COMP2300/6300

Computer Organisation and Prog[illegible]tion

Operating Systems

Dr Charles Martin

Semester 1, 2022

Admin Time

Little wires and alternatives—se ere

Quiz 2 **open**!

Midsem feedback: marks out on streams, stats here, detailed feedback tonight.

Assignment 2! Go work on it enough to *ask a question*.

(This week I'm working through some old issues on Piazza – thanks for your patience!)

Week 10: Operating Systems

Outline

- what is an OS?
- privilege levels
- processes & scheduling

What is an OS?

















talk

what *is* an **operating system** (OS)

…it's a virtual machine

offering a more familiar, comfort[illegible]afer environment for your programs to run in

- memory management
- hardware abstraction
- process management
- inter-process communication (IPC)

…it's a resource manager

co-ordinating access to hardware resources

- processors
- memory
- mass storage
- communication channels
- devices (timers, GPUs, DSPs, other peripherals…)

multiple tasks/processes/programs may be applying for access to these resources!

A brief history of operating systems...

1950s-60s: system monitors

IBM 704 mainframe at NACA in 1

NASA / Public Domain

1960s - multi-programming system

University of Manchester Atlas, 1963

1970s - multi-tasking systems

DEC PDP-11

1970s - early workstations...

Xerox Alto

Joho345 / Public domain

1970s-80s: Consumer OSs

1990s+

NeXTStation

A brief history of operating systems (1)

in the beginning: single user, single prog sk, serial processing—no OS

- 50s: system monitors/batch pr
  - the monitor ordered the sequence of jobs and triggered their sequential execution
- 50s-60s: advanced system monitors/batch processing:
  - the monitor handles interrupts and timers
  - first support for memory protection
  - first implementations of privileged instructions (accessible by the monitor only)
- early 60s: multi-programming systems:
  - use the long device I/O delays for switches to other runnable programs
- early 60s: multi-programming, time-sharing systems:
  - assign time-slices to each program and switch regularly

A brief history of operating systems (2)

- early 70s: multi-tasking systems – mu[illegible]oments resulting in UNIX (and others)
- early 80s: single user, single tasking s[illegible]emphasis on user interface or APIs. MS-DOS, CP/M, MacOS and others first employed 'small scale[illegible]onal computers).
- mid-80s: Distributed/multiprocessor operating systems - modern UNIX systems (SYSV, BSD)
- late 70s: Workstations starting by porting UNIX or VMS to 'smaller' computers.
- 80s: PCs starting with almost none of the classical OS-features and services, but with an user-interface (MacOS) and simple device drivers (MS-DOS)

A brief history of operating systems (3)

- last 20 years: evolving and expanding [illegible] general purpose OSes, like for instance:
  - Solaris (based on SVR4, BSD, and S[illegible] etty much dead now)
  - Linux (open source UNIX re-impler[illegible] x86 processors and others)
  - current Windows (used to be partly based on Windows NT, which is ‘related’ to VMS)
  - MacOS (Mach kernel with BSD Unix and a proprietary user-interface)
- multi-processing is supported by all these OSes to some extent
  - but not (really) suitable for embedded, or real-time systems

Standard features?

is there a standard set of feature ting systems?

no.

the term *operating system* covers everything from 4 kB microkernels, to > 1 GB installations of desktop general purpose operating systems

Minimal set of features?

is there a *minimal* set of features

almost: memory management, process management and inter-process communication/synchronisation would be considered essential in most systems

is there always an explicit operating system?

no: some languages and development systems operate with standalone runtime environments

Process management

(we'll talk more about this in a m

basically, this is the task of keep e things going all at once…

…while tricking them all into thinking they're the main game

talk

what's a task?

Memory management

remember memory? the OS is re[illegible] for sharing it around

- allocation / deallocation
- virtual memory: logical vs. physical addresses, segments, paging, swapping, etc.
- memory protection (privilege levels, separate virtual memory segments, …)
- shared memory (for performance, communication, …)

Synchronisation/inter-process communication

remember all the **asynchronism** OS is responsible for managing that as well

semaphores, mutexes, condition channels, mailboxes, MPI, etc.

this is tightly coupled to scheduling / task switching!

Hardware abstraction

remember all the specific load-t[illegible]e addresses in the labs? no?

good news everyone! the OS doe[illegible]on't have to

- device drivers
- protocols, file systems, networking, everything else…

all through a consistent API

Kernel: definition

the **kernel** is the program (functi tructures in memory, etc.) which performs the *core* role(s) of the OS

access to the CPU, memory, peripherals all happens *through* the kernel through a **system call**


程序代写代做 CS编程辅导
WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com


程序代写代做 CS编程辅导
WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com


程序代写代做 CS编程辅导
WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com

info

if you want to look at some real s APIs

on Linux,

- `syscalls.h` **header file**
- **how to add a new system call**

on Windows,

- **Windows API Index**

writing an OS seems complicated

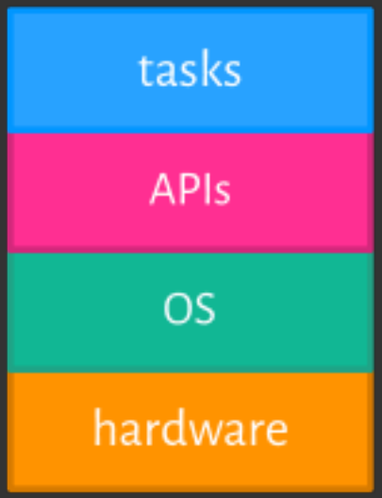
how is it done in practice?

Monolithic OS

(or 'the big mess…')

- non-portable/hard to maintain
- lacks reliability
- all services are in the kernel (on the same privilege level)
- but: may reach high efficiency

e.g.: most early UNIX systems, MS-DOS (80s), Windows (all non-NT based versions) MacOS (until version 9), etc…



monolithic OS

Monolithic & Modular OS

- modules can be platform independer
- easier to maintain and to develop
- reliability is increased
- all services are still in the kernel (on the same privilege level)
- may reach high efficiency

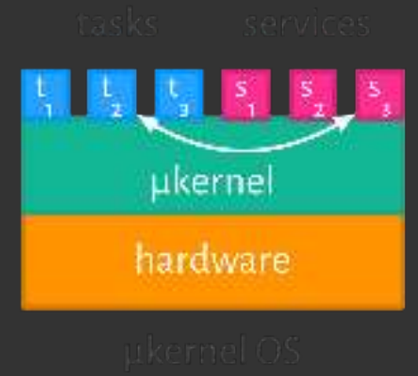
e.g., current Linux versions

monolithic & modular OS

µKernels & client-server models

- µkernel implements essential process[illegible]d message handling
- all 'higher' services are user level ser[illegible]
- kernel ensures reliable message pass[illegible]clients and servers
- highly modular, flexible & maintainable
- servers can be redundant and easily replaced
- (possibly) reduced efficiency through increased communications

e.g., current research projects, µ, L4, Minix 3, etc.



Example: UNIX

- hierarchical file-system (main… `mount` and `unmount`)
- universal file-interface applied… evices (I/O), as well as IPC
- dynamic process creation via duplication
- choice of shells
- internal structure as well as all APIs are based on C
- relatively high degree of portability

many versions/flavours: UNICS, UNIX, BSD, XENIX, System V, QNX, IRIX, SunOS, Ultrix, Sinix, Mach, Plan 9, NeXTSTEP, AIX, HP-UX, Solaris, NetBSD, FreeBSD, Linux, OPENSTEP, OpenBSD, Darwin, QNX/Neutrino, OS X, QNX RTOS, …

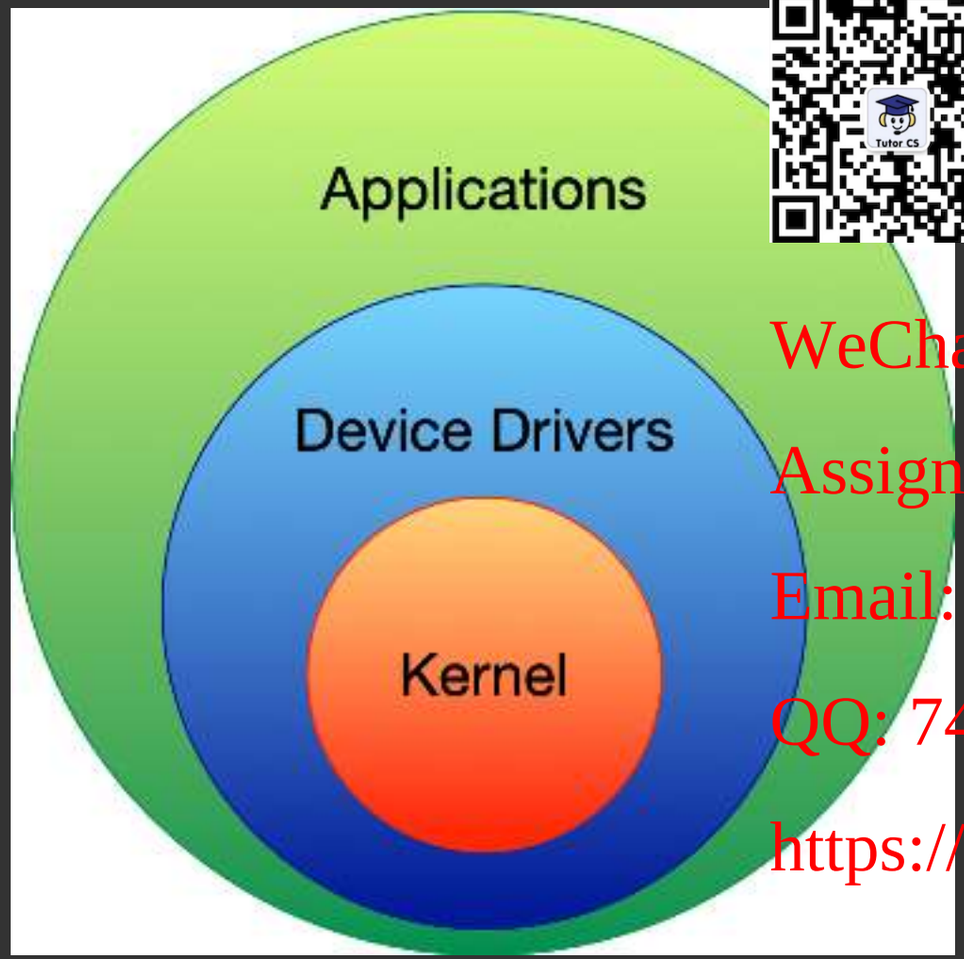
Privilege levels

talk

what do you think privilege mean

how does it affect your code running on microbit?

Privilege



Privilege levels

certain instructions can only be e “privileged” mode—this is enforced in **hardware**

different architectures enforce th ent ways

check the manual (e.g. *Section A2.3.4* on p32 or *Table B1-1 Mode* on p568 of the **ARMv7-M reference manual**

Fun video for the nostalgic: What is DOS protected mode?

ARMv7-M execution levels

	thread mode	**dler** mode
privileged	regular code	xceptions (including interrupts)
unprivileged	regular code	

priviliges may control:

- code execution
- memory read/write access
- register access (e.g., for peripherals)

“Supervisor call” instruction

have you noticed these entries in table in labs?

```
        .word    SVC_Handler
@ ...
        .word    PendSV_Handler
```

the `svc` instruction (A7.7.175 in the reference manual) runs the `SVC_Handler` immediately

Encoding T1 All versions of the Thumb instruction set.

SVC<c> #<imm8>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	1	imm8							

Deferred supervisor call (PendSV)

there's a **PENDSVSET** bit (bit 28)  rupt Control and State Register (`ICSR`)

if set, the `PendSV_Handler` will be called according to the usual interrupt/exception priority rules

both `SVC_Handler` and `PendSV_Handler` run in privileged mode, like *all* interrupts

System calls with `SVC`

How might we implement a sys a microbit?

How can we get at the argument? (Hint: see Figure B1-3 from the ARM-v7 M architecture reference)

How do **system calls work in linux**?

Types of Operating Systems

Distributed operating systems

- all CPUs carry a small kernel ystem for communication services
- all other OS services are distri available CPUs

- services may migrate
- services can be multiplied in order to guarantee availability (hot stand-by), or to increase throughput (heavy duty servers)

Real-time operating systems

- fast context switches? should be fast anyway
- small size? should be small anyway

- quick response to external interrupts? not *quick*, but predictable
- multitasking? often, not always
- 'low level' programming interfaces? needed in many operating systems
- interprocess communication tools? needed in almost all operating systems
- high processor utilisation? fault tolerance builds on redundancy

Real-time OS for music?

Real-time OS for robots?


Real-time OS for cars?
程序代写代做 CS编程辅导
WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com

Real-time operating systems need to provide…

the **logical correctness** of the results as well as the correctness of the **time**: *what* and *when* are both important

all results are to be delivered **just-in-time**—not too early, not too late.

timing constraints are specified in many different ways… often as a response to **external** events (reactive systems)

predictability, not performance!

Embedded Operating Systems

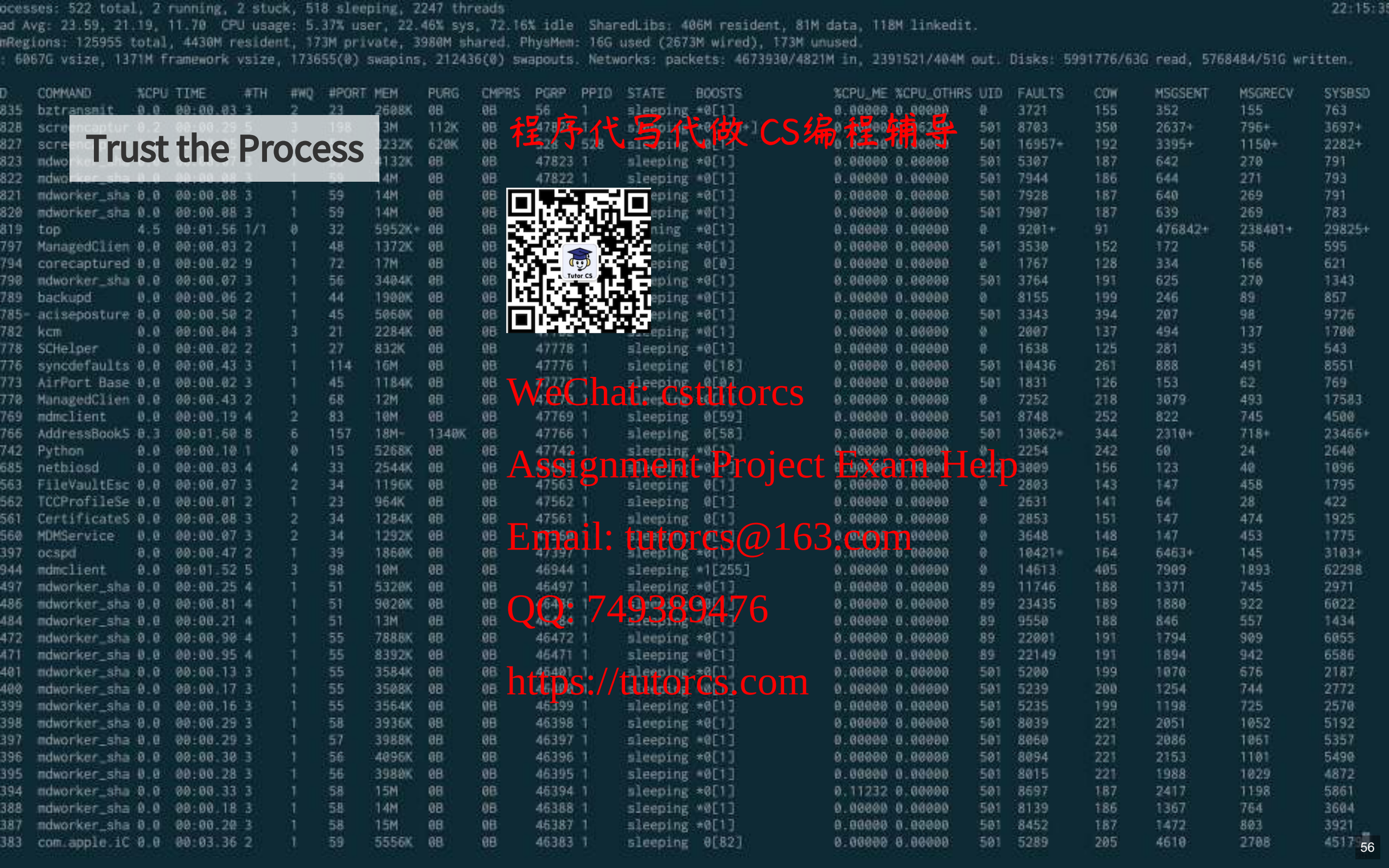
usually real-time systems, often hard real-time systems

very small footprint (often a few kBytes)

none or limited user-interaction

most processors in the world are in embedded systems

Processes & scheduling


Trust the Process
程序代写代做CS编程辅导
WeChat: cstutorcs
Assignment Project Exam Help
Email: tutorcs@163.com
QQ: 749389476
https://tutorcs.com

talk

(if you've got your laptop here) h 

rocesses are running on your machine right now?

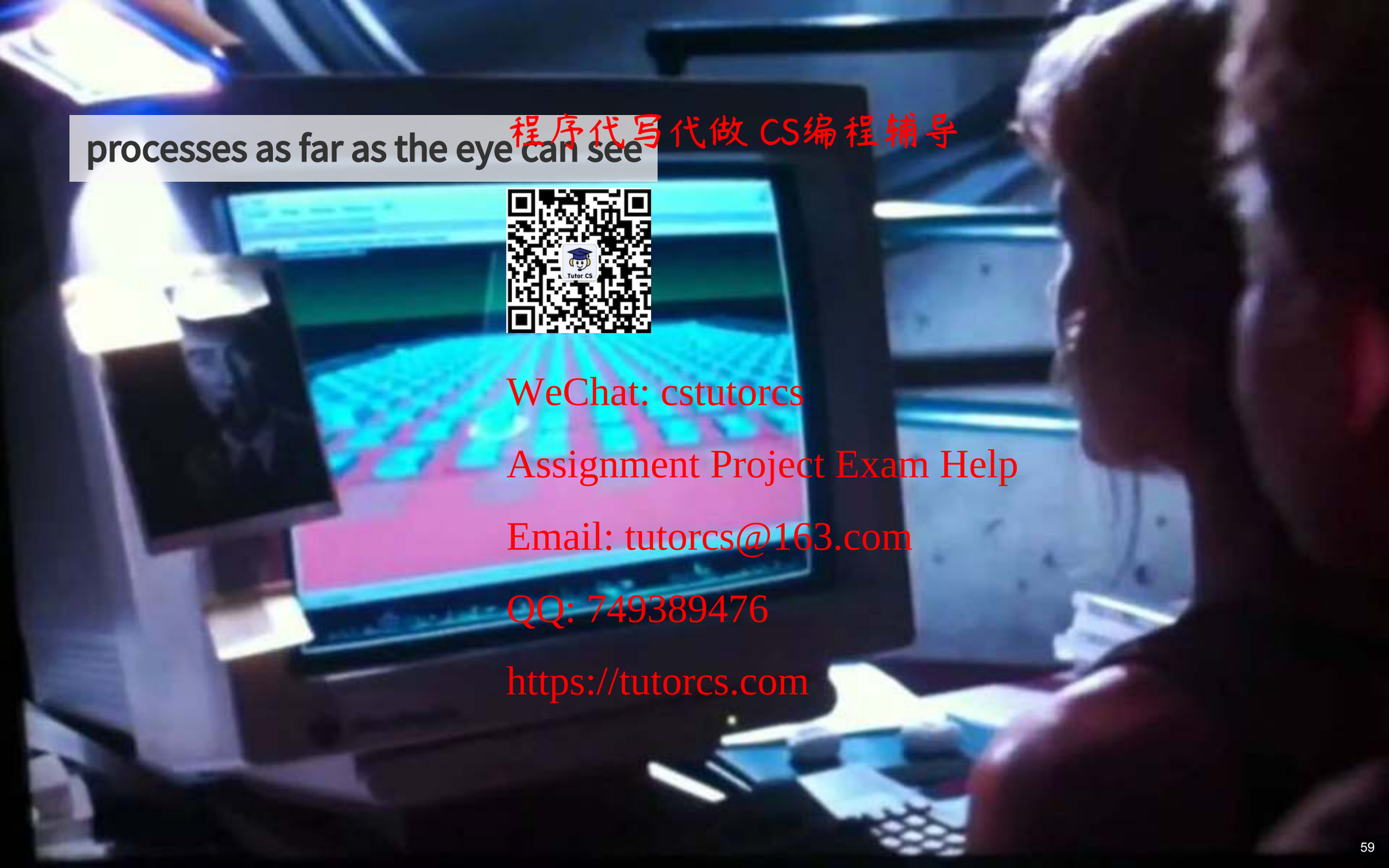
how about on your phone?

Process: definition

basically: a *running* program

includes the code (instructions) for the program, and the current state/context:

- registers/flags
- memory (stack and heap)
- permissions/privileges
- other resources (e.g. global variables, open files & network connections, address space mappings)



exact definition of process

depends on the OS

so how do we manage them?

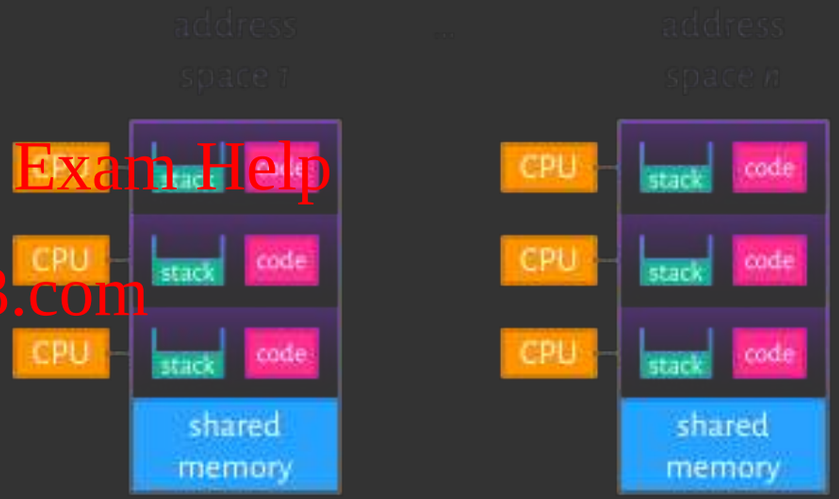
1 CPU per control-flow

specific configurations only, e.g.:

- distributed microcontrollers
- physical process control systems

1 cpu per task, connected via a bus system

- Process management (scheduling) not required
- Shared memory access need to be coordinated

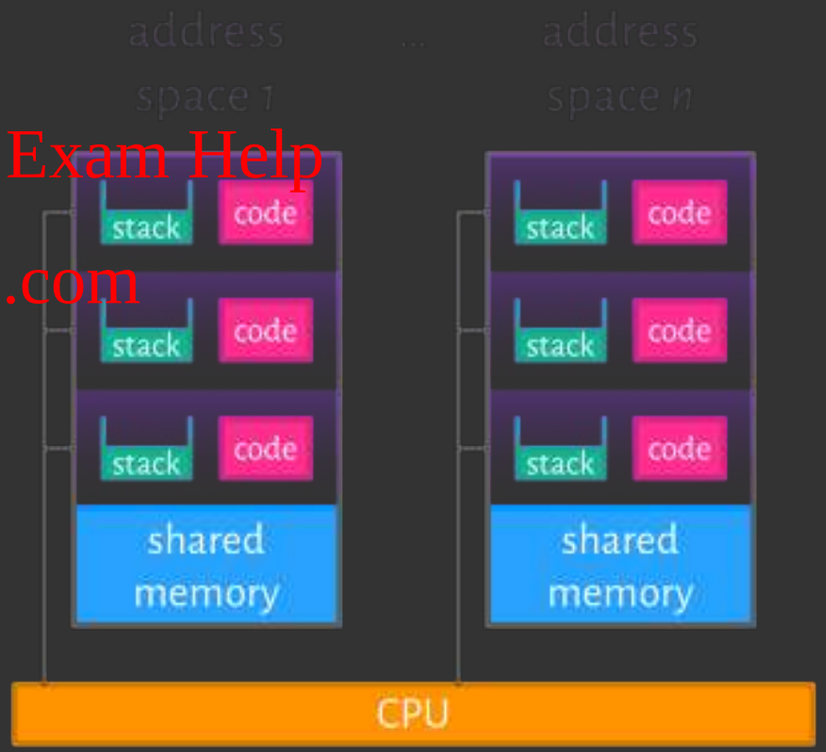


1 CPU for all control-flows

the OS may “emulate” one CPU f ntrol-flow

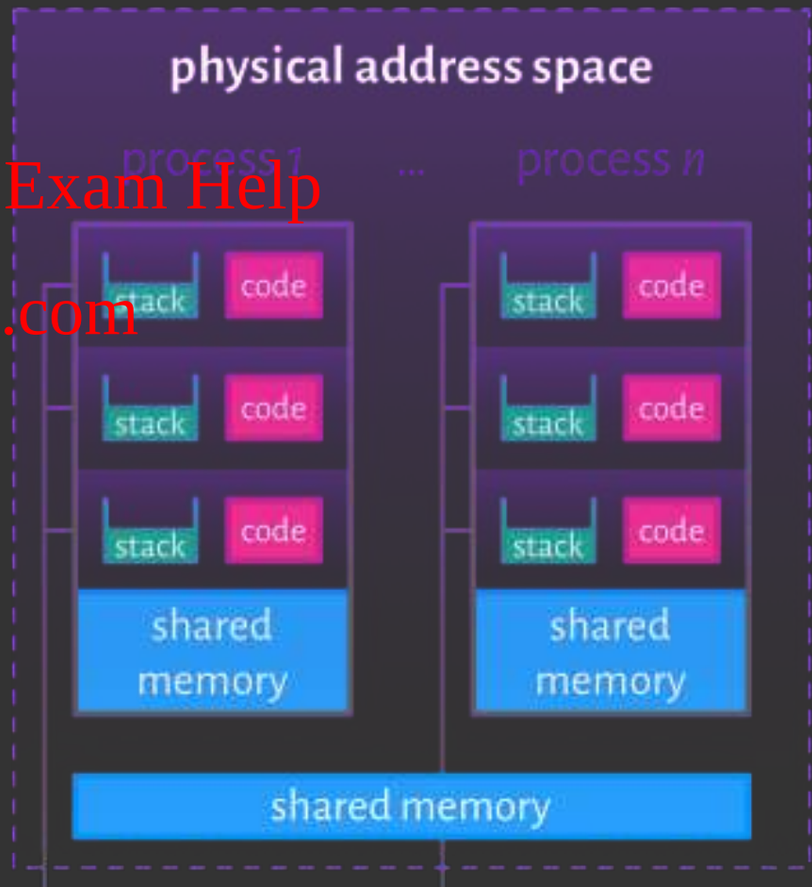
this is a **multi-tasking operating**

- support for memory protection essential
- process management (scheduling) required
- shared memory access need to be coordinated



Symmetric multiprocessing (SMP)

all CPUs share the same physica pace (and have access to the same resources) so any process can be executed on ble CPU

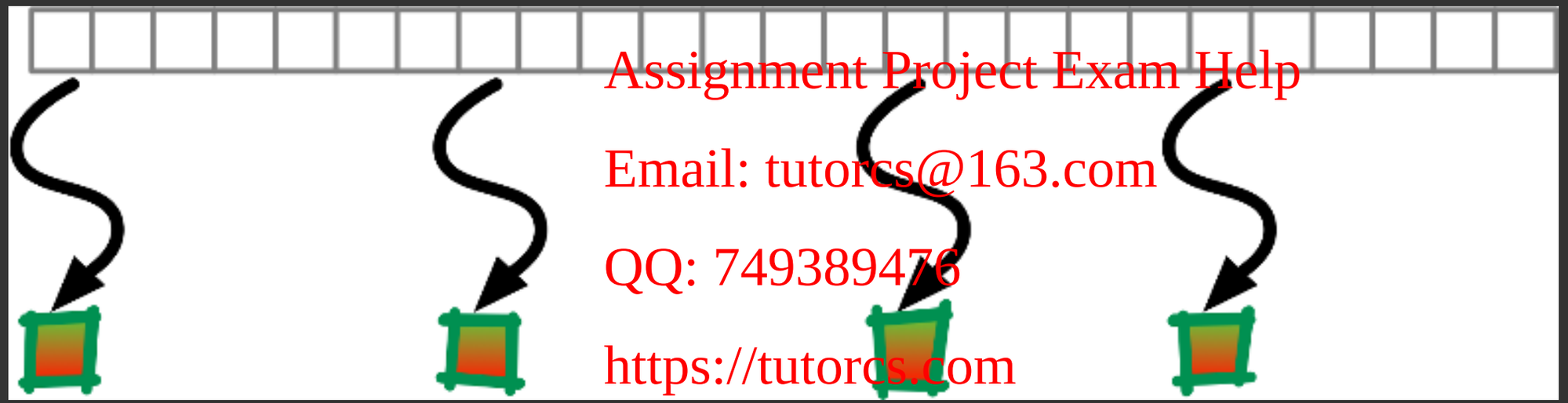


Processes vs threads

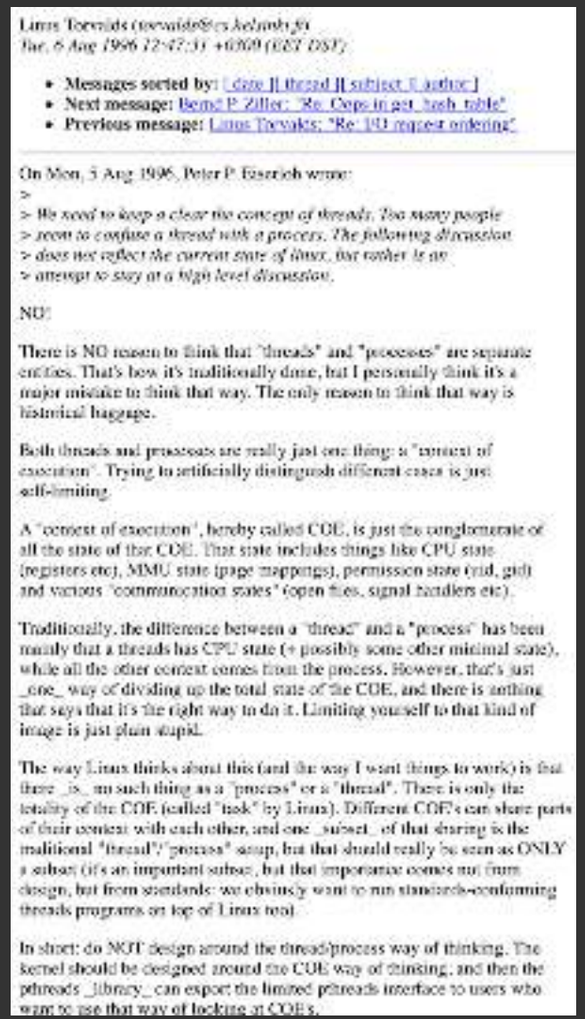
processes (as discussed earlier) own registers, stack, resources, etc.

threads have their own registers but share the other process resources

one process can create/manage many threads



Torvalds vs Threads



Linus Torvalds (torvalds@cs.helsinki.fi)
Tue, 6 Aug 1996 12:47:31 +0300 (EET DST)

- **Messages sorted by:** [date][thread][subject][author]
- **Next message:** Bernd P. Ziller: "Re: Oops in get_hash_table"
- **Previous message:** Linus Torvalds: "Re: 143 request ordering"

On Mon, 5 Aug 1996, Peter P. Eisenlohr wrote:
>
> *We need to keep a clear the concept of threads. Too many people*
> *seem to confuse a thread with a process. The following discussion*
> *does not reflect the current state of linux, but rather is an*
> *attempt to stay at a high level discussion.*

NO!

There is NO reason to think that "threads" and "processes" are separate entities. That's how it's traditionally done, but I personally think it's a major mistake to think that way. The only reason to think that way is historical baggage.

Both threads and processes are really just one thing: a "context of execution". Trying to artificially distinguish different cases is just self-limiting.

A "context of execution", hereby called COE, is just the conglomerate of all the state of that COE. That state includes things like CPU state (registers etc), MMU state (page mappings), permission state (uid, gid) and various "communication states" (open files, signal handlers etc).

Traditionally, the difference between a "thread" and a "process" has been mainly that a threads has CPU state (+ possibly some other minimal state), while all the other context comes from the process. However, that's just _one_ way of dividing up the total state of the COE, and there is nothing that says that it's the right way to do it. Limiting yourself to that kind of image is just plain stupid.

The way Linux thinks about this (and the way I want things to work) is that there _is_ no such thing as a "process" or a "thread". There is only the totality of the COE (called "task" by Linux). Different COE's can share parts of their context with each other, and one _subset_ of that sharing is the traditional "thread"/"process" setup, but that should really be seen as ONLY a subset (it's an important subset, but that importance comes not from design, but from standards: we obviously want to run standards-conforming threads programs on top of Linux too).

In short: do NOT design around the thread/process way of thinking. The kernel should be designed around the COE way of thinking, and then the pthreads _library_ can export the limited pthreads interface to users who want to use that way of looking at COE's.

"Depends on the implementation"...

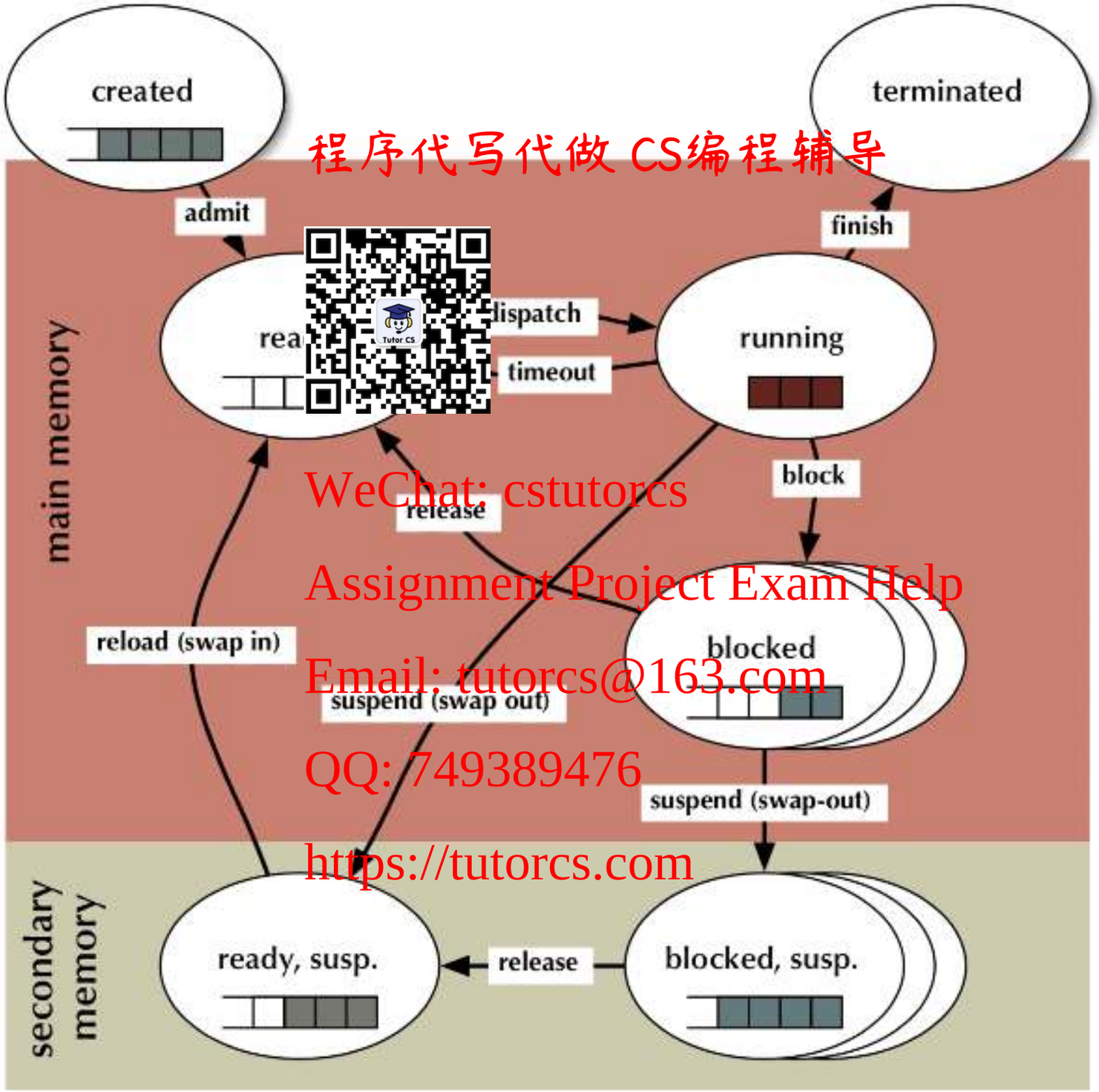
Process Control Blocks (PCBs)

- process ID
- process state: {created, ready, ex[illegible]ocked, suspended, bored …}
- scheduling attributes: priorities, deadlines
- CPU state: (e.g. registers, stack pointer)
- memory attributes/privileges: permissions, limits, shared areas
- allocated resources: open/requested devices and files, etc.

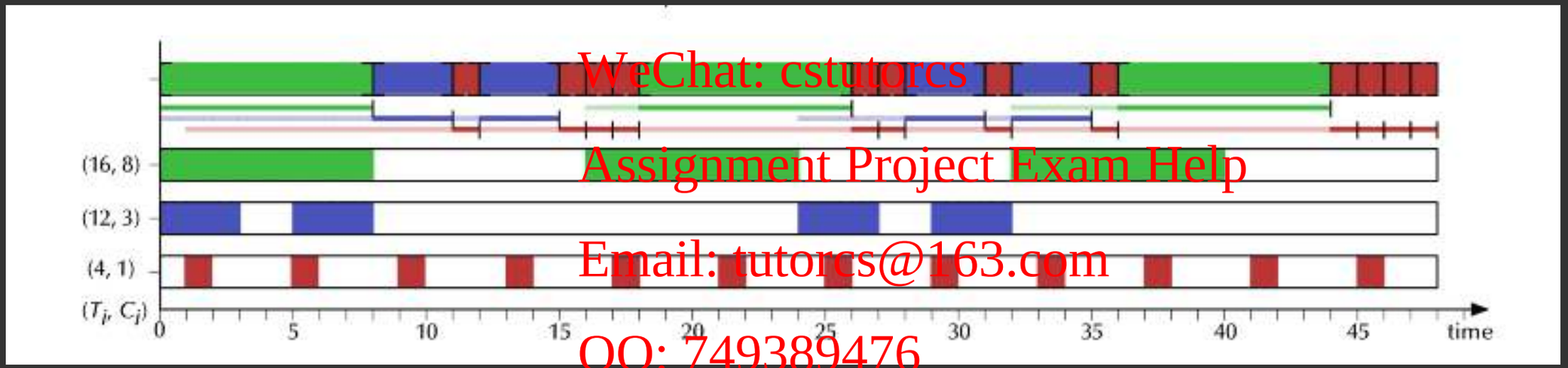
a data structure for processes

Process states

- **created**: the task is ready to ru[illegible]yet considered by any dispatcher
- **ready**: ready to run (waiting fo[illegible]U)
- **running**: holds a CPU and executes
- **blocked**: not ready to run (waiting for a resource)
- **suspended**: swapped out of main memory (e.g. waiting for main memory space)

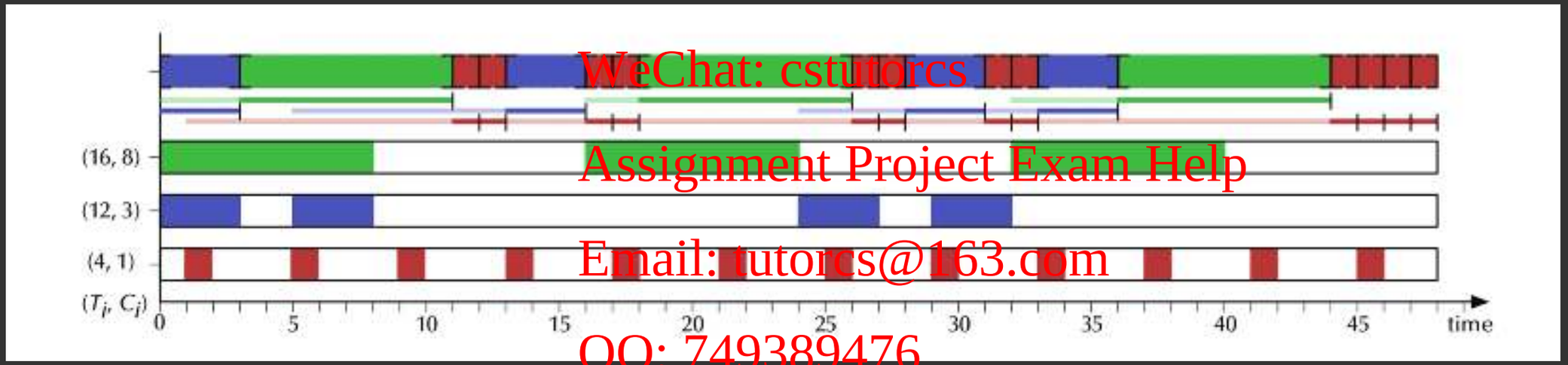


First come, first served (FCFS) scheduling



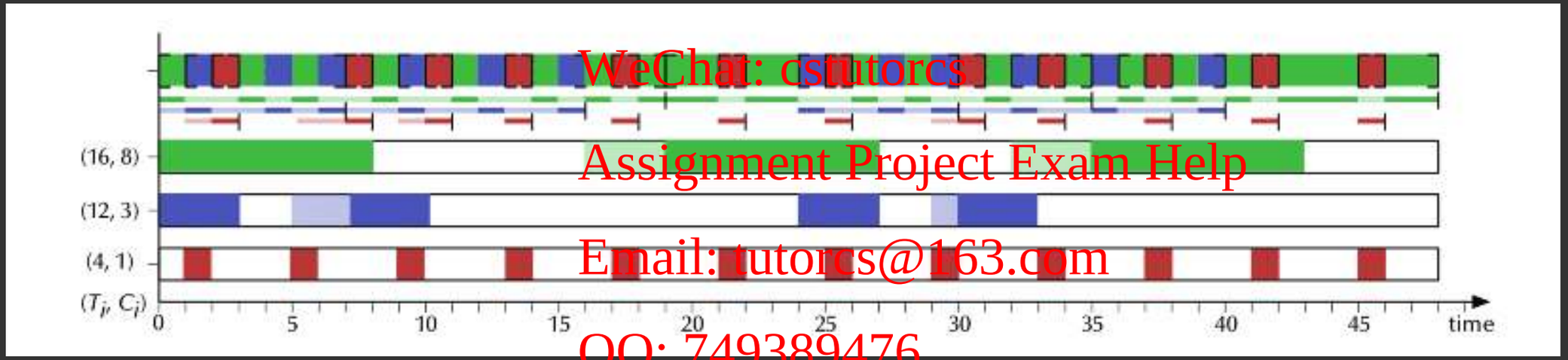
- **Waiting time**: 0..11, average: 5.9
- **Turnaround time**: 3..12, average: 8.4

FCFS (again)



- **Waiting time**: 0..11, average: 5.4 (was 5.9 before)
- **Turnaround time**: 3..12, average: 8.0 (was 8.4 before)

Round-robin (RR) scheduling



- **Waiting time**: 0..5, average: 1.2
- **Turnaround time**: 1..20, average: 5.8

optimised for swift initial responses, but “stretches out” long tasks

talk

when might you want to use FCF ng? how about RR?

go build your own in labs.

Questions

Fun With Operating Systems

- Kernel writing 101
- Linux on an 8bit AVR

- How to make an operating system (WikiHow)

Further Reading

Essentials of Computer Organis Architecture - Chapter 8.2: Operating Systems

Nick Moffitt's $7 History of Unix

LGR Tech Tales - How Digital Research Almost Ruled PCs