

# Resource Sharing & Inter-Process Interactions

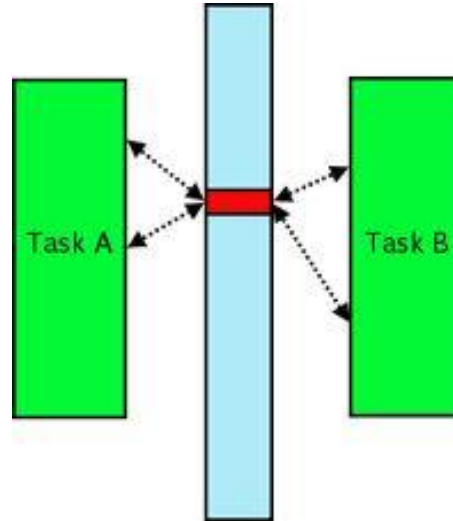
- The Issues: Deadlocks, Critical Sections, ME...
- Task Orderings (Scheduling issues and solutions)
- The Algorithmic Solutions (Races, ME...)
- The Program Level Solutions (Semaphores, Monitors)

❑ **How to achieve ME...** exclusive access to shared resources



# Why is lack of ME bad?

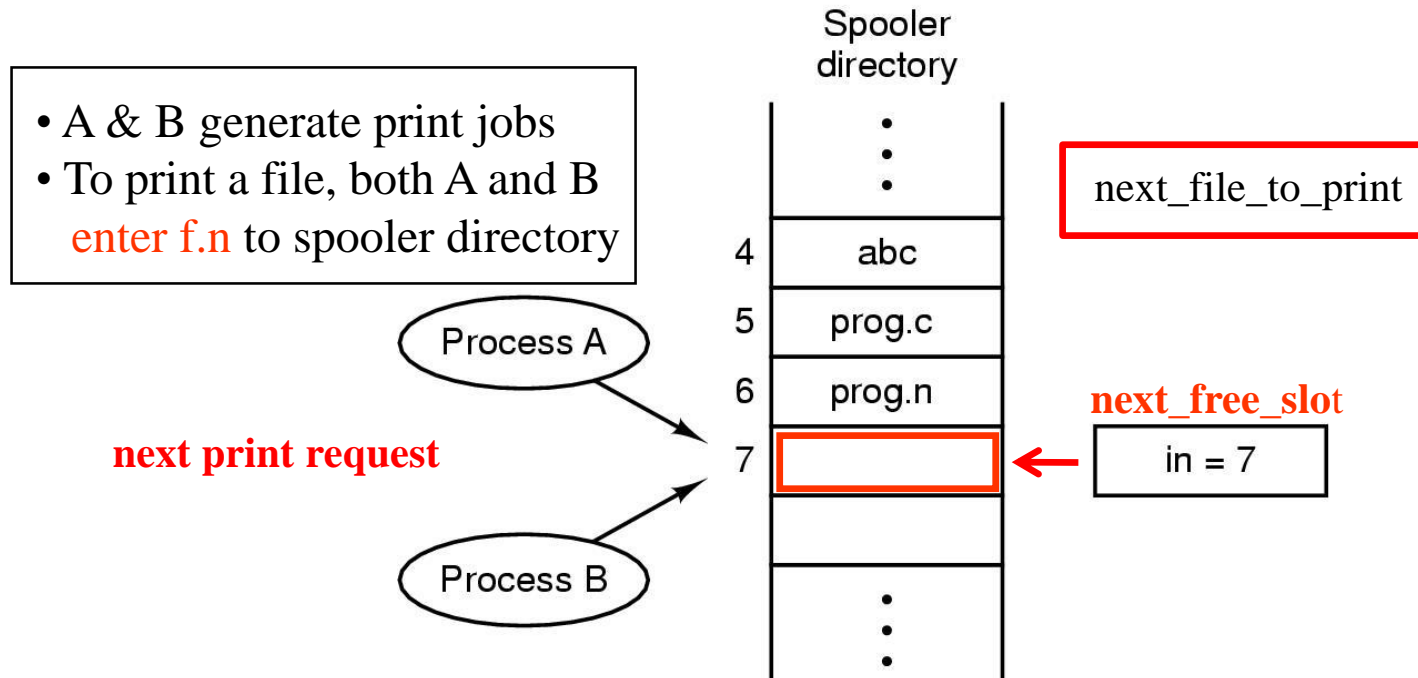
Scenario: Concurrent access to shared data by >1 processes?



- Outcome “**can**” depend solely on the order of accesses as **seen** by the resource instead of the proper request order

**Unpredictable Execution Order → Race Condition**

# Race Condition: Eg. Concurrent Accesses to Shared Memory



A (next\_free\_slot)  $\leftarrow$  7 (reads “in”)

A interrupted; B switches in (and then A comes back);

→ B should print

B  $\leftarrow$  reads in (7)

B  $\rightarrow$  f.n slot 7

in = 7+1 = 8

A  $\leftarrow$  next\_free\_slot (7)

A  $\rightarrow$  f.n slot 7

in = 7+1 = 8

A prints, B stuck ... or if A was delayed then B prints

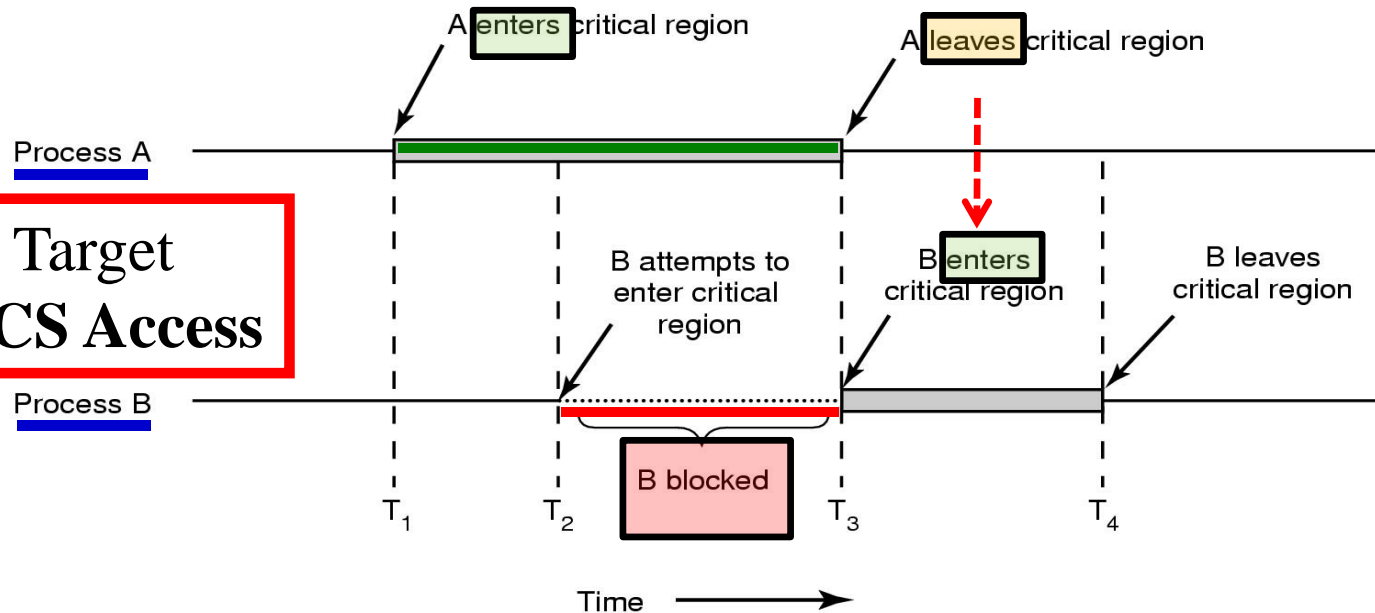
predictability?

# Mutual Exclusion & CS

Four conditions need to hold to provide mutual exclusion

1. **Unity:** No two processes are simultaneously in the CS
2. **Fairness:** No assumptions can be made about speeds or numbers of CPUs except that each process executes at non-zero speed
3. **Progress:** No process running outside its CS may block another process (from accessing the CS)
4. **Bounded Waiting:** No process must wait forever to enter its CS

**ME Solutions Target  
→ Exclusive CS Access**



# The Possibilities to Achieve ME

- Interrupts (recall Priority Scheduling)
- Locks
- TSL
- Busy-waiting
- IPC
- Semaphores

# ME Using Interrupts?

## □ Classical Interrupt Handling

- each interrupt has a specific “interrupt priority level” (ipl)
- for each executing process, the kernel maintains “current ipl”
- If `incoming_interrupt > current_ipl` → handle interrupt, else wait

## □ For ME → Do Interrupt Masking (ie. Disable Interrupts)

Enter CS

Set **Current\_ipl** ← **max\_ipl** ; disable interrupts (locked exclusive access)

enter CS

Finished CS; Restore\_ipl

Exit CS

➤ Restore\_ipl lost? Process hangs → other processes blocked?

Solution: segregated kernel\_ipl and user\_ipl levels...

➤ Distributed Sys: CPU #1 disabled, process from CPU #2 can come in...

How do we realize “ipl” across processors?

# “The Priority Inversion Case”

- **Current\_ipl  $\leftarrow$  max\_ipl ; interrupts disabled**

2 processes: H (high priority); L (low priority) Policy: L not scheduled if H is there!

Case 1: L holding max\_ipl; H waits .... Fair priority? If L hangs then ...?

Case 2:

- L in CS; H gets “ready”; H busy-waits (~ running state for H)
- BUT: L is not scheduled anytime H is running; so L is stuck in CS (cannot exit) while H loops forever (waiting for L to get out).

**Does H really have high priority with L over-riding it?**

- Is ME condition 4: Bounded Waiting holding?

# ME Using Locks for Access?

unlocked? → acquire lock

enter CS

done → release lock

do non-CS



# ME: Lock Variables

□ **flag(lock)** as global variable for access to shared section

- lock = 0 ; resource\_free
- lock = 1 ; resource\_in\_use

\* Unix flags: wanted, locked

□ **Check lock**; if free (0); set lock to 1 and then access CS

- A reads lock(0); initiates set\_to\_1
- B comes in before lock(1) finished; sees lock(0), sets lock(1)
- Both A and B have access rights → race condition ☹

➤ Happening as “**locking**” (the global var.) is not an atomic action


“**Atomic**”: All sub-actions finish for action to finish or nothing (All or Nothing)

## □ Is finer granularity & speed possible at the HW level?

- Provide “concurrency control” or “synchronizer” instructions
- Make “test & set lock” as an atomic op (ie “uninterruptible op”)

# ME with Busy Waiting: Atomic TSL

LOCK = 's global variable!

enter\_region:   
TSL REGISTER, LOCK  
CMP REGISTER, #0  
JNE enter\_region  
RET | return to caller; critical region entered

Done as a single indivisible/atomic op!!!

| copy lock to register and set lock to 1  
| was lock zero? 0 = 's unlocked  
| if it was non zero, lock was set, so loop

leave\_region:   
MOVE LOCK, #0  
RET | return to caller

| store a 0 in lock

- Entering and leaving CS using TSL (HW based – fast!!)
- Works for multiple processors as well (unlike ipl)
  - Lots of work going on in this area! Transactional Memories

# Process Alternation: ME with 'Busy Waiting'

## Turn 0

```
while (TRUE) {  
    while (turn != 0)      /* loop */ ;  
    critical_region( );  
    turn = 1;  
    noncritical_region( );  
}
```

(a)

## Turn 1

```
while (TRUE) {  
    while (turn != 1)      /* loop */ ;  
    critical_region( );  
    turn = 0;  
    noncritical_region( );  
}
```

(b)

A sees turn=0; enters CS

A exits CS, sets turn =1

.....→ B sees turn=0; busy\_waits (CPU waste)

.....→ B sees turn=1; enters CS

B finishes CS; sets turn=0; A enters CS; finishes CS quickly; sets turn=1;  
A finishes non-CS exec & wants CS again, but B is still exec non-CS; **AND** turn=1  
→ A waits (Condition 3 of ME? Process seeking CS should not be blocked  
by a process not using CS! But, no race condition given the strict alternation!)

# Peterson's 'Busy Waiting' Sans Strict Alternation

if both accesses arrive ~simultaneously?

```
#define FALSE 0
#define TRUE 1
#define N      2
```

```
int turn; shared var
int interested[N];
```

```
/* number of processes */
```

```
/* whose turn is it? */
```

```
/* all values initially 0 (FALSE) */
```

Step 1: void enter\_region(int process);

```
{
```

```
    int other;
```

```
/* process is 0 or 1 */
```

```
/* number of the other process */
```

```
    other = 1 - process;
```

```
/* the opposite of process */
```

```
    interested[process] = TRUE;
```

```
/* show that you are interested */
```

```
    turn = process;
```

```
/* set flag */ turn is "global" var; written by last requester
```

```
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
```

```
} Process 0: other = 1; interested[0] = TRUE; turn = 0; interested[1] = FALSE; P0 exits call and gets CS
```

Step 2: **Process 1: other = 0; interested[1] = TRUE; turn = 1; interested[0] = TRUE; loops**

```
void leave_region(int process)
```

```
/* process: who is leaving */
```

```
{
```

```
    interested[process] = FALSE;
```

```
/* indicate departure from critical region */
```

```
}
```

# Petersons ME (Alternate implementation)

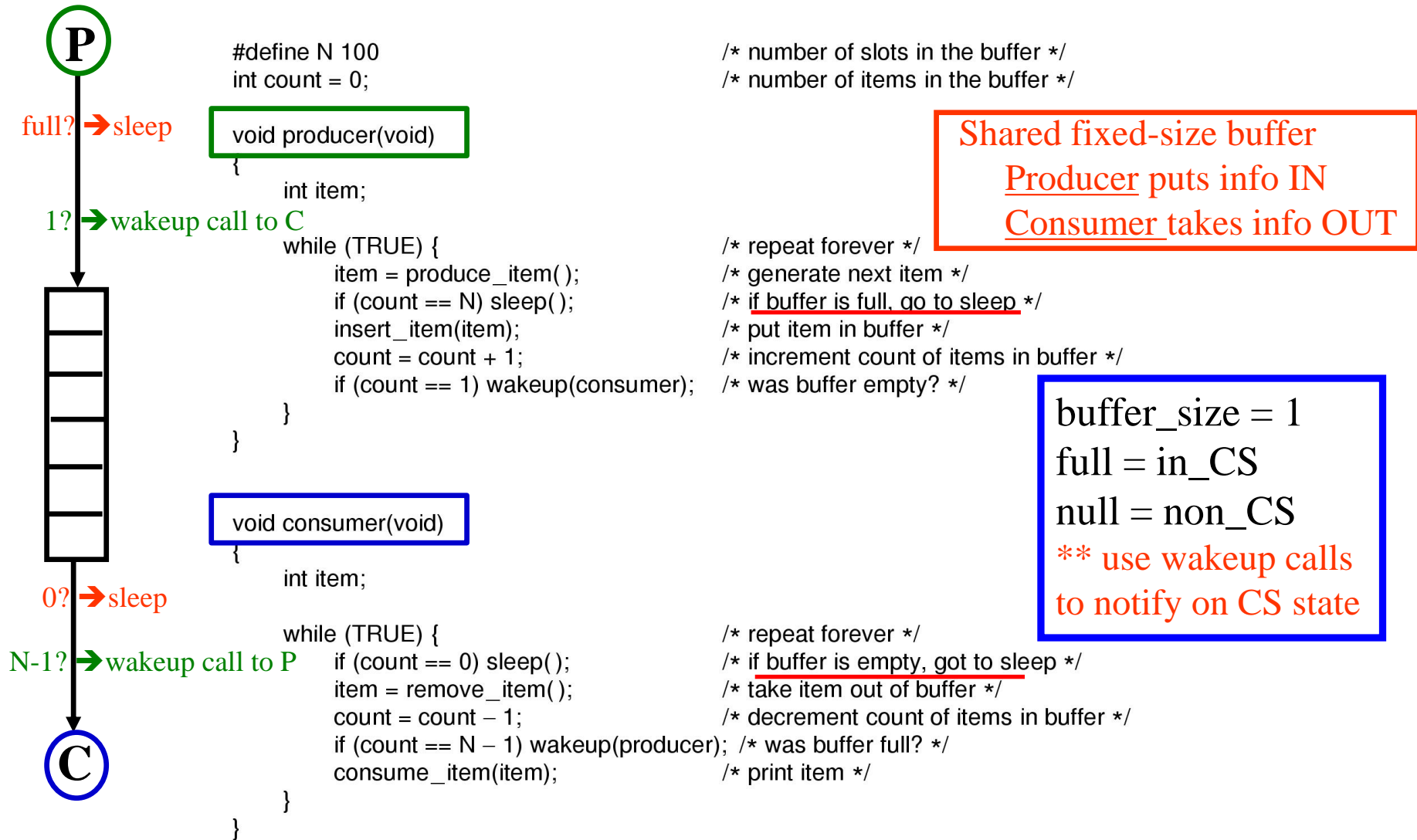
- `int turn;` “turn” to enter CS
- `boolean flag[2];` TRUE indicates ready (access) to enter CS

```
do {  
    flag[i] = TRUE;  
    turn = j ;                set access for next_CS access  
    while (flag[j] && turn == j); CS only if flag[j]=FALSE or turn = i  
CS  
    flag[i] = FALSE;  
non-CS  
} while(TRUE);
```

\* Check that conditions of ME, Fairness, Progress, Bounded-Waiting hold!

- Locks, TSL, Alternation, Petersons' ...  
All are Busy Waiting → Wasted CPU cycles
- Can we avoid busy-waiting?

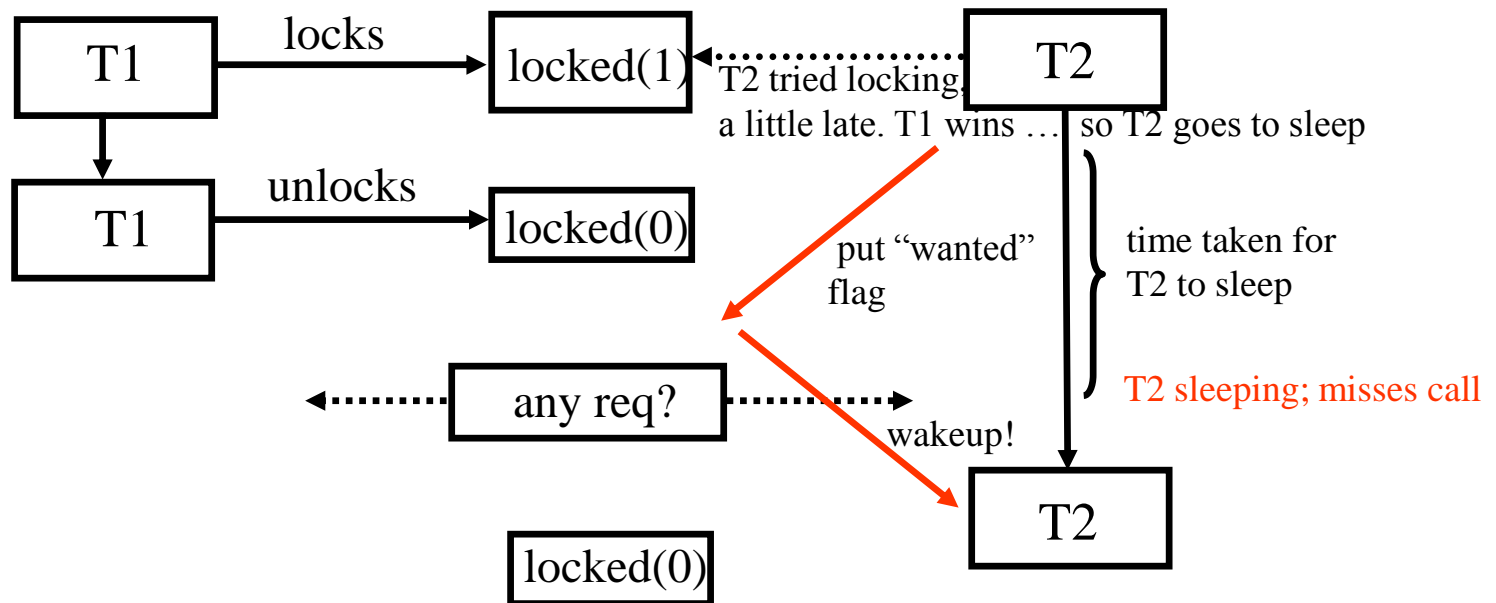
# Sleep and Wakeup (Bounded Buffer PC, IPC)



Fatal race condition?? (a) Sleep duration undefined (b) count access unconstrained



# Producer-Consumer: Sleep/Race Condition



# Do Semaphores help?

- Do SW mechanisms exist for synchronization than HW TSL?
- Semaphores (S): Integer Variables [System Calls]
  - Accessible only through 2 standard atomic operations (i.e., operation must execute indivisibly) of:
    - wait()
    - signal()

```
❑ Wait(S) {           ; sleep
    while S ≤ 0
        ; // no-op
        S--;
    }
❑ Signal(S) {         ; wakeup
    S++;
    }
```

S can be an integer resource counter;  
If S is binary, it is called a “**mutex**”

wait(0) → block if 0

wait(1) → decrement & progress (DOWN)

signal(0) → increment and unblock (UP)

# ME using Semaphores

```
do {  
    wait (mutex);           "mutex" variable initialized to 1  
        // critical section  
    signal (mutex);  
        // non critical section  
} while (TRUE);
```

wait(0) → block if 0  
wait(1) → decrement & progress (DOWN)  
signal(0) → increment and unblock (UP)