

Synchronizing Threads and Processes

Marek Biskup

Warsaw University

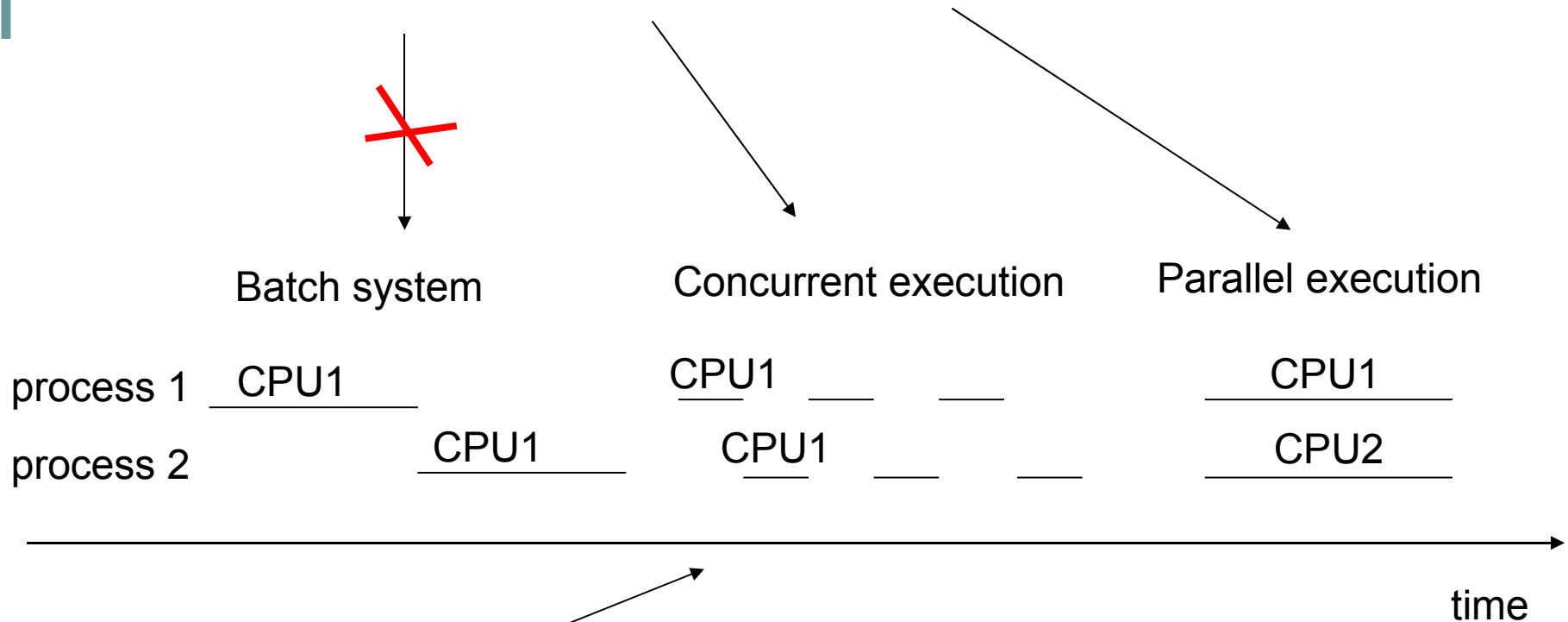
Outline

- **Why to synchronize processes and threads**
- **Means of synchronization**
 - Mutexes
 - Semaphores
 - Monitors
 - Condition variables
- **Threads in Java and C**

Motivation

- **Multiprogramming operating systems**
 - Single CPU may execute several programs (time sharing)
- **Some problems are intrinsically concurrent**
 - e.g. a web server
- **Hyper-threading technology**
 - Single processor executing a couple of threads at the same time
- **Multi-core CPUs emerging**
 - To achieve enough speed all processors have to be used

Concurrent programs

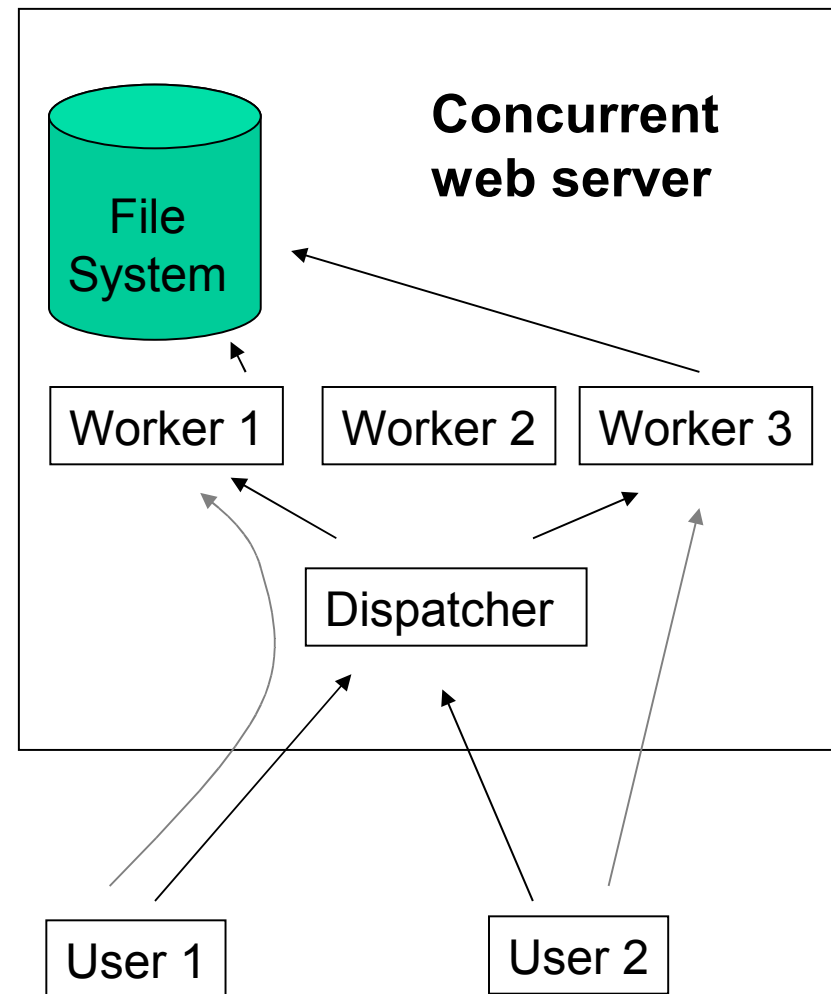


- **Time sharing**

- several programs executed concurrently on a single CPU
- a processor executes one program for certain time
- and switches to another one

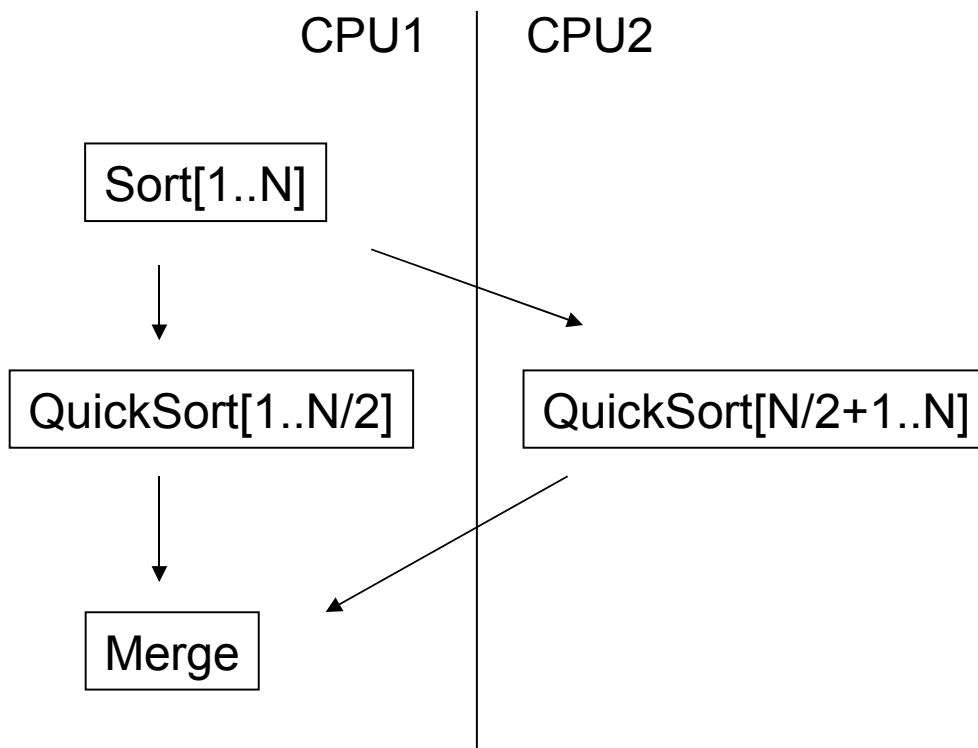
Example – concurrent web server

- **Before sending a web page the server has to read it from disk**
 - Synchronous IO with a single process would block the whole server
 - Solution:
 - Use asynchronous IO (a bit complicated)
 - **Use multiple processes/threads**
- **Each process serves one client:**
 - Reads the request from a socket
 - Reads a local file to be send
 - Writes results to the socket



A parallel program

Parallel Sort



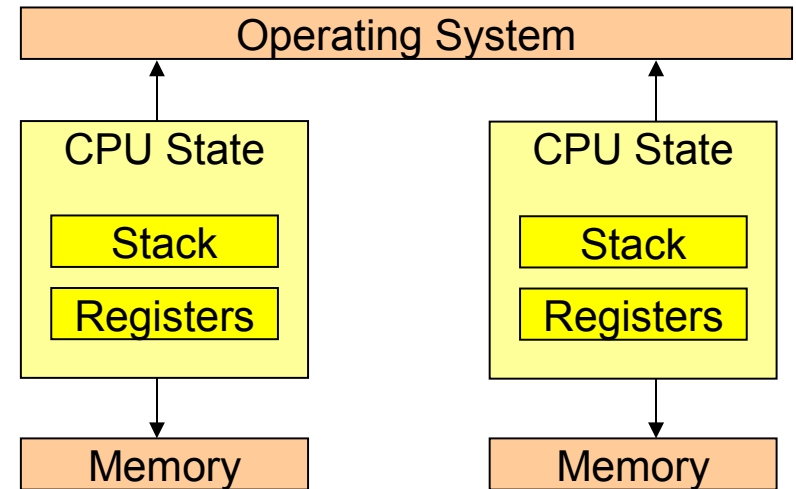
- Problems
 - Start a program on the second CPU
 - Exchange data between CPUs
 - Wait for results

- **With two processors: ~ 2x faster**

Multiprogramming systems

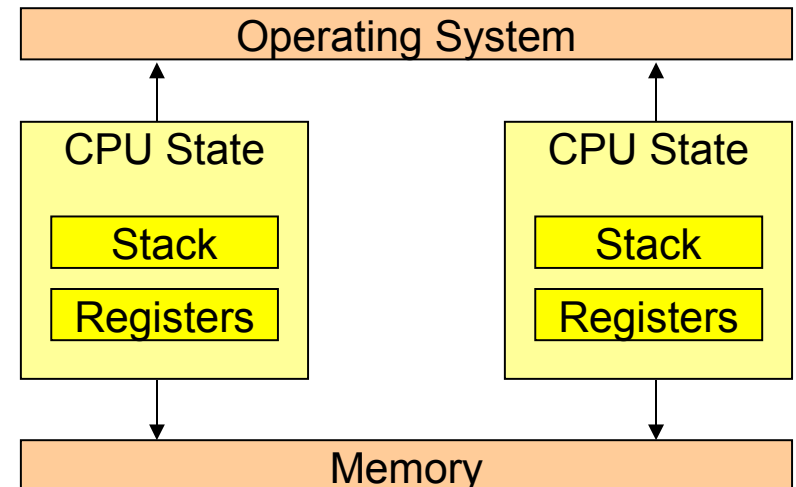
Processes

- Different CPU state
- Different memory space
- Communication via operating system mechanisms
 - Messages, shared memory (allocated using O.S.), Sockets



Threads

- Exist within a single process
- Same (global) memory space
- Different CPU state
- Communication via memory
- Faster switching

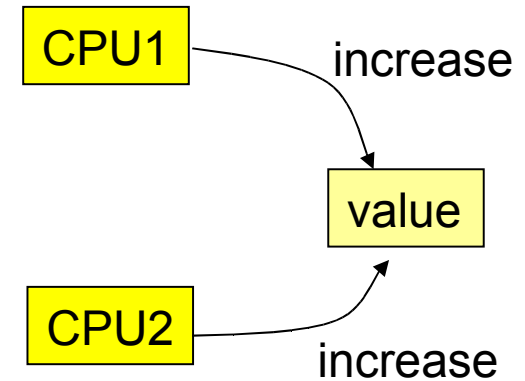


Accessing a memory location

- Example: Two CPUs execute the same function: add one to a global variable

```

int value; // global
void increase() {
    int x = value
    x++;
    value = x;
}
  
```



		value == 1		
CPU1:			CPU2:	
1.	<code>int x = value;</code>			<code>// 1</code>
2.			<code>int x = value;</code>	<code>// 1</code>
3.			<code>x++;</code>	<code>// 2</code>
4.	<code>x++;</code>			
5.			<code>value = x;</code>	<code>// 2</code>
6.	<code>value = x;</code>			

value is 2 instead of 3!

Accessing a memory location

- **The same example. Reading/writing an int (4 bytes) is byte-by-byte**
 - CPU1 reads the value (255: |0|0|0|255|)
 - CPU1 writes the new result: 256 : |0|0|1|0|, first the 3 upper bytes
 - CPU2 reads the value (511: |0|0|1|255|)
 - CPU1 writes the last byte (256: |0|0|1|0|)
 - CPU2 writes the result (512: |0|0|2|0|)
- **We get 512 instead of 257!**
- **Luckily most architectures read whole ints at once**

The access to a common variable must be blocked when it is accessed by a processor

Critical section

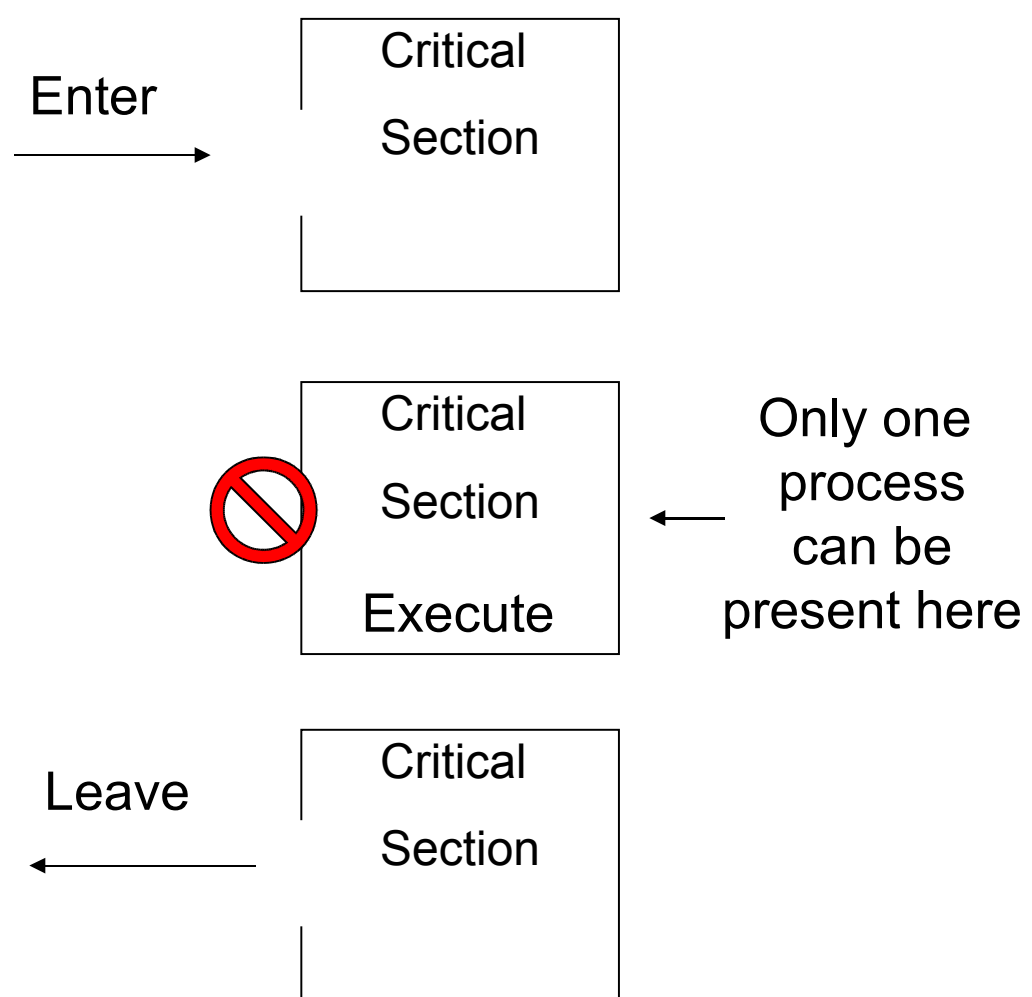
- A region of code that can only be executed by one thread or process at a time
- Atomic execution (synonym) – execution of code that cannot be interfered with by another process

```

int value; // global
void increase() {
    int x = value
    x++;
    value = x;
}
  
```

Critical section

time



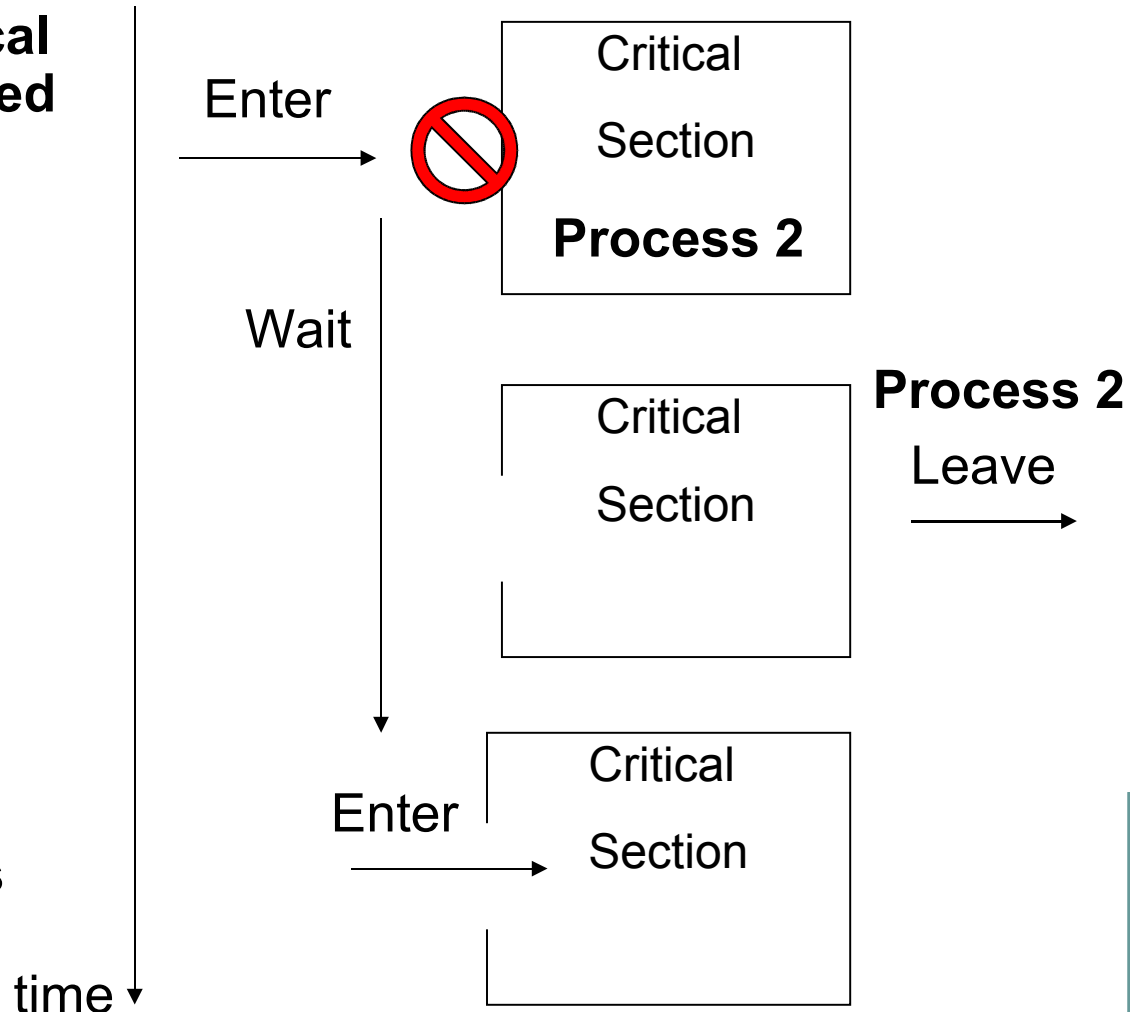
Critical Section 2

- Process entering a critical section should be blocked until the section is free
- How to check if the section is free?

```

bool lock = false;
while (lock == true)
    ; // empty instruction
lock = true;
    // critical section
lock = false;
  
```

- Wrong! – two processes may check lock at the same time and enter



Hardware support

- **The test-and-set instruction**
 - Executed atomically

This is sufficient to synchronize processes

```
bool testAndSet(int* lock) {
    // atomic instruction !
    if (*lock == false) {
        *lock = true;
        return false;
    }
    else
        return true;
}
```

Often called
SpinLock

```
int value;
int lock = 0;
void increase() {
    while (testAndSet(&lock))
        ; // just wait
    int x = value; // now lock == 1
    x++;
    value = x;
    lock = 0; // release
}
```

Active waiting! – the CPU keeps on testing the variable.

On multiprogramming systems, the CPU should be assigned to another task.


O.S. support is needed.

Mutexes

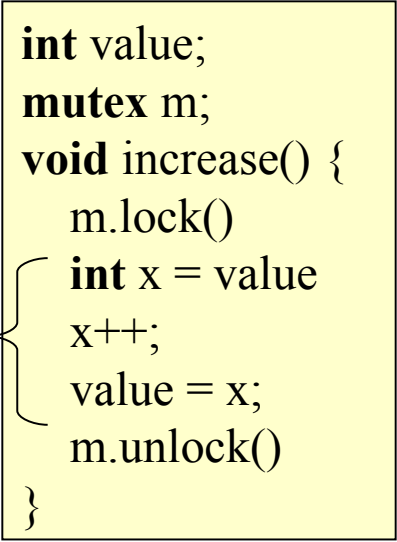
Operations:

- **lock()**
 - If the mutex is locked
 - Release the CPU (sleep)
 - Wake up when the mutex is unlocked
 - Lock the mutex and pass over
- **unlock()**
 - Unlock the mutex
 - Wake up one process waiting for the lock (if any)
 - May be executed only by the process that owns the mutex (executed lock())

Atomic once the process is woken up



```
int value;  
mutex m;  
void increase() {  
    m.lock()  
    int x = value  
    x++;  
    value = x;  
    m.unlock()  
}
```



Critical section

Bounded buffer

- **Buffer of a fixed size**
- **Processes can read from it and write to it**
 - Readers should be blocked if nothing to read
 - Writers should be blocked if no space to write
- **put/get must be exclusive (critical section)**

```
mutex m;
int buffer[N];
int items = 0;
```

```
void put(int value) {
    m.lock();
    if (items == N)
        wait(); // ???
    buffer[items++] = value;
    m.unlock();
}
```

```
int get() {
    m.lock();
    if (items == 0)
        wait(); // ???
    int rv = buffer[--items];
    m.unlock();
    return rv;
}
```

How can we wait?

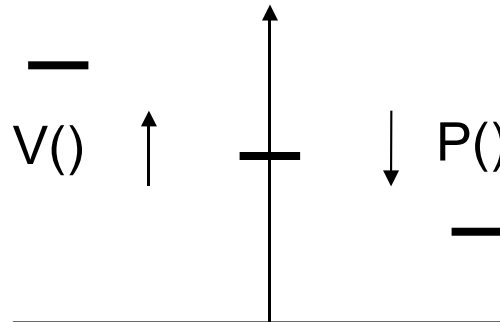
- use another mutex (complicated)
- introduce more advanced means of synchronization

Semaphores

- **An object which holds a value $n \geq 0$**
- **Operations**
 - `Init(int v)`
 - $n := v;$
 - `V()` – atomic
 - $n++;$
 - `P()` – atomic once $n > 0$ is detected
 - `await $n > 0;$`
 - $n--;$
- **Semaphores are considered nonstructural, obsolete, error-prone**
 - But they are useful

Bounded Buffer

```
semaphore empty(N);
semaphore full(0);
mutex m;
int buffer[N];
int items = 0;
```



```
void put(int value) {
    empty.P();
    m.lock();
    buffer[items++] = value;
    m.unlock();
    full.V();
}
```

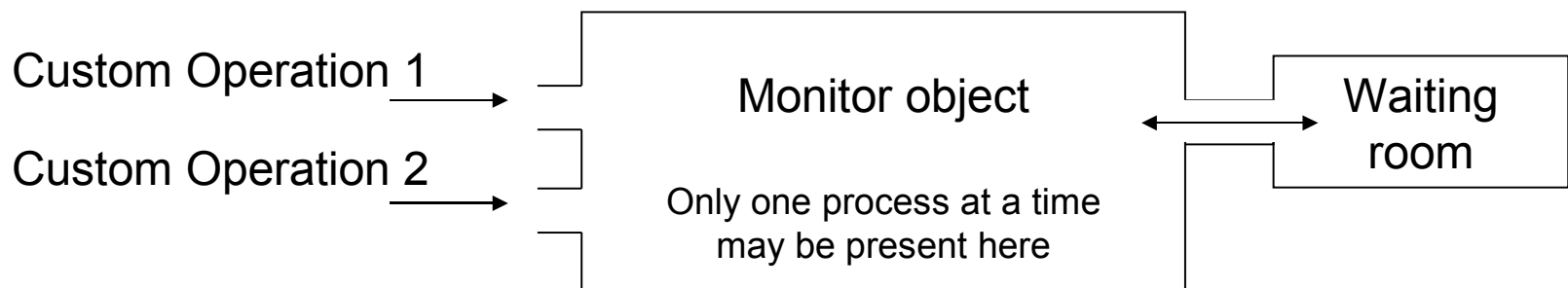
```
int get(int value) {
    full.P();
    m.lock();
    int rv = buffer[--items];
    m.unlock();
    empty.V();
    return rv;
}
```

Monitor Objects

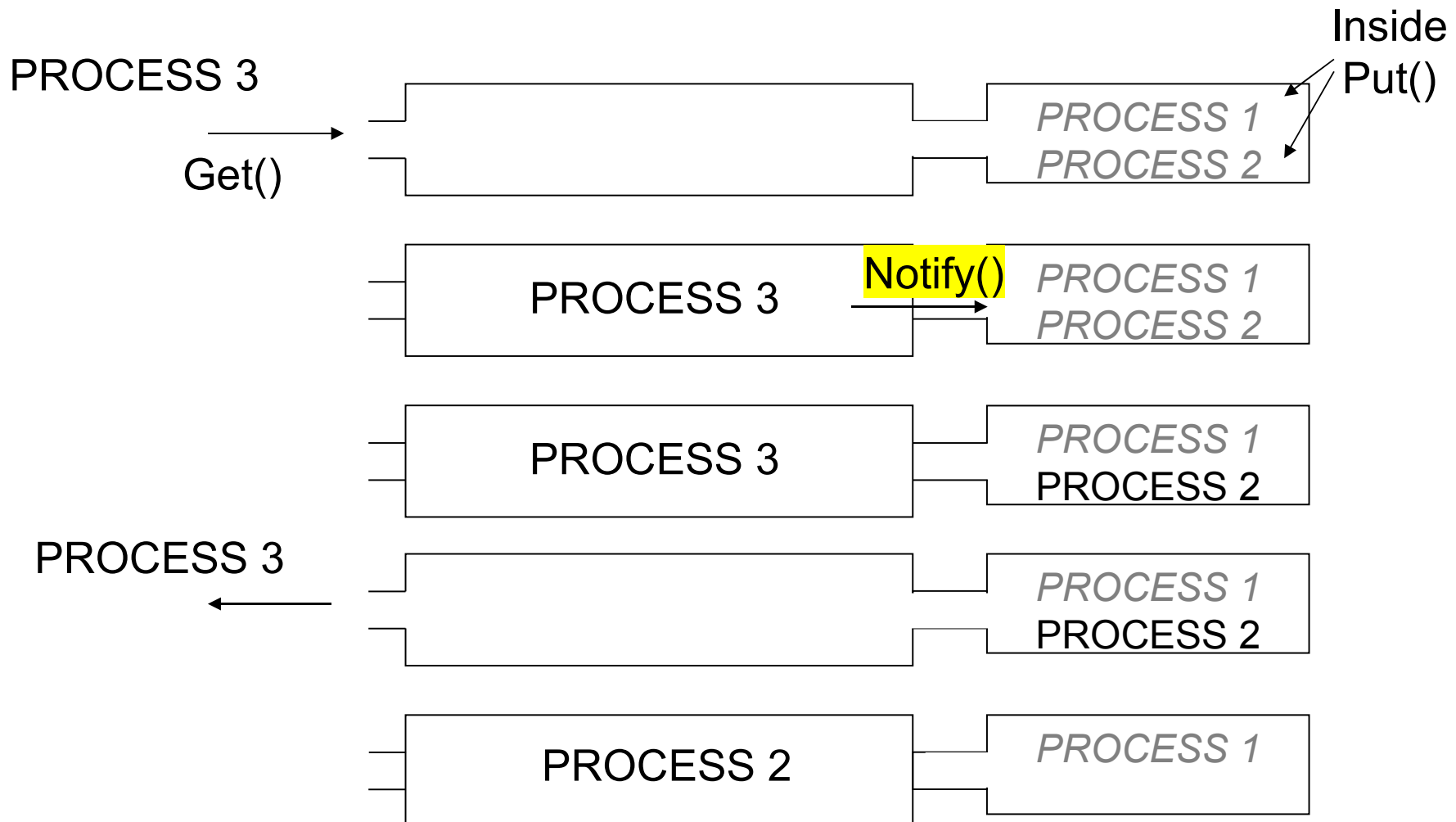
- **Previous example was rather complicated**
 - We'd prefer to use the lock-wait strategy as in the first trial with mutexes
- **Monitors – higher level synchronization**
- **Operations:**
 - Custom operations (fully synchronized)
 - Wait() – release the monitor and wait
 - Notify() – wake up one of the waiting processes
 - NotifyAll() – wake up all the waiting processes

```

void put(int value) {
    m.lock();
    if (items == N)
        wait(); // ???
    buffer[items++] = value;
    m.unlock();
}
  
```



Monitor Objects – Notify()



Java monitors

- Exactly this kind of monitors has been implemented in Java

```

public class BoundedBuffer {
    int A[], N; // buffer and its size
    int k = 0; // filled elements

    public BoundedBuffer(int size) {
        N = size;
        A = new int[N];
    }
    // custom operations
    public synchronized void put(int r) {...}
    public synchronized int get() { ... }
}
  
```

```

public synchronized void put(int r)
    throws InterruptedException {
    while (k == N)
        wait();           // full buffer
    A[k++] = r;
    notifyAll();
}
  
```

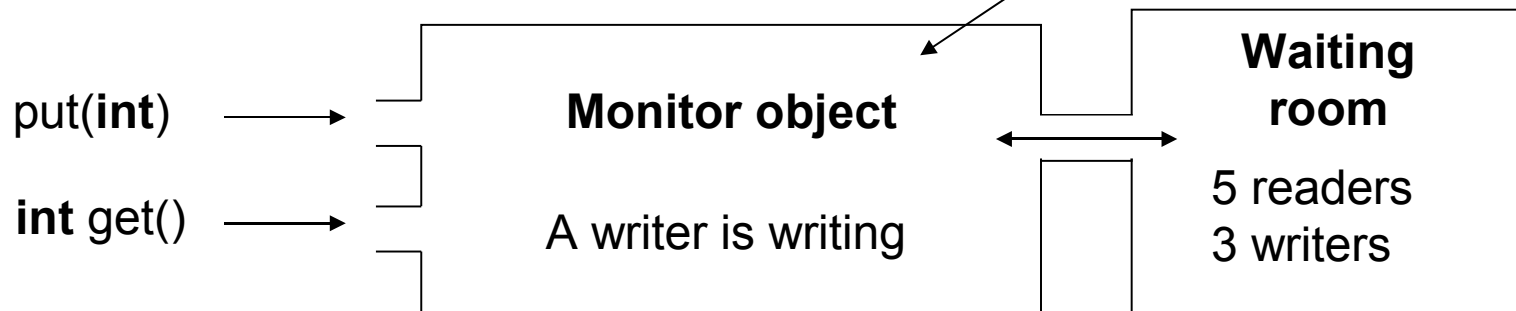
```

public synchronized int get()
    throws InterruptedException {
    while (k == 0)
        wait();           // empty buffer
    notifyAll();
    return A[--k];
}
  
```

Java monitors cont.

- **Why notifyAll() instead of notify()?**
- **Why this is not optimal?**

The buffer will be full after this write



Notify() would wake up just one process. Writer => deadlock!

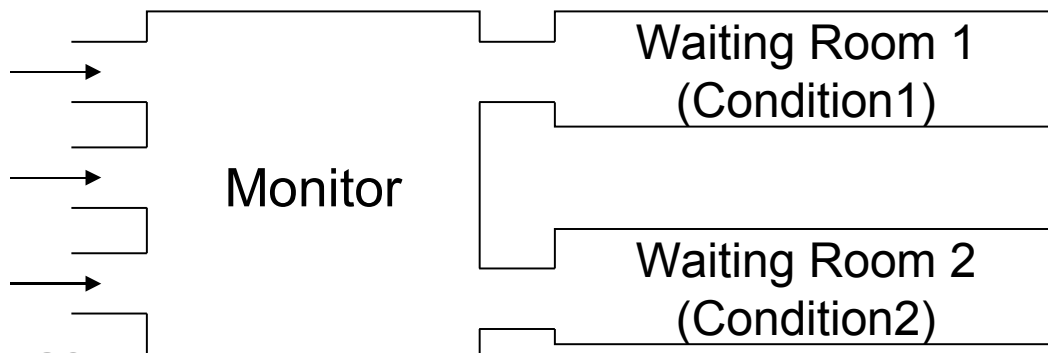
NotifyAll() wakes up everybody in the waiting room

But only readers can continue!

We would prefer to have separate waiting rooms for readers and writers.

Condition Variables

- A monitor objects holds several *condition variables* – waiting rooms
- When waiting, a condition should be specified
- When notifying, a condition should be specified
 - Wakes up only processes waiting on that condition



Pseudo-java code:

```
Condition readers, writers;
```

```
put(int r) {
    while (k == N)
        writers.wait(); // full buffer
    A[k++] = r;
    readers.notify();
}
```

```
int get() {
    while (k == 0)
        readers.wait(); // empty buffer
    writers.notify();
    return A[--k];
}
```

Bounded buffer with conditions

- **Real Java code:**
 - Monitor's lock must be managed explicitly (synchronized keyword doesn't work)
 - Condition variables are associated with that lock. In the wait method:
 - Lock is released when waiting
 - Lock is reacquired when waking up
 - Operation names: `await()`, `signal()` and `signalAll()`

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class BoundedBuffer {
    int N, A[], k = 0;
    private ReentrantLock lock = new ReentrantLock();
    Condition readers, writers; // ...
}
```

The constructor:

```
public BoundedBuffer(int size) {
    N = size;
    A = new int[N];
    readers = lock.newCondition();
    writers = lock.newCondition();
}
```

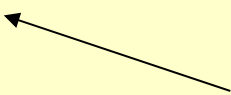
Bounded buffer with conditions 2

■ Methods code

- Note: there is no **synchronized** statement (explicit locking)

```
public int get()
    throws InterruptedException {
    lock.lock();
    try {
        while (k == 0)
            readers.await();
        writers.signal();
        return A[--k];
    } finally {
        lock.unlock();
    }
}
```

lock must be
locked when
calling signal!



```
public void put(int r)
    throws InterruptedException {
    lock.lock();
    try {
        while (k == N)
            writers.await();
        A[k++] = r;
        readers.signal();
    } finally {
        lock.unlock();
    }
}
```

Java: how to create a thread

- **Create your thread's class:**
 - Inherit from the Thread class
 - Implement the run() method
- **Create such an object and call start()**
 - Calling start() will create a new thread which starts in the run() method
- **Use the constructor to pass initial data to the thread**
 - E.g. in the BoundedBuffer example all readers and writers have to know the buffer object

```
public class MyThread extends Thread {  
    // local thread's data  
    public MyThread(/* Parameters */) {  
        // use constructor parameters to pass  
        // initial data to the thread  
        // e.g. global data structures  
    }  
    public void run() {  
        // thread's own code  
    }  
}
```

```
myThread = new MyThread(globalData);  
myThread.start();
```

Java: Complete Example

```
public class Producer extends Thread {  
    BoundedBuffer buff;  
    public Producer(BoundedBuffer b) {  
        buff = b;  
    }  
    public void run() {  
        try {  
            for (int i = 0; i < 100; i++)  
                buff.put(i);  
        } catch (InterruptedException e) {}  
    }  
}
```

```
public class Consumer extends Thread {  
    BoundedBuffer buff;  
    public Consumer(BoundedBuffer b) {  
        buff = b;  
    }  
    public void run() {  
        try {  
            for (int i = 0; i < 100; i++)  
                System.out.println( buff.get() );  
        } catch (InterruptedException e) {}  
    }  
}
```

```
public class TestBoundedBuffer {  
    public static void main(String[] args) {  
        BoundedBuffer b = new BoundedBuffer(10);  
        Producer p = new Producer(b);  
        p.start();  
        Consumer c = new Consumer(b);  
        c.start();  
    }  
}
```

C language threads

- **Pthreads – POSIX threads standard**
 - Just API specification
 - Implementations available for various systems (windows, linux, other unices)
- **Synchronization:**
 - mutexes
 - condition variables
- **Same semantics as in Java (almost)**

Datatypes:

```
pthread_t
pthread_cond_t
pthread_mutex_t
```

```
void* start_routine(void*);
```

Thread functions:

```
pthread_create(&thread,
               attr, start_routine, arg)
pthread_exit(ret_val)
pthread_join(thread_id);
```

Mutex functions:

```
pthread_mutex_init(mutex, attr);
pthread_mutex_destroy(mutex);
pthread_mutex_lock(mutex);
pthread_mutex_unlock(mutex);
```

Condition functions:

```
pthread_cond_init(condition, 0);
pthread_cond_destroy(condition);
pthread_cond_wait(condition, mutex);
pthread_cond_signal(condition);
pthread_cond_broadcast(condition);
```

New concurrent Java API

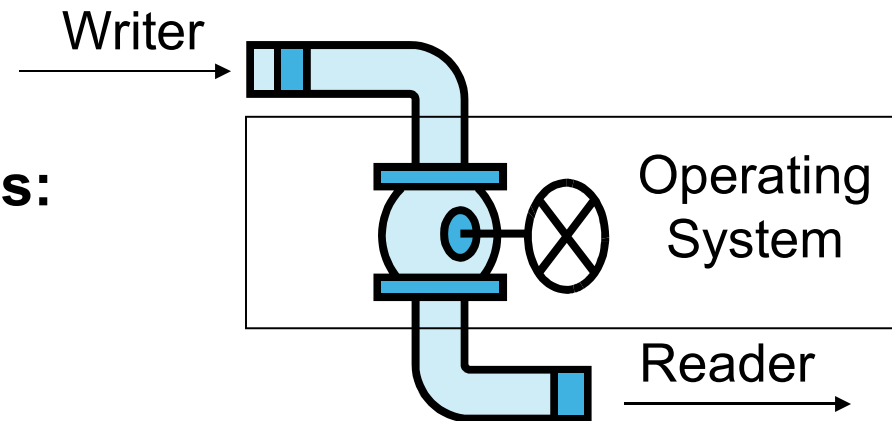
- **From Java 1.5 – new concurrent API: `java.util.concurrent`**
- **Atomic datatypes: `java.util.concurrent.atomic`**
 - `AtomicBoolean`
 - `AtomicInteger`
 - `AtomicReference<E>`
 - `AtomicIntegerArray`
 - Operations: `get`, `set`, `getAndSet`, `compareAndSet`, `incrementAndGet`
- **New means of synchronization:**
 - `Semaphores`
 - `Condition`, `Lock`
 - `ReadWriteLock` (allow many readers but writing is exclusive)
- **Synchronized datastructures**
 - `BlockingQueue<E>` – more or less our `BoundedBuffer`)
 - `ConcurrentMap<K, V>` – additional synchronized methods: `putIfAbsent()`, `remove()`, `replace()`

Interprocess communication in UNIX

- **Pipes**
 - Two file descriptors connected to each other
 - Created before forking the second process
- **Fifos**
 - Special file in a file system.
 - One process opens it for reading and the other for writing
- **Message queues**
 - Structured messages
- **Shared memory**
 - Processes can locate shared memory segments using *keys*
- **Signals**
 - One process may send a signal to another one
 - Breaks the normal execution of a sequential program to execute the signal handler
 - Very primitive way of communication

Pipes, FIFOs

- „Virtual” file in memory
 - One process writes data
 - The other reads data
- **Standard file access functions:**
 - `read()`, `write()`
 - `fprintf()`, `fscanf()`
- **Pipes**
 - Only for related processes (parent-child)
- **FIFOs**
 - For any processes
 - Can be found via the file system (special file)



Message queues, shared memory

- **Both reside within the kernel**
 - Can be found using O.S. calls
 - ID of the queue/shmem has to be known to the processes
- **Message queues:**
 - `msgsnd()`
 - `msgrcv()`
- **Shared memory**
 - Just read/write data as with threads
 - Don't forget about synchronization (semaphores, mutexes, etc.)!



Difficulties

- **Testing is not enough**
 - Problems with synchronization may occur very rarely. E.g. the famous AT&T crash ('90)
- **Deadlocks**
 - Two processes waiting for each other
- **Starvation**
 - Some processes may never be allowed to enter a critical section

Summary

- **Concurrent programs are inevitable for efficient usage of a multiprocessor system**
- **Threads and processes within a concurrent application have to be synchronized**
- **There are typical means of synchronization: mutexes, semaphores, monitors**
- **Other interprocess communication mechanisms are based on messages**
- **Writing and testing a concurrent application is much more difficult than a sequential one**

Further reading and materials used

- ***Concurrent Programming*** course at Warsaw University
- **Stevens – *UNIX Network Programming, Volume 2: Interprocess Communication***
- **Wikipedia**
- **Contact: mbiskup@mimuw.edu.pl**