

CS 370: OPERATING SYSTEMS

[PROCESS SYNCHRONIZATION]

Computer Science
Colorado State University

Instructor: Louis-Noel Pouchet
Spring 2024

** Lecture slides created by: SHRIDEEP PALICKARA

Topics covered in the lecture

- Synchronization hardware
- Using `TestAndSet` to satisfy critical section requirements
- Semaphores
- Classical process synchronization problems
- Midterm

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)
    }
```

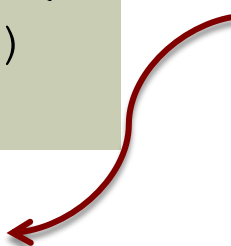
```
    critical section
```

```
    lock = FALSE;
```

```
    remainder section
```

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

Cannot enter critical section
UNLESS lock == FALSE



lock is a SHARED variable
key is a LOCAL variable

Note: If two Swap() are executed
simultaneously, they will be executed
sequentially in some arbitrary order

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

USING TEST-AND-SET TO SATISFY CRITICAL SECTION REQUIREMENTS

Using TestAndSet to satisfy all critical section requirements

- N processes
- Data structures initialized to FALSE
 - `boolean waiting[n];`
 - `boolean lock;`

These data structures are maintained in shared memory.

The entry section for process i

```
waiting[i] = TRUE;
key = TRUE;

while (waiting[i] && key) {
    key = TestAndSet(&lock);
}

waiting[i] = FALSE;
```

**First process to execute TestAndSet will find `key == false` ;
ENTER critical section
EVERYONE else must wait**

The exit section: Part I

Finding a suitable waiting process

If a process is not waiting
move to the next one

```
j = (i + 1) % n;
```

```
while ( (j != i) && !waiting[j] ) {  
    j = (j+1) % n  
}
```

Will break out at $j == i$ if
there are no waiting
processes

If a process is
waiting:
break out of loop

The exit section: Part II

Finding a suitable waiting process

Could NOT find a suitable waiting process

```
if (j==i) {  
    lock = FALSE;  
} else {  
    waiting[j] = FALSE;  
}
```

Found a suitable waiting process

Mutual exclusion

- The variable `waiting[i]` can become `false` **ONLY** if another process leaves its critical section
 - **Only one** `waiting[i]` is set to `FALSE`

Progress

- A process exiting the critical section
 - ① Sets `lock` to `FALSE`
 - OR
 - ② `waiting[j]` to `FALSE`
- Allows a process that is *waiting* to **proceed**

Bounded waiting requirement

```
j = (i + 1) % n;
```

```
while ( (j != i) && !waiting[j] ) {  
    j = (j+1) % n  
}
```

- **Scans** `waiting[]` in the **cyclic** ordering
($i+1, i+2, \dots, n, 0, \dots, i-1$)
- ANY waiting process trying to enter critical section will do so in **($n-1$)** turns

SEMAPHORES

Semaphores

- Semaphore **S** is an integer variable
- Once *initialized*, accessed through **atomic** operations
 - `wait()`
 - `signal()`

Modifications to the integer value of semaphore execute indivisibly

```
wait(S) {  
    while (S<=0) {  
        ; //no operation  
    }  
    S--;  
}
```

```
signal(S) {  
    S++;  
}
```

Types of semaphores

- Binary semaphores
 - ▣ The value of S can be 0 or 1
 - Also known as **mutex locks**
- Counting semaphores
 - ▣ Value of S can range over an **unrestricted domain**

Using the Binary semaphore to deal with the critical section problem

mutex is initialized to 1

do {

wait(**mutex**) ;

critical section

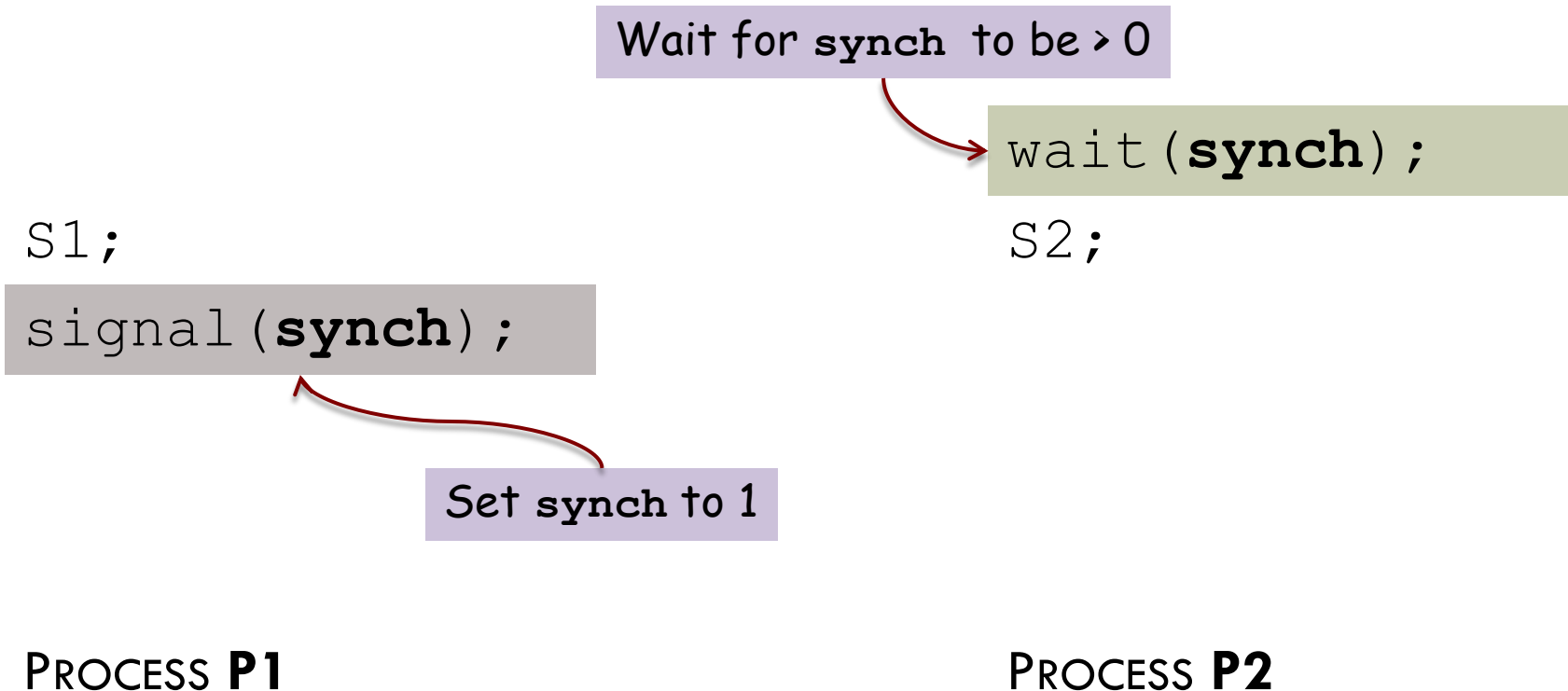
signal(**mutex**) ;

remainder section

} while (TRUE) ;

Suppose we require S2 to execute only after S1 has executed


Semaphore **synch** is initialized to 0



The counting semaphore

- Controls access to a **finite** set of resource instances
- INITIALIZED to the number of resources available
- Resource Usage
 - `wait()` : To **use** a resource
 - `signal()` : To **release** a resource
- When all resources are being used: $S == 0$
 - Block until $S > 0$ to use the resource

Problems with the basic semaphore implementation

- **{C1}** If there is a process in the critical section
 - **{C2}** If another process tries to enter its critical section
 - ▣ Must loop continuously in entry code
 - ▣ **Busy waiting!**
 - Some other process could have used this more productively!
 - ▣ Sometimes these locks are called **spinlocks**
 - One advantage: No context switch needed when process must wait on a lock
- 

Overcoming the need to busy wait

- During `wait` if `S==0`
 - ▣ Instead of *busy waiting*, the process **blocks** itself
 - ▣ Place process in waiting queue for `S`
 - ▣ **Process state** switched to **waiting**
 - ▣ CPU scheduler picks *another* process to execute
- **Restart** process when another process does `signal`
 - ▣ Restarted using `wakeup()`
 - ▣ Changes process state from *waiting* to **ready**

Defining the semaphore

```
typedef struct {  
    int value;  
    struct process *list;  
} semaphore;
```



list of processes

The `wait()` operation to eliminate busy waiting

```
wait(semaphore *S) {  
    S->value--;
```

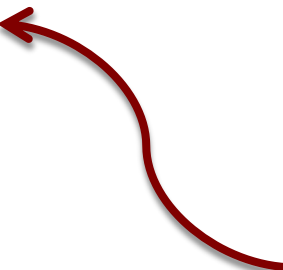
If $\text{value} < 0$
 $\text{abs}(\text{value})$ is the number
of waiting processes

```
    if (S->value < 0) {  
        add process to S->sleeping_list;  
        block();  
    }  
}
```

`block()` suspends the
process that invokes it

The `signal()` operation to eliminate busy waiting

```
signal(semaphore *S) {  
    S->value++;  
  
    if (S->value <= 0) {  
        remove a process P from S->sleeping_list;  
        wakeup(P);  
    }  
}
```



**wakeup(P) resumes the
execution of process P**

Deadlocks and Starvation: Implementation of semaphore with a waiting queue

PROCESS P0

```
wait(S) ;  
wait(Q) ;
```

```
signal(S) ;  
signal(Q) ;
```

PROCESS P1

```
wait(Q) ;  
wait(S) ;
```


```
signal(Q) ;  
signal(S) ;
```

Say: **P0** executes `wait(S)` and *then* **P1** executes `wait(Q)`

P0 must wait till **P1** executes `signal(Q)`

P1 must wait till **P0** executes `signal(S)`

Cannot be
executed
so deadlock



Semaphores and atomic operations

- Once a semaphore action has started
 - ▣ **No other process** can access the semaphore UNTIL
 - Operation has *completed* or *process has blocked*
- Atomic operations
 - ▣ Group of related operations
 - ▣ Performed without interruptions
 - Or not at all

PRIORITY INVERSION

Priority inversion

- Processes **L**, **M**, **H** (priority of $L < M < H$)
- Process **H** requires
 - ▣ Resource **R** being accessed by process **L**
 - ▣ Typically, **H** will wait for **L** to finish resource use
- **M** becomes runnable and preempts **L**
 - ▣ Process (**M**) with lower priority affects *how long* process **H** has to wait for **L** to release **R**

Priority inheritance protocol

- Process accessing resource needed by higher priority process
 - ▣ **Inherits** higher priority till it finishes resource use
 - ▣ Once done, process **reverts** to lower priority

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. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620. [Chapter 2]*