

Harnessing the Power of Threads

CS439: Principles of Computer Systems

Bringing It All Together

- Processes
 - Abstraction for protection
 - Define address space
- Threads
 - Share (and communicate) through global and static data, share the heap, each has its own stack and full use of the registers
 - Race conditions may be a problem!
- CPU Schedulers
 - May pre-empt a process or thread at any time
- Solving race conditions (Ensuring correctness)
 - Safety and liveness
 - Atomic instructions
 - Synchronization: mutual exclusion
 - Locks! Semaphores!

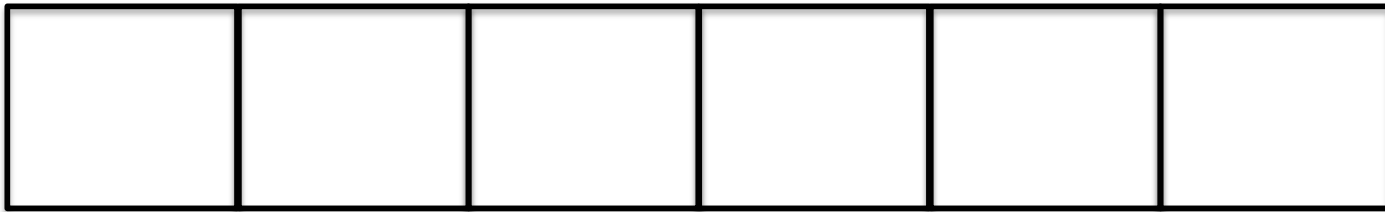
Vocabulary

logically atomic	binary semaphore	release
processes	threads	acquire
busy waiting	bounded waiting	atomic operation
free	failure atomicity	critical section
synchronization	mutual exclusion	read-modify-write
liveness	counted semaphore	lock
semaphore	disable interrupts	test&set
enable interrupts	down	safety
up	sleep	race condition
rendezvous pattern	busy	

Producers and Consumers

CS439: Principles of Computer Systems

Producers/Consumers



- Buffer is a global data structure of fixed size n
- Producer produces items and puts them in the buffer
- Consumer consumes items from the buffer
- What happens if the producer produces an item and the buffer is full?
- What happens if the consumer consumes an item and the buffer is empty?

When to Use Semaphores

- Mutual Exclusion
 - Use to protect the critical section (see Too Much Milk Example)
- Control Access to a Pool of Resources
 - Counted semaphore
- General Synchronization
 - Use to enforce general scheduling constraints where the threads must wait for some circumstance
 - Value is typically 0 to start
 - `down()` and `up()` are called by different threads from different parts of the code base

Semaphores: Atomic Operations

- Down()
 - Actually P() (*Proberen*, or “pass” in Dutch)
 - Decrements the value
 - When down() returns, the thread has the resource
 - Can block: if resource not available (as indicated by count), the thread will be placed on a wait queue and put to sleep
- Up()
 - Actually V() (*Verhogen*, or “release” in Dutch)
 - Increments the value
 - Never blocks
 - If a thread is asleep on the wait queue, it will be awakened

Producers/Consumers

```
int buffer[N]
```

```
BoundedBuffer::Producer(){
```

```
<produce item>
```

```
<add item to buffer>
```

```
}
```

```
BoundedBuffer::Consumer(){
```

```
<remove item from buffer>
```

```
<use item>
```

```
}
```

Producers/Consumers

```
int buffer[N]
Sema mutex = 1;
Sema empty = N;
Sema full = 0;
```

```
BoundedBuffer::Producer(){
```

```
<produce item>
```

```
<add item to buffer>
```

```
}
```

- 1) Access to the buffer
- 2) producers wait if buffer is full
- 3) Consumers wait if buffer is empty

```
BoundedBuffer::Consumer(){
```

```
<remove item from buffer>
```

```
<use item>
```

```
}
```

Producers/Consumers

```
Semaphore mutex = 1 //access to buffer
Semaphore empty = N //count of empty slots
Semaphore full = 0 //count of full slots
int buffer[N]
```

```
BoundedBuffer::Producer(){
    <produce item>

    <add item to buffer>

}
```

```
BoundedBuffer::Consumer(){

    <remove item from buffer>

    <use item>

}
```

Producers/Consumers

```
Semaphore mutex = 1 //access to buffer
Semaphore empty = N //count of empty slots
Semaphore full = 0 //count of full slots
int buffer[N]
```

```
BoundedBuffer::Producer(){
    <produce item>
    empty->Down() //get empty spot
    mutex->Down() //get access to buffer

    <add item to buffer>

    mutex->Up() //release buffer
    full->Up() //another item in buffer
}
```

```
BoundedBuffer::Consumer(){
    full->Down() //get item
    mutex->Down() //get access to buffer

    <remove item from buffer>

    mutex->Up() //release buffer
    empty->Up() //another empty slot
    <use item>
}
```

Let's make a small change...

```
Semaphore mutex = 1 //access to buffer
Semaphore empty = N //count of empty slots
Semaphore full = 0 //count of full slots
int buffer[N]
```

```
BoundedBuffer::Producer(){
    <produce item>
    empty->Down() //get empty spot
    mutex->Down() //get access to buffer

    <add item to buffer>

    mutex->Up() //release buffer
    full->Up() //another item in buffer
}
```

```
BoundedBuffer::Consumer(){
    mutex->Down() //get access to buffer
    full->Down() //get item

    <remove item from buffer>

    mutex->Up() //release buffer
    empty->Up() //another empty slot
    <use item>
}
```

Problems with Semaphores

- Huge step up from what we had, but...
- Essentially shared global variables
- Too many purposes
 - Waiting for a condition is independent of mutual exclusion
- No control or guarantee of proper usage
- Difficult to read (and develop) code
- Not typically used in new *user* code
 - Where are they used?
- So... *what do we use in user code?*

Exit Ticket

- What is mutual exclusion?
- How do you ensure mutual exclusion around a section of code (i.e., create a critical section) using a lock?