

CS 370: OPERATING SYSTEMS

[PROCESS SYNCHRONIZATION]

Instructor: Louis-Noel Pouchet
Spring 2024

Computer Science
Colorado State University

** Lecture slides created by: SHRIDEEP PALICKARA

Topics covered in the lecture

- Critical section
- Critical section problem
- Peterson's solution
- Hardware assists

*A cooperating process can affect or be affected by
other processes within the system*

PROCESS SYNCHRONIZATION

Process synchronization

- How can processes **pass information** to one another?
- Make sure two or more processes **do not get in each other's way**
 - ▣ E.g., 2 processes in an airline reservation system, each trying to grab the last seat for a different passenger
- Ensure proper **sequencing** when dependencies are present

Applicability to threads

- Passing information between threads is easy
 - ▣ They share the same address space of the parent process
- Other two aspects of process synchronization are applicable to threads
 - ▣ Keeping out of each other's hair
 - ▣ Proper sequencing

A look at the producer consumer problem

```
while (true) {  
    while (counter == BUFFER_SIZE) {  
        ; /*do nothing */  
    }  
    buffer[in] = nextProduced  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

Producer

```
while (true) {  
    while (counter == 0) {  
        ; /*do nothing */  
    }  
    nextConsumed = buffer[out]  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
}
```

Consumer

Implementation of ++/-- in machine language

counter++

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

counter--

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

Lower-level statements may be interleaved in any order

Producer execute: `register1 = counter`

Producer execute: `register1 = register1 + 1`

Producer execute: `counter = register1`

Consumer execute: `register2 = counter`

Consumer execute: `register2 = register2 - 1`

Consumer execute: `counter = register2`

Lower-level statements may be interleaved in any order

Producer execute: `register1 = counter`

Consumer execute: `register2 = counter`

Producer execute: `register1 = register1 + 1`

Consumer execute: `register2 = register2 - 1`

Producer execute: `counter = register1`

Consumer execute: `counter = register2`

The **order** of statements *within* each high-level statement is **preserved**

Lower-level statements may be interleaved in any order (counter = 5)

<i>Producer</i> execute: register1 = counter	{register1 = 5}
<i>Producer</i> execute: register1 = register1 + 1	{register1 = 6}
<i>Consumer</i> execute: register2 = counter	{register2 = 5}
<i>Consumer</i> execute: register2 = register2 - 1	{register2 = 4}
<i>Producer</i> execute: counter = register1	{counter = 6}
<i>Consumer</i> execute: counter = register2	{counter = 4}

Counter has **incorrect** state of 4

Lower-level statements may be interleaved in any order (counter = 5)

<i>Producer</i> execute: register1 = counter	{register1 = 5}
<i>Producer</i> execute: register1 = register1 + 1	{register1 = 6}
<i>Consumer</i> execute: register2 = counter	{register2 = 5}
<i>Consumer</i> execute: register2 = register2 - 1	{register2 = 4}
<i>Consumer</i> execute: counter = register2	{counter = 4}
<i>Producer</i> execute: counter = register1	{counter = 6}

Counter has **incorrect** state of 6

Race condition

- Several processes access and manipulate data **concurrently**
- **Outcome** of execution *depends* on
 - ▣ Particular **order** in which accesses takes place
- Debugging programs with race conditions?
 - ▣ Painful!
 - ▣ Program runs fine most of the time, but once in a rare while something weird and unexpected happens

Race condition: Example

[1 / 3]

- When process wants to print file, adds file to a special **spooler directory**
- Printer daemon periodically checks to see if there are files to be printed
 - ▣ If there are, print them
- In our example, spooler directory has a large number of slots
- Two variables
 - ▣ `in`: Next free slot in directory
 - ▣ `out`: Next file to be printed

Race condition: Example

[2/3]

- In jurisdictions where *Murphy's Law* hold ...
- Process A reads `in`, and stores the value 7, in local variable `next_free_slot`
- Context switch occurs
- Process B also reads `in`, and stores the value 7, in local variable `next_free_slot`
 - ▣ Stores name of the file in slot 7
- Process A context switches again, and stores the name of the file it wants to print in slot 7

Race condition: Example

[3/3]

- Spooler directory is internally consistent
- But process B will never receive any output
 - ▣ User B loiters around printer room for years, wistfully hoping for an output that never comes ...

The kernel is subject to several possible race conditions

- E.g.: Kernel maintains list of all open files
 - ▣ 2 processes open files simultaneously
 - ▣ Separate updates to kernel list may result in a race condition
- Other kernel data structures
 - ▣ Memory allocation
 - ▣ Process lists
 - ▣ Interrupt handling

Segment of code where processes change common variables

CRITICAL SECTION

Critical-Section

- System of n processes $\{P_0, P_1, \dots, P_{n-1}\}$
- Each process has a segment of code (**critical section**) where it:
 - **Changes common variables**, updates a table, etc
- No two processes can execute in their critical sections at the same time

The Critical-Section problem


- Design a **protocol** that processes can use to cooperate
- Each process must **request permission** to enter its critical section
 - ▣ The **entry** section

General structure of a participating process

```
do {
```

entry section


Request permission
to enter



critical section

exit section

Housekeeping to let
other processes enter



remainder section

```
} while (TRUE);
```

REQUIREMENTS FOR A SOLUTION TO THE CRITICAL SECTION PROBLEM

Requirements for a solution to the critical section problem

- ① Mutual exclusion
- ② Progress
- ③ Bounded wait

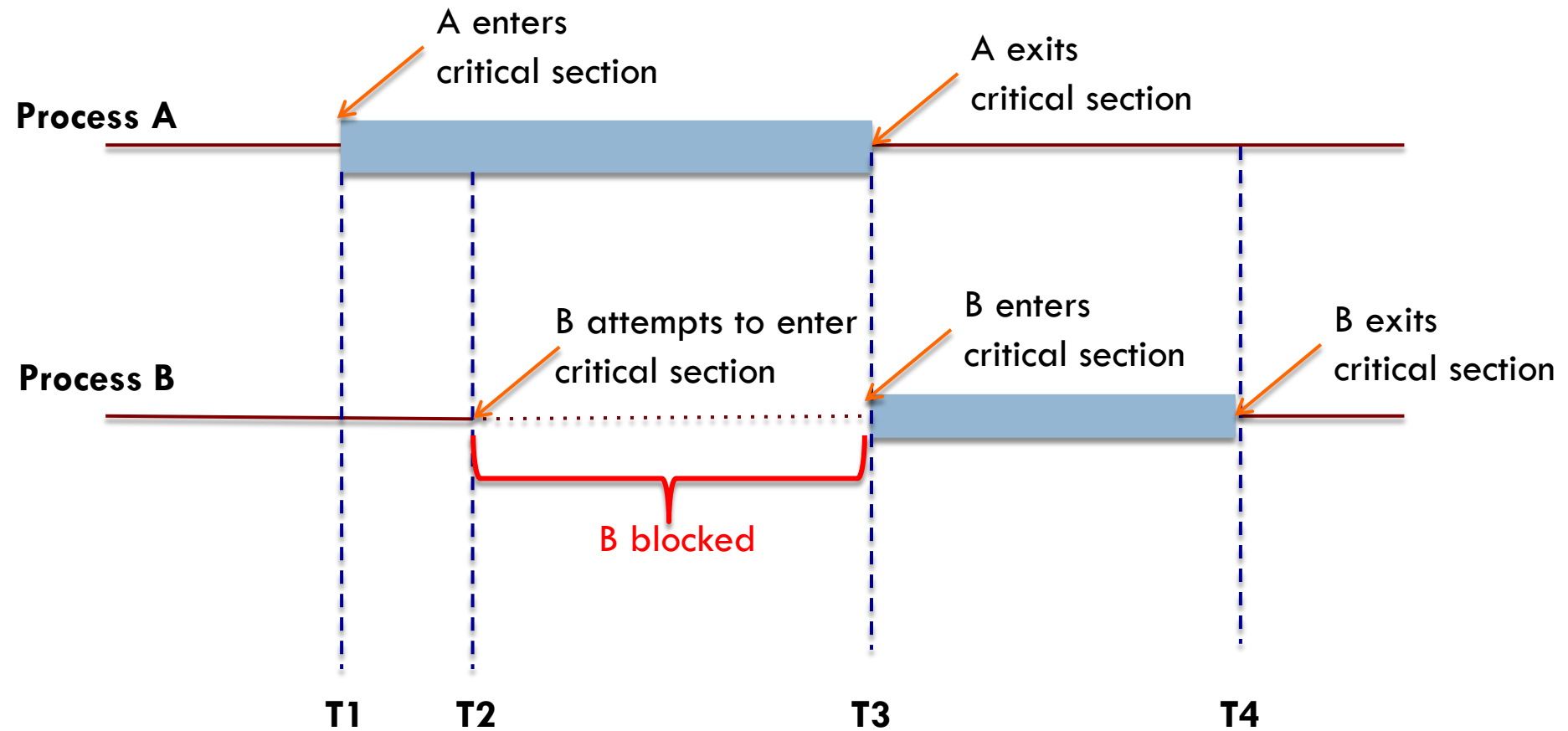
□ PROCESS SPEED

- ▣ Each process operates at *non-zero* speed
- ▣ Make no assumption about the *relative speed* of the n processes

Mutual Exclusion

- Only **one** process can execute in its critical section
- When a process executes in its critical section
 - **No other process** is allowed to execute in *its* critical section

Mutual Exclusion: Depiction



Progress

- **{C1}** If **No** process is executing in its critical section, and ...
- **{C2}** **Some** processes wish to enter their critical sections

- **Decision** on who gets to enter the critical section
 - ▣ Is made by processes that are NOT executing in their remainder section
 - ▣ Selection **cannot be postponed indefinitely**

Bounded waiting

- *After* a process has made a **request** to enter its critical section
 - ▣ AND *before* this request is granted
- **Limit number** of times other processes are allowed to enter their critical sections

Approaches to handling critical sections in the OS

- Nonpreemptive kernel
 - ▣ If a process runs in kernel mode: no preemption
 - ▣ **Free** from race conditions on kernel data structures
- Preemptive kernels
 - ▣ Must ensure shared kernel data is free from race conditions
 - ▣ Difficult on SMP (Symmetric Multi Processor) architectures
 - 2 processes may run simultaneously on different processors

Kernels: Why preempt?

- Suitable for real-time
 - ▣ A real-time process may preempt a kernel process
- More **responsive**
 - ▣ *Less risk* that kernel mode process will run arbitrarily long

Software based solution

PETERSON'S SOLUTION

Peterson's Solution

- **Software solution** to the critical section problem
 - Restricted to two processes
- No guarantees on modern architectures
 - Machine language instructions such as `load` and `store` implemented differently
- Good algorithmic description
 - Shows how to address the 3 requirements

Peterson's Solution: The components

- Restricted to two processes in this example (but generalizable to n)
 - P_i and P_j
- **Share** two data items
 - `int turn`
 - Indicates whose *turn* it is to enter the critical section
 - `boolean flag[2]`
 - Whether process *is ready* to enter the critical section

Peterson's solution: Structure of process P_i

do {

```
flag[0] = TRUE;  
turn = 1;  
while (flag[0] && turn==1) {;}
```

critical section

```
flag[0] = FALSE;
```

remainder section

} while (TRUE);

Peterson's solution: Structure of process P_i

do {

```
flag[1] = TRUE;  
turn = 0;  
while (flag[0] && turn==0) {;}
```

critical section

```
flag[0] = FALSE;
```

remainder section

} while (TRUE);

Peterson's solution: Mutual exclusion

```
while (flag[j] && turn==j) {;
```

- P_i enters critical section only if
 `flag[j] == false OR turn == i`
- If both processes try to execute in critical section at the same time
 - `flag[0] == flag[1] == true`
 - **But** `turn` can be 0 or 1, not BOTH
- If P_j entered critical section
 - `flag[j] == true AND turn == j`
 - Will persist as long as P_j is in the critical section

Peterson's Solution:

Progress and Bounded wait

- P_i can be stuck only if `flag[j]==true` AND `turn==j`
 - ▣ If P_j is *not ready*: `flag[j]== false`, and P_i can enter
 - ▣ Once P_j *exits*: it resets `flag[j]` to false
- If P_j resets `flag[j]` to true
 - ▣ Must set `turn = i;`
- P_i **will enter** critical section (*progress*) after at most one entry by P_j (*bounded wait*)

SYNCHRONIZATION HARDWARE

Solving the critical section problem using locks

```
do {
```

```
    acquire lock
```

```
    critical section
```

```
    release lock
```

```
    remainder section
```

```
} while (TRUE);
```

Possible assists for solving critical section problem (1/2)

- Uniprocessor environment
 - ▣ **Prevent interrupts** from occurring when shared variable is being modified
 - *No unexpected modifications!*
- Multiprocessor environment
 - ▣ Disabling interrupts is *time consuming*
 - Message passed to ALL processors

Possible assists for solving critical section problem (2/2)

- Special **atomic** hardware instructions
 - ▣ Swap content of two words
 - ▣ Modify word

Swap ()

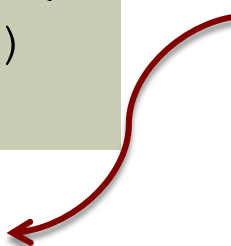
```
void Swap(boolean *a, boolean *b ) {  
  
    boolean temp = *a;  
    *a = *b;  
    *b = temp;  
}
```


Swap: Shared variable LOCK is initialized to false

```
do {
```

```
    key = TRUE;
    while (key == TRUE) {
        Swap(&lock, &key)
    }
```

Cannot enter critical section
UNLESS lock == FALSE



```
critical section
```

```
    lock = FALSE;
```

```
remainder section
```

lock is a SHARED variable
key is a LOCAL variable

```
} while (TRUE);
```

TestAndSet ()

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

Sets target **to** true and returns old value of target

TestAndSet: Shared boolean variable lock initialized to false

```
do {
```

```
    while (TestAndSet(&lock)) {;
```


```
    critical section
```

```
    lock = FALSE;
```

```
    remainder section
```

```
} while (TRUE);
```

To break out:
Return value of TestAndSet
should be FALSE



**If two TestAndSet() are executed
simultaneously, they will be executed
sequentially in some arbitrary order**

Entering and leaving critical regions using TestAndSet and Swap (Exchange)

```
enter_region:
    TSL REGISTER, LOCK
    CMP REGISTER, #0
    JNE enter_region
    RET
```

```
leave_region:
    MOVE LOCK, #0
    RET
```

```
enter_region:
    MOVE REGISTER, #1
    XCHNG REGISTER, LOCK
    CMP REGISTER, #0
    JNE enter_region
    RET
```

```
leave_region:
    MOVE LOCK, #0
    RET
```

All Intel x86 CPUs have the XCHG instruction for low-level synchronization

The contents of this slide set are based on the following references

- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330. [Chapter 5]*
- *Andrew S Tanenbaum and Herbert Bos. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620. [Chapter 2]*