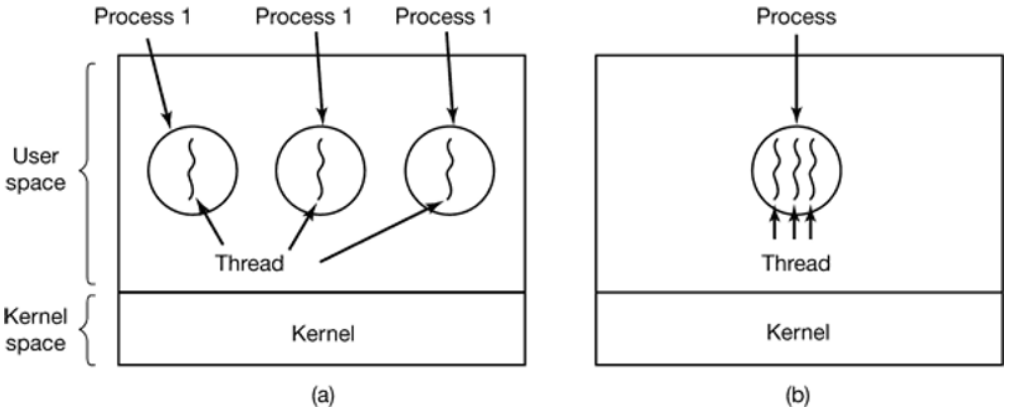

Thread

Thread คืออะไร

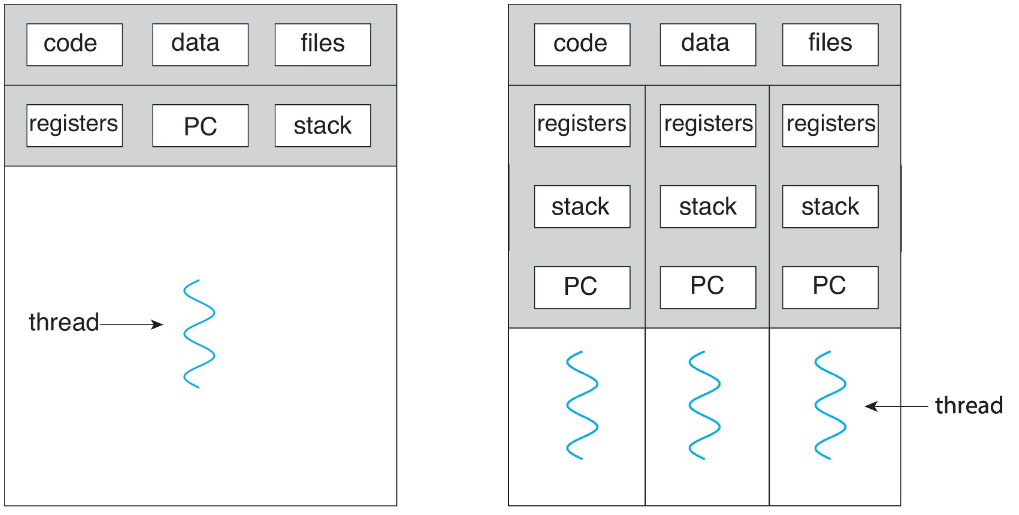
- เธรด คือ หน่วยการทำงานย่อยที่อยู่ในโพรเซสที่มีการแบ่งปันทรัพยากรต่าง ๆ ในโพร เซสนั้น ๆ โดยปกติ โพรเซส ที่มี เพียง 1 เธรดจะถูก เรียกว่า Single thread หรือเรียก อีก ชื่อว่า Heavy Weight Process ซึ่งมักพบในระบบปฏิบัติการรุ่นเก่า
- แต่ถ้า 1 โพรเซสมีเธรดหลายเธรดจะเรียกว่า Light Weight Process (LWP) หรือ Multithread ซึ่งพบได้ในระบบปฏิบัติการรุ่นใหม่ที่ใช้กันในปัจจุบันทั่วไป
- Multithread ก็เป็นที่นิยมมากกว่า Single thread

Thread คืออะไร (2)



- (a) คุณจะเห็นโพรเซส 3 โพรเซส แต่ละโพรเซสจะมีเลขที่ตำแหน่งเป็นของตนเอง และควบคุมเพียง 1 เท่านั้น
- (b) จะเห็นว่าในแต่ละโพรเซสจะควบคุมสามเธรด โดยใช้เลขที่ตำแหน่งเดียวกันอยู่

Thread คืออะไร (3)



single-threaded process

multithreaded process

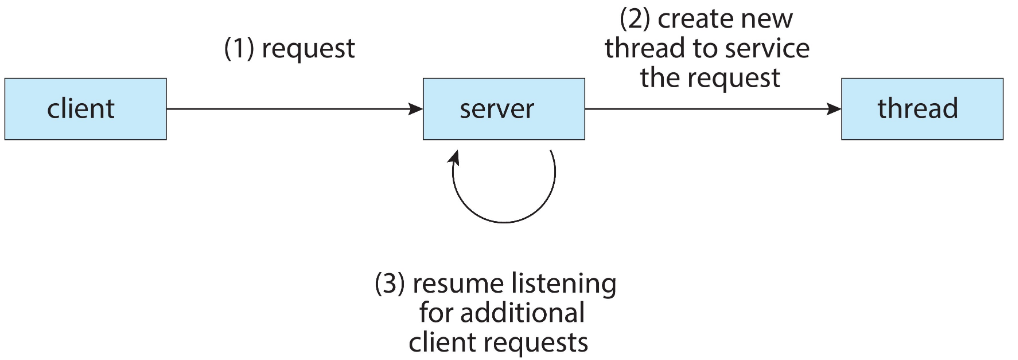
- เธรดเป็นหน่วยพื้นฐานของการจัดสรรการใช้ประโยชน์ของซีพียู ภายในโพรเซสจะประกอบด้วยเธรดจะมีการแชร์โค้ด ข้อมูล และทรัพยากร เช่น ไฟล์ อุปกรณ์ต่าง ๆ เป็นต้น โพรเซสดั้งเดิม (ที่เรียกว่า Heavy weight) ที่มีการควบคุมเพียง 1 เธรด แสดงว่าทำงานได้ 1 งาน แต่ถ้าโพรเซสมีหลายเธรด (อาจเรียกว่า Multithread) จะทำงานได้หลายงานในเวลาเดียวกัน

ตัวอย่างการใช้ Thread

- วิธีหนึ่งคือให้เซริฟเวอร์เรียกใช้งานโพรเซสขึ้นมาหนึ่งโพรเซสและรอรับการร้องขอเมื่อได้รับแล้ว จะสร้างโพรเซสแยกออกมาเพื่อให้บริการในทุกการร้องขอที่ขอมา

- ข้อเสียคือ การสร้างโพรเซสต้องใช้เวลาในการสร้างมาก

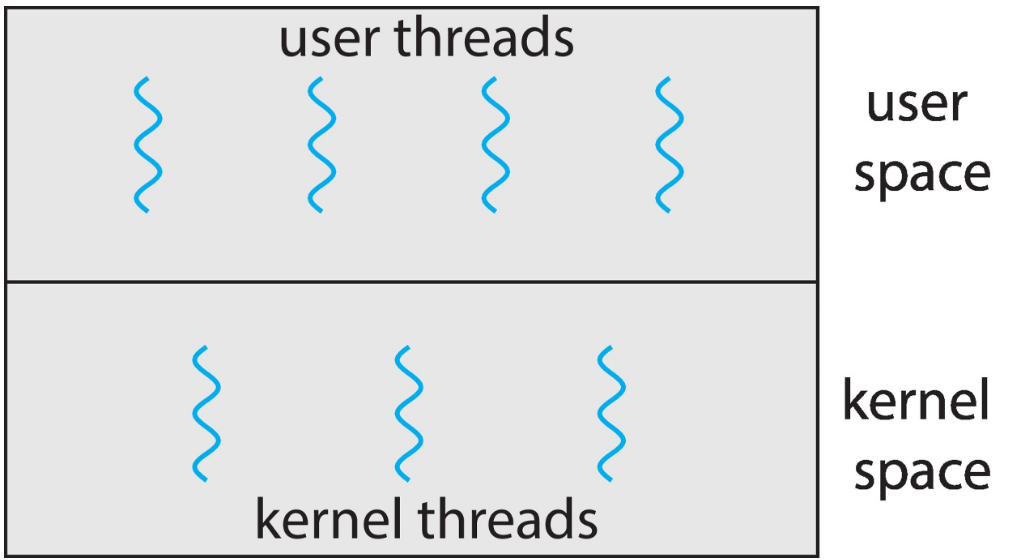
ตัวอย่างการใช้ Thread (2)



1. เมื่อมีการร้องขอ (Request)
2. สร้าง Thread ขึ้นมาเพื่อจัดการกับ Request
3. Server จะได้ไปจัดการกับ Request ถัดๆมาได้

Thread สำหรับผู้ใช้และ Thread สำหรับระบบปฏิบัติการ

- เธรดอาจจะแบ่งตามระดับการสนับสนุนได้ 2 แบบที่สัมพันธ์กัน คือ
  1. เธรดสำหรับผู้ใช้ - ง่ายที่จะถูกสร้างและอาจถูกยกเลิกก่อนเข้าเธรดสำหรับระบบปฏิบัติการได้และ
  2. เธรดสำหรับระบบปฏิบัติการ - รองรับเธรดสำหรับผู้ใช้และการปฏิบัติงาน



Thread สำหรับผู้ใช้

- การสร้างเธรดและการจัดเวลา เธรดทั้งหมดจะกระทำเสร็จสิ้นภายในพื้นที่ของผู้ใช้โดยไม่จำเป็นต้องใช้ Kernel ดังนั้นเธรดในระดับผู้ใช้สามารถสร้างและจัดการได้อย่างรวดเร็ว
- อย่างไรก็ตามถ้า Kernel เป็น Single thread แล้ว เธรดระดับผู้ใช้จะบล็อก System call จนเป็นเหตุให้ทุกโพรเซสถูกบล็อก ถึงแม้ว่าเธรดอื่นจะยังคงรันอยู่ในแอปพลิเคชั่นก็ตาม

Thread สำหรับระบบปฏิบัติการ

- โดย Kernel จะสร้าง จัดเวลา และจัดการเธรดภายในพื้นที่ของ Kernel เอง เนื่องจากระบบปฏิบัติการเป็นผู้จัดการเกี่ยวกับการสร้างและจัดการเธรดเอง จึงทำให้เธรดสำหรับระบบปฏิบัติการจะสร้างและจัดการได้ช้ากว่าเธรดสำหรับผู้ใช้
- อย่างไรก็ตาม เพราะ Kernel จัดการเกี่ยวกับเธรด ดังนั้นถ้าเธรดเกิดการบล็อก System call จะทำให้ Kernel จัดการนำเอาเธรดอื่นในแอปพลิเคชั่นเข้ามาดำเนินการแทนได้ เช่นเดียวกับในสภาวะมัลติโพรเซสเซอร์ที่ Kernel สามารถจัดเธรดลงในโพรเซสเซอร์อื่นได้

รูปแบบของ Thread

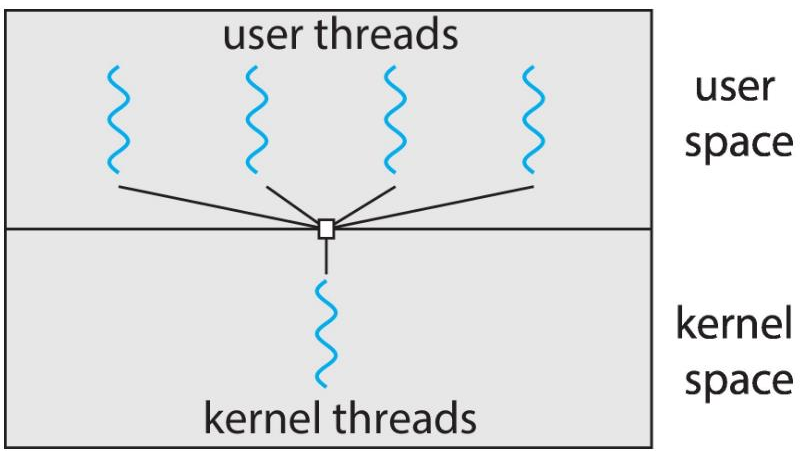
- การสนับสนุนการทำงานของเธรดจะขึ้นอยู่กับระดับของผู้ใช้จากเธรดของผู้ใช้ หรือจาก Kernel
- แต่เธรดของผู้ใช้จะสนับสนุนมากกว่า Kernel และสามารถควบคุมโดยไม่ต้องใช้การสนับสนุนจาก Kernel
- ส่วนเธรดของ Kernel นั้นจะสนับสนุนและควบคุมโดยตรงจากระบบปฏิบัติการ
- ในที่สุดแล้วเธรดของผู้ใช้และเธรดของ Kernel ก็ยังเชื่อมโยงกันอยู่ดี

จากการที่มี เธรดของผู้ใช้ กับ เธรดของ Kernel จึงสรุปรูปแบบความสัมพันธ์ได้เป็น 3 แบบ

1. Many-to-One
2. One-to-One
3. Many-to-Many

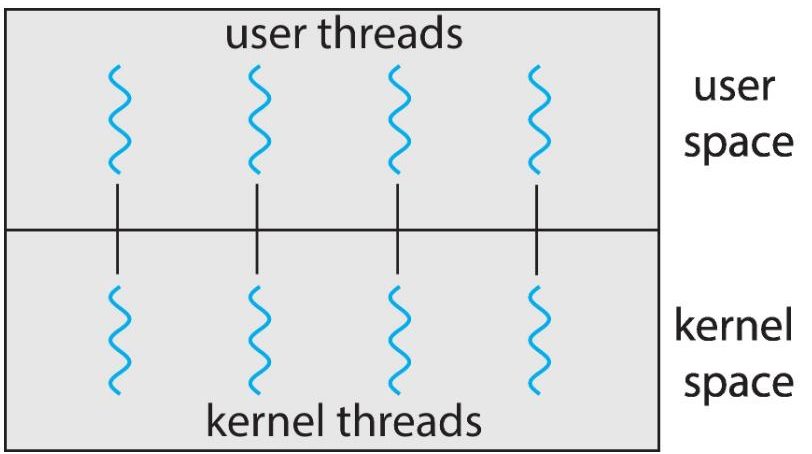
Thread แบบ Many-to-One

- รูปแบบ Many-to-One เป็นรูปแบบที่ใช้เธรดสำหรับระบบปฏิบัติการ 1 หน่วย กับเธรดสำหรับผู้ใช้หลายหน่วย (Thread ผู้ใช้ผลัดกันทำงานใน Kernel)
- การจัดการเธรดจะอยู่ในพื้นที่ของผู้ใช้ซึ่งมีประสิทธิภาพ แต่ถ้าเธรดบล็อก System call โพรเซสทั้งหมดจะถูกบล็อกไปด้วย เนื่องจากจะมีเพียงเธรดเดียวเท่านั้นที่เข้าถึง Kernel ในเวลาหนึ่ง ๆ
- เธรดหลาย ๆ เธรด ไม่สามารถรันขนานกันในระบบมัลติโพรเซสเซอร์ได้ ระบบที่ใช้รูปแบบนี้เช่น Green thread ซึ่งเป็นไลบรารีในโซลาริสทู (Solaris 2)



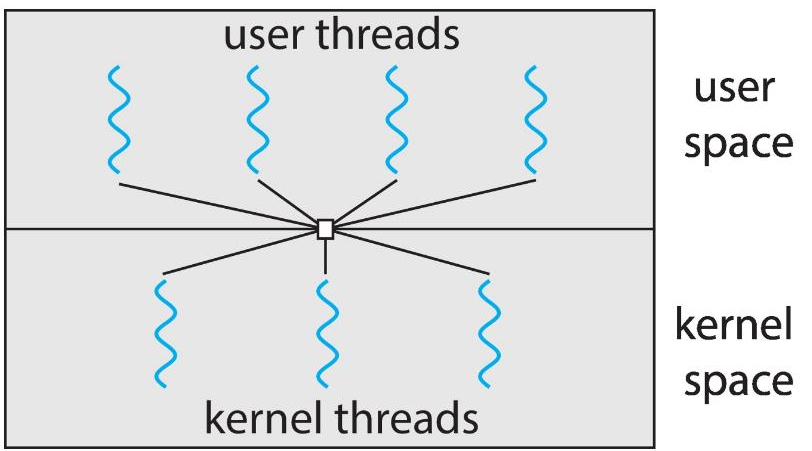
Thread แบบ One-to-One

- รูปแบบ One-to-One เป็นรูปแบบที่แต่ละเธรดสำหรับผู้ใช้ จะจับคู่กับเธรดสำหรับระบบปฏิบัติการ ในลักษณะ 1 ต่อ 1
- ทำให้สามารถทำงานพร้อมกันดีกว่าแบบ Many-to-One โดยยอมให้เธรดอื่นรันได้เมื่อเธรดบล็อก System call
- นอกจากนี้โมเดลนี้ยังยอมให้หลาย ๆ เธรดทำงานแบบขนานกันได้ในระบบมัลติโพรเซสเซอร์ได้อีกด้วย
- การสร้างเธรดสำหรับผู้ใช้ จำเป็นต้องสร้างเธรดสำหรับระบบปฏิบัติการที่สัมพันธ์กัน ระบบที่โมเดลมีข้อจำกัดที่จำนวน เธรดที่สนับสนุนในระบบได้ โมเดลนี้นำมาใช้ในระบบ เช่น ในระบบปฏิบัติการวินโดว์ กับ Linux



Thread แบบ Many-to-Many

- รูปแบบ Many-to-Many เป็นรูปแบบที่อาจจะมีจำนวนเธรดสำหรับผู้ใช้ มากกว่าหรือเท่ากับจำนวนเธรดสำหรับระบบปฏิบัติการ
- ผู้พัฒนาสร้างเธรดสำหรับผู้ใช้ ได้ตามที่เขาต้องการ แต่จะไม่สามารถทำงานได้พร้อมกัน
- จำนวนเธรดสำหรับระบบปฏิบัติการ อาจจะเป็นตัวกำหนดแอปพลิเคชันเฉพาะหรือเครื่องเฉพาะ - ระบบปฏิบัติการสามารถเลือกจำนวนเธรตเคอร์เนลที่จะสร้างได้ ในจำนวนที่ตนเห็นว่าเหมาะสม ไม่ถูกบังคับให้ต้องสร้างเท่ากับจำนวนเธรตผู้ใช้ เมื่อเธรดเกิดการบล็อก System call แล้ว Kernel จะจัดเวลาเพื่อนำเธรดอื่นขึ้นมารันก่อนก็ได้



POSIX Pthreads

- Pthreads เป็นตัวพื้นฐานของ POSIX (IEEE 103.1C) เรียกได้ว่าเป็น API สำหรับการสร้างเธรดและสิ่งที่เกิดขึ้นในเวลาเดียวกัน เป็นตัวบ่งบอกถึงพฤติกรรมของเธรดโดยไม่ใช้เครื่องมือ การออกแบบระบบปฏิบัติการมักจะใช้เครื่องมือในงานที่ต้องการ ระบบปฏิบัติการส่วนใหญ่ใช้ Pthreads เป็นเครื่องมือ ไม่ว่าจะเป็นระบบปฏิบัติการโซราริส ลินุกส์ แมคโอเอส และยูนิกส์

  1. pthreads_create() - สร้าง Thread
  2. pthread_join( ) - เธรดพ่อจะรอคำสั่งสิ้นสุดการทำงาน
  3. pthread_exit( ) - เธรดลูกจะสิ้นสุดการทำงานเมื่อมันเรียกฟังก์ชัน

ตัวอย่างถัดไปแสดงการทำงานแยกกันของ Thread กับ main Process

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>

void * thread_function(void * arg); //user define function
int i, j;
int main() {
  pthread_t a_thread; //thread declaration
  pthread_create( & a_thread, NULL, thread_function, NULL);
  //thread is created
  pthread_join(a_thread, NULL); //process waits for thread to finish . //Comment this line to see the difference
  printf("Inside Main Program\n");
  for (j = 20; j < 25; j++) {
    printf("%d\n", j);
    sleep(1);
  }
}
void * thread_function(void * arg) {
  // the work to be done by the thread is defined in this function
  printf("Inside Thread\n");
  for (i = 0; i < 5; i++) {
    printf("%d\n", i);
    sleep(1);
  }
}
```

ตัวอย่างถัดไปแสดงการทำงานแยกกันของ Thread กับ main Process

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>

void * thread_function(void * arg); //user define function
int i, j;
int main() {
  pthread_t a_thread; //thread declaration
  pthread_create( & a_thread, NULL, thread_function, NULL);
  //thread is created
  pthread_join(a_thread, NULL); //process waits for thread to finish . //Comment this line to see the difference
  printf("Inside Main Program\n");
  for (j = 20; j < 25; j++) {
    printf("%d\n", j);
    sleep(1);
  }
}
void * thread_function(void * arg) {
  // the work to be done by the thread is defined in this function
  printf("Inside Thread\n");
  for (i = 0; i < 5; i++) {
    printf("%d\n", i);
    sleep(1);
  }
}
```

ตัวอย่างถัดไปแสดงการทำงานแยกกันของ Thread กับ main Process

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>

void * thread_function(void * arg); //user define function
int i, j;
int main() {
  pthread_t a_thread; //thread declaration
  pthread_create( & a_thread, NULL, thread_function, NULL);
  //thread is created
  pthread_join(a_thread, NULL); //process waits for thread to finish . //Comment this line to see the difference
  printf("Inside Main Program\n");
  for (j = 20; j < 25; j++) {
    printf("%d\n", j);
    sleep(1);
  }
}
void * thread_function(void * arg) {
  // the work to be done by the thread is defined in this function
  printf("Inside Thread\n");
  for (i = 0; i < 5; i++) {
    printf("%d\n", i);
    sleep(1);
  }
}
```

ตัวอย่างถัดไปแสดงการทำงานแยกกันของ Thread กับ main Process

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>

void * thread_function(void * arg); //user define function
int i, j;
int main() {
  pthread_t a_thread; //thread declaration
  pthread_create( & a_thread, NULL, thread_function, NULL);
  //thread is created
  pthread_join(a_thread, NULL); //process waits for thread to finish . //Comment this line to see the difference
  printf("Inside Main Program\n");
  for (j = 20; j < 25; j++) {
    printf("%d\n", j);
    sleep(1);
  }
}
void * thread_function(void * arg) {
  // the work to be done by the thread is defined in this function
  printf("Inside Thread\n");
  for (i = 0; i < 5; i++) {
    printf("%d\n", i);
    sleep(1);
  }
}
```

ตัวอย่างถัดไปแสดงการทำงานแยกกันของ Thread กับ main Process

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>

void * thread_function(void * arg); //user define function
int i, j;
int main() {
  pthread_t a_thread; //thread declaration
  pthread_create( & a_thread, NULL, thread_function, NULL);
  //thread is created
  pthread_join(a_thread, NULL); //process waits for thread to finish . //Comment this line to see the difference
  printf("Inside Main Program\n");
  for (j = 20; j < 25; j++) {
    printf("%d\n", j);
    sleep(1);
  }
}
void * thread_function(void * arg) {
  // the work to be done by the thread is defined in this function
  printf("Inside Thread\n");
  for (i = 0; i < 5; i++) {
    printf("%d\n", i);
    sleep(1);
  }
}
```

ตัวอย่างถัดไปแสดงการทำงานแยกกันของ Thread กับ main Process

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>

void * thread_function(void * arg); //user define function
int i, j;
int main() {
  pthread_t a_thread; //thread declaration
  pthread_create( & a_thread, NULL, thread_function, NULL);
  //thread is created
  pthread_join(a_thread, NULL); //process waits for thread to finish . //Comment this line to see the difference
  printf("Inside Main Program\n");
  for (j = 20; j < 25; j++) {
    printf("%d\n", j);
    sleep(1);
  }
}
void * thread_function(void * arg) {
  // the work to be done by the thread is defined in this function
  printf("Inside Thread\n");
  for (i = 0; i < 5; i++) {
    printf("%d\n", i);
    sleep(1);
  }
}
```

OUTPUT

```
0
1
2
3
4
Inside Main Program
20
21
22
23
24
```

OUTPUT

```
0
1
2
3
4
Inside Main Program
20
21
22
23
24
```

OUTPUT

```
0
1
2
3
4
Inside Main Program
20
21
22
23
24
```

```c
#include<unistd.h>
#include<pthread.h>

void * thread_function(void * arg);
int i, j;
int main() {
  pthread_t a_thread; //thread declaration
  pthread_create( & a_thread, NULL, thread_function, NULL);
  //thread is created
  printf("Inside Main Program\n");
  for (j = 50; j < 60; j++) {
    printf("Main %d\n", j);
    sleep(1);
  }
  printf("Main thread end\n");
  pthread_join(a_thread, NULL); //process waits for thread to finish . //Comment this line to see the difference
  printf("Thread end\n");
}
void * thread_function(void * arg) {
  // the work to be done by the thread is defined in this function
  printf("Inside Thread\n");
  for (i = 0; i < 50; i++) {
    printf("Thread %d\n", i);
    sleep(1);
  }
```


0:00

```c
#include<pthread.h>

void * thread_function(void * arg);
int i, j;
int main() {
  pthread_t a_thread; //thread declaration
  pthread_create( & a_thread, NULL, thread_function, NULL);
  //thread is created
  printf("Inside Main Program\n");
  for (j = 50; j < 60; j++) {
    printf("Main %d\n", j);
    sleep(1);
  }
  printf("Main thread end\n");
  // pthread_join(a_thread, NULL); //process waits for thread to finish . //Comment this line to see the difference
  printf("Thread end\n");
}
void * thread_function(void * arg) {
  // the work to be done by the thread is defined in this function
  printf("Inside Thread\n");
  for (i = 0; i < 50; i++) {
    printf("Thread %d\n", i);
    sleep(1);
  }
}
```


0:00

การยกเลิก Thread

- เธรตที่จะถูกยกเลิกเรียกว่าเธรตเป้าหมาย (target thread)
- การยกเลิกทันที (Asynchronous Cancellation) หยุดการทำงานของเธรตทันที ผู้ใช้ไม่ต้องรอ (asynchronous) แต่เธรตอาจจะไม่มีโอกาสได้คืนทรัพยากรที่สำคัญ ไม่แนะนำให้ใช้
- การยกเลิกแบบถ่วงเวลา (Deferred Cancellation) เธรตจะตรวจเป็นระยะว่าผู้ใช้ต้องการให้ตนหยุดทำงานหรือไม่ เธรตจึงมีโอกาสที่จะคืนทรัพยากรที่สำคัญก่อนหยุดทำงาน แต่ในระหว่างนี้ผู้ใช้อาจจะต้องรอ