# Operating Systems 2024/2025

## TP Class 04 – Threads and synchronization (1/2)

Vasco Pereira (vasco@dei.uc.pt)
Dep. Eng. Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Some slides based on previous versions from Bruno Cabral, Paulo Marques and Luis Silva.

**operating system**
noun
the collection of software that directs a computer's operations, controlling and scheduling the execution of other programs, and managing storage, input/output, and communication resources.

Abbreviation: OS

Source: Dictionary.com

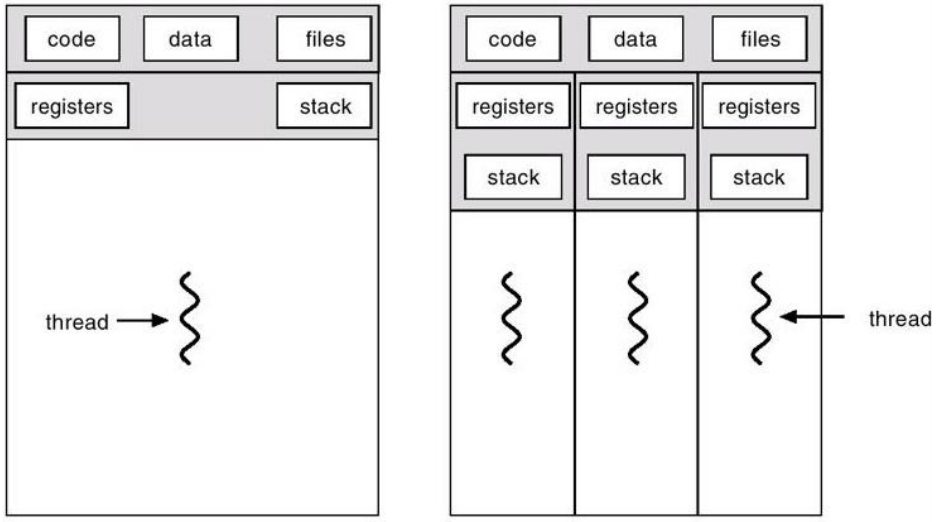
1 2 9 0
Departamento de Engenharia Informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D COIMBRA

Updated on
22 February 2025

# THREADS

# Threads

- Single vs multithreaded process




single-threaded multi-threaded

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

# Threads

- Why use threads?
  - Very light weight compared to processes
  - Light context switches
  - Fast to create and terminate
  - Fast to synchronize
  - Resource sharing

Be careful to synchronize the access to shared resources!

# POSIX Threads – Thread Management

- POSIX Thread management functions summary

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

  - Create a new thread

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

  - Terminate the current thread and return a pointer to a value

```
void pthread_exit(void * retval);
```

  - Cancel a thread

```
int pthread_cancel(pthread_t thread);
```

  - Wait for the termination of a given thread

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

  - Return the identifier of the current thread

```
pthread_t pthread_self(void);
```

All Pthreads functions return 0 on success or a positive value on failure

# POSIX Threads – Thread Management

- Thread creation

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
      const pthread_attr_t *attr,
      void*(*start_function)(void*), void *arg);
```

  - Creates a new thread;
  - The function prototype for Pthread creation (the start function) is:

```
void * start_function(void * arg);
```

  - The new thread starts execution by calling function `start_function` with arguments `arg` ( `start_function(arg)`);
  - If multiple arguments are needed, then `arg` should point to a structure
  - If `attr` is specified as `NULL`, then the thread is created with default attributes; (use NULL)
  - After a call to `pthread_create()`, a program has no guarantees about which thread will next be scheduled.

# POSIX Threads – Thread Management

- Thread termination
  - The execution of a thread terminates in one of the following ways:
    - The thread's start function performs a return specifying a return value for the thread.
    - The thread calls `pthread_exit()`
    - The thread is canceled using `pthread_cancel()`
    - Any of the threads calls `exit()`, or the main thread performs a return (in the `main()` function), which causes all threads in the process to terminate immediately.

# POSIX Threads – Thread Management

- Thread termination (2)

```
#include <pthread.h>
int pthread_exit(void* retval);
```

  - Terminates the calling thread;
  - The `retval` argument specifies the return value for the thread;
  - If the main thread calls `pthread_exit()` instead of calling `exit()` or performing a `return`, then the other threads continue to execute.

# POSIX Threads – Thread Management

- Thread termination (3)

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

  - Requests a thread cancellation;
  - Having made the cancellation request, `pthread_cancel` returns immediately, it doesn't wait for the target thread to terminate;
  - What happens to the target thread, and when it happens, depends on that thread's cancellation state (specifies if a specific thread is cancelable or not) and type (the thread can be cancelable at any time or remains pending until a cancellation point – a call to some specific functions);
  - By default a new thread is cancelable.

# POSIX Threads – Thread Management

- Thread join

```
#include <pthread.h>
int pthread_join(pthread_t thread, void** retval);
```

  - Waits for the thread identified by `thread` to terminate;
  - If `retval` is a non-NULL pointer, it receives the terminated thread reurn value (the value that was specified when the thread performed a return or called `pthread_exit()` ).
  - Detached threads cannot be joined
    - Detached threads are threads from which we do not want to receive the return status; the system automatically cleans up and removes the thread when it terminates.
    - A thread can be marked as detached by using `pthread_detach()` (not covered in these slides)

# POSIX Threads – Thread Management

- Thread ID

```
#include <pthread.h>
pthread_t pthread_self(void);
```

  - Returns the thread ID of the calling thread;
  - Each thread in a process is uniquely identified by a thread ID.

# Simple thread creation example

## simple_thread.c

```c
// Ids used by the threads
pthread_t my_thread[N];
int       id[N];

// Worker thread
void* worker(void* idp) {
  int my_id = *((int*) idp);

  printf("Hello, I'm thread %d\n", my_id);
  sleep(rand()%3);
  printf("Hello, I'm thread %d, going away!\n", my_id);

  pthread_exit(NULL);
  return NULL;
}

int main()
{
  // Creates N threads
  for (int i=0; i<N; i++) {
    id[i] = i;
    pthread_create(&my_thread[i], NULL, worker, &id[i]);
  }

  // Waits for them to die
  for (int i=0; i<N; i++)
    pthread_join(my_thread[i], NULL);

  return 0;
}
```

Not executed

# Compiling with threads

- Linux

```
gcc –lpthread –D_REENTRANT –Wall fich.c –o fich
```

- -D_REENTRANT is quite important in LinuxThreads (Kernel 2.4) but gcc calls it implicitly
  - It instructs the compiler to use special re-entrant routine functions
  - If you don't... it ONLY appears to work, until you get in trouble!
  - In some systems **–pthread** is sufficient

Beware: Many routines are not re-entrant, they cannot be directly used with threads since they use common storage in an unsynchronized way (e.g., stktok())!

In some cases, there are re-entrant versions (e.g., strtok_r()). Check the manual! Don't trust common sense.

# Example of non-reentrant routine

```
char buffer[MAX_SIZE];

void send_to_network(const char* msg)
{
  strcpy(buffer, "START_MSG | ");
  strcat(buffer, msg);
  strcat(buffer, " | END_MSG");
}
```

What happens if this is called from two different threads at the same time??

| Thread-unsafe function | Reentrant version |
|---|---|
| asctime | asctime_r |
| ctime | ctime_r |
| gethostbyaddr gethostbyname | gethostbyaddr_r |
| inet_ntoa | gethostbyname_r |
| localtime | (none) |
| rand | localtime_r |
| | rand_r |

# Example of reentrant/non-reentrant funcs

```
// An example where func1() and func2() are non-reentrant
int i;
int func1() {   // func1() is NOT reentrant because it uses global variable i
     return i * 2;
}

int func2() {  // func2() is NOT reentrant because it calls a non-reentrant function
   return func1() * 2;
}

// An example where func1() and func2() are reentrant
int func1(int i) {
     return i * 2;
}

int func2(int i) {
     return func1(i) * 2;
}
```

# Other things to beware of…

```
// Creates N threads
for (int i=0; i<N; i++) {
  pthread_create(&my_thread[i], NULL, worker, &i);
}
```

Doesn't work, "i" is on the stack an constantly changing

```
int main()
{
  // Ids used by the threads
  pthread_t my_thread[N];
  int       id[N];

  // Creates N threads
  for (int i=0; i<N; i++) {
    id[i] = i;
    pthread_create(&my_thread[i], NULL, worker, &id[i]);
  }

  return 0;
}
```

Doesn't work!

(1) after main() dies, its variables disappear – race condition with the starting threads;

(2) main() dies everything dies!

# If you need to terminate the main() thread...

```
// Ids used by the threads
pthread_t my_thread[N];
int       id[N];

int main()
{
  // Creates N threads
  for (int i=0; i<N; i++) {
    id[i] = i;
    pthread_create(&my_thread[i], NULL, worker, &id[i]);
  }

  // Kill the main thread
  pthread_exit(NULL);
  return 0;
}
```

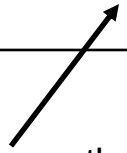
This is OK!

**Note:** the other threads continue to execute.

# Beware of... shared variables

```
char **ptr;  /* global */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    pthread_exit(NULL);
}
```

```
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;

    printf("[%d]: %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
}
```



The new threads access
the main thread's stack through **ptr**

# Beware of...

- What happens when you...
  - fork() in a thread
    - When a multithreaded process calls fork(), only the calling thread is replicated in the child process.
  - exec() in thread
    - When any thread calls one of the exec() functions, the calling program is completely replaced.
- Signals are received
  - More on this later…

# THREADS SYNCHRONIZATION

# POSIX synchronization of threads

- Mutexes
  - Provide mutual exclusion zones between threads (processes can also use them if shared memory is used)
  - Similar to a binary semaphore, but the thread that locks the mutex must be the one to unlock it
- POSIX Semaphores (named and unnamed can be used)
  - Used to signal events across threads
  - Used to count objects in a synchronized way
- Condition Variables
  - Allow a thread to block or to notify others on any condition
  - Semaphores are a kind of condition variable:
  - the implicit condition is the semaphore being greater than 0

# POSIX Mutexes

```
#include <pthread.h>

// Initialize a mutex with the specified attributes
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
           const pthread_mutexattr_t *restrict attr);

// Declares and creates a mutex with default attributes
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;

// Performs a lock on a mutex
int pthread_mutex_lock(pthread_mutex_t *mutex);

// Performs an unlock on a mutex
int pthread_mutex_unlock(pthread_mutex_t *mutex);

// Tries to performs a lock on a mutex
int pthread_mutex_trylock(pthread_mutex_t *mutex);

//Releases a mutex
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Initialization

If NULL, the attributes used are the default

Static initialization using a MACRO. Uses default attributes.

# POSIX Unnamed Semaphores

```
#include <semaphore.h>

// Create an unnamed semaphore
int sem_init(sem_t *sem, int pshared, unsigned int value);

// Wait and post
int sem_wait(sem_t *sem);

int sem_post(sem_t *sem);

int sem_trywait(sem_t *sem);

int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);

// Get the value of a semaphore
int sem_getvalue(sem_t *sem, int *sval)

// Destroy an unnamed semaphore
int sem_destroy(sem_t *sem);
```

POSIX unnamed semaphores were already seen before…)

# POSIX Named Semaphores

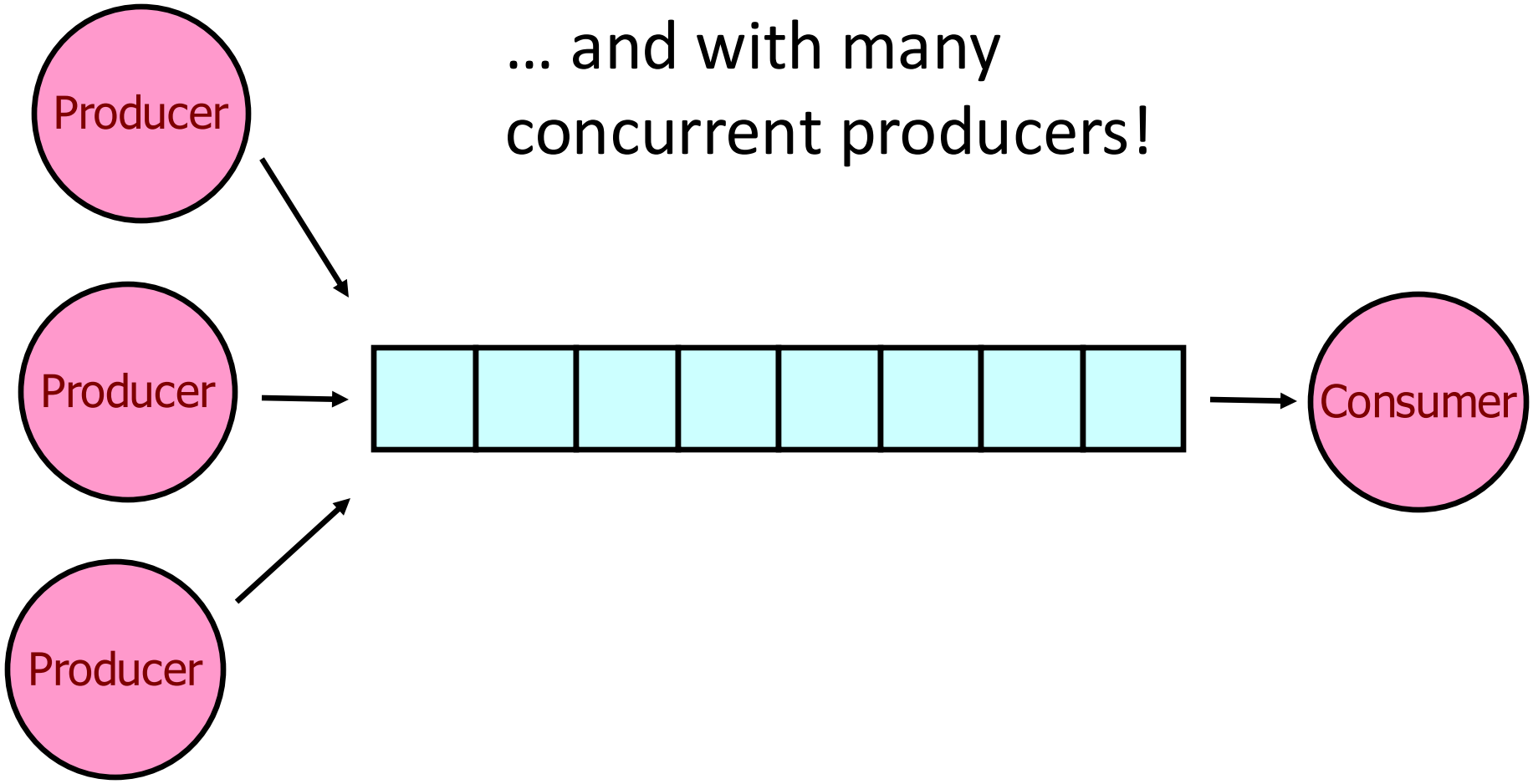
```
#include <semaphore.h>

// Create a named semaphore
sem_t *sem_open(const char *name, int oflag);

sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);

// Wait and post
int sem_wait(sem_t *sem);

int sem_post(sem_t *sem);

int sem_trywait(sem_t *sem);

int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);

// Get the value of a semaphore
int sem_getvalue(sem_t *sem, int *sval)

// Close a named semaphore (remove association with a semaphore)
int sem_close(sem_t *sem);

// Delete a named semaphore
int sem_unlink(const char *name);
```

POSIX named semaphores were already seen before…)

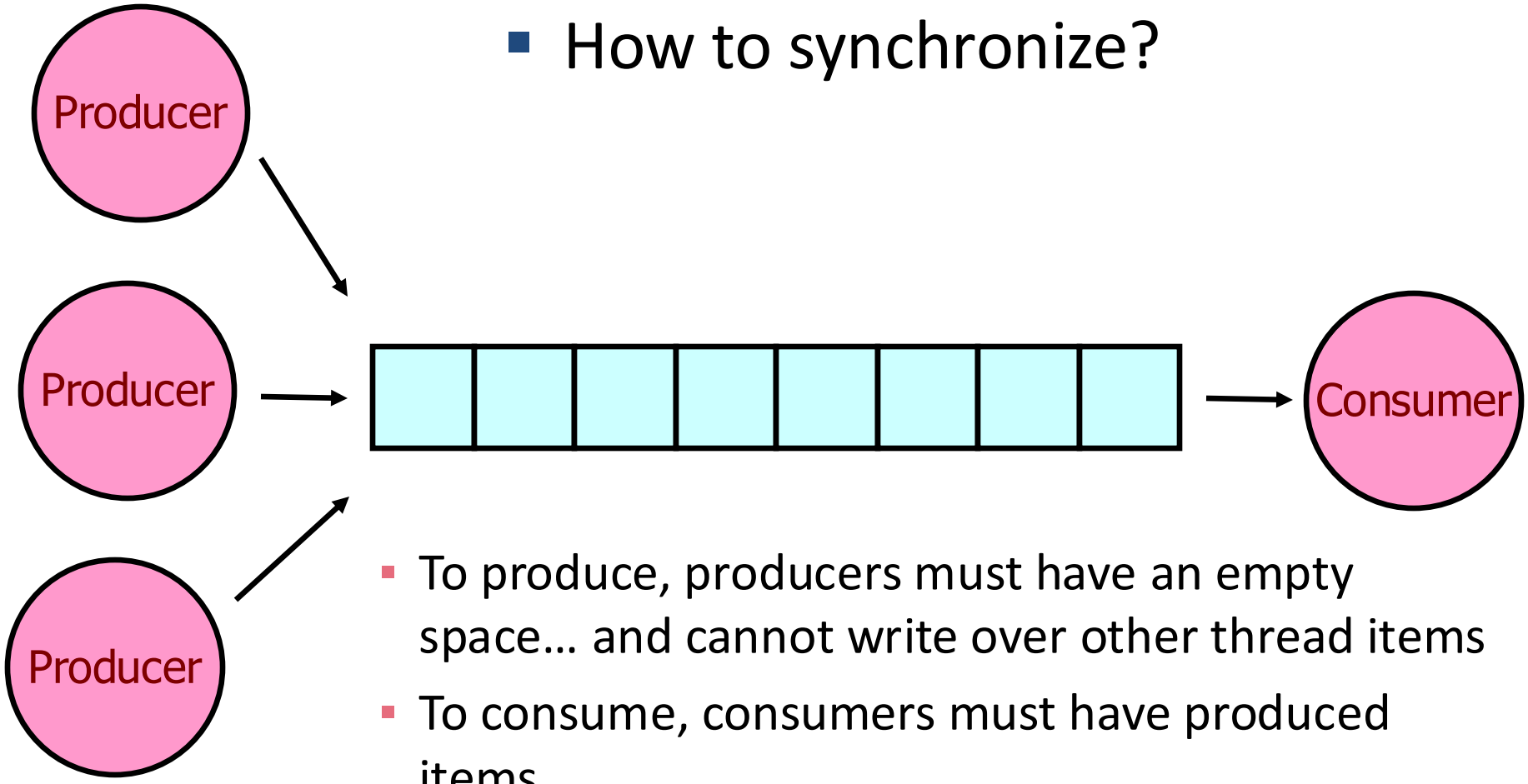
# POSIX mutexes and semaphores between processes

- POSIX unnamed semaphores and mutexes can also be used between (threads in different) processes
  - Set semaphores pshared or mutex init attributes to the appropriate value
  - Use shared memory

# Producer/consumer with threads



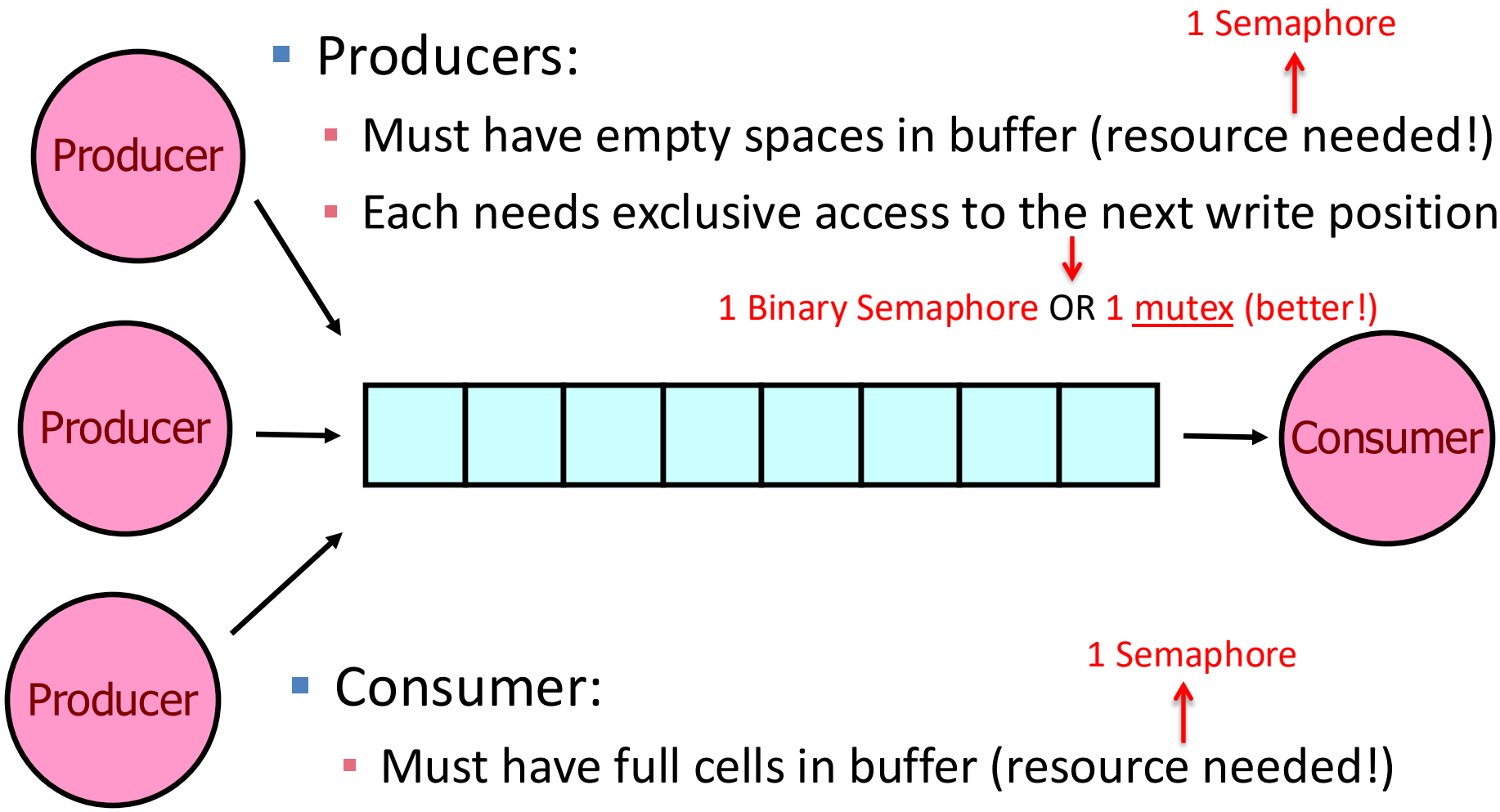
# Producer/consumer with threads
## synchronization




- How to synchronize?

- To produce, producers must have an empty space… and cannot write over other thread items
- To consume, consumers must have produced items

# Producer/consumer with threads
## synchronization



# prod_cons_threads.c

```c
int write_pos, read_pos;
int buf[N];

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t empty;
sem_t full;

int id[PRODUCERS];

void init() {
  sem_init(&empty, 0, N);
  sem_init(&full, 0, 0);

  write_pos = read_pos = 0;
}
```

```c
int main(int argc, char *argv[]) {
  int i;
  init();
  pthread_t thr;
  pthread_create(&thr, NULL, consumer, NULL);

  for (i = 0; i < PRODUCERS; i++) {
    id[i] = i;
    pthread_create(&thr, NULL, producer, &id[i]);
  }
  pthread_exit(NULL);
  return 0;
}
```

# prod_cons_threads.c (2)

```c
void* producer(void *id) {
  int my_id = *((int*) id);
  int i = my_id;

  while (1) {
    sem_wait(&empty);
    pthread_mutex_lock(&mutex);

    printf("[PRODUCER %3d] Writing %d\n", my_id, i);
    buf[write_pos] = i;
    write_pos = (write_pos+1) % N;

    pthread_mutex_unlock(&mutex);
    sem_post(&full);
    ++i;
    //sleep(1);
  }
}
```

Not necessary – there is only one consumer

```c
void* consumer(void *arg) {
  while (1) {
    sem_wait(&full);
    pthread_mutex_lock(&mutex);

    int e = buf[read_pos];
    read_pos = (read_pos+1) % N;
    printf("[CONSUMER    ] Read %d\n", e);

    pthread_mutex_unlock(&mutex);
    sem_post(&empty);
    sleep(1);
  }
  return NULL;
}
```

# Result...

```
[vasco@student2 SO]$ gcc -Wall -D_REENTRANT -lpthread prod_cons_threads.c -o pc
[vasco@student2 SO]$ ./pc
[PRODUCER   1] Writing 1
[PRODUCER   1] Writing 2
[PRODUCER   1] Writing 3
[PRODUCER   0] Writing 0
[PRODUCER   0] Writing 1
[PRODUCER   3] Writing 3
[PRODUCER   4] Writing 4
[CONSUMER    ] Read 1
[PRODUCER   0] Writing 2
[PRODUCER   1] Writing 4
[PRODUCER   2] Writing 2
[PRODUCER   5] Writing 5
[CONSUMER    ] Read 2
[PRODUCER   6] Writing 6
[CONSUMER    ] Read 3
[PRODUCER   3] Writing 4
^C
[vasco@student2 SO]$
```

# Class demos included

- Demo01

  `simple_thread.c`
- Demo02

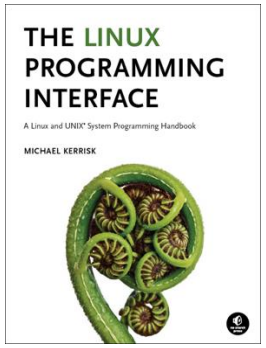
  `prod_cons_threads.c`
- Demo 03

  `prod_cons_threads-named_semaphores.c`
- Demo04

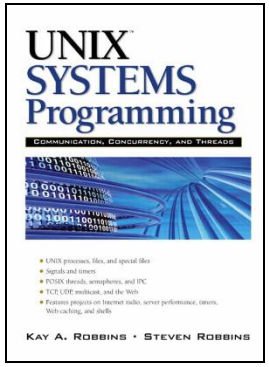
  `mutex_between_procs_no.c  // does not work!`
- Demo05

  `mutex_between_procs_yes.c // correct implementation`

# References




- [kerrisk10]
- Chapter 29: Threads: Introduction
- Chapter 30: Threads: Thread Synchronization
- Chapter 31: Threads: Thread Safety (…)
- Chapter 32: Threads: Thread Cancellation




- [Robbins03]

  Chapter 12: POSIX Threads

  Chapter 13: Thread Synchronization

  Chapter 14: Critical Sections and Semaphores

# INTRODUCTION TO ASSIGNMENT 05– "THREADS AND SYNCHRONIZATION I"

# Thank you! Questions?

*I keep six honest serving men. They taught me all I knew. Their names are What and Why and When and How and Where and Who.*
*—Rudyard Kipling*