(int)))(int);
```

  where signo refers to signal number

- The following shows some of the signal numbers
  - SIGHUP: Hangup, report that user's terminal is disconnected
  - SIGTERM: Termination, it can be blocked, handled, and ignored. Generated by "kill" command
  - SIGINT: Interrupt, Program interrupt signal from keyboard (normally Ctrl-c)
  - SIGQUIT: Quit, terminate process and generate core dump
  - SIGFPE: Floating point arithmetic exception, e.g. division by zero
  - SIGKILL: Kill, unblockable, cause immediate program termination, cannot be handled, blocked or ignored
  - SIGCHLD: Child status has changed, child process stopped or terminated
  - SIGSEGV: Segmentation violation, dereferencing a bad or NULL pointer

UNIX / Linux Signal Handling: Example 1

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

// Define the function to be called when ctrl-c
// (SIGINT) signal is sent to process
void signal_callback_handler(int signum) {
    printf("Caught signal %d\n",signum);
    // Terminate program
    exit(1);
}

int main() {
    // register signal and signal handler
    signal(SIGINT, signal_callback_handler);

    while(1) {
        printf("Processing here.\n");
        sleep(1);
    }
    return 0;
}
```

- Run the program and press Ctrl-c:

  ```
  Processing here.
  Processing here.
  Processing here.
  Caught signal 2
  ```

- Run the program at one terminal and type a command at another terminal
  - **First terminal running the program**

    ```
    Processing here.
    Processing here.
    Processing here.
    Caught signal 2
    ```

  - **Second terminal (Type command) Assume the process id of the program is 12570 (check using ps -all)**

    ```
    kill -INT 12570
    ```

UNIX / Linux Signal Handling: Example 2

```c
// Example of how two processes can
// communicate to each other using
// kill() and signal(). We will fork()
// 2 process and let the parent
// send a few signals to it's child

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

void sighup();
void sigint();
void sigquit();

int main() {
  int pid;

  if((pid = fork()) < 0) {
      perror("Fail to create child");
      exit(1);
  }

  if(pid == 0) {                                  // child
      // register functions to the kernel
      signal(SIGHUP,sighup);
      signal(SIGINT,sigint);
      signal(SIGQUIT, sigquit);
      for(;;);                                    // infinite loop
  }
  else {                                          // parent
      printf("\nParent: send SIGHUP\n\n");
      kill(pid,SIGHUP);
      sleep(3);                                   // pause for 3 seconds
      printf("\nParent: send SIGINT\n\n");
      kill(pid,SIGINT);
      sleep(3);                                   // pause for 3 seconds
      printf("\nParent: send SIGQUIT\n\n");
      kill(pid,SIGQUIT);
      sleep(3);                                   // pause for 3 seconds
  }
}
```

UNIX / Linux Signal Handling: Example 2 (Cont'd)

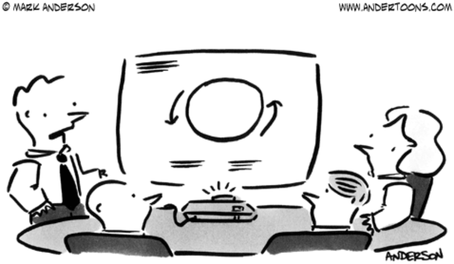
```
void sighup() {
  signal(SIGHUP,sighup);                      // reset signal
  printf("Child:  I have received a SIGHUP\n");
}

void sigint() {
  signal(SIGINT,sigint);                      // reset signal
  printf("Child:  I have received a SIGINT\n");
}

void sigquit() {
  printf("My parent has killed me!!!\n");
  exit(0);
}
```

Output:

```
Parent: send SIGHUP

Child: I have received a SIGHUP

Parent: send SIGINT

Child: I have received a SIGINT

Parent: send SIGQUIT

My parent has killed me!!!
```

Threading Issues - Signal in UNIX (Cont'd)

- Where should a signal be delivered a multi-threaded program?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process
- The method for delivering a signal depends on the type of signal
  - Synchronous signals need to be delivered to the thread causing the signal, not other threads
  - Terminating a process signal should be sent to all threads with the process

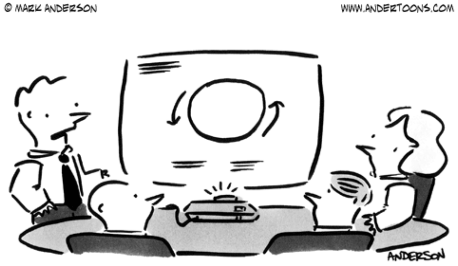


"Normally I can see both sides of an issue. But this..."

Threading Issues - Thread Cancellation

- Thread cancellation involves terminating a thread before it has completed.

  For example: Multiple threads are concurrently searching through a database, one thread returns the result, the remaining threads might be canceled
- Thread to be canceled is called target thread
- Cancellation of a target thread may occur in two different scenarios
  - Asynchronous cancellation terminates the target thread immediately
  - Deferred cancellation allows the target thread to periodically check if it should be canceled



"Normally I can see both sides of an issue. But this..."

Thread-Local Storage

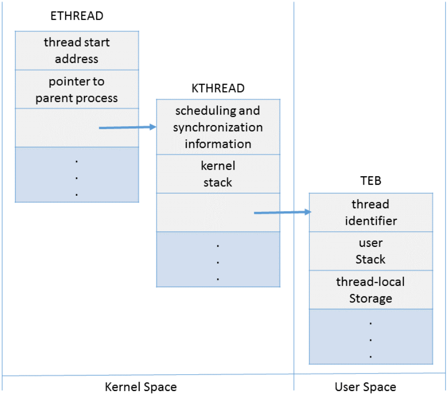
- Thread-local storage (TLS) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations, i.e. similar to static data
- TLS is unique to each thread

Windows Threads

- Windows implements the Windows API for Windows 98, NT, 2000, WinXP, and Windows 7
- Implements the one-to-one mapping, one user-level thread $\rightarrow$ one kernel thread
- Each thread contains
  - A thread ID uniquely identifying the thread
  - A register set representing the status of the processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - A private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the context of the thread

Windows Threads (Cont'd)

- The primary data structures of a thread include
  - ETHREAD (executive thread block), which includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - TEB (thread environment block), which has thread ID, user-mode stack, thread-local storage, in user space
  - KTHREAD (kernel thread block), which has scheduling and synchronization information, kernel-mode stack, pointer to TEB, in kernel space



Linux Threads

- Linux refers to processes and threads as tasks rather than threads
- Thread creation is done through clone() system call
- clone() allows a child tasks to determine how to share the address space of the parent task (process)
  - Flags control behavior

flag	**meaning**
CLONE_FS	File-system information is shared
CLONE_VM	The same memory space is shared
CLONE_SIGNAND	Signal handlers are shared
CLONE_FILES	The set of open files is shared

That's all!

Any questions?