KOÇ UNIVERSITY

Process Synchronization

Didem Unat

Lecture 9

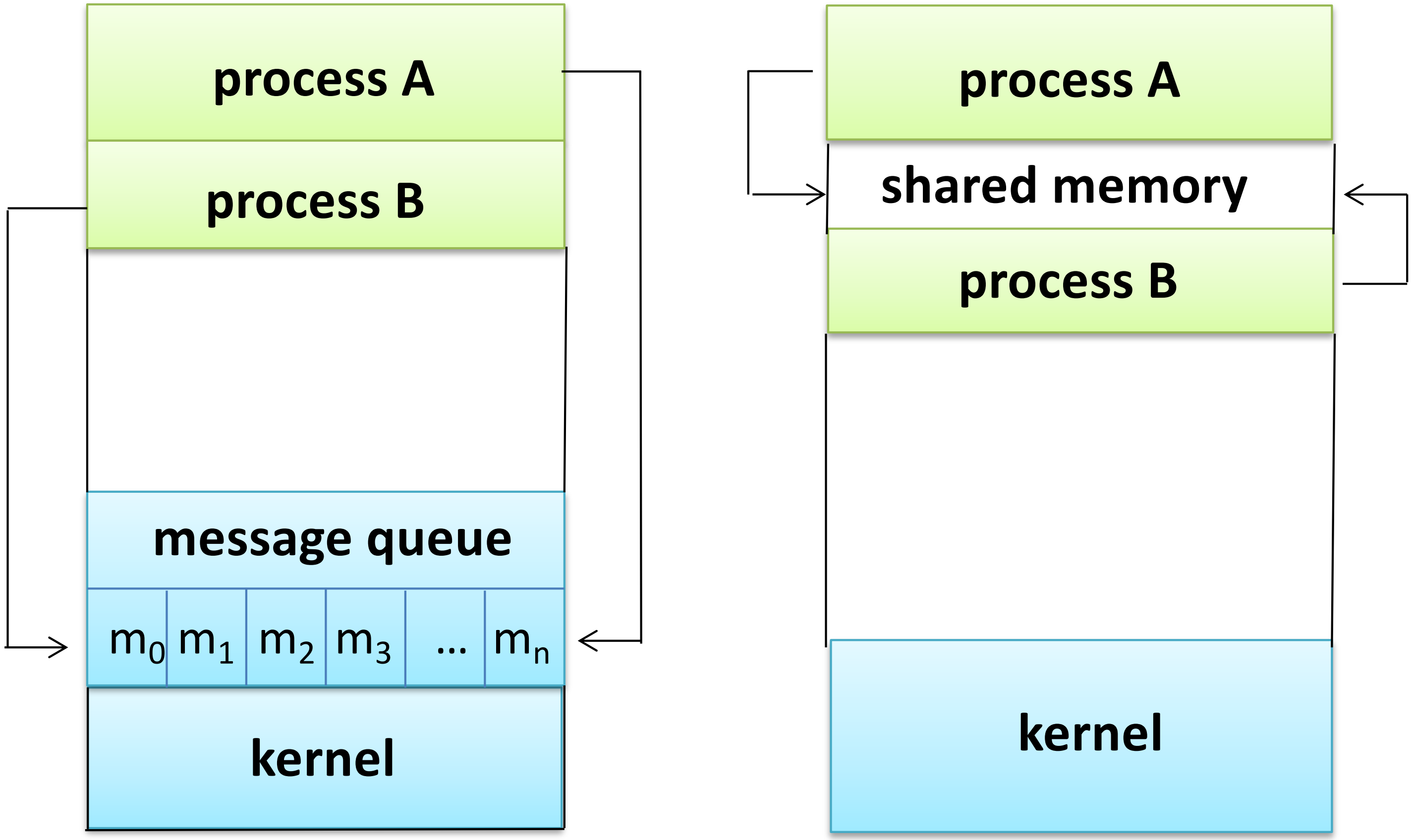
COMP304 - Operating Systems (OS)

Process Synchronization

- **Race Condition**
- **The Critical-Section Problem**
- **Synchronization Hardware**
- **Semaphores**
- **Classical Problems of Synchronization**
- **Monitors**
- **Synchronization Examples**

Two Models of Communication

Message Passing vs Shared Memory



- Message passing requires the message of A to be copied to a buffer and copied to process B's memory – thus it is slower but safer

Shared Memory

- Communication through shared memory takes place with shared variables.

- Threads or processes share common variables

- Access to these variables should be coordinated (synchronized) so that the data is not corrupted.

Problem

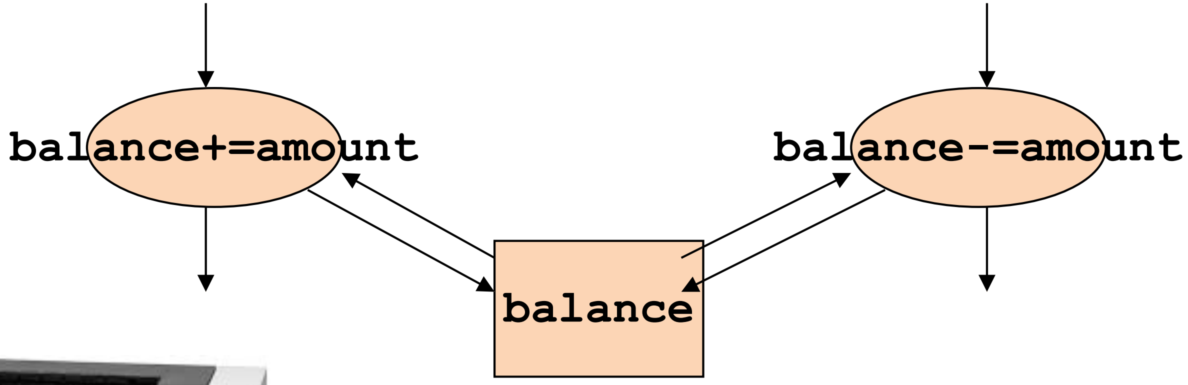
```
shared double balance;
```

Code for p_1 (deposit)

```
  . . .
balance = balance + amount;
  . . .
```

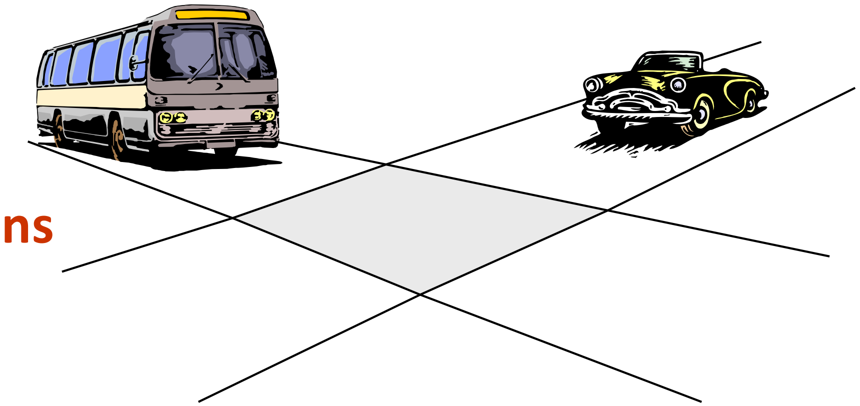
Code for p_2 (withdraw)

```
 . . .
balance = balance - amount;
 . . .
```



Interleaved Printing

Traffic Intersections



Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution

- Concurrent access to shared data may result in data inconsistency.

- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Producer-Consumer Problem

- A **producer** process "produces" information "consumed" by a **consumer** process.
- Here are the variables needed to define the problem:

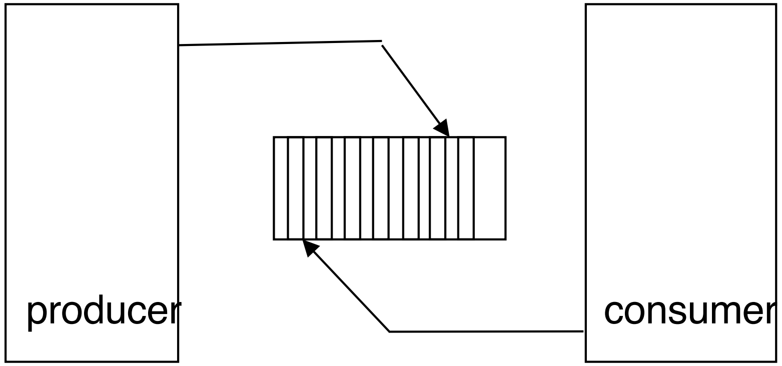
```
#define BUFFER_SIZE 10
typedef struct {
    DATA     data;
} item;

item   buffer[BUFFER_SIZE];
int    in = 0;       // Location of next input to buffer
int    out = 0;      // Location of next removal from buffer
int    counter = 0;  // Number of items in the buffer
```

Counter is initialized to 0 and incremented each time a new item is added to the buffer and decremented when an item is consumed.

Producer-Consumer Problem

```
item   nextProduced;          //PRODUCER

while (TRUE) {
    while (counter == BUFFER_SIZE);
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

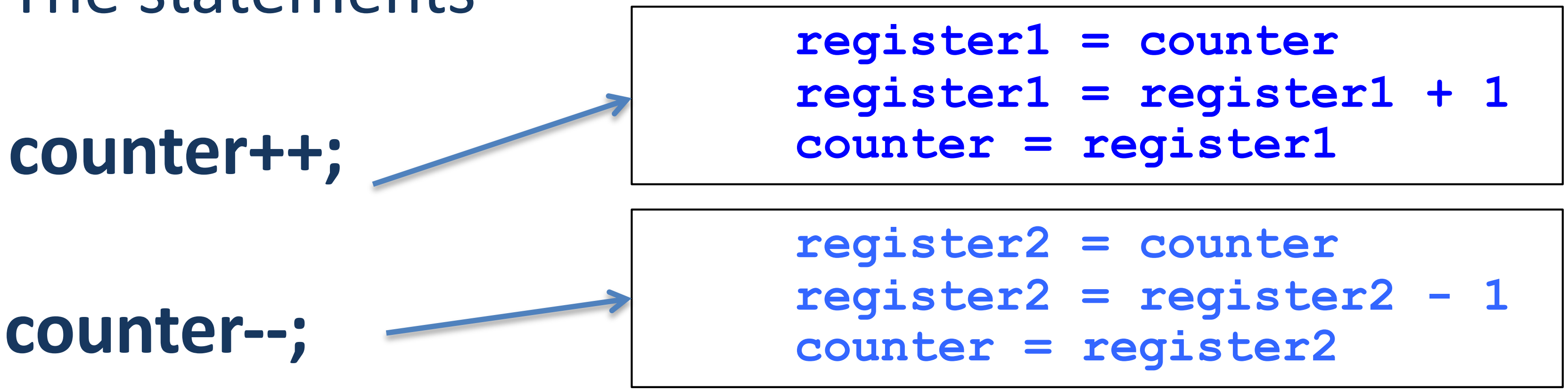
```
item  nextConsumed;          //CONSUMER

while (TRUE) {
    while (counter == 0);
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```



Example:
Producer produces papers to be printed.
Consumer prints the papers.
They use a common queue.

Counter

- The statements



counter++;

```
register1 = counter
register1 = register1 + 1
counter = register1
```

counter--;

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Even though these are single statements in the code, they are compiled into and executed as multiple instructions in the hardware.

Race Condition

Producer	Consumer		Counter
			5
read value		←	5
increase value			5
write back		→	6
	read value	←	6
	Decrease value		6
	write back	→	5

Producer	Consumer		Counter
			5
read value		←	5
	read value	←	5
increase value			5
	decrease value		5
write back		→	6
	write back	→	4

Consider this execution (shown on the right) interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 – 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Problem

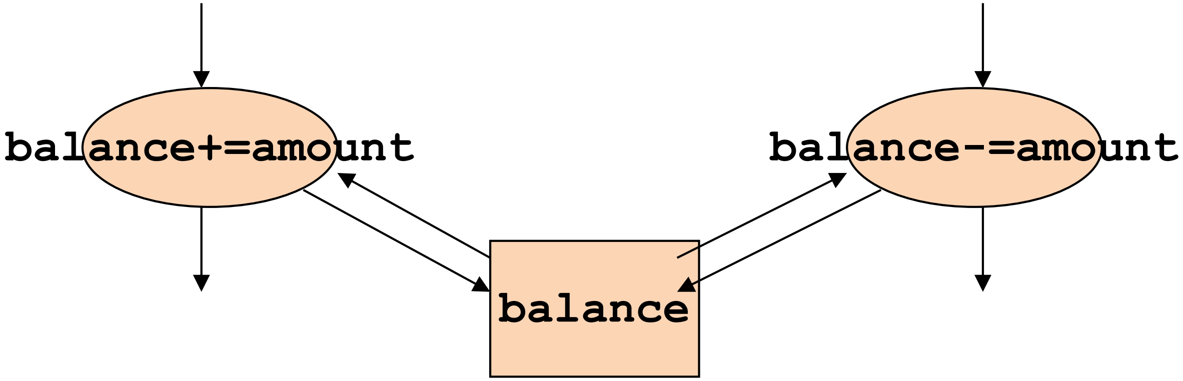
```
shared double balance;
```

Code for p_1 (deposit)

```
 . . .
balance = balance + amount;
 . . .
```

Code for p_2 (withdraw)

```
. . .
balance = balance - amount;
. . .
```

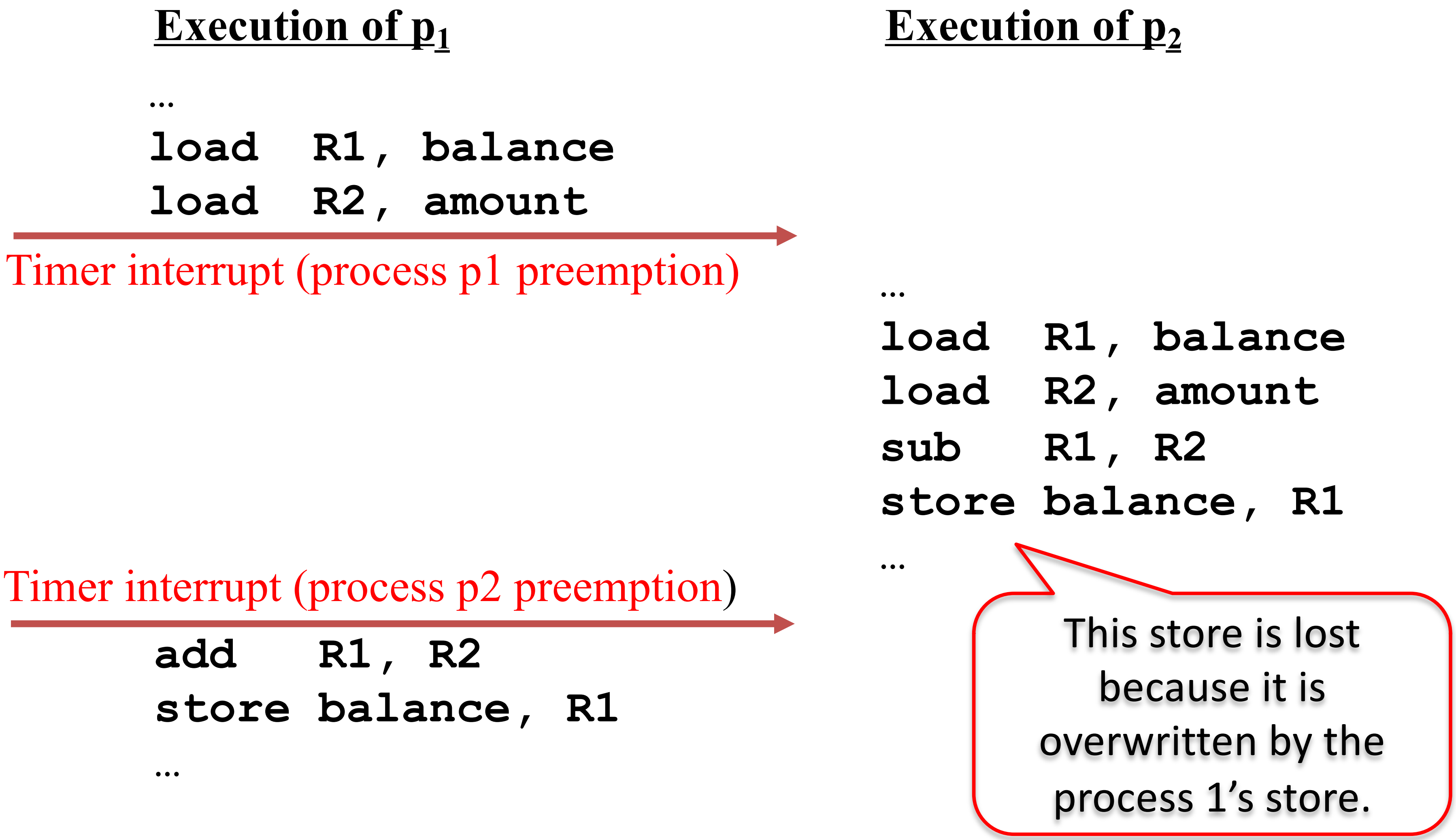


balance+=amount

balance-=amount

balance

Load, Execute, Store



Execution of p_1

```
…
load  R1, balance
load  R2, amount
```

Timer interrupt (process p1 preemption)

Execution of p_2

```
…
load  R1, balance
load  R2, amount
sub   R1, R2
store balance, R1
…
```

Timer interrupt (process p2 preemption)

```
add   R1, R2
store balance, R1
…
```

This store is lost because it is overwritten by the process 1’s store.

Race Condition

- **Race condition:** The situation where several processes access and manipulate shared data concurrently. The final value of the shared data is **non-deterministic** and depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be **synchronized**.

Preemptive Kernels

- A non-preemptive kernel is free from race conditions on kernel data structures

- A preemptive kernel allows a process to be preempted while it is running in kernel mode
  - Need to ensure that shared kernel data are free from race conditions
  - Examples of shared kernel data
    - List of open files, interrupt handlers, process list / queues, memory allocation management

The Critical Section Problem

- Consider system of ***n*** processes {$\boldsymbol{p_0}$, $\boldsymbol{p_1}$, **…** $\boldsymbol{p_{n-1}}$} and all competing to use some shared data
  - Updating a table, writing into a file etc.

- Each process has a code segment, called **critical section**, in which the shared data is accessed.

- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Critical Section

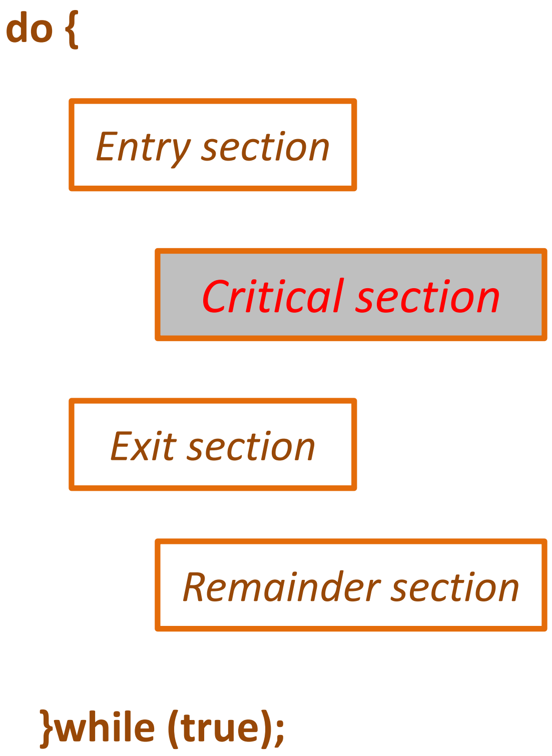
A Critical Section Environment contains:

Entry Section Code requesting entry into the critical section.

Critical Section Code in which only one process can execute at any one time.

Exit Section The end of the critical section, releasing or allowing others in.

Remainder Section Rest of the code AFTER the critical section.



```
do {
    Entry section
        Critical section
    Exit section
        Remainder section
}while (true);
```

Solution to Critical Section Problem

Any solution to Critical Section problem must provide all of the followings

1. **Mutual Exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

Solution to Critical Section Problem

Any solution to Critical Section problem must provide all of the followings

1. **Mutual Exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress**. If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

Solution to Critical Section Problem

Any solution to Critical Section problem must provide all of the followings

1. **Mutual Exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress**. If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. **Bounded Waiting**. A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Attempt 1 to Solve Problem

- Only 2 processes, P_0 and P_1

i is the current process, **j** the "other" process.

```
//Pi:
//turn =>shared variable, initially i
//if turn == i => Pi can enter its
critical

do {
    while  ( turn  !=  i );
    /* critical section  */
    turn  =  j;
    /* remainder section */

} while(TRUE);
```

```
//Pj:
//turn =>shared variable, initially i
//if turn == j => Pj can enter its
critical

do {
    while  ( turn  !=  j );
    /* critical section  */
    turn  =  i;
    /* remainder section */

} while(TRUE);
```

Are the three Critical Section Requirements Met?

Satisfies mutual exclusion, but not progress because Pi needs to enter the critical section, then only Pj can enter.

Attempt 2 to Solve Problem

Shared variables

- `boolean flag[2];`
  `//initially flag[0] = flag[1] = false.`
- `If flag[i] == true` ⇒ P_i ready to enter its critical section

```
do {
   flag[i] = true;
   while (flag[j]) ;

   //critical section

   flag [i] = false;

  //remainder section
} while (true);
```

Each process maintains a flag indicating that it wants to get into the critical section. It checks the flag of the other process and doesn't enter the critical section if other process wants to get in.

Are the three Critical Section Requirements Met?

Satisfies mutual exclusion, but not progress and not bounded waiting

Attempt 3: Peterson’s Solution

Combined shared variables of algorithms 1 and 2.

```
bool flag[0] = false;
bool flag[1] = false;
int turn;  //shared variable
```

```
P0: flag[0] = true;
P0: turn = 1;

while (flag[1] && turn == 1) {
// busy wait }

// critical section

flag[0] = false;

//remainder section
```

```
P1: flag[1] = true;
P1: turn = 0;

while (flag[0] && turn == 0) {
// busy wait }

// critical section

flag[1] = false;

//remainder section
```

Meets all three requirements; solves the critical-section problem for two processes.

Peterson’s Solution

- **Mutual exclusion:** P0 and P1 can never be in the critical section at the same time: If P0 is in its critical section, then flag[0] is true. In addition, either flag[1] is false (meaning P1 has left its critical section), or turn is 0 (meaning P1 is just now trying to enter the critical section, but graciously waiting), or P1 is trying to enter its critical section, after setting flag[1] to true but before setting turn to 0. So if both processes are in their critical sections then we conclude that the state must satisfy flag[0] and flag[1] and turn = 0 and turn = 1. No state can satisfy both turn = 0 and turn = 1, so there can be no state where both processes are in their critical sections.
- **Progress:** A process cannot immediately re-enter the critical section if the other process has set its flag to say that it would like to enter its critical section.
- **Bounded waiting:** In Peterson's algorithm, a process will not wait longer than one turn for entrance to the critical section: After giving priority to the other process, this process will run to completion and set its flag to 0, thereby allowing the other process to enter the critical section.

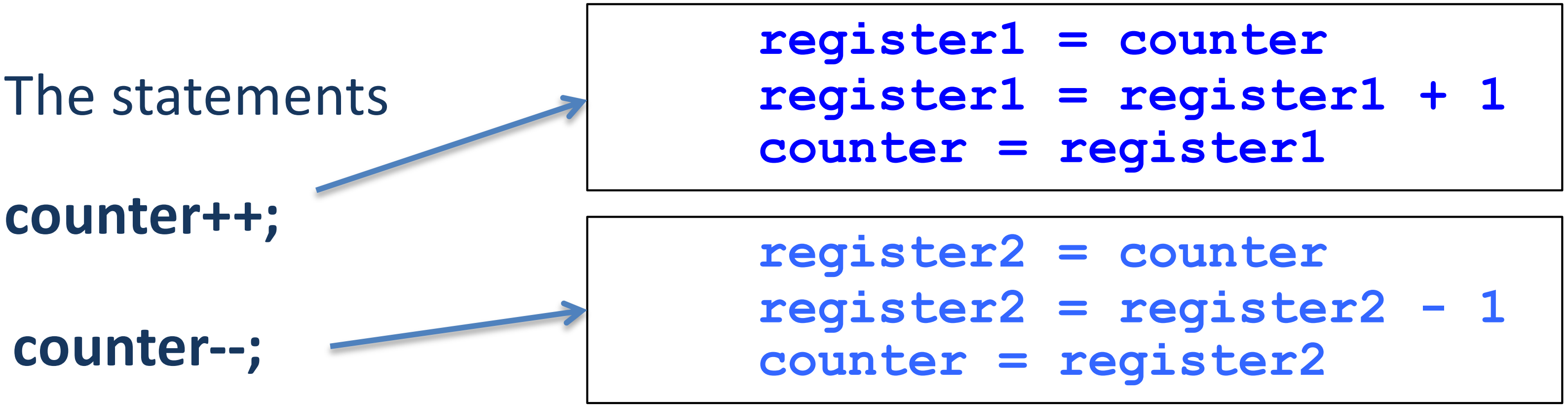
Peterson’s Solution

- Peterson solution is **not correct** on today’s modern computers
  - Because update to turn is not specified as **atomic**
  - We have **caches and multiple copies** of the same data (turn variable) in the hardware
- Process 0 may see turn as 1
- Process 1 may see turn as 0
- Resulting in data race

- We need hardware support for avoiding race conditions.

Atomic Instructions

- Peterson’s solution is a software-based solution
- Modern machines provide special atomic hardware instructions
  - **Atomic** = indivisible instructions
- The statements



counter++;

```
register1 = counter
register1 = register1 + 1
counter = register1
```

counter--;

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- must be performed **atomically**.
- Atomic operation means an operation that completes in its entirety **without interruption (preemption)**.

Atomic Test-and-Set Instruction

- The term *locks* are used to indicate getting a key to enter a critical section.
- The **test-and-set** instruction is an instruction used to write to a memory location and return its old value as a single atomic (i.e., non-interruptible) operation.
- If multiple processes may access the same memory location, and if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process is done.

```
boolean TestAndSet(boolean *lock) {
  boolean initial = *lock;
  *lock = true;
  return initial;
}
```

Mutual Exclusion with Test-and-Set

- Shared variable:

```
boolean lock = false;
```

- Process P_i

```
do {
    while (TestAndSet(&lock)) { };
    critical section
    lock = false;
    remainder section

} while (true);
```

Mutual Exclusion with Test-and-Set

- Shared variable:

```
boolean lock = false;
```

- Process P_i

```
do {
     while (TestAndSet(&lock)) { };
     critical section
     lock = false;
     remainder section

} while (true);
```

```
boolean TestAndSet(boolean *lock) {
  boolean initial = *lock;
  *lock = true;
  return initial;
}
```

The calling process obtains the lock if the old value was **False**. It spins until it acquires the lock. When it acquires, the value turns to **True** preventing other processes to acquire the lock.

Must be careful if these approaches are to satisfy a bounded wait condition - must use round robin

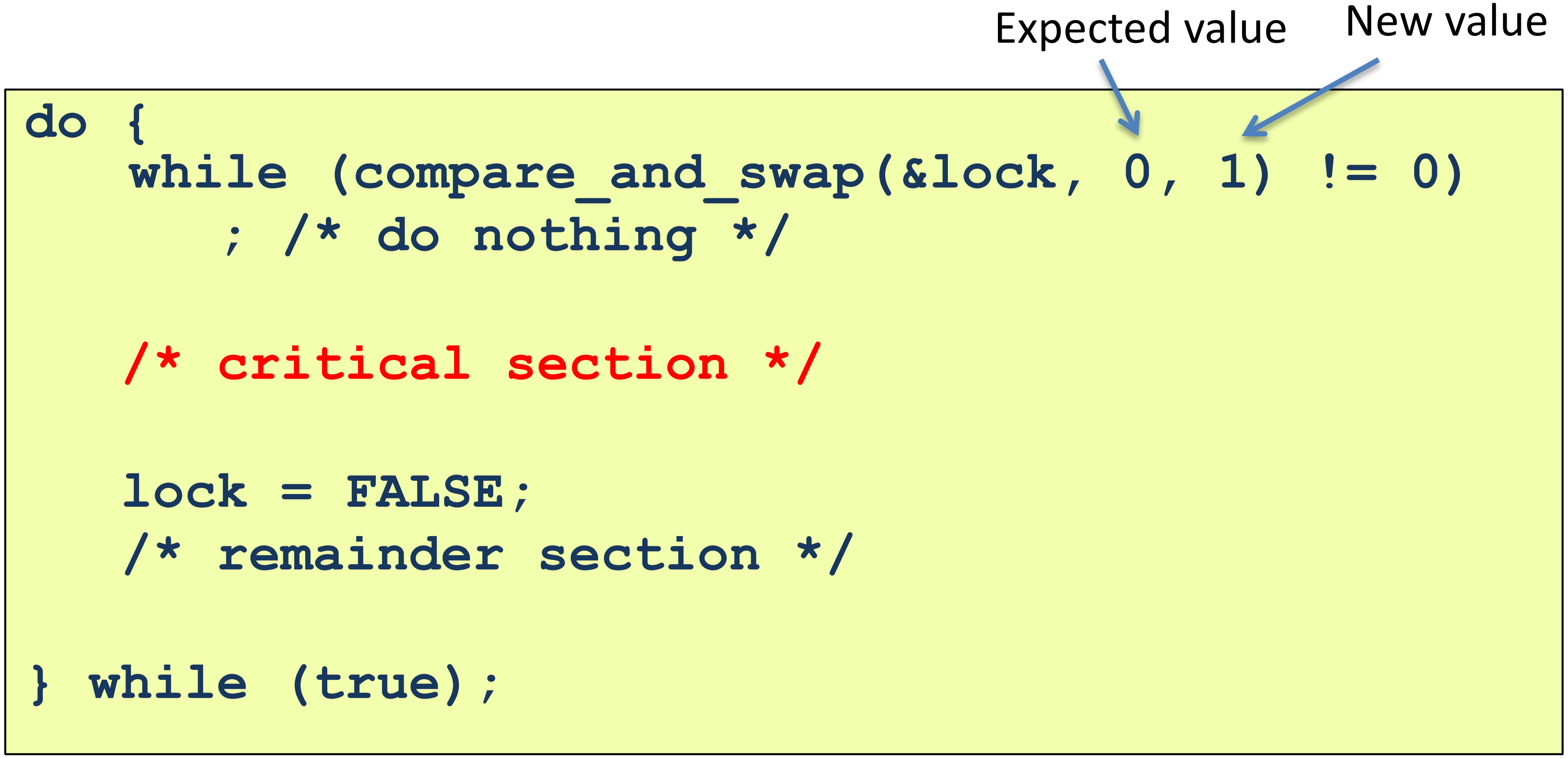
compare_and_swap instruction

- It compares the contents of a memory location to a given value and, **only if they are the same**, modifies the contents of that memory location to a given new value.
- Done as a **single atomic operation**
- if the value had been updated by another process in the meantime, the write would fail.

```
int compare_and_swap(int *value,
                     int expected, int new_value) {
   int oldValue = *value;
   if (*value == expected)
      *value = new_value;
   return oldValue;
}
```

Use of compare_and_swap instruction

- Shared boolean variable *lock* initialized to FALSE (0)



```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = FALSE;
    /* remainder section */

} while (true);
```

Mutex Locks

- Previous hardware solutions are complicated and generally too low level to application programmers

- OS designers build software tools to solve critical section problem on top of these hardware solutions

- Enter critical regions by first **acquire()** a lock then **release()** it
  - Boolean variable indicates if lock is available or not
  - Next lecture, we will cover those
- Note that these solutions still use hardware solutions/support underneath

acquire() and release()

```
do {
   acquire lock
      critical section
   release lock
      remainder section
} while (true);
```

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;;
}

release() {
   available = true;
}
```

- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via *hardware atomic instructions* discussed few slides back.
- This solution requires **busy waiting**
  - This type of lock is called a **spinlock**

Bounded-waiting Mutual Exclusion with test_and_set

```
//Round-robin implementation
Boolean waiting[N];
int j;
//takes on values from 0 to N-1
Boolean key;
```

Each process tries to test_and_set for the lock. Only one succeeds with key=false, then it sets its waiting to false. Enters the critical section. Before it exits, it tries to find a process j who is waiting. If j is not i, then process i hands in the lock to process j.

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

   /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;

   /* remainder section */

} while (true);
```

Reading

- Read Chapter 6

- Acknowledgments
  - These slides are adapted from
    - Öznur Özkasap (Koç University)
    - Operating System and Concepts (9th edition) Wiley
    - Jerry Breecher