รายงานวิชา Operating Systems

หัวข้อที่เลือก

- Copy on Write
- Deadlocks

ชื่อนิสิต

นายจิรเมธ วัฒนไพบูลย์ เลขประจำตัว 6510503263

บทคัดย่อ (Abstract)

รายงานนี้นำเสนอการทดลองสองหัวข้อของวิชา OS ได้แก่ (1) การสาธิต Copy-on-Write (CoW) หลังการ fork() และ (2) การสาธิตภาวะเดดล็อก ครอบคลุมการหลีกเลี่ยง (avoidance), การตรวจจับ (detection) และการแก้ไข (resolution) โดยยึดหลัก DO NO HARM และทำงานบน Linux เท่านั้น งาน CoW มีวัตถุประสงค์เพื่ออธิบายการแชร์เพจระหว่างโปรเซสหลัก/ลูกและผลของการเขียนที่กระตุ้นให้ระบบคัดลอกเพจเป็นส่วนตัว วิธีทดลองคือจองหน่วยความจำขนาด 50/75/100 MB แตะหนึ่งไบต์ต่อเพจเพื่อคอมมิต จากนั้น fork() วัดค่า VmRSS ของทั้งสองโปรเซสแบบซิงก์ด้วยท่อ แล้วให้โปรเซสลูกเขียนหนึ่งไบต์ต่อเพจ ผลสำคัญคือหลัง fork() ค่า VmRSS ของทั้งสองใกล้เคียงกัน และหลังลูกเขียน ค่า VmRSS ของลูกเพิ่มขึ้น ขณะที่ของพ่อคงเดิม แนวโน้มการเพิ่มชัดขึ้นตามขนาดหน่วยความจำที่ทดสอบ

ในงานเดดล็อก เราจำลองทรัพยากร (เช่น เวกเตอร์รวม [3,3,2]) และใช้เธรดแทนโปรเซส เพื่อความปลอดภัย โหมด avoidance ใช้ Banker's Algorithm อนุมัติคำขอเฉพาะเมื่อยังคงอยู่ในสถานะปลอดภัย จึงจบได้โดยไม่เกิดเดดล็อก ส่วนโหมด detection อนุมัติตามมีและบล็อกเมื่อไม่พอ โดยตัวตรวจจับสร้าง Wait-for Graph ค้นหาวงจรและยุติเหยื่อ เพื่อคืนทรัพยากร ผลลัพธ์แสดงให้เห็นว่า avoidance ป้องกันเดดล็อกได้อย่างเป็นระบบ ขณะที่ detection + resolution สามารถปลดล็อกระบบเมื่อเกิดวงจรจริง พร้อมข้อแลกเปลี่ยนด้านเวลารอและโอเวอร์เฮดในการตรวจจับ/แก้ไข

คำเตือน / DO NO HARM

- โค้ดทั้งหมดทำขึ้นเพื่อการศึกษาเท่านั้น
- ข้อจำกัดและมาตรการลดผลกระทบ เช่น จำกัดหน่วยความจำ, ใช้ทรัพยากรจำลอง, ไม่แตะทรัพยากรระบบจริง
- เครื่องมือ/แพลตฟอร์มที่รองรับ (Linux เท่านั้นในบางส่วน)

1. ภาพรวมงาน (Overview)

- เลือกทำข้อ 2 และ 3 ตามใบงาน "OS Report 2025"
- รายการไฟล์ที่ส่ง: ส่งเป็นไฟล์ zip ที่มีโครงสร้างโฟลเดอร์, README, ไฟล์โค้ดหลัก, BUILD, รายงานฉบับนี้

2. Copy-on-Write (CoW)

2.1 วัตถุประสงค์

แสดงพฤติกรรม CoW หลัง fork() โดยสังเกต VmRSS ของ parent/child ก่อนและหลังการเขียนในโปรเซสลูก

2.2 แนวคิดและภาพรวมการทดลอง

Copy-on-Write (CoW) คือกลไกที่หลังการ fork() แล้ว พื้นที่หน่วยความจำของโปรเซสลูกและโปรเซสหลักจะอ้างถึง page เดียวกันไปก่อน เมื่อมีการเขียนลง page ใด ๆ ในโปรเซสหนึ่ง ระบบปฏิบัติการจึงค่อยทำสำเนา page นั้นให้เป็นของโปรเซสนั้นโดยเฉพาะ ส่งผลให้เกิดการเพิ่มขึ้นของ RSS เฉพาะในโปรเซสที่เขียน ส่วนอีกโปรเซสยังใช้ page ร่วมตามเดิม

การทดลองของเราประกอบด้วยขั้นตอน:

1. จองหน่วยความจำขนาดมากพอให้สังเกตได้โดยค่าเริ่มต้น 50/75/100 MB และ touch per page เพื่อให้ page ถูก commit ใน RAM
2. fork() สร้างโปรเซสลูก
3. วัดและพิมพ์ค่า VmRSS ของทั้งโปรเซสหลักและโปรเซสลูกทันทีหลัง fork() จะเห็นว่าค่าใกล้เคียงกัน
4. ให้โปรเซสลูกเขียนลงหนึ่งไบต์ต่อหนึ่ง page เพื่อกระตุ้นให้เกิด CoW
5. วัด VmRSS ของโปรเซสลูกอีกครั้งและวัดของโปรเซสหลัก
6. ทำซ้ำด้วยขนาดข้อมูลที่ต่างกันเพื่อสังเกตแนวโน้ม

เพื่อให้ลำดับการวัดแน่นอน โปรแกรมใช้ pipe สำหรับซิงโครไนซ์ เช่น Child พิมพ์ค่าแรก -> แจ้ง Parent -> Parent สั่งเริ่มเขียน -> Child เขียนและพิมพ์ค่าใหม่ -> แจ้ง Parent -> Parent พิมพ์ค่าหลังเด็กเขียน

2.3 ผลการทดลอง

วิธีรันย่อ:

➢ *python3 2_cow_6510503263/cow.py --sizes 50,75,100 --smaps*

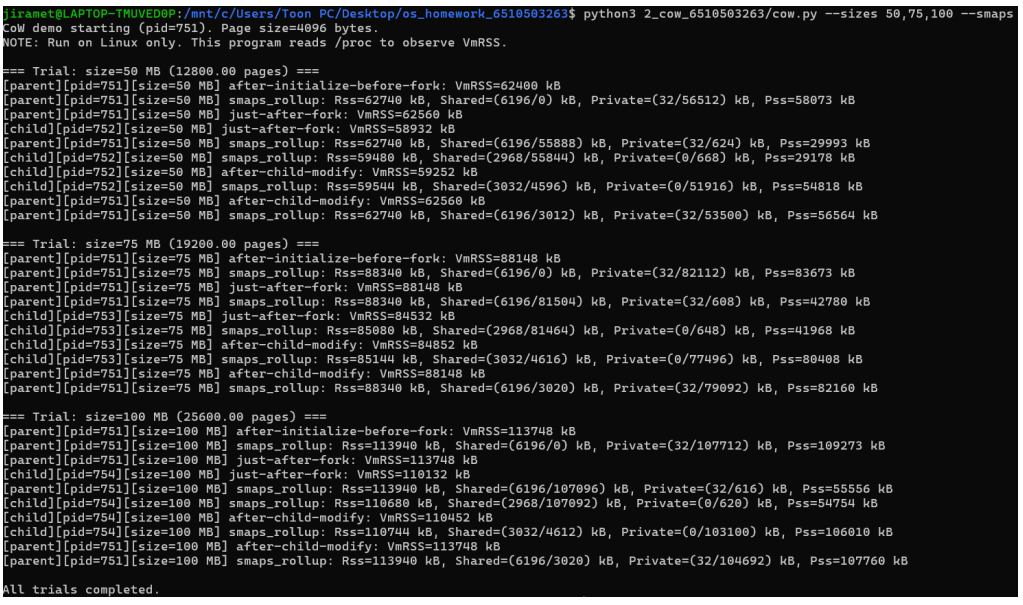
ตาราง A บันทึกค่า VmRSS (kB) จากการรันจริงบนเครื่องของผู้ทำรายงาน

ขนาด (MB)	VmRSS Parent: ก่อน fork	VmRSS Parent: หลัง fork	VmRSS Child: หลัง fork	VmRSS Child: หลังแก้ไข	VmRSS Parent: หลัง Child แก้ไข

50	62,400	62,560	58,932	59,544	62,560
75	88,148	88,148	84532	85,144	88,148
100	113,748	113,748	110132	110,744	113,748

ตาราง B บันทึกค่า smaps_rollup (kB) ของ Child (คีย์ที่ชี้ CoW)

ขนาด (MB)	**Phase**	**RSS**	**Shared_ Clean**	**Shared_ Dirty**	**Private_ Clean**	**Private_ Dirty**	**PSS**
50	หลัง fork	59,480	2,968	55,844	0	668	29,178
	หลังแก้ไข	59,544	3,032	4,596	0	51,916	54,818
75	หลัง fork	85,808	2,968	84,164	0	648	41,968
	หลังแก้ไข	85,144	3,032	4,616	0	77,496	80,408
100	หลัง fork	110,680	2,968	107,092	0	620	54,754
	หลังแก้ไข	110,744	3,032	4,612	0	103,100	106,010



รูปที่ 1: Console log ของรอบ 50/75/100 MB แสดงค่า VmRSS และ smaps_rollup (Shared/Private) ต่อเฟส

2.4 การวิเคราะห์

ภาพรวมจากตาราง A (VmRSS):

- ค่า VmRSS ของ Parent ก่อน/หลัง fork แทบไม่เปลี่ยน แปลว่า parent ไม่ได้สร้างสำเนาหน้าเพจใด ๆ เพิ่มหลังจาก fork
- Child หลัง fork มี VmRSS ใกล้เคียงกับ Parent เพราะ VmRSS นับเพจที่อาศัยอยู่ใน RAM ทั้งที่แชร์และเป็นของตนเอง ดังนั้นช่วงนี้ parent/child จึงนับเพจร่วมกัน CoW ยังไม่เกิด
- หลัง Child แก้ไข ค่า VmRSS ของ Child แทบไม่เปลี่ยน ต่างเพียงหลักร้อยกิโลไบต์ สาเหตุคือ VmRSS ไม่บอกความเป็นเจ้าของของเพจ แต่บอกจำนวนเพจที่อาศัยอยู่ใน RAM ซึ่งยังเท่าเดิม แม้สถานะจะเปลี่ยนจาก shared เป็น private

ภาพรวมจากตาราง B (smaps_rollup ของ Child):

ชี้ CoW ได้ชัดเจนกว่ามาก เมื่อเทียบ “หลัง fork” กับ “หลังแก้ไข”:

- Private_Dirty เพิ่มขึ้นเกือบเท่าขนาดที่ทดสอบ ในแต่ละรอบ
  - 50 MB: +51,248 kB
  - 75 MB: +76,848 kB
  - 100 MB: +102,480 kB
- Shared_Dirty ลดลงพอ ๆ กัน ย้ายจากเพจที่แชร์ไปเป็นเพจของ Child เอง
- Shared_Clean ~3 MB คงที่ สื่อว่าหน้ารหัส/ไลบรารีแบบอ่านอย่างเดียว ยังคงแชร์ได้
- RSS ของ Child แทบคงที่
- PSS เพิ่มเกือบ 2 เท่า เพราะเมื่อนับแบบถ่วงตามสัดส่วนการแชร์เพจส่วนใหญ่กลายเป็นของ Child เอง
  - 50 MB: 29,178 -> 54,818 kB (≈1.88×)
  - 75 MB: 41,968 -> 80,408 kB (≈1.92×)
  - 100 MB: 54,754 -> 106,010 kB (≈1.94×)

ตีความเชิงกลไก:

1. หลัง fork เพจของบัฟเฟอร์ถูกทำเครื่องหมาย read-only และ แชร์ ระหว่าง parent/child (Shared_Dirty สูง, Private_Dirty ต่ำมาก)
2. เมื่อ Child เขียนทีละหน้า kernel จึง คัดลอกหน้า นั้นมาทำเป็นของ Child (เกิด Private_Dirty สูงขึ้นเท่าขนาดงาน) ขณะที่ Shared_Dirty ดรอปลง
3. ระบบรวม (system-wide) จึงใช้ RAM มากขึ้นตามจำนวนเพจที่ถูกคัดลอก

2.5 บทสรุปย่อย

- ผลทดลอง ยืนยัน CoW ตรงตามทฤษฎี: ทันทีหลัง fork ทั้งสองโปรเซสแชร์เพจกัน ทำให้ VmRSS ใกล้เคียงกันมาก
- เมื่อ Child แก้ไขข้อมูล โควต้าเพจของตนเพิ่มขึ้นอย่างชัดเจนใน Private_Dirty และ Shared_Dirty ลด

ลง ตามสัดส่วน

- VmRSS ไม่ใช่ตัวชี้วัดความเป็นเจ้าของเพจ จึงแทบไม่เปลี่ยน แต่ PSS สะท้อนการเลิกแชร์ได้ดี เกือบ x2 ทุกขนาด
- โดยสรุป: CoW ช่วยประหยัดหน่วยความจำได้จนกว่าจะมีการเขียน และเมื่อเขียนจริง ค่า private จะพุ่งขึ้นเท่าขนาดงาน ขณะที่ parent ไม่ได้รับผลกระทบ VmRSS ของ parent คงเดิม และเพจอ่านอย่างเดียว Shared_Clean ยังคงแชร์ต่อไปได้อย่างปลอดภัย

3. Deadlocks (Avoidance, Detection, Resolution)

3.1 วัตถุประสงค์

- แสดง Coffman's Deadlock Conditions ด้วยสถานการณ์จำลอง
- สาธิต Deadlock Avoidance โดยใช้ Banker's Algorithm
- สาธิต Deadlock Detection โดยสร้าง Wait-for Graph (WFG)
- สาธิต Deadlock Resolution โดยการยกเลิก (abort) เธรดที่เป็นเหยื่อเพื่อคืนทรัพยากร

3.2 แนวคิดและภาพรวมการทดลอง

องค์ประกอบของการจำลอง

- Simulated Resources: เวกเตอร์total = [3,3,2] (ค่าเริ่มต้น) และกำหนด max demand ของแต่ละเธรดแบบสุ่มในช่วง 0 0 จนถึงทั้งหมด
- Processes/Threads: จำนวนเริ่มต้น 5 เธรด
- Allocation/Need: ใช้โครงสร้างเดียวกับปัญหาการจัดสรรทรัพยากร (RAG) และ Banker's algorithm

Coffman's Deadlock Conditions

1. Mutual Exclusion: ทรัพยากรแต่ละชนิดมีจำนวนจำกัด จึงใช้ร่วมกันแบบแบ่งชิ้นไม่ได้
2. Hold and Wait: เธรดสามารถถือครองทรัพยากรบางส่วนและร้องขอเพิ่มได้
3. No Preemption: ในโหมด detection เธรดไม่ถูกพรากทรัพยากรคืนโดยอัตโนมัติ จนกว่าจะเข้าสู่ขั้น resolution
4. Circular Wait: โหมด detection จะทำให้เกิดสถานการณ์รอเป็นวงจรได้ และตัวตรวจจับจะสร้าง WFG เพื่อหาวงจรดังกล่าว

โหมดการทำงาน

- Avoidance: ใช้ Banker's Algorithm ตรวจสอบก่อนอนุมัติคำขอ หากเข้าภาวะไม่ปลอดภัยจะรอ ทำให้ระบบไม่มี deadlock
- Detection: ใช้การอนุมัติแบบ naive พร้อมตัวตรวจจับที่สร้าง WFG เป็นระยะ ๆ หากพบวงจรจะเลือกเหยื่อหนึ่งตัว เช่น ผู้ที่มี allocation รวมมากที่สุดในวงจร เพื่อ abort และปล่อย

ทรัพยากร

การทดลองของเราประกอบด้วยขั้นตอน:

1. สร้าง max demand แบบสุ่มต่อเธรดจาก total
2. เธรดแต่ละตัวขอทรัพยากรทีละน้อย (random bounded request) จนกว่าจะครบ max
3. โหมด avoidance: ตรวจสภาพปลอดภัยก่อนให้ (safe state) -> สำเร็จทั้งหมด ไม่มี deadlock
4. โหมด detection: อนุมัติได้เมื่อมีพอ มิฉะนั้นเธรดจะรอ -> ตัวตรวจจับสร้าง WFG และหาวงจร -> ทำ Resolution โดย abort อย่างน้อยหนึ่งเธรด -> ระบบเดินหน้าต่อ

3.3 ผลการทดลอง

วิธีรันย่อ:

➢ *python3 3_deadlock_6510503263/deadlock.py --mode avoidance*

➢ *python3 3_deadlock_6510503263/deadlock.py --mode detection --seed 1 --n 5 --resources 3,3,2*

โหมด Avoidance (Banker's Algorithm)

- มีทั้ง [avoid][grant] และบางจังหวะ [avoid][wait]
- ลำดับจบ: P0 ⟶ P3 ⟶ P2 ⟶ P1 ⟶ P4

โหมด Detection (+Resolution)

- พบวงจร 3 ครั้ง และมีการ abort เหยื่อเพื่อแก้ deadlock ต่อเนื่อง
    1. cycle: [2, 3, 2] -> abort P2
    2. cycle: [3, 4, 3] -> abort P4
    3. cycle: [1, 3, 1] -> abort P1
- หลังแก้ไข ระบบเดินหน้าต่อ จบด้วย P3 finished

ตาราง A: สรุปต่อโหมด

โหมด	**Total**	**#Proc**	**ลักษณะการทำงานที่เห็น**	**พบวงจร?**	**เหยื่อที่ถูก abort**	**ผลลัพธ์**
avoidance	[3,3,2]	5	มี grant และบางจังหวะ wait	ไม่	–	ทุกโปรเซส finished

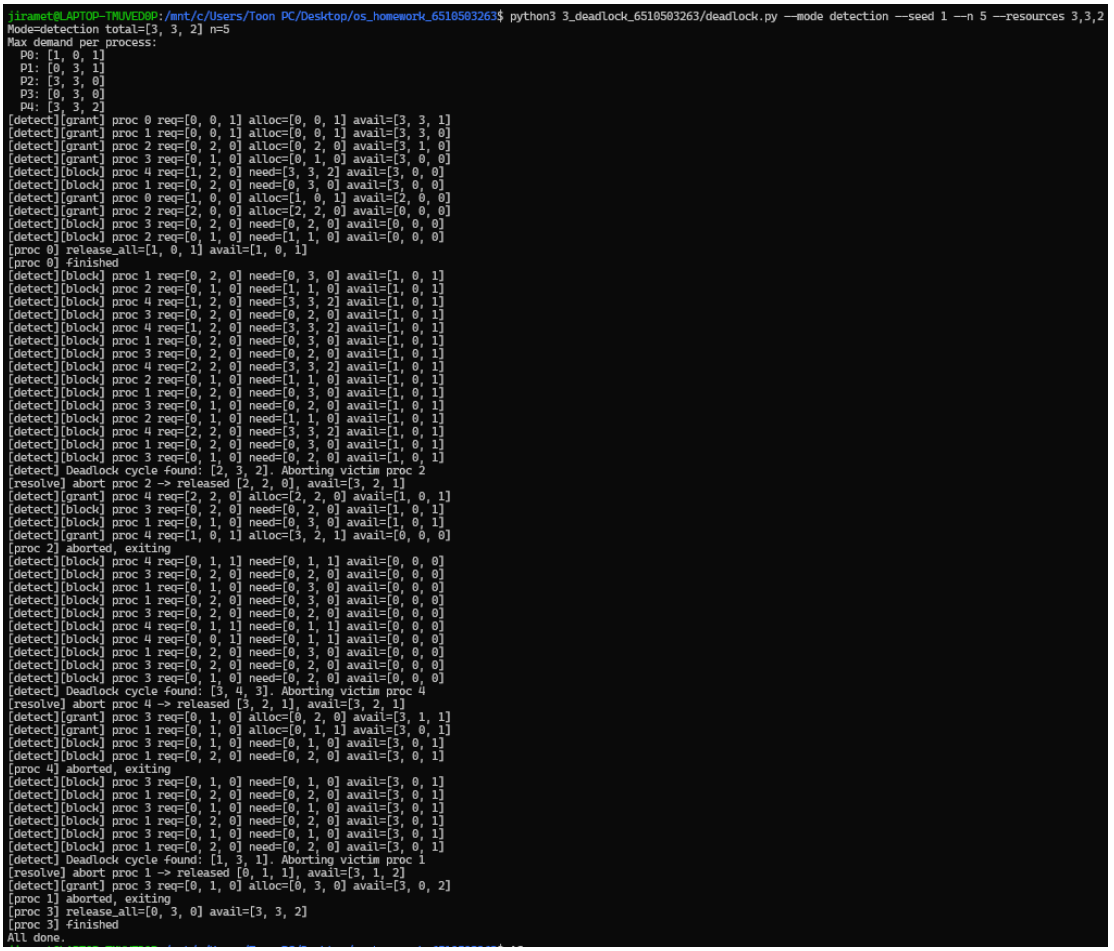
detection	[3,3,2]	5	มี block ต่อเนื่อง, ตัวตรวจจับหาวงจร	ใช่ (3 ครั้ง)	P2, P4, P1	ระบบเดินหน้าต่อหลัง abort, จบ “All done.”

ตาราง B: ไทม์ไลน์เหตุการณ์สำคัญ สำหรับโหมด Detection

ลำดับ	เหตุการณ์	ผลกับทรัพยากร & ระบบ
1	พบ cycle [2,3,2]	abort P2 -> ปล่อยทรัพยากรของ P2 -> โปรเซสอื่นขยับได้
2	พบ cycle [3,4,3]	abort P4 -> ระบบมี avail เพิ่มขึ้นอีก
3	พบ cycle [1,3,1]	abort P1 -> เปิดทางให้ P3 ขอครบและ finished
4	สรุป	P0, P3 จบ; ระบบ “All done.”

```
jiramet@LAPTOP-TMUVED0P:/mnt/c/Users/Toon PC/Desktop/os_homework_6510503263$ python3 3_deadlock_6510503263/deadlock.py --mode avoidance
Mode=avoidance total=[3, 3, 2] n=5
Max demand per process:
  P0: [0, 2, 0]
  P1: [3, 2, 0]
  P2: [0, 3, 2]
  P3: [2, 2, 0]
  P4: [1, 1, 1]
[avoid][grant] proc 0 req=[0, 2, 0] alloc=[0, 2, 0] avail=[3, 1, 2]
[avoid][grant] proc 1 req=[1, 0, 0] alloc=[1, 0, 0] avail=[2, 1, 2]
[avoid][grant] proc 2 req=[0, 0, 2] alloc=[0, 0, 2] avail=[2, 1, 0]
[avoid][grant] proc 3 req=[0, 1, 0] alloc=[0, 1, 0] avail=[2, 0, 0]
[avoid][wait ] proc 4 req=[0, 0, 1] need=[1, 1, 1] avail=[2, 0, 0]
[proc 0] release_all=[0, 2, 0] avail=[2, 2, 0]
[proc 0] finished
[avoid][wait ] proc 4 req=[0, 0, 1] need=[1, 1, 1] avail=[2, 2, 0]
[avoid][grant] proc 2 req=[0, 1, 0] alloc=[0, 1, 2] avail=[2, 1, 0]
[avoid][grant] proc 3 req=[0, 1, 0] alloc=[0, 2, 0] avail=[2, 0, 0]
[avoid][wait ] proc 1 req=[2, 1, 0] need=[2, 2, 0] avail=[2, 0, 0]
[avoid][wait ] proc 2 req=[0, 1, 0] need=[0, 2, 0] avail=[2, 0, 0]
[avoid][grant] proc 3 req=[1, 0, 0] alloc=[1, 2, 0] avail=[1, 0, 0]
[avoid][grant] proc 3 req=[1, 0, 0] alloc=[2, 2, 0] avail=[0, 0, 0]
[proc 3] release_all=[2, 2, 0] avail=[2, 2, 0]
[proc 3] finished
[avoid][grant] proc 2 req=[0, 1, 0] alloc=[0, 2, 2] avail=[2, 1, 0]
[avoid][wait ] proc 4 req=[0, 0, 1] need=[1, 1, 1] avail=[2, 1, 0]
[avoid][wait ] proc 1 req=[2, 1, 0] need=[2, 2, 0] avail=[2, 1, 0]
[avoid][grant] proc 2 req=[0, 1, 0] alloc=[0, 3, 2] avail=[2, 0, 0]
[proc 2] release_all=[0, 3, 2] avail=[2, 3, 2]
[proc 2] finished
[avoid][grant] proc 4 req=[0, 0, 1] alloc=[0, 0, 1] avail=[2, 3, 1]
[avoid][grant] proc 1 req=[2, 1, 0] alloc=[3, 1, 0] avail=[0, 2, 1]
[avoid][grant] proc 1 req=[0, 1, 0] alloc=[3, 2, 0] avail=[0, 1, 1]
[avoid][wait ] proc 4 req=[1, 1, 0] need=[1, 1, 0] avail=[0, 1, 1]
[proc 1] release_all=[3, 2, 0] avail=[3, 3, 1]
[proc 1] finished
[avoid][grant] proc 4 req=[1, 1, 0] alloc=[1, 1, 1] avail=[2, 2, 1]
[proc 4] release_all=[1, 1, 1] avail=[3, 3, 2]
[proc 4] finished
All done.
```

รูปที่ 2: Console log โหมด Avoidance (Banker’s Algorithm) ด้วย Total resources = [3, 3, 2], n=5 แสดงเหตุการณ์ grant/wait/release ต่อเฟส จนได้ safe sequence: P0 ⟶ P3 ⟶ P2 ⟶ P1 ⟶ P4 และ ทุกโปรเซสเสร็จสมบูรณ์ (ไม่เกิดเดดล็อก)



รูปที่ 3: Console log โหมด Detection + Resolution (seed=1) ด้วย Total resources = [3, 3, 2], n=5 แสดงการ grant/block และการตรวจจับ WFG จนพบเดดล็อก [2,3,2], [3,4,3], [1,3,1] พร้อมการ abort เหยื่อ ตามลำดับ P2 ⟶ P4 ⟶ P1 ก่อนที่ P3 จะได้รับทรัพยากรครบและ finish

3.4 การวิเคราะห์

โหมด Avoidance (Banker's Algorithm):

- ก่อนอนุมัติคำขอทุกครั้ง ระบบจำลองใช้หลัก Banker เพื่อตรวจสอบว่า ถ้าจ่ายแล้ว ยังมีลำดับที่ทุกโปรเซสสามารถเสร็จได้หรือไม่ โดยใช้เวกเตอร์ Work = avail และตรวจเงื่อนไข Need[i] ≤ Work เพื่อสร้างลำดับปลอดภัย (safe sequence)
- ข้อสังเกตจากผลรัน
    - มีหลายจังหวะที่ ไม่อนุมัติ แม้มีทรัพยากรว่างบางส่วน เช่น P4 ถูกรอเมื่อ need=[1,1,1] แต่ avail ไม่เพียงพอให้เกิดลำดับปลอดภัย ระบบจึงชะลอเพื่อป้องกันการเกิดวงจร
    - เมื่อโปรเซสหนึ่งเสร็จและ release_all (โดยเฉพาะ P0 และ P3) ค่าของ avail เพิ่มขึ้น ทำให้ตรวจผ่าน Banker และอนุมัติคำขอของโปรเซสถัด ๆ ไปได้
    - ได้ลำดับปลอดภัยชัดเจน P0 ⟶ P3 ⟶ P2 ⟶ P1 ⟶ P4 และ ไม่มีเดดล็อกตลอดการทดลอง

- ข้อดีเชิงระบบ
    - ป้องกันเดดล็อกได้ 100% ตามแบบจำลอง
    - ไม่จำเป็นต้องยกเลิกงานหรือบังคับคืนทรัพยากร ทำให้ไม่สูญเสียงาน
- ข้อจำกัด
    - ต้องทราบ/ประมาณ Max demand ล่วงหน้า
    - อาจเกิดการรอที่ยาวขึ้นในบางช่วง (ลดการขนานบ้าง) เพื่อรักษาความปลอดภัยของสถานะ

โหมด Detection การสร้างและอ่าน WFG:

- โครงสร้างกราฟที่ปรากฏจริง
    - ก่อน Deadlock #1: ปรากฏ P2 <-> P3 และเส้นรอจาก P1, P4 ไปยังผู้ถือ R1 หลายตัว แสดงว่า R1 เป็นคอขวดหลัก
    - ก่อน Deadlock #2: หลัง P4 ได้รับอนุมัติชุดใหญ่ กลายเป็นผู้ถือทรัพยากรจำนวนมาก (R0,R1,R2) ทำให้เกิดวงจร P3 <-> P4 และมีขอบจาก P1 -> P4 เพิ่ม
    - ก่อน Deadlock #3: กลับมาที่คอขวด R1 อีกครั้ง เกิดวงจร P1 <-> P3 ชัดเจน
- บทเรียนที่ได้คือ คอขวดชัดเจนที่ R1 เกิดทั้งใน DL#1 และ DL#3 การให้สิทธิ์ทรัพยากรกับโปรเซสเดียวจำนวนมาก เพิ่มความเสี่ยงที่กราฟจะปิดเป็นวงจรกับผู้อื่น

Resolution วิธีแก้ Deadlock ที่ใช้จริงในผลรัน:

- ยกเลิกโปรเซสเหยื่อ (Abort) เพื่อคืนทรัพยากรทั้งหมดทันที
    - รอบที่ 1: ยกเลิก P2 และ คืน [2,2,0]
    - รอบที่ 2: ยกเลิก P4 และ คืน [3,2,1]
    - รอบที่ 3: ยกเลิก P1 และ คืน [0,1,1]
- ผลกระทบหลังการแก้
    - ค่า avail เพิ่มขึ้นทันที ทำให้เส้นขอบใน WFG ลดลงหรือหายไป ส่งผลให้สามารถอนุมัติคำขอที่ค้างอยู่ได้ เช่น หลัง abort P4 มีการ grant ให้ P3, P1 ต่อทันที
    - หลังรอบที่ 3 P3 ได้รับ R1 ครบ จึงเสร็จและปล่อยคืนทั้งหมด กราฟว่างลง ระบบจบการทำงาน

ตีความเชิงกลไก:

1. สาเหตุหลักของ Deadlock
    - ทรัพยากร R1 เป็นคอขวดซ้ำ ๆ โดยเฉพาะเมื่อกระจุกอยู่ที่โปรเซสบางตัว P2, P3 และภายหลัง P4 ทำให้เกิดการรอไขว้กลับไปกลับมา
2. ความต่างระหว่างสองโหมด
    - Avoidance เหมาะกับระบบที่รู้เพดานความต้องการและไม่ยอมสูญเสียงาน และไม่มี deadlock แต่มีต้นทุนเป็นระยะเวลารอเพื่อรักษาสถานะปลอดภัย
    - Detection+Resolution ยืดหยุ่นกับสถานการณ์ที่คาดเดาความต้องการยาก ยอมให้

เกิด Deadlock จริง แล้วแก้ด้วยการยกเลิก แต่ต้องยอมรับการสูญเสียงานและความผันผวนของความก้าวหน้า

3.5 บทสรุปย่อย

- ผลทดลองยืนยันแนวคิด Avoidance (Banker's Algorithm) ตรวจสภาวะปลอดภัยก่อนอนุมัติทุกคำขอ ทำให้ได้ safe sequence P0⟶P3⟶P2⟶P1⟶P4 และทุกโปรเซสจบโดยไม่เกิดเดดล็อก
- ในโหมด Detection สร้าง Wait-for Graph (WFG) จากสถานะการถือ/ความต้องการ พบวงจร [2,3,2], [3,4,3], [1,3,1] โดยมี R1 เป็นคอขวดหลัก ช่วงที่ avail(R1)=0 จะเกิดการ block ต่อเนื่อง
- Resolution แก้เดดล็อกด้วยการยกเลิกเหยื่อ (abort) เพื่อคืนทรัพยากรทันที ลำดับที่เกิดคือ P2 ⟶ P4 ⟶ P1 ทำให้ avail กระโดดขึ้น กราฟคลายวงจร และโปรเซสที่เหลือได้รับการอนุมัติต่อจนเสร็จ
- โดยสรุป Avoidance เหมาะเมื่ออยากเลี่ยงการสูญงาน (zero-deadlock) ส่วน Detection+Resolution เหมาะเมื่อรูปแบบการขอทรัพยากรไม่แน่นอน แต่ต้องมีนโยบายเลือกเหยื่อที่ชัดเจน และจัดการคอขวด (เช่น เพิ่ม R1 หรือกำหนด request ordering) เพื่อให้ระบบคืบหน้าเสมอ