Runtime Environment

สภาวะแวดล้อมในขณะโปรแกรมทำงาน

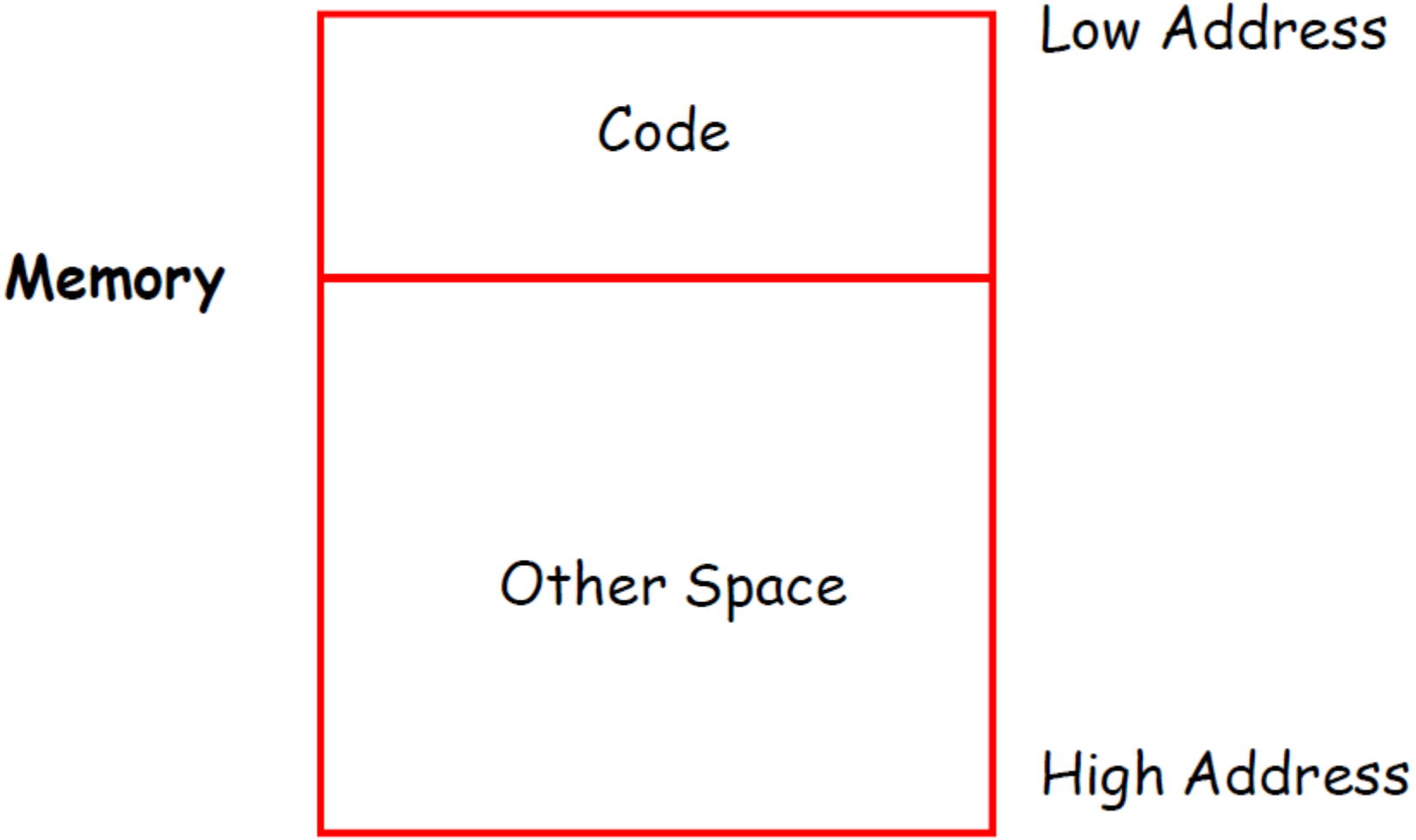
หัวข้อหลักวันนี้

- การจัดการและลักษณะของ executable code
- การจัดการทรัพยากรในขณะโปรแกรมทำงาน (management of runtime resources)
- ความสัมพันธ์ระหว่างส่วนที่เป็น static (compile-time) กับ dynamic (runtime)
- การจัดการหน่วยความจำ
- Activation Record (AR)

Runtime Resources

- การทำงานของโปรแกรมอยู่ภายใต้การควบคุมของระบบปฏิบัติการ (Operating Systems)
- ในขณะที่โปรแกรมจะเริ่มต้นทำงาน
  - OS จองพื้นที่สำหรับโปรแกรม
  - โหลดโค๊ดเข้าไปที่ address space ที่จะใช้ในการรันโปรแกรม
  - OS เปลี่ยน control flow ไปที่ entry point ของโปรแกรม
    - เช่นฟังก์ชั่น main

ตัวอย่างการใช้พื้นที่หน่วยความจำ



Virtual Memory

- Layout ของหน่วยความจำกำหนดโดย OS
- Address ทุกๆอันเป็น virtual address
  - อยู่เรียงติดกันจากน้อยไปมาก (หรือมากไปน้อย)
  - Physical address อาจจะแยกกันอยู่คนละ page
- เราสนใจในส่วนที่เกี่ยวข้องกับ user code (ที่เป็น machine instruction) ที่เราจะคอมไพล์จาก source code (ที่เป็นโปรแกรมในภาษาระดับสูง) และ load เข้าหน่วยความจำหลัก
- โปรแกรมที่จะรันจะต้องอยู่ในหน่วยความจำหลักก่อนเสมอ

การใช้พื้นที่ส่วน user code

- ส่วน code บรรจุ binary ที่เป็น machine instruction ที่จะถูกประมวลโดย CPU ฮาร์ดแวร์
- ส่วน other บรรจุ data
  - เราจะได้เจาะรายละเอียดในส่วนนี้ต่อไป
- หน้าที่ของคอมไพเลอร์คือ
  - ผลิตโค๊ดและ
  - ประสานงานการใช้ data ในส่วน other

การผลิตโค๊ด

- มีวัตถุประสงค์สองส่วน
  - ความถูกต้อง (correctness)
  - ความเร็วหรือสมรรถนะ (speed)
- ความซับซ้อน ยุ่งยาก ในการผลิตโค๊ดคือจะต้องพยายามบรรลุวัตถุประสงค์ทั้งสองนี้ไปพร้อมๆ กัน

Assumptions ในการรันโปรแกรม

- การรันโปรแกรม (program execution) เป็นแบบ sequential
  - การทำงานของโปรแกรมเริ่มจากจุดหนึ่งไปยังอีกจุดหนึ่งเรียงตามลำดับที่ชัดเจน
- เมื่อมีการเรียกใช้งาน procedure โปรแกรมจะกลับมาเริ่มทำงานต่อไป ณ คำสั่งที่อยู่หลังจากการเรียกใช้งาน procedure นั้น

Activations

- การเรียกใช้งาน procedure P คือ activation ของ P
- Lifetime (ช่วงอายุ) ของ activation ของ P คือ
  - ขั้นตอนการทำงานทั้งหมดที่จะ execute P
  - รวมไปถึงการเรียกใช้งาน P

Variable Lifetime

- Lifetime ของตัวแปร x คือส่วนของการรันโปรแกรมที่มีการนิยาม x (x is defined)
- Lifetime จะเกี่ยวข้องกับสิ่งที่จะรันในเวลาจริง (dynamic)
- Scope จะเกี่ยวข้องกับส่วนการคอมไพล์โปรแกรม (static)

Activation Tree

- อยู่บนพื้นฐานของ assumption การเรียกใช้งาน procedure
  - เมื่อ P เรียก Q จะได้ว่า Q return ก่อน P return
- Lifetime ของ Q จะ nested (ซ้อนทับฝังตัว) อยู่ใน lifetime ของ P
  - เป็นธรรมชาติของการ activate procedure
- เราสามารถแสดง activation lifetime ได้โดยใช้ tree

ตัวอย่าง activation tree

```
int g() {return 1;}
int f() {return g();}
int main() { g(); f(); return
0; }
```

ลองดูตัวอย่างต่อไปนี้

```
int g() {return 1;}
int f(int x) {if (x == 0) return g() else return
f(x-1);}
int main() { f(3); return 0; }
```

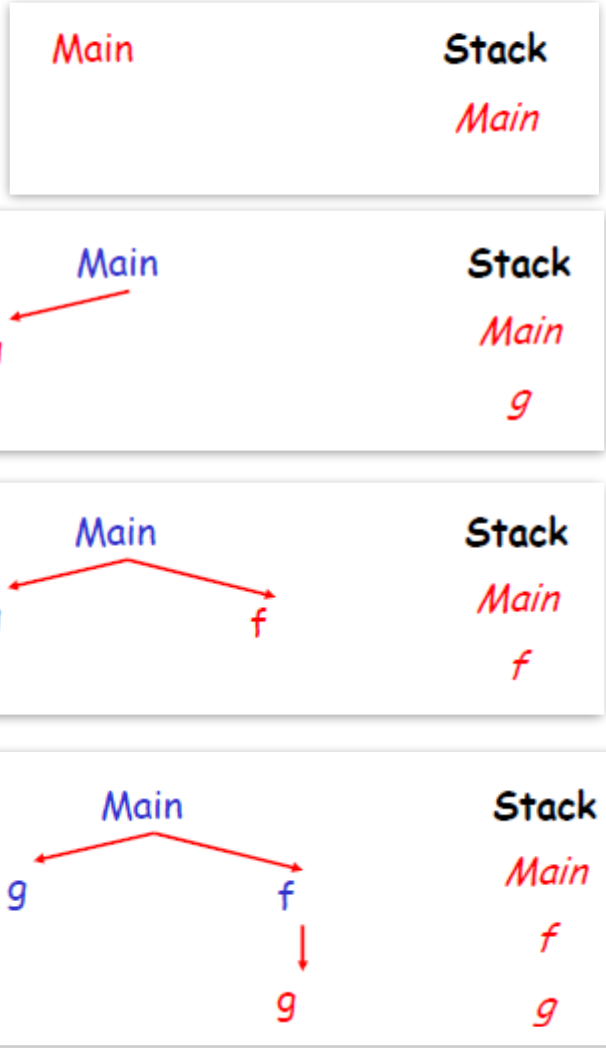
นิสิตเขียน diagram ของ activation tree ของโปรแกรมด้านบนได้หรือไม่

Activation Tree

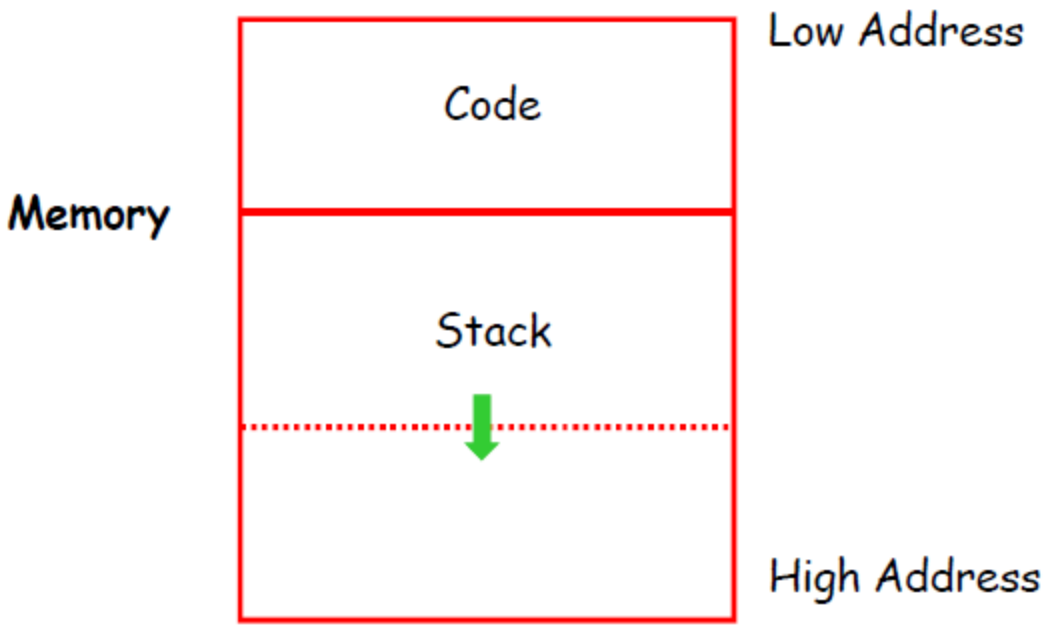
- ขึ้นอยู่กับพฤติกรรมของโปรแกรมขณะทำงาน
- อาจมีการเปลี่ยนแปลงถ้า input ของโปรแกรมมีการเปลี่ยนแปลง
- เนื่องจาก activation ของ procedure มีคุณสมบัติ nested เราสามารถติดตามบันทึก (track) การเรียกใช้งาน procedure ได้โดยใช้ stack
  - Procedure ที่ถูกเรียกครั้งล่าสุดจะต้อง return ออกมาก่อน
  - LIFO (Last-In First-Out)

ตัวอย่าง activation tree และ stack

```
int g() {return 1;}
int f() {return g();}
int main() { g(); f(); return
0; }
```



เพิ่ม stack ในพื้นที่หน่วยความจำ



Activation Records

- เกี่ยวกับชื่อ: activation record = record (การเก็บข้อมูล) ของ activation (การเรียกใช้งาน procedure)
- ข้อมูลที่ใช้จัดการ procedure activation อันใดอันหนึ่งเรียกว่า activation record (AR) หรือ frame
- ถ้า F เรียกใช้งาน G AR ของ G มีข้อมูลทั้งที่เกี่ยวกับ F และ G

เมื่อ procedure F เรียก G

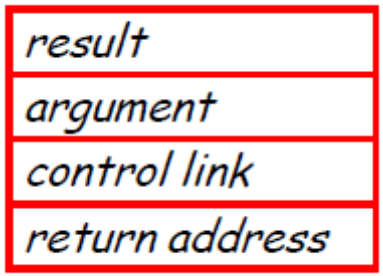
- F จะถูกแขวนอยู่ (suspend) จนกว่า G จะจบการทำงาน (return กลับออกมา)
- จากนั้น F จะทำงานต่อ (resume)
- AR ของ G จะต้องมีข้อมูลเพียงพอที่จะทำให้ F ทำงานต่อได้
- ARของ G บรรจุข้อมูลต่อไปนี้
  - ค่าที่ได้จากการ return ของ G
  - Actual parameter ที่ส่งผ่านไปให้ G โดย F
  - Local variable ใน G

สิ่งที่อยู่ใน AR ของ G

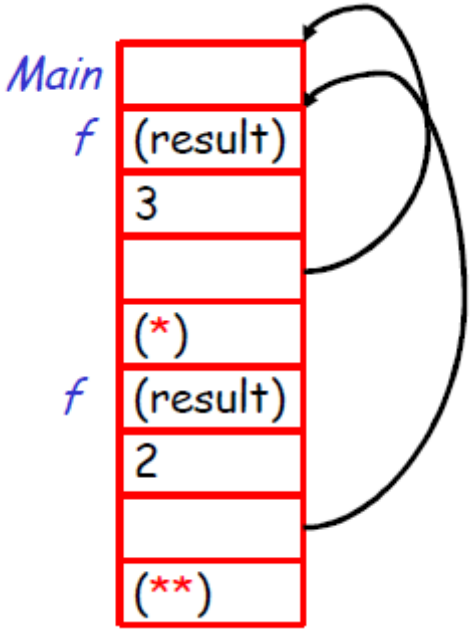
- Return value ของ G
- Actual parameter ที่ถูกส่งผ่านมา
- Pointer กลับไปหา AR ก่อนหน้านี้
  - เรียก pointer นี้ว่า control link
  - Link กลับไปหา caller ของ G (ในกรณีนี้คือ F)
- สถานะของโปรแกรมและ CPU ก่อนหน้าที่จะเรียก G
  - สถานะของ registers และ program counter

ตัวอย่าง AR

```
int g() {return 1;}
int f(int x) {if (x == 0) return g() else return f(x-1)
(**);}
int main() { f(3)(*); return 0; }
```



Stack หลังจากเรียก f สองครั้ง



สิ่งที่ได้เรียนรู้จากตัวอย่าง

- AR ของ main ไม่ได้นำค่า return ไปใช้งานต่อ
- จุด (**) และ (*) เป็นจุดที่เป็น return address หลังการทำงานของ f
  - แตกต่างออกไปเมื่อเรียกใช้งานคนละที่ คนละเวลา
- อาจมี AR แบบอื่นๆได้อีก
- คอมไพเลอร์ทำหน้าที่
  - กำหนด layout ของ AR
  - ผลิตโค๊ดเพื่อใช้ข้อมูลใน AR อย่างถูกต้อง
- **การออกแบบ AR และการผลิตโค๊ดจะต้องทำไปในทิศทางเดียวกัน**

Global และ heap

- Reference ไปหา name ที่เป็น global ชี้ไปที่วัตถุเดียว
- Global จะถูกกำหนดให้อยู่ที่ address ที่ตายตัว
  - ต้องจองหน่วยความจำให้แบบ static (statically allocated)
- ข้อมูลใน heap ถูกให้กำเนิดขณะโปรแกรมทำงาน
  - มี lifetime มากกว่า procedure
  - เป็นข้อมูลที่ต้องจองหน่วยความจำแบบ dynamic

เพิ่ม global และ heap ในพื้นที่หน่วยความจำ