Concurrent Programming Lecture Notes

Lecture by Prakash Panangaden, 6th April, 2017

In traditional programming, we have a single locus of control and an unambigious notion of *next step.* However in many applications - notably operating systems but also user interfaces, real-time systems and multithreaded programs, we have multiple independent processes executing at the same time.

Parallel programming: you have several CPUs all dedicated to the same task and you want to make your task run as fast as possible. You get to control the behaviours of the processes.

Concurrent Programming: You have many independent processes competing for resources and you have to manage the resources. The concurrency is a conceptual organization of the code.

Many of the problems are common but the overall paradigm is different. Multithreaded programming sits in between. Now Java, C, F# and many other languages provide facilities for multithreaded programming so it is no longer a speciality topic for operating systems alone but a basic paradigm that many programmers need to know.

New Features

1. Different programs are running at the same time; you cannot precdict which instruction will execute next - NONDETERMINISM
2. Programs are competing for resources and one may unfairly get all the resources all the time - FAIRNESS
3. Programs may grab part of their needed resources and refuse to give it up, thus causing the whole system to halt - DEADLOCK

Paradigmatic Example: Critical Section Mutual Exclusion

Two processes, each has a critical section and a non-critical section. A process may run forever or it may halt, but it will never halt in its critical section. It is imperative that if one process is in its critical section the other must wait for it. The code below shows attempts at symmetric solutions. Last year, Amanda Ivey asked a brilliant question: "Why should the solutions be symmetric"? The answer is that allowing asymmetric solutions makes it even harder to guarantee fairness and does not help arrive at easier solutions.

First Attempt

shared variable turn=1 or 2

Pro1

loop

NCS-1
while turn!=1 do skip
CS-1
turn=2

end loop

Proc-2

loop

NCS-2
while turn!=2 do skip
CS-2
turn=1

end loop

- **Good:** Satisfies mutual exclusion , Fair(?), No deadlock.
- **Bad:** **F**orces the two processes to alternate, what if one of them terminates?

Second Attempt

Use 2 variables, C1,C2=0 or 1. Ci=0 means Proc_i wants to enter its critical section. Only Proc_i can set ci but the other one can test it.

Proc_1

loop

NCS-1
while C2=0 do skip
C1=0
CS-1
C1=1

end loop

Proc_2

loop

NCS-2
while C1=0 do skip
C2=0
CS-2

C2=1
end loop

Problem: Can violate mutex! 1. Proc_1 checks and finds C2=1 2. Proc_2 checks C1 and finds C1=1
3. Proc_1 sets C1=0 4.Proc_2 sets C2=0 5.Proc_2 enters CS 6. Proc_1 enters CS

Third Attempt

Idea: Set before test

Proc_1
loop
NCS-1
C1=0
while C2!=1 do skip
CS-1
C1=1
end loop

Proc_2
loop
NCS-2
C2=0
while C1!=1 do Skip
CS-2
C2=1
end loop

- **Good:** Satisfies mutual exclusion
- **Bad:** Can easily deadlock

Fourth Attempt

Back off if you cannot make progrees

Proc_1
loop
NCS-1
C1 = 0
while C2!=1 do
{C1=1;C1=0;}
CS-1
C1=1

end loop
Proc_2
loop
NCS-2
C2 = 0
while C1!=1 do
{C2=1;C2=0;}
CS-2
C2=1
end loop

- **Good:** Satisfies mutual exclusion and is dead lock free.
- **Bad:** A process can be starved. So this solution is unfair. The system can also **livelock:**

Proc_1 sets C1 to 0
Proc_2 sets C1 to 0
Proc_1 checks C2 and stays in loop
Proc_2 checks C1 and stays in loop
Proc_1 resets C1 to 1
Proc_2 resets C2 to 1
Proc_1 sets C1 to 0
Proc_2 sets C2 to 0
Proc_1 checks C2 and stays in loop
Proc_2 checks C1 and stays in loop

Fifth Attempt

Use C1, C2 and turn
Proc_1
loop
NCS-1
C1=0
while (C2!=1 and turn=2) do skip
CS-1
C1=1
turn=2
end loop

Proc_2
loop
NCS-2
C2=0

while(C1!=1and turn=1) do skip
CS-2
C2=1
turn=1
end loop
Cannot have deadlock, because turn will have 1 value if there is contention. If one of the processes stops in its non-critical section then turn is not relevant. Unfortunately this is also wrong!!
turn is set to 2 initially
Proc_1 sets C1 to 0
Proc_1 checks C2 and sees it is 1 so goes to CS
Proc_2 sets C2 to 0
Proc_2 checks C1 and sees it is 0 but turn=2 so it goes to its CS
both Proc_1 and Proc_2 are in the critical section

Sixth Attempt

Proc_1
loop
NCS-1
C1=0
turn=2
while(C2=0 and turn=2) do skip
CS-1
C1=1
end loop

Proc_2
loop
NCS-2
C2=0
turn=1
while(C1=0 and turn=1) do skip
CS-2
C2=1
end loop

- **Good:** No deadlock, fair, no livelock, mutex.
- **Bad:** Very hard to believe that this works!