Computer Systems

Synchronizing Threads

with

Semaphores

Review: Multi-Threaded Processes
Review: badcnt.c

```c
#define NITERS 1000000

unsigned int cnt = 0; /* shared */
int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, Count, NULL);
    pthread_create(&tid2, NULL, Count, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
}
/* thread routine */
void * Count(void *arg) {
    int i;
    for (i=0; i<NITERS; i++)
        cnt++;
    return NULL;
}
```

```bash
./badcnt
BOOM! cnt=1988411
./badcnt
BOOM! cnt=1982618
./badcnt
BOOM! cnt=1982696
```

**cnt** should be equal to 2,000,000. What went wrong?!

Review: Race Conditions

- Assume that two threads execute concurrently
  ```c
cnt++; /* cnt is global shared data */
  ```

- One possible interleaving of statements is:

```
R1 = cnt
R1 = R1 + 1
Timer interrupt!
R2 = cnt
R2 = R2 + 1
Timer interrupt!
cnt = R2
```

**Race condition:**
The situation where several threads access shared data concurrently. The final value of the shared data may vary from one execution to the next.

- Then **cnt** ends up incremented once only!
Review: Critical Sections

- Blocks of code that access shared data:

```c
/* thread routine */
void * Count(void *arg) {
    int i;
    for (i=0; i<NITERS; i++)
        cnt++;
    return NULL;
}
```

- Threads need mutual exclusive access to critical sections:
  two threads cannot simultaneously execute `cnt++`

The Critical-Section Problem

- **Critical section (CS):** block of code that uses shared data
- **CS problem:** correctness of the program should not depend on one thread reaching a certain point before another thread
- **CS solution:** allow a **single thread** in the CS at one time

```
while(1) {
    entry section
    Critical Section (CS)
    exit section
    Remainder Section (RS)
}
```

Thread $T_i$
while(1) {
    entry section
    Critical Section (CS)
    exit section
    Remainder Section (RS)
}

while(1) {
    entry section
    Critical Section (CS)
    exit section
    Remainder Section (RS)
}

Solving the CS Problem - Semaphores (1)

- A semaphore is a synchronization tool provided by the operating system.
- A semaphore S can be viewed as an integer variable that can be accessed through 2 atomic operations:

  - **DOWN(S)** also known as **wait(S)**
  - **UP(S)** also known as **signal(S)**

*Atomic means indivisible.*

- When a thread has to wait, put it in a queue of blocked threads waiting on the semaphore.
In fact, a semaphore is a structure:

```c
struct Semaphore {
    int count;
    Thread * queue; /* blocked threads */
};
struct Semaphore S; /* global variable */
```

- `S.count` must be initialized to a nonnegative value (depending on application)

**OS Semaphores – `DOWN(S)` or `wait(S)`**

- When a process must wait for a semaphore $S$, it is blocked and placed on the semaphore’s queue

  ```c
  DOWN(S):
  <disable interrupts>
  S.count--;
  if (S.count < 0) {
    block this thread
    place this thread in S.queue
  }
  <enable interrupts>
  ```

- Threads waiting on a semaphore $S$: 

  ![Diagram showing threads waiting on a semaphore]

  - `S.queue` 
  - `T3` 
  - `T1` 
  - `T5`
OS Semaphores – \textit{UP}(S) or signal(S)

- The \textit{UP} or \textit{signal} operation removes one thread from the queue and puts it in the list of ready threads

\begin{verbatim}
\textbf{UP(S):}
\begin{verbatim}
<disable interrupts>
S.count++;
if (S.count \leq 0) {
  remove a thread T from S.queue
  place this thread T on Ready list
}
<enable interrupts>
\end{verbatim}
\end{verbatim}

\end{verbatim}

OS Semaphores - Observations

- When \texttt{S.count} \geq 0
  - the number of threads that can execute \textit{wait(S)} without being blocked is equal to \texttt{S.count}

- When \texttt{S.count} < 0
  - the number of threads waiting on \texttt{S} is equal to |\texttt{S.count}|

- Atomicity and mutual exclusion
  - no two threads can be in \textit{wait(S)} and \textit{signal(S)} (on the same \texttt{S}) at the same time
  - hence the blocks of code defining \textit{wait(S)} and \textit{signal(S)} are, in fact, critical sections
Using Semaphores to Solve CS Problems

Thread Ti:
DOWN(S);
<critical section CS>
UP(S);
<remaining section RS>

- To allow only one thread in the CS (mutual exclusion):
  - initialize S.count to ____

- To allow n threads in the CS, for some n > 1:
  - initialize S.count to ____

Solving the Earlier Problem

```c
/* Thread T1 routine */
void * Count(void *arg) {
    int i;
    for (i=0; i<10; i++) {
        cnt++;
    }
    return NULL;
}
```

```c
/* Thread T2 routine */
void * Count(void *arg) {
    int i;
    for (i=0; i<10; i++) {
        cnt++;
    }
    return NULL;
}
```

- Solution: use Semaphores to impose mutual exclusion to executing cnt++
## How are Races Prevented?

Semaphore mutex; 

<table>
<thead>
<tr>
<th>Thread T1 routine</th>
<th>Thread T2 routine</th>
</tr>
</thead>
</table>
| ```c
void * Count(void *arg)
{
    int i;
    for (i=0; i<10; i++) {
        DOWN(mutex);
        R1 ← cnt
        R1 ← R1+1
        cnt ← R1
        UP(mutex);
    }
    return NULL;
}
``` |
| ```c
void * Count(void *arg)
{
    int i;
    for (i=0; i<10; i++) {
        DOWN(mutex);
        cnt++;
        UP(mutex);
    }
    return NULL;
}
``` |

## Solving the Earlier Problem

Semaphore mutex; 

<table>
<thead>
<tr>
<th>Thread T1 routine</th>
<th>Thread T2 routine</th>
</tr>
</thead>
</table>
| ```c
void * count(void *arg)
{
    int i;
    for (i=0; i<10; i++) {
        DOWN(mutex);
        cnt++;
        UP(mutex);
    }
    return NULL;
}
``` |
| ```c
void * count(void *arg)
{
    int i;
    for (i=0; i<10; i++) {
        DOWN(mutex);
        cnt++;
        UP(mutex);
    }
    return NULL;
}
``` |
Activity – Understanding Semaphores

What are possible values for \( g \), after the three threads below finish executing?

```c
/* global shared variable */
int g = 10;
Semaphore s = 0;
```

**Thread A**

\[
\begin{align*}
g &= g + 1; \\
UP(s); \\
g &= g \times 2;
\end{align*}
\]

**Thread B**

\[
\begin{align*}
DOWN(s); \\
g &= g - 2; \\
UP(s);
\end{align*}
\]
Using OS Semaphores

- Semaphores have two uses:
  - **Mutual exclusion**: making sure that only one thread is in a critical section at one time
  - **Synchronization**: making sure that T1 completes execution before T2 starts?

Synchronizing Threads with Semaphores

- Suppose that we have 2 threads: T1 and T2
- How can we ensure that a statement S1 in T1 executes before statement S2 in T2?

<table>
<thead>
<tr>
<th>Thread T1</th>
<th>Thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphore sync; 0</td>
<td></td>
</tr>
<tr>
<td>S1;</td>
<td>DOWN(sync);</td>
</tr>
<tr>
<td>UP(sync);</td>
<td>S2;</td>
</tr>
</tbody>
</table>
Activity

- Consider two concurrent threads T1 and T2. T1 executes statements S1 and S3 and T2 executes statement S2.

<table>
<thead>
<tr>
<th>Thread T1</th>
<th>Thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>S2</td>
</tr>
<tr>
<td>S3</td>
<td></td>
</tr>
</tbody>
</table>

- Use semaphores to ensure that the statements are always executed in the order S1, S2, S3.

Review: Mutual Exclusion

```
Semaphore mutex; 1

Thread T1
  DOWN(mutex);
  critical section
  UP(mutex);

Thread T2
  DOWN(mutex);
  critical section
  UP(mutex);
```
Review: Synchronization

- T2 cannot begin execution until T1 has finished:
  
  ```
  Semaphore mutex; 0
  ```

<table>
<thead>
<tr>
<th>Thread T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code for T1</td>
</tr>
<tr>
<td>UP(mutex);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOWN(mutex);</td>
</tr>
<tr>
<td>Code for T2</td>
</tr>
</tbody>
</table>

Deadlock and Starvation

- **Deadlock** occurs when two or more threads are waiting indefinitely for an event that can be caused by only one of the waiting threads
- Let S and Q be two semaphores initialized to 1

  ```
  Thread T_1          Thread T_2
  wait (S);           wait (Q);
  wait (Q);           wait (S);
  ...                 ...                 ...
  signal (S);         signal (Q);
  signal (Q);         signal (S);
  ```

- **Starvation** – indefinite blocking
  - a thread may never be removed from the semaphore queue in which it is suspended
Summary

- Threads
  - Share global data
- Problem with Threads
  - Races
- Eliminating Races
  - Mutual Exclusion with Semaphores
- Thread Synchronization
  - Use Semaphores
- POSIX Threads and Semaphores

Hands-on Session

- POSIX semaphores
  - `sem_init`
  - `sem_wait`
  - `sem_post`
  - `sem_getvalue`
  - `sem_destroy`

- Fix the `badcnt.c` program to produce correct result
  - Exercise 2 on class website