
AWS
AWS

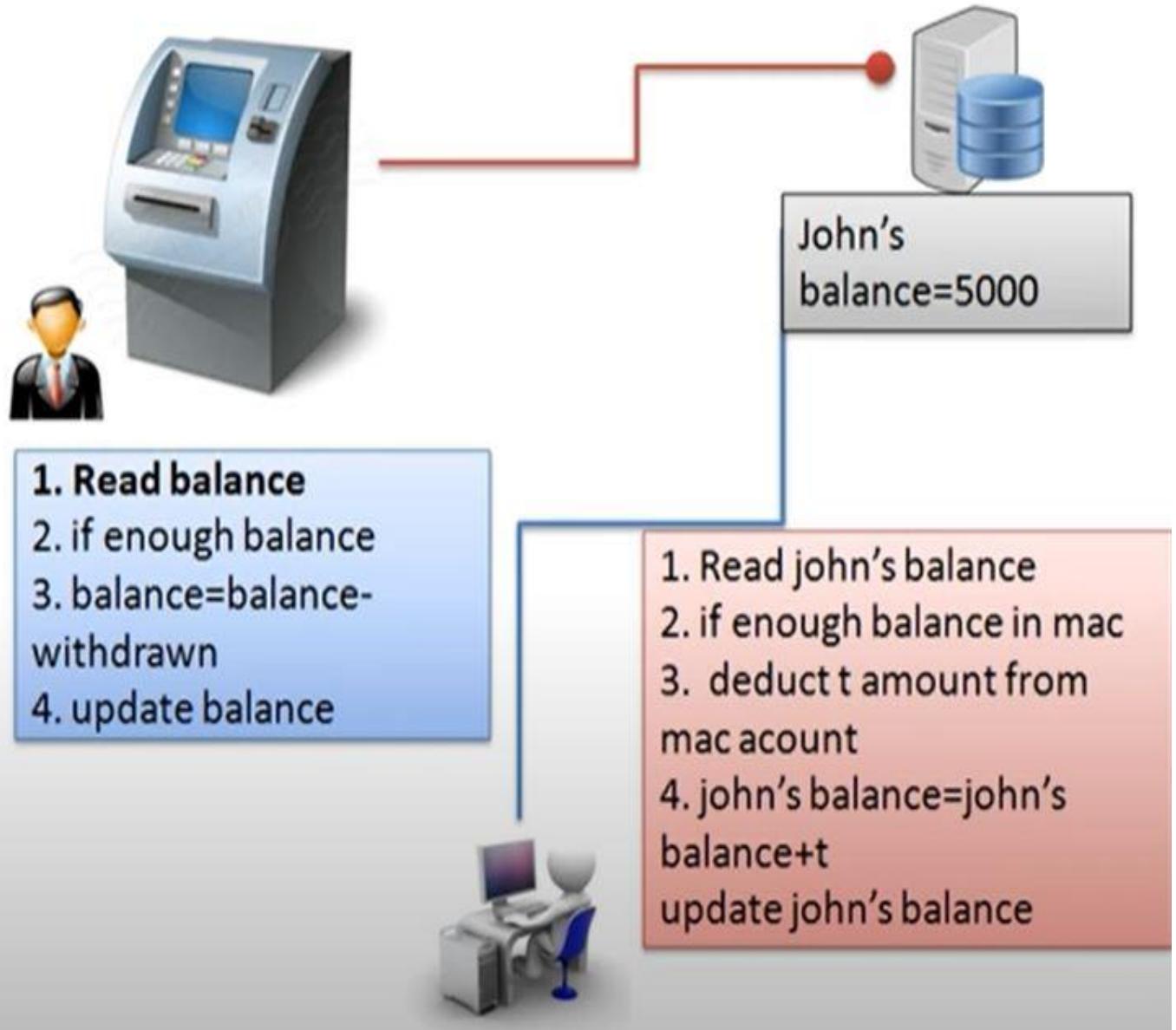
Process Synchronization

- Process Synchronization is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources at the same time.
- It is specially needed in a multi-process system when multiple processes are running together, and more than one process try to gain access to the same shared resource or data at the same time.
- This can lead to the inconsistency of shared data.

Race Condition

- Process synchronization is mainly used for cooperating process which shared the resources concurrently.
- A situation, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called race condition.
- Race condition leads to the data inconsistency.

RACE CONDITION



Race Condition

John withdraw 2000.

John's balance=5000

balance=5000

balance=3000

1. Read balance
2. if enough balance
3. balance=balance-withdrawn
4. update balance

1. Read john's balance
2. if enough balance in mac
3. deduct t amount from mac acount
4. john's balance=john's balance+t

update john's balance

balance=5000
balance=9000

mac transferred 4000.

RACE CONDITION

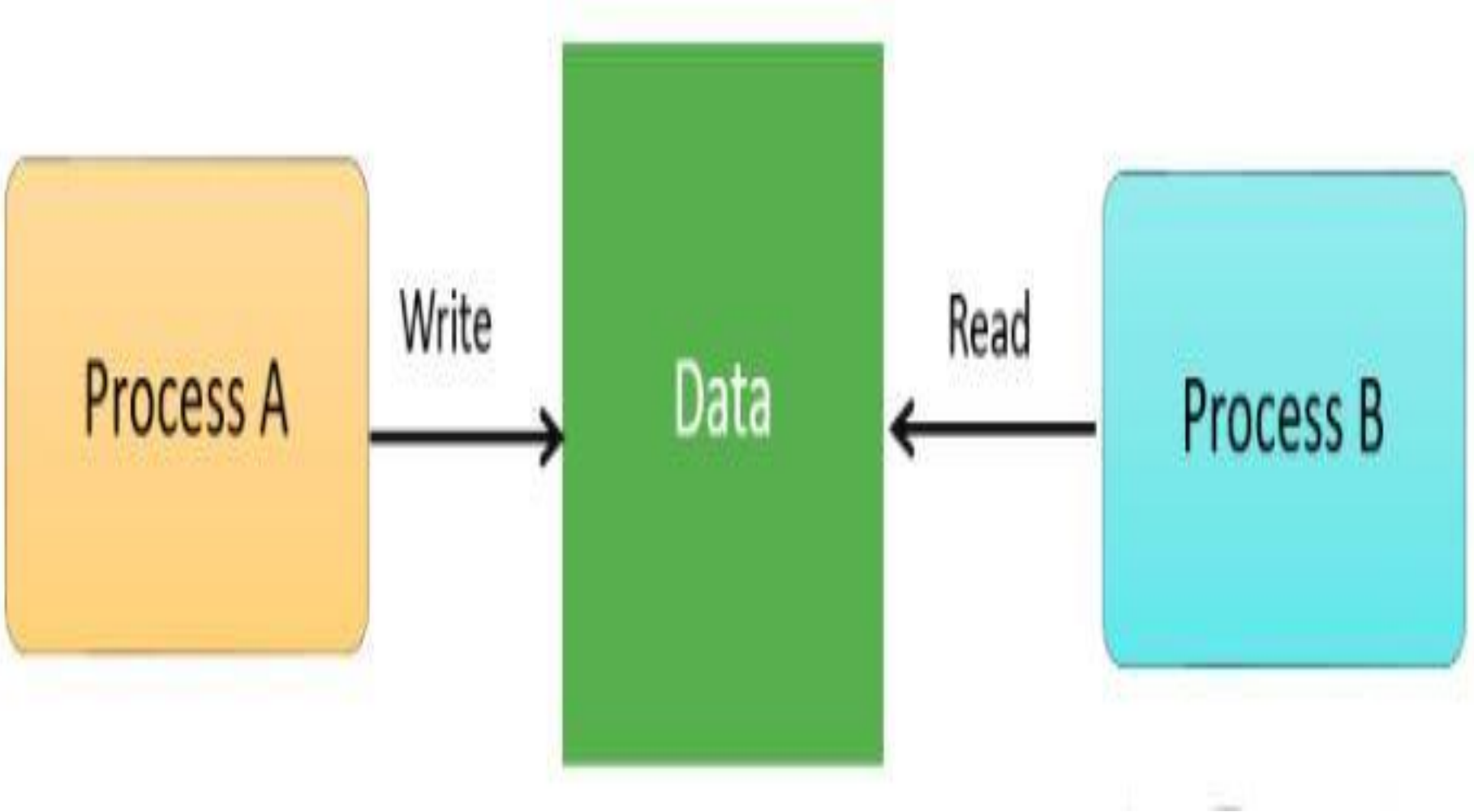
John withdraw 2000.

John's balance=5000

Final amount must be 7000.

mac transferred 4000.

CRITICAL SECTION

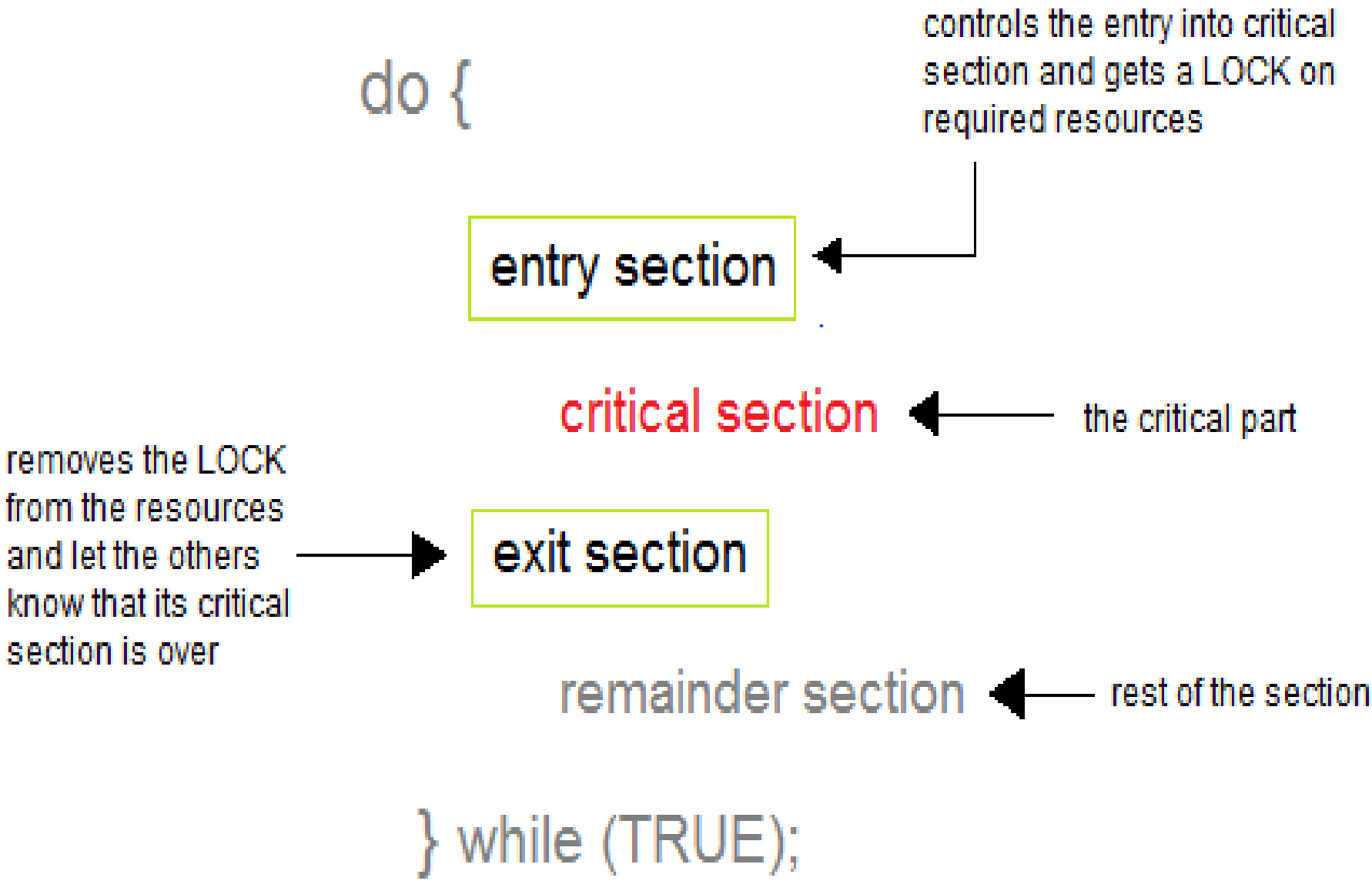


CRITICAL SECTION

- The critical section is a code segment where the shared variables can be accessed.
- A race condition is a situation that may occur inside a critical section.
- Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction.i.e. only one process can execute in its critical section at a time.
- All the other processes have to wait to execute in their critical sections.

Constituents of Critical Section

- **Entry Section –** To enter the critical section code, a process must request permission. Entry Section code implements this request.
- **Critical Section –** This is the segment of code where process changes common variables, updates a table, writes to a file and so on. When one process is executing in its critical section, no other process is allowed to execute in its critical section.
- Exit Section – After the critical section is executed , this is followed by exit section code which marks the end of critical section code.
- Remainder Section – The remaining code of the process is known as remaining section.


do {
controls the entry into critical section and gets a LOCK on required resources
entry section
critical section
the critical part
removes the LOCK from the resources and let the others know that its critical section is over
exit section
remainder section
rest of the section
} while (TRUE);

Properties to Implement Critical Section

1. Mutual exclusion
2. Progress
3. Bounded waiting

Mutual exclusion: When a process is executing in its critical section, no other process can be executing in their critical sections.

Progress & Bounded waiting

Properties to Implement Critical Section

Progress: If no process is executing in its critical section, and if there are some processes that wish to enter their critical sections, then one of these processes will get into the critical section.

Bounded waiting: After a process makes a request to enter its critical section, there is a bound on the number of times that other processes are allowed to enter their critical sections, before the request is granted.

Two-Process Solution: Algorithm 1

- Shared variables:
  - **int turn**;
    initially **turn = 0**
  - **if turn = i** $\Rightarrow$ P_i can enter its critical section
- Process P_i

```
do {
    while (turn != i) ;    //if not my turn
        critical section
    turn = j;              // others turn
        reminder section
} while (1);
```

- Satisfies mutual exclusion, but not progress

- Shared variables
  - **boolean flag[2];**
    initially **flag [0] = flag [1] = false.**
  - **flag [i] = true** $\Rightarrow$ P_i ready to enter its critical section

- Process P_i

```
do {
    flag [i] := true;          // I want to go in
    while (flag [j]) ;         // Proceed if other not trying
    critical section
    flag [i] = false;          // I am out
    remainder section
} while (1);
```

- Satisfies mutual exclusion, and progress but not bounded waiting requirement.

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process P_i is ready!

Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process P_i

```
do {
    flag [i]:= true;          // I want to go in
    turn = j;

                              // Proceed if other not trying
    while (flag [j]==true && turn = j) ;
        critical section
    flag [i] = false;
        remainder section
} while (1);
```

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

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     `pi` enters CS only if: either

     `flag[j] = false` or **`turn = i`**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
    {
        boolean rv = *target;
        *target = TRUE;
        return rv:
    }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Solution using test_and_set()

Shared Boolean variable lock, initialized to FALSE

Solution:

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */
         /* critical section */
   lock = false;
         /* remainder section */
} while (true);
```

Solution using test_and_set()

```
boolean lock = false;

boolean TestAndSet(boolean &target){
   boolean returnValue = target;
   target = true;
   return returnValue;
}

while(1){
while(TestAndSet(lock));
CRITICAL SECTION CODE;
lock = false;
REMAINDER SECTION CODE;

}
```

Swap Pseudocode

```
boolean lock;
Individual key;
void swap(boolean &a, boolean &b){
    boolean temp = a;
    a = b;
    b = temp;
}
while (1){
    key = true;
    while(key==true)
        swap(lock,key);
    critical section
    lock = false;
    remainder section
```

TestandSet – Bounded-Waiting – Mutual Exclusion

- The algorithm given below makes use of the TestAndSet instruction and satisfies all the three requirements for a solution to the critical section problem. This is also a solution when there are n processes.

- The shared variables used in the algorithm are as follows:

```
boolean waiting[n];          //initially false
boolean lock  ;              //initially false
```

TestandSet – Bounded-Waiting – Mutual Exclusion

The algorithm for process Pi is given below:

```
boolean waiting[n];              //initially false
boolean lock  ;
do {
waiting[i] = TRUE; key = TRUE;
while (waiting[i] && key)
key = TestAndSet(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);
```