Dividing work among threads/processes

- Different threads/processes execute same function on different data
  - *Data decomposition*
- Different threads/processes execute different functions
  - *Task decomposition*
- Third option
  - One thread executes a function & produces output data
  - Other thread executes a different function & consumes data
  - *Data-flow decomposition*

Shared buffer to facilitate this ...

- *Producer* thread/process produces data items & places them in shared buffer
- *Consumer* thread/process removes data items from shared buffer & consumes them
- Need to maintain consistency of buffer at all times
  - Producer should not put items into full buffer
  - Consumer should not remove items from empty buffer
- *Busy waiting* should be avoided

Producer-consumer problem

- Also called *bounded buffer* problem
- Solution
  - Keep count of number of items in buffer
  - Producer can check number of items
    - Go to *sleep* when buffer is full
    - Be *woken up* when buffer has at least one empty space
  - Consumer can check number of items
    - Go to *sleep* when buffer is empty
    - Be *woken up* when buffer has at least one item

Synchronization based on *state/value* of *shared data*

Pthread library provides a mechanism that can be used for synchronization based on value/state of data

Pthread condition variable

data type for condition variable

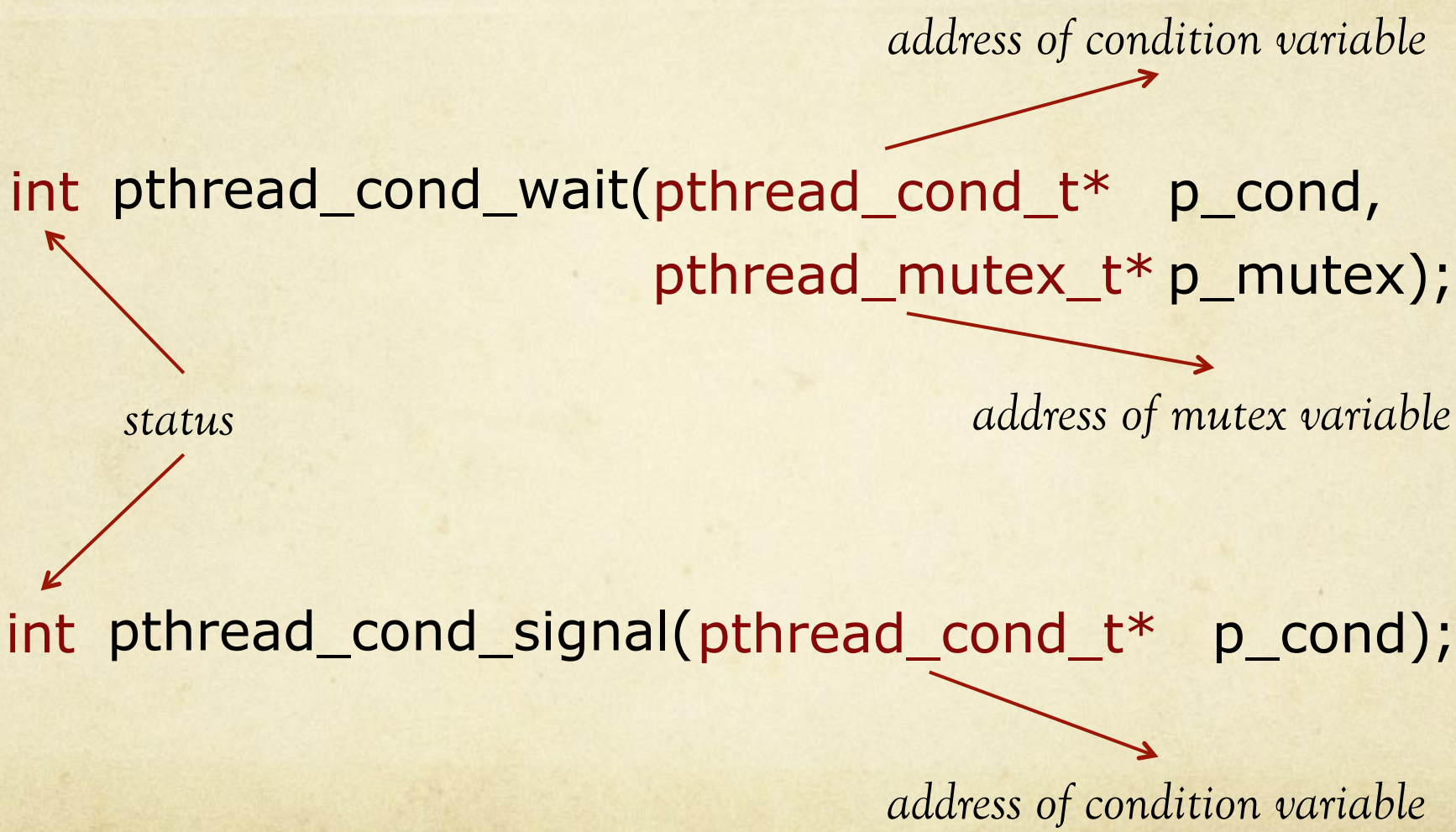
name for condition variable

Make it meaningful so it reflects the condition that it represents

```
pthread_cond_t   cond_var
        = PTHREAD_COND_INITIALIZER;
```

macro to initialize condition variable with default attributes

Pthread condition variable



Pthread condition variable

Shared data x

Thread A

Condition variable C

Thread B

Mutex M

Sample program – 7

```
#include <pthread.h>
#include <iostream>
#define N 10
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;
int buffer[N];
int count = 0;
```

```
void* producer(void* arg) {
  int item; int curr_pos = 0;
  for(int i = 0; i < 100; ++i) {
    pthread_mutex_lock(&mutex);
    if(count == N)
       pthread_cond_wait(&not_full, &mutex);
    buffer[curr_pos++] = rand();
    cout << "Producing " <<
              buffer[curr_pos-1] << endl;
    if(curr_pos == N) curr_pos = 0;
    ++count;
    pthread_cond_signal(&not_empty);
    pthread_mutex_unlock(&mutex);
}}
```

```
void* consumer(void* arg) {
  int item; int curr_pos = 0;
  for(int i = 0; i < 100; ++i) {
    pthread_mutex_lock(&mutex);
    if(count == 0)
         pthread_cond_wait(&not_empty, &mutex);
    cout << "Consuming " <<
              buffer[curr_pos++] << endl;
    if(curr_pos == N) curr_pos = 0;
    --count;
    pthread_cond_signal(&not_full);
    pthread_mutex_unlock(&mutex); }}
```

```
int main() {
  pthread_t id1, id2;
  pthread_create(&id1, NULL, producer, NULL);
  pthread_create(&id1, NULL, consumer, NULL);
  pthread_exit(NULL); }
```