
# SISTEM OPERASI

Departemen Ilmu Komputer/ Informatika
Universitas Diponegoro
Semester Gasal 2018/ 2019

# Komunikasi Antar Proses

- Proses-proses saling berkomunikasi.
  - Bagaimana proses dapat mengirimkan informasi ke proses yang lain?
  - Bagaimana memastikan dua atau lebih proses tidak berebut pemakaian sumber daya?
  - Bagaimana mengatur urutan yang tepat? Misal proses A memberi data, proses B mencetak.
- Isu-isu tersebut berlaku dan diterapkan pula pada Thread

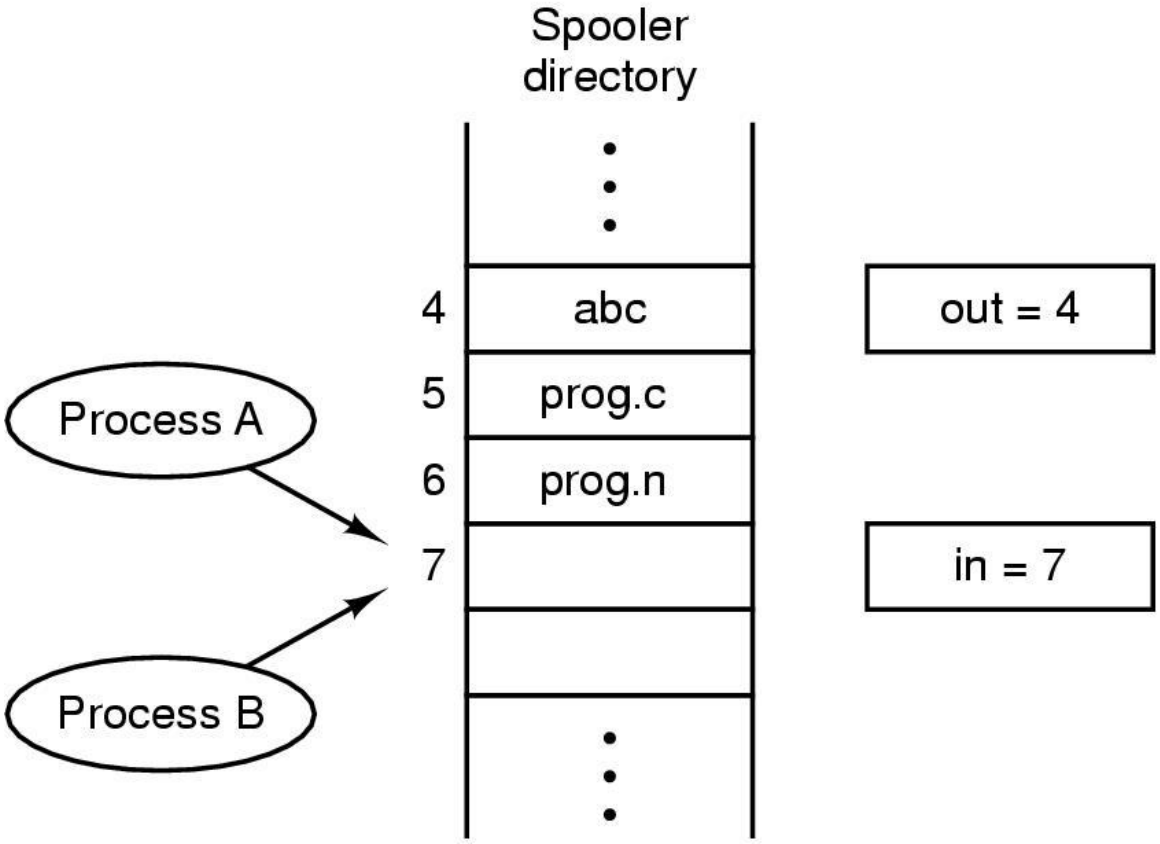
# Agenda

1. Race Condition
2. Pengaturan Mutual Exclusion
   - ❑ Solution with Busy Waiting
   - ❑ Solution with Non Busy Waiting

# Race Condition

- Akses-akses yang dilakukan (secara bersama-sama) ke data yang sama, dapat menyebabkan data menjadi tidak konsisten.
- ***Race Condition***: Situasi ketika beberapa proses mengakses dan memanipulasi suatu data secara bersamaan. Nilai terakhir data tersebut bergantung dari proses mana yang selesai terakhir.
- Untuk menghindari *Race Condition*, proses-proses tersebut harus disinkronkan (*synchronise*).

# Example of Race Conditions In Spooler Directory




- In is shared variable containing pointer to next free slot
- Out is shared variable pointing to next file to be printed

# ... Spooler Directory (2)

- Proses A membaca in = 7, menyimpannya dalam variable local next_free_slot
- Proses A time out, beralih ke proses B
- Proses B membaca in = 7, menyimpannya dalam variable local next_free_slot
- Proses B menaruh nama filenya di slot 7, meng-update in = 8. Proses B selesai dg printer, menunggu hasil pencetakan.

# … Spooler Directory (3)

- Ketika Proses A melanjutkan, dia melihat ke isi next_free_slot lalu menaruh nama filenya di slot 7, menghapus nama file yang ditaruh oleh B, dan mengupdate in = 8.
- Spooler directory menjadi tidak konsisten, namun printer tidak menyadari adanya permasalahan disini
- Kenyataannya proses B tidak akan pernah memperoleh hasil output (karena *tertimpa* update dari A)

# Solusi Race Condition

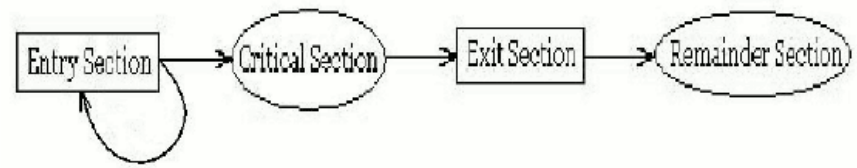
- Membuat suatu abstraksi → Critical Section
- **Critical Section**

  Bagian kode/ program yang ketika dieksekusi akan mengakses data yang digunakan bersama-sama (dengan proses lain) sehingga dapat membawa proses itu ke bahaya ***race condition***
- Contoh aktifitas yang dapat membahayakan antara lain:
  - Mengubah variable,
  - meng-update suatu tabel,
  - menulis ke suatu file, dll

# Struktur umum Critical Section

```
do {
        entry section
           critical section
        exit section
           remainder section
        } while (1);
```

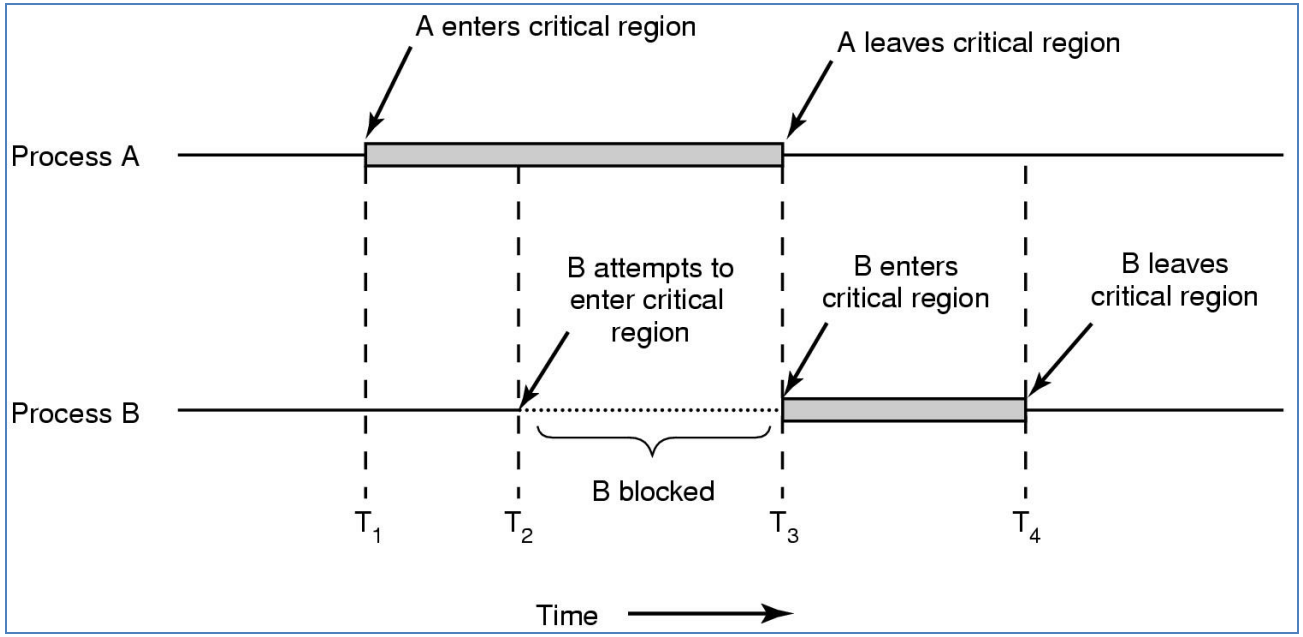
- Entry section : kode untuk masuk ke dalam critical section
- Critical section : segmen kode yang hanya mengijinkan satu proses yg dapat dieksekusi pada satu waktu
- Exit Section : Akhir dari Critical Section; mengijinkan proses lain
- Remainder Section : Kode lain setelah keluar critical section




*Gambar Proses Critical Section*

# Mutual Exclusion

- Karena keberadaan critical section dapat menimbulkan race condition, maka dua proses tidak boleh berada di critical section secara bersamaan (disebut: mutual exclusion)



# Solusi Yang Baik?

- Namun mutual exclusion saja belum cukup.
- Agar kerjasama antar proses dapat menjadi solusi yang baik bagi *race condition*, ada tiga persyaratan yang harus dipenuhi :

1. **Mutual Exclusion**
   - Tidak boleh ada dua proses secara bersamaan dalam critical section
2. **Progress** (terjadi kemajuan)
   - Tidak boleh ada proses diluar critical section yang dapat melakukan blok proses lain → menghindari *deadlock*
3. **Bounded Waiting** (ada batas waktu tunggu)
   - Tidak boleh ada proses yang menunggu selamanya untuk memasuki critical section → menghindari *starvation*

# REVIEW

1. Apa yang dimaksud dengan **race condition** apa hubungannya dengan **critical section?**
2. Jelaskan 3 syarat untuk menyelesaikan masalah *race condition* yang baik.

# Agenda

1. Race Condition
2. Pengaturan Mutual Exclusion
   - ❑ **Disabling Interrupt**
   - ❑ Solution with Busy Waiting
     - Software Solution
     - Hardware Solution
   - ❑ Solution with Non Busy Waiting

# Interrupt disabling (1)

- Proses **mematikan interupsi ke prosesor dan segera memasuki critical section**
- Proses segera **mengaktifkan interupsi ketika meninggalkan critical section**

```
While(true)
    {
    <disable interrupt>
    <critical section>
    <enable interrupt>
    <remainder>
    }
```

- Efek yang diharapkan :
  - Prosesor tidak dapat beralih ke proses lain. Krn penjadwal tidak beroperasi maka tidak terjadi alih proses
  - Proses dapat memakai sharing-memori tanpa takut intervensi proses lain krn memang tidak akan ada proses lain yg dieksekusi saat itu.

# Interrupt disabling (2)

- Kelemahan utama:
  - Bila proses yg mematikan interupsi crash, proses tidak akan pernah menghidupkan interupsi kembali → kematian seluruh sistem
  - Jika terdapat 2 prosesor/ lebih, pematian interupsi hanya berpengaruh pada prosesor yg mengeksekusi instruksi itu → prosesor lain masih bebas memasuki critical section → kesimpulannya, tidak cocok untuk sistem multiprosesor
- Cocok diterapkan untuk proses2 kernel (uniprosesor) yg sangat kritis yg tidak boleh diintervensi proses lain, mis : saat memodifikasi PCB

# Agenda

1. Race Condition
2. Pengaturan Mutual Exclusion
   - ❑ Disabling Interrupt
   - ❑ **Solution with Busy Waiting**
     - Software Solution
     - Hardware Solution
   - ❑ Solution with Non Busy Waiting

# *Solution With Busy Waiting*

- **Busy waiting?**

  Proses menggunakan CPU namun “tidak bekerja” karena menunggu giliran masuk ke *critical section*

# Lock Variable (1)

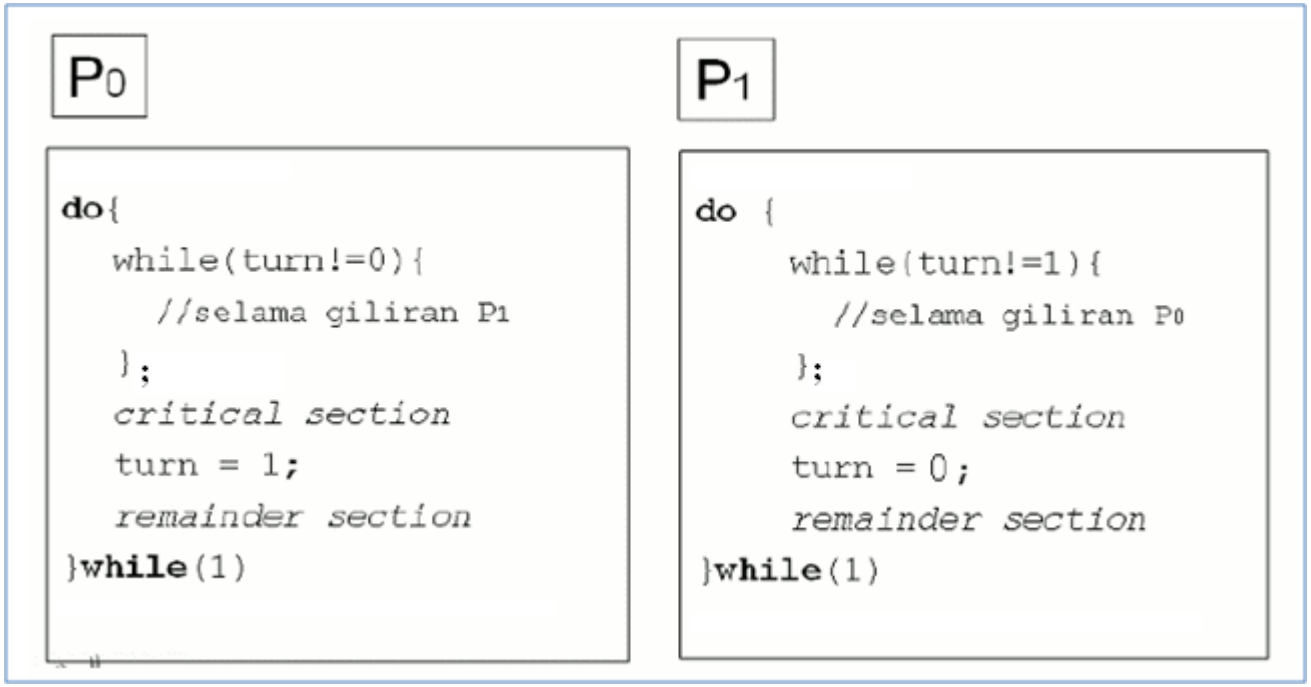
- Software solution. **Single shared (lock) variable**, initially 0 (tidak ada yang mengakses critical section)
- Jika proses perlu masuk critical section dia harus memeriksa nilai lock
  - Jika 0, proses mengubahnya menjadi 1, masuk critical section, keluar dari critical section, mengubah jadi 0.
  - Jika 1, proses menunggu hingga bernilai 0.
- Berpotensi race condition
  - Sesaat setelah membaca lock =0, time out, proses lain melihat lock masih 0
  - Dua proses dapat mengakses critical section

# Strict Alternation dengan "turn"

- **Shared variable**, *lock variable* → turn, nilai awal turn = 0
- Masalah:
  1. bila P1 tidak menggunakan gilirannya → Tidak memenuhi syarat ke-2 (***progress)***
  2. Bila P1 selesai/ error padahal belum keluar dari critical section, belum mengeset turn=0 →Tidak memenuhi syarat ke-3 (***bounded waiting)***

Global variable is a Shared variable

turn = 0



# Strict Alternation dengan "Flag"

- Menggunakan global variabel array bertipe boolean → flag[0] dan flag[1]
- **Flag[x]** = true → Proses x meminta akses ke Critical section
- Proses yg menge-set flagnya menjadi **true** harus melihat apakah sudah ada proses lain yang menjalankan critical section (flag[teman] = true), bila sudah ada maka proses ini harus menunggu (mempersilahkan)
- Mengantisipasi masalah yg muncul pada Algoritma Sebelumnya, dg cara :
  1. Problem menunggu giliran → menunjuk diri sendiri, flag[x] = true
  2. Crash dalam critical section → flag otomatis akan terdeteksi sebagai false

Global variable

flag

[0] = false

[1] = false

**P0**

```
do{
   flag[0] = true;
   while(flag[1])
   {
     //selama P1 memerlukan
   }
   critical section
   flag[0] = false;
   remainder section
}while(1);
```

**P1**

```
do{
   flag[1] = true;
   while(flag[0])
   {
     //selama P0 memerlukan
   }
   critical section
   flag[1] = false;
   remainder section
}while(1);
```

# … "Flag"

- **Masalah** : bila keduanya secara bersamaan menginginkan critical section (bersamaan melakukan set flag)
  - Saling menunggu, dan saling mempersilahkan → tidak ada yg mengakses critical section → Melanggar syarat ***progress*** dan ***bounded waiting***

# Algoritma Petterson

- **G.L Petterson**, pada tahun 1981.
- Menggabungkan penggunaan **shared variable** turn dan flag
  - **turn** : menunjukkan giliran proses yg diijinkan memasuki Critical Section (Nilai turn hanya bisa 0 atau 1) → shared variable
  - **flag[x]** : tanda suatu proses meminta akses ke Critical Section
- Memenuhi ketiga syarat yang dibutuhkan
- Dapat digeneralisasi untuk n Proses

turn = 0

flag

[0] = false

[1] = false

**P0**

```
do{
   flag[0] = true;
   turn = 1;

   while(flag[1])&&(turn==1
   ){
     //selama P1 memerlukan
   dan giliran P1
   }
   critical section
   flag[0] = false;
   remainder section
}while(1)
```

**P1**

```
do{
   flag[1] = true;
   turn = 0;

   while(flag[0])&&(turn==0
   ){
     //selama P0 memerlukan
   dan giliran P0
   }
   critical section
   flag[1] = false;
   remainder section
}while(1)
```

# Solusi Hardware

- Dilakukan dg cara mengunci bus, shg prosesor2 lain tidak dapat menggunakan bus.
- Contoh instruksi atomik dengan bantuan hardware:
  - TSL (test and set lock)
  - XCHG (exchange) → intel x86

# TSL (1)

- Menggunakan shared variable: LOCK
- Komputer dengan banyak (multi) prosesor disertai instruksi : TSL RX, LOCK
  - Instruksi tersebut membaca isi word memori *lock* ke dalam register RX kemudian menyimpan nilai *nonzero* pada alamat memori lock
  - *Guaranted to be indivisible* – tidak ada prosesor lain yang dapat mengakses word memori tersebut hingga instruksi ini selesai → hardware level

# TSL (2)

```
enter_region:
        TSL REGISTER,LOCK                | copy lock to register and set lock to 1
        CMP REGISTER,#0                  | was lock zero?
        JNE enter_region                 | if it was non zero, lock was set, so loop
        RET | return to caller; critical region entered


leave_region:
        MOVE LOCK,#0                     | store a 0 in lock
        RET | return to caller
```

- Masuk dan keluar Critical Region menggunakan TSL
- Instruksi bersifat Atomic, diimplementasikan pada hardware

# TSL (3)

- Keunggulan:
  - Sederhana dan mudah diverifikasi
  - Dapat diterapkan ke sembarang jumlah prosesor, baik tunggal maupun multiprosesor yg memakai memori bersama
  - Dapat digunakan utk mendukung banyak Critical Section, masing2 CS didefinisikan dg suatu variable.
- Kelemahan:
  - Masih merupakan metode dg busy waiting
  - Adanya busy waiting memungkinkan deadlock dan starvation

# REVIEW

1. Jelaskan bagaimana Algoritma Petterson dapat memenuhi 3 persyaratan solusi race condition.
2. Jelaskan keunggulan Test and Set Lock (TSL) dibandingkan algoritma Petterson.
3. Jelaskan kekurangan: baik pada Algoritma Petterson maupun TSL.

# Agenda

1. Race Condition
2. Mutual Exclusion
   - ❑ Solution with Busy Waiting
   - ❑ **Solution with Non Busy Waiting**
     - ✓ Sleep and Wake-up
     - ✓ Semaphores
     - ✓ Monitors

# Sleep and Wakeup

- Menggunakan Primitive → **sleep** and **wakeup**
  - Let a process **blocked**, rather than waste CPU time, if not allowed to enter its Critical Section
  - Sleep → system call to **block** caller
  - Wakeup(p) → system call to **wakeup** p

# Producer-consumer sleep dan wakeup

```
#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 = produc tem( );                 /* 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 */
     }
}
```

Sinyal *wakeup* tidak tertangkap

# Problem

- Deadlock
  - Konsumen mengambil item, krn kosong maka bersiap ***sleep***,
  - Kebetulan saat itu penjadwal menghentikan konsumen, beralih ke produsen
  - Produsen menambah item, menaikkan ***count*** jadi 1, shg harus membangunkan konsumen, mengirimkan sinyal ***wakeup***
  - Sayangnya, konsumen belum benar-benar *sleep*
  - Sehingga sinyal wakeup **tidak tertangkap**
  - Konsumen melanjutkan benar-benar ***sleep***
  - Produsen terus menambah item hingga penuh, lalu ***sleep***
  - Keduanya..... Sleep
- Penyebabnya karena variable ***count*** tidak dijaga
  - Seharusnya, Count dapat naik jadi 1 hanya apabila konsumen benar-benar sleep

# Semaphore (1)

- **Semaphores** : Variabel integer dengan dua operasi atomik
  - *Down* : if 0, then go to **sleep** (blocked);
    if >0, then decrement value
  - *Up* : increment value, and let a sleeping process to perform a *down*
- Mengatasi permasalahan busy waiting, dg menempatkan proses ke antrian blocked.
- Up dan down di implementasikan menggunakan **system call** dg men-disable interrupt

# Semaphore (2)

- Ada dua jenis variable semaphore:
  - **Binary Semaphore**, untuk mutual exclusion
  - **Counting Semaphore**, untuk sinkronisasi

# Semaphore (3)

```
#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 */
    }
}
```

# Semaphore (4)

- Untuk menjamin **mutual exclusive** dan **sinkronisasi** pada kasus produsen konsumen, maka digunakan 3 variabel semaphore:
  - Mutex → mutual exclusion, awal=1
    - Menjamin hanya satu proses yang dapat mengakses buffer
    - Bila mutex=0, blocked, hingga ada yang melakukan up(&mutex)
  - Empty → sinkronisasi, jumlah slot kosong, awal=N
    - Memastikan producer berhenti ketika slot penuh
    - Setiap menaruh item akan melakukan **Down(&empty)**
    - Bila empty=0, akan di blok, hingga consumer melakukan **Up(&empty)** setelah me-remove item
  - Full → sinkronisasi, jumlah slot terisi, awal=0
    - Memastikan consumer berhenti ketika slot kosong
    - Setiap ambil item akan melakukan **Down(&full)**
    - Bila full=0, akan di blok, hingga Producer melakukan **Up(&full)** setelah insert item

# Problem Semaphore

- Jika di implementasikan pada sistem yang kompleks akan menjadi terlalu rumit
  - Rawan, berefek **deadlock.** Contoh, jika pada **producer** terbalik urutan down(&empty) dan down(&mutex)
    - Setelah producer berhasil down(&mutex), namun karena buffer penuh sehingga gagal down(&empty), maka **blocked** → mutex = 0
    - Sehingga ketika Consumer mencoba down(&mutex) akan gagal, maka **blocked**

# Monitors (1)

- **Process can call monitors**
  - but cannot access the monitor's internal data structure from procedures declared outside
- Advantage
  - **Automatic mutual exclusion**, so paralel programming less error prone than with **semaphores**
- Disadvantage
  - Not implemented in C, C++, etc
  - Design (like semaphores) for mutual exclusion on CPU having shared memory → not for distributed system, each cpu with private memory

# Monitors (2)

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

- Outline of producer-consumer problem with monitors
  - only one monitor procedure active at one time
  - buffer has *N* slots

  Condition variables with *wait* and *signal*

# REVIEW

1. Jelaskan solusi producer-consumer menggunakan semaphore.
2. Jelaskan solusi producer-consumer menggunakan monitor.