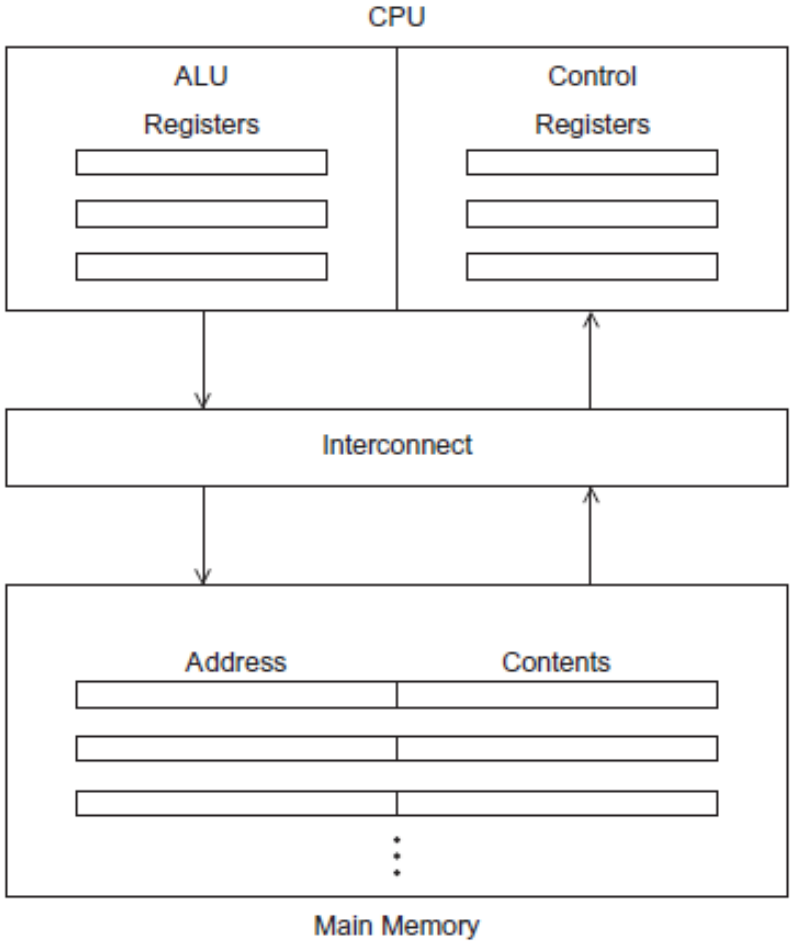
Østfold University College

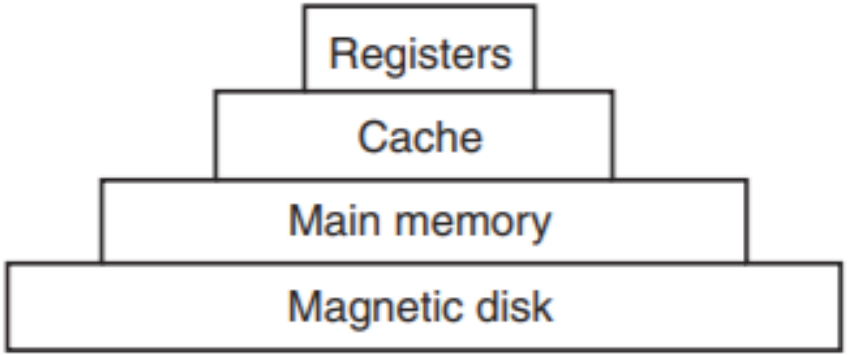
Thread Synchronization

- Implementation point of views

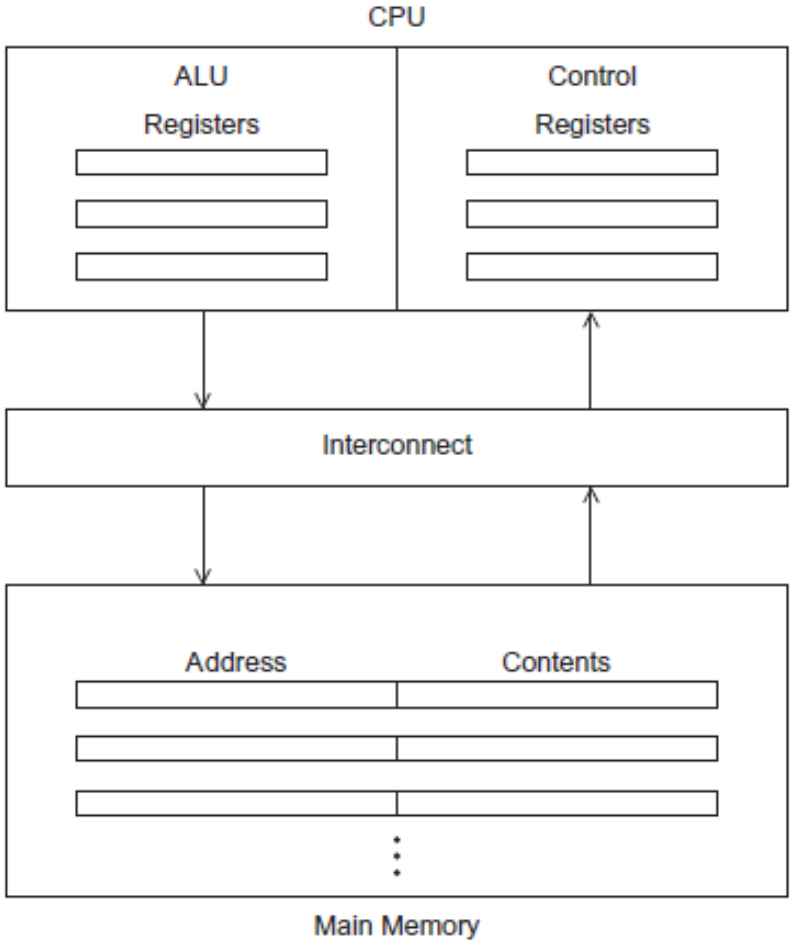
How an instruction is executed?



Computer Architecture



Typical memory hierarchy



Computer Architecture

Østfold University College

Machine executes a single instruction at a time, and each instruction operates on only a few pieces of data.

[1] Peter Pacheco, “An Introduction to Parallel Programming”, Elsevier, 2011, 2.1-2.3

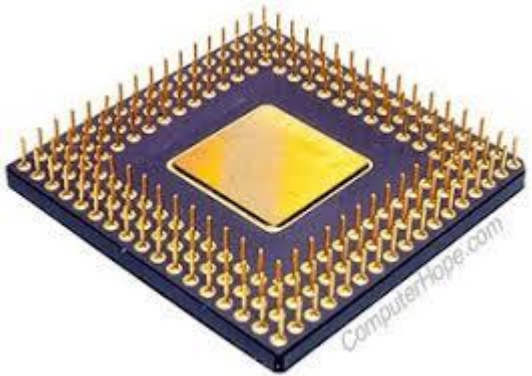
Read data from memory

Østfold University College

Data/instruction

RAM

fetched/read



CPU

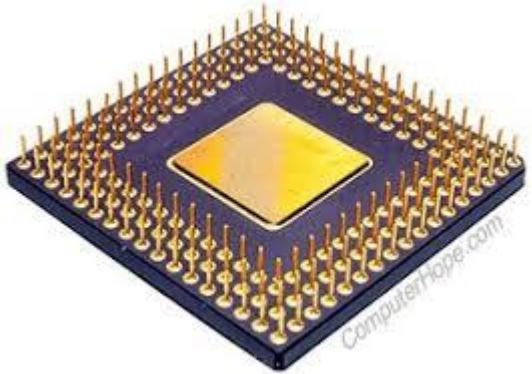
Write data to the memory

Østfold University College

RAM

Write/store

Data/instruction



CPU

Example

Østfold University College

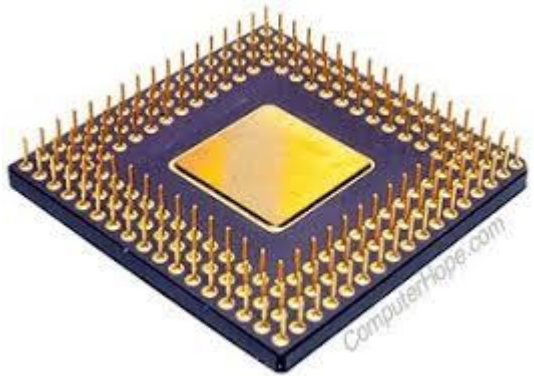
Data/instruction

count++;

R ← count

RAM

fetched/read

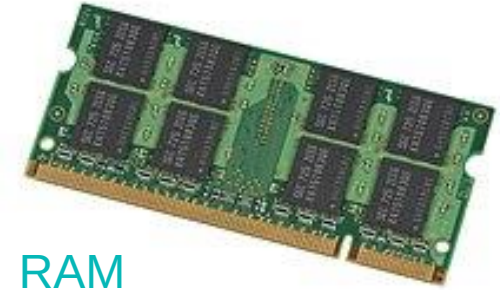


CPU

Example

Østfold University College

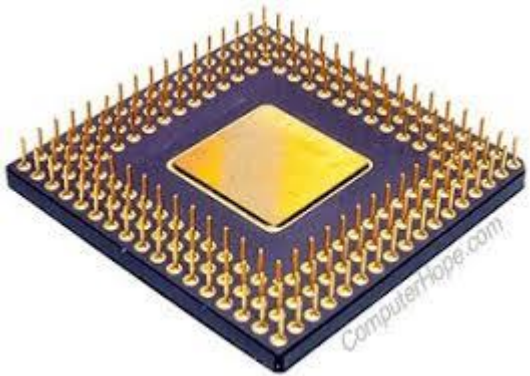
Data/instruction



RAM

count++;

R ← count
R ← R+1



CPU

Write data to the memory

Østfold University College

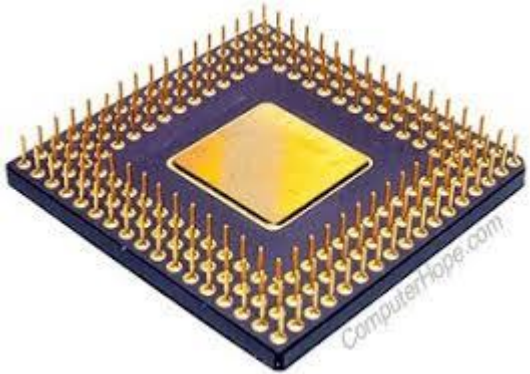
count++;

R ← count
R ← R+1
count ← R

Data/instruction

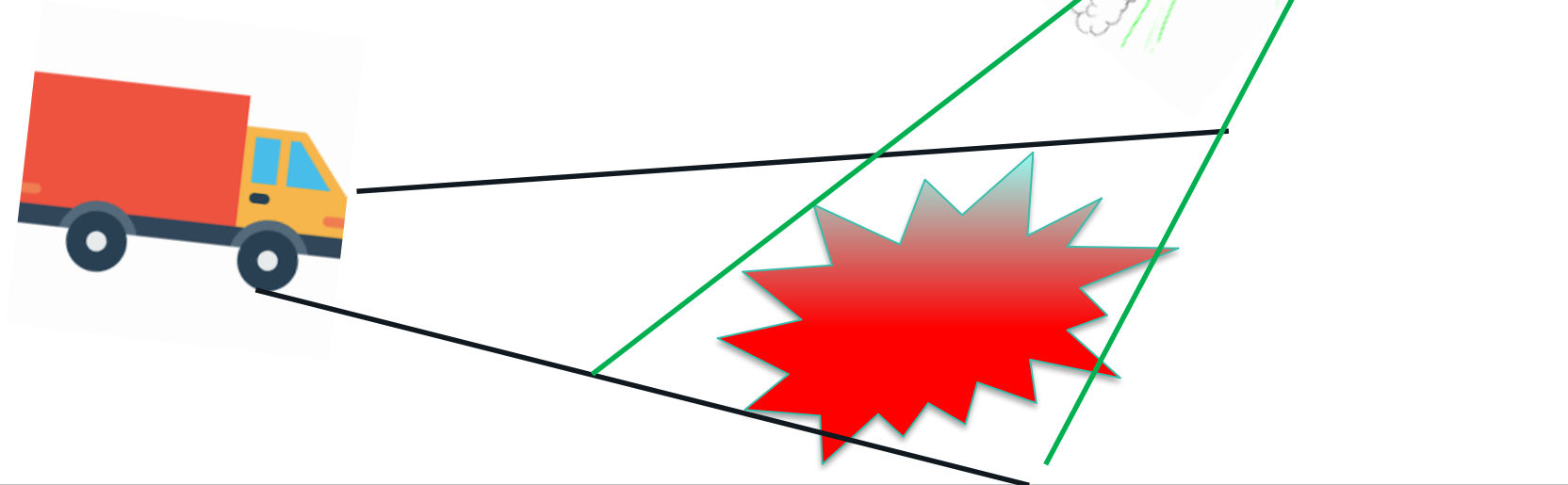
RAM

Write/store

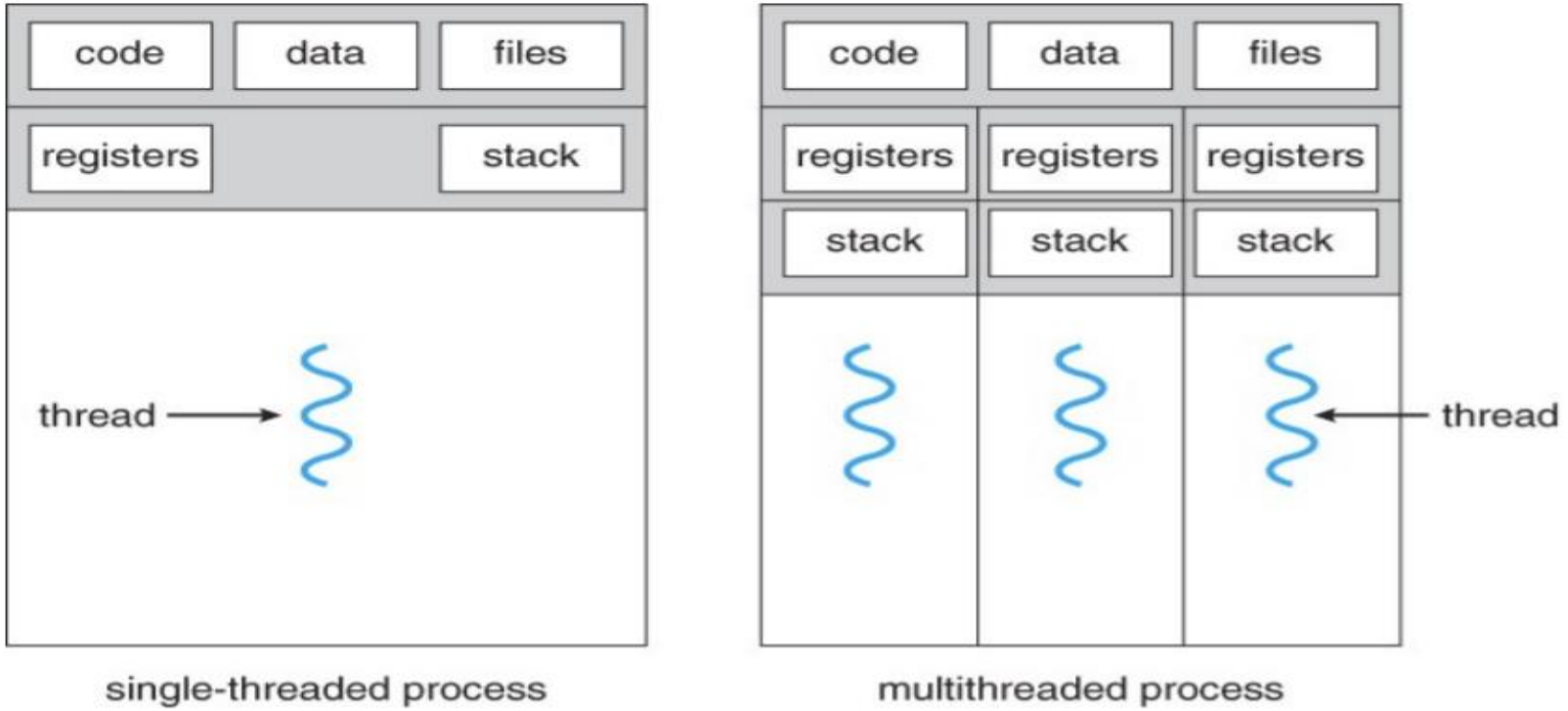


CPU

Race conditions



Threads in a process



single-threaded process

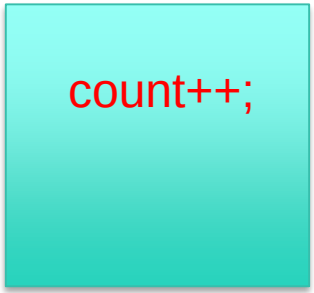
multithreaded process

Østfold University College

Critical section:

count = 5;

Thread 1



R1 ← count
R1 ← R1+1
count ← R1

Thread 2

R2 ← count
R2 ← R2 - 1
count ← R2

Østfold University College

Critical section:

count = 5;

Thread 1

count++;

R1 ← count
R1 ← R1+1
count ← R1

R2 ← count
R2 ← R2-1
count ← R2

Thread 2

count --;

count = 5;

Critical section:

count = 5;

Thread 1

count++;

R2 ← count
R2 ← R2-1
R1 ← count
R1 ← R1+1
count ← R2
count ← R1

Thread 2

count --;

count = 6;

Østfold University College

Critical section:

count = 5;

Thread 1

count++;

R1 ← count
R2 ← count
R2 ← R2-1
R1 ← R1+1
count ← R1
count ← R2

Thread 2

count --;

count = 4;

Mutex

Østfold University College

Critical section:

count = 5;

Thread 1

Thread 2

Østfold University College

Critical section:

count = 5;

Thread 1

count++;

Thread 2

count --;

Mutex

```
void* PrintMessage(void* ThreadId) {
      long tid;
      int i;

      tid = (long)ThreadId;
      for (i = 0; i < 5; i++) {
            printf("Hello World from Thread #%ld, count = %d!\n", tid, count);
            sleep(2);
            count++;
      }
```

Pthread calls for mutexes

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Pthread_mutex_trylock tries to lock mutex. If it fails it returns an error code, and can do something else.

Østfold University College

Semaphores

The Producer-Consumer Problem (Bounded Buffer)

- Two processes share the same buffer.
- Producer pushes items to the buffer.
- Consumer takes out items.

```
#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 Problem (2)

- What can be the problem?

- Signal missing
  - Shared variable: counter
  - Same old problem caused by concurrency
  - When consumer read count with a 0 but didn't fall asleep in time, then the signal will be lost

```
void producer(void)
{
        int item;

        while (TRUE) {
                item = produce_item( );
                if (count == N) sleep( );
                insert_item(item);
                count = count + 1;
                if (count == 1) wakeup(consumer);
        }
}
```

```
void consumer(void)
{
        int item;

        while (TRUE) {
                if (count == 0) sleep( );
                item = remove_item( );
                count = count – 1;
                if (count == N – 1) wakeup(producer);
                consume_item(item);
        }
}
```

Insert item into the buffer

Remove item from the buffer

```
void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item( );
        if (count == N) sleep( );
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
}
```

```
void consumer(void)
{
    int item;

    while (TRUE) {
        if (count == 0) sleep( );
        item = remove_item( );
        count = count – 1;
        if (count == N – 1) wakeup(producer);
        consume_item(item);
    }
}
```

Insert item into the buffer

Remove item from the buffer

```
void producer(void)
{
        int item;

        while (TRUE) {
                item = produce_item( );
                if (count == N) sleep( );
                insert_item(item);
                count = count + 1;
                if (count == 1) wakeup(consumer);
        }
}
```

```
void consumer(void)
{
        int item;

        while (TRUE) {
                if (count == 0) sleep( );
                item = remove_item( );
                count = count – 1;
                if (count == N – 1) wakeup(producer);
                consume_item(item);
        }
}
```

Insert item into the buffer

Remove item from the buffer

Semaphore (2)

- Solve producer-consumer problem
  - Full: counting the slots that are full; initial value 0
  - Empty: counting the slots that are empty, initial value N
  - Mutex: prevent access the buffer at the same time, initial value 1 (**binary semaphore**)

  - Synchronization/mutual exclusion

```
void producer(void)
{
        int item;

        while (TRUE) {
                item = produce_item( );
                if (count == N) sleep( );
                insert_item(item);
                count = count + 1;
                if (count == 1) wakeup(consumer);
        }
}
```

```
void consumer(void)
{
        int item;

        while (TRUE) {
                if (count == 0) sleep( );
                item = remove_item( );
                count = count – 1;
                if (count == N – 1) wakeup(producer);
                consume_item(item);
        }
}
```

Producer

Consumer

Insert item into the buffer

Remove item from the buffer

Using three mutexes: empty, full, mutex

```
void producer(void)
{
        int item;

        while (TRUE) {
                item = produce_item( );
                if (count == N) sleep( );
                insert_item(item);
                count = count + 1;
                if (count == 1) wakeup(consumer);
        }
}
```

```
void consumer(void)
{
        int item;

        while (TRUE) {
                if (count == 0) sleep( );
                item = remove_item( );
                count = count – 1;
                if (count == N – 1) wakeup(producer);
                consume_item(item);
        }
}
```

Producer

Consumer

Insert item into the buffer

Remove item from the buffer

Using three mutexes: empty, full, mutex

```
void producer(void)
{
        int item;

        while (TRUE) {
                item = produce_item( );
                if (count == N) sleep( );
                insert_item(item);
                count = count + 1;
                if (count == 1) wakeup(consumer);
        }
}
```

```
void consumer(void)
{
        int item;

        while (TRUE) {
                if (count == 0) sleep( );
                item = remove_item( );
                count = count – 1;
                if (count == N – 1) wakeup(producer);
                consume_item(item);
        }
}
```

Producer

Consumer

Insert item into the buffer

Remove item from the buffer

Using three mutexes: empty, full, mutex

R ← count = 0

Consumer

Deadlock

R ← count = 0

Consumer

```
item = produce_item( );
if (count == N) sleep( );
insert_item(item);
count = count + 1;
if (count == 1) wakeup(consumer);
```

Producer

Deadlock

```
R ← count = 0
```

Consumer

```
item = produce_item( );
if (count == N) sleep( );
insert_item(item);
count = count + 1;
if (count == 1) wakeup(consumer);
```

Producer

```
if (count == 0) sleep( );
```

Consumer

Deadlock

R ← count = 0

```
item = produce_item( );
if (count == N) sleep( );
insert_item(item);
count = count + 1;
if (count == 1) wakeup(consumer);
```

Consumer

Producer

```
if (count == 0) sleep( );
```

Consumer

Deadlock

Consumer: waits for producer to insert an item

Producer: waits for consumer to remove an item

R ← count = 0

```
item = produce_item( );
if (count == N) sleep( );
insert_item(item);
count = count + 1;
if (count == 1) wakeup(consumer);
```

Consumer

Producer

```
if (count == 0) sleep( );
```

Consumer

Deadlock

Consumer: waits for producer to insert an item

Producer: waits for consumer to remove an item

They wait forever!

Semaphore (1)

- Proposed by Dijkstra, introducing a new type of variable
- Atomic Action
  - A single, indivisible action
- Down (P)
  - Check a semaphore to see whether it's 0, if so, sleep; else, decrements the value and go on
- Up (v)
  - Check the semaphore
  - If processes are waiting on the semaphore, OS will choose one to proceed, and complete its **down**
  - **Consider as a sign of number of resources**

Semaphore (2)

- Solve producer-consumer problem
  - Full: counting the slots that are full; initial value 0
  - Empty: counting the slots that are empty, initial value N
  - Mutex: prevent access the buffer at the same time, initial value 1 (**binary semaphore**)

  - Synchronization/mutual exclusion

Østfold University College

Conditional Variables

Condition Variables

- Allows a thread to block if a condition is not met.
  - e.g. Producer needs to block if the buffer is full.
- Mutex make it possible to check if buffer is full
- Condition variable makes it possible to put producer to sleep if buffer is full
- Both are present in pthreads and are used together


Q&A