แบบฝึกหัด บทที่ 3

1. ฟังก์ชันfork( ) ได้สร้างโปรเซสขึ้นมาและคัดลอกค่าทั้งหมดมาจากโปรเซสเดิม โดยที่โปรเซส  
 ใหม่นั้นมีหน่วยความจำเป็นของตัวเองและมีค่าที่คัดลอกมาทั้งหมด

if (pid == 0) {

value += 15;

return 0;

}  
 ในโปรเซสใหม่นั้น ได้returnค่าไปที่ค่าpid โดยค่าที่returnไปคือ 0 และ ได้เพิ่มค่าเข้าไปอีก 15 ไป   
 ที่ตัวแปรvalue

else if (pid > 0) {

wait (NULL);

printf("Value = %d", value);

return 0;

}

ในโปรเซสดั้งเดิมนั้นมีตัวแปรvalueนั้นได้เก็บค่าไว้ทั้งหมด 5 และในโปรเซสดั้งเดิมมีค่าอยู่  
 มากกว่า 0 และได้ปริ้นค่าออกมาคือ 5

\*แต่หลังจากทำการทดลองแล้ว ได้ค่าออกมาเป็น0

2. จากcode มี fork( ); อยู่ทั้งหมด 3 บรรทัด โดยที่บรรทัดแรกสร้างโปรเซสขึ้นมา 1 โปรเซส  
 บรรทัดที่2 สร้างขึ้นมา 2 โปรเซส และบรรทัดที่ 3 สร้างขึ้นมา 4 โปรเซส และมีโปรเซสพ่อแม่  
 อีก 1 โปรเซส  
 \*สรุป มีโปรเซสทั้งหมด 7 โปรเซส

3. shared memory segments

4. - Short-Term Scheduling คือ ตัวจัดตารางระยะสั้น ใช้สำหรับเลือกโปรเซสที่อยู่ในสถานะรอ   
 เพื่อนำมาใช้เป็นหน่วยประมวลผลกลาง  
 - Medium-Term Scheduling คือ ตัวจัดตารางระยะกลาง ใช้สำหรับย้ายโปรเซสออกจาก   
 หน่วยความจำหลัก เพื่อลดจำนวนโปรเซสที่มีมากเกินไปในหน่วยความจำ เพื่อให้CPUมี   
 ประสิทธิ์ภาพในการทำงานดีและเร็วขึ้น  
 - Long-Term Scheduling คือ ตัวจัดตารางระยะยาว ใช้เลือกโปรเซสจากหน่วยเก็บข้อมูลเพื่อ  
 นำเข้าสู่หน่วยความจำ

5. สมมติ Context switch ระหว่างProcess-AและProcess-B  
 1. Process-A เข้าสู่ Kernal และเปลี่ยนจาก User mode เป็น Kernal mode  
 2. Process-A ใน Kernal ทำการContext switch เป็นProcess-B  
 3. Process-B ออกจาก Kernal และทำการเป็นจากKernal mode เป็น User mode

6. คือโปรเซสแรกสุดที่สร้างขึ้นตอนทำการเปิดระบบปฏิบัติการ

7. - Zombie Process คือ โปรเซสที่ทำงานเสร็จเร็วเกินไป จนโปรเซสพ่อแม่ทำการwaitingไม่ทัน   
 และพอwaiting ก็ไม่มีProcessไหนสั่งให้ทำงาน จนทำให้โปรเซสนั้นอยู่ในสถานะwaiting   
 ตลอดไป  
 - Orphan Process คือ โปรเซสที่โปรเซสพ่อแม่ทำการปิดตัวเร็วเกินไป ในขณะที่โปรเซสลูกยังทำงาน  
 ไม่เสร็จ ทำให้โปรเซสลูกทำงานต่อไปโดยที่ไม่มีโปรเซสพ่อแม่

แบบฝึกหัด บทที่ 4

1. 1.1 – User-level threads การทำงานของ thread จะเป็นแบบ Many to one คือ ระบบจะทำการวาง   
 threadทั้งหมดของMultithread process ไว้ใน Execution context เพียงตัวเดียว   
 และทำงานอยู่ในพื้นที่ของUser ไม่เกี่ยวข้องกับของพื้นที่Kernal ทำให้ไม่สามารถใช้งาน   
 Kernalได้โดยตรง  
 - Kernal-level threads การทำงานจะเป็นแบบ One to one คือ ระบบจะวาง Threadทั้งหมด   
 ไว้ในExecution context ของตัวเอง ในการวาง Thread แต่ละตัวจะต้องให้ระบบปฏิบัติการ-  
 ช่วยในเรื่อง Mapping ตัว User-level threads แต่ละตัวเข้ากับ Kernal-level threads   
 1.2 - Kernal-level threads และ User-level threads เหมาะสำหรับพัฒนาThreadบน   
 ระบบปฏิบัติการที่ไม่สนับสนุนThread  
 1.3 - Kernal-level threads เหมาะสำหรับแอฟพิเคชั่นที่ต้องการความละเอียดสูง

2. เมื่อ Kernal thread ได้รับความเสียหายจากอีกKernal

3. Mutithread process จะแชร์ Heap memory และ Global variables ระหว่างMutithread process   
 และแต่ละThread จะมีRegister ValuesและStack Memory แยกกัน

4. - 6 โปรเซส  
 - 8 Thread

5. Thread Pool คือคลาสที่ไว้ใช้ทำงานกี่ยวกับMultithread   
 Thread pool นั้นทำงานแบบBackground Thread และแอฟจะหยุดทำงานทันทีที่   
 Foreground Tread สิ้นสุดการทำงาน

6. Java Thread

**public** **class** demo1 **extends** Thread{

**public** **static** **int** *count* = 0;

**public** **static** **synchronized** **void** count() {

*count*++;

}

**public** **static** **void** main(String[] args) {

Thread t1 = **new** Thread(**new** Runnable() {

**public** **void** run() {

**for**(**int** i=0; i<100; i++) {

*count*();

System.***out***.println(Thread.*currentThread*().getName());

}

}

});

Thread t2 = **new** Thread(**new** Runnable() {

**public** **void** run() {

**for**(**int** i=0; i<100; i++) {

*count*();

System.***out***.println(Thread.*currentThread*().getName());

}

}

});

t1.start();

t2.start();

**try** {

t1.join();

t2.join();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.***out***.println("count= "+*count*);

}

}

พฤติกรรมของโปรแกรมนี้คือ จะให้Threadทั้ง2ตัวทำงานพร้อมกันและนับจำนวนThreadที่ทำงานทั้งหมดตอนจบการทำงาน

ผลลัพธ์ที่ได้คือ

Thread-0

Thread-0

Thread-0

Thread-0

Thread-0

Thread-0

Thread-0

Thread-0

Thread-0

Thread-0

Thread-1

Thread-1

Thread-1

Thread-1

Thread-1

Thread-1

Thread-1

Thread-1

Thread-1

Thread-1

count= 20

7. Line-C = 5 Line-P = 0