Betriebssysteme

9. Tutorium - IPC

Peter Bohner

9. Januar 2024

ITEC - Operating Systems Group

Begrüßung

- Frohes Neues!

Begrüßung

- Frohes Neues!
- Neujahresvorsatz: OS-Übungsblätter bearbeiten

Begrüßung

- Frohes Neues!
- Neujahresvorsatz: OS-Übungsblätter bearbeiten
- `<rant>` Wenn man vorher explizit sagt das man kommt, hat man das auch zu tun, oder sich abzumelden! `</rant>`

Inter-Process-Communication

IPC

How can different processes interact?

IPC

How can different processes interact?

- Shared memory (implicitly

IPC

How can different processes interact?

- Shared memory (implicitly as they share an address space and explicitly via a shared memory area)

IPC

How can different processes interact?

- Shared memory (implicitly as they share an address space and explicitly via a shared memory area)
- OS facilities (e.g. messages, pipes, Signals, sockets)

IPC

How can different processes interact?

- Shared memory (implicitly as they share an address space and explicitly via a shared memory area)
- OS facilities (e.g. messages, pipes, Signals, sockets)
- High level abstractions (files, database entries)

IPC - Getting Stuck

What mechanism do you need for IPC in an imperfect world?

IPC - Getting Stuck

What mechanism do you need for IPC in an imperfect world?

Timeouts! You don't want to wait for buggy programs or poor dead ones :(

IPC - Buffering

Asynchronous `send` Operations require a buffer. Where do you put that?

- Actually, *why do they need a buffer*?

IPC - Buffering

Asynchronous `send` Operations require a buffer. Where do you put that?

- Actually, *why do they need a buffer*? Sent but not yet received messages

IPC - Buffering

Asynchronous `send` Operations require a buffer. Where do you put that?

- Actually, *why do they need a buffer*? Sent but not yet received messages
- They can be in the *receiver's* address space, the *sender's* AS or in the kernel

IPC - Buffering

Asynchronous `send` Operations require a buffer. Where do you put that?

- Actually, *why do they need a buffer*? Sent but not yet received messages
- They can be in the *receiver's* address space, the *sender's* AS or in the kernel

Buffering in the Receiver's Address Space

- Is that a good idea?

IPC - Buffering

Asynchronous `send` Operations require a buffer. Where do you put that?

- Actually, *why do they need a buffer*? Sent but not yet received messages
- They can be in the *receiver's* address space, the *sender's* AS or in the kernel

Buffering in the Receiver's Address Space

- Is that a good idea? What happens when you have many clients?

IPC - Buffering

Asynchronous `send` Operations require a buffer. Where do you put that?

- Actually, *why do they need a buffer*? Sent but not yet received messages
- They can be in the *receiver's* address space, the *sender's* AS or in the kernel

Buffering in the Receiver's Address Space

- Is that a good idea? What happens when you have many clients?

$\Rightarrow$ Does not scale well. You either allow every sender to allocate memory for you (*ouch*) or you might run out with many clients

IPC - Buffering

Buffering in the Kernel

- Is that a good idea?

IPC - Buffering

Buffering in the Kernel

- Is that a good idea? What happens when you have many clients?

IPC - Buffering

Buffering in the Kernel

- Is that a good idea? What happens when you have many clients?

$\Rightarrow$ Does not scale well. You either allow every sender to allocate kernel memory (***ouch***) or you might run out with many clients

IPC - Buffering

Buffering in the Kernel

- Is that a good idea? What happens when you have many clients?

$\Rightarrow$ Does not scale well. You either allow every sender to allocate kernel memory (***ouch***) or you might run out with many clients

Buffering in the Sender's Address Space

- Sounds good?

IPC - Buffering

Buffering in the Kernel

- Is that a good idea? What happens when you have many clients?

$\Rightarrow$ Does not scale well. You either allow every sender to allocate kernel memory (***ouch***) or you might run out with many clients

Buffering in the Sender's Address Space

- Sounds good? Yea, we are running out of options.

IPC - Buffering

Buffering in the Kernel

- Is that a good idea? What happens when you have many clients?

$\Rightarrow$ Does not scale well. You either allow every sender to allocate kernel memory (***ouch***) or you might run out with many clients

Buffering in the Sender's Address Space

- Sounds good? Yea, we are running out of options. But it's mostly alright!

IPC - Buffering

Buffering in the Kernel

- Is that a good idea? What happens when you have many clients?

⇒ Does not scale well. You either allow every sender to allocate kernel memory (***ouch***) or you might run out with many clients

Buffering in the Sender's Address Space

- Sounds good? Yea, we are running out of options. But it's mostly alright!

\+ We only need to store it once: The sender has it in a buffer somewhere anyways

IPC - Buffering

Buffering in the Kernel

- Is that a good idea? What happens when you have many clients?

⇒ Does not scale well. You either allow every sender to allocate kernel memory (***ouch***) or you might run out with many clients

Buffering in the Sender's Address Space

- Sounds good? Yea, we are running out of options. But it's mostly alright!

\+ We only need to store it once: The sender has it in a buffer somewhere anyways

\+ Scales better, as each sender keeps their messages

± We need to tell the client when it can reclaim the buffer

The Joys of Synchronous IPC

You are a very popular process that receives and handles many messages.

- For simplicities sake you chose *Synchronous IPC*. What problem might occur with many clients?

The Joys of Synchronous IPC

You are a very popular process that receives and handles many messages.

- For simplicities sake you chose *Synchronous IPC*. What problem might occur with many clients?
- You spent your whole life *waiting for timeouts to expire*

The Joys of Synchronous IPC

You are a very popular process that receives and handles many messages.

- For simplicities sake you chose *Synchronous IPC*. What problem might occur with many clients?
- You spent your whole life *waiting for timeouts to expire*
- How could you solve that with a new syscall?

The Joys of Synchronous IPC

You are a very popular process that receives and handles many messages.

- For simplicities sake you chose *Synchronous IPC*. What problem might occur with many clients?
- You spent your whole life *waiting for timeouts to expire*
- How could you solve that with a new syscall? How does `send-and-receive`, which sends and instantly receives help?

The Joys of Synchronous IPC

You are a very popular process that receives and handles many messages.

- For simplicities sake you chose *Synchronous IPC*. What problem might occur with many clients?
- You spent your whole life *waiting for timeouts to expire*
- How could you solve that with a new syscall? How does `send-and-receive`, which sends and instantly receives help?
- The server can assume you are using it and set a zero timeout. After all, if you are using that syscall you *will* be waiting

Emulating Async IPC

How can you emulate asynchronous IPC using synchronous IPC?

Emulating Async IPC

How can you emulate asynchronous IPC using synchronous IPC?

- Use a Proxy thread

Emulating Async IPC

How can you emulate asynchronous IPC using synchronous IPC?

- Use a Proxy thread
  1. Copy message to proxy thread
  2. Proxy threads sends synchronously and might block until recipient calls receive

Emulating Async IPC

How can you emulate asynchronous IPC using synchronous IPC?

- Use a Proxy thread
  1. Copy message to proxy thread
  2. Proxy threads sends synchronously and might block until recipient calls receive

\+ Allows async I/O

\-

Emulating Async IPC

How can you emulate asynchronous IPC using synchronous IPC?

- Use a Proxy thread
  1. Copy message to proxy thread
  2. Proxy threads sends synchronously and might block until recipient calls receive

\+ Allows async I/O

\- How many messages can you send?

Emulating Async IPC

How can you emulate asynchronous IPC using synchronous IPC?

- Use a Proxy thread
  1. Copy message to proxy thread
  2. Proxy threads sends synchronously and might block until recipient calls receive

\+ Allows async I/O

\- How many messages can you send? Yea, one per thread...

Emulating Async IPC

How can you emulate Synchronous IPC using asynchronous IPC?

Emulating Async IPC

How can you emulate Synchronous IPC using asynchronous IPC?

What about

```
1  do {
2    res = async_send(message);
3  } while(res != MESSAGE_SENT);
```

Emulating Async IPC

How can you emulate Synchronous IPC using asynchronous IPC?

What about

```
1  do {
2    res = async_send(message);
3  } while(res != MESSAGE_SENT);
```

Will wait until it was *sent*, not until it was *received*.

Emulating Async IPC

How can you emulate Synchronous IPC using asynchronous IPC?

What about

```
1  do {
2    res = async_send(message);
3  } while(res != MESSAGE_SENT);
```

Will wait until it was *sent*, not until it was *received*.

⇒ Implement a very simple protocol involving multiple messages

Emulating Async IPC

How can you emulate Synchronous IPC using asynchronous IPC?

What about

```
1  do {
2    res = async_send(message);
3  } while(res != MESSAGE_SENT);
```

Will wait until it was *sent*, not until it was *received*.

⇒ Implement a very simple protocol involving multiple messages

⇒ Send `ACK` message on receiver side, wait for `ACK` to be received.

And receive?

Emulating Async IPC

How can you emulate Synchronous IPC using asynchronous IPC?

What about

```
1  do {
2    res = async_send(message);
3  } while(res != MESSAGE_SENT);
```

Will wait until it was *sent*, not until it was *received*.

⇒ Implement a very simple protocol involving multiple messages

⇒ Send `ACK` message on receiver side, wait for `ACK` to be received.

And receive?

Just loop until `async_receive` receives a message (that is not an `ACK`)

Critical Sections

Synchronizing Properly - Core Properties

Does this work?

```
1 void foo() { // called in parallel
2   if(random() < 0.5) {
3     sleep(10);
4   }
5   a += 10;
6 }
```

Synchronizing Properly - Core Properties

Does this work?

```
1 void foo() { // called in parallel
2   if(random() < 0.5) {
3     sleep(10);
4   }
5   a += 10;
6 }
```

Nope. Multiple threads can call `a += 10` at the same time.

Property 1: Mutual Exclusion

Synchronizing Properly - Notes on Terminology

Some boring definitions

```
1  void foo() { // called in parallel
2
3    // This is the code before the critical section
4    // ==> Entry section
5
6    // Here common data is accessed (e.g. shared variable).
7    // Only one thread might be in here at a time.
8    // ==> Critical Section
9    a += 10;
10
11   // This is the code after the critical section
12   // ==> Exit section
13 }
```

Everything else is the Remainder section

Synchronizing Properly - Core Properties

Does this work?

```
1 void foo() { // called in parallel
2   // The late bird catches the worm
3   while(me != selectLastWaiting()) {
4     sleep(10);
5   }
6   a += 10;
7 }
```

Synchronizing Properly - Core Properties

Does this work?

```
1 void foo() { // called in parallel
2   // The late bird catches the worm
3   while(me != selectLastWaiting()) {
4     sleep(10);
5   }
6   a += 10;
7 }
```

How long could a thread wait?

Synchronizing Properly - Core Properties

Does this work?

```
1 void foo() { // called in parallel
2   // The late bird catches the worm
3   while(me != selectLastWaiting()) {
4     sleep(10);
5   }
6   a += 10;
7 }
```

How long could a thread wait?

Forever :(

Property 2: Bounded Waiting

Synchronizing Properly - Core Properties

Does this work?

```
1 void foo() { // called in parallel
2   while(waitingThreadCount > 1) {
3     sleep(10);
4   }
5   a += 10;
6 }
```

Synchronizing Properly - Core Properties

Does this work?

```
1 void foo() { // called in parallel
2   while(waitingThreadCount > 1) {
3     sleep(10);
4   }
5   a += 10;
6 }
```

Threads *outside* the critical section prevent threads from *entering* it

$\Rightarrow$ There's no progress!

Property 3: Progress

Synchronizing Properly - Core Properties

And the last one isn't really a property of a correct solution

Property 4: Performance

Synchronizing Properly - Core Properties

Mutual Exclusion

Only *one thread* can enter the critical section at once.

Synchronizing Properly - Core Properties

Mutual Exclusion

Only *one thread* can enter the critical section at once.

Progress

Threads in the *remainder* section do not prevent threads from *entering* the critical section.

Synchronizing Properly - Core Properties

Mutual Exclusion

Only *one thread* can enter the critical section at once.

Progress

Threads in the *remainder* section do not prevent threads from *entering* the critical section.

Bounded Waiting

There is an upper bound on *how many* different threads can enter the CS while a thread is waiting.

Important: This is not a *time* bound!

Synchronizing Properly - Core Properties

Mutual Exclusion

Only *one thread* can enter the critical section at once.

Progress

Threads in the *remainder* section do not prevent threads from *entering* the critical section.

Bounded Waiting

There is an upper bound on *how many* different threads can enter the CS while a thread is waiting.

Important: This is not a *time* bound!

Performance

The time overhead of the synchronization primitive is low (for low / medium / high contention).

Synchronizing Properly - Core Properties

Does this code fulfill all properties?

```
1  /* aligned to cache lines */
2  volatile int next = 0;
3  volatile int executing = 0;
4
5  lock_acquire() {
6    /* atomic version of "ticket = next; next += 1;" */
7    int ticket = fetch_and_add(next, 1);
8    /* busy wait until the counter matches */
9    while (ticket != executing ) {}
10  }
11
12 lock_release() {
13   executing++;
14 }
```

Synchronizing Properly - Core Properties

Does this code fulfill all properties?

- Mutual Exclusion:

Synchronizing Properly - Core Properties

Does this code fulfill all properties?

- Mutual Exclusion: Yes (though if you have 2^{32} threads waiting it doesn't)
- Bounded Waiting:

Synchronizing Properly - Core Properties

Does this code fulfill all properties?

- Mutual Exclusion: Yes (though if you have 2^{32} threads waiting it doesn't)
- Bounded Waiting: Yes. Eventually my number is the next one!
- Progress:

Synchronizing Properly - Core Properties

Does this code fulfill all properties?

- Mutual Exclusion: Yes (though if you have 2^{32} threads waiting it doesn't)
- Bounded Waiting: Yes. Eventually my number is the next one!
- Progress: Yes, threads in the remainder section do not hinder any thread from entering the critical section. Only threads in the *entry section* can do that temporarily.

Fixing The Bank

Remember the bank example?

Fixing The Bank

Remember the bank example?

```
1 current = get_balance();
2 current += delta; // delta ∈ {−50, 100}
3 set_balance(current);
```

How could you fix that?

Fixing The Bank

Remember the bank example?

```
1 current = get_balance();
2 current += delta; // delta ∈ {−50, 100}
3 set_balance(current);
```

How could you fix that?

Synchronize it!

Fixing The Bank

Remember the bank example?

```
1 current = get_balance();
2 current += delta; // delta ∈ {−50, 100}
3 set_balance(current);
```

How could you fix that?

Synchronize it!

```
1 lock(L);
2 current = get_balance();
3 current += delta; // delta ∈ {−50, 100}
4 set_balance(current);
5 unlock(L);
```

Race Conditions

The Parallel Wizard Strikes Again

```
1 current = get_balance();
2 current += delta; // delta ∈ {−50, 100}
3 set_balance(current);
```

What happens if this code is executed in *parallel* for the two values of `delta`?

The Parallel Wizard Strikes Again

```
1 current = get_balance();
2 current += delta; // delta ∈ {−50, 100}
3 set_balance(current);
```

What happens if this code is executed in *parallel* for the two values of `delta`?

Unrolled

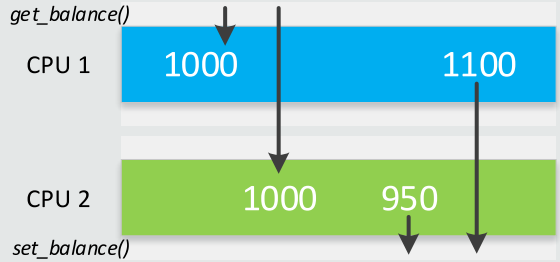
```
1 current = get_balance();
2 int tmp = current;
3 current = tmp + delta; // delta ∈ {−50, 100}
4 set_balance(current);
```

Race Conditions

Thread execution

Race Conditions

Thread execution

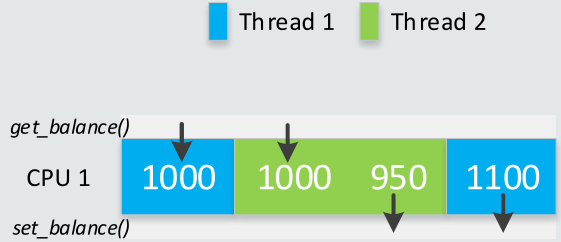


Race Conditions

And if you run it concurrently on a single-core system?

Race Conditions

And if you run it concurrently on a single-core system?

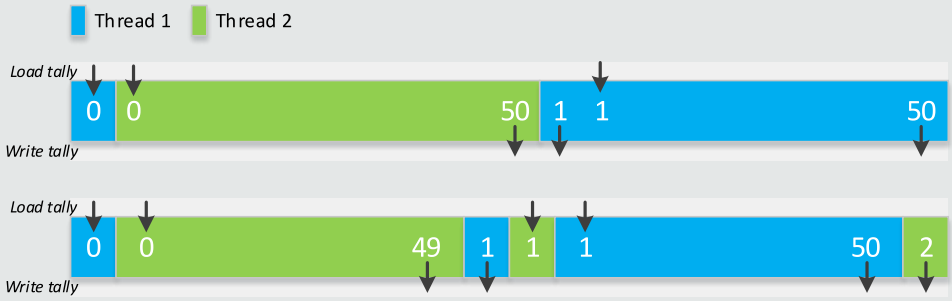


Having Fun With It...

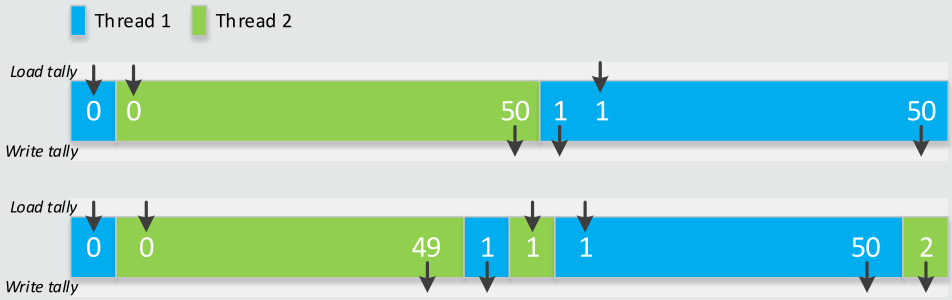
What is the value of `tally` at the end?

```
1  #include <stdio.h>
2
3  int tally;
4  void total(int N) {
5      for(int i = 0; i < N; i++)
6          tally += 1;
7  }
8
9  int main() {
10     tally = 0;
11
12     #pragma omp parallel for
13     for(int i = 0; i < 2; i++)
14         total(50);
15
16     printf("%d\n" , tally);
17     return 0;
18 }
```

This!


Thread 1
Thread 2
Load tally
0
0
50
1
1
50
Write tally
Load tally
0
0
49
1
1
1
50
2
Write tally

This!

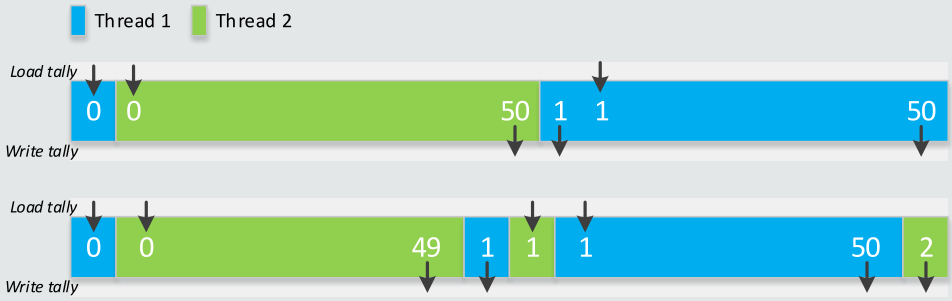


And what happens when we use $N, N > 0$ threads instead of 2?

The range is now

Having Fun With It...

This!



And what happens when we use $N, N > 0$ threads instead of 2?

The range is now $[2, 50 \cdot N]$. We can still have the same problem.

...And Adding ULTs!

What would happen if we used ULTs?

...And Adding ULTs!

What would happen if we used ULTs?

Can ULTs be pre-empted?

...And Adding ULTs!

What would happen if we used ULTs?

Can ULTs be pre-empted? With cooperatively-scheduled (the common case) no!

...And Adding ULTs!

What would happen if we used ULTs?

Can ULTs be pre-empted? With cooperatively-scheduled (the common case) no!

$\Rightarrow$ Everything will be fine, *unless* the threads `yield` *during the increment*

...And Adding ULTs!

What would happen if we used ULTs?

Can ULTs be pre-empted? With cooperatively-scheduled (the common case) no!

$\Rightarrow$ Everything will be fine, *unless* the threads `yield` *during the increment*

In that spirit: Does this work (with 1:1 threads)?

```
1 void total(int N) {
2   for(int i = 0; i < N; ++ i) {
3     tally += 1;
4     sched_yield();
5   }
6 }
```

...And Adding ULTs!

What would happen if we used ULTs?

Can ULTs be pre-empted? With cooperatively-scheduled (the common case) no!

$\Rightarrow$ Everything will be fine, *unless* the threads `yield` *during the increment*

In that spirit: Does this work (with 1:1 threads)?

```
1 void total(int N) {
2   for(int i = 0; i < N; ++ i) {
3     tally += 1;
4     sched_yield();
5   }
6 }
```

Nope. It might be less likely that it is interrupted during the increment (Why?

...And Adding ULTs!

What would happen if we used ULTs?

Can ULTs be pre-empted? With cooperatively-scheduled (the common case) no!

$\Rightarrow$ Everything will be fine, *unless* the threads `yield` *during the increment*

In that spirit: Does this work (with 1:1 threads)?

```
1 void total(int N) {
2   for(int i = 0; i < N; ++ i) {
3     tally += 1;
4     sched_yield();
5   }
6 }
```

Nope. It might be less likely that it is interrupted during the increment (Why? it never uses its timeslice), but it is still possible (e.g. due to

...And Adding ULTs!

What would happen if we used ULTs?

Can ULTs be pre-empted? With cooperatively-scheduled (the common case) no!

$\Rightarrow$ Everything will be fine, *unless* the threads `yield` *during the increment*

In that spirit: Does this work (with 1:1 threads)?

```
1 void total(int N) {
2   for(int i = 0; i < N; ++ i) {
3     tally += 1;
4     sched_yield();
5   }
6 }
```

Nope. It might be less likely that it is interrupted during the increment (Why? it never uses its timeslice), but it is still possible (e.g. due to a hardware interrupt)

Deadlock Empire

https://deadlockempire.github.io

Ende

XKCD 281 - Package Tracking

F R A G E N?

https://forms.gle/9CwJSKidKibubran9

Bis nächste Woche