SUPSI

Lab: Introduction

Operating Systems

Amos Brocco, Lecturer & Researcher

14 settembre 2018

Objectives

- Understand the concept of thread
- Understand how to create and manage threads with Pthread in C
- Understand how to work with synchronization mechanisms in C

Browsing
- Get a rapid overview.

Reading
- Read it and try to understand the concepts.

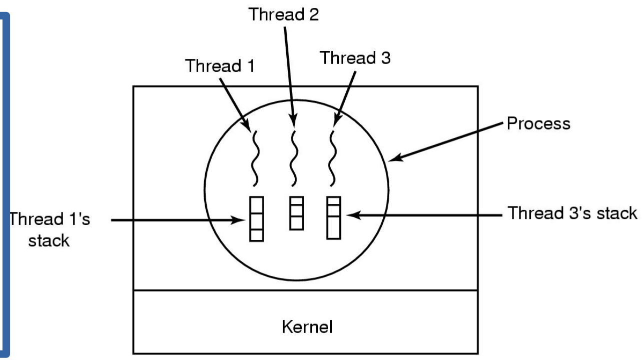
Studying
- Read in depth, understand the concepts as well as the principles behind the concepts.

You are also encouraged to try out (compile and run) code examples!

The active part of a process: threads

- A process can have one or more **threads** (or **paths**) **of execution** *
  - Threads in a process share some resources (→ concurrency problems)

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

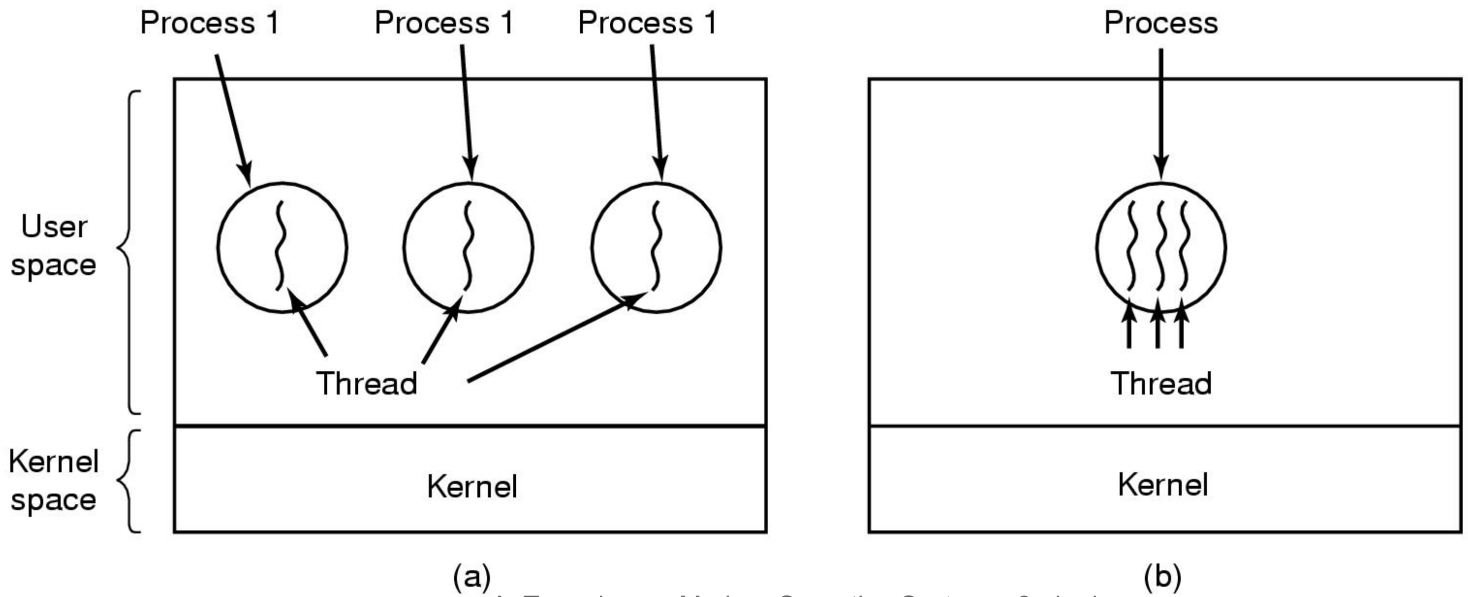


A. Tanenbaum, Modern Operating Systems, 2nd ed

  - When a process has multiple threads of execution we call it a **multi-threaded process**, otherwise it is called a **single-threaded process**

* typically simply referred to as **threads**

The active part of a process: threads

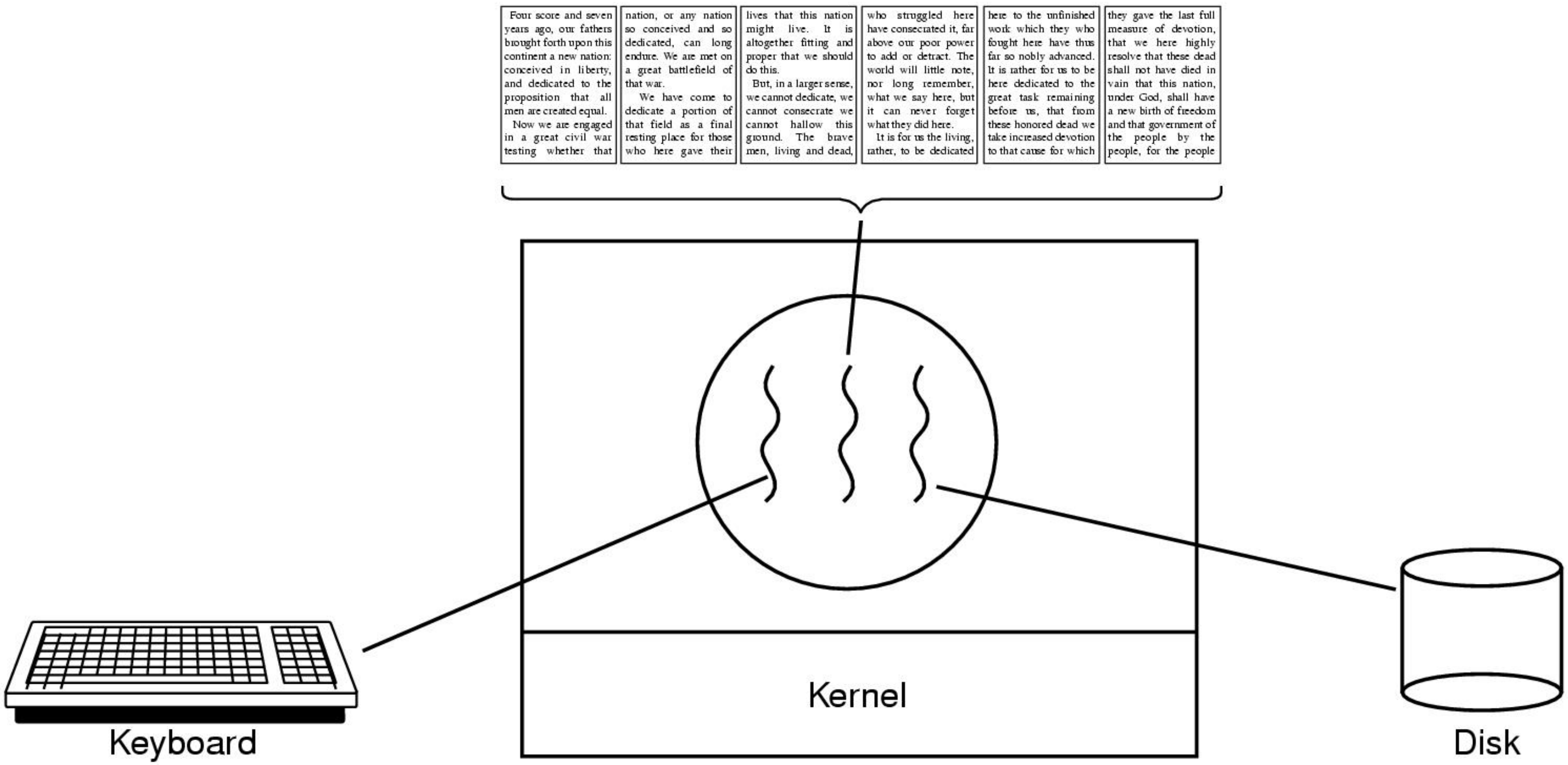


A. Tanenbaum, Modern Operating Systems, 2nd ed

Multiple single-threaded processes

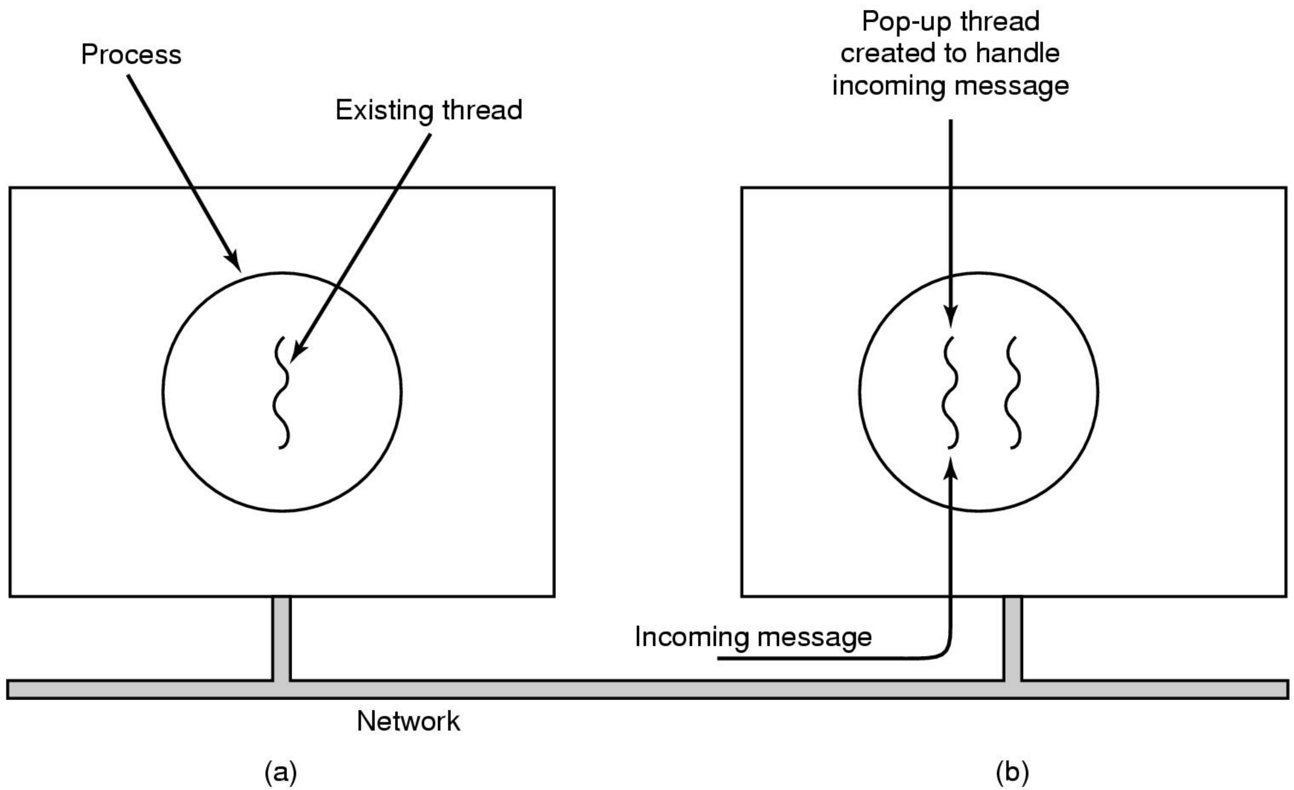
One multi-threaded process

Why multi-threading?



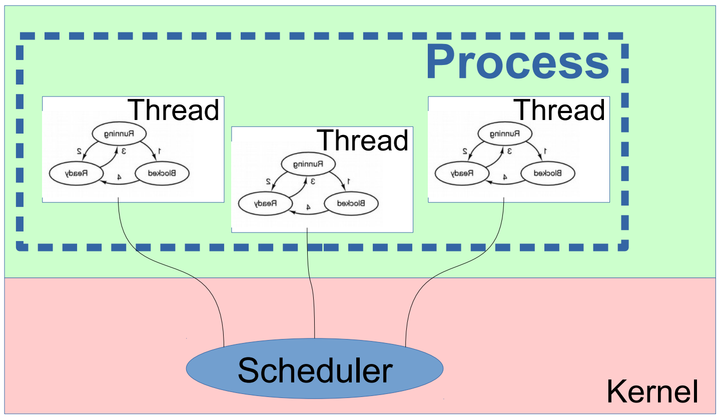
A. Tanenbaum, Modern Operating Systems, 2nd ed

Why multi-threading?

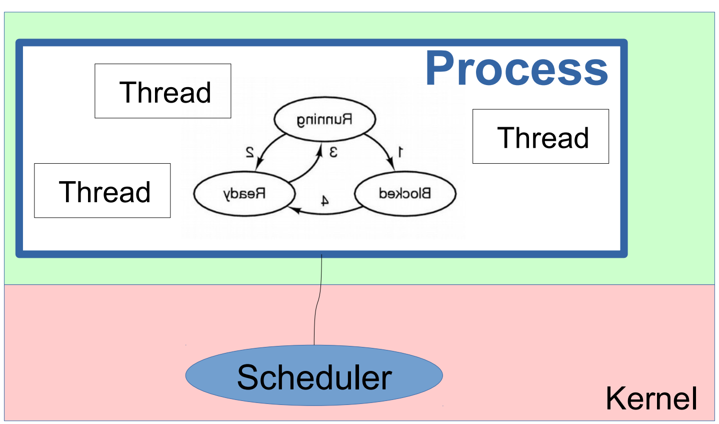


A. Tanenbaum, Modern Operating Systems, 2nd ed

Threads implementation



- **Kernel level threads**
  - “the kernel knows what threads are”
  - Thread scheduling is done by the kernel
  - If a thread blocks, other threads within the same process can continue executing
  - Note: kernel level threads still run in unprivileged (user) mode!



- **User level threads**
  - “the kernel doesn't know anything about threads”
  - Thread scheduling is done by the process
    - When the kernel schedules the process its threads are given a chance to run
  - If a thread blocks, the whole process (including other user threads) is blocked

Creating a thread (with pthread)

Compile and link with `-pthread`

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

- Each thread is associated with a **`pthread_t`** structure
- The “body” of the thread is defined by the **`start_routine`** procedure, which can receive parameters using the **`arg`** pointer
- The new thread is started immediately

Terminating a thread

- A thread terminates when `start_routine` returns
- A thread can explicitly terminate its execution and return a value:

```
#include <pthread.h>

void pthread_exit(void *retval);
```

- A thread can also ask another thread to terminate:

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

  - Exit happens as soon as possible (when a *cancellation point* is reached)

Cancellation point

```
#include <pthread.h>

void pthread_testcancel(void);
```

- With this procedure we can define a cancellation point where the thread will respond to pending cancellation requests:
  - It is possible to ignore the request with `pthread_setcancelstate`
  - Many functions provide pre-defined cancellation points (see `man pthreads`)

Who gets the exit value?

- A thread can wait for another thread to terminate and obtain its exit value:

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

- The return value can be obtained from **`retval`**
- Only threads which are **`JOINABLE`** *(default) can be waited for, **`DETACHED`** ones can't be waited for:
  - A joinable thread waits until the join before being freed
  - **`pthread_join`** returns 0 if the thread terminates correctly, a negative value in case of errors

* see `man pthread_attr_init`

Example

```c
#include <pthread.h>
#include <stdio.h>

void *mythread (void *name)
{
    printf("Hello, I'm a new thread %s\n", (char*) name);
    sleep(3);
    return (void*) 42;
}

int main()
{
    pthread_t thread;
    int i;

    pthread_create(&thread, NULL, &mythread, "Alfred");
    sleep(2);
    pthread_join(thread, (void**) &i);
    printf("Return value is %d\n", i);
    return 0;
}
```

Example (detached thread)

```c
#include <pthread.h>
#include <stdio.h>

void *mythread (void *name)
{
    printf("Hello, I'm a new thread %s\n", (char*) name);
    sleep(3);
    return (void*) 42;
}

int main()
{
    pthread_t thread;
    pthread_attr_t attr;
    int i;

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    pthread_create(&thread, &attr, &mythread, "Alfred");
    sleep(2);
    pthread_join(thread, (void**) &i); // Error, cannot join detached thread
    printf("Return value is %d\n", i); // Return value is bogus
    return 0;
}
```

Example (alternate stack)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define STACK_SIZE 2<<15

void *mythread (void *name)
{
    printf("Hello, I'm a new thread %s\n", (char*) name);
    sleep(3);
    return (void*) 42;
}

int main()
{
    pthread_t thread;
    pthread_attr_t attr;
    int i;
    void *sp;
    pthread_attr_init(&attr);
    sp = malloc(STACK_SIZE);
    pthread_attr_setstack(&attr, sp, STACK_SIZE);
    pthread_create(&thread, &attr, &mythread, "Alfred");
    sleep(2);
    pthread_join(thread, (void**) &i); // Error, cannot join detached thread
    free(sp);
    printf("Return value is %d\n", i); // Return value is bogus
    return 0;
}
```

Example (pthread_exit)

```c
#include <pthread.h>
#include <stdio.h>

void *mythread (void *name)
{
    printf("Hello, I'm a new thread %s\n", (char*) name);
    sleep(3);
    pthread_exit((void*) 13);
    return (void*) 42;
}

int main()
{
    pthread_t thread;
    int i;

    pthread_create(&thread, NULL, &mythread, "Alfred");
    sleep(2);
    pthread_join(thread, (void**) &i);
    printf("Return value is %d\n", i);
    return 0;
}
```

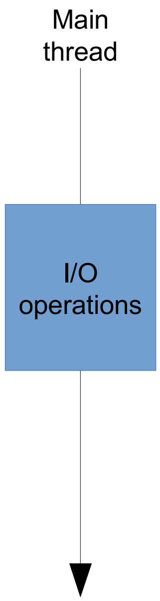
Example (pthread_testcancel)

```c
#include <pthread.h>
#include <stdio.h>

void *mythread (void *arg)
{
    printf("Thread start\n");
    while (1) {
        pthread_testcancel(); /* Cancellation point */
    }
    printf("Exiting!\n"); /* This code is never executed */
    return (void*) 42; /* This code is never executed */
}
int main()
{
    pthread_t thread;
    int i;
    pthread_create(&thread, NULL, &mythread, NULL);
    sleep(3);
    pthread_cancel(thread);
    sleep(4);
    pthread_join(thread, (void**) &i);  /* The return value 'i' is PTHREAD_CANCELED (-1)  */
    printf("Return value %d\n", i);
    return 0;
}
```

Use case for kernel threads: I/O in a separate thread

- I/O operations normally block the execution (until data is read/written)

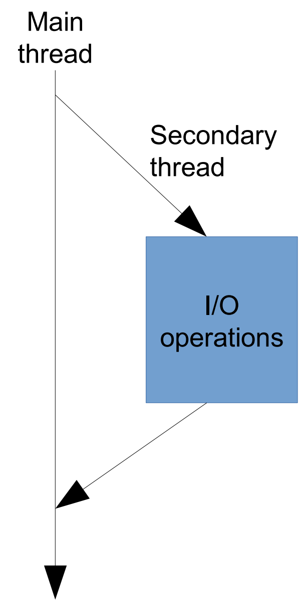
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define NBYTES 10000

void main(void)
{
        char* buffer[NBYTES];
        unsigned int bytes;

        FILE* file = fopen("/var/log/syslog", "r");
        bytes = read(fileno(file), buffer, NBYTES);
        printf("Synchronous read, got %d bytes.\n", bytes);
        close(fileno(file));
}
```



Use case for kernel threads: I/O in a separate thread

- I/O operations can be moved to a separate thread

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#define NBYTES 10000
FILE* file;
char* buffer[NBYTES];
unsigned int bytes;

void* reader() {
       bytes = read(fileno(file), buffer, NBYTES);
}

void main(void) {
       pthread_t thread_reader;
       file = fopen("/var/log/syslog", "r");
       pthread_create(&thread_reader, NULL, &reader, NULL);
       printf("... reader thread is doing its work...\n");
       sleep(5);
       pthread_join(thread_reader, NULL);
       printf("Read finished, got %d bytes.\n", bytes);
       close(fileno(file));
}
```



Race condition

- When working with shared resources (for example, in a multi-thread program with global shared variables) if the correctness of the output depends on the sequence or timing of execution of each task there is a **race condition**
  - the problem originates from **concurrent access** to a shared resource by multiple processes or threads
    - to solve this situation we first have to identify critical sections of the program...

Critical regions

Critical regions are part of the source code which access shared resources

```c
int account_balance = 100;
```

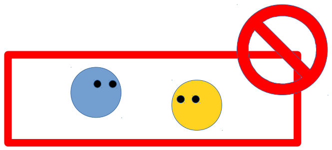
```c
void* thread1(void* arg)
{
    int wallet = 0;
    if (account_balance >= 100) {
        account_balance -= 100;
        wallet = 100;
    } else {
        printf("Cannot draw money\n");
        pthread_exit((void*)-1);
    }
    assert(account_balance >= 0);
    account_balance += 50;
    printf("Balance: %d\n",
                account_balance);
}
```

```c
void* thread2(void* arg)
{
    account_balance += 50;

    if (account_balance >= 100) {
        account_balance -= 100;
        printf("Pay bills\n");
    } else {
        printf("Not enough money\n");
    }
    assert(account_balance >= 0);
}
```

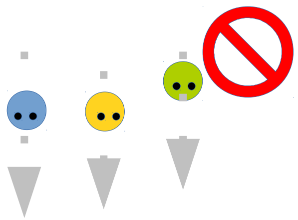
How to avoid race conditions

- Two processes/threads must not be simultaneously inside critical regions of the same resource

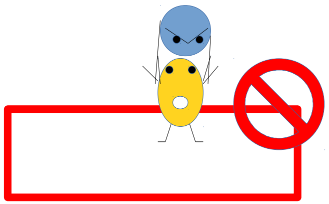
  → mutual exclusion



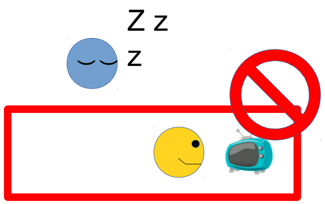
- The program must operate correctly no matter what is the speed of execution, number of processes/threads, scheduling policy or timing



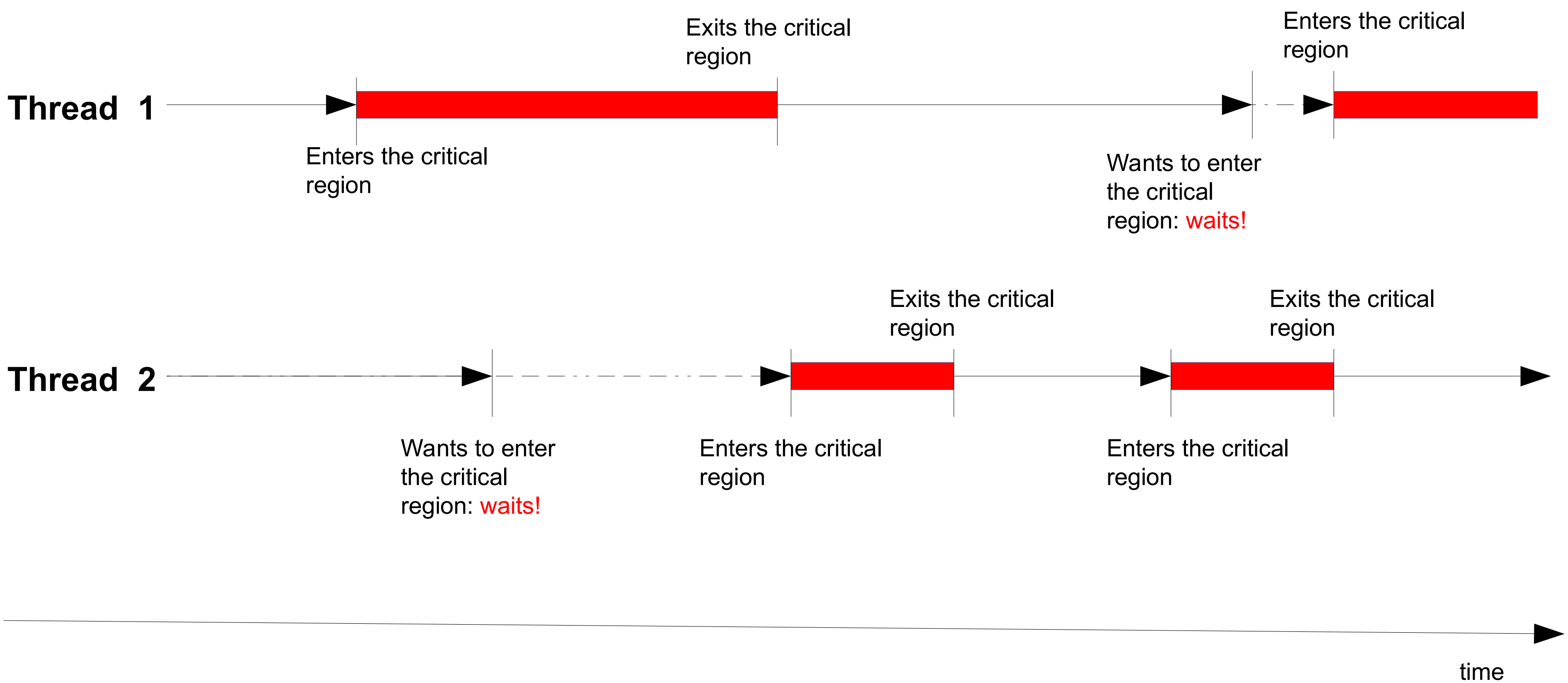
- When outside critical regions a thread/process cannot block another thread (for example, by preventing it from entering a critical region)



- No thread should have to wait indefinitely before entering a critical region (should not starve)



Example


Thread 1
Exits the critical region
Enters the critical region
Enters the critical region
Wants to enter the critical region: waits!
Thread 2
Exits the critical region
Exits the critical region
Wants to enter the critical region: waits!
Enters the critical region
Enters the critical region
time

Atomicity

- The mechanisms that we will study to ensure **mutual exclusion** work on the principle of **atomicity** and on **atomic operations**/instructions
  - an operation is atomic if it completes **in a single step** relative to other threads (i.e. cannot be interrupted)
  - no other thread can observe an inconsistent state (for example, half-complete modification)

Achieving mutual exclusion: a **non-solution**

- **A variable named lock**

```
int lock = 0;

void* thread1(void* arg)
{
    for(;;) {
        while (lock != 0);
        lock = 1;
        do_things();
        lock = 0;
    }
}

void* thread2(void* arg)
{
    for(;;) {
        while (lock != 0);
        lock = 1;
        do_things();
        lock = 0;
    }
}
```

Achieving mutual exclusion: some valid **solution**

- **Sequential execution**
  - If threads are executed sequentially (one at at time, from beginning to the end) there cannot be two of them at the same time in a critical section

- **Disabling interrupts**
  - On single processor/core systems disabling interrupts prevents preemption and thus forces sequential execution
    - <u>does not work</u> on multiprocessor/multicore systems!

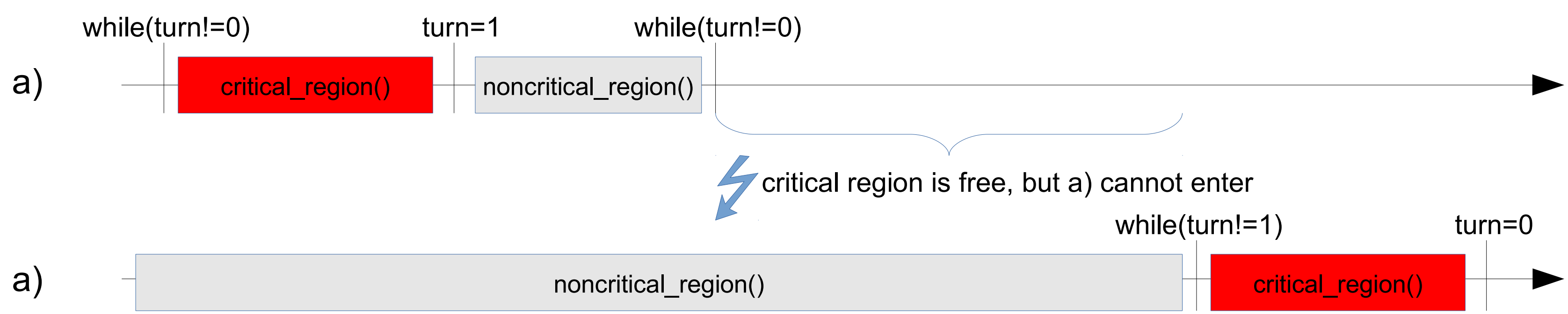
Achieving mutual exclusion: some valid **solution**

- **Strict alternation** *(working but not optimal *)*

```
while (TRUE) {                              while (TRUE) {
    while (turn != 0)      /* loop */ ;         while (turn != 1)      /* loop */ ;
    critical_region( );                         critical_region( );
    turn = 1;                                   turn = 0;
    noncritical_region( );                      noncritical_region( );
}                                           }
```

(a) A. Tanenbaum, Modern Operating Systems, 2nd ed (b)

* example:



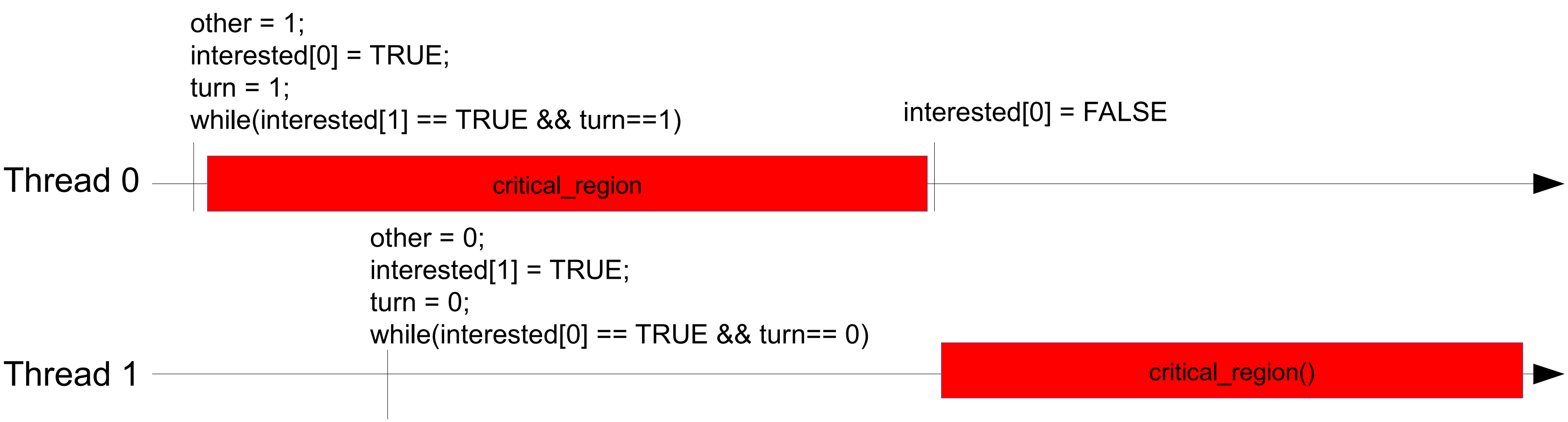
Achieving mutual exclusion: some valid **solution**

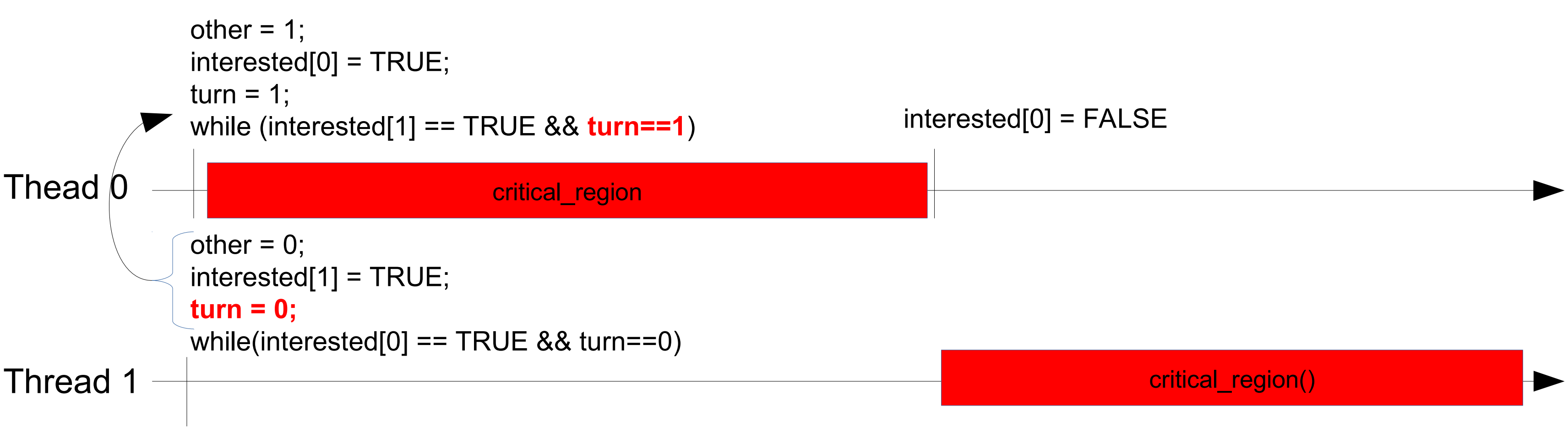
- **Peterson solution** (similar to strict alternation, but avoids flaw)

```c
int turn;
int interested[2];

#define TRUE 1
#define FALSE 0

int count = 0;

void enter_region(int thread)
{
    int other;
    other = 1 - thread;
    interested[thread] = TRUE;
    turn = other;
    // Other process might be in the region now, wait...
    while (interested[other] && turn == other);
}

void leave_region(int thread)
{
    // Leave the region
    interested[thread] = FALSE;
}
```

Might not work on multi-core/multiprocessor unless memory fences are used

Peterson solution





Achieving mutual exclusion: some valid **solution**

- **Solution based on the hardware TSL Instruction (spinlock)**
  - Many CPUs implement a TSL (**Test and Set Lock**) instruction which test and sets the value of a memory address in one atomic step
  - **TSL Source, Destination**

```
enter_region:
  TSL Lock, Reg         ; copy lock in reg, set lock to 1
  CMP Reg,#0            ; test if Reg (= previous lock value) is 0
  JNE enter_region      ; if it wasn't 0, repeat (loop)
  RET                   ; otherwise enter the critical region
exit_region:
  MOV #0, Lock          ; set lock to 0
  RET                   ; return
```

"spin"

Achieving mutual exclusion: some valid **solution**

- **Solution based on the hardware XCHG Instruction (spinlock)**
  - Many CPUs implement a **XCHG** instruction which performs an atomic swap of the two operands
  - **XCHG Source, Destination**

```
enter_region:
    MOVE REGISTER,#1          | put a 1 in the register
    XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
    CMP REGISTER,#0           | was lock zero?
    JNE enter_region          | if it was non zero, lock was set, so loop
    RET                       | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0              | store a 0 in lock
    RET                       | return to caller
```

“spin”

From A. Tanenbaum, Modern Operating Systems, 2nd ed

Busy waiting... handle with care

- Strict alternation, Peterson solution and spinlock solutions based on TSL or XCHG employ **busy waiting** to enter the critical section
  - they perform a loop continuously retrying to enter the region

- If the loop is short or there are multiple CPUs or cores drawbacks are neglibible
- ...but in some situations it can cause priority inversion:
  - high-priority is busy waiting to enter critical region **R**. Thread remains runnable, and preempts lower priority threads that are executing in **R** → might never exit critical region → deadlock

Synchronization primitives

- To avoid race conditions operating systems, with the help of atomic CPU instructions, implement **synchronization primitives** which are easier to work with and don't require busy waiting:
  - Mutex
  - Semaphores
  - Barriers
  - Condition variables

Mutex

- To enter a critical section a thread tries to **acquire** the corresponding mutex (**lock the mutex**)
  - Only one thread at a time can own the mutex
  - If the mutex is in the unlocked state, the thread can acquire it
    - From that moment, only the thread owning the mutex can **unlock** it
  - If the mutex is already locked, all threads which try to acquire it will wait (will not be scheduled for execution)

pthread mutex

- To create and initialize a mutex:

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
            const pthread_mutexattr_t *attr)
```

- **`pthread_mutex_t`** identifies the mutex
- **`pthread_mutexattr_t`** defines the type of mutex
  - can be **NULL** if we do not want to set attributes

Mutex types

```
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_DEFAULT);
```

FAST (default)

- Defined also with **PTHREAD_MUTEX_NORMAL**
- Can be acquired only once: if a thread owns the mutex and tries to re-acquire it it blocks (deadlock)
- Fastest mutex

```
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK);
```

NON RECURSIVE

- Returns a negative value (error) when the thread that owns the mutex tries to re-acquire it (useful for debugging)

```
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
```

RECURSIVE

- Can be re-acquired multiple times
- To unlock, the thread must call **pthread_mutex_unlock()** the same number of times it has called **pthread_mutex_lock()**

Destroy a mutex

- To destroy a mutex and release associated resources

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

- A locked mutex cannot be destroyed (EBUSY is returned)

Lock and unlock a mutex

- To lock a mutex:

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex)
```

- To unlock a mutex (only from the thread that owns it):

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <assert.h>
int account_balance = 100;
pthread_mutex_t mutex;

void* thread1(void* arg) {
    int wallet = 0;
    pthread_mutex_lock(&mutex);
    if (account_balance >= 100) {
        account_balance -= 100;
        pthread_mutex_unlock(&mutex);
        wallet = 100;
    } else {
        pthread_mutex_unlock(&mutex);
        printf("Cannot draw money\n");
        pthread_exit((void*)-1);
    }
    assert(account_balance >= 0);
    pthread_mutex_lock(&mutex);
    account_balance += 50;
    pthread_mutex_unlock(&mutex);
    printf("Balance: %d\n", account_balance);
}
```

```c
void* thread2(void* arg) {
    pthread_mutex_lock(&mutex);
    account_balance += 50;
    if (account_balance >= 100) {
        account_balance -= 100;
        pthread_mutex_unlock(&mutex);
        printf("Pay bills\n");
    } else {
        pthread_mutex_unlock(&mutex);
        printf("Not enough money\n");
    }
    assert(account_balance >= 0);
}

void main(void) {
    pthread_t t1, t2;
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&t1, NULL, thread1, NULL);
    pthread_create(&t2, NULL, thread2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
```

mutextrace

- To analyze lock/unlock sequences we can use a Linux tool called **mutextrace**

```
$ mutextrace ~/bankaccount_mutex
[1] mutex_init(1, FAST)
[2] started
[2] mutex_lock(1)
[2] mutex_unlock(1)
[2] mutex_lock(1)
[2] mutex_unlock(1)
Balance: 50
[3] started
[3] mutex_lock(1)
[3] mutex_unlock(1)
[2] finished (normal exit)
Pay bills
[3] finished (normal exit)
```

Source and packages at
http://packages.debian.org/squeeze/mutextrace

Semaphores

- A **semaphore** represents an integer value
  - At initialization the programmer can specify the initial value of the semaphore: subsequently the semaphore can be **incremented** (+1) or **decremented** (-1) atomically
  - When a thread decrements the semaphore, if the result is negative, the threads blocks and has to wait until another thread increases the semaphore
  - When a thread increments a semaphore and some threads are waiting, one of them is woken up

Semaphore initialization

- Initialize and assign a value to a semaphore

```
#include <semaphore.h>

int sem_init(sem_t *sem,
             int pshared,
             unsigned int value);
```

Specifies how a semaphore is to be shared:
- zero if the semaphore is shared only between threads in the same process
- nonzero if the semaphore is to be shared between processes (when using shared memory)

Initial value of the semaphore

Incrementing a semaphore

- To increment a semaphore (+ 1, atomically)

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

- If there are any waiting threads, one of them is unblocked

Decrementing a semaphore

- To decrement a semaphore (-1, atomically)

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
```

  - if the resulting semaphore value is less than zero the thread is put in a waiting list: when the thread wakes up, decrements the value and resumes execution

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
```

  - like **`sem_wait()`** but returns an error (**`EAGAIN`**) without blocking if the resulting value is less than zero

Getting the semaphore current value

- To get the semaphore's current value

```
#include <semaphore.h>

int sem_getvalue(sem_t *sem, unsigned int *sval);
```

The value is stored at the address pointed by sval

Destroying a semaphore

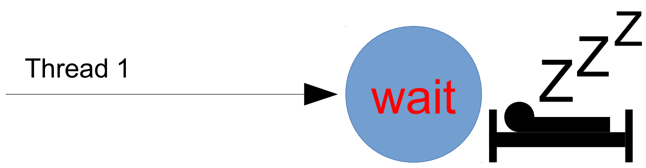
- To destroy a semaphore

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

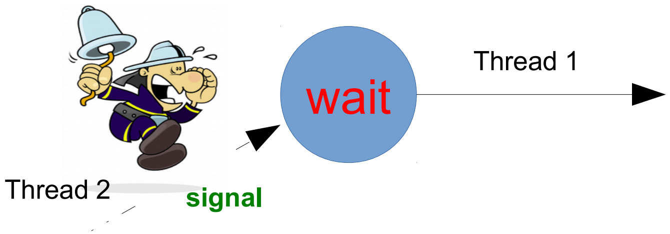
Destroying a semaphore which has some waiting threads results in an undefined (i.e bad) behavior!

Condition variables

- Alternative to semaphores
- A **condition variable** can be manipulated using two primitives
  - **wait** *(for an event to happen)*
  - **signal** *(that some event has happened)*



Thread 1 waits on the condition variable



Thread 2 signals the condition variable, waking up Thread 1 which resumes execution

Initialize a condition variable

- Initialize a condition variable

```
#include <semaphore.h>

int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
```

Wait for an event

- Wait on a condition variable

```
#include <semaphore.h>

int pthread_cond_wait(pthread_cond_t *cond,
            pthread_mutex_t *mutex);
```

what's mutex???

Condition variable and mutex

- A condition variable is a synchronization mechanism which does not manage mutual exclusion...
  - ... for this purpose we have mutexes!

- Frequently the event of a condition variable is related to a shared resource (which should be protected)
  - We can pass a mutex to **`p_thread _cond_wait()`** that will be released while the thread is waiting for the event... as soon as execution resumes the thread will try to re-acquire the thread

Signaling an event

- Signal on a condition variable

```
#include <semaphore.h>

int pthread_cond_signal(pthread_cond_t *cond);
```

- This call wakes up **at least one** waiting thread
  - if there is not waiting thread this call does nothing
- If more than one thread is waiting on the condition variable and there is a mutex, those threads will all try to re-acquire it but just one will succeed (the others will wait)

Signaling an event

- Signal on a condition variable

```
#include <semaphore.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
```

- This call wakes up **all** waiting thread
  - if there is not waiting thread this call does nothing
- All the threads waiting on the condition variable resume execution: if there is a mutex, those threads will all try to re-acquire it but just one will succeed (the others will wait)

Destroying a condition variable

- Destroy a condition variable

```
#include <semaphore.h>

int pthread_cond_destroy(pthread_cond_t *cond);
```

Destroying a condition variable which has some waiting threads results in an undefined (i.e bad) behavior!

Spurious wakeup

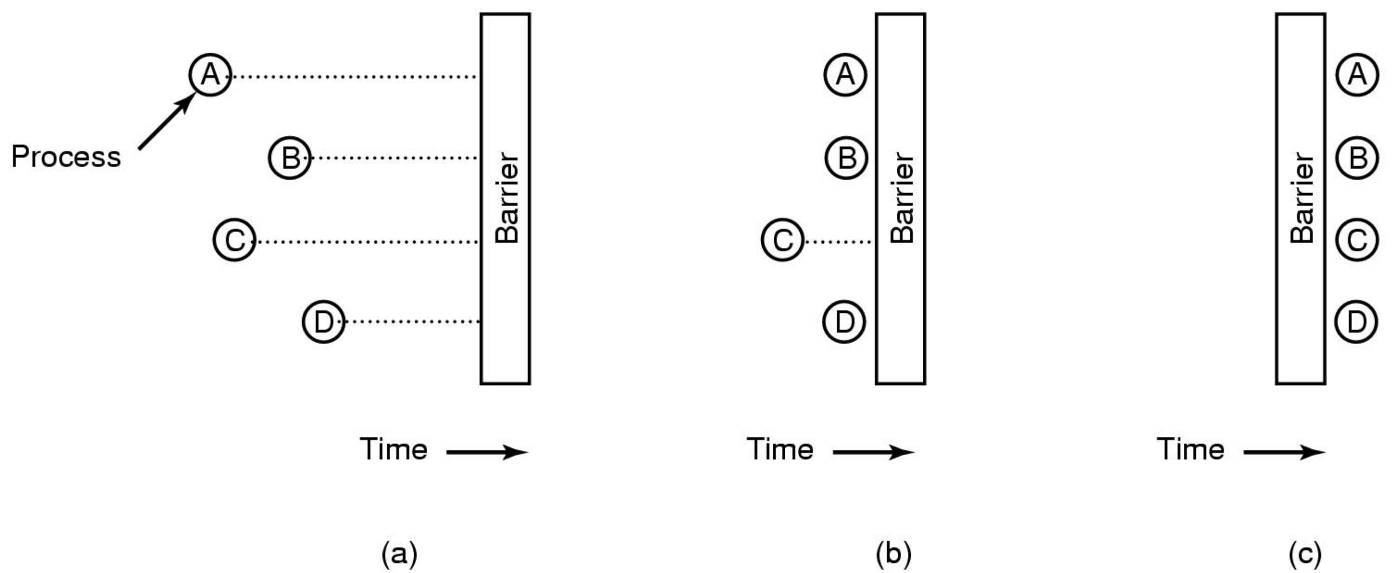
- **`pthread_cond_wait`** might return unexpectedly* or in more than one waiting thread** even when **`pthread_cond_signal`** (and not **`pthread_cond_broadcast`**) is called (mostly in multiprocessor systems):
  - These issues are called **spurious wakeups**:
  - Putting the wait inside a while loop is good practice to ensure that the condition is always checked before continuing execution

* In multiprocessor systems

** On Linux **pthread_cond_wait** is based on a futex call which returns if interrupted by a system call

Barrier

- The barrier is a synchronization mechanism which is useful to control the execution of a multi-step parallel algorithm, where each step depends on results computed at the previous step



From A. Tanenbaum, Modern Operating Systems, 2nd Edition

Creating a barrier

- Initializing a barrier

```
#include <pthread.h>

int pthread_barrier_init
            (pthread_barrier_t *restrict barrier,
      const pthread_barrierattr_t *restrict attr,
            unsigned count);
```

Number of threads that need to synchronize on the barrier

Barrier properties (typically NULL, for default parameters; used for setting sharing options)

Destroying a barrier

- Destroy a barrier

```
#include <pthread.h>

int pthread_barrier_destroy(pthread_barrier_t
                             *barrier);
```

Waiting on a barrier

- A thread can wait on a barrier with

```
#include <pthread.h>

int pthread_barrier_wait(pthread_barrier_t
                         *barrier);
```

- When all threads arrive at the barrier, the latter unlocks and this function returns **`PTHREAD_BARRIER_SERIAL_THREAD`**

Barrier example

- We consider matrix multiplication:

$$\begin{array}{c} \text{row} \\ \begin{bmatrix} a0 & a1 & a2 \\ b0 & b1 & b2 \\ c0 & c1 & c2 \end{bmatrix} \end{array} * \begin{array}{l} \text{column} \\ \begin{bmatrix} x0 & x1 & x2 \\ y0 & y1 & y2 \\ z0 & z1 & z2 \end{bmatrix} \end{array} =$$

$$\begin{bmatrix} a0x0+a1y0+a2z0 & a0x1+a1y1+a2z1 & a0x2+a1y2+a2z2 \\ b0x0+b1y0+b2z0 & b0x1+b1y1+b2z1 & b0x2+b1y2+b2z2 \\ c0x0+c1y0+c2z0 & c0x1+c1y1+c2z1 & c0x2+c1y2+c2z2 \end{bmatrix}$$

Matrix multiplication

- The problem can be easily divided in multiple independent subproblems which can execute in parallel
  - we can assign each of these problems to one thread

- ... but consider if, given a matrix A, we want to compute A^3
  - we need to perform two steps
    1) multiply A * A = B
    2) multiply B * A =C

... the second step depends on the result of the first one!

Barrier example: matrix cube (1)

```
#include <pthread.h>
#include <stdio.h>

double A[3][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9} };
double B[3][3];
double C[3][3];

pthread_barrier_t barrier;

void multiply(double X[3][3], double Y[3][3], double Z[3][3],
                int row, int column) {
  int i;
  Z[row][column] = 0;
  for(i=0; i<3; i++) {
    Z[row][column] += X[row][i] * Y[i][column];
  }
}
```

Barrier example: matrix cube (2)

```
void* thread_multiply (void* args)
{
  int result;
  int row = (int) args / 3;
  int column = (int) args % 3;

  multiply(A, A, B, row, column);

  result = pthread_barrier_wait(&barrier);

  if (result != 0 && result != PTHREAD_BARRIER_SERIAL_THREAD) {
    perror("Error!\n");
    exit(-1);
  }

  multiply(A, B, C, row, column);
}
```

Barrier example: matrix cube (3)

```
void main()
{
  pthread_t threads[9]; /* One thread for each matrix element */
  int t, r, c;
  if(!pthread_barrier_init(&barrier, NULL, 9)) {
    for (t=0; t<9; t++) {
      pthread_create(&threads[t], NULL, &thread_multiply, (void*) t);
    }
    for (t=0; t<9; t++) {
      pthread_join(threads[t], NULL);
    }
    printf("The cube is:\n");
    for (r=0; r<3; r++) {
      for (c=0; c<3; c++) {
        printf(" %3.2f ", C[r][c]);
      }
      printf("\n");
    }
    pthread_barrier_destroy(&barrier);
  }
}
```