50004 - Operating Systems - Lecture 7

Oliver Killane

09/11/21

Lecture Recording

Lecture recording is available here

Semaphores

A blocking synchronisation mechanism invented by **Dijkstra**. Processes cooperate by means of signals:

- Process stops, waiting for a signal.
- Process continues if it has received a specific signal.

Semaphores allow **i** processes to be between **down** and **up**. They are special variables containing a value and a queue of waiting processes, they are accessible through the following atomic operations:

- **down(s)** wait to receive signal via **semaphore s**.

```
1  void down(Sema *s)
2  {
3      if (s->counter > 0) {
4          s->counter--;
5      } else {
6          Process *cur = current_process()
7          queue_add(cur)
8          suspend_process(cur)
9      }
10
11 }
```

- **up(s)** transmit a signal via **semaphore s**.

```
1  void up(Sema *s)
2  {
3      if (queue_empty(s->waiters))
4          s->counter++;
5      else
6          resume_process(queue_pop(s->waiters))
7  }
```

- **init(s,i)** initialise semaphore **s** with value **i**

```
1  typedef struct {
2      int counter;
3      queue waiters;
4  } Sema;
5
6  void init(Sema *s, int i)
7  {
8      s->counter = i;
9      queue_init(s->waiters);
10 }
```

Mutual Exclusion

```
Sema s;

int main(int arc, char **argv)
{
    sema_init(&s, 1);

    /* Start processes. */
    ...
}


void some_function(void)
{
    sema_down(&s);

    /* Critical region. */

    sema_up(&s);
}
```

A semaphore with an initial value of 1 acts similarly to a **lock** (**lock** also enforces that only the thread that acquired the access to the critical region can release it).

Ordering Events

```
Sema s;

int main(int argc, char **argv)
{
    sema_init(&s, 0);
}
```

```
void process_A(void)
{
    ...
    /* Critical region */
    sema_up(&s);
}
```

```
void process_B(void)
{
    ...
    sema_down(&s);
    /* Critical region */
}
```

No matter the order of processes being started, process B cannot continue to the critical region until process A has called **sema_up**.

Producer/Consumer

- **Producer Constraints**
  - Items can only be deposited if there is space.
  - Items can only be deposited in the buffer if mutual exclusion is ensured.
- **Consumer Constraints**
  - Items can only be fetched if the buffer is not empty.
  - Items can only be fetched from the buffer if mutual exclusion is ensured.

- **Buffer Constraints**
  Buffer can hold between 0 and N items.

```c
/* Global semaphores and buffer size. */
Sema item, space, mutex;
int N;

int main(int argc, char **argv)
{
    sema_init(item, 0);  /* Number of items available, blocks consumer if zero. */
    sema_init(space, N); /* Remaining space available, block producer if zero. */
    sema_init(mutex, 1); /* Mutual exclusion access to buffer. */

    /* Create processes. */
    ...
}
```

```c
void producer(void)
{
    for (;;) {
        /* Produce item. */
        sema_down(space);
        sema_down(mutex);
        /* Desposit item. */
        sema_up(mutex);
        sema_up(item);
    }
}
```

```c
void consumer(void)
{
    for (;;) {
        sema_down(item);
        sema_down(mutex);
        /* Fetch item. */
        sema_up(mutex);
        sema_up(space);
        /* Consume item. */
    }
}
```

Monitors

Monitors allow processes to wait on high level conditions. Entry procedures are called from outside the monitor, internal procedures only from inside (cannot access monitor data, only monitor procedures).

There is an (implicit) monitor lock, with one process being in the monitor at a time.

The high-level conditions could be:

- *Some space is available in the buffer*
- *Some data has arrived in the buffer*
- *There is an available resource*

Signals do not accumulate, if a condition is signalled and no processes/threads are waiting for it, the signal is lost.

Typically a language construct, and are not supported by **C**, however we can explicitly implement them (as in **PINTOS**).
Java uses synchronize, **wait** and **notify** with no condition variables.

Monitor Procedures

- **wait(condition)** Releases monitor lock, waits for **condition** to be signaled.
- **signal(condition)** Wakes up one process waiting for **condition**.
- **broadcast(condition)** Wakes up all processes waiting for **condition**.

Monitor Code

```
1  montior ProducerConsumer {
2      condition not_full, not_empty;
3      int count = 0;
4
5      entry procedure insert(item) {
6          while (count == N) wait(not_full);
7          insert_item(item);
8          count++;
9          signal(not_empty);
10     }
11
12     entry procedure remove(item) {
13         while (count == 0) wait(not_empty);
14         remove_item(item);
15         count--;
16         signal(not_full);
17     }
18 }
```

Synchronisation Summary

- **Lock**
  - Reader/Writer Locks.
  - Often exposed in monitor language construct.
  - Within a process.
  - 1 process/thread in the critical section.
- **Mutex**
  - Like lock, but works across processes.
- **Semaphore**
  - Like mutex, but can let N processes / threads pass & does not need the same process/thread to release as acquired.