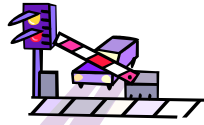


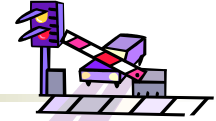
## Introduction to Synchronization



Copyright ©: Nahstedt, Angrave, Abdelzaher, Kravets, Gupta

1

## Overview



- Introduction to synchronization
  - Why do we need synchronization?
  - What is a data race condition?
  - Critical region, mutual exclusion, progress, bounded waiting



## Why Synchronization



- Processes and threads can be preempted at arbitrary times
  - Why is this a problem?
- Example:
  - What is the execution outcome of the following two threads (initially  $x=0$ )?

Thread 1:

**Read X**  
**Add 1**  
**Write X**

Thread 2:

**Read X**  
**Add 1**  
**Write X**



Copyright ©: Nahstedt, Angrave, Abdelzaher, Kravets, Gupta

4



## Producer/Consumer Problem

- Producer
  - Process "produces" information
- Consumer
  - Process "consumes" produced information

## Producer

### 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
}
```

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

5



## Incrementing

- How is `count++` implemented?

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

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

6



## Race Condition (count = 5)

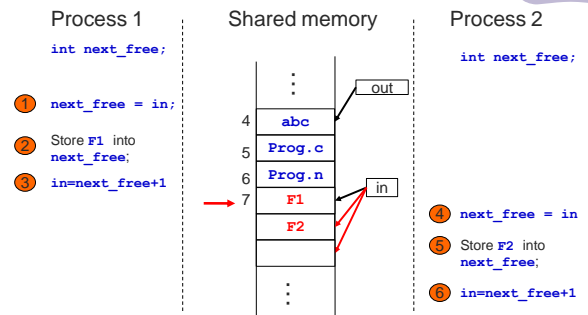
|              |                           |                 |
|--------------|---------------------------|-----------------|
| S0: producer | register1 = count         | {register1 = 5} |
| S1: producer | register1 = register1 + 1 | {register1 = 6} |
| S2: consumer | register2 = count         | {register2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

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

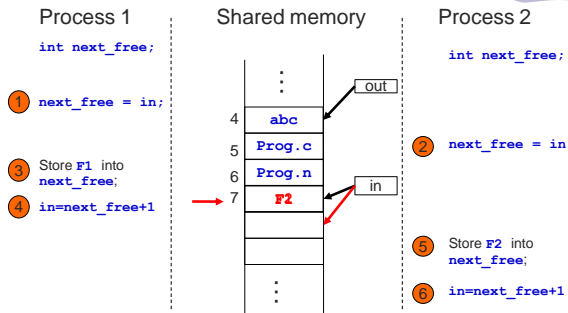
7



## Spooling Example: Correct



## Spooling Example: Problem



## Introducing: Critical Region (Critical Section)

```
Process {
  while (true) {
    Access shared variables;
    Do other work
  }
}
```

## Introducing: Critical Region (Critical Section)

```
Process {
  while (true) {
    ENTER CRITICAL SECTION
    Access shared variables;
    LEAVE CRITICAL SECTION
    Do other work
  }
}
```

## Critical Region Requirements

- Mutual Exclusion
- Progress
- Bounded Wait



No assumptions about Speed and Number of CPUs

## Critical Region Requirements

- Mutual Exclusion:
  - No other process must execute within the critical section while a process is in it
- Progress:
  - If no process is waiting in its critical section and several processes are trying to get into their critical section, then entry to the critical section cannot be postponed indefinitely



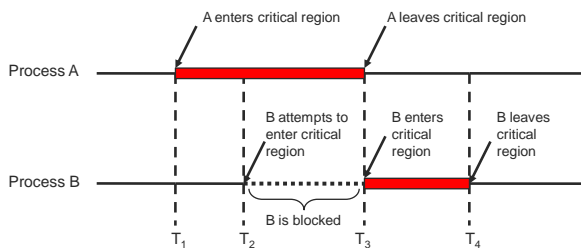
## Critical Region Requirements

- Bounded Wait:
  - A process requesting entry to a critical section should only have to wait for a bounded number of other processes to enter and leave the critical section
- Speed and Number of CPUs:
  - No assumption may be made about speeds or number of CPUs





## Critical Regions



Mutual exclusion using critical regions



## Mutual Exclusion With Busy Waiting

- Software-only candidate solutions (Two-Process Solutions)
  - Lock Variables
  - Turn Mutual Exclusion
  - Other Flag Mutual Exclusion
  - Two Flag Mutual Exclusion
  - Two Flag and Turn Mutual Exclusion
- Hardware solutions
  - Disabling Interrupts; Test-and-set; Swap (Exchange)
- Semaphores



## [ Lock Variables ]



```
...
while (lock) {
    /* spin spin spin spin */
}
lock = 1;
/* EnterCriticalSection; */
access shared variable;
/* LeaveCriticalSection; */
lock = 0;
...
```

What's the problem?



## [ Turn-based Mutual Exclusion with Strict Alternation ]



```
...
while (turn != my_process_id) {
    /* wait your turn */
}
access shared variables;
turn = other_process_id;
...
```

What's the problem?



## [ Other Flag Mutual Exclusion ]



```
int owner[2] = {false, false};
...
while (owner[other_process_id]) {
    /* wait your turn */
}
owner[my_process_id] = true;
access shared variables;
owner[my_process_id] = false;
...
```

What's the problem?



## [ Two Flag Mutual Exclusion ]



```
int owner[2] = {false, false};
...
owner[my_process_id] = true;
while (owner[other_process_id]) {
    /* wait your turn */
}
access shared variables;
owner[my_process_id] = false;
...
```

What's the problem?



## Two Flag and Turn Mutual Exclusion

```
int owner[2]={false, false};
int turn;
...
owner[my_process_id] = true;
turn = other_process_id;
while (owner[other_process_id] and
      turn == other_process_id) {
    /* wait your turn */
}
access shared variables;
owner[my_process_id] = false;
...
```



Peterson's Solution



## Discussion



- In uni-processors
  - Concurrent processes cannot be overlapped, only **interleaved**
  - A process runs until it **invokes a system call**, or is **interrupted**
  - To guarantee mutual exclusion, **hardware support** could help by allowing the **disabling of interrupts**

```
While(true) {
    /* disable interrupts */
    /* critical section */
    /* enable interrupts */
    /* remainder */
}
```
  - What's the problem with this solution?



## Discussion



- In multi-processors
  - Several processors share memory
  - Processors behave independently in a peer relationship
  - Interrupt disabling will not work
  - We need **hardware support** where access to a memory location excludes any other access to that same location
  - The hardware support is based on execution of multiple instructions **atomically** (test and set)



## Summary

- Synchronization is important for correct multi-threading programs
- Data races
- Critical regions
- Solutions to protect critical regions
  - Software-only approaches
- Next lecture:
  - Semaphores
  - Other hardware solutions

