Operating Systems

Tutorial 7

Dr. Eng. Catherine M. Elias
Dr. Aya Salama
Eng. Farah Shams
Eng. Mariam ElOraby
Eng. Mostafa Moataz

Processes Synchronization

- Shared HW/SW resources = conflicts!
- Race conditions, deadlocks, starvation, priority inversion …

Race Condition

A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes.

Race Condition Example 1

Consider a banking system that maintains an account balance with two functions: `withdraw(account, amount)`and `deposit(account, amount)`. These two functions are passed the bank account and the amount that is to be deposited or withdrawn from the bank account balance. Assume that two persons **A** and **B** share a bank account with an initial balance of 300 LE, and that, simultaneously, **A** calls `withdraw(account, 100)` and **B** calls `deposit(account, 50)`.

```
int withdraw(account, amount) {
     int balance = get_balance(account);
     balance -= amount;
     put_balance(account, balance);
     return balance;
}
```

```
int deposit(account, amount) {
     int balance = get_balance(account);
     balance += amount;
     put_balance(account, balance);
     return balance;
}
```

Race Condition Example 1 - continued

Consider the following execution sequence:

```
int balance = get_balance(account);
balance -= amount;
```

```
int balance = get_balance(account);
balance += amount;
put_balance(account, balance);
```

```
put_balance(account, balance);
```

What is the final account balance given this execution sequence?

Race Condition Example 1 - continued

Consider the following execution sequence:

```
int balance = get_balance(account);
balance -= amount;
```

```
int balance = get_balance(account);
balance += amount;
put_balance(account, balance);
```

```
put_balance(account, balance);
```

What is the final account balance given this execution sequence? 200 LE.
(Note: the correct balance should be 250 LE)

Race Condition Example 2

```
void echo()
{
    chin = getchar();
    chout = chin;
    putchar(chout);
}
```

Character echo is a procedure where each key pressed is immediately displayed on the screen. In the above procedure, input is obtained from a keyboard one keystroke at a time. Each input character is stored in the variable `chin`. It is then transferred to variable `chout` and sent to the display. Any program can call this procedure repeatedly to accept user input and display it on the user's screen. Because each application needs to use the procedure `echo`, it is loaded into a portion of memory global to all applications.

Race Condition Example 2 - continued

Two processes P1 and P2 invoke the `echo` procedure:

```
void echo()
{
     chin = getchar();
     chout = chin;
     putchar(chout);
}
```

P1

```
void echo()
{
     chin = getchar();
     chout = chin;
     putchar(chout);
}
```

P2

Race Condition Example 2 - continued

Consider the following execution sequence:

```
chin = getchar();
```

```
chin = getchar();
chout = chin;
putchar(chout);
```

```
chout = chin;
putchar(chout);
```

What is displayed?

Race Condition Example 2 - continued

Consider the following execution sequence:

```
chin = getchar();
```

```
chin = getchar();
chout = chin;
putchar(chout);
```

```
chout = chin;
putchar(chout);
```

What is displayed? The first character is lost and the second character is displayed twice.

Critical Section and Mutual Exclusion

- **Critical Section:** A section of code within a process that requires access to shared resources.
- **Mutual Exclusion:** The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
- To solve the critical section problem and ensure mutual exclusion, each process must request permission to enter its critical section. This can be achieved using **locks**, which protect critical regions. A process must acquire the lock before entering and release it when exiting the critical section.

Spinlock

```
void acquire(lock) {
     while (!lock.available);   /* busy waiting */
     lock.available = false;
}


void release(lock) {
     lock.available = true;
}
```

Race Condition Example 2 - revisited

```
void echo()
{

    chin = getchar();
    chout = chin;
    putchar(chout);

}
```

Race Condition Example 2 - revisited

```
void echo()
{
    acquire(lock);
    chin = getchar();
    chout = chin;
    putchar(chout);
    release(lock);
}
```

Race Condition Example 2 - revisited

Consider the following execution sequence.
What is displayed?

```
acquire(lock);
chin = getchar();
```

```
acquire(lock);
```

```
chout = chin;
putchar(chout);
release(lock);
```

```
chin = getchar();
chout = chin;
putchar(chout);
release(lock);
```

GUC German University in Cairo

Race Condition Example 2 - revisited

Consider the following execution sequence.
What is displayed?
The first character is displayed followed by the 2nd.

```
acquire(lock);
chin = getchar();
```

```
acquire(lock);
```

```
chout = chin;
putchar(chout);
release(lock);
```

```
chin = getchar();
chout = chin;
putchar(chout);
release(lock);
```

Race Condition Example 1 - revisited

```
int withdraw(account, amount) {
     int balance = get_balance(account);
     balance -= amount;
     put_balance(account, balance);
     return balance;
}
```

```
int deposit(account, amount) {
     int balance = get_balance(account);
     balance += amount;
     put_balance(account, balance);
     return balance;
}
```

GUC
German University in Cairo

Race Condition Example 1 - revisited

```
int withdraw(account, amount) {
    acquire(lock);
    int balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}
```

```
int deposit(account, amount) {
    acquire(lock);
    int balance = get_balance(account);
    balance += amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}
```

Race Condition Example 1 - revisited

```
acquire(lock);
int balance = get_balance(account);
balance -= amount;
```

```
acquire(lock);
```

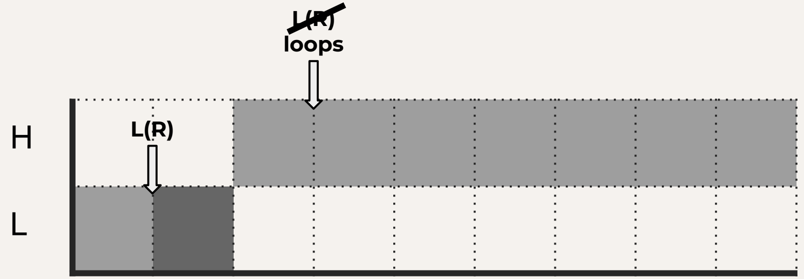
```
put_balance(account, balance);
release(lock);
```

```
int balance = get_balance(account);
balance += amount;
put_balance(account, balance);
release(lock);
```

What is the final account balance given this execution sequence? 250 LE

Spinlock Drawbacks

- **Busy waiting wastes CPU time:**
  - Processes waiting to enter their critical sections waste processor time checking to see if they can proceed.
- **Starvation** (and *priority inversion*)**:**
  - **Scenario:** Consider a preemptive priority-based scheduler with two processes: **H**, with high priority, and **L** with low priority. **L** starts running first and enters its critical section by acquiring a lock, then **H** arrives and **L** is preempted. **H** is now busy waiting, but since **L** cannot be scheduled while **H** is running, **L** never gets the chance to leave its critical region, so **H** loops forever.



Blocking Locks

A better solution would be to block processes when they are denied access to their critical sections. A pseudo example*:

```
void acquire(lock) {
     If (!lock.available){
          sleep();
     }
     lock.available = false;
}


void release(lock) {
     lock.available = true;
     wakeup(other_process);
}
```

- `Sleep` is a system call that causes the *caller* to block and be suspended until another process wakes it up.

  The `wakeup` system call unblocks the process passed as a parameter.

- **Scenario: L** starts running and enters its critical section, then **H** arrives and **L** is preempted. **H** starts running, tries to acquire lock but is blocked. **L** runs and exits its critical section, releasing the lock. **H** is unblocked and starts running.

! **Priority Inversion problem remains.**

*Different implementations of locks which do *not* require busy waiting will be discussed later in lecs/tuts.

Priority Inversion

Priority inversion occurs when a low-priority process/thread holds a resource (lock, HW resource, memory, etc) required by a high-priority process/thread, causing the high-priority process to wait longer than expected. This can happen if the low-priority process is preempted by a medium-priority one while holding the resource.

Priority Inversion Example

Preemptive priority-based scheduler:

Task Name	Arrival Time	Service Time	Priority	R_i	$t_{L(R_i)}$
T1	5	4	1	T1: $[R_1,3]$	$t_{L(R1)}$ =exec.St.+1
T2	4	3	2	T2: $[R_1,2]$	$t_{L(R1)}$ =exec.St.+1
T3	2	4	3	-	-
T4	0	6	4	T4: $[R_1,3]$	$t_{L(R1)}$ =exec.St.+1

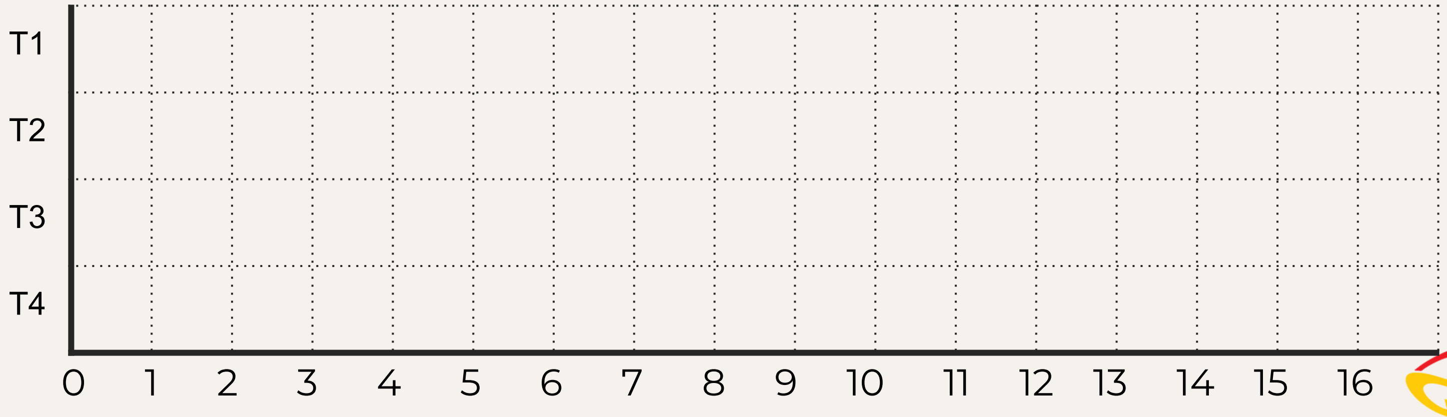
Priority Inversion Example

Notations:

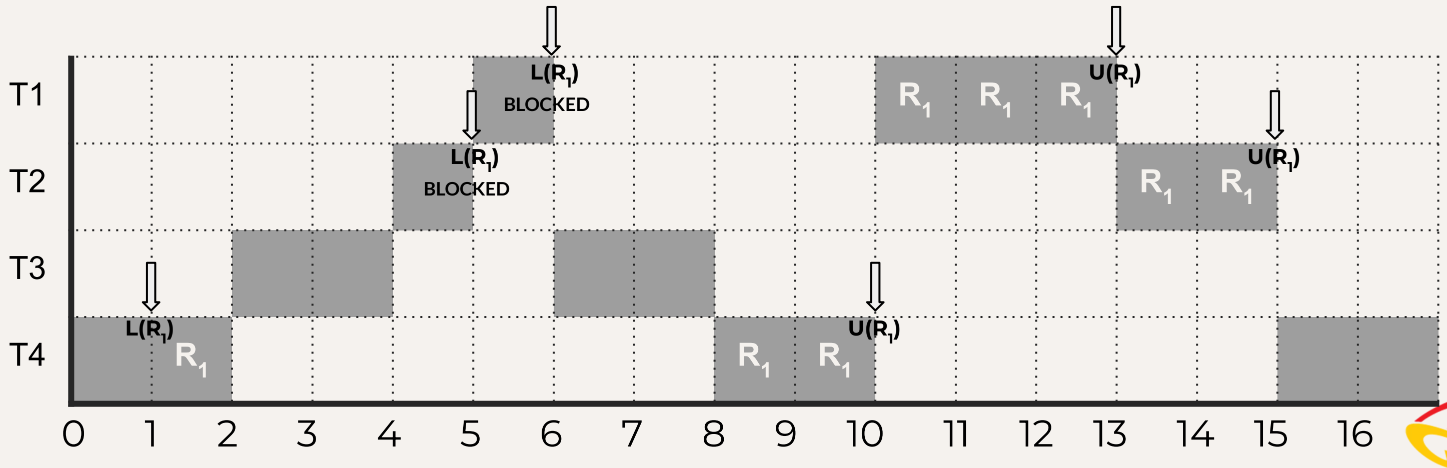
- **Ti**: Task.
- **R_j**: Resource.
- **$L(R_j)$**: Locking request to resource R_j .
- **$U(R_j)$**: Unlocking request to resource R_j .
- **C_j**: time spent in the critical section corresponding to resource R_j .
- **$t_{L(Rj)}$**: start time of critical section.
- **exec.St**: start time of execution.

A task can be written as: **$Ti=(tr_i, e_i):\{ [R_j,c_j], ...\}$**

Task Name	Arrival	Service Time	Priority	R_i	$t_{L(Ri)}$
T1	5	4	1	T1: $[R_1,3]$	$tL(R_1)$ =exec.St.+1
T2	4	3	2	T2: $[R_1,2]$	$tL(R_1)$ =exec.St.+1
T3	2	4	3	-	-
T4	0	6	4	T4: $[R_1,3]$	$tL(R_1)$ =exec.St.+1



Task Name	Arrival	Service Time	Priority	R_i	$t_{L(Ri)}$
T1	5	4	1	T1: $[R_1,3]$	$tL(R_1)$ =exec.St.+1
T2	4	3	2	T2: $[R_1,2]$	$tL(R_1)$ =exec.St.+1
T3	2	4	3	-	-
T4	0	6	4	T4: $[R_1,3]$	$tL(R_1)$ =exec.St.+1



All

Done!