

```

Sep 19 14:20:18 amd64 sshd[20494]: Accepted publickey for esser from ::ffff:87.234.201.207 port 61557
Sep 19 14:27:41 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 01:00:01 amd64 /usr/sbin/cron[6232]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 20 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 02:00:01 amd64 /usr/sbin/cron[6232]: (root) CMD (/sbin/evlogmgr -c "age > *30d*")
Sep 20 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 12:46:44 amd64 sshd[62004]: Accepted publickey for esser from ::ffff:87.234.201.207 port 62004
Sep 20 12:46:44 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 12:48:44 amd64 sshd[62105]: Accepted publickey for esser from ::ffff:87.234.201.207 port 62105
Sep 20 12:54:44 amd64 sshd[62514]: Accepted publickey for esser from ::ffff:87.234.201.207 port 62514
Sep 20 15:27:33 amd64 sshd[64242]: Accepted publickey for esser from ::ffff:87.234.201.207 port 64242
Sep 20 16:15:18 amd64 sshd[63375]: Accepted publickey for esser from ::ffff:87.234.201.207 port 63375
Sep 20 16:15:18 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 16:15:18 amd64 /usr/sbin/cron[63546]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 20 16:15:18 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 16:15:18 amd64 /usr/sbin/cron[63546]: (root) CMD (/sbin/evlogmgr -c "age > *30d*")
Sep 20 16:15:18 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 16:15:18 amd64 /usr/sbin/cron[63397]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 20 16:15:18 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 16:15:18 amd64 /usr/sbin/cron[64391]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 20 16:15:18 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 16:15:18 amd64 /usr/sbin/cron[54991]: (root) CMD (/sbin/evlogmgr -c "age > *30d*")
Sep 20 16:15:18 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 20 16:15:18 amd64 /usr/sbin/cron[6232]: (root) CMD (/sbin/evlogmgr -c "age > *30d*")
Sep 20 16:15:18 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 02:00:01 amd64 /usr/sbin/cron[6232]: (root) CMD (/sbin/evlogmgr -c "age > *30d*")
Sep 23 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 02:00:01 amd64 sshd[59771]: Accepted publickey for esser from ::ffff:192.168.1.5 port 59771 ssh2
Sep 23 18:04:05 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 23 18:04:34 amd64 sshd[62093]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62093
Sep 24 01:00:01 amd64 /usr/sbin/cron[12436]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 24 01:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 02:00:01 amd64 /usr/sbin/cron[12553]: (root) CMD (/sbin/evlogmgr -c "age > *30d*")
Sep 24 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 11:15:48 amd64 sshd[64456]: Accepted publickey for esser from ::ffff:87.234.201.207 port 64456
Sep 24 11:15:48 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 13:49:08 amd64 sshd[61330]: Accepted publickey for esser from ::ffff:87.234.201.207 port 61330
Sep 24 13:49:08 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 15:42:07 amd64 kernel: snd_seq_midi_event: unsupported module, tainting kernel.
Sep 24 15:42:07 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 24 15:42:07 amd64 kernel: snd_seq_oss: unsupported module, tainting kernel.
Sep 24 20:25:31 amd64 sshd[62566]: Accepted publickey for esser from ::ffff:87.234.201.207 port 62566
Sep 24 20:25:31 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 01:00:02 amd64 /usr/sbin/cron[6621]: (root) CMD (/sbin/evlogmgr -c "severity=DEBUG")
Sep 25 01:00:02 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 02:00:01 amd64 /usr/sbin/cron[1484]: (root) CMD (/sbin/evlogmgr -c "age > *30d*")
Sep 25 02:00:01 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 10:59:25 amd64 sshd[64183]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64183
Sep 25 10:59:25 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 10:59:47 amd64 sshd[64253]: Accepted rsa for esser from ::ffff:87.234.201.207 port 64253
Sep 25 11:30:02 amd64 sshd[62029]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62029
Sep 25 11:59:25 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 14:05:37 amd64 sshd[11554]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62822
Sep 25 14:05:37 amd64 syslog-ng[7653]: STATS: dropped 0
Sep 25 14:06:10 amd64 sshd[11586]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62951
Sep 25 14:07:17 amd64 sshd[11608]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63392
Sep 25 14:08:33 amd64 sshd[11630]: Accepted rsa for esser from ::ffff:87.234.201.207 port 63709
Sep 25 15:25:33 amd64 sshd[12930]: Accepted rsa for esser from ::ffff:87.234.201.207 port 62778

```



# 5 Synchronization (1)

- 5. Synchronization
- 5.1 Introduction
- 5.2 Vocabulary
- 5.3 Synchroniz. methods

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

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

```

result after P1, P2: w=11 – not 12!

## 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 (3)

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

<pre> Initial situation: w=10  P1: w=read(Addr); // 10     w=w+1; // 11     write(Addr,w); // 11  P2: w=read(Addr); // 10     w=w+1; // 11     write(Addr,w); // 11 </pre>	<pre> Initial situation: w=10  P1: w=read(Addr); // 11     w=w+1; // 12     write(Addr,w); // 12  P2: w=read(Addr); // 11     w=w+1; // 12     write(Addr,w); // 12 </pre>
--	--

result after P1, P2: 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 data (mutual exclusion)

## 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 (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 (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 (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.**

## 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

## 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

## 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

## 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)

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

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

## Software-based synchroniz. (2)

2nd 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. (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. (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. (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  $P_1$  is blocked in the while loop, i.e.  
 $C_2 = \text{true}$  and  $\text{turn} = 2$  ( $P_1$  can enter the critical region when one of the conditions no longer holds, i.e. either  $C_2 = \text{false}$  or  $\text{turn} = 1$ )

then only two possibilities:

- $P_2$  waits for entering its critical region -> this cannot be the case since with  $\text{turn} = 2$  it can enter immediately
- $P_2$  goes through its critical region repeatedly, monopolizing access to it -> this cannot be either since  $P_2$  sets  $\text{turn} = 1$  before entering (thus letting  $P_1$  try before)