Universität Bamberg

Operating Systems Engineering

Lecture 10: Synchronization

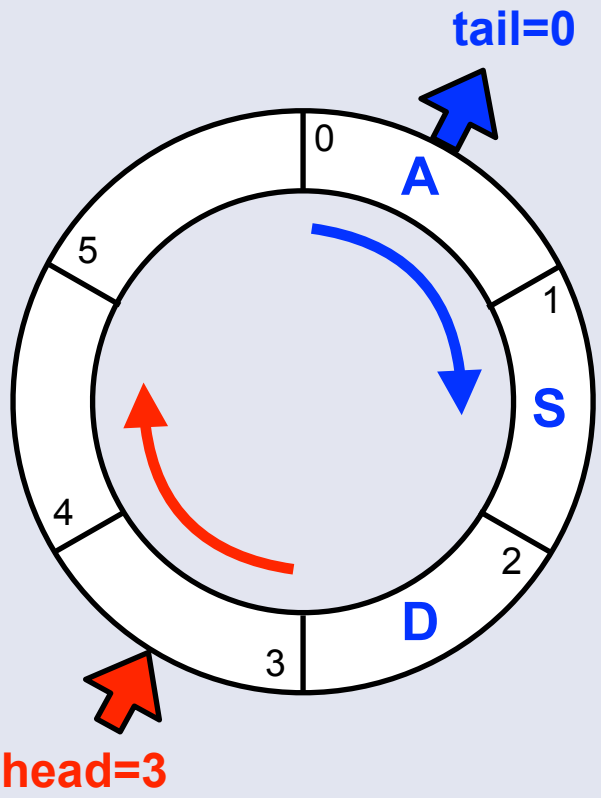
Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insb. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

Ring buffer implementation


```
#define BUFFER_SIZE 6
char ringbuffer[BUFFER_SIZE];
int  head, tail, full_flag;
```

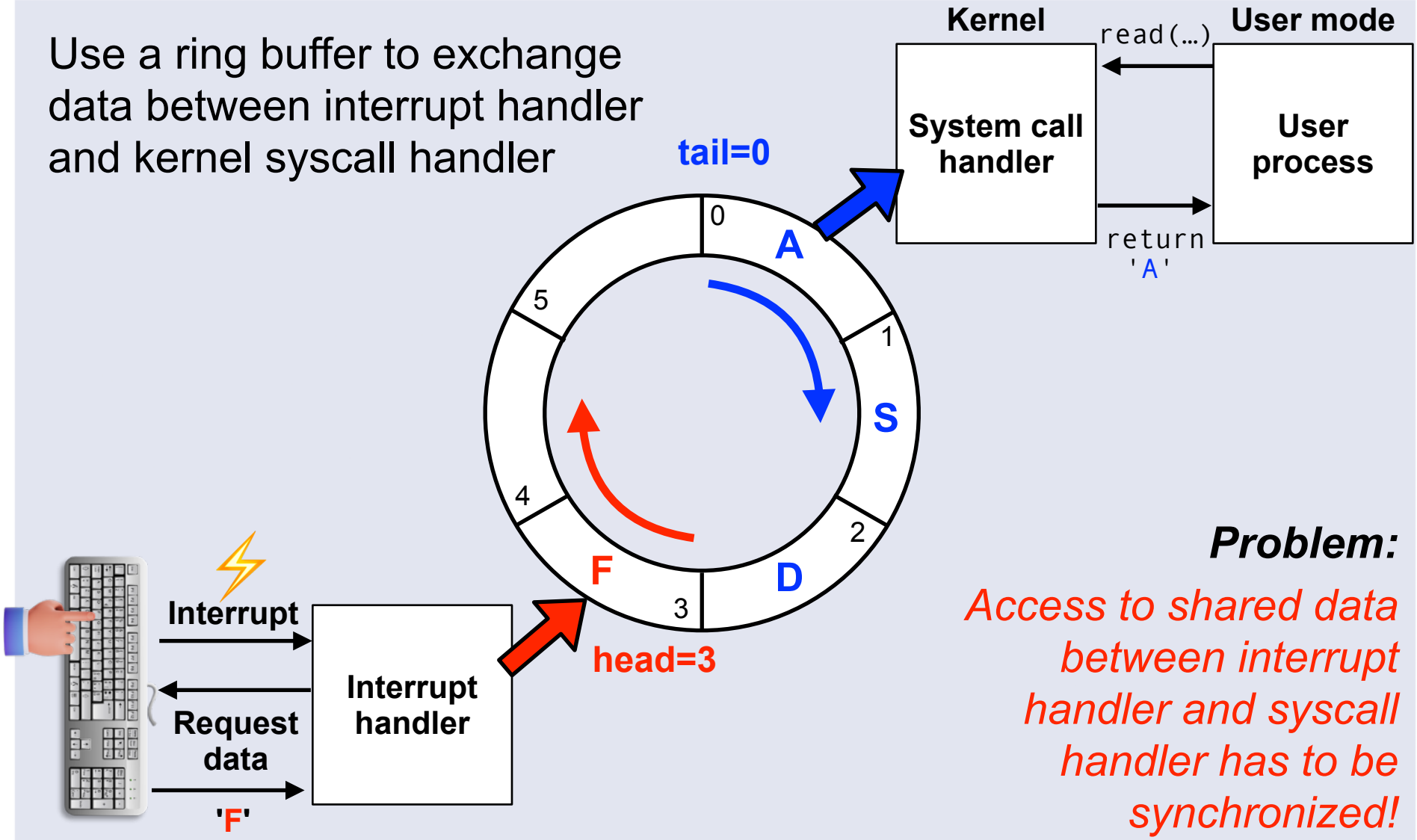
```
int rb_write(char c) {
  if (buffer_is_full()) {
    return -1;
  } else {
    ringbuffer[head] = c;
    head = (head+1) % BUFFER_SIZE;
    if (head == tail)
      full_flag = 1;
  }
}

int rb_read(char *c) {
  if (buffer_is_empty()) {
    return -1;
  } else {
    *c = ringbuffer[tail];
    tail = (tail+1) % BUFFER_SIZE;
    full_flag = 0;
  }
}
```



Kernel↔interrupt handler communication

Use a ring buffer to exchange data between interrupt handler and kernel syscall handler



Problem:

Access to shared data between interrupt handler and syscall handler has to be synchronized!

The problem: ***race conditions***

- A ***race condition*** is a situation in which multiple processes access shared data concurrently and at least one of the processes manipulates the data
  - When a race condition occurs, the resulting value of the shared data is dependent on the order of access by the processes
  - The result is therefore not predictable and can also be incorrect in case of overlapping accesses!

- To avoid race conditions, concurrent processes need to be ***synchronized***

Synchronization

- The coordination of the cooperation of processes is called ***synchronization***
  - Synchronization creates an order for the activities of concurrent processes
  - Thus, on a global level, synchronization enables the ***sequentiality*** of activities

Source: Herrtwich/Hommel (1989), Kooperation und Konkurrenz, p. 26

Critical section

- In the case of a race condition, N processes compete for the access to shared data
- The code fragments accessing these critical data are called ***critical sections***

- **Problem**
  - We need to ensure that ***only a single process*** can be in the critical section at the same time

Solution: Lock variables

A lock variable is an abstract data type with two operations: `acquire` **and** `release`

```
Lock lock;

/* Example code for enqueue */
void enqueue (struct list *list, struct element *item) {
  item->next  = NULL;

  acquire(&lock);

  *list->tail = item;
  list->tail  = &item->next;

  release(&lock);
}
```

- blocks a process until the specified lock is open
- then locks the lock itself "from the inside"

- opens the specified lock without blocking the calling process

Implementations like these are called **lock(ing) algorithms**

Implementing locks: incorrect

This naïve lock implementation does not work!

```
/* Lock variable (initial value is 0) */
typedef unsigned char Lock;

/* enter the critical section */
void acquire (Lock *lock) {
  while (*lock); /* note: empty loop body! */
  *lock = 1;
}

/* leave the critical section */
void release (Lock *lock) {
  *lock = 0;
}
```

Implementing locks: incorrect


```
/* Lock variable */
typedef unsigned char Lock;

/* enter the critical section */
void acquire (Lock *lock) {
  while (*lock);
  *lock = 1;
}

/* leave the critical section */
void release (Lock *lock) {
  *lock = 0;
}
```

`acquire` must protect a critical section – but it is critical itself!

- the critical moment is the point in time after leaving the waiting loop and before setting the lock variable!
- If the current process is preempted between the two lines of code, another process sees the critical section as free and would also enter!

If this happens, (at least) two processes could enter the critical section simultaneously that should be protected by `acquire`!

Locks with atomic operations

- Many CPUs support indivisible (**atomic**) read/modify/write cycles that can be used to implement lock algorithms
- We have to use special machine instructions for atomic operations, e.g.:
  - Motorola 68K: **TAS** (test and set)
    - sets bit 7 of the destination operand and returns its previous state in the CPU's condition code bits
  - Intel x86: **XCHG** (exchange)
    - Exchanges the content of a register with that of a memory location (i.e. a variable in memory)
  - ARM: **LDREX/STREX** (load/store exclusive)
    - STREX checks if any write to the address has occurred since the last LDREX
    - More recent ARM CPUs (v8/v8.1) provide additional (better performing) atomic instructions

```
acquire  TAS lock
         BNE acquire
```

```
         mov  ax, 1
acquire  xchg lock
         cmp  ax, 0
         jne  acquire
```

```
        MOV   r1, #0xFF
acquire LDREX r0, [LockAddr]
        CMP   r0, #0
        STREXEQ r0, r1, [LockAddr]
        CMPEQ r0, #0
        BNE   acquire
```

Atomic operations on RISC-V [2]

- **Load Reserve & Store Conditional**
  - like ARM's LDREX/STREX or MIPS' LL/SC
  - `LR.W` loads a word from the address in `rs1`, places the sign-extended value in `rd`, and registers a reservation set–a set of bytes that subsumes the bytes in the addressed word
  - `SC.W` conditionally writes a word in `rs2` to the address in `rs1`: the `SC.W` succeeds only if the reservation is still valid and the reservation set contains the bytes being written
- **Atomic Memory Operations**
  - `Amoswap`: Atomically, let `t` be the value of the memory word at address `x[rs1]`, then set that memory word to `x[rs2]`. Set `x[rd]` to (the sign extension of) `t` – like x86 XCHG
  - other atomic instructions exist: `amoadd`, `amo{and,or,xor}`, `amo{min,max}`

Atomic operations on RISC-V examples

Test-and-set (address of lock variable in `a0`):

```
tas:
  li           t0, 1          # locked state is 1
  amoswap.w.aq t1, t0, (a0)
  bne          t1, x0, tas    # if lock is not free, retry
  ...                         # critical section
  amoswap.w.rl x0, x0, a0     # release lock
  ret
```

Compare-and-swap (CAS, address of lock variable in `a0`):

```
# a1: expected value, a2: desired value, result in a0 (!=0 -> fail)
cas:
  lr.w  t0, (a0)       # load original value
  bne   t0, a1, fail   # doesn't match -> fail
  sc.w  t0, a2, (a0)   # try to update value
  bnez  t0, cas        # retry if store-conditional failed
  li    a0, 0          # set return to success
  jr    ra             # return
fail:
  li    a0, 1          # set return to failure
  jr    ra
```

Compiler intrinsics for atomic operations

- Modern compilers have ***intrinsic functions*** for atomic ops:

```
__sync_lock_test_and_set(&lk->locked, value);
```

…is compiled for a RISC-V target to:

```
a5 = 1
s1 = &lk->locked
amoswap.w.aq a5, a5, (s1)
```

This is also used in the xv6 RISC-V implementation of locks [3,4]

Suppressing interrupts

What is the reason for a process switch inside of a critical section?

- The operating system interferes (e.g. due to a process using too much CPU time) and moves another process to the RUNNING state
- This can only happen if the OS regains control
  ➛ a timer or device interrupt occurs

Idea: **disable interrupts**
to ensure a process can stay in the critical section!

```
/* enter critical section */
void acquire (Lock *lock) {
  asm ("cli");
}

/* leave critical section */
void release (Lock *lock) {
  asm ("sti");
}
```

`cli` and `sti` are used in x86 processors to disable and enable interrupts

Suppressing interrupts on RISC-V

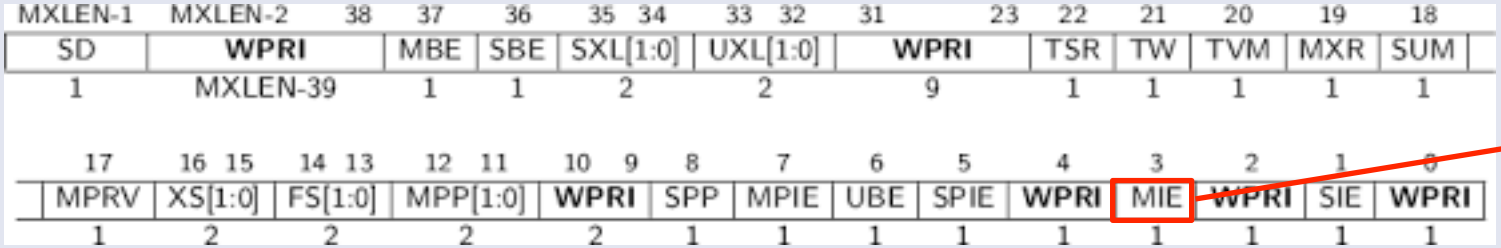
On RISC-V, there is ***no instruction*** for this, but you can use the `mstatus` CSR by setting the `mie` bit to 0 or 1, respectively

However, reading a CSR using `csrr`, clearing the `mie` bit and writing it back using `csrw` would be a critical section again!

Solution: use the `csrc/csrs` or `csrci/csrsi` (atomic clear/set bits in CSR) instruction to atomically clear the `mie` bit [5]

```
/* enter critical section */
void acquire (Lock *lock) {
  asm ("csrci mstatus, 0x8");
}
```

```
/* leave critical section */
void release (Lock *lock) {
  asm ("csrsi mstatus, 0x8");
}
```



`mie` is bit 3, so use $2^3 = 8$

`mstatus` CSR

Alternative: ***passive waiting***

- Idea: processes release the CPU while they wait for events
  - in the case of synchronization, a process “blocks itself” waiting for an event
    - the process is entered into a waiting queue
  - when the event occurs, ***one of the processes*** waiting for it is unblocked (there can be more than one waiting)
- The waiting phase of a process is realized as a blocking phase (“I/O burst”)
  - the process schedule is updated
  - another process in state READY will be moved to state RUNNING (dispatching)
  - *what happens if no process is in READY at that moment?*
- with the start of the blocking phase of a process, its CPU burst ends

Semaphores [1]

- A semaphore is defined as “a non-negative integer number” with **two atomic operations**:

P (from Dutch “prolaag” = “decrement”; also down or ***wait***)

- if the semaphore has the value 0, the process calling P is blocked
- otherwise, the semaphore value is **decremented**

V (from Dutch “verhoog” = “increment”; also up or ***signal***)

- a process waiting for the semaphore (due to a previous call to P) is unblocked
- otherwise, the semaphore is **incremented** by 1

- Semaphores are an **operating system abstraction** to exchange synchronization signals between concurrent processes

Using semaphores

"Mutual exclusion": a semaphore initialized to 1 can function as lock variable

```
Semaphore lock; /* = 1: use semaphore as lock variable */

/* Example code: enqueue */
void enqueue (struct list *list, struct element *item) {
  item->next  = NULL;

  wait (&lock);

  *list->tail = item;
  list->tail  = &item->next;

  signal (&lock);
}
```

- **the first process entering the critical section decrements the counter to 0**
- **all others block**

- **when leaving the critical section, either a blocked process is woken up *or* the counter is incremented back to 1**

…and this is not the only application of semaphores…

Semaphores: discussion

- Semaphore extensions and variants
  - binary semaphore or mutex
  - non blocking **wait()**
  - timeout
  - arrays of counters
- Sources of errors
  - risk of "***deadlocks***" → next lecture
  - difficult to implement more complex synchronization patterns
  - cooperating processes depend on each other
    - all of them must precisely follow the protocols
  - use of semaphores is not enforced
- **Implementing semaphores**…
  - …your next task :)

Conclusion

- Uncontrolled concurrent data access can lead to errors
  - **synchronisation methods** provide coordination
  - One needs to ensure that the selection strategy after signalling is not in contradiction to the scheduling approach used
- Ad hoc approach: **active waiting**
  - **Caution! Waste** of compute time
  - But: a short active wait is better than blocking, especially in multi processor systems
- Operating system-supported approach: **semaphores**
  - **Flexible** (enables many different synchronization patterns), but error-prone (→ next lecture)

References

1. Allen Downey, *The Little Book of Semaphores,* Green Tea Press 2016, https://greenteapress.com/wp/semaphores/
2. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA* Document Version 20191213 – Chapter 8:“A” Standard Extension for Atomic Instructions, Version 2.1
3. Russ Cox, Frans Kaashoek and Robert Morris, *xv6: a simple, Unix-like teaching operating system* (31.8.2020) Sections 5.3 "Concurrency in drivers" and 6 "Locking" https://pdos.csail.mit.edu/6.S081/2020/xv6/book-riscv-rev1.pdf
4. Sources for spinlock and sleeplock in xv6: https://github.com/mit-pdos/xv6-riscv/blob/riscv/kernel/sleeplock.c https://github.com/mit-pdos/xv6-riscv/blob/riscv/kernel/spinlock.c
5. Palmer Dabbelt, Michael Clark and Alex Bradbury, *RISC-V Assembly Programmer's Manual*, section "Pseudoinstructions for accessing control and status registers" https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md