OPERATING SYSTEMS

Mutex Locks and Semaphores

Chandravva Hebbi
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

Mutex Locks

- Previous solutions i.e hardware and software solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **acquire()** a lock then **release()** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
- This lock therefore called a **spinlock**
- It is problem in real time systems
- Busy waiting wastes CPU cycles

Advantages of spinlocks

- no context switch is required when a process must wait on a lock.
- context switch may take considerable time.
- When locks are expected to be held for short times, spinlocks are useful
- These are employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor.

Solution to Critical-section Problem Using Locks

```
while (TRUE) {
    acquire lock
        critical section
    release lock
remainder section
}
```

- **Implementation of acquire and release**

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}
release() {
    available = true;
}
```

acquire() and release()

- Solution to a critical section problem using mutex
- **do {**

acquire lock

critical section

release lock

remainder section

} while (true);

OPERATING SYSTEMS

Scenario 1

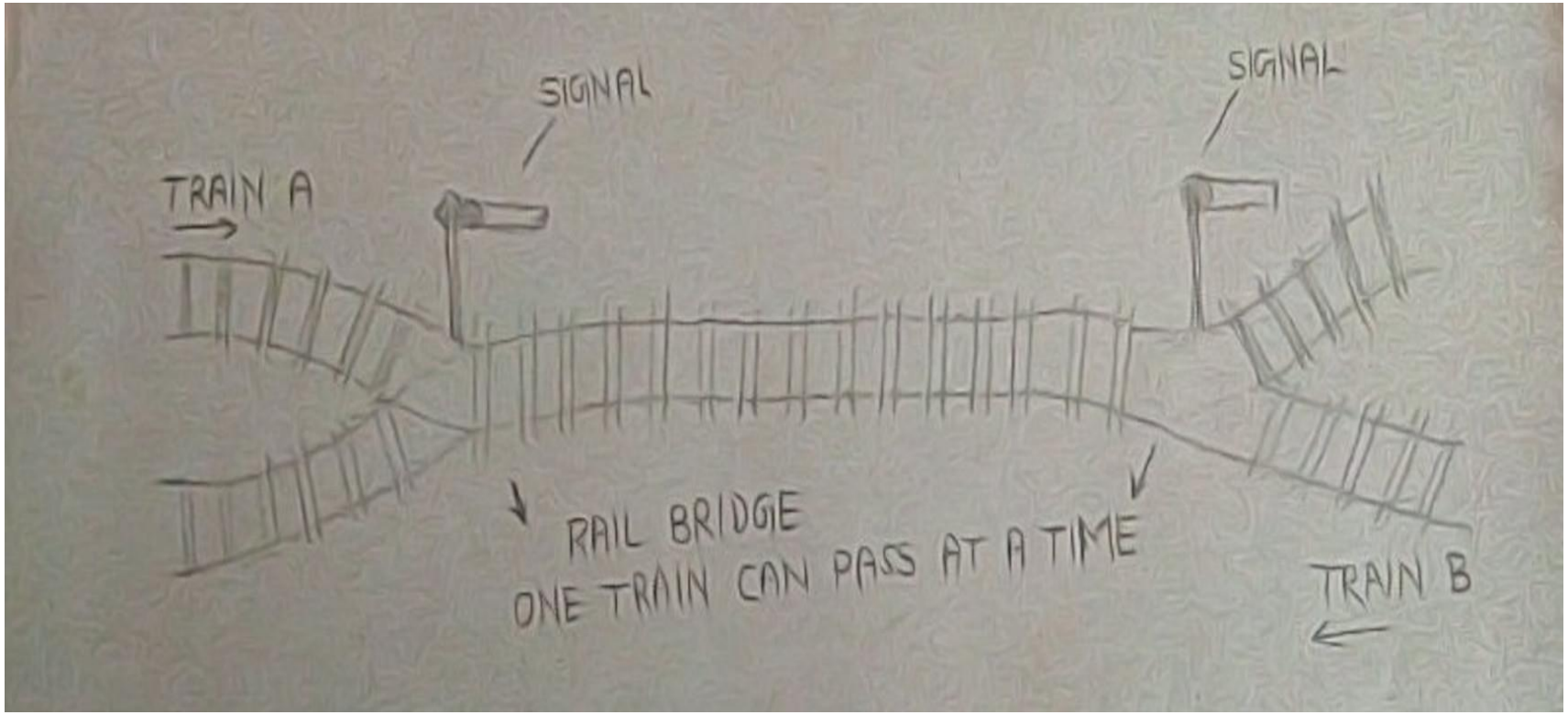


Image src: https://www.linkedin.com/pulse/concurrency-semaphore-real-life-kunal-saxena/

Scenario 2

- Consider a library of an university with 10 rooms
- At a time one room can be used by only one student by informing the front desk for reading.
- Once he completes reading, he has to inform the front desk.
- Person at front desk knows how many rooms are available for use and how many are occupied, how may of them are waiting.
- Once the room is vacant, who will get the chance to occupy the room?

Idea Src:https://en.wikipedia.org/wiki/Semaphore_(programming)

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **wait()** and **signal()**
    - Originally called **P()** and **V()**
- Definition of the **wait() operation**

```
wait(S) {
   while (S <= 0)
     ; // busy wait
   S--;}
```

- Definition of the **signal() operation**

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

Semaphore

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**synch**" initialized to 0

```
P1:
  S1;
  signal(synch);
P2:
  wait(synch);
  S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore

Semaphore Implementation

- Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **wait** and **signal** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
- **typedef struct{**

int value;

struct process *list;

} semaphore;

Semaphore Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
  S->value--;
  if (S->value < 0) {
    add this process to S->list;
    block();
  }
}
signal(semaphore *S) {
  S->value++;
  if (S->value <= 0) {
    remove a process P from S->list;
    wakeup(P);
  } }
```

Process P0

wait()// entry section

// critical section

Signal()// exit section

Remainder section

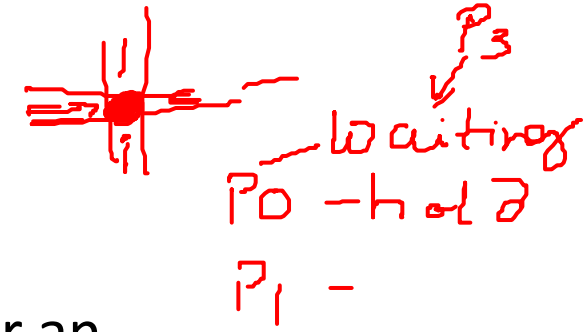
S = 4

S - - 4

OPERATING SYSTEMS

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)



- Deadlock and starvation are possible.
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

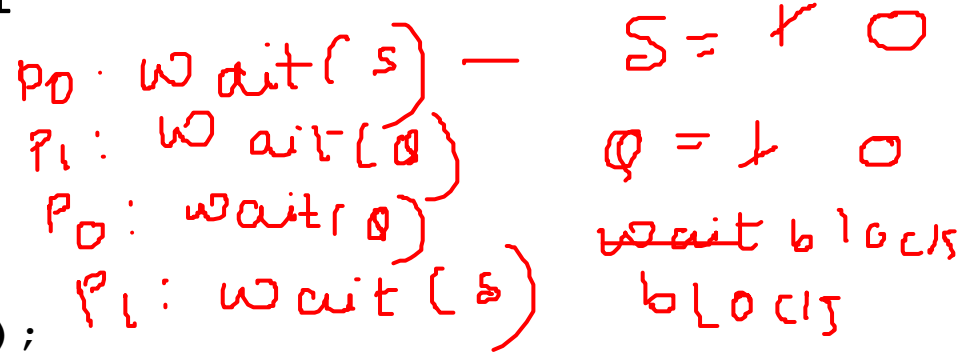
- Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0

```
wait(S);
wait(Q);
 ...
signal(S);
signal(Q);
```

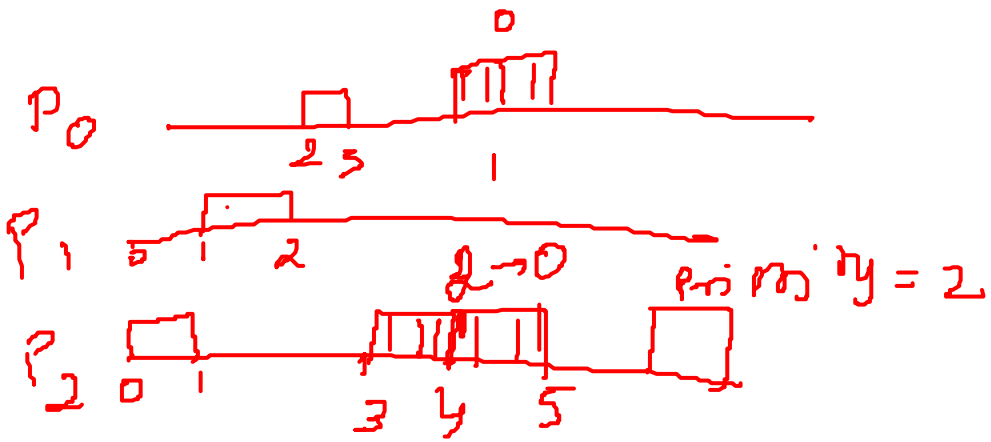
P_1

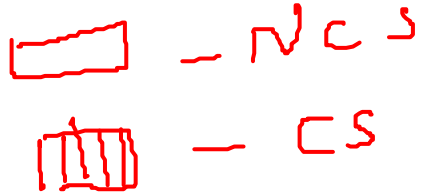
```
wait(Q);
wait(S);
 ...
signal(Q);
signal(S);
```



Problems with Semaphores (Cont.)

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**





Priority Inheritance Protocol

- When several tasks are waiting for the same critical resource, the task which is currently holding this critical resource is given the highest priority among all the tasks which are waiting for the same critical resource.
- Now after the lower priority task having the critical resource is given the highest priority then the intermediate priority tasks can not preempt this task. This helps in avoiding priority inversion.
- When the task which is given the highest priority among all tasks, finishes the job and releases the critical resource then it gets back to its original priority value (which may be less or equal).
- It allows the different priority tasks to share the critical resources.

Priority Inheritance Protocol

- Consider the scenario with three processes **P1, P2**, and **P3**.
  - **P1** has the highest priority, **P2** the next highest, and **P3** the lowest.
- Assume that **P3** is holding semaphore **S** and that **P1** is waiting for **S** to be released
- Assume that **P2** is assigned the CPU and preempts **P3**
  - **P3** is still holding semaphore **S**
  - **P1** is waiting for **S** to be released
- What has happened is that **P2** - a process with a lower priority than **P1** - has indirectly prevented **P3** from gaining access to the resource.
- To prevent this from occurring, a **priority inheritance protocol** is used.

THANK YOU

Chandravva Hebbi
Department of Computer Science Engineering

chandravvahebbi@pes.edu