NTNU | Norwegian University of Science and Technology

# Operating Systems

## Lecture 6: Concurrency: Synchronization

Michael Engel

# Processes, once more…

Processes again

- Processes are **programs in execution** (under the control of the OS)
  - ***The*** abstraction for control flows in computers
  - Processes are conceptionally independent
  - Technically, the CPU is ***multiplexed***
  - The OS determines when a process is to be preempted and in which order processes are executed
- Processes have an **address space**
  - Logical addresses of a process are mapped to physical addresses using the hardware (MMU)
- Processes can share code and data areas
  - Threads and fibers operate in the same address space
  - The OS can map a single memory area into multiple address spaces using the MMU
  - Data of the OS itself is also shared (in a controlled way)

# Example: Shared data

**Synchronization**

A simple linked list implementation in C:

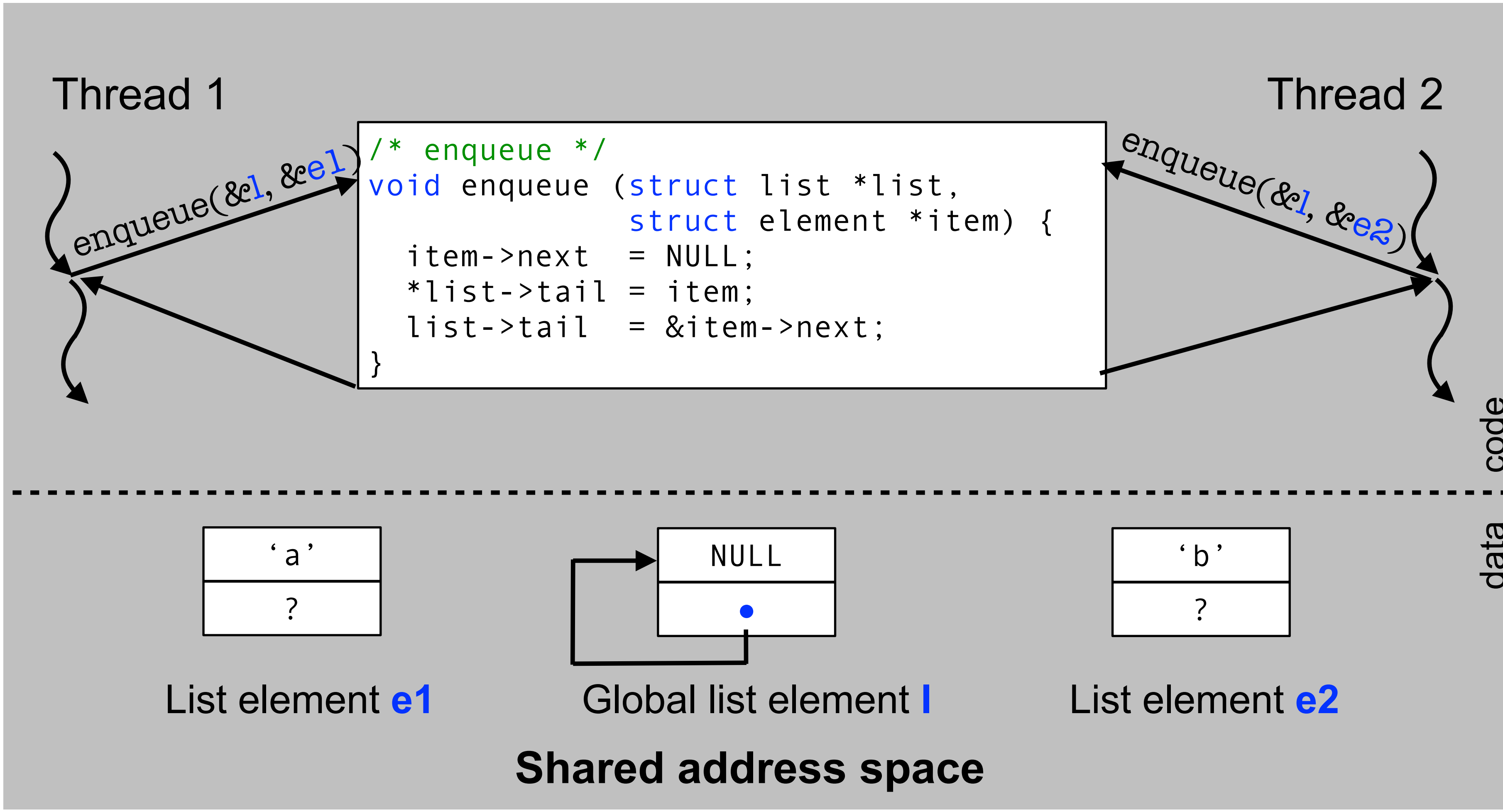
```
/* Data type for list elements */
struct element {
  char payload;           /* the data to be stored */
  struct element *next;   /* pointer to next list element */
};

/* Data type for list administration */
struct list {
  struct element *head;   /* first element */
  struct element **tail;  /* 'next' pointer in last element */
};

/* Function to add a new element to the end of the list */
void enqueue (struct list *list, struct element *item) {
  item->next  = NULL;
  *list->tail = item;
  list->tail  = &item->next;
}
```

**This list implementation is a bit sophisticated. Since tail does not point to the last list element, but to its next pointer, we don't need any special case to add an element to an empty list.**

# Example: simple linked list in C

**Synchronization**

## Scenario



# Example: simple linked list in C

Synchronization

## Case 1: thread 2 *after* thread 1

enqueue(&l, &e1)

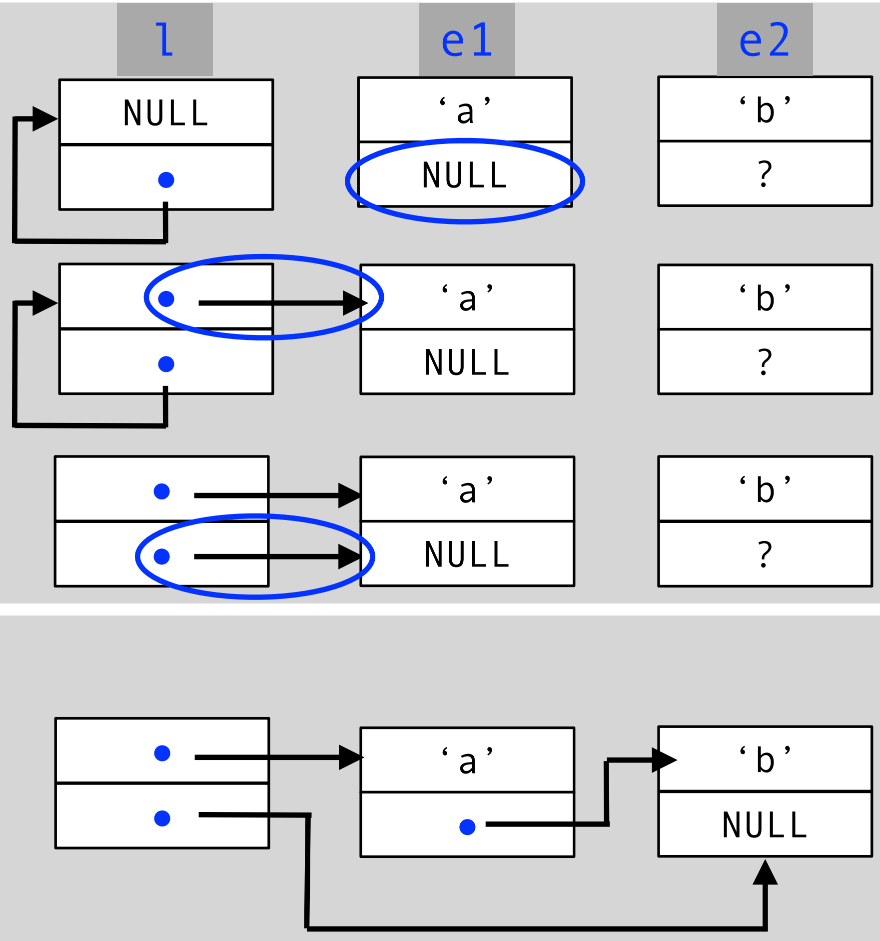
```
item->next  = NULL;
```

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

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

enqueue(&l, &e2)

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



# Example: simple linked list in C

Synchronization

## Case 2: thread 2 *overlaps* thread 1

`enqueue(&l, &e1)`

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

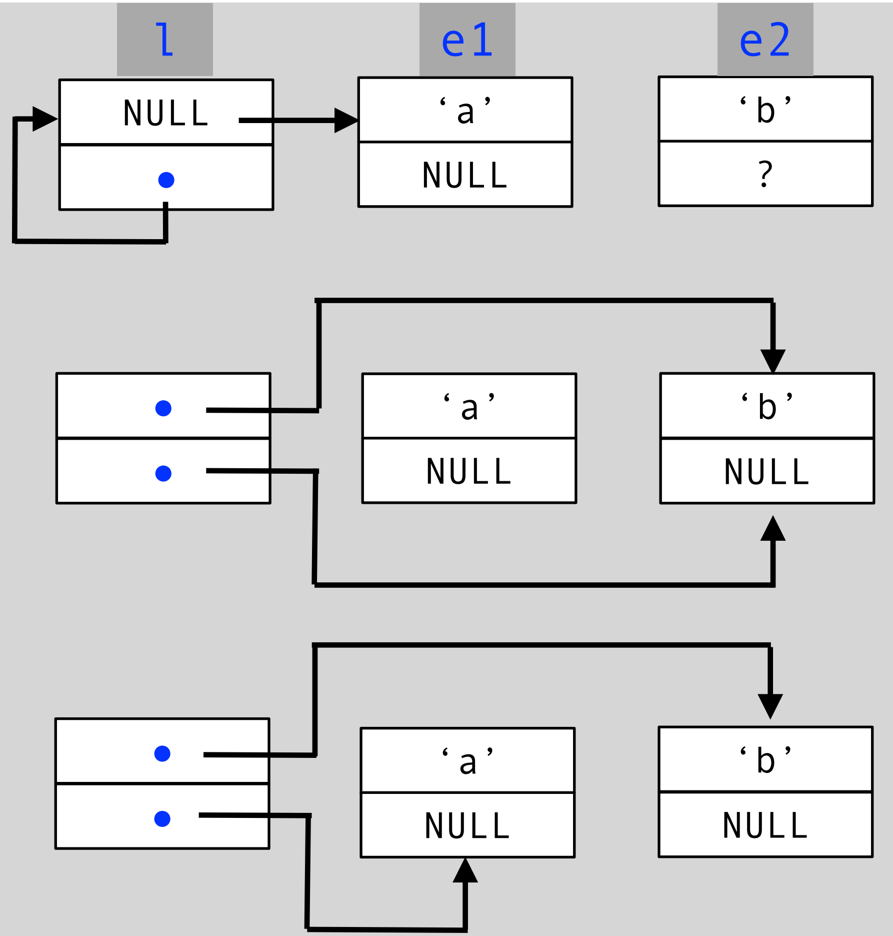
↯ process switch

`enqueue(&l, &e2)`

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

↯ process switch

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



# Where else does this problem occur?

Synchronization

- **Shared memory** used to communicate between processes
  - Systems with a shared memory service
- **Threads and fibers**
  - Concurrent access to the same variables
- **Operating system data** which are used to coordinate the access of processes to non-divisible resources
  - File system structures, process table, memory management, …
  - Devices (terminals, printers, network interfaces, …)
- Similar special case: **interrupt synchronization**
  - Caution: methods that work for synchronizing processes do not necessarily work for interrupts!

# The problem: *race conditions*

Synchronization

- 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

Synchronization

- The coordination of to 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

Synchronization

- 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

Synchronization

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

Synchronization

**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

Synchronization

```
/* 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!**

# A working solution: “bakery” algorithm

Synchronization


(probably not that common in Norway…)

- A process takes a ***waiting number (ticket)*** before it is allowed to enter the critical section [1]
- Admission in order of the waiting numbers
  - i.e. the process with the lowest number is allowed to enter the critical section when the section is free
  - When leaving the critical section, its waiting number is invalidated
- **Problem**
  - The algorithm cannot guarantee that a waiting number is given to only one process
  - In this case, a process ID (0..N-1) decides about the priority

# A working solution: “bakery” algorithm

```
typedef struct { /* lock variables (initially all 0) */
  bool choosing[N]; int number[N];
} 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;
}
```

***Careful: this is pseudo code!***

# Discussion: bakery algorithm

Synchronization

The bakery algorithm is a provably correct solution for the problem of critical sections, but...

- in most cases, it is not known beforehand how many processes will compete to enter a critical section
- process IDs are not necessarily in a range 0…N-1
- the `acquire` function has a long runtime even in cases where the critical section is already free → O(N)

**Can we find a correct algorithm that is as simple as the** (incorrect) **naïve approach?**

# Locks with atomic operations

Synchronization

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
```

# Discussion: active waiting

Synchronization

- So far, our lock algorithms have a significant drawback:

  The **actively waiting** process…
  - is unable to change the condition it is waiting for on its own
  - It unnecessarily impedes other processes which would be able to use the CPU for “useful” work
  - It harms itself due to active waiting:
    - The longer a process holds the processor, the longer it has to wait for other processes to fulfill the condition it is waiting for
    - This problem does not occur in multi processor systems

# Suppressing interrupts

Synchronization

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 Intel x86 processors to disable and enable the handling of interrupts

# Alternative: *passive waiting*

Synchronization

- 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

**Synchronization**

- 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

# Example semaphore implementation

```
/* C++ implementation taken from the teaching OS OO-StuBS */
class Semaphore : public WaitingRoom {
  int counter;
public:
  Semaphore(int c) : counter(c) {}
  void wait() {
    if (counter == 0) {
      Customer *life = (Customer*)scheduler.active();
      enqueue(life);
      scheduler.block(life, this);
    }
    else
      counter--;
  }
  void signal() {
    Customer *customer = (Customer*)dequeue();
    if (customer)
      scheduler.wakeup(customer);
    else
      counter++;
  }
};
```

**A "WaitingRoom" is a list of processes (PCBs) with the access methods** `enqueue` **and** `dequeue`

**The *scheduler* has to provide three operations:**

- `active` **returns the PCB of the running process**
- `block` **moves a process into state BLOCKED**
- `wakeup` **puts a blocked process back on the READY list**

# Using semaphores

**Synchronization**

**"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: simple interactions

Synchronization

- “one sided synchronization”

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

```
/* initialization */
elem = 0;
```

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

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

- “resource oriented synchronization”

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

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

the rest: same as with mutual exclusion

# Semaphores: complex interactions

Synchronization

- Example: the first ***reader/writer problem***

As with mutual exclusion, a critical section also has to be protected in this example

However, here we have two classes of concurrent processes:

- ***Writers:*** they change data and thus need a guarantee for mutual exclusion
- ***Readers:*** these only read data, thus multiple readers are allowed to enter the critical section at the same time

# Semaphores: complex interactions

Synchronization

- Example: the first ***reader/writer problem***

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

```
/* initialization */
mutex = 1;
wrt   = 1;
readcount = 0;
```

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

... write data ...

signal(&wrt);
```

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

... read data ...

wait(&mutex);
readcount--;
if (readcount == 0)
  signal(&wrt);
signal(&mutex);
```

# 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
- Support in programming languages

# Language support: Monitors

Synchronization

- A **monitor** is an **abstract data type** [3,4] with implicit synchronization properties:

  **multilateral** synchronization at the interface to the monitor

  - mutual exclusion of the execution of all monitor methods

  **unilateral** synchronization inside of the monitors using **condition variables**

  - **wait** blocks a process until a signal or condition occurs and implicitly releases the monitor again
  - **signal** indicates that a signal or condition has occured and unblocks (exactly one or all) processes blocking on this event

- Language-supported mechanism:
  Concurrent Pascal [5], PL/I, CHILL, . . . , **Java**

# Monitors: example code

**Synchronization**

***Careful: this is pseudo code!***

```
/* A synchronized queue */
monitor SyncQueue {
  Queue queue;
  condition not_empty;
public:
  /* add an element */
  void enqueue(Element element) {
    queue.enqueue(element);
    not_empty.signal();
  }
  /* remove an element */
  Element dequeue() {
    while (queue.is_empty())
      not_empty.wait();
    return queue.dequeue();
  }
};
```

**The language guarantees mutual exclusion of the access methods *per SyncQueue object***

enqueue **signals that the queue is no longer empty**

**If no process is waiting, nothing happens**

dequeue **first waits until at least one element is in the queue**

# Signaling semantics in monitors

Synchronization

- In the case of waiting processes, a monitor has to fulfill the following **requirements**:
  - at *least* one process waiting for the condition variable is ***and***
  - at *most* one process continues to run after the monitor operation

- There are different solution approaches, each with its own semantics:
  - Number of processes that are activated (all or only one)
    - If only one, then which one?
      ➢ Possible conflict with CPU allocation
  - Change of the monitor owner or no change
    - If no immediate change of the owner takes place, the waiting condition has to be checked again

# Monitors in Java

Synchronization

- **synchronized** is a keyword indicating mutual exclusion
- ***One*** implicit condition variable
  - **notify** or **notifyAll** instead of **signal**, no change of owner

```
/* A synchronized queue */
class SyncQueue {
  private Queue queue;
  /* add element */
  public synchronized void enqueue(Element element) {
    queue.enqueue(element);
    notifyAll();
  }
  /* remove element */
  public synchronized Element dequeue() {
    while (queue.empty()) wait();
    return queue.dequeue();
  }
};
```

# Conclusion

- Uncontrolled concurrent data access can lead to errors
  - **synchronisation methods** provide coordination
  - Grundsätzlich muss man bei der Implementierung aufpassen, dass die  Auswahlstrategien nicht im Widerspruch zum Scheduler stehen.
- Ad hoc approach: **active waiting**
  - **Caution! Waste** of compute time
  - But: a short active wait is better than blocking, especially in multi processor systems → lecture on multiprocessors
- Operating system-supported approach: **semaphores**
  - **Flexible** (enables many different synchronization patterns), but error-prone
- Language-supported approach: **monitors**
  - Less versatile compared to semaphores
  - Expensive, since many context switches are required
  - But monitors are a very safe approach

# References

1. Leslie Lamport (1974). "A new solution of Dijkstra's concurrent programming problem". Communications of the ACM. 17 (8): 453–455. doi:10.1145/361082.361093
2. Allen B. Downey. The Little Book of Semaphores. Green Tea Press 2016. https://greenteapress.com/wp/semaphores/
3. Per Brinch Hansen (1973). "7.2 Class Concept" (PDF). In Operating System Principles. Prentice Hall. ISBN 978-0-13-637843-3
4. C. A. R. Hoare (1974). "Monitors: an operating system structuring concept". Comm. ACM. 17 (10): 549–557. CiteSeerX 10.1.1.24.6394. doi:10.1145/355620.361161
5. Per Brinch Hansen (1975). "The programming language Concurrent Pascal". IEEE Trans. Softw. Eng. SE-1 (2): 199–207. doi:10.1109/TSE.1975.6312840