Operating Systems CMPSC 473

Concurrency: Channels

November 9, 2023 – Lecture 24

Instructor: Ruslan Nikolaev

Summary

- Previous lectures
  - Data races and race conditions
  - The mutual exclusion approach
  - How to implement locks
  - Pthreads lock API
  - How to use locks
  - Liveness conditions
  - Thread safety
  - Condition variables (+ pthreads API), semaphores
  - Reader-write lock
- Next
  - Channels and Project 3 information
  - Several synchronization problems

Buffered channel

- Send/receive messages between threads

Buffered channel

- Send/receive messages between threads
- Can be used for synchronization

Buffered channel

- Send/receive messages between threads
- Can be used for synchronization
- Finite message buffer – can be full/empty

Buffered channel

- Send/receive messages between threads
- Can be used for synchronization
- Finite message buffer – can be full/empty

- Interface:

Buffered channel

- Send/receive messages between threads
- Can be used for synchronization
- Finite message buffer – can be full/empty

- Interface:
- Send() – adds message to buffer; if buffer full, block and wait

Buffered channel

- Send/receive messages between threads
- Can be used for synchronization
- Finite message buffer – can be full/empty

- Interface:
- Send() – adds message to buffer; if buffer full, block and wait
- Recv() – removes message from buffer; if buffer empty, block and wait

Channels to track resources

```
for (int i = 0; i < num_jobs; i++) {

  thread_create(run_job, &jobs[i]);
}

run_job() {
  ...

}
```

Channels to track resources

```
for (int i = 0; i < num_jobs; i++) {
  channel_send();
  thread_create(run_job, &jobs[i]);
}

run_job() {
  ...

}
```

Channels to track resources

```
for (int i = 0; i < num_jobs; i++) {
  channel_send();
  thread_create(run_job, &jobs[i]);
}

run_job() {
  ...
  channel_recv();
}
```

Channels to track resources

```
channel_init(# cpu cores);
for (int i = 0; i < num_jobs; i++) {
  channel_send();
  thread_create(run_job, &jobs[i]);
}

run_job() {
  ...
  channel_recv();
}
```

Sharing data using channels

```
average_first_n_results(n) {
  for (int i = 0; i < n; i++) {
    sum += channel_recv();
  }
  return sum / n;
}
run_job() {
  ...
  channel_send(result);
}
```

Channel interface

- channel_create
- channel_send (blocking/non-blocking)
- channel_receive (blocking/non-blocking)
- channel_close
- channel_destroy
- channel_select

Channel assignment

Things to note:

Channel assignment

Things to note:

- Basics for send/receive should be straightforward

Channel assignment

Things to note:

- Basics for send/receive should be straightforward
- Close a bit tricky to ensure threads are not stuck

Channel assignment

Things to note:

- Basics for send/receive should be straightforward
- Close a bit tricky to ensure threads are not stuck
- Select requires some thought to get correct

Channel assignment

Things to note:

- Basics for send/receive should be straightforward
- Close a bit tricky to ensure threads are not stuck
- Select requires some thought to get correct
  - Need to figure out some way to properly block

Channel assignment

Things to note:

- Basics for send/receive should be straightforward
- Close a bit tricky to ensure threads are not stuck
- Select requires some thought to get correct
  - Need to figure out some way to properly block
  - May need to modify other channel functions

Channel assignment

Things to note:

- Basics for send/receive should be straightforward
- Close a bit tricky to ensure threads are not stuck
- Select requires some thought to get correct
  - Need to figure out some way to properly block
  - May need to modify other channel functions
  - Can use extra structs to store relevant data

Channel assignment

Things to note:

- Basics for send/receive should be straightforward
- Close a bit tricky to ensure threads are not stuck
- Select requires some thought to get correct
  - Need to figure out some way to properly block
  - May need to modify other channel functions
  - Can use extra structs to store relevant data
- Correctly implementing unbuffered version with select is very hard

Channel assignment

Things to note:

- Basics for send/receive should be straightforward
- Close a bit tricky to ensure threads are not stuck
- Select requires some thought to get correct
  - Need to figure out some way to properly block
  - May need to modify other channel functions
  - Can use extra structs to store relevant data
- Correctly implementing unbuffered version with select is very hard
  - Small amount of extra credit (done individually)

Channel assignment

Things to note:

- Basics for send/receive should be straightforward
- Close a bit tricky to ensure threads are not stuck
- Select requires some thought to get correct
  - Need to figure out some way to properly block
  - May need to modify other channel functions
  - Can use extra structs to store relevant data
- Correctly implementing unbuffered version with select is very hard
  - Small amount of extra credit (done individually)
- Tests are not exhaustive – we may release more

channel_create(size_t size)

channel_create(size_t size)

- **size**: in terms of messages, not bytes
- 1 send = 1 message

channel_create(size_t size)

- **size**: in terms of messages, not bytes
- 1 send = 1 message

Purpose: create a channel “object”

- channel = chan_t struct

channel_create(size_t size)

- **size**: in terms of messages, not bytes
- 1 send = 1 message

Purpose: create a channel "object"

- channel = chan_t struct

```
typedef struct {
    // DO NOT REMOVE buffer (OR CHANGE ITS NAME) FROM THE STRUCT
    // YOU MUST USE buffer TO STORE YOUR BUFFERED CHANNEL MESSAGES
    buffer_t* buffer;

    /* ADD ANY STRUCT ENTRIES YOU NEED HERE */

} chan_t;
```

channel_create(size_t size)

- **size**: in terms of messages, not bytes
- 1 send = 1 message

malloc

Purpose: create a channel "object"

- channel = chan_t struct

```
typedef struct {
    // DO NOT REMOVE buffer (OR CHANGE ITS NAME) FROM THE STRUCT
    // YOU MUST USE buffer TO STORE YOUR BUFFERED CHANNEL MESSAGES
    buffer_t* buffer;

    /* ADD ANY STRUCT ENTRIES YOU NEED HERE */

} chan_t;
```

channel_create(size_t size)

- **size**: in terms of messages, not bytes
- 1 send = 1 message

malloc and initialize channel_t data

Purpose: create a channel "object"

- channel = chan_t struct

```
typedef struct {
    // DO NOT REMOVE buffer (OR CHANGE ITS NAME) FROM THE STRUCT
    // YOU MUST USE buffer TO STORE YOUR BUFFERED CHANNEL MESSAGES
    buffer_t* buffer;

    /* ADD ANY STRUCT ENTRIES YOU NEED HERE */

} chan_t;
```

channel_create(size_t size)

- **size**: in terms of messages, not bytes
- 1 send = 1 message

malloc and initialize channel_t data

Purpose: create a channel "object"

- channel = chan_t struct

```
typedef struct {
    // DO NOT REMOVE buffer (OR CHANGE ITS NAME) FROM THE STRUCT
    // YOU MUST USE buffer TO STORE YOUR BUFFERED CHANNEL MESSAGES
    buffer_t* buffer;

    /* ADD ANY STRUCT ENTRIES YOU NEED HERE */

} chan_t;
```

There's a buffer_create function

channel_create(size_t size)

- **size**: in terms of messages, not bytes
- 1 send = 1 message

malloc and initialize channel_t data

Purpose: create a channel "object"

- channel = chan_t struct

```
typedef struct {
    // DO NOT REMOVE buffer (OR CHANGE ITS NAME) FROM THE STRUCT
    // YOU MUST USE buffer TO STORE YOUR BUFFERED CHANNEL MESSAGES
    buffer_t* buffer;

    /* ADD ANY STRUCT ENTRIES YOU NEED HERE */
    pthread_mutex_t mutex;
} chan_t;
```

There's a buffer_create function

channel_create(size_t size)

- **size**: in terms of messages, not bytes
- 1 send = 1 message

malloc and initialize channel_t data

Purpose: create a channel "object"

- channel = chan_t struct

```
typedef struct {
    // DO NOT REMOVE buffer (OR CHANGE ITS NAME) FROM THE STRUCT
    // YOU MUST USE buffer TO STORE YOUR BUFFERED CHANNEL MESSAGES
    buffer_t* buffer;

    /* ADD ANY STRUCT ENTRIES YOU NEED HERE */
    pthread_mutex_t mutex;
} chan_t;
```

There's a buffer_create function

Use pthread_mutex_init

channel_send(chan_t* channel, void* data)

channel_send(chan_t* channel, void* data)

- **channel**: the channel to perform the operation
- **data**: a single message to send (no size)

channel_send(chan_t* channel, void* data)

- **channel**: the channel to perform the operation
- **data**: a single message to send (no size)

NULL is a valid message

channel_send(chan_t* channel, void* data)

- **channel**: the channel to perform the operation
- **data**: a single message to send (no size)

NULL is a valid message

Channel = finite buffer to add/remove data

Purpose: add one message to buffer

channel_send(chan_t* channel, void* data)

- **channel**: the channel to perform the operation
- **data**: a single message to send (no size)

NULL is a valid message

Channel = finite buffer to add/remove data

Purpose: add one message to buffer

Look at buffer_add

channel_send(chan_t* channel, void* data)

- **channel**: the channel to perform the operation
- **data**: a single message to send (no size)

NULL is a valid message

Channel = finite buffer to add/remove data

Purpose: add one message to buffer

Look at buffer_add

Concerns:

channel_send(chan_t* channel, void* data)

- **channel**: the channel to perform the operation
- **data**: a single message to send (no size)

NULL is a valid message

Channel = finite buffer to add/remove data

Purpose: add one message to buffer

Look at buffer_add

Concerns:

- Many threads call channel_send → need mutex

channel_send(chan_t* channel, void* data)

- **channel**: the channel to perform the operation
- **data**: a single message to send (no size)

NULL is a valid message

Channel = finite buffer to add/remove data

Purpose: add one message to buffer

Look at buffer_add

Concerns:

- Many threads call channel_send → need mutex
- Buffer is full → need to wait
  (condition variable or semaphore)

channel_receive(chan_t* channel, void** data)

channel_receive(chan_t* channel, void** data)

- **channel**: the channel to perform the operation
- **data**: pointer to return a single message

channel_receive(chan_t* channel, void** data)

- **channel**: the channel to perform the operation
- **data**: pointer to return a single message

Why a double pointer?

channel_receive(chan_t* channel, void** data)

- **channel**: the channel to perform the operation
- **data**: pointer to return a single message

Why a double pointer?

Channel = finite buffer to add/remove data

Purpose: remove one message from buffer

channel_receive(chan_t* channel, void** data)

- **channel**: the channel to perform the operation
- **data**: pointer to return a single message

Why a double pointer?

Channel = finite buffer to add/remove data

Purpose: remove one message from buffer

Look at buffer_remove

channel_receive(chan_t* channel, void** data)

- **channel**: the channel to perform the operation
- **data**: pointer to return a single message

Why a double pointer?

Channel = finite buffer to add/remove data

Purpose: remove one message from buffer

Look at buffer_remove

Concerns:

channel_receive(chan_t* channel, void** data)

- **channel**: the channel to perform the operation
- **data**: pointer to return a single message

Why a double pointer?

Channel = finite buffer to add/remove data

Purpose: remove one message from buffer

Look at buffer_remove

Concerns:

- Many threads call channel_recieve → same mutex

channel_receive(chan_t* channel, void** data)

- **channel**: the channel to perform the operation
- **data**: pointer to return a single message

Why a double pointer?

Channel = finite buffer to add/remove data

Purpose: remove one message from buffer

Look at buffer_remove

Concerns:

- Many threads call channel_recieve → same mutex
- Buffer is empty → need to wait
  (different condition variable or semaphore)

Question

Q: Suppose you have a variable void* result that you want to return in a parameter void data. What do you do?**

A. data = result;

B. *data = result;

C. **data = result;

D. data = &result;

E. *data = &result;

Question

Q: Suppose you have a variable void* result that you want to return in a parameter void data. What do you do?**

A. data = result;

B. *data = result;

C. **data = result;

D. data = &result;

E. *data = &result;

Blocking vs non-blocking

- Blocking = wait
- Non-blocking = no waiting, just return WOULDBLOCK

Blocking vs non-blocking

- Blocking = wait
- Non-blocking = no waiting, just return WOULDBLOCK

Q: Do I need to lock non-blocking send/receive?

Blocking vs non-blocking

- Blocking = wait
- Non-blocking = no waiting, just return WOULDBLOCK

Q: Do I need to lock non-blocking send/receive?

Purpose of lock: ensure atomic data access

Blocking vs non-blocking

- Blocking = wait
- Non-blocking = no waiting, just return WOULDBLOCK

Make operations seem uninterruptible

Q: Do I need to lock non-blocking send/receive?

Purpose of lock: ensure atomic data access

Blocking vs non-blocking

- Blocking = wait
- Non-blocking = no waiting, just return WOULDBLOCK

Make operations seem uninterruptible

Q: Do I need to lock non-blocking send/receive?

Purpose of lock: ensure atomic data access

Only works if everyone accessing data uses lock

Blocking vs non-blocking

- Blocking = wait
- Non-blocking = no waiting, just return WOULDBLOCK

Make operations seem uninterruptible

Q: Do I need to lock non-blocking send/receive?

Purpose of lock: ensure atomic data access

Only works if everyone accessing data uses lock

Q: Do these functions access shared data?

Blocking vs non-blocking

- Blocking = wait
- Non-blocking = no waiting, just return WOULDBLOCK

Make operations seem uninterruptible

Q: Do I need to lock non-blocking send/receive?

Purpose of lock: ensure atomic data access

Only works if everyone accessing data uses lock

Q: Do these functions access shared data?

A: Yes, they read/modify the buffer → lock needed

channel_close(chan_t* channel)

channel_close(chan_t* channel)

Purpose: stop current/future operations

channel_close(chan_t* channel)

Purpose: stop current/future operations

Concerns:

channel_close(chan_t* channel)

Purpose: stop current/future operations

Concerns:

- Current threads need to exit send/receive/select

channel_close(chan_t* channel)

Purpose: stop current/future operations

Concerns:

- Current threads need to exit send/receive/select
  - Should return CLOSED_ERROR

channel_close(chan_t* channel)

Purpose: stop current/future operations

Concerns:

- Current threads need to exit send/receive/select
  - Should return CLOSED_ERROR
- Future threads should return CLOSED_ERROR

channel_close(chan_t* channel)

Purpose: stop current/future operations

Concerns:

- Current threads need to exit send/receive/select
  - Should return CLOSED_ERROR
- Future threads should return CLOSED_ERROR
  - Add a closed/open flag in chan_t

channel_close(chan_t* channel)

Purpose: stop current/future operations

Concerns:

- Current threads need to exit send/receive/select
  - Should return CLOSED_ERROR
- Future threads should return CLOSED_ERROR
  - Add a closed/open flag in chan_t
  - Need to modify other functions

channel_close(chan_t* channel)

Purpose: stop current/future operations

Concerns:

- Current threads need to exit send/receive/select
  - Should return CLOSED_ERROR
- Future threads should return CLOSED_ERROR
  - Add a closed/open flag in chan_t
  - Need to modify other functions
- What if close is called at the same time as send?

channel_close(chan_t* channel)

Purpose: stop current/future operations

Concerns:

- Current threads need to exit send/receive/select
  - Should return CLOSED_ERROR
- Future threads should return CLOSED_ERROR
  - Add a closed/open flag in chan_t
  - Need to modify other functions
- What if close is called at the same time as send?
  - How do you make things atomic?

channel_close(chan_t* channel)

```
channel_send(chan_t* channel, void* data) {
 if (channel->closed) {
  return CLOSED_ERROR;
 }

 pthread_mutex_lock(&channel->mutex);
 ...
}



channel_close(chan_t* channel) {
 channel->closed = 1;
 return SUCCESS;
}
```

channel_close(chan_t* channel)

```
channel_send(chan_t* channel, void* data) {
 if (channel->closed) {
  return CLOSED_ERROR;
 }

 pthread_mutex_lock(&channel->mutex);
 ...
}
```

Q: What about current waiting threads?

```
channel_close(chan_t* channel) {
 channel->closed = 1;
 return SUCCESS;
}
```

channel_close(chan_t* channel)

```
channel_send(chan_t* channel, void* data) {
 if (channel->closed) {
  return CLOSED_ERROR;
 }

 pthread_mutex_lock(&channel->mutex);
 ...
}
```

Q: What about current waiting threads?

```
channel_close(chan_t* channel) {
 channel->closed = 1;
 return SUCCESS;
}
```

A: Need to wakeup

channel_close(chan_t* channel)

```
channel_send(chan_t* channel, void* data) {
 if (channel->closed) {
  return CLOSED_ERROR;
 }

 pthread_mutex_lock(&channel->mutex);
 ...
}
```

Q: What about current waiting threads?

```
channel_close(chan_t* channel) {
 channel->closed = 1;
 return SUCCESS;
}
```

A: Need to wakeup

Q: Signal or broadcast?

channel_close(chan_t* channel)

```
channel_send(chan_t* channel, void* data) {
 if (channel->closed) {
  return CLOSED_ERROR;
 }

 pthread_mutex_lock(&channel->mutex);
 ...
}
```

Q: What about current waiting threads?

```
channel_close(chan_t* channel) {
 channel->closed = 1;
 return SUCCESS;
}
```

A: Need to wakeup

Q: Signal or broadcast?

A: Think about it

channel_close(chan_t* channel)

```
channel_send(chan_t* channel, void* data) {
 if (channel->closed) {
  return CLOSED_ERROR;
 }
```

Q: What if channel_close called here?

```
 pthread_mutex_lock(&channel->mutex);
 ...
}
```

Q: What about current waiting threads?

```
channel_close(chan_t* channel) {
 channel->closed = 1;
 return SUCCESS;
}
```

A: Need to wakeup

Q: Signal or broadcast?

A: Think about it

channel_close(chan_t* channel)

```
channel_send(chan_t* channel, void* data) {
 if (channel->closed) {
  return CLOSED_ERROR;
 }

 pthread_mutex_lock(&channel->mutex);
 ...
}
```

Q: What if channel_close called here?

A: Race condition

Q: What about current waiting threads?

```
channel_close(chan_t* channel) {
 channel->closed = 1;
 return SUCCESS;
}
```

A: Need to wakeup

Q: Signal or broadcast?

A: Think about it

channel_destroy(chan_t* channel)

channel_destroy(chan_t* channel)

Purpose: free memory and destroy objects

channel_destroy(chan_t* channel)

Purpose: free memory and destroy objects

(e.g., pthread_mutex_destroy)

channel_destroy(chan_t* channel)

Purpose: free memory and destroy objects

(e.g., pthread_mutex_destroy)

Rule of thumb: destroy = opposite of create

- malloc ←→ free
- buffer_create ←→ buffer_free
- pthread_mutex_init ←→ pthread_mutex_destroy

channel_destroy(chan_t* channel)

Purpose: free memory and destroy objects

(e.g., pthread_mutex_destroy)

Rule of thumb: destroy = opposite of create

- malloc ←→ free
- buffer_create ←→ buffer_free
- pthread_mutex_init ←→ pthread_mutex_destroy

Remarks:

channel_destroy(chan_t* channel)

Purpose: free memory and destroy objects

(e.g., pthread_mutex_destroy)

Rule of thumb: destroy = opposite of create

- malloc ←→ free
- buffer_create ←→ buffer_free
- pthread_mutex_init ←→ pthread_mutex_destroy

Remarks:

- Do not call sem_close. Use sem_destroy.

channel_destroy(chan_t* channel)

Purpose: free memory and destroy objects

(e.g., pthread_mutex_destroy)

Rule of thumb: destroy = opposite of create

- malloc ←→ free
- buffer_create ←→ buffer_free
- pthread_mutex_init ←→ pthread_mutex_destroy

Remarks:

- Do not call sem_close. Use sem_destroy.
- Q: Why channel_destroy/channel_close separate?

channel_destroy(chan_t* channel)

Purpose: free memory and destroy objects

(e.g., pthread_mutex_destroy)

Rule of thumb: destroy = opposite of create

- malloc ←→ free
- buffer_create ←→ buffer_free
- pthread_mutex_init ←→ pthread_mutex_destroy

Remarks:

- Do not call sem_close. Use sem_destroy.
- Q: Why channel_destroy/channel_close separate?
- A: caller ensures no threads using channel in destroy

channel_select(size_t channel_count, select_t* channel_list, size_t* selected_index)

- **channel_list**: array of channels/operations
- **channel_count**: # of elements in channel_list
- **selected_index**: output index of operation

channel_select(size_t channel_count, select_t* channel_list, size_t* selected_index)

- **channel_list**: array of channels/operations
- **channel_count**: # of elements in channel_list
- **selected_index**: output index of operation

Purpose: perform exactly 1 of the operations

channel_select(size_t channel_count, select_t* channel_list, size_t* selected_index)

- **channel_list**: array of channels/operations
- **channel_count**: # of elements in channel_list
- **selected_index**: output index of operation

Purpose: perform exactly 1 of the operations

- Must perform the operation, not just choose

channel_select(size_t channel_count, select_t* channel_list, size_t* selected_index)

- **channel_list**: array of channels/operations
- **channel_count**: # of elements in channel_list
- **selected_index**: output index of operation

Purpose: perform exactly 1 of the operations

- Must perform the operation, not just choose
- channel_list is an array – arrays are pointers in C

channel_select(size_t channel_count, select_t* channel_list, size_t* selected_index)

- **channel_list**: array of channels/operations
- **channel_count**: # of elements in channel_list
- **selected_index**: output index of operation

Purpose: perform exactly 1 of the operations

- Must perform the operation, not just choose
- channel_list is an array – arrays are pointers in C
  - channel_list[index] is the typical way to access

channel_select(size_t channel_count, select_t* channel_list, size_t* selected_index)

- **channel_list**: array of channels/operations
- **channel_count**: # of elements in channel_list
- **selected_index**: output index of operation

Purpose: perform exactly 1 of the operations

- Must perform the operation, not just choose
- channel_list is an array – arrays are pointers in C
  - channel_list[index] is the typical way to access
- Must indicate chosen operation

channel_select(size_t channel_count, select_t* channel_list, size_t* selected_index)

- **channel_list**: array of channels/operations
- **channel_count**: # of elements in channel_list
- **selected_index**: output index of operation

Purpose: perform exactly 1 of the operations

- Must perform the operation, not just choose
- channel_list is an array – arrays are pointers in C
  - channel_list[index] is the typical way to access
- Must indicate chosen operation
  - *selected_index = chosen_index;

channel_select(size_t channel_count, select_t* channel_list, size_t* selected_index)

- **channel_list**: array of channels/operations
- **channel_count**: # of elements in channel_list
- **selected_index**: output index of operation

Purpose: perform exactly 1 of the operations

- Must perform the operation, not just choose
- channel_list is an array – arrays are pointers in C
  - channel_list[index] is the typical way to access
- Must indicate chosen operation
  - *selected_index = chosen_index;
- If chose receive, then store data in
  channel_list[chosen_index].data

Channel Select

Channel Select

Suppose:

select { recv channelA, send channelB }

Channel Select

Suppose:

select { recv channelA, send channelB }

select: Perform exactly 1 of the operations

Channel Select

Suppose:

select { recv channelA, send channelB }

select: Perform exactly 1 of the operations

What if:

channelA has data?

Channel Select

Suppose:

select { recv channelA, send channelB }

select: Perform exactly 1 of the operations

What if:

channelA has data? **recv from channelA**

Channel Select

Suppose:

select { recv channelA, send channelB }

select: Perform exactly 1 of the operations

What if:

channelA has data? **recv from channelA**

channelA empty, channelB not full?

Channel Select

Suppose:

select { recv channelA, send channelB }

select: Perform exactly 1 of the operations

What if:

channelA has data? **recv from channelA**

channelA empty, channelB not full? **send on channelB**

Channel Select

Suppose:

select { recv channelA, send channelB }

select: Perform exactly 1 of the operations

What if:

channelA has data? **recv from channelA**

channelA empty, channelB not full? **send on channelB**

channelA empty, channelB full?

Channel Select

Suppose:

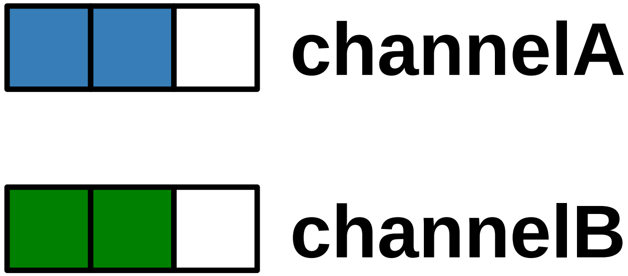
select { recv channelA, send channelB }

select: Perform exactly 1 of the operations

What if:

channelA has data?	**recv from channelA**
channelA empty, channelB not full?	**send on channelB**
channelA empty, channelB full?	**wait**

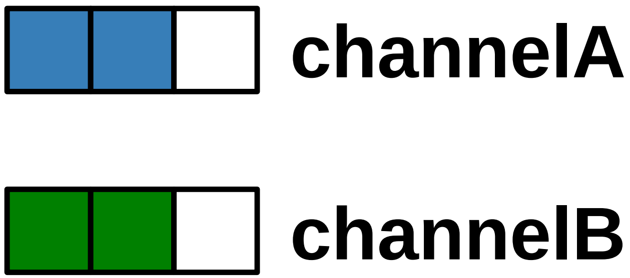
Select example



Select example

Thread1

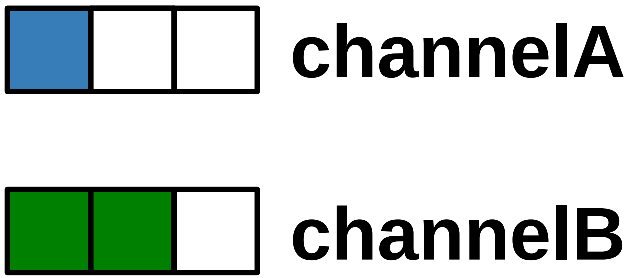
select { recv channelA, send channelB }



Select example

Thread1

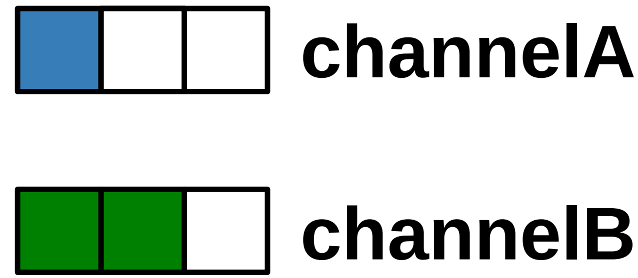
select { recv channelA, send channelB }



Select example

<u>Thread1</u>

select { recv channelA, send channelB }
select { recv channelA, send channelB }

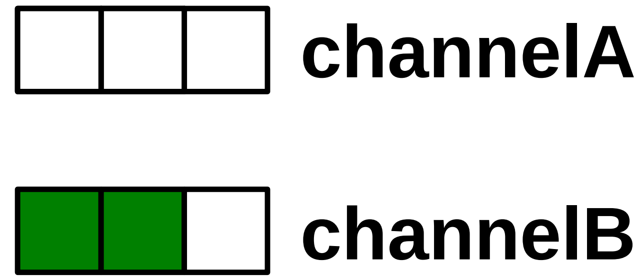


Select example

<u>Thread1</u>

select { recv channelA, send channelB }

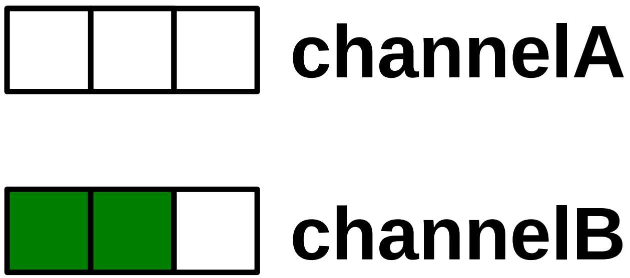
select { recv channelA, send channelB }



Select example

Thread1

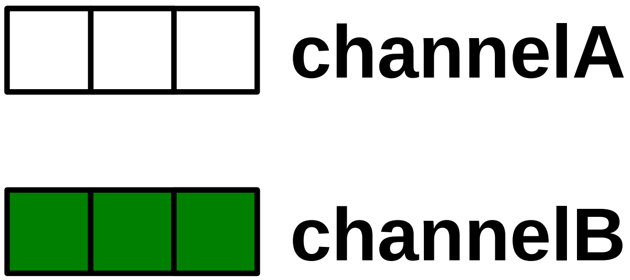
select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }



Select example

Thread1

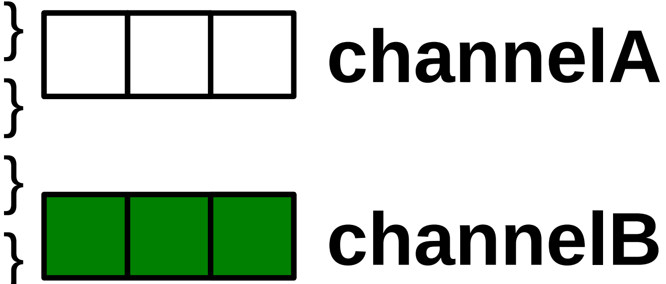
select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }



Select example

<u>Thread1</u>

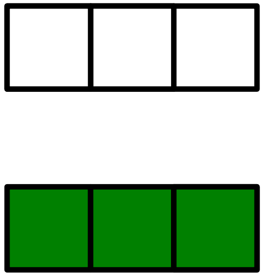
select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }



Select example

Thread1

select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
wait() zZz…



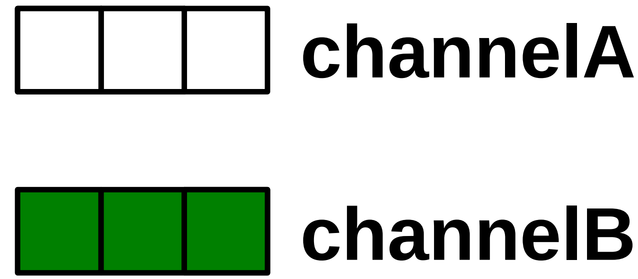
channelA

channelB

Select example

Thread1

select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
 wait() zZz…



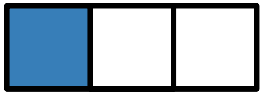
Thread2

send channelA

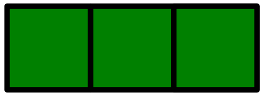
Select example

Thread1

select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
wait() zZz…



channelA



channelB

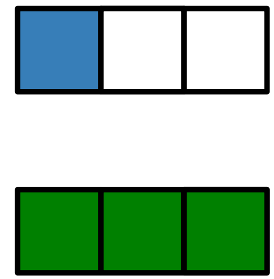
Thread2

send channelA

Select example

Thread1

select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
wait() zZz…



channelA

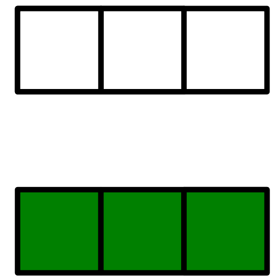
channelB

Thread2

send channelA

Select example

<u>Thread1</u>
select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
wait() zZz…



channelA

channelB

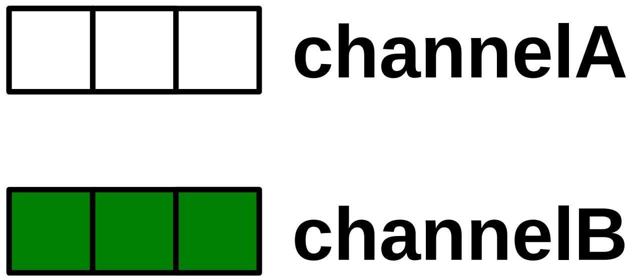
<u>Thread2</u>
send channelA

Find work to do → recv channelA

Select example

Thread1

select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
    wait() zZz…



Thread2
send channelA

Find work to do → recv channelA
select { recv channelA, send channelB }

Select example

Thread1

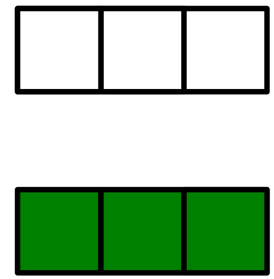
select { recv channelA, send channelB }

select { recv channelA, send channelB }

select { recv channelA, send channelB }

select { recv channelA, send channelB }

wait() zZz…



channelA

channelB

Thread2

send channelA

Find work to do → recv channelA

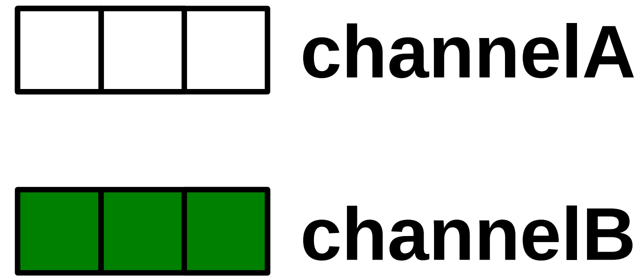
select { recv channelA, send channelB }

wait() zZz…

Select example

<u>Thread1</u>

select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
  wait() zZz…



<u>Thread2</u>

send channelA

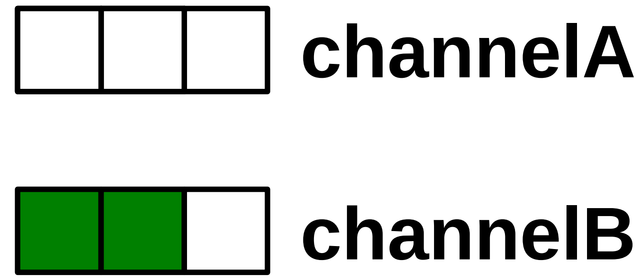
Find work to do → recv channelA
select { recv channelA, send channelB }
  wait() zZz…

recv channelB

Select example

Thread1

select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
  wait() zZz…



Thread2

send channelA

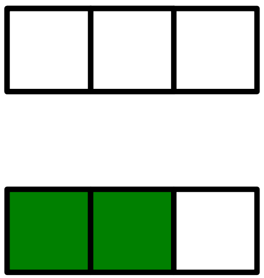
Find work to do → recv channelA
select { recv channelA, send channelB }
  wait() zZz…

recv channelB

Select example

Thread1

select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
  wait() zZz…



Thread2

send channelA

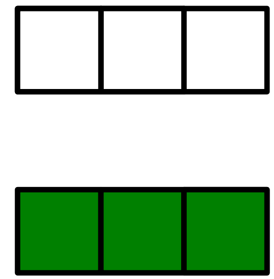
Find work to do → recv channelA
select { recv channelA, send channelB }
  wait() zZz…

recv channelB

Select example

Thread1

select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
select { recv channelA, send channelB }
wait() zZz…



channelA

channelB

Thread2

send channelA

Find work to do → recv channelA
select { recv channelA, send channelB }
wait() zZz…

recv channelB

Find work to do → send channelB

Select challenge

Select challenge

- Key challenge: waiting on multiple channels

Select challenge

- Key challenge: waiting on multiple channels
  - (e.g., either recv channelA <u>OR</u> send channelB)

Select challenge

- Key challenge: waiting on multiple channels
  - (e.g., either recv channelA <u>OR</u> send channelB)
- Q: How to wait?

Select challenge

- Key challenge: waiting on multiple channels
  - (e.g., either recv channelA <u>OR</u> send channelB)
- Q: How to wait?
- A: Semaphore/condition variable, but can’t wait on multiple

Select challenge

- Key challenge: waiting on multiple channels
  - (e.g., either recv channelA OR send channelB)
- Q: How to wait? **This is why select exists**
- A: Semaphore/condition variable, but can't wait on multiple

Select challenge

- Key challenge: waiting on multiple channels
  - (e.g., either recv channelA OR send channelB)
- Q: How to wait? **This is why select exists**
- A: Semaphore/condition variable, but can't wait on multiple
- Q: Can we store variable with channel?

Select challenge

- Key challenge: waiting on multiple channels
  - (e.g., either recv channelA <u>OR</u> send channelB)
- Q: How to wait? **This is why select exists**
- A: Semaphore/condition variable, but can't wait on multiple
- Q: Can we store variable with channel?
- A: No, variable is related to multiple channels

Select challenge

- Key challenge: waiting on multiple channels
    - (e.g., either recv channelA OR send channelB)
- Q: How to wait? **This is why select exists**
- A: Semaphore/condition variable, but can't wait on multiple
- Q: Can we store variable with channel?
- A: No, variable is related to multiple channels
- Q: Where else to store semaphore/condition variable?

Select challenge

- Key challenge: waiting on multiple channels
  - (e.g., either recv channelA OR send channelB)
- Q: How to wait? **This is why select exists**
- A: Semaphore/condition variable, but can't wait on multiple
- Q: Can we store variable with channel?
- A: No, variable is related to multiple channels
- Q: Where else to store semaphore/condition variable?
- A: Global?

Select challenge

- Key challenge: waiting on multiple channels
  - (e.g., either recv channelA OR send channelB)
- Q: How to wait? **This is why select exists**
- A: Semaphore/condition variable, but can't wait on multiple
- Q: Can we store variable with channel?
- A: No, variable is related to multiple channels
- Q: Where else to store semaphore/condition variable?
- A: Global? **NO! You will wake up everyone**

Select challenge

- Key challenge: waiting on multiple channels
  - (e.g., either recv channelA OR send channelB)
- Q: How to wait? **This is why select exists**
- A: Semaphore/condition variable, but can't wait on multiple
- Q: Can we store variable with channel?
- A: No, variable is related to multiple channels
- Q: Where else to store semaphore/condition variable?
- A: Global? **NO! You will wake up everyone**
- A: Within select function as a local variable

Select challenge

- Key challenge: waiting on multiple channels
  - (e.g., either recv channelA OR send channelB)
- Q: How to wait? **This is why select exists**
- A: Semaphore/condition variable, but can't wait on multiple
- Q: Can we store variable with channel?
- A: No, variable is related to multiple channels
- Q: Where else to store semaphore/condition variable?
- A: Global? **NO! You will wake up everyone**
- A: Within select function as a local variable
- Q: How does the send/receive know about it?

Select challenge

- Key challenge: waiting on multiple channels
  - (e.g., either recv channelA <u>OR</u> send channelB)
- Q: How to wait? **This is why select exists**
- A: Semaphore/condition variable, but can't wait on multiple
- Q: Can we store variable with channel?
- A: No, variable is related to multiple channels
- Q: Where else to store semaphore/condition variable?
- A: Global? **NO! You will wake up everyone**
- A: Within select function as a local variable
- Q: How does the send/receive know about it?
- A: It just needs a pointer

Select challenge

- Key challenge: waiting on multiple channels
  - (e.g., either recv channelA <u>OR</u> send channelB)
- Q: How to wait? **This is why select exists**
- A: Semaphore/condition variable, but can't wait on multiple
- Q: Can we store variable with channel?
- A: No, variable is related to multiple channels
- Q: Where else to store semaphore/condition variable?
- A: Global? **NO! You will wake up everyone**
- A: Within select function as a local variable
- Q: How does the send/receive know about it?
- A: It just needs a pointer
- Q: What about multiple simultaneous select calls…

Select challenge

- Key challenge: waiting on multiple channels
  - (e.g., either recv channelA OR send channelB)
- Q: How to wait? **This is why select exists**
- A: Semaphore/condition variable, but can't wait on multiple
- Q: Can we store variable with channel?
- A: No, variable is related to multiple channels
- Q: Where else to store semaphore/condition variable?
- A: Global? **NO! You will wake up everyone**
- A: Within select function as a local variable
- Q: How does the send/receive know about it?
- A: It just needs a pointer
- Q: What about multiple simultaneous select calls…
- A: Think about it

Selected food for thought

Selected food for thought

- Q: Why use a semaphore vs condition variable for signaling?

Selected food for thought

- Q: Why use a semaphore vs condition variable for signaling?
- Hint: think about missing signals

Selected food for thought

- Q: Why use a semaphore vs condition variable for signaling?
- Hint: think about missing signals
- Q: Wake up one or all select functions?

Selected food for thought

- Q: Why use a semaphore vs condition variable for signaling?
- Hint: think about missing signals
- Q: Wake up one or all select functions?
- Hint: depends on how you find work to do

Selected food for thought

- Q: Why use a semaphore vs condition variable for signaling?
- Hint: think about missing signals
- Q: Wake up one or all select functions?
- Hint: depends on how you find work to do
- Q: What if a channel is ready right before you make your local variable visible to the channel?

Selected food for thought

- Q: Why use a semaphore vs condition variable for signaling?
- Hint: think about missing signals
- Q: Wake up one or all select functions?
- Hint: depends on how you find work to do
- Q: What if a channel is ready right before you make your local variable visible to the channel?
- Hint: depends on how you add to channel

Selected food for thought

- Q: Why use a semaphore vs condition variable for signaling?
- Hint: think about missing signals
- Q: Wake up one or all select functions?
- Hint: depends on how you find work to do
- Q: What if a channel is ready right before you make your local variable visible to the channel?
- Hint: depends on how you add to channel
- Q: What do you need to do before/after you modify/access channel data?

Selected food for thought

- Q: Why use a semaphore vs condition variable for signaling?
- Hint: think about missing signals
- Q: Wake up one or all select functions?
- Hint: depends on how you find work to do
- Q: What if a channel is ready right before you make your local variable visible to the channel?
- Hint: depends on how you add to channel
- Q: What do you need to do before/after you modify/access channel data?
- Hint: what is the purpose of this assignment?

Multi-threading gdb commands

- info threads – lists all your threads
- thread NUM – switch to thread; see info threads for NUM
- thread apply all CMD – run CMD on all threads
  (e.g., thread apply all backtrace)
- backtrace is your friend – see where threads are stuck
- frame NUM – switch function call; see backtrace for NUM
- info locals – show all local variables in current function
- p VAR – p for printing any variable or parameter VAR
- p EXPR – print any expression EXPR
  (e.g., p channel->closed, p *channel, p *channel_list[1].channel->buffer)
- Ctrl-C – stop executing and break
- attach PID – connect to program with gdb after the fact

Summary

- Today's lecture
  - Channels
    - Model for synchronization via message passing
    - E.g. heavily used in Google's Go programming language and are very useful frameworks for high-level concurrent programming
  - Relevant information for Project 3
    - Challenges with select
  - GDB cheatsheet