

INF113: Conditional Variables

Kirill Simonov
22.10.2025

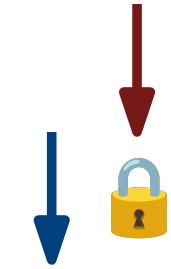


Final exam

- Takes place November 21
 - Exact time and location announced two weeks before the exam
- Exam score is 70% of the final grade
- No prerequisites for taking the exam
- Consists of multiple-choice and text answer questions
- Questions will be in English, Bokmål and Nynorsk
- No support materials on the exam
- Past exams published under **Modules** on MittUiB

Reminder

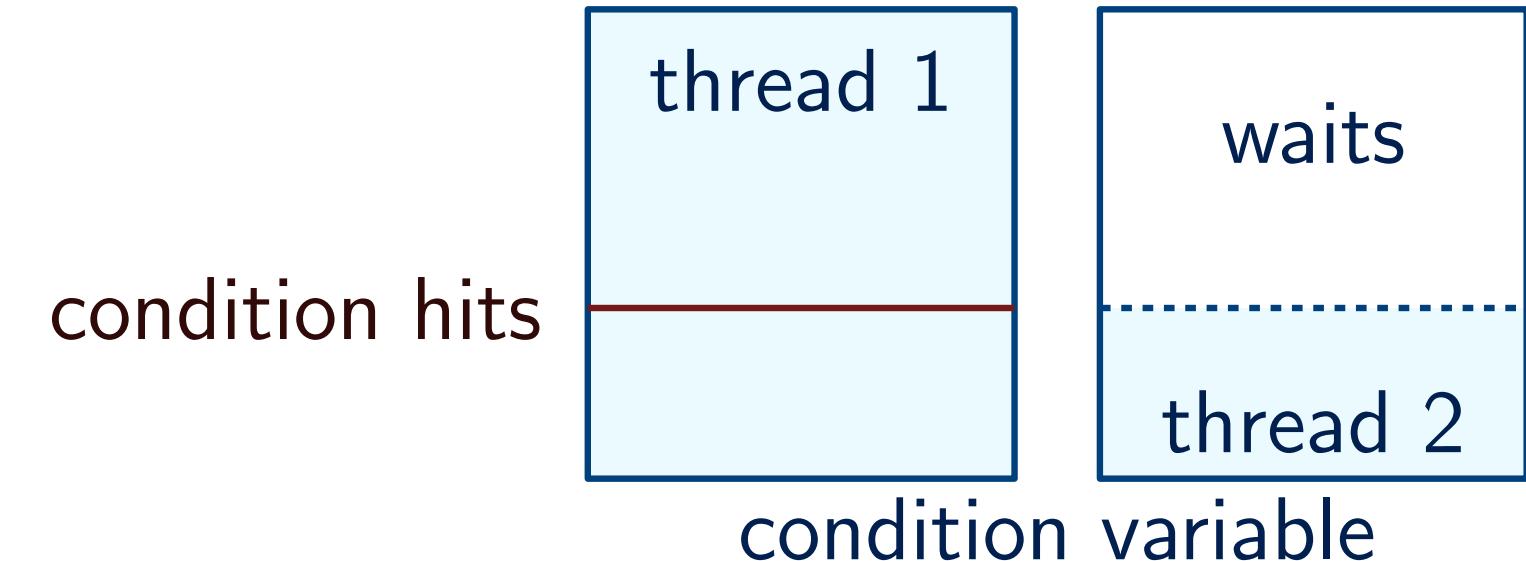
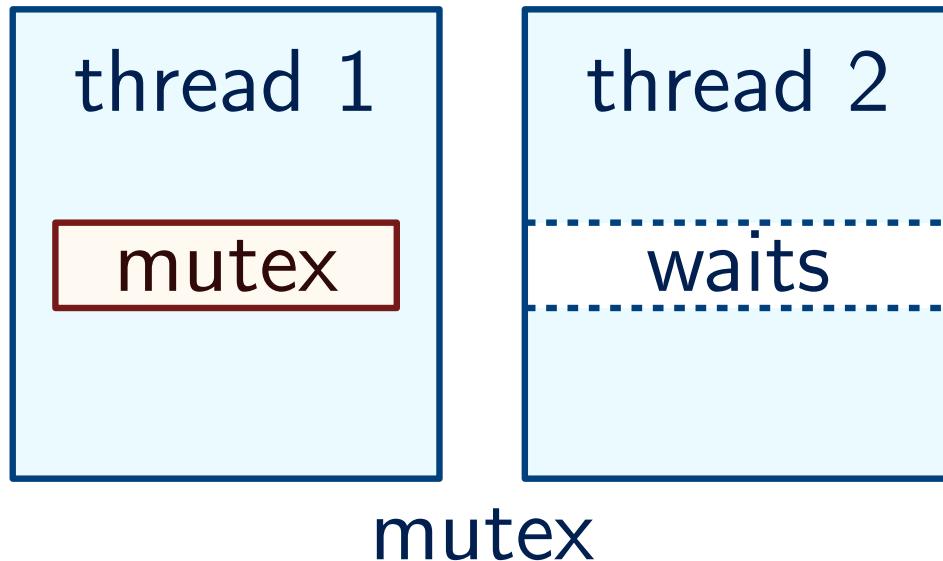
- We aim to design a set of general **synchronization primitives** based on limited hardware instructions
- Lock a.k.a mutex: “blocks” a part of code when in use by a thread
- Uses a test-and-set or similar hardware instruction to modify the state atomically
- Uses spinning to keep waiting threads “locked”
- Real implementations use system calls to put to sleep and wake threads, to reduce spinning
- But locks alone are not enough



```
pthread_mutex_t mutex;  
  
void *mythread(void *arg) {  
    ...  
    pthread_mutex_lock(&mutex);  
    counter = counter + 1;  
    pthread_mutex_unlock(&mutex);  
    ...  
}
```

Condition variables

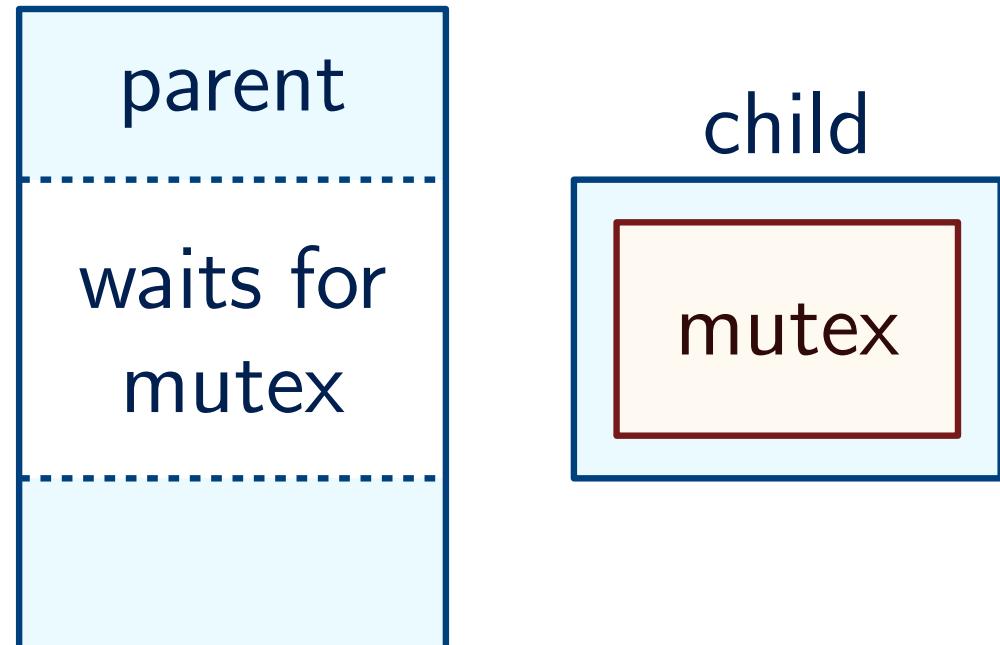
- Threads often wait for certain **conditions**:
 - Parent waits for the child to finish
 - Server thread waits for a request
 - Worker thread waits for a resource to be available
- Semantics of mutex vs condition variable:
 - Mutex targets a small critical section and is held by one thread
 - Condition may stay false for long
 - Condition may be triggered by multiple sources, and may affect multiple threads



Why not use a lock?

- Example: parent thread waits for the child thread to finish
- The child thread could hold a mutex while it runs

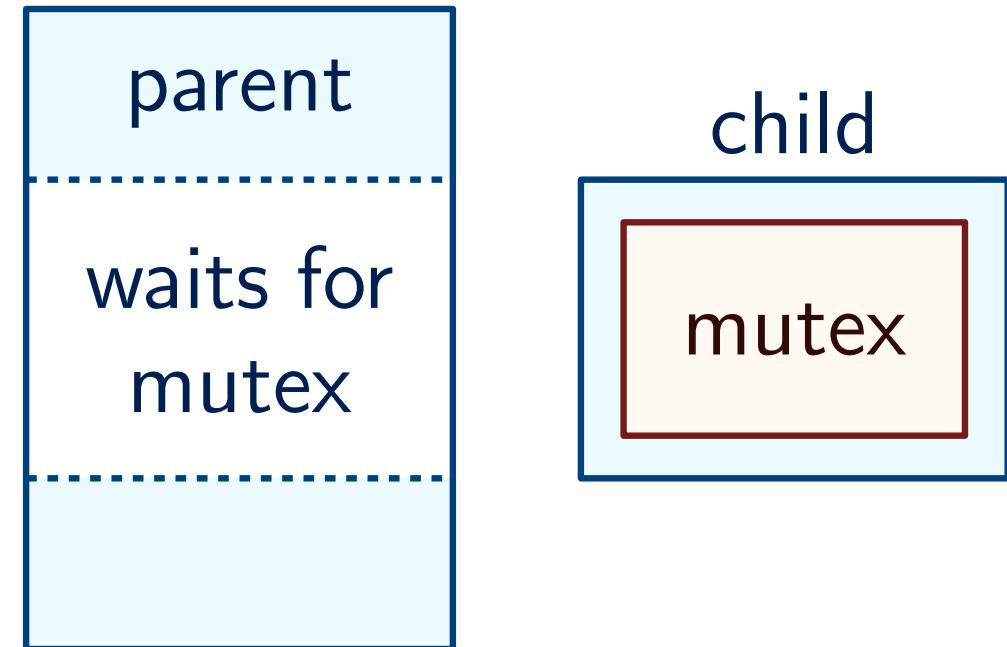
```
void *mythread(void *arg) {  
    pthread_mutex_lock(&child_lock);  
    ...  
    pthread_mutex_unlock(&child_lock);  
}  
  
int main() {  
    ...  
    pthread_create(&p, NULL, mythread, "A");  
    //parent waits for child  
    pthread_mutex_lock(&lock); ← parent could steal the lock  
    ...  
}
```



Why not use a lock?

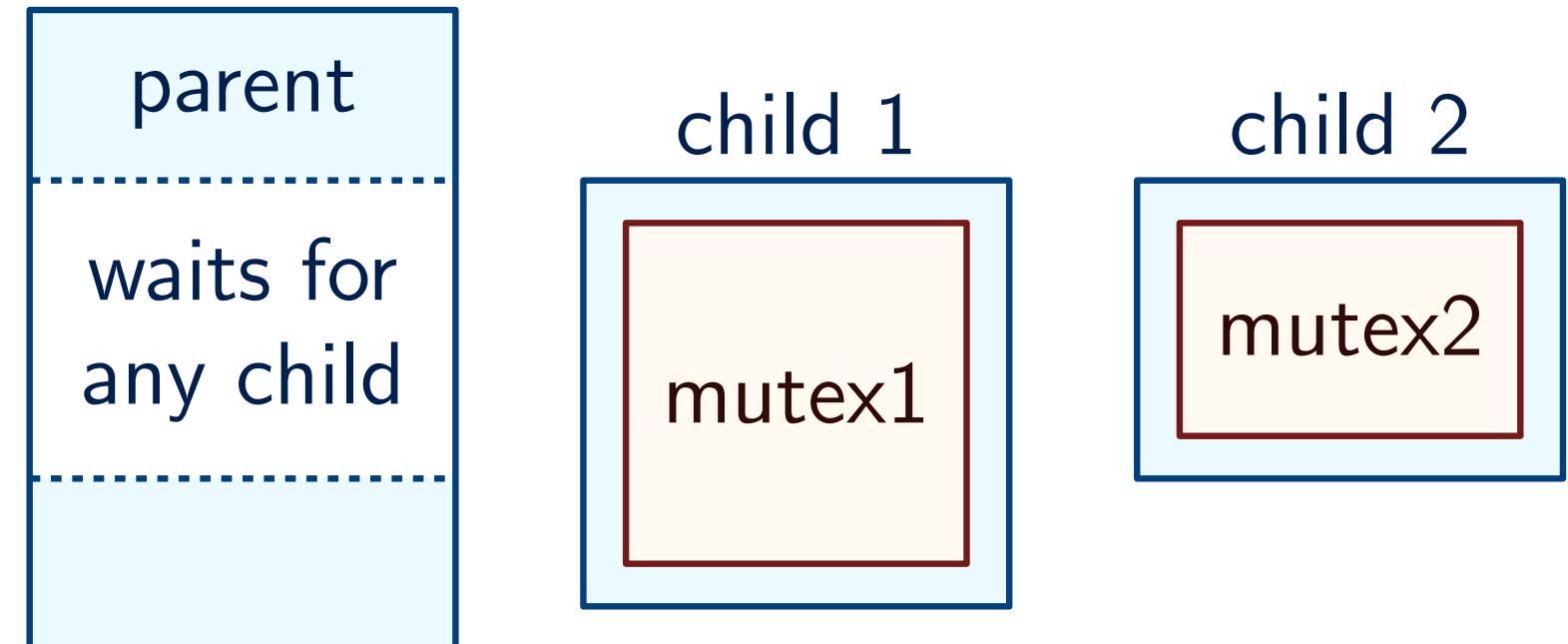
- Example: parent thread waits for the child thread to finish
- The child thread could hold a mutex while it runs

```
void *mythread(void *arg) {  
    ...  
    pthread_mutex_unlock(&child_lock);  
}  
  
int main() {  
    ...  
    pthread_mutex_lock(&child_lock);  
    pthread_create(&p, NULL, mythread, "A");  
    //parent waits for child  
    pthread_mutex_lock(&lock);  
    ...  
}
```



Why not use a lock?

- Example: parent thread waits for the child thread to finish
- The child thread could hold a mutex while it runs
- Could be (somewhat) made to work, but goes against the lock semantics
- Bigger issue: several children
 - If parent locks into mutex1, it will not notice mutex2 unlocking
- Even more problems if multiple threads wait for the condition



Let's try spinning

- Implement waiting by checking the condition in an infinite loop

```
volatile int done = 0;

void *child(void *arg) {
    // child works
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    pthread_create(&p, NULL, child, NULL);
    while (done == 0); // spin
    return 0;
}
```

- Pros: Simple
- Cons: Wastes CPU cycles for the whole wait duration
- We'd like to put a thread to “sleep” instead, until the condition is done

Condition variables: Interface

- Condition variable holds a queue of threads that wait for the condition to happen

```
pthread_cond_t c;
```

- If the expected condition is not met, the thread adds itself to the queue and goes to sleep

```
pthread_mutex_lock(&m);
while (done == 0)
    pthread_cond_wait(&c, &m);
```

wait releases m and
reacquires it upon waking

- The thread that realizes the condition wakes a waiting thread

```
pthread_mutex_lock(&m);
done = 1;
pthread_mutex_cond_signal(&c);
pthread_mutex_unlock(&m);
```

Example: Join

- Complete implementation for a join from parent:

```
void *child(void *arg) {  
    printf("child\n");  
    thr_exit();  
    return NULL;  
}  
  
int main(int argc, char *argv[]) {  
    printf("parent: begin\n");  
    pthread_t p;  
    pthread_create(&p, NULL, child, NULL);  
    thr_join();  
    printf("parent: end\n");  
    return 0;  
}
```

```
void thr_exit() {  
    pthread_mutex_lock(&m);  
    done = 1;  
    pthread_cond_signal(&c);  
    pthread_mutex_unlock(&m);  
}  
  
void thr_join() {  
    pthread_mutex_lock(&m);  
    while (done == 0)  
        pthread_cond_wait(&c, &m);  
    pthread_mutex_unlock(&m);  
}
```

Why also a mutex?

- Assume there's no lock reserved upon sending/receiving the signal:

```
void *child(void *arg) {  
    printf("child\n");  
    thr_exit();  
    return NULL;  
}  
  
int main(int argc, char *argv[]) {  
    printf("parent: begin\n");  
    pthread_t p;  
    pthread_create(&p, NULL, child, NULL);  
    thr_join();  
    printf("parent: end\n");  
    return 0;  
}
```

```
void thr_exit() {  
    done = 1;  
    pthread_cond_signal(&c);  
}  
  
void thr_join() {  
    while (done == 0)  
        pthread_cond_wait(&c);  
}
```

child interrupts between check and wait
now parent goes to sleep
after child has sent the signal
no one left to wake

Why the state variable?

- Seemingly, mutex itself tracks the state, however:

```
void *child(void *arg) {  
    printf("child\n");  
    thr_exit();  
    return NULL;  
}  
  
int main(int argc, char *argv[]) {  
    printf("parent: begin\n");  
    pthread_t p;  
    pthread_create(&p, NULL, child, NULL);  
    thr_join();  
    printf("parent: end\n");  
    return 0;  
}
```

```
void thr_exit() {  
    pthread_mutex_lock(&m);  
    pthread_cond_signal(&c);  
    pthread_mutex_unlock(&m);  
}  
  
void thr_join() {  
    pthread_mutex_lock(&m);  
    pthread_cond_wait(&c, &m);  
    pthread_mutex_unlock(&m);  
}
```

if `thr_exit()` completes first
then `thr_join()` will wait forever

Back to working solution

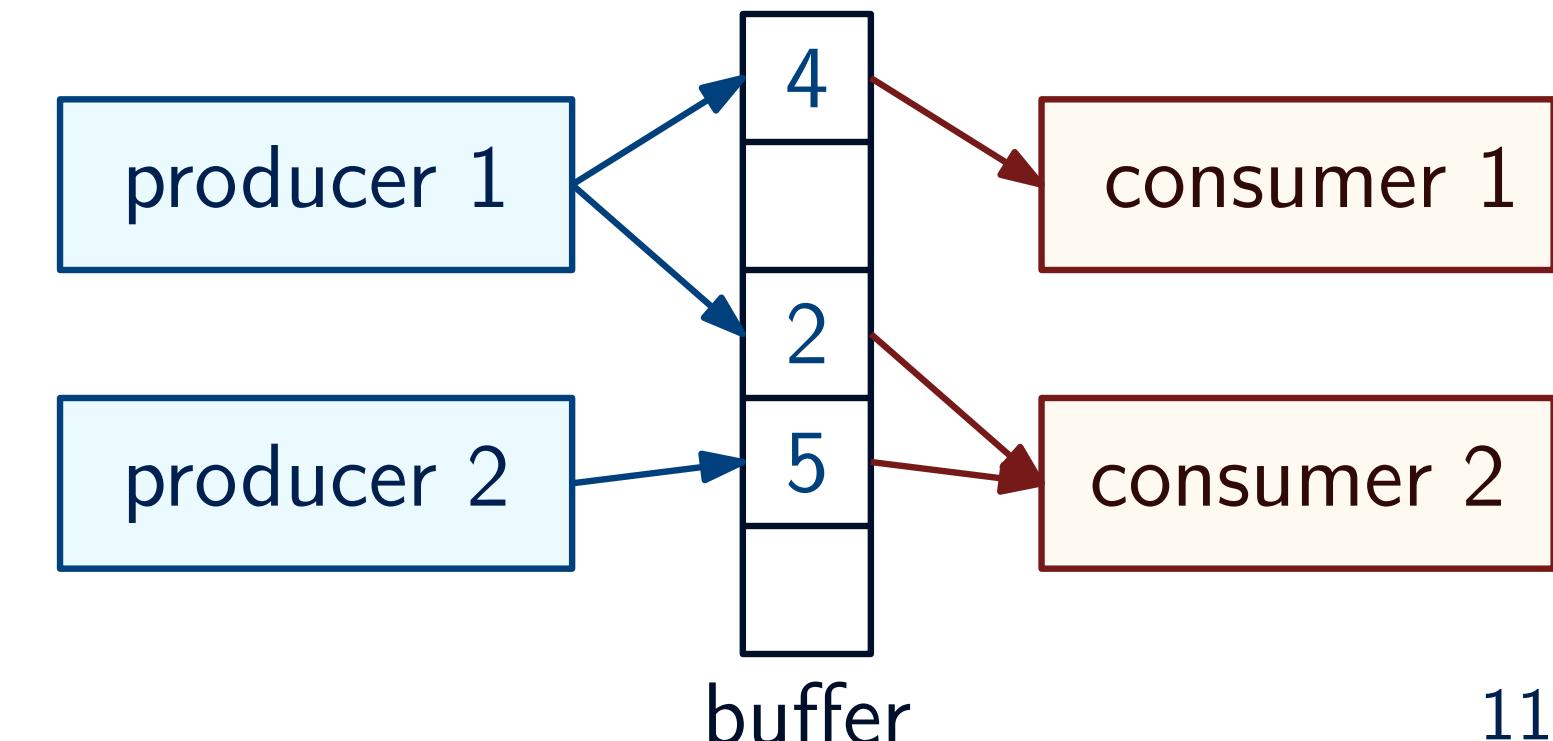
- Complete implementation for a join from parent:

```
void *child(void *arg) {  
    printf("child\n");  
    thr_exit();  
    return NULL;  
}  
  
int main(int argc, char *argv[]) {  
    printf("parent: begin\n");  
    pthread_t p;  
    pthread_create(&p, NULL, child, NULL);  
    thr_join();  
    printf("parent: end\n");  
    return 0;  
}
```

```
void thr_exit() {  
    pthread_mutex_lock(&m);  
    done = 1;  
    pthread_cond_signal(&c);  
    pthread_mutex_unlock(&m);  
}  
  
void thr_join() {  
    pthread_mutex_lock(&m);  
    while (done == 0)  
        pthread_cond_wait(&c, &m);  
    pthread_mutex_unlock(&m);  
}
```

Producer/consumer problem

- **Producer** threads generate data items and place them in the buffer
- **Consumers** pick the items from the buffer and process them
- Example: multi-threaded web server
 - Producer puts HTTP requests into the work queue
 - Consumer threads take requests from the queue and serve them
- Example: `grep foo file.txt | wc -l`
 - Producer: `grep foo file.txt`
 - Consumer: `wc -l`
- Challenge: buffer protection and waiting
 - Producers wait while buffer is full
 - Consumers wait while buffer is empty



Producer/consumer: Templates

- Assume buffer is a single integer
- Clearly, protection is necessary

```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        put(i);
    }
}

void *consumer(void *arg) {
    while (1) {
        int tmp = get();
        printf("%d\n", tmp);
    }
}
```

template producer and consumer

```
int buffer;
int count = 0;

void put(int value) {
    assert(count == 0);
    count = 1;
    buffer = value;
}

int get() {
    assert(count == 1);
    count = 0;
    return buffer;
}
```

helper functions

Producer/consumer: Attempt 1

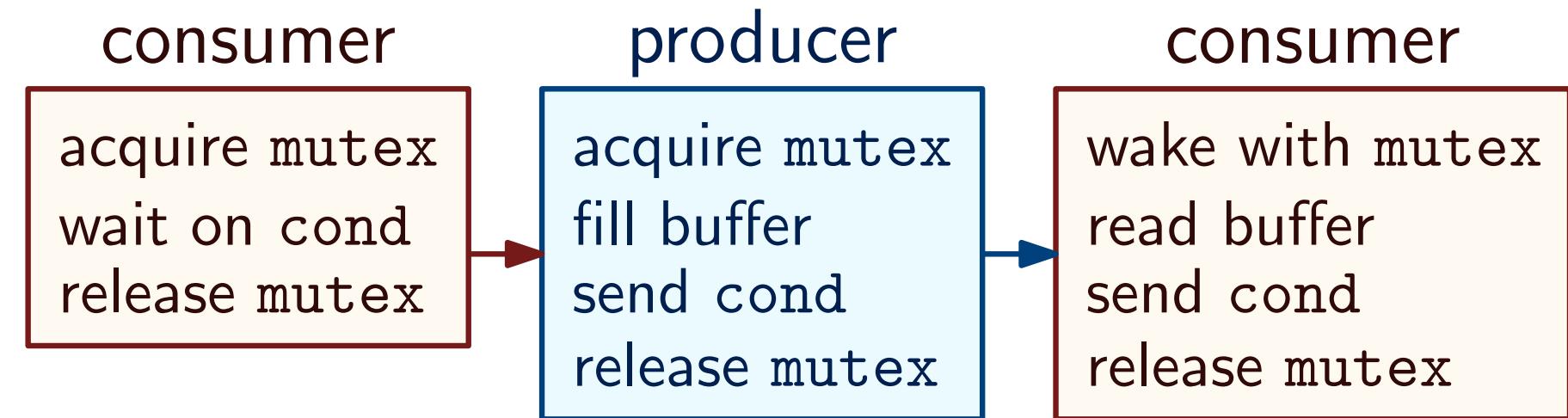
- Measure 1: wrap the critical section with a mutex
- Measure 2: wait on a conditional variable if buffer is full/empty
- Same for consumer

```
void *consumer(void *arg) {  
    while (1) {  
        pthread_mutex_lock(&mutex);  
        if (count == 0)  
            pthread_cond_wait(&cond, &mutex);  
        int tmp = get();  
        pthread_cond_signal(&cond);  
        pthread_mutex_unlock(&mutex);  
        printf("%d\n", tmp);  
    }  
}
```

first wake other threads
and release the lock
then go into a system call

Producer/consumer: 1v1

- This works safely with one producer and one consumer

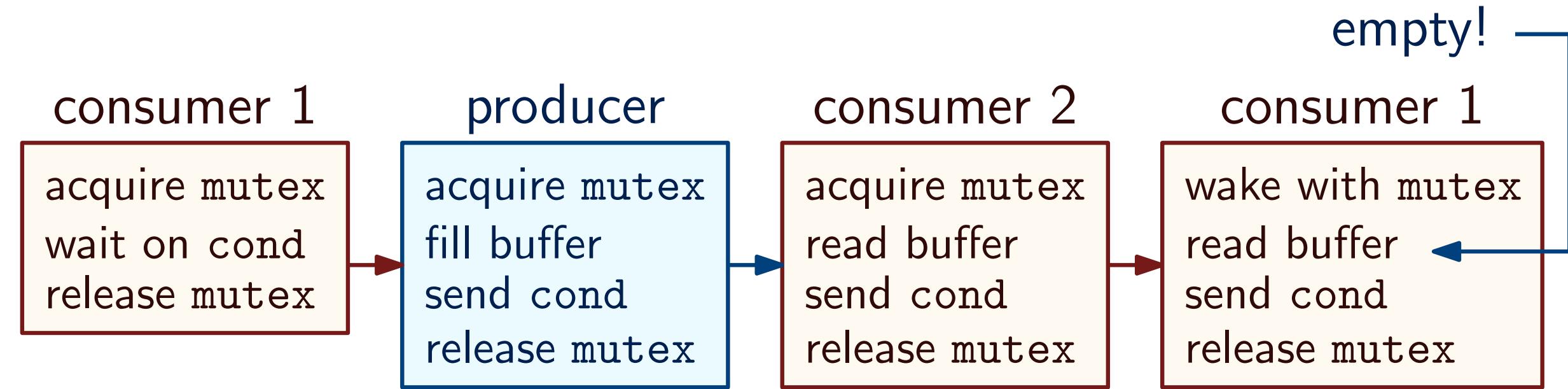


```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        if (count == 1)
            pthread_cond_wait(&cond, &mutex);
        put(i);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        pthread_mutex_lock(&mutex);
        if (count == 0)
            pthread_cond_wait(&cond, &mutex);
        int tmp = get();
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Producer/consumer: 1v2

- 2 consumers can have problems:
- State of buffer can change before waking



```
void *producer(void *arg) {  
    for (int i = 0; i < loops; i++) {  
        pthread_mutex_lock(&mutex);  
        if (count == 1)  
            pthread_cond_wait(&cond, &mutex);  
        put(i);  
        pthread_cond_signal(&cond);  
        pthread_mutex_unlock(&mutex);  
    }  
}
```

```
void *consumer(void *arg) {  
    while (1) {  
        pthread_mutex_lock(&mutex);  
        if (count == 0)  
            pthread_cond_wait(&cond, &mutex);  
        int tmp = get();  
        pthread_cond_signal(&cond);  
        pthread_mutex_unlock(&mutex);  
        printf("%d\n", tmp);  
    }  
}
```

Solving issue 1

- We need to check the condition again upon waking
- **Solution:** replace if with while
- Now even if another consumers clears the buffer, we notice and go into wait again

```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == 1)
            pthread_cond_wait(&cond, &mutex);
        put(i);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        pthread_mutex_lock(&mutex);
        while (count == 0)
            pthread_cond_wait(&cond, &mutex);
        int tmp = get();
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Issue 2

buffer is empty, no one can wait the producer

consumer 1

```
acquire mutex  
wait on cond  
release mutex
```

producer

```
acquire mutex  
fill buffer  
send cond  
release mutex
```

producer

```
acquire mutex  
wait on cond  
release mutex
```

consumer 2

```
acquire mutex  
read buffer  
send cond  
release mutex
```

consumer 1

```
wake with mutex  
wait on cond  
release mutex
```

```
void *producer(void *arg) {  
    for (int i = 0; i < loops; i++) {  
        pthread_mutex_lock(&mutex);  
        while (count == 1)  
            pthread_cond_wait(&cond, &mutex);  
        put(i);  
        pthread_cond_signal(&cond);  
        pthread_mutex_unlock(&mutex);  
    }  
}
```

```
void *consumer(void *arg) {  
    while (1) {  
        pthread_mutex_lock(&mutex);  
        while (count == 0)  
            pthread_cond_wait(&cond, &mutex);  
        int tmp = get();  
        pthread_cond_signal(&cond);  
        pthread_mutex_unlock(&mutex);  
        printf("%d\n", tmp);  
    }  
}
```

Solving issue 2

- A consumer should wake a producer, and vice versa
- **Solution:** use two different condition variables
- producer waits on `empt`, but sends `full`
consumer waits on `full`, but sends `empt`

```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == 1)
            pthread_cond_wait(&empt, &mutex);
        put(i);
        pthread_cond_signal(&full);
        pthread_mutex_unlock(&mutex);
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        pthread_mutex_lock(&mutex);
        while (count == 0)
            pthread_cond_wait(&full, &mutex);
        int tmp = get();
        pthread_cond_signal(&empt);
        pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Producer/consumer: Full solution

- Can extend the buffer too

```
int buffer[MAX];
int fill_ptr = 0;
int use_ptr = 0;
int count = 0;

void put(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % MAX;
    count--;
    return tmp;
}
```

```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == MAX)
            pthread_cond_wait(&empt, &mutex);
        put(i);
        pthread_cond_signal(&full);
        pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    while (1) {
        pthread_mutex_lock(&mutex);
        while (count == 0)
            pthread_cond_wait(&full, &mutex);
        int tmp = get();
        pthread_cond_signal(&empt);
        pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Summary

- Threads can wait on conditions and wake other waiting threads via conditional variables
- A conditional variable stores a queue of waiting threads
 - We have already seen how to implement a queue holding threads within a lock
- `pthread_cond_signal` and `pthread_cond_broadcast` calls
 - signal wakes one or several threads
 - broadcast wakes **all** waiting threads
- Tip: lock when you signal
- Tip: check conditions in a while
- **Next time:** Semaphores—one primitive to rule them all