Synchronization Issues

IPC Issues

1. How do the processes communicate?
2. One process should not get into the way of another process when doing critical activities
3. Proper sequence of execution when dependencies are present
   - A produces data, B prints it
   - Before printing B should wait while A is producing data
   - B is dependent on A

Multithreading issues

- All the 3 mentioned issues apply to multiple threads as well

1. How do the threads communicate
   - Simpler due to shared address space
2. One thread should not get into the way of another thread when doing critical activities
3. proper sequencing when dependencies are present
   - A produces data, B prints it
   - Before printing B should wait while A is producing data
   - B is dependent on A

IPC Issues vs. Multithreading Issues

- Same solutions exists
- The only difference could be the level at which the solution is applied
  - Kernel level
  - User level
- From now on threads and processes both mean the same i.e. “Execution path”, unless otherwise specified

Common Storage

- Multiple processes may be sharing a common storage
- Read, Write, Update, Delete, Insert, etc
  - Common Main memory
  - Or Common File
  - Or Common xyz
- The nature of the common storage does not change the nature of the problem
- “Common storage” is an abstract concept
- Implementation may differ

Race Condition: Example Print Spooler

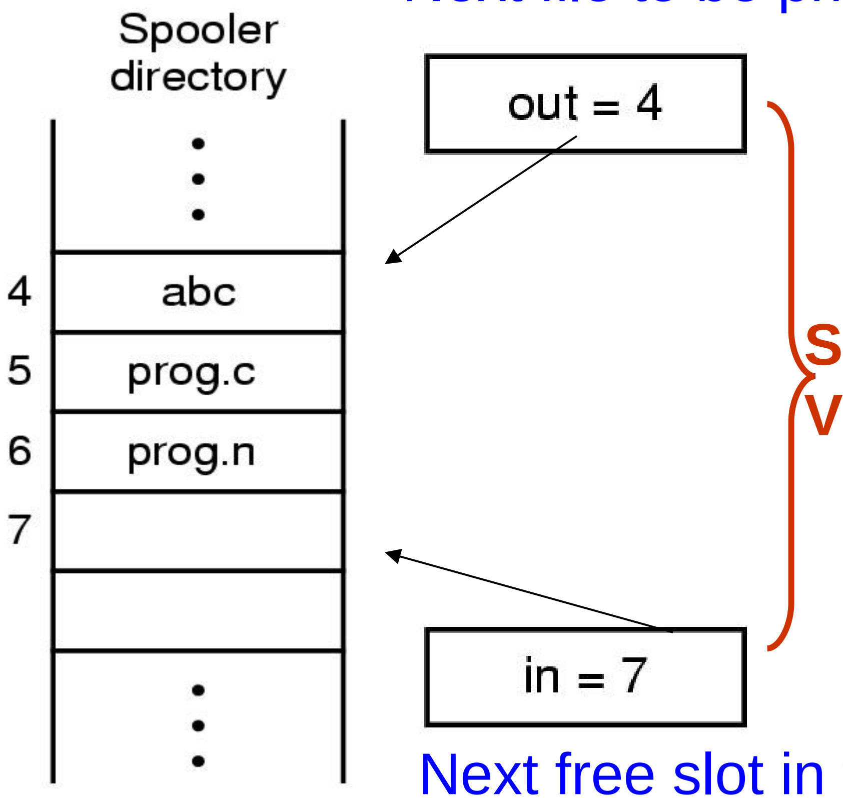
- What is race condition?
- a situation in which multiple processes read and write a shared data item and the final result depends on the relative timing of their execution
- If a process wishes to print a file it adds its name in a **Spooler Directory**
- The **Printer process**
  - Periodically checks the spooler directory
  - Prints a file
  - Removes its name from the directory

Example Print Spooler

If any process wants to print a file it will execute the following code

1. Read the value of **`in`** in a local variable **`next_free_slot`**
2. Store the name of its file in the **`next_free_slot`**
3. Increment **`next_free_slot`**
4. Store back in **`in`**

Next file to be printed



Shared Variables

Next free slot in the directory

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {

        /*produce an item and put in nextProduced  */
        while (count == BUFFER_SIZE); // do nothing

                buffer [in] = nextProduced;
                in = (in + 1) % BUFFER_SIZE;
                count++;
}
```

Consumer

```
while (true)  {
    while (count == 0); // do nothing

          nextConsumed =  buffer[out];
          out = (out + 1) % BUFFER_SIZE;
        count--;

        /*consume the item in nextConsumed*/
}
```

Race Condition

- count++ could be implemented as

  ```
  register1 = count
  register1 = register1 + 1
  count = register1
  ```

- count-- could be implemented as

  ```
  register2 = count
  register2 = register2 - 1
  count = register2
  ```

- Consider this execution interleaving with “count = 5” initially:

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

Solution to Critical-Section Problem

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
    - Assume that each process executes at a nonzero speed
    - No assumption concerning relative speed of the N processes

Example Print Spooler

Let A and B be processes who want to print their files

Process B will never receive any output

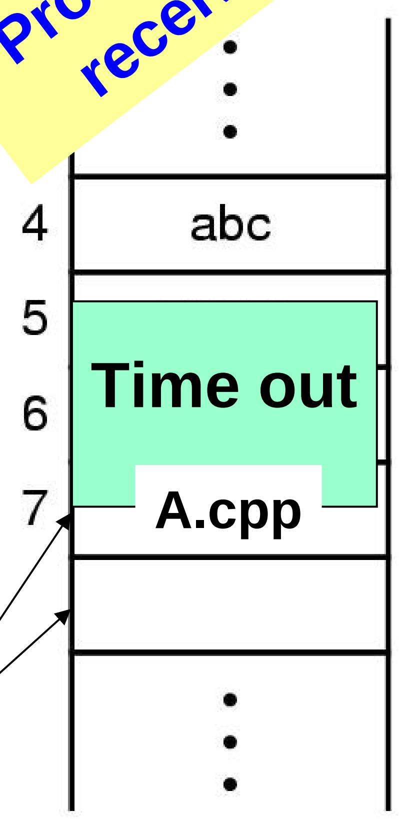
Process A

1. Read the value of **in** in a local variable **next_free_slot**

2. Store the name of its file in the **next_free_slot**

3. Increment **next_free_slot**

4. Store back in **in**



Process B

1. Read the value of **in** in a local variable **next_free_slot**

2. Store the name of its file in the **next_free_slot**

3 .Increment **next_free_slot**

4. Store back in **in**

$next\_free\_slot_a = 7$

in = 8

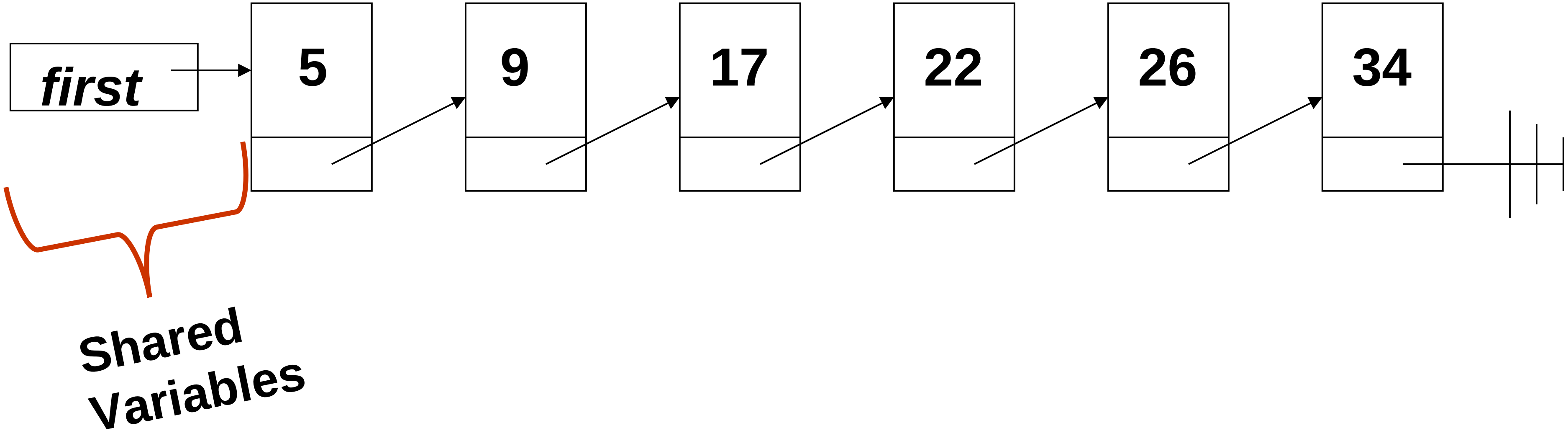
$next\_free\_slot_b = 7$

Race condition

- Data is shared
- The final result depends on which process runs first
- Debugging is not easy
- Most test runs will run fine

Other examples of Race conditions

- Adding node to a shared linked list



Reason behind Race Condition

- Process B started using one of the shared variables before process A was finished with it.
- At any given time a Process is either
  - Doing internal computation => no race conditions
  - Or accessing shared data that can lead to race conditions
- Part of the program where the shared memory is accessed is called **Critical Region**
- Races can be avoided
  - **If no two processes are in the critical region at the same time.**

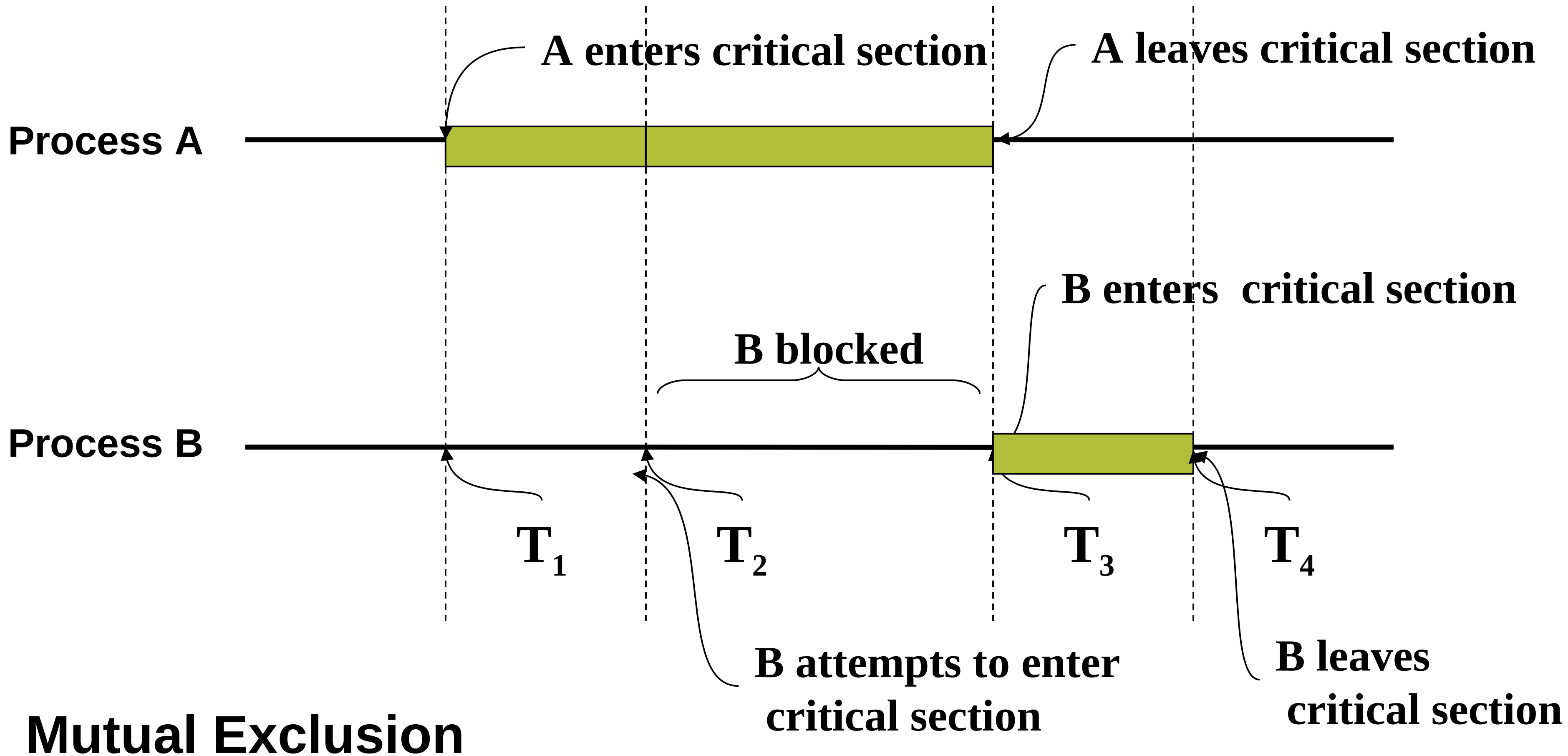
Critical Section

```
void threadRoutine()
{
    int y, x, z;
    y = 3 + 5x;
    y = Global_Var;
    y = y + 1;
    Global_Var = y;
    …
    …
}
```

```
Global_Var = 10;
ya = Global_Var;
ya = ya + 1;
Global_Var = ya;
```

```
yb = Global_Var;
yb = yb + 1;
Global_Var = yb;
```

Critical Section



A enters critical section
A leaves critical section
Process A
B enters critical section
B blocked
Process B
T1
T2
T3
T4
B attempts to enter critical section
B leaves critical section

Mutual Exclusion

At any given time, only one process is in the critical section

Critical Section

- Avoid race conditions by not allowing two processes to be in their critical sections at the same time
- We need a mechanism of mutual exclusion
- Some way of ensuring that one processes, whilst using the shared variable, does not allow another process to access that variable

Critical Section

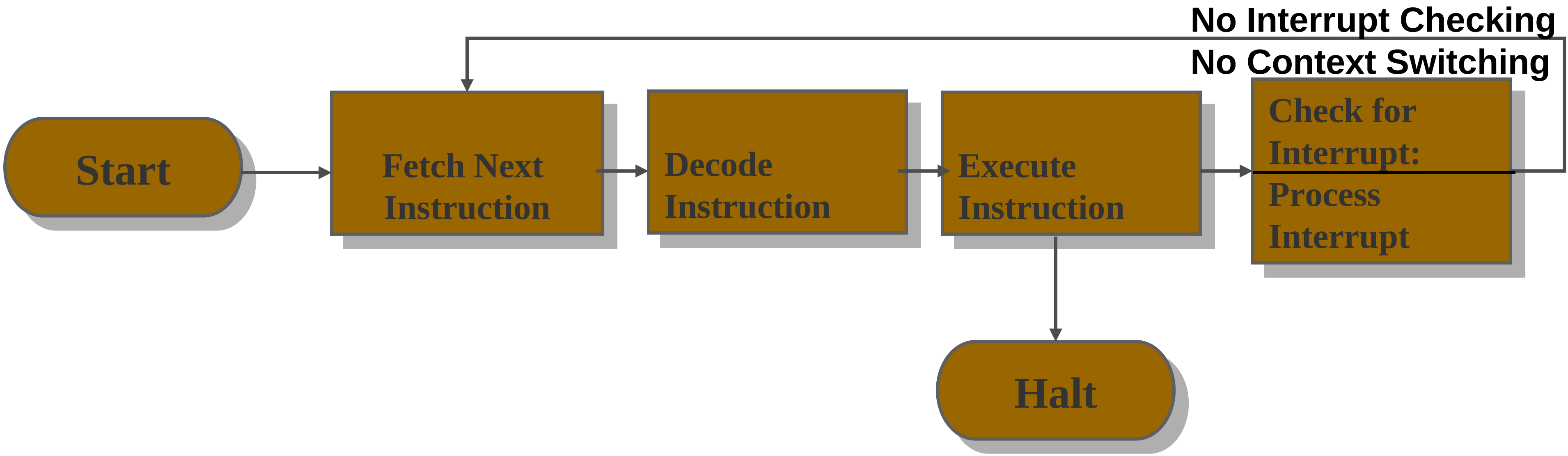
- In fact we need four conditions to hold.
  1. No two processes may be simultaneously inside their critical sections
  2. No assumptions may be made about the speed or the number of processors
  3. No process running outside its critical section may block other processes
  4. No process should have to wait forever to enter its critical section
- It is difficult to devise a method that meets all these conditions.

Implementing Mutual Exclusion

1. Disabling Interrupts
2. Lock Variables
3. Strict Alternation

Disabling Interrupts

- The problem occurred because the CPU switched to another process due to clock interrupt
- Remember the CPU cycle



Disabling Interrupts

- Solution: A Process
  - Disable interrupts before it enters its critical section
  - Enable interrupts after it leaves its critical section
- CPU will be unable to switch a process while it is in its critical section
- Guarantees that the process can use the shared variable without another process accessing it
- Disadvantage:
  - Unwise to give user processes this much power
  - The computer will not be able to service useful interrupts
  - The process may never enable interrupts, thus (effectively) crashing the system
- However, the kernel itself can disable the interrupts