Operating Systems

Timo Hönig
Bochum Operating Systems and System Software (BOSS)
Ruhr University Bochum (RUB)

IV. Synchronisation
May 3, 2023 (Summer Term 2023)

www.informatik.rub.de
Chair of Operating Systems and System Software

Agenda

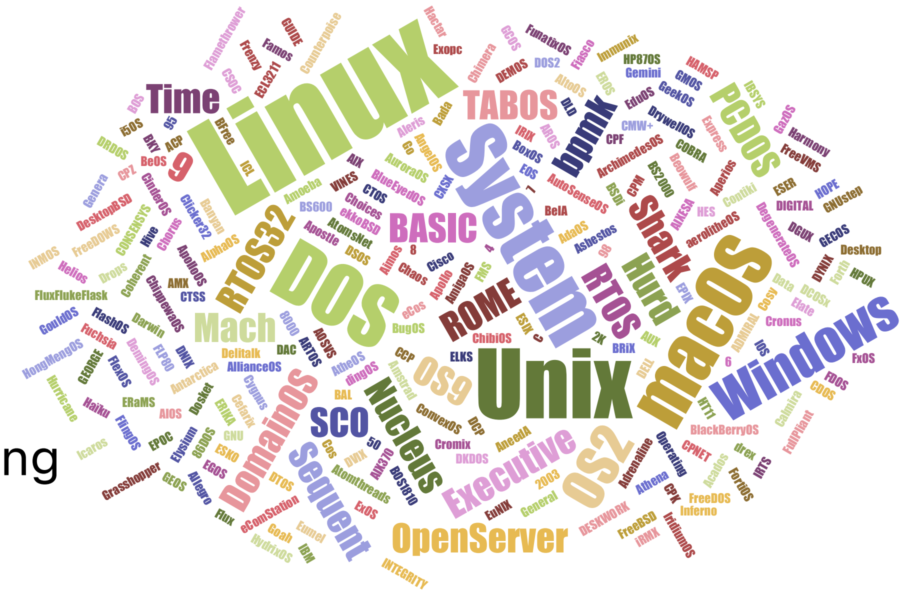
- Recap
- Organizational Matters
- Problem Scenario and Concepts
  - Critical Sections, Race Conditions
- Synchronisation
  - Ad-hoc Solutions, Busy Waiting
  - Hardware Support, Atomic Operations
  - Operating System Support, Passive Waiting
- Summary and Outlook

Literature References

Silberschatz, Chapter 6

Tanenbaum, Chapters 2.3, 2.5

⏮ Recap

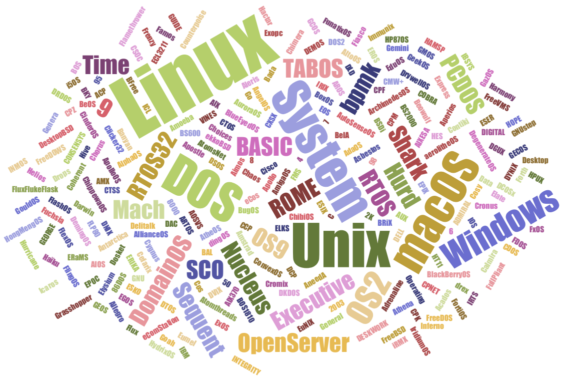


- processes: **key abstraction** for **activities** in operating systems
  - conceptually independent **sequential control flows** (alternating CPU and I/O bursts)
  - **multiplexing** of the CPU
- **process models**
  - discussion: the **weight** of processes
  - processes vs. threads vs. user-level threads
- processes in practice: **Unix process model**
  - UNIX systems provide **system calls** to create, manage, and link processes

Agenda

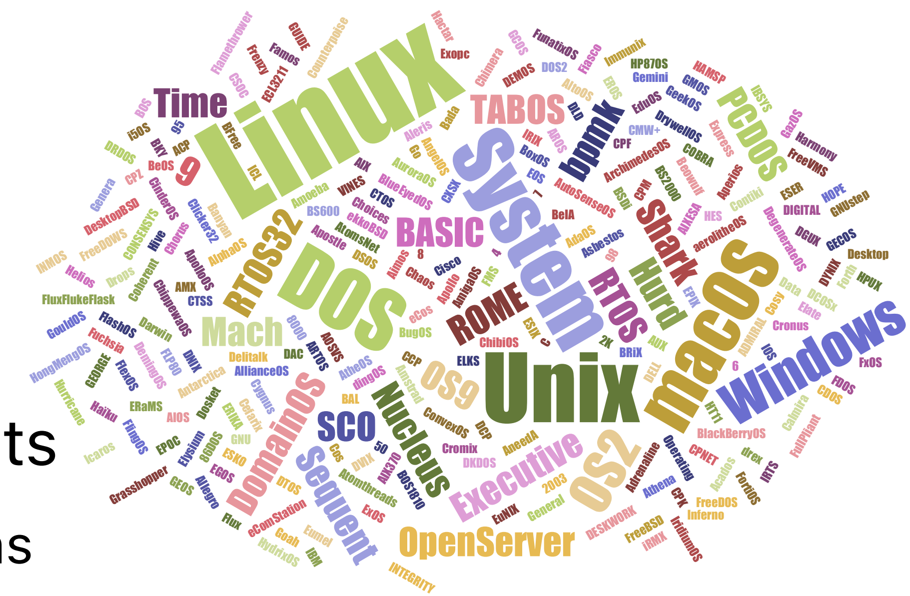
- Recap
- **Organizational Matters**
- Problem Scenario and Concepts
  - Critical Sections, Race Conditions
- Synchronisation
  - Ad-hoc Solutions, Busy Waiting
  - Hardware Support, Atomic Operations
  - Operating System Support, Passive Waiting
- Summary and Outlook

Organizational Matters



- lecture
  - Wednesday, 10:15 – 11:45
  - format: synchronous, **hybrid**
    - → in presence (Room HID, Building ID)
    - → online lecture (Zoom)
  - language: English/German
  - exercises: **group allocation ~~almost~~ complete**
    - if you are not yet signed up → Moodle
    - make use of group work - for your own benefit!
- manage course material, asynchronous communication: Moodle
- `https://moodle.ruhr-uni-bochum.de/course/view.php?id=50698`

Agenda



- Recap
- Organizational Matters
- Problem Scenario and Concepts
  - Critical Sections, Race Conditions
- Synchronisation
  - Ad-hoc Solutions, Busy Waiting
  - Hardware Support, Atomic Operations
  - Operating System Support, Passive Waiting
- Summary and Outlook

Scenario: Processes and Shared Resources

- process := **program in execution** (under OS control)
  - the core abstraction for control flows in computer systems
  - conceptually independent
  - technically a multiplexing of the CPU takes place
  - the operating system determines the time of **preemption** and the dispatching order of the processes in the ready list

- sharing of resources: **code** and **data**
  - **threads** (and user-level threads) operate in the **same address space**
  - the operating system can use the MMU to map a **single memory area** into **multiple address spaces (i.e., different processes)**
  - operating system data is also shared (in a **controlled** manner)

Definition: Critical Section

- in the case of race conditions, n processes fight over access to shared data
- the code fragments in which this critical data is accessed are called **critical sections**

Problem

- it must be ensured that only one process can be in a critical section at any time

Example: Singly Linked List in C

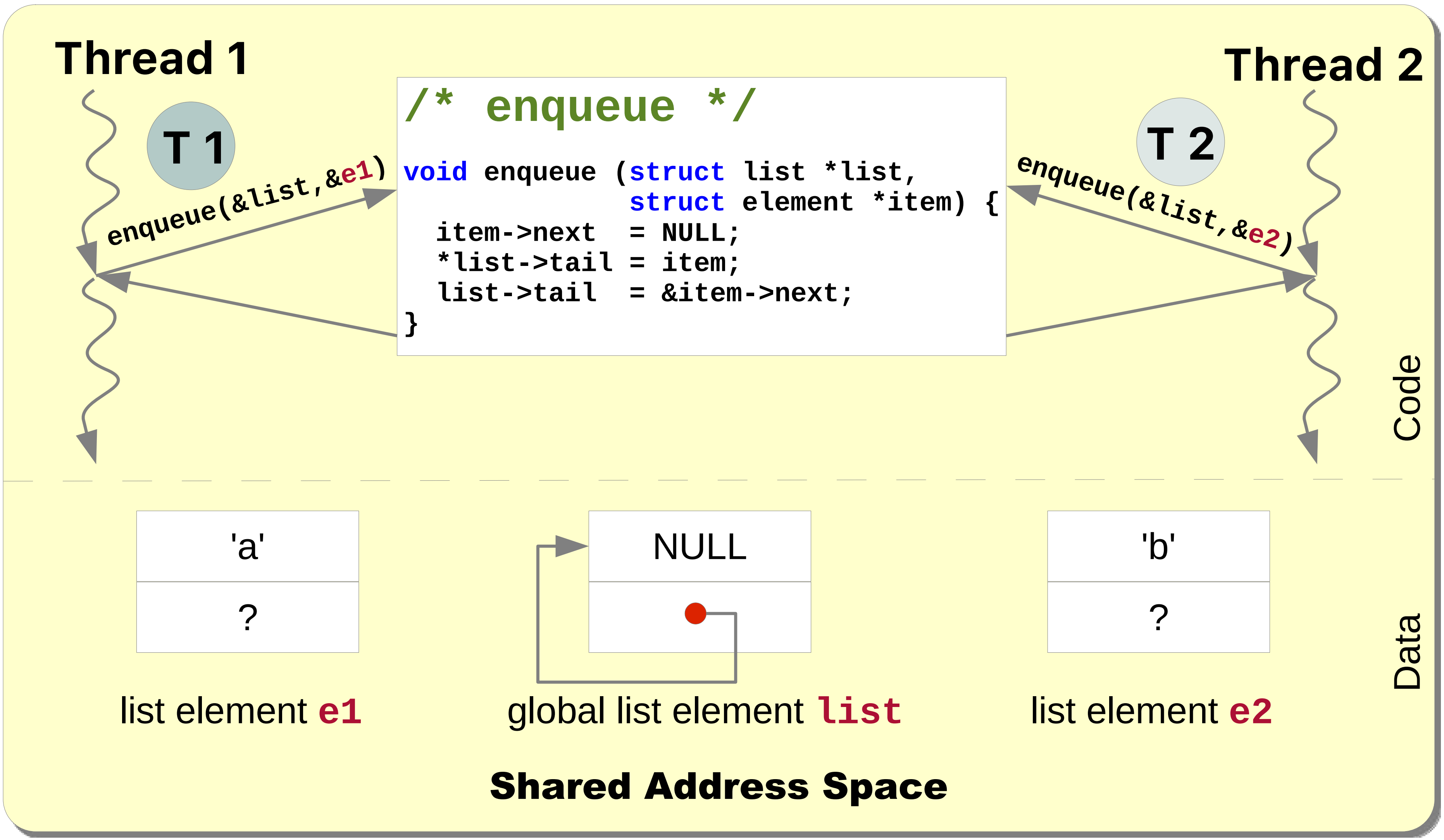
```
/* data type for list elements */
struct element {
  char payload;             /* actual “payload” */
  struct element *next;     /* next pointer */
};

/* data type for managing lists */
struct list {
  struct element *head;     /* first element */
  struct element **tail;    /* 'next' of the last element */
};

/* function to insert a new list element */
void enqueue (struct list *list, struct element *item) {
  item->next  = NULL;
  *list->tail = item;
  list->tail  = &item->next;
}
```

Note: The list implementation is clever because tail does *not* refer to the last element but to the next pointer, there is *no* special treatment for insertion into an empty list.

Example: Singly Linked List in C



Example: Singly Linked List in C

First Scenario:
Thread 2 *after* Thread 1

T 1

```
enqueue(&list,&e1)
```

```
e1->next = NULL;
```

```
*list->tail = e1;
```

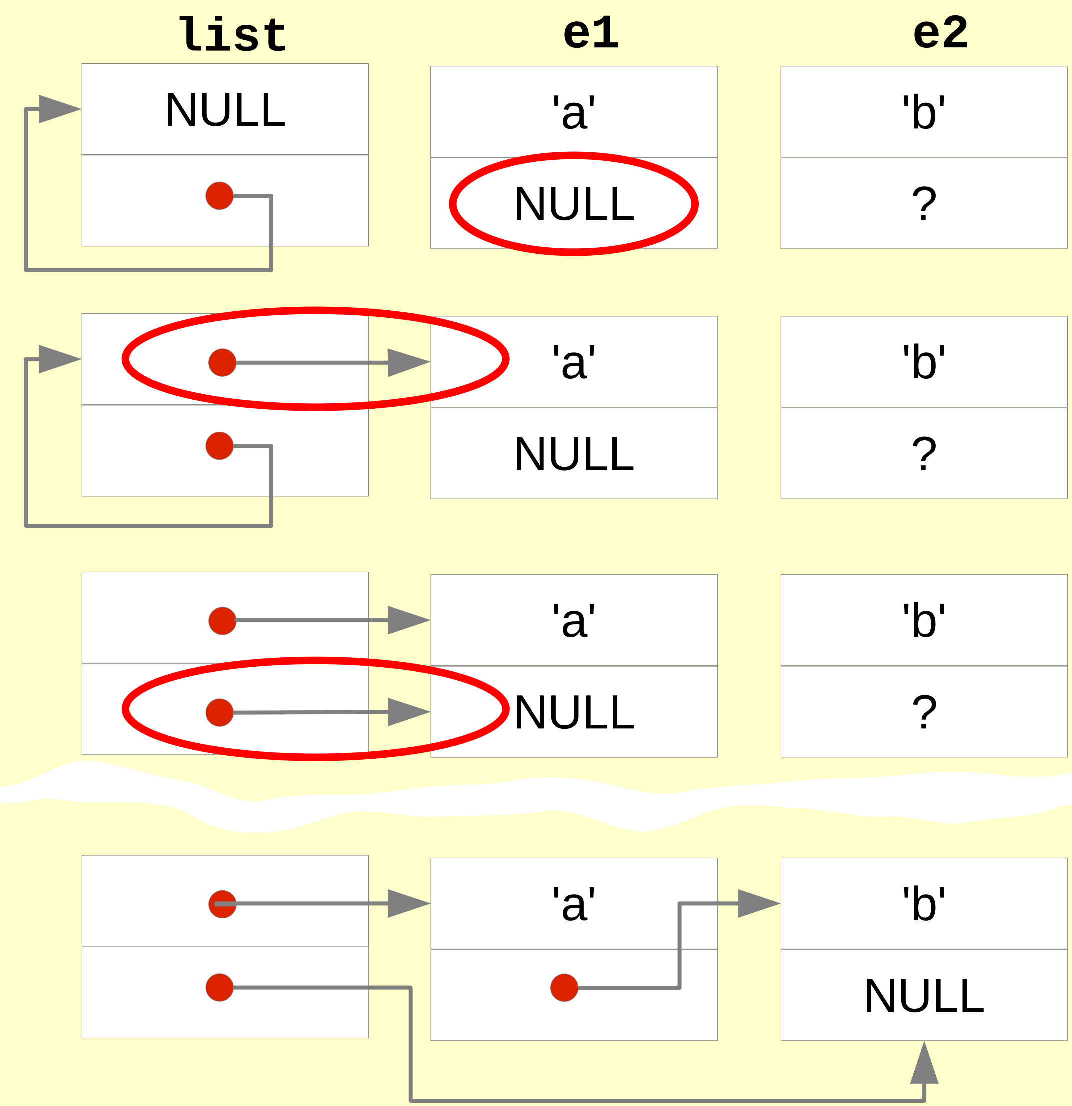
```
list->tail = &e1->next;
```

⋮

T 2

```
enqueue(&list,&e2)
```

```
e2->next  = NULL;
*list->tail = e2;
list->tail  = &e2->next;
```



Example: Singly Linked List in C

Second Scenario:
Thread 2 *overlaps* Thread 1

T 1

```
enqueue(&list,&e1)
```

```
e1->next = NULL;
*list->tail = e1;
```

context switch

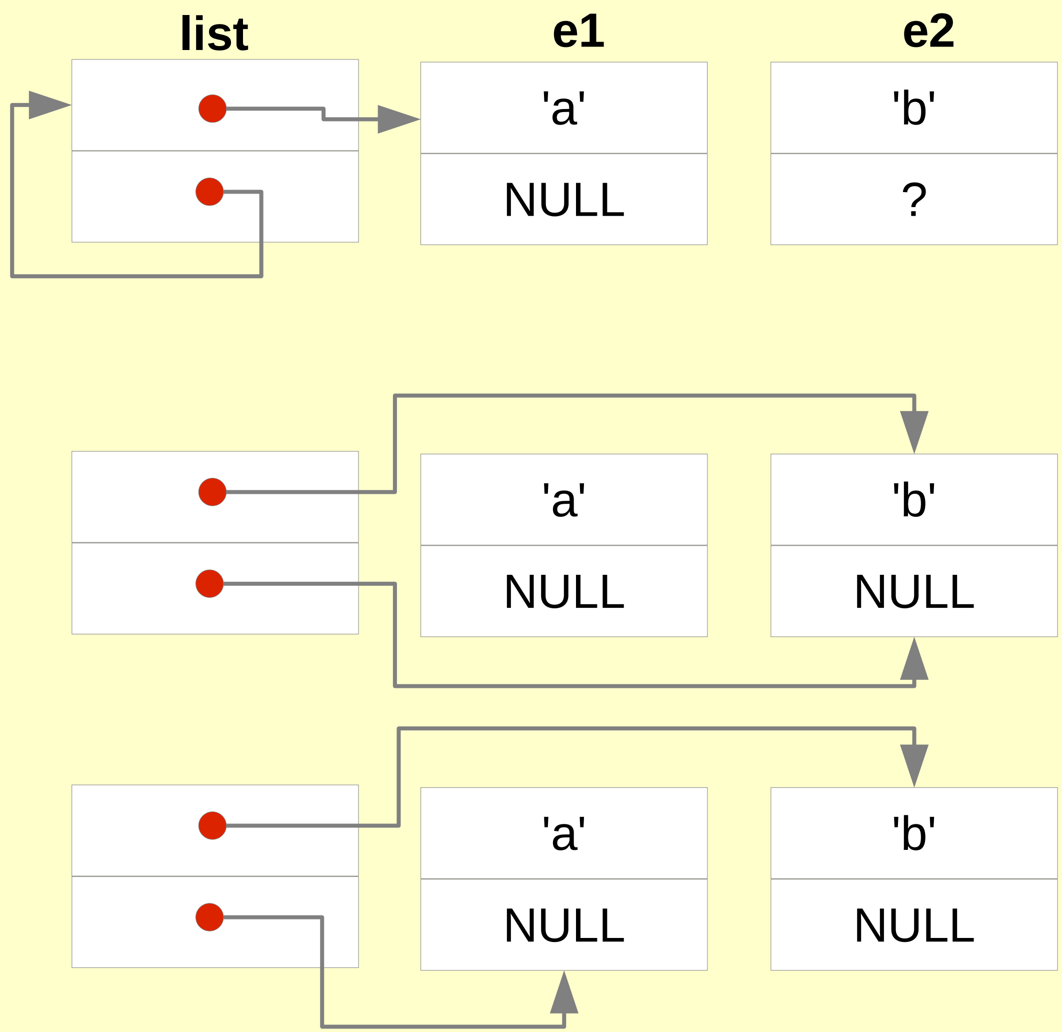
T 2

```
enqueue(&list,&e2)
```

```
e2->next  = NULL;
*list->tail = e2;
list->tail  = &e2->next;
```

context switch

```
list->tail = &e1->next;
```



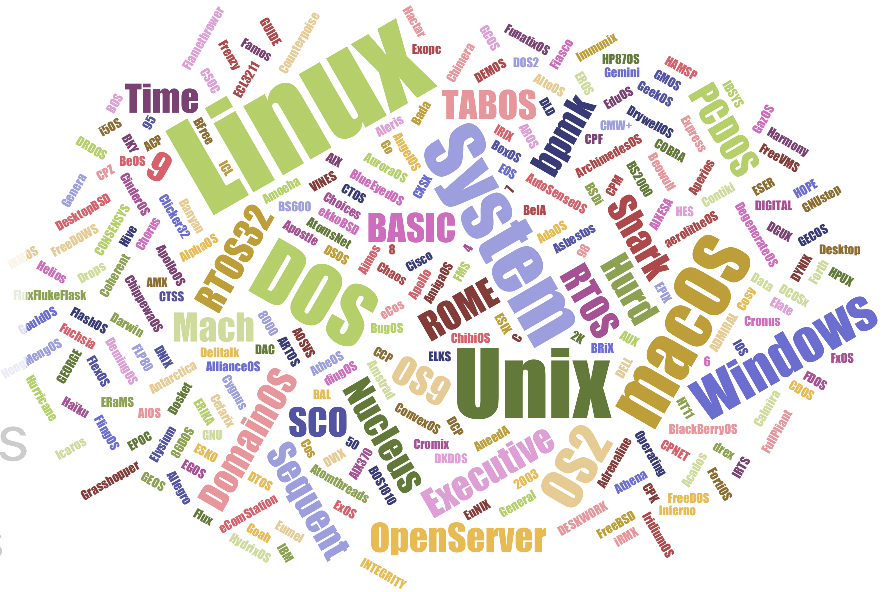
Similar Patterns

- **shared memory** used for interprocess communication
  - systems with “shared memory” service
- **threads (and user-level threads)**
  - concurrent (write) access to shared variables
- **operating system data** needed to coordinate the access of processes to indivisible resources
  - file system structures, process table, memory management
  - devices (e.g., terminals, printers, network interfaces)
- **interrupt synchronisation**
  - **caution:** synchronisation methods that are suitable for *process synchronisation* do not necessarily work for *interrupt synchronisation*!

Definition: Race Condition

- a race condition (dt. Wettlaufsituation) is a situation in which **multiple processes concurrently access shared data** and **at least one manipulates** it
- the ultimate value of the shared data in such a race condition depends on the **order** in which the processes access it
- the result is therefore **unpredictable** and may even be **incorrect** in the case of overlapping accesses!
- to **avoid** race conditions, concurrent processes that work on shared data must be **synchronised**

Agenda



- Recap
- Organizational Matters
- Problem Scenario and Concepts
  - Critical Sections, Race Conditions
- Synchronisation
  - Ad-hoc Solutions, Busy Waiting
  - Hardware Support, Atomic Operations
  - Operating System Support, Passive Waiting
- Summary and Outlook

Definition: Synchronisation

- synchronisation := **coordination of cooperation and competition** between processes
- synchronisation brings the activities of different concurrent processes into a **<u>certain</u> order**
- it thus achieves across processes what the sequentiality of activities ensures within a process

Based on: Herrtwich/Hommel (1989), Cooperation and Competition

Approach: Lock Variables

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

```
Lock lock; /* global lock variable */

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

  acquire (&lock);

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

  release (&lock);
}
```

- block a process until the associated lock is *released*
- then *acquires* the lock "from the inside"

- *release* the associated lock - without delaying (or even blocking) the calling process

basic pattern of a **locking algorithm**

Naive Solution with Busy Waiting

```c
/* lock variable (initially zero) */
typedef unsigned char Lock;

/* enter critical section */
void acquire (Lock *lock) {
  while (*lock); /* empty loop body, block */
  *lock = 1;
}

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

Naive Solution with Busy Waiting

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

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

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

- **`acquire`** is supposed to protect a critical section, but its execution itself is critical!
  - problematic is the moment *after* leaving the while-loop and *before* setting the lock variable
  - when the running process is preempted at this moment, another process could find the critical section free and enter it, **too**

In the further course (at least) two processes could overlap the critical section that actually is (pseudo-)protected by **`acquire`**!

Lamport: Bakery Algorithm

- **before** a process is allowed to enter the critical section, it is given a **wait number**
- **entrance** to the critical section in **order** of the **wait number**
- if the critical section is free, the process with the **lowest wait number** may enter the critical section
- wait number **expires** upon leaving the critical section

Caveat:

- the algorithm *cannot* guarantee that a wait number is assigned to only one process
- consideration of process priorities to resolve this limitation

Lamport: Bakery Algorithm

```
typedef struct { /* lock variable (initially everything 0) */
  bool choosing[N]; /* is x drawing a waiting number? */
  int number[N];    /* which waiting number has x?    */
} Lock;

void acquire (Lock *lock) { /* enter critical section */
  int j; int i = pid();
  lock->choosing[i] = true;
  lock->number[i]   = max(lock->number[0], …, number[N-1]) + 1;
  lock->choosing[i] = false;
  for (j = 0; j < N; j++) {
    while (lock->choosing[j]);
    while (lock->number[j] != 0 &&
           (lock->number[j] < lock->number[i] ||
            (lock->number[j] == lock->number[i] && j < i)));
} }

void release (Lock *lock) { /* leave critical section */
  int i = pid(); lock->number[i] = 0;
}
```

note: pseudo code

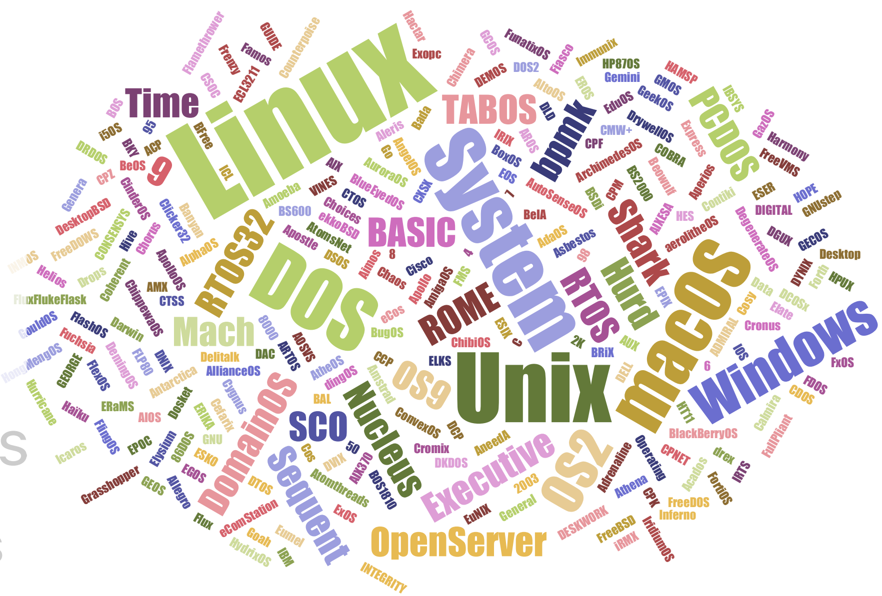
Discussion: Bakery Algorithm

the algorithm is a provably correct solution to the problem of critical sections, but:

- as a rule, it is **unknown** in advance **how many processes** will compete for entry into a critical section
- process IDs are not in the value range from 0 to N-1
- execution time of the function acquire is $O(N)$ even **if the critical section is free**

Wanted: A algorithm which is correct *and* as simple as the naive approach!

Agenda



- Recap
- Organizational Matters
- Problem Scenario and Concepts
  - Critical Sections, Race Conditions
- Synchronisation
  - Ad-hoc Solutions, Busy Waiting
  - Hardware Support, Atomic Operations
  - Operating System Support, Passive Waiting
- Summary and Outlook

Locks with Atomic Operations

- at the hardware level, CPUs implement (atomic) **read/modify/write (RMW) operations** which are suitable for the implementation of lock algorithms
- Motorola 68K: TAS (***test-and-set***)
  - sets bit 7 of the destination operand and returns the previous state in condition code bits

```
acquire:    TAS    lock
            BNE    acquire
```

- Intel x86: XCHG (***exchange***)
  - atomically exchanges the contents of a register with that of a variable in memory

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

- PowerPC: LL/SC (***load linked/store conditional***)

Discussion: Busy Waiting

Issue of the lock algorithms shown so far:
the actively waiting process (i.e., busy waiting)

- since the process is waiting, it cannot provoke a change in the condition it is blocked on
- therefore the waiting process unnecessarily restricts the progress of other processes
- thus ultimately also harms itself:
  - the longer the process keeps the processor to itself → the longer it has to wait for other processes to fulfill the condition it is blocked on

Disabling Interrupts

- note: only interrupts (e.g., timer) lead to the CPU revocation from a process during its execution of a critical section

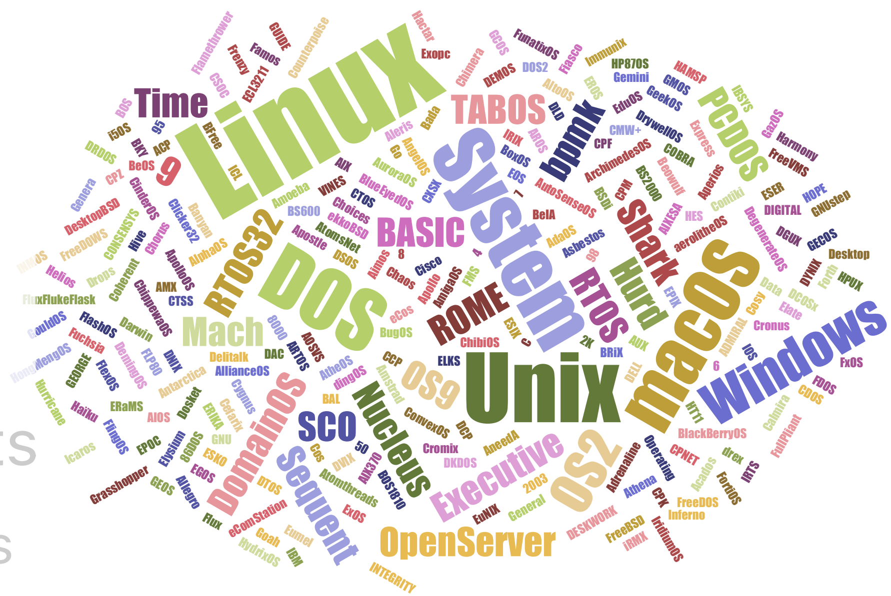
```
/* enter critical section */
void acquire (Lock *lock) {
  asm ("cli");
}

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

`cli` and `sti` are used with Intel x86{,-64} processors to **disable** and **enable** interrupts

- this "solution" will affect all processes and the operating system itself (including device drivers)
  - **`sti`** and **`cli`** must therefore not be used in user mode

Agenda



- Recap
- Organizational Matters
- Problem Scenario and Concepts
  - Critical Sections, Race Conditions
- Synchronisation
  - Ad-hoc Solutions, Busy Waiting
  - Hardware Support, Atomic Operations
  - Operating System Support, Passive Waiting
- Summary and Outlook

Discussion: Passive Waiting

- processes yield control of the CPU while waiting for events to occur
  - in case of synchronisation a process "blocks itself" on an event
  - PCB of the process placed in a queue
  - if the event occurs, a process waiting for it is deblocked
- the waiting phase of a process is designed as a blocking phase (cf. I/O burst)
  - the schedule for the processes is updated (scheduling)
  - another, ready process is dispatched according to schedule (dispatching)
  - if no process is ready anymore, the CPU runs "empty" (idle phase)
- with the beginning of its blocking phase the CPU burst of a process also ends

Semaphore (dt. Semaphor {m})

- OS abstraction to implement synchronisation of processes using **passive waiting**

- semaphore: a "non-negative integer", for which two distinct atomic (**<u>indivisible operations</u>)** are defined:

  P (hol. prolaag, „decrement“; also: *down*, ***wait***)
  - if the semaphore has the value 0: the running process is blocked
  - else: the semaphore is *decremented* by 1

  V (hol. verhoog, „increase“; also: *up*, ***signal***)
  - if existing: another process which is blocked on the semaphore is deblocked
  - else: the semaphore is *incremented* by 1

- semaphore is an operating system abstraction for exchanging synchronisation signals between **concurrent processes**

Semaphore

```
/* Semaphore Implementation, Pseudo-Code */
Semaphore semaphore;

semaphore.wait() {
    if (counter == 0) {
      worker *self = scheduler.active(); /*
      enqueue(self);
      scheduler.block(self);
    }
    else
      counter--;
}

semaphore.signal() {
    worker *worker = dequeue();
    if (worker)
      scheduler.wakeup(worker);
    else
      counter++;
}
```

Semaphore provides a list of PCBs with the access methods **enqueue** and **dequeue**.

The scheduler must provide 3 operations:

- **active:** provides PCB of the running process
- **block:** puts a process into the BLOCKED state
- **wakeup:** puts a blocked process back on the ready list

Semaphore: Simple Interaction Patterns

- **unilateral synchronisation** (dt. einseitige Synchronisation)

```
/* shared memory */
Semaphore elem;
struct list l;
struct element e;
```

```
void producer() {
  enqueue(&l, &e);
  signal(&elem);
}
```

```
void consumer() {
  struct element *x;
  wait(&elem);
  x = dequeue(&l);
}
```

```
/* initialisation */
elem = 0;
```

- example: producer/consumer relationship between threads
- only consumer may block (i.e., when no consumable resource is available)
- blocked consumers resume when consumable resources are available (i.e., when a producer calls `signal()`)

Semaphore: Simple Interaction Patterns

- **multilateral synchronisation** (dt. mehrseitige Synchronisation)

```
/* shared memory */
Semaphore resource;
```

```
/* initialisation */
resource = N; /* N >= 1 */
```

modus operandi:
as with mutual exclusion

- example: threads share reusable resources (e.g., buffers), only a limited number of the shared resource is available for simultaneous use
- access to/allocation of the shared resource must be synchronized between all threads involved
- all threads are potentially blocked if all available resources have already been spent

Semaphore: Interactions

- example: **the first readers-writers problem**

As with mutual exclusion, a critical section should be protected in this example. However, there are **two classes** of competing processes:

- **writer**: they change data and must therefore get mutual exclusion guaranteed
  - concurrent writes are unsafe
- **reader**: they access data read-only and therefore multiple readers may enter the critical section at the same time
  - concurrent reads are safe

Semaphore: Interactions

- example: **the first readers-writers problem**

```
/* shared memory */
Semaphore mutex;
Semaphore writer;
int readcount;
```

```
/* initialisation */
mutex     = 1;
writer    = 1;
readcount = 0;
```

```
/* writer */
wait (&writer);

/* write operations */
signal (&writer);
```

```
/* reader */
wait(&mutex);
readcount++;
if (readcount == 1)
  wait(&writer);
signal(&mutex);

/* read operations */
wait(&mutex);
readcount--;
if (readcount == 0)
   signal(&writer);
signal(&mutex):
```

Semaphore: Discussion

- **semaphore extensions and variants**
  - binary semaphore or ***mutex***
  - non-blocking `wait()`
  - timeout: `wait()` with expiration (e.g., 500 ms)
  - counter arrays (several semaphores)
- **pitfalls and sources of error**
  - risk of deadlocks
  - complex synchronisation patterns become difficult

    dependence of cooperating processes
    - all processes must follow the protocols strictly
  - semaphore usage is *not* enforced

→ monitor concept: programming language support

Putting a Semaphore to Good Use

mutual exclusion: a semaphore initialized with 1 semaphore can act as a lock variable

```
Semaphore lock; /* = 1; semaphore as locking 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 that **enters** the critical section **decreases** the counter **to 0**
- all subsequent **processes block**

- when **leaving** either
  (I) a blocked **process** is **woken** up *or*
  (II) the counter is **increased to 1** again

Agenda

- Recap
- Organizational Matters
- Problem Scenario and Concepts
  - Critical Sections, Race Conditions
- Synchronisation
  - Ad-hoc Solutions, Busy Waiting
  - Hardware Support, Atomic Operations
  - Operating System Support, Passive Waiting
- **Summary and Outlook**

⏭ Summary and Outlook

- **summary**
  - uncontrolled, concurrent data access leads to errors
    - **synchronisation** methods ensure coordination
    - necessary for: **cooperation** (producer/consumer) and **competition** (concurrent use of system resources)
  - **ad-hoc method:** busy waiting
    - **waste of resources**, in particular previous CPU time
  - synchronisation methods with **hardware** and **operating system support**
    - **mutual exclusion** using **atomic operations**
    - **semaphores, unilateral** and **multilateral synchronisation**
    - flexible but sensitive to errors when **synchronisation protoco**l is violated
- **outlook: deadlocks**
  - **process deadlocks:** mutual blocking of concurrent but independent control flows
  - deadlock **prevention**, **avoidance**, **detection**

References and Acknowledgments

Lecture

- Systemnahe Programmierung in C (SPiC), Betriebssysteme (Jürgen Kleinöder, Wolfgang Schröder-Preikschat)
- Betriebssysteme und Rechnernetze (Olaf Spinczyk, Embedded Software Systems Group, Universität Osnabrück)

Teaching Books and Reference Book

[1] Avi Silberschatz, Peter Baer Galvin, Greg Gagne: *Operating System Concepts*, John Wiley & Sons*, 2018*.

[2] Andrew Tanenbaum, Herbert Bos: *Modern Operating Systems*, Pearson, 2015.

[3] Wolfgang Schröder-Preikschat: *Grundlage von Betriebssystemen – Sachwortverzeichnis*, 2023.
https://www4.cs.fau.de/~wosch/glossar.pdf