Process Synchronization - II

Lecture 14

March 8 2017
Process Synchronization

- Race Condition
- Critical Section
- Mutual Exclusion
- Peterson’s Solution
- Disabling Interrupts
- Test and Lock Instruction (TSL)
- Semaphores
- Deadlock
Mutual Exclusion in Critical Sections
Semaphores

- Semaphore is a process synchronization technique supported by the OS.

- Semaphores are used by programmers to:
  - ensure mutual exclusion
  - send signals from one process to another

- Today there are libraries that provide application programmers with semaphores.
What is a semaphore?

- A semaphore allows multiple processes to cooperate by using signal. Semaphore is an integer variable with the following three operations:
  
  - **Initialize**: Semaphore is initialized to any non-negative value (1).
  
  - **Decrement**: *wait (down operation)* decrements the semaphore and if \( S < 0 \) then the process is blocked (and placed into a queue). If \( S > -1 \) then process enter into Critical Section.
  
  - **Increment**: *signal (up operation)* increments the semaphore value, and if \( S \leq 0 \) and if there is a process in the queue then a process is unblocked and enters into Critical Section.
What is a Semaphore?

- Use **wait / down** before entering a critical section
- Use **signal / up** after finishing with a critical section
- Example: Assume $S$ is initialized to 1.

```java
    S = 1;
    while (true)
    {
        down (S);
        critical section
        up(S);
        remainder section;
    }
```
Semaphores Example

Process $P_0$

- $S = 1$
- `down(S);`
- `critical section`
- `up(S);`

Process $P_1$

- $S = 1$
- `down(S);`
- `critical section`
- `up(S);`

- Initialize the semaphore variable, $S$, to 1
- Now what would happen if $P_0$ executes the `down` operation?
  - The semaphore $S$ is currently 1.
  - $S$ becomes 0 and $P_0$ enters the critical section
Semaphores Example

Process $P_0$

$S = 1$
down(S);
    critical section
up(S);

Process $P_1$

$S = 1$
down(S);
    critical section
up(S);

- Now what would happen if $P_1$ executes the down operation?
  - The semaphore $S$ is currently 0, after down operation by $P_1$, $S$ becomes -1
  - $P_1$ is blocked and placed into a queue
Semaphores Example

Process $P_0$

$S = 1$
down($S$);
critical section
up($S$);

Process $P_1$

$S = 1$
down($S$);
critical section
up($S$);

- Now what would happen if $P_0$ is done with critical section?
  - $P_0$ calls the up function
    - $S$ becomes 0
    - $P_1$ is unblocked and $P_1$ enters into Critical Section
  - If there was no process waiting in the queue then the value of $S$ would become 1
Semaphores Example

- What happens if there are three processes: $P_0, P_1, P_2$?
- Assume $P_0$ enters its critical section.
- If $P_1$ and $P_2$ execute the `down` operation they will block.
- When $P_0$ leaves the critical section then $P_1$ is unblocked allowing $P_1$ to enter its critical section.
  - $P_2$ is still blocked.
- What if $P_0$ wants to enter its critical section again when $P_1$ is in it?
Semaphore Types

- **Binary Semaphore:**
  - Allows only ONE process to be in critical section at a time
  - Initialized to 1
  - Often referred to as a mutex

- **Counting Semaphore:**
  - Allows multiple processes to be in critical section at a time
  - Initialized to N where N is the max processes that can be in critical section simultaneously
Implementation

- 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
Implementation

- A semaphore can be defined as a C struct along these lines:

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

- The `down()` operation can be defined as

  ```c
  down(semaphore *S) {
    S->value--;  
    if (S->value < 0) {
      add this process to S->list;  
      block();
    }
  }
  ```
Implementation

- `up()` operation can be defined as

```c
up(semaphore *S) {
    S->value++;
    if (S->value ≤ 0) {
        remove process P from S->list;
        wakeup(P);
    }
}
```
Implementation

- BTW, the implementation just described is how Linux implements semaphores
- Need hardware/OS support e.g.,
  - Signals, TSL
  - Signals allow for a “message” to be sent to processes. When the up is executed a blocked process is woken up. This is done using signals
Deadlock

- **Deadlock** - Two or more processes are waiting indefinitely for an event that can only be caused by one of the waiting processes.
Deadlock

- **Avoidance Approaches**
  - Avoid cycle in the resource allocation graph
  - Avoid Mutual Exclusion
  - Avoid Hold & Wait
    - Release resources that may not be needed immediately
    - Do not request resource ahead of time
  - Block a process that requesting large number of resources
Deadlock

- Deadlock Recovery
  - Abort all deadlock processes
  - Back up all deadlock processes to the previous safe state and then restart
  - Selectively abort processes until deadlock broken