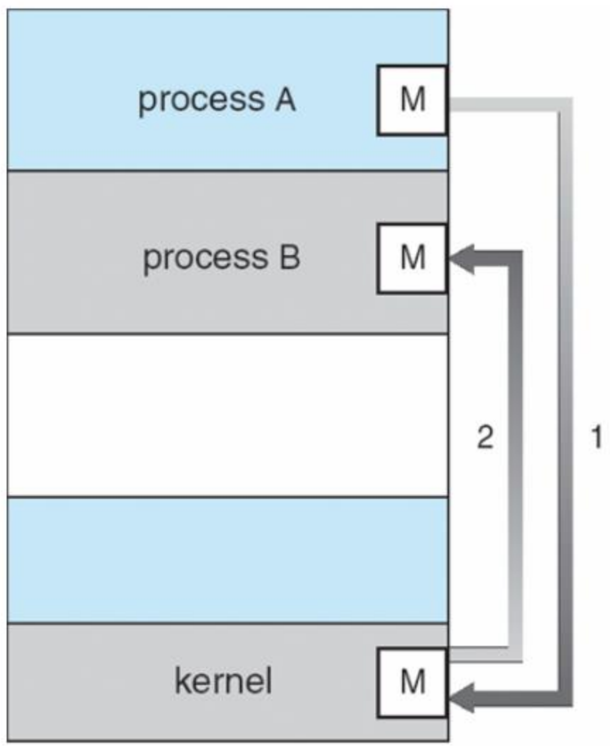
Inter-Process Communication

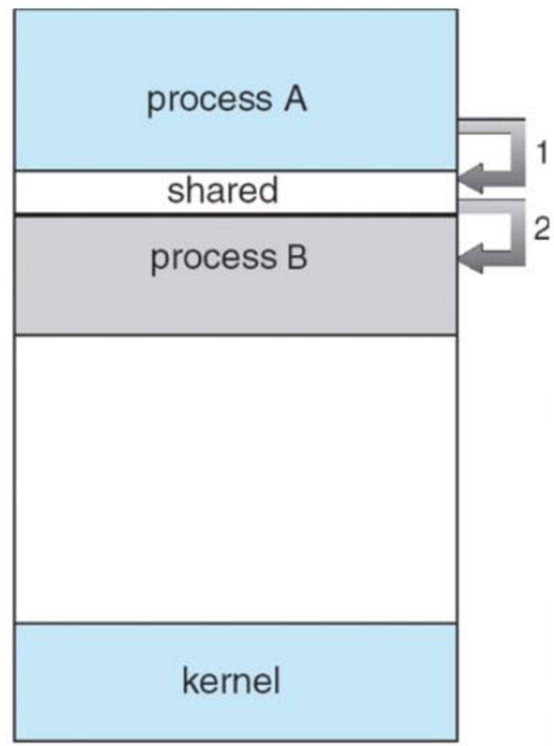
How can we make processes communicate (sharing information)?

Two methods: **message passing** and **shared memory**.

Message passing an shared memory



Communication take place by means of messages exchanged between the cooperating processes.



A region of memory that is shared by cooperating process is established. Process can then exchange information by reading and writing data to the shared region.

Message passing

Messages can be exchanged between processes either directly or indirectly using a common mailbox.

The recipient process usually must give its permission for communication to take place with an accept connection system call.

Message-passing is **useful** for exchanging **smaller amounts of data**, because no conflicts need be avoided.

Easier to implement compared to the shared memory approach.

shared memory

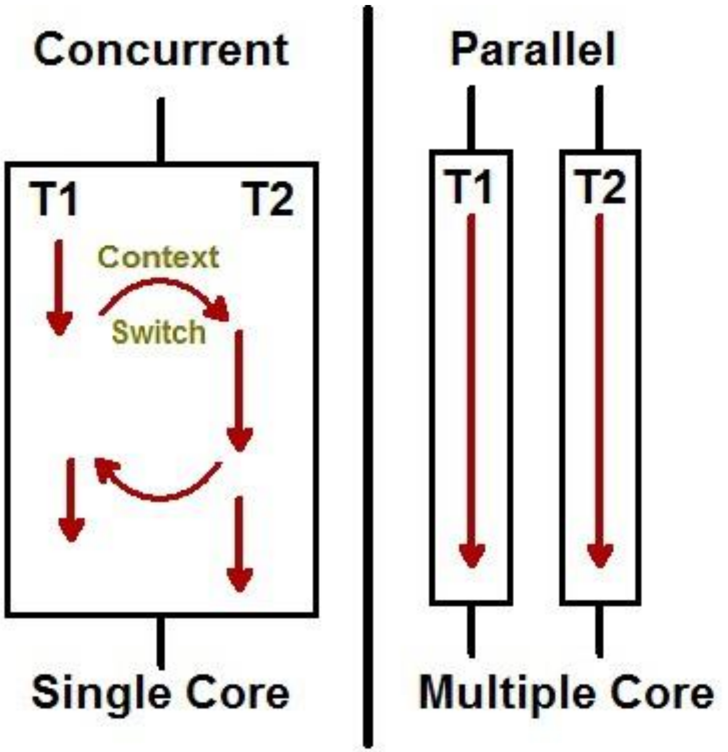
Processes can communicate by reading and writing to shared memory.

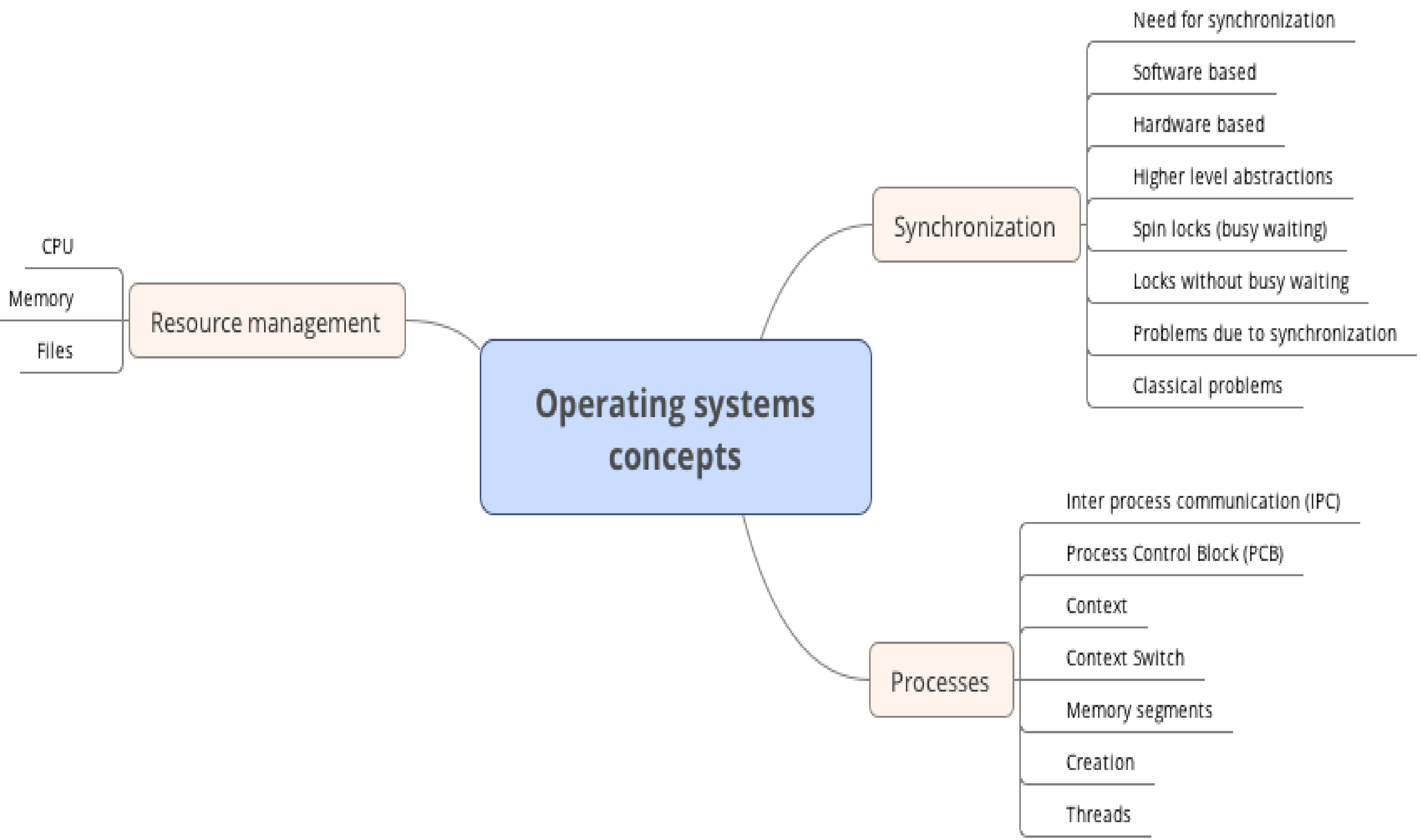
Normally, the OS tries to prevent one process from accessing another process's memory. Shared-memory requires that two or more processes agree to remove this restriction.

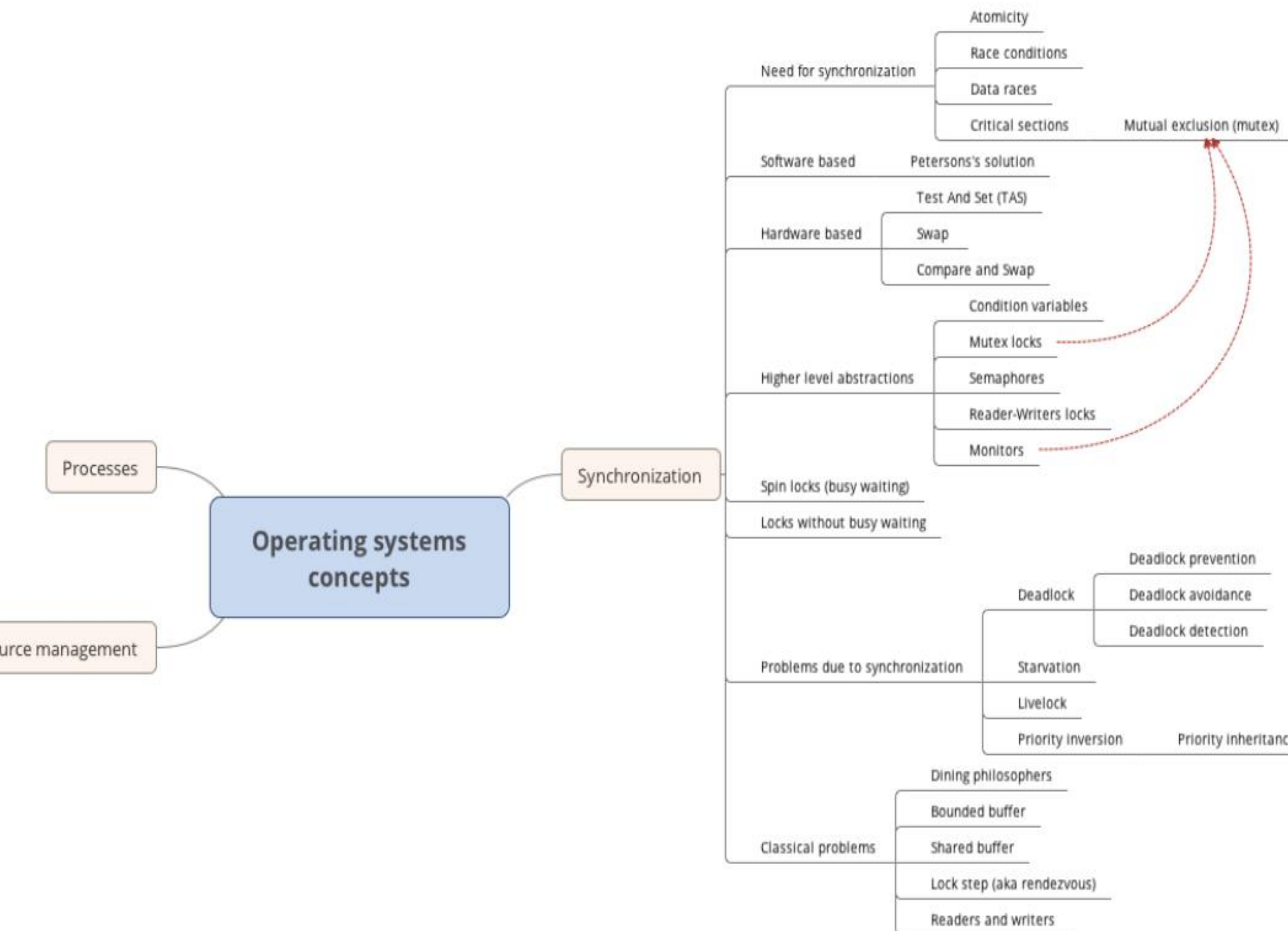
Shared memory **allows maximum speed** and convenience of communication, since it can be done at memory transfer speeds.

Problems: protection and synchronization between the processes sharing memory.

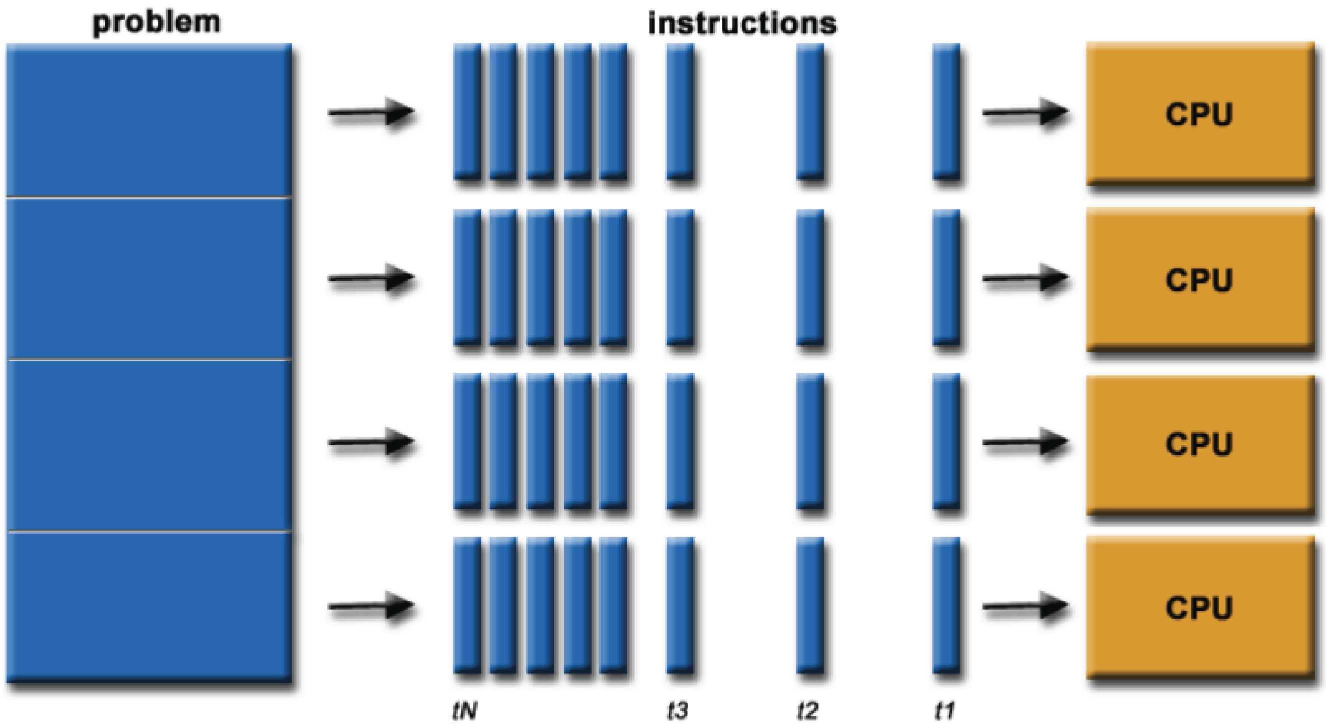
Concurrency/ Synchronization/ Deadlocks




Operating systems concepts
Resource management
CPU
Memory
Files
Synchronization
Need for synchronization
Software based
Hardware based
Higher level abstractions
Spin locks (busy waiting)
Locks without busy waiting
Problems due to synchronization
Classical problems
Processes
Inter process communication (IPC)
Process Control Block (PCB)
Context
Context Switch
Memory segments
Creation
Threads


Operating systems concepts
Processes
urce management
Synchronization
Need for synchronization
Atomicity
Race conditions
Data races
Critical sections
Mutual exclusion (mutex)
Software based
Petersons's solution
Hardware based
Test And Set (TAS)
Swap
Compare and Swap
Higher level abstractions
Condition variables
Mutex locks
Semaphores
Reader-Writers locks
Monitors
Spin locks (busy waiting)
Locks without busy waiting
Problems due to synchronization
Deadlock
Deadlock prevention
Deadlock avoidance
Deadlock detection
Starvation
Livelock
Priority inversion
Priority inheritanc
Classical problems
Dining philosophers
Bounded buffer
Shared buffer
Lock step (aka rendezvous)
Readers and writers

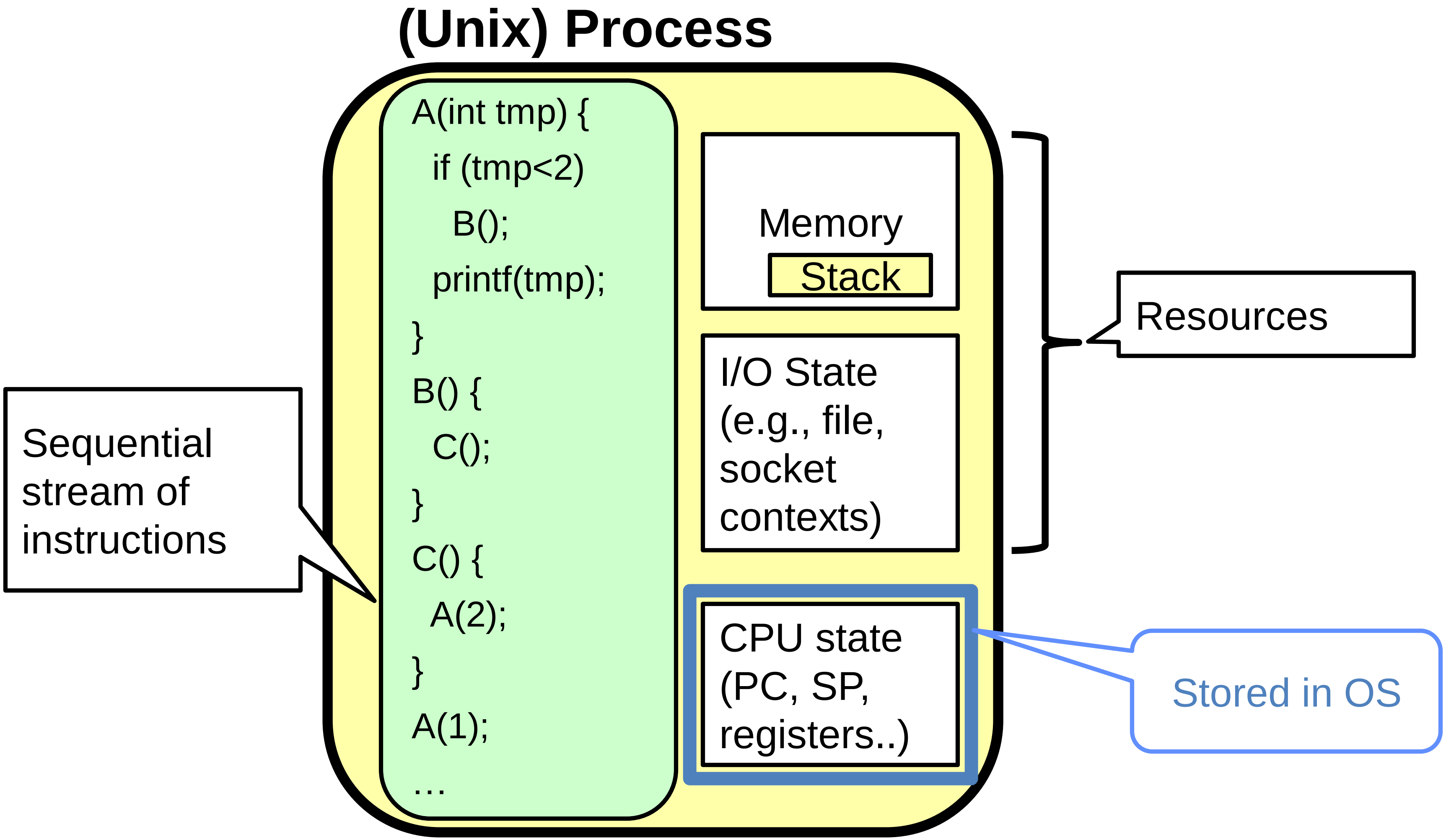
Large problems can often be divided into smaller ones, which are then solved in **parallel** (at the same time) on multiple physical CPUs.



This is not the same as multithreading. Multithreading can be done on a single core CPU. In such a case, two threads can never execute at the same time on the CPU**.**

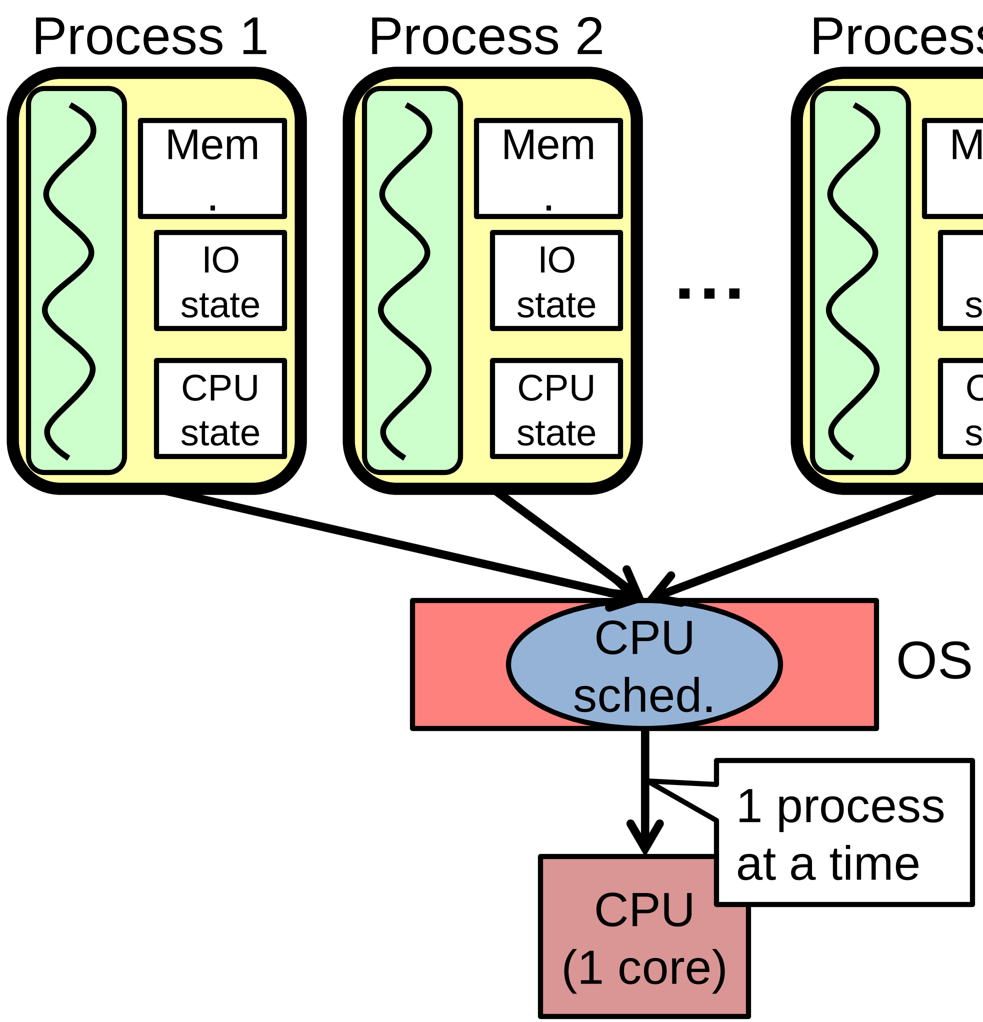
Source: https://en.wikipedia.org/wiki/Parallel_computing

Process



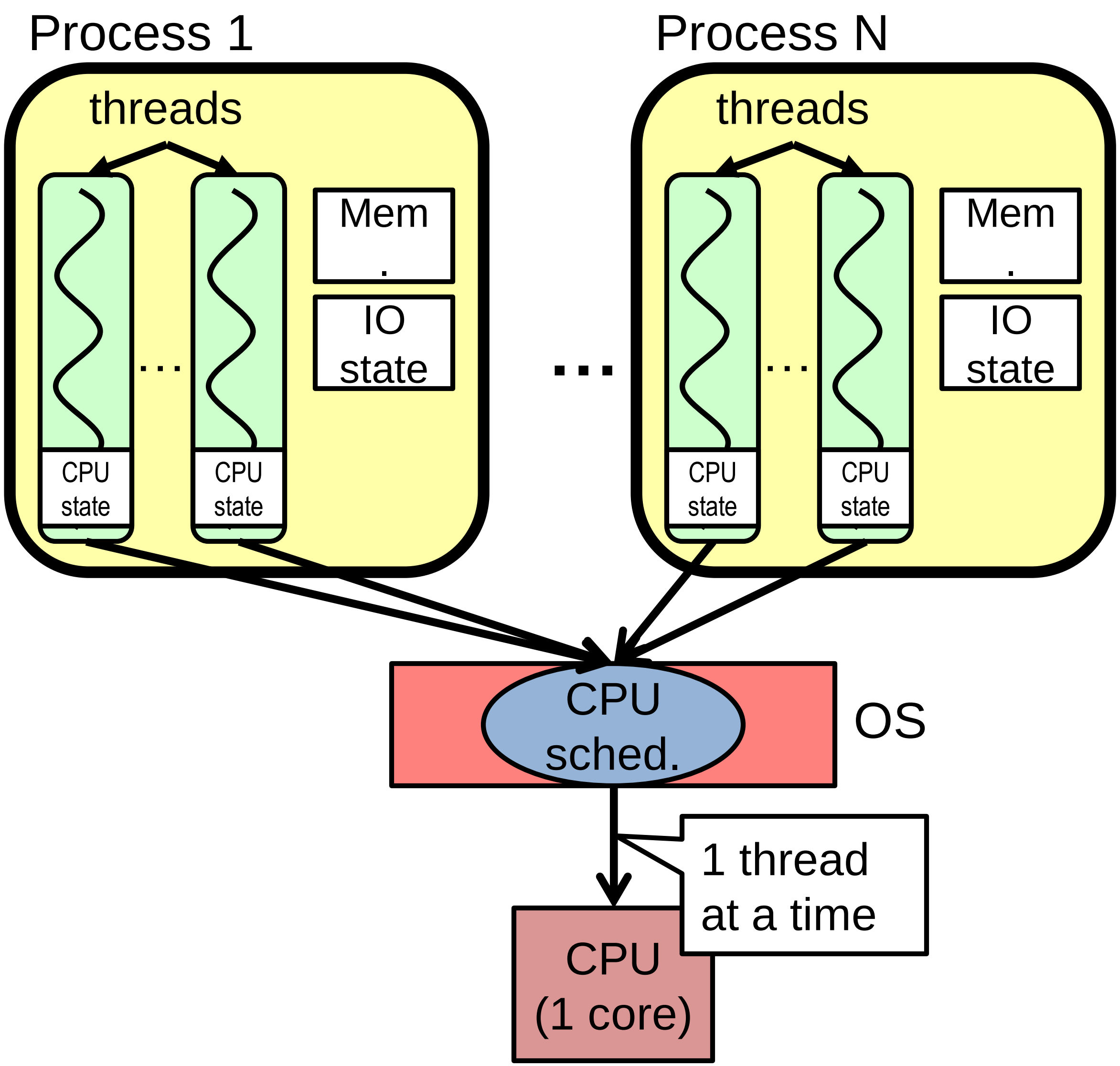
(Unix) Process
A(int tmp) {
if (tmp<2)
B();
printf(tmp);
}
B() {
C();
}
C() {
A(2);
}
A(1);
…
Memory
Stack
I/O State (e.g., file, socket contexts)
CPU state (PC, SP, registers..)
Resources
Sequential stream of instructions
Stored in OS

Putting it together: Processes



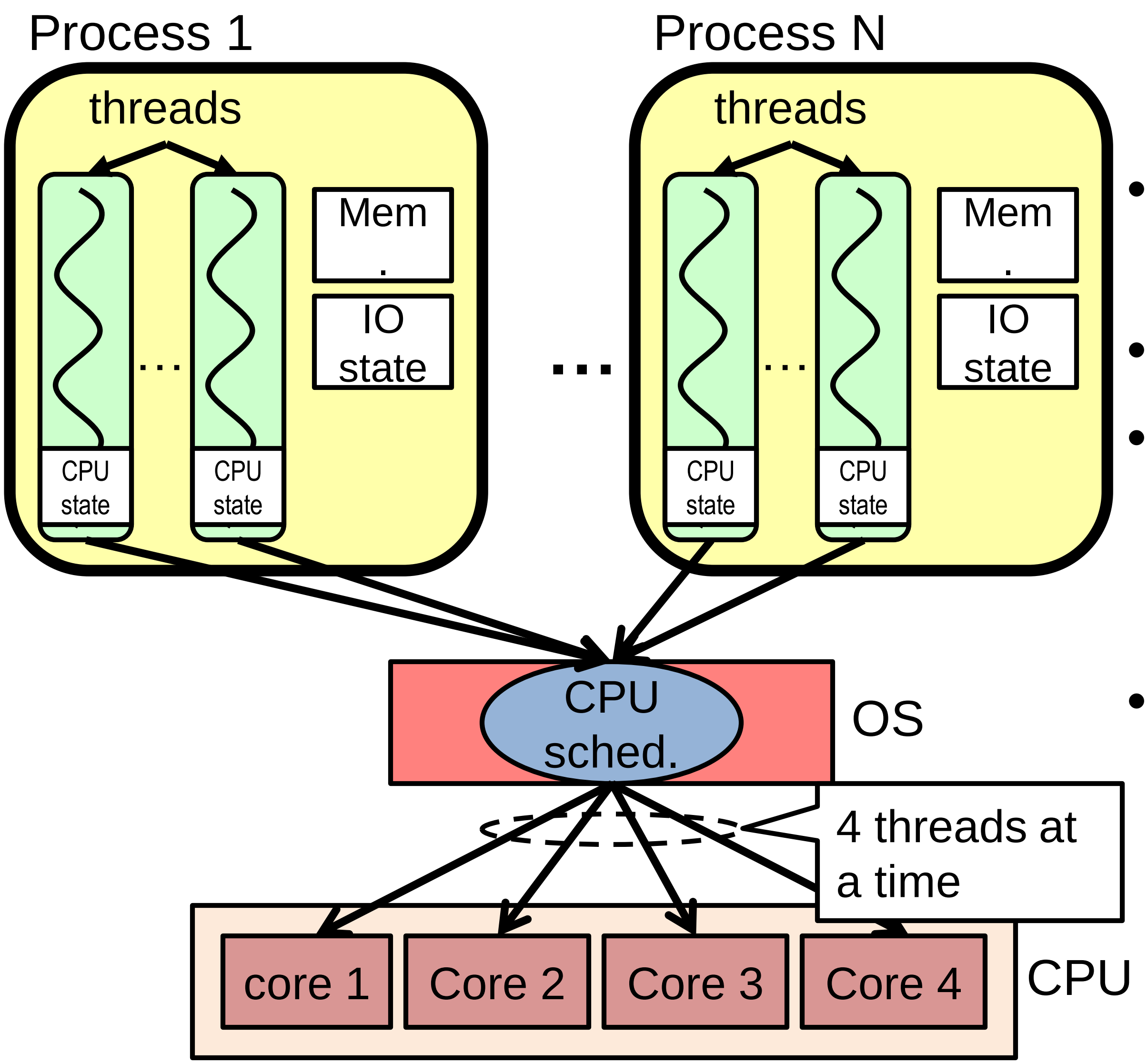
- Switch overhead: high
  - CPU state: low
  - Memory/IO state: high
- Process creation: high
- Protection
  - CPU: yes
  - Memory/IO: yes
- Sharing overhead: high (involves at least a context switch)

Putting it together: Threads



- Switch overhead: low (only CPU state)
- Thread creation: low
- Protection
  - CPU: yes
  - Memory/IO: No
- Sharing overhead: low (thread switch overhead low)

Putting it together: Multi-Cores

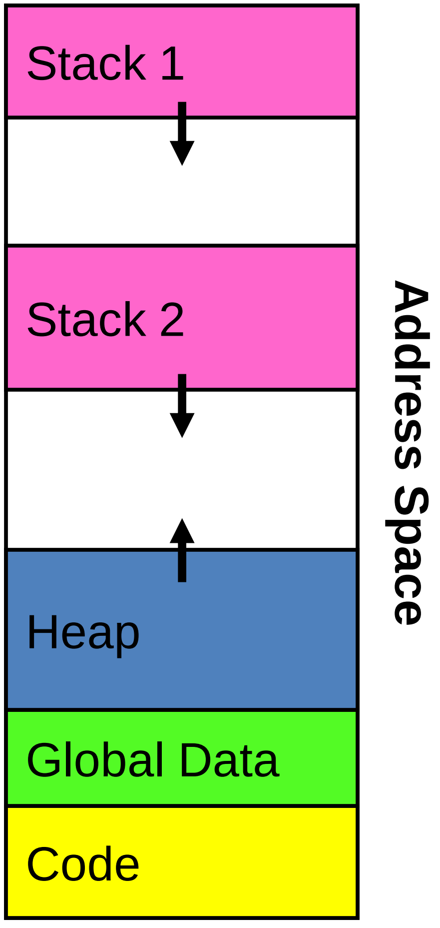


- Switch overhead: low (only CPU state)
- Thread creation: low
- Protection
  - CPU: yes
  - Memory/IO: No
- Sharing overhead: low (thread switch overhead low)

Memory Footprint of Two-Thread Example

- If we stopped this program and examined it with a debugger, we would see
  – Two sets of CPU registers
  – Two sets of Stacks

- Questions:
  – How do we position stacks relative to each other?
  – What maximum size should we choose for the stacks?
  – What happens if threads violate this?
  – How might you catch violations?



Concurrency

The ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome.

This allows for parallel execution of the concurrent units, which can significantly improve overall speed of the execution in multi-processor and multi-core systems.

Concurrency

≠

Parallelism

Concurrency is often referred to as the decomposability property of a program, algorithm, or problem into order-independent or partially-ordered components or units.

Source: Lamport, Leslie (July 1978). "Time, Clocks, and the Ordering of Events in a Distributed System"

Concurrent execution of threads

On a single core CPU

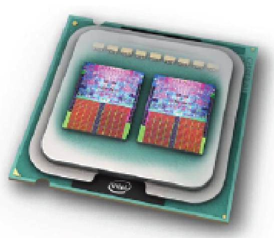
single core	T_1	T_2	T_3	T_4	T_1	T_2	T_3	T_4	T_1	...

time →

Threads take turn executing on the single CPU core. By switching fast enough between the threads they appear to be executing ”at the same time”.

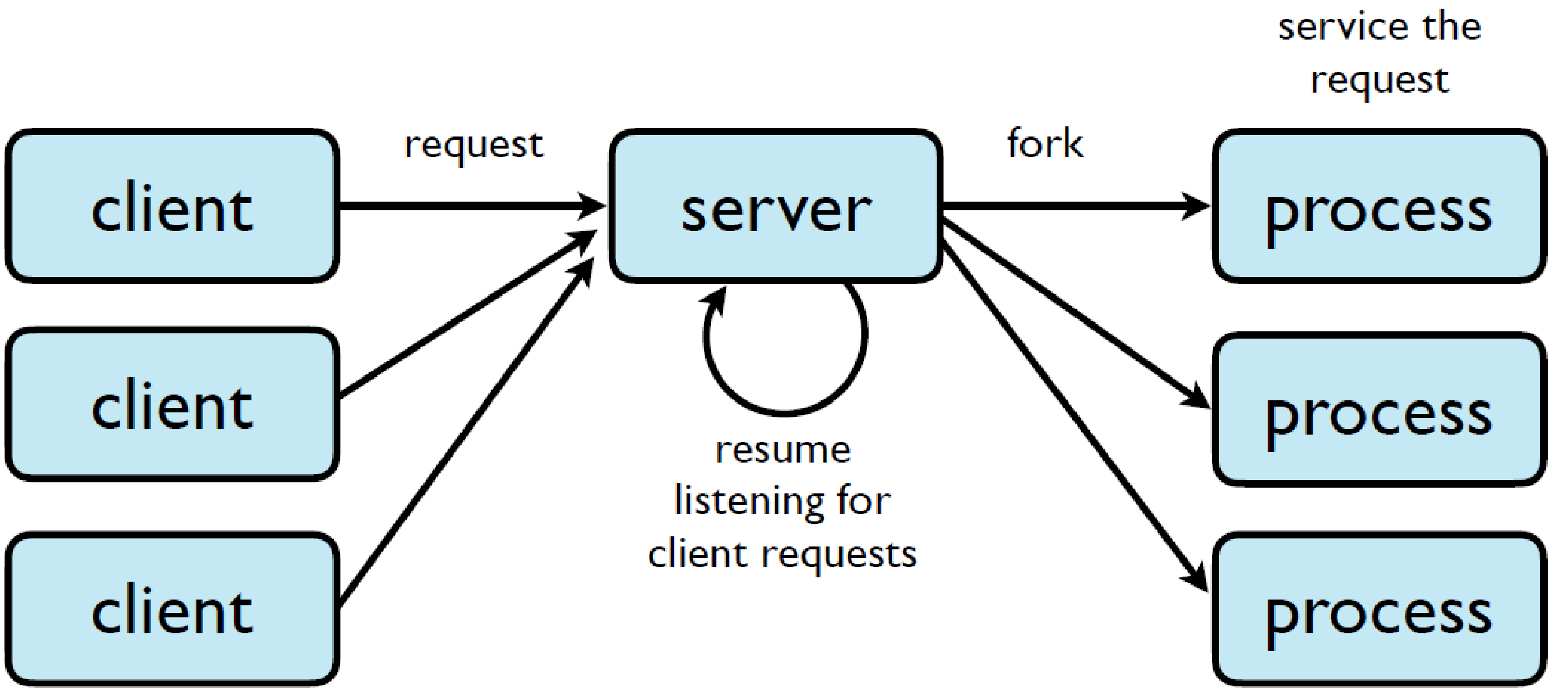
$\boxed{\text{concurrent} \neq \text{parallel}}$

On a dual core CPU

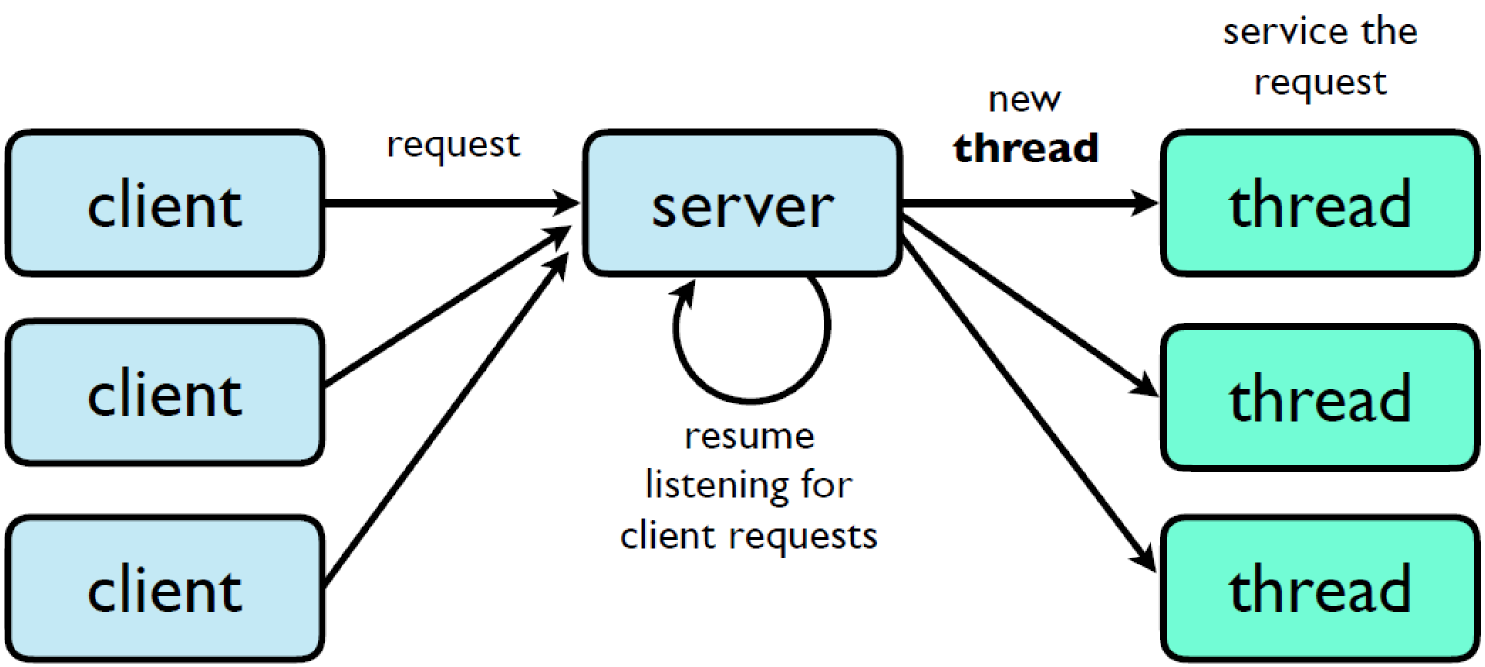


core 1	T_1	T_3	T_1	T_3	T_1	...
core 2	T_2	T_4	T_2	T_4	T_2	...

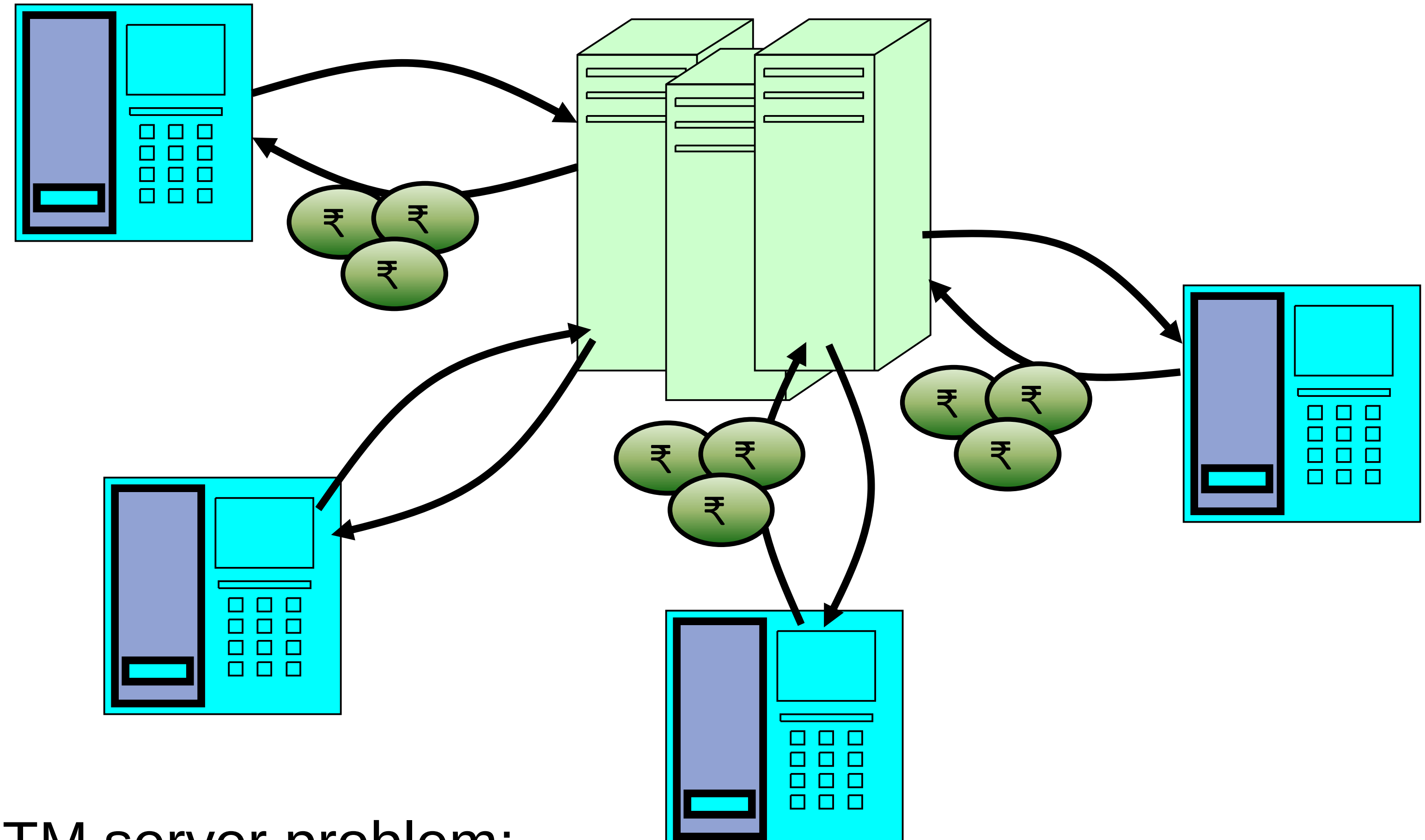
Concurrency and response time



Creating a new process is time consuming and resource intensive.

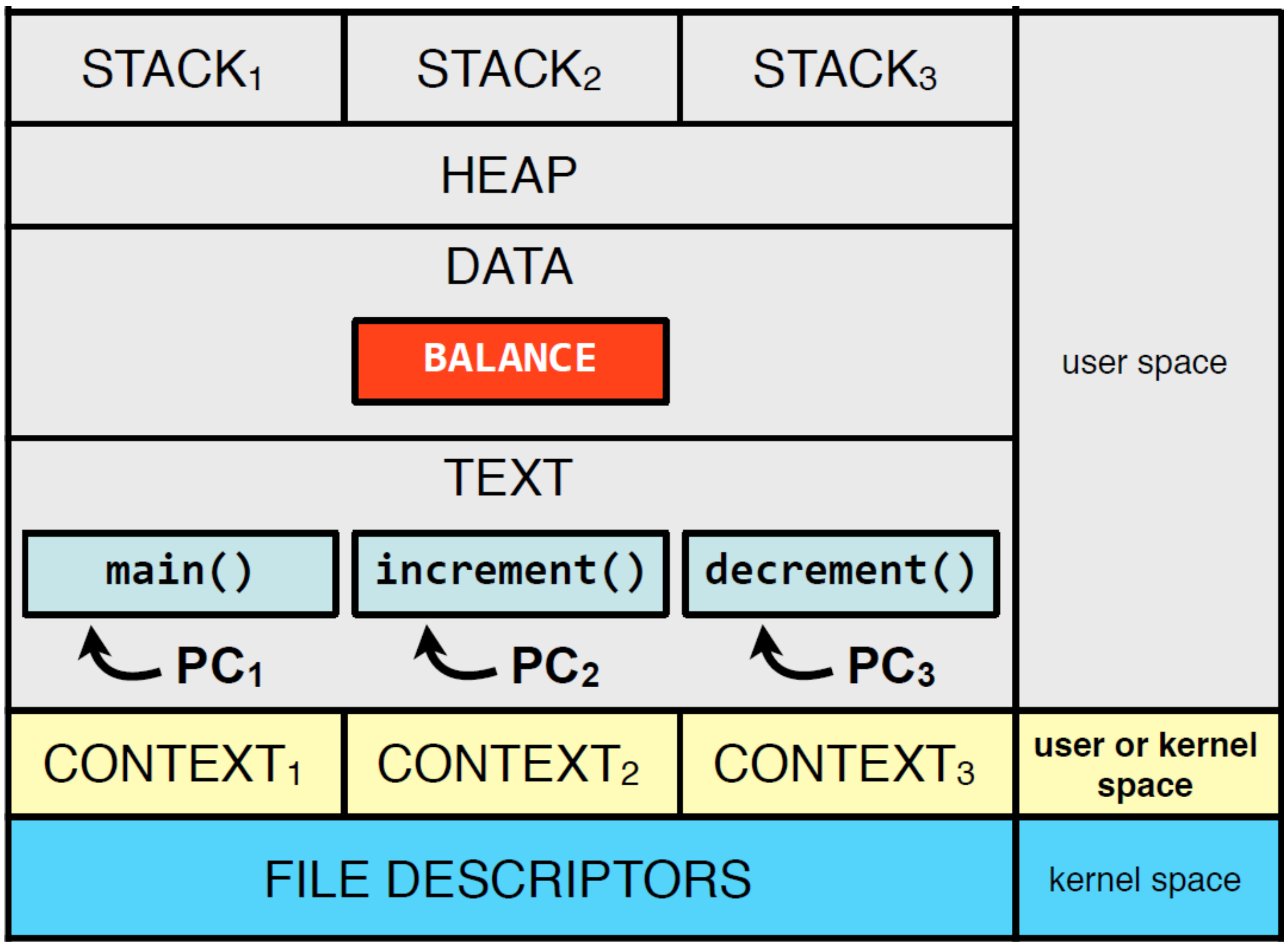

client
client
client
request
server
resume
listening for
client requests
new
thread
service the
request
thread
thread
thread

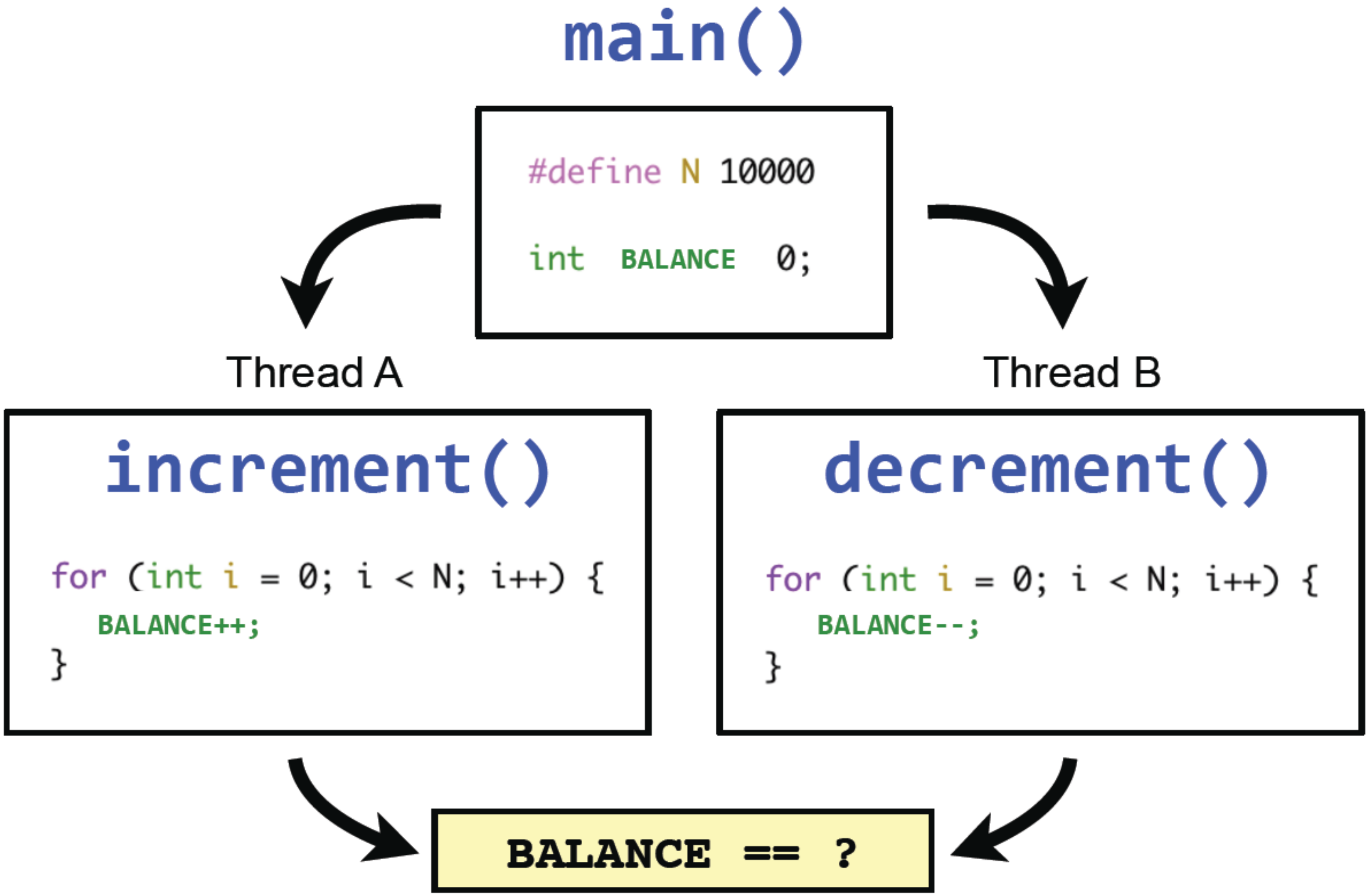
ATM Bank Server



- ATM server problem:
  - Service a set of requests
  - Do so without corrupting database
  - Don't hand out too much money

A process with three threads




main()
#define N 10000
int BALANCE 0;
Thread A
Thread B
increment()
for (int i = 0; i < N; i++) {
BALANCE++;
}
decrement()
for (int i = 0; i < N; i++) {
BALANCE--;
}
BALANCE == ?

Correctness with Concurrent Threads

- Non-determinism:
  - Scheduler can run threads in **any order**
  - Scheduler can switch threads **at any time**
  - This can make testing very difficult
- *Independent Threads*
  - No state shared with other threads
  - Deterministic, reproducible conditions
- *Cooperating Threads*
  - Shared state between multiple threads
- **Goal: Correctness by Design**

Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, What about (Initially, y = 12):

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  – What are the possible values of x?

Thread A	Thread B
x = 1;	
x = y+1;	
	y = 2;
	y = y*2

x=13

Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, What about (Initially, y = 12):

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  – What are the possible values of x?

Thread A	Thread B
	y = 2;
	y = y*2;
x = 1;	
x = y+1;	

x=5

Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, What about (Initially, y = 12):

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  – What are the possible values of x?

Thread A	Thread B
	y = 2;
x = 1;	
x = y+1;	
	y= y*2;

x=3

Correctness Requirements

- Threaded programs must work for all interleavings of thread instruction sequences
  - Cooperating threads inherently non-deterministic and non-reproducible
  - Really hard to debug unless carefully designed!
- Example: Therac-25
  - Machine for radiation therapy
    - Software control of electron accelerator and electron beam/ Xray production
    - Software control of dosage
  - Software errors caused overdoses and the death of several patients
    - A series of race conditions on shared variables and poor software design
    - “They determined that data entry speed during editing was the key factor in producing the error condition: If the prescription data was edited at a fast pace, the overdose occurred.”

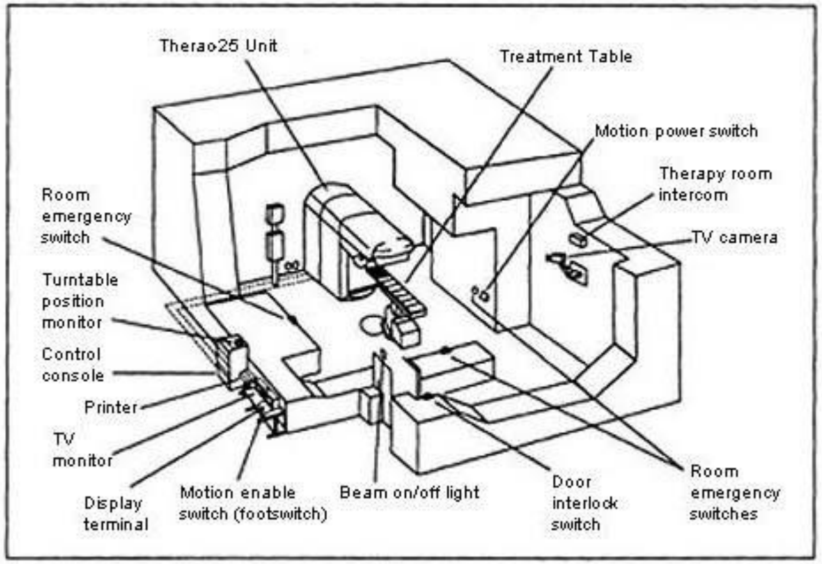
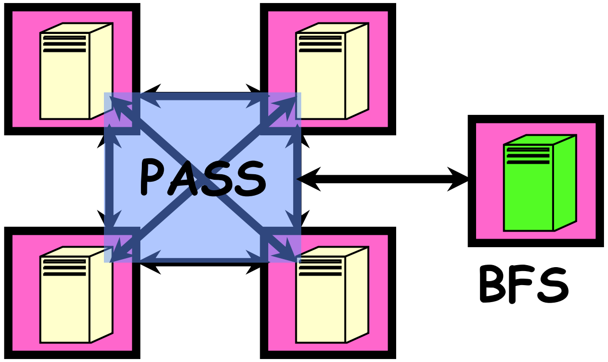


Figure 1. Typical Therac-25 facility

http://brinch-hansen.net/

Space Shuttle Example

- Original Space Shuttle launch aborted 20 minutes before scheduled launch
- Shuttle has five computers:
  - Four run the "Primary Avionics Software System" (PASS)
    - Asynchronous and real-time
    - Runs all of the control systems
    - Results synchronized and compared 440 times per second
  - The Fifth computer is the "Backup Flight System" (BFS)
    - Stays synchronized in case it is needed
    - Written by completely different team than PASS



- Countdown aborted because BFS disagreed with PASS
  - A 1/67 chance that PASS was out of sync one cycle
  - Bug due to modifications in initialization code of PASS
    - A delayed init request placed into timer queue
    - As a result, timer queue not empty at expected time to force use of hardware clock
  - Bug not found during extensive simulation

Atomic Operations

- To understand a concurrent program, we need to know what the underlying atomic operations are!

- Atomic Operation: an operation that always runs to completion or not at all
  - It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for threads to work together

- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic

- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy a whole array

Atomic Operations

- uint64_t sharedValue = 0;
- void storeValue()
```
{
 sharedValue = 0x100000002;
}
```

- $ gcc -O2 -S -masm=intel test.c
- $ cat test.s ...
```
      mov DWORD PTR sharedValue, 2
      mov DWORD PTR sharedValue+4, 1
       ret
```

Concurrency Challenges

- Multiple computations (threads) executing in parallel to
  - share resources, and/or
  - share data
- Fine grain sharing:
  - ⬆ increase concurrency → better performance
  - ⇓ more complex
- Coarse grain sharing:
  - ⇑ Simpler to implement
  - ⇓ Lower performance
- Examples:
  - Sharing CPU for 10ms vs. 1min
  - Sharing a database at the row vs. table granularity

Concurrency poses challenges for:

- Correctness
  - Threads accessing shared memory should not interfere with each other
- Liveness
  - Threads should not get stuck, should make forward progress
- Efficiency
  - Program should make good use of available computing resources (e.g., processors).
- Fairness
  - Resources apportioned fairly between threads

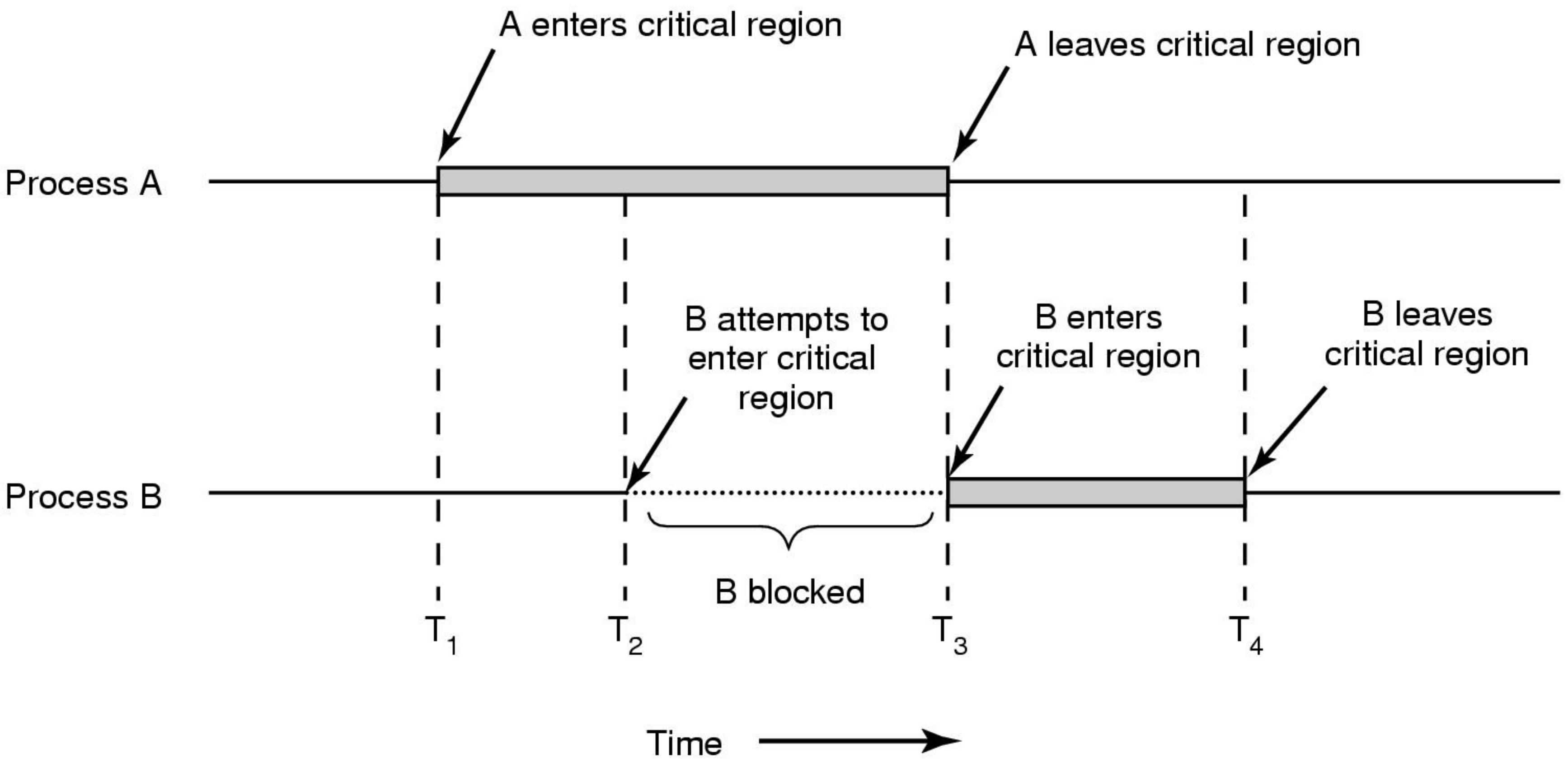
Definitions

- Synchronization: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic

- Critical Section: piece of code that only one thread can execute at once

- Mutual Exclusion: ensuring that only one thread executes critical section
  - One thread *excludes* the other while doing its task
  - Critical section and mutual exclusion are two ways of describing the same thing

Critical Section Problem



Mutual exclusion using critical regions

Race Conditions

- What are the possible values of **`x`** below?
- Initially **`x = y = 0;`**

<u>Thread A</u>	<u>Thread B</u>
`x = 1;`	**`y = 2;`**

- Must be **1**. Thread B cannot interfere.

Race Conditions

- What are the possible values of **`x`** below?
- Initially **`x = y = 0;`**

Thread A

```
x = y + 1;
```

Thread B

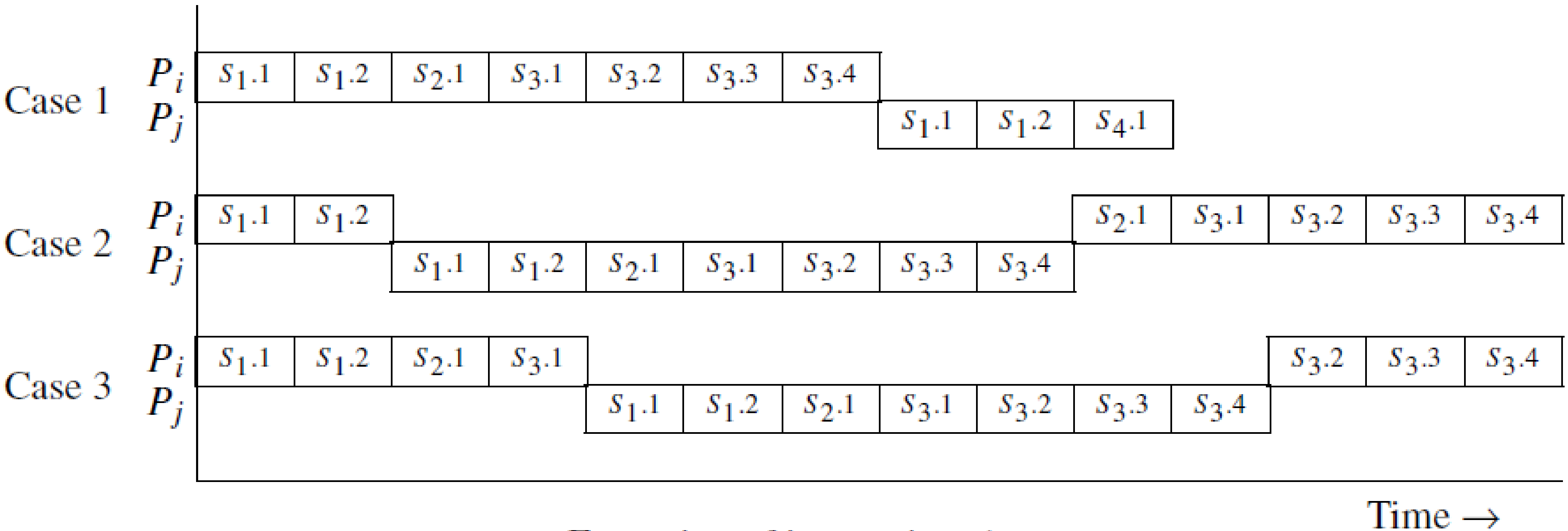
```
y = 2;
y = y * 2;
```

- 1 or 3 or 5 (non-deterministic)
- Race Condition: Thread A races against Thread B

Race Condition

	Code of processes		Corresponding machine instructions
S_1	**if** *nextseatno* $\leq$ *capacity*	$S_1.1$	Load *nextseatno* in reg_k
		$S_1.2$	If $reg_k > capacity$ goto $S_4.1$
	then		
S_2	*allotedno:=nextseatno;*	$S_2.1$	Move *nextseatno* to *allotedno*
S_3	*nextseatno:=nextseatno+1;*	$S_3.1$	Load *nextseatno* in reg_j
		$S_3.2$	Add 1 to reg_j
		$S_3.3$	Store reg_j in *nextseatno*
		$S_3.4$	Go to $S_5.1$
	else		
S_4	*display "sorry, no seats available"*	$S_4.1$	Display "*sorry,* · · · "
S_5	. . .	$S_5.1$	. . .

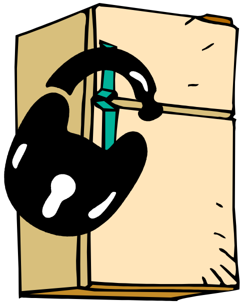
Some execution cases



Execution of instructions by processes

More Definitions

- Lock: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - Important idea: all synchronization involves waiting
- Example: fix the milk problem by putting a lock on refrigerator
  - Lock it and take key if you are going to go buy milk



Motivation: “Too much milk”

- Example: People need to coordinate:

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Too Much Milk: Correctness

1. At most one person buys milk (not more than 1)

2. At least one person buys milk if needed

Solution Attempt #1

- Leave a note
  - Place on fridge before buying
  - Remove after buying
  - Don’t go to store if there’s already a note
- Leaving/checking a note is atomic (word load/store)

```
if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
    remove Note;
  }
}
```

Attempt #1in Action

A

```
if (noMilk) {
  if (noNote) {


    leave Note;
    buy milk;
    remove Note;
  }
}
```

B

```
if (noMilk) {
  if (noNote) {





    leave Note;
    buy milk;
    remove note;
  }
}
```

Solution Attempt #2

```
leave Note;
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
   }
}
remove Note;
```

But there's always a note – you just left one!

At least you don't buy milk twice…

Solution Attempt #3

- Leave a named note – each person ignores their own

A

```
leave note A
if (noMilk) {
  if (noNote B) {
    buy milk
  }
}
remove note A;
```

B

```
leave note B
if (noMilk) {
  if (noNote A) {
    buy milk
  }
}
remove note B;
```

Attempt #3 in Action

A

```
leave note A
if (noMilk) {

  if (noNote B) {
    buy milk
  }
}







remove note A
```

B

```



leave note B





if (noMilk) {
  if (noNote A) {
    buy milk
  }
remove note B
```

Solution Attempt #4

A

```
leave note A
while (note B) {
  do nothing
}
if (noMilk) {
  buy milk
}
remove note A;
```

B

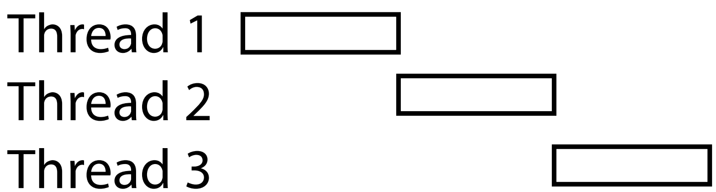
```
leave note B
if (noNote A) {
  if (noMilk) {
    buy milk
  }
}
remove note B;
```

- This is a correct solution, but …

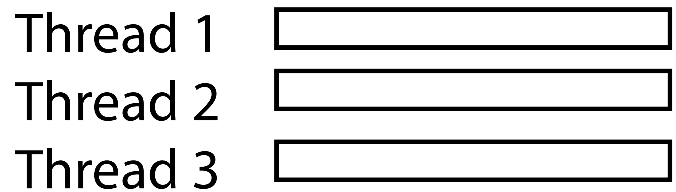
Issues with Solution 4

- Complexity
  - Proving that it works is hard
  - How do you add another thread?
- Busy-waiting
  - A **consumes CPU time to wait**
- Fairness
  - Who is more likely to buy milk?

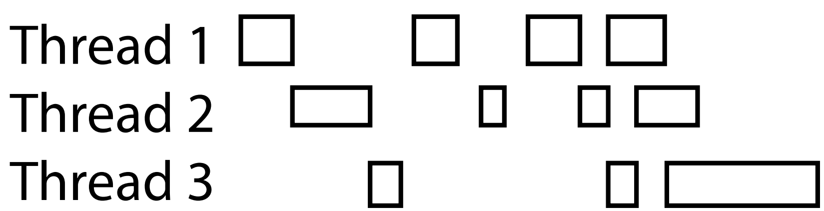
All Possible Executions” ?



a) One execution



b) Another execution



c) Another execution

Relevant Definitions

- **Lock:** An object only one thread can hold at a time
  - **Provides** mutual exclusion
- Offers two **atomic** operations:
  - Lock.Acquire() – wait until lock is free; then grab
  - Lock.Release() – Unlock, wake up waiters

Using Locks

```
MilkLock.Acquire()
if (noMilk) {
  buy milk
}
MilkLock.Release()
```

But how do we implement this?

First, how do we use it?

Relevant Definitions

- **Lock:** An object only one thread can hold at a time
  - **Provides** mutual exclusion

- Offers two **atomic** operations:
  - Lock.Acquire() – wait until lock is free; then grab
  - Lock.Release() – Unlock, wake up waiters

Mutexes

- Critical sections typically associated with mutual exclusion locks (*mutexes*)
- Only one thread can hold a given mutex at a time
- Acquire (lock) mutex on entry to critical section
  - Or block if another thread already holds it
- Release (unlock) mutex on exit
  - Allow one waiting thread (if any) to acquire & proceed

```
pthread_mutex_init(&m);
```

```
pthread_mutex_lock(&m);
    hits = hits+1;
pthread_mutex_unlock(&m);
```

T1

```
pthread_mutex_lock(&m);
    hits = hits+1;
pthread_mutex_unlock(&m);
```

T2

Using Locks

```
MilkLock.Acquire()
if (noMilk) {
  buy milk
}
MilkLock.Release()
```

But how do we implement this?

First, how do we use it?

Implementing Too Much Milk with Locks

Thread A

```
Lock.Acquire();
if (noMilk){
    buy milk;
}
Lock.Release();
```

Thread B

```
Lock.Acquire();
if (noMilk){
    buy milk;
}
Lock.Release();
```

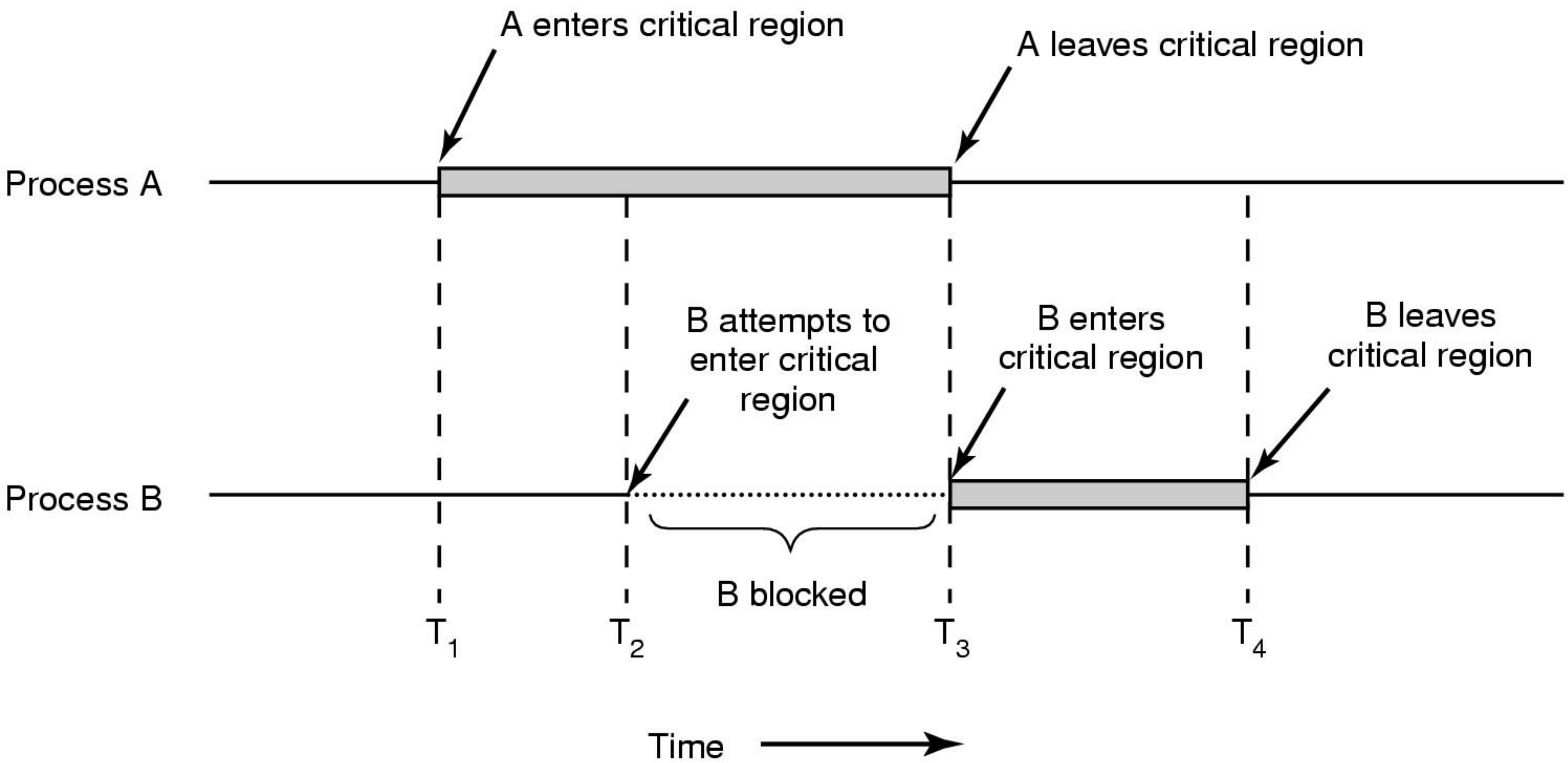
- This solution is clean and symmetric.
- How do we make Lock.Acquire and Lock.Release atomic?

Support for synchronization?

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Comp&Swap

- We are going to implement various higher-level synchronization primitives using atomic operations
  – Everything is pretty painful if only atomic primitives are load and store
  – Need to provide primitives useful at user-level

Re-call Critical Section Problem



Mutual exclusion using critical regions

“Solution 1 ”:

Disable interrupts while holding lock

Implementing Locks: Single Core

- Idea: A context switch can only happen (assuming threads don't yield) if there's an **interrupt**

- "Solution": **Disable interrupts** while holding lock

- x86 has `cli` and `sti` instructions that only operate in system mode (PL=0)
  - Interrupts enabled bit in FLAGS register

Interrupt Enable/Disable

```
Acquire() {
  disable interrupts;
}
```

```
Release() {
  enable interrupts;
}
```

- Problem: can stall the entire system

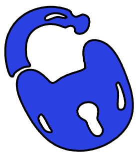
```
Lock.Acquire()
While (1) {}
```

- Problem: What if we want to do I/O?

```
Lock.Acquire()
Read from disk
/* OS waits for (disabled) interrupt)!
*/
```

Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable



```
int value = FREE;

Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```

Atomic Read-Modify-Write instructions

- Problems with interrupt-based lock solution:
  - Can’t give lock implementation to users
  - Doesn’t work well on multiprocessor
    - Disabling interrupts on all processors requires messages and would be very time consuming
- Alternative: atomic instruction sequences
  - These instructions read a value from memory and write a new value atomically
  - Hardware is responsible for implementing this correctly
    - on both uniprocessors (not too hard)
    - and multiprocessors (requires help from cache coherence protocol)
  - Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors

An atomic test_and_set instruction

- Assume we add the following instruction
- **Test_And_Set Rx memory-location**
- Perform the following atomically:
  - Read the current value of memory-location into some processor register (Rx)
  - Set the memory-location to a 1

Test_and_Set

```
boolean Test_and_Set( boolean memory[m] )
{ [
   if( memory[m] )        // lock denied
      return True;
   else {                 // lock granted
      memory[m] = True;
      return False;
   }
 ]
}
```

An atomic test_and_set instruction

If the mutex is unlocked (=0)

- test_and_set will return 0 which will mean you have the mutex lock
- It will also set the mutex variable to 1

if the mutex is locked (=1)

- test_and_set will return 1 which will mean you don't have the mutex lock
- It will also set the mutex variable to 1 (which is what it was anyway)

(mutual exclusion- “**mutex**”)

Examples of Read-Modify-Write

- ```
  test&set (&address) {       /* most architectures */
      result = M[address];
      M[address] = 1;
      return result;
  }
  ```

- ```
  swap (&address, register) { /* x86 */
      temp = M[address];
      M[address] = register;
      register = temp;
  }
  ```

- ```
  compare&swap (&address, reg1, reg2) { /* 68000 */
      if (reg1 == M[address]) {
          M[address] = reg2;
          return success;
      } else {
          return failure;
      }
  }
  ```
  ```

Implementing Locks with test&set

```
test&set (&address) {
  result = M[address];
  M[address] = 1;
  return result;
}
```

- Simple solution:

```
int value = 0; // Free
Acquire() {
   while (test&set(value)); // while busy
}
Release() {
   value = 0;
}
```

- Simple explanation:
  - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits
  - If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues
  - When we set value = 0, someone else can get lock

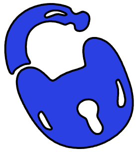
Problem: Busy-Waiting for Lock

- Positives for this solution
  - Machine can receive interrupts
  - User code can use this lock
  - Works on a multiprocessor
- Negatives
  - Inefficient: busy-waiting thread will consume cycles waiting
  - Waiting thread may take cycles away from thread holding lock!
  - Priority Inversion: If busy-waiting thread has higher priority than thread holding lock $\Rightarrow$ no progress!
- Priority Inversion problem with original Mars Rover

Locks using test&set vs. Interrupts

- Compare to "disable interrupt" solution

```
int value = FREE;
```



```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

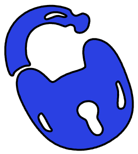
```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```

- Basically replace
  - `disable interrupts` → `while (test&set(guard));`
  - `enable interrupts` → `guard = 0;`

Better Locks using test&set

- Can we build test&set locks without busy-waiting?
  – Can't entirely, but can minimize!
  – Idea: only busy-wait to atomically check lock value

```
int guard = 0;
int value = FREE;
```



```
Acquire() {
   // Short busy-wait time
   while (test&set(guard));
   if (value == BUSY) {
      put thread on wait queue;
      go to sleep() & guard = 0;
   } else {
      value = BUSY;
      guard = 0;
   }
}
```

```
Release() {
   // Short busy-wait time
   while (test&set(guard));
   if anyone on wait queue {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   guard = 0;
```

- Note: sleep has to be sure to reset the guard variable
  – Why can't we do it just before or just after the sleep?

Recap: Locks

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Acquire() {
  disable interrupts;
}

Release() {
  enable interrupts;
}
```

If one thread in critical section, no other activity (including OS) can run!

```
int value = 0;
Acquire() {
  // Short busy-wait time
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}

Release() {
  // Short busy-wait time
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Recap: Locks

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
int value = 0;
Acquire() {
  while(test&set(value));
}
```

```
Release() {
  value = 0;
}
```

Threads waiting to enter critical section busy-wait

```
int guard = 0;
int value = 0;
Acquire() {
  // Short busy-wait time
  while(test&set(guard));
  if (value == 1) {
    put thread on wait-queue;
    go to sleep()& guard = 0;
  } else {
    value = 1;
    guard = 0;
  }
}
```

```
Release() {
  // Short busy-wait time
  while (test&set(guard));
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  guard = 0;
}
```

Support for synchronization?

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Comp&Swap

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

Peterson’s solution

Peterson's algorithm (aka Peterson's solution) is a concurrent programming algorithm for **mutual exclusion** that allows **two tasks** to **share** a single use **resource** without conflict, **using only shared memory** for communication. It was formulated by Gary L. Peterson in 1981.

Faulty Algorithm 1 - Turn taking

- The shared variable turn is initialized (to 0 or 1) before executing any Pi
- Pi's critical section is executed iff turn = i
- Pi is busy waiting if Pj is in CS

```
Process Pi:
            //i,j= 0 or 1
repeat
   while(turn!=i){};
      CS
   turn:=j;
      RS
forever
```

```
Process P0:
repeat

while(turn!=0){};
      CS
   turn:=1;
      RS
forever
```

```
Process P1:
repeat

while(turn!=1){};
      CS
   turn:=0;
      RS
forever
```

Faulty Algorithm 1 side-by-side view

Analysis

- Achieves Mutual Exclusion (busy wait)

- But Progress requirement is not satisfied since it requires strict alternation of CS's.
  - If one process requires its CS more often than the other, it can't get it.

Faulty Algorithm 2 – Ready flag

- Keep a Boolean variable for each process: flag[0] and flag[1]
- Pi signals that it is ready to enter its CS by: flag[i]:=true but waits until the other has finished its CS.

```
Process Pi:
repeat
  flag[i]:=true;
  while(flag[j]){};
     CS
  flag[i]:=false;
     RS
forever
```

```
Process P0:
repeat
  flag[0]:=true;

while(flag[1]){};
     CS
  flag[0]:=false;
     RS
forever
```

```
Process P1:
repeat
  flag[1]:=true;

while(flag[0]){};
     CS
  flag[1]:=false;
     RS
forever
```

Faulty Algorithm 2 side-by-side view

Analysis

- Mutual Exclusion is satisfied but not the progress requirement
- For the (interleaved) sequence:
  - flag[0]:=true
  - flag[1]:=true
- Both processes will wait forever

Algorithm 3
(Peterson's Algorithm)

- Initialization:
  flag[0]:=flag[1]:=false
  turn:= 0 or 1
- Wish to enter CS specified by flag[i]:=true
- Even if both flags go up, and no matter how the instructions are interleaved,
  - ..turn will always end up as either 0 or 1

```
Process Pi:
repeat
   flag[i]:=true;
     // I want in
   turn:=j;
    // but you can go first!
   while(flag[j]&& turn==j);
      CS
   flag[i]:=false;
    // I'm done
      RS
forever
```

Peterson's Algorithm

```
Process P0:
repeat
   flag[0]:=true;
     // 0 wants in
   turn:= 1;
    // 0 gives a chance to 1
   while
    (flag[1]&turn=1);
      CS
   flag[0]:=false;
    // 0 is done
      RS
forever
```

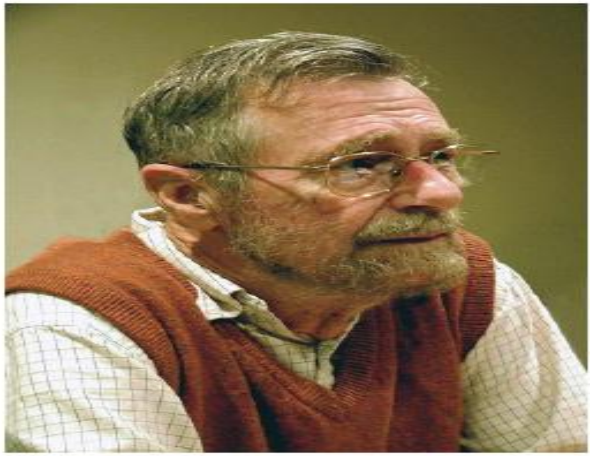
```
Process P1:
repeat
   flag[1]:=true;
     // 1 wants in
   turn:=0;
    // 1 gives a chance to 0
   while
    (flag[0]&turn=0);
      CS
   flag[1]:=false;
    // 1 is done
      RS
forever
```

Peterson's algorithm side-by-side view

Peterson's Algorithm: Proof of Correctness

- Mutual exclusion holds since:
  - For both P_0 and P_1 to be in their CS
    - both flag[0] and flag[1] must be true and:
    - turn=0 and turn=1 (at same time): impossible

Semaphores



Edsger Dijkstra
1930 - 2002

- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX (& Pintos)
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P() or down(): atomic operation that waits for semaphore to become positive, then decrements it by 1
  - V() or up(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any

P() stands for *"proberen"* (to test) and V() stands for *"verhogen"* (to increment) in Dutch

Semaphores: Key Concepts

- Like locks, a semaphore supports two atomic operations, Semaphore.Wait() and Semaphore.Signal().

```
S.Wait()        // wait until semaphore S
                 // is available
<critical section>

S.Signal()      // signal to other processes
                 // that semaphore S is free
```

- Each semaphore supports a queue of processes that are waiting to access the critical section (e.g., to buy milk).
- If a process executes **S.Wait()** and semaphore S is free (non-zero), it continues executing. If semaphore S is not free, the OS puts the process on the wait queue for semaphore S.
- A **S.Signal()** unblocks one process on semaphore S's wait queue.

Semaphore

- Does not require busy-waiting
  - CPU is not held unnecessarily while the process is waiting
- A Semaphore *S* is
  - A data structure with an integer variable *S.value* and a queue *S.list* of processes (shared variable)
  - The data structure can only be accessed by two atomic operations, *wait(S)* and *signal(S)* (also called *down(S), P(S)* and *Up(s), V(S)*)
- Value of the semaphore S = value of the integer *S.value*

```
typedef struct {
        int value;
        struct process *list;
        } semaphore
```

Semaphore

Wait(S) S<= semaphore variable

- When a process P executes the wait(S) and finds
- S==0
  - Process must wait => block()
  - Places the process into a waiting queue associated with S
  - Switch from running to waiting state

Signal(S)

When a process P executes the signal(S)

- Check, if some other process Q is waiting on the semaphore S
- Wakeup(Q)
- Wakeup(Q) changes the process from waiting to ready state

Semaphore (wait and signal)

- Implementation of wait:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

List of PCB

- Implementation of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Atomic/
Indivisible

Note: which process is picked for unblocking may depend on policy.

Binary Semaphores: Example

- Too Much Milk using locks:

Thread A

```
Lock.Acquire();
if (noMilk){
  buy milk;
}
Lock.Release();
```

Thread B

```
Lock.Acquire();
if (noMilk){
      buy milk;
}
Lock.Release();
```

- Too Much Milk using semaphores:

Thread A

```
Semaphore.Wait();
if (noMilk){
  buy milk;
}
Semaphore.Signal();
```

Thread B

```
Semaphore.Wait();
if (noMilk){
      buy milk;
}
Semaphore.Signal();
```

Semaphore -> P() (*Passeren*; wait)

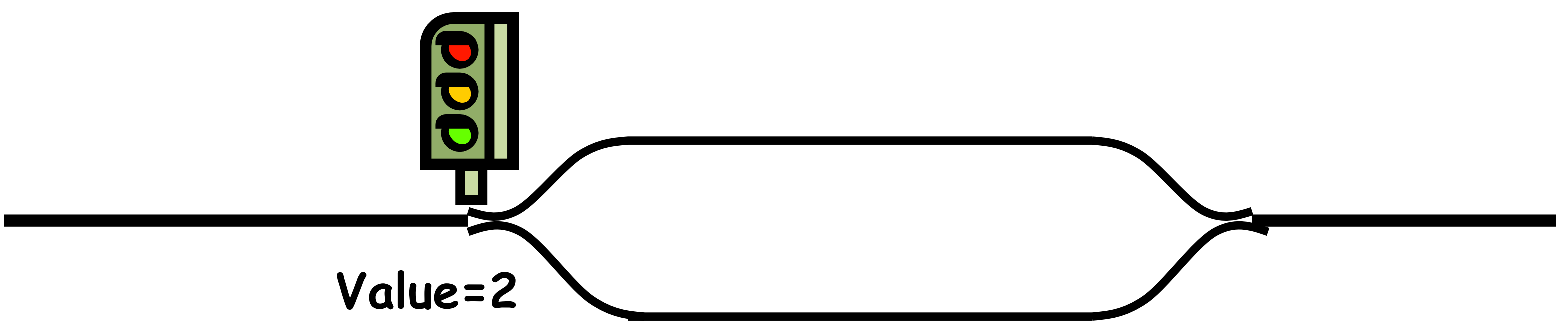
- If sem > 0, then decrement sem by 1
- Otherwise "wait" until sem > 0 and then decrement

Semaphore -> V() (*Vrijgeven*; signal)

- Increment sem by 1
- Wake up a thread waiting in P()

Semaphores Like Integers Except

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V – can't read or write value, except to set it initially
  - Operations must be atomic
    - Two P's together can't decrement value below zero
    - Similarly, thread going to sleep in P won't miss wakeup from V – even if they both happen at same time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



Signal and Wait: Example

```
P1:  S.Wait();
     S.Wait();
     S.Signal();
     S.Signal();

P2:  S.Wait();
     S.Signal();
```

```
P1:   S->Wait();
P2:   S->Wait();
P1:   S->Wait();
P2:   S->Signal();
P1:   S->Signal();
P1:   S->Signal();
```

value	Queue	process state: execute or block	
		P1	P2
2	empty	execute	execute
1	empty	execute	execute
0	empty	execute	execute
-1	P1	blocked	execute
0	empty	execute	execute
1	empty	execute	execute
2	empty	execute	execute

Two Uses of Semaphores

- Mutual Exclusion (initial value = 1)
  - Also called “Binary Semaphore”.
  - Can be used for mutual exclusion:

```
semaphore.P();
// Critical section goes here
semaphore.V();
```

- Scheduling Constraints (initial value = 0)
  - Allow thread 1 to wait for a signal from thread 2, i.e., thread 2 schedules thread 1 when a given constrained is satisfied
  - Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:

```
Initial value of semaphore = 0
ThreadJoin {
  semaphore.P();
}
ThreadFinish {
  semaphore.V();
}
```

Deadlocks

Deadlock: A condition where two or more threads are waiting for an event that can only be generated by these same threads.

Example:

Process A:

```
printer.Wait();
disk.Wait();

// copy from disk
// to printer

printer.Signal();
disk.Signal();
```

Process B:

```
disk.Wait();
printer.Wait();

// copy from disk
// to printer

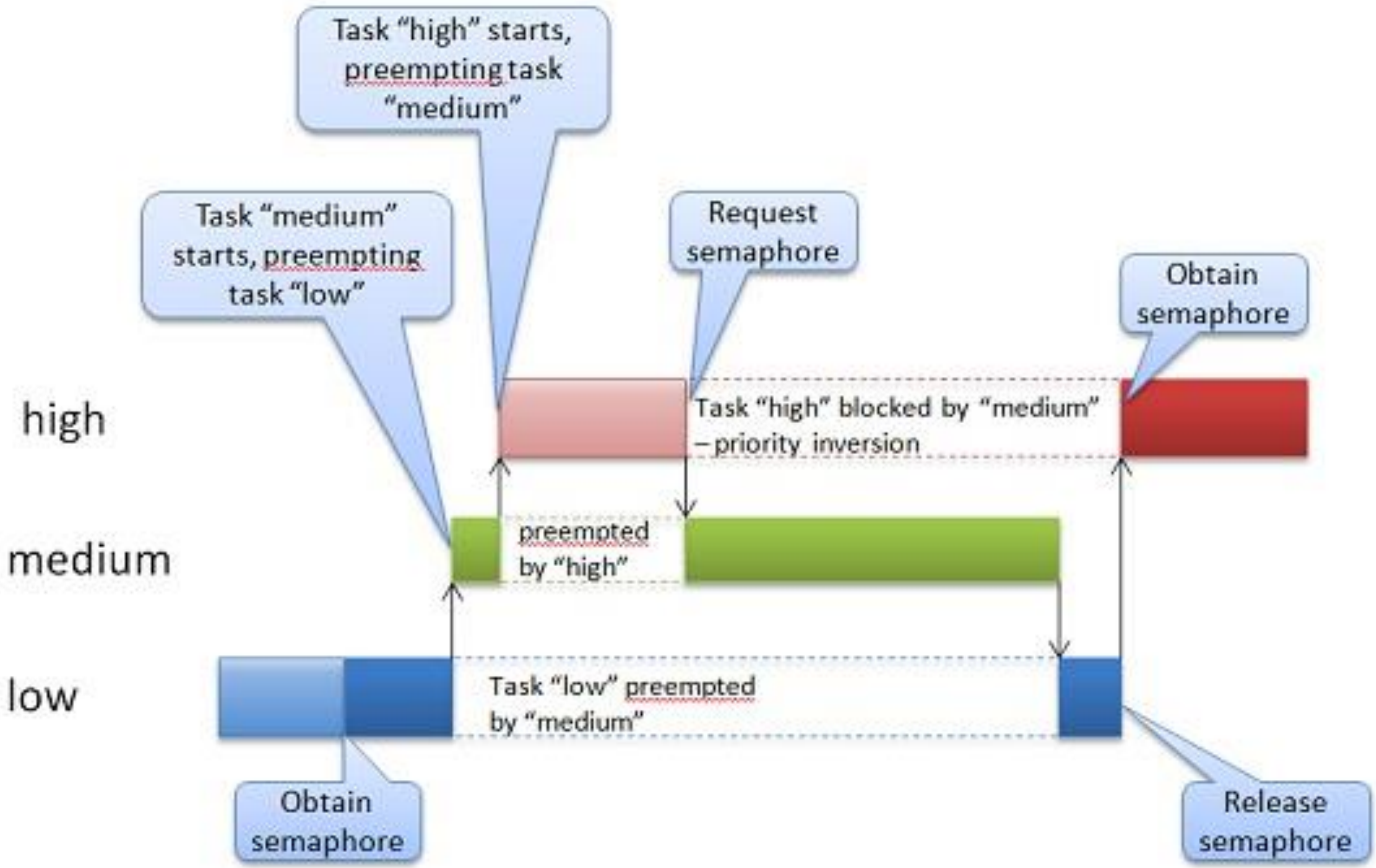
printer.Signal();
disk.Signal();
```

Deadlock and Starvation

- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- **Starvation** – indefinite blocking
  - LIFO queue
  - A process may never be removed from the semaphore queue in which it is suspended



https://www.rapitasystems.com/blog/what-really-happened-to-the-software-on-the-mars-pathfinder-spacecraft

Ordering Execution of Processes using Semaphores

- Execute statement *B* in P_j only after statement *A* executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
⋮	⋮
Stmt. *A*	*wait*(*flag*)
signal(*flag*)	Stmt. *B*

- Multiple such points of synchronization can be enforced using one or more semaphores

Synchronization Problems- Real-World Examples

- **Producer-consumer**

– Audio/Video player: network and display threads; shared buffer

– Web servers: master thread and slave thread

- Reader-writer

– Banking system: read account balances versus update

- Dining Philosophers

– Cooperating processes that need to share limited resources

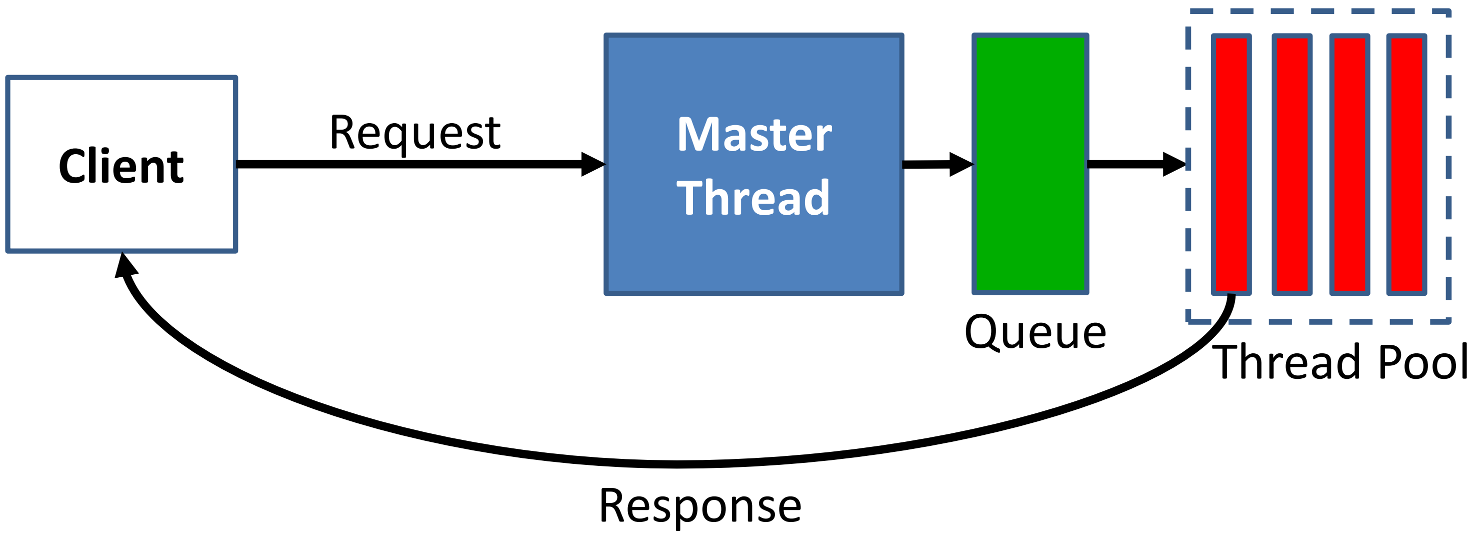
- Set of processes that need to lock multiple resources

– Disk and tape (backup),

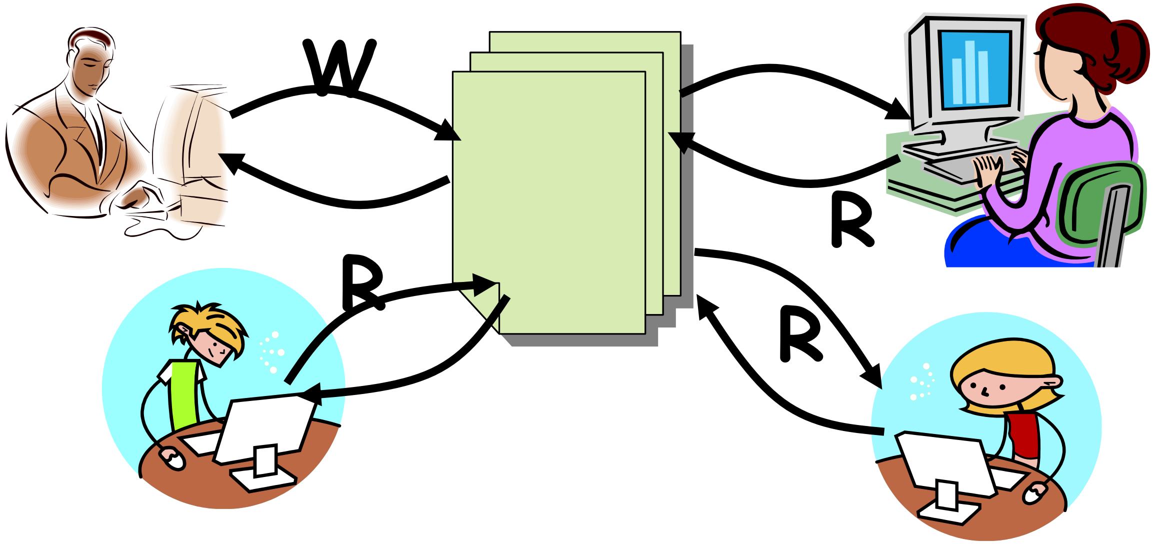
- Travel reservation: hotel, airline, car rental databases

Web Server: Thread Pools

- **Bounded** pool of worker threads
  - Allocated in **advance:** no thread creation overhead
  - **Queue** of pending requests
  - **Limited number** of requests in progress



Readers/Writers Problem



- Motivation: Consider a shared database
  - Two classes of users:
    - Readers – never modify database
    - Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    - Like to have many readers at the same time
    - Only one writer at a time

Basic Readers/Writers Solution

- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers or writers
  - Only one thread manipulates state variables at a time
- Basic structure of a solution:
  - **`Reader()`**
    ```
    Wait until no writers
    Access database
    Check out – wake up a waiting writer
    ```
  - **`Writer()`**
    ```
    Wait until no active readers or writers
    Access database
    Check out – wake up waiting readers or writer
    ```
  - State variables (Protected by a lock called “lock”):
    - int AR: Number of active readers; initially = 0
    - int WR: Number of waiting readers; initially = 0
    - int AW: Number of active writers; initially = 0
    - int WW: Number of waiting writers; initially = 0
    - Condition okToRead = NIL
    - Condition okToWrite = NIL

Solution to Bounded Buffer (coke machine)

```
Semaphore fullSlots = 0;     // Initially, no coke
 Semaphore emptySlots = bufSize;
                             // Initially, num empty slots
 Semaphore mutex = 1;        // No one using machine

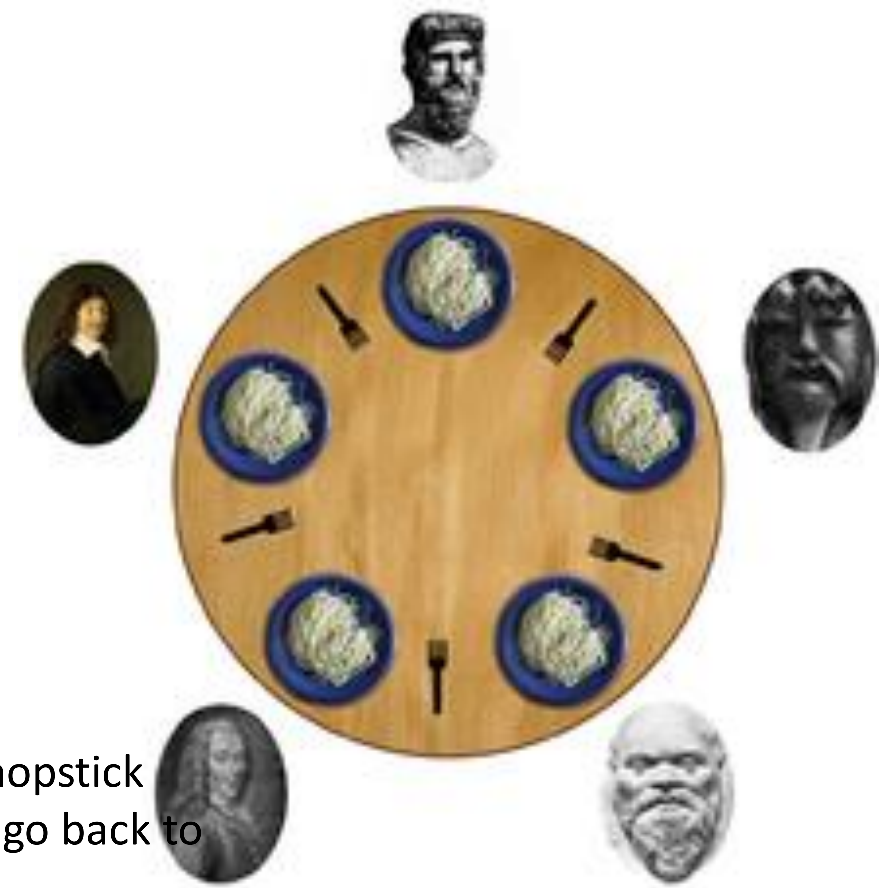
 Producer(item) {
   semaP(&emptySlots);       // Wait until space
   semaP(&mutex);            // Wait until machine free
   Enqueue(item);
   semaV(&mutex);
   semaV(&fullSlots);        // Tell consumers there is
                             // more coke
 }
 Consumer() {
   semaP(&fullSlots);        // Check if there's a coke
   semaP(&mutex);            // Wait until machine free
   item = Dequeue();
   semaV(&mutex);
   semaV(&emptySlots);       // tell producer need more
   return item;
 }
```

fullSlots signals coke

Critical sections using mutex protect integrity of the queue

emptySlots signals space

Dining Philosophers Problem



- Five philosophers, each either eats or thinks
- Share a circular table with five chopsticks
- Thinking: do nothing
- Eating => need two chopsticks, try to pick up two closest chopsticks
  – Block if neighbor has already picked up a chopstick
- After eating, put down both chopsticks and go back to thinking

Dining Philosophers attempt 1

```
Semaphore  chopsticks[5];

do{
   wait(chopstick[i]);  // left chopstick
   wait(chopstick[(i+1)%5 ]); // right chopstick
      // eat
   signal(chopstick[i]);  // left chopstick
   signal(chopstick[(i+1)%5 ]); // right chopstick
      // think
   } while(TRUE);
```

Dining Philosophers attempt 2

```
monitor DP  {
        enum { THINKING; HUNGRY,
EATING) state [5] ;
        condition self [5];

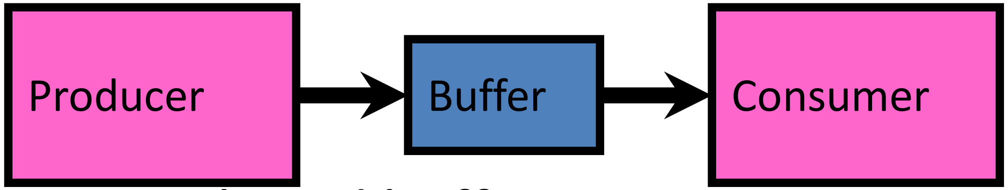
void synchronized pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING)
            self[i].wait;
        }

void synchronized putdown (int i) {
           state[i] = THINKING;
       //test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
        }
```

```
void test (int i) {
if ( (state[(i + 4) % 5] != EATING)&&
(state[i] == HUNGRY) &&
     (state[(i + 1) % 5] != EATING) ) {
                 state[i] = EATING ;
                 self[i].signal () ;
  }
}

       initialization_code() {
          for (int i = 0; i < 5; i++)
               state[i] = THINKING;
        }
}
```

Producer-consumer with a bounded buffer



- Problem Definition
  - Producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization to coordinate producer/consumer
- Don't want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty

- Example: Coke machine
  - Producer can put limited number of cokes in machine
  - Consumer can't take cokes out if machine is empty

Full Solution to Bounded Buffer

```
Semaphore fullSlots = 0;             // Initially, no coke
Semaphore emptySlots = bufSize;
                                     // Initially, num empty slots
Semaphore mutex = 1;                 // No one using machine

Producer(item) {
    emptySlots.P();                  // Wait until space
    mutex.P();                       // Wait until machine free
    Enqueue(item);
    mutex.V();
    fullSlots.V();                   // Tell consumers there is
                                     // more coke
}
Consumer() {
    fullSlots.P();                   // Check if there's a coke
    mutex.P();                       // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptySlots.V();                  // tell producer need more
    return item;
}
```

Motivation for Monitors and Condition Variables

- Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints

- Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
  - Some languages like Java provide this natively
  - Most others use actual locks and condition variables

Motivation for Monitors and Condition Variables

- Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores
- Problem is that semaphores are dual purpose:
  - They are used for both mutex and scheduling constraints
  - Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?

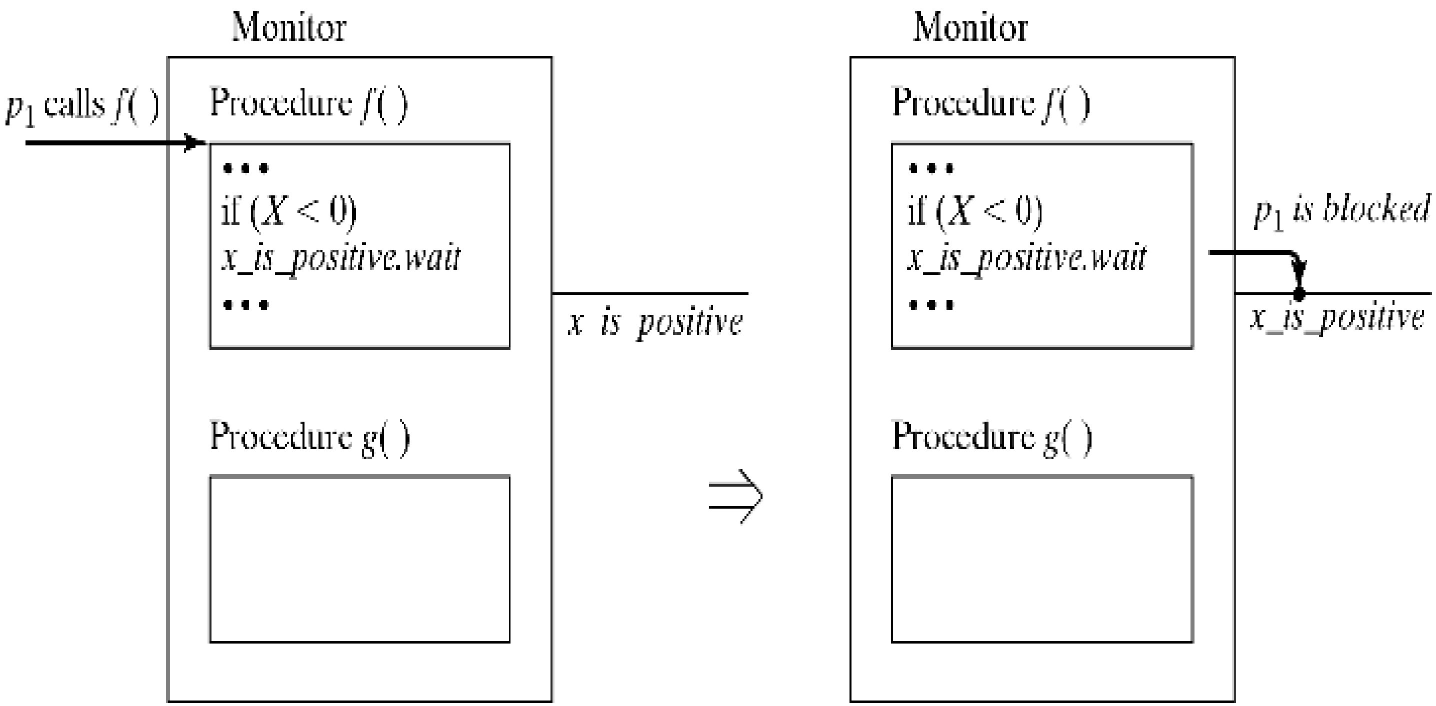
Motivation for Monitors and Condition Variables

- Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints

- Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
  - Some languages like Java provide this natively
  - Most others use actual locks and condition variables

Monitors

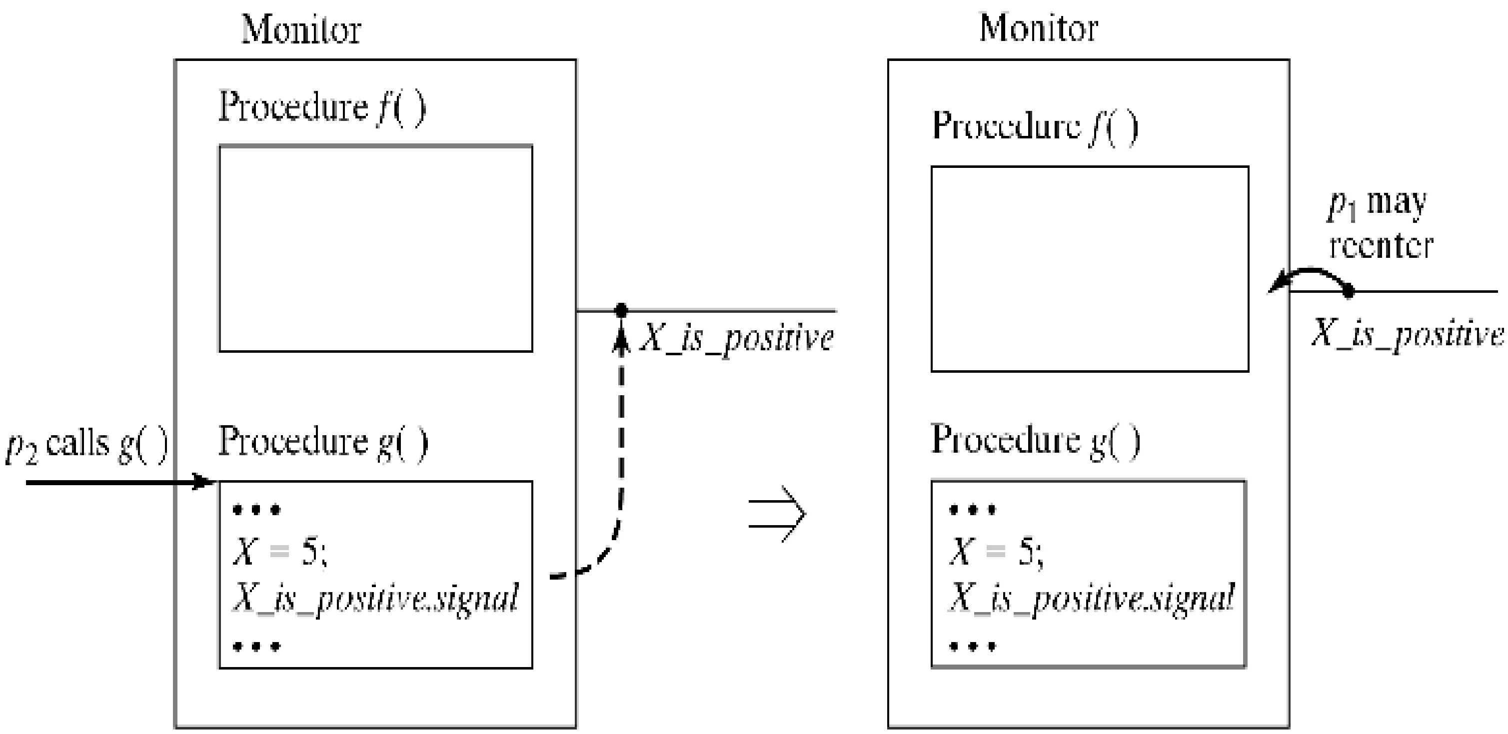
- Example: process p1 needs to wait until a variable X becomes positive before it can proceed



- p1 steps out to wait on queue associated with condition variable X_is_positive

Monitors

- Another process may execute X_is_positive.signal to wake up p1 when X becomes positive



- process p1 may reenter the monitor, however …

Monitors

- Design Issue:
  - After c.signal, there are **2 ready processes**:
    - The **calling** process that did the c.signal
    - The **waiting** process that the c.signal woke up
  - Which should continue?

    (Only one can be executing inside the monitor!)

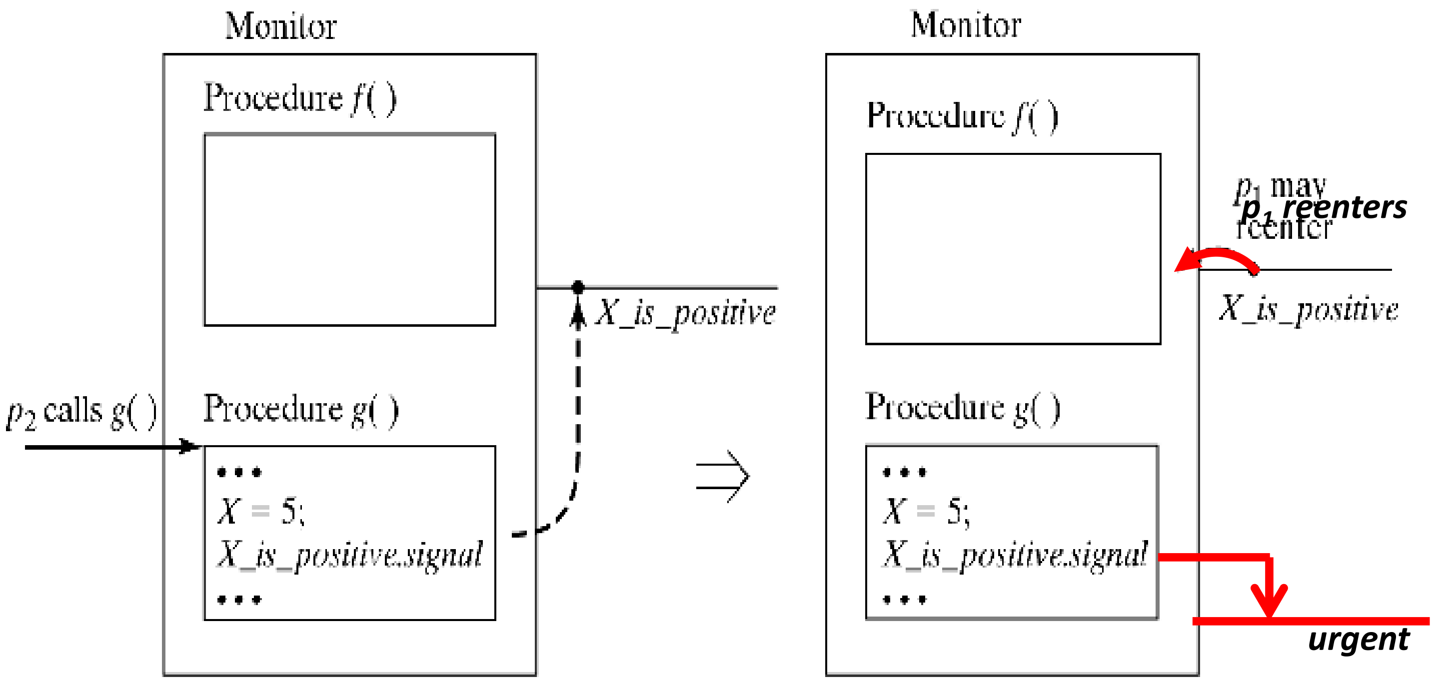
  Two different approaches
  - **Hoare** monitors
  - **Mesa-style** monitors

Hoare Monitors

- Introduced by Tony Hoare in a 1974
  http://wikipedia.org/wiki/C._A._R._Hoare
- First implemented by Per Brinch Hansen in Concurrent Pascal
  http://wikipedia.org/wiki/Per_Brinch_Hansen
- Approach taken by Hoare monitor:
  - After c.signal,
    - **Awakened process continues**
    - **Calling process is suspended**, and placed on **high-priority** queue

Hoare Monitors

- Effect of c.signal



- Signaling process has the highest priority
  - It reenters as soon as p1 terminates

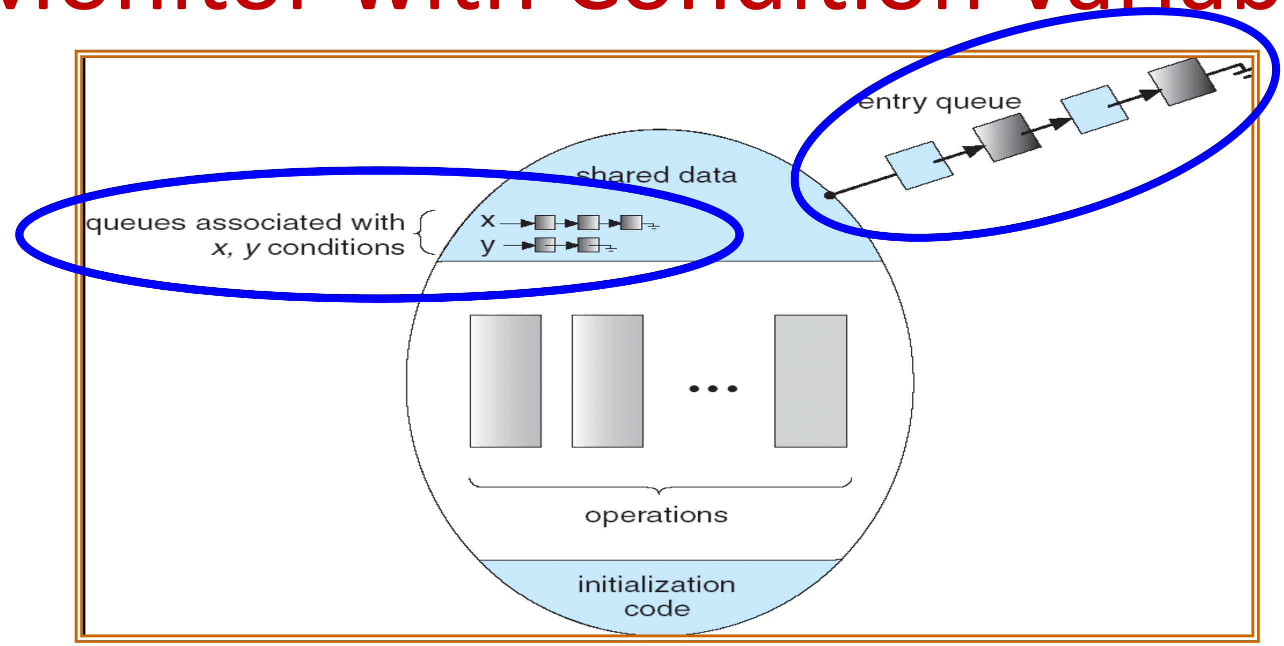
Mesa-style monitors

- After issuing c.signal, **calling process continues** executing
- Problem: **Condition cannot be guaranteed** after wakeup
  - Assume that p_1 and p_2 are waiting for some condition c
  - Caller could satisfying c and wake up both processes
  - Assume p_1 starts and makes the condition false again
  - When p_2 resumes, c is not guaranteed to be true
- Solution: process must **retest** c after wakeup

  instead of: if (!condition) c.wait

  use: while (!condition) c.wait
- This form of signal is sometimes called **notify**

Monitor with Condition Variables



- Lock: the lock provides mutual exclusion to shared data
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free
- Condition Variable: a queue of threads waiting for something *inside* a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep

Simple Monitor Example

- Here is an (infinite) synchronized queue

```
Lock lock;
Queue queue;

AddToQueue(item) {
    lock.Acquire();              // Lock shared data
    queue.enqueue(item);         // Add item
    lock.Release();              // Release Lock
}

RemoveFromQueue() {
    lock.Acquire();              // Lock shared data
    item = queue.dequeue();// Get next item or null
    lock.Release();              // Release Lock
    return(item);                // Might return null
}
```

- Not very interesting use of “Monitor”
  – It only uses a lock with no condition variables
  – Cannot put consumer to sleep if no work!

Complete Monitor Example (with condition variable)

- Here is an (infinite) synchronized queue

```
Lock lock;
Condition dataready;
Queue queue;

AddToQueue(item) {
    lock.Acquire();                    // Get Lock
    queue.enqueue(item);               // Add item
    dataready.signal();                // Signal any waiters
    lock.Release();                    // Release Lock
}

RemoveFromQueue() {
    lock.Acquire();                    // Get Lock
    while (queue.isEmpty()) {
        dataready.wait(&lock); // If nothing, sleep
    }
    item = queue.dequeue();            // Get next item
    lock.Release();                    // Release Lock
    return(item);
}
```

Summary

- Locks construction based on atomic seq. of instructions
  - Must be very careful not to waste/tie up machine resources
    - Shouldn't spin wait for long
  - Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable
- Semaphores
  - Generalized locks
  - Two operations: P(), V()
- Monitors: A synchronous object plus one or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
    - Three Operations: Wait(), Signal(), and Broadcast()