**Jurnal Praktikum**

**Sistem Operasi (CII-2H3)**

**Modul 4: Thread dan Mutex**

**Tujuan**

1. Mahasiswa mampu membuat program dengan thread pada Linux
2. Mahasiswa menguasai teknik sinkronisasi menggunakan semaphore

**Tutorial POSIX Thread**

1. Semua fungsi dan tipe posix semaphore didefinisikan pada semaphore.h
2. Untuk mendeklarasikan semaphore gunakan "sem\_t nama\_semaphore"

contoh:

sem\_t my\_semaphore;

Letakkan deklarasi semaphore sebagai global variabel (ada di luar fungsi main dan fungsi-fungsi lain).

1. Untuk inisiasi semaphore gunakan "sem\_init(sem\_t \*sem, int pshared, unsigned int value)"

sem menunjuk ke semaphore yang digunakan

pshared merupakan indikasi apakah semaphore dishare ketika ada fork(). Untuk mata kuliah ini, nilainya diset selalu 0.

value merupakan nilai inisial semaphore

contoh:

sem\_init(&my\_semaphore, 0, 5); // my\_semaphore diinisiasi dengan nilai 5

Letakkan sem\_init() pada fungsi main()

1. Untuk wait gunakan sem\_wait(sem\_t \*sem)

contoh:

sem\_wait(&my\_semaphore);

Jika nilai semaphore negatif maka proses yang memanggil sem\_wait() akan block

Letakkan sesuai dengan kebutuhan dan spesifikasi program (dapat pada fungsi manapun).

1. Untuk signal gunakan sem\_post(sem\_t \*sem)

contoh:

sem\_post(&my\_semaphore);

Letakkan sesuai dengan kebutuhan dan spesifikasi program (dapat pada fungsi manapun).

**Jurnal Praktikum**

1. bad\_code\_1.c
   1. Beri komentar pada SETIAP BARIS kode bad\_code\_1.c:
   2. Jelaskan apa yang dilakukan oleh kode ini!

* Agar bisa mengakses threads dan memproses execute pada thread
  1. Compile source code (nama executable adalah bad\_code\_1).
  2. Jalankan executable beberapa kali (min 10x)! Amati hasil yang diperoleh!
* Hasilnya berbeda – beda
  1. Jelaskan hasil observasi Anda! Mengapa output bisa berbeda-beda? Hubungkan dengan race condition!
* Ketika suatu thread/proses nya banyak maka akan diakses satu command yang sama, oleh karena itu maka kita membutuhkan sinkronisasi yang merupakan suatu upaya yang dilakukan agar proses-proses yang saling bekerjasama dieksekusi secara beraturan demi mencegah timbulnya suatu keadaan race condition.

1. good\_code\_1.c
2. Ubahlah source code bad\_code\_1.c dengan spesifikasi: **x selalu berjalan sebelum y**. Thread z boleh berjalan kapanpun.

x,y,z = benar

x,z,y = benar

z,y,x = salah

y,z,x = salah

y,x,z = salah

Hint: gunakan signalling!

* Ketika di running akan ada kombinasi x berada dibawah y, dengan menggunakan signalling maka akan membuat semua proses berjalan sequantial sehingga x berjalan lebih dahulu dari y.

1. Compile source code yang telah diperbaiki dengan nama executable good\_code\_1

* Ketika mendapatkan y diproses lebih dahulu dari x maka proses tersebut tidak boleh sehingga harus melakukan good\_code\_1

1. Jalankan executable beberapa kali (min 10x)!

* Pada good code terdapat sem\_post dan sem\_wait sehingga ketika di running bagaimanapun hasilnya akan tetap x akan selalu diatas y.

1. good\_code\_xyz.c
   1. Ubahlah source code bad\_code\_1 dengan spesifikasi: urutan selalu x, y, z!

* Menggunakan teknik semaphore, ketikasem\_wait dikurangi/dieksekusi akan menjadi 0 lagi selanjutnya menge-print y, sem\_post dari dari sem\_1 +1 baru bisa menjalankan sem wait selanjutnya tanpa harus di block.
  1. Compile source code yang telah diperbaiki dengan nama executable good\_code\_1
  2. Jalankan executable beberapa kali (min 10x)!
* Hasil yang akan keluar ialah urutan selalu x, y, z

1. bad\_code\_2.c

Program berikut membuat dua buah thread. Dua thread tersebut akan melakukan increment pada global variable bernama cnt.

Jika diincrement sebanyak 4000x oleh masing-masing thread maka seharusnya program menghasilkan cnt = 8000. Thread 1 akan melakukan increment 4000x dan Thread 2 akan melakukan increment 4000x sehingga diakhir program nilai cnt adalah 8000. Kita tidak mengetahui bagaimana os akan menjadwalkan thread 1 maupun thread 2 tetapi os pasti akan menjalankan thread 1 dan thread 2 masing-masing sebanyak 4000x.

* 1. Beri komentar pada SETIAP BARIS kode bad\_code\_2.c:
  2. Compile source code (nama executable adalah bad\_code\_2).
  3. Jalankan executable beberapa kali (min 10x)! Amati hasil yang diperoleh!
  4. Jelaskan hasil observasi Anda! Mengapa output bisa berbeda-beda? Hubungkan dengan race condition!

1. good\_code\_1.c
2. Ubahlah source code bad\_code\_2.c sehingga menghasilkan cnt yang benar. Hint: gunakan mutex!
3. Compile source code yang telah diperbaiki dengan nama executable good\_code\_2
4. Jalankan executable beberapa kali (min 10x)!

**bad\_code\_1.c**

//========================SOF================

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <stdlib.h>

sem\_t my\_sem;

void \* Print\_x(void \* a)

{

printf("Print x\n");

}

void \* Print\_y(void \* a)

{

printf("Print y\n");

}

void \* Print\_z(void \* a)

{

printf("Print z\n");

}

int main(int argc, char \* argv[])

{

pthread\_t tid1, tid2, tid3;

if(pthread\_create(&tid1, NULL, Print\_x, NULL))

{

printf("\n Eror! tidak bisa membuat thread 1");

exit(1);

}

if(pthread\_create(&tid2, NULL, Print\_y, NULL))

{

printf("\n Eror! tidak bisa membuat thread 2");

exit(1);

}

if(pthread\_create(&tid3, NULL, Print\_z, NULL))

{

printf("\n Eror! tidak bisa membuat thread 3");

exit(1);

}

if(pthread\_join(tid1, NULL))

{

printf("\n Eror, join thread");

exit(1);

}

if(pthread\_join(tid2, NULL))

{

printf("\n Eror join thread");

exit(1);

}

if(pthread\_join(tid3, NULL))

{

printf("\n Eror join thread");

exit(1);

}

pthread\_exit(NULL);

}

//========================EOF================

**bad\_code\_2.c**

//========================SOF================

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <stdlib.h>

#define COUNT 1000000

int cnt = 0;

sem\_t my\_mutex;

void \* Count(void \* a)

{

int i, tmp;

for(i = 0; i < COUNT; i++)

{

tmp = cnt;

tmp = tmp+1;

cnt = tmp;

}

}

int main(int argc, char \* argv[])

{

pthread\_t tid1, tid2;

if(pthread\_create(&tid1, NULL, Count, NULL))

{

printf("\n Eror! tidak bisa membuat thread 1");

exit(1);

}

if(pthread\_create(&tid2, NULL, Count, NULL))

{

printf("\n Eror! tidak bisa membuat thread 2");

exit(1);

}

if(pthread\_join(tid1, NULL))

{

printf("\n Eror, join thread");

exit(1);

}

if(pthread\_join(tid2, NULL))

{

printf("\n Eror join thread");

exit(1);

}

if (cnt < 2 \* COUNT)

printf("\n NOT OK! cnt = [%d], seharusnya %d\n", cnt, 2\*COUNT);

else

printf("\n OK! cnt = [%d]\n", cnt);

pthread\_exit(NULL);

}

//========================EOF================