5. Synchronization (2)

5. Synchronization
5.3 Sync methods (cont'd)

Test-and-Set-Lock (TSL) (1)

- machine instruction (e.g. called TSL = Test and Set Lock), which **atomically** reads and sets a lock variable, i.e., without being interrupted in the meantime.

```
enter:
  tsl register, flag  ; copy variable value into register and
                      ; then set variable to 1
  cmp register, 0     ; was the variable 0?
  jnz enter           ; not 0: Lock was set, therefore loop
  ret

leave:
  mov   flag, 0       ; save 0 in flag: free lock
  ret
```


Test-and-Set-Lock (TSL) (2)

- TSL must handle two things:
  - disable interrupts, so that the test-and-set action will not be interrupted by a different process (that was picked by the scheduler)
  - in case of more than one CPU it must block access to the memory bus so that no process on a different CPU (which does not have interrupts disabled) can access the same variable

Active and passive Waiting (1)

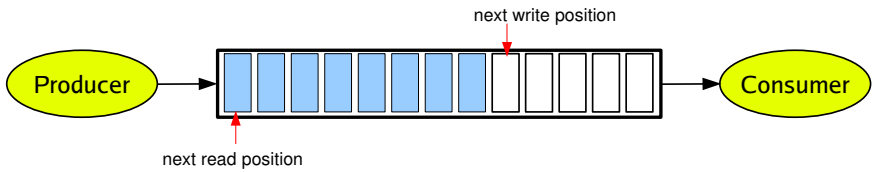
- active / busy waiting
  - execution of a loop until a variable holds a certain value.
  - the thread is ready and occupies the CPU.
  - Variable must be set by a different thread.
    - (big) problem, if the other thread terminates
    - (big) problem, if the other thread never sets the variable – e.g. because for priority reasons it is never scheduled

Active and passive Waiting (2)

- passive waiting (sleep and wake):
  - a thread blocks and waits for an event that will set it back to the „ready“ state.
  - the blocked thread wastes no CPU time.
  - a different thread must cause the event.
    - (smaller) problem if the other thread terminates.
  - when the event occurs, the blocked thread must be wokenup, e.g.
    - explicitely by another thread, or
    - through mechanisms of the operating system.

Producer Consumer Problem (1)

- in the producer consumer problem (bounded buffer problem) there are two cooperating threads:
  - the producer stores information items in a bounded buffer.
  - the consumer reads those items out of the buffer.



Producer Consumer Problem (2)

- **Synchronization**
  - **Don't overfill the buffer:**
    when the buffer is full, the producer must wait, until the consumer has taken an item out of the buffer (thus freeing one space) – then it can go on.
  - **Don't read from empty buffer:**
    when the buffer is empty, the consumer must wait until the producer has stored an item in the buffer; then it can continue.

Producer Consumer Problem (3)

- Implementation with passive waiting:
  - a shared variable „count“ counts the occupied positions in the buffer.
  - when the producer stores an item in the buffer and it has previously been empty (count == 0), it will wake up the consumer;
    in case of a full buffer it blocks (goes to sleep).
  - when the consumer retrieves an item from the buffer and it has previously been full (count == max), it wakes up the producer;
    in case of an empty buffer it blocks (goes to sleep).

Producer Consumer Problem with sleep / wake

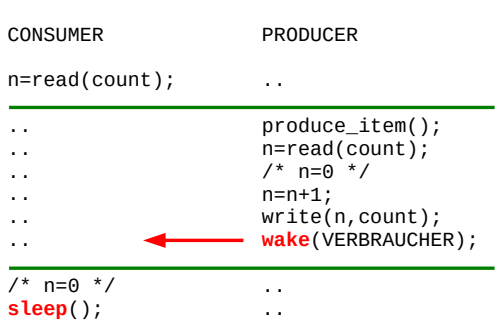
```
#define  N  100                      // # buffer size
int  count = 0;                      // # occupied positions in the buffer

producer () {
  while (TRUE) {                     // infinite loop
    produce_item (item);             // create something for the buffer
    if (count == N) sleep();         // if buffer full: sleep
    enter_item (item);               // store in the buffer
    count = count + 1;               // increment  # occupied positions
    if (count == 1) wake(consumer);  // was the buffer empty before?
  }
}

consumer () {
  while (TRUE) {                     // inifinte loop
    if (count == 0) sleep();         // if buffer empty: sleep
    remove_item (item);              // retrieve item from the buffer
    count = count - 1;               // decrement  # occupied positions
    if (count == N-1) wake(producer); // was the buffer full?
    consume_item (item);             // do something with the item
  }
}
```


Deadlock problem with sleep / wake (1)

- program contains a race condition that can lead to a deadlock, e.g. the following way:
  - consumer reads counter (with value 0)
  - context switch -> producer
  - producer puts an item into the buffer, increases the counter and wakes up the consumer, since counte was 0.
  - consumer goes to sleep since it still remembers the counter to be 0 (which was incremented in the meantime).
  - producer fills the buffer and goes to sleep as well

Deadlock problem with sleep / wake (2)

- **Cause of the problem:**
  wakeup signal for a – not yet – sleeping process is ignored
- wrong order of execution
- try to remember wakeup call for later usage...



Deadlock problem with sleep / wake (3)

- solution attempt: system calls *sleep* und *wake* use a „wakeup pending bit“:
  - when *wake()*-ing a non-sleeping thread, set its wakeup pending bit.
  - when *sleep()*-ing check the thread's wakeup pending bit – if it is set, don't put the thread to sleep.

but: solution cannot be generalized (several synchronized threads might need extra pending bits)

Semaphores (1)

a **semaphore** is an integer (counter) variable that can be used as follows:

- semaphore has a defined initial value N („number of available resources“).
- when requesting a semaphore (P or **Wait** operation):
  - decrease semaphore value by 1 if it is positive,
  - block thread and put it into a queue if the semaphore value is 0.

Semaphores (2)

- when releasing a semaphore (V or **Signal** operation):
  - wake up one thread in the queue if it is non-empty,
  - increment semaphore value by 1 (when there is no thread waiting for the semaphore)
- code always looks like this:

```
wait (&sem);
/* Code that uses the resource */
signal (&sem);
```


Semaphores (3)

- variant: negative semaphore values
  - semaphore counts number of waiting threads
  - request (wait):
    - decrement semaphore value by 1 (~~if it is positive~~)
    - block thread and put it into a queue when the semaphore value is ≤ 0.
  - release (signal):
    - wake up thread in the queue (if it is non-empty)
    - increment semaphore value by 1 (~~when there's no waiting thread in the queue~~)

Semaphores (4)

standard variant: semaphore can only have values ≥ 0

```
wait (sem) {
  if (sem>0)
    sem--;
    else BLOCK_CALLER;
}
```

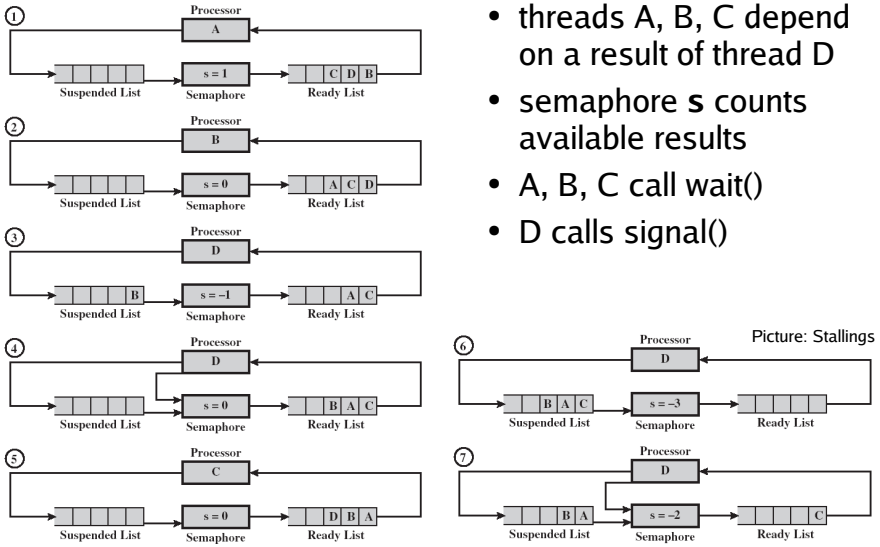
```
signal (sem) {
  if (P in QUEUE(sem)) {
    wakeup (P);
    remove (P, QUEUE);
  }
  else sem++;
}
```

variant: semaphore also negative, memorizes size of the queue

```
wait (sem) {
  if (sem<1)
    BLOCK_CALLER;
  sem--;
}
```

```
signal (sem) {
  if (P in QUEUE(sem)) {
    wakeup (P);
    remove (P, QUEUE); }
  sem++;
}
```

Semaphores (5): Example



- threads A, B, C depend on a result of thread D
- semaphore **s** counts available results
- A, B, C call wait()
- D calls signal()

Picture: Stallings

Mutexes (1)

- **Mutex:** boolean variable (true/false) that synchronizes access to shared data
  - true: access allowed
  - false: access forbidden
- **blocking:** a thread which attempts access while another thread already accesses the data will block ➞ queue
- when releasing:
  - queue contains thread(s) ➞ wake up one of them
  - queue empty: set mutex to true

Mutexes (2)

- **Mutex** (**mut**ual **ex**clusion) = **binary semaphore**, i.e. a semaphore that only used values 0 / 1

```
wait (mutex) {
  if (mutex==1)
    mutex=0;
    else BLOCK_CALLER;
}
```

```
signal (mutex) {
  if (P in QUEUE(mutex)) {
    wakeup (P);
    remove (P, QUEUE);
  }
  else mutex=1;
}
```

- new interpretation: wait ➞ lock
  signal ➞ unlock
- mutexes for exclusive access (critical regions)

Blocking or Non-blocking?

- operating systems cann implement mutexes and semaphores **blocking** or **non-blocking**
- blocking:
  when an attempt to decrement the counter fails
  ➞ warten
- non-blocking:
  when the attempt fails
  ➞ possibly do something else
  (yet still don't enter the critical region)

Atomic Operations

- the wait() and signal() operations for mutexes / semaphores must be implemented **atomically**:

  during the execution of wait() / signal() no other thread must be scheduled

Queues

- mutexes / semaphores manage queues (for the processes that were put to sleep, waiting on the mutex / semaphore)
- when calling signal(), a process might have to be woken up
- selection of the process to wake up is a task that is comparable with the scheduler's selection of a process to get the CPU next
  - FIFO: **strong** semaphore / mutex
  - random: **weak** semaphore / mutex

Producer Consumer Problem with Semaphores and Mutexes

```
typedef int semaphore;
semaphore mutex = 1;        // synchronizes buffer access
semaphore empty = N;        // counts free spaces in buffer
semaphore full = 0;         // counts used spaces in buffer

producer() {
  while (TRUE) {            // inifinite loop
    produce_item(item);     // create something for the buffer
    wait (empty);           // decrement empty spaces or block
    wait (mutex);           // enter critical region
    enter_item (item);      // store item in the buffer
    signal (mutex);         // leave critical region
    signal (full);          // increase used spaces; possibly wake consumer
  }
}

consumer() {
  while (TRUE) {            // inifinite loop
    wait (full);            // decrement used spaces or block
    wait (mutex);           // enter critical region
    remove_item(item);      // withdraw item from the buffer
    signal (mutex);         // leave critical region
    signal (empty);         // increase free spaces, possibly wake producer
    consume_entry (item);   // consume the item
  }
}
```