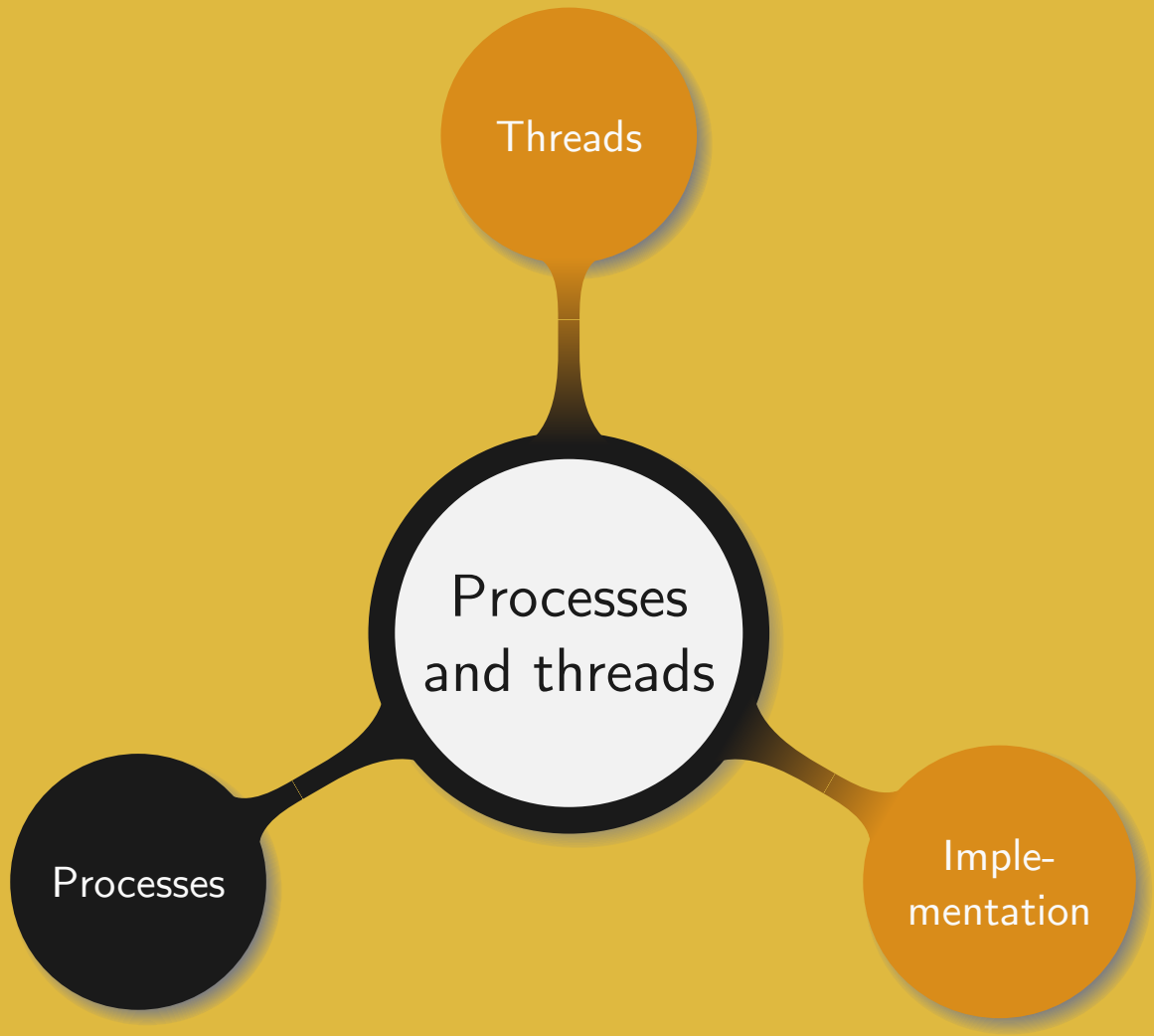
# Introduction to Operating Systems

## 2. Processes and threads

Manuel – Fall 2021

# Chapter organisation

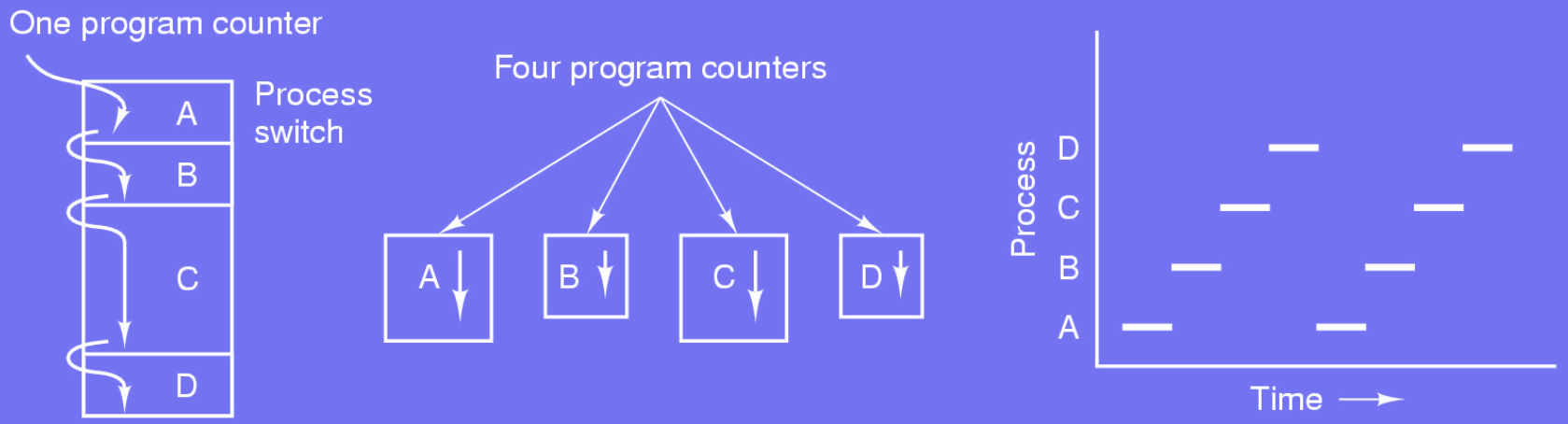


# Basics

A *process* is an abstraction of a running program:

- At the core of the OS
- Process is the unit for resource management
- Oldest and most important concept
- Turn a single CPU into multiple virtual CPUs
- CPU quickly switches from process to process
- Each process run for 10-100 ms
- Processes hide the effect of interrupts

# Multiprogramming




Multiprogramming strategies and issues:

- CPU switches rapidly back and forth among all the processes
- Rate of computation of a process is not uniform/reproducible
- Potential issue under time constraints; e.g.
  - Read from tape
  - Idle loop for tape to get up to speed
  - Switch to another process
  - Switch back... too late

# Program vs. process

Differences between programs and processes:

- Running twice a program generates two processes
- Program: sequence of operations to perform
- Process: program, input, output, state

# Program vs. process

Differences between programs and processes:

- Running twice a program generates two processes
- Program: sequence of operations to perform
- Process: program, input, output, state

*Describe the process of baking a cake when the phone rings*

# Process creation and termination

Times at which processes are created:

- System initialization
- Upon a user launching a new program
- Initialization of a batch job

Any created processes ends at some stage:

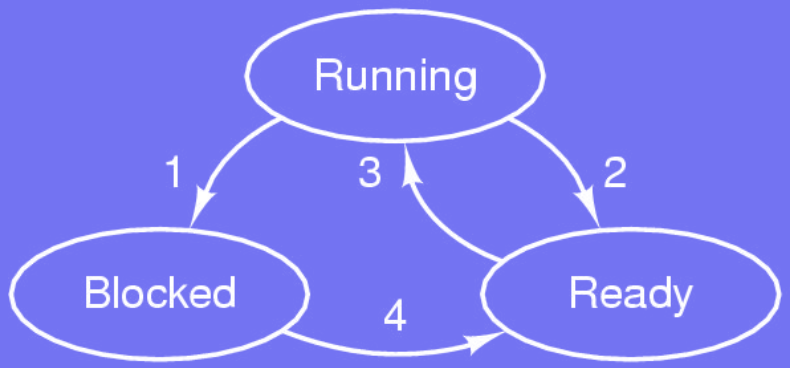
- Voluntarily:
  - The work is completed, issue a system call to inform the OS
  - An error is noticed, the process exits nicely
- Involuntary:
  - Fatal error, program crashes
  - Another process kills it

# Process hierarchies

Two main approaches:

- UNIX-like systems:
  - A parent creates a child
  - A child can create its own child
  - The hierarchy is called *process group*
  - It is impossible to disinherit a child
- Windows system:
  - All processes are equal
  - A parent has a token to control its child
  - A token can be given to another process

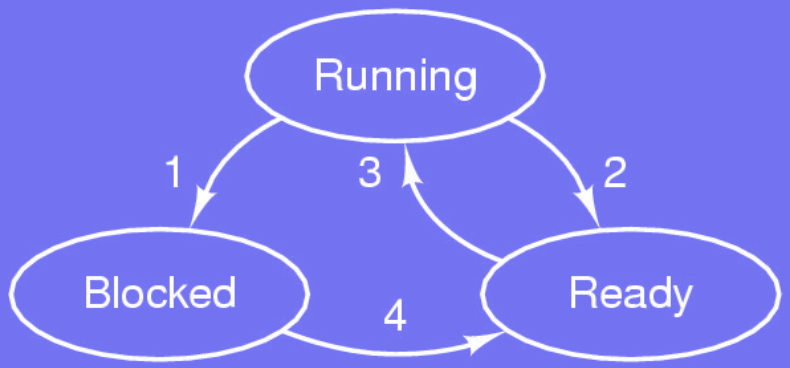
# Process states




Possible states:

1. Waiting for some input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

# Process states




Possible states:

1. Waiting for some input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Change of perspective on the inside of the OS:

- Do not think in terms of interrupt but of process
- Lowest level of the OS is the scheduler
- Interrupt handling, starting/stopping processes are hidden in the scheduler

# Modeling processes

A simple model for processes:

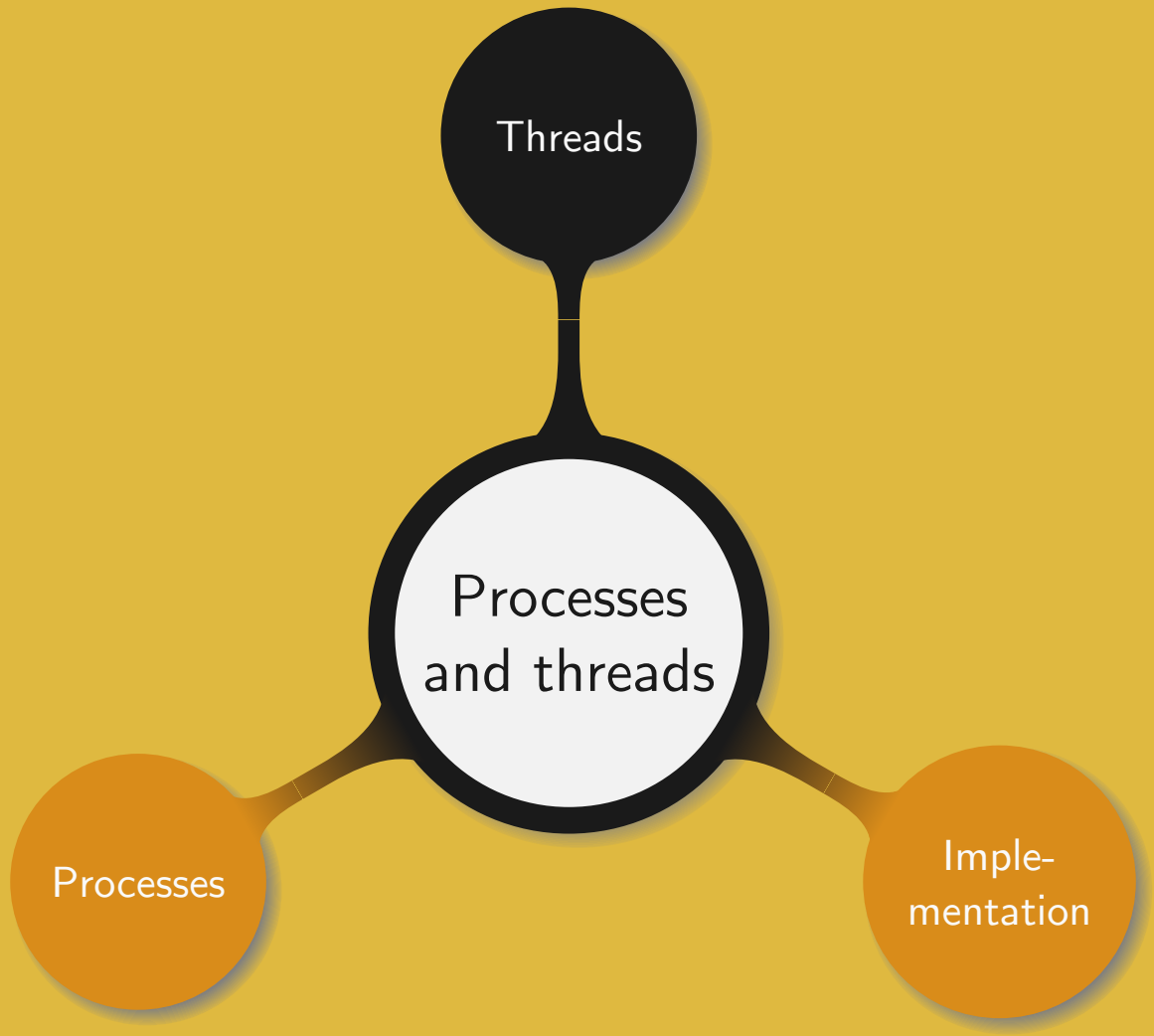
- A process is a data structure called *process control block*
- The structure contains important information such as:
  - State
  - Program counter
  - Stack pointer
  - Memory allocation
  - Open files
  - Scheduling information
- All the processes are stored in an array called *process table*

# Interrupts and processes

Upon an interrupt the running process must be paused:

1. Push on the stack the user program counter, PSW, etc.
2. Load information from interrupt vector
3. Save registers (assembly)
4. Setup new stack (assembly)
5. Finish up the work for the interrupt
6. Decides which process to run next
7. Load the new process, i.e. memory map, registers, etc. (assembly)

# Chapter organisation



# Overview

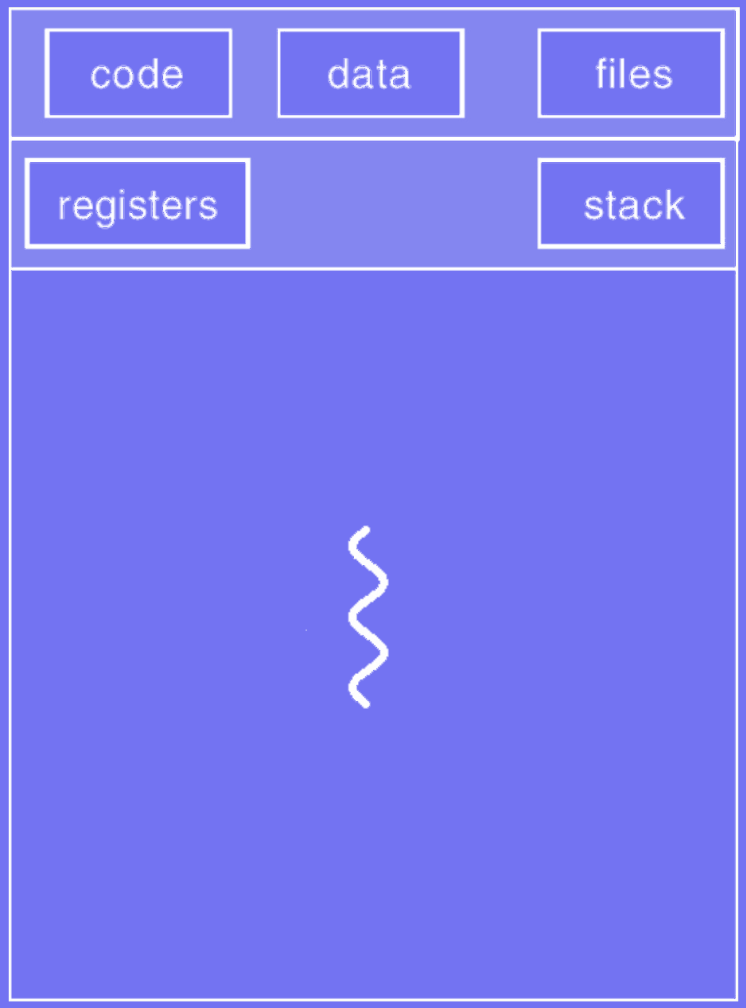
A thread is the basic unit of CPU utilisation consisting of:

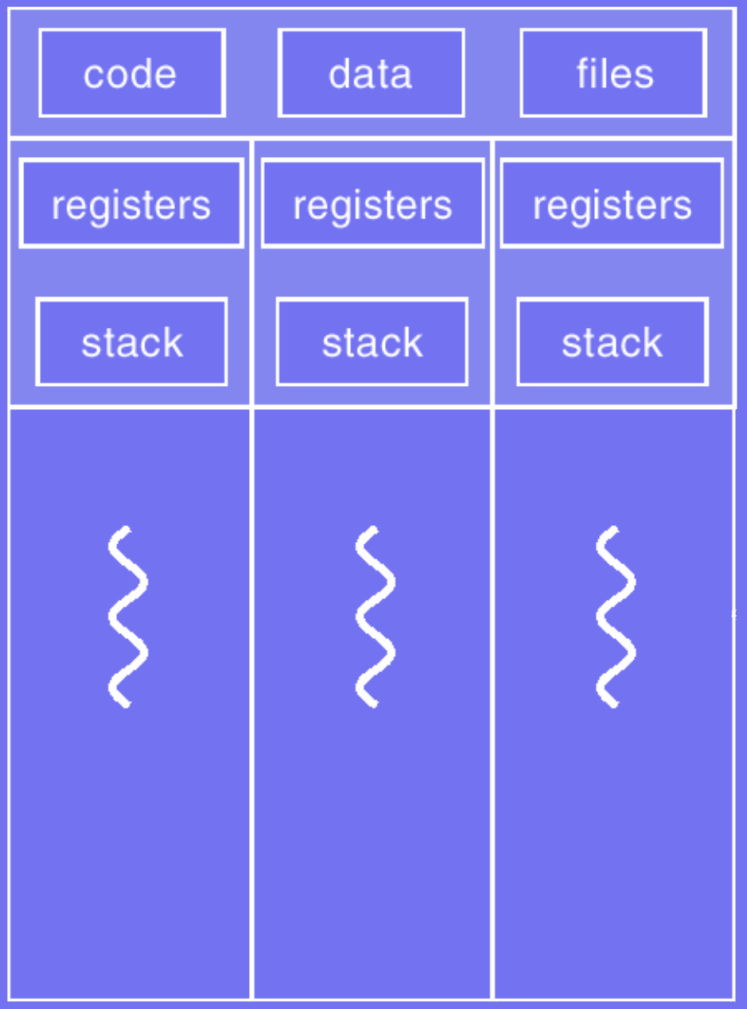
- A thread ID
- The program counter
- A register set
- A stack space

All the threads within a process share the same:

- Code section
- Data section
- Operating system resources

# Single vs. multi-threaded



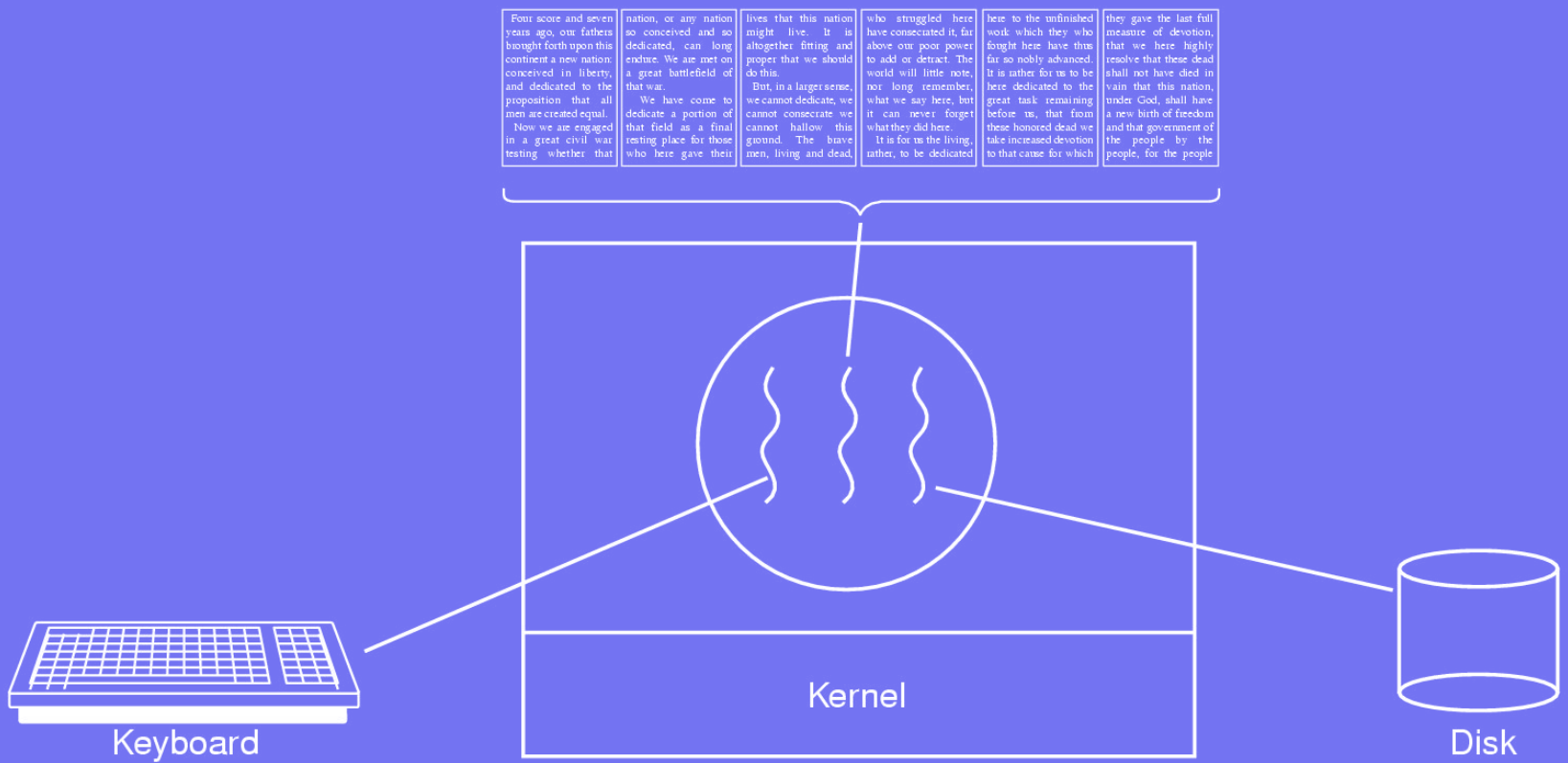



# Notes and remarks

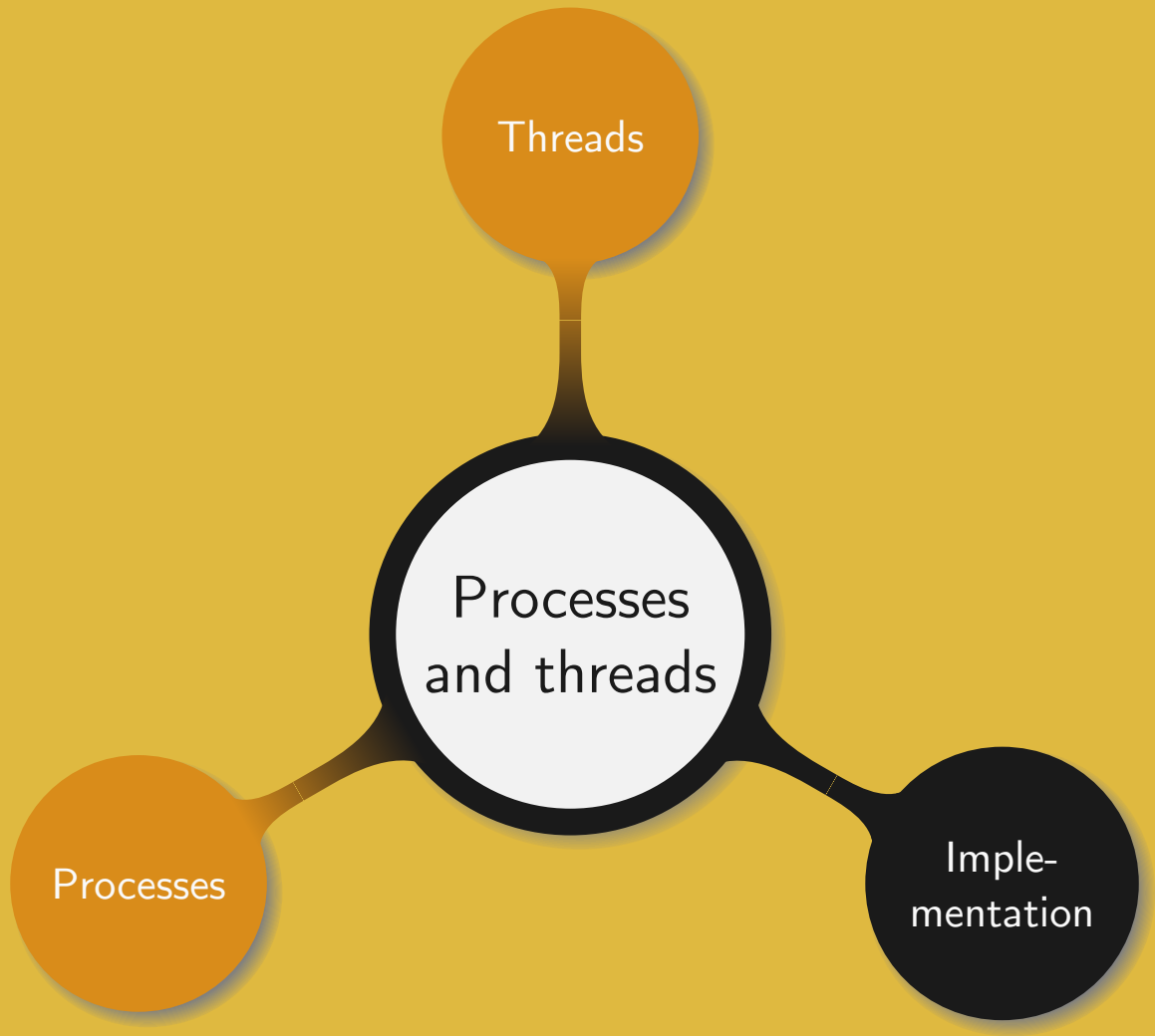
Processes and threads:

- A thread has the same possible states as a process
- Transitions are similar to the case of a process
- Threads are sometimes called lightweight processes
- No protection is required for threads, compared to processes
- A process starts with one threads and can create more
- Processes want as much CPU as they can
- Threads can give up the CPU to let others using it

# Example of a word processor



# Chapter organisation



# POSIX threads

The `pthread` library has over 60 function calls:

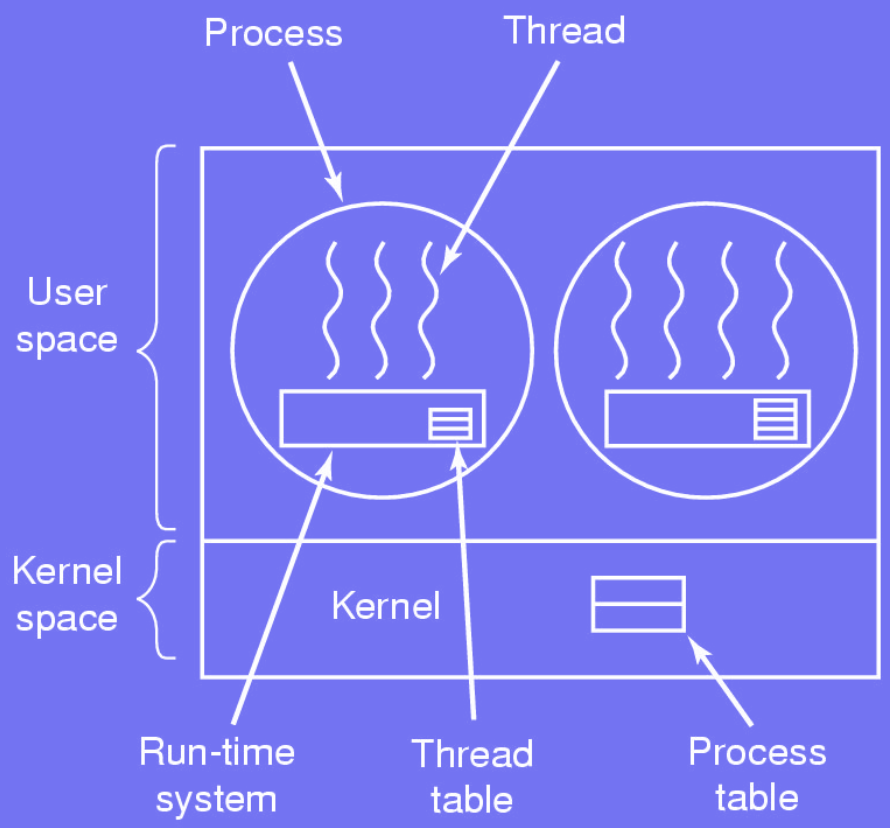
- Create a thread: `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);`
- Terminate a thread: `void pthread_exit(void *retval);`
- Wait for a specific thread to end: `int pthread_join(pthread_t thread, void **retval);`
- Release CPU to let another thread run: `int pthread_yield(void);`
- Create and initialise a thread attribute structure: `int pthread_attr_init(pthread_attr_t *attr);`
- Delete a thread attribute object: `int pthread_attr_destroy(pthread_attr_t *attr);`

# Creating ten threads and printing their ID

threads.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define THREADS 10
void *gm(void *tid) {
  printf("Good morning from thread %lu\n",*(unsigned long int*)tid);
  pthread_exit(NULL);
}
int main () {
  int status, i; pthread_t threads[THREADS];
  for(i=0;i< THREADS;i++) {
    printf("thread %d\n",i);
    status=pthread_create(&threads[i],NULL,gm,(void*)&(threads[i]));
    if(status!=0) {
      fprintf(stderr,"thread %d failed with error %d\n",i,status);
      exit(-1);
    }
  }
  exit(0);
}
```

# Threads in user-space – N:1



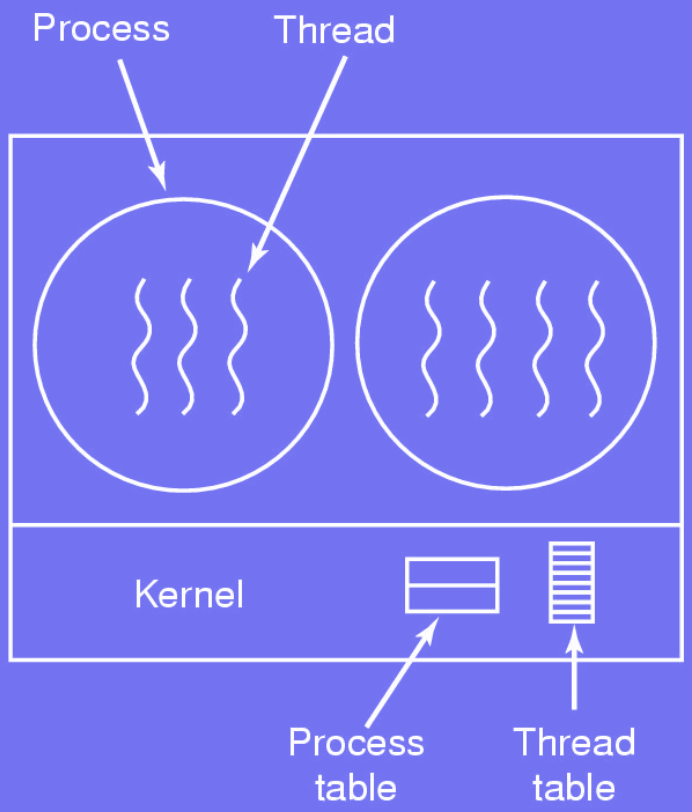

User-space threads:

- Kernel thinks it manages single threaded processes
- Threads implemented in a library
- Thread table similar to process table, managed by run-time system
- Switching thread does not require to trap the kernel

Questions.

- What if a thread issues a blocking system call?
- Threads within a process have to voluntarily give up the CPU

# Thread in the kernel – 1:1



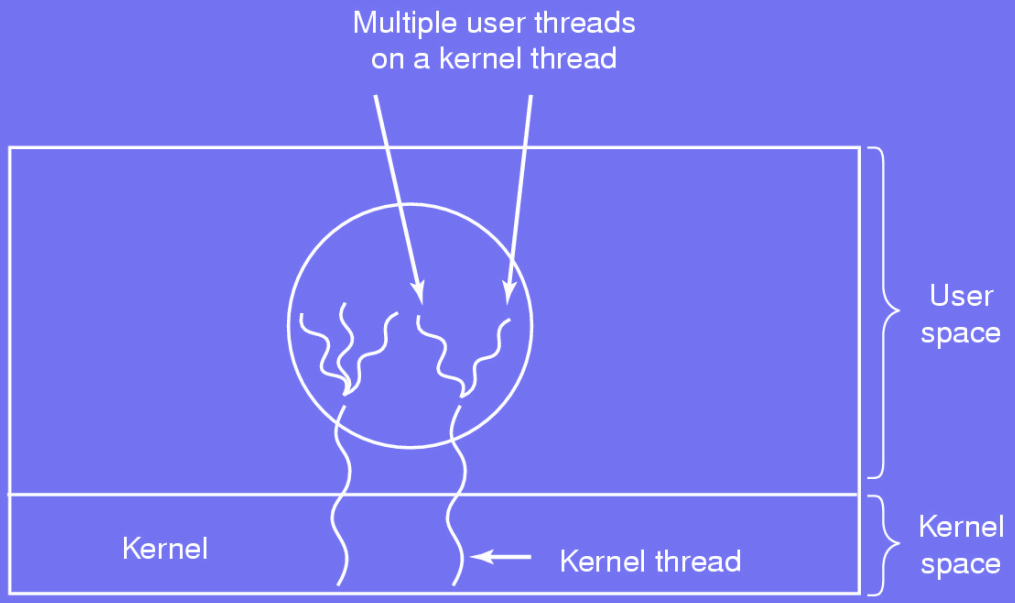

Kernel space thread:

- Kernel manages the thread table
- Kernel calls are issued to request a new thread
- Calls that might block a thread are implemented as system call
- Kernel can run another thread in the meantime

Questions.

- Why does it have a much higher cost than user space threads?
- Signals are sent to processes, which thread should received it?

# Hybrid threads – M:N




Hybrid threads:

- Compromise between user-level and kernel-level
- Threading library schedules user threads on available kernel threads

Questions.

- How to implement hybrid threads?
- How to handle scheduling?

# Notes and remarks

Best thread approach:

- Hybrid looked attractive
- Most systems are coming back to 1:1
- Different approaches exist on how to use threads
  e.g. thread bocks on “receive system call” vs. pop up threads
- Switching implementation from single thread to multiple thread is not easy task
- Requires redesigning the whole system
- Backward compatibility must be preserved
- Research still going on to find better ways to handle threads

# Key points

- What is a process?
- How can processes be created and terminated?
- What are the possible states of a process?
- What is the difference between single thread and multi-threads?
- What approaches can be taken to handle threads?

# Thank you!