Operating Systems
Courtesy of BGU-CSE and Dr. Itamar Cohen

Tutorial 3: Threads

Ben-Gurion University of the Negev
Communication Systems Engineering Department

Outline

- **Motivation & basics**
- User-level threads
- Kernel-level threads
- Coding threads

Motivation

- Suppose editing a large WORD doc. This requires
  - User interaction: eg deleting a line and echoing it on the screen
  - Editing the whole document: eg, after a line was deleted, the rest of document is move 1 line upward
  - Automatic disk backups
- Similar scenarios: Editing a .xls, (un)zipping, ...
- One process
  - □ Too slow
- Many processes:
  - □ Problem: processes cannot access the same place (file) in memory
  - □ Long *context switches*
  - □ High system cost (entries in the process table)
- □ Solution: thread – a refined division of tasks

Threads - General

- One / multiple thread(s) within a process
- Allows multiple independent executions under the same process
- Possible states:
  - Running
  - Ready
  - Blocked
  - (Terminated)

Threads - Advantages

- **Share** open files, data structures, global variables, child processes, etc.
- Peer threads can **communicate** without using system calls.
- Threads are 10-100 times **faster** to create / terminate / switch than processes.

Threads - Disadvantages

- Security & stability: open files, data structures, global variables, child processes etc are shared
- Signaling a thread affects all threads of that process

Threads vs. Processes

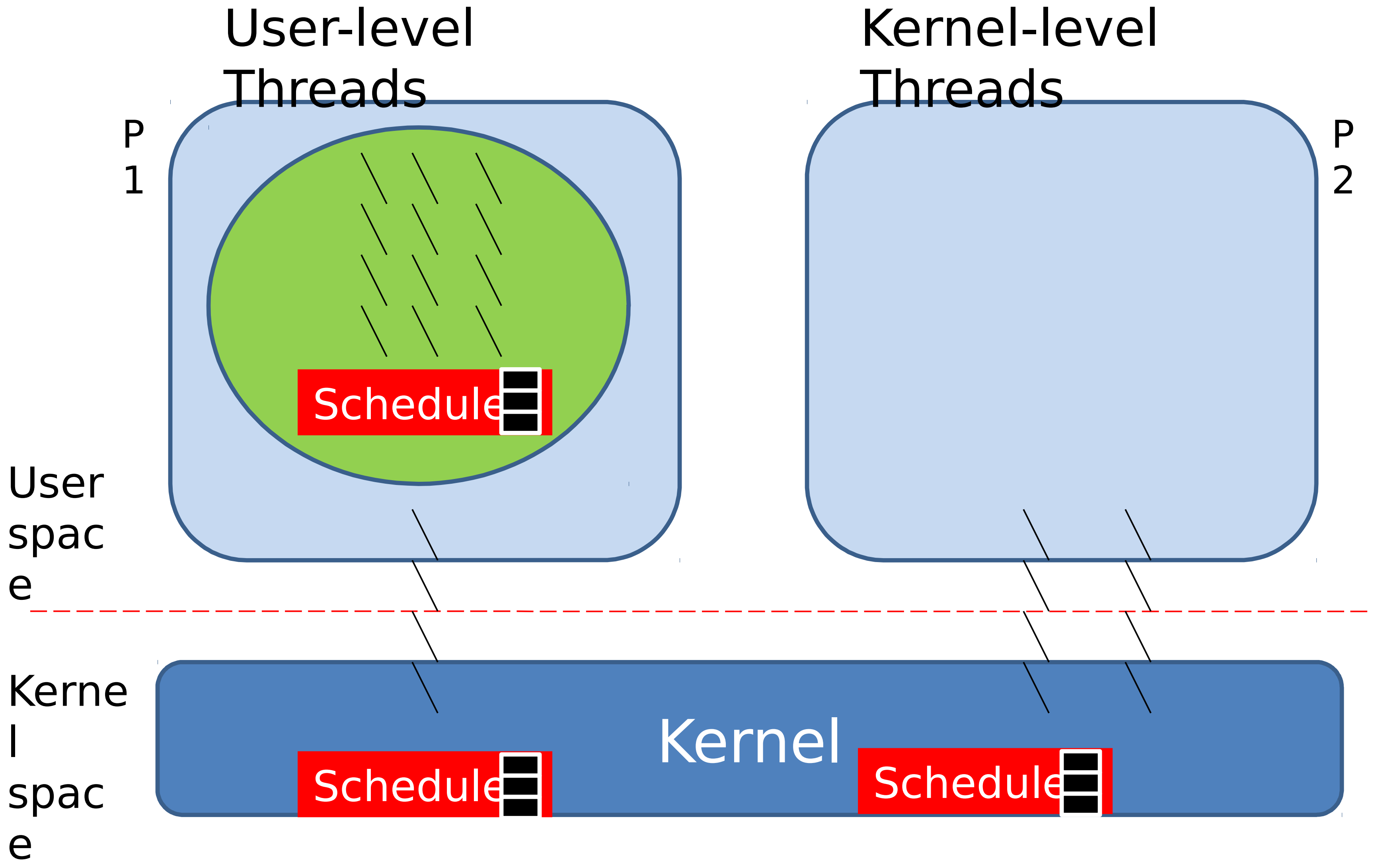
Threads	Processes
shared data	unique data
shared code	unique code
shared open I/O	unique open I/O
*shared signal table	*unique signal table
unique stack	unique stack
unique PC	unique PC
unique registers	unique registers
unique state	unique state
light context switch	heavy context switch

* Signal handlers must be shared among all threads of a multithreaded app. However, each thread must have its own mask of pending / blocked signals: use pthread_mask rather then sigprocmask.

Implementation dilemmas

- fork()
  - Duplicate the calling thread / all threads?
  - OS dependent
    - Many UNIX systems implement both types of fork() (e.g. Solaris 10).
    - In Linux, only the forking thread is duplicated. However, this causes problems, eg in case where the child holds mutex.
- exec()
  - Does the command replace the entire process?
    - Yes.
- Divide process into threads by…
  - the user / the kernel?

User-Level / Kernel-Level Threads



Outline

- Motivation & basics
- **User-level threads**
- Kernel-level threads
- Coding threads

User-Level Threads

- Implemented in user-level libraries
- Thread switching does not need to call the OS or to cause interrupts.
- The user application schedules the process’s CPU time for its internal threads
- Utilizes only a single CPU, as the OS won’t allocate multiple CPUs for one process

User-Level Threads - Advantages

- ▯ ***Compatibility***: Can be implemented on an Operating System that does not support threads.
- ▯ ***Simple Representation****:* Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
- ▯ ***Simple & fast management****:* Creating a thread, synchronization and switching between threads can all be done without intervention of the kernel, and are therefore cheaper and ~100 times faster then in kernel-level threads.

User-Level Threads - Disadvantages

- □ Lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespective of whether process has one thread or 1000 threads within.
- □ A system call (in one of the thread) causes the OS to block the whole process, even if there are runnable threads left in that processes.
- □ The kernel’s inability to notice between user level threads makes it difficult to design preemptive (timed?) scheduling between threads of the same process

Outline

- Motivation & basics
- User-level threads
- **Kernel-level threads**
- Coding threads

Kernel-Level Threads

- All threads are visible to the kernel
- The kernel manages and schedules the threads
- There exist system calls to create and manage threads

Kernel-Level Threads - Advantages

- ▯ The kernel can smartly schedule between processes with different number of threads
- ▯ Kernel-level threads are especially good for applications that frequently block
- ▯ In a multi-processor, a few CPUs can run simultaneously different threads of the same process

Kernel-Level Threads - Disadvantages

- □ Creation, management and switching of threads is MUCH more expensive and slow then user-level treads.

User-Level vs. Kernel-Level Threads

Kernel-level threads	User-level threads	
Visible to the kerne	[redacted]	[redacted]
Kernel defined	[redacted]	[redacted]
Preemptive	[redacted]	[redacted]
Slower, done by the kernel	[redacted]	[redacted]
a single thread	[redacted]	[redacted]
kernel	process	Thread table held by…

Outline

- Motivation & basics
- User-level threads
- Kernel-level threads
- **Coding threads**

POSIX Threads & global variables

- Historically, POSIX functions assumed a single thread per process.
    - E.g., consider a naive implementation of `errno` in a multi threaded environment.
- Hence, the need for reentrant functions.
- While this is supported by many standard functions, the compiler must be aware of the need for re-entrant functions:
    - **`gcc –D_REENTRANT –lpthread …`**

Threads in POSIX: pthreads

int **pthread_create** (pthread_t* thread, pthread_attr_t* attr, void* (*start_func)(void*) , void* arg)
Creates a new thread of control that executes concurrently with the calling thread.
On success, the identifier of the newly created thread is stored in the location pointed by the *thread* argument, and a 0 is returned. On error, a non-zero error code is returned.
attr enables applying attributes to the new thread (e.g. detached, scheduling-policy). Can be NULL (default attributes).
start_func is a pointer to the function the thread will start executing. The function receives one argument of type *void* and returns a void**.
arg is the parameter to be given to *func*.
pthread_t **pthread_self** ()_
Returns this thread's identifier.

Threads in POSIX (pthreads) – cont.

int **pthread_join** (pthread_t th, void** thread_return)
Suspends the execution of the calling thread until the thread identified by *th* terminates.
On success, the return value of *th* is stored in the location pointed by *thread_return*, and a 0 is returned. On error, a non-zero error code is returned.
At most one thread can wait for the termination of a given thread. Calling **pthread_join** on a thread *th* on which another thread is already waiting for termination returns an error.
th is the identifier of the thread that needs to be waited for
thread_return is a pointer to the returned value of the *th* thread (can

void **pthread_exit** (void* ret_val)_
Terminates the execution of the calling thread. Doesn't terminate the whole process if called from the main function.
If *ret_val* is not null, then *ret_val* is saved, and its value is given to the thread which performed *join* on this thread; that is, it will be written to the *thread_return* parameter in the **pthread_join** call.

Example files

- thread_hello_world.c
- thread_exercise2.c