Advanced topics: reentrant function

- Functions executed in a multi-threaded environment need to be re-entrant
  - it can be safely called again before its previous invocation has been completed
- Requirements:
  - Must not hold static/global non-constant data.
  - Must not return the address to global, non-constant data.
  - Must work only on the data provided to it by the caller.
  - Must not rely on locks to singleton resources.
  - Must not call non-reentrant functions
- Note: some POSIX functions are not reentrant and have reentrant counterparts (e.g. `strtok` vs. `strtok_r`)
Advanced topics: Threads vs. processes in Linux

- In Linux: no difference on the OS level between processes and threads
  - Threads are processes which have access to the same resources, e.g. address space etc.
  - E.g. internally Linux creates a thread by calling
    \[
    \text{clone(CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND, 0)};
    \]
- In contrast, a new process is created internally by Linux calling
  \[
  \text{clone(SIGCHLD, 0)}
  \]

Advanced topics: Thread and Process Affinity

- Each thread/process has an \textit{affinity mask}
  - Specifies what processors a thread is allowed to use
  - Different threads can have different masks
  - Affinities are inherited across process creation
- Example: 4-way multi-core

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>core 3</td>
<td>core 2</td>
<td>core 1</td>
<td>core 0</td>
<td></td>
</tr>
</tbody>
</table>

Process/thread is allowed to run on cores 0,2,3, but not on core 1

Slide based on a lecture of Jernej Barbic, MIT.
http://people.csail.mit.edu/barbic/multi-core-15213-sp07.ppt
Linux Kernel scheduler API

- Setting affinity mask of a thread

```c
#define __USE_GNU
pthread_setaffinity_np(thread_t t, len, mask);
pthread_attr_setaffinity_np (thread_attr_t a, len, mask);
```

- First function modifies the affinity mask of an existing thread
- Second function sets the affinity mask of a thread before it is created
  - A thread inherits the affinity mask of the main thread and will run on the same core initially as the main thread otherwise

```c
#define _GNU_SOURCE
#define _GNU_SOURCE 1

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

pthread_t tid;
pthread_attr_t attr;
cpu_set_t cpuset;

CPU_ZERO(&cpuset);
CPU_SET(1, &cpuset); // thread will be allowed to run on
                     // core 1 only
pthread_attr_init(&attr);

pthread_attr_setaffinity_np(&attr, sizeof(cpuset),
                           &cpuset);
pthread_create (&tid, &attr, worker, (void *)
                (intptr_t) 1);
```
Thread affinity

- E.g. Intel Tigerton Processor

  ![Thread affinity diagram]

  - For threads that have to often access shared data items, costs of update operations are smaller if threads are ‘close to each other’
  - Cache coherence effects can be limited if threads are close to each other

Advanced topics: Process scheduler

- Component of the kernel that selects which process to run next
- Processes are either waiting to run (blocked) or running
- Multi-tasking: interleave the execution of more than one process
  - Cooperative multitasking: scheduler does not stop a running process until it voluntarily decides to do so (yielding)
  - Preemptive multitasking: scheduler decides when to cease a running process
    - timeslice: amount of time a process is run before being preempted
Process scheduler

- Scheduling policy must satisfy two conflicting goals:
  - Process response time (low latency)
  - Maximum system utilization (high throughput)
- Priority based scheduling in Linux:
  - Rank processes based on their worth for need for processing time
  - Processes with higher priority run before those with lower priority.
  - Processes with the same priority are scheduled round-robin
  - Higher priority processes in Linux receive larger timeslices

SMP support in Linux

- Linux maintains one *runqueue* per processor
- Each *runqueue* contains two priority queues:
  - *active queues*: list of tasks to be executed
  - *expired queues*: list of tasks that have been executed
  - A task is moved from the *active* to the *expired* queue after execution
  - When a task is moved from the active to the expired queue, dynamic priority and *timeslice* length are recalculated
  - After all tasks in the active queue have executed, both queues are swapped
Load balancing in Linux kernel

- Load balancer tries to ensure that the number of processes is evenly distributed across the different runqueues (= processors)
- Load balancing function called periodically on all processors
- Load balance algorithm:
  - check all other runqueues for busiest runqueue, i.e. runqueue with the largest number of processes
  - If no runqueue with more than 25% more processes than on the own processors found, exit
  - Else find tasks that can be migrated to own runqueue

Load balancing in Linux

- To migrate a task from one runqueue to another:
  - Search in highest priority queues first on the busiest runqueue
  - Preferably tasks from the expired queue
    - These tasks are not running, and are least likely to have data in the cache
  - Need to take affinity setting of task/process into account
    - Is the process/thread allowed to run on another processor?
- A processor ‘steals’ work from other processors, it never ‘pushes’ work to other runqueues
Thread-creation on Linux

- Whenever a new thread is created:
  - Inherits parent’s process ( = main thread) priority value
  - Inherits half of parent’s process timeslice
  - Has the same affinity mask as the parent process (unless explicitly changed by user)
  - Is most probably in the same runqueue ( = on the same processor) as the parent process

- Once thread has been moved to the expired queue
  - Timeslice for each thread is recalculated ( = increased)
  - Other processors can steal a thread