Module 3 - Threads

Reading: Chapter 4 (Silberschatz)

Goal:

- **Understanding the concept of threads and its relationship to the process**
- **Understand how the operating systems manage and use the threads.**

Topics

- **The execution thread in the processes**
- **Multi-thread versus single thread (the thread)**
  - User level threads and kernel level threads
- **The challenges of "Threading"**
- **Examples of threads**

Characteristics of processes

- **Resource ownership unit - a process has:**
  - an addressable virtual space containing the process image
  - other resources (files, I / O units ...)
- **Execution unit (dispatching) - a process is executed along a path among several programs**
  - execution nested among the execution of several processes
  - the process has an execution state and a priority for scheduling

Process

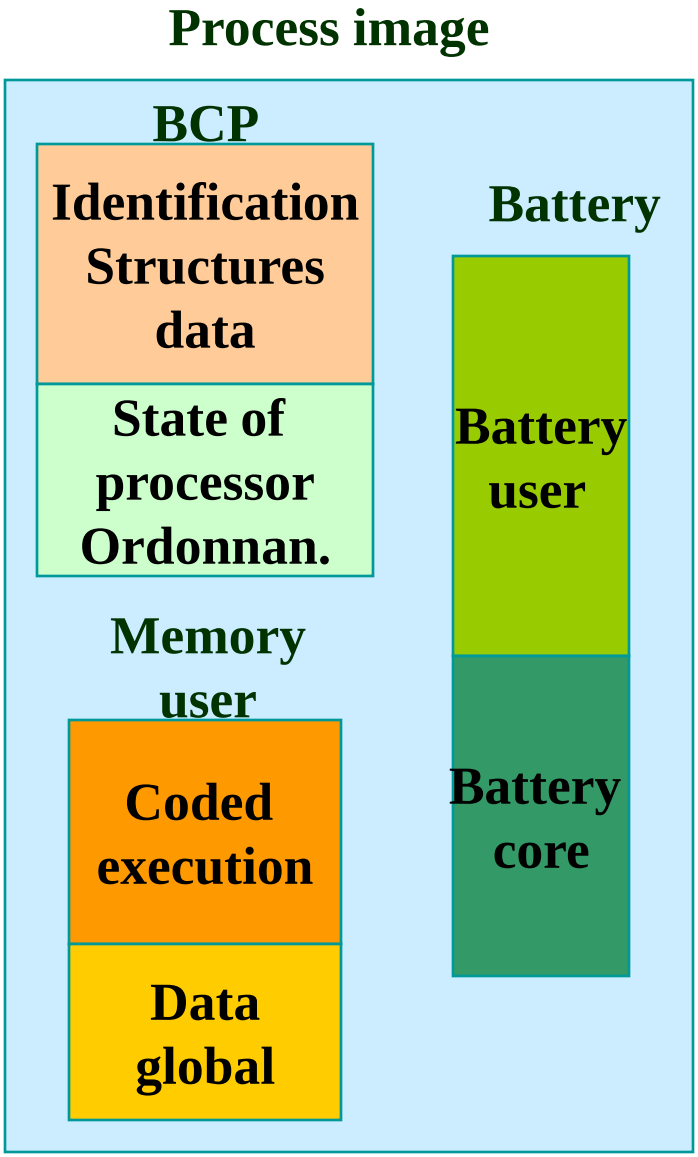
- **Owns its memory, files, resources, etc.**
- **Protected access to memory, files, resources of other processes**

Process characteristics

- These 2 characteristics are perceived as independent by some OS
- The execution unit is usually denoted by execution thread
- The unit of resource ownership is usually referred to as process

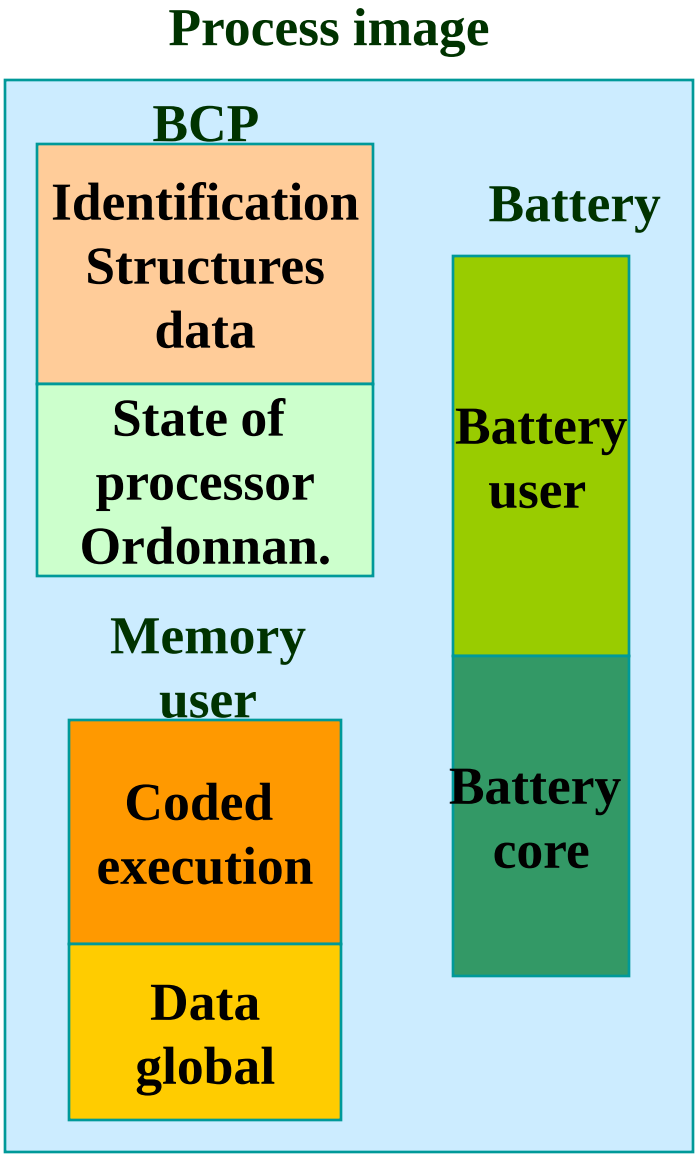
Resource ownership unit

- Related to the following components of the process image
  - The part of the PCB which contains identification and resource structures
  - Memory containing the execution code
  - Memory containing global data



Execution unit

- **Related to the following components of the process image**
  - PCB
    - **Processor state**
    - **Scheduling structure**
  - Stack



Topics

- **The execution thread in the processes**
- **Multi-thread versus single thread (the thread)**
  - User level threads and kernel level threads
- **The challenges of "Threading"**
- **Examples of threads**

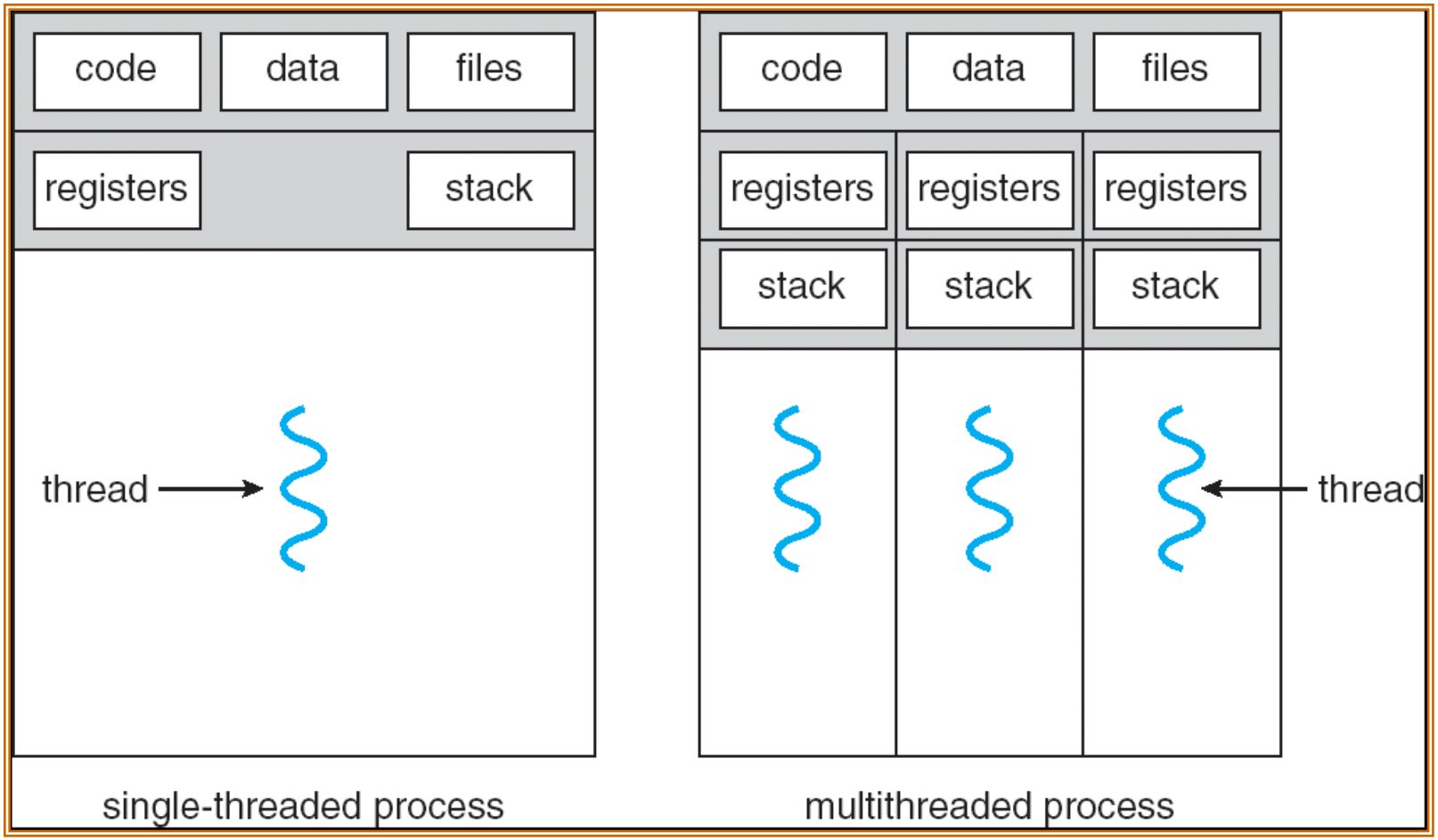
Threads = lightweight processes

- **A thread is a subdivision of a process**
  - A control thread in a process
- **The different threads of a process share addressable space and resources of a process**
  - when a thread modifies a variable (not local), all other threads see the modification
  - a file opened by one thread is accessible to other threads (of the same process)

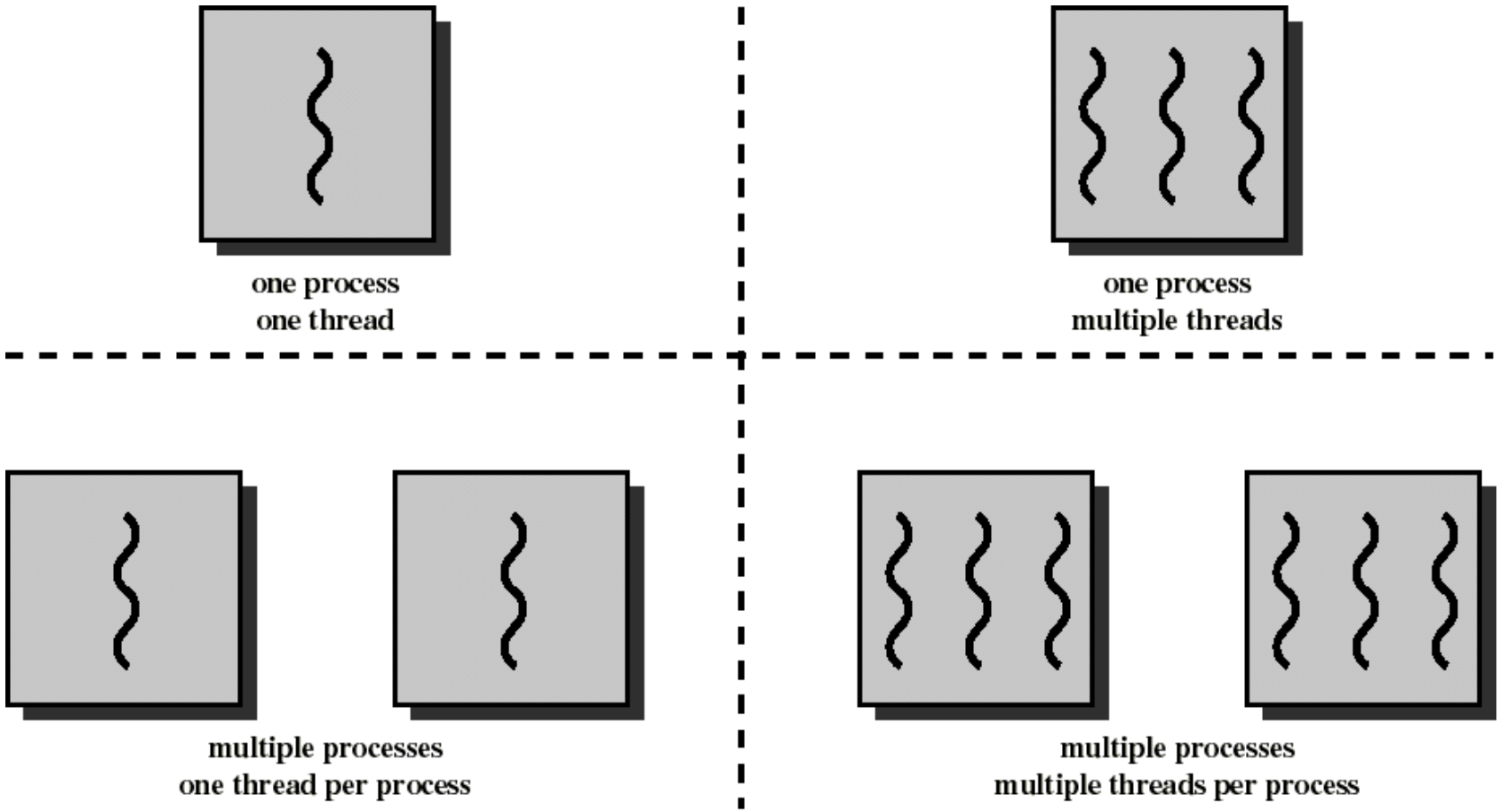
Example

- **The MS-Word process involves several threads:**
  - Interacting with the keyboard
  - Arrangement of characters on the page
  - Regular backup of work done
  - Spell check
  - Etc.
- **These threads share all the same document**

Single and multi-threaded processes



Threads and processes [Stallings]



Thread

- **Has an execution status (ready, blocked, etc.)**
- **Has its stack and a private space for local variables**
- **Has access to the addressable space, files and resources of the process to which it belongs**
  - In common with the other threads of the same process

Why threads

- **Responsiveness: a process can be subdivided into several threads, e.g. one dedicated to interacting with users, the other dedicated to processing data**
  - One can run as long as the other is blocked
- **Use of multiprocessors: threads can run in parallel on different CPUs**

Switching between threads is less expensive than switching between processes

- A process has memory, files, other resources
- Changing from one process to another involves saving and restoring the state of it all.
- Switch from one thread to another *in the same process* is much simpler, involves saving CPU registers, stack, and little else

Communication is also less expensive between threads than between processes.

- **Since threads share their memory,**
  - communication between threads in the same process is more efficient than communication between processes

Creation is less expensive

- **Creating and terminating new threads in an existing proc is also less expensive than creating a proc.**

Kernel and user threads

- **Where to implement the threads:**
  - In user libraries
    - **user-controlled**
    - **POSIX Pthreads, Java threads, Win32 threads**
  - In the kernel of the OS:
    - **kernel controlled**
    - **Windows XP / 2000, Solaris, Linux, True64 UNIX, Mac OS X**
  - Mixed solutions
    - **Solaris 2, Windows 2000 / NT**

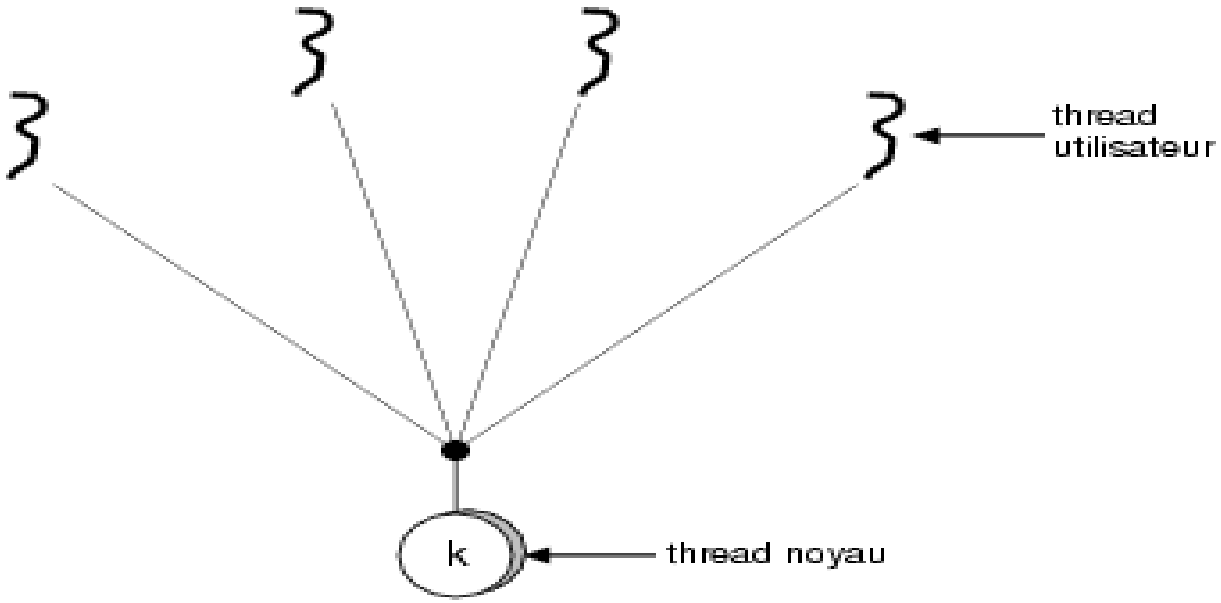
User and kernel threads (kernel)

- **user threads: supported by user libraries or prog language**
  - efficient because ops on threads do not request system calls
  - disadvantage: the kernel is not able to distinguish between state of process and state of threads in the process
    - **blocking a thread implies blocking the process**
- **kernel threads: directly supported by the OS kernel (WIN NT, Solaris)**
  - the kernel is able to directly manage the states of the threads
  - It can assign different threads to different CPUs

Mixed solutions: user and kernel threads

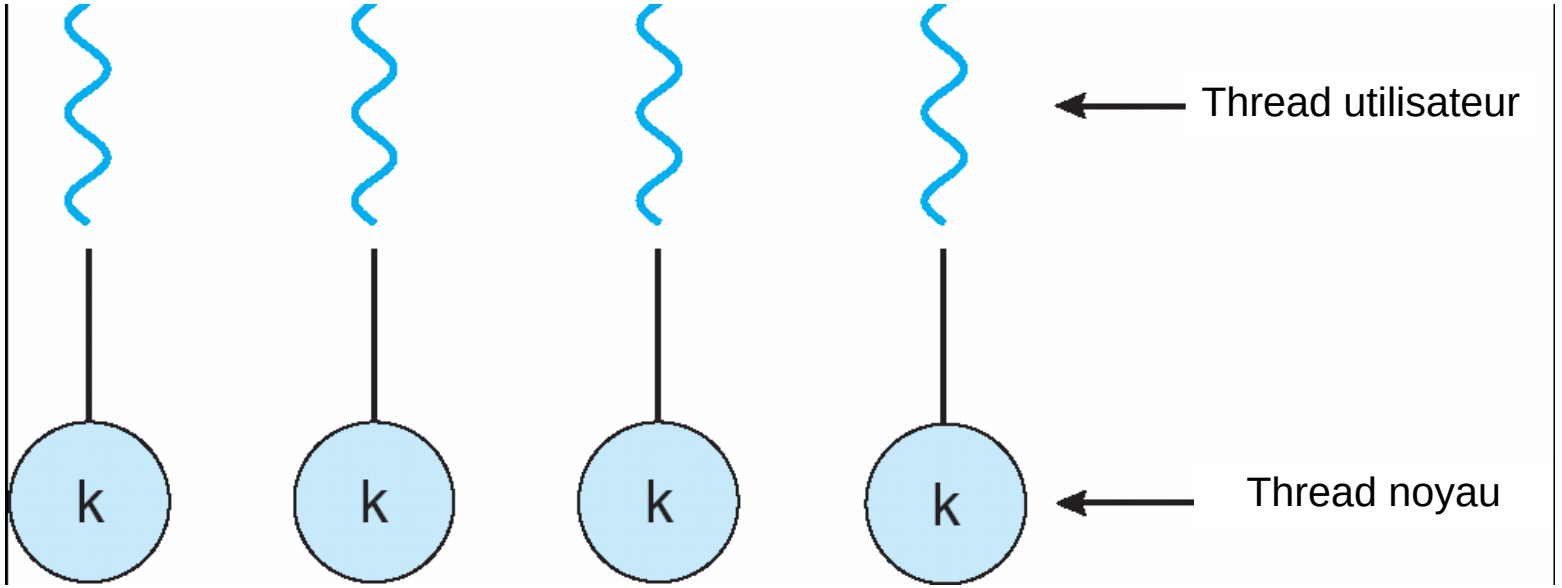
- **Relationship between user threads and kernel threads**
  - many to one
  - one by one
  - many to many (2 models)
- **We must take into consideration several levels:**
  - Process
  - User thread
  - Kernel thread
  - Processor (CPU)

Multiple user threads for a kernel thread:
the user controls the threads



- **OS does not know user threads**
  - v. advantages and disadvantages mentioned before
- **Examples**
  - Solaris Green Threads
  - GNU Portable Threads

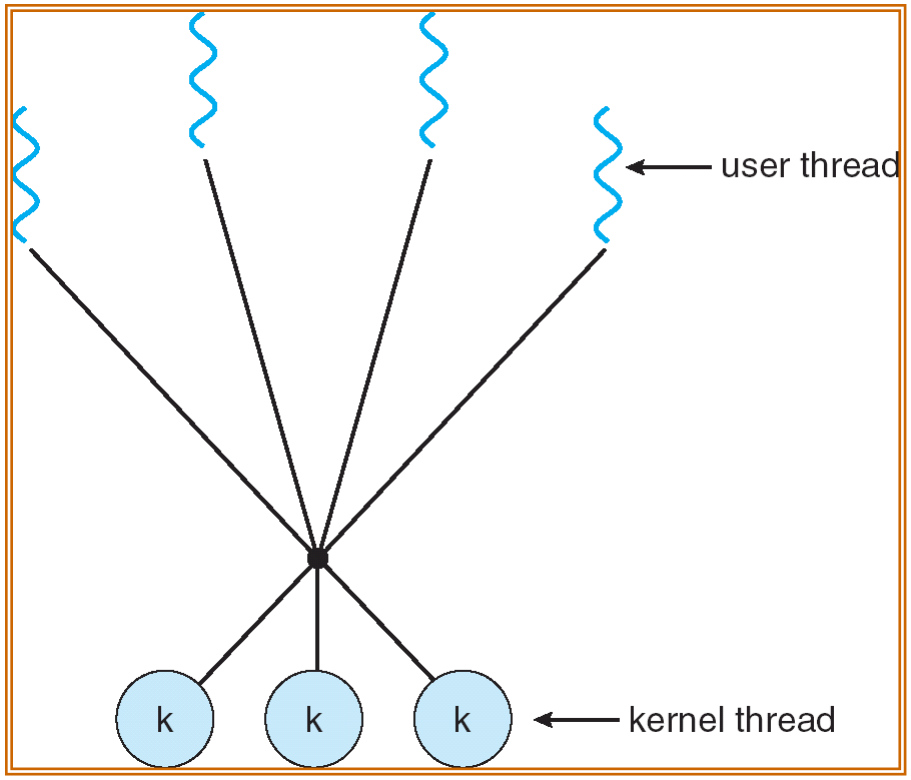
One to one: the OS controls the threads



- **Ops on threads are system calls**
- **Allows another thread to run when a thread executes a blocking system call**
- **Win NT, XP, OS / 2**
- **Linux, Solaris 9**

Many to many: mixed solution (M: M - many to many)

- Uses both user threads and kernel threads
- Flexibility for the user to use the technique he prefers
- If a user thread blocks, its kernel thread can be assigned to another
- If more. UCT are available, plus. kernel threads can run at the same time
- Some versions of Unix, including Solaris before version 9
- Windows NT / 2000 with the *ThreadFiber* package



Multithreads and monothreads

- **MS-DOS supports a single-threaded user process**
- **UNIX SVR4 supports multiple single-threaded processes**
- **Solaris, Widows NT, XP and OS2 support multiple multithreaded processes**

Topics

- **The execution thread in the processes**
- **Multi-thread versus single thread (the thread)**
  - User level threads and kernel level threads
- **The challenges of "Threading"**
- **Examples of threads**

Threading challenges

How beautiful it is to have children, but what are the practical consequences?

Challenges:

- Semantics of system calls **fork ()** and **exec()**
- Canceling threads
- A group of threads (pools)
- Thread-specific data
- Scheduling

Fork () and exec () semantics

- **Does** fork () **copy only the calling thread or all the threads?**
  - Often two versions available
- **What does exec() do?**
  - It replaces the address space, so all threads are replaced

Cancellation of thread

- **The termination of the thread before it is finished.**
- **Two general approaches:**
  - **Asynchronous cancellation** which ends the child immediately
    - **May leave shared data in a bad state**
    - **Some resources are not released**.
  - **Deferred cancellation**
    - **Use a flag that the child checks to see if it should cancel its execution**
    - **Gives a smooth ending**

Wire groupings (Thread Pools)

- **A server process can service its requests by creating a thread for each request**
  - Thread creation takes time
  - No control over the number of threads, which can increase the load on the system.
- **Solution**
  - Let's create a number of threads waiting for work
  - Advantages:
    - **The creation time only takes place at the beginning of the creation of the group of children**
    - **The number of running threads is limited by the size of the group**

thread-specific data

- **Allows each thread to have a private copy of data**
- **Useful when the control of the creation of the thread is limited (ie in a group of threads).**

Topics

- **The execution thread in the processes**
- **Multi-thread versus single thread (the thread)**
  - User level threads and kernel level threads
- **The challenges of "Threading"**
- **Examples of threads**

Examples of thread libraries

- **Pthreads**
- **Win32**
- **Java threads**

Pthreads

- **A POSIX standard (IEEE 1003.1c) of an API for the creation and synchronization of threads**
- **The API specifies the behavior of the thread library (its realization depends on the developer)**
- **Common in UNIX operating systems (Solaris, Linux, Mac OS X)**
- **Typical functions:**
  - pthread_create (& threadid, & attr, start_routine, arg)
  - pthread_exit (status)
  - pthread_join (threadid, status)
  - pthread_attr_init (& attr)

192.168.57.2 - siteDev* - SSH Secure Shell

File Edit View Window Help

Quick Connect Profiles

```
PTHREAD_CREATE(3)                                        PTHREAD_CREATE(3)

NAME
       pthread_create - create a new thread

SYNOPSIS
       #include <pthread.h>

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

DESCRIPTION
       pthread_create creates a new thread of control  that  executes  concur-
       rently  with  the  calling  thread. The new thread applies the function
       start_routine passing it arg as first argument. The new  thread  termi-
       nates  either explicitly, by calling pthread_exit(3), or implicitly, by
       returning from the start_routine function. The latter case  is  equiva-
       lent  to calling pthread_exit(3) with the result returned by start_rou-
       tine as exit code.

       The attr argument specifies thread attributes to be applied to the  new
       thread.   See  pthread_attr_init(3)  for  a  complete  list  of  thread
       attributes. The attr argument can also be NULL, in which  case  default
:
```

Connected to 192.168.57.2 SSH2 - aes128-cbc - hmac-md5 - none 80x24

Programming exercise with threads

Goal: Write a matrix multiplication program with several threads, to take advantage of several CPUs.

Program for multiplication with single thread of matrix A and B of order nxn

```
for (i = 0; i <n; i ++)
    for (j = 0; j <n; j ++) {
        C [i, j] = 0;
        for (k = 0; k <n; k ++)
            C [i, j] + = A [i, k] * B [k, j];
    }
```

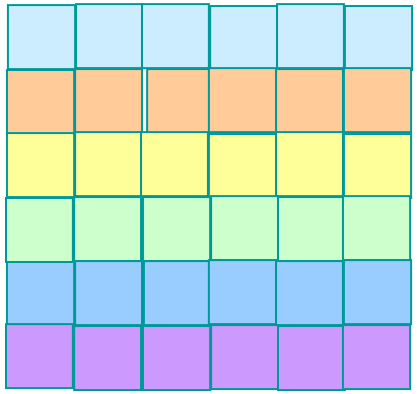
To make our life easier:
We have 6 CPUs and n is a multiple of 6

How to start? Ideas?

Matrix multiplication with multi-threads

Idea:

- creation of 6 threads
- each thread solves 1/6 of matrix C
- wait for the end of the 6 threads
- matrix C can now be used



Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5

Let's go!

```
pthread_t tid [6];
pthread_attr_t attr;
int i;

pthread_init_attr (& attr);
for (i = 0; i <6; i ++) / * creation of work threads * /
   pthread_create (& tid [i], & attr, worker, & i);

for (i = 0; i <6; i ++) / * let's wait until all are
finished * /
   pthread_join (tid [i], NULL);

/ * matrix C can now be used * /
...
```

Let's go!

```
void *worker (void * param)
{
   int i, j, k;
   int id = * ((int *) param); / * interpret the param
    as pointer to an integer * /
   int low = id * n / 6;
   int high = (id + 1) * n / 6;

   for (i = low; i <high; i ++)
      for (j = 0; j <n; j ++)
      {
         C [i, j] = 0;
         for (k = 0; k <n; k ++)
            C [i, j] = A [i, k] * B [k, j];
      }
   pthread_exit (0);
}
```

Let's go!

Does it work?

- Do we need to pass A, B, C and n as a parameter?
  - no, they are in shared memory, we are good
- Have the IDs been passed?
  - not really, pointers all receive the same address.

```
int id[6];
.
.
for (i = 0; i <6; i ++) / * create the working
threads * / {
    id [i] = i;
    pthread_create (& tid [i], & attr, worker, & id
    [i]);
}
```

Now does it work?

- should, ...

Win32 thread API

```
// create a thread
ThreadHandle = CreateThread (
 NULL, // default security attributes
 0, // default stack size
 Summation, // function to execute
 & Param, // parameter to thread function
 0, // default creation flags
& ThreadId); // returns the thread ID

if (ThreadHandle! = NULL) {
 WaitForSingleObject (ThreadHandle, INFINITE);
 CloseHandle (ThreadHandle);
 printf ("sum =% d \ n", Sum);
}
```

Java Threads

- **Java threads are created with a call to the start () method of a class that**
  - Extends the Thread class, or
  - Uses the Runnable interface:

```
public interface Runnable
{
 public abstract void run ();
}
```

- **Java threads are an important part of the Java language which offers a feature rich API.**

Extend the Thread class

```
class Worker1 extends Thread
{
 public void run () {
 System.out.println ("I Am a Worker Thread");
 }
}

public class First
{
 public static void main (String args []) {
 Worker1 runner = new Worker1 ();
 runner.start ();
 System.out.println ("I Am The Main Thread");
 }
}
```

The Runnable interface

```
class Worker2 implements Runnable
{
 public void run () {
 System.out.println ("I Am a Worker Thread");
 }
}
public class Second
{
 public static void main (String args []) {
 Runnable runner = new Worker2 ();
 Thread thrd = new Thread (runner);
 thrd.start ();

 System.out.println ("I Am The Main Thread");
 }
}
```

Attach Threads

```
class JoinableWorker implements Runnable
{
 public void run () {
 System.out.println ("Worker working");
 }
}

public class JoinExample
{
 public static void main (String [] args) {
 Thread task = new Thread (new JoinableWorker ());
 task.start ();

 try{task.join (); }
 catch (InterruptedException ie) {}

 System.out.println ("Worker done");
 }
}
```

Thread cancellation

```
Thread thrd = new Thread (new InterruptibleThread ());
Thrd.start ();

. . .

// now interrupt it
Thrd.interrupt ();
```

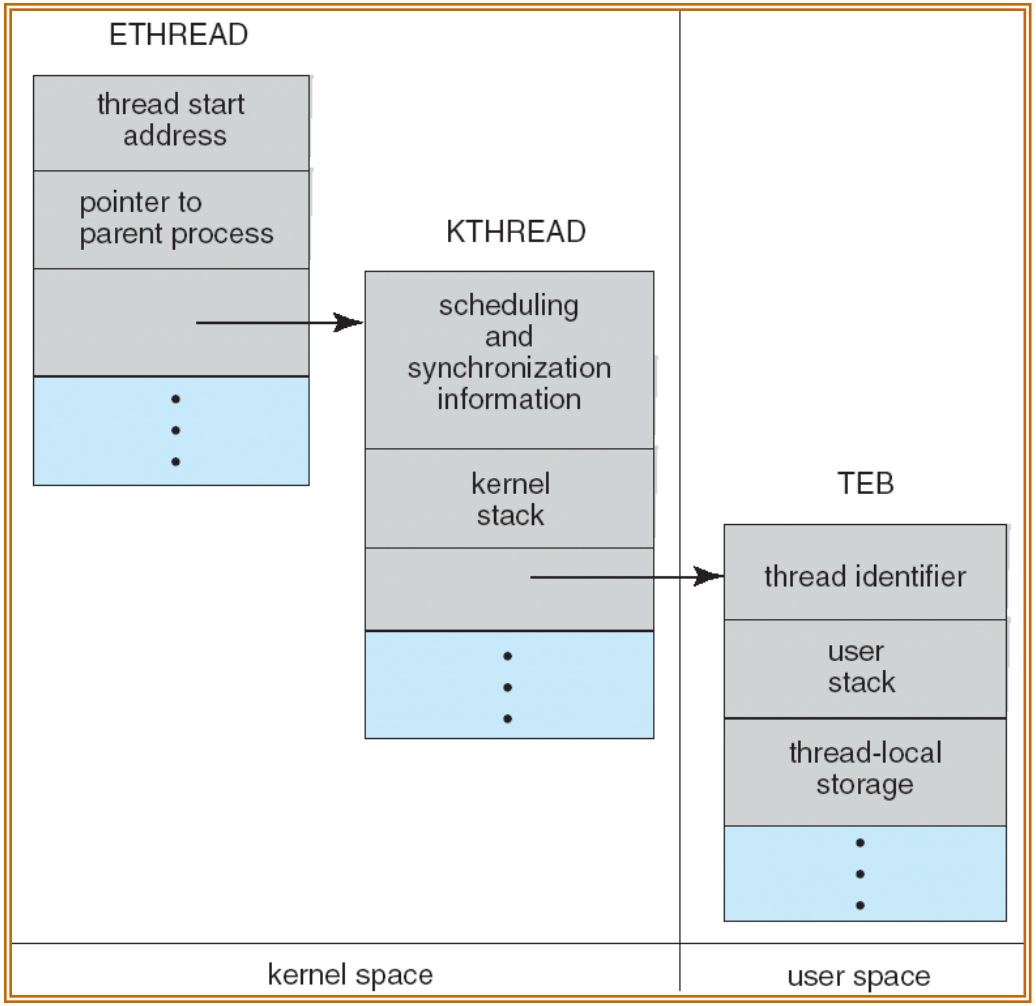
Examples of I/O Thread Implementation

- **Windows XP**
- **Linux**
- **Java**

Windows XP Threads

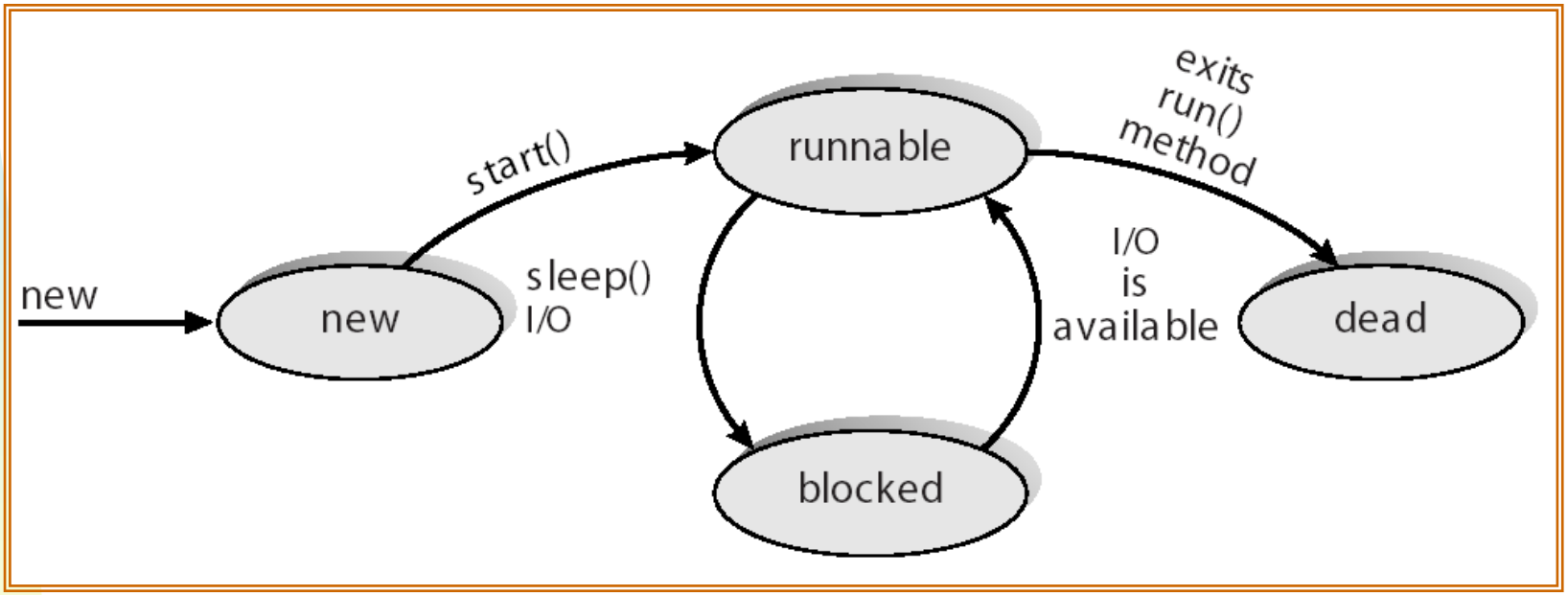
- **One to one model**
- **Each thread contains**
  - A thread identifier (id)
  - A set of registers
  - Different user and kernel stacks
  - Private data memory
- **The set of registers, stacks, and private memory form the** context **of the thread**
- **The main data structures of a thread include:**
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Windows XP Threads



Java threads

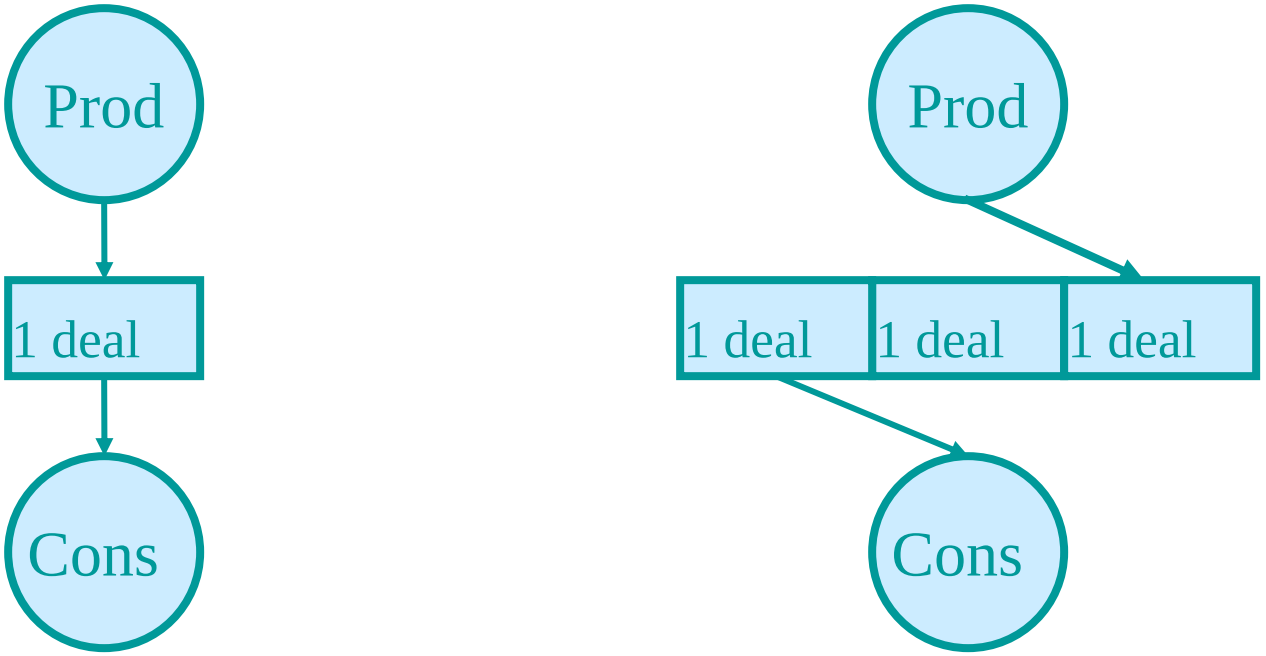
Java threads are managed by the JVM



The producer - consumer pb

- **A classic problem in the study of communicating processes**
  - a process *producer* produces data (e.g. records from a file) for a process *consumer*
  - a print pgm produces characters - consumed by a printer
  - an assembler produces object modules which will be consumed by the loader
- **Requires a buffer to store produced items (waiting to be consumed)**

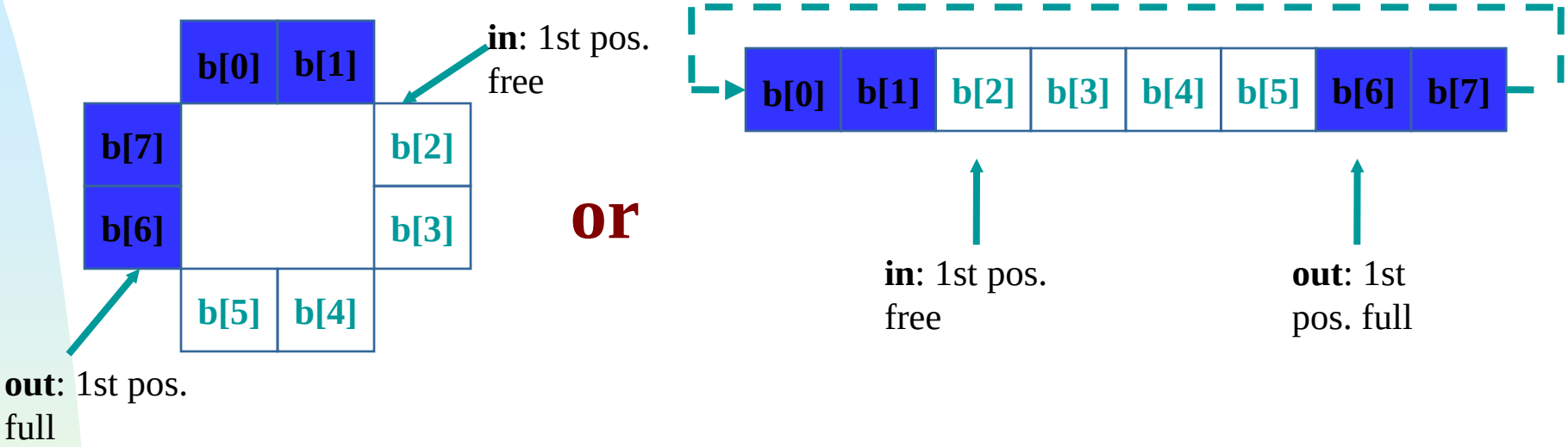
Communication buffers



If the buffer is of length 1, the producer and consumer must necessarily go at the same speed

Longer length buffers allow some independence. Eg. on the right the consumer was slower

The bounded buffer (bounded buffer)
a fundamental data structure in OS



The bounded buffer is in the memory shared between consumer and user

blue: full, white: free

The producer - consumer pb

```
public class Factory
{
 public Factory () {
 // first create the message buffer
 MessageQueue mailBox = new MessageQueue ();

 // now create the producer and consumer threads
 Thread producerThread = new Thread (new Producer (mailBox));
 Thread consumerThread = new Thread (new Consumer (mailBox));

 producerThread.start ();
 consumerThread.start ();
 }

 public static void main (String args []) {
 Factory server = new Factory ();
 }
}
```

Producer thread

```
class Producer implements Runnable
{
 private MessageQueue mbox;

 public Producer (MessageQueue mbox) {
 this.mbox = mbox;
 }

 public void run () {
 Message date;

 while (true) {
 SleepUtilities.nap ();
 message = new Dated();
 System.out.println ("Producer produced" + message);

 // produce an item & enter it into the buffer
 mbox.send (message);
 }
 }
}
```

Consumer thread

```
class Consumer implements Runnable
{
 private MessageQueue mbox;

 public Consumer (MessageQueue mbox) {
 this.mbox = mbox;
 }

 public void run () {
 Message date;

 while (true) {
 SleepUtilities.nap ();
 // consume an item from the buffer
 System.out.println ("Consumer wants to consume.");

 message = (Date) mbox.receive ();
 yew (message! = null)
  System.out.println ("Consumer consumed" + message);
 }
 }
}
```


Thank You!
Ευχαριστώ
ขอบคุณ
Vielen Dank
Merci
Teşekkürler
Dmnvwd
شكراً
متشكرم
Gracias
Grazie
Bedankt
Dankie
THANK YOU
Köszönettel
Obrigado!
شكريا
Díky
謝謝
GADDA GUEY
WAD MAHAD
SAN TAHAY
감사합니다
Urakoze