IF3055 – Sinkronisasi & Komunikasi antar Proses

Henny Y. Zubir
STEI - ITB

Ikhtisar

- Critical Section
- Mutual Exclusion
- Algoritma Sinkronisasi
- Komunikasi antar Proses
- Permasalahan Klasik

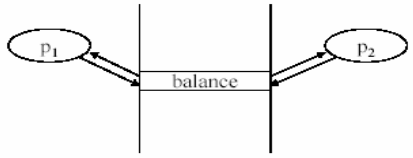
Latar Belakang

- Akses konkuren terhadap data yg dipakai bersama dapat mengakibatkan inkonsistensi data
- Akses memori bersama dapat mengakibatkan terjadinya **kondisi berlomba** (***race condition***)
- Pemeliharaan konsistensi data memerlukan mekanisme khusus utk menjamin eksekusi antar proses yg bekerjasama berlangsung dengan benar (terurut)

Game Sederhana

- 2 pemain (produsen dan konsumen)
- Aturan permainan
  - **Produsen**: memproduksi 1 spidol per iterasi
    - Langkah 1: naikkan counter
    - Langkah 2: taruh spidol di meja
  - **Konsumen**:
    - Langkah 1: periksa counter jika 0
    - Langkah 2a: jika counter 0, kembali ke tahap 1
    - Langkah 2b: jika counter tidak 0, ambil spidol dari meja
    - Langkah 3: turunkan counter
  - **OS**
    - Menentukan siapa yg jalan dan siapa yg berhenti

Kondisi Berlomba



Dua/lebih proses ingin mengakses ruang memori yg digunakan bersama

Kode utk p1

```
  . . .  . . .
balance = balance + amount;
 . . .  . . .
```

Kode utk p2

```
balance = balance - amount;
```

Kode utk p1

```
load     R1, balance
load     R2, amount
add      R1, R2
store    R1, balance
```

Kode utk p2

```
load     R1, balance
load     R2, amount
sub      R1, R2
store    R1, balance
```


Critical Section (1)

```
Process {
   while (true) {
      ENTER CS
      Akses variabel bersama; // Critical Section;
      LEAVE CS
      Lakukan pekerjaan lainnya      }
}
```

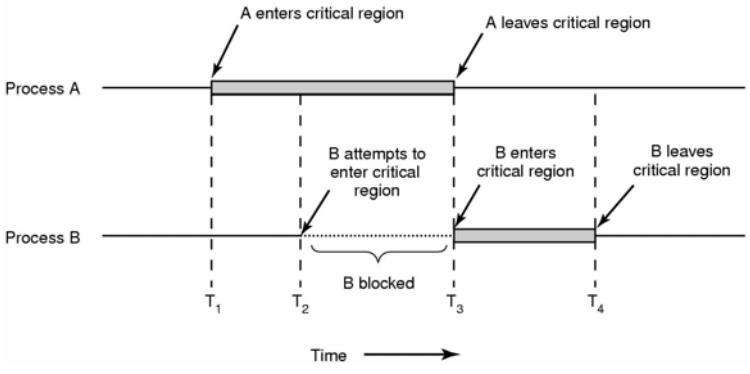
- Terjadi perlombaan beberapa proses untuk mengeksekusi **critical section** (CS) → hasil eksekusi tidak bisa diprediksi (indeterminate)
- CS bisa berada pada kode dan proses yg berbeda

Critical Section (2)

- Persyaratan:
  - **Mutual Exclusion:** tidak ada proses lain yg boleh masuk ke CS selama masih ada proses lain di dalamnya
  - **Progress:** jika tidak ada proses yg sedang berada di CS dan ada proses yg mencoba masuk ke CS, maka proses tsb harus diberi akses ke CS
  - **Bounded Waiting:** proses yg menunggu akses ke CS tidak boleh menunggu dlm waktu yg tak terbatas
  - **Kecepatan dan banyak CPU:** tidak ada asumsi mengenai kecepatan atau banyaknya CPU

Critical Section (3)

- Mutual Exclusion (Mutex) menggunakan CS



Implementasi Mutex: Busy Waiting

- Kemungkinan solusi:
  - Disabling Interrupts
  - Variabel Lock
  - Strict Alternation
  - Solusi Peterson
  - Instruksi TSL

Disabling Interrupt (1)

- Bagaimana cara kerjanya?
  - Disable semua interrupt tepat sebelum masuk ke CS dan di-enable kembali tepat sebelum keluar

Kode utk p1

```
disableInterrupts();
balance = balance + amount;
enableInterrupts();
```

Kode utk p2

```
disableInterrupts();
balance = balance - amount;
enableInterrupts();
```

Disabling Interrupt (2)

- Jika interrupt dlm keadaan disable, tdk ada clock interrupt yg akan terjadi → tidak akan terjadi process switching
- Masalah:
  - Bgm jika proses lupa utk me-enable-kan kembali interrupt?
  - Multiprosesor? (disabling interrupts hanya berpengaruh pd satu CPU)
- Hanya digunakan utk internal OS

Variabel Lock

```
shared boolean lock = FALSE;
shared double balance;
```

Kode utk p1	**Kode utk p2**
/* Ambil lock */	/* Ambil lock */
`while(lock) ;`	`while(lock) ;`
`lock = TRUE;`	`lock = TRUE;`
/* Eksekusi CS */	/* Eksekusi CS */
`balance = balance + amount;`	`balance = balance - amount;`
/* Lepaskan lock */	/* Lepaskan lock */
`lock = FALSE;`	`lock = FALSE;`

- Menimbulkan permasalahan kondisi berlomba yg baru

Strict Alternation

Kode utk p1

```
while(TRUE){
  while(turn!=2)/*tunggu giliran*/
  critical_section();
  turn = 2; /* gantian */
  non_critical_section();
}
```

Kode utk p2

```
while(TRUE) {
  while(turn!=1)/*tunggu giliran*/
  critical_section();
  turn = 1; /* gantian */
  non_critical_section();
}
```

- Menyelesaikan masalah mutex, tapi tidak memenuhi kriteria progres

Solusi Peterson

```
#define FALSE 0
#define TRUE  1
#define N     2                    /* banyaknya proses*/

int turn;                          /* giliran siapa? */
int interested[N]                  /* diinisialisasi 0 (FALSE) */

void enter_region(int process){    /* proses 1 atau 0 */
  int other;                       /* nomor proses lainnya */

  other = 1 - process;             /* proses lainnya */
  interested[process] = TRUE;      /* anda tertarik */
  turn = process;                  /* set flag */
  while (turn==process && interested[other]==TRUE); /* null statement */
}

void leave_region(int process) {   /* siapa yg keluar */
   interested[process] = FALSE;
}
```

Instruksi Test-Set-Lock (TSL)

- Melakukan test dan set secara atomik
- Membutuhkan dukungan hardware

```
enter_region:
  TSL REGISTER, LOCK      | salin lock ke register dan set lock=1
  CMP REGISTER, #0        | lock=0?
  JNE enter_region        | jika tdk 0, lock diset, dan loop
  RET                     | kembali ke caller; masuk ke CS

leave_region:
  MOVE LOCK, #0           | simpan 0 pada lock
  RET                     | kembali ke caller
```


Implementasi Mutex dengan TSL

- Menggunakan instruksi TSL utk mengimplementasikan mutex:
  - `mutex_lock`
  - `mutex_unlock`

```
mutex_lock:
    TSL REGISTER, MUTEX    | salin mutex ke register dan set mutex=1
    CMP REGISTER, #0       | mutex=0?
    JZE ok                 | jika 0, kunci mutex dibuka, dan kembali
    CALL thread_yield      | mutex sibuk, jadwalkan thread lainnya
ok: RET                    | kembali ke caller; masuk ke CS

mutex_unlock:
  MOVE MUTEX, #0           | simpan 0 ke mutex
  RET                      | kembali ke caller
```

SLEEP dan WAKEUP

- Masalah dengan solusi sebelumnya:
  - Busy waiting
  - Membuang waktu CPU
  - Priority Inversion:
    - Proses dgn prioritas tinggi menunggu proses dgn prioritas yg lebih rendah keluar dari critical section
    - Proses dgn prioritas rendah tidak pernah bisa dieksekusi karena proses dgn prioritas tinggi tidak di-block
- Solusi: sleep dan wakeup
  - Jika diblokir, tidur dulu
  - Bangun jika sudah bisa mencoba kembali masuk ke critical section

Problem Produsen-Konsumen

```
#define N 100                                   /* banyaknya slot di buffer*/
int count=0;                                    /* banyaknya item di buffer*/

void producer(void){
  int item;
  while(TRUE) {                                 /* ulang terus */
    item=produce_item();                        /* produksi item berikutnya */
    if(count==N) sleep();                       /* jika buffer penuh, tidur dulu */
    insert_item(item);                          /* taruh item di buffer */
    count=count+1;                              /* tingkatkan counter item di buffer */
    if(count==1)wakeup(consumer); } }           /* apakah buffer kosong? */

void consumer(void){
  int item;
  while(TRUE) {                                 /* ulang terus */
    if(count==0) sleep();                       /* jika buffer kosong, tidur dulu */
    item=remove_item();                         /* ambil item dari buffer */
    count=count-1;                              /* kurangi counter item di buffer */
    if(count==N-1)wakeup(producer);             /* apakah buffer penuh? */
    consume_item(item); } }                     /* gunakan item */
```

Semaphore

- **Semaphore**: variabel global yg merepresentasi-kan banyaknya sumberdaya yg dipakai bersama
- Memiliki 2 operasi:
  - **down** (atau P): digunakan utk memperoleh sumberdaya dan mengurangi count
  - **up** (atau V): digunakan utk melepas sumberdaya dan meningkatkan count
- Operasi semaphore bersifat atomik (indivisible)
- Memecahkan masalah wakeup-bit

Semaphore: Solusi Permasalahan Prod-Kon

```
#define N 100                                   /* banyaknya slot di buffer*/

typedef int semaphore;                          /* tipe semaphore */
semaphore mutex=1;                              /* mengontrol akses ke critical section */
semaphore empty=N;                              /* banyaknya slot buffer yg kosong */
semaphore full=0;                               /* banyaknya slot buffer yg penuh */

void producer(void){
  int item;
  while(TRUE) {                                 /* ulang terus */
    item=produce_item();                        /* produksi item */
    down(&empty);                               /* kurangi counter buffer kosong */
    down(&mutex);                               /* masuk ke critical section */
    insert_item(item);                          /* taruh item di buffer */
    up(&mutex);                                 /* keluar dari critical section */
    up(&empty); } }                             /* tambah counter buffer penuh */

void consumer(void){
  int item;
  while(TRUE) {                                 /* ulang terus */
    down(&full);                                /* kurangi counter buffer penuh */
    down(&mutex);                               /* masuk ke critical section */
    item=remove_item();                         /* ambil item dari buffer */
    up(&mutex);                                 /* keluar dari critical section */
    up(&full);                                  /* tambah counter buffer kosong */
    item=produce_item(); } }                    /* gunakan item */
```

Operasi Up dan Down

- Operasi Down(atau P):

```
down(S) {
  while (S <= 0) { }; // no-op
  S= S-1; }
```

- Operasi Up (atau V):

```
up(S) {  S++; }
```

- Counting semaphores: 0..N
- Binary semaphores: 0,1
- Mutual exclusion dpt diimplementasikan dgn binary semaphore
  - state: **lock** dan **unlock**

Implementasi Semaphore dgn Busy Waiting

- Menggunakan mutex untuk mengimplementasikan counter semaphore
  - up
  - down

```
void down() {
  mutex_lock(m);
  while(count==0) {
    mutex_unlock(m);
    yield();
    mutex_lock(m);
  }
  count = count - 1;
  mutex_unlock();
}
```

```
void up(){
  mutex_lock(m);
  count = count + 1;
  mutex_unlock(m);
}
```

Implementasi Semaphore dgn Sleep&Wakeup

```
type Semaphore = record
{  value:integer;
   L: list of processes; }
Semaphore S;

down(S) {                          up(S){
S.Value = S.value - 1;               S.value = S.value + 1;
if(S. value < 0)                     if(S.value > 0)
{ tambahkan proses P ke S.L;         { pindahkan proses P dari S.L;
  block;                               wakeup(P);
};                                   };
```


Busy Waiting vs Sleep&Wakeup

- Busy waiting (spinlock)
  - menghabiskan siklus CPU
- Sleep&Wakeup (blocked lock)
  - overhead context switch
- Solusi hybrid (spin-block)
  - gunakan spinlock jika waktu tunggu lebih singkat dari waktu utk context switch
  - gunakan sleep & wakeup jika waktu tunggu lebih lama dari waktu utk context switch

Kemungkinan Deadlock dgn Semaphore

- Contoh: menggunakan 2 semaphores S dan Q

P0		P1
`S:= 1;`		`Q:=1;`
`wait(S); // S=0`	`---------->`	`wait(Q); //Q=0`
`wait(Q); // Q=-1`	`<----------`	`wait(S); // S=-1`
// P0 diblokir		// P1 diblokir
	Deadlock	
`signal(S);`		`signal(Q);`
`signal(Q);`		`signal(S);`

Hati-hati Menggunakan Semaphore

- Pelanggaran Mutual Exclusion

```
signal(mutex);          mutexUnlock();
critical section        criticalSection();
wait(mutex);            mutexLock();
```

- Situasi deadlock

```
wait(mutex);            mutexLock();
critical section        criticalSection();
wait(mutex);            mutexLock(P);
```

- Pelanggaran mutual exclusion
  (menghilangkan `wait(mutex)` atau `mutexLock()`)

```
critical section        critical Section();
signal(mutex);          mutexUnlock();
```

- Situasi deadlock
  (menghilangkan `signal(mutex)` atau `mutexUnlock()`)

```
wait(mutex);            mutexLock();
critical section        criticalSection();
```

Monitor

- Cara yang lebih sederhana utk sinkronisasi
- Berupa sekumpulan operator yg didefinisikan oleh programmer

```
monitor nama_monitor
// deklarasi variabel
public entry P1(..);
{... };

......

public entry Pn(..);
{...};

begin
  kode inisialisasi
end
```


Properti Monitor

- Implementasi internal tipe monitor tidak bisa diakses secara langsung melalui berbagai thread
- Enkapsulasi yg disediakan oleh monitor membatasi akses ke variabel lokal hanya oleh prosedur lokal
- Monitor tidak mengizinkan akses secara konkuren terhadap semua prosedur yg didefinisikan dalam monitor
- Hanya satu thread/process yg dapat aktif di monitor pada satu saat
- Sinkronisasi bersifat built-in

Monitor: Solusi Permasalahan Prod-Kon (1)

```
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;
```


Monitor: Solusi Permasalahan Prod-Kon (2)

```
monitor ProducerConsumer
  condition full, empty;
  integer count;

  procedure enter;
  begin
    if count=N then wait(full);
    enter_item;
    count := count + 1;
    if count = 1 then signal(empty);
  end;

  procedure remove;
  begin
    if count = 0 then wait(empty);
    remove_item;
    count := count - 1;
    if count = N - 1 then signal(full);
  end;
  count := 0;
end monitor;
```

Message Passing

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```


Permasalahan Klasik IPC

- Produsen-Konsumen
- Bounded buffer
- Permasalahan Reader-writer 1
- Permasalahan dining philosophers
- Permasalahan Sleeping Barber

Permasalahan Bounded Buffer

- Diskusi kelompok (2 menit)
  - **Produsen**: dalam infinite loop dan menghasilkan 1 item untuk buffer pd tiap iterasi
  - **Konsumen**: dalam infinite loop dan mengkonsumsi 1 item dari buffer pd tiap iterasi
  - Ukuran buffer: dapat menyimpan paling banyak N item
- Tunjukkan di papan tulis

Permasalahan Reader Writer 1

- Reader: membaca data; Writer: menulis data
- Aturan:
  - Lebih dari satu reader dapat membaca data secara simultan
  - Hanya satu writer yg dapat menulis data pd satu saat
  - Reader and writer tidak bisa berada di CS bersama
- Tabel locking: menentukan apakah dua proses (R/W) dapat berada di CS sekaligus

	Reader	**Writer**
Reader	Ya	Tidak
Writer	Tidak	Tidak

Permasalahan Reader Writer 1: Solusi

```
Semaphore mutex, wrt; // shared and initialized to 1;
int readcount;        // shared and initialized to 0
// Writer                    // Reader
                             wait(mutex);
                             readcount:=readcount+1;
wait(wrt);                   if readcount == 1 then wait(wrt);
......                       signal(mutex);
writing performed            ....
.....                        reading performed
                             wait(mutex);
signal(wrt);                 readcount:=readcount-1;
                             if readcount == 0 then signal(wrt);
                             signal(mutex);
```

- Masalah dengan solusi ini?

Permasalahan Dining Philosophers (1)

- Kegiatan philosophers: makan dan berpikir
- Makan perlu 2 garpu
- Mengambil satu garpu pada satu saat
- Kemungkinan deadlock?
- Bagaimana mencegah deadlock?

Permasalahan Dining Philosophers (2)

```
#define N 5                                 /* number of philosophers */

void philosopher(int i)                     /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                           /* philosopher is thinking */
        take_fork(i);                       /* take left fork */
        take_fork((i+1) % N);               /* take right fork; % is modulo operator */
        eat( );                             /* yum-yum, spaghetti */
        put_fork(i);                        /* put left fork back on the table */
        put_fork((i+1) % N);                /* put right fork back on the table */
    }
}
```

- Bukan solusi permasalahan dining philosophers!

Permasalahan Dining Philosophers: Solusi (1)

```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N–1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      1                   /* philosopher is trying to get forks */
#define EATING      2                   /* philosopher is eating */
typedef int semaphore;                  /* semaphores are a special kind of int */
int state[N];                           /* array to keep track of everyone's state */
semaphore mutex = 1;                    /* mutual exclusion for critical regions */
semaphore s[N];                         /* one semaphore per philosopher */

void philosopher(int i)                 /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {                      /* repeat forever */
        think( );                       /* philosopher is thinking */
        take_forks(i);                  /* acquire two forks or block */
        eat( );                         /* yum-yum, spaghetti */
        put_forks(i);                   /* put both forks back on table */
    }
}
```

Permasalahan Dining Philosophers: Solusi (2)

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
     down(&mutex);                      /* enter critical region */
     state[i] = HUNGRY;                 /* record fact that philosopher i is hungry */
     test(i);                           /* try to acquire 2 forks */
     up(&mutex);                        /* exit critical region */
     down(&s[i]);                       /* block if forks were not acquired */
}
void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
     down(&mutex);                      /* enter critical region */
     state[i] = THINKING;               /* philosopher has finished eating */
     test(LEFT);                        /* see if left neighbor can now eat */
     test(RIGHT);                       /* see if right neighbor can now eat */
     up(&mutex);                        /* exit critical region */
}

void test(i)                            /* i: philosopher number, from 0 to N–1 */
{
     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
     }
}
```


Permasalahan Sleeping Barber

- Ada N kursi pelanggan
- Seorang tukang cukur mencukur rambut seorang pelanggan pada satu saat
- Tidur jika tidak ada pelanggan
- Diskusi kelompok (2 menit)

Permasalahan Sleeping Barber: Solusi (1)

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */
```


Permasalahan Sleeping Barber: Solusi (2)

```
void barber(void)
{
     while (TRUE) {
          down(&customers);          /* go to sleep if # of customers is 0 */
          down(&mutex);              /* acquire access to 'waiting' */
          waiting = waiting – 1;     /* decrement count of waiting customers */
          up(&barbers);              /* one barber is now ready to cut hair */
          up(&mutex);                /* release 'waiting' */
          cut_hair( );               /* cut hair (outside critical region) */
     }
}

void customer(void)
{
     down(&mutex);                   /* enter critical region */
     if (waiting < CHAIRS) {         /* if there are no free chairs, leave */
          waiting = waiting + 1;     /* increment count of waiting customers */
          up(&customers);            /* wake up barber if necessary */
          up(&mutex);                /* release access to 'waiting' */
          down(&barbers);            /* go to sleep if # of free barbers is 0 */
          get_haircut( );            /* be seated and be serviced */
     } else {
          up(&mutex);                /* shop is full; do not wait */
     }
}
```

Message Passing

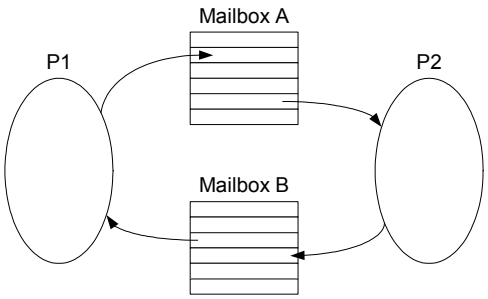
- Send (destination, &message)
- Receive (source, &message)
- Ukuran pesan: fixed atau bervariasi
- Analogi dunia nyata: percakapan

Message Passing: Solusi Produsen Konsumen

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

- Permasalahan Produsen Konsumen dengan N pesan

Komunikasi Tidak Langsung



- `send(A,message)` /* kirim pesan ke mailbox A */
- `receive(A,message)` /* terima pesan dari mailbox A */

Komunikasi Tidak Langsung: Kelebihan

- Memungkinkan lebih beragam skema:
  - 2 proses per link
  - 1 link per pasangan proses
  - uni atau bidirectional
  - Memungkinkan 1 proses untuk menerima pesan dari link
  - Memungkinkan 1 proses untuk semua menerima pesan dari 1 link