Operating Systems 2010/2011

Threads

Johan Lukkien

Agenda

- Thread creation
- Thread execution model

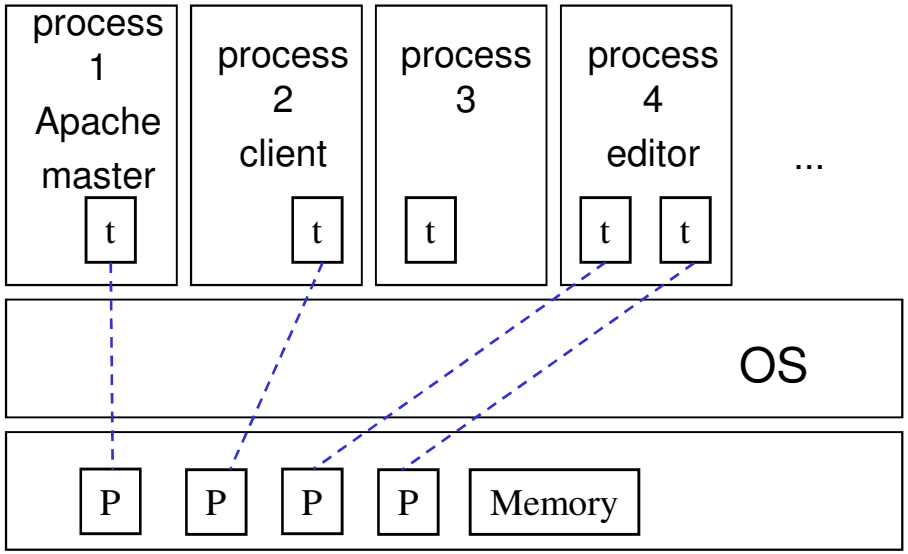
Concepts in OS: Process, Thread

- Process ("program in execution")
  - defines a data space (virtualization of the memory)
  - realizes transparency for multiple activities
  - defines ownership of resources
  - has at least one associated thread
  - unit of deployment (distribution)
  - unit of fault containment

- Thread
  - unit of concurrency (virtualization of the processor)
  - unit of scheduling
    - though in case of one thread per process, the *process* is often said to be the unit of scheduling
  - operates in an address space
    - i.e., in a process; several threads may share this
  - has an associated *execution state*
    - place where it is in the code, stack image and return addresses of function calls, values of processor registers

Reasons to introduce threads

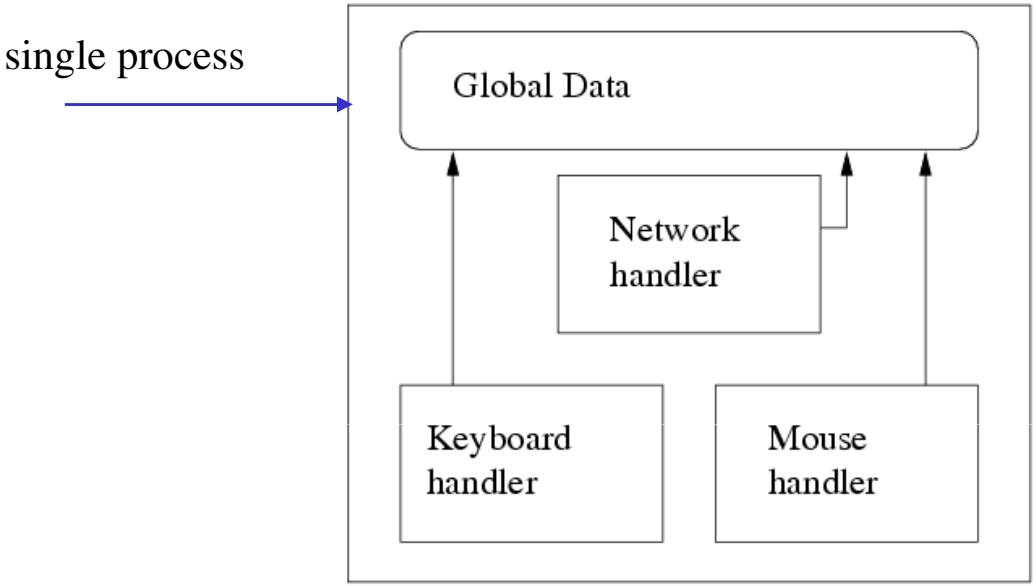
- Increase concurrency level
  - performance
    - hide latency
    - increase responsiveness
    - exploit platform concurrency
  - discriminate importance levels in activities
    - e.g. interrupt routines



- Deal with the natural concurrency
  - natural organization, structure ... e.g.
    - thread per event
    - thread per resource
    - thread per (active) external interaction sequence
      - e.g. in user interfaces

Structuring by threads

- This program must deal with three input sources



- Possible program structure:
  - while ... do
    - check keyboard;
    - check network;
    - check mouse;
    - wait a while

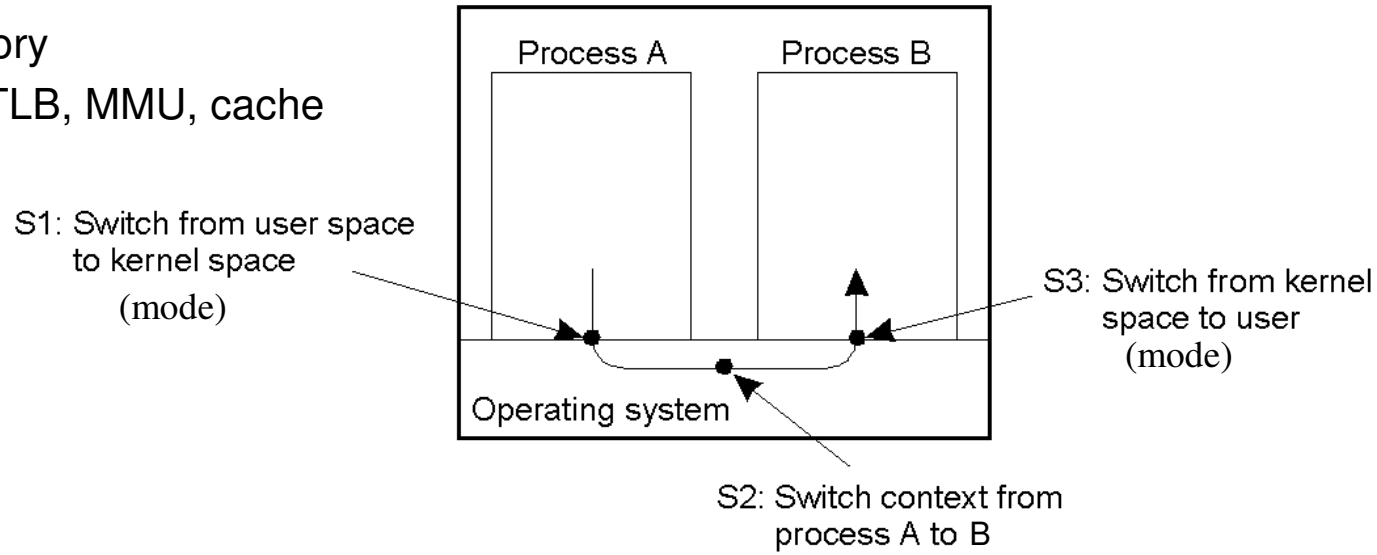
    od

- Simpler organization
  - start three threads
  - each thread takes care of one source

- Downside: shared resource (global data)

Process shortcomings

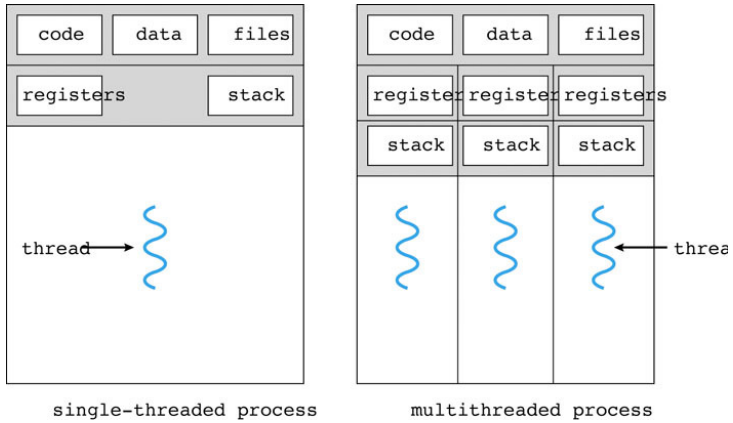
- Creation overhead
- No shared memory (in principle)
- Switching overhead: *mode and context switch*
  - for each Inter-Process Communication (IPC)
  - state save/restore
    - CPU state
    - memory
      - TLB, MMU, cache



Threads

- Concurrency on shared memory
  - no protection agains other threads (faults, memory sharing)
  - no expensive IPCs between threads
  - easier to construct programs (?)
    - *exploit platform concurrency*: multiple processors will run multiple threads
    - *latency hiding*: while waiting do something useful in another thread

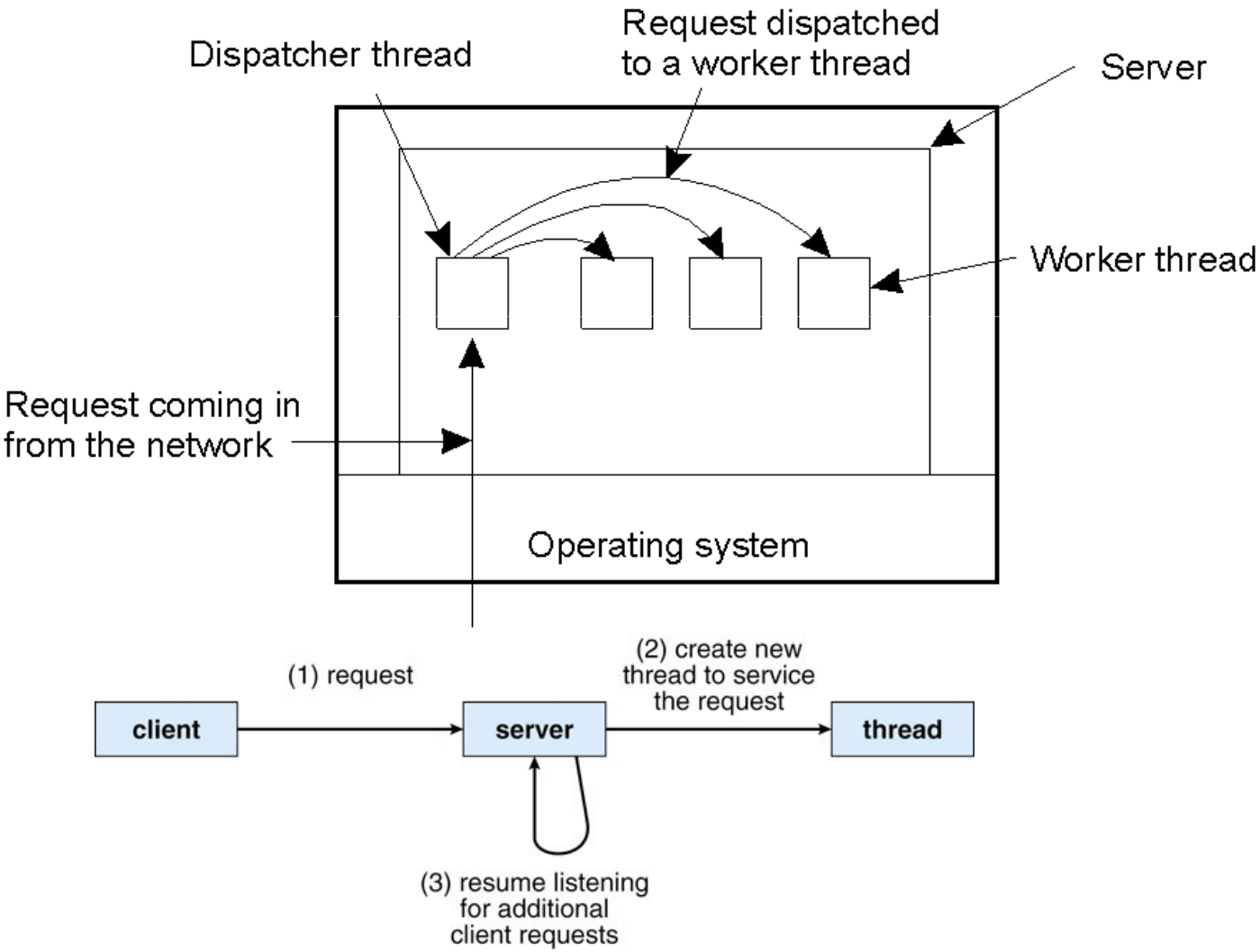
- State per thread
  - control state ('context'): CPU registers, masks
  - stack



- Generally, little overhead for switching: no context!
  - depending on underlying thread execution model

Example: multithreaded server

- Dispatch/worker model



POSIX 1003.1c: Thread interface (pthreads)

- Start a *function* as a new thread
  - setting attributes, e.g. stacksize;
  - thread terminates when this function returns

- Extensive interface
  - e.g. detach, join, cancel

```
pthread_t thread_id;

status = pthread_create (&thread_id, attr, func, arg_of_func);
/* func(arg_of_func) is started as a new thread; when attr == NULL some
 * internal defaults are chose */

status = pthread_join (thread_id, &result);
/* wait for thread to terminate */
```

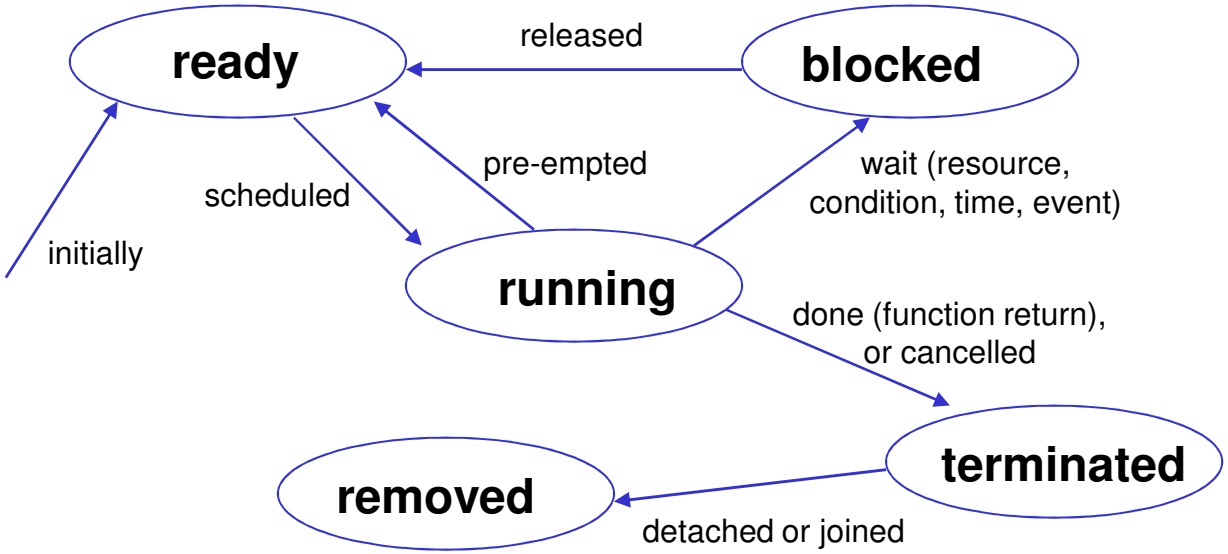
Example program

```
#include <stdio.h>
#include <pthread.h>

int x;

void *Count_100 ()
{
  int i;
  for (i = 0; i < 100; i ++) {
    x = x+1;
  }
  return (NULL);
}

int main ()
{
  pthread_t thread_id;

  x = 0;
  pthread_create (&thread_id, NULL, Count_100, NULL);
  Count_100 ();
  pthread_join (thread_id, NULL);
  printf ("x = %d\n", x);
}
```

- How many threads are here?
- How many processes?
- What are *possible* final values of *x?*

The life of a thread in POSIX

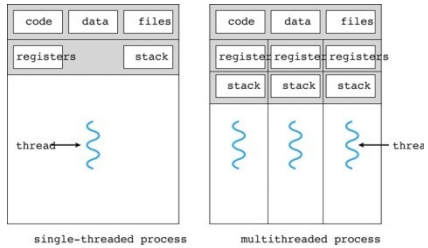
- Main program: own thread
- Additional threads:
  - created on demand or on receipt of signal
  - thread code: a function in the program



- **Notice**:
  - process states are super-imposed
  - the effect of process suspension depends on threads is system dependent

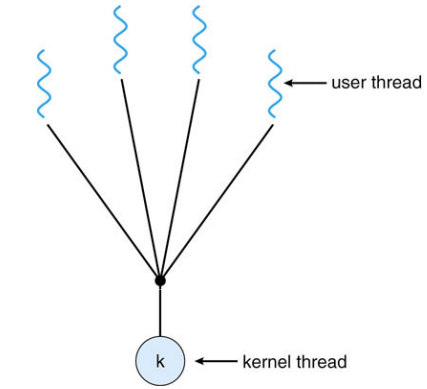
Thread switching

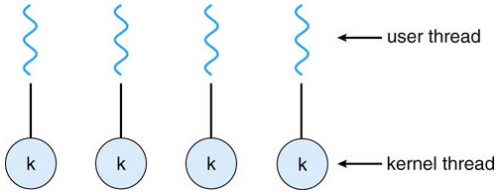
- Threads are executed in an interleaved way, even entirely concurrent
- Switching between threads: two possibilities
  - kernel activity
    - kernel maintains execution state of thread
    - similar to process switching, but without context

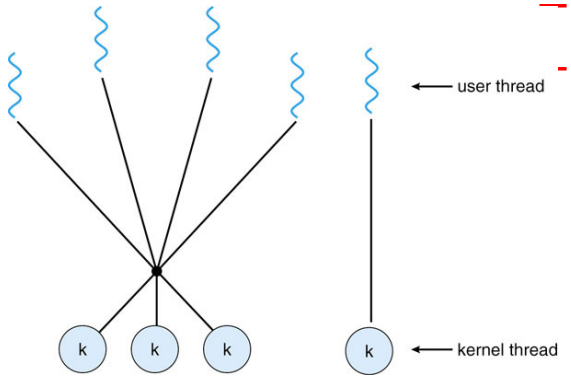


  - user level thread package (library) (user level thread: 'fiber')
    - the package maintains execution state of threads (kernel knows just one thread)
    - the package must obtain control in order to switch threads
    - usually, limited number of switching points
      - only within the package (typically: change return address and stack pointer, other registers not needed)
      - responsibility of programmer to call package often enough – 'cooperative multithreading'
      - could use timer interrupt – however, accessing *detailed* execution state in user mode (registers etc.) is not straightforward or even impossible

Thread execution model



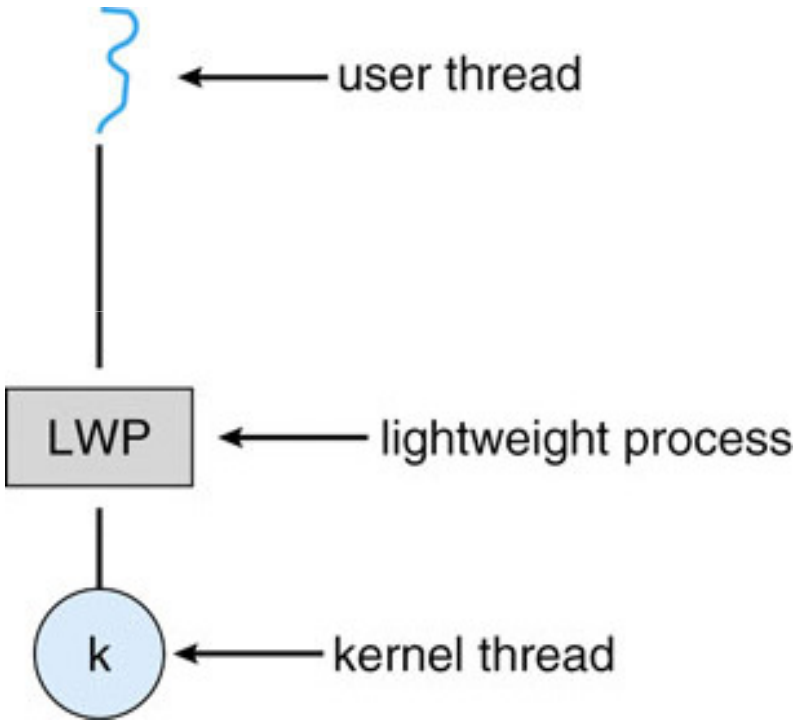




- **Concurrency level**: number of “engines” (virtual processors) actually executing the threaded program
  - virtual processor: called light-weight process, kernel-level thread, ...
- Virtual processors are scheduled by the kernel
- Concurrency level choices
  - 1: no concurrency, no kernel activity in switching.
    - *user level threads*
    - thread switching happens in the library, which must be called often enough
  - # threads: switching always becomes kernel activity
  - in between: only blocking kernel calls and virtual processor scheduling requires kernel activity
    - many-to-many or two-level model
    - blocking call: kernel warns the thread package using an ‘up-call’ (actually, an API extension to help user thread implementation)

Thread execution

- For the two-level model: lightweight process as a user-level represention of a kernel thread
  - LWP just executes user-level threads
  - blocking calls in user space or call to thread package switch LWP to new user thread
    - possibly using an 'upcall' (= callback of the kernel to the thread package)
  - user threads can be bound to an LWP, or unbound (mapped to an available one)



Further thread issues

- What happens upon a blocking system call?
  - suspend the entire process?
- What to do upon *fork(), exec()* ?
- How to handle signals? Which thread receives it?
- How to kill threads and recover resources?
- How to manage priorities?