

# Systems 3

## Coordination

Marcel Waldvogel

Department of Computer and Information Science  
University of Konstanz

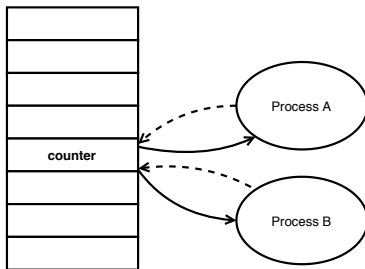
Winter 2019/2020

# Chapter Goals

- What is a critical region?
- How can mutual exclusion be achieved?
- What happens if mutual exclusion would be required but is not enforced?
- What are the preconditions for deadlocks?
- How can deadlocks be avoided?
  
- How do Mutexes, Semaphores, Monitors, and Message Passing work?
- How do they achieve mutual exclusion?
- What are their relationships (similarities, differences)?

# Race conditions

```
1 int counter = 0;
2
3 int main()
4 {
5     while(1) {
6         counter = counter + 1;
7     }
8     return 0;
9 }
```



- 1 Why does this **not** work? (Multiple reasons)
- 2 Why `counter = counter + 1` instead of `counter++`?

# Critical Sections: Avoiding Race Conditions

Basic assumptions necessary:

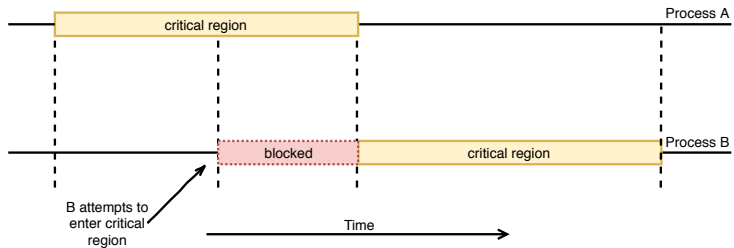
- 1 No two processes may be simultaneously inside their critical regions.
- 2 No assumptions may be made about speeds or the number of CPUs.<sup>1</sup>
- 3 No process running outside its critical region may block other processes.
- 4 No process should have to wait forever to enter its critical region.

⇒ Mutual exclusion

---

<sup>1</sup>One(!) or more. Are CPUs always the same speed?

# Mutual exclusion



# Strict Alternation

```
1  int turn = 0;
```

Thread 0:

```
2  while(1)
3  {
4      while (turn != 0) {}
5      critical_region();
6      turn = 1;
7      noncritical_region();
8  }
```

Thread 1:

```
9  while(1)
10 {
11     while (turn != 1) {}
12     critical_region();
13     turn = 0;
14     noncritical_region();
15 }
```

Two processes, **alternating** order.

# Peterson's Solution

```
1 int turn;          /* tie-breaker */
2 int interested[2]; /* all values initially 0 */
3
4 void enter_region(int process) /* process is 0 or 1 */
5 {
6     int other = 1 - process;    /* number of the other process */
7     interested[process] = 1;    /* show that you are interested */
8     turn = other;               /* set flag */
9     while (turn == other && interested[other]) {}
10 }
11
12 void leave_region(int process)
13 {
14     interested[process] = 0;    /* no longer in critical region */
15 }
```

- 1 At most one process in critical region?
- 2 Progress is being made (no dead-lock)?
- 3 Bounded waiting time?

Two processes, any order.

# Test-Set-Lock Instruction

```
1 enter_region:
2     tsl register,lock    | atomically copy lock to register and set lock to 1
3     cmp register,#0      | was lock zero?
4     jne enter_region     | if it was non zero, lock was set, so loop
5     ret                  | return to caller; cr entered
6
7 leave_region:
8     move lock,#0         | store a 0 in lock
9     ret                  | return to caller
```

**Any number of** processes, **any** order.



# Atomic Support Examples

```
1 bool __atomic_clear(void *ptr, int memorder);
2 bool __atomic_test_and_set(void *ptr, int memorder);
3
4 char lock;
5
6 /* Acquire */
7 while (__atomic_test_and_set(&lock, __ATOMIC_ACQUIRE)) {
8     /* Wait for lock */
9 }
10
11 /* Critical section here */
12
13 /* Release */
14 __atomic_clear(&lock, __ATOMIC_RELEASE);
```

# Producer-Consumer

## Bank Teller

- Any number of customers
- Customers can come whenever they want
- Any number of tellers
- Tellers work in parallel

## Holiday card writers

- Any number of card writers
- Put finished cards on the shared desk
- Any number of envelope packagers
- Pick up cards for packaging

```
1  #define N 100    // number of slots in the buffer
2  int count = 0;   // number of items in the buffer
```

```
1 void producer(void)
2 {
3     int item;
4     while (1) {
5         item = produce_item();
6         if (count == N) sleep();
7         insert_item(item);
8         count = count + 1;
9         if (count == 1) wakeup(consumer);
10    }
11 }
12 }
```

```
1 void consumer(void)
2 {
3     int item;
4     while (1) {
5
6         if (count == 0) sleep();
7         item = remove_item();
8         count = count - 1;
9         if (count == N-1) wakeup(producer);
10        consume_item(item);
11    }
12 }
```

# Semaphores

## Semaphore

- Counter
- Special functions
  - `up()` aka `signal()`<sup>2</sup> aka `wakeup()`
  - `down()` aka `wait()` aka `sleep()`<sup>3</sup>
- Initialized to a value  $N$ <sup>4</sup>
- When `down()` would like to make the counter negative, the calling thread is blocked until another thread calls. `up()`

---

<sup>2</sup>Not to be confused with Unix/POSIX `signal(2)` handlers!

<sup>3</sup>Not to be confused with `sleep(2)`!

<sup>4</sup>How often one may call `down()` before an `up()` is needed.

For **mutual exclusion**, this is 1, for a buffer with  $N$  spaces, this is  $N$

```
1 #define N 100
2
3 typedef int semaphore;
4
5 semaphore mutex = 1;
6 semaphore empty = N;
7 semaphore full = 0;
```

```
8 void producer(void)
9 {
10     int item;
11     while (1) {
12         item = produce_item();
13         // Reserve empty space, any order
14         down(&empty);
15         // Enforce ordering now
16         down(&mutex);
17         insert_item(item);
18         up(&mutex);
19         // One more item is in
20         up(&full);
21     }
22 }
23 }
```

```
24 void consumer(void)
25 {
26     int item;
27     while (1) {
28
29         // Reserve one item, any order
30         down(&full);
31         // Enforce ordering
32         down(&mutex);
33         item = remove_item();
34         up(&mutex);
35         // One more empty space
36         up(&empty);
37         consume_item(item);
38     }
39 }
```

# Monitors

## Mutexes/Semaphores are hard to handle

- Hard to use (3 semaphores for the simple mechanism above!)
- Hard to maintain
- Hard to find (timing) bugs

## Need a simpler construct: Monitors

- Simple concept
  - Often slightly larger critical regions
  - Easy to maintain
  - Race conditions, deadlocks rare
- Only one of the monitor functions active at a time<sup>5</sup>

---

<sup>5</sup>synchronized in Java

```
1 monitor ProducerConsumer
2   condition full, empty;
3   integer count;
4
5   procedure insert(item: integer);
6   begin
7       if count = N then wait(full);
8       insert_item(item);
9       count := count + 1;
10      if count = 1 then signal(empty)
11      end;
12
13      function remove: integer;
14      begin
15          if count = 0 then wait(empty);
16          remove = remove_item;
17          count := count - 1;
18          if count = N - 1 then signal(full)
19          end;
20
21          count := 0;
22      end monitor;
```

```
1 procedure producer;
2 begin
3     while true do
4         begin
5             item = produce_item;
6             ProducerConsumer.insert(item)
7         end
8     end;
9
10 procedure consumer;
11 begin
12     while true do
13         begin
14             item = ProducerConsumer.remove;
15             consume_item(item)
16         end
17     end;
```

# Message Passing

## Message Passing

- Alternative to Mutex, Semaphore, Monitor
- Conceptually simple(r)<sup>6</sup>
- Does not require shared memory<sup>7</sup>

```
1 // could be implemented as
2
3 send(destination, &message);
4
5 receive(source, &message);
```

---

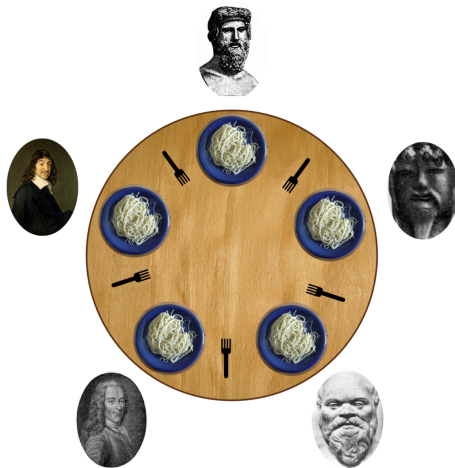
<sup>6</sup>Often implemented using the mechanisms described above

<sup>7</sup>I.e., also works over machine boundaries, over the network



```
1 #define N 100
2 void producer(void)
3 {
4     message m;
5     while (TRUE) {
6         m.item = produce_item();
7         receive(consumer, &m);
8         build_message(&m, item);
9         send(consumer, &m);
10    }
11 }
12
13 void consumer(void)
14 {
15     int item, i;
16     message m;
17     for (i = 0; i < N; i++) send(producer, &m);
18     while (TRUE) {
19         receive(producer, &m);
20         send(producer, &m);
21         consume_item(m.item);
22     }
23 }
```

# The Dining Philosophers Problem



CC BY-SA 3.0 Benjamin D. Esham / Wikimedia Commons

# Deadlock

“A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.”

## Conditions for Deadlock

- Mutual exclusion
- Hold and wait
- No (lock) preemption
- Circular wait

# Examples

- User level
  - Data structures: Hash tables, trees, linked lists, ...
  - SQL with two tables, A and B
  - Locking two files, A and B
- Kernel level
  - Locking two directories, A and B, when moving files between them
  - Writeback buffer, especially under memory pressure
- Linux
  - Interrupt disable
  - Big Kernel Lock<sup>8</sup>
  - Individual locks

---

<sup>8</sup>Introduced 1996 in Linux 2.0 for multi-processor support; gone as of Linux 3.0 (2011).

# Evolution: Fighting programmer's mistakes

## ■ Memory allocation: premature/missing release

- 1 C: Ownership strategy
- 2 C: Reference counting
- 3 Not exception-safe (break, return, ...)
- 4 HLL: Garbage collection

## ■ Locks: Premature/missing release

- 1 Ownership, counting semaphores
- 2 Java: synchronized
- 3 Python: with

# A nonsolution to the dining philosophers problem

```
1 #define N 5    // number of philosophers
2
3 void philosopher(int i)
4 {
5     while (TRUE) {
6         think();
7         take_fork(i);           // left
8         take_fork((i+1) % N);  // right
9         eat();
10        put_fork(i);           // left
11        put_fork((i+1) % N);   // right
12    }
13 }
```

Why not?

```
1 #define N 5
2 #define LEFT (i+N-1)%N // Left neighbor
3 #define RIGHT (i+1)%N // Right neighbor
4
5 typedef int semaphore;
6
7 enum{THINKING, HUNGRY, EATING} state[N];
8 // Is initialized to THINKING (=0)
9
10 semaphore mutex = 1;
11 semaphore s[N]; // Initialized to all 0s
12
13 void philosopher(int i)
14 {
15     while (1) {
16         think();
17         take_forks(i);
18         eat();
19         put_forks(i);
20     }
21 }
```

```
1 void take_forks(int i)
2 {
3     down(&mutex);
4     state[i] = HUNGRY;
5     test(i);
6     up(&mutex);
7     down(&s[i]);
8 }
9 void put_forks(int i)
10 {
11     down(&mutex);
12     state[i] = THINKING;
13     test(LEFT);
14     test(RIGHT);
15     up(&mutex);
16 }
17 void test(int i)
18 {
19     if (state[i] == HUNGRY &&
20         state[LEFT] != EATING &&
21         state[RIGHT] != EATING) {
22         state[i] = EATING;
23         up(&s[i]);
24     }
25 }
```

# Readers and Writers

## Example

- Given a shared data structure (hash, linked list, tree, database, ...)
  - Some threads want to modify the data structure (read-write access<sup>9</sup>)
  - Some threads only want to look up information (read-only access<sup>10</sup>)
- 1 Mutual exclusion **among readers** is **wasteful**.
  - 2 Only required **among writers** or **between readers and writers**.

## Problem

How to allow concurrent readers? How to still lock out writers?

- 1 Only **first reader in** locks the database; **last reader out** unlocks.
- 2 Writers always lock/unlock.

---

<sup>9</sup> aka “writer”

<sup>10</sup> aka “reader”



# A solution to the readers and writers problem

```
1  typedef int semaphore;
2  semaphore mutex = 1;
3  semaphore db = 1;
4  int rc = 0; // Reader count
5
6  void reader(void)
7  {
8      while (1) {
9          down(&mutex); // First in?
10         rc = rc + 1;
11         if (rc == 1) down(&db);
12         up(&mutex);
13         read_database();
14         down(&mutex); // Last out?
15         rc = rc - 1;
16         if (rc == 0) up(&db);
17         up(&mutex);
18         use_data_read();
19     }
20 }
```

```
21 void writer(void)
22 {
23     while (1) {
24         think_up_data();
25         down(&db);
26         write_database();
27         up(&db);
28     }
29 }
```

Expensive operations in **blue** and **red**; only the latter need mutual exclusion.