Process Management

Synchronization Primitives

Lecture 6

https://archipelago.rocks/app/resend-invite/04498013008

Overview

- **Race Condition**
  - Problems with concurrent execution
- **Critical Section**
  - Properties of correct implementation
  - Symptoms of incorrect implementation
- **Implementations of Critical Section**
  - High-level programming language solution
  - Low level (hardware) solution
  - High-level OS abstraction
- **Classical synchronization problems (next lecture)**

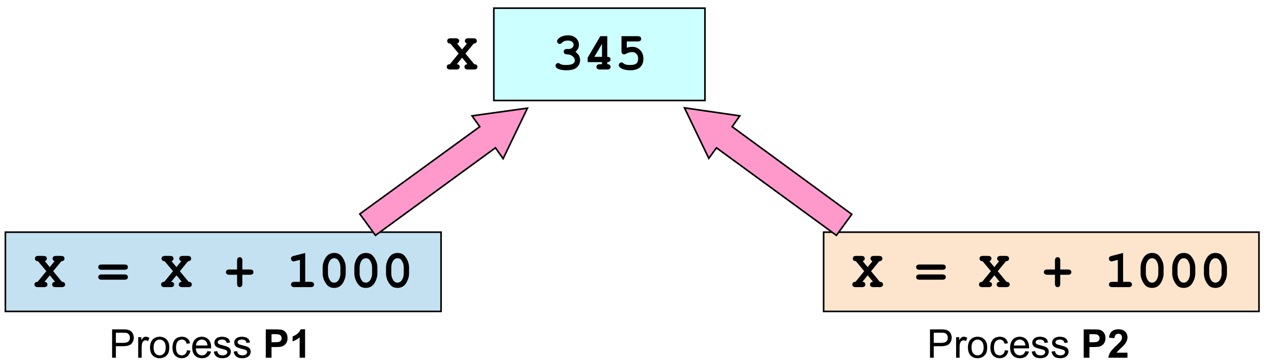
Problems with Concurrent Execution

- Execution of a single sequential process is ***deterministic***
  - Repeated execution gives the same result

- When two or more processes:
  - Execute concurrently in interleaving fashion AND
  - Share a modifiable resource
  - → **Can cause synchronization problems**

- Execution of concurrent processes may be **non-deterministic**
  - Execution outcome depends on the order in which the shared resource is accessed/modified
  - These problems are known as **race conditions** or **data races**

Race Condition: Illustration



- Process `P1` and `P2` shares a variable `X`
- Rough translation for `X = X + 1000`

```
1. Load X → Register1
2. Add 1000 to Register1
3. Store Register1 → X
```

Race Condition: **Round 1**

Time	Value of X	P1	P2
1	345		

Race Condition: Round 2

Time	Value of X	P1	P2
1	345	`Load X → Reg1`	
2	345	`Add 1000 to Reg1`	

Race Condition: **Solution**

Incorrect execution is due to the **unsynchronized access to a shared modifiable resource**

- Needed: synchronization to control the interleaving of accesses to a shared resource
  - Allow all (or as many as possible) correct interleaving scenarios
  - Not allow any incorrect interleaving scenario
- Solution outline:
  - Designate code segment with race condition as **critical section**
  - At any point in time, at most **one process** can be in the critical section

Critical Section (CS)

Critical Section

```
//Normal code
Enter CS
    //Critical Work
Exit CS
    //Normal code
```

Generic Skeleton of code with Critical Section(s):

Example:

```
Enter CS
      X = X + 1000
Exit CS
```

Process **P1**

```
Enter CS
      X = X + 1000
Exit CS
```

Process **P2**

Race Condition Solution: "Illustration"

- **Mutual Exclusion:**
- **Progress:**
- **Bounded Wait:**
- **Independence:**

Properties of Correct CS Implementation

Mutual Exclusion:

- If process P_i is executing in critical section, all other processes are prevented from entering the critical section.

Progress:

- If no process is in a critical section, one of the waiting processes should be granted access.

Bounded Wait:

- After process P_i request to enter critical section, there exists an upper bound on the number of times other processes can enter the critical section before P_i.

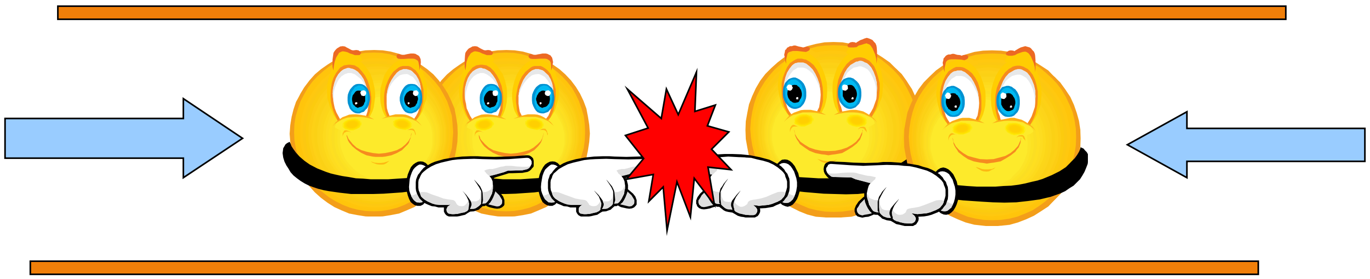
Independence:

- Process **not** executing in critical section should never block other process.

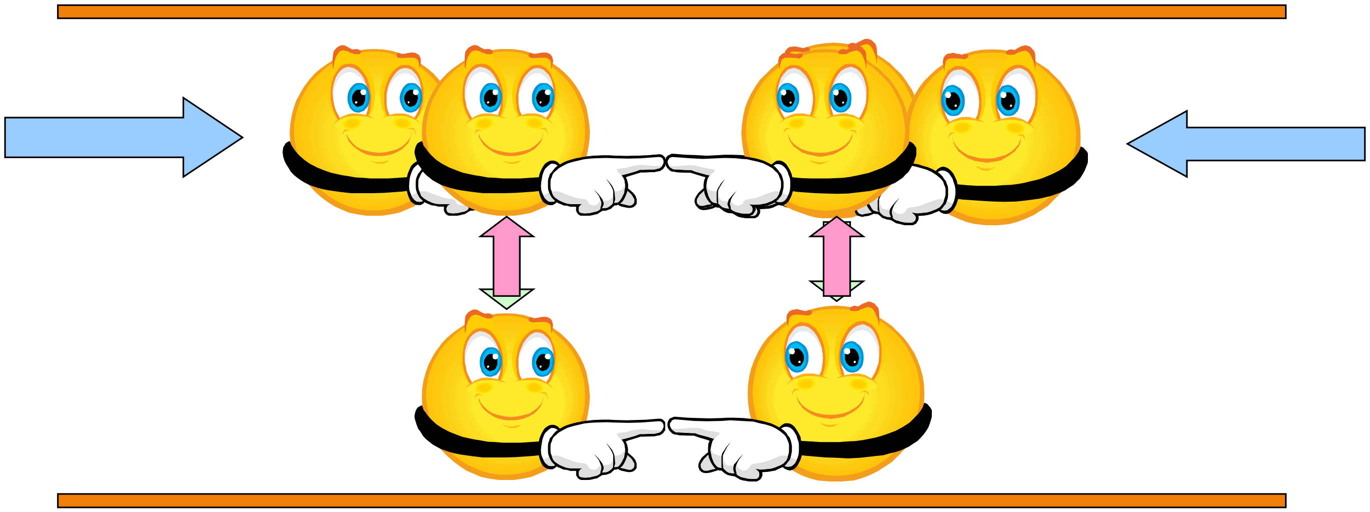
Symptoms of Incorrect Synchronization

- **Incorrect output/behavior**
  - Usually due to lack of mutual exclusion
- **Deadlock:**
  - All processes blocked ➔ ***no progress***
- **Livelock:**
  - Typically, processes are not in a blocked state
  - Processes keep changing state to avoid deadlock but make no other progress
  - Usually related to *deadlock avoidance* mechanisms
- **Starvation:**
  - Some processes are blocked forever

Deadlock



Livelock



CS Implementations Overview

- **High-level programming language implementations:**
  - Using only normal programming constructs

- **Assembly-level implementations:**
  - Mechanisms provided by the hardware
    - through special instructions

- **High level abstractions:**
  - Provide abstracted mechanisms that provide additional useful features
  - Commonly implemented by assembly level mechanisms

Using only your brain power…. ☺

HIGH LEVEL LANGUAGE IMPLEMENTATION

Using HLL: **Attempt 1**

Lock 0

```
while (Lock != 0);
Loc
Loc
```

```
while (Lock != 0);

Lock = 1;
```

Critical Section

Process **P0**

```
while (Lock != 0);
```

```
while (Lock != 0);

Lock = 1;
```

Critical Section

Process **P1**

- Makes intuitive sense ☺
  - But it doesn't work properly ☹
- It violates the "Mutual Exclusion" requirement!
  - How?

Mutual Exclusion:

Progress:

Bounded Wait:

Independence:

Using HLL : Attempt 1 Fixed*

Lock | 0

```
//Disable Interrupts
while (Lock != 0);
Lock = 1;
    Critical Section
Lock = 0;
//Enable Interrupts
```

Process **P0**

```
//Disable Interrupts
while (Lock != 0);
Lock = 1;
    Critical Section
Lock = 0;
//Enable Interrupts
```

Process **P1**

- Solves the problem by preventing context switch
- However:
  - Buggy critical section may stall the WHOLE system
  - Busy waiting
  - Requires permission to disable/enable interrupts

Using High Level Language: **Attempt 2**

Turn

```
while (Turn != 0);
Critical Section
Turn = 1;
```

Process **P0**

```
while (Turn != 1);
Critical Section
Turn = 0;
```

Process **P1**

- **Assumption:**
  - P0 and P1 executes the above in loop
  - Take turn to enter **critical section**
- **Problems:**
  - Starvation:
    - E.g. If P0 never enters CS, P1 starve
  - Violates the **independence** property!

Mutual Exclusion:

Progress:

Bounded Wait:

Independence:

Using High Level Language: Attempt 3

```
Want[0]
Want[1]
```

```
Want[0] = 1;
while (Want[1]);
    Critical Section
Want[0] = 0;
```

Process **P0**

```
Want[1] = 1;
while (Want[0]);
    Critical Section
Want[1] = 0;
```

Process **P1**

Mutual Exclusion:

Progress:

Bounded Wait:

Independence:

Solves the independence problem

- If P0 or P1 is not around, another process can still enter CS

Problem:

- Deadlock! Try identify the execution sequence that causes deadlock

Using High Level Language: Attempt 4

Turn	
Want[0]	
Want[1]	

```
Want[0] = 1;
Turn = 1;
while (Want[1] &&
        Turn == 1);
   Critical Section
Want[0] = 0;
```

Process **P0**

```
Want[1] = 1;
Turn = 0;
while (Want[0] &&
        Turn == 0);
   Critical Section
Want[1] = 0;
```

Process **P1**

Peterson's Algorithm: **Disadvantages**

- **Busy Waiting:**
  - The waiting process repeatedly test the while-loop condition instead of going into blocked state

- **Too Low-level:**
  - Higher-level programming construct is desirable
    - Simpler and less error prone

- **Not general:**
  - General synchronization mechanism is desirable
    - Not just mutual exclusion

Don’t worry! The processor has all the answers!

ASSEMBLY LEVEL IMPLEMENTATION

Inspiration: **HLL Attempt 1**

Lock 0

```
while (Lock != 0);
Lock = 1;
    Critical Section
Lock = 0;
```

Process **P0**

```
while (Lock != 0);
Lock = 1;
    Critical Section
Lock = 0;
```

Process **P1**

Test and Set: An Atomic Instruction

- Machine instruction to aid synchronization
  - Commonly found in modern processors

```
TestAndSet Register, MemoryLocation
```

- **Behavior:**
  1. Load the current content at `MemoryLocation` into `Register`
  2. Stores a **1** into `MemoryLocation`

  - Important: The above is performed as a **single atomic machine operation**
    - Even in multi-core systems!

Using Test and Set

```
void EnterCS( int* Lock )
{
   while( TestAndSet( Lock ) == 1);
}
```

Assume a wrapper function in *C* that uses test and set:
- takes a memory address M:
- returns the current content at M
- sets content of M to 1

$reg

$reg

Lock

```
EnterCS()
  X = X + 1000
ExitCS()
```

Process **P0**

```
EnterCS()
  X = X + 1000
ExitCS()
```

Process **P1**

```
void ExitCS( int* Lock )
{
   *Lock = 0;
}
```

Mutual Exclusion:

Progress:

Bounded Wait:

Independence:

Observations and Comments

- The implementation **works**!
  - However, it employs **busy waiting**
    - Keeps checking the condition until it's safe to enter

    ➔ Wasteful use of processing power
  - Does not guarantee bounded-wait out of the box:
    - **Unless the scheduling is fair**

    ➔ But there are algorithms based on test-and-set that address this

- Variants of this instruction exists on most processors:
  - **Compare and Exchange**
  - **Atomic Swap**
  - **Load Link / Store Conditional**

Let's go higher……

HIGH-LEVEL ABSTRACTION

High Level Synchronization Mechanism

- **Semaphore:**
  - A generalized synchronization mechanism
  - **Only functional behavior is specified**

    ➔ can have different implementations
  - Provides means to:
    - **block** a number of processes
      - Known as **sleeping processes**
    - **unblock/wake up** one or more sleeping process

- History:
  - Proposed by **Edgar W. Dijkstra** in 1965

Semaphore: Wait( ) and Signal( )

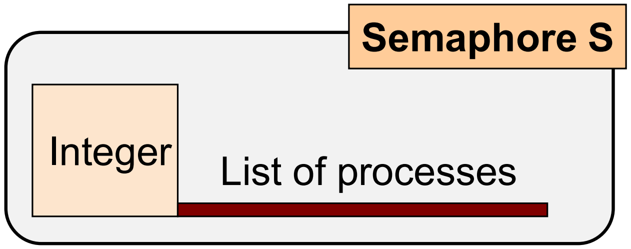
- A semaphore **S** contains an integer value
  - Can be initialized to any non-negative values initially
- Two atomic operations:

`Wait( S )`	**`Signal( S )`**
- **If `S` <= 0, blocks** (go to sleep) - **Decrement `S`** - Also known as **`P()`** or **`Down()`**	- **Increments `S`** - Wakes up one sleeping process if any - This operation **never** blocks - Also known as **`V()`** or **`Up()`**

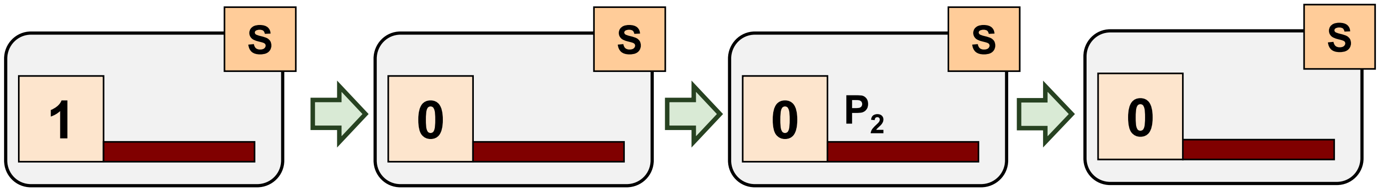
Reminder: The above specifies the **behavior**, not the implementations

Semaphore: **Visualization**

- To aid understanding, you can visualize semaphore as:
  - A protected integer
  - A list to keep track of waiting processes



- Example:



After initialization

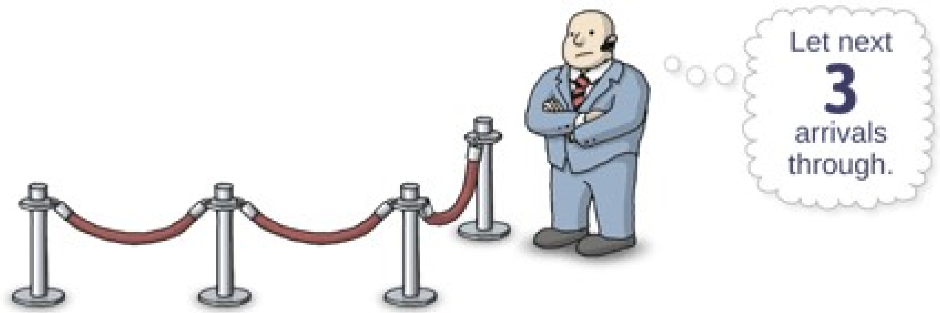
After P_1 wait(S)
P_1 proceeds

After P_2 wait(S)
P_2 suspended

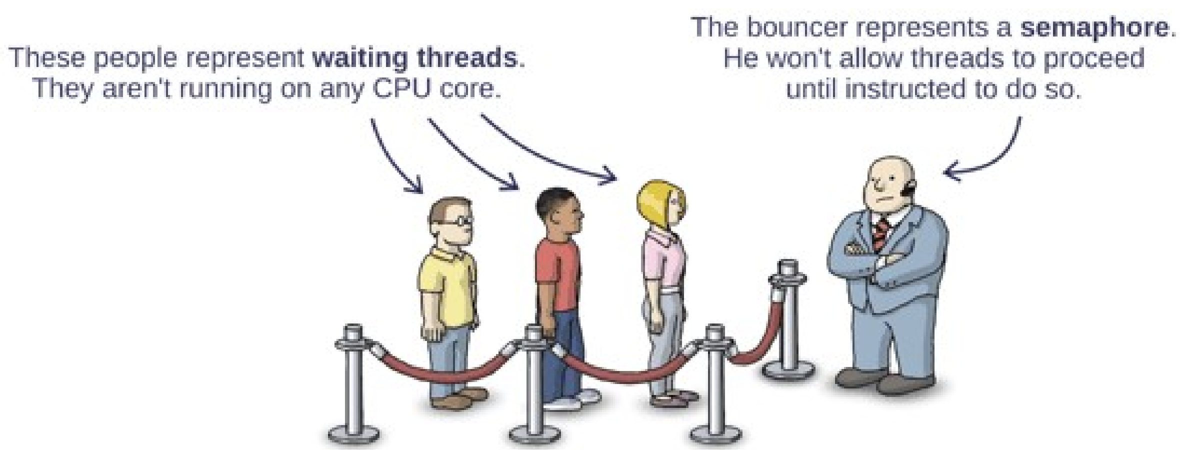
After P_3 signal(S)
P_3 proceeds
P_2 wakes up, decreases S and proceeds

Safe Distancing Problem: Only 3 people allowed

•



- Initially semaphore = 3



- Now semaphore = 0 as 3 people are already in

- New people are waiting in the queue

Semaphores: **Properties**

- Given:
  - $S_{Initial} \geq 0$

- Then, the following **invariant** must be true:

$$S_{current} = S_{Initial} + \#signal(S) - \#wait(S)$$

`#signal(S)` :

- number of signals() operations executed

`#wait(S)` :

- number of wait() operations **completed**

General and Binary Semaphores

- **General semaphore** S:
  - **$S \geq 0$ ($S = 0, 1, 2, 3, ...$)**
  - also called counting semaphores

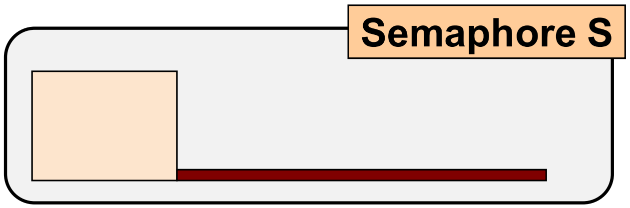
- **Binary semaphore** S:
  - **S = 0** or **1**

- General semaphore is provided for convenience
  - **Binary semaphore is sufficient**
  - i.e., general semaphore can be mimicked by binary semaphores

Semaphore Example: **Critical Section**

- Binary semaphore `s = 1`

```
Wait( s );
    Critical Section
Signal( s );
```



- In this case, **S can only be 0** or **1**
  - From the semaphore invariant
- This semaphore is commonly known as **mutex** (**mut**ual **ex**clusion)

Mutex: Correct CS - Informal Proof

```
Wait( s );
```

Critical Section

```
Signal( s );
```

- **Mutual Exclusion:**
  - N_{CS} = Number of process in critical section
    = Process that completed `wait()` but not `signal()`

$$N_{CS} = \#Wait(S) - \#Signal(S)$$

  - $S_{Initial} = 1$
  - $S_{current} = 1 + \#Signal(S) - \#Wait(S)$

$$S_{current} + N_{CS} = 1$$

  - **Since $S_{current} \geq 0 \rightarrow N_{CS} \leq 1$**

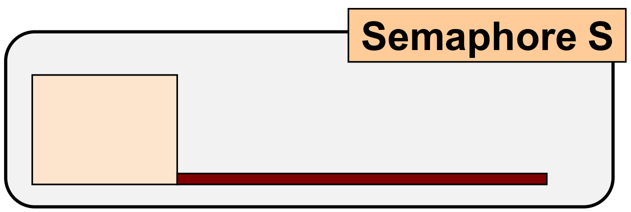
Mutex: Correct CS - Informal Proof (cont)

- **Deadlock:**
  - Deadlock means **all** processes stuck at **`wait( S )`**

    ➔ $S_{current}$ **= 0** and N_{CS} **= 0**

    $$S_{current} + N_{CS} = 1$$

  - ➔ ← (contradiction)



- **Starvation:**
  - Suppose **`P1`** is blocked at **`wait( S )`**
  - **`P2`** is in CS, exits CS with **`signal( S )`**
    a. If no other process sleeping, **`P1`** wakes up
    b. If there are other process, **`P1`** eventually wakes up (under fair scheduling)

Incorrect Use of Semaphore: Deadlock

- Deadlock is still possible with **incorrect use of semaphore**

- Example: Semaphore **P = 1**, **Q = 1** initially

```
1  Wait( P )
3  Wait( Q )
     Some Code
   Signal( Q )
   Signal( P )
```

Process **P0**

```
2  Wait( Q )
4  Wait( P )
     Some Code
   Signal( P )
   Signal( Q )
```

Process **P1**

Semaphore as General Synchronization Tool

- Execute *B* in P_1 **only after** *A* executed in P_o
- Use semaphore *s* initialized to 0
- Code:

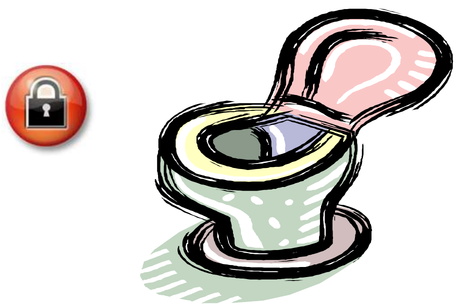
P0	P1
.	.
.	.
.	.
A	wait(s)
signal(s)	B
.	.

Summary: Most Common Uses Of Semaphores

Critical Section

```
BinarySemapore mutex = 1;

void use_restroom(){
   wait(mutex);
   critical_section();
   signal(mutex);
}
```



Safe-distancing problem

```
Semaphore sem = N;

void cient(){
    wait(sem);
    safe_eating();
    signal(sem);
}
```

General synchronization

```
Semaphore sem = 0;

Process P1:
    produce(X);
    signal(sem);
```

```
process P2:
    wait(sem);
    consume(X);
```

Other High-Level Abstractions

- Semaphore is **very powerful**:
  - There are no known unsolvable synchronization problem with semaphore (so far ☺)
  - Other high-level abstractions essentially provide extended features
    - Usually features that are troublesome to express using semaphore alone

- Common alternative: **Conditional Variable**
  - Allow a task to wait for certain event to happens
  - Has the ability to ***broadcast***, i.e., wakes up all waiting tasks
  - related to **monitors**

POSIX Semaphore

- Popular implementation of semaphores in Unix
- Header File:
  - `#include <semaphore.h>`
- Compilation Flag:
  - `gcc something.c -lrt`
  - Stand for "real time library"
- Basic Usage:
  - Initialize a semaphore
  - Perform **`wait()`** or **`signal()`** on semaphore

pthread Mutex and Conditional Variables

- Synchronization mechanisms for pthreads
- Mutex (pthread_mutex):
  - Binary semaphore (i.e. equivalent Semaphore(1)).
  - Lock: **`pthread_mutex_lock()`**
  - Unlock: **`pthread_mutex_unlock()`**

- Conditional Variables(pthread_cond):
  - Wait: **`pthread_cond_wait()`**
  - Signal: **`pthread_cond_signal()`**
  - Broadcast: **`pthread_cond_broadcast()`**

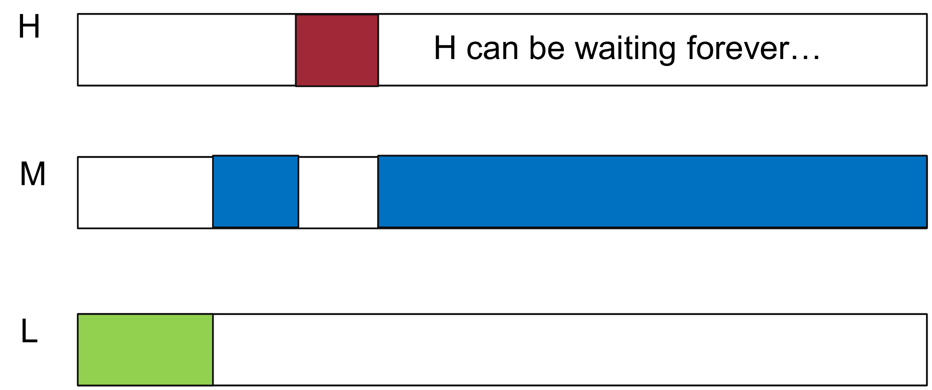
Programming language Support

- Programming languages with thread support must have some synchronization mechanisms

- Examples:
  - **Java**: all object has built-in lock (mutex), **synchronized** method access, monitors, etc.
  - **Python**: supports mutex, semaphores, conditional variables, etc.
  - **C++**: Added built-in thread in C++11; Support mutexes, conditional variable

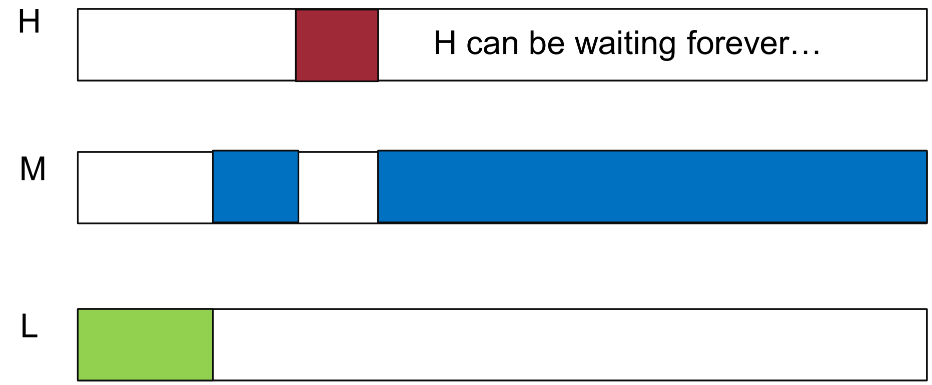
Recall **Priority Scheduling**: **Priority Inversion**

- Consider the scenario:
  - Priority: {H = 1, M=3, L= 5} (1 is highest)
  - **L** starts and **locks** a resource (e.g., a file)
  - **M** arrives and preempts **L** (higher priority)
    - **L** is unable to unlock the file
  - **H** arrives and need the same resource as **L**
    - but the resource is locked! And L can't run because of M
  - ➔ **M** continues executes even if Task **H** has higher priority
  - ➔ **M** can starve **H** for as long as it wants!



- Known as **Priority Inversion**:
  - Lower priority task effectively preempts higher priority task

Priority Inversion: **Solution**



Priority inheritance

- Temporarily increase priority of **L** to **H**
  - Until it unlocks the lock

- Low-priority job *inherits* the priority of the higher priority process
  - Happens when the high-priority process requests a lock held by low-priority process
  - Priority restored upon successful unlock()

Summary

Synchronization:

- Problem: Race conditions
- Solution: Critical Section
- Criteria for a good solution:
  - Mutual Exclusion,
  - progress,
  - bounded waiting time,
  - independence
- Important High-Level Construct: Semaphore
  - Examples in Unix

References

- Modern Operating System (4th Edition)
  - Chapter 2.5
- Operating System Concepts (9th Edition)
  - Chapter 5
- Three Easy Pieces
  - Chapters 25, 26, …., 34!
- Edgar W. Dijkstra, "Note No.123: Cooperating Sequential Processes"
  - http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF