CS 370: Operating Systems
[Process Synchronization]

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- Other real-world examples?
- Critical Sections
  - Parametrized
  - How can wait be bounded when each process is doing its own thing?
  - Entry/Exit section protocol: Responsibility of the programmer?
- What happens in the critical section? Access to shared memory?
- Atomic
- How do hardware-assisted locks deal with priority?

Topics covered in the lecture

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

`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

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;
```

Will break out only if `waiting[i]==FALSE` OR `key==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, …n, 0, …, 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**

Wait for **synch** to be > 0

```
wait(synch);
S2;
```

```
S1;
signal(synch);
```

Set **synch** to 1

PROCESS **P1** | PROCESS **P2**

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 **block**s 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 (S→value <0) {
      add process to S→list;
      block();
    }

}
```

If value < 0
abs(value) is the number of waiting processes

`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→list;
      wakeup(P);
    }

}
```

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

Deadlocks and Starvation: Implementation of semaphore with a waiting queue

Process P0	Process P1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`signal(S);`	`signal(Q);`
`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

CLASSIC PROBLEMS OF SYNCHRONIZATION

The bounded buffer problem

- Binary semaphore (`mutex`)
  - Provides mutual exclusion for accesses to buffer pool
  - Initialized to 1
- Counting semaphores
  - `empty`: Number of empty slots available to produce
    - Initialized to ***n***
  - `full`: Number of filled slots available to consume
    - Initialized to **0**

Some other things to bear in mind

- Producer and consumer must be **ready** before they **attempt to enter** critical section
- Producer readiness?
  - When a slot is available **to add** produced item
    - `wait(empty)`: `empty` is initialized to **n**
- Consumer readiness?
  - When a **producer has added** new item to the buffer
    - `wait(full)` : `full` initialized to **0**

The Producer

```
do {
    produce item nextp
    wait(empty);
    wait(mutex);

    add nextp to buffer

    signal(mutex);
    signal(full);

    remainder section

} while (TRUE);
```

- `wait` till slot available
- Only producer **OR** consumer can be in critical section
- Allow producer **OR** consumer to (re)enter critical section
- `signal` consumer that a slot is available

The Consumer

```
do {
    wait(full);
    wait(mutex);

    remove item from buffer
        (nextc)

    signal(mutex);
    signal(empty);

    consume nextc

} while (TRUE);
```

- `wait` till slot available for consumption
- Only producer **OR** consumer can be in critical section
- Allow producer **OR** consumer to (re)enter critical section
- `signal` producer that a slot is available to add

THE READERS-WRITERS PROBLEM

The Readers-Writers problem

- A database is **shared** among several concurrent processes
- Two types of processes
  - Readers
  - Writers

Readers-Writers: Potential for adverse effects

- If *two readers* access shared data simultaneously?
  - No problems
- If a *writer and some other reader* (or writer) access shared data simultaneously?
  - Chaos

Writers must have exclusive access to shared database while writing

- FIRST readers-writers problem:
  - No reader should wait for other readers to finish; simply because a writer is waiting
    - Writers may starve
- SECOND readers-writers problem:
  - If a writer is ready it performs its write ASAP
    - Readers may starve

Solution to the FIRST readers-writers problem

- Variable `int readcount`
  - Tracks how many readers are reading object
- Semaphore **`mutex`** {1}
  - Ensure mutual exclusion when `readcount` is accessed
- Semaphore **`wrt`** {1}
  1. Mutual exclusion for the writers
  2. First (last) reader that enters (exits) critical section
     - Not used by readers, when **other** readers **are in** their critical section

The Writer: When a writer signals either a waiting writer or the readers resume

```
do {
     wait(wrt);

     writing is performed

     signal(wrt);

} while (TRUE);
```

When:
writer in critical section
and if n readers waiting

1 reader is queued on **`wrt`**
(n-1) readers queued on **`mutex`**

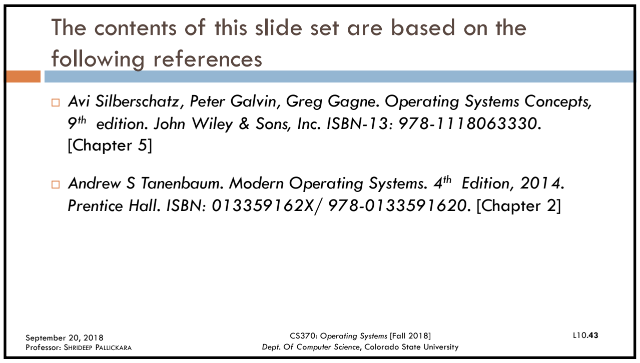
The Reader process

```
do {
     wait(mutex);
     readcount++;
     if (readcount ==1) {
       wait(wrt);
     }
     signal(mutex);

     reading is performed

     wait(mutex);
     readcount--;
     if (readcount ==0) {
       signal(wrt);
     }
     signal(mutex);
} while (TRUE);
```

`mutex` for mutual exclusion to `readcount`

When:
writer in critical section
and if n readers waiting

1 is queued on **`wrt`**
(n-1) queued on **`mutex`**


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]
September 20, 2018
Professor: Shrideep Pallickara
CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University
L10.43