Concurrent and Distributed Systems Introduction

- 8 lectures on concurrency control in centralised systems
  - interaction of components in main memory
  - interactions involving main memory and persistent storage (concurrency control and crashes)
- 8 lectures on distributed systems
- Part 1A Operating Systems concepts are needed

Let’s look at the total system picture first

How do distributed systems differ fundamentally from centralised systems?

Fundamental properties of *distributed* systems

1. Concurrent execution of components on different nodes
2. Independent failure modes of nodes and connections
3. Network delay between nodes
4. No global time – each node has its own clock

Implications: to be studied in lectures 9 – 16

1 components do not all fail together and connections may also fail

2, 3 - can't know why there's no reply – node/comms. failure and/or node/comms. congestion

4 - can't use locally generated timestamps for ordering events from different nodes in a DS

1, 3 - inconsistent views of state/data when it's distributed

1 - can't wait for quiescence to resolve inconsistencies

What are the fundamental problems for a single node?

single node characteristics cf. distributed systems

1. Concurrent execution of components in a single node
2. Failure modes - all components crash together, but disc failure modes are independent
3. Network delay not relevant – but consider interactions with disc (persistent store)
4. Single clock – event ordering not a problem

single node characteristics: concurrent execution

1. Concurrent execution of components in a single node

When a program is executing in main memory all components fail together on a crash, e.g. power failure.

Some old systems, e.g. original UNIX, assumed ***uniprocessor*** operation. Concurrent execution of components is achieved on uniprocessors by interrupt-driven scheduling of components. Preemptive scheduling creates most potential flexibility and most difficulties.

Multiprocessors are now the norm.
Multi-core instruction sets are being examined in detail and found problematic (sequential consistency).

single node characteristics: failure modes

2 Failure modes - all components crash together, but disc failure modes are independent

3 Network delay not relevant – but consider interactions with disc (persistent store)

In lectures 5-8 we consider programs that operate on persistent data on disc. We define transactions: composite operations in the presence of concurrent execution and crashes.

single node characteristics: time and event ordering

4. Single clock – event ordering “not a problem”?

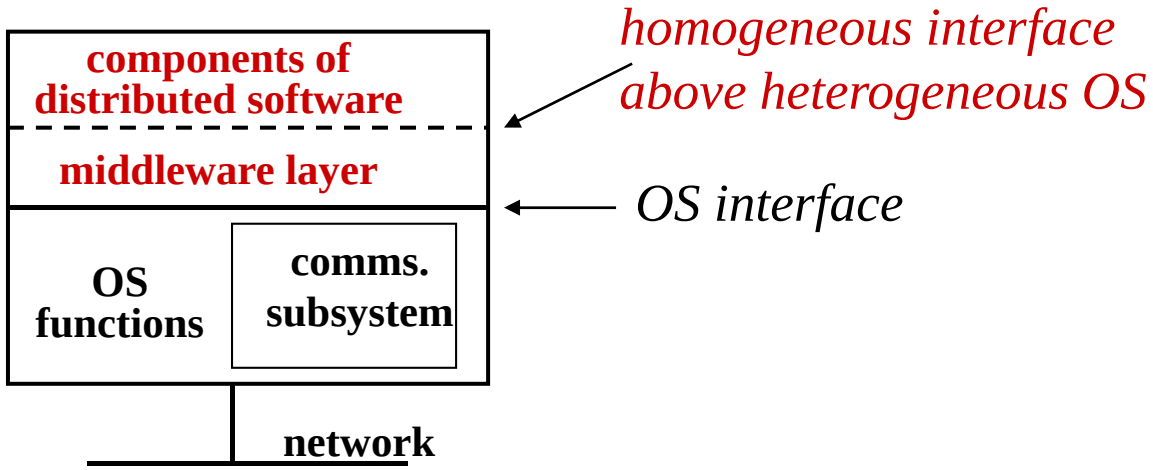
In distributed systems we can assume that the timestamps generated by a given node and appended to the messages it sends are ordered sequentially.

We could previously assume sequential ordering of instructions on a single computer, including multiprocessors. But multi-core computers now reorder instructions in complex ways. Sequential consistency is proving problematic.

We shall concentrate on classical concurrency control concepts. Multi-core will be covered in depth in later years.

single node as DS component (for lectures 9-16)

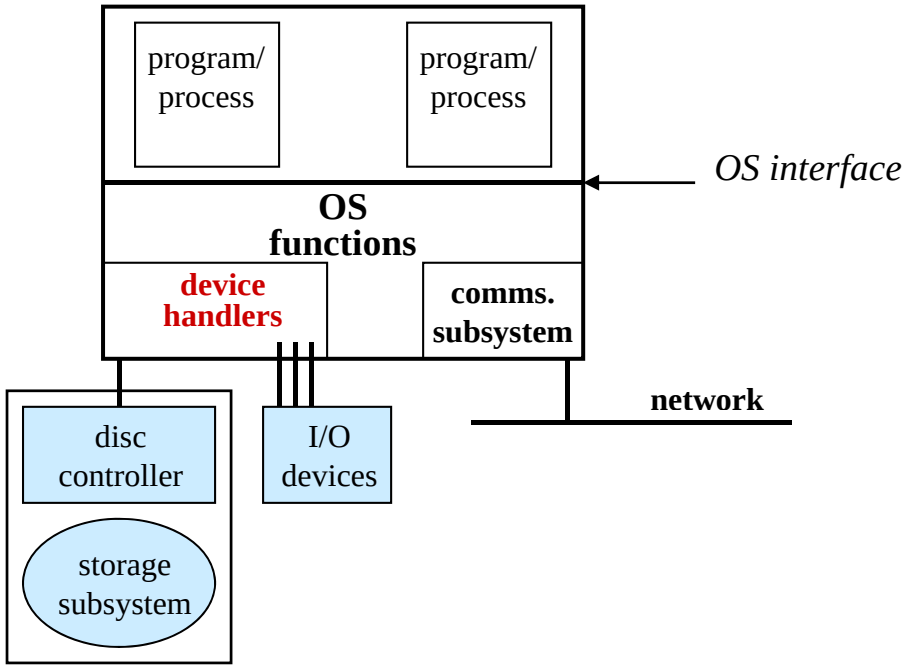
Support for distributed software components is by a software layer (middleware) above potentially heterogeneous OS



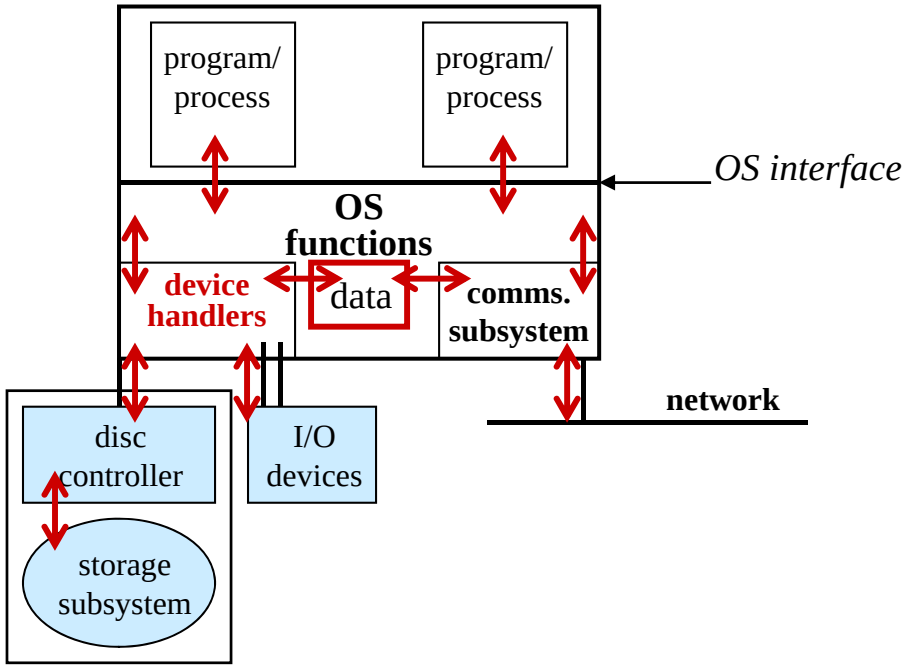
We shall first consider a single node's software structure and dynamic execution

single node: software components

- Software structure
- Support for persistent storage
- Dynamic concurrent execution – see next slide

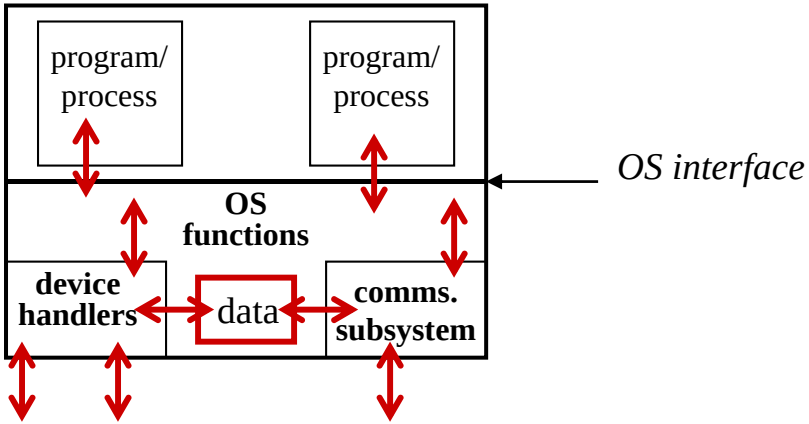


single node – concurrent execution



- note shared data areas in OS
- also, programs may share data
- also, “threads” in a concurrent program may share data

concurrent execution and process scheduling

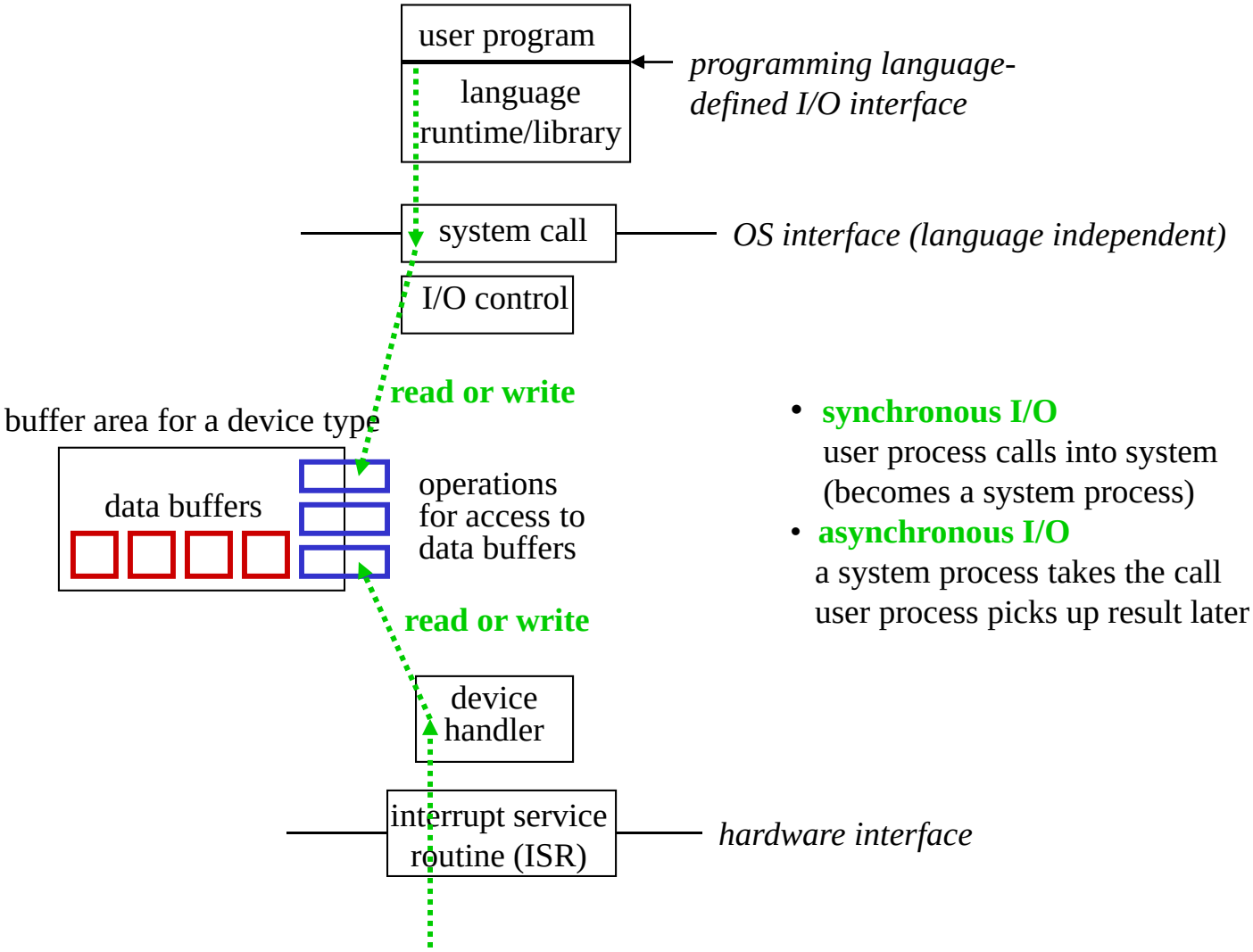


RECALL from part 1A OS:

- preemptive scheduling
- interrupt-driven execution – devices, timers, system calls
- OS processes have static priority, page fault > disc > …. > system call handling
- OS process priority higher than application process priority

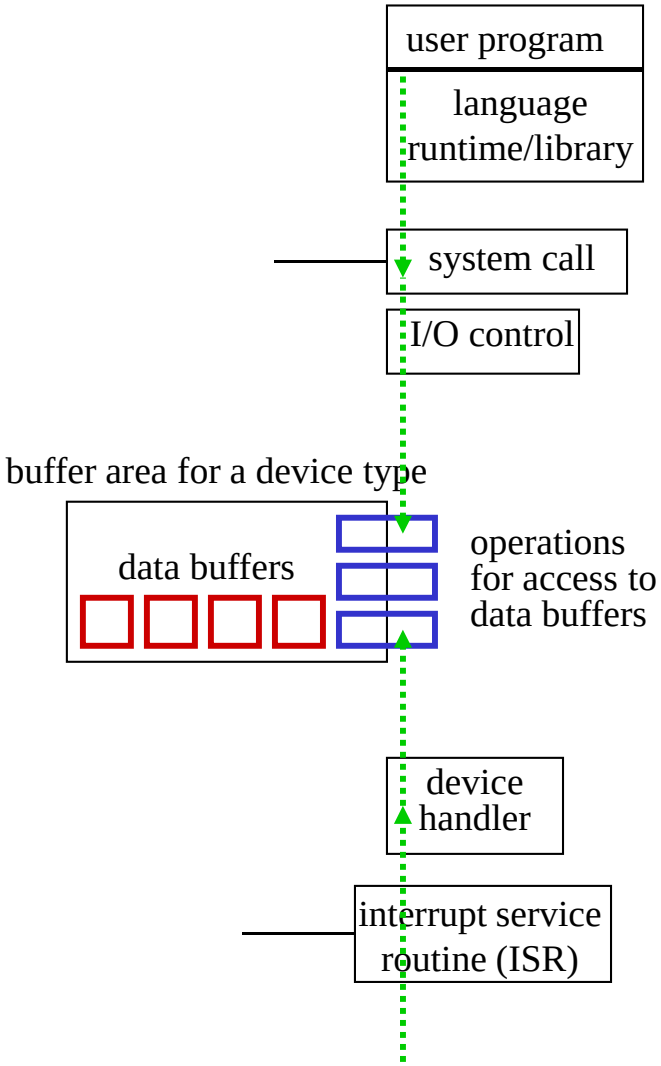
PROBLEM: process preemption while reading/writing shared data

Some OS components and processes - 1



- **synchronous I/O**
  user process calls into system (becomes a system process)
- **asynchronous I/O**
  a system process takes the call
  user process picks up result later

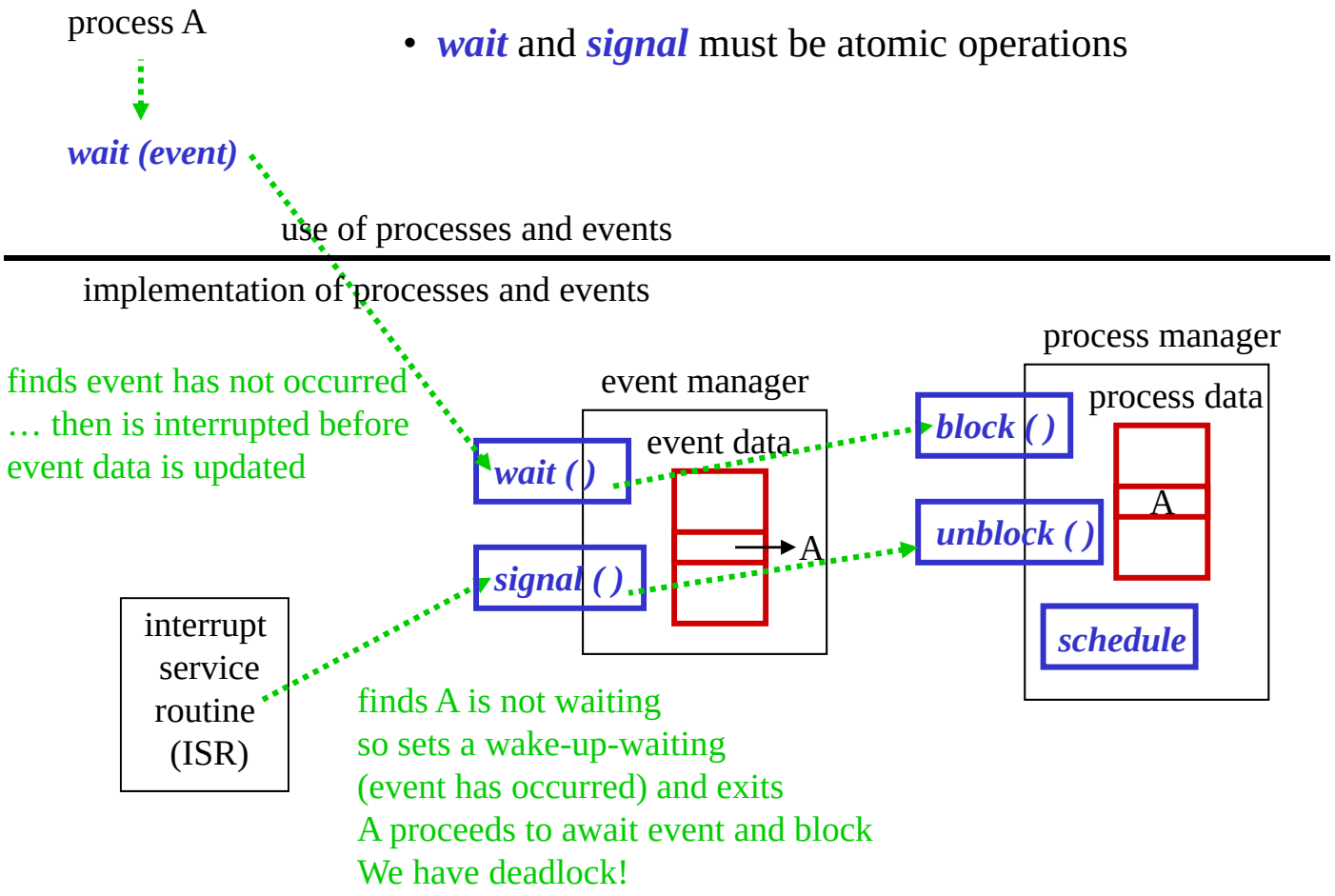
Some OS components and processes - 2



- buffers must be accessed under mutual exclusion
- condition synchronisation is also needed:
  process gets mutex access to buffer (for how, see later)
  process finds buffer full on write or empty on read
  process must BLOCK until space or data available
  process ***must not block while holding mutex*** access
  (else we have deadlock!)
- note priority of device handlers > priority of user calls
- interrupts are independent of process execution
- if mutex access to buffer is not enforced, top-down access could be preempted by interrupt-driven, bottom-up access, resulting in deadlock or incorrect data.
- same issue in hardware-software synchronisation
  – see next slide:

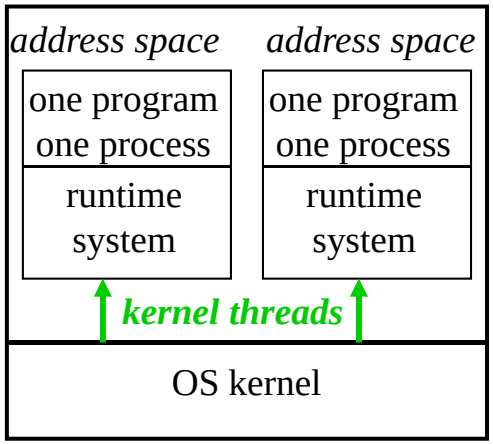
Some OS components and processes - 3

- interrupts are independent of process execution
- care is needed over priority of ISR and interrupt-driven code
- ***wait*** and ***signal*** must be atomic operations

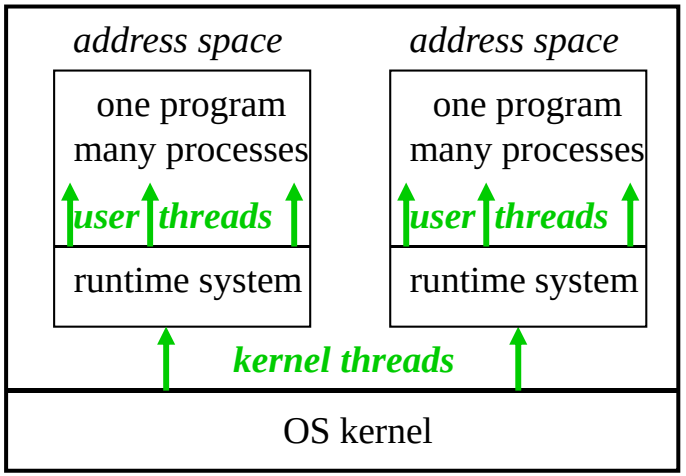


Processes and threads

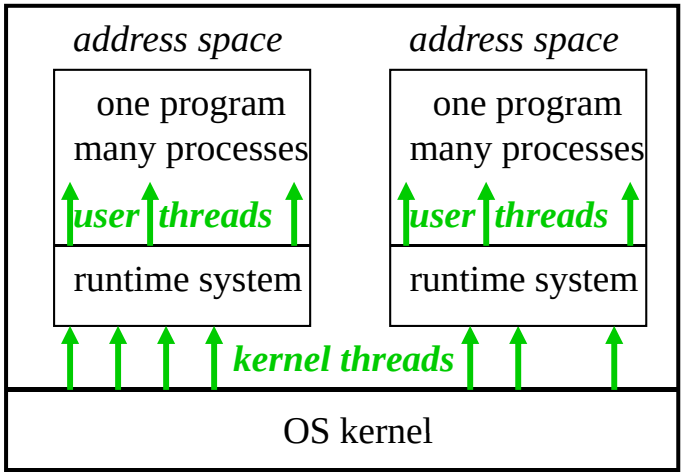
a) Sequential programming languages



b) Concurrent programming language, no OS support (user threads only)

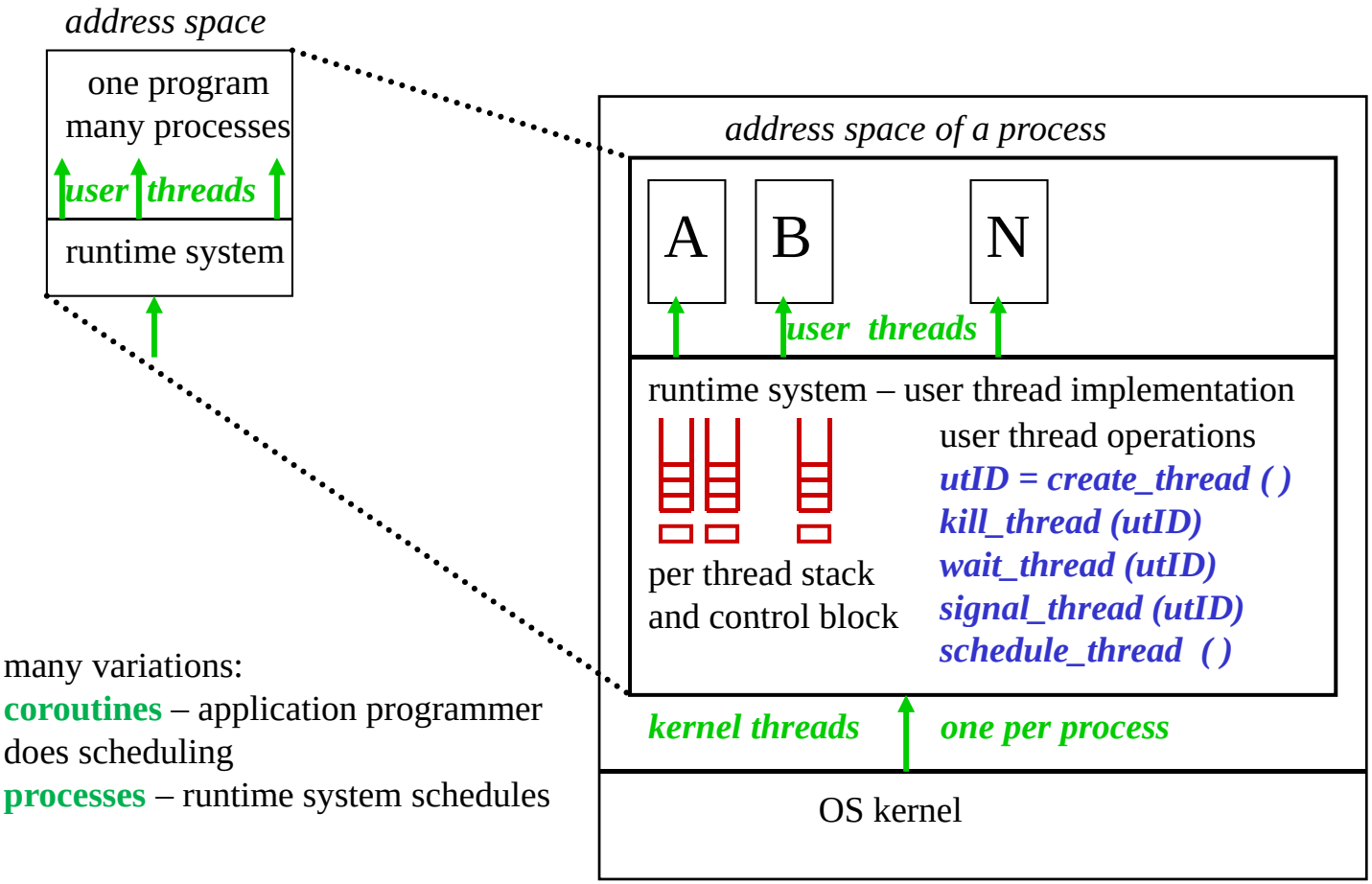


c) Concurrent programming language, OS kernel threads for user threads



Runtime system - user threads

b) Concurrent programming language, no OS support (user threads only)



see later - may not
wait and *signal*
a thread, but a mutex

many variations:
coroutines – application programmer does scheduling
processes – runtime system schedules

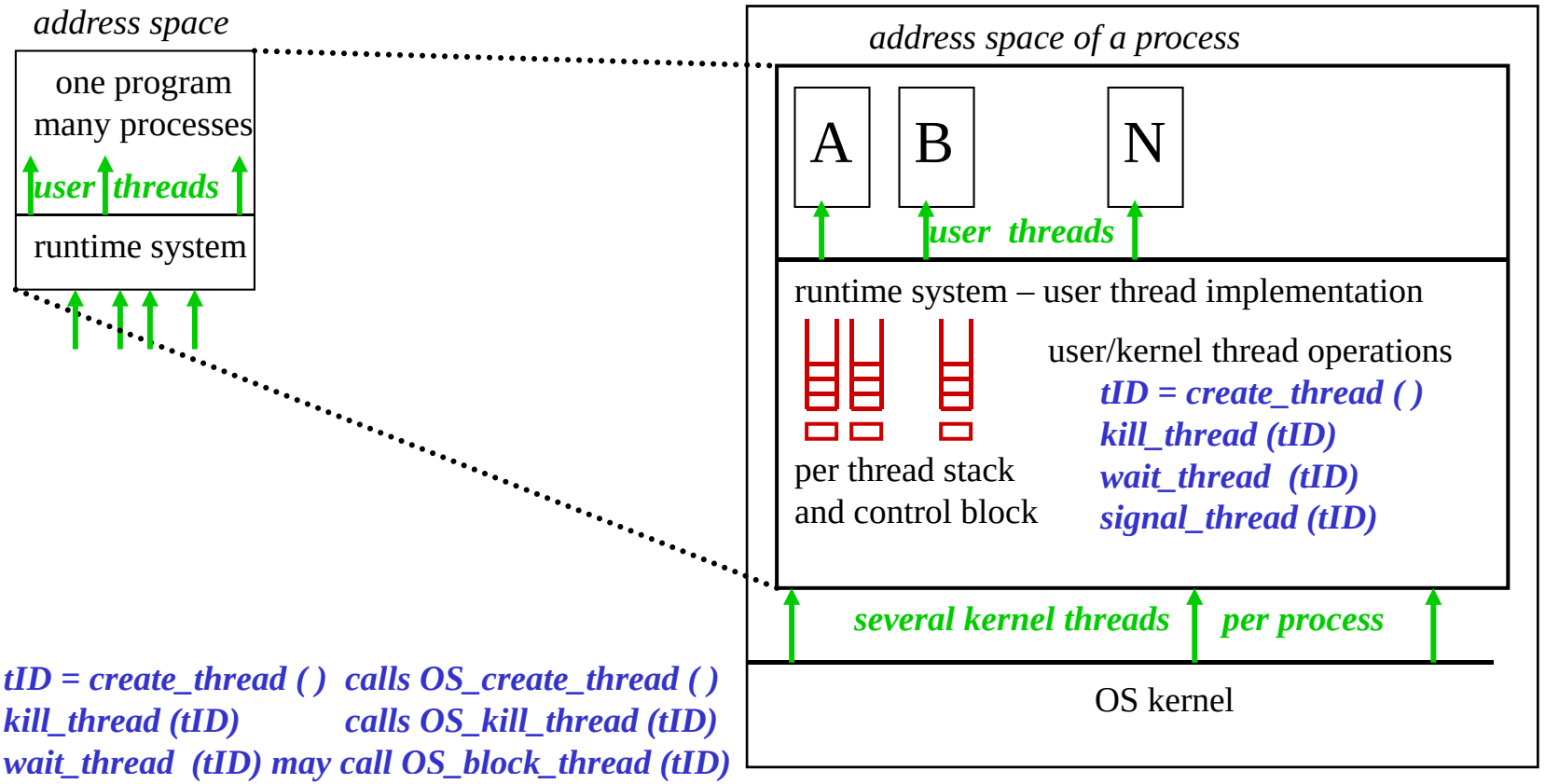
user threads only

1. the application can’t respond to OS events by switching user-threads
2. can’t use for real-time applications – delay is unbounded
3. the whole process is blocked if any thread makes a system call and blocks
4. applications can’t exploit a multiprocessor. The OS knows, and can schedule, only one kernel thread
5. BUT handling shared data in the concurrent program is simple. There is no user-thread preemption i.e. threads are ONLY switched on calls to the runtime system.

   After an interrupt, control returns to the point at which the interrupt occurred.

Runtime system - kernel threads

c) Concurrent programming language, OS can have several kernel threads per process



tID = create_thread ( ) calls OS_create_thread ( )
kill_thread (tID) calls OS_kill_thread (tID)
wait_thread (tID) may call OS_block_thread (tID)
signal_thread (tID) may call OS_unblock_thread (tID)

The OS schedules threads

kernel threads and user threads

1. thread scheduling is via the OS scheduling algorithm
2. applications can respond to OS events by switching threads, but only if OS scheduling is preemptive and priority-based. Real-time response is therefore OS-dependent
3. user threads can make blocking system calls without blocking the whole process – other threads can run
4. applications can exploit a multiprocessor
5. managing shared writeable data becomes complex
6. there are different thread packages – needn't create exactly one kernel thread per user thread.

   modern applications may create large numbers of threads

   kernel may allow a maximum number of threads per process