# Uppgift 1,
# The Unisex Bathroom Problem

Parallella processer G1F

Grupp A4

Adam Näslund a13adana & Victor Karlsson c12vicka

## 1 Inledning

I uppgiften gavs ett grundproblem och två utökade versioner av detta problem. Grundproblemet bestod av en resurs som flera processer av två olika typer försökte få tillgång till där en lösning skulle garantera att processer av olika sort aldrig hade samtidig tillgång till resursen. De två utökade versionerna lade respektive till en gräns på hur många processer som fick ha tillgång till resursen samtidigt och krav på fairness och liveness.

Samtliga tre problem löstes med hjälp av programmeringsspråket SR och lösningarna för de första två problemen modellerades och verifierades i verktyget UPPAAL.

## 2 Problembeskrivning

Grundproblemet består av ett antal processer av två olika typer som vill ha tillgång till en delad resurs. Flera processer får använda resursen samtidigt om de är av samma typ, men processer av olika typ får aldrig använda resursen samtidigt. Processerna ska dessutom aldrig kunna hamna i deadlock.

Detta problem är en variant av det klassiska problemet Readers/Writers och en lösning av detta problem måste garantera att processer av olika typ aldrig har tillgång till resursen samtidigt, samt att processerna aldrig kan hamna i deadlock.

Global invariant för en lösning av detta problem blir därmed
$\forall i,j\ (\mathrm{inCS}(i) \rightarrow \neg \mathrm{inCS}(j) \vee \mathrm{typeOf}(i) = \mathrm{typeOf}(j))$.

I nästa version av problemet ställdes det även krav på att max ett visst antal processer fick ha tillgång till resursen samtidigt, vilket innebär att en lösning på detta andra problem dessutom behöver garantera att antalet processer med tillgång till resursen inte överstiger maxvärdet.

Global invariant för en lösning av detta problem innehåller därmed även
$\mathrm{procType1\_inCS} + \mathrm{procType2\_inCS} \leq \max$.

Till sist gavs ytterligare en version av problemet där krav även ställdes på fairness och liveness. Fairness är ett begrepp som kan tolkas på olika sätt, den tolkning som valdes var att det innebär att ingen typ av process någonsin prioriteras före den andra typen, utan att alla processer ska få tillgång till resursen i den ordning de begär tillgång.

Liveness innebär att varje process som vill ha tillgång till resursen någon gång kommer att få det och en lösning på problemet måste garantera både detta och den tidigare nämnda jämlika behandlingen av processtyper.

# 3 Metodik och lösning

## 3.1 Design

Då det första problemet var mycket likt Readers/Writers-problemet och inte hade krav på vare sig liveness eller fairness skapades en nerskalad version av en känd lösning på detta klassiska problem. I lösningen kommunicerar processerna via globala variabler som lagrar hur många av de båda typerna som använder resursen. De globala variablerna skyddas av ett lås och en process som vill ha tillgång till resursen erhåller låset, kontrollerar att inte någon process av motsatt typ använder resursen, uppdaterar de globala variablerna, släpper låset och använder resursen.

För att säkerställa en korrekt design skapades lösningen först som en modell i UPPAAL. Efter att designens korrekthet hade verifierats implementerades lösningen i SR.

Denna lösning kunde sedan enkelt byggas ut för att lösa det andra problemet, där en gräns på antalet samtida användare av resursen finns, genom att ytterligare en global variabel lades till. Även i detta fall modellerades lösningen först i UPPAAL innan den implementerades i SR.

När det tredje problemet skulle lösas, där fairness och liveness dessutom krävdes, blev det dock uppenbart att den enkla lösning som använts tidigare inte räckte till. Istället behövde en ny lösning skapas från grunden.

Flera olika modeller, inspirerade av kända lösningar på Readers/Writers-problemet, skissades upp och diskuterades. Slutligen valdes en design baserad på en gemensam kö, där processer av båda typer ställer sig när de vill ha tillgång till resursen. I denna modell finns en gemensam funktion som genom att undersöka typen på den process som är näst i kö och de globala variabler som håller reda på hur många processer av varje typ som har tillgång till resursen, bestämmer om processen ska få tillgång till resursen eller om den ska få fortsätta vänta. (Observera att implementationen av tekniska skäl använder flera köer.)

I detta fall skapades ingen modell i UPPAAL utan lösningen implementerades direkt i SR.

## 3.2 Verifiering

Den första lösningen hade som krav att enbart en processtyp fick ha tillgång till resursen åt gången och att den inte skulle kunna hamna i deadlock. För att verifiera detta ställdes förfrågningar i UPPAAL gentemot den modell av programmet som skapats.

Lösningen av det andra problemet skulle, utöver kraven på den föregående lösningen, även garantera att det aldrig fanns fler än ett givet max antal processer som hade tillgång till resursen. Även denna lösning modellerades i UPPAAL och verifierades.

Lösningen på det sista problemet skulle innehålla alla föregående krav och utöver dessa även garantera fairness och liveness. En modell av lösningen i UPPAAL skulle kunna verifiera liveness men inte fairness och en sådan modell skulle blivit mycket komplex och tagit lång tid att skapa. Testkod i SR samt manuell granskning av implementationens resultat användes därför för att testa dess korrekthet.

# 4 Resultat

## 4.1 Implementation

De första två lösningarna är mycket enkla men använder sig av busy-wait för att få tillgång till resursen vilket gör att de använder onödigt mycket systemresurser. De har även båda stor risk för starvation, vilket inte är önskvärt.

Den sista lösningen använder istället semaforer för att vänta, vilket är mycket bättre. Lösningen implementerar en design där en gemensam kö används för de processer som vill ha tillgång till resursen oavsett typ, men då det inte är möjligt att undersöka typen på en process i en semafor utan att ta ut den, används istället en cirkulär buffer för att lagra den logiska kön och två semaforer i vilka processerna väntar på sin tur.

När en process ska gå ur den kritiska sektionen anropar den sedan en funktion (let_in) som använder den logiska kön för att avgöra hur många och vilken sorts processer som ska släppas in genom att deras semafor signaleras.

Ett mindre problem med denna lösning är att under mycket specifika omständigheter kan en process byta köplats med en annan process av samma typ. (Om den första processen pausas efter det att den redigerat den cirkulära buffern men innan den börjat vänta i semaforen, och den andra processen sedan hinner göra båda sakerna innan den första får köra igen.)

## 4.2 Verifiering

Att den första lösningen garanterade den globala invarianten verifierades genom att lösningen modellerades i UPPAAL och nedanstående förfrågningar kördes mot modellen med positivt resultat.

- A[] !((M1.inCS || M2.inCS) && (W1.inCS || W2.inCS))
- A[] !deadlock
- E<> (M1.inCS || M2.inCS || W1.inCS || W2.inCS)

Den första förfrågningen verifierar att de två processtyperna inte har tillgång till resursen samtidigt och den andra att lösningen inte kan hamna i deadlock. Den sista förfrågningen visar att någon process kommer åt resursen någon gång.

Att den andra lösningen garanterade den globala invarianten för det problemet verifierades även det genom att den modellerades i UPPAAL. Utöver de förfrågningar som användes för att verifiera föregående lösning användes även nedanstående för att verifiera att lösningen aldrig gav för många processer tillgång till resursen.

- A[] !((Men_in_Bathroom + Women_in_Bathroom) > Bathroom_limit)

Därmed går det att med tämligen stor säkerhet säga att de två första lösningarna är korrekta, givet att modellerna överensstämmer med implementationerna.

Den sista lösningen verifierades dels genom att den har en funktion som regelbundet kontrollerar dess invariant, samt genom att dess tillstånd under exekvering sparades i en logg som sedan granskades manuellt. I den manuella granskningen ingick att säkerställa både att resursen fördelats rättvist (att fairness fanns) samt att alla processer fått tillgång till den (att liveness fanns). Även om dessa tester alla gav positiva resultat utesluter det dock inte att fel kan förekomma i lösningen.

# 5 Diskussion

Enligt de tester och den verifiering som utförts löser samtliga lösningar sina respektive problem. De första två lösningarna är väldigt enkla men har några allvarliga problem som gör dem olämpliga för verkliga scenarion.

Sista lösningen löser problemet på ett bra sätt då den fördelar resurserna jämnt mellan de olika processtyperna och även försöker öka tillgången till resursen. Lösningen prioriterar dock en rättvis fördelning framför effektivitet, och kan ge scenarion där en process blockerar tillgång till resursen för ett flertal processer av den andra typen.

Då ingen formell verifiering utan enbart testning av den sista lösningen genomförts råder dock viss osäkerhet över huruvida den verkligen garanterar liveness och fairness, samt huruvida den är helt fri från fel.