# Shared Memory Programming with Pthreads

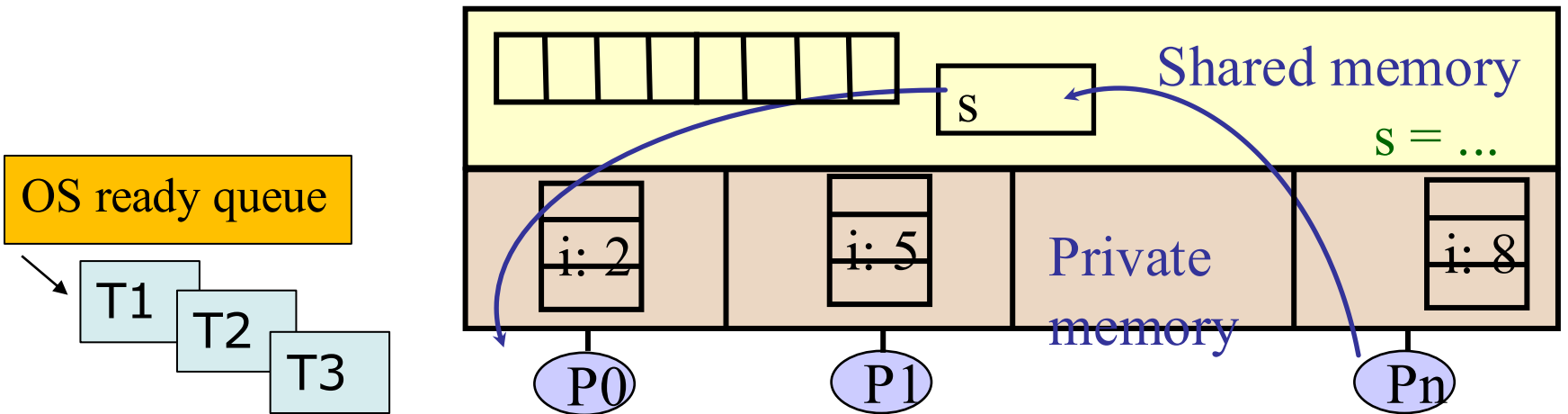
T. Yang. UCSB CS140

# Outline

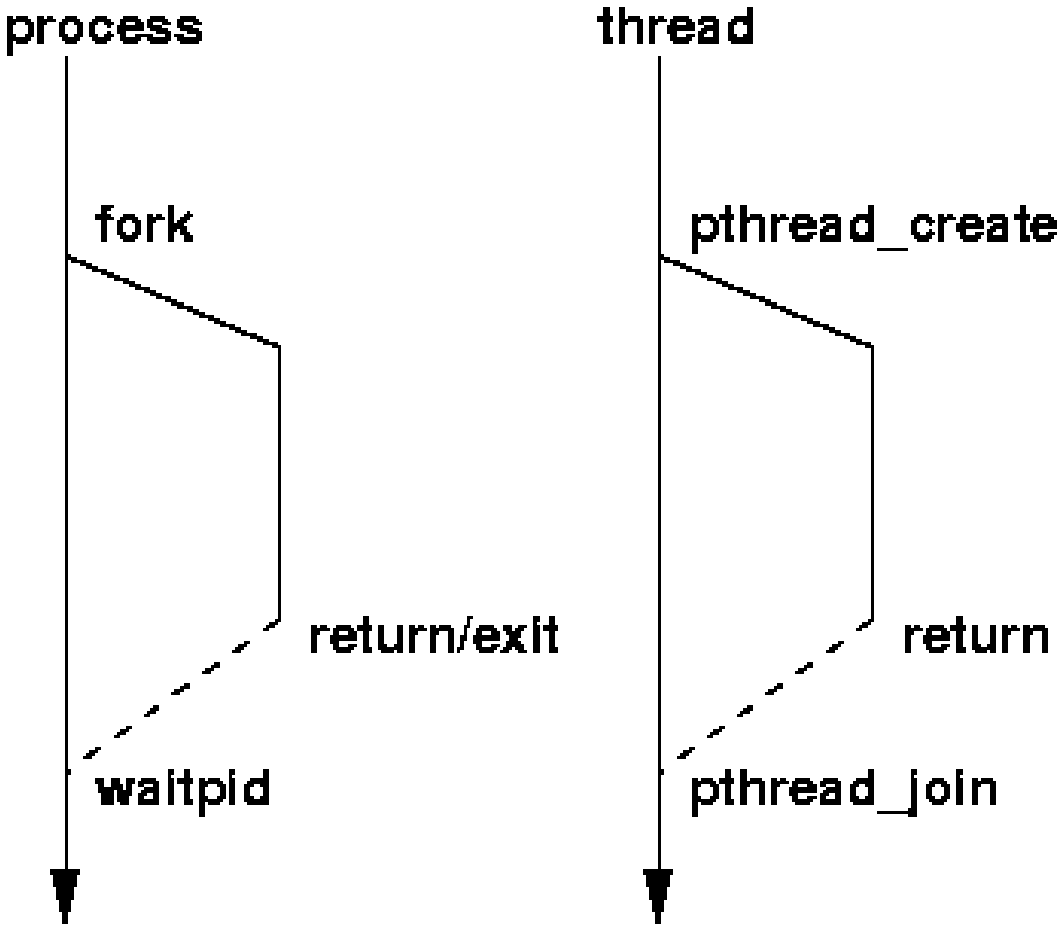
- Overview and examples of parallel programming with Pthreads
- Synchronization with Pthreads
  - Primitives
  - Implementation of barrier
- Use of threads in Python packages

# Pthread Programming with Shared Memory:

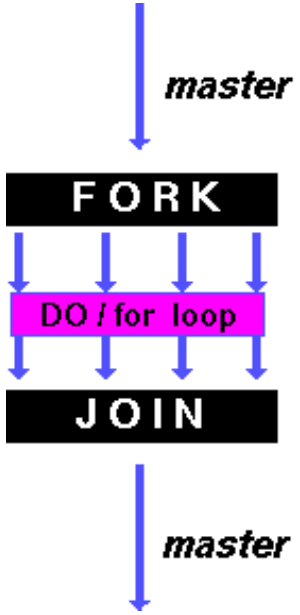
- Pthreads: The POSIX threading interface
  - POSIX: *Portable Operating System Interface for UNIX*
- Program is a set of explicitly created threads executed by OS
- Each thread has private variables, e.g., local stack variables
- Also have a set of shared variables, e.g., global variables/heap
  - Threads coordinate by synchronizing on shared variables



# Explicit creation of Unix processes/Pthreads vs. Implicit thread creation in OpenMP




```
#pragma omp parallel for
  for (i=0; i<max; i++)
        x[i] = 0;
```




Fundamental primitives for high level software layers

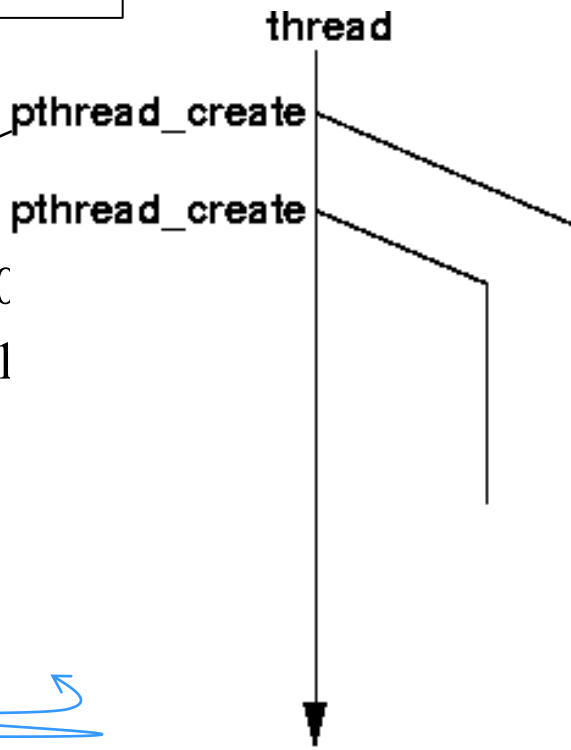
# Example of Pthreads

```
#include <pthread.h>
#include <stdio.h>
void *PrintHello(void * id){
   printf("Thread%d: Hello World!\n", id);
}

void main (){
   pthread_t thread0, thread1;
   pthread_create(&thread0, NULL, PrintHello, (void *) 0
   pthread_create(&thread1, NULL, PrintHello, (void *) 1
}
```

Pthreads header

SPMD code for each thread



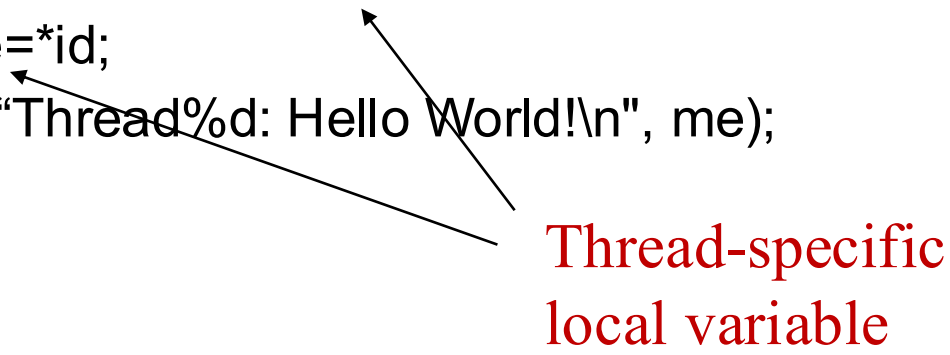

gcc pth_hello . c −lpthread

Compilation with Pthreads library

# Example of Pthreads with join

```
#include <pthread.h>
#include <stdio.h>
void *PrintHello(void * id){
   int me=*id;
   printf("Thread%d: Hello World!\n", me);
}

void main (){
   pthread_t thread0, thread1;
   pthread_create(&thread0, NULL, PrintHello, (void *) 0);
   pthread_create(&thread1, NULL, PrintHello, (void *) 1);
   pthread_join(thread0, NULL);
   pthread_join(thread1, NULL);
}
```

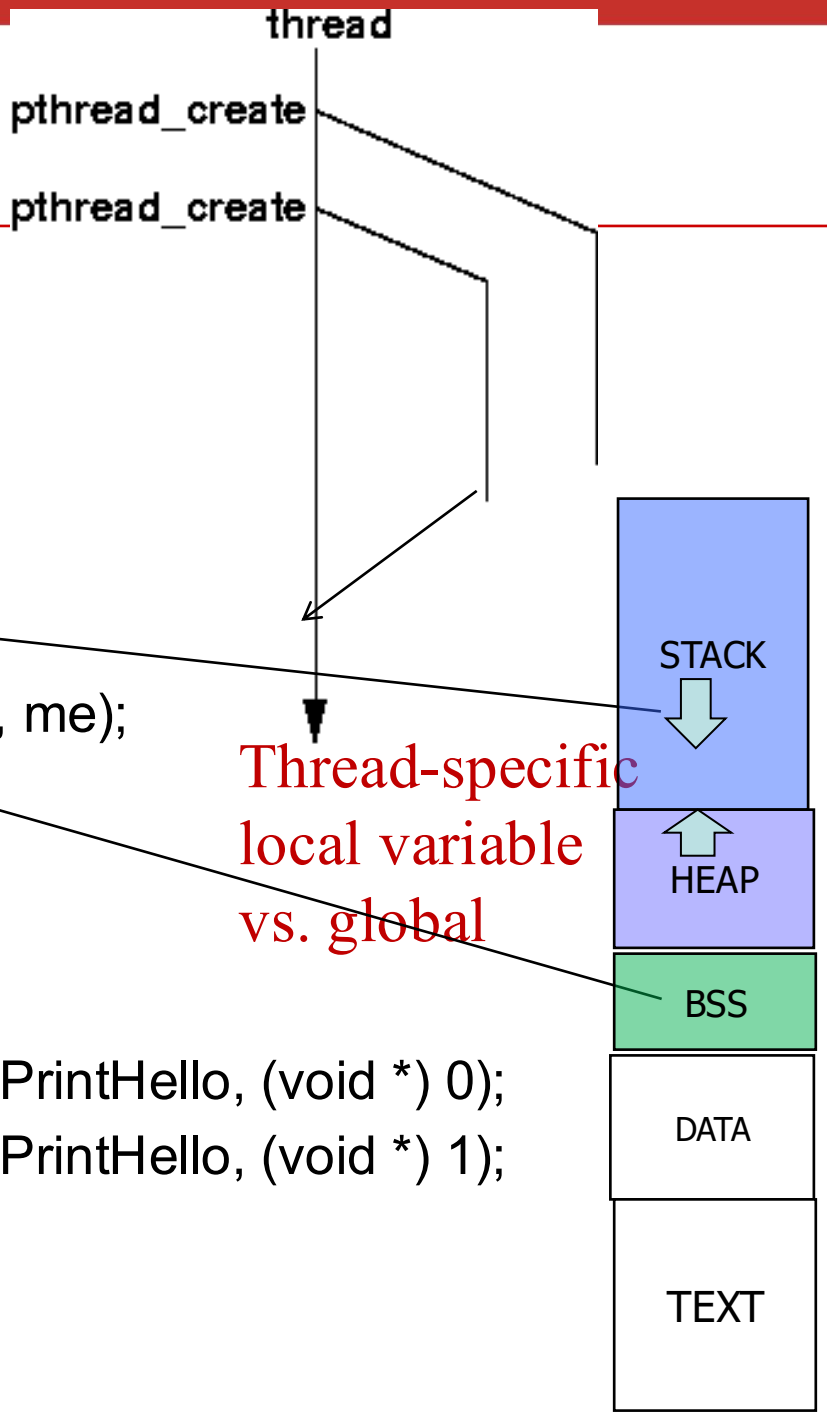


# Shared vs local variables

```
#include <pthread.h>
#include <stdio.h>
int me;
void *PrintHello(void * id){
   int x =10;
   me=*id +x;
   printf("Thread%d: Hello World!\n", me);
}

void main (){
   pthread_t thread0, thread1;
   pthread_create(&thread0, NULL, PrintHello, (void *) 0);
   pthread_create(&thread1, NULL, PrintHello, (void *) 1);
   pthread_join(thread0, NULL);
   pthread_join(thread1, NULL);
}
```

SPMD code




Thread-specific local variable vs. global

# How to Map Parallel Iterations to Pthreads

Write SPMD code for thread functions with explicit mapping

```
sum = 0.0;
for (i = 0; i <= n; i++)
    sum += f(i);
```

| Thread | Iterations |
|---|---|
| ID=0 | 0, 1, 2, …, b-1 |
| ID=1 | b, b+1, …, 2b-1 |

Assignment of work using block mapping

```
For (i= ID*b; i< ID*b+b; i++)
        localsum +=f(i)
```

| Thread | Iterations |
|---|---|
| ID=0 | 0, 2, 4, … |
| ID=1 | 1, 3, 5… |

Cyclic mapping

```
For (i= ID; i<=n; i= i+ 2)
        localsum +=f(i)
```

| Thread | Iterations |
|---|---|
| 0 | 0, 1, 4, 5… |
| 1 | 2, 3, 6,7… |

Block cyclic mapping

Which one is better?

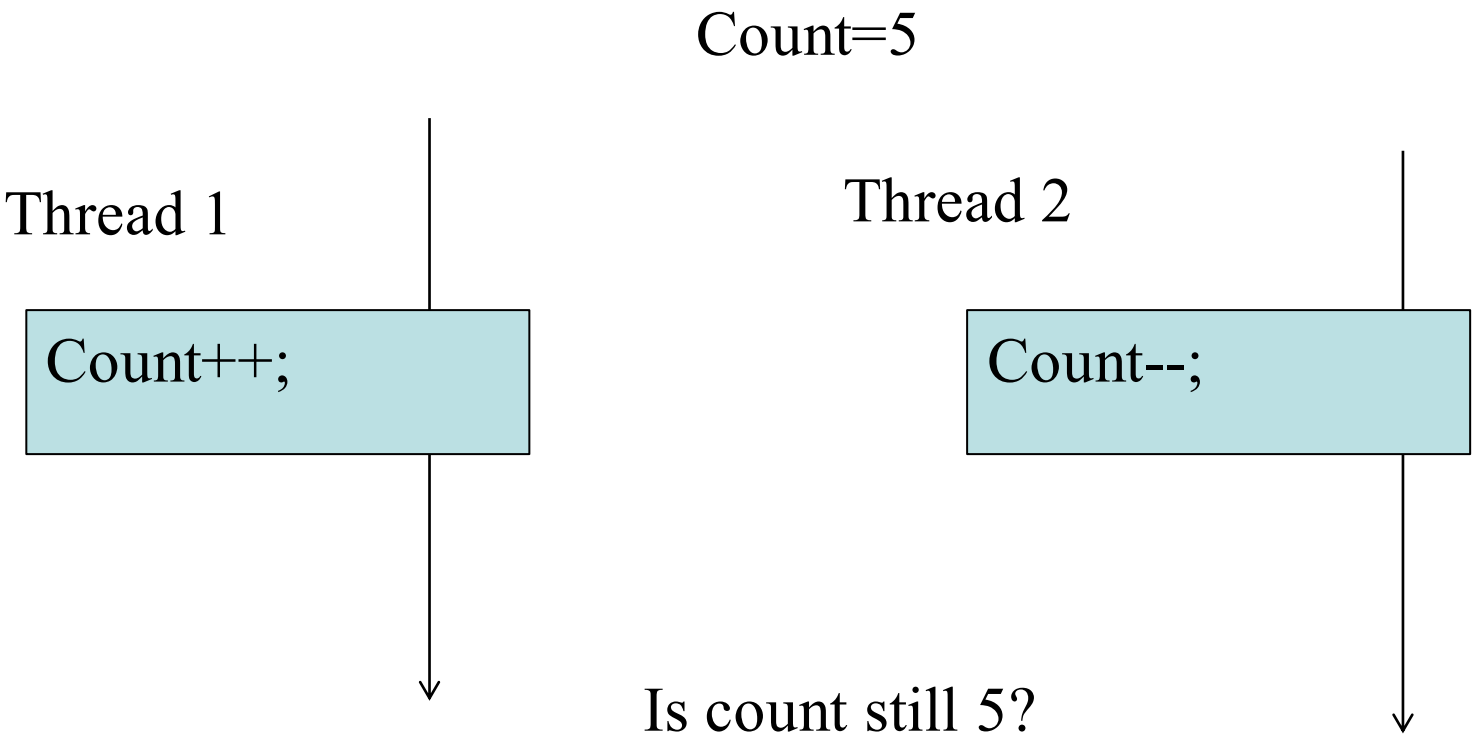
# Impact of Mapping/Scheduling Choices

- **Load balance**
  - Same work in each iteration?
  - Processors working at same speed?
  - Cyclic mapping typically improves load balancing
- **Data locality and false sharing issue**
  - Particularly within cache lines for small chunk sizes
  - Also impacts data reuse on same processor
- **Static vs. dynamic scheduling**
  - Static decisions are cheap because they require no run-time coordination, but less adaptive to dynamic load change
  - Dynamic decisions have scheduling overhead  impacted by complexity and frequency of decisions

# RACE CONDITIONS AND SYNCHRONIZATION

1. Mutex (lock)
2. Conditional Variables
3. Barriers

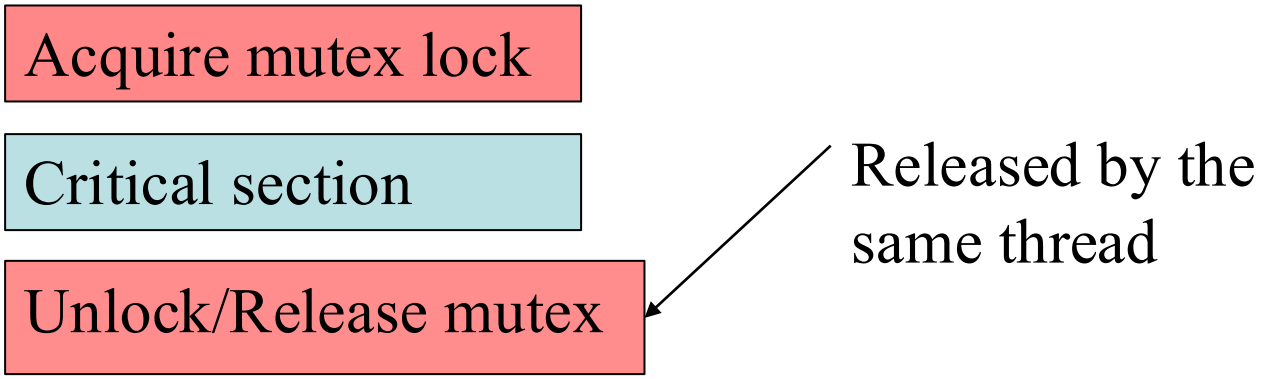
# Race Conditions: Example




Race Conditions: two or more threads are reading and writing on shared data and the final result depends on who runs precisely & when

# Mutexes for Mutual Exclusion (Locks)

- Code structure




- Use one variable to restrict access to a critical section to a single thread at a time.
  - Guarantee that one thread “excludes” all other threads while it executes the critical section.
- When a thread waits on a mutex/lock,

CPU resource can be used by others.

- Implement “OpenMP critical”

# Mutexes in Pthreads

- A special type for mutexes: pthread_mutex_t.

```
int pthread_mutex_init(
      pthread_mutex_t*              mutex_p   /* out */
      const pthread_mutexattr_t*    attr_p    /* in  */);
```

- To gain access to a critical section, call

```
int pthread_mutex_lock(pthread_mutex_t* mutex_p  /* in/out */);
```

- To release

```
int pthread_mutex_unlock(pthread_mutex_t* mutex_p  /* in/out */);
```

- When finishing use of a mutex, call

```
int pthread_mutex_destroy(pthread_mutex_t* mutex_p  /* in/out */);
```

# Mutex lock solution for critical section problem

Global variable
int s = 0;

Thread 0
```
for i = 0, n/2-1
    pthread_mutex_lock(&mutex)
    s = s + f(A[i])
    pthread_mutex_unlock(&mutex)
```

Thread 1
```
for i = n/2, n-1
    pthread_mutex_lock(&mutex)
    s = s + f(A[i])
    pthread_mutex_unlock(&mutex)
```

**SPMD code**:
```
Let me be my thread ID
for i = me*r to me*r +r-1
    pthread_mutex_lock(&mutex)
    s = s + f(A[i])
    pthread_mutex_unlock(&mutex)
```

Thread ID starts from 0.
Block mapping with size r

# SYNCHRONIZATION

1. Mutex (lock)
2. Conditional Variables
3. Barriers

# Condition Variables

- More programming primitives to simplify code for synchronization of threads

| Synchronization options | Functionality |
|---|---|
| Busy waiting | Spinning for a condition. Waste resource. Not safe |
| Mutex lock | Support code with simple mutual exclusion |
| Semaphore | Signal-based synchronization. Allow sharing (not wait unless semaphore=0) |
| Condition variables | More complex synchronization: Let threads wait until a user-defined condition becomes true |
| Barrier | Rendezvous-based synchronization |

# How to Use Condition Variables: Typical Flow

- Thread 1: //try to get into critical section.

  Check&wait for some condition

```
Mutex_lock(mutex);
While (condition is not satisfied)
        Cond_Wait(mutex, cond);
Do something in this critical section;
Mutex_unlock(mutex)
```

- Thread 2: // Try to make a condition true.

```
Mutex_lock(mutex);
Make a condition true;
Signal(cond);
Mutex_unlock(mutex);
```

Check condition needs mutex protection too

Condition checking is application-specific

Condition variable is used together mutex lock

# How to Use Condition Variables: Typical Flow

- Thread 1: //try to get into critical section.

  Check&wait for some condition

```
Mutex_lock(mutex);
While (condition is not satisfied)
        Cond_Wait(mutex, cond);
Do something in this critical section;
Mutex_unlock(mutex)
```

Calling thread gives up the lock, and then this thread is placed in a waiting queue of this condition variable.

Calling thread regains lock after wakeup

- Thread 2: // Try to make a condition true.

```
Mutex_lock(mutex);
Make a condition true;
Signal(cond);
Mutex_unlock(mutex);
```

Notify some thread in the waiting queue of this condition variable to wake up

# Pthread synchronization: Condition variables

int status;  pthread_condition_t cond;

const pthread_condattr_t attr;

pthread_mutex mutex;

status = pthread_cond_init(&cond,&attr);

status = pthread_cond_destroy(&cond);

status = pthread_cond_wait(&cond,&mutex);

-Wait in a queue until somebody wakes up. Then the mutex is reacquired.

status = pthread_cond_signal(&cond);

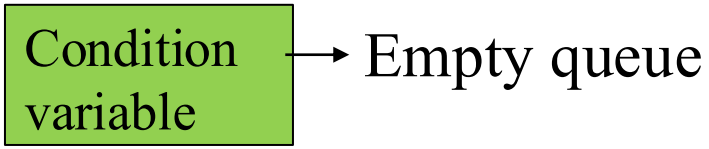
- Wake up one waiting thread.

status = pthread_cond_broadcast(&cond);

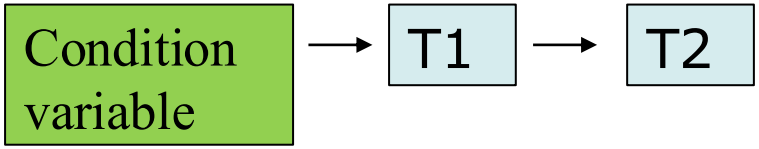
- Wake up all waiting threads in that condition. Used when waking up one thread in the queue of a condition variable  to run is not sufficient.

# How can Pthread_cond_signal() be implemented

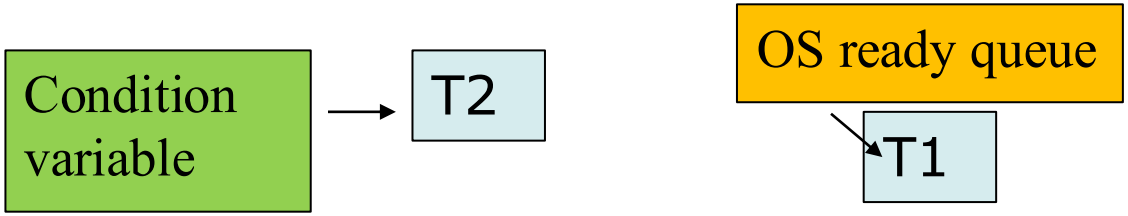
- Pthread_cond_init() sets up a queue for a condition variable




- Pthread_cond_wait() places a thread to a queue.
  Note the associated lock is released



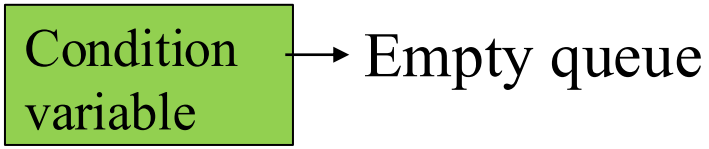

- Pthread_cond_signal() wakes up a thread and places into OS ready queue for future execution



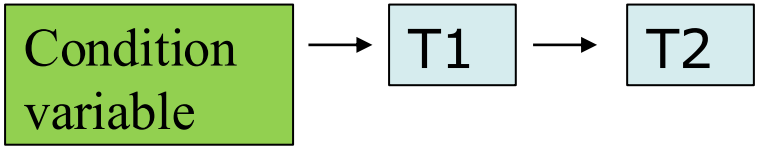

- When T1 is selected to execute, Pthread_cond_wait() resumes and the associated lock is reacquired.

# How to implement Pthread_cond_broadcast()

- Pthread_cond_init() sets up a queue for a condition variable




- Pthread_cond_wait() places a thread to a queue.
  Note the associated lock is released

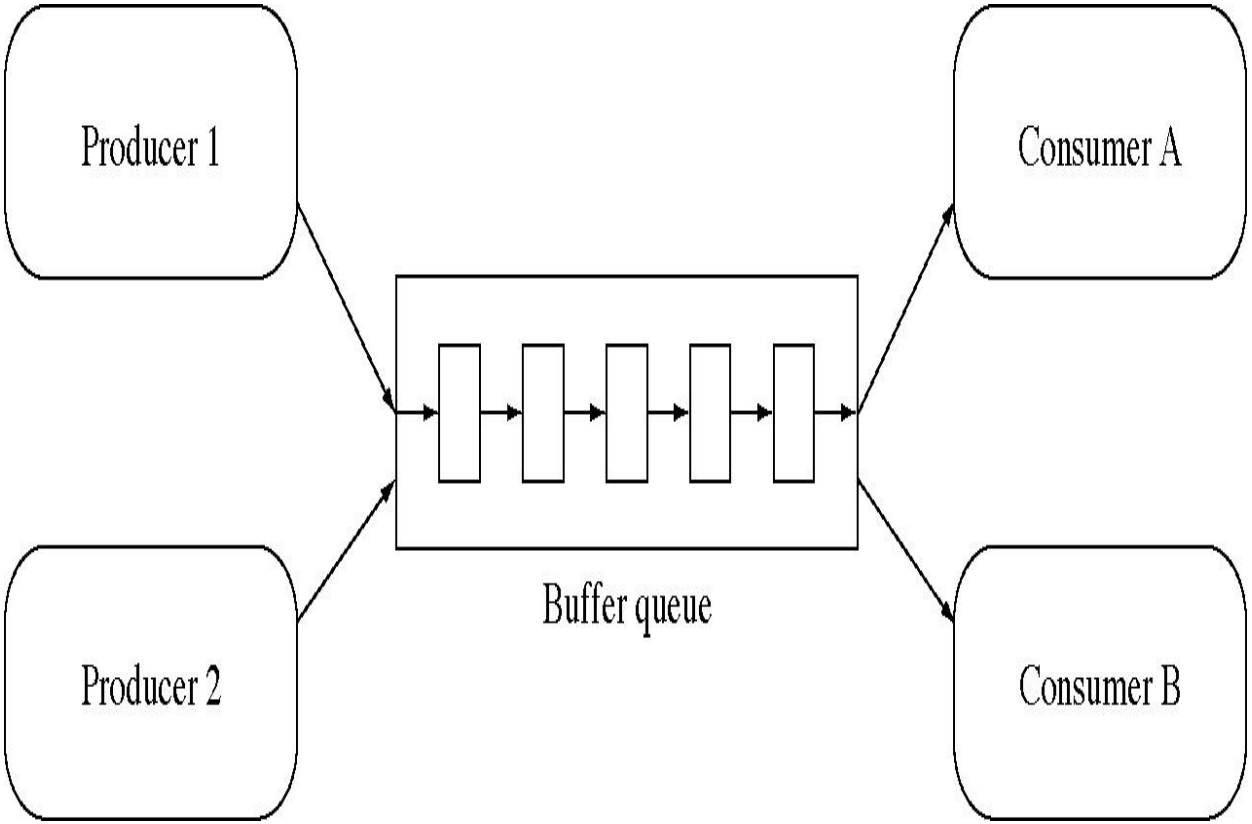



- Pthread_cond_broadcast() wakes up all threads and places into OS ready queue for future execution


- When T2 is selected to execute, Pthread_cond_wait() resumes and the associated lock is reacquired.

# Application example: Condition variables for in producer-consumer problem

Producer deposits data in a buffer  for others to consume




A generalization from the previous message-sending example:
Many producers and many consumers. Assume buffer has enough space

# First version for consumer-producer problem with unbounded buffer

- int avail=0;  // # of data items available for consumption
- Consumer  thread:

```
while (avail <=0); //wait
 Consume next item;  avail = avail-1;
```

  - *Producer thread:*

```
Produce next item;  avail = avail+1;
//notify an item is available
```

# Condition Variables for consumer-producer problem with unbounded buffer

- int avail=0;  // # of data items available for consumption
- Pthread mutex  m and condition cond;
- Consumer  thread:

```
multex_lock(&m)
 while (avail <=0) Cond_Wait(&cond, &m);
  Consume next item;  avail = avail-1;
 mutex_unlock(&m)
```

  - *Producer thread:*

```
mutex_lock(&m);
Produce next item;  availl = avail+1;
Cond_signal(&cond); //notify an item is available
mutex_unlock(&m);
```

# What happens if we do not put mutex_lock?

- int avail=0;  // # of data items available for consumption
- Pthread mutex  m and condition cond;
- Consumer  thread:

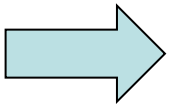
```
multex_lock(&m)
 while (avail <=0) Cond_Wait(&cond, &m);
  Consume next item;  avail = avail-1;
mutex_unlock(&m)
```

  - *Producer thread:*

```
mutex_lock(&m);
Produce next item;  availl = avail+1;
Cond_signal(&cond); //notify an item is available
mutex_unlock(&m);
```

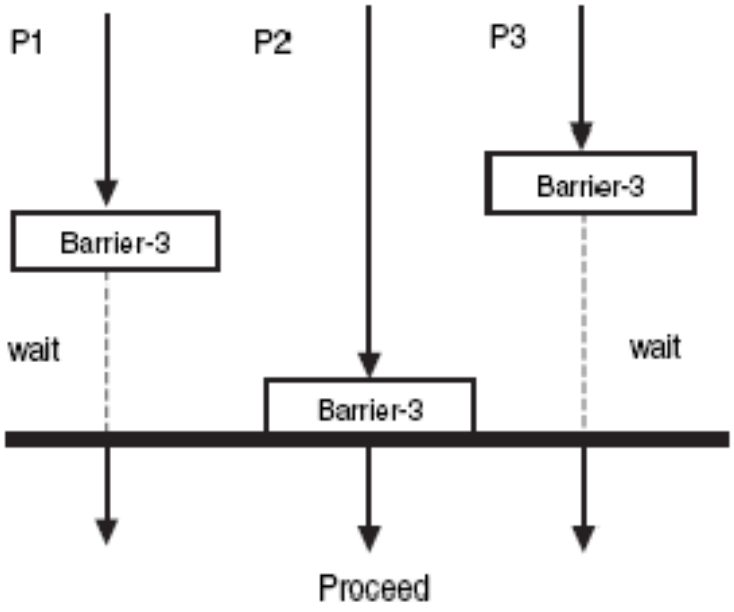
# SYNCHRONIZATION

1. **Mutex (lock)**
2. **Conditional Variables**
3. **Barriers**



# Barriers

- Synchronize the threads to make sure that they all are at the same point in a program is called a barrier.
- No thread can cross the barrier until all the threads have reached it.

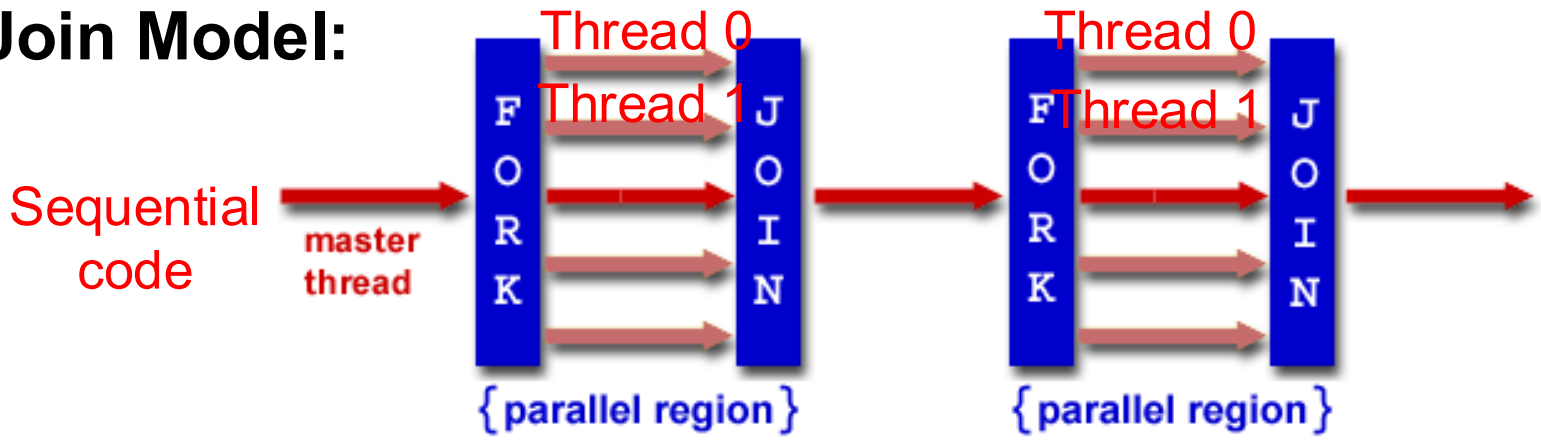



Pthread barrier is available in some OS

- Available on Linux
- Not on MAC OS

# Use of Barrier for Parallel Execution with Fork-Join Parallelism

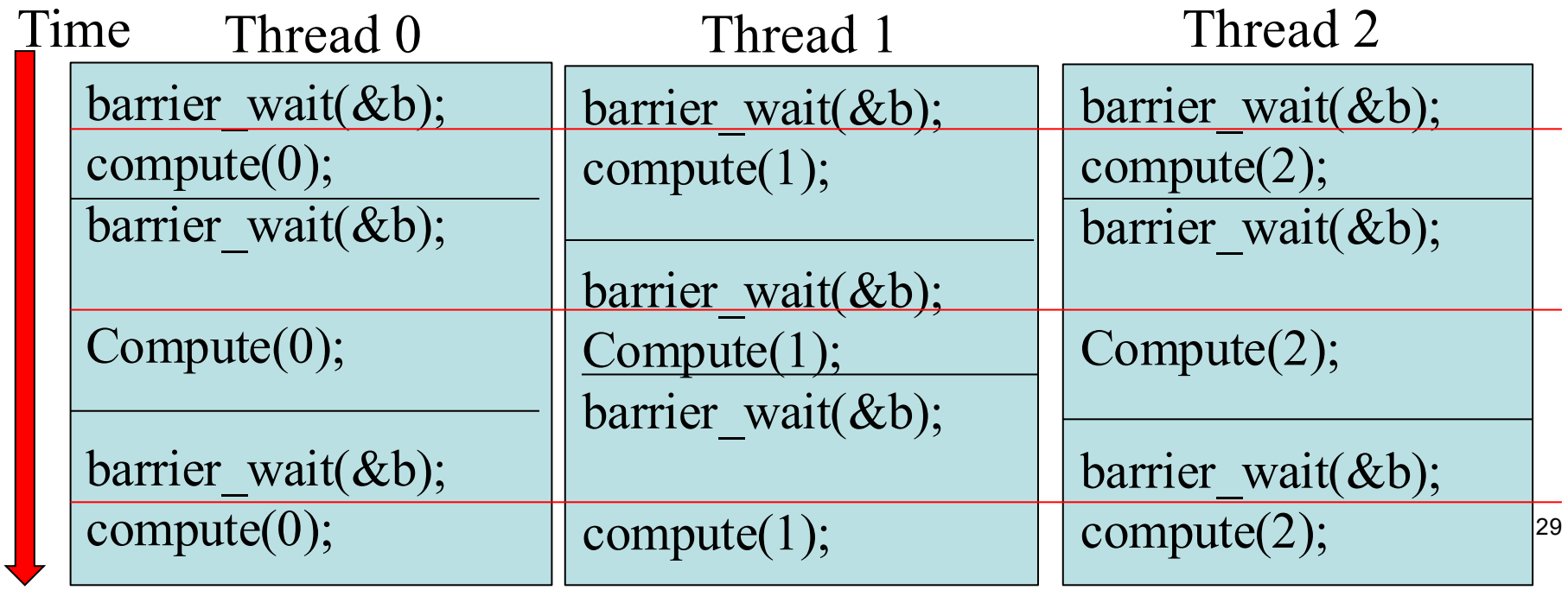
- **Fork - Join Model:**




- Begin as single thread and executes sequentially until a parallel region construct is encountered
  - *FORK:* Master thread creates a team of parallel slave threads
  - Statements in program that are enclosed by the parallel region construct are executed in parallel among the various threads
  - *JOIN:* When the team threads complete the statements in the parallel region construct, they synchronize (e.g. with a barrier), leaving only the master thread

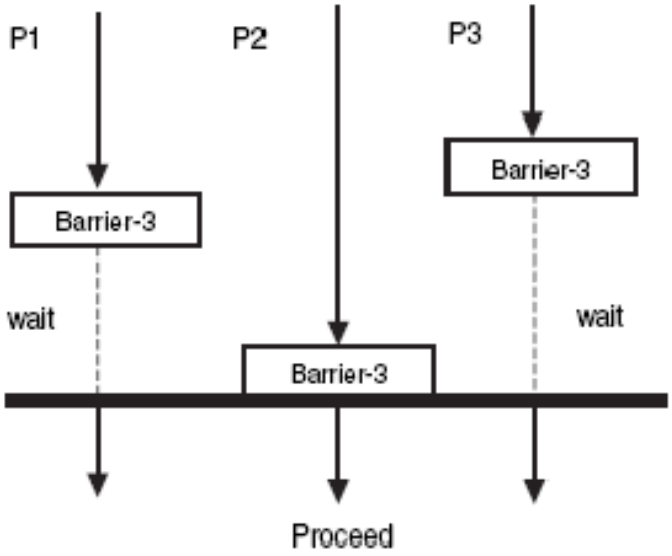
# Example of running 3 threads with Pthread barrier

```
pthread_barrier_t b;
pthread_barrier_init(&b, NULL, 3);
for (i=0; i<3; i++){
      pthread_barrier_wait( &b);
      compute(my_rank);
}
```



# How to implement a Barrier using conditional variables?

- Create a counter when the first thread calls a barrier.
- Keep a counter on how many threads have enterer the barrier call.
- No thread can move forward if the counter is not equal to the total number of threads. They have to wait.
- All threads can leave from the waiting status when the counter = the number of threads used in the system.



# Implementing a barrier with condition variable: Draft Idea

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread_work(. . .) {
    . . .
    /* Barrier */
    pthread_mutex_lock(&mutex);
    counter++;
    if (counter == thread_count) {
        counter = 0;
        pthread_cond_broadcast(&cond_var);
    } else {
        while (pthread_cond_wait(&cond_var, &mutex) != 0);
    }
    pthread_mutex_unlock(&mutex);
    . . .
}
```

Keep a counter on #threads entered in a barrier

Protect counter checking/operation

Set up a condition of thread waiting

counter++ for all threads

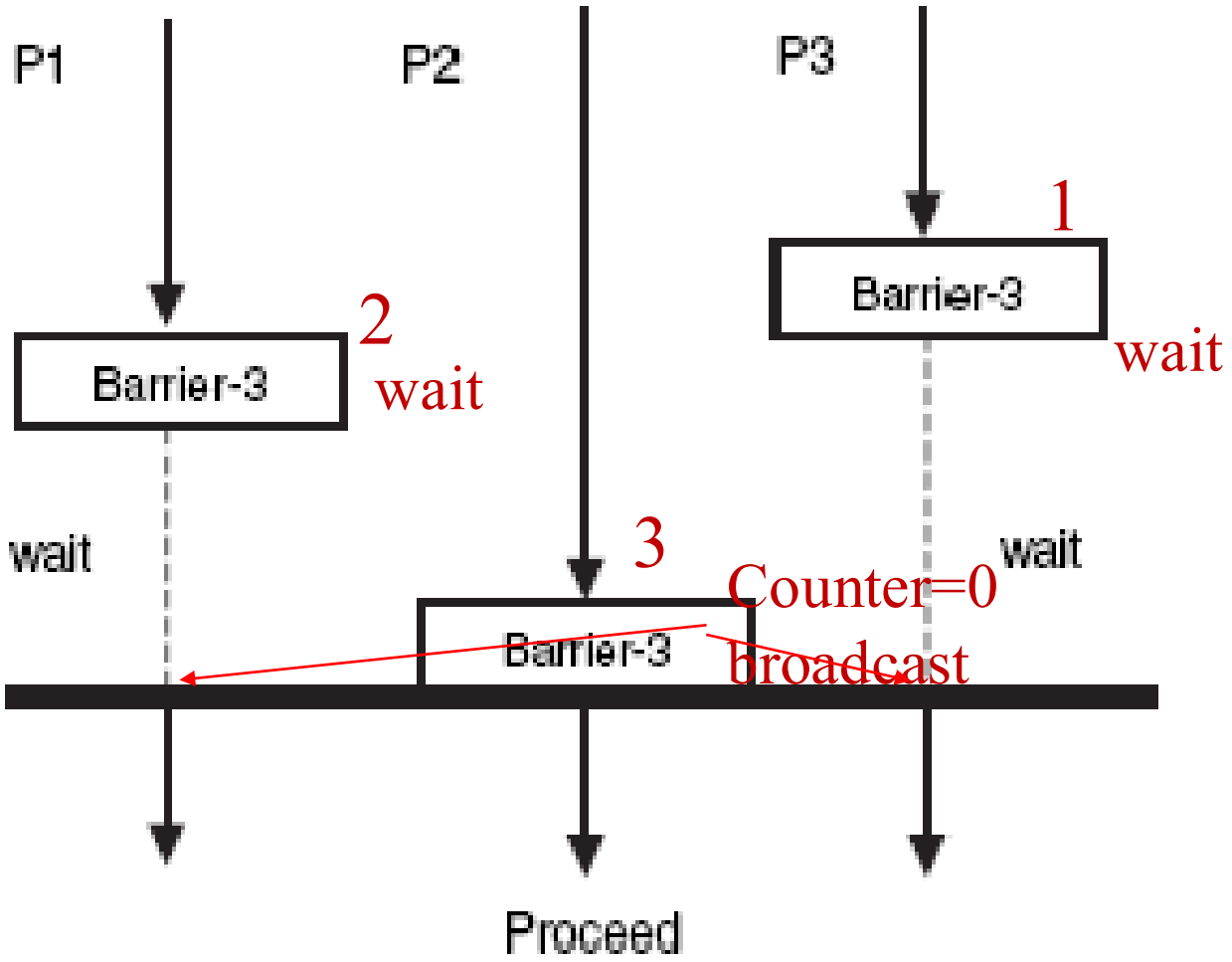
Last thread resets counter
Broadcast to wake

Not last thread:
cond_wait

When this thread wakes up, can it continue?
Can it possibly enter another barrier round?

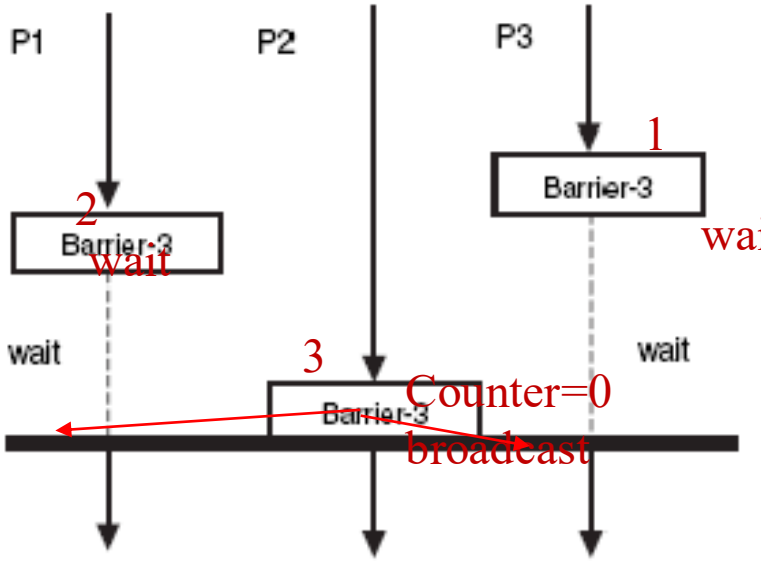
# Flow of threads in barrier synchronization

- Barrier counter =0.



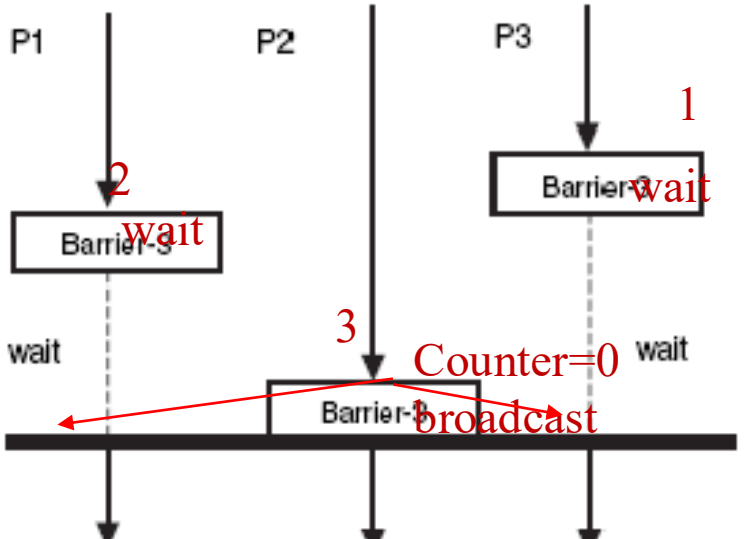
# Example of barrier flow and design issue

- Barrier counter =0.




Round 0

- Barrier counter =0.

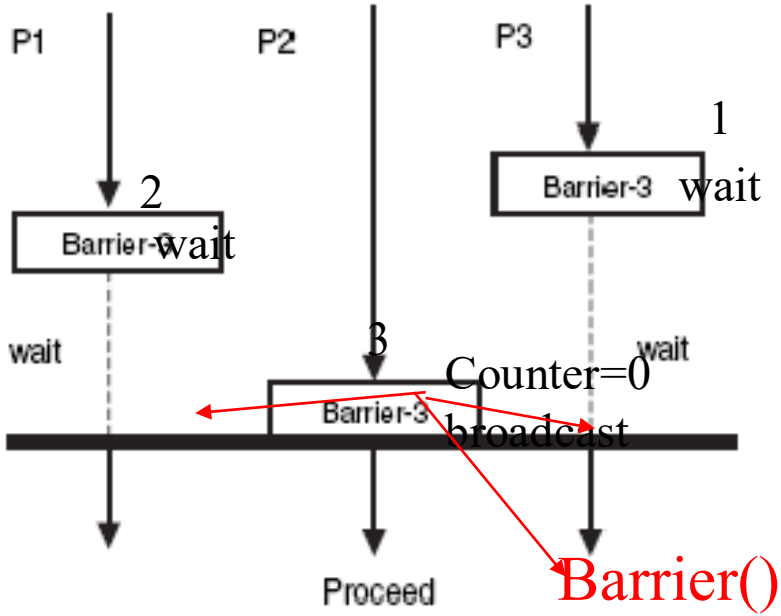



Round 1

Barrier counter begins with 0 and ends with 0 after all threads have arrived at current barrier round

# Barrier design challenge

- Ensure that all threads have left the previous barrier round.
  - Without this, a thread can mistakenly pass through current barrier round while state still being used by previous round, causing improper synchronization




Round 0 (even)

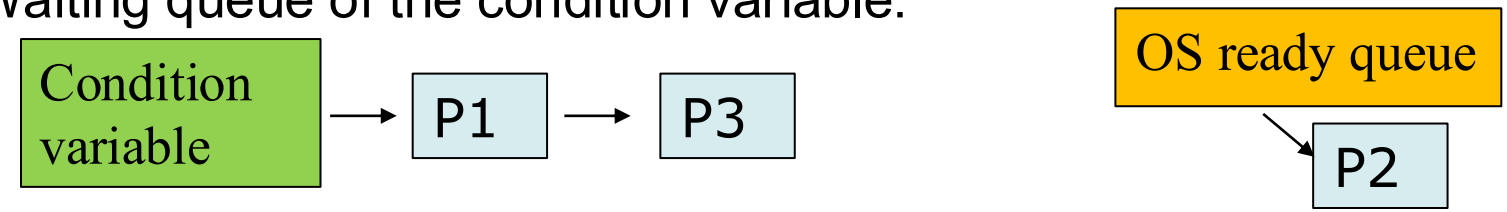
Odd and even barrier rounds can overlap in time

—Some threads (e.g. P1) may still be exiting the $k^{th}$ barrier

—Other threads (e.g. P3) may be entering the $k+1^{st}$ barrier . P3 may escape Round 1 based on the broadcasted message from P2 at Round 0.

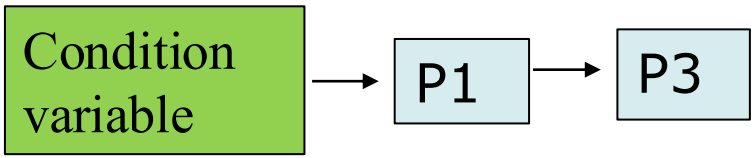
# Can Pthread_cond_broadcast() interleave with Pthread_cond_wait()? Possibility exists

- Waiting queue of the condition variable:




- Thread P2 calls Pthread_cond_broadcast() which moves waiting threads out one by one

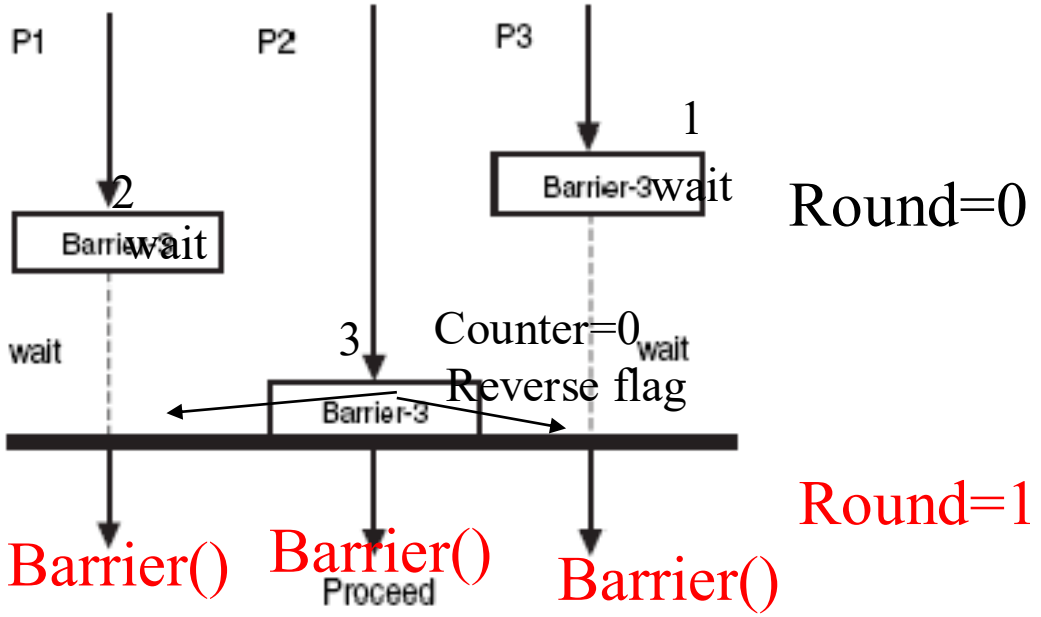

- P3 is selected to execute and enter next barrier round




- Thread P2 continues Pthread_cond_broadcast() which moves waiting threads out one by one. P3 is moved again and escapes next round

# Solution: Count the round number

- Termination of one round legitimizes next round
  - Last thread of a barrier increases the round number
  - Threads need to wait if the round number has not changed




**All threads** : counter++:

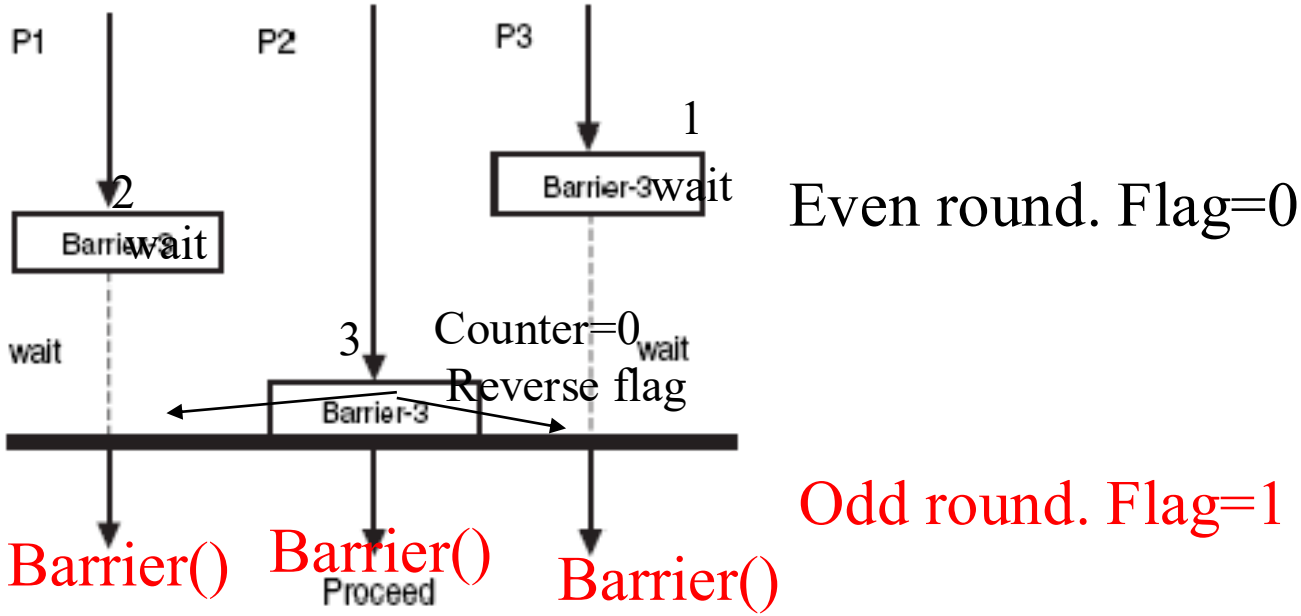
**Not last thread**: Loop to do cond_wait if #round is not changed

**Last thread when counter becomes 3:**

Set counter=0. Round++. Broadcast.

# Equivalent Solution: Sense Reversal

- Detect termination of one round when #round number changes odd/even. Implement it as a flag
  - Barriers wait until flag changes from 1 to 0 or from 0 to 1




**All threads** : counter++:

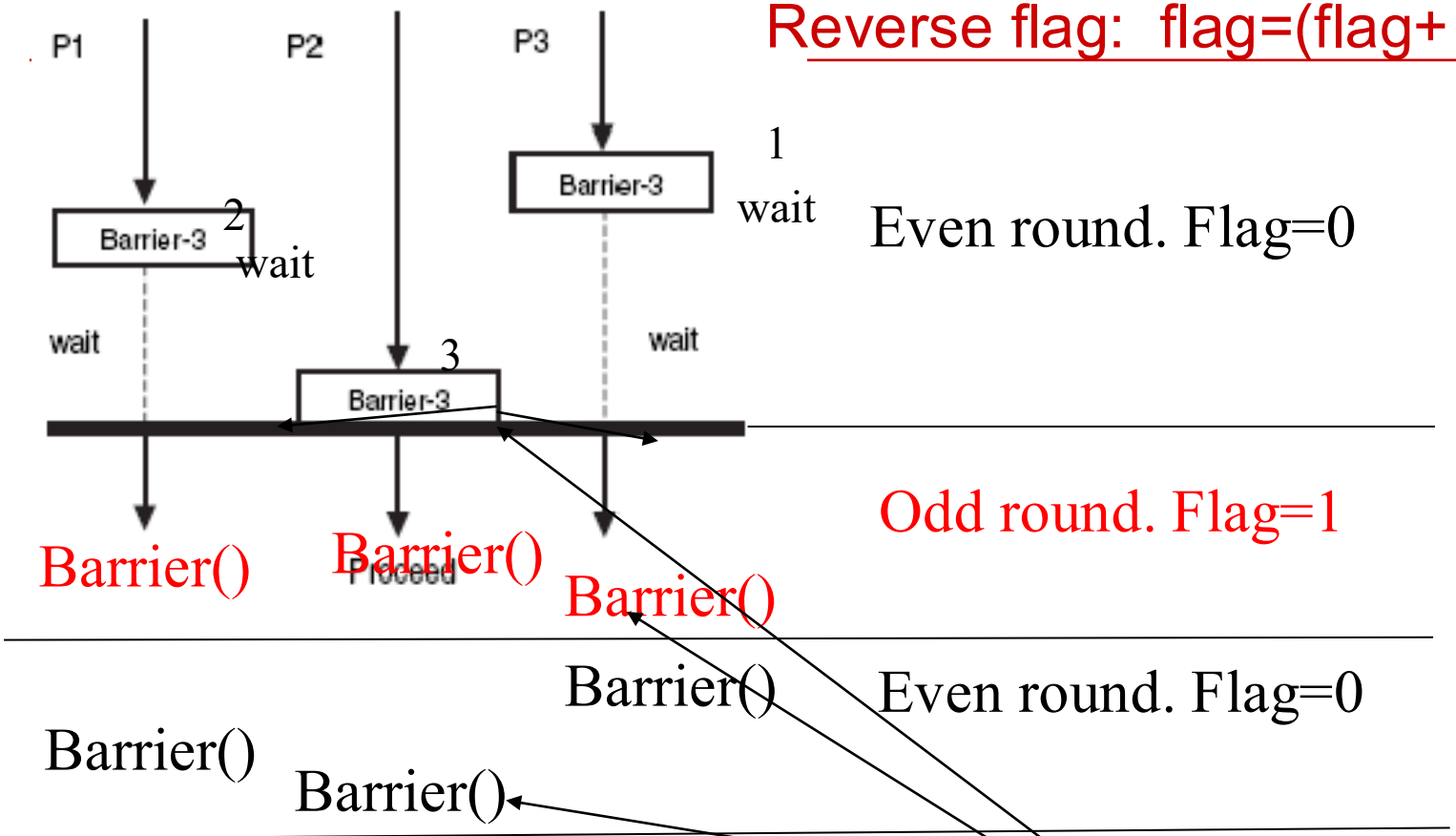
**Not last thread**: Loop to do cond_wait if flag is not changed

**Last thread when counter becomes 3:**

Set counter=0. Reverse flag. Broadcast.

# Example: Sense Reversal

Reverse flag: flag=(flag+1) %2




**All threads**: counter++;

**Not last thread**:

Do { cond_wait }

while flag is not changed;

**Last thread:**

Set counter=0;

Reverse flag;

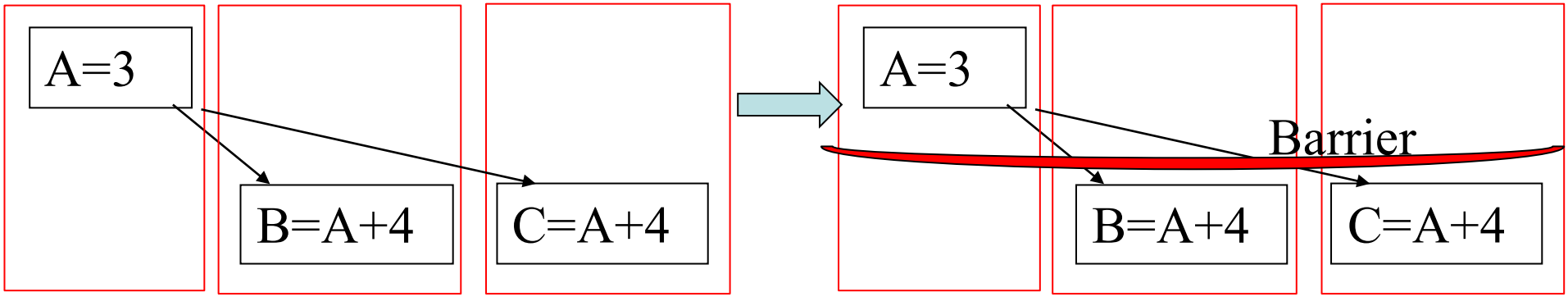
Broadcast;

# Concluding Remarks on Pthreads

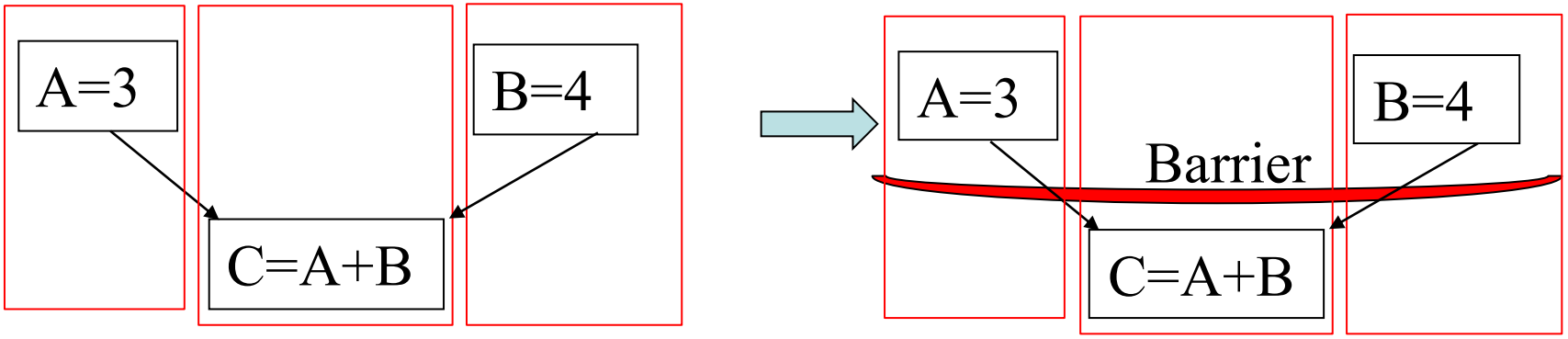
- In a Pthread program, all the threads have access to global variables, while local variables usually are private to each thread running the thread function.
- When multiple threads access/update a shared resource without controlling, we have a race condition.
  - A critical section is a block of code that updates a shared resource that can only be updated by one thread at a time
  - A mutex arranges for mutually exclusive access to a critical section.
- Semaphore & Condition variables
  - more powerful synchronization primitives.
- A barrier is a point in a program at which the threads block until all of the threads have reached this point.
- PA2b: Pthread shared-memory programming

# How to synchronize a dependence with a barrier

- Synchronize fork-shaped dependence




- Synchronize join-shaped dependence



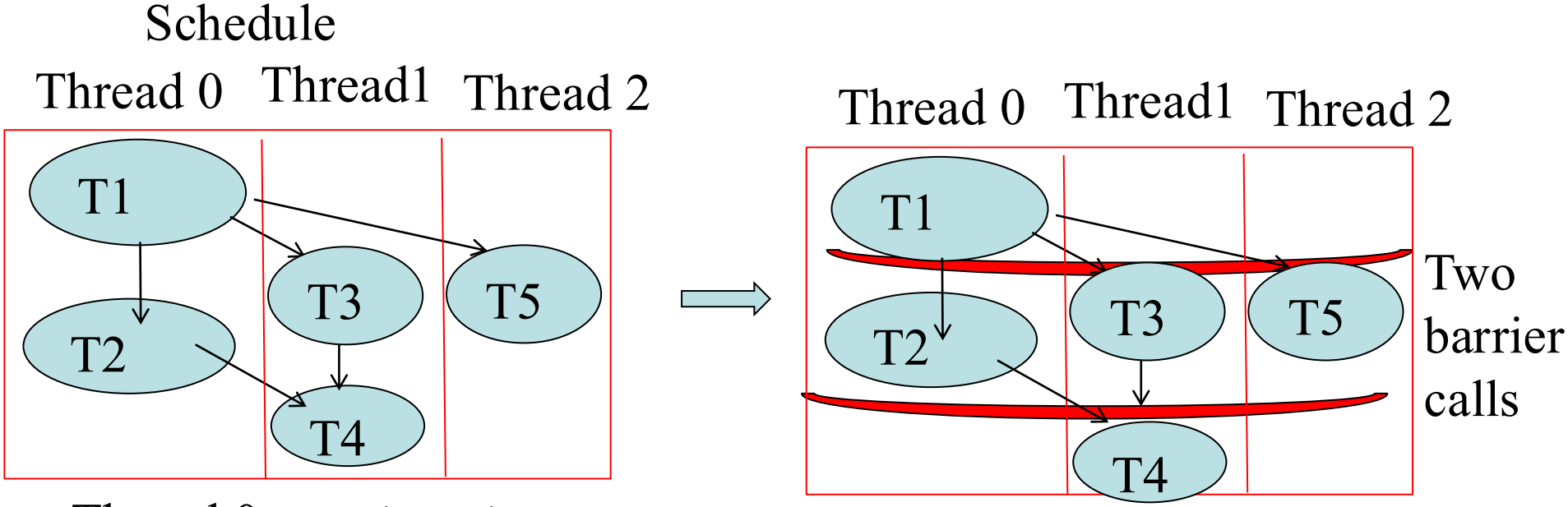
# Barrier vs. other synchronization primitives

- **Pros:** Convenient/clean way to synchronize threads
  - Suitable for fork/join parallelism with lots of dependence
- **Cons:** Very expensive overhead
  - Minimize the use of barrier calls

**When to use Barriers or other primitives?**

- **Critical section**
  - Out-of-order thread execution, but one thread at a time
  - Mutex
- **Consumer-producer dependence**
  - One-to one, many-to-one, many-to-many asynchronous
  - Condition variable with a mutex.
- **Fork parallelism or join parallelism**
  - Barrier

# How many barrier calls are needed?



# Tips for PA2

- Only barriers can be used
- Minimum number of barrier calls

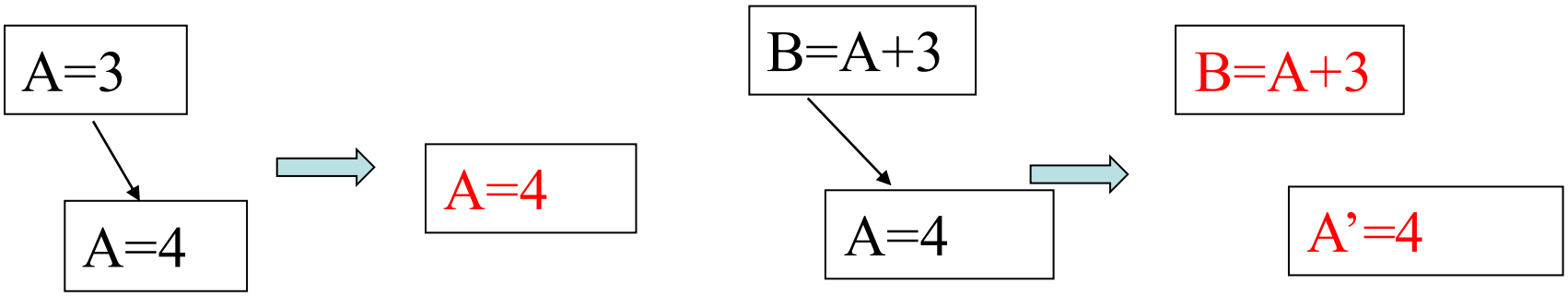
```
For k = 1 to t
        y= d +Ax
        error =||y-x||
        x=y
        if error < 0.0001
                break
Endfor
```

```
Naive solution:
For k = 1 to t
        barrier();
        y= d +Ax
        barrier();
        error =||y-x||
        barrier();
        x=y
        barrier();
        if error < 0.0001
                break
        barrier();
```

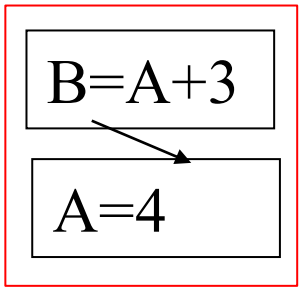
- **Steps to find a minimum number of barrier calls**
  - Derive a task graph which captures reasonable parallelism
    - Consider ”if error<0.0001 then break” as a conditional dependence
  - Map tasks to threads and schedule
  - Place barrier calls . If needed, simplify dependence.

# Simplify a dependence graph

- **If T1 → T2, and two tasks are executed on different threads, we need a synchronization.**
  - Remove anti or output dependence
  - Rename by allocating extra space if space complexity is still reasonable




- **If T1 → T2, and execute two tasks on the same thread, naturally enforce a synchronization.**



# Ex: How many barrier calls are needed?

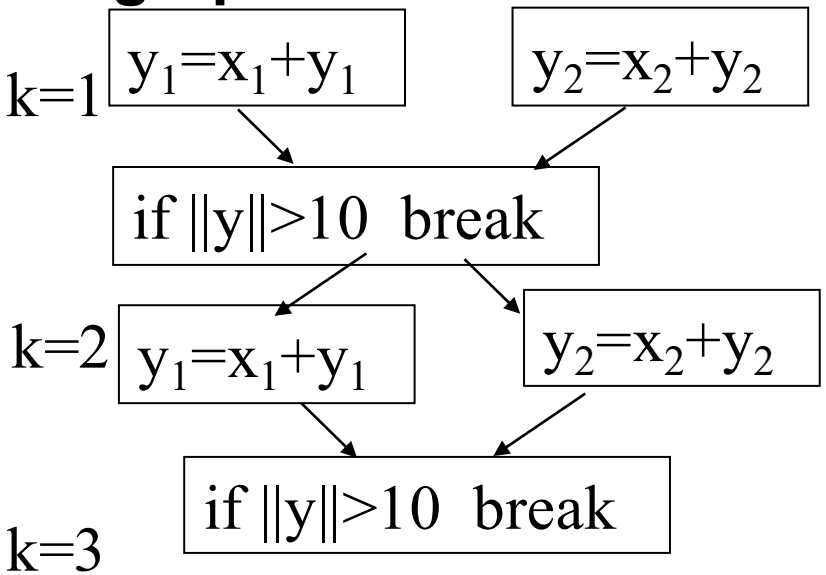
- x, y are vectors

```
For k = 1 to t
        y=x+y
        if ||y|| >10 then
                break
Endfor
```

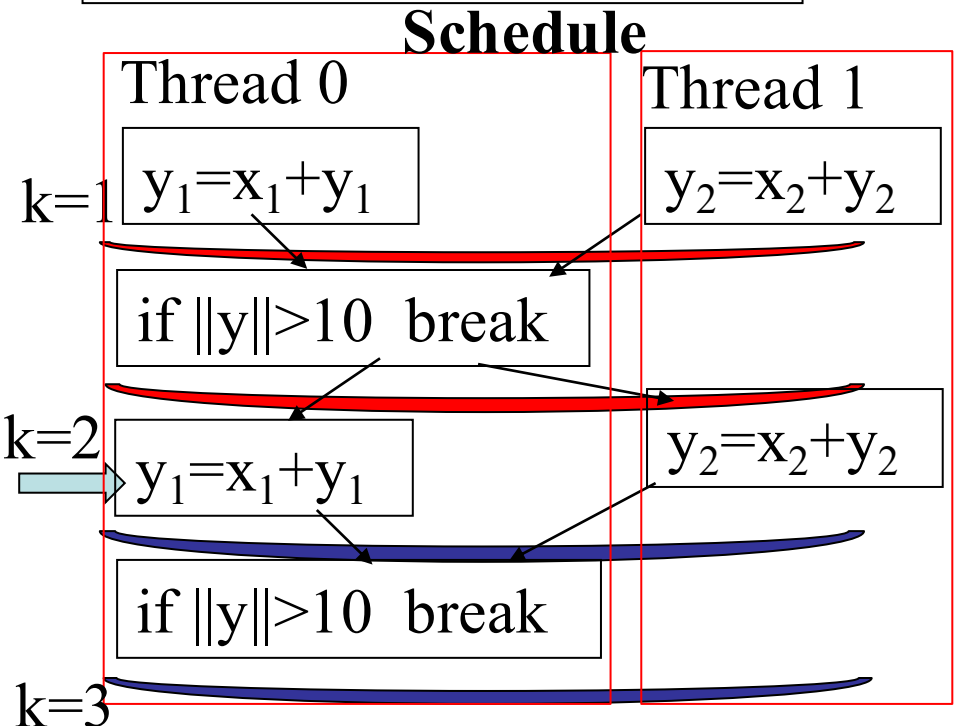
## Program partitioning

```
For k = 1 to t
        for i=1 to n
            yi=xi+yi
        Endfor
        if ||y|| >10 then
                break
Endfor
```

## Task graph with t=2




## Schedule

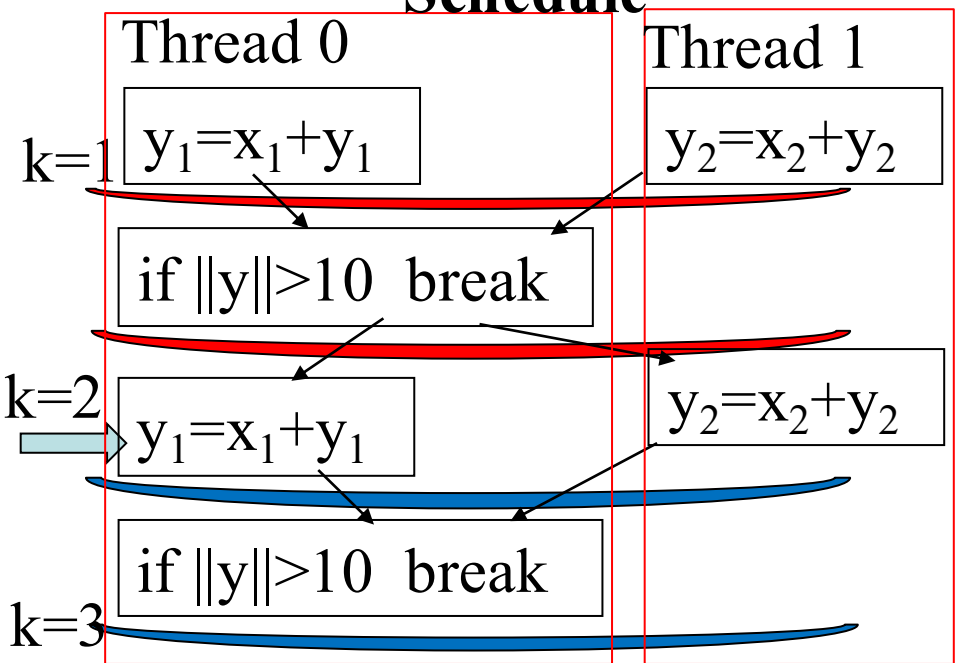


# Ex: How many barrier calls are needed?

- x, y are vectors

```
For k = 1 to t
        y=x+y
        if ||y|| >10 then
                break
Endfor
```

**Schedule**




**Parallel code**

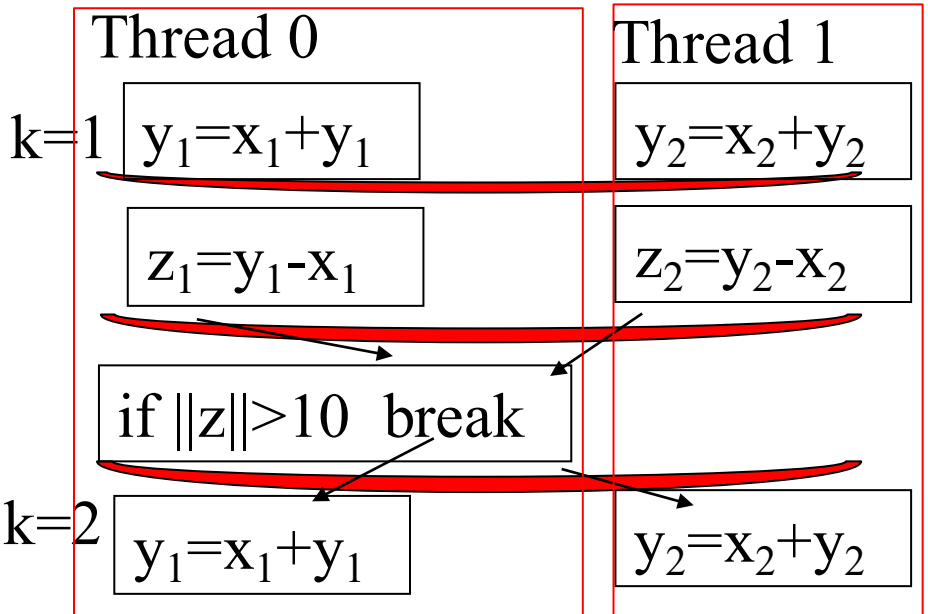
```
For k = 1 to t
        Parallel for i=1 to n
          yi=xi+yi
        Endfor
        barrier();
        if ||y|| >10 then
                break
        barrier();
Endfor
```

# Ex 2: How many barrier calls are needed?

**Are 3 barrier calls minimum?**

- x, y are vectors

```
For k = 1 to t
        y=x+y
        z=y-x
        if ||z|| >10 then
                break
Endfor
```



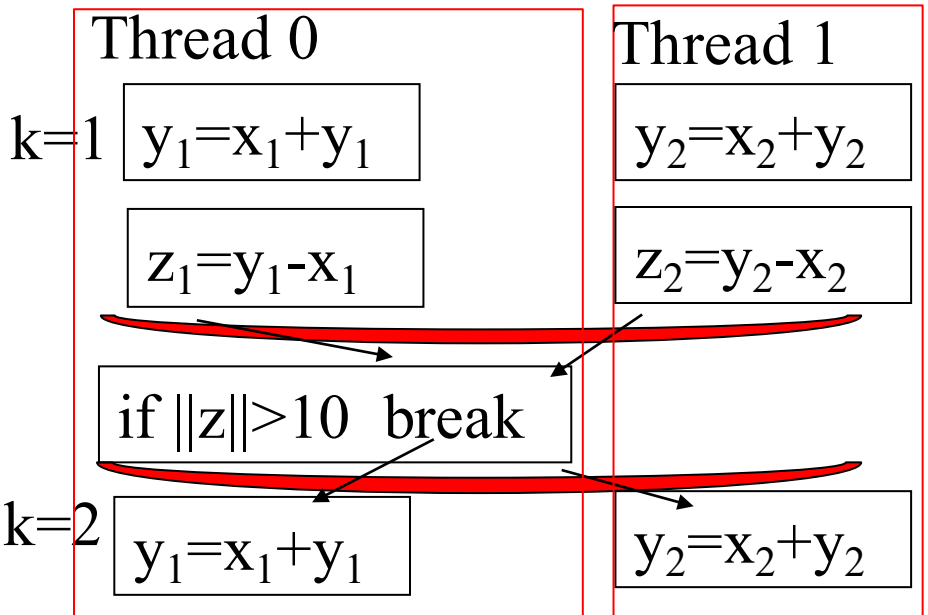

**Parallel code**

```
For k = 1 to t
        Parallel for i=1 to n
          yi=xi+yi
        Endfor
        barrier();
        Parallel for i=1 to n
                zi=yi -xi
        Endfor
        barrier();
        if ||z|| >10 then
                break
        barrier();
Endfor
```

# Ex 2: How many barrier calls are needed?

- x, y are vectors

```
For k = 1 to t
        y=x+y
        z=y-x
        if ||z|| >10 then
                break
Endfor
```




**Parallel code**

```
For k = 1 to t
        Parallel for i=1 to n
          yi=xi+yi
          zi=yi -xi
        Endfor
        barrier();
        if ||z|| >10 then
                break
        barrier();
Endfor
```

**2 barrier calls are minimum**

# Use of Threads in Python Packages

- **Compute-intensive Python packages use threads on CPU cores, with OpenMP or Pthreads implementation**

```
import time
import torch
x = torch.randn(1024, 1024)
y = torch.randn(1024, 1024)
threads = [2, 4, 8]
for t in threads:
    torch.set_num_threads(t)
    start= time.time()
    for i in range (0,100):
        z= torch.mm(x, y)
    print("the number of cpu threads: {}, time:
          {}".format(torch.get_num_threads(), time.time()-start))
```

Test parallel time of PyTorch matrix multiplication on a CSIL machine with 8 cores

# Use of Threads in Python PyTorch Machine Learning Package

```
import time
import torch
x = torch.randn(1024, 1024)
y = torch.randn(1024, 1024)
threads = [2, 4, 8]
for t in threads:
    torch.set_num_threads(t)
    start= time.time()
    for i in range (0,100):
        z= torch.mm(x, y)
    print("the number of cpu threads: {}, time:
          {}".format(torch.get_num_threads(), time.time()-start))
```

Output on a CSIL 8-core machine

```
$ python3 timethread1.py

#the number of cpu threads: 2, time:
4.336086988449097
#the number of cpu threads: 4, time:
2.210944652557373
#the number of cpu threads: 8, time:
1.1844263076782227
```

# Use of GPU in Python PyTorch

- **PyTorch use threads executed on GPUs if available**

Check if PyTorch uses GPU

```
>>> import torch
>>> torch.cuda.is_available()
```

```
#Perform some computation with GPU 0
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
X_train = X_train.to(device)
print(X_train.is_cuda)  #True if GPU is used
```

```
#Use another GPU
torch.cuda.set_device(0) # or 1,2,3
```