Tentamen PC: Processen & Concurrency
9 mei 2006

Geef bij al je antwoorden voldoende motivatie: voor alleen getallen, ja/nee antwoorden en slogans krijg je geen punten. Succes!!

Vraag 1 (1 punt)

Figuur 3.8(b) uit Stallings suggereert dat een proces zich in hooguit één event queue tegelijkertijd kan bevinden. Beschrijf een situatie waarin het wenselijk zou zijn dat een proces op meer dan één event tegelijkertijd kan wachten. Hoe zou je in dit geval Figuur 3.8(b) moeten aanpassen?

Vraag 2 (2 punten)

Beschouw het mutual exclusion algoritme dat is afgebeeld in Figuur 1 en dat is bedacht door James Burns. Het algoritme werkt voor `n>0` processen en maakt gebruik van `n` gedeelde binaire variabelen die initieel allemaal op `down` staan. Voor iedere `i` in het interval `[0..n-1]` voert proces `i` de code uit Figuur 1 uit. Bewijs dat het algoritme mutual exclusion garandeert. Hint: maak gebruik van een bewijs uit het ongerijmde. Neem aan dat er een executie bestaat waarin zowel proces `i` als proces `j` de kritieke sectie binnenkomen. Zonder verlies van algemeenheid mag je aannemen dat proces `i` als eerste zijn vlag op `up` zet. Leidt nu een tegenspraak af.

Vraag 3 (2 punten)

Een kannibalenstam eet een gezamelijk maaltijd uit een grote pot die M porties gestoofde missionaris kan bevatten. Wanneer een kannibaal wil eten pakt hij/zij zelf een maaltijd uit de pot tenzij deze leeg is. Wanneer de pot leeg is maakt de kannibaal de kok wakker en wacht totdat deze de pot opnieuw heeft gevuld.

Vertaald naar informatica-jargon is er een willekeurig aantal kannibaal-threads die de volgende code uitvoeren:

```
while true
   getServingFromPot()
   eat()
```

```
type flagtype = {down, up};
shared var Flag: array [0..n-1] of flagtype initially {down,..,down};

program Process_i;
  local var j:0..n-1;
begin
  while true do begin
    remainder;                                    {*Remainder region*}
    repeat                                      {*Trying region entry*}
      Flag[i] := down;
      repeat until for all j in {0,..,i-1} Flag[j] = down;
      Flag[i] := up
    until for all j in {i+1,..,n-1} Flag[j] = down
    critical                                        {*Critical region*}
    Flag[i] := down;                                    {*Exit region*}
  end
```

Figure 1: Mutual exclusion algoritme van Burns.

en er is één kok-thread die de volgende code uitvoert:

```
while true
   putServingsInPot(M)
```

De synchronisatieconstraints zijn dat een kannibaal `getServingFromPot` niet kan uitvoeren wanneer de pot leeg is, terwijl de kok `putServingsInPot` alleen kan uitvoeren als de pot leeg is. Voeg, gebruikmakend van semaforen, code toe aan de bovenstaande threads zodat voldaan wordt aan de synchronisatieconstraints.

Vraag 4 (1.5 punt)

Beschouw een systeem bestaande uit processen $P_1, \ldots, P_n$, die ieder een unieke prioriteit hebben. Schrijf een monitor die drie identieke printers toewijst aan deze processen, en gebruik maakt van de prioriteiten om de volgorde van toewijzing te bepalen.

Vraag 5 (1 punt)

Beschouw de deadlock die kan ontstaan bij het dinerende filosofen probleem wanneer alle filosofen tegelijk een vork vasthouden. Leg uit waarom de vier condities voor deadlock in dit geval gelden. Bespreek voor iedere conditie

hoe de deadlock voorkomen zou kunnen worden indien deze conditie wordt opgeheven.

Vraag 6 (0.5 punt)

Welk voordeel heeft het om tijds-quanta van verschillende grootte te hebben voor de verschillende niveaus in een "multilevel queuing systeem"?

Vraag 7 (1 punt)

Stel dat een operating system twee soorten processen ondersteunt: interactieve processen met een hoge prioriteit, en niet-interactieve processen met een lage prioriteit. De processen met een hoge prioriteit wisselen periodes ter lengte t_c waarin ze rekenen af met periodes ter lengte t_b waarin ze geblokkeerd zijn en wachten op input. De processen met een lage prioriteit rekenen permanent en zijn nooit geblokkeerd. De scheduling strategie van het operating system is round-robin met prioriteiten en een quantum ter grootte q, waarbij $t_c < q$. Scheduling beslissingen worden alleen genomen wanneer het quantum verbruikt is of wanneer het lopende proces blokkeert. De scheduler selecteert alleen een proces met lage prioriteit indien er geen proces met hoge prioriteit beschikbaar is. Stel dat er 1 proces is met hoge prioriteit en 1 proces met lage prioriteit, en dat beide processen zeer lange tijd lopen. Welke fractie van de tijd draait het proces met lage prioriteit?

Vraag 8 (1 punt)

Beschouw twee processen P_1 en P_2 met $T_1 = 50$, $C_1 = 25$, $T_2 = 75$ en $C_2 = 30$. Kunnen deze twee processen gescheduled worden met behulp van rate monotonic scheduling? Laat zien wat er gebeurt wanneer earliest-deadline-first scheduling wordt gebruikt.