OPERATING SYSTEMS

by
Marwa Yusuf

Lecture 7
Sunday 15-11-2020

Chapter 2 (2.3.4 to 2.3.5)

Processes and Threads

Sleep and Wakeup

- Both solutions require **busy waiting**.
  - Waste CPU time.
  - **Priority inversion problem**: lower priority process in its critical region.
- Use **sleep** and **wakeup** instead.

Producer-Consumer (Bounded-Buffer)

- Shared buffer between 2 processes (for simplification).
- Producer *sleeps* waiting for an empty slot (awakened by Consumer), Consumer *sleeps* waiting for a non-empty slot (awakened by producer).
- Leads to race condition.

Producer-Consumer (cont.)

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                item = produce_item( );         /* generate next item */
                if (count == N) sleep( );       /* if buffer is full, go to sleep */
                insert_item(item);              /* put item in buffer */
                count = count + 1;              /* increment count of items in buffer */
                if (count == 1) wakeup(consumer);   /* was buffer empty? */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                if (count == 0) sleep( );       /* if buffer is empty, got to sleep */
                item = remove_item( );          /* take item out of buffer */
                count = count - 1;              /* decrement count of items in buffer */
                if (count == N - 1) wakeup(producer);   /* was buffer full? */
                consume_item(item);             /* print item */
        }
}
```

Producer-Consumer (cont.)

• Producer	• Consumer
	read *count*
Insert item count++ wakeup (consumer)	
	count == 0? → sleep
…	
sleep	

Producer-Consumer (cont.)

• Producer	• Consumer
	read *count*
Insert item count++ Is consumer awake? → **wakeup** **waiting bit** = 1	
wakeup (consumer)	
	count == 0? → **wakeup waiting bit == 1?** don’t sleep
…	
sleep	

Producer-Consumer (cont.)

- What about more processes? 16, 32 or more? Shall we keep adding more **wakeup waiting bits**?

Semaphores (by Dijkstra)

- **Semaphore** variables to count the number of wakeups.
- **Up** and **down** (generalizations of wakeup and sleep)
  - **Down** checks semaphore value, if > 0, decrement and continue. Otherwise, sleep.
  - Checking, changing ans sleep are done as an atomic operation.
  - **Up** increments semaphore, if some process is sleeping waiting for it, it is waked up and it completed its **down** operation.
  - Incrementing and waking up also is atomic.
  - In the original paper **P** for **down** and **V** for **up**.

Semaphores (by Dijkstra)

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
        int item;

        while (TRUE) {                          /* TRUE is the constant 1 */
                item = produce_item();          /* generate something to put in buffer */
                down(&empty);                   /* decrement empty count */
                down(&mutex);                   /* enter critical region */
                insert_item(item);              /* put new item in buffer */
                up(&mutex);                     /* leave critical region */
                up(&full);                      /* increment count of full slots */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                          /* infinite loop */
                down(&full);                    /* decrement full count */
                down(&mutex);                   /* enter critical region */
                item = remove_item();           /* take item from buffer */
                up(&mutex);                     /* leave critical region */
                up(&empty);                     /* increment count of empty slots */
                consume_item(item);             /* do something with the item */
        }
}
```

Semaphores (by Dijkstra)

- **Binary Semaphores**.
- To make **up** and **down** operations atomic:
  - Make them as system calls where OS disables interrupts during them.
  - In multiple CPUs case, protect them with a lock variable using TSL or XCHG.
- Note that as the operations are short (few instructions) there is no problem in both solutions.
- Semaphores were used for **synchronization** (full and empty) and for **mutual exclusion** (mutex).