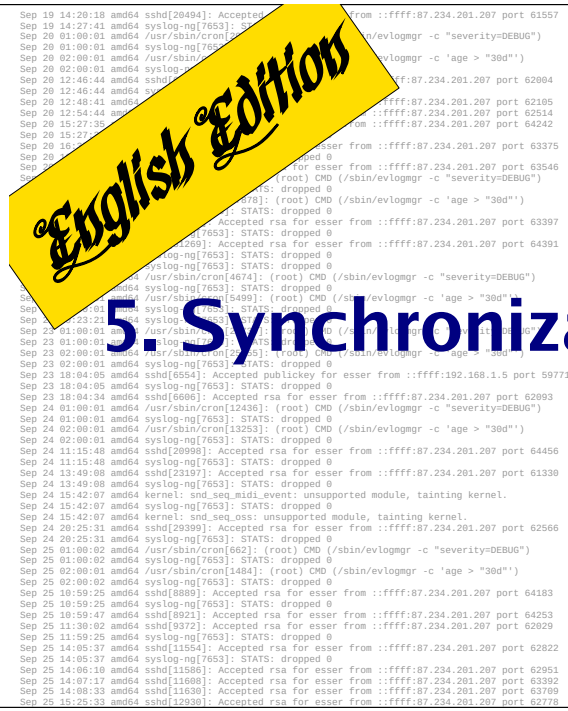

5. Synchronization (1)

Introduction (1)

- There are processes (or threads, Kernel functions etc.) with shared access on certain data, e.g.
  - threads of the same process: shared memory
  - processes with common memory-mapped file
  - processes / threads open the same file for reading / writing
  - SMP system: scheduler (one for each CPU) access the same process lists / queues

Introduction (2)

- Synchronization: problems with „parallel“ data access
- Example: two processes increase a counter

```
inc_counter( )
{
  w=read(Address);
  w=w+1;
  write(Address,w);
}
```

Initial situation: w=10

P1:	P2:
w=read(Address); // 10	
w=w+1; // 11	
	w=read(Address); // 10
	w=w+1; // 11
	write(Address,w); // 11
write(Address,w); // 11 !!	

result after P1, P2: w=11 – not 12!

Introduction (3)

- Programmer's intention is one of the following exceution orderings:

Initial situation: w=10

P1:	P2:
w=read(Addr); // 10	
w=w+1; // 11	
write(Addr,w); // 11	
	w=read(Addr); // 11
	w=w+1; // 12
	write(Addr,w); // 12

result after P1, P2: w=12

Initial situation: w=10

P1:	P2:
	w=read(Addr); // 10
	w=w+1; // 11
	write(Addr,w); // 11
w=read(Addr); // 11	
w=w+1; // 12	
write(Addr,w); // 12	

result after P1, P2: w=12

Introduction (4)

- Reason: `inc_counter()` does not work **atomically**:
  - scheduler can preempt function
  - function could be running on several CPUs simultaneously
- Solution: Find all code parts which reference shared data, and guarantee that there is always at most one process that accesses the date (mutual exclusion)

Introduction (5)

- Analogous problem with databases:

```
exec sql CONNECT ...
exec sql SELECT balance INTO $var FROM accounts
         WHERE accno = $no
$var = $var - $withdrawal
exec sql UPDATE accounts SET balance = $var
         WHERE accno = $no
exec sql DISCONNECT
```

  Accessing the same data record in parallel can lead to errors
- Definition of (database) **transaction** which must be **atomic and isolated**

Introduction (6)

Race Condition:

- several parallel threads / processes use a shared resource
- state depends on order of execution
- race: threads „race“ for first / fastest access

Introduction (7)

Why avoid Race Conditions?

- results of parallel computations are well-defined (i. e. potentially false)
- when testing the program, the developer could (accidentally) always see a „correct“ execution order; but later there might occasionally appear a „false“ one.
- Race Conditions are security risks

Introduction (8)

Race Condition as security risk

- used by attackers
- simplified example:

```
read(command)
f=open("/tmp/script","w")
write(f,command)
f.close()
chmod("/tmp/script","a+x")
system("/tmp/script")
```

Attacker changes file contents before the chmod; program executes with victim's rights

Introduction (9)

- Idea: use a lock to restrict concurrent access to one process (thread, ...):

```
inc_counter( ) {
  flag=read(Lock);
  if (flag == LOCK_UNSET) {
    set(Lock);
    /*  start of „critical region“  */
    w=read(Address);  w=w+1;
    write(Address,w);
    /*  end of „critical region“  */
    release(Lock);
  };
}
```

- Problem: lock variable is not protected

Introduction (10)

- not all attempts to access data are risky:
  - concurrent reading data does not cause harm
  - „disjoint“ processes (those which share no data) can always access without protection
- Whenever several processes / threads / ... concurrently access an object – and at least one of them writes –, the overall system behavior is **unpredictable** and **irreproducible**.

Content overview: Synchronization

- 5.1 Introduction, Race Conditions
- 5.2 critical sections and mutual exclusion
- 5.3 Synchronisation methods
  - software-based synchronization
  - Standard primitives: mutexes, semaphores, events, monitors
  - locking
  - messages
- 5.4 Synchronization on Unix/Linux
  - Locking
  - Signals
  - System V IPC: Message queues, SV Semaphores, Shared memory
- 5.5 Applications
  - Mutex objects
  - Scope of synchronization

Critical regions (1)

- program section which accesses shared data
  - need not be different programs: could be several instances (processes) of the same program!
- code block from first to last access
- don't protect the code, but the data!
- terminology: „enter“ and „leave“ a critical region

Critical regions (2)

- Requirements for parallel threads:
  - No more than one thread may be inside a critical region (at the same time)
  - No thread that is outside of critical region is allowed to block another process
  - No thread should wait forever to enter a critical region
  - Deadlocks should be avoided (e.g.: two processes are inside different critical regions and block one another)

Mutual exclusion

- If there's never more than one process in the same critical region, that is called „**mutual exclusion**“ (mutex)
- It is the programmer's task to satisfy this requirement
- the operating systems offers tools to implement mutual exclusion, but it doesn't protect the software from programming faults

Software-based synchroniz. (1)

1st attempt: lock variable (is in the introduction)

- initialize lock variable to *false*
- process that wants to enter the critical region tests lock==*false* – if condition is fulfilled:
  - set lock=*true*,
  - enter (and then leave) critical region
  - (re)set lock=*false*

```
while ( lock ) {
  /* wait */
};
lock=true;
critical_region();
lock=false;
```

- this simply moves the problem from the original variable to the lock variable

Software-based synchroniz. (2)

2nt attempt: remember „next process“

- lock variable *turn* holds information: which process may enter the critical region next?

```
while (true) {
  while (turn != 1) {
    /* wait */
  };
  critical_region();
  turn=2;
}
```

```
while (true) {
  while (turn != 2) {
    /* wait */
  };
  critical_region();
  turn=1;
}
```

- avoids Race Conditions
- but: critical region can only be used in an alternating fashion

Software-based synchroniz. (3)

3rd attempt: for each thread a separate variable that says: „thread is in crit. region“

```
while (true) {
  C1=true;
  while (C2) {
    /* wait */
  };
  kritischer_bereich();
  C1=false;
}
```

```
while (true) {
  C2=true;
  while (C1) {
    /* wait */
  };
  kritischer_bereich();
  C2=false;
}
```

- avoids Race Conditions
- Deadlocks happen when both processes want to enter the critical region simultaneously

Software-based synchroniz. (4)

4th attempt (Dekker): combination of lock variables and alternation

```
while (true) {
  C1=true;
  while (C2) {
    if (turn != 1) {
      C1=false;
      while (turn != 1) {
        /* wait */
      };
      C1=true;
    };
  };
  critical_region();
  turn=2;
  C1=false;
}
```

```
while (true) {
  C2=true;
  while (C1) {
    if (turn != 2) {
      C2=false;
      while (turn != 2) {
        /* wait */
      };
      C2=true;
    };
  };
  critical_region();
  turn=1;
  C2=false;
}
```


Software-based synchroniz. (5)

Alternative: Peterson's algorithm

```
C1=true;
turn=2;
while (C2 && turn==2)
  /* warten */;
critical_region();
C1=false;
```

```
C2=true;
turn=1;
while (C1 && turn==1)
  /* warten */;
critical_region();
C2=false;
```

Software-based synchroniz. (6)

Peterson's Algorithmus –
guarantees mutual exclusion:

- when P_1 sets C_1 to *true*, P_2 can no longer enter its critical region
- If P_2 already was in the critical region then C_2 was already *true*, i.e., P_1 was not allowed to enter its critical region

Software-based synchroniz. (7)

Peterson's Algorithmus –
no mutual blocking:

Assume that P1 is blocked in the while loop, i.e.
C2=true and turn=2 (P1 can enter the critical region when one of the conditions no longer holds, i.e. either C2=false or turn=1)

then only two possibilities:

- P2 waits for entering its critical region -> this cannot be the case since with turn=2 it can enter immediately
- P2 goes through its critical region repeatedly, monopolizing access to it -> this cannot be either since P2 sets turn=1 before entering (thus letting P1 try before)