Thread Synchronization

CS 360 Internet Programming

Daniel Zappala

CS 360 Internet Programming
Brigham Young University

Semaphores

- semaphore is a shared variable maintained by OS
  - contains an integer and a queue
  - value initialized $>= 0$
- `wait(s)`: wait for a signal on semaphore *s*
  - decrements semaphore, blocks if value < 0
  - if blocked, process put on the queue, suspends until signal is sent
- `signal(s)`: transmit a signal to semaphore *s*
  - increments semaphore
  - if value $<= 0$ then unblock someone
- `wait()` and `signal()` are atomic operations and cannot be interrupted

Types of Sempahores

- binary semaphore
  - only one process at a time may be in the critical section
- counting semaphore
  - a fixed number of processes > 0 may be in the critical section
- OS determines whether processes are released from queue in FIFO order or otherwise; usually FIFO in order to prevent starvation

Using Semaphores

```
semaphore s = 1;

void thread(int i) {
    while (true) {
        wait(s);
        /* critical section */
        signal(s);
       /* remainder */
    }
}
```

- semaphore protects critical section
- can set s to > 1 to let more than one process in the critical section
  - $s >= 0$: number that can enter
  - $s < 0$: number that are waiting

POSIX Semaphores

POSIX Semphores

```c
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t * sem);
int sem_trywait(sem_t * sem);
int sem_post(sem_t * sem);
```

- `sem_init()`: sets initial value of semaphore; `pshared = 0` indicates semaphore is local to the process
- `sem_wait()`: suspends process until semaphore is > 0, then decrements semaphore
- `sem_trywait()`: returns EAGAIN if semaphore count is $= 0$
- `sem_post()`: increments semaphore, may cause another thread to wake from `sem_wait()`

Example Code

- see example code `semaphore.cc`

▸ GitHub

Producer Consumer

Producer Consumer Problem

- one or more producers are generating data and placing them in a buffer
- one or more consumers are taking items out of the buffer
- only one producer or consumer may access the buffer at any time

Producer Consumer

```
vector buffer;
append(item) {
  buffer.append(item);
}
take() {
  return buffer.remove();
}
```

producer:

```
while (true) {
  item = produce();
  append(item);
}
```

consumer:

```
while (True) {
  item = take();
  consume(item);
}
```

Producer Consumer with Infinite Buffer

```
sem_t s, n;
sem_init(&s,0,1);
sem_init(&n,0,0);
```

producer:

```
while (True) {
   item = produce();
   sem_wait(&s);
   append(item);
   sem_post(&s);
   sem_post(&n);
}
```

consumer:

```
while (True) {
      sem_wait(&n);
      sem_wait(&s);
      item = take();
      sem_post(&s);
      consume(item);
}
```

Looking at the Code ...

1. *What is the purpose of semaphore s?*
2. *What is the purpose of semaphore n?*
3. *Why is semaphore s initialized to 1 but semaphore n is initialized to 0?*
4. *Why can the producer signal n every time an item is added to the buffer ?*
5. *Can the producer swap the signals for n and s?*
6. *Can the consumer swap the waits for n and s?*

Important Insights

- two purposes for semaphores
  - mutual exclusion: semaphore *s* controls access to critical section
  - signalling: semaphore *n* coordinates when the buffer is empty: consumer waits if buffer is empty, producer signals when buffer becomes non-empty
- avoid race conditions
  - *item* keeps a local copy of the data protected by the semaphore so that it can be accessed later
  - reduces amount of processing inside the critical section

Important Insights

- n: semaphore value is number of items in buffer
  - if $n == 0$, consumer must wait
  - can swap `sem_post(&n);` and `sem_post(&s);` in producer and be OK
  - can't swap `sem_wait(&n);` and `sem_wait(&s);` in consumer: otherwise consumer enters and then waits and deadlocks the producer!
- ordering of semaphore operations is important

Producer Consumer with Finite Buffer

```
sem_t s, n, e;
sem_init(&s,0,1);
sem_init(&n,0,0);
sem_init(&e,0,BUFFER_SIZE);
```

producer:

```
while (True) {
    produce();
    sem_wait(&e);
    sem_wait(&s);
    append();
    sem_post(&s);
    sem_post(&n);
}
```

consumer:

```
while (True) {
    sem_wait(&n);
    sem_wait(&s);
    take();
    sem_post(&s);
    sem_post(&e);
    consume();
}
```

Looking at the Code ...

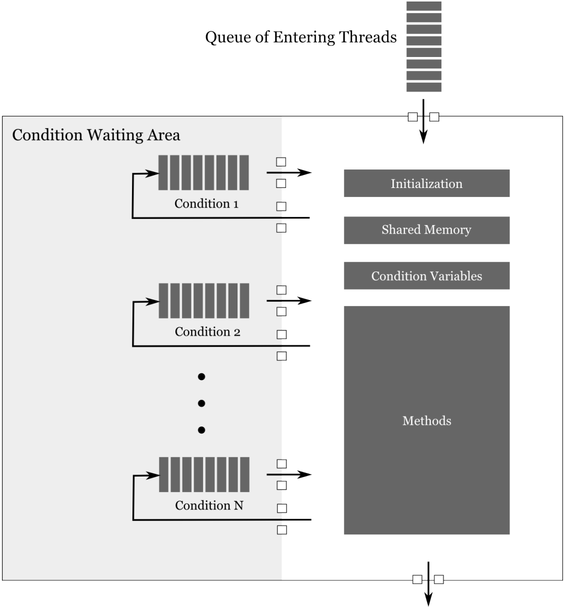
1. *What is the difference between semaphore e and semaphore n?*

Monitors

Monitor

- difficult to get semaphores right
  - match wait and signal
  - put in right order
  - scattered throughout code
- monitor: programming language construct
  - equivalent functionality
  - easier to control
  - mutual exclusion constraints can be checked by the compiler
  - used in versions of Pascal, Modula, Mesa
  - Java also has a Monitor object but compliance cannot be checked at compile time

Hoare Monitor



Hoare Monitor

- monitor can only be entered through methods
- shared memory can only be accessed by methods
- only one process or thread in monitor at any time
- may suspend and wait on a condition variable
- like object-oriented programming with mutual exclusion added in

Hoare Synchronization

- `cwait(c)`: suspend on condition c
- `csignal(c)`: wake up one thread waiting for condition c
  - do nothing if no threads waiting (signal is lost)
  - different from semaphore (number of signals represented in semaphore value)

Producer Consumer with a Hoare Monitor

```
vector buffer;
condition notfull, notempty;
```

```
append(item) {
  if buffer.full()
    cwait(notfull);
  buffer.append(item);
  csignal(notempty);
}
```

```
take() {
  if buffer.empty();
    cwait(notempty);
  item = buffer.remove();
  csignal(notfull);
  return item;
```

Producer Consumer with a Hoare Monitor

producer:

```
while (True) {
  item = produce();
  append(item);
}
```

consume:

```
while (True) {
  item = take();
  consume(item);
}
```

- advantages
  - moves all synchronization code into the monitor
  - monitor handles mutual exclusion
  - programmer handles synchronization (buffer full or empty)
  - synchronization is confined to monitor, so it is easier to check for correctness
  - write a correct monitor, any thread can use it

Lampson and Redell Monitor

- Hoare monitor requires that signaled thread must run immediately
    - thread that calls `csignal()` must exit the monitor or be suspended
    - for example, when `notempty` condition signaled, thread waiting must be activated immediately or else the condition may no longer be true when it is activated
    - usually restrict `csignal()` to be the last instruction in a method (Concurrent Pascal)
- Lampson and Redell
    - replace `csignal()` with `cnotify()`
    - `cnotify(x)` signals the condition variable, but thread may continue
    - thread at head of condition queue will run at some future time
    - must recheck the condition!
    - used in Mesa, Modula-3

Producer Consumer with a Lampson Redell Monitor

```
vector buffer;
condition notfull, notempty;
```

```
append() {
  while buffer.full()
    cwait(notfull);
  buffer.append(item);
  cnotify(notempty);
}
```

```
take() {
while buffer.empty()
  cwait(notempty);
  item = buffer.remove();
  cnotify(notfull);
  return item;
}
```

Lampson Redell Advantages

- allows processes in waiting queue to awaken periodically and reenter monitor, recheck condition
  - prevents starvation
- can also add `cbroadcast(x)`: wake up all processes waiting for condition
  - for example, append variable block of data, consumer consumes variable amount
  - for example, memory manager that frees k bytes, wake all to see who can go with k more bytes
- less prone to error
  - process always checks condition before doing work

What Can You Do?

- emulate a Lampson Redell Monitor with semaphores
    - create a class with private data only
    - use the same semaphore to protect all class methods
    - use semaphores to replace `cwait()` and `cnotify()`
    - use while loops to recheck conditions
- take your semaphores and move them inside the method call instead of outside of it