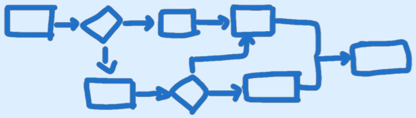
Imperial College
London



Operating Systems

Threads

Course 211
Spring Term 2018-2019

http://www.imperial.ac.uk/computing/current-students/courses/211/calendar/

(Slides courtesy of Cristian Cadar)

Peter Pietzuch

prp@doc.ic.ac.uk
http://www.doc.ic.ac.uk/~prp

What Are Threads?

Execution streams that share the **same address space**

When multithreading is used, each process can contain one or more threads

Per process items	**Per thread items**
Address space	Program counter (PC)
Global variables	Registers
Open files	Stack
Child processes	
Signals	

Why Threads?

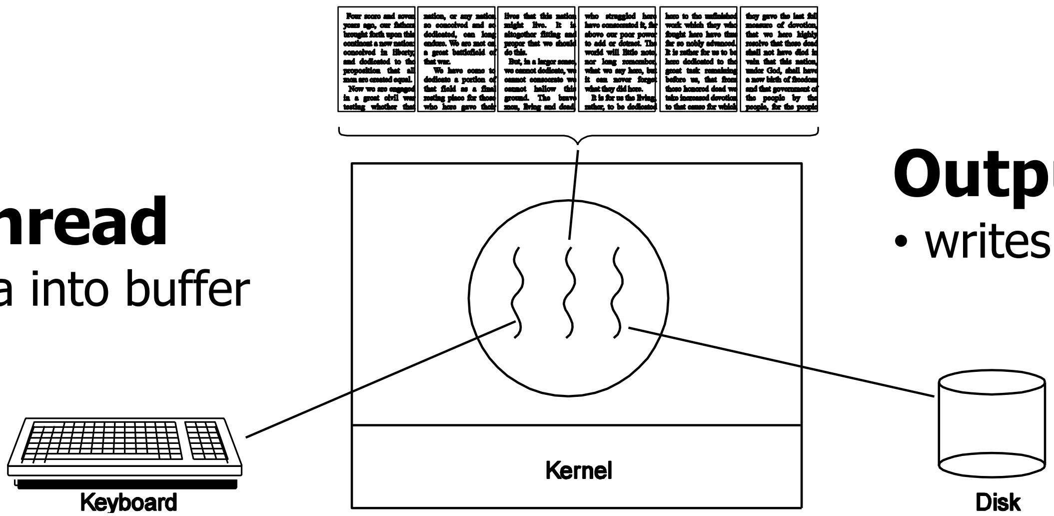
Many applications contain multiple activities

- Which execute in parallel
- Which access and process the same data
- Some of which may block

Processing thread
- processes input buffer
- writes result into output buffer

Input thread
- reads data into buffer

Output thread
- writes output buffer to disk



Why Not Processes?

Many applications contain multiple activities

- Which execute in parallel
- Which access and process the same data
- Some of which might block

Processes are too heavyweight

- Difficult to communicate between different address spaces
- Activity that blocks may switch out the entire application
- Expensive to context switch between activities
- Expensive to create/destroy activities

Threads – Problems/Concerns

Shared address space

- Memory corruption
  - One thread can write another thread's stack
- Concurrency bugs
  - Concurrent access to shared data (e.g. global variables)

Forking

- What happens on a `fork()`?
  - Create a new process with the same number of threads?
  - Create a new process with a single thread?

Signals

- When a signal arrives, which thread should handle it?

Case Study: PThreads

PThreads (Posix Threads)

Defined by IEEE standard 1003.1c

- Implemented by most UNIX systems

```
#include <pthread.h>
#include <sys/types.h>
```

`pthread_t` → type representing a thread

`pthread_attr_t` → type representing the attributes of a thread

Creating Threads

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

Creates a new thread

- Newly created thread is stored in **`*thread`**
- Function returns 0 if thread was successfully created, or error code

Arguments:

- **`attr`** **->** specifies thread attributes, can be **`NULL`** for default attributes (attributes include: minimum stack size, guard size, detached/ joinable, ...)
- **`start_routine`** -> C function the thread will start execute once created
- **`arg`** -> Argument to be passed to **`start_routine`** (of pointer type **`void*`**).
  Can be **`NULL`** if no arguments are to be passed.

Terminating Threads

```
void pthread_exit(void *value_ptr);
```

Terminates the thread and makes **`value_ptr`** available to any successful join with the terminating thread

Called implicitly when the thread's start routine returns

- But not for the initial thread which started **`main()`**
- If **`main()`** terminates before other threads w/o calling **`pthread_exit()`**, the entire process is terminated
- If **`pthread_exit()`** is called in **`main()`**, the process continues executing until last thread terminates (or **`exit()`** is called)

PThread Example (1)

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

void *thread_work(void *threadid) {
  long id = (long) threadid;
  printf("Thread %ld\n", id);
}

int main (int argc, char *argv[]) {
  pthread_t threads[5];
  long t;
  for (t=0; t<5; t++)
      pthread_create(&threads[t], NULL,
                     thread_work, (void *)t);

}
```

```
$ gcc pt.c -lpthread
$ ./a.out
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
```

Question: Passing Arguments to Threads

What if we want to pass more than one argument to the start routine?

Create structure containing arguments and pass pointer to that structure to **`pthread_create()`**

Yielding the CPU

```
int pthread_yield(void)
```

Releases CPU to let another thread run

Returns 0 on success, or an error code

Always succeeds on Linux

Why would a thread ever yield? (think of `nice()` for processes)

Joining Other Threads

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Blocks until **`thread`** terminates

Value passed to **`pthread_exit()`** by terminating thread is available in location referenced by **`value_ptr`**

`value_ptr` can be **`NULL`**

Join Example

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

long a, b, c;
void *work1(void *x) { a = (long)x *
  (long)x;}
void *work2(void *y) { b = (long)y *
  (long)y;}

int main (int argc, char *argv[]) {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, work1, (void*)
  3);
  pthread_create(&t2, NULL, work2, (void*)
  4);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  c = a + b;
  printf("3^2 + 4^2 = %ld\n", c);
}
```

```
$ ./a.out
3^2 + 4^2 = 25
```

Two Ways to Implement Threads

User-level threads

- The kernel is not aware of threads
- Each process manages its own threads

Kernel-level threads

- Managed by the kernel

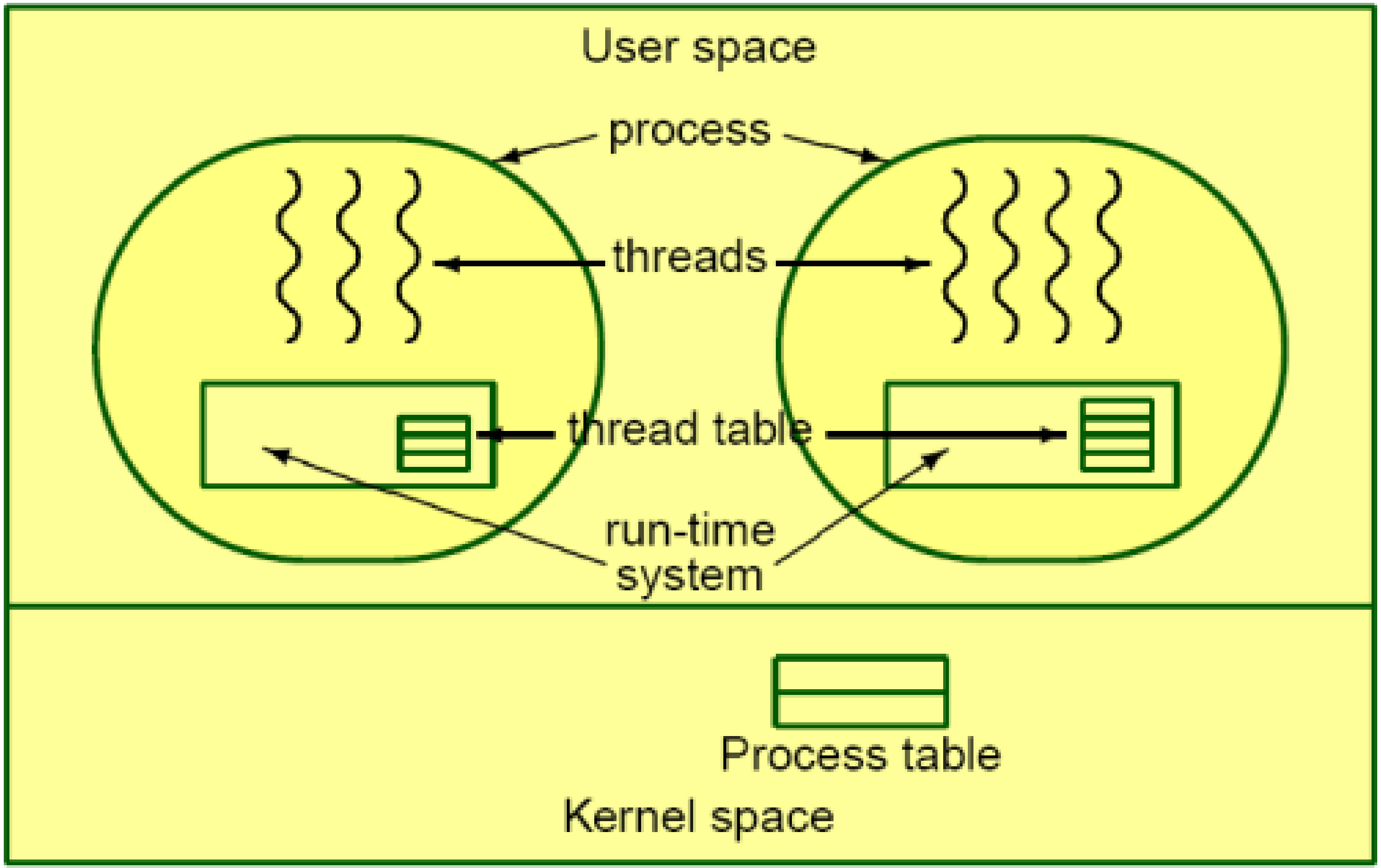
Trade-offs on each side

Various hybrid approaches possible

User-Level Threads

Kernel thinks it is managing processes only

- Threads implemented by software library
- Process maintains **thread table** for **thread scheduling**



Advantages of User-Level Threads

Better performance

- Thread creation and termination are fast
- Thread switching is fast
- Thread synchronisation (e.g. joining other threads) is fast
- All these operations do not require any kernel involvement

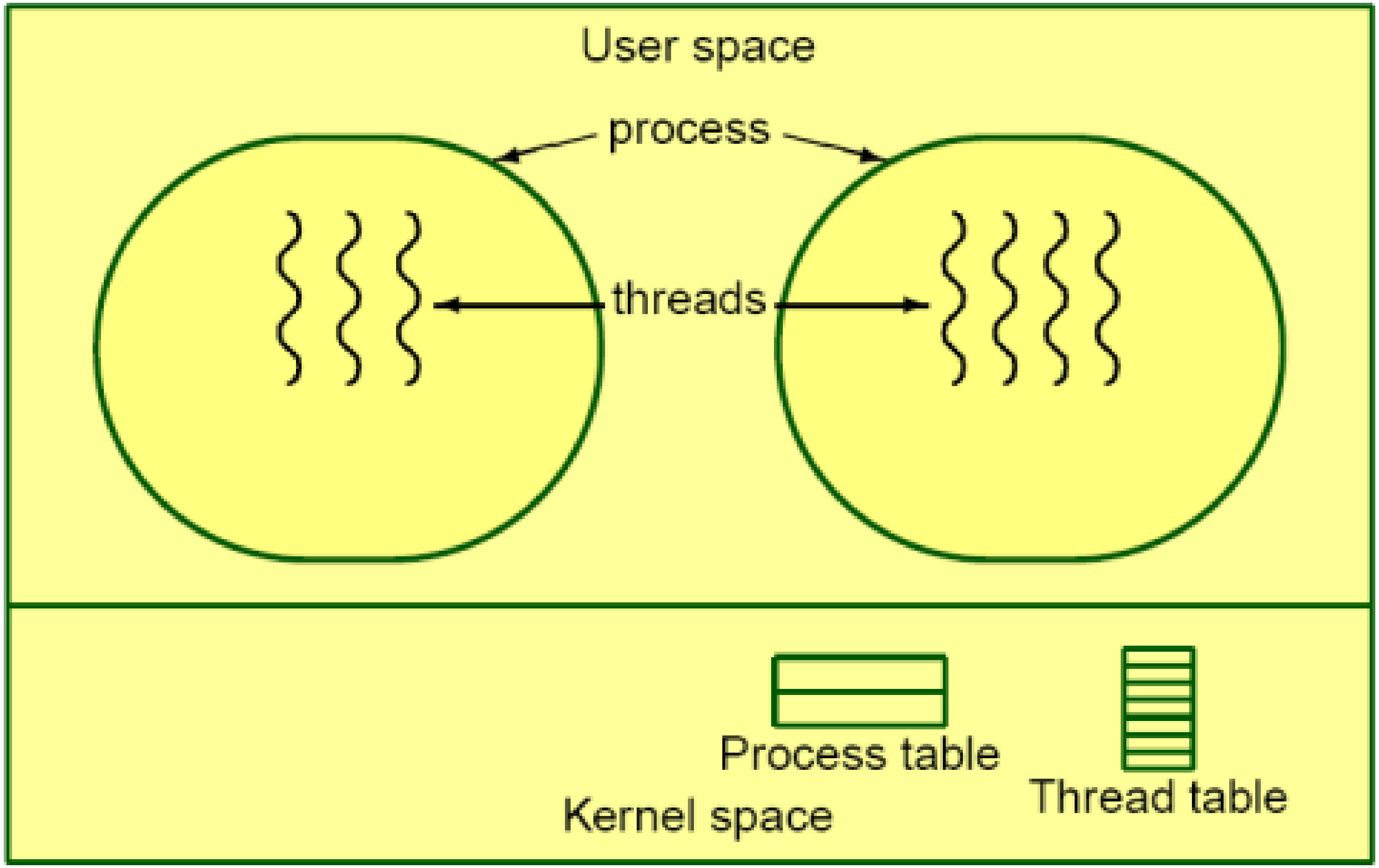
Each application can have its own scheduling algorithm

Disadvantages of User-Level Threads

- Blocking system calls stops **all threads** in process
  - Denies one of core motivations for using threads
- Non-blocking I/O can be used (e.g. **`select()`**)
  - Harder to use and understand, inelegant

- During page fault, OS blocks whole process…
  - But other threads may be runnable

Kernel Threads



Advantages of Kernel Threads

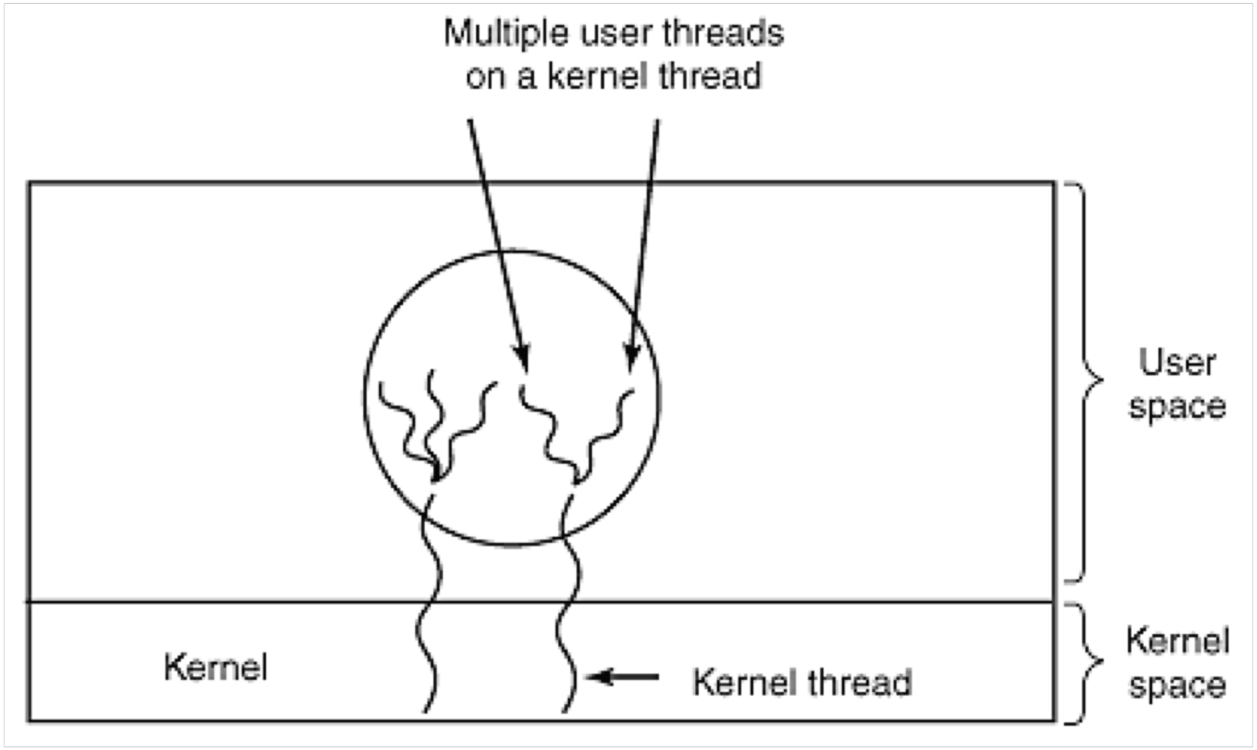
- Blocking system calls/page faults can be easily accommodated
  - If one thread calls a blocking system call or causes a page fault, the kernel can schedule a runnable thread from the same process

Disadvantages of Kernel Threads

- Thread creation and termination more expensive
  - Require system calls
    - But still much cheaper than process creation/termination
  - One mitigation strategy is to recycle threads (**thread pools**)
- Thread synchronisation more expensive
  - Requires blocking system calls
- Thread switching more expensive
  - Requires system call
    - But still much cheaper than process switches (same address space)
- No application-specific schedulers

Hybrid Approaches

Use kernel threads and multiplex user-level threads onto some (or all) kernel threads



Tutorial Question

If in a multithreaded web server the only way to read from a file is the normal blocking `read()` system call, do you think user-level threads or kernel-level threads are being used? Why?

A worker thread will block when it has to read a web page from disk. If user-level threads were used, this action would block the entire process, destroying the value of multi-threading. Thus is is essential that kernel threads are used to permit some threads to block without affecting others.

Tutorial Question

You are to compare reading a file using a **single-threaded file server** and a **multithreaded server**, running on a single-CPU machine.

It takes 15 ms to get a request for work, dispatch it, and do the rest of the necessary processing, assuming that the data needed are in the block cache.

A disk operation is needed 1/3 of the time, requiring an additional 75 ms, during which time the thread sleeps. Assume that thread switching time is negligible.

How many requests/sec can the server handle if it is (a) single-threaded and (b) multi-threaded?

Tutorial Question - Answer

Single-threaded case:

Cache hit = 15ms Cache miss = 90ms

Weighted average: 2/3 * 15ms + 1/3 * 90 = 40 ms

Server can do 25 req/sec

Multi-threaded case (with preemptive scheduling):

On average, each requests needs 15 ms CPU time and 1/3 * 75 = 25 ms I/O time

Probability of all n threads sleeping: $(25/40)^n = (5/8)^n$

CPU utilization: $1 – (5/8)^n$

In 1000ms, CPU handles $[1-(5/8)^n] * 1000/15$ requests

n=1: 25 req/s; n=2: 40.62 req/s; n=6: 62.69 req/s