Concurrency: *Mutual Exclusion* dan Sinkronisasi

(Pertemuan ke-8)

Pendahuluan

- Apa yang akan dipelajari ?
  - Ruang lingkup *concurrency*
  - Contoh kasus perlunya *concurrency*
  - Jenis interaksi antar proses
  - Mekanisme *mutual exclusion*
  - Implementasi *mutual exclusion*
  - Semaphore
  - Monitor
  - *Message passing*
  - Beberapa contoh kasus

Concurrency (1)

- *Concurrency* = kebersamaan
- Apa yang menjadi ruang lingkup *concurrency* ?
  - Komunikasi antar proses
  - *Sharing* dan kompetisi penggunaan *resource*
  - Sinkronisasi antar berbagai proses
  - Pengalokasian waktu prosesor untuk setiap proses

Concurrency (2)

- Di mana *concurrency* diperlukan ?
  - *Multiprogramming*
    - Banyak proses - satu prosesor
  - *Multiprocessing*
    - Banyak proses - banyak prosesor (dalam satu komputer)
  - *Distributed processing*
    - Banyak proses – banyak komputer

Concurrency (3)

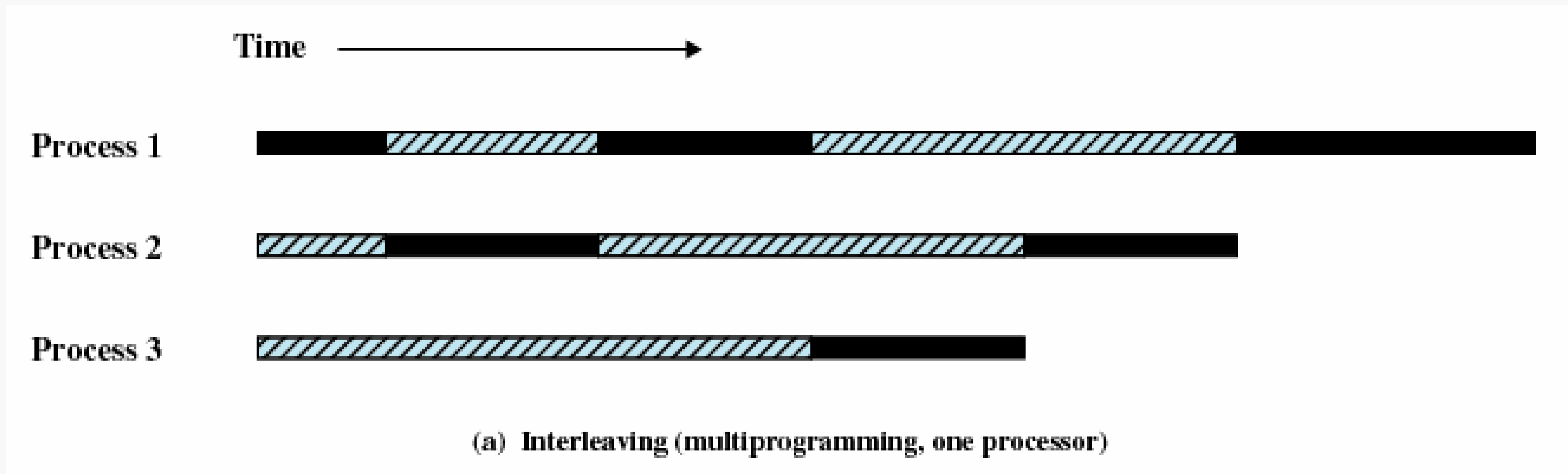
- Istilah-istilah berhubungan dg *concurrency*:
  - *Critical section*:
    - Resource yang dalam satu waktu hanya boleh diakses oleh satu proses saja
    - Contoh resource: printer, baris-baris program, file, dll
  - *Deadlock*:
    - Keadaan dimana dua proses atau lebih tidak dapat meneruskan eksekusi akibat saling menunggu aksi/data
  - *Livelock*:
    - Keadaan dimana dua proses atau lebih saling mengubah status sebagai respon terhadap perubahan status proses yang lain tanpa mengerjakan pekerjaan yang berarti

Concurrency (4)

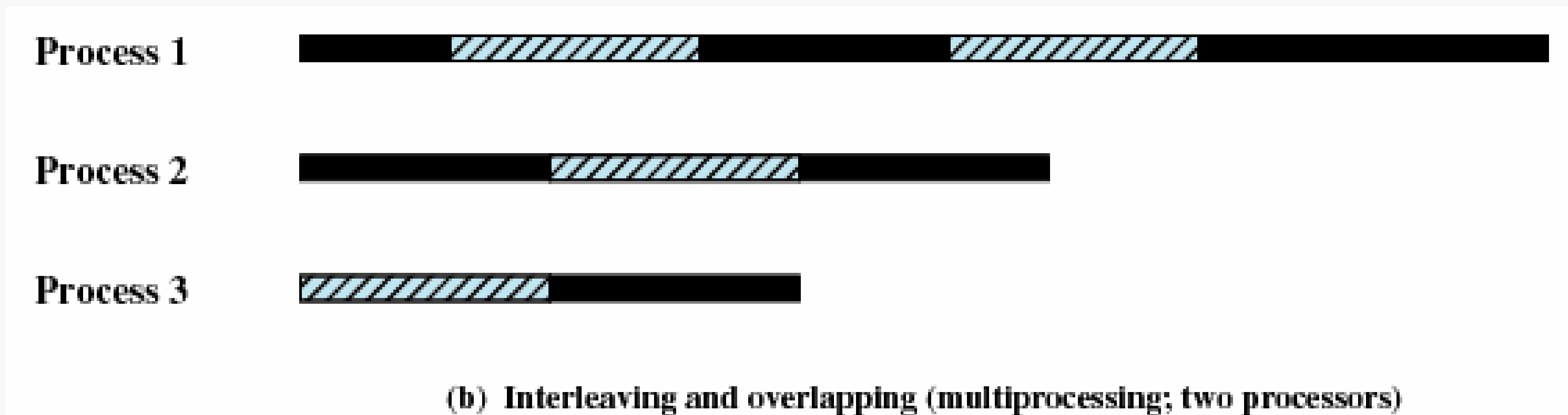
- Istilah-istilah berhubungan dg *concurrency*: (lanjutan)
  - *Mutual exclusion*:
    - Syarat/kondisi yang harus dipenuhi untuk mencegah terjadinya pengaksesan *critical section* oleh lebih dari satu proses dalam satu saat
  - *Race condition*:
    - Keadaan dimana terdapat banyak thread atau proses mengakses data bersama (shared data) yang menyebabkan hasil akhir sulit dipastikan (bergantung pada lama waktu eksekusi setiap proses)
  - *Starvation*:
    - Keadaan dimana suatu proses yang siap dieksekusi terus menerus tidak diberi kesempatan untuk melakukan aksinya

Concurrency (5)

- *Interleaving* dan *overlapping*



(a) Interleaving (multiprogramming, one processor)



(b) Interleaving and overlapping (multiprocessing; two processors)

Concurrency (6)

- Permasalahan pada *multiprogramming* dan *multiprocessor*:
  - Pemanfaatan resource bersama-sama (global) penuh resiko
    - Misal: variabel global
  - Pengaturan alokasi resource sukar dilakukan
    - Misal: bila suatu device I/O sedang diakses oleh suatu proses dan tiba-tiba proses tersebut ter-interrupt, maka device I/O harus tetap dapat digunakan oleh proses yang lain
  - Sulit untuk melacak kesalahan program, karena kesalahan yang terjadi sukar diprediksi dan diulangi

Contoh Kasus 1

- Sebuah prosedur digunakan 2 buah proses dalam uniprosesor:

```
void echo()
{
   chin = getchar();// dari keyboard
   chout = chin;
   putchar(chout); // ke monitor
}
```

- Masalah:
  - Ada 2 proses (P1 dan P2) yang masing-masing dapat memanggil prosedur di atas
  - Pada saat P1 sedang menjalankan prosedur tsb pada baris getchar tiba-tiba P1 terinterrupt oleh P2, misal chin = x
  - P2 menjalankan prosedur dan mengubah nilai chin = y
  - Saat P1 kembali → nilai chin berbeda → y ditampilkan 2 kali
- Solusi:
  - Prosedur echo diproteksi sehingga dalam satu saat hanya satu proses yang boleh menggunakannya → proses lain diblok

Contoh Kasus 2

- Sebuah prosedur digunakan 2 buah proses dalam multiprosesor:

```
Process P1
.
chin = getchar();
.
chout = chin;
putchar(chout);
.
.
```

```
Process P2
.
.
chin = getchar();
chout = chin;
.
putchar(chout);
.
```

- Masalah:
  - Ada 2 proses (P1 dan P2) yang masing-masing dapat memanggil prosedur di atas
  - Pada saat P1 sedang menjalankan prosedur tsb pada baris getchar tiba-tiba P2 juga menjalankan prosedur yang sama → data P1 pada chin **tertimpa**
  - Baik P1 dan P2 sama-sama menampilkan hasil dari P2
- Solusi:
  - Prosedur echo diproteksi sehingga dalam satu saat hanya satu proses yang boleh menggunakannya → proses lain diblok

Race Condition (1)

- Hasil proses bergantung pada urutan eksekusi setiap proses
- Contoh 1:
  - Dua buah proses P1 dan P2 sama-sama menggunakan variabel global a
  - P1 mengubah nilai a = 1, P2 mengubah nilai a = 2
  - Nilai a ditentukan oleh proses yang "kalah" (yang mengakses variabel a belakangan)

Race Condition (2)

- Contoh 2:
  - Dua buah proses P3 dan P4 sama-sama menggunakan variabel global b dan c yang masing-masing bernilai b=1 dan c=2
  - P3 menjalankan baris program b=b+c
  - P4 menjalankan baris pogram c=b+c
  - Jika P3 yang dieksekusi lebih dahulu, maka hasilnya: b=3 dan c=5
  - Jika P4 yang dieksekusi lebih dahulu, maka hasilnya: b=4 dan c=3

  → Hasil akhir sangat berbeda !!!

Peranan OS dalam Concurrency

- Apa yang harus dilakukan OS untuk memperoleh concurrency ?
  - OS harus dapat menjaga track (info state, prioritas, *resource*, dll) setiap proses
  - OS harus dapat memberi dan mengambil resource (waktu prosesor, memory, file, I/O device) kepada setiap proses yang aktif
  - OS harus dapat memproteksi data dan resource yang sedang digunakan suatu proses
  - OS harus dapat menjamin hasil suatu proses tidak dipengaruhi oleh kecepatan pemrosesan

Jenis Interaksi Antar Proses

- Misal: multiprogramming (Winamp, media player, dll)
- Misal: penggunaan variabel global, share memory, I/O buffer, dll

Degree of Awareness	Relationship	Influence that one Process has on the Other	Potential Control Problems
Processes unaware of each other	Competition	•Results of one process independent of the action of others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation •Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Deadlock (consumable resource) •Starvation

Kompetisi Antar Proses (1)

- Dua proses atau lebih saling berkompetisi memperebutkan sebuah resource
- Masalah apa yang mungkin terjadi ?
  - Mutual Exclusion
    - Sebuah resource diakses oleh dua buah proses atau lebih secara bersamaan
    - Misal: dua buah proses ingin mengakses sebuah printer
  - Deadlock
    - Dua proses atau lebih saling menunggu data atau resource lain yang sedang digunakan oleh proses yang lain
    - Misal:
      - Dua buah proses P1 dan P2 sama-sama membutuhkan resource R1 dan R2
      - OS telah memberikan R2 kepada P1 dan R1 kepada P2
      - P1 dan P2 sama-sama tidak mau melepaskan resource yang sedang digunakan karena kebutuhannya belum terpenuhi → deadlock

Kompetisi Antar Proses (2)

- *Starvation*
  - Terdapat satu proses atau lebih yang tidak pernah mendapat giliran dieksekusi (memperoleh *resource*)
  - Misal:
    - Ada 3 proses P1, P2, dan P3
    - Mula-mula P1 dieksekusi, P2 dan P3 menunggu giliran
    - Setelah P1 selesai, P3 mendapat giliran
    - Sebelum P3 selesai, P1 telah melakukan interrupt minta untuk dieksekusi → P2 tertunda lagi
    - Bila kodisi seperti di atas terjadi terus menerus → P2 <u>tidak pernah</u> mendapatkan giliran → starvation (kelaparan)

Kerjasama Antar Proses melalui Sharing

- Beberapa proses berbagi data yang sama tanpa mengetahui identitas proses yang lain, tetapi sama-sama sepakat menjaga integritas data
- Membutuhkan mekanisme untuk menjaga koherensi/konsistensi/integritas data
- Apa yang di-*share* ?
  - Variabel, file, atau database
- Model akses:
  - *Read* (baca): semua proses boleh membaca secara bersama-sama
  - *Write* (tulis): dalam satu saat hanya satu proses yang boleh menulis

Kerjasama Antar Proses melalui Komunikasi

- Beberapa proses saling berkomunikasi untuk memperoleh sinkronisasi atau berkoordinasi dalam melakukan aktifitas
- Komunikasi diwujudkan dalam bentuk kirim dan terima pesan primitif yang biasanya telah disediakan oleh bahasa pemrograman atau kernel OS
- Masalah yang mungkin terjadi:
  - *Deadlock*: saling menunggu pesan
  - *Starvation*:
    - Misal:
      - P1 terus menerus mencoba berkomunikasi dengan P2 dan P3 demikian pula sebaliknya
      - Bila P1 berhasil komunikasi dengan P2 dalam waktu yang lama, maka P3 mengalami starvation

Syarat *Mutual Exclusion* (1)

- Dalam satu waktu hanya satu proses saja yang boleh mengakses *critical section*
- Proses yang berada di luar critical section harus dapat melakukan aktifitas lain dengan tidak mengganggu proses yang lain
- Tidak boleh terjadi *deadlock* atau *starvation*

Syarat Mutual Exclusion (2)

- Dalam pengaksesan *critical section* tidak boleh ada tunda waktu *(delay)* bila sedang tidak ada yang mengakses *critical section* tersebut
- Kecepatan relatif proses dan jumlah proses tidak boleh mempengaruhi *mutual exclusion (race condition)*
- Sebuah proses berada pada *critical section* dalam waktu terbatas

Mekanisme Mutual Exclusion (Mutex)

```
/* Process 1 */
void P1
{
  while (true)
  {
     /*preceding code*/;
    entercritical (Ra);
     /*critical section*/;
    exitcritical (Ra);
    /*following code*/;
  }
}
```

```
/* Process 2 */
void P2
{
  while (true)
  {
     /*preceding code*/;
    entercritical (Ra);
     /*critical section*/;
    exitcritical (Ra);
    /*following code*/;
  }
}
```

...

```
/* Process n */
void Pn
{
  while (true)
  {
     /*preceding code*/;
    entercritical (Ra);
     /*critical section*/;
    exitcritical (Ra);
    /*following code*/;
  }
}
```

Implementasi Mutual Exclusion (1)

- Mutex dengan Enable-disable interrupt
  - Perintah primitive yang disediakan oleh sistem kernel
  - Bertujuan untuk melindungi critical section agar proses yang sedang mengakses critical section tidak dapat diinterrupt
  - Mekanisme:

```
while (true)
{
    /* disable interrupts */
    /* critical section   */
    /* enable interrupts */
    /* remainder          */
}
```

  - Hanya dapat diterapkan pada sistem uniprocessor, why ?

Implementasi Mutual Exclusion (2)

- Mutex dengan Instruksi khusus yang atomik:
  - Dua buah instruksi dibungkus menjadi sebuah instruksi yang atomik (tidak dapat disela/diinterrupt)
  - Misal:
    - Instruksi Test and Set
    - Instruksi Exchange

Instruksi Test and Set (1)

- Fungsi testset:
  - Memeriksa nilai suatu variabel dan mengubah nilainya

```
boolean testset (int i) {
    if (i == 0) {
        i = 1;
        return true;
    }
    else {
        return false;
    }
}
```

- Cek apakah nilai i = 0
  - 0 → 1, true
  - 1 → 1, false

Instruksi Test and Set (2)

- Pemanfaat-an fungsi **`testset`** dalam *mutex*
- bolt = 0 → dapat akses *critical section*
- bolt = 1 → *looping*

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true)
   {
      while (!testset (bolt))
          /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . . ,P(n));

}
```

Instruksi Test and Set (3)

- Apa arti *parbegin*:
  - Tunda eksekusi main program
  - Lakukan eksekusi *concurrent* terhadap prosedur P1, P2, …, Pn
  - Hanya proses yang kebetulan mendapatkan nilai `bolt=0` saja yang dapat mengakses *critical section*
  - Proses yang lain masuk dalam mode *busy waiting* atau *spin waiting* (terus menerus *looping* melakukan pemeriksaan nilai `bolt`)
  - Bila eksekusi concurrent semua prosedur telah selesai → lanjutkan eksekusi main program

Instruksi Exchange (1)

- Prosedur exchange:
  - Untuk mempertukarkan data dari variabel `memory` ke variabel `register` atau sebaliknya

```
void exchange (int register, int memory)
  {
   int temp;
   temp = memory;
   memory = register;
   register = temp;
}
```

Instruksi Exchange (2)

- Pemanfaat-an prosedur **exchange** dalam *mutex*
- bolt = 0 → dapat akses *critical section*
- bolt = 1 → *looping*

```
/* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi = 1;
   while (true)
   {
     do exchange (keyi, bolt)
     while (keyi != 0)
     /* critical section */
     exchange (keyi, bolt);
     /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . ., P(n));
}
```

Instruksi *Exchange* (3)

- Keterangan:
  - Hanya proses yang kebetulan mendapatkan `bolt=0` saja yang dapat mengakses *critical section*
  - Proses yang telah selesai mengakses *critical section* akan menjalankan instruksi `exchange` lagi untuk me-*reset* nilai `bolt` menjadi 0 lagi
  - Pada saat `bolt=0` → tidak ada proses yang mengakses *critical section*, `bolt=1` → hanya ada satu proses yang sedang mengakses *critical section*

Kelebihan Instruksi Atomik

- Dapat diterapkan pada sistem dengan jumlah proses berbeda-beda
- Dapat diterapkan pada prosesor tunggal maupun multiprosesor yang mengakses *share memory*
- Sederhana → mudah ditelusuri
- Dapat menangani beberapa *critical section* berbeda-beda → tiap *critical section* menggunakan variabel berbeda-beda

Kekurangan Instruksi Atomik

- Dapat terjadi *busy-waiting* → masih tetap menggunakan waktu prosesor
- Dapat terjadi *starvation* bila terdapat banyak proses yang mengantri *critical section*
- Dapat terjadi *deadlock*
  - Jika proses dengan prioritas rendah sedang mengakses *critical section* tiba-tiba di-*interrupt* oleh proses dengan prioritas lebih tinggi
    - → waktu prosesor diberikan kepada proses dengan prioritas lebih tinggi, tetapi
    - → *critical section* masih dikuasai oleh proses dengan prioritas lebih rendah

Semaphore

- Diperkenalkan oleh Dijkstra pada tahun 1965
- Merupakan mekanisme concurrency yang disediakan oleh OS dan bahasa pemrograman
- Digunakan variabel khusus yang disebut semaphore yang digunakan sebagai tanda (signaling)
- Proses yang sedang menunggu signal akan berada pada status suspend hingga signal diterima, apakah masih menggunakan waktu prosesor ???

Semaphore Primitif (1)

- Prosedur yang digunakan:
  - semSignal (s)
    - Untuk mengirimkan signal semaphore s
    - semSignal = V = *verhogen* = increment
  - semWait (s)
    - Untuk menerima signal semaphore s
    - semWait = P = *proberen* = test
- Semaphore primitif

  = *general semaphore*

  = *counting semaphore*

Semaphore Primitif (2)

- Ketentuan:
  - Inisialisasi variabel semaphore <u>tidak boleh negatif</u>
  - Prosedur semWait:
    - Akan mengurangi nilai variabel semaphore
    - Jika nilai variabel menjadi negatif → proses yang mengeksekusi semWait akan di-blok
    - Jika tidak → proses tersebut akan dilayani
  - Prosedur semSignal:
    - Akan menambah nilai variabel semaphore
    - Jika nilai variabel menjadi ≤ 0 → sebuah proses yang di-blok oleh semWait akan dibebaskan

Semaphore Primitif (3)

- Definisi semaphore primitif

```
struct semaphore {
     int count;
     queueType queue;
}

void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0)
     {
          place this process in s.queue;
          block this process
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0)
     {
          remove a process P from s.queue;
          place process P on ready list;
     }
}
```

Semaphore Primitif (4)

- Apa pengaruh variabel `s.count` tsb ?
  - Jika `s.count` ≥ 0:
    - `s.count` merupakan jumlah proses yang dapat mengeksekusi `semWait(s)` tanpa penundaan
  - Jika `s.count` < 0:
    - `s.count` merupakan jumlah proses yang di-blok dan berada di dalam antrian

Referensi:

[STA09] Stallings, William. 2009. *Operating System: Internal and Design Principles*. 6th edition. Prentice Hall