OS1 kolokvijumi

kategorisani zadaci sa kolokvijuma 2006-2023 sa stranice predmeta

pripremili Luka Simić i Aleksa Marković

Mart 2023

Sadržaj

Predgovor **8**

Kompajler/linker **9**
1. zadatak, prvi kolokvijum, mart 2023. . . . 9
1. zadatak, prvi kolokvijum, jun 2022. . . . 10
2. zadatak, drugi kolokvijum, septembar 2016. . . . 10
2. zadatak, drugi kolokvijum, septembar 2014. . . . 11
3. zadatak, prvi kolokvijum, mart 2012. . . . 12
3. zadatak, prvi kolokvijum, april 2011. . . . 13
3. zadatak, prvi kolokvijum, maj 2011. . . . 14
3. zadatak, drugi kolokvijum, maj 2008. . . . 15
3. zadatak, drugi kolokvijum, maj 2006. . . . 16

Stranična organizacija **17**
3. zadatak, prvi kolokvijum, mart 2023. . . . 17
2. zadatak, prvi kolokvijum, mart 2022. . . . 17
3. zadatak, prvi kolokvijum, maj 2022. . . . 18
2. zadatak, prvi kolokvijum, jun 2022. . . . 19
3. zadatak, kolokvijum, jun 2020. . . . 20
2. zadatak, kolokvijum, jul 2020. . . . 21
3. zadatak, drugi kolokvijum, maj 2019. . . . 22
3. zadatak, drugi kolokvijum, jun 2019. . . . 23
3. zadatak, drugi kolokvijum, april 2018. . . . 24
3. zadatak, drugi kolokvijum, jun 2018. . . . 25
3. zadatak, drugi kolokvijum, april 2017. . . . 26
3. zadatak, drugi kolokvijum, jun 2017. . . . 27
3. zadatak, drugi kolokvijum, maj 2016. . . . 28
3. zadatak, drugi kolokvijum, septembar 2016. . . . 29
3. zadatak, drugi kolokvijum, maj 2015. . . . 30
3. zadatak, drugi kolokvijum, septembar 2015. . . . 31
3. zadatak, drugi kolokvijum, april 2014. . . . 32
3. zadatak, drugi kolokvijum, septembar 2014. . . . 33
2. zadatak, prvi kolokvijum, april 2013. . . . 33
4. zadatak, drugi kolokvijum, april 2013. . . . 34
4. zadatak, drugi kolokvijum, septembar 2013. . . . 34
2. zadatak, prvi kolokvijum, mart 2012. . . . 35
4. zadatak, drugi kolokvijum, maj 2012. . . . 35
2. zadatak, prvi kolokvijum, septembar 2012. . . . 36
4. zadatak, drugi kolokvijum, septembar 2012. . . . 37
2. zadatak, prvi kolokvijum, april 2011. . . . 37
2. zadatak, prvi kolokvijum, maj 2011. . . . 38

4. zadatak, drugi kolokvijum, maj 2011. . . . 39
2. zadatak, prvi kolokvijum, april 2010. . . . 40
4. zadatak, drugi kolokvijum, maj 2010. . . . 41
4. zadatak, drugi kolokvijum, maj 2009. . . . 41
3. zadatak, prvi kolokvijum, april 2008. . . . 42
4. zadatak, drugi kolokvijum, maj 2008. . . . 43
5. zadatak, drugi kolokvijum, maj 2008. . . . 43
2. zadatak, prvi kolokvijum, april 2007. . . . 44
4. zadatak, drugi kolokvijum, maj 2007. . . . 44
4. zadatak, prvi kolokvijum, april 2006. . . . 45
5. zadatak, drugi kolokvijum, maj 2006. . . . 45

Kontinualna alokacija **47**
1. zadatak, kolokvijum, avgust 2021. . . . 47
2. zadatak, drugi kolokvijum, jun 2019. . . . 48
2. zadatak, drugi kolokvijum, april 2018. . . . 49
2. zadatak, drugi kolokvijum, jun 2018. . . . 50
2. zadatak, drugi kolokvijum, april 2017. . . . 50
2. zadatak, drugi kolokvijum, maj 2015. . . . 51
2. zadatak, drugi kolokvijum, septembar 2015. . . . 52
2. zadatak, drugi kolokvijum, april 2014. . . . 54
3. zadatak, drugi kolokvijum, april 2013. . . . 55
3. zadatak, drugi kolokvijum, septembar 2012. . . . 55
4. zadatak, drugi kolokvijum, septembar 2011. . . . 56
5. zadatak, drugi kolokvijum, maj 2010. . . . 56
5. zadatak, drugi kolokvijum, maj 2009. . . . 57
5. zadatak, drugi kolokvijum, maj 2007. . . . 57

Segmentna organizacija **58**
2. zadatak, prvi kolokvijum, mart 2023. . . . 58
1. zadatak, prvi kolokvijum, mart 2022. . . . 59
3. zadatak, prvi kolokvijum, mart 2022. . . . 60
1. zadatak, prvi kolokvijum, maj 2022. . . . 60
1. zadatak, kolokvijum, jun 2021. . . . 61
2. zadatak, kolokvijum, oktobar 2020. . . . 62
3. zadatak, kolokvijum, septembar 2020. . . . 63
2. zadatak, prvi kolokvijum, mart 2014. . . . 63
2. zadatak, prvi kolokvijum, april 2014. . . . 64
2. zadatak, prvi kolokvijum, septembar 2011. . . . 64
3. zadatak, prvi kolokvijum, april 2009. . . . 65
4. zadatak, drugi kolokvijum, maj 2006. . . . 65

Segmentno-stranična organizacija **67**
2. zadatak, prvi kolokvijum, maj 2022. . . . 67
3. zadatak, prvi kolokvijum, jun 2022. . . . 67
2. zadatak, kolokvijum, avgust 2020. . . . 69
2. zadatak, prvi kolokvijum, septembar 2014. . . . 70
2. zadatak, prvi kolokvijum, mart 2013. . . . 70
2. zadatak, prvi kolokvijum, septembar 2013. . . . 70
2. zadatak, prvi kolokvijum, maj 2012. . . . 71
3. zadatak, prvi kolokvijum, april 2007. . . . 71
3. zadatak, prvi kolokvijum, april 2006. . . . 71

Dinamičko učitavanje **72**
2. zadatak, drugi kolokvijum, maj 2019. . . . 72
3. zadatak, drugi kolokvijum, septembar 2011. . . . 73
3. zadatak, drugi kolokvijum, maj 2009. . . . 74

3. zadatak, drugi kolokvijum, maj 2007. 74

Preklopi **76**
1. zadatak, kolokvijum, jul 2021. 76
2. zadatak, drugi kolokvijum, jun 2017. 76
2. zadatak, drugi kolokvijum, maj 2016. 78
3. zadatak, drugi kolokvijum, septembar 2013. 79
3. zadatak, drugi kolokvijum, maj 2012. 81
3. zadatak, drugi kolokvijum, maj 2011. 81
3. zadatak, drugi kolokvijum, maj 2010. 83

Prekidi **85**
2. zadatak, drugi kolokvijum, maj 2022. 85
1. zadatak, kolokvijum, jun 2020. 85
2. zadatak, prvi kolokvijum, mart 2019. 86
2. zadatak, prvi kolokvijum, maj 2019. 88
2. zadatak, prvi kolokvijum, mart 2018. 89
2. zadatak, prvi kolokvijum, april 2018. 90
2. zadatak, prvi kolokvijum, jun 2018. 91
2. zadatak, prvi kolokvijum, mart 2017. 92
2. zadatak, prvi kolokvijum, mart 2016. 93
2. zadatak, prvi kolokvijum, mart 2015. 94
2. zadatak, prvi kolokvijum, maj 2015. 95
3. zadatak, prvi kolokvijum, mart 2014. 97
3. zadatak, prvi kolokvijum, april 2014. 97
3. zadatak, prvi kolokvijum, septembar 2014. 98
3. zadatak, prvi kolokvijum, mart 2013. 99
3. zadatak, prvi kolokvijum, april 2013. 100
3. zadatak, prvi kolokvijum, septembar 2011. 101
4. zadatak, prvi kolokvijum, april 2010. 102

Sistemski pozivi **103**
1. zadatak, drugi kolokvijum, maj 2022. 103
1. zadatak, drugi kolokvijum, jun 2022. 104
1. zadatak, kolokvijum, oktobar 2020. 105
1. zadatak, kolokvijum, jul 2020. 106
4. zadatak, kolokvijum, avgust 2020. 106
2. zadatak, kolokvijum, septembar 2020. 107
3. zadatak, prvi kolokvijum, mart 2019. 108
3. zadatak, prvi kolokvijum, maj 2019. 109
3. zadatak, prvi kolokvijum, jun 2019. 110
3. zadatak, prvi kolokvijum, mart 2018. 111
3. zadatak, prvi kolokvijum, april 2018. 112
3. zadatak, prvi kolokvijum, jun 2018. 112
3. zadatak, prvi kolokvijum, mart 2017. 113
3. zadatak, prvi kolokvijum, mart 2016. 114
3. zadatak, prvi kolokvijum, maj 2016. 115
3. zadatak, prvi kolokvijum, septembar 2015. 116
4. zadatak, prvi kolokvijum, mart 2014. 116
4. zadatak, prvi kolokvijum, mart 2013. 118
4. zadatak, prvi kolokvijum, april 2013. 118
4. zadatak, prvi kolokvijum, septembar 2013. 119
4. zadatak, prvi kolokvijum, mart 2012. 120
4. zadatak, prvi kolokvijum, maj 2012. 121
4. zadatak, prvi kolokvijum, septembar 2012. 122
4. zadatak, prvi kolokvijum, maj 2011. 123
4. zadatak, prvi kolokvijum, septembar 2011. 123

4. zadatak, prvi kolokvijum, april 2009. . . . 124
5. zadatak, prvi kolokvijum, april 2009. . . . 125
4. zadatak, prvi kolokvijum, april 2008. . . . 126
5. zadatak, prvi kolokvijum, april 2007. . . . 127
6. zadatak, prvi kolokvijum, april 2006. . . . 127

Interfejs niti **129**
2. zadatak, kolokvijum, avgust 2021. . . . 129
3. zadatak, prvi kolokvijum, april 2017. . . . 130
3. zadatak, prvi kolokvijum, jun 2017. . . . 130
3. zadatak, prvi kolokvijum, septembar 2016. . . . 131
3. zadatak, prvi kolokvijum, mart 2015. . . . 132
3. zadatak, prvi kolokvijum, maj 2015. . . . 133
4. zadatak, prvi kolokvijum, septembar 2014. . . . 134
3. zadatak, prvi kolokvijum, septembar 2013. . . . 135
2. zadatak, drugi kolokvijum, maj 2012. . . . 135
4. zadatak, prvi kolokvijum, april 2011. . . . 136
5. zadatak, prvi kolokvijum, april 2010. . . . 137

Promena konteksta **140**
1. zadatak, kolokvijum, septembar 2020. . . . 140
2. zadatak, prvi kolokvijum, jun 2019. . . . 141
2. zadatak, prvi kolokvijum, april 2017. . . . 141
2. zadatak, prvi kolokvijum, jun 2017. . . . 142
2. zadatak, prvi kolokvijum, maj 2016. . . . 144
2. zadatak, prvi kolokvijum, septembar 2015. . . . 145
4. zadatak, prvi kolokvijum, april 2014. . . . 146
3. zadatak, prvi kolokvijum, maj 2012. . . . 147
3. zadatak, prvi kolokvijum, septembar 2012. . . . 147
3. zadatak, prvi kolokvijum, april 2010. . . . 148
5. zadatak, prvi kolokvijum, april 2008. . . . 148
4. zadatak, prvi kolokvijum, april 2007. . . . 150
5. zadatak, prvi kolokvijum, april 2006. . . . 152
1. zadatak, drugi kolokvijum, maj 2006. . . . 153

Sinhronizacija procesa (implementacija) **156**
2. zadatak, drugi kolokvijum, jun 2022. . . . 156
2. zadatak, kolokvijum, jun 2021. . . . 156
1. zadatak, kolokvijum, avgust 2020. . . . 157
1. zadatak, drugi kolokvijum, maj 2019. . . . 158
1. zadatak, drugi kolokvijum, april 2017. . . . 160
1. zadatak, drugi kolokvijum, jun 2017. . . . 160
1. zadatak, drugi kolokvijum, maj 2016. . . . 162
2. zadatak, prvi kolokvijum, septembar 2016. . . . 162
1. zadatak, drugi kolokvijum, septembar 2015. . . . 164
1. zadatak, drugi kolokvijum, april 2014. . . . 165
1. zadatak, drugi kolokvijum, septembar 2014. . . . 165
2. zadatak, drugi kolokvijum, april 2013. . . . 166
2. zadatak, drugi kolokvijum, septembar 2013. . . . 166
2. zadatak, drugi kolokvijum, septembar 2012. . . . 167
2. zadatak, drugi kolokvijum, maj 2011. . . . 168
2. zadatak, drugi kolokvijum, septembar 2011. . . . 168
1. zadatak, drugi kolokvijum, maj 2010. . . . 168
1. zadatak, drugi kolokvijum, maj 2009. . . . 169
1. zadatak, drugi kolokvijum, maj 2008. . . . 170
1. zadatak, drugi kolokvijum, maj 2007. . . . 171

Sinhronizacija procesa (interfejs) **173**
3. zadatak, drugi kolokvijum, jun 2022. . . . 173
2. zadatak, kolokvijum, jul 2021. . . . 173
2. zadatak, kolokvijum, jun 2020. . . . 174
1. zadatak, drugi kolokvijum, jun 2019. . . . 175
1. zadatak, drugi kolokvijum, april 2018. . . . 176
1. zadatak, drugi kolokvijum, jun 2018. . . . 177
1. zadatak, treći kolokvijum, jun 2016. . . . 179
1. zadatak, drugi kolokvijum, septembar 2016. . . . 180
1. zadatak, drugi kolokvijum, maj 2015. . . . 180
1. zadatak, drugi kolokvijum, april 2013. . . . 181
1. zadatak, drugi kolokvijum, septembar 2013. . . . 183
1. zadatak, drugi kolokvijum, maj 2012. . . . 183
1. zadatak, drugi kolokvijum, septembar 2012. . . . 184
1. zadatak, drugi kolokvijum, maj 2011. . . . 184
1. zadatak, drugi kolokvijum, septembar 2011. . . . 185
2. zadatak, drugi kolokvijum, maj 2010. . . . 186
2. zadatak, drugi kolokvijum, maj 2009. . . . 187
2. zadatak, drugi kolokvijum, maj 2006. . . . 187

Baferi, proizvođač/potrošač **189**
3. zadatak, drugi kolokvijum, maj 2022. . . . 189
3. zadatak, kolokvijum, jul 2020. . . . 190
1. zadatak, treći kolokvijum, septembar 2013. . . . 191
1. zadatak, treći kolokvijum, septembar 2012. . . . 193
1. zadatak, treći kolokvijum, jun 2011. . . . 194
2. zadatak, drugi kolokvijum, maj 2008. . . . 195
2. zadatak, drugi kolokvijum, maj 2007. . . . 197

Ulaz/izlaz **199**
1. zadatak, treći kolokvijum, jun 2022. . . . 199
3. zadatak, kolokvijum, jul 2021. . . . 200
3. zadatak, kolokvijum, avgust 2021. . . . 201
3. zadatak, kolokvijum, oktobar 2020. . . . 202
3. zadatak, kolokvijum, avgust 2020. . . . 203
1. zadatak, prvi kolokvijum, mart 2019. . . . 204
1. zadatak, prvi kolokvijum, maj 2019. . . . 205
1. zadatak, prvi kolokvijum, jun 2019. . . . 207
1. zadatak, prvi kolokvijum, mart 2018. . . . 208
1. zadatak, prvi kolokvijum, april 2018. . . . 209
1. zadatak, prvi kolokvijum, jun 2018. . . . 210
1. zadatak, prvi kolokvijum, mart 2017. . . . 211
1. zadatak, prvi kolokvijum, april 2017. . . . 212
1. zadatak, prvi kolokvijum, jun 2017. . . . 213
1. zadatak, prvi kolokvijum, mart 2016. . . . 214
1. zadatak, prvi kolokvijum, maj 2016. . . . 216
1. zadatak, prvi kolokvijum, septembar 2016. . . . 217
1. zadatak, prvi kolokvijum, mart 2015. . . . 217
1. zadatak, prvi kolokvijum, maj 2015. . . . 219
1. zadatak, prvi kolokvijum, septembar 2015. . . . 220
1. zadatak, prvi kolokvijum, mart 2014. . . . 221
1. zadatak, prvi kolokvijum, april 2014. . . . 222
1. zadatak, prvi kolokvijum, septembar 2014. . . . 224
1. zadatak, prvi kolokvijum, mart 2013. . . . 225
1. zadatak, prvi kolokvijum, april 2013. . . . 226
1. zadatak, prvi kolokvijum, septembar 2013. . . . 227
1. zadatak, prvi kolokvijum, mart 2012. . . . 228

1. zadatak, prvi kolokvijum, maj 2012. . . . 229
1. zadatak, prvi kolokvijum, septembar 2012. . . . 230
1. zadatak, prvi kolokvijum, april 2011. . . . 231
1. zadatak, prvi kolokvijum, maj 2011. . . . 232
1. zadatak, prvi kolokvijum, septembar 2011. . . . 233
1. zadatak, prvi kolokvijum, april 2010. . . . 234
2. zadatak, prvi kolokvijum, april 2009. . . . 235
2. zadatak, prvi kolokvijum, april 2008. . . . 235
1. zadatak, prvi kolokvijum, april 2007. . . . 236
2. zadatak, prvi kolokvijum, april 2006. . . . 237

Ulaz/izlaz (blokovski uređaji) **238**
3. zadatak, kolokvijum, jun 2021. . . . 238
1. zadatak, treći kolokvijum, jun 2019. . . . 239
1. zadatak, treći kolokvijum, jun 2018. . . . 240
1. zadatak, treći kolokvijum, jun 2017. . . . 240
1. zadatak, treći kolokvijum, septembar 2016. . . . 242
1. zadatak, treći kolokvijum, jun 2015. . . . 243
1. zadatak, treći kolokvijum, septembar 2015. . . . 245
1. zadatak, treći kolokvijum, jun 2014. . . . 245
1. zadatak, treći kolokvijum, septembar 2014. . . . 246
1. zadatak, treći kolokvijum, jun 2013. . . . 246
1. zadatak, treći kolokvijum, jun 2012. . . . 247
1. zadatak, treći kolokvijum, septembar 2011. . . . 248

Komandna linija **251**
2. zadatak, treći kolokvijum, jun 2017. . . . 251
2. zadatak, treći kolokvijum, septembar 2012. . . . 252

Fajl sistem (interfejs) **253**
2. zadatak, treći kolokvijum, jun 2022. . . . 253
4. zadatak, kolokvijum, jun 2021. . . . 254
4. zadatak, kolokvijum, jul 2020. . . . 256
2. zadatak, treći kolokvijum, jun 2019. . . . 257
2. zadatak, treći kolokvijum, jun 2018. . . . 258
2. zadatak, treći kolokvijum, jun 2016. . . . 259
2. zadatak, treći kolokvijum, septembar 2016. . . . 260
2. zadatak, treći kolokvijum, jun 2015. . . . 261
2. zadatak, treći kolokvijum, septembar 2015. . . . 262
2. zadatak, treći kolokvijum, jun 2014. . . . 262
2. zadatak, treći kolokvijum, septembar 2014. . . . 263
2. zadatak, treći kolokvijum, jun 2013. . . . 264
2. zadatak, treći kolokvijum, septembar 2013. . . . 265
2. zadatak, treći kolokvijum, jun 2012. . . . 266
2. zadatak, treći kolokvijum, jun 2011. . . . 266
2. zadatak, treći kolokvijum, septembar 2011. . . . 266

Fajl sistem (implementacija) **268**
3. zadatak, treći kolokvijum, jun 2022. . . . 268
4. zadatak, kolokvijum, jul 2021. . . . 269
4. zadatak, kolokvijum, avgust 2021. . . . 270
4. zadatak, kolokvijum, oktobar 2020. . . . 271
4. zadatak, kolokvijum, jun 2020. . . . 271
4. zadatak, kolokvijum, septembar 2020. . . . 272
3. zadatak, treći kolokvijum, jun 2019. . . . 273
3. zadatak, treći kolokvijum, jun 2018. . . . 274
3. zadatak, treći kolokvijum, jun 2017. . . . 274

3. zadatak, treći kolokvijum, jun 2016. . . . 275
3. zadatak, treći kolokvijum, septembar 2016. . . . 275
3. zadatak, treći kolokvijum, jun 2015. . . . 276
3. zadatak, treći kolokvijum, septembar 2015. . . . 278
3. zadatak, treći kolokvijum, jun 2014. . . . 279
3. zadatak, treći kolokvijum, septembar 2014. . . . 279
3. zadatak, treći kolokvijum, jun 2013. . . . 280
3. zadatak, treći kolokvijum, septembar 2013. . . . 281
3. zadatak, treći kolokvijum, jun 2012. . . . 281
3. zadatak, treći kolokvijum, septembar 2012. . . . 282
3. zadatak, treći kolokvijum, jun 2011. . . . 282
3. zadatak, treći kolokvijum, septembar 2011. . . . 283

Uvod u operativne sisteme **284**
1. zadatak, prvi kolokvijum, april 2009. . . . 284
1. zadatak, prvi kolokvijum, april 2008. . . . 284
1. zadatak, prvi kolokvijum, april 2006. . . . 284

Predgovor

Svrha ove zbirke jeste da objedini sve do sada dostupne kolokvijume iz Operativnih sistema 1 iz nekoliko razloga:

- Na kolokvijumima iz OS1 je dozvoljena literatura, i studenti često pristupaju tome tako što odštampaju kolokvijume i njihova rešenja pa se na kolokvijumu snalaze kroz taj odštampani materijal ako vide neki sličan zadatak. Ovom zbirkom postiže se organizacija takvih materijala radi lakšeg snalaženja na kolokvijumu i ušteda u papirima potrebnim za štampanje svih tih rokova.
- Zadaci su kategorisani po oblastima i sličnosti kako bi se lakše vežbali određeni tipovi zadataka na kolokvijumu.

Poglavlja zbirke idu istim redom kao kolokvijumsko gradivo na predmetu, konkretno:

1. Kompajler/linker, stranična organizacija, kontinualna alokacija, segmentna organizacija, segmentno-stranična organizacija, dinamičko učitavanje, preklopi
2. Prekidi, sistemski pozivi, interfejs niti, promena konteksta, sinhronizacija procesa, baferi, proizvođač/potrošač
3. Ulaz/izlaz, komandna linija, fajl sistem

Raspored ovih oblasti po kolokvijumima se retko menja, ali se takva situacija može desiti (usled predviđenih ili nepredviđenih okolnosti). U tom slučaju, proverite koje oblasti dolaze na kolokvijumu sa predmetnim asistentima. Bitno je napomenuti da se ovaj redosled u prošlosti menjao, pa ako se neki zadatak ranije pojavljivao na prvom kolokvijumu to ne mora da znači da je ta oblast i tekuće godine na prvom kolokvijumu. U suštini, **da li neki zadatak može doći na kolokvijumu određujte prvenstveno na osnovu toga iz koje je oblasti, a ne na osnovu toga na kom se kolokvijumu pojavio u prošlosti**.

Greške u formatiranju i kategorizaciji su sigurno prisutne. Ukoliko ih uočite, možete se javiti jednom od autora ili poslati *pull request* na repozitorijum projekta: https://github.com/KockaAdmiralac/OS-kolokvijumi. Ispravljeni dokumenti će biti dostupni u Releases. Svaka pomoć je dobrodošla.

Srećno na kolokvijumu.

Autori

Kompajler/linker

1. zadatak, prvi kolokvijum, mart 2023.

Dat je sadržaj jednog izvornog fajla sa C kodom.

```
int a[256], n;
int max_a(int n) {
    if (n <= 1) return a[0];
    int m = max_a(n-1);
    return (a[n - 1] > m) ? a[n - 1] : m;
}
```

1. Napisati asemblerski kod za 32-bitni procesor picoRISC, sa sintaksom direktiva pokazanom na predavanjima, kakav bi prevodilac mogao da napravi prevođenjem ovog fajla. Logički segment se na asembleru deklariše direktivom `seg` uz koju ide kvalifikator za tip segmenta (`text`, `bss` ili `data`); npr. `seg text`. Stek raste ka nižim adresama, SP ukazuje na poslednju popunjenu lokaciju, adresibilna jedinica je bajt, a instrukcija poziva potprograma na steku čuva PC i PSW tim redom.
2. Za svaku labelu unutar asemblerskog koda prevoda funkcije `max_a` odrediti pomeraj (razliku) njene vrednosti (adrese u koju se preslikava) u odnosu na vrednost labele koja označava prvu instrukciju ove funkcije. Vrednosti pisati decimalno i obrazložiti rezultat.
3. Ukoliko se logički segmenti nastali prevodom ovog fajla alociraju redom, jedan odmah iza drugog u virtuelnom adresnom prostoru, svaki logički segment poravnat na početak stranice veličine 4 KB, a prvi od njih počinje od virtuelne adrese 0, odrediti virtuelne adrese u koje se preslikavaju svi simboli u asemblerskom prevodu (labele za sve definisane podatke i one koje označavaju instrukcije prevodene funkcije `max_a`). Kratko obrazložiti rezultat.

Rešenje

1.
```
    seg bss
        a dd 256 dup 0
        n dd 0
    endseg
    seg text
        max_a:  load r1, [sp+8]     ; r1 := n
                load r2, #1         ; if (n<=1)
                cmp r1, r2
                jg L0001
                load r0, a          ; return a[0]
                ret
        L0001:  sub r1, r1, r2      ; r1 := n-1
                push r1             ; r0 := max_a(n-1)
                call max_a
                pop r1
                load r2, #2         ; r1:= a[n-1]
                shl r1, r1, r2
                load r1, a[r1]
                cmp r1, r0          ; (a[n-1]>m)?
                jle L0002
                load r0, r1
        L0002:  ret
    endseg
```

2. Instrukcije koje u sebi sadrže zapis neposrednog operanda, pomeraja ili apsolutne adrese zauzimaju dve 32-bitne reči (8 bajtova), ostale zauzimaju po jednu (4 bajta).

```
L0001 = max_a + 4*8 + 2*4 = max_a + 40
L0002 = L0001 + 4*8 + 6*4 = L0001 + 56 = max_a + 96
```

3.
```
    a = 0
```

```
n = a+256*4 = 1024
max_a = 4*1024 = 4096
L0001 = max_a + 40 = 4136
L0002 = max_a + 96 = 4192
```

1. zadatak, prvi kolokvijum, jun 2022.

Posmatraju se četiri odvojena i nezavisna slučaja sadržaja tri fajla sa C kodom, `a.h`, `a.c` i `b.c`, data u tabeli i označena rednim brojem 1 do 4. U svakom od tih slučajeva se sprovodi sledeći postupak:

- najpre se prevodi fajl `a.c`; ako prevodilac prijavi grešku, postupak se prekida;
- ako u prevođenju fajla `a.c` nije bilo grešaka, prevodi se fajl `b.c`; ako prevodilac prijavi grešku, postupak se prekida;
- ako ni u prevođenju fajla b.c nije bilo grešaka, povezuju se fajlovi `a.o` i `b.o`.

Za svaki od datih slučajeva precizno objasniti grešku koju će prijaviti prevodilac, odnosno linker, ukoliko grešaka ima: navesti ko će prijaviti grešku i kakvog tipa je ta greška (i zbog kog simbola). Ukoliko grešaka nema, to naglasiti ("nema grešaka").

#	a.h	a.c	b.c
1	`int f ();`	`#include "a.h"` `int f ();`	`#include "a.h"` `int main () { return f(); }`
2	`extern int x;`	`#include "a.h"` `int x = 0;`	`#include "a.h"` `int main () { return x; }`
3	`int x = 0;`	`#include "a.h"` `int x = 0;`	`extern int x;` `int main () { return x; }`
4	`int x = 0;`	`#include "a.h"` `extern int x;`	`#include "a.h"` `int main () { return x; }`

Rešenje

1. Greška u povezivanju jer simbol `f` nije definisan.
2. Nema grešaka.
3. Greška u prevođenju `a.c` zbog višestruke definicije identifikatora `x`.
4. Greška u povezivanju jer je simbol `x` višestruko definisan (u fajlu `a.o` i `b.o`).

2. zadatak, drugi kolokvijum, septembar 2016.

Potrebno je implementirati proceduru `resolveSymbols` koja se koristi u drugom prolazu jednog linkera. Ova procedura obrađuje jedan ulazni .obj fajl i treba da razreši adresna polja mašinskih instrukcija koja koriste simbole koje taj fajl uvozi. Ulazni i izlazni fajl su memorijski preslikani, tehnikom virtuelne memorije, tako da se njihov sadržaj može jednostavno posmatrati kao sadržaj memorije procesa. Na početak memorijski preslikanog sadržaja ulaznog .obj fajla ukazuje prvi, a na početak tog prepisanog sadržaja u izlaznom (.exe) fajlu ukazuje drugi argument ove procedure; pre poziva ove procedure, linker je već prepisao sadržaj binarnog prevedenog koda (bez zaglavlja) svih ulaznih .obj fajlova u sadržaj izlaznog fajla.

Na samom početku ulaznog fajla nalazi se zaglavlje. Svi pomeraji (*offset*, odnosno relativne adrese) u njemu izraženi su u jedinicama `sizeof(char)==1`, a veličine su `unsigned long` (skraćeno `ulong`). Sadržaj početka zaglavlja je, redom, sledeći:

- jedan `ulong` koji sadrži pomeraj početka binarnog prevedenog koda u .obj fajlu u odnosu na početak celog sadržaja tog fajla (zapravo sadrži veličinu celog zaglavlja iza koga sledi binarni prevedeni kod);
- jedan `ulong` koji sadrži ukupan broj simbola koji se uvoze (n);
- n redom poređanih parova: ime simbola koji se uvozi (niz znakova proizvoljne dužine, završen znakom `'\0'`), iza koga sledi jedan `ulong` koji predstavlja pomeraj prvog nerazrešenog adresnog polja u mašinskoj instrukciji koje treba da sadrži vrednost razrešene adrese tog simbola; takva polja su dalje ulančana u jednostruku listu, tako da svako polje sadrži pomeraj narednog takvog polja za isti simbol, s tim da vrednost pomeraja 0 označava

kraj liste (poslednje takvo polje za taj simbol); ovi pomeraji su relativni u odnosu na početak prevedenog binarnog koda unutar sadržaja .obj fajla.

Linker poseduje tabelu simbola čija operacija:

```
ulong SymbolTable::resolveSymbol(char* symbol);
```

vraća pomeraj (relativnu adresu) u odnosu na početak izlaznog fajla (.exe) u koji se dati simbol prevodi, ukoliko on postoji u tabeli, a 0 ako ga nema. Grešku nedefinisanog simbola treba obraditi pozivom funkcije:

```
int errorSymbolUndefined(char* symbol);
```

Ova funkcija ispisuje korisniku poruku o nedefinisanom datom simbolu i vraća -1, što u tom slučaju treba da vrati i funkcija `resolveSymbols`. U slučaju uspeha, funkcija `resolveSymbols` treba da vrati 0.

```
int resolveSymbols (char* inputObj, char* output);
```

Rešenje

```
typedef unsigned long ulong;
const ulong OffsBinaryStartOffset = 0,
         OffsNumOfImportedSymbols = OffsBinaryStartOffset + sizeof(ulong),
         OffsImportedSymbols = OffsNumOfImportedSymbols + sizeof(ulong);

int resolveSymbols (char* inputObj, char* output) {
  ulong binaryStartOffs = *(ulong*)(inputObj + OffsBinaryStartOffset);
  char* binaryStart = inputObj + binaryStartOffs;
  ulong numOfSymbols = *(ulong*)(inputObj + OffsNumOfImportedSymbols);
  char* symbol = inputObj + OffsImportedSymbols;
  for (ulong i=0; i<numOfSymbols; i++) {
     ulong addr = SymbolTable::resolveSymbol(symbol);
     if (addr==0) return errorSymbolUndefined(symbol);
     ulong symbolLen = strlen(symbol)+1;
     ulong fieldOffs = *(ulong*)(symbol+symbolLen);
     for (; fieldOffs>0; fieldOffs=*(ulong*)(binaryStart+fieldOffs))
       *(ulong*)(output+fieldOffs) = addr;
     symbol = symbol+symbolLen+sizeof(fieldOffs);
  }
  return 0;
}
```

2. zadatak, drugi kolokvijum, septembar 2014.

Kod za prvi prolaz nekog linkera dat je u nastavku. (U prvom prolazu linker samo prikuplja izvezene simbole, a ne proverava i da li su svi uvezeni simboli i definisani; tu proveru radi prilikom obrade uvezenih simbola u drugom prolazu.) Popuniti izostavljene delove koda ozna čene komentarima `/***1***/` do `/***5***/`.

```
void Linker::firstPass () {
  this.status = OK;
  this.binarySize = /***1***/; //BinarySize of processed output
  for (FileReader::reset(); !FileReader::isDone(); FileReader::next()) {
    ObjectFile* objFile = FileReader::currentObjectFile();
    if (objFile==0) {
      Output::error("Fatal internal error: null pointer exception.");
      exit(-1);
    }
    for (objFile.reset(); !objFile.isDone(); objFile.nextSymbol()) {
      Symbol* sym = objFile->getCurrentSymbol();
      if (sym==0) {
        /***2***/
```

```
      }
      if (sym->getKind() == Symbol::export) {
        int offset = sym->getOffset(); // offset in objFile
        offset += /***3***/;
        int status =
          SymbolTable::addSymDef(sym->getName(),offset,objFile->getName());
        if (status==-1) {
          /***4***/
          this.status = ERROR;
        }
      }
    }
    int size = objFile->getBinarySize();
    /***5***/
  }
}
```

Rešenje

```
void Linker::firstPass () {
  this.status = OK;
  this.binarySize = 0;
  for (FileReader::reset(); !FileReader::isDone(); FileReader::next()) {
    ObjectFile* objFile = FileReader::currentObjectFile();
    if (objFile==0) {
      Output::error("Fatal internal error: null pointer exception.");
      exit(-1);
    }
    for (objFile.reset(); !objFile.isDone(); objFile.nextSymbol()) {
      Symbol* sym = objFile->getCurrentSymbol();
      if (sym==0) {
        Output::error("Fatal internal error: null pointer exception.");
        exit(-1);
      }
      if (sym->getKind() == Symbol::export) {
        int offset = sym->getOffset();
        offset += this.binarySize;
        int status =
          SymbolTable::addSymDef(sym->getName(),offset,objFile->getName());
        if (status==-1) {
          Output::errorMsg("Symbol %d already defined.", sym->getName());
          this.status = ERROR;
        }
      }
    }
    int size = objFile->getBinarySize();
    this.binarySize += size;
  }
}
```

3. zadatak, prvi kolokvijum, mart 2012.

Neki C kompajler ne koristi uobičajeni pristup alokaciji lokalnih promenljivih i argumenata funkcija, pa ih ne smešta na kontrolni procesorski stek (onaj na kome procesor implicitno čuva povratnu adresu prilikom skoka u potprogram mašinskom instrukcijom `call`). Umesto toga, ovaj kompajler koristi sledeću tehniku za alokaciju i pristup lokalnim promenljivim i argumentima funkcija.

Za svaku lokalnu promenljivu i argument funkcije definisanu u programu, kompajler statički (u vreme prevođenja) alocira jedan (i samo jedan) deo memorije za smeštanje jedne instance te promenljive/argumenta. Pristup toj promenljivoj ili argumentu u kodu funkcije je onda rešen statičkim vezivanjem, direktnim memorijskim adresiranjem te jedne i uvek iste lokacije koja je poznata u vreme prevođenja. Da bi se podržali rekurzivni pozivi, svakoj takvoj promenljivoj i argumentu pridružena je jedna (i samo jedna) globalna struktura podataka koju kompajler organizuje. Ta struktura podataka implementira LIFO protokol; drugim rečima, svakoj lokalnoj promenljivoj i argumentu pridružen je jedan LIFO stek na kome se čuvaju prethodne vrednosti te promenljive/argumenta u slučaju ugnježdenih i rekurzivnih poziva. Prilikom poziva funkcije, tekuća vrednost iz statičke lokacije za svaku lokalnu promenljivu/argument se smešta na vrh njegovog sopstvenog steka, sa koga se restaurira prilikom izlaska iz pozvane funkcije. Ovo radi kod koga generiše kompajler na odgovarajućim mestima.

1. Na asembleru napisati prevod koji bi napravio ovaj kompajler za sledeću rekurzivnu C funkciju:

```
int f (int n) {
  if (n==0) return 0;
  else return f(n-1)+1;
}
```

   Kod koji generiše kompajler za smeštanje vrednosti lokalne promenljive/argumenta koji je alociran na adresi simbolički označenoj sa `X` predstaviti sledećim makroima (ti makroi biće zamenjeni odgovarajućim kodom koji održava pomenuti stek pridružen promenljivoj `X` i koji ima određeni isti oblik parametrizovan adresom `X`, a koji ovde nije od interesa):

```
push(X)
pop(X)
```

2. Objasniti šta je problem koji se mora prevazići ako se za ovaj kompajler želi napraviti višenitno jezgro poput školskog jezgra i kako se taj problem može prevazići u implementaciji jezgra, a bez izmene kompajlera.

Rešenje

1.

```
f:    load r0,[n] ; if (n==0)
      cmp r0,#0
      jne else
      ret ; r0==0, return 0
else: dec r0 ; f(n-1)
      push(n)
      store [n],r0
      call f
      pop(n)
      inc r0 ; return f(n-1)+1
      ret
```

2. Problem je to što je svakoj lokalnoj promenljivoj i argumentu pridružen jedan i samo jedan globalni i statički alocirani stek. Zbog toga taj stek može da „prati“ samo instance lokalnih promenljivih samo jedne niti, a ne više njih. Na primer, jedna nit bi mogla da pozove funkciju f sa datim argumentom `n` i druga učini to isto i uporedo, pokvarivši i tekuću vrednost i stek starih vrednosti za `n` prve niti. Za potrebe uporednih niti neophodno je imati zaseban skup instanci lokalnih promenljivih i argumenata pridružen svakoj niti. Prema tome, ceo skup statički alociranih lokalnih promenljivih i argumenata, zajedno sa njima pridruženim LIFO strukturama (pojedinačnim stekovima), mora da bude deo konteksta niti, što znači da se mora čuvati i restaurirati iz PCB prilikom promene konteksta niti, na sličan način kako se čuvaju i restauriraju registri procesora, odnosno analogno odvajanju zasebnog kontrolnog steka za svaku nit.

3. zadatak, prvi kolokvijum, april 2011.

U nekom 32-bitnom RISC procerosu svi registri su 32-bitni, adrese su 32-bitne, a adresibilna jedinica je bajt. Prevodilac za jezik C za taj procesor povratnu vrednost funkcije prenosi kroz registar `r0`. Data je implementacija standardne funkcije `setjmp()` za taj procesor i taj prevodilac:

```
int setjmp (jmp_buf buf) {
  asm {
    load r0,-1*4[sp]; // r0:=buf
    store psw,0*4[r0];
    store r1,1*4[r0];
    store r2,2*4[r0];
    ...
    store r31,31*4[r0];
    pop r1;           // pop return address to save the caller's context
    store sp,32*4[r0]; // save sp from the caller's context
    store r1,33*4[r0]; // save pc from the caller's context
    push r1;          // push return address back to the stack
    load r1,1*4[r0];  // restore r1
    clr r0; // r0:=0
  }
}
```

Napisati implementaciju standardne funkcije `longjmp()`:

```
void longjmp (jmp_buf buf, int val);
```

Rešenje

```
void longjmp (jmp_buf buf, int val) {
  asm {

    load r0,-2*4[sp]; // r0:=val
    and r0,r0,r0;     // fix r0 if it is zero
    jnz continue
    load r0,#1

continue:
    load r1,-1*4[sp]; // r1:=buf

    load psw,0*4[r1];
    load r3,3*4[r1];  // restore regs r3..r31
    load r4,4*4[r1];
    ...
    load r31,31*4[r1];

    load sp,32*4[r1]; // restore sp
    load r2,33*4[r1]; // restore pc and push it on the stack
    push r2
    load r2,2*4[r1];  // restore r2
    load r1,1*4[r1];  // restore r1
    ret;              // now return
  }
}
```

3. zadatak, prvi kolokvijum, maj 2011.

U nekom 32-bitnom RISC procesoru svi registri su 32-bitni, adrese su 32-bitne, a adresibilna jedinica je bajt. Prevodilac za jezik C za taj procesor povratnu vrednost funkcije prenosi kroz registar `r0`. Data je jedna nekorektna implementacija standardnih funkcija `setjmp()` i `longjmp()` za taj procesor i taj prevodilac:

```
int setjmp (jmp_buf buf) {
  asm {
    clr r0; // r0:=0
```

```
    push r1;
    load r1,-2*4[sp]; // r1:=buf
    store sp,0*4[r1];
    store psw,1*4[r1];
    store r2,2*4[r1];
    store r3,3*4[r1];
    ...
    store r31,31*4[r1];
    store pc,32*4[r1];
    pop r1
  }
}

void longjmp (jmp_buf buf, int val) {
  asm {
    load r0,-2*4[sp]; // r0:=val
    and r0,r0,r0;    // fix r0 if it is zero
    jnz continue
    load r0,#1
continue:
    load r1,-1*4[sp]; // r1:=buf
    load sp,0*4[r1];
    load psw,1*4[r1];
    load r2,2*4[r1];
    load r3,3*4[r1];
    ...
    load r31,31*4[r1];
    load pc,32*4[r1];
  }
}
```

Posmatrati upotrebu ovih funkcija, na primer kao u operaciji `dispatch()` školskog jezgra i precizno objasniti šta je problem sa ovom implementacijom.

Rešenje

Problem je u tome što se povratak iz funkcije `setjmp()`, nakon skoka iz funkcije `longjmp()`, oslanja na sačuvanu vrednost povratne adrese originalnog pozivaoca funkcije `setjmp()`, kao i na sačuvanu vrednost `r1`. Međutim, nakon povratka iz „običnog“ poziva funkcije `setjmp()`, u kontekstu pozivajuće funkcije može se pozvati neka druga funkcija ili dogoditi prekid, koji će onda prepisati te vrednosti povratne adrese i sačuvanog `r1` koje su ostale iznad vrha steka.

Drugim rečima, prilikom povratka iz „običnog“ poziva funkcije `setjmp()`, povratna adresa na koju se oslanja povratak nakon skoka iz `longjmp()` ostala je iznad vrha steka i može se lako dogoditi da bude „pregažena“ nekom drugom vrednošću. Tada povratak iz `setjmp()` nakon skoka iz `longjmp()` neće biti korektan. Jedan primer je kod funkcije `dispatch()` školskog jezgra, a koji nakon poziva `setjmp()` ima pozive drugih funkcija (klase `Scheduler`).

3. zadatak, drugi kolokvijum, maj 2008.

Objasniti na koji način prevodilac i/ili linker podržava to što je u nekim jezicima dozvoljeno da se u različitim modulima pojave definicije više entiteta sa istim imenom, ali u različitim tzv. prostorima imena (engl. *namespace*). Na primer, u jeziku C++, statički podatak-član sa istim imenom `m` (a koji po definiciji ima eksterno vezivanje, odnosno izvozi se kao simbol iz modula) može sasvim regularno da se definiše u različitim klasama `X` i `Y`.

Rešenje

Iako ovakvi entiteti imaju isto *nekvalifikovano* ime u datom programu, oni imaju različita potpuno *kvalifikovana* imena (engl. *fully qualified name*) koja se sastoje od pune staze imena njihovih okružujućih prostora imena. Na primer, statički podatak-član `m` klase `X` ima potpuno kvalifikovano ime `X::m`, dok istoimeni član klase `Y` ima potpuno

kvalifikovano ime `Y::m`; slično važi za ugnežđene prostore imena (npr. `P::Q::R::S::t`). Da bi se omogućilo definisanje istoimenih simbola u različitim prostorima imena, prevodilac jednostavno kao simbol u `.obj` fajlu definiše potpuno kvalifikovano ime, a ne nekvalifikovano ime, tako da linker vidi to jednoznačno puno ime. Drugim rečima, linker ne poznaje pojam prostora imena niti kvalifikovanog imena, za njega su svi simboli jednostavni nizovi znakova koji moraju biti jednoznačno definisani. Prevodilac obezbeđuje ovu jednoznačnost korišćenjem punih kvalifikovanih imena kao imena simbola koje ostavlja linkeru.

3. zadatak, drugi kolokvijum, maj 2006.

Precizno objasniti zašto je potrebno linkeru navesti da kao svoj proizvod treba da napravi statičku biblioteku (lib), a ne izvršni program (exe)? Precizno objasniti razlike između postupka i rezultata pravljenja ove dve vrste proizvoda povezivanja.

Rešenje

Statička biblioteka sadrži zaglavlje sa spiskom simbola koje izvozi i uvozi, i samo telo biblioteke (kod), drugim rečima, proizvod je istog oblika kao i proizvod prevođenja, dok izvršni fajl sadrži telo (kod) i zaglavlje u kome se nalazi adresa prve instrukcije koja treba da se izvrši. Zbog ove razlike u proizvodima, linkeru je potrebna informacija šta da napravi. Osim toga, u samom postupku razrešavanja simbola, prilikom pravljenja izvršnog fajla, postojanje nedefinisanog a referisanog simbola se nakon prvog prolaza prijavljuje kao greška. U slučaju pravljenja biblioteke, ovakav slučaj je dozvoljen.

Stranična organizacija

3. zadatak, prvi kolokvijum, mart 2023.

Neki sistem sa straničnom organizacijom memorije koristi tehniku kopiranja pri upisu (*copy on write*). Deskriptor stranice u PMT je veličine 32 bita, s tim da najnižih 16 bita sadrži broj okvira, a najviša tri bita koduju prava pristupa *RWX* tim redom (bit *R* je najviši bit).

Za evidenciju svih okvira raspoložive operativne memorije sistem koristi niz `frames`; i-ti element ovog niza odgovara okviru broj i. Vrednost ovog elementa označava broj stranica koje dele isti okvir pre razdvajanja pri upisu; ako je ta vrednost 0, okvir je slobodan; ako je ta vrednost 1, okvir koristi samo jedna stranica jednog procesa; ako je ta vrednost veća od 1, taj okvir dele stranice različitih procesa.

Kada obrađuje hardverski izuzetak zbog pokušaja nedozvoljenog upisa, nakon provere da adresirana stranica pripada segmentu koji je logički dozvoljen za upis i da treba raditi kopiranje pri upisu, jezgro poziva operaciju `copyOnWrite` sa argumentom koji pokazuje na deskriptor stranice u koju je pokušan upis. Ta funkcija tada treba da iskopira sadržaj okvira u nov okvir i da stranicu koja je referencirana preusmeri u taj okvir i dozvoli upis u tu stranicu. Ostale stranice koje su koristile isti okvir, čak i ako je to samo jedna preostala, se ne menjaju. Kada se pokuša upis u poslednju preostalu stranicu koja je delila taj okvir, ova funkcija će samo dozvoliti upis u nju (nema potrebe kopirati je, jer je jedina).

Implementirati funkciju: `int copyOnWrite(uint32* pd)`. Na raspolaganju je sledeće:

- `getFreeFrame()`: vraća broj prvog slobodnog okvira; ukoliko takvog nema, vraća 0;
- `copyFrame(uint16 oldFrame, uint16 newFrame)`: kopira sadržaj iz okvira broj `oldFrame` u okvir broj `newFrame`.

Ako slobodnog okvira nema, funkcija `copyOnWrite` ne treba ništa da menja, samo da vrati status -1. U slučaju uspeha, ona treba da vrati status 0. Tipovi `uint16` i `uint32` predstavljaju neoznačene 16-bitne, odnosno 32-bitne cele brojeve, respektivno.

Rešenje

```
typedef PgDsc uint32;
typedef Frame uint16;
inline void setPgRW(PgDsc* pd) {
    *pd |= ((PgDsc)3) << 30;
}
int copyOnWrite(PgDsc* pd) {
    Frame oldFrame = (Frame)(*pd);
    if (frames[oldFrame] > 1) {
        Frame newFrame = getFreeFrame();
        if (newFrame == 0) return -1;
        frames[newFrame] = 1;
        frames[oldFrame]--;
        *pd = newFrame;
        copyFrame(oldFrame, newFrame);
    }
    setPgRW(pd);
    return 0;
}
```

2. zadatak, prvi kolokvijum, mart 2022.

U nekom sistemu adresibilna jedinica je bajt, virtuelni i fizički adresni prostori su iste velične 4 GB, a stranica je veličine 4 KB. Sistem koristi straničenje u dva nivoa. PMT oba nivoa imaju isti broj ulaza i svaki ulaz zauzima po 32 bita. U jednom ulazu u PMT drugog nivoa najniža tri bita zadaju prava pristupa do stranice (RWX), a biti do njih sadrže broj okvira (ili 0 ako stranica nije alocirana). PMT prvog nivoa sadrži celu početnu adresu PMT-a drugog nivoa (PMT ne mora biti poravnata na stranicu).

Da bi obradio sistemske pozive u kojima se neki parametar zadaje kao pokazivač (virtuelna adresa u adresnom prostoru pozivajućeg procesa), operativni sistem mora da konvertuje datu virtuelnu u fizičku adresu, jer se kod kernela izvršava u režimu bez preslikavanja adresa. Implementirati funkciju koja obavlja ovu konverziju (vraća *null* ako virtuelna adresa ne pripada alociranom delu virtuelnog adresnog prostora); prvi parametar ove funkcije je pokazivač na PMT prvog nivoa pozivajućeg procesa. Pretpostaviti da je `uint32` deklarisan kao neoznačeni 32-bitni celobrojni tip. Napisati sve potrebne deklaracije za tipove tabele prvog (`PMT0`) i drugog (`PMT1`) nivoa.

```
void* v2pAddr(PMT0 pmt, void* vaddr);
```

Rešenje

```
const uint32 offsetw = 12;
const uint32 page1w = 10;
const uint32 PMT0_size = 1024;
const uint32 PMT1_size = 1024;

typedef uint32 PMT1[PMT1_SIZE];
typedef uint32 PMT0[PMT0_SIZE];

void* v2pAddr(PMT0 pmt, void* vaddr) {
    uint32 page = (uint32)vaddr >> offsetw;
    uint32 offset = (uint32)vaddr & ~((uint32)-1 << offsetw);
    uint32 page0 = page >> page1w;
    uint32 page1 = page & ~((uint32)-1 << page1w);
    uint32* pmt1 = (uint32*)pmt[page0];
    if (!pmt1) {
        return nullptr;
    }
    uint32 dsc = pmt1[page1];
    uint32 frame = dsc >> 3;
    if (!frame) {
        return nullptr;
    }
    uint32 paddr = (frame << offsetw) + offset;
    return (void*)paddr;
}
```

3. zadatak, prvi kolokvijum, maj 2022.

Neki sistem koristi straničnu organizaciju memorije sa stranicom veličine 128 KB. Adresibilna jedinica je 16-bitna reč, virtuelni adresni prostor je veličine 8 GB, a fizički adresni prostor je veličine 2 GB. PMT je u jednom nivou. U deskriptoru stranice u PMT najviša dva bita koduju prava pristupa, dok je u najnižim bitima broj okvira; ako stranica nije alocirana, ceo deskriptor ima vrednost 0.

1. Prikazati logičku strukturu virtuelne i fizičke adrese i označiti širinu svih polja.
2. Implementirati funkciju `sharePages` koja se koristi u implementaciji sistemskog poziva kojim se zahteva logičko deljenje niza od `cnt` susednih stranica počev od stranice pg1 procesa čija je PMT data parametrom `pmt1` sa nizom od `cnt` susednih stranica počev od stranice `pg2` procesa čija je PMT data parametrom `pmt2`, pri čemu su pomenute stranice drugog procesa već alocirane u okvire, a pomenute stranice prvog procesa nisu, već treba da se dele sa drugim. Ukoliko drugi proces nije alocirao sve pomenute svoje stranice ili je prvi proces već alocirao neku od svojih pomenutih stranica, ova funkcija treba da vrati grešku (negativnu vrednost). Tip `long` je širine 64 bita, tip `int` 32 bita, a tip `short` 16 bita.

```
typedef unsigned short ushort;
typedef ushort PMT[PMT_SIZE];

int sharePages(PMT pmt1, ushort pg1, PMT pmt2, ushort pg2, ushort cnt);
```

Rešenje

1. VA: Page(16):Offset(16); PA: Frame(14):Offset(16)
2.
```
ushort MAXPAGE = -1;
int sharePages(PMT pmt1, ushort pg1, PMT pmt2, ushort pg2, ushort cnt) {
    if (MAXPAGE-cnt <= pg1 || MAXPAGE-cnt <= pg2) {
        return -1; // Overflow
    }
    for (ushort i = 0; i < cnt; i++) {
        if (pmt1[pg1 + i] != 0 || pmt2[pg2 + i] == 0) {
            return -1;
        }
    }
    for (ushort i = 0; i < cnt; i++) {
        pmt1[pg1 + i] = pmt2[pg2 + i];
    }
    return 0;
}
```

2. zadatak, prvi kolokvijum, jun 2022.

U nekom sistemu jedan objekat klase `FreeFrames`, čiji je interfejs dat dole, vodi evidenciju o slobodnim okvirima memorije predviđene za smeštanje stranica procesa. Objekat ove klase inicijalizuje se zadavanjem kontinualnog memorijskog prostora od `size` susednih stranica počev od adrese `mem`. Veličina stranice je `PAGE_SIZE` (u jedinicama `sizeof(char)`). Operacija `alloc` vraća jedan slobodan okvir i izbacuje ga iz evidencije slobodnih okvira. Operacija `free` dodaje slobodni okvir na koga ukazuje argument u evidenciju slobodnih okvira.

Implementirati klasu `FreeFrames` u potpunosti tako da operacije `alloc` i `free` budu najjefikasnije moguće ($\mathcal{O}(1)$).

```
class FreeFrames {
public:
    FreeFrames(void* mem, size_t size);
    void* alloc();
    void free(void* frame);
};
```

Rešenje

```
class FreeFrames {
public:
    FreeFrames(void* mem, size_t size);
    void* alloc();
    void free(void* frame);
private:
    char* head;
};

void FreeFrames::FreeFrames (void* mem, size_t size) {
    char* frame = head = (char*)mem;
    for (int i = 0; i < size - 1; i++) {
        frame = (*(char**)frame) = frame + PAGE_SIZE;
    }
    *(char**)frame = 0;
}

inline void* FreeFrames::alloc () {
    void* ret = head;
    if (head) {
```

```
        head = *(char**)head;
    }
    return ret;
}

inline void FreeFrames::free (void* frame) {
    *(char**)frame = head;
    head = (char*)frame;
}
```

3. zadatak, kolokvijum, jun 2020.

Virtuelni adresni prostor nekog sistema je 8TB (terabajta) i organizovan je stranično, adresibilna jedinica je bajt, a stranica je veličine 4KB. PMT (*page map table*) je organizovana u dva nivoa, s tim da je broj ulaza u PMT prvog nivoa dva puta manji od broja ulaza u PMT drugog nivoa.

Posmatra se jedan proces kreiran nad sledećim programom sa greškom:

```
#define M 4
#define N 0x1000
int src[M][N], dst[M][N];

int main (int argc, const char* argv[]) {
  for (int i=0; i<=M; i++)
    for (int j=0; j<=N; j++)
      dst[i][j] = src[i][j];
}
```

pod sledećim pretpostavkama:

- tip `int` je veličine 32 bita; promenljive `i` i `j` je prevodilac formirao kao automatske promenljive na steku;
- logički segment za kod ovog programa veličine je jedne stranice i nalazi se na dnu virtuelnog adresnog prostora procesa (na najnižim adresama), a segment za stek je veličine 32 stranice i nalazi se odmah iznad segmenta za kod;
- nizovi `src` i `dst` smešteni su jedan odmah iza drugog, tim redom, u isti logički segment memorije alociran za statičke podatke koji se nalazi iznad segmenta za stek; ovaj segment je onoliki koliko je najmanje stranica potrebno za smeštanje ovih nizova i dozvoljen je za upis;
- operativni sistem ne učitava nijednu stranicu pri kreiranju procesa, već sve stranice učitava tek na zahtev (*demand paging*).

1. Prikazati logičku strukturu virtuelne adrese i označiti veličinu svakog polja.
2. Za koje vredosti promenljivih `i` i `j` i pristup do kog od dva niza `src` i `dst` će procesor prvi put generisati izuzetak koji će operativni sistem smatrati kao prestup (grešku) procesa? Obrazložiti.
3. Koliko straničnih grešaka (*page fault*) će operativni sistem obraditi uspešno za ovaj proces dok ga ne ugasi zbog prestupa, ako je operativni sistem za ovaj proces odvojio dovoljno okvira operativne memorije da smesti sve stranice koje taj proces regularno adresira? Obrazložiti.

Rešenje

1. VA(43): Page1(15):Page2(16):Offset(12).
2. Kod pogrešnog pristupa elementu iza kraja svakog reda osim poslednjeg (tj. za sve `i<M-1` i `j=N`), proces pristupa prvom elementu sledećeg reda istog niza, tako da ti pristupi ne predstavaljaju prestup. Kod prekoračenja poslednjeg reda niza src (`i=M-1` i `j=N`), proces zapravo pristupa prvom elementu prvog reda niza `dst`, tako da ni to nije prestup. Međutim, kod upisa tog elementa u niz `dst`, proces će izazvati straničnu grešku (page fault) koju će operativni sistem tumačiti kao prestup, jer adresira stranicu koja ne pripada nekom alociranom segmentu. Ovo se dešava za `i=3` i `j=N`.
3. Ovaj proces tokom izvršavanja regularno adresira sledeće stranice:

- jednu stranicu segmenta za kod, za dohvatanje instrukcija programa

- samo jednu stranicu segmenta za stek, jer je to dovoljno za samo jedan poziv potprograma `main` sa argumentima ove funkcije i njene dve automatske celobrojne promenljive na steku
- po 16 celih stranica za svaki niz `src` i `dst`, jer svaki niz sadrži $4 \cdot 0x1000 = 4 \cdot 2^{12}$ elemenata po 4 bajta, odnosno 2^{16} bajtova, što je po 16 stranica po 2^{12} bajtova.

Sve ukupno, proces regularno adresira 34 stranice. Kako svaka od njih ostaje u memoriji nakon prvog adresiranja i eventaulnog učitavanja, i pošto je procesu dodeljeno dovoljno okvira, ovaj proces će generisati isto toliki broj (34) straničnih grešaka koje će biti uspešno obrađene.

2. zadatak, kolokvijum, jul 2020.

Neki sistem koristi straničnu organizaciju memorije. Logičke segmente koje je proces alocirao sistem opisuje strukturama tipa `SegDesc` u dvostruko ulančanoj listi za svaki proces. Pritom se za svaki proces pri kreiranju uvek alociraju najmanje dva logička segmenta koji pokrivaju najniže i najviše adrese virtuelnog adresnog prostora procesa (preslikavaju se u prostor koji koristi kernel). Logički segment je uvek poravnat i zaokružen na stranice. U strukturi `SegDesc` polja `next` i `prev` služe za ulančavanje u listu, polje `pg` sadrži broj prve stranice logičkog segmenta, a polje `sz` sadrži veličinu segmenta izraženu u broju stranica. Tip `size_t` je neoznačen celobrojni tip dovoljno velik da predstavi veličinu virtuelnog adresnog prostora. Konstanta `PAGE_SZ` predstavlja veličinu stranice u bajtovima (adresibilnim jedinicama), a konstanta `PAGE_OFFS_SZ` veličinu polja u virtuelnoj adresi za broj bajta unutar stranice; obe ove konstante su tipa `size_t`. Data je i pomoćna funkcija `insert_seg_desc` koja alocira jednu strukturu `SegDesc` i ulančava je u listu kao što je pokazano.

```
struct SegDesc {
  SegDesc *prev, *next;
  size_t pg, sz;
  ...
};

int insert_seg_desc (SegDesc* sd, size_t pg, size_t sz) {
  SegDesc* nsd = alloc_seg_desc();
  if (!nsd) return -1;  // Error: cannot allocate SegDesc
  nsd->pg = pg; nsd->sz = sz;
  nsd->next = sd->next; nsd->prev = sd;
  sd->next->prev = nsd; sd->next = nsd;
  return 0;
}
```

Implementirati internu funkciju kernela `vmalloc` koja treba da alocira logički segment kako je proces tražio. Traženi logički segment je proizvoljne veličine `size` u bajtovima (iako ovaj parametar ne mora biti zaokružen, segment se uvek alocira kao ceo broj stranica). Ukoliko je parametar `addr` različit od *null*, pokušava se alokacija počev od stranice kojoj pripada ova adresa; ukoliko tu nije moguće alocirati segment tražene veličine, funkcija odbija alokaciju i treba da vrati grešku (vrednost *null*). Ukoliko je parametar `addr` jednak *null*, sistem treba da pokuša alokaciju segmenta bilo gde gde je to moguće; ukoliko nije moguće, treba da vrati *null*. U slučaju uspeha, u oba slučaja ova funkcija treba da vrati adresu početka segmenta (poravnatu na stranicu). Prvi parametar je glava liste struktura `SegDesc` procesa koji je tražio ovu uslugu (sigurno različito od *null*).

```
void* vmalloc (SegDesc* head, void* addr, size_t size);
```

Rešenje

```
void* vmalloc (SegDesc* head, void* addr, size_t size) {
  size_t sz = (size+PAGE_SIZE-1)>>PAGE_OFFS_SZ;
  SegDesc* sg = head;
  if (!addr) {
    for (; sg && sg->next; sg=sg->next) {
      size_t pg = sg->pg+sg->sz;
      if (sg->next->pg - pg >= sz) {
        if (insert_seg_desc(sg,pg,sz)<0) return 0;
        return (void*)(pg<<PAGE_OFFS_SZ);
```

```
      }
    }
    return 0;
  }
  else
  {
    size_t pg = addr>>PAGE_OFFS_SZ;
    for (; sg && sg->next; sg=sg->next) {
      if (sg->pg+sg->sz<=pg && pg+sz<=sg->next->pg) {
        if (insert_seg_desc(sg,pg,sz)<0) return 0;
        return (void*)(pg<<PAGE_OFFS_SZ);
      }
    }
    return 0;
  }
}
```

3. zadatak, drugi kolokvijum, maj 2019.

Virtuelni adresni prostor nekog računara je 4GB i organizovan je stranično, adresibilna jedinica je bajt, a PMT je organizovana u dva nivoa. Broj ulaza u PMT oba nivoa je isti, kao i širina svakog ulaza koja je po 32 bita. PMT svakog nivoa zauzima tačno jednu celu stranicu.

Neki operativni sistem na ovom računaru zauzima najnižih 4MB raspoložive fizičke memorije za potrebe kernela. Kako ne bi menjao memorijski kontekst prilikom obrade sistemskih poziva, sistem preslikava (mapira) najviše stranice svakog kreiranog procesa u ovaj memorijski prostor kernela, s tim što postavlja indikatore u deskriptoru tih stranica u PMT tako da one nisu dostupne u neprivilegovanom (korisničkom) režimu rada procesora.

Kernel koristi tehniku *copy-on-write*. U implementaciji sistemskog poziva *fork*, sve PMT procesa roditelja i deteta, uključujući i PMT prvog nivoa, se ne kopiraju, nego se dele, sve dok je to moguće, odnosno dok kernel ne mora da ih razdvoji (kopira). Osim toga, pošto je deo memorije koji pripada kernelu deljen u virtuelnim adresnim prostorima svih procesa na istom mestu, one PMT koje se odnose na taj deo memorije svi procesi dele sve vreme.

1. Kolika je veličina stranice i koliko ulaza ima PMT prvog nivoa? Prikazati postupak kojim se došlo do odgovora.
2. Neki proces izvršio je sistemski poziv *exec* koji inicijalizuje memorijsku mapu tog procesa tako da, osim dela koji se preslikava u memoriju kernela, alocira jedan segment za kod veličine 1MB na početku svog virtuelnog adresnog prostora, jedan segment za podatke veličine 3MB odmah nakon segmenta koda, i segment za stek veličine 4MB odmah ispod memorijskog dela koji pripada kernelu. Koliko operativne memorije (osim one koja je prethodno već bila zauzeta zbog postojanja drugih procesa) je alocirano prilikom izvršavanja ovog sistemskog poziva za potrebe smeštanja PMT ovog procesa? Obrazložiti.
3. Ovaj proces sada izvršava sistemski poziv *fork*. Koliko dodatne memorije (osim one koja je već bila zauzeta zbog postojanja ovog roditeljskog i drugih procesa) je dodatno alocirano za potrebe smeštanja PMT novog procesa deteta prilikom izvršavanja ovog sistemskog poziva? Obrazložiti.
4. Posmatrani proces (roditelj) je potom odmah izvršio instrukciju poziva potprograma, što je uzrokovalo upisom u samo jednu stranicu na vrhu steka. Koliko dodatne memorije (osim one koja je već bila zauzeta pre toga) je dodatno alocirano za potrebe smeštanja PMT oba procesa, i roditelja i deteta zbog izvršavanja ove instrukcije? Obrazložiti.

Rešenje

1. Neka n označava broj bita za adresiranje ulaza u PMT svakog nivoa (broj ulaza u PMT svakog nivoa je 2^n), a neka je veličina stranice 2^p bajtova. Na osnovu uslova zadatka važi: $n + n + p = 32$, jer je takva struktura 32-bitne virtuelne adrese; $2^n \cdot 2^2 = 2^p$, odnosno $n + 2 = p$, jer je veličina jedne PMT jednaka veličini stranice. Rešavanjem se dobija $n = 10, p = 12$, pa jedna PMT ima $2^{10} = 1024 = 1K$ ulaza, a stranica je veličine 4KB, koliko zauzima i jedna PMT prvog ili drugog nivoa.

2. Prema tome, jedna PMT drugog nivoa „pokriva“ $1K \cdot 4KB = 4MB$ virtuelnog adresnog prostora. Zbog toga, za memorijski prostor kernela, procesi dele tačno jednu PMT drugog nivoa na koju ukazuje poslednji ulaz u PMT prvog nivoa i ta PMT drugog nivoa se ne alocira za svaki proces posebno.

PMT prvog nivoa ovog procesa, koja je svakako morala biti alocirana, zauzima 4KB. Za potrebe segmenata koda i podataka ovog procesa potrebna je i dovoljna samo jedna PMT drugog nivoa (1MB + 3MB = 4MB), a za segment steka još jedna. Tako je za PMT prvog i drugog nivoa ukupno alocirano tri stranice, odnosno 12KB memorije.

3. Pošto su i PMT prvog i PMT drugog nivoa procesa roditelja i procesa deteta inicijalno potpuno iste, one se mogu u potpunosti deliti odmah nakon sistemskog poziva *fork* (ovi procesi imaju inicijalno isti i PMTP), i to sve dok neki od procesa ne izvrši upis u neku svoju stranicu. Prema tome, nije potrebno alocirati nikakav dodatni memorijski prostor za PMT procesa deteta.

4. Kada jedan proces izvrši upis u jednu svoju stranicu, potrebno je „razdvojiti“, odnosno kopirati jednu PMT drugog nivoa procesa roditelja i procesa deteta. Osim toga, i odgovarajući ulazi u PMT prvog nivoa koji ukazuju na tu PMT drugog nivoa tako postaju različiti, jer moraju da ukazuju na različite PMT drugog nivoa, pa je potrebno „razdvojiti“, odnosno kopirati i PMT prvog nivoa. Zato je alocirano dve dodatne stranice, za smeštanje ove dve dodatne PMT, odnosno 8KB memorije.

3. zadatak, drugi kolokvijum, jun 2019.

Virtuelni adresni prostor nekog sistema je 4GB i organizovan je stranično, adresibilna jedinica je bajt, a stranica je veličine 8KB. PMT (*page map table*) je organizovana u dva nivoa, s tim da je broj ulaza u PMT prvog nivoa dva puta manji od broja ulaza u PMT drugog nivoa. Posmatra se jedan proces kreiran nad sledećim programom:

```
#define N 0x1000
int src[N], dst[N];

int main () {
  for (int i=0; i<N; i++) dst[i] = src[i];
}
```

pod sledećim pretpostavkama:

- tip `int` je veličine 32 bita; promenljivu `i` je prevodilac formirao kao registarsku promenljivu (njena vrednost se ne čuva u operativnoj memoriji);
- segment za kod ovog programa veličine je jedne stranice, a segment za stek je veličine 32 stranice;
- nizovi `src` i `dst` smešteni su jedan odmah iza drugog u segment memorije alociran za statičke podatke; ovaj segment je onoliki koliko je najmanje stranica potrebno za smeštanje ovih nizova;
- operativni sistem ne učitava nijednu stranicu pri kreiranju procesa, već sve stranice učitava tek na zahtev (*demand paging*).

1. Prikazati logičku strukturu virtuelne adrese i označiti veličinu svakog polja.
2. Koliko straničnih grešaka (*page fault*) će izazvati izvršavanje ovog procesa ako je operativni sistem za ovaj proces odvojio dovoljno okvira operativne memorije da smesti sve stranice koje taj proces adresira? Obrazložiti.
3. Ako je operativni sistem za ovaj proces odvojio samo 4 okvira operativne memorije, a za zamenu (izbacivanje) bira onu stranicu istog tog procesa koja je najdavnije učitana u memoriju, koja stranica će biti prva izbačena iz memorije i kojom stranicom će biti zamenjena (stranice imenovati kao n-te stranice segmenta za kod, stek ili podatke)? Obrazložiti.

Rešenje

1. VA(32): Page1(9):Page2(10):Offset(13).
2. Ovaj proces adresira sledeće stranice:
   - jednu stranicu segmenta za kod, za dohvatanje instrukcija programa
   - samo jednu stranicu iz segmenta za stek, jer je to dovoljno za samo jedan poziv potprograma `main` bez automatskih objekata na steku (argumenata i lokalnih varijabli)
   - po 2 stranice za svaki niz `src` i `dst`, jer svaki niz sadrži $0x1000 = 2^{12}$ elemenata po 4 bajta, odnosno 2^{14} bajtova, što je 2 stranice po 2^{13} bajtova.

   Sve ukupno, proces adresira 6 stranica. Kako svaka od njih ostaje u memoriji nakon prvog adresiranja i učitavanja, i pošto je procesu dodeljeno dovoljno okvira, ovaj proces će generisati isto toliki broj (6) straničnih grešaka.

3. U memoriju će biti najpre učitana stranica segmenta za kod, stranica segmenta za stek, i po jedna stranica segmenta za podatke, tj. prva i treća stranica tog segmenta, u kojima se nalaze prve polovine nizova `src` i `dst`, tim redom. Kada proces bude adresirao prvi element druge polovine niza `src`, za tu stranicu (drugu u segmentu za podatke) neće biti mesta, pa će biti izbačena najdavnije učitana stranica, a to je stranica segmenta za kod.

3. zadatak, drugi kolokvijum, april 2018.

Neki sistem zauzima najnižih n okvira raspoložive fizičke memorije za potrebe kernela. Kako ne bi menjao memorijski kontekst prilikom obrade sistemskih poziva, sistem preslikava (mapira) najviših n stranica svakog kreiranog procesa u ovaj memorijski prostor kernela, s tim da postavlja indikatore u deskriptoru tih stranica u PMT (*page map table*) tako da one nisu dostupne u neprivilegovanom (korisničkom) režimu rada procesora. Ostatak virtuelnog adresnog prostora dostupan je samom procesu za njegove potrebe i u korisničkom režimu. Virtuelni adresni prostor je 16EB (eksabajt, 1E=2^{60}) i organizovan je stranično, adresibilna jedinica je bajt, a stranica je veličine 16KB. Fizički adresni prostor je veličine 1TB (terabajt).

PMT je organizovana u dva nivoa, s tim da su i broj ulaza, kao i širina ulaza u PMT prvog i drugog nivoa isti (PMT oba nivoa su iste veličine). PMT oba nivoa smeštaju se u memoriju uvek poravnate na fizički okvir, odnosno uvek počinju na početku okvira. Zbog toga se u ulazu prvog nivoa čuva samo broj okvira u kom počinje PMT drugog nivoa, dok se preostali biti do celog broja bajtova u ulazu ne koriste; vrednost 0 u svim bitima označava da preslikavanje nije dozvoljeno. U jednom ulazu PMT drugog nivoa čuva se broj okvira u koji se stranica preslikava u najnižim bitima, dok 5 najviših bita ima sledeće značenje (tim redom, od najvišeg ka najnižem): bit koji govori da li je data stranica dostupna u neprivilegovanom (korisničkom) režimu rada procesora, bit koji govori da li je preslikavanje stranice moguće ili ne, i još 3 bita koja koduju prava pristupa (*rwx*), dok se ostali biti ne koriste. Jedan ulaz u PMT prvog i drugog nivoa zauzima minimalan, ali ceo broj bajtova. Implementirati sledeću funkciju:

```
void initPMT (unsigned* pmt, unsigned long pageFrom, unsigned long nPages,
              unsigned long frameFrom, unsigned short rwx);
```

Ovu funkciju poziva kod kernela kada inicijalizuje PMT novokreiranog procesa, na čiju (već alociranu) PMT prvog nivoa ukazuje argument `pmt`, kako bi obavio opisano preslikavanje dela virtuelnog prostora u kernel prostor. Ona treba da preslika `nPages` susednih stranica počev od stranice `pageFrom` u isto toliko susednih okvira počev od okvira `frameFrom`. Argument `rwx` sadrži bite prava pristupa koje treba postaviti u deskriptore svih tih stranica.

Na raspolaganju je interna funkcija kernela `alloc_pmt2` koja alocira jednu PMT drugog nivoa u memoriji, popunjava sve njene ulaze nulama i vraća broj okvira u kom počinje ta alocirana PMT. Pretpostaviti da će ova funkcija uvek uspeti da alocira PMT u memoriji (ignorisati greške). Veličine tipova su sledeće: `int` – 32 bita, `long` – 64 bita, `short` – 16 bita.

Obratiti pažnju na to da se ceo kernel kod izvršava u memorijskom kontekstu nekog procesa, što znači da su sve adrese (vrednosti pokazivača) virtuelne. Za konverziju neke fizičke adrese (vrednosti pokazivača) u kernel prostoru u virtuelnu adresu (u najvišem delu virtuelnog prostora svakog procesa), na raspolaganju je sledeća funkcija:

```
void* kaddrPtoV (void* physicalAddr);
```

Rešenje

VA(64): Page1(25):Page2(25):Offset(14)

PA(40): Frame(26):Offset(14)

```
const unsigned short pg1w = 25, pg2w = 25, offsw = 14;

inline unsigned getPMT1EntryForPage (unsigned long pg) {
  return pg>>pg2w;
}

inline unsigned getPMT2EntryForPage (unsigned long pg) {
  return pg & ~(-1L<<pg2w);
}
```

```
void setPMTEntryForPage (unsigned* pmt1, unsigned long pg, unsigned val) {
  unsigned pmt1entry = getPMT1EntryForPage(pg);
  if (pmt1[pmt1entry]==0)
    pmt1[pmt1entry] = alloc_pmt2();
  unsigned* pmt2 = (unsigned*)(((unsigned long)pmt1[pmt1entry])<<offsw);
  pmt2 = (unsigned*)kaddrPtoV(pmt2);
  unsigned pmt2entry = getPMT2EntryForPage(pg);
  pmt2[pmt2entry] = val;
}

void initPMT (unsigned* pmt, unsigned long pageFrom, unsigned long nPages,
              unsigned long frameFrom, unsigned short rwx) {
  for (unsigned long i=0; i<nPages; i++) {
    unsigned descr = (frameFrom+i) | (((01<<3)|rwx)<<27);
    setPMTEntryForPage(pmt,pageFrom+i,descr);
}
```

3. zadatak, drugi kolokvijum, jun 2018.

Virtuelni adresni prostor nekog sistema je 4GB i organizovan je stranično, adresibilna jedinica je bajt, a stranica je veličine 16KB. Fizički adresni prostor je veličine 16MB. PMT (*page map table*) je organizovana u dva nivoa, s tim da je broj ulaza u PMT prvog i drugog nivoa isti.

Za svaki proces operativni sistem kreira memorijski kontekst kao skup deskriptora logičkih memorijskih segmenata. Deskriptor segmenta predstavljen je strukturom `SegDesc` u kojoj, između ostalog, postoje sledeća polja:

- `unsigned long startingPage`: početna stranica segmenta u virtuelnom prostoru;
- `unsigned long size`: veličina segmenta izražena u broju stranica;
- `SegDesc *next, *prev`: sledeći i prethodni deskriptor segmenta u dvostruko ulančanoj listi segmenata istog procesa; na prvi segment u listi ukazuje polje `segDesc` u PCB procesa;
- `short rwx`: u tri najniža bita definisana su prava pristupa stranicama datog segmenta, odnosno dozvoljene operacije za taj logički segment.

Kada stranica nije u memoriji ili ne pripada alociranom segmentu, u odgovarajućem ulazu u PMT nalazi se vrednost 0 koja hardveru indikuje da preslikavanje nije moguće; procesor tada generiše izuzetak tipa stranične greške (*page fault*). U slučaju povrede prava pristupa stranici, odnosno nedozvoljene operacije, procesor generiše izuzetak koji naziva *operation denied*. Sistem primenjuje *copy on write* tehniku.

1. Prikazati logičku strukturu virtuelne i fizičke adrese i označiti veličinu svakog polja.
2. Implementirati funkciju `resolveVAddrExc`:

```
enum AddrExcKind {pageFault, opDenied};

int resolveVAddrExc (PCB* pcb, unsigned long vaddr, AddrExcKind kind,
                     short rwx, SegDesc** ret);
```

Ovu funkciju poziva kod kernela kada obrađuje neki izuzetak prilikom adresiranja virtuelne adrese za proces na čiji PCB ukazuje prvi argument, prilikom pristupa adresi datoj drugim argumentom. Tip izuzetka daje argument `kind`, a operaciju koja je pokušana argument `rwx`. Ukoliko data virtuelna adresa nije regularna (ne pripada definisanom logičkom segmentu), ili tražena operacija nije dozvoljena za tu adresu, ova funkcija treba da vrati rezultat `MEM_ACCESS_FAULT`, na osnovu koga će kernel ugasiti proces. U suprotnom, radi se o slučaju kada je adresiranje dozvoljeno, ali stranica nije u memoriji i treba je učitati, i kada funkcija treba da vrati `LOAD_PAGE`, ili o slučaju kada treba iskopirati stranicu prilikom upisa, i kada funkcija treba da vrati `COPY_ON_WRITE`. Kad god je virtuelna adresa unutar definisanog logičkog segmenta, u izlazni argument na koga ukazuje argument `ret` treba upisati pokazivač na deskriptor segmenta kom pripada data (regularna) virtuelna adresa.

Rešenje

1. VA(32): Page1(9):Page2(9):Offset(14).

PA(24): Frame(10):Offset(14).

2.

```
const unsigned offsw = 14;
enum AddrExcKind {pageFault, opDenied};

int resolveVAddrExc (PCB* pcb, unsigned long vaddr, AddrExcKind kind,
                     short rwx, SegDesc** ret) {
  if (pcb==0) return ERR_FATAL; // Fatal exception!
  unsigned long page = addr>>offsw;
  for (SegDesc* sd = pcb->segDesc; sd!=0; sd = sd->next) {
    if (page>=sd->startingPage && page<sd->startingPage+sd->size) {
      *ret = sd;
      switch (kind) {
        case pageFault: return LOAD_PAGE;
        case opDenied:
          if ((rwx&2) && (sd->rwx&2))
            return COPY_ON_WRITE;
          else
            return MEM_ACCESS_FAULT;
      }
    }
  }
  *ret = 0;
  return MEM_ACCESS_FAULT;
}
```

3. zadatak, drugi kolokvijum, april 2017.

Virtuelni adresni prostor nekog sistema je 16EB (eksabajt, 1E=2^{60}) i organizovan je stranično, adresibilna jedinica je bajt, a stranica je veličine 16KB. Fizički adresni prostor je veličine 1TB (terabajt). PMT (*page map table*) je organizovana u dva nivoa, s tim da su i broj ulaza, kao i širina ulaza u PMT prvog i drugog nivoa isti (PMT oba nivoa su iste veličine). PMT oba nivoa smeštaju se u memoriju uvek poravnate na fizički okvir, odnosno uvek počinju na početku okvira. Zbog toga se u ulazu prvog nivoa čuva samo broj okvira u kom počinje PMT drugog nivoa, dok se preostali biti do celog broja bajtova u ulazu ne koriste; vrednost 0 u svim bitima označava da preslikavanje nije dozvoljeno. U jednom ulazu PMT drugog nivoa čuva se broj okvira u koji se stranica preslikava i još 2 najniža bita koja koduju prava pristupa (00 – nedozvoljen pristup, stranica nije alocirana, 01 – dozvoljeno samo izvršavanje instrukcije, 10 – dozvoljeno samo čitanje podataka, 11 – dozvoljeno i čitanje i upis podataka), dok se ostali biti ne koriste. Jedan ulaz u PMT prvog i drugog nivoa zauzima minimalan, ali ceo broj bajtova. Sistem ne vrši učitavanje stranica na zahtev niti zamenu stranica, već su sve stranice procesa (odnosno sve one i samo one koje pripadaju logičkim segmentima virtuelne memorije koje je proces alocirao) učitane prilikom njegovog kreiranja i stalno su u memoriji do njegovog gašenja.

1. Prikazati logičku strukturu virtuelne i fizičke adrese i označiti veličinu svakog polja.
2. Implementirati sledeću funkciju:

   ```
   int sharePage (unsigned* pmtFrom, unsigned long pageFrom, unsigned* pmtTo, unsigned long pageTo);
   ```

   Ovu funkciju poziva kod kernela kada obrađuje sistemski poziv kojim jedan proces, na čiju PMT prvog nivoa ukazuje `pmtTo`, zatraži deljenje svoje stranice `pageTo` sa stranicom `pageFrom` drugog procesa, na čiju PMT prvog nivoa ukazuje `pmtFrom`, odnosno želi da svoju stranicu `pageTo` preusmeri na isti okvir koji već koristi stranica `pageFrom`. U slučaju da data stranica nije alocirana u `pmtFrom`, treba vratiti -1 kao signal greške. Pretpostavlja se da proces koji traži ovu uslugu (`pmtTo`) nije već imao alociranu datu stranicu `pageTo` (ona se alocira tek pri ovom pozivu).

Na raspolaganju je interna funkcija kernela `alloc_pmt` koja alocira jednu PMT u memoriji, popunjava sve njene ulaze nulama i vraća broj okvira u kom počinje ta alocirana PMT. Veličine tipova su sledeće: `int` – 32 bita, `long` – 64 bita, `short` – 16 bita.

Rešenje

1. VA(64): Page1(25):Page2(25):Offset(14).

   PA(40): Frame(26):Offset(14).

2.

```
const unsigned short pg1w = 25, pg2w = 25, offsw = 14;

inline unsigned getPMT1Entry (unsigned long vaddr) {
  return vaddr>>(pg2w+offsw);
}

inline unsigned getPMT2Entry (unsigned long vaddr) {
  return (vaddr>>offsw) & ~(-1L<<pg2w);
}

int getPMTEntry (unsigned* pmt1, unsigned long vaddr, unsigned* value) {
  unsigned pmt1entry = getPMT1Entry(vaddr);
  if (pmt1[pmt1entry]==0){
    *value = 0;
    return -1;
  }
  unsigned* pmt2 = (unsigned*)(((unsigned long)pmt1[pmt1entry])<<offsw);
  unsigned pmt2entry = getPMT2Entry(vaddr);
  *value = pmt2[pmt2entry];
  return (*value&3==0)?-1:0;
}

void setPMTEntry (unsigned* pmt1, unsigned long vaddr, unsigned value) {
  unsigned pmt1entry = getPMT1Entry(vaddr);
  if (pmt1[pmt1entry]==0)
    pmt1[pmt1entry] = alloc_pmt();
  unsigned* pmt2 = (unsigned*)(((unsigned long)pmt1[pmt1entry])<<offsw);
  unsigned pmt2entry = getPMT2Entry(vaddr);
  pmt2[pmt2entry] = value;
}

int sharePage (unsigned* pmtFrom, unsigned long pageFrom,
 unsigned* pmtTo, unsigned long pageTo){
  unsigned long vaddrFrom = pageFrom<<offsw;
  unsigned long vaddrTo = pageTo<<offsw;
  unsigned value;
  if (getPMTEntry(pmtFrom,vaddrFrom,&value)<0)
    return -1;
  setPMTEntry(pmtTo,vaddrTo,&value);
  return 0;
}
```

3. zadatak, drugi kolokvijum, jun 2017.

Virtuelni adresni prostor nekog sistema je 4GB i organizovan je stranično, adresibilna jedinica je bajt, a stranica je veličine 4KB. Fizički adresni prostor je veličine 256MB. PMT (*page map table*) je organizovana u dva nivoa, s tim da su i broj ulaza, kao i širina ulaza u PMT prvog i drugog nivoa isti (PMT oba nivoa su iste veličine).

Kada sistem kreira nov proces, ne učitava inicijalno nijednu njegovu stranicu, niti alocira ijednu PMT drugog nivoa, veg samo alocira PMT prvog nivoa, čije sve ulaze inicijalizuje nulama. Stranice se potom dohvataju na zahtev, tokom izvršavanja procesa, kada se po potrebi alociraju i PMT drugog nivoa. Prilikom incijalizacije procesa, sistem

samo kreira memorijski kontekst kao skup deskriptora logičkih memorijskih segmenata definisanih u izvršnom fajlu. Deskriptor segmenta je predstavljen strukturom `SegDesc` u kojoj, između ostalog, postoje sledeća polja:

- `unsigned long startingPage`: početna stranica segmenta u virtuelnom prostoru;
- `unsigned long size`: veličina segmenta izražena u broju stranica;
- `SegDesc *next, *prev`: sledeći i prethodni deskriptor segmenta u dvostruko ulančanoj listi segmenata istog procesa; na prvi segment u listi ukazuje polje `segDesc` u PCB procesa.

Kada stranica nije u memoriji ili ne pripada alociranom segmentu, u odgovarajućem ulazu u PMT nalazi se vrednost 0 koja hardveru indikuje da preslikavanje nije moguće; procesor tada generiše izuzetak tipa stranične greške (*page fault*).

1. Prikazati logičku strukturu virtuelne i fizičke adrese i označiti veličinu svakog polja.
2. Implementirati sledeću funkciju:

   ```
   int resolvePageFault (PCB* pcb, unsigned long addr, SegDesc** ret);
   ```

   Ovu funkciju poziva kod kernela kada obrađuje izuzetak tipa stranične greške za proces na čiji PCB ukazuje prvi argument, prilikom adresiranja adrese date drugim argumentom. Ova funkcija treba da proveri da li je ovo adresiranje dozvoljeno ili nije; ukoliko nije, treba da vrati rezultat `MEM_ACCESS_FAULT`, na osnovu koga će kernel ugasiti proces; u suprotnom, radi se o slučaju kada je adresiranje dozvoljeno, ali stranica nije u memoriji i treba je učitati, pa treba vratiti rezultat `LOAD_PAGE`, a u izlazni argument na koga ukazuje argument `ret` upisati pokazivač na deskriptor segmenta kom pripada data adresa.

Rešenje

1. VA(32): Page1(10):Page2(10):Offset(12).

   PA(28): Frame(16):Offset(12).

2.

```
const unsigned offsw = 12;

int resolvePageFault (PCB* pcb, unsigned long addr, SegDesc** ret) {
  if (pcb==0) return ERR_FATAL; // Exception: fatal exception!
  unsigned long page = addr>>offsw;
  SegDesc* sd = pcb->segDesc;
  while (sd!=0) {
    if (page>=sd->startingPage && page<sd->startingPage+sd->size) {
      *ret = sd;
      return LOAD_PAGE;
    }
    sd = sd->next;
  }
  return MEM_ACCESS_FAULT;
}
```

3. zadatak, drugi kolokvijum, maj 2016.

U nekom sistemu koristi se straničenje sa PMT u jednom nivou i tehnika *copy-on-write*. Tabela deskriptora alociranih memorijskih segmenata jednog procesa ima sledeći sadržaj:

Segment#	Starting page	Size in pages	RWX
1	00h	2h	001
2	A0h	4h	100
3	C1h	3h	110

PMT ovog procesa u posmatranom stanju ima sledeći sadržaj (svi ulazi koji nisu navedeni imaju sadržaj 000 u polju RWX, što hardveru znači da preslikavanje nije dozvoljeno iz bilo kog razloga i da treba generisati *page fault*):

Page#	Frame#	RWX
00h	210h	
01h	211h	
A0h	212h	
A1h	213h	
A2h	214h	
A3h	215h	
C1h	216h	
C2h	217h	
C3h	218h	

1. U prethodnu tabelu upisati vrednosti u kolonu *RWX*, ako su sve stranice učitane.
2. U opisanom stanju ovaj proces kreira proces-dete pozivom `fork()`. Prikazati PMT procesa-roditelja nakon uspešnog završetka ovog poziva.
3. Nakon toga, proces-roditelj izvršava instrukciju koja upisuje vrednost u lokaciju na stranici C2h. Prikazati PMT oba procesa nakon što je operativni sistem obradio izuzetak koji je nastao tokom izvršavanja ove instrukcije. Pretpostaviti da je prvi slobodan okvir 219h.

Rešenje

Page#	Frame#	RWX
00h	210h	001
01h	211h	001
A0h	212h	100
A1h	213h	100
A2h	214h	100
A3h	215h	100
C1h	216h	110
C2h	217h	110
C3h	218h	110

(a) Rešenje pod a

Page#	Frame#	RWX
00h	210h	001
01h	211h	001
A0h	212h	100
A1h	213h	100
A2h	214h	100
A3h	215h	100
C1h	216h	100
C2h	217h	100
C3h	218h	100

(b) PMT procesa-roditelja

Page#	Frame#	RWX
00h	210h	001
01h	211h	001
A0h	212h	100
A1h	213h	100
A2h	214h	100
A3h	215h	100
C1h	216h	100
C2h	219h*	110
C3h	218h	100

(c) PMT procesa-roditelja

Page#	Frame#	RWX
00h	210h	001
01h	211h	001
A0h	212h	100
A1h	213h	100
A2h	214h	100
A3h	215h	100
C1h	216h	100
C2h	217h*	110
C3h	218h	100

(d) PMT procesa-deteta

*) Moguće je i da je OS alocirao nov okvir za proces-dete, pa je korektan odgovor i ako 217h i 219h zamene mesta.

3. zadatak, drugi kolokvijum, septembar 2016.

Virtuelni adresni prostor nekog sistema je 16EB (eksabajt, 1E=2^{60}) i organizovan je stranično, adresibilna jedinica je bajt, a stranica je veličine 64KB. Fizički adresni prostor je veličine 4TB (terabajt). PMT (*page map table*) je organizovana u dva nivoa, s tim da su i broj ulaza, kao i širina ulaza u PMT prvog i drugog nivoa isti (PMT oba nivoa su iste veličine). PMT oba nivoa smeštaju se u memoriju uvek poravnate na fizički okvir, odnosno uvek počinju na početku okvira. Zbog toga se u ulazu prvog nivoa čuva samo broj okvira u kom počinje PMT drugog nivoa, dok se preostali biti do celog broja bajtova u ulazu ne koriste; vrednost 0 u svim bitima označava hardveru da preslikavanje nije dozvoljeno ili moguće. U jednom ulazu PMT drugog nivoa čuva se broj okvira u koji se stranica preslikava i još 2 najniža bita koja koduju prava pristupa (00 – nedozvoljen pristup, stranica nije u memoriji, 01 – dozvoljeno samo izvršavanje instrukcije, 10 – dozvoljeno samo čitanje podataka, 11 – dozvoljeno i čitanje i upis podataka), dok se ostali biti ne koriste. Jedan ulaz u PMT prvog i drugog nivoa zauzima minimalan, ali ceo broj bajtova. Kada sistem kreira nov proces, ne učitava inicijalno ni jednu njegovu stranicu, niti alocira ijednu PMT drugog nivoa, već samo alocira PMT prvog nivoa, čije sve ulaze inicijalizuje nulama. Stranice se potom dohvataju na zahtev, tokom izvršavanja procesa, kada se po potrebi alociraju i PMT drugog nivoa.

1. Prikazati logičku strukturu virtuelne i fizičke adrese i označiti veličinu svakog polja.
2. Implementirati sledeću funkciju:

```
void releasePMTEntry (unsigned* pmt1, unsigned long page);
```

Ovu funkciju poziva kod kernela kada iz memorije izbacuje datu stranicu `page` procesa čija je PMT prvog nivoa zadata prvim argumentom. Ova funkcija treba da ažurira odgovarajuće ulaze u PMT (po potrebi, oba nivoa), s tim što treba i da dealocira PMT drugog nivoa, ako ona više nije potrebna. Na raspolaganju je interna funkcija kernela `dealloc_pmt(unsigned* pmt)` koja dealocira jednu PMT u memoriji. Ignorisati sve eventualne greške.

Rešenje

1. VA(64): Page1(24):Page2(24):Offset(16).

   PA(42): Frame(26):Offset(16).

2.

```
const unsigned short pg1w = 24, pg2w = 24, offsw = 16;
const unsigned pmt1size = 1<<pg1w, pmt2size = 1<<pg2w;

void releasePMTEntry (unsigned* pmt1, unsigned long page) {
  unsigned pmt1entry = page>>(pg2w);
  unsigned* pmt2 = (unsigned*)(((unsigned long)pmt1[pmt1entry])<<offsw);
  unsigned pmt2entry = (page) & ~(-1L<<pg2w);
  pmt2[pmt2entry] = 0;
  for (pmt2entry=0; pmt2entry<pmt2size; pmt2entry++)
    if (pmt2[pmt2entry]!=0) return;
  // PMT2 empty, release it:
  dealloc_pmt(pmt2);
  pmt1[pmt1entry] = 0;
}
```

3. zadatak, drugi kolokvijum, maj 2015.

U nekom sistemu koristi se straničenje sa PMT u jednom nivou i tehnika *copy-on-write*. Tabela deskriptora alociranih memorijskih segmenata jednog procesa ima sledeći sadržaj:

Segment#	Starting page	Size in pages	RWX
1	00h	4h	001
2	12h	3h	100
3	1Ah	2h	110

PMT ovog procesa u posmatranom stanju ima sledeći sadržaj (svi ulazi koji nisu navedeni imaju sadržaj 000 u polju RWX, što hardveru znači da preslikavanje nije dozvoljeno iz bilo kog razloga i da treba generisati *page fault*):

Page#	Frame#	RWX
00h	10h	
01h	11h	
02h	12h	
03h	13h	
12h	14h	
13h	15h	
14h	16h	
1Ah	17h	
1Bh	18h	

1. U prethodnu tabelu upisati vrednosti u kolonu *RWX*, ako su sve stranice učitane.
2. U opisanom stanju ovaj proces kreira proces-dete pozivom `fork()`. Prikazati PMT oba procesa (i roditelja i deteta) nakon uspešnog završetka ovog poziva.
3. Nakon toga, proces-dete izvršava instrukciju koja upisuje vrednost u lokaciju na stranici 1Ah. Prikazati PMT procesa-deteta nakon što je operativni sistem obradio izuzetak koji je nastao tokom izvršavanja ove instrukcije. Pretpostaviti da je prvi slobodan okvir 19h.

Rešenje

Page#	Frame#	RWX
00h	10h	001
01h	11h	001
02h	12h	001
03h	13h	001
12h	14h	100
13h	15h	100
14h	16h	100
1Ah	17h	110
1Bh	18h	110

(e) Rešenje pod a

Page#	Frame#	RWX
00h	10h	001
01h	11h	001
02h	12h	001
03h	13h	001
12h	14h	100
13h	15h	100
14h	16h	100
1Ah	17h	100
1Bh	18h	100

(f) PMT procesa-roditelja

Page#	Frame#	RWX
00h	10h	001
01h	11h	001
02h	12h	001
03h	13h	001
12h	14h	100
13h	15h	100
14h	16h	100
1Ah	17h	100
1Bh	18h	100

(g) PMT procesa-deteta

Page#	Frame#	RWX
00h	10h	001
01h	11h	001
02h	12h	001
03h	13h	001
12h	14h	100
13h	15h	100
14h	16h	100
1Ah	19h	110
1Bh	18h	100

(h) PMT procesa-deteta

3. zadatak, drugi kolokvijum, septembar 2015.

Virtuelni adresni prostor nekog sistema je 1EB (eksabajt, $1E = 2^{60}$) i organizovan je stranično, adresibilna jedinica je bajt, a stranica je veličine 4KB. Fizički adresni prostor je veličine 4TB (terabajt). PMT (*page map table*) je organizovana u tri nivoa, s tim da su i broj ulaza, kao i širina ulaza u PMT sva tri nivoa isti (PMT sva tri nivoa su iste veličine). PMT sva tri nivoa smeštaju se u memoriju uvek poravnate na fizički okvir, odnosno uvek počinju na početku okvira. Zbog toga se u jednom ulazu u PMT prvog i drugog nivoa, u najnižim bitima, čuva samo broj okvira u kom počinje PMT drugog/trećeg nivoa, dok se preostali biti do celog broja bajtova u ulazu ne koriste; vrednost 0 u svim bitima označava da preslikavanje nije dozvoljeno. U jednom ulazu PMT trećeg nivoa, u najnižim bitima čuva se broj okvira u koji se stranica preslikava i još 2 bita sdesna koja koduju prava pristupa (00 – nedozvoljen pristup, stranica nije u memoriji, 01 – dozvoljeno samo izvršavanje instrukcije, 10 – dozvoljeno samo čitanje podataka, 11 – dozvoljeno i čitanje i upis podataka); jedan ulaz u PMT trećeg nivoa zauzima minimalan, ali ceo broj bajtova.

Kada sistem kreira nov proces, ne učitava inicijalno nijednu njegovu stranicu, niti alocira ijednu PMT drugog i trećeg nivoa, već samo alocira PMT prvog nivoa, čije sve ulaze inicijalizuje nulama. Stranice se potom dohvataju na zahtev, tokom izvršavanja procesa, kada se po potrebi alociraju i PMT drugog i trećeg nivoa.

1. Prikazati logičku strukturu virtuelne i fizičke adrese i označiti veličinu svakog polja.
2. U PCB procesa polje `pmt1` tipa `unsigned` sadrži broj okvira od koga počinje PMT prvog nivoa. Na raspolaganju je sledeća funkcija koja okvir sa datim brojem proglašava slobodnim (upisuje ga u evidenciju slobodnih okvira):

```
void deallocateFrame (unsigned frameNumber);
```

Implementirati sledeću funkciju koja se koristi prilikom gašenja procesa ili njegovog izbacivanja iz memorije (*swap out*) i koja oslobađa sve okvire koje proces koristi:

```
void deallocateAllFrames (PCB* pcb);
```

Pretpostavlja se da proces ne deli stranice sa drugim procesima. Mašinska reč, kao i tip `int` su veličine 32 bita, dok su pokazivači, kao i tip `long` veličine 64 bita.

Rešenje

1. VA(60): Page1(16):Page2(16):Page3(16):Offset(12).

   PA(42): Frame(30):Offset(12).

2.

```
const unsigned short pg1w = 16, pg2w = 16, pg3w = 16, offsw = 12;
const unsigned pmt1Size = 1U<<pg1w,
               pmt2Size = 1U<<pg2w, pmt3Size = 1U<<pg3w;
typedef unsigned long ulong;

void deallocateAllFrames (PCB* pcb) {
  if (pcb==0) return;  // Exception
```

```
  // Traverse PMT1:
  unsigned* pmt1 = (unsigned*)((ulong)(pcb->pmt1)<<offsw);
  for (unsigned i1 = 0; i1<pmt1Size; i1++) {
    if (pmt1[i1]==0) continue;

    // Traverse PMT2:
    unsigned* pmt2 = (unsigned*)((ulong)(pmt1[i1])<<offsw);
    for (unsigned i2 = 0; i2<pmt2Size; i2++) {
      if (pmt2[i2]==0) continue;

      // Traverse PMT3:
      unsigned* pmt3 = (unsigned*)((ulong)(pmt2[i2])<<offsw);
      for (unsigned i3 = 0; i3<pmt3Size; i3++) {
        if (pmt3[i3]==0) continue;
        // Deallocate frame:
        unsigned frame = pmt3[i3]>>2;
        deallocateFrame(frame);
      }
    }
  }
}
```

3. zadatak, drugi kolokvijum, april 2014.

Virtuelni adresni prostor nekog sistema je 1EB (eksabajt, 1E=2^{60}) i organizovan je stranično, adresibilna jedinica je bajt, a stranica je veličine 4KB. Fizički adresni prostor je veličine 4TB (terabajt). PMT (*page map table*) je organizovana u tri nivoa, s tim da su i broj ulaza, kao i širina ulaza u PMT sva tri nivoa isti (PMT sva tri nivoa su iste veličine). PMT sva tri nivoa smeštaju se u memoriju uvek poravnate na fizički okvir, odnosno uvek počinju na početku okvira. Zbog toga se u jednom ulazu u PMT prvog/drugog nivoa čuva samo broj okvira u kom počinje PMT drugog/trećeg nivoa, dok se preostali biti do celog broja bajtova u ulazu ne koriste; vrednost 0 u svim bitima označava da preslikavanje nije dozvoljeno. U jednom ulazu PMT trećeg nivoa čuva se broj okvira u koji se stranica preslikava i još 2 bita koja koduju prava pristupa (00 – nedozvoljen pristup, stranica nije u memoriji, 01 – dozvoljeno samo izvršavanje instrukcije, 10 – dozvoljeno samo čitanje podataka, 11 – dozvoljeno i čitanje i upis podataka); jedan ulaz u PMT trećeg nivoa zauzima minimalan, ali ceo broj bajtova. Kada sistem kreira nov proces, ne učitava inicijalno nijednu njegovu stranicu, niti alocira ijednu PMT drugog i trećeg nivoa, već samo alocira PMT prvog nivoa, čije sve ulaze inicijalizuje nulama. Stranice se potom dohvataju na zahtev, tokom izvršavanja procesa, kada se po potrebi alociraju i PMT drugog i trećeg nivoa. Odgovoriti na sledeća pitanja uz detaljna obrazloženja postupka.

1. Prikazati logičku strukturu virtuelne i fizičke adrese i označiti veličinu svakog polja.
2. Koliko memorije minimalno zauzima PMT alocirana za proces prilikom njegovog kreiranja?
3. Koliko ukupno memorije zauzimaju sve PMT alocirane za proces koji je u dosadašnjem toku izvršavanja adresirao najnižih 1GB svog virtuelnog adresnog prostora?

Rešenje

1. VA(60): Page1(16):Page2(16):Page3(16):Offset(12).

   PA(42): Frame(30):Offset(12).

2. Širina PMT3 je 30+2=32 bita. Ista je i širina PMT1 i PMT2. PMT1 ima 2^{16} ulaza širine 32 bita (4B), što je ukupno: 2^{18}B=256KB.

3. Ovaj proces koristio je 2^{30} svojih najnižih adresa, što je $2^{30-12} = 2^{18}$ stranica. Jedna PMT trećeg nivoa pokriva 2^{16} stranica, pa je ovaj proces alocirao PMT prvog nivoa, jednu PMT drugog nivoa i četiri PMT trećeg nivoa. Zato ukupna veličina PMT iznosi $6 \cdot 256$KB=1,5MB.

3. zadatak, drugi kolokvijum, septembar 2014.

Virtuelni adresni prostor nekog sistema je 16EB (eksabajt, 1E=2^{60}) i organizovan je stranično, adresibilna jedinica je bajt, a stranica je veličine 16KB. Fizički adresni prostor je veličine 1TB (terabajt). PMT (*page map table*) je organizovana u dva nivoa, s tim da su i broj ulaza, kao i širina ulaza u PMT prvog i drugog nivoa isti (PMT oba nivoa su iste veličine). PMT oba nivoa smeštaju se u memoriju uvek poravnate na fizički okvir, odnosno uvek počinju na početku okvira. Zbog toga se u ulazu prvog nivoa čuva samo broj okvira u kom počinje PMT drugog nivoa, dok se preostali biti do celog broja bajtova u ulazu ne koriste; vrednost 0 u svim bitima označava da preslikavanje nije dozvoljeno. U jednom ulazu PMT drugog nivoa čuva se broj okvira u koji se stranica preslikava i još 2 najniža bita koja koduju prava pristupa (00 – nedozvoljen pristup, stranica nije u memoriji, 01 – dozvoljeno samo izvršavanje instrukcije, 10 – dozvoljeno samo čitanje podataka, 11 – dozvoljeno i čitanje i upis podataka), dok se ostali biti ne koriste. Jedan ulaz u PMT prvog i drugog nivoa zauzima minimalan, ali ceo broj bajtova. Kada sistem kreira nov proces, ne učitava inicijalno ni jednu njegovu stranicu, niti alocira ijednu PMT drugog nivoa, već samo alocira PMT prvog nivoa, čije sve ulaze inicijalizuje nulama. Stranice se potom dohvataju na zahtev, tokom izvršavanja procesa, kada se po potrebi alociraju i PMT drugog nivoa.

1. Prikazati logičku strukturu virtuelne i fizičke adrese i označiti veličinu svakog polja.
2. Implementirati sledeću funkciju:

```
void setPMTEntry (unsigned* pmt1, unsigned long vaddr, unsigned frame, short r, short w, short x);
```

Ovu funkciju poziva kod kernela kada treba da postavi ulaze u PMT (po potrebi, oba nivoa) kada je alocirao okvir za neku stranicu virtuelne memorije prilikom obrade strani čne greke.

Značenje argumenata je sledeće:

- `pmt1`: pokazivač na PMT prvog nivoa (već alocirana prilikom kreiranja procesa);
- `vaddr`: puna virtuelna adresa koja se obrađuje (pripada stranici koja je učitana);
- `frame`: broj okvira koji je već alociran za datu stranicu;
- `r`, `w`, `x`: prava pristupa koja treba postaviti (0 ili 1; čitanje, upis, izvršavanje, respektivno).

Veličine tipova su sledeće: `int` – 32 bita, `long` – 64 bita, `short` – 16 bita.

Na raspolaganju je interna funkcija kernela `alloc_pmt()` koja alocira jednu PMT u memoriji, popunjava sve njene ulaze nulama i vraća broj okvira u kom počinje ta alocirana PMT. Ignorisati sve eventualne greške.

Rešenje

1. VA(64): Page1(25):Page2(25):Offset(14).

   PA(40): Frame(26):Offset(14).

2.

```
const unsigned short pg1w = 25, pg2w = 25, offsw = 14;

void setPMTEntry (unsigned* pmt1, unsigned long vaddr, unsigned fr,
      short r, short w, short x) {
  unsigned pmt1entry = vaddr>>(pg2w+offsw);
  if (pmt1[pmt1entry]==0)
    pmt1[pmt1entry] = alloc_pmt();
  unsigned* pmt2 = (unsigned*)((unsigned long)pmt1[pmt1entry] << offsw);
  unsigned pmt2entry = (vaddr>>offsw) & ~(-1L<<pg2w);
  pmt2[pmt2entry] = (fr<<2) | ((r||w)<<1) | (x||w);
}
```

2. zadatak, prvi kolokvijum, april 2013.

U nekom sistemu sa straničnom organizacijom memorije virtuelni adresni prostor je veličine 4 GB, adresibilna jedinica je 32-bitna reč, a fizička adresa je veličine 32 bita. Ceo virtuelni adresni prostor ima 64 K stranica.

1. Prikazati logičku strukturu virtuelne i fizičke adrese i navesti širinu svakog polja.
2. Koliko 32-bitnih reči zauzima PMT? Obrazložiti.

Rešenje

1. VA: Page(16):Offset(14); PA: Frame(18):Offset(14).
2. 64K 32-bitnih reči.

   Pošto virtuelni prostor ima 64K = 2^{16} stranica, toliko ulaza ima i PMT. Svaki ulaz je veličine najmanje jedne adresibilne jedinice (32-bitne reči), što je i dovoljno za smeštanje broja okvira i eventualnih dodatnih bita, pa PMT zauzima 64 K reči.

4. zadatak, drugi kolokvijum, april 2013.

Virtuelni adresni prostor nekog sistema je 16EB (eksabajta, 1E=2^{60}) i organizovan je stranično, adresibilna jedinica je bajt, a stranica je veličine 1KB. Fizički adresni prostor je veličine 1TB (terabajt). PMT (*page map table*) je organizovana u tri nivoa, s tim da su i broj ulaza, kao i širina ulaza u PMT sva tri nivoa isti (PMT sva tri nivoa su iste veličine). PMT sva tri nivoa smeštaju se u memoriju uvek poravnate na fizički okvir, odnosno uvek počinju na početku okvira. Zbog toga se u jednom ulazu u PMT prvog/drugog nivoa čuva samo broj okvira u kom počinje PMT drugog/trećeg nivoa, dok se preostali biti do celog broja bajtova u ulazu ne koriste; vrednost 0 u svim bitima označava da preslikavanje nije dozvoljeno. U jednom ulazu PMT trećeg nivoa čuva se broj okvira u koji se stranica preslikava i još 2 bita koja koduju prava pristupa (00 – nedozvoljen pristup, stranica nije u memoriji, 01 – dozvoljeno samo izvršavanje instrukcije, 10 – dozvoljeno samo čitanje podataka, 11 – dozvoljeno i čitanje i upis podataka); jedan ulaz u PMT trećeg nivoa zauzima minimalan, ali ceo broj bajtova.

Kada sistem kreira nov proces, ne učitava inicijalno ni jednu njegovu stranicu, niti alocira ijednu PMT drugog i trećeg nivoa, već samo alocira PMT prvog nivoa, čije sve ulaze inicijalizuje nulama. Stranice se potom dohvataju na zahtev, tokom izvršavanja procesa, kada se po potrebi alociraju i PMT drugog i trećeg nivoa.

Odgovoriti na sledeća pitanja uz detaljna obrazloženja postupka.

1. Prikazati logičku strukturu virtuelne i fizičke adrese i označiti veličinu svakog polja.
2. Koliko memorije minimalno zauzima PMT alocirana za proces prilikom njegovog kreiranja?
3. Koliko ukupno memorije zauzimaju sve PMT alocirane za proces koji je u dosadašnjem toku izvršavanja adresirao najnižih 1GB svog virtuelnog adresnog prostora?

Rešenje

1. VA(64): Page1(18):Page2(18):Page3(18):Offset(10).

   PA(40): Frame(30):Offset(10).
2. Širina PMT3 je 30+2=32 bita. Ista je i širina PMT1 i PMT2. PMT1 ima 2^{18} ulaza širine 32 bita (4B), što je ukupno: 2^{20}B=1MB.
3. Ovaj proces koristio je 2^{30} svojih najnižih adresa, što je $2^{30-10} = 2^{20}$ stranica. Jedna PMT trećeg nivoa pokriva 2^{18} stranica, pa je ovaj proces alocirao PMT prvog nivoa, jednu PMT drugog nivoa i četiri PMT trećeg nivoa. Zato ukupna veličina PMT iznosi $6 \cdot 1$MB=6MB.

4. zadatak, drugi kolokvijum, septembar 2013.

Virtuelni adresni prostor nekog sistema je 16EB (eksabajt, 1E=2^{60}) i organizovan je stranično, adresibilna jedinica je bajt, a stranica je veličine 16KB. Fizički adresni prostor je veličine 1TB (terabajt). PMT (*page map table*) je organizovana u dva nivoa, s tim da su i broj ulaza, kao i širina ulaza u PMT prvog i drugog nivoa isti (PMT oba nivoa su iste veličine). PMT oba nivoa smeštaju se u memoriju uvek poravnate na fizički okvir, odnosno uvek počinju na početku okvira. Zbog toga se u ulazu prvog nivoa čuva samo broj okvira u kom počinje PMT drugog nivoa, dok se preostali biti do celog broja bajtova u ulazu ne koriste; vrednost 0 u svim bitima označava da preslikavanje nije dozvoljeno. U jednom ulazu PMT drugog nivoa čuva se broj okvira u koji se stranica preslikava i još 2 bita koja koduju prava pristupa (00 – nedozvoljen pristup, stranica nije u memoriji, 01 – dozvoljeno samo izvršavanje instrukcije, 10 – dozvoljeno samo čitanje podataka, 11 – dozvoljeno i čitanje i upis podataka), dok se ostali biti ne koriste. Jedan ulaz u PMT prvog i drugog nivoa zauzima minimalan, ali ceo broj bajtova.

Kada sistem kreira nov proces, ne učitava inicijalno ni jednu njegovu stranicu, niti alocira ijednu PMT drugog nivoa, već samo alocira PMT prvog nivoa, čije sve ulaze inicijalizuje nulama. Stranice se potom dohvataju na zahtev, tokom izvršavanja procesa, kada se po potrebi alociraju i PMT drugog nivoa.

Odgovoriti na sledeća pitanja uz detaljna obrazloženja postupka.

1. Prikazati logičku strukturu virtuelne i fizičke adrese i označiti veličinu svakog polja.
2. Koliko memorije minimalno zauzima PMT alocirana za proces prilikom njegovog kreiranja?
3. Koliko ukupno memorije zauzimaju sve PMT alocirane za proces koji je u dosadašnjem toku izvršavanja koristio najnižih 256GB svog virtuelnog adresnog prostora?

Rešenje

1. VA(64): Page1(25):Page2(25):Offset(14).

   PA(40): Frame(26):Offset(14).

2. Širina PMT2 je 32 bita, od koga 26 sadrže broj okvira, 2 bite zaštite, a ostali su neiskorišćeni. Ista je i širina PMT1. PMT1 ima 2^{25} ulaza širine 32 bita (4B), što je ukupno: 2^{27}B=128MB.

3. Ovaj proces koristio je 2^{38} svojih najnižih adresa, što je $2^{38-14} = 2^{24}$ stranica. Jedna PMT drugog nivoa pokriva 2^{25} stranica, pa je ovaj proces alocirao PMT prvog nivoa i jednu PMT drugog nivoa. Zato ukupna veličina PMT iznosi: $2 \cdot 128$MB=256MB.

2. zadatak, prvi kolokvijum, mart 2012.

Neki računar podržava straničnu organizaciju virtuelne memorije, pri čemu je virtuelni adresni prostor veličine 16GB, adresibilna jedinica je 32-bitna reč, a fizički adresni prostor je veličine 1GB. Stranica je veličine 1KB. U deskriptoru stranice u jednom ulazu u tabeli preslikavanja stranica (PMT) najviši bit ukazuje na to da li je stranica u memoriji ili ne, a najniži biti predstavljaju broj okvira u fizičkoj memoriji u koji se stranica preslikava. Deskriptor stranice ne sadrži druge informacije.

1. Prikazati logičku strukturu virtuelne i fizičke adrese i označiti širinu svakog polja. Ako je početak PMT nekog procesa na fizičkoj adresi FF00h, na kojoj adresi je deskriptor stranice ABCh tog procesa?

2. Na jeziku C napisati kod funkcije

   ```
   void setPageDescr(unsigned* pmtp, unsigned page, unsigned frame);
   ```

   koja u tabelu na čiji početak ukazuje dati pokazivač `pmtp`, za stranicu sa datim brojem `page`, upisuje deskriptor tako da se ta stranica preslikava u okvir sa datim brojem `frame`.

Rešenje

1. VA: Page(24):Offset(8);

   PA: Frame(20):Offset(8).

   Nalazi se na adresi 109BCh.

2. ```
   void setPageDescr(unsigned* pmtp, unsigned page, unsigned frame){
     pmtp[page] = frame | ~((unsigned int)~0 / 2);
   }
   ```

4. zadatak, drugi kolokvijum, maj 2012.

Virtuelni adresni prostor nekog sistema je 16EB (eksabajta, $1E = 2^{60}$) i organizovan je straničnо, adresibilna jedinica je bajt, a stranica je veličine 4KB. Fizički adresni prostor je veličine 4TB (terabajta). PMT (*page map table*) je organizovana u dva nivoa, s tim da su i broj ulaza, kao i širina ulaza u PMT prvog i drugog nivoa isti (PMT oba nivoa su iste veličine). PMT oba nivoa smeštaju se u memoriju uvek poravnate na fizički okvir, odnosno uvek počinju na početku okvira. Zbog toga se u ulazu prvog nivoa čuva samo broj okvira u kom počinje PMT drugog nivoa, dok se preostali biti do celog broja bajtova u ulazu ne koriste; vrednost 0 u svim bitima označava da preslikavanje nije dozvoljeno. U jednom ulazu PMT drugog nivoa čuva se broj okvira u koji se stranica preslikava i još 2 bita
   ```

koja koduju prava pristupa (00 – nedozvoljen pristup, stranica nije u memoriji, 01 – dozvoljeno samo izvršavanje instrukcije, 10 – dozvoljeno samo čitanje podataka, 11 – dozvoljeno i čitanje i upis podataka); jedan ulaz u PMT drugog nivoa zauzima minimalan, ali ceo broj bajtova.

Kada sistem kreira nov proces, ne učitava inicijalno ni jednu njegovu stranicu, niti alocira ijednu PMT drugog nivoa, već samo alocira PMT prvog nivoa, čije sve ulaze inicijalizuje nulama. Stranice se potom dohvataju na zahtev, tokom izvršavanja procesa, kada se po potrebi alociraju i PMT drugog nivoa. Odgovoriti na sledeća pitanja uz detaljna obrazloženja postupka.

1. Prikazati logičku strukturu virtuelne i fizičke adrese i označiti veličinu svakog polja.
2. Koliko memorije minimalno zauzima PMT alocirana za proces prilikom njegovog kreiranja?
3. Koliko ukupno memorije zauzimaju sve PMT alocirane za proces koji je u dosadašnjem toku izvršavanja koristio najnižih 512GB svog virtuelnog adresnog prostora?

Rešenje

1. VA(64): Page1(26):Page2(26):Offset(12).

   PA(42): Frame(30):Offset(12).

2. Širina PMT2 je 30+2=32 bita. Ista je i širina PMT1. PMT1 ima 2^{26} ulaza širine 32 bita (4B), što je ukupno: $2^{28}B = 256MB$.

3. Ovaj proces koristio je 2^{39} svojih najnižih adresa, što je $2^{39-12} = 2^{27}$ stranica. Jedna PMT drugog nivoa pokriva 2^{26} stranica, pa je ovaj proces alocirao PMT prvog nivoa i dve PMT drugog nivoa.Zato ukupna veličina PMT iznosi: $3 \cdot 256MB = 768MB$.

2. zadatak, prvi kolokvijum, septembar 2012.

Virtuelna memorija nekog računara organizovana je straničnо. Veličina virtuelnog adresnog prostora je 16 MB, adresibilna jedinica je bajt, a veličina stranice je 64 KB. Veličina fizičkog adresnog prostora je 16 MB. Operativni sistem učitava stranice na zahtev, tako što se stranica učitava u prvi slobodni okvir fizičke memorije kada joj se pristupi. Kada se kreira proces, ni jedna njegova stranica se ne učitava odmah, već tek kad joj se prvi put pristupi. U početnom trenutku, slobodni okviri fizičke memorije su okviri počev od 20h zaključno sa 2Fh. Neki proces generiše sledeću sekvencu virtuelnih adresa tokom svog izvršavanja (sve vrednosti su heksadecimalne):

30F00, 30F02, 30F04, 822F0, 822F2, 322F0, 322F2, 322F4, 522F0, 522F2, 602F0, 602F2

Prikazati izgled prvih 16 ulaza tabele preslikavanja stranica (PMT) za ovaj proces posle izvršavanja date sekvence. Za svaki ulaz u PMT prikazati indikator prisutnosti stranice u fizičkoj memoriji (0 ili 1) i broj okvira u fizičkoj memoriji u koji se stranica preslikava, ukoliko je stranica učitana; ukoliko nije, prikazati samo ovaj indikator.

Rešenje

VA: Page(8):Offset(16)

PA: Frame(8):Offset(16)

Sekvenca stranica koje se traže: $3, 3, 3, 8, 8, 3, 3, 3, 5, 5, 6, 6$

PMT na kraju sekvence:

Entry	Flag	Frame
0	0	
1	0	
2	0	
3	1	20h
4	0	
5	1	22h
6	1	23h
7	0	
8	1	21h
9	0	
A	0	
B	0	
C	0	
D	0	
E	0	
F	0	

4. zadatak, drugi kolokvijum, septembar 2012.

Neki sistem sa straničnom organizacijom virtuelne memorije koristi tehniku *copy-on-write* i sistemski poziv *fork* za kreiranje procesa.

Odgovoriti na sledeća pitanja uz precizna obrazloženja.

Da li ovakav sistem mora da kreira i novu PMT za novokreirani proces odmah pri kreiranju procesa u sistemskom pozivu *fork*? Ako mora, zašto mora? Ako ne mora, u kom trenutku najkasnije mora da kreira novu PMT za proces? Šta se dešava sa vrednošću PMTP u PCB procesa?

Rešenje

Sistem ne mora da kreira novu PMT za novokreirani proces prilikom izvršavanja sistemskog poziva *fork*, jer oba procesa inicijalno dele sve stranice, pa su njihove PMT inicijalno potpuno iste. Isto važi i za PMTP koji ukazuje na istu PMT, pa su i oni isti. Kada bilo koji od ovih procesa generiše izuzetak zbog zabranjenog upisa u deljenu stranicu, sistem mora da razdvoji stranice kopiranjem u različite fizičke okvire (*copy-on-write*). Kako sada procesi imaju razdvojenu stranicu preslikanu u različite fizičke okvire, njihove PMT postaju različite (različit je sadržaj deskriptora za razdvojenu stranicu), pa je to najkasniji trenutak kada sistem mora da formira sopstvenu PMT za novokreirani proces. Kako sada procesi imaju različite PMT, i njihovi PMTP postaju tada različiti.

2. zadatak, prvi kolokvijum, april 2011.

Neki sistem podržava straničnu organizaciju virtuelne memorije. Virtuelna adresa je 32-bitna, stranica je veličine 64KB, a adresibilna jedinica je bajt. Fizički adresni prostor je veličine 4GB. U tabeli preslikavanja stranica (PMT) svaki ulaz zauzima samo onoliko prostora koliko je potrebno da se smesti broj okvira u koji se data stranica preslikava, pošto se informacije o smeštaju stranica na disku čuvaju u drugoj strukturi. Pri tome, vrednost 0 u ulazu u PMT označava da data stranica nije učitana u fizičku memoriju (nijedna stranica korisničkog procesa ne preslikava se u okvir 0 gde se inače nalazi interapt vektor tabela). Trenutno stanje PMT dva posmatrana procesa A i B je sledeće (brojevi ulaza i vrednosti su dati heksadecimalno):

Proces A:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	FE12	FEFF	0	0	0	0	14	FE	0

Proces B:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	0	12	2314	01AD	0	22	01AE	0	0

Prikazati sadržaj ovih tabela nakon što je operativni sistem završio sve sledeće akcije tim redom:

- Obradio straničnu grešku (*page fault*) koju je generisao proces A kada je adresirao adresu 30203h u svom adresnom prostoru, tako što je izbacio stranicu broj 1 procesa B i na njeno mesto učitao traženu stranicu procesa A.
- Obradio straničnu grešku (*page fault*) koju je generisao proces B kada je adresirao adresu 1F00Fh u svom adresnom prostoru, tako što je izbacio stranicu broj FFh procesa A i na njeno mesto učitao traženu stranicu procesa B.
- Obradio sistemski poziv procesa A kojim je taj proces zahtevao deljenje svoje stranice broj 2 sa stranicom broj 1 procesa B.

Rešenje

Proces A:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	FE12	FEFF	14	12	0	0	0	FE	0

Proces B:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	0	14	2314	01AD	0	22	01AE	0	0

2. zadatak, prvi kolokvijum, maj 2011.

Virtuelna memorija nekog računara organizovana je straničnо. Veličina virtuelnog adresnog prostora je 2 MB, adresibilna jedinica je 16-bitna reč, a veličina stranice je 128 KB. Veličina fizičkog adresnog prostora je 32 MB. Operativni sistem učitava stranice na zahtev, tako što se stranica učitava u prvi slobodni okvir fizičke memorije kada joj se pristupi. Kada se kreira proces, ni jedna njegova stranica se ne učitava odmah, već tek kad joj se prvi put pristupi. U početnom trenutku, slobodni okviri fizičke memorije su okviri počev od 10h zaključno sa 1Fh. Neki proces generiše sledeću sekvencu virtuelnih adresa tokom svog izvršavanja (sve vrednosti su heksadecimalne):

30F00, 30F02, 30F04, 922F0, 922F2, 322F0, 322F2, 322F4, 522F0, 522F2, 402F0, 402F2

Prikazati izgled cele tabele preslikavanja stranica (PMT) za ovaj proces posle izvršavanja ove sekvence. Za svaki ulaz u PMT prikazati indikator prisutnosti stranice u fizičkoj memoriji (0 ili 1) i broj okvira u fizičkoj memoriji u koji se stranica preslikava, ukoliko je stranica učitana; ukoliko nije, prikazati samo ovaj indikator.

Rešenje

VA: Page(4):Offset(16)

PA: Block(8):Offset(16)

Sekvenca stranica koje se traže: 3, 3, 3, 9, 9, 3, 3, 3, 5, 5, 4, 4

PMT na kraju sekvence:

Entry	Flag	Frame
0	0	
1	0	
2	0	
3	1	10h
4	1	13h
5	1	12h
6	0	
7	0	
8	0	
9	1	11h
A	0	
B	0	
C	0	
D	0	
E	0	
F	0	

4. zadatak, drugi kolokvijum, maj 2011.

U nekom sistemu sa straničnom organizacijom virtuelne memorije koristi se *copy-on-write* tehnika deljenja stranica između procesa. U deskriptoru stranice u tabeli preslikavanja stranica (PMT) procesa nalaze se samo sledeće informacije: broj okvira u koji se stranica preslikava (0 označava da stranica nije u memoriji, okvir broj 0 se ne dodeljuje procesima) i biti prava pristupa (*R*-dozvoljeno čitanje u fazi izvršavanja instrukcije/dohvatanja operanda, *W*- dozvoljen upis, *E*-dozvoljeno čitanje u fazi dohvatanja instrukcije). Pored PMT, operativni sistem za svaki proces vodi posebnu strukturu podataka koju naziva *VMStruct* i u kojoj se čuvaju podaci koje koristi operativni sistem, a koji nisu potrebni hardveru za preslikavanje adresa. Svaki deskriptor u ovoj strukturi opisuje čitav skup susednih stranica koje predstavljaju logičku celinu, jer su svi ovi podaci za njih isti, i koji se u ovom kontekstu naziva *region*. U ovom deskriptoru čuvaju se sledeći podaci: prva stranica u skupu stranica na koje se deskriptor odnosi (*Start Page#*), broj susednih stranica na koje se deskriptor odnosi (*Region Length*), biti prava pristupa na logičkom nivou (*R*-dozvoljeno čitanje u fazi izvršavanja instrukcije/dohvatanja operanda, *W*-dozvoljen upis, *E*-dozvoljeno čitanje u fazi dohvatanja instrukcije), da li su stranice u ovom regionu deljene tako da ih treba kopirati pri prvom upisu (*Copy-On-Write*: 0-nisu deljene, 1-jesu deljene i treba ih kopirati pri upisu, samo ako je upis dozvoljen na logičkom nivou), kao i mesto na disku gde se zamenjuju stranice iz tog regiona (nije relevantno za ovaj zadatak). Posmatra se proces *Parent* čiji su delovi struktura *PMT* i *VMStruct* dati u nastavku.

Page#	Frame#	RWE
A04h	23h	?
BF0h	14h	?
C0Ah	7Ah	?

(i) PMT

StartPage#	Region Length	RWE-Copy-On-Write	Opis
A00h	50h	001-0	Code Region
B00h	FFh	110-0	Data Region
C00h	70h	100-0	Input Buffer Region

(j) VMStruct

Ovaj proces izvršava sistemski poziv `fork()` i uspešno kreira proces potomak *Child*. Prikazati iste delove struktura *PMT* i *VMStruct* za oba procesa *Parent* i *Child* neposredno nakon ovog poziva. Sistem primenjuje *demand paging* strategiju (dohvata stranicu tek kada se prvi put zatraži, ne alocira je odmah pri kreiranju procesa).

Rešenje

Proces Parent:

Page#	Frame#	RWE
A04h	23h	001
BF0h	14h	100
C0Ah	7Ah	100

(k) PMT

StartPage#	Region Length	RWE-Copy-On-Write	Opis
A00h	50h	001-0	Code Region
B00h	FFh	110-0	Data Region
C00h	70h	100-0	Input Buffer Region

(l) VMStruct

Proces Child: Sve isto.

Napomena: Primetiti da se u opisanom sistemu bit Copy-On-Write (u daljem tekstu CoW) odnosi na čitav set stranica. Zbog toga ovaj bit nije dovoljan da bi se odredilo da li pri upisu u neku stranicu iz seta, tu stranicu treba kopirati. Stranice koje pri upisu stvarno treba kopirati su one koje pripadaju regionu za koji je postavljen bit CoW i kojima je u trnutku upisa bitom W u PMT zabranjen upis. Kada jednom dođe do upisa i samim tim i do kopiranja stranice, za novu kopiju se setuje bit W u PMT i pri kasnijim upisima u tu stranicu ne treba vršiti kopiranje te stranice bez obzira što stranica pripada regionu za koji je setovan CoW bit. Stoga bi se sistem mogao implementirati tako da se u CoW bit jednom upiše vrednost pri pokretanju procesa i više nikada ne menja. U takvoj varijanti, CoW bit bi čak bio suvišan jer bi uvek imao istu vrednost kao i W bit u VMStruct strukturi.

2. zadatak, prvi kolokvijum, april 2010.

Neki sistem podržava straničnu organizaciju virtuelne memorije. Virtuelna adresa je 64-bitna, stranica je veličine 64KB, a adresibilna jedinica je bajt. Fizički adresni prostor je veličine 4GB. U tabeli preslikavanja stranica (PMT) svaki ulaz zauzima samo onoliko prostora koliko je potrebno da se smesti broj okvira u koji se data stranica preslikava, pošto se informacije o smeštaju stranica na disku čuvaju u drugoj strukturi. Pri tome, vrednost 0 u ulazu u PMT označava da data stranica nije učitana u fizičku memoriju (nijedna stranica korisničkog procesa ne preslikava se u okvir 0 gde se inače nalazi interapt vektor tabela). Trenutno stanje PMT dva posmatrana procesa A i B je sledeće (brojevi ulaza i vrednosti su dati heksadecimalno):

Proces A:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	FE12	FEFF	0	0	0	0	14	FE	0

Proces B:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	0	12	2314	01AD	0	22	01AE	0	0

Prikazati sadržaj ovih tabela nakon što operativni sistem završi sve sledeće akcije tim redom:

- Obradio straničnu grešku (*page fault*) koju je generisao proces A kada je adresirao adresu FE030203 u svom adresnom prostoru, tako što je izbacio stranicu broj 2 procesa B i na njeno mesto učitao traženu stranicu procesa A.
- Obradio straničnu grešku (*page fault*) koju je generisao proces B kada je adresirao adresu 02FEFF u svom adresnom prostoru, tako što je izbacio stranicu broj FF procesa A i na njeno mesto učitao traženu stranicu procesa B.
- Obradio sistemski poziv procesa A kojim je taj proces zahtevao deljenje svoje stranice broj 2 sa stranicom broj 1 procesa B.

Rešenje

Proces A:

Ulaz	0	1	2	3	...	FE	FF	100	...	FE03	...
Vrednost	FE12	FEFF	12	0	0	0	0	FE	0	2314	0

Proces B:

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	0	12	14	01AD	0	22	01AE	0	0

4. zadatak, drugi kolokvijum, maj 2010.

Virtuelni adresni prostor sistema je 4GB, adresibilna jedinica je bajt, a virtuelni adresni prostor je organizovan stranično sa stranicom veličine 64KB. Fizički adresni prostor je veličine 4GB. Deskriptor stranice koji se čuva u tabeli preslikavanja (PMT) i koga hardver za preslikavanje adresa učitava i koristi za proveru prisutnosti stranice i preslikavanje sadrži samo broj okvira u fizičkoj memoriji u koji se data stranica preslikava, s tim da vrednost 0 označava da data stranica nije u memoriji, pošto se stranica nikada ne preslikava u okvir 0 u fizičkoj memoriji koji je rezervisan za interapt vektor tabelu; ostale informacije o stranici operativni sistem čuva u zasebnim strukturama van PMT. Tabele preslikavanja stranica su organizovane u dva nivoa, s tim da tabela prvog nivoa ima 1K ulaza. Posmatra se proces koji koristi prvih 400 stranica i poslednjih 64 stranica svog virtuelnog adresnog prostora, dok su mu ostale zabranjene za pristup jer ih ne koristi. Odgovoriti na sledeća pitanja i precizno obrazložiti odgovore:

1. Prikazati logičku strukturu virtuelne adrese i označiti širinu svakog polja. Označiti i podelu polja za broj stranice na polja za indeksiranje PMT prvog i drugog nivoa.
2. Kolika je veličina jednog ulaza u PMT prvog nivoa i šta taj ulaz sadrži?
3. Koliko bi maksimalno ukupno memorije zauzimale PMT nekog procesa koji bi koristio ceo svoj virtuelni adresni prostor?
4. Koliko ukupno memorije zauzimaju PMT za opisani proces?

Rešenje

1. Virtuelni adresni prostor: 4GB = $2^2 \cdot 2^{30}$B = 2^{32}B, pa je virtuelna adresa širine 32 bita.

   Fizički adresni prostor: 4GB = 2^{32}B, pa je fizička adresa širine 32 bita.

   Veličina stranice i okvira: 64KB = $2^6 \cdot 2^{10}$B = 2^{16}B, pa je širina polja za pomeraj unutar stranice i okvira 16 bita.

   Odatle sledi da je širina polja unutar virtuelne adrese za broj stranice 32-16 = 16 bita, širina polja za broj okvira unutar fizičke adrese 32-16 = 16 bita, a širina deskriptora (ulaza u PMT drugog nivoa) isto toliko – 16 bita, odnosno 2 bajta.

   Stranica prvog nivoa ima 1K = 2^{10} ulaza, pa je širina polja za indeksiranje PMT prvog nivoa 10 bita, a za indeksiranje PMT drugog nivoa 16-10 = 6 bita. Prema tome, struktura virtuelne adrese je: Page_L1(10):Page_L2(6):Offset(16).

2. Ulaz u PMT prvog nivoa sadrži adresu početka PMT drugog nivoa u fizičkoj memoriji, s tim da vrednost 0 može da označava nekorišćeni opseg stranica (invalidan ulaz), pošto se ni PMT drugog nivoa ne može smestiti počev od adrese 0. Prema tome, širina ulaza u PMT prvog nivoa je najmanje jednaka širini fizičke adrese, što je 32 bita. Drugim rečima, jedan ulaz u PMT prvog nivoa zauzima 4 bajta.

3. PMT prvog nivoa zauzima 1K ulaza po 4 bajta, dakle 4KB.

   Jedan ulaz u PMT drugog nivoa sadrži broj okvira, koji je širine 16 bita, pa zauzima 2 bajta. PMT drugog nivoa ima 2^6 = 64 ulaza, pa zauzima 128B. Prema tome, PMT ukupno zauzimaju maksimalno: $4 \cdot 2^{10}$B (veličina PMT prvog nivoa) + 2^{10} (broj PMT drugog nivoa) $\cdot 2^7$B (veličina PMT drugog nivoa) = 2^{12}B + 2^{17}B, što iznosi 132KB.

4. Dati proces ima validna samo prvih sedam i poslednji ulaz u PMT prvog nivoa, dakle za njega postoje samo osam PMT drugog nivoa u memoriji. Ukupna veličina PMT za ovaj proces je zato: $4 \cdot 2^{10}$B (veličina PMT prvog nivoa) + 8 (broj PMT drugog nivoa) $\cdot 2^7$B (veličina PMT drugog nivoa) = $4 \cdot 2^{10}$B + 2^{10}B = 5KB.

4. zadatak, drugi kolokvijum, maj 2009.

Virtuelni adresni prostor sistema je 4GB, adresibilna jedinica je bajt, a virtuelni adresni prostor je organizovan stranično sa stranicom veličine 16KB. Fizički adresni prostor je veličine 1GB. Deskriptor stranice koji se čuva u tabeli preslikavanja (PMT) i koga hardver za preslikavanje adresa učitava i koristi za proveru prisutnosti stranice i preslikavanje sadrži samo broj okvira u fizičkoj memoriji u koji se data stranica preslikava, s tim da vrednost 0 označava da data stranica nije u memoriji, pošto se stranica nikada ne preslikava u okvir 0 u fizičkoj memoriji koji je rezervisan za interapt vektor tabelu; ostale informacije o stranici operativni sistem čuva u zasebnim strukturama van PMT. Tabele preslikavanja stranica su organizovane u dva nivoa, s tim da tabela prvog nivoa ima 2K ulaza.

Posmatra se proces koji koristi prvih 800 stranica i poslednjih 56 stranica svog virtuelnog adresnog prostora, dok su mu ostale zabranjene za pristup jer ih ne koristi. Odgovoriti na sledeća pitanja i precizno obrazložiti odgovore:

1. Prikazati logičku strukturu virtuelne adrese i označiti širinu svakog polja. Označiti i podelu polja za broj stranice na polja za indeksiranje PMT prvog i drugog nivoa.
2. Kolika je veličina jednog ulaza u PMT prvog nivoa i šta taj ulaz sadrži?
3. Koliko bi maksimalno ukupno memorije zauzimale PMT nekog procesa koji bi koristio ceo svoj virtuelni adresni prostor?
4. Koliko ukupno memorije zauzimaju PMT za opisani proces?

Rešenje

1. Virtuelni adresni prostor: $4GB = 2^2 \cdot 2^{30}B = 2^{32}B$, pa je virtuelna adresa širine 32 bita. Fizički adresni prostor: $1GB = 2^{30}B$, pa je fizička adresa širine 30 bita. Veličina stranice i okvira: $16KB = 2^4 \cdot 2^{10}B = 2^{14}B$, pa je širina polja za pomeraj unutar stranice i okvira 14 bita. Odatle sledi da je širina polja unutar virtuelne adrese za broj stranice 32-14 = 18 bita, širina polja za broj okvira unutar fizičke adrese 30-14 = 16 bita, a širina deskriptora (ulaza u PMT drugog nivoa) isto toliko – 16 bita, odnosno 2 bajta. Stranica prvog nivoa ima $2K = 2^{11}$ ulaza, pa je širina polja za indeksiranje PMT prvog nivoa 11 bita, a za indeksiranje PMT drugog nivoa 18-11 = 7 bita. Prema tome, struktura virtuelne adrese je: Page_L1(11):Page_L2(7):Offset(14).
2. Ulaz u PMT prvog nivoa sadrži adresu početka PMT drugog nivoa u fizičkoj memoriji, s tim da vrednost 0 može da označava nekorišćeni opseg stranica (invalidan ulaz), pošto se ni PMT drugog nivoa ne može smestiti počev od adrese 0. Prema tome, širina ulaza u PMT prvog nivoa je najmanje jednaka širini fizičke adrese, što je 30 bita. Drugim rečima, jedan ulaz u PMT prvog nivoa zauzima 4 bajta.
3. PMT prvog nivoa zauzima 2K ulaza po 4 bajta, dakle 8KB. Jedan ulaz u PMT drugog nivoa sadrži broj okvira, koji je širine 16 bita, pa zauzima 2 bajta. PMT drugog nivoa ima 27 = 128 ulaza, pa zauzima 256B. Prema tome, PMT ukupno zauzimaju maksimalno: $4 \cdot 2^{11}B$ (veličina PMT prvog nivoa) + 2^{11} (broj PMT drugog nivoa) $\cdot 2^8B$ (veličina PMT drugog nivoa) = $2^{13}B + 2^{19}B$, što iznosi 520KB.
4. Dati proces ima validna samo prvih sedam i poslednji ulaz u PMT prvog nivoa, dakle za njega postoje samo osam PMT drugog nivoa u memoriji. Ukupna veličina PMT za ovaj proces je zato: $4 \cdot 2^{11}B$ (veličina PMT prvog nivoa) +8 (broj PMT drugog nivoa) $\cdot 2^8B$ (veličina PMT drugog nivoa) = $8 \cdot 2^{10}B + 2 \cdot 2^{10}B = 10KB$.

3. zadatak, prvi kolokvijum, april 2008.

Neki sistem podržava straničnu organizaciju virtuelne memorije. Virtuelni adresni prostor je veličine 4GB, stranica je veličine 256KB, a adresibilna jedinica je bajt. Fizički adresni prostor je veličine 16GB. U tabeli preslikavanja stranica (PMT) svaki ulaz zauzima samo onoliko prostora koliko je potrebno da se smesti broj okvira u koji se data stranica kojoj odgovara taj ulaz preslikava, pošto se informacije o smeštaju stranica na disku čuvaju u drugoj strukturi. Pri tome, vrednost -1 (sve jedinice) u ulazu u PMT označava da data stranica nije dozvoljena za pristup, jer proces nije deklarisao da koristi taj deo adresnog prostora, a vrednost 0 označava da data stranica jeste dozvoljena za pristup, ali nije u fizičkoj memoriji. (Naravno, prvi i poslednji okvir u fizičkom adresnom prostoru se zbog toga nikada ne koriste za smeštanje stranica procesa.) Trenutno stanje PMT je sledeće (brojevi ulaza i vrednosti su dati heksa decimalno):

Ulaz	0	1	2	3	...	3FE	3FF	400	...
Vrednost	0	25F	0	FF0	-1	2AD0	0	14	-1

U donjoj tabeli date su virtuelne adrese koje se posmatraju nezavisno jedna od druge (ne u sekvenci, već se posmatra svaka za dato stanje PMT nezavisno od drugih, kao da druge nisu generisane).

Popuniti donju tabelu za svaku od datih virtuelnih adresa (zapis je heksadecimalan) na sledeći način: ukoliko je pristup toj virtuelnoj adresi nedozvoljen, odnosno ako se generiše izuzetak zbog nelegalnog pristupa memoriji (engl. *memory access violation*), u polje upisati MAV; ako data adresa generiše izuzetak tipa straničnе greške (engl. *page fault*), upisati PF; inače, upisati vrednost (pune) fizičke adrese u koju se ova preslikava, u heksadecimalnom zapisu.

Virtuelna adresa	FFC2673	D385A	FF7FB32	FFFA8C4	10012EB8
Rezultat adresiranja					

Rešenje

Virtuelna adresa	FFC2673	D385A	FF7FB32	FFFA8C4	10012EB8
Rezultat adresiranja	PF	3FC1385A	MAV	PF	512EB8

4. zadatak, drugi kolokvijum, maj 2008.

Virtuelni adresni prostor sistema je 4GB, adresibilna jedinica je bajt, a virtuelni adresni prostor je organizovan stranično sa stranicom veličine 4KB. Fizički adresni prostor je veličine 256MB. Deskriptor stranice koji se čuva u tabeli preslikavanja (PMT) i koga hardver za preslikavanje adresa učitava i koristi za proveru prisutnosti stranice i preslikavanje sadrži samo broj okvira u fizičkoj memoriji u koji se data stranica preslikava, s tim da vrednost 0 označava da data stranica nije u memoriji, pošto se stranica nikada ne preslikava u okvir 0 u fizičkoj memoriji koji je rezervisan za interapt vektor tabelu; ostale informacije o stranici operativni sistem čuva u zasebnim strukturama van PMT. Tabele preslikavanja stranica su organizovane u dva nivoa, s tim da tabela prvog nivoa ima 1K ulaza. Posmatra se proces koji koristi prve 1032 stranice i poslednjih 10 stranica svog virtuelnog adresnog prostora, dok su mu ostale zabranjene za pristup jer ih ne koristi. Odgovoriti na sledeća pitanja i precizno obrazložiti odgovore:

1. Prikazati logičku strukturu virtuelne adrese i označiti širinu svakog polja. Označiti i podelu polja za broj stranice na polja za indeksiranje PMT prvog i drugog nivoa.
2. Kolika je veličina jednog ulaza u PMT prvog nivoa i šta taj ulaz sadrži?
3. Koliko bi maksimalno ukupno memorije zauzimale PMT nekog procesa koji bi koristio ceo svoj virtuelni adresni prostor?
4. Koliko ukupno memorije zauzimaju PMT za opisani proces?

Rešenje

1. Virtuelni adresni prostor: $4GB = 22 \cdot 2^{30}B = 2^{32}B$, pa je virtuelna adresa širine 32 bita. Fizički adresni prostor: $256MB = 28 \cdot 2^{20}B = 2^{28}B$, pa je fizička adresa širine 28 bita. Veličina stranice i okvira: $4KB = 22 * 2^{10}B = 2^{12}B$, pa je širina polja za pomeraj unutar stranice i okvira 12 bita.

   Odatle sledi da je širina polja unutar virtuelne adrese za broj stranice 32-12 = 20 bita, širina polja za broj okvira unutar fizičke adrese 28-12 = 16 bita, a širina deskriptora (ulaza u PMT drugog nivoa) isto toliko – 16 bita, odnosno 2 bajta.

   Stranica prvog nivoa ima $1K = 2^{10}$ ulaza, pa je širina polja za indeksiranje PMT prvog nivoa 10 bita, a za indeksiranje PMT drugog nivoa 20-10 = 10 bita.

   Prema tome, struktura virtuelne adrese je: Page_L1(10):Page_L2(10):Offset(12).

2. Ulaz u PMT prvog nivoa sadrži adresu početka PMT drugog nivoa u fizičkoj memoriji, s tim da vrednost 0 može da označava nekorišćeni opseg stranica (invalidan ulaz), pošto se ni PMT drugog nivoa ne može smestiti počev od adrese 0. Prema tome, širina ulaza u PMT prvog nivoa je najmanje jednaka širini fizičke adrese, što je 28 bita. Drugim rečima, jedan ulaz u PMT prvog nivoa zauzima 4 bajta.

3. PMT prvog nivoa zauzima 1K ulaza po 4 bajta, dakle 4KB. Jedan ulaz u PMT drugog nivoa sadrži broj okvira, koji je širine 16 bita, pa zauzima 2 bajta. PMT drugog nivoa ima $2^{10} = 1K$ ulaza, pa zauzima 2KB. Prema tome, PMT ukupno zauzimaju maksimalno: $4 \cdot 2^{10}B$ (veličina PMT prvog nivoa) $+2^{10}$ (broj PMT drugog nivoa) $\cdot 2^{11}B$ (veličina PMT drugog nivoa) $= 4 \cdot 2^{10}B + 2^{21}B$, što je približno (odnosno nešto veće od) $2^{21}B = 2MB$.

4. Dati proces ima validna samo prva dva i poslednji ulaz u PMT prvog nivoa, dakle za njega postoje samo tri PMT drugog nivoa u memoriji. Ukupna veličina PMT za ovaj proces je zato: $4 \cdot 2^{10}B$ (veličina PMT prvog nivoa) $+3$ (broj PMT drugog nivoa) $\cdot 2^{11}B$ (veličina PMT drugog nivoa) $= 4 \cdot 2^{10}B + 6 \cdot 2^{10}B = 10KB$.

5. zadatak, drugi kolokvijum, maj 2008.

U nekom operativnom sistemu primenjuje se tehnika *copy-on-write* i učitavanje stranica virtuelne memorije na zahtev (*demand paging*). Za potrebe preslikavanja stranica postoje dve odvojene strukture: tabela preslikavanja stranica (PMT) koju hardver za preslikavanje adresa jedino koristi i koja sadrži *minimum* potrebnih informacija za ovo hardversko preslikavanje, kako bi bila što manja, i struktura koju isključivo koristi operativni sistem i koja čuva ostale informacije o virtuelnom adresnom prostoru procesa. Drugim rečima, u PMT su uključene one i samo one

informacije koje su potrebne hardveru za preslikavanje adresa. Za svaku od sledećih informacija navesti da li se nalazi u ovoj PMT ili ne:

- Indikator da li se stranica nalazi u fizičkoj memoriji.
- Indikator da li je stranica uopšte dozvoljena za pristup (registrovana kao korišćeni deo virtuelnog adresnog prostora).
- Indikator da li je hardveru dozvoljen upis u stranicu.
- Indikator da je logički dozvoljen upis u stranicu, ali je ona deljena sa *copy-on-write* semantikom.
- Indikator da li je stranica „prljava“, odnosno menjana od svog učitavanja.
- Broj okvira u koji se stranica preslikava.
- Broj bloka na particiji za zamenu stranica.

Rešenje

- Indikator da li se stranica nalazi u fizičkoj memoriji. Da
- Indikator da li je stranica uopšte dozvoljena za pristup (registrovana kao korišćeni deo virtuelnog adresnog prostora). Ne
- Indikator da li je hardveru dozvoljena upis u stranicu. Da
- Indikator da li je dozvoljen upis u stranicu, ali je ona deljena sa *copy-on-write* semantikom. Ne
- Indikator da li je stranica „prljava“, odnosno menjana od svog učitavanja. Da
- Broj okvira u koji se stranica preslikava. Da
- Broj bloka na particiji za zamenu stranica. Ne

2. zadatak, prvi kolokvijum, april 2007.

Neki računar podržava straničnu organizaciju virtuelne memorije, pri čemu je virtuelni adresni prostor veličine 16GB, adresibilna jedinica je 32-bitna reč, a fizički adresni prostor je veličine 1GB. Stranica je veličine 1KB. Jedan ulaz u tabeli preslikavanja stranica (PMT) zauzima jednu reč, s tim da najviši bit ukazuje na to da li je stranica u memoriji ili ne, a najniži biti predstavljaju broj okvira u fizičkoj memoriji u koji se stranica preslikava.

1. Prikazati logičku strukturu virtuelne i fizičke adrese i označiti širinu svakog polja.
2. Na jeziku C napisati kod funkcije

```
void setPageDescr(unsigned int* pmtp, unsigned int page, unsigned int frame);
```

koja u tabelu na čiji početak ukazuje dati pokazivač `pmtp`, za stranicu sa datim brojem `page`, upisuje deskriptor tako da se ta stranica preslikava u okvir sa datim brojem `frame`.

Rešenje

1. VA: Page(24) Offset(8)

   PA: Block(20) Offset(8)

2.
```
void setPageDescr(unsigned int* pmtp, unsigned int page, unsigned int frame){
    pmtp[page] = frame | ~((unsigned int)~0 / 2);
}
```

4. zadatak, drugi kolokvijum, maj 2007.

Virtuelni adresni prostor sistema je 16EB (EB je oznaka za eksabajt, engl. exabyte, kilo- mega-giga-tera-peta-eksa), adresibilna jedinica je bajt, a v irtuelni adresni prostor je organizovan straničnо sa stranicom veličine 16MB. Fizički adresni prostor je veličine 1TB (terabajt). Deskriptor stranice koji se čuva u tabeli preslikavanja (PMT) i koga hardver za preslikavanje adresa učitava i koristi za proveru prisutnosti stranice i preslikavanje sadrži samo broj okvira u fizičkoj memoriji u koji se data stranica preslikava, s tim da vrednost 0 označava da data stranica nije u memoriji, pošto se stranica nikada ne preslikava u okvir 0 u fizičkoj memoriji koji je rezervisan za interapt vektor tabelu; ostale informacije o stranici operativni sistem čuva u zasebnim strukturama van PMT. Tabele preslikavanja stranica su organizovane u dva nivoa, s tim da tabela prvog nivoa ima 1M ulaza. Posmatra se proces koji koristi prvih 1K stranica i poslednjih 200 stranica svog virtuelnog adresnog prostora, dok su mu ostale zabranjene za pristup jer ih ne koristi. Odgovoriti na sledeća pitanja i precizno obrazložiti odgovore:

1. Prikazati logičku strukturu virtuelne adrese i označiti širinu svakog polja. Označiti i podelu polja za broj stranice na polja za indeksiranje PMT prvog i drugog nivoa.
2. Kolika je veličina jednog ulaza u PMT prvog nivoa i šta taj ulaz sadrži?
3. Koliko bi maksimalno ukupno memorije zauzimale PMT nekog procesa koji bi koristio ceo svoj virtuelni adresni prostor?
4. Koliko ukupno memorije zauzimaju PMT za opisani proces?

Rešenje

1. Virtuelni adresni prostor: $16EB = 2^4 \cdot 2^{60}B = 2^{64}B$, pa je virtuelna adresa širine 64 bita. Fizički adresni prostor: $1TB = 2^{40}B$, pa je fizička adresa širine 40 bita. Veličina stranice i okvira: $16MB = 2^4 \cdot 2^{20}B = 2^{24}B$, pa je širina polja za pomeraj unutar stranice i okvira 24 bita.

   Odatle sledi da je širina polja unutar virtuelne adrese za broj stranice 64-24 = 40 bita, širina polja za broj okvira unutar fizičke adrese 40-24 = 16 bita, a širina deskriptora (ulaza u PMT drugog nivoa) isto toliko – 16 bita, odnosno 2 bajta.

   Stranica prvog nivoa ima $1M = 2^{20}$ ulaza, pa je širina polja za indeksiranje PMT prvog nivoa 20 bita, a za indeksiranje PMT drugog nivoa 40 - 20 = 20 bita. Prema tome, struktura virtuelne adrese je: Page_L1(20):Page_L2(20):Offset(24).

2. Ulaz u PMT prvog nivoa sadrži adresu početka PMT drugog nivoa u fizičkoj memoriji, s tim da vrednost 0 može da označava nekorišćeni opseg stranica (invalidan ulaz), pošto se ni PMT drugog nivoa ne može smestiti počev od adrese 0. Prema tome, širina ulaza u PMT prvog nivoa je jednaka širini fizičke adrese, što je 40 bita. Drugim rečima, jedan ulaz u PMT prvog nivoa zauzima 5 bajtova.

3. PMT prvog nivoa zauzima 1M ulaza po 5 bajtova, dakle 5MB. Jedan ulaz u PMT drugog nivoa sadrži broj okvira, koji je širine 16 bita, pa zauzima 2 bajta. PMT drugog nivoa ima 2^{20} = 1M ulaza, pa zauzima 2MB. Prema tome, PMT ukupno zauzimaju maksimalno: $5 \cdot 2^{20}B$(veličina PMT prvog nivoa) $+2^{20}$ (broj PMT drugog nivoa) $\cdot 2^{21}B$ (veličina PMT drugog nivoa) $= 5 \cdot 2^{20}B + 2^{41}B$, što je približno (odnosno nešto veće od) $2^{41}B = 2TB$ (terabajta).

4. Dati proces ima validan samo prvi i poslednji ulaz u PMT prvog nivoa, dakle za njega postoje samo dve PMT drugog nivoa u memoriji. Ukupna veličina PMT za ovaj proces je zato: $5 \cdot 2^{20}B$ (veličina PMT prvog nivoa) + 2 (broj PMT drugog nivoa) $\cdot 2^{21}B$ (veličina PMT drugog nivoa) $= 5 \cdot 2^{20}B + 4 \cdot 2^{20}B = 9MB$, što je značajno manje od fizičkog adresnog prostora.

4. zadatak, prvi kolokvijum, april 2006.

U nekom operativnom sistemu tabele preslikavanja stranica virtuelne memorije (PMT) složene su u niz od N elemenata, tako da procesu sa identifikatorom i pripada tabela u i-tom elementu ovog niza. Da li vrednost koju treba upisati u registar procesora koji ukazuje na tabelu preslikavanja (MTP) treba da bude element strukture PCB? Precizno obrazložiti odgovor.

Rešenje

Ne treba, pošto je identifikator procesa ujedno i ulaz u opisani niz. Nije potrebno dodatno čuvati tu informaciju.

5. zadatak, drugi kolokvijum, maj 2006.

U nekom sistemu sa straničnom organizacijom virtuelne memorije u datom trenutku raspored stranica tri procesa po okvirima u fizičkoj memoriji izgleda redom ovako (oznake su složene redom po rastućem broju okvira, slova A, B i C označavaju procese kojima pripadaju stranice u tim okvirima, brojevi označavaju brojeve stranica tih procesa, K označava okvir koji pripada jezgru OS-a, a DLL označava okvir čiji je sadržaj DLL koga koriste sva tri procesa A, B i C):

```
K, A1, B3, C1, DLL, C0, B2, K, K
```

U virtuelnom adresnom prostoru procesa A dati DLL se nalazi na stranici broj 2, a u prostorima procesa B i C na stranici broj 5. Napisati kako izgledaju tabele preslikavanja stranica za sva tri procesa A, B i C (u tabelu upisati brojeve okvira i T ili F kao oznaku da li je stranica u memoriji ili ne, za prvih 6 stranica).

Strana	0	1	2	3	4	5
Proces A						
Proces B						
Proces C						

Rešenje

Strana	0	1	2	3	4	5
Proces A	F	T 1	T 4	F	F	F
Proces B	F	F	T 6	T 2	F	T 4
Proces C	T 5	T 3	F	F	F	T 4

Kontinualna alokacija

1. zadatak, kolokvijum, avgust 2021.

Implementirati bibliotečnu funkciju `malloc` za dinamičku alokaciju memorijskog bloka veličine size (sve deklaracije date su dole). Ako nema dovoljno slobodne memorije, ova funkcija vraća `nullptr`.

Evidencija slobodnih blokova u virtuelnom adresnom prostoru procesa koju vodi ovaj alokator organizovana je kao neuređena jednostruko ulančana lista sa glavom u `freeMemHead`. Svaki element liste je jedan slobodan deo memorije koji na početku ima zaglavlje tipa `BlockHeader` u kome je pokazivač na sledeći takav zapis, kao i veličina slobodnog bloka iza ovog zaglavlja (neto, bez veličine zaglavlja). Funkcija `malloc` najpre traži slobodan blok dovoljne veličine u ovoj svojoj evidenciji slobodnih blokova algoritmom *first fit*. Ako takav blok ne nađe, ona alocira nov logički segment veličine `BLOCK_SIZE` ili tražene veličine `size` (šta god je veće) u virtuelnom adresnom prostoru sistemskim pozivom `mmap` i dodaje ga u svoju evidenciju. Ukoliko bi nakon alokacije ostao slobodan fragment veličine manje od `MIN_BLOCK_SIZE`, alocira se ceo slobodan blok (drugim rečima, ne ostavlja se slobodan fragment manji od `MIN_BLOCK_SIZE`).

Da bi funkcija `free` za oslobađanje memorijskog bloka znala njegovu veličinu, uz alociran blok ostaje zaglavlje, s tim da funkcija `malloc` vraća pokazivač na slobodan blok ispred koga je zaglavlje, a ne na zaglavlje.

```
void* malloc (size_t size);
struct BlockHeader {
  BlockHeader* next;
  size_t size;
};
static BlockHeader* freeMemHead;
const size_t BLOCK_SIZE = ..., MIN_BLOCK_SIZE = ...;
void* mmap (void* addr, size_t size, int protection);
const int PROT_RD = ..., PROT_WR = ..., PROT_RW = PROT_RD | PROT_WR;
```

Rešenje

```
void* malloc (size_t size) {

  // Try to find an existing free block in the list (first fit):
  BlockHeader *blk = freeMemHead, *prev = nullptr;
  for (; blk!=nullptr; prev=blk, blk=blk->next)
    if (blk->size>=size) break;

  // If not found, allocate a new memory segment and add it to the list:
  if (blk == nullptr) {
    size_t sz = max(size,BLOCK_SIZE);
    blk = (BlockHeader*)mmap(nullptr,sz+sizeof(BlockHeader),PROT_RW);
    if (blk == nullptr) return nullptr;  // No free memory
    if (prev) prev->next = blk;
    else freeMemHead = blk;
    blk->next = nullptr;
    blk->size = sz;
  }

  // Allocate the requested block:
  size_t remainingSize = blk->size - size;
  if (remainingSize >= sizeof(BlockHeader) + MIN_BLOCK_SIZE) {
    // A fragment remains
    blk->size = size;
    size_t offset = sizeof(BlockHeader) + size;
    BlockHeader* newBlk = (BlockHeader*)((char*)blk + offset);
    if (prev) prev->next = newBlk;
    else freeMemHead = newBlk;
```

```
    newBlk->next = blk->next;
    newBlk->size = remainingSize - sizeof(BlockHeader);
  } else {
    // No remaining fragment, allocate the entire block
    if (prev) prev->next = blk->next;
    else freeMemHead = blk->next;
  }
  blk->next = nullptr;
  return (char*)blk + sizeof(BlockHeader);
}
```

2. zadatak, drugi kolokvijum, jun 2019.

Neki operativni sistem primenjuje kontinualnu alokaciju memorije i nudi sistemski poziv kojim proces može da zatraži „skupljanje“ prostora koji zauzima, tj. oslobađanje vršnog dela svog prostora koji je do sada zauzimao. Implementirati operaciju

```
int shrink (PCB* pcb, size_t by);
```

koja se koristi u implementaciji ovog sistemskog poziva i koja treba da smanji alociran memorijski prostor procesa na čiji PCB ukazuje prvi argument za veličinu datu drugim argumentom. Pri tom, ovu priliku sistem po potrebi koristi i da izvrši lokalnu defragmentaciju na sledeći način: ako je neposredno ispred memorije koju proces zauzima već postojao slobodan fragment, a kako je njegovim „skupljanjem“ iza njega sigurno nastao slobodan fragment, proces se relocira na početak slobodnog fragmenta ispred njega, kako bi se ova dva slobodna fragmenta spojila u jedan. U slučaju uspeha, ova funkcija treba da vrati 0, a u slučaju neke greške -1.

U strukturi PCB postoje, između ostalog, i sledeća polja:

- `char* base`: početna (bazna) fizička adresa procesa u memoriji;
- `size_t size`: veličina memorijskog prostora koji proces trenutno zauzima.

Na raspolaganju su i sledeće funkcije:

- `mem_free(void* at, size_t sz)`: oslobađa prostor na adresi datoj prvim argumentom i veličine date drugim argumentom, uz spajanje sa slobodnim fragmenatima ispred i iza novonastalog po potrebi;
- `relocate(PCB* pcb, void* to)`: relocira dati proces na novo zadato mesto, uz svo neophodno ažuriranje strukture za evidenciju slobodne memorije.

Slobodni fragmenti dvostruko su ulančani u listu na čiji prvi element ukazuje `fmem_head`. Fragmenti su ulančani u listu po rastućem redosledu svojih početnih adresa. Svaki slobodni fragment predstavljen je strukturom `FreeMem` koja je smeštena na sam početak tog slobodnog fragmenta:

```
struct FreeMem {
  FreeMem* next; // Next in the list
  FreeMem* prev; // Previous in the list
  size_t size;   // Size of the free fragment
};
```

Rešenje

```
int shrink (PCB* pcb, size_t by) {
  if (pcb==0 || by>=pcb->size) return -1; // Exception
  if (by==0) return 0; // Nothing to do
  pcb->size -= by;
  mem_free(pcb->base+pcb->size,by);
  FreeMem* above = (FreeMem*)(pcb->base+pcb->size);
  FreeMem* under = above->prev;
  if (under && ((char*)under+under->size == pcb->base)
    relocate(pcb,under);
  return 0;
}
```

2. zadatak, drugi kolokvijum, april 2018.

U nekom sistemu koristi se kontinualna alokacija memorije. U PCB postoje sledeća polja:

- `char* mem_base_addr`: početna adresa u memoriji na koju je smešten proces;
- `size_t mem_size`: veličina dela memorije koju zauzima proces u jedinicama `sizeof(char)`.

Fragmenti slobodne memorije ulančani su u jednostruko ulančanu listu, uređenu po rastućem poretku adresa slobodnih fragmenata, pri čemu se svaki element te liste, tipa `FreeSegment`, smešta na sam početak slobodnog fragmenta. Glava ove liste je u globalnoj promenljivoj `mem_free_head`. Struktura `FreeSegment` izgleda ovako:

```
struct FreeSegment {
  FreeSegment* next;  // Next free segment in the list
  size_t size; // Total size of the free segment in sizeof(char)
};
```

Napisati funkciju `proc_relocate()` koja vrši relokaciju datog procesa sa ciljem obavljanja lokalizovane, ograničene kompakcije na sledeći način. Ako i samo ako i ispred i iza datog procesa u memoriji postoji slobodan fragment, onda se taj proces pomera na početak slobodnog fragmenta ispred njega, kako bi se slobodan prostor ta dva fragmenta spojio u jedan, iza tog procesa. Ukoliko je ova kompakcija urađena, funkcija vraća 1, inače vraća 0.

```
int proc_relocate (PCB* pcb);
```

Rešenje

```
int proc_relocate (PCB* pcb) {
  FreeMem *cur=fmem_head, *prev=0;
  for (; cur && (char*)cur+cur->size!=pcb->mem_base_addr;
         prev=cur, cur=cur->next);

  if (cur==0 || cur->next==0 ||
      (char*)cur->next!=pcb->mem_base_addr+pcb->size)
    // No room or no need for relocation
    return 0;

  // Relocate the process to cur
  // and join the two fragments (cur and cur->next):

  size_t new_size = cur->size + cur->next->size;
  FreeMem* new_next = cur->next->next;

  memcpy(cur,pcb->mem_base_addr,pcb->mem_size);
  pcb->mem_base_addr = cur;

  if (prev) {
    prev->next = (FreeMem*)(pcb->mem_base_addr+pcb->mem_size);
    prev->next->next = new_next;
    prev->next->size = new_size;
  } else {
    fmem_head = (FreeMem*)(pcb->mem_base_addr+pcb->mem_size);
    fmem_head->next = new_next;
    fmem_head->size = new_size;
  }

  return 1;
}
```

2. zadatak, drugi kolokvijum, jun 2018.

Neki sistem primenjuje kontinualnu alokaciju memorije sa *best-fit* algoritmom. Zapisi o slobodnim delovima memorije organizovani su u ulančanu listu uređenu neopadajuće po veličini slobodnih delova memorije. U svakom zapisu je informacija o adresi početka slobodnog dela memorije i njegovoj veličini. Dat je sadržaj ove liste u nekom trenutku (sve vrednosti su heksadecimalne).

Zapis broj	Adresa početka	Veličina
1	3510	20
2	3680	30
3	3470	50
4	35A0	90

1. Prikazati tu listu nakon alokacije dela memorije veličine 40h.
2. Prikazati tu listu nakon što se, posle akcije pod a), izvrši oslobađanje dela memorije veličine 70h, na adresi 3530h.
3. Prikazati tu listu nakon što se, posle akcije pod 2), izvrši kompakcija slobodnog prostora pomeranjem alociranih delova na sam početak prvog slobodnog dela. Iza poslednjeg slobodnog dela nalazi se memorijski prostor koji se ne koristi u ovoj alokaciji.

Rešenje

1.

Zapis broj	Adresa početka	Veličina
1	34B0	10
2	3510	20
3	3680	30
4	35A0	90

2.

Zapis broj	Adresa početka	Veličina
1	34B0	10
2	3680	30
3	3510	120

3.

Zapis broj	Adresa početka	Veličina
1	3550	160

2. zadatak, drugi kolokvijum, april 2017.

Neki operativni sistem primenjuje kontinualnu alokaciju memorije i nudi sistemski poziv kojim proces može da zatraži proširenje prostora koji zauzima. Implementirati operaciju

```
int extend (PCB* pcb, size_t by);
```

koja se koristi u implementaciji ovog sistemskog poziva i koja treba da proširi memorijski prostor procesa na čiji PCB ukazuje prvi argument za veličinu datu drugim argumentom. Ova funkcija treba da pokuša proširenje prostora procesa najpre bez njegove relokacije, ako je moguće (ukoliko je prostor iza njega slobodan i dovoljne je veličine), a potom i uz relokaciju, ako je to neophodno i moguće. U slučaju uspeha, ova funkcija treba da vrati 0, a u slučaju neuspeha -1.

U strukturi PCB postoje, između ostalog, i sledeća polja:

- `void* base`: početna (bazna) fizička adresa procesa u memoriji;
- `size_t size`: veličina memorijskog prostora koji proces trenutno zauzima.

Na raspolaganju su i sledeće funkcije:

- `mem_allocate_at(size_t sz, void* at)`: alocira prostor veličine date prvim argumentom na (fizičkoj) adresi datoj drugim argumentom; u slučaju uspeha vraća 0, a u slučaju neuspeha (na datoj adresi nije slobodan prostor potrebne veličine) vraća -1;
- `mem_allocate(size_t sz, void** location)`: alocira prostor veličine date prvim argumentom gde god je to moguće; u slučaju uspeha vraća 0, a adresu alociranog prostora smešta u pokazivač na koji ukazuje drugi argument; u slučaju neuspeha (nema slobodnog prostora potrebne veličine) vraća -1;
- `mem_free(void* at, size_t sz)`: oslobađa prostor na adresi datoj prvim argumentom i veličine date drugim argumentom;
- `mem_copy(void* from, void* to, size_t sz)`: kopira memorijski sadržaj veličine date trećim argumentom sa prve na drugu adresu.

Rešenje

```
int extend (PCB* pcb, size_t by) {
  if (pcb==0) return -1; // Exception
  if (mem_allocate_at(by,pcb->base+pcb->size)) {
     // Extension without relocation
     pcb->size+=by;
     return 0;
  }
  void* newLocation;
  if (mem_allocate(pcb->size+by,&newLocation)) {
     // Extension with relocation
     mem_copy(pcb->base,newLocation,pcb->size);
     mem_free(pcb->base,pcb->size);
     pcb->base = newLocation;
     pcb->size+=by;
     return 0;
  }
  return -1; // No free memory, cannot extend
}
```

2. zadatak, drugi kolokvijum, maj 2015.

U nekom sistemu koristi se kontinualna alokacija memorije. U PCB postoje sledeća polja:

- `char* mem_base_addr`: početna adresa u memoriji na koju je smešten proces;
- `size_t mem_size`: veličina dela memorije koju zauzima proces;
- `PCB* mem_next`: pokazivač na sledeći u listi procesa; PCB procesa su ulančani u ovu listu po redosledu njihovog smeštanja u memoriji (po rastućem redosledu `mem_base_addr`); glava ove liste je globalna promenljiva `proc_mem_head`.

Na početak dela operativne memorije koja se koristi za smeštanje procesa ukazuje pokazivač `user_proc_mem_start`, a na njegov kraj (na poslednji znak) pokazivač `user_proc_mem_end`; oba su tipa `char*`.

Fragmenti slobodne memorije ulančani su u jednostruko ulančanu listu, pri čemu se svaki element te liste, tipa `FreeSegment`, smešta na sam početak slobodnog fragmenta. Glava ove liste je u globalnoj promenljivoj `mem_free_head`. Struktura `FreeSegment` izgleda ovako:

```
struct FreeSegment {
  FreeSegment* next;  // Next free segment in the list
  size_t size; // Total size of the free segment
};
```

Napisati proceduru `mem_compact()` koja vrši kompakciju slobodnog prostora relokacijom procesa.

Rešenje

```
void proc_relocate (PCB* pcb, char* to) {
  if (pcb->mem_base_addr==to) return;
```

```
  memcpy(to,pcb->mem_base_addr,pcb->mem_size);
  pcb->mem_base_addr = to;
}

void mem_compact () {
  if (mem_free_head==0) return; // No free memory, no need for compaction
  char* to = user_proc_mem_start;
  for (PCB* pcb=proc_mem_head; pcb!=0; pcb=pcb->mem_next) {
    proc_relocate(pcb,to);
    to+=pcb->mem_size;
  }
  size_t free_mem = user_proc_mem_end-to+1;
  if (free_mem>=sizeof(FreeSegment)) {
    mem_free_head = (FreeSegment*)to;
    mem_free_head->size = free_mem;
    mem_free_head->next = 0;
  }
  else mem_free_head = 0; // No more free memory (should not ever happen)
}
```

2. zadatak, drugi kolokvijum, septembar 2015.

Neki sistem primenjuje kontinualnu alokaciju memorije. Slobodni fragmenti dvostruko su ulančani u listu na čiji prvi element ukazuje `fmem_head`. Fragmenti su ulančani u listu po rastućem redosledu svojih početnih adresa. Svaki slobodni fragment predstavljen je strukturom `FreeMem` koja je smeštena na sam početak tog slobodnog fragmenta:

```
struct FreeMem {
  FreeMem* next; // Next in the list
  FreeMem* prev; // Previous in the list
  size_t size;   // Size of the free fragment
};
```

Implementirati funkciju

```
int mem_free(char* address, size_t size);
```

Ova funkcija treba da dealocira zauzeti kontinualni segment memorije na datoj adresi i date veličine, uz eventualno (po potrebi) spajanje novooslobođenog fragmenta sa susednim slobodnim fragmentima ispred i iza njega.

Rešenje

Jednostavnije, ali manje efikasno rešenje:

```
// Helper: Try to join cur with the cur->next free segment:
int tryToJoin (FreeMem* cur) {
  if (!cur) return 0;
  if (cur->next && (char*)cur+cur->size == (char*)(cur->next)) {
    // Remove the cur->next segment:
    cur->size += cur->next->size;
    cur->next = cur->next->next;
    if (cur->next) cur->next->prev = cur;
    return 1;
  } else
    return 0;
}

int mem_free(char* addr, size_t size) {
  // Find the place where to insert the new free segment (just after cur):
  FreeMem* cur=0;
```

```
  if (!fmem_head || addr<(char*)fmem_head)
    cur = 0; // insert as the first
  else
    for (cur=fmem_head; cur->next!=0 && addr>(char*)(cur->next);
         cur=cur->next);

  // Insert the new segment after cur:
  FreeMem* newSeg = (FreeMem*)addr;
  newSeg->size = size;
  newSeg->prev = cur;
  if (cur) newSeg->next = cur->next;
  else newSeg->next = fmem_head;
  if (newSeg->next) newSeg->next->prev = newSeg;
  if (cur) cur->next = newSeg;
  else fmem_head = newSeg;

  // Try to merge with the previous and next segments:
  tryToJoin(newSeg);
  tryToJoin(cur);
}
```

Složenije, ali nešto efikasnije rešenje:

```
int mem_free(char* addr, size_t size) {
  // Find the place where to insert the new free segment (just after cur):
  FreeMem* cur=0;
  if (!fmem_head || addr<(char*)fmem_head)
    cur = 0; // insert as the first
  else
    for (cur=fmem_head; cur->next!=0 && addr>(char*)(cur->next);
         cur=cur->next);

  // Try to append it to the previous free segment cur:
  if (cur && (char*)cur+cur->size==addr) {
    cur->size+=size;
    // Try to join cur with the next free segment:
    if (cur->next && (char*)cur+cur->size == (char*)(cur->next)) {
      // Remove the cur->next segment:
      cur->size += cur->next->size;
      cur->next = cur->next->next;
      if (cur->next) cur->next->prev = cur;
    }
    return;
  }
  else

  // Try to append it to the next free segment:
  FreeMem* nxtSeg = cur?cur->next:fmem_head;
  if (nxtSeg && addr+size==(char*)nxtSeg) {
    FreeMem* newSeg = (FreeMem*)addr;
    newSeg->size = nxtSeg->size+size;
    newSeg->prev=nxtSeg->prev;
    newSeg->next=nxtSeg->next;
    if (nxtSeg->next) nxtSeg->next->prev = newSeg;
    if (nxtSeg->prev) nxtSeg->prev->next = newSeg;
    else fmem_head = newSeg;
    return;
```

```
  }
  else

  // No need to join; insert the new segment after cur:
  FreeMem* newSeg = (FreeMem*)addr;
  newSeg->size = size;
  newSeg->prev = cur;
  if (cur) newSeg->next = cur->next;
  else newSeg->next = fmem_head;
  if (newSeg->next) newSeg->next->prev = newSeg;
  if (cur) cur->next = newSeg;
  else fmem_head = newSeg;
}
```

2. zadatak, drugi kolokvijum, april 2014.

Neki sistem primenjuje kontinualnu alokaciju memorije. Slobodni fragmenti dvostruko su ulančani u listu na čiji prvi element ukazuje `fmem_head`. Svaki slobodni fragment predstavljen je strukturom `FreeMem` koja je smeštena na sam početak tog slobodnog fragmenta:

```
struct FreeMem {
  FreeMem* next; // Next in the list
  FreeMem* prev; // Previous in the list
  size_t size;   // Size of the free fragment
};
```

Implementirati funkciju

```
int mem_alloc(void* address, size_t by);
```

Ova funkcija pokušava da pronađe slobodan fragment na adresi datoj prvim argumentom i da iz njega alocira deo veličine date drugim argumentom. Ukoliko na datoj adresi ne počinje slobodan fragment ili on nije dovoljne veličine, ova funkcija vraća -1. Ukoliko na datoj adresi postoji slobodan fragment dovoljne veličine, ova funkcija ažurira listu slobodnih framenata na sledeći način:

- ukoliko bi iza alociranog dela preostao fragment koji je manji ili jednak veličini strukture `FreeMem`, ceo taj slobodni fragment se alocira (ne ostavlja se ostatak koji je premali za evidenciju) i izbacuje iz liste slobodnih;
- u suprotnom, preostali deo se ostavlja kao slobodan fragment.

U oba slučaja, funkcija vraća veličinu stvarno alociranog dela (veće ili jednako traženoj veličini).

Rešenje

```
int mem_alloc(void* addr, size_t size) {
  for (FreeMem* cur=fmem_head; cur!=0; cur=cur->next) {
    if (cur!=addr || cur->size<size) continue;
    // Found
    if (cur->size-size<=sizeof(FreeMem)) {
      // No remaining fragment
      if (cur->prev) cur->prev->next = cur->next;
      else fmem_head = cur->next;
      if (cur->next) cur->next->prev = cur->prev;
      return cur->size;
    else {
      FreeMem* newfrgm = (FreeMem*)((char*)cur+size);
      if (cur->prev) cur->prev->next = newfrgm;
      else fmem_head = newfrgm;
      if (cur->next) cur->next->prev = newfrgm;
      newfrgm->prev = cur->prev;
      newfrgm->next = cur->next;
```

```
      newfrgm->size = cur->size-size;
      return size;
    }
  }
  return -1;
}
```

3. zadatak, drugi kolokvijum, april 2013.

Neki sistem primenjuje kontinualnu alokaciju memorije za procese. U strukturi PCB procesa postoje sledeća polja:

- `Word* mem_base`: pokazivač na lokaciju u memoriji na kojoj je proces smešten; tip `Word` predstavlja jednu adresibilnu jedinicu memorije;
- `size_t mem_size`: veličina memorijskog segmenta koju zauzima proces; `size_t` je celobrojni tip za izražavanje veličina memorijskih segmenata, u jedinicama `Word`.

Sistemskim pozivom `mem_extend` proces može da traži povećanje svog memorijskog prostora za traženu veličinu (proširenje iza kraja). Sistem će ispuniti ovaj zahtev ukoliko iza prostora koji proces već zauzima postoji slobodan segment dovoljne veličine. U suprotnom, odbiće taj zahtev (ne radi relokaciju procesa).

Slobodni segmenti ulančani su u listu, a funkcija

```
int mem_alloc(Word* address, size_t by);
```

pokušava da pronađe slobodan segment na adresi datoj prvim argumentom i da iz njega alocira deo veličine date drugim argumentom. U slučaju uspeha, funkcija vraća 0. Ukoliko na datoj adresi ne počinje slobodan segment ili on nije dovoljne veličine, ova funkcija vraća negativnu vrednost (kod greške).

Korišćenjem već implementirane funkcije `mem_alloc`, implementirati funkciju

```
int mem_extend (PCB* p, size_t by);
```

koja se koristi u implementaciji opisanog sistemskog poziva `mem_extend`. Ova funkcija treba da vrati 0 u slučaju uspeha, a negativnu vrednost u slučaju greške.

Rešenje

```
int mem_extend (PCB* p, size_t by) {
    // Error: invalid argument
    if (p == 0 || by < 0) return -1;
    if (by == 0) return 0;
    Word* tail = p->mem_base + p->mem_size;
    // Error: allocation failed
    if (mem_alloc(tail, by) < 0) return -2;
    // Extend
    p->mem_size += by;
    return 0;
}
```

3. zadatak, drugi kolokvijum, septembar 2012.

Neki sistem primenjuje kontinualnu alokaciju memorije sa *best-fit* algoritmom. Zapisi o slobodnim delovima memorije organizovani su u ulančanu listu uređenu neopadajuće po veličini slobodnih delova memorije. U svakom zapisu je informacija o adresi početka slobodnog dela memorije i njegovoj veličini. Dat je sadržaj ove liste u nekom trenutku (sve vrednosti su heksadecimalne). Prikazati tu listu nakon oslobađanja dela memorije veličine 70h, na adresi 2520h.

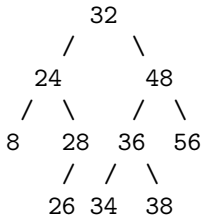
Zapis broj	Adresa početka	Veličina
1	2500	20
2	2670	30
3	2460	40
4	2590	90

Rešenje

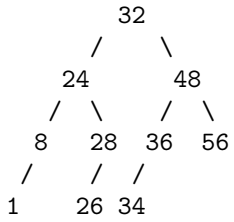
Zapis broj	Adresa početka	Veličina
1	2670	30
2	2460	40
3	2500	120

4. zadatak, drugi kolokvijum, septembar 2011.

Neki sistem primenjuje kontinualnu alokaciju memorije sa *best-fit* algoritmom. Radi efikasnijeg pronalaženja najpovoljnijeg dela memorije, sistem održava binarno stablo slobodnih fragmenata memorije, pri čemu svaki čvor ovog binarnog stabla predstavlja jedan slobodni fragment, u levom podstablu su manji, a u desnom veći fragmenti. Na slici je prikazano stanje ovog stabla u datom trenutku (u čvor je upisana veličina fragmenta u jedinicama alokacije). Prikazati stanje ovog stabla nakon alokacije memorije veličine 37 jedinica.



Rešenje



5. zadatak, drugi kolokvijum, maj 2010.

U nekom sistemu primenjuje se kontinualna alokacija operativne memorije. Deo definicije strukture PCB je sledeći:

```
struct PCB {
  ...
  void* memLocation;  // Current place in memory
  size_t memSize;  // Size of memory space
};
```

Implementirati operaciju:

```
void relocate(PCB* process, void* newPlace);
```

kojom sistem premešta memorijski prostor procesa sa tekućeg na novozadato (već alocirano) mesto u memoriji.

U datom sistemu, operacija

```
void free (void* addr, size_t size);
```

dealocira (proglašava slobodnim) memorijski prostor veličine `size` počev od adrese `addr`.

Pomoć: Standardna bibliotečna funkcija `memcpy` ima sledeću deklaraciju:

```
void* memcpy (void* destination, const void* source, size_t size);
```

Rešenje

```
void relocate(PCB* p, void* newPlace) {
  if (p==0) return; // Exception!
```

```
  if (newPlace==p->memLocation || p->memSize==0) return; // Nothing to do
  memcpy(newPlace,p->memLocation,p->memSize); // Move memory contents
  free(p->memLocation,p->memSize); // Free the old memory space
  p ->memLocation=newPlace;  // Move relocation register
}
```

5. zadatak, drugi kolokvijum, maj 2009.

U nekom sistemu primenjuje se kontinualna alokacija operativne memorije, uz primenu algoritma *first fit* i alokaciju u jedinicama veličine 1KB. Ukupan prostor za alokaciju memorije za korisničke procese je veličine 128KB. Posmatra se sledeća sekvenca zahteva za alokacijom i dealokacijom memorije:

$A+45, B+23, C+38, B-, D+16, E+16, F+5, C-, G+25$

U oznaci svakog zahteva prvo slovo označava proces koji izdaje zahtev, simbol „+“ označava operaciju alokacije memorije, simbol „–„ označava operaciju dealokacije memorije, dok broj označava veličinu memorije koja se alocira izraženu u KB.

1. Popuniti sledeću tabelu vrednostima koje opisuju stanje zauzetosti memorije nakon ove sekvence:

Proces	A	D	E	F	G
Adresa početka ($\cdot 2^{10}$)					

2. Broj slobodnih fragmenata je _____ .

   Ukupna veličina slobodne memorije je _____ .

   Veličina najvećeg slobodnog fragmenta je _____ .

   Veličina najmanjeg slobodnog fragmenta je _____ .

Rešenje

1.

Proces	A	D	E	F	G
Adresa početka ($\cdot 2^{10}$)	0	45	106	61	66

2. Broj slobodnih fragmenata je 2.

   Ukupna veličina slobodne memorije je 21 KB.

   Veličina najvećeg slobodnog fragmenta je 15 KB.

   Veličina najmanjeg slobodnog fragmenta je 6 KB.

5. zadatak, drugi kolokvijum, maj 2007.

U nekom operativnom sistemu primenjuje se kontinualna alokacija operativne memorije. Alokator održava spisak slobodnih delova memorije kao ulančanu listu uređenih parova (adresa početka slobodnog dela, veličina slobodnog dela). U nekom trenutku lista ima sledeće elemente redom (sve vrednosti su heksadecimalne):

`(7F1A, 24), (1FFC, 42), (A000, 20), (0770, 7A), (4010, 68), (3A0A, A0), ( 01B0, 30)`

Koji deo će biti alociran ako se traži alokacija prostora veličine 47h (navesti samo adresu slobodnog dela koji je odabran), ako sistem primenjuje sledeći algoritam alokacije:

1. *first-fit*
2. *best-fit*
3. *worst-fit*.

Rešenje

1. 0770h
2. 4010h
3. 3A0Ah

Segmentna organizacija

2. zadatak, prvi kolokvijum, mart 2023.

U nekom sistemu sa segmentnom organizacijom memorije adresibilna jedinica je bajt, virtuelni adresni prostor je veličine 1 MB, a maksimalna veličina fizičkog segmenta je 4 KB. Dat je spisak početnih (virtuelnih) adresa, veličina (obe vrednosti su zapisane heksadecimalno) i vrsta logičkih segmenata (regiona) koje je alocirao neki proces.

Adresa segmenta (hex)	Veličina (hex)	Vrsta segmenta
0	2890	instrukcije
3000	FF0	konstantni podaci incijalizovani statički
4000	3E68	instrukcije
28000	189A	promenljivi podaci
FE000	2000	stek

1. U datu tabelu upisati parametre svih fizičkih segmenata koje je operativni sistem organizovao u SMT (sve ulaze u SMT koji nisu *null*) za ovaj proces, po rastućem redosledu broja segmenta. Broj segmenta i granicu segmenta (*limit*, najvišu dozvoljenu vrednost pomeraja u segmentu koji se sme adresirati) zapisati heksadecimalno, a bite prava pristupa *RWX* binarno tim redom. (Broj redova u datoj tabeli ne mora da odgovara broju segmenata koje treba upisati; eventualni višak redova date tabele ostaviti prazne.)

2. Ako stek raste ka nižim adresama, koja virtuelna adresa je prva koja je van dozvoljenog opsega i nije dozvoljena za adresiranje ako stek prekorači svoj dozvoljeni kapacitet?

   Odgovor (hex): _____

   Kom fizičkom segmentu pripada ta adresa?

   Odgovor (hex): _____

Segment # (hex)	Limit (hex)	RWX (bin)

Slika 1: Tabela data uz stavku 1 drugog zadatka.

Rešenje

1.

Segment # (hex)	Limit (hex)	RWX (bin)
0	FFF	001
1	FFF	001
2	88F	001
3	FEF	100
4	FFF	001
5	FFF	001
6	FFF	001
7	E67	001
28	FFF	110
29	899	110
FE	FFF	110
FF	FFF	110

2. Adresa FDFFFh pripada segmentu broj FDh.

1. zadatak, prvi kolokvijum, mart 2022.

Neki sistem vodi evidenciju alociranih (deklarisanih) logičkih segmenata (regiona) za svaki proces u jednostruko ulančanoj listi elemenata tipa `RegionDesc` uređenih po početnoj virtuelnoj adresi segmenta (polje `addr`). Svaki logički segment opisuje jedan deskriptor tipa `RegionDesc` u kom polje `size` sadrži veličinu segmenta.

```
struct RegionDesc {
    byte* addr;
    size_t size;
    RegionDesc* next;
};

int createRegion(RegionDesc** phead, byte* addr, size_t sz);
```

Implementirati funkciju `createRegion` koja se koristi u implementaciji sistemskog poziva `mmap`. Ova funkcija treba da napravi nov deskriptor logičkog segmenta koji se deklariše na zadatoj adresi (`addr`) tražene veličine (`sz`) i uključi ga u evidenciju segmenata procesa, ali samo ako na traženom mestu u virtuelnom adresnom prostoru procesa ima dovoljno prostora između već alociranih segmenata, odnosno ako se traženi segment ne preklapa ni sa jednim već deklarisanim. Ukoliko to nije slučaj, ova funkcija treba da vrati grešku (negativnu vrednost). Najveća adresa u virtuelnom adresnom prostoru je `MAX_VADDR`. Argument phead je pokazivač na glavu liste deskriptora segmenata datog procesa. Tip `byte` predstavlja celobrojni tip veličine jedne adresibilne jedinice. Pretpostaviti da su argumenti `addr` i `size` po potrebi već poravnati na stranice. Dinamičku alokaciju prostora za potrebe struktura jezgra radi funkcija `kmalloc` koja ima isti potpis i dejstvo kao standardna C funkcija `malloc`.

Rešenje

```
int createRegion(RegionDesc** phead, byte* addr, size_t sz) {
    if (!phead || !sz || addr + sz - 1 > MAX_VADDR) {
        return -1;
    }
    RegionDesc *prev = nullptr, *next = *phead;
    while (next && addr >= next->addr) {
        prev = next;
        next = next->next;
    }
    if (prev && prev->addr + prev->size > addr) {
        return -1;
    }
    if (next && addr + sz > next->addr) {
        return -1;
    }
```

```
    RegionDesc* dsc = (RegionDesc*) kmalloc(sizeof(RegionDesc));
    if (!dsc) {
        return -2;
    }
    dsc->addr = addr;
    dsc->size=sz;
    dsg->next = next;
    if (prev) {
        prev->next = dsc;
    } else {
        *phead = desc;
    }
    return 0;
}
```

3. zadatak, prvi kolokvijum, mart 2022.

Neki sistem koristi segmentnu organizaciju memorije sa fizičkim segmentom maksimalne veličine 128 KB. Adresibilna jedinica je 16-bitna reč, virtuelni adresni prostor je veličine 8 GB, a fizički adresni prostor je veličine 1 GB. U deskriptoru segmenta u SMT najviša tri bita predstavljaju prava pristupa (RWX), u bitima do njih je maksimalna dozvoljena adresa unutar segmenta (*limit*), dok je u najnižim bitima bazna fizička adresa segmenta; ako segment nije alociran, ceo deskriptor ima vrednost 0.

1. Prikazati logičku strukturu virtuelne adrese i označiti širinu svih polja.
2. Implementirati funkciju `shareSeg` koja se koristi u implementaciji sistemskog poziva kojim se zahteva logičko deljenje segmenta broj `seg1` procesa čija je SMT data parametrom `smt1` sa segmentom `seg2` procesa čija je SMT data parametrom `smt2`, tako da ovaj drugi već postoji, a ovaj prvi postaje deljen sa drugim. Ukoliko drugi proces nije alocirao segment `seg2` ili je prvi proces već alocirao segment `seg1`, ova funkcija treba da vrati grešku (negativnu vrednost). Tip `long` je širine 64 bita, tip `int` 32 bita, a tip `short` 16 bita.

```
typedef unsigned long SMT[SMT_SIZE];
int shareSeg(SMT smt1, unsigned seg1, SMT smt2, unsigned seg2);
```

Rešenje

1. VA: Segment(16):Offset(16)
2.
```
int shareSeg(SMT smt1, unsigned seg1, SMT smt2, unsigned seg2) {
    if (smt1[seg1] || !smt2[seg2]) {
        return -1;
    }
    smt1[seg1] = smt2[seg2];
    return 0;
}
```

1. zadatak, prvi kolokvijum, maj 2022.

Neki sistem vodi evidenciju alociranih (deklarisanih) logičkih segmenata (regiona) za svaki proces u jednostruko ulančanoj listi elemenata tipa `RegionDesc` uređenih po početnoj virtuelnoj adresi segmenta (polje `addr`). Svaki logički segment opisuje jedan deskriptor tipa `RegionDesc` u kom polje `size` sadrži veličinu segmenta. Ova lista je uvek neprazna, jer kernel za svaki proces, prilikom njegovog pokretanja, alocira jedan logički segment na najnižim adresama virtuelnog adresnog prostora (počev od adrese 0) koji preslikava u svoj memorijski prostor.

```
struct RegionDesc {
    byte* addr;
    size_t size;
    RegionDesc* next;
};
void* createRegion(RegionDesc* phead, size_t sz);
```

Implementirati funkciju `createRegion` koja se koristi u implementaciji sistemskog poziva `mmap` sa parametrom `start_addr` jednakim `NULL`. Ova funkcija treba da alocira (deklariše) nov logički segment tražene veličine (`sz`) na bilo kom mestu u virtuelnom adresnom prostoru procesa u kom ima dovoljno prostora tražene veličine tako da se ne preklapa sa već alociranim segmentima i uključi ga u evidenciju segmenata procesa. U slučaju uspeha funkcija vraća početnu adresu alociranog segmenta, a u slučaju neuspeha vraća *null*. Najveća adresa u virtuelnom adresnom prostoru je `MAX_VADDR`. Argument `phead` je glava liste deskriptora segmenata datog procesa. Tip `byte` predstavlja celobrojni tip veličine jedne adresibilne jedinice. Ne treba poravnavati adrese i veličine segmenata. Dinamičku alokaciju prostora za potrebe struktura jezgra radi funkcija `kmalloc` koja ima isti potpis i dejstvo kao standardna C funkcija `malloc`.

Rešenje

```
void* createRegion(RegionDesc* phead, size_t sz) {
    if (!phead || !sz) {
        return nullptr;
    }
    RegionDesc* prev = phead;
    while (prev->next && prev->addr + prev->size + sz > prev->next->addr) {
        prev = prev->next;
    }
    byte* addr = prev->addr + prev->size;
    if (!prev->next && addr + sz - 1 > MAX_VADDR) {
        return nullptr;
    }
    RegionDesc* dsc = (RegionDesc*)kmalloc(sizeof(RegionDesc));
    if (!dsc) {
        return nullptr;
    }
    dsc->addr = addr;
    dsc->size=sz;
    dsg->next = prev->next;
    prev->next = dsc;
    return addr;
}
```

1. zadatak, kolokvijum, jun 2021.

Neki sistem ima segmentnu organizaciju memorije. Virtuelna adresa je 32-bitna, adresibilna jedinica je bajt, a maksimalna veličina fizičkog segmenta je 1 MB. Fizički adresni prostor je veličine 1 TB. Jedan ulaz u SMT-u sadrži prava pristupa *rwx* u najniža tri bita, granicu fizičkog segmenta (*limit* u opsegu od 0 do maksimalne veličine segmenta minus 1) u bitima do njih, a zatim fizičku adresu u bitima do njih; ovaj deskriptor (ulaz u SMT-u) zauzima najmanji potreban ceo broj bajtova.

Operativni sistem alocira segmente na zahtev, tako da pri kreiranju procesa ne alocira i ne učitava nijedan fizički segment. Vrednost 0 u polju za fizičku adresu u deskriptoru segmenta u SMT-u označava da preslikavanje nije moguće (segment nije alociran ili nije učitan).

Jedan alociran logički segment procesa opisan je deskriptorom tipa `SegDesc` u kom su, između ostalog, sledeća polja:

- `unsigned startAddr`: početna virtuelna adresa logičkog segmenta, svakako poravnata na početak fizičkog segmenta;
- `unsigned size`: veličina logičkog segmenta u bajtovima (može biti i veća od maksimalne veličine fizičkog segmenta);
- `unsigned short rwx`: prava pristupa za ceo logički segment u tri najniža bita.

Implementirati sledeću funkciju:

```
void initSegment (SegDesc* sd, unsigned long* smt);
```

Ovu funkciju poziva kod kernela kada inicijalizuje SMT novokreiranog procesa za svaki logički segment sa datim deskriptorom `sd`. Na već alociran SMT ukazuje `smt`. Veličine tipova su sledeće: `int` – 32 bita, `long` – 64 bita, `short` – 16 bita.

Rešenje

- VA(32): Segment(12). Offset(20)
- PA(40)

```
const unsigned SEG_WIDTH = 12, OFFS_WIDTH = 20,
               MAX_SEG_SIZE = 1U<<OFFS_WIDTH;
inline void setSMTEntry(unsigned long* smt,
                        unsigned seg,
                        unsigned limit,
                        unsigned long paddr,
                        unsigned short rwx) {
    smt[seg] = (paddr<<(OFFS_WIDTH+3)) | (limit<<3) | rwx;
}

void initSegment(SegDesc* sd, unsigned long* smt) {
    unsigned size = sd->size;
    unsigned saddr = sd->startAddr;
    while (size > 0) {
        setSMTEntry(smt, saddr>>OFFS_WIDTH,
                    (size>MAX_SEG_SIZE?MAX_SEG_SIZE-1:size-1), 0, sd->rwx);
        if (size < MAX_SEG_SIZE) break;
        size -= MAX_SEG_SIZE;
        saddr += MAX_SEG_SIZE;
    }
}
```

2. zadatak, kolokvijum, oktobar 2020.

Neki sistem koristi segmentnu organizaciju memorije. Slobodne fragmente operativne memorije sistem opisuje strukturama tipa `FreeSegDesc` za svaki proces. Kako bi pretraga pri alokaciji memorije bila efikasnija, ovi deskriptori organizovani su u uređeno binarno stablo, tako da su u levom podstablu svakog čvora deskriptori slobodnih fragmenata koji su manji ili jednaki, a desno onih koji su veći od fragmenta datog čvora. U strukturi `FreeSegDesc` polja `left` i `right` ukazuju na koren levog odnosno desnog podstabla, polje `addr` sadrži početnu adresu fragmenta, a polje `sz` sadrži veličinu fragmenta. Tip `size_t` je neoznačen celobrojni tip dovoljno velik da predstavi veličinu adresnog prostora.

```
struct FreeSegDesc {
  FreeSegDesc *left, *right;
  size_t sz;
  void* addr;
  ...
};
```

Implementirati operaciju koja pretragom u stablu na čiji koren ukazuje prvi parametar pronalazi i vraća deskriptor slobodnog fragmenta u koji se može alocirati segment zadate veličine, a *null* ako takvog nema, i to radi primenom *best fit* algoritma alokacije. Ova operacija ne treba da ažurira ni stablo ni pronađeni deskriptor, već samo da pronađe i vrati pokazivač na odgovarajući deskriptor.

```
SegDesc* findSegDesc (SegDesc* root, size_t size);
```

Rešenje

```
SegDesc* findSegDesc (SegDesc* root, size_t size) {
  SegDesc *sd = root, *bestFit = nullptr;
```

```
  while (sd) {
    if (sd->sz==size) return sd;
    else
    if (sd->sz<size) sd = sd->right;
    else {
      bestFit = sd;
      sd = sd->left;
    }
  }
  return bestFit;
}
```

3. zadatak, kolokvijum, septembar 2020.

Neki sistem koristi segmentnu organizaciju memorije. Slobodne fragmente operativne memorije sistem opisuje strukturama tipa `FreeSegDesc` za svaki proces. Kako bi pretraga pri alokaciji memorije bila efikasnija, ovi deskriptori organizovani su u uređeno binarno stablo, tako da su u levom podstablu svakog čvora deskriptori slobodnih fragmenata koji su manji ili jednaki, a desno onih koji su veći od fragmenta datog čvora. U strukturi `FreeSegDesc` polja `left` i `right` ukazuju na koren levog odnosno desnog podstabla, polje `addr` sadrži početnu adresu fragmenta, a polje `sz` sadrži veličinu fragmenta. Tip `size_t` je neoznačen celobrojni tip dovoljno velik da predstavi veličinu adresnog prostora.

```
struct FreeSegDesc {
  FreeSegDesc *left, *right;
  size_t addr, sz;
  ...
};
```

Implementirati operaciju koja pretragom u stablu na čiji koren ukazuje prvi parametar pronalazi i vraća deskriptor slobodnog fragmenta u koji se može alocirati segment zadate veličine, a *null* ako takvog nema, i to radi primenom *first fit* algoritma alokacije. Ova operacija ne treba da ažurira ni stablo ni pronađeni deskriptor, već samo da pronađe i vrati pokazivač na odgovarajući deskriptor.

```
SegDesc* findSegDesc (SegDesc* root, size_t size);
```

Rešenje

```
SegDesc* findSegDesc(SegDesc* root, size_t size) {
    SegDesc* sd = root;
    while (sd != nullptr) {
        if (sd->sz >= size) return sd;
        else sd = sd->right;
    }
    return nullptr;
}
```

2. zadatak, prvi kolokvijum, mart 2014.

Neki računar podržava segmentnu organizaciju virtuelne memorije, pri čemu je virtuelna adresa 16-bitna, fizički adresni prostor je veličine 4GB, a adresibilna jedinica je bajt. Maksimalna veličina segmenta je 4KB. U sledećoj tabeli dat je sadržaj nekoliko ulaza u tabeli preslikavanja nekog procesa (sve vrednosti su heksadecimnalne):

Segment#	Size	In memory?	Starting physical address
1	500	Yes	250000
2	600	Yes	AFF00
A	700	No	-
B	800	No	-
C	900	Yes	D6700

Popuniti sledeću tabelu na sledeći način: ako data virtuelna adresa uzrokuje grešku prekoračenja - napisati X, ukoliko izaziva straničnu grešku (*page fault*) - napisati P, a ukoliko je preslikavanje uspešno, upisati fizičku adresu u koju se preslikava (heksadecimalno).

Virtual address (hex)	Mapping result (hex)
12FA	
C0FF	
1675	
B014	
CDAB	

Rešenje

Virtual address (hex)	Mapping result (hex)
12FA	2502FA
C0FF	D67FF
1675	X
B014	P
CDAB	X

2. zadatak, prvi kolokvijum, april 2014.

Neki računar podržava segmentnu organizaciju virtuelne memorije, pri čemu je virtuelna adresa 20-bitna, fizički adresni prostor je veličine 16MB, a adresibilna jedinica je bajt. Maksimalna veličina segmenta je 4KB. U sledećoj tabeli dat je sadržaj nekoliko ulaza u tabeli preslikavanja nekog procesa (sve vrednosti su heksadecimnalne):

Segment#	Size	In memory?	Starting physical address
70	500	Yes	250000
12	600	Yes	AFF000
A0	700	No	-
B0	800	No	-
C0	900	Yes	D67000

Popuniti sledeću tabelu na sledeći način: ako data virtuelna adresa uzrokuje grešku prekoračenja - napisati X, ukoliko izaziva straničnu grešku (*page fault*) - napisati P, a ukoliko je preslikavanje uspešno, upisati fizičku adresu u koju se preslikava (heksadecimalno).

Virtual address (hex)	Mapping result (hex)
12FA0	
C00F0	
70750	
B0140	
C02AB	

Rešenje

Virtual address (hex)	Mapping result (hex)
12FA0	X
C00F0	D670F0
70750	X
B0140	P
C02AB	D672AB

2. zadatak, prvi kolokvijum, septembar 2011.

Neki sistem podržava segmentnu organizaciju virtuelne memorije. Virtuelni adresni prostor je veličine 16MB, maksimalna veličina segmenta je 256B, a adresibilna jedinica je bajt. Fizički adresni prostor je veličine 16MB. U tabeli preslikavanja segmenata (SMT) svaki ulaz zauzima samo onoliko prostora koliko je potrebno da se smesti početna adresa segmenta i stvarna veličina segmenta umanjena za 1 (najveći pomeraj koji se unutar segmenta sme

generisati), pošto se informacije o smeštaju segmenata na disku čuvaju u drugoj strukturi. Pri tome, vrednost 0 u polju za stvarnu veličinu segmenta u ulazu u SMT označava da dati segment nije dozvoljen za pristup, jer proces nije deklarisao da koristi taj deo adresnog prostora, a za segmente koje proces sme da koristi, vrednost 0 u polju za fizičku adresu segmenta označava da dati segment trenutno nije u fizičkoj memoriji. Naravno, nijedan segment ne sme početi pre fizičke adrese 1. Trenutno stanje SMT je sledeće (brojevi ulaza i vrednosti su dati heksadecimalno):

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	0245	01000140	0127	5400094C	0	CDD01227	34	402001A9	0

U donjoj tabeli date su virtuelne adrese koje se posmatraju nezavisno jedna od druge (ne u sekvenci, već se posmatra svaka za dato stanje SMT nezavisno od drugih, kao da druge nisu generisane). Popuniti donju tabelu za svaku od datih virtuelnih adresa (zapis je heksadecimalan) na sledeći način: ukoliko je pristup toj virtuelnoj adresi nedozvoljen, odnosno ako se generiše izuzetak zbog nelegalnog pristupa memoriji (engl. *memory access violation*), u polje upisati MAV; ako data adresa generiše izuzetak tipa stranične greške (engl. *page fault*), upisati PF; ako data adresa upada u zamku (engl. *trap*) usled prekoračenja veličine segmenta, upisati T; inače, upisati vrednost (pune) fizičke adrese u koju se ova preslikava, u heksadecimalnom zapisu.

Virtuelna adresa	222	FF32	4002D8	FE3A	FE14
Rezultat adresiranja					

Rešenje

Virtuelna adresa	222	FF32	4002D8	FE3A	FE14
Rezultat adresiranja	000023	PF	MAV	T	CDD026

3. zadatak, prvi kolokvijum, april 2009.

Neki sistem podržava segmentnu organizaciju virtuelne memorije. Virtuelni adresni prostor je veličine 4GB, maksimalna veličina segmenta je 4MB, a adresibilna jedinica je bajt. Fizički adresni prostor je veličine 64MB. U tabeli preslikavanja segmenata (SMT) svaki ulaz zauzima samo onoliko prostora koliko je potrebno da se smesti početna adresa segmenta i stvarna veličina segmenta umanjena za 1 (najveći pomeraj koji se unutar segmenta sme generisati), pošto se informacije o smeštaju stranica na disku čuvaju u drugoj strukturi. Pri tome, vrednost 0 u polju za stvarnu veličinu segmenta u ulazu u SMT označava da dati segment nije dozvoljen za pristup, jer proces nije deklarisao da koristi taj deo adresnog prostora, a za segmente koje proces sme da koristi, vrednost 0 u polju za fizičku adresu segmenta označava da dati segment trenutno nije u fizičkoj memoriji. Naravno, nijedan segment ne sme početi pre fizičke adrese 1. Trenutno stanje SMT je sledeće (brojevi ulaza i vrednosti su dati heksadecimalno):

Ulaz	0	1	2	3	...	FE	FF	100	...
Vrednost	0245	1000140	0127	5400094C	0	2CDD00027	034	43C002001A9	0

U donjoj tabeli date su virtuelne adrese koje se posmatraju nezavisno jedna od druge (ne u sekvenci, već se posmatra svaka za dato stanje SMT nezavisno od drugih, kao da druge nisu generisane). Popuniti donju tabelu za svaku od datih virtuelnih adresa (zapis je heksadecimalan) na sledeći način: ukoliko je pristup toj virtuelnoj adresi nedozvoljen, odnosno ako se generiše izuzetak zbog nelegalnog pristupa memoriji (engl. *memory access violation*), u polje upisati MAV; ako data adresa generiše izuzetak tipa stranične greške (engl. *page fault*), upisati PF; ako data adresa upada u zamku (engl. *trap*) usled prekoračenja veličine segmenta, upisati T; inače, upisati vrednost (pune) fizičke adrese u koju se ova preslikava, u heksadecimalnom zapisu.

Virtuelna adresa	80013A	EB28C32	3F80028D	403001E2	F0A001E4
Rezultat adresiranja					

Rešenje

Virtuelna adresa	80013A	EB28C32	3F80028D	403001E2	F0A001E4
Rezultat adresiranja	PF	MAV	DC4 (28D+B37)	T	MAV

4. zadatak, drugi kolokvijum, maj 2006.

Virtuelni adresni prostor sistema je 1GB, adresibilna jedinica je bajt, a virtuelni adresni prostor je organizovan segmentno. Svaki proces može imati najviše 64 segmenta. Fizički adresni prostor je veličine 512MB. Prostor za

zamenu (*swap space*) na disku je duplo veći od fizi čke memorije i koristi se za čuvanje zamenjenih segmenata kojima se direktno pristupa na disku (na „presnoj“ particiji, ne kroz fajl-sistem). Pristup segmentima se kontroliše pomoću tri bita zaštite koji se označavaju sa R (*read*), W (*write*) i E (*execute*). Kada se stranica nalazi u memoriji, prostor koji je zauzimala na disku se oslobađa za zamenu drugih stranica. Odgovoriti na sledeće pitanja i kratko, ali precizno obrazložiti odgovor: ako se deskriptori segmenata u tabeli preslikavanja segmenata (SMT) maksimalno kompaktno smeštaju, kolika je veličina SMT za jedan proces?

Rešenje

Virtuelna adresa (VA), 30 bita: Segment (6) : Offset(24)

Fizička adresa (PA), 29 bita

Swap 1GB $\implies$ Disk adr. (30)

Deskriptor stranice: Da li je u memoriji (1) : RWE(3) : Size(24) : Segment adr. (29) ili Disk Adr. (30)

Odatle sledi da je deskriptor segmenta veličine 58 bita, SMT ima po jedan ulaz za svaki segment, što znači 64 ulaza. Tako da je veličina SMTa $58 \cdot 64/8$ B = 464B

Napomena: Nije logično da se deskriptori pakuju tako “gusto”, tako da se jedan deskriptor rasprostire u više bajtova/reči, jer se njima pristupa prilikom preslikavanja adresa, što bi bilo neefikasno. Nije nemoguće, ali nije baš ni logično. Tako da je rešenje u kome se jedan deskriptor zaokruže na ceo broj bajtova/reči takođe prihvatljivo. Odnosno velivina deskriptora 64b=8B $\implies$ veličina SMT $8 \cdot 64$B = 512B. Na ovaj način je SMT veća za 10%, ali su zato performanse drastično bolje.

Segmentno-stranična organizacija

2. zadatak, prvi kolokvijum, maj 2022.

U nekom sistemu adresibilna jedinica je bajt, virtuelni adresni prostor je veličine 4 GB, a fizički adresni prostor je velične 1 GB. Sistem koristi segmentno-straničnu organizaciju memorije sa stranicom veličine 1 KB i segmentom maksimalne veličine 4 MB. Jedan ulaz u SMT zauzima dve 32-bitne reči; u nižoj je pokazivač (cela adresa) na PMT (ili 0 ako segment nije alociran), a u višoj reči su prava pristupa (RWX) u najniža tri bita i maksimalan dozvoljen broj stranice u segmentu (*limit*) u bitima do njih; preostali biti se ne koriste. Jedan ulaz u PMT zauzima jednu 32-bitnu reč (koriste se najniži biti).

Da bi obradio sistemske pozive u kojima se neki parametar zadaje kao pokazivač (virtuelna adresa u adresnom prostoru pozivajućeg procesa), operativni sistem mora da konvertuje datu virtuelnu u fizičku adresu, jer se kod kernela izvršava u režimu bez preslikavanja adresa. Implementirati funkciju koja obavlja ovu konverziju (vraća *null* ako virtuelna adresa ne pripada alociranom delu virtuelnog adresnog prostora). Prvi parametar ove funkcije je pokazivač na SMT pozivajućeg procesa. Pretpostaviti da je `uint32` deklarisan kao neoznačeni 32-bitni celobrojni tip. Napisati sve potrebne deklaracije za tipove SMT i PMT.

```
void* v2pAddr(SMT pmt, void* vaddr);
```

Rešenje

```
const uint32 offsetw = 10;
const uint32 pagew = 12;
const uint32 PMT_size = 4096;
const uint32 SMT_size = 1024;

typedef uint32 PMT[PMT_SIZE];
typedef uint32 SMT[SMT_SIZE][2];

void* v2pAddr(SMT smt, void* vaddr) {
    uint32 seg = (uint32)vaddr >> (pagew + offsetw);
    uint32 page = ((uint32)vaddr >> offsetw)) & ~((uint32)-1<<pagew);
    uint32 offset = (uint32)vaddr & ~((uint32)-1 << offsetw);
    uint32* pmt = (uint32*)smt[seg][0];
    if (!pmt) {
        return nullptr;
    }
    uint32 limit = smt[seg][1] >> 3;
    if (page > limit) {
        return nullptr;
    }
    uint32 frame = pmt[page];
    if (!frame) {
        return nullptr;
    }
    uint32 paddr = (frame << offsetw) + offset;
    return (void*)paddr;
}
```

3. zadatak, prvi kolokvijum, jun 2022.

Neki sistem ima segmentno-straničnu organizaciju memorije. Virtuelna adresa je 32-bitna, adresibilna jedinica je bajt, a maksimalna veličina fizičkog segmenta je 16 MB. Stranica je veličine 64 KB. Fizički adresni prostor je veličine 4 GB. Jedan ulaz u SMT-u zauzima dve 32- bitne reči; u prvoj (nižoj) su prava pristupa rwx u najviša tri bita, a granica fizičkog segmenta (*limit* u opsegu od 0 do maksimalno dozvoljenog broja stranice koji se sme adresirati) u najnižim bitima; druga (viša) reč ulaza u SMT-u sadrži 32-bitni pokazivač na PMT tog segmenta.

Operativni sistem alocira stranice na zahtev, tako da pri kreiranju procesa ne alocira nijednu stranicu. Vrednost 0 u

polju za pokazivač na PMT u deskriptoru segmenta u SMT-u, kao i u deskriptoru stranice u PMT-u označava da preslikavanje nije moguće.

Jedan alociran logički segment procesa opisan je deskriptorom tipa `SegDesc` u kom su, između ostalog, sledeća polja:

- `unsigned startAddr`: početna virtuelna adresa logičkog segmenta, svakako poravnata na početak fizičkog segmenta;
- `unsigned size`: veličina logičkog segmenta u bajtovima (može biti i veća od maksimalne veličine fizičkog segmenta);
- `unsigned short rwx`: prava pristupa za ceo logički segment u tri najniža bita.

Implementirati sledeću funkciju:

```
int initSegment(SegDesc* sd, unsigned long smt[][2]);
```

Ovu funkciju poziva kod kernela kada inicijalizuje SMT novokreiranog procesa za svaki logički segment sa datim deskriptorom `sd`. Na već alociran SMT ukazuje `smt`. Potrebno je inicijalizovati ulaze u SMT i napraviti pridružene PMT za sve korišćene fizičke segmente. Veličine tipova su sledeće: `int` – 32 bita, `long` – 64 bita, `short` – 16 bita.

Statička funkcija `PMT::alloc()` alocira prostor u memoriji kernela za smeštanje PMT jednog segmenta, inicijalizuje ceo taj PMT na 0 i vraća 32-pokazivač taj PMT. Kako bi se obezbedilo da bude dovoljno prostora za sve PMT-ove potrebne za sve fizičke segmente koje zauzima jedan logički segment, funkcija `initSegment` treba najpre da proveri da li ima dovoljno prostora za njih. Ovu proveru obavlja statička funkcija `PMT::reserve(int segs)` koja proverava da li ima dovoljno prostora za PMT-ove `segs` segmenata, rezerviše taj prostor (da bi narednih segs poziva `PMT::alloc` sigurno uspelo) i vraća 0 u slučaju uspeha, a negativnu vrednost u slučaju nedostatka prostora. Funkcija `initSegment` treba da vrati 0 u slučaju uspeha, a negativnu vrednost u suprotnom.

Rešenje

- VA(32): Segment(8).Page(8).Offset(16);
- PA: Frame(16).Offset(16)

```
const unsigned SEG_WIDTH = 8, PAGE_WIDTH = 8, OFFS_WIDTH = 16,
               MAX_SEG_SIZE = 1U << PAGE_WIDTH << OFFS_WIDTH,
               PAGE_SIZE = 1U << OFFS_WIDTH;

inline void setSMTEntry(unsigned smt[][2], unsigned seg, unsigned limit,
    unsigned short* pmt, unsigned short rwx) {
    smt[seg][0] = ((unsigned)rwx << 29) | limit;
    smt[seg][1] = (unsigned)pmt;
}
int initSegment(SegDesc* sd, unsigned smt[][2]) {
    unsigned size = sd->size;
    unsigned saddr = sd->startAddr;
    unsigned segs = (size + MAX_SEG_SIZE - 1) / MAX_SEG_SIZE;
    if (PMT::reserve(segs) < 0) {
        return -1;
    }
    while (size > 0) {
        unsigned short* pmt = PMT::alloc();
        unsigned limit = size > MAX_SEG_SIZE ? MAX_SEG_SIZE-1 : size-1;
        limit >>= OFFS_WIDTH;
        unsigned seg = saddr >> (PAGE_WIDTH + OFFS_WIDTH);
        setSMTEntry(smt, seg, limit, pmt, sd->rwx);
        if (size < MAX_SEG_SIZE) {
            break;
        }
        size -= MAX_SEG_SIZE;
        saddr += MAX_SEG_SIZE;
    }
```

```
    return 0;
}
```

2. zadatak, kolokvijum, avgust 2020.

Neki sistem koristi segmentno-straničnu organizaciju memorije. Logičke segmente koje je proces alocirao sistem opisuje strukturama tipa `SegDesc` za svaki proces. Kako bi se pretraga za logičkim segmentom kom odgovara adresirana stranica koja je generisala straničnu grešku što efikasnije izvršila, ovi deskriptori organizovani su u uređeno binarno stablo za svaki proces, tako da su u levom podstablu svakog čvora segmenti koji zauzimaju niže adrese, a desno oni koji zauzimaju više adrese od segmenta datog čvora. Logički segment je uvek poravnat i zaokružen na stranice. U strukturi `SegDesc` polja `left` i `right` ukazuju na koren levog odnosno desnog podstabla, polje `pg` sadrži broj prve stranice logičkog segmenta, a polje `sz` sadrži veličinu segmenta izraženu u broju stranica. Tip `size_t` je neoznačen celobrojni tip dovoljno velik da predstavi veličinu virtuelnog adresnog prostora.

```
struct SegDesc {
  SegDesc *left, *right;
  size_t pg, sz;
  ...
};
```

1. Implementirati operaciju koja pretragom u stablu na čiji koren ukazuje prvi parametar pronalazi i vraća deskriptor segmenta kom pripada data stranica, a *null* ako takvog nema:

   ```
   SegDesc* findSegDesc (SegDesc* root, size_t page);
   ```

   U strukturi PCB svakog procesa postoji polje `segdesc` tipa `SegDesc*` koje ukazuje na koren strukture stabla deskriptora logičkih segmenata tog procesa. U strukturi `SegDesc` postoji polje `vtp` (*virtual table pointer*) kao pokazivač na strukturu (tabelu) pokazivača na funkcije koje za dati tip logičkog segmenta implementiraju operacije koje kernel poziva u različitim situacijama. Ova struktura ima polje `loadPage` koje ukazuje na funkciju koja datu stranicu učitava u dati okvir i ima sledeći potpis (vraća negativnu vrednost u slučaju greške):

   ```
   int loadPage (size_t page, size_t frame);
   ```

   Postoji i globalna funkcija bez parametara `allocateFrame` koja alocira slobodan okvir u operativnoj memoriji i vraća njegov broj tipa `size_t`, a 0 u slučaju neuspeha. Konačno, postoji funkcija kernela koja postavlja vrednost ulaza u PMT procesa na čiji PCB ukazuje prvi parametar, za stranicu datu drugim parametrom, i u taj ulaz upisuje broj okvira dat trećim parametrom, a prava pristupa podešava prema zapisu u deskriptoru segmenta datom poslednjim parametrom:

   ```
   void setPMTEntry (PCB* pcb, size_t page, size_t frame, SegDesc* sd);
   ```

2. Implementirati internu funkciju kernela `handlePageFault` koju kernel poziva kada treba da obradi straničnu grešku procesa na čiji PCB ukazuje prvi parametar, generisanu za stranicu datu drugim parametrom. Ova funkcija treba da vrati 0 u slučaju uspeha, a različite negativne kodove u slučaju različitih grešaka. Ova funkcija ne treba da rukuje stanjem procesa (to rade drugi delovi).

   ```
   int handlePageFault (PCB* pcb, size_t page);
   ```

Rešenje

1.
```
SegDesc* findSegDesc (SegDesc* root, size_t page) {
  SegDesc* sd = root;
  while (sd) {
    if (sd->pg<=page && page<sd->pg+sd->sz) return sd;
    if (page<sd->pg) sd = sd->left;
    else sd = sd->right;
  }
  return nullptr;
}
```

2.
```
int handlePageFault (PCB* pcb, size_t page) {
  SegDesc* sd = findSegDesc(pcb->segdesc,page);
  if (!sd) return -1; // Memory access violation
  size_t frame = allocateFrame();
  if (!frame) return -2; // No free memory
  int s = sd->vtp->loadPage(page,frame);
  if (s<0) return s; // Error loading page
  setPMTEntry(pcb,page,frame,sd);
  return 0;
}
```

2. zadatak, prvi kolokvijum, septembar 2014.

Neki računar podržava segmentno-straničnu organizaciju virtuelne memorije, pri čemu je virtuelni adresni prostor veličine 16GB, adresibilna jedinica je 32-bitna reč, a fizički adresni prostor je veličine 1GB. Maksimalna veličina segmenta je 64MB, a stranica je veličine 1KB.

1. Prikazati logičku strukturu virtuelne i fizičke adrese i označiti širinu svakog polja.
2. Napisati heksadecimalni kod fizičke adrese u koju se preslikava virtuelna adresa u segmentu 56h, stranici broj 34h u tom segementu, reč broj DDh u toj stranici, ako se ta stranica preslikava u okvir broj FF00h.

Rešenje

1. VA: Segment(8):Page(16):Offset(8)

   PA: Frame(20):Offset(8).

2. FF00DDh

2. zadatak, prvi kolokvijum, mart 2013.

Neki računar podržava segmentno-straničnu organizaciju virtuelne memorije, pri čemu je virtuelna adresa 16-bitna, fizički adresni prostor je veličine 8GB, a adresibilna jedinica je 16-bitna reč. Stranica je veličine 512B. Maksimalan broj segmenata u virtuelnom adresnom prostoru je 4. Prikazati logičku strukturu virtuelne i fizičke adrese i navesti širinu svakog polja.

Rešenje

VA(16): Segment(2):Page(6):Offset(8)

PA(32): Frame(24):Offset(8)

2. zadatak, prvi kolokvijum, septembar 2013.

Neki računar podržava segmentno-straničnu organizaciju virtuelne memorije, pri čemu je virtuelni adresni prostor veličine 8GB, adresibilna jedinica je 16-bitna reč, a fizički adresni prostor je veličine 512MB. Maksimalna veličina segmenta je 32MB, a stranica je veličine 512B.

1. Prikazati logičku strukturu virtuelne i fizičke adrese i označiti širinu svakog polja.
2. Napisati heksadecimalni kod fizičke adrese u koju se preslikava virtuelna adresa u segmentu 45h, stranici broj 23h u tom segementu, reč broj FFh u toj stranici, ako se ta stranica preslikava u okvir broj FF00h.

Rešenje

1. VA: Segment(8):Page(16):Offset(8)

   PA: Frame(20):Offset(8).

2. FF00FFh

2. zadatak, prvi kolokvijum, maj 2012.

Neki računar podržava segmentno-straničnu organizaciju virtuelne memorije, pri čemu je virtuelni adresni prostor veličine 16GB, adresibilna jedinica je 32-bitna reč, a fizički adresni prostor je veličine 1GB. Maksimalna veličina segmenta je 64MB, a stranica je veličine 1KB.

1. Prikazati logičku strukturu virtuelne i fizičke adrese i označiti širinu svakog polja.
2. Napisati heksadecimalni kod fizičke adrese u koju se preslikava virtuelna adresa u segmentu 24h, stranici broj 45h u tom segementu, reč broj 15h u toj stranici, ako se ta stranica preslikava u okvir broj FF00h.

Rešenje

1. VA: Segment(8):Page(16):Offset(8)

   PA: Frame(20):Offset(8).

2. FF0015h

3. zadatak, prvi kolokvijum, april 2007.

Neki sistem podržava segmentno-straničnu organizaciju virtuelne memorije. Za procese kreirane nad istim programom moguće je uštedeti na potrošnji fizičke memorije tako što se programski kod tog programa u fizičku memoriju smešta samo na jedno mesto, pri čemu se okviri u kojima se nalaze stranice koje pripadaju segmentima koji sadrže samo programski kod fizički dele između ovih procesa (zajednički su).

1. Precizno objasniti kako se ovo deljenje stranica može implementirati.
2. Čija je odgovornost da obezbedi ovo deljenje stranica, hardvera ili operativnog sistema?
3. Precizno objasniti zašto ovakvo deljenje nije moguće za stranice koje pripadaju segmentima koji sadrže stek.

Rešenje

1. Potrebno je da postoji tabela svih pokrenutih programa. Za svaki program u tabeli se čuva lista (pokazivač na prvi u listi) PCB-ova procesa koji su pokrenuti nad tim programom. Pri pokretanju novog, potrebno ga je uvezati u odgovarajuću listu. Ukoliko već postoji neki proces pokrenutim nad istim programom, potrebno je u SMT i PMT tabele novog procesa upisati trenutno stanje prisutnosti stranica u memoriji i odgovarajuće brojeve okvira. Jedan način da to bude moguće uraditi je da se uz svaki proces pamti u koji dio virtuelnog adresnog prostora tog procesa je učitan program.

   Kada se neka stranica učita u OM, potrebno je ažurirati deskriptore u svim PMT svih procesa koji dijele tu stranicu. Isto je i kada neka stranica bude izbačena iz OM. Moguće je i efikasnije rješenje (razmisliti o još po jednoj tabeli po programu u kojoj se čuvaju deskriptori njegovih stranica, a da se u SMT/PMT procesa vrši redirekcija na ovu tabelu).

2. Opisani način se u potpunosti realizuje u OS.

3. Stek je dio konteksta svakog procesa i kao takav pravi razliku izmedju različitih izvršavanja jednog koda. Stek je struktura podataka u koju se podaci i upisuju, a ne samo čitaju. To znači da svaki proces ima svoj zaseban stek i iz tog razloga nije moguće djeljenje stranica sa stekom. (Stranice su male da bi u njih moglo stati više stekova koji se neće preklopiti).

3. zadatak, prvi kolokvijum, april 2006.

Neki računar podržava segmentno-stranični mehanizam virtuelne memorije, pri čemu je virtuelna adresa 16-bitna, fizički adresni prostor je veličine 8GB, a adresibilna jedinica je 16-bitna reč. Stranica je veličine 512B. Maksimalan broj segmenata u virtuelnom adresnom prostoru je 4. Prikazati logičku strukturu virtuelne i fizičke adrese i navesti širinu svakog polja.

Rešenje

- VA(16): seg(2), page(6), offset(8)
- PA(32): block(24), offset(8)

Dinamičko učitavanje

2. zadatak, drugi kolokvijum, maj 2019.

Neki program koristi dinamičko učitavanje. Proces uvek zauzima kontinualan deo svog virtuelnog adresnog prostora određene potrebne veličine, počev od virtuelne adrese 0. Za svaki modul predviđen za dinamičko učitavanje prevodilac u glavnom modulu programa organizuje sledeću strukturu – deskriptor tog modula:

```
struct ModDesc {
  char* base;  // Base address of the module
  size_t size; // Size of the module in sizeof(char)
  const char* name;  // Name of the module's file
};
```

U polju `base` je početna virtuelna adresa modula, ukoliko je taj modul učitan; ako modul još nije učitan, ovo polje ima vrednost *null*. Polje `size` definiše veličinu modula, a polje `name` naziv fajla sa sadržajem modula.

1. Na jeziku C napisati pomoćnu funkciju `load_module` koja treba da proširi alocirani deo virtuelnog adresnog prostora procesa za veličinu datog modula i učita taj modul u to proširenje, pod sledećim pretpostavkama:

   ```
   void load_module (ModDesc* mod);
   ```

   - globalna promenljiva programa `mem_size` tipa `size_t` sadrži trenutnu veličinu zauzetog (alociranog) dela virtuelnog adresnog prostora (u odnosu na početnu virtuelnu adresu 0); ovaj prostor treba proširiti i modul učitati u to proširenje;
   - `mem_extend(size_t)`: sistemski poziv koji proširuje alocirani deo virtuelnog adresnog prostora za zadatu veličinu;
   - `load(const char* filename, char* addr)`: sistemski poziv koji na zadatu adresu `addr` u virtuelnom adresnom prostoru procesa učitava sadržaj fajla sa zadatim imenom;
   - ignorisati greške u sistemskim pozivima (ukoliko ne može da izvrši uslugu traženu sistemskim pozivom, sistem gasi pozivajući proces).

2. Na asembleru nekog zamišljenog jednostavnog RISC procesora napisati kod koji koristi opisanu funkciju `load_module`, a koji prevodilac generiše za svaki poziv nekog potprograma `fun` koji se nalazi u nekom modulu koji se dinamički učitava, pod sledećim pretpostavkama:

   - svi registri i pokazivači su 32-bitni, kao i sva polja u strukturi `ModDesc`; adresibilna jedinica je bajt;
   - pre izvršavanja traženog koda, stvarni argumenti potprograma `fun` već su stavljeni na stek, a u registru `R0` je adresa deskriptora modula (`ModDesc*`) u kome se nalazi potprogram `fun` (ovo prevodilac zna u toku prevođenja);
   - pomeraj (relativna adresa u odnosu na početak modula) potprograma `fun` je poznat prevodiocu u toku prevođenja; ovaj pomeraj označiti simboličkom konstantom `fun`.

Rešenje

1.
```
void load_module (ModDesc* mod) {
  mem_extend(mod->size);  // Extend memory
  mod->base = (char*)0 + mem_size;
  mem_size += mod->size;
  load(mod->name,mod->base);
}
```

2.
```
            load r1, [r0] ; Load module's base address
            and r1, r1, r1 ; If loaded,
            jnz call_fun ; call the subroutine
            push r0 ; Else, load the module
            call load_module
            pop r0
            load r1, [r0] ; Load module's base address
call_fun: call #fun[r1] ; Add the subroutine's offset
            ; to the module's base addr and call it
```

3. zadatak, drugi kolokvijum, septembar 2011.

Klasa `GeoRegion`, čiji je interfejs dat u nastavku, apstrahuje geografski region i implementirana je u potpunosti. Objekti ove klase zauzimaju mnogo prostora u memoriji, pa se mogu učitavati po potrebi, dinamički. Ovo učitavanje obavlja statička operacija `GeoRegion::load`, pri čemu se objekat identifikuje datim nazivom geografskog regiona (niz znakova). Ostale operacije ove klase vraćaju vrednosti nekih svojstava geografskog regiona. Potrebno je u potpunosti implementirati klasu `GeoRegionProxy` čiji je intefejs dat u nastavku. Objekti ove klase služe kao posrednici (*proxy*) do objekata klase `GeoRegion`, pri čemu pružaju isti interfejs kao i „originali“, s tim da skrivaju detalje implementacije i tehniku dinamičkog učitavanja od svojih korisnika. Korisnici klase `GeoRegionProxy` vide njene objekte na sasvim uobičajen način, mogu ih kreirati datim konstruktorom i pozivati date operacije interfejsa, ne znajući da odgovarajuća struktura podataka možda nije učitana u memoriju.

```
class GeoRegion {
public:
  static GeoRegion* load (char* regionName);
  double getSurface ();
  double getHighestPeak ();
};
class GeoRegionProxy {
public:
  GeoRegionProxy (char* regionName);
  double getSurface ();
  double getHighestPeak ();
};
```

Rešenje

```
class GeoRegion {
public:
  static GeoRegion* load (char* regionName);
  double getSurface ();
  double getHighestPeak ();
};

class GeoRegionProxy {
public:
  GeoRegionProxy (char* regionName);
  double getSurface ();
  double getHighestPeak ();
protected:
  GeoRegion* getServer ();
private:
  GeoRegion* myServer;
  char* myName;
};

GeoRegionProxy::GeoRegionProxy (char* regionName)
  : myServer(0), myName(regionName) {}

GeoRegion* GeoRegionProxy::getServer() {
  if (myServer==0) {
    myServer=GeoRegion::load(myName);
    if (myServer==0) ... // Raise exception
  }
  return myServer;
}

double GeoRegionProxy::getSurface () {
```

```
  return getServer()->getSurface();
}

double GeoRegionProxy::getHighestPeak () {
  return getServer()->getHighestPeak();
}
```

3. zadatak, drugi kolokvijum, maj 2009.

Sledeća dva potprograma čine jedan programski modul koji treba učitavati dinamički (engl. *dynamic loading*):

```
int f (int,int);
double g (double);
```

Ovi potprogrami implementirani su u jednom C fajlu `p.c` koji izgleda ovako:

```
int _f (int,int);
double _g (double);
void* map[2] = {&_f, &_g};

int _f (int x, int y){
  ...
}

double _g (double x) {
  ...
}
```

Ovaj modul biće preveden u fajl `p.obj`. Ovaj modul ne uvozi spoljašnje simbole koji su definisani u drugim modulima, već su u njemu sva adresiranja internih simbola. Napisati C deo koda glavnog modula programa sa *stub* procedurama za ove potprograme, koje obezbeđuju dinamičko učitavanje ovog modula. Na raspolaganju je sistemska usluga:

```
void* load_module(char* filename);
```

koja učitava izvršni kod (bez zaglavlja sa simbolima) iz .obj fajla sa datim imenom u adresni prostor pozivajućeg procesa i vraća adresu na koju je učitan taj modul, a prilikom učitavanja razrešava sva adresiranja internih simbola (u ovom slučaju elemente niza `map` postavlja na apsolutne adrese funkcija `_f` i `_g` određene prilikom učitavanja). Zanemariti moguće greške prilikom učitavanja modula.

Rešenje

```
static void* module_p = NULL;

int f (int x, int y) {
  if (module_p == NULL) module_p = load_module("p.obj");
  int (*_f)(int,int) = (int(*)(int,int))(module_p[0]);
  if (module_p!=NULL && _f != NULL) return _f(x,y);
}

double g (double x) {
  if (module_p == NULL) module_p = load_module("p.obj");
  double (*_g)(double) = (double(*)(double))(module_p[1]);
  if (module_p!=NULL && _g != NULL) return _g(x);
}
```

3. zadatak, drugi kolokvijum, maj 2007.

Neki prevodilac koji podržava dinamičko učitavanje (*dynamic loading*) na početku svakog modula koji je predviđen za dinamičko učitavanje generiše tabelu adresa potprograma koji su definisani u tom modulu. Adrese su relativne u odnosu na početak modula. Na primer, za modul `M` u kome su definisani sledeći potprogrami:

```
void f(int);
int g(int,int);
double h(double,int);
```

prevodilac će na samom početku binarnog fajla sa prevodom koda tog modula generisati sledeću strukturu („tabelu“):

```
void* _funtbl[] = { &f, &g, &h };
```

Osim toga, u glavnom modulu datog programa koji se uvek inicijalno učitava pri kreiranju procesa nad tim programom, prevodilac generiše *stub* („patrljak“) za svaki potprogram koji je definisan u nekom modulu koji je predviđen za dinamičko učitavanje. Računar i operativni sistem na kome se izvršavaju ovakvi programi ne podržavaju virtuelnu memoriju. Operativni sistem obezbeđuje uslugu alokacije dela operativne memorije i učitavanja sadržaja iz fajla u taj deo memorije sledećim sistemskim pozivom:

```
void* alloc_and_load(char* filename);
```

Ovaj poziv vraća adresu u operativnoj memoriji gde je alociran prostor i učitan sadržaj fajla sa datim imenom, odnosno NULL ako je došlo do greške (npr. nema slobodnog prostora ili ne postoji fajl sa datim imenom).

Napisati kod na jeziku C za *stub* („patrljak“) funkcije `h` iz gore pomenutog modula `M` čiji je binarni oblik u fajlu „m.dlm“.

Rešenje

```
double h (double _1, int _2) {
  typedef double (*PFUN)(double,int);
  static PFUN _my_impl = NULL;
  if (_my_impl == NULL) {
    void* _m = alloc_and_load("m.dlm");
    if (_m ==   NULL) exit(1);  // Error
    _my_impl = (PFUN)((int)((void**)_m)[2]+(int)_m)
  }
  return _my_impl(_1,_2);
}
```

Preklopi

1. zadatak, kolokvijum, jul 2021.

U nekom velikom binarnom fajlu zapisane su dve velike kvadratne matrice a i b elemenata tipa `int`, dimenzije $N \times N$ i obezbeđen prostor za smeštanje još jedne takve matrice c u koju treba upisati rezultat množenja $c = a \times b$. Korišćenjem principa preklopa (*overlay*), implementirati funkciju `mat_mul()` koja realizuje ovo množenje. N je velik broj tipa `size_t`, pa u memoriju ne mogu da stanu cele navedene matrice, već samo po jedna vrsta ili kolona svake od njih. Na raspolaganju su sledeće implementirane funkcije:

- `void load_arow (int (*)[], size_t i)`: u niz na koji ukazuje prvi parametar iz fajla učitava vrstu matrice a datu drugim parametrom;
- `void load_bcol (int (*)[], size_t i)`: u niz na koji ukazuje prvi parametar iz fajla učitava kolonu matrice b datu drugim parametrom;
- `void store_crow (int (*)[], size_t i)`: iz niza na koji ukazuje prvi parametar u fajl upisuje vrstu matrice c datu drugim parametrom.

Ne treba otvarati dati fajl (on je već otvoren) niti ga zatvarati (zatvoriće ga pozivalac). Sve eventualne greške obrađuju date implementirane funkcije.

Rešenje

```
const size_t N = ...;

int ai[N], bj[N], ci[N];

void mat_mul() {
    for (size_t i = 0; i < N; i++) {
        load_arow(&ai, i);
        for (size_t j = 0; j < N; j++) {
            load_bcol(&bj, j);
            ci[j] = 0;
            for (size_t k = 0; k < N; k++) {
                ci[j] += ai[k] * bj[k];
            }
        }
        store_crow(&ci, i);
    }
}
```

2. zadatak, drugi kolokvijum, jun 2017.

U nastavku je data implementacija jednog programa koji ciklično vrši neku obradu podeljenu u dve faze. Prva faza obrade koristi (čita i menja) podatke koji se samo koriste u toj fazi; svi ovakvi podaci grupisani su u jednu veliku strukturu podataka tipa `Phase1Data`; ali koristi i podatke koji su zajednički za obe faze, odnosno preko kojih obrade u ove dve faze razmenjuju informacije; ovi podaci grupisani su u strukturu tipa `CommonData`. Analogno radi i obrada u drugoj fazi.

Sa ciljem značajnog smanjenja potrebe za operativnom memorijom, potrebno je restrukturirati ovaj program tako da koristi preklop (*overlay*) u koji se smeštaju podaci prve, odnosno druge faze koji se ne koriste istovremeno. Za čuvanje izbačenog sadržaja treba kreirati dva privremena fajla u tekućem direktorijumu procesa. Za rad sa fajlovima na raspolaganju su sledeći sistemski pozivi; sve ove funkcije vraćaju 0 u slučaju uspeha, a negativnu vrednost u slučaju greške:

- `FILE fopen(char* filename)`: otvara fajl sa zadatim imenom za čitanje i upis; ako fajl ne postoji, kreira ga;
- `int fread(FILE file, int offset, int size, void* buffer)`: iz datog fajla, sa pozicije rednog broja bajta `offset` (numeracija počev od 0), čita niz bajtova dužine `size` u memoriju na lokaciju na koju ukazuje `buffer`; ukoliko se čitanjem prekorači granica sadržaja fajla, vraća grešku;

- `int fwrite(FILE file, int offset, int size, void* buffer)`: u dati fajl, na poziciju rednog broja bajta `offset` (numeracija počev od 0), upisuje niz bajtova dužine `size` sa lokacije na koju ukazuje `buffer`; ukoliko se upisom prekorači granica sadržaja fajla, sadržaj fajla se proširuje tako da primi sav upisani sadržaj;
- `int fclose(FILE file)`: zatvara dati fajl.

```
struct Phase1Data p1data;
struct Phase2Data p2data;
struct CommonData cdata;

void main () {
  init_cdata(&cdata);   // Initialize common data (cdata)
  init_p1data(&p1data); // Initialize data for phase 1 (p1data)
  init_p2data(&p2data); // Initialize data for phase 2 (p2data)

  do {
    process_phase_1(&p1data,&cdata);  // Perform phase 1 processing
    process_phase_2(&p2data,&cdata);  // Perform phase 2 processing
  } while (!cdata.completed);
}
```

Rešenje

```
union OverlaidData {
  struct Phase1Data p1data;
  struct Phase2Data p2data;
} data;
struct CommonData cdata;

int main () {
  FILE fd1 = fopen("overlay1.tmp");
  if (fd1<0) return fd1;
  FILE fd2 = fopen("overlay2.tmp");
  if (fd2<0) {
    fclose(fd1);
    return fd2;
  }

  int ret = 0;

  init_cdata(&cdata);
  init_p1data(&data.p1data);
  process_phase_1(&data.p1data,&cdata);
  if ((ret=fwrite(fd1,0,sizeof(data.p1data),&data.p1data))<0) {
    fclose(fd1);
    fclose(fd2);
    return ret;
  }

  init_p2data(&data.p2data);
  process_phase_2(&data.p2data,&cdata);

  while (!cdata.completed) {
    if ((ret=fwrite(fd2,0,sizeof(data.p2data),&data.p2data))<0) break;

    if ((ret=fread(fd1,0,sizeof(data.p1data),&data.p1data))<0) break;
    process_phase_1(&data.p1data,&cdata);
    if ((ret=fwrite(fd1,0,sizeof(data.p1data),&data.p1data))<0) break;
```

```
    if ((ret=fread(fd2,0,sizeof(data.p2data),&data.p2data))<0) break;
    process_phase_2(&data.p2data,&cdata);
  }

  fclose(fd1);
  fclose(fd2);
  return ret;
}
```

2. zadatak, drugi kolokvijum, maj 2016.

Neki sistem koristi preklope (*overlays*). Prevodilac i linker u generisanom kodu prevedenog programa koji koristi preklope statički alociraju i adekvatno inicijalizuju sledeće strukture podataka:

- za svaki modul (preklop, *overlay*) postoji sledeći deskriptor; moduli koji se preklapaju imaju istu početnu adresu:

  ```
  struct OverlayDescr {
    const char* filename; // Naziv fajla u kome je binarni sadržaj preklopa
    void* addr; // Adresa u adresnom prostoru procesa na kojoj se nalazi
    bool isLoaded; // Da li je preklop trenutno učitan?
  }
  ```

- tabela svih preklopa-modula:

  ```
  const int numOfOverlays = ...; // Ukupan broj preklopa
  OverlayDescr overlays[numOfOverlays];  // Tabela svih preklopa
  ```

- svakom potprogramu koji se nalazi u nekom preklopu prevodilac pridružuje jedan jedinstveni ceo broj (identifikator), koji predstavlja ulaz u tabelu tih potprograma; svaki ulaz u ovoj tabeli sadrži pokazivač na deskriptor preklopa u kome se nalazi taj potprogram:

  ```
  const int numOfProcs = ...; // Ukupan broj potprograma
  OverlayDescr* procedureMap[numOfProcs];  // Tabela potprograma
  ```

Za svaki poziv potprograma koji se nalazi u nekom preklopu, na uvek istoj i prevodiocu poznatoj adresi u adresnom prostoru procesa, npr. `proc(a,b,c)`, prevodilac generiše kod koji je ekvivalent sledećeg koda:

```
ensureOverlay (proc_id); // proc_id je identifikator za proc
proc(a,b,c); // standardan poziv na poznatoj adresi
```

Funkcija `ensureOverlay(int procID)` treba da obezbedi da je potprogram sa datim identifikatorom prisutan u memoriji, odnosno po potrebi u čita njegov preklop. Potrebno je implementirati ovu funkciju, pri čemu je na raspolaganju sistemski poziv koji učitava binarni sadržaj iz fajla sa zadatim imenom na zadatu adresu u adresnom prostoru procesa:

```
void sys_loadBinary(const char* filename, void* address);
```

Rešenje

```
void ensureOverlay (int procedureID) {
  OverlayDescr* ovrl = procedureMap[procedureID];
  if (!ovrl->isLoaded) {
    for (int i=0; i<numOfOverlays; i++)
      if (overlays[i].addr==ovrl->addr) overlays[i].isLoaded = false;
    sys_loadBinary(ovrl->filename,ovrl->addr);
    ovrl->isLoaded = true;
  }
}
```

3. zadatak, drugi kolokvijum, septembar 2013.

Data je jedna realizacija klase `DLArray` koja apstrahuje dinamički niz elemenata tipa `double` velike dimenzije zadate prilikom inicijalizacije parametrom `size`. U toj implementaciji, u svakom trenutku se u memoriji drži samo jedan keširani blok ovog niza veličine zadate parametrom `blockSize`. Implementacija koristi dinamičko učitavanje bloka u kome se nalazi element kome se pristupa uz zamenu (*swapping*), zapravo neku vrstu preklopa (*overlay*), tako da se u memoriji uvek nalazi samo jedan keširani blok kome se trenutno pristupa, dok se ceo niz nalazi u fajlu. Interfejs te klase je u njenom javnom delu.

U funkcijama `fread` i `fwrite` za pristup fajlu i učitavanje, odnosno snimanje datog niza elemenata tipa `double` na zadatu poziciju u fajlu, pozicija se izražava u jedinicama veličine binarnog zapisa tipa `double`, počev od 0, što znači da se fajl kroz ove funkcije može posmatrati kao veliki niz elemenata tipa `double`:

```
void fread (FHANDLE, int position, double* buffer, int bufferSize);
void fwrite(FHANDLE, int position, double* buffer, int bufferSize);
```

Modifikovati datu implementaciju ove klase (bez modifikacije njenog interfejsa) tako da se u memoriji nalaze uvek dva bloka iz niza u dva slota (umesto u jednom, kako je sada), pri čemu se u jedan slot preslikavaju parni, a u drugi neparni blokovi niza.

```
class DLArray {
public:
  inline DLArray (int size, int blockSize, FHANDLE fromFile);

  inline double get (int i);  // Get element [i]
  inline void set (int i, double x); // Set element [i]

protected:
  inline void save();
  inline void load(int blockNo);
  inline void fetch(int blockNo);

private:
  FHANDLE file;
  int size, blockSize;
  int curBlock;
  int dirty;
  double* block;
};


DLArray::DLArray (int s, int bs, FHANDLE f) :
  file(f), size(s), blockSize(bs), curBlock(0), dirty(0) {
  block = new double[bs];
  if (block) load(curBlock);
}

void DLArray::save () {
  fwrite(file,curBlock*blockSize,block,blockSize);
  dirty=0;
}


void DLArray::load (int b) {
  curBlock = b;
  fread(file,curBlock*blockSize,block,blockSize);
  dirty = 0;
}
```

```
void DLArray::fetch(int b) {
  if (curBlock!=b) {
    if (dirty) save();
    load(b);
  }
}

double DLArray::get (int i) {
  if (block==0 || i<0 || i>=size) return 0; // Exception
  fetch(i/blockSize);
  return block[i%blockSize];
}

void DLArray::set (int i, double x) {
  if (block==0 || i<0 || i>=size) return; // Exception
  fetch(i/blockSize);
  if (block[i%blockSize]!=x) {
    block[i%blockSize]=x;
    dirty=1;
  }
}
```

Rešenje

```
class DLArray {
public:
  DLArray (int size, int blockSize, FHANDLE fromFile);

  inline double get (int i);  // Get element [i]
  inline void set (int i, double x); // Set element [i]

protected:
  inline void save(int slot);
  inline void load(int blockNo, int slot);
  inline int fetch(int blockNo);

private:
  FHANDLE file;
  int size, blockSize;
  int curBlock[2];
  int dirty[2];
  double* block[2];
};

DLArray::DLArray (int s, int bs, FHANDLE f) :
  file(f), size(s), blockSize(bs) {
  for (int slot=0; slot<2; slot++) {
    curBlock[slot]=slot;
    dirty[slot]=0;
    block[slot] = new double[bs];
    load(curBlock[slot],slot);
  }
}

void DLArray::save (int slot) {
  fwrite(file,curBlock[slot]*blockSize,block[slot],blockSize);
  dirty[slot]=0;
```

```
}


void DLArray::load (int b, int slot) {
  curBlock[slot] = b;
  fread(file,curBlock[slot]*blockSize,block[slot],blockSize);
  dirty[slot] = 0;
}

int DLArray::fetch(int b) {
  int slot = b%2;
  if (curBlock[slot]!=b) {
    if (dirty[slot]) save(slot);
    load(b,slot);
  }
  return slot;
}

double DLArray::get (int i) {
  if (block==0 || i<0 || i>=size) return 0; // Exception
  int slot = fetch(i/blockSize);
  return block[slot][i%blockSize];
}

void DLArray::set (int i, double x) {
  if (block==0 || i<0 || i>=size) return; // Exception
  int slot = fetch(i/blockSize);
  if (block[slot][i%blockSize]!=x) {
    block[slot][i%blockSize]=x;
    dirty[slot]=1;
  }
}
```

3. zadatak, drugi kolokvijum, maj 2012.

Neki program treba da izračuna proizvod dve ogromne matrice $A[m \times k] \times B[k \times n] = C[m \times n]$, gde su dimenzije matrica veoma velike (i po nekoliko miliona). Predložiti i precizno opisati kako biste primenili tehniku preklopa (*overlays*) u ovom programu. Posebno objasniti način smeštanja matrica u preklope i redosled učitavanja u preklope.

Rešenje

Dovoljno je, na primer, da se samo jedna vrsta (i) matrice A smesti u jedan preklop. U drugi preklop smeštena je jedna kolona (j) matrice B. Tako se iz njih može izračunati `C[i,j]`. Ako se u unutrašnjoj petlji varira j (`j:=1..n`), u jednoj iteraciji spoljašnje petlje dobija se jedna vrsta i matrice C. Ova vrsta je u svom, trećem preklopu. U narednoj iteraciji spoljašnje petlje po i, u prvi preklop biće učitana nova vrsta matrice A, u drugom preklopu biće učitavana jedna po jedna kolona (`j:=1..n`) matrice B u unutrašnjoj petlji, a u trećem preklopu biće izračunavana nova vrsta matrice C.

3. zadatak, drugi kolokvijum, maj 2011.

U potpunosti realizovati klasu koja apstrahuje dinamički niz elemenata tipa `double` velike dimenzije zadate prilikom inicijalizacije parametrom `size`. U svakom trenutku se u memoriji drži samo jedan keširani blok ovog niza veličine zadate parametrom `blockSize`. Implementacija treba da koristi dinamičko učitavanje bloka u kome se nalazi element kome se pristupa uz zamenu (*swapping*), zapravo neku vrstu preklopa (*overlay*), tako da se u memoriji uvek nalazi samo jedan keširani blok kome se trenutno pristupa, dok se ceo niz nalazi u fajlu. Interfejs ove klase treba da bude:

```
class DLArray {
public:
```

```
  DLArray (int size, int blockSize, FHANDLE fromFile);
  double get (int i);  // Get element [i]
  void set (int i, double x); // Set element [i]
};
```

Inicijalno se vrednosti celog niza nalaze u binarnom fajlu zadatom trećim argumentom (otvaranje i zatvaranje tog fajla je u odgovornosti korisnika ove klase). Na raspolaganju su sledeće funkcije za pristup fajlu i učitavanje, odnosno snimanje datog niza elemenata tipa `double` na zadatu poziciju u fajlu. Pozicija se izražava u jedinicama veličine binarnog zapisa tipa `double`, počev od 0, što znači da se fajl kroz ove funkcije može posmatrati kao veliki niz elemenata tipa `double`:

```
void fread (FHANDLE, int position, double[], int bufferSize);
void fwrite(FHANDLE, int position, double[], int bufferSize);
```

Rešenje

```
class DLArray {
public:
  inline DLArray (int size, int blockSize, FHANDLE fromFile);

  inline double get (int i);  // Get element [i]
  inline void set (int i, double x); // Set element [i]

protected:
  inline void save();
  inline void load(int blockNo);
  inline void fetch(int blockNo);

private:
  FHANDLE file;
  int size, blockSize;
  int curBlock;
  int dirty;
  double* block;
};


DLArray::DLArray (int s, int bs, FHANDLE f) :
  file(f), size(s), blockSize(bs), curBlock(0), dirty(0) {
  block = new double[bs];
  if (block) load(curBlock);
}

void DLArray::save () {
  fwrite(file,curBlock*blockSize,block,blockSize);

  dirty=0;
}


void DLArray::load (int b) {
  curBlock = b;
  fread(file,curBlock*blockSize,block,blockSize);
  dirty = 0;
}

void DLArray::fetch(int b) {
  if (curBlock!=b) {
```

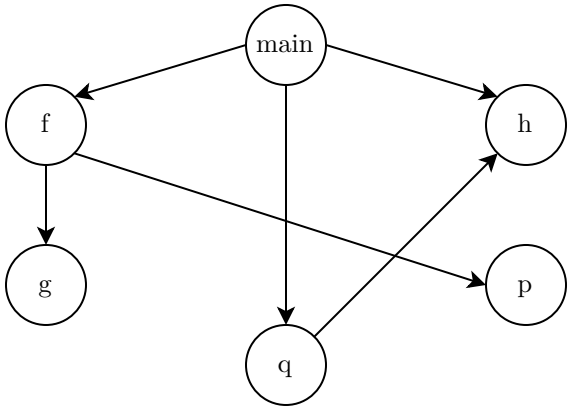
```
    if (dirty) save();
    load(b);
  }
}

double DLArray::get (int i) {
  if (block==0 || i<0 || i>=size) return 0; // Exception
  fetch(i/blockSize);
  return block[i%blockSize];
}

void DLArray::set (int i, double x) {
  if (block==0 || i<0 || i>=size) return; // Exception
  fetch(i/blockSize);
  if (block[i%blockSize]!=x) {
    block[i%blockSize]=x;
    dirty=1;
  }
}
```

3. zadatak, drugi kolokvijum, maj 2010.

1. Ukoliko prevodilac generiše samo uobičajeni kod za povratak iz potprograma (skidanje samo sačuvane povratne adrese sa steka i skok na tu adresu), da li se dve procedure, od kojih jedna poziva onu drugu, mogu nalaziti u dva različita modula-preklopa (*overlays*) koji se učitavaju na isto mesto (jedan preko drugog)? Obrazložiti odgovor.
2. Na slici je dat graf poziva potprograma nekog programa. Čvorovi grafa predstavljaju potprograme, a grana je usmerena od pozivaoca prema pozvanom potprogramu. Za pretpostavke u tački a) i na osnovu zaključka iz te tačke, odgovoriti da li je korektna svaka od sledeće dve konfiguracije preklopa.



Slika 2: Graf

1. Konfiguracija 1:
   - Modul A: *main*, h
   - Modul B: f, g, p
   - Modul C: q
   - Preklapaju se modul B i modul C, modul A je uvek učitan. Korektna?
2. Konfiguracija 2:
   - Modul A: *main*, p
   - Modul B: f, g
   - Modul C: q, h
   - Preklapaju se modul B i modul C, modul A je uvek učitan. Korektna?

Rešenje

1. Ne. Kada se iz pozivajućeg potprograma pozove onaj drugi, na isto mesto modula kome pripada pozivajući potprogram se učitava modul u kome je pozvani potprogram. Kada se vrši povratak iz tog pozvanog potprograma, ukoliko prevodilac generiše samo kod za jednostavni indirektni skok preko adrese skinute sa steka, skok će biti na adresu unutar istog modula, a ne na kod unutar pozivajućeg potprograma, jer je on u modulu koji je izbačen, što nije korekno. Prema tome, potprogrami koji su u relaciji pozivalac-pozvani se mogu nalaziti ili u istom modulu, ili u dva modula koji se ne preklapaju (ne učitavaju na isto mesto jedan preko drugog).
2. Obe konfiguracije su korektne, pošto je za sve grane zadovoljen uslov iz zaključka prethodne tačke.

Prekidi

2. zadatak, drugi kolokvijum, maj 2022.

Neki procesor pri obradi prekida, sistemskog poziva i izuzetka prelazi na sistemski stek. Taj stek alociran je u delu memorije koju koristi kernel, a na vrh tog steka ukazuje poseban registar SSP procesora koji je dostupan samo u privilegovanom režimu.

Prilikom obrade ovih situacija, procesor ništa ne stavlja na stek, već zatečene, neizmenjene vrednosti registara PC i PSW koje je koristio prekinuti proces sačuva u posebne, za to namenjene registre, SPC i SPSW, respektivno, a vrednosti registara SP i SSP međusobno zameni (*swap*). Prilikom povratka iz prekidne rutine instrukcijom `iret` procesor radi inverznu operaciju. Procesor je RISC, sa *load/store* arhitekturom i ima 32 registra opšte namene (R0..R31).

Kernel je višenitni, a svakom toku kontrole (procesu ili niti), uključujući i niti kernela, pridružen je poseban stek koji se koristi u sistemskom režimu. Prilikom promene konteksta, kontekst procesora treba sačuvati na tom steku, dok informaciju o vrhu steka treba čuvati u polju PCB čiji je pomeraj u odnosu na početak strukture PCB označen simboličkom konstantom `offsSP`.

U kodu kernela postoji statički pokazivač `oldRunning` koji ukazuje na PCB tekućeg procesa, kao i pokazivač `newRunning` koji ukazuje na PCB procesa koji je izabran za izvršavanje.

Napisati kod funkcije `yield` koju koristi kernel kada želi da promeni kontekst (prebaci se sa izvršavanja jednog toka kontrole, `oldRunning`, na drugi, `newRunning`), na bilo kom mestu gde se za to odluči.

Rešenje

```
void yield() {
    asm {
        ; Save the current context
        push spc ; save regs on the process stack
        push ssp
        push spsw
        push r0
        push r1
        ...
        push r31
        load r0, oldRrunning ; r0 now points to the running PCB
        store sp, [r0 + offsSP] ; save SP
        ; Restore the new context
        load r0, newRunning
        load sp, [r0 + offsSP] ; restore SP
        pop r31 ; restore regs
        ...
        pop r0
        pop spsw
        pop ssp
        pop spc
    }
}
```

1. zadatak, kolokvijum, jun 2020.

Neki procesor obrađuje prekide (hardverske i softverske) tako što tokom izvršavanja prekidne rutine koristi poseban stek koji se koristi u sistemskom, privilegovanom režimu rada procesora, u kome se izvršava kod kernela (čiji je deo i prekidna rutina). Taj stek alociran je u delu memorije koju koristi kernel, a na vrh tog steka ukazuje poseban registar SSP procesora koji je dostupan samo u privilegovanom režimu.

Prilikom obrade prekida, procesor ništa ne stavlja na stek, pa tako ni ne menja sadržaj pokazivača vrha korisničkog steka (SP) koji je koristio prekinuti proces (SP zadržava staru vrednost), dok tekuću (staru) vrednost statusnog

registra (PSW) i programskog brojača (PC) sačuva u posebne, za to namenjene registre, SPSW i SPC, respektivno. Prilikom povratka iz prekidne rutine instrukcijom `iret`, procesor restaurira registre PSW i PC prepisujući vrednosti iz registara SPSW i SPC i vraća se u korisnički režim, a time i na korisnički stek.

Ovaj isti sistemski stek se koristi tokom izvršavanja bilo kog koda kernela; kernel ima samo jedan takav stek (nema više niti). Prilikom promene konteksta, kontekst procesora treba sačuvati u odgovarajućim poljima strukture PCB, u kojoj postoji polje za čuvanje svakog od programski dostupnih registara; pomeraj ovog polja u odnosu na početak strukture PCB označen je simboličkim konstantama `offsPC`, `offsSP`, `offsPSW`, `offsR0`, `offsR1` itd. Procesor je RISC, sa *load/store* arhitekturom i ima 32 registra opšte namene (R0..R31).

U kodu kernela postoji statički definisan pokazivač `running` koji ukazuje na PCB tekućeg procesa. Potprogram `s_call`, koji nema argumente, realizuje sistemski poziv zadat od strane korisničkog procesa vrednostima u registrima i po potrebi raspoređivanje, tako što smešta PCB na koji ukazuje pokazivač `running` u listu spremnih procesa ili na drugo mesto, a iz liste spremnih uzima jedan odabrani proces i postavlja pokazivač `running` na njega. Na asembleru datog procesora napisati kod prekidne rutine `sys_call` koja vrši sistemski poziv korišćenjem potprograma `s_call`. Ova prekidna rutina poziva se softverskim prekidom iz korisničkog procesa, pri čemu se sistemski poziv realizuje softverskim prekidom.

Rešenje

```
sys_call: ; Save the current context
          push r0 ; save r0 temporarily on the (kernel) stack
          load r0, [running] ; r0 now points to the running PCB
          store r1, #offsR1[r0] ; save r1
          store r2, #offsR2[r0] ; save r2
          ...                   ; save r3-r30
          store r31, #offsR31[r0] ; save r31
          pop r1            ; save r0
          push r1
          store r1, #offsR0[r0]
          store SPC, #offsPC[r0] ; save PC
          store SPSW,#offsPSW[r0] ; save PSW
          store SP, #offsSP[r0] ; save SP
          load r1, #offstR1[r0]  ; restore sys call args in r1
          pop r0            ; and r0

          ; Execute the system call
          call s_call

          ; Restore the new context
          load r0, [running]
          load SP, #offsSP[r0] ; restore SP
          load SPSW,#offsPSW[r0] ; restore PSW
          load SPC, #offsPC[r0] ; restore PC
          load r31, #offsR31[r0] ; restore R31
          ...                    ; restore r30-r2
          load r1, #offsR1[r0] ; restore R1
          load r0, #offsR0[r0] ; restore R0

          ; Return
          iret
```

2. zadatak, prvi kolokvijum, mart 2019.

Neki procesor obrađuje prekide (hardverske i softverske) tako što tokom izvršavanja prekidne rutine koristi poseban stek koji se koristi u sistemskom, privilegovanom režimu rada procesora, u kome se izvršava kod kernela (čiji je deo i prekidna rutina). Taj stek alociran je u delu memorije koju koristi kernel, a na vrh tog steka ukazuje poseban registar SSP procesora koji je dostupan samo u privilegovanom režimu.

Prilikom obrade prekida, procesor na ovom steku čuva: pokazivač vrha korisničkog steka (SP) koji je koristio prekinuti proces, programsku statusnu reč (PSW) i programski brojač (PC), tim redom, ali *ne* i ostale programski dostupne registre. Prilikom povratka iz prekidne rutine instrukcijom `iret`, procesor restaurira ove registre sa sistemskog steka i vraća se u korisnički režim, a time i na korisnički stek.

Ovaj isti sistemski stek se koristi tokom izvršavanja bilo kog koda kernela; kernel ima samo jedan takav stek (nema više niti). Prilikom promene konteksta, programski dostupne registre treba sačuvati u odgovarajućim poljima strukture PCB, u kojoj postoji polje za čuvanje svakog od programski dostupnih registara; pomeraj ovog polja u odnosu na početak strukture PCB označen je simboličkim konstantama `offsPC`, `offsSP`, `offsPSW`, `offsR0`, `offsR1` itd.

Procesor je RISC, sa *load/store* arhitekturom i ima sledeće programski dostupne registre: 32 registra opšte namene (R0..R31), SP, PSW i PC.

U kodu kernela postoji statički definisan pokazivač `running` koji ukazuje na PCB tekućeg procesa. Potprogram `scheduler`, koji nema argumente, realizuje raspoređivanje, tako što smešta PCB na koji ukazuje pokazivač `running` u listu spremnih procesa, a iz nje uzima jedan odabrani proces i postavlja pokazivač `running` na njega.

Na asembleru datog procesora napisati kod prekidne rutine `dispatch` koja vrši promenu konteksta korišćenjem potprograma `scheduler`. Ova prekidna rutina poziva se sistemskim pozivom iz korisničkog procesa, pri čemu se sistemski poziv realizuje softverskim prekidom.

Rešenje

```
dispatch: ; Save the current context
          push r0 ; save r0 temporarily on the stack
          load r0, [running] ; r0 now points to the running PCB
          store r1, #offsR1[r0] ; save r1
          store r2, #offsR2[r0] ; save r2
          ...                   ; save r3-r30
          store r31, #offsR31[r0] ; save r31
          pop r1            ; save r0
          store r1, #offsR0[r0]
          pop r1            ; save PC
          store r1, #offsPC[r0]
          pop r1            ; save PSW
          store r1, #offsPSW[r0]
          pop r1            ; save SP
          store r1, #offsSP[r0]

          ; Select the next running process
          call scheduler

          ; Restore the new context
          load r0, [running]
          load r1, #offsSP[r0] ; restore SP
          push r1
          load r1, #offsPSW[r0] ; restore PSW
          push r1
          load r1, #offsPC[r0] ; restore PC
          push r1
          load r31, #offsR31[r0] ; restore R31
          ...                    ; restore r30-r2
          load r1, #offsR1[r0] ; restore R1
          load r0, #offsR0[r0] ; restore R0

          ; Return
          iret
```

2. zadatak, prvi kolokvijum, maj 2019.

Neki sistem čuva kontekst niti u strukturi PCB u kojoj postoje polja za čuvanje vrednosti svih programski dostupnih registara procesora: pokazivača na vrh steka niti (SP), programske statusne reči (PSW) i programskog brojača (PC), kao i registara opšte namene R0..R31. Adrese i svi registri su 32-bitni, pa su takvi i tipovi `int` i svi pokazivački tipovi na jeziku C.

Dole je dat segment asemblerskog koda kojim ovaj sistem restaurira kontekst niti nakon svakog sistemskog poziva. Simboličke konstante `pc`, `psw`, `sp` itd. predstavljaju pomeraje (engl. *offset*) istoimenih polja u strukturi PCB. Registar R0 prenosi vrednost koja se vraća iz sistemskog poziva, pa se zato ne restaurira.

```
; Restore the new context
load r1, [running]
load r2, #sp[r1] ; restore SP
push r2
load r2, #psw[r1] ; restore PSW
push r2
load r2, #pc[r1] ; restore PC
push r2
load r31, #r31[r1] ; restore R31
...                    ; restore r30-r2
load r1, #r1[r1] ; restore R1
; R0 holds the return value from the system call
; Return from system call (interrupt)
iret
```

Potrebno je dopuniti implementaciju operacije `createThread` kodom na mestu označenom sa `/***/`. Ovu funkciju poziva kernel u implementaciji sistemskog poziva za kreiranje nove niti, a kod koji nedostaje treba da formira inicijalni procesorski kontekst. Nit treba kreirati nad korisničkom funkcijom na koju ukazuje argument `pf`, tako da ta nit izvršava tu funkciju sa argumentom datim u `arg`. Nakon izvršavanja ove korisničke funkcije `pf`, nit treba da se ugasi sistemskim pozivom implementiranim u funkciji `exit`.

Funkcija `createPCB` alocira novu strukturu PCB i adresu te strukture upisuje u pokazivač na koga ukazuje argument. Funkcija `allocateStack` alocira stek za novu nit i adresu najviše slobodne lokacije steka upisuje u pokazivač na koga ukazuje argument. Stek raste ka nižim adresama, a pokazivač vrha steka ukazuje na prvu slobodnu lokaciju.

```
void exit ();  // Terminate the calling thread
const int PSW_INIT = ...; // Initial value for PSW

int createThread (void (*pf) (void*), void* arg) {
  PCB* pcb;
  int stat = createPCB(&pcb);
  if (stat!=0) return stat;
  void** sp;
  stat = allocateStack(&sp);
  if (stat!=0) return stat;
  /***/
  pcb->psw = PSW_INIT;
  stat = Scheduler::put(pcb);
  return stat;
}
```

Rešenje

```
  *sp-- = arg;
  *sp-- = &exit;
  pcb->sp = sp;
  pcb->pc = pf;
```

2. zadatak, prvi kolokvijum, mart 2018.

U implementaciji jezgra nekog jednoprocesorskog *time-sharing* operativnog sistema, radi pojednostavljenja celog mehanizma promene konteksta, primenjeno je sledeće neobično rešenje. Promena konteksta vrši se isključivo kao posledica prekida, na samo jednom mestu u kodu koji se izvršava na prekid. Prekidi dolaze od raznih uređaja, u najmanju ruku od vremenskog brojača, jer on u svakom slučaju generiše prekid zbog toga što je tekućoj niti isteklo dodeljeno procesorsko vreme. Zbog toga tekuća nit nikada ne gubi procesor sinhrono, čak ni kada poziva blokirajuću operaciju (sistemski poziv). Umesto toga, ukoliko je potrebno da se nit suspenduje (blokira) u nekom blokirajućem pozivu, nit se samo „označi“ suspendovanom i nastavlja sa izvršavanjem (uposlenim čekanjem) sve dok ne stigne sledeći prekid. Kada takav prekid stigne, prekidna rutina vrši samu promenu konteksta.

Na primer, implementacija operacije `suspend`, koja suspenduje pozivajući proces, i `resume`, koja ponovo deblokira dati proces, izgledaju ovako:

```
void suspend () {
  running->status = suspended; // Mark as suspended
  while (running->status==suspended); // and then busy-wait
  // until it is preempted, suspended, and then resumed later
}

void resume (int pid) {
  processes[pid].status = ready; // Mark as ready
}
```

Procesor je RISC sa *load/store* arhitekturom, ima 32 registra opšte namene i SP. Prilikom prekida na steku čuva samo PC i PSW. Tajmer se restartuje upisom odgovarajuće vrednosti u registar koji se nalazi na adresi simbolički označenoj sa `Timer`. PCB procesa je dat strukturom definisanom dole, a svi procesi zapisani su u nizu `processes`. U asembleru, simboličko ime polja strukture ima vrednost pomeraja (engl. *offset*) tog polja od početka strukture.

```
enum ProcessStatus { unused, initiating, terminating, ready, suspended };
typedef unsigned long Time;
typedef unsigned Register;

struct PCB {
  ProcessStatus status;  // Process status
  Time timeSlice;        // Time slice for time sharing
  Register savedSP;      // Saved stack pointer
  ...
}

const unsigned long NumOfProcesses = ...;
PCB processes[NumOfProcesses];
PCB* running;  // Running process
```

1. Na asembleru datog procesora napisati kod prekidne rutine koja vrši promenu konteksta. Ova prekidna rutina, pored čuvanja i restauracije konteksta na steku procesa, treba da pozove potprogram `scheduler` koji će u pokazivač `running` smestiti vrednost koja ukazuje na novoizabrani tekući proces, i da tajmer restartuje sa vremenskim kvantumom dodeljenim tom procesu. Pretpostavlja se da uvek postoji barem jedan spreman proces.
2. Na jeziku C napisati potprogram `scheduler` koji bira sledeći proces za izvršavanje. Spremne procese treba da bira redom, u krug, a ne svaki put od početka niza `processes`.

Rešenje

```
dispatch: ; Save the current context
          push r0 ; save regs
          push r1
          ...
          push r31
          load r0, running
```

```
          store sp, #savedSP[r0] ; save sp

          ; Select the next running process
          call scheduler

          ; Restore the new context
          load r0, running
          load r1, #timeSlice[r0]; restart timer
          store r1, [Timer]
          load sp, #savedSP[r0] ; restore sp
          pop r31
          pop r30 ; restore regs
          ...
          pop r0

          ; Return
          iret
```

```
void scheduler () {
  do {
    running = processes + (running - processes + 1) % NUM_OF_PROCESSES;
  } while (running->status!=ready);
}
```

2. zadatak, prvi kolokvijum, april 2018.

Neki procesor obrađuje prekide (hardverske i softverske) tako što tokom izvršavanja prekidne rutine koristi poseban stek koji se koristi u sistemskom, privilegovanom režimu rada procesora, u kome se izvršava kod kernela (čiji je deo i prekidna rutina). Taj stek alociran je u delu memorije koju koristi kernel, a na vrh tog steka ukazuje poseban registar SSP procesora koji je dostupan samo u privilegovanom režimu.

Prilikom obrade prekida, procesor na ovom steku čuva programski brojač (PC) i programsku statusnu reč (PSW), tim redom, ali ne i ostale programski dostupne registre. Prilikom povratka iz prekidne rutine instrukcijom `iret`, procesor restaurira ove registre sa sistemskog steka i vraća se u korisnički režim, a time i na korisnički stek.

Postoji samo jedan kernel stek koji se koristi za izvršavanje celog koda kernela. Svi potprogrami kernela pisani su tako da na ovom steku čuvaju (i sa njega potom restauriraju) sve registre procesora koje koriste. Za sve vreme izvršavanja kernel koda prekidi su maskirani (procesor ih implicitno maskira prilikom prihvatanja prekida, a kernel kod ih ne demaskira).

U strukturi PCB postoje polja za čuvanje vrednosti svih programski dostupnih registara procesora; pomeraj polja za neki registar R*i* u odnosu na početak strukture PCB označen je simboličkom konstantom `offsRi` (npr. `offsSP`, `offsPC`, `offsPSW`, `offsR0`, ..., `offsR31`).

Procesor je RISC, sa *load/store* arhitekturom i ima sledeće programski dostupne registre: 32 registra opšte namene (R0..R31), SP, PSW i PC. Svi registri su 32-bitni.

U kodu kernela postoji statički definisan pokazivač `running` koji ukazuje na PCB tekućeg procesa. Potprogram `sys_call_proc`, koji nema argumente, realizuje obradu sistemskog poziva, kao i raspoređivanje, tako što, nakon obrade samog sisteskog poziva, postavlja pokazivač `running` da ukazuje na PCB odabranog novog tekućeg procesa.

Na asembleru datog procesora napisati kod prekidne rutine `sys_call` za sistemske pozive softverskim prekidom, s tim da ona, zbog efikasnijeg rada, ne treba da vrši promenu konteksta (čuvanje i restauraciju registara) ako za tim nema potrebe, odnosno ukoliko procedura `sys_call_proc` nije promenila pokazivač `running`.

Rešenje

```
sys_call: ; Save r0 and r1 on the (kernel) stack
          push r0
          push r1
```

```
        load r0, [running] ; the old running is in r0

        ; Perform the system call:
        call sys_call_proc

        ; Compare the old and the new running,
        load r1, [running] ; the new running is in r1
        cmp r0, r1
        jne switch

        ; and do not switch the context if they are equal,
        ; but restore r0 and r1, and return
        pop r1
        pop r0
        iret
switch: ; Save the context of the old running
        store r2,#offsR2[r0] ; save other regs
        store r3,#offsR3[r0]
        ...
        store r31,#offsR31[r0]
        store sp,#offsSP[r0] ; save sp
        pop r1 ; save r1 through r1
        store r1,#offsR1[r0]
        pop r1 ; save r0 through r1
        store r1,#offsR0[r0]
        pop r1 ; save psw through r1
        store r1,#offsPSW[r0]
        pop r1 ; save pc through r1
        store r1,#offsPC[r0]

        ; Restore the context of the new running
        load r0,[running]
        load r1, #offsPC[r0] ; restore pc through the stack
        push r1
        load r1, #offsPSW[r0] ; restore psw through the stack
        push r1
        load sp,#offsSP[r0] ; restore sp
        load r31,#offsR31[r0] ; restore r31
        ... ; restore other regs
        load r1,#offsR1[r0]
        load r0,#offsR0[r0] ; restore r0
        ; and return
        iret
```

2. zadatak, prvi kolokvijum, jun 2018.

Neki procesor obrađuje prekide (hardverske i softverske) tako što tokom izvršavanja prekidne rutine koristi poseban stek koji se koristi u sistemskom, privilegovanom režimu rada procesora, u kome se izvršava kod kernela (čiji je deo i prekidna rutina). Taj stek alociran je u delu memorije koju koristi kernel, a na vrh tog steka ukazuje poseban registar SSP procesora koji je dostupan samo u privilegovanom režimu.

Prilikom obrade prekida, procesor na ovom steku čuva: pokazivač vrha korisničkog steka (SP) koji je koristio prekinuti proces, programsku statusnu reč (PSW) i programski brojač (PC), tim redom, ali ne i ostale programski dostupne registre. Prilikom povratka iz prekidne rutine instrukcijom `iret`, procesor restaurira ove registre sa sistemskog steka i vraća se u korisnički režim, a time i na korisnički stek.

Svaki proces ima sopstveni takav sistemski stek. Prilikom promene konteksta, ostale programski dostupne registre treba sačuvati na ovom sistemskom steku prekinutog procesa. U strukturi PCB postoji polje za čuvanje vrednosti

SSP steka procesa; pomeraj ovog polja u odnosu na početak strukture PCB označen je simboličkom konstantom `offsSSP`.

Procesor je RISC, sa *load/store* arhitekturom i ima sledeće programski dostupne registre: 32 registra opšte namene (R0..R31), SP, PSW i PC.

Sistem je multiprocesorski. Poseban registar procesora RPID služi za identifikaciju tekućeg procesa koji taj procesor trenutno izvršava i dostupan je samo u privilegovanom režimu. Potprogram `scheduler`, koji nema argumente, realizuje raspoređivanje, tako što smešta adresu PCB onog procesa koji se raspoređuje na datom procesoru (na kome se izvršava kod) u registar RPID.

1. Na asembleru datog procesora napisati kod prekidne rutine `dispatch` koja vrši promenu konteksta korišćenjem potprograma `scheduler`. Ova prekidna rutina poziva se sistemskim pozivom iz korisničkog procesa, pri čemu se sistemski poziv realizuje softverskim prekidom.
2. Pod pretpostavkom da procedura `scheduler` obezbeđuje međusobno isključenje pristupa deljenim strukturama podataka (npr. redu spremnih procesa) od strane različitih procesora, da li i ostatak prekidne rutine `dispatch` treba da obezbedi isto, tj. da li je u nju potrebno ugraditi *spin lock*? Precizno obrazložiti odgovor.

Rešenje

1.

```
dispatch: ; Save the current context
  push r0 ; save regs
  push r1
  ...
  push r31
  store ssp, #offsSSP[rpid] ; save ssp

  ; Select the next running process and store its PCB* in rpid
  call scheduler

  ; Restore the new context
  load ssp, #offsSSP[rpid] ; restore ssp
  pop r31
  pop r30 ; restore regs
  ...
  pop r0
  ; Return
  iret
```

2. Ne treba. Ova prekidna rutina pristupa samo registrima procesora, strukturi PCB tekućeg procesa i njegovom sistemskom steku. Kako su sve tri stvari korišćene isključivo od strane tog procesora (jer je taj proces raspoređen samo tom procesoru), nema potrebe za međusobnim isključenjem.

2. zadatak, prvi kolokvijum, mart 2017.

Neki procesor obrađuje prekide (hardverske i softverske) tako što tokom izvršavanja prekidne rutine koristi poseban stek koji se koristi u sistemskom, privilegovanom režimu rada procesora, u kome se izvršava kod kernela (čiji je deo i prekidna rutina). Taj stek alociran je u delu memorije koju koristi kernel, a na vrh tog steka ukazuje poseban registar SSP procesora koji je dostupan samo u privilegovanom režimu.

Prilikom obrade prekida, procesor na ovom steku čuva: pokazivač vrha korisničkog steka (SP) koji je koristio prekinuti proces, programsku statusnu reč (PSW) i programski brojač (PC), tim redom, ali ne i ostale programski dostupne registre. Prilikom povratka iz prekidne rutine instrukcijom `iret`, procesor restaurira ove registre sa sistemskog steka i vraća se u korisnički režim, a time i na korisnički stek.

Svaki proces ima takav sopstveni sistemski stek. Prilikom promene konteksta, ostale programski dostupne registre treba sačuvati na ovom sistemskom steku prekinutog procesa. U strukturi PCB postoji polje za čuvanje vrednosti

SSP steka procesa; pomeraj ovog polja u odnosu na početak strukture PCB označen je simboličkom konstantom `offsSSP`.

Procesor je RISC, sa *load/store* arhitekturom i ima sledeće programski dostupne registre: 32 registra opšte namene (R0..R31), SP, PSW i PC.

U kodu kernela postoji statički definisan pokazivač `running` koji ukazuje na PCB tekućeg procesa. Potprogram `scheduler`, koji nema argumente, realizuje raspoređivanje, tako što smešta PCB na koji ukazuje pokazivač `running` u listu spremnih procesa, a iz nje uzima jedan odabrani proces i postavlja pokazivač `running` na njega.

Na asembleru datog procesora napisati kod prekidne rutine `dispatch` koja vrši promenu konteksta korišćenjem potprograma `scheduler`. Ova prekidna rutina poziva se sistemskim pozivom iz korisničkog procesa, pri čemu se sistemski poziv realizuje softverskim prekidom.

Rešenje

```
dispatch:   ; Save the current context
            push r0 ; save regs
            push r1
            ...
            push r31
            load r0, running
            store ssp, #offsSSP[r0] ; save ssp

            ; Select the next running process
            call scheduler

            ; Restore the new context
            load r0, running
            load ssp, #offsSSP[r0] ; restore ssp
            pop r31
            pop r30 ; restore regs
            ...
            pop r0
            ; Return
            iret
```

2. zadatak, prvi kolokvijum, mart 2016.

Neki procesor obrađuje prekide (hardverske i softverske) u sistemskom, privilegovanom režimu rada procesora, u kome se izvršava kod kernela (čiji je deo i prekidna rutina), koristeći posebne registre SPC (*System Program Counter*), SPSW (*System Processor Status Word*) i SSP (*System Stack Pointer*) koji nisu dostupni u korisničkom (neprivilegovanom) režimu. Registre PC, PSW i SP, kao ni sve druge programski dostupne registre koji se koriste u neprivilegovanom režimu, procesor ne čuva nigde implicitno prilikom obrade prekida, jer se oni ni ne menjaju implicitno tokom skoka u prekidnu rutinu niti tokom izvršavanja te rutine (izvršavanje instrukcija u privilegovanom režimu koristi SPC, SPSW i SSP umesto PC, PSW i SP); njihove vrednosti se mogu menjati eksplicitno, uobičajenim instrukcijama u privilegovanom režimu (npr. `load` i `store`). Prema tome, izvršavanje u privilegovanom režimu koristi i poseban stek alociran u delu memorije koju koristi kernel. Taj stek je jedan i kernel ga ne menja.

Prilikom povratka iz prekidne rutine instrukcijom `iret` procesor ništa ne restaurira, samo se prebacuje na korišćenje registara PC, SP i PSW umesto njihovih sistemskih parnjaka SPC, SSP i SPSW.

Procesor je RISC, sa *load/store* arhitekturom i ima sledeće registre dostupne u neprivilegovanom režimu: 32 registra opšte namene (R0..R31), SP, PSW i PC. Poseban registar RX je dostupan samo u privilegovanom režimu rada procesora, pa ga korisnički procesi ne koriste i kernel ga može koristiti samo za svoje potrebe.

Registre dostupne u neprivilegovanom režimu treba sačuvati u odgovarajuća polja strukture PCB. U strukturi PCB postoje polja za čuvanje svih tih registara; pomeraji ovih polja u odnosu na početak strukture PCB označeni su simboličkim konstantama `offsPC`, `offsPSW`, `offsSP`, `offsR0`, ..., `offsR31`.

U kodu kernela postoji statički definisan pokazivač `running` koji ukazuje na PCB tekućeg procesa. Potprogram `scheduler`, koji nema argumente, realizuje raspoređivanje, tako što smešta PCB na koji ukazuje pokazivač `running` u listu spremnih procesa, a iz nje uzima jedan odabrani proces i postavlja pokazivač `running` na njega.

Na asembleru datog procesora napisati kod prekidne rutine `dispatch` koja vrši promenu konteksta korišćenjem potprograma `scheduler`. Ova prekidna rutina poziva se sistemskim pozivom iz korisničkog procesa, pri čemu se sistemski poziv realizuje softverskim prekidom.

Rešenje

```
dispatch:   ; Save the current context
            load rx,running
            store r0,#offsR0[rx] ; save regs
            store r1,#offsR1[rx]
            ...
            store r31,#offsR31[rx]
            store pc,#offsPC[rx] ; save pc
            store psw,#offsPSW[rx] ; same psw
            store sp,#offsSP[rx] ; save sp

            ; Select the next running process
            call scheduler

            ; Restore the new context
            load rx,running
            load sp,#offsSP[rx] ; restore sp
            load psw,#offsPSW[rx] ; restore psw
            load pc,#offsPC[rx] ; restore pc
            load r0,#offsR0[rx] ; restore regs
            load r1,#offsR1[rx]
            ...
            load r31,#offsR31[rx]
            ; Return
            iret
```

2. zadatak, prvi kolokvijum, mart 2015.

Neki procesor obrađuje prekide (hardverske i softverske) tako što tokom izvršavanja prekidne rutine koristi poseban stek koji se koristi u sistemskom. privilegovanom režimu rada procesora, u kome se izvršava kod kernela (čiji je deo i prekidna rutina). Taj stek alociran je u delu memorije koju koristi kernel, a na vrh tog steka ukazuje poseban registar procesora koji je dostupan samo u privilegovanom režimu. Taj stek je uvek isti i kernel ga ne menja.

Prilikom obrade prekida. procesor na ovom sistemskom steku čuva: pokazivač vrha korisničkog steka (SP) koji je koristio prekinuti program, programsku statusnu reg (PSW) i programski brojai (PC). tim redom, ali *ne* i ostale programski dostupne registre. Prilikom povratka iz prekidne rutine instrukcijom `iret`, procesor restaurira ove registre sa sistemskog steka i vraća se u korisnički režim, a time i na korisnički stek.

Procesor je RISC. sa *load/store* arhitekturom i ima sledeće programski dostupne registre: 32 registra opšte namene (R0..R31). SP. PSW i PC. Poseban registar RX je dostupan samo u privilegovanom režimu rada procesora. pa ga korisnički procesi ne koriste i kernel ga može koristiti samo za svoje potrebe.

Registre PC. SP. PSW i R0..R31 treba sačuvati u odgovarajuća polja strukture PCB. U strukturi PCB postoje polja za čuvanje svih tih registara; pomeraji ovih polja u odnosu na početak strukture PCB označeni su simboličkim konstantama `offsPC`, `offsPSW`, `offsSP`, `offsR0`, ..., `offsR31`.

U kodu kernela postoji statički definisan pokazivač `running` koji ukazuje na PCB tekućeg procesa. Potprogram `scheduler`. koji nema argumente. realizuje raspoređivanje. tako što smešta PCB na koji ukazuje pokazivač `running` u listu spremnih procesa, a iz nje uzima jedan odabrani proces i postavlja pokazivač `running` na njega.

Na asembleru datog procesora napisati kod prekidne rutine dispatch koja vrši promenu konteksta korišćenjem potprograma `scheduler`. Ova prekidna rutina poziva se sistemskim pozivom iz korisničkog procesa, pri čemu se sistemski poziv realizuje softverskim prekidom.

Rešenje

```
dispatch:   ; Save the current context
            load rx,running
            store r0,#offsR0[rx] ; save regs
            store r1,#offsR1[rx]
            ...
            store r31,#offsR31[rx]
            pop r0 ; save pc
            store r0,#offsPC[rx]
            pop r0 ; save psw
            store r0,#offsPSW[rx]
            pop r0 ; save original sp
            store r0,#offsSP[rx]

            ; Select the next running process
            call scheduler

            ; Restore the new context
            load rx,running
            load r0,#offsSP[rx] ; restore original sp
            push r0
            load r0,#offsPSW[rx] ; restore original psw
            push r0
            load r0,#offsPC[rx] ; restore pc
            push r0
            load r0,#offsR0[rx] ; restore regs
            load r1,#offsR1[rx]
            ...
            load r31,#offsR31[rx]
            ; Return
            iret
```

2. zadatak, prvi kolokvijum, maj 2015.

Neki procesor obrađuje prekide (hardverske i softverske) tako što tokom izvršavanja prekidne rutine koristi poseban stek. Taj stek alociran je u delu memorije koju koristi kernel, a na vrh tog steka ukazuje poseban registar procesora koji je dostupan samo u privilegovanom režimu rada procesora. Na tom steku samo se obrađuju prekidi, i uvek je isti (kernel ga ne menja).

Prilikom obrade prekida, procesor na ovom posebnom steku čuva: pokazivač vrha steka (SP) koji je koristio prekinuti tok kontrole, programsku statusnu reč (PSW) i programski brojač (PC), tim redom, ali *ne* i ostale programski dostupne registre. Prilikom povratka iz prekidne rutine instrukcijom `iret`, procesor restaurira ove registre sa ovog steka, a time prelazi na drugi stek.

Procesor ima dva režima rada: privilegovani (sistemski), u kome se izvršava kod kernela, i korisnički, u kome se izvršavaju korisnički procesi. Prilikom prihvatanja prekida, procesor obavezno prelazi u privilegovani režim rada. Prilikom povratka iz prekidne rutine, procesor ne menja implicitno režim rada, već ostaje u istom režimu rada u kome se nalazi prilikom obrade instrukcije `iret`. Ako je potrebno preći u korisnički režim rada, potrebno je izvršiti eksplicitnu instrukciju `setusr` koja prebacuje procesor u korisnički režim.

Operativni sistem za ovaj procesor je višenitni (engl. *multithreaded*). Sve funkcije jezgra, uključujući i obradu sistemskog poziva, raspoređivanje itd. obavljaju interne kernel niti. Svi sistemski pozivi izvršavaju se kao softverski prekid koji skače na prekidnu rutinu označenu kao `sys_call`, dok se sama identifikacija sistemskog poziva i njegovi

parametri prenose kroz registre procesora.

Procesor je RISC, sa *load/store* arhitekturom i ima sledeće programski dostupne registre: 32 registra opšte namene (R0..R31), SP, PSW i PC. Registre PC, SP, PSW i R0..R31 treba sačuvati u odgovarajuća polja strukture PCB. U strukturi PCB, koja postoji i ista je i za korisničke procese i za kernel niti, postoje polja za čuvanje svih tih registara; pomeraji ovih polja u odnosu na početak strukture PCB označeni su simboličkim konstantama `offsPC`, `offsPSW`, `offsSP`, `offsR0`, ..., `offsR31`.

U kodu kernela postoje dva statički definisana pokazivača, `userRunning` i `kernelRunning`, koji ukazuju na PCB tekućeg korisničkog procesa, odnosno kernel niti, respektivno.

1. Na asembleru datog procesora napisati prekidnu rutinu `sys_call`. Ova rutina treba samo da izvrši promenu konteksta sa tekućeg korisničkog procesa na tekuću kernel nit.
2. Na asembleru datog procesora napisati prekidnu rutinu `switch_to_user` (dovoljno je i precizno objasniti razlike u odnosu na rutinu `sys_call`) koja se poziva softverskim prekidom iz kernel koda, kada je kernel završio ceo posao i kada želi da promeni kontekst sa tekuće kernel niti na korisnički proces i vrati se u korisnički režim rada.

Rešenje

1.

```
sys_call:   ; Save the current context
            push r0 ; save r0 temporarily on the stack
            load r0,userRunning
            store r1,#offsR1[r0] ; save regs
            pop r1
            store r1,#offsR0[r0]
            store r2,#offsR2[r0]
            ...
            store r31,#offsR31[r0]
            pop r1 ; save pc
            store r1,#offsPC[r0]
            pop r1 ; save psw
            store r1,#offsPSW[r0]
            pop r1 ; save original sp
            store r1,#offsSP[r0]

            ; Restore the new context
            load r0,kernelRunning
            load r1,#offsSP[r0] ; restore original sp
            push r1
            load r1,#offsPSW[r0] ; restore original psw
            push r1
            load r1,#offsPC[r0] ; restore pc
            push r1
            load r1,#offsR1[r0] ; restore regs
            load r2,#offsR2[r0]
            ...
            load r31,#offsR31[r0]
            load r0,#offsR0[r0]
            ; Return (but stay in kernel mode)
            iret
```

2. Procedura `switch_to_user` izgleda potpuno analogno (skoro potpuno isto) kao i data procedura `sys_call`, samo što promenljive `userRunning` i `kernelRunning` zamenjuju mesta (uloge), a neposredno pre instrukcije `iret` stoji još samo instrukcija `setusr`.

3. zadatak, prvi kolokvijum, mart 2014.

Na asembleru datog procesora napisati kod operacije

```
void yield (PCB* cur, PCB* nxt);
```

poput one date na predavanjima, a koja u PCB na koji ukazuje prvi argument čuva kontekst izvršavanja koje se napušta i restaurira kontekst koji je sačuvan u PCB na koga ukazuje drugi argument.

Procesor je RISC, sa *load/store* arhitekturom i ima sledeće programski dostupne registre: 32 registra opšte namene (R0..R31), SP, PSW i PC. Registre (PSW, R0..R31 i SP) treba sačuvati u odgovarajuća polja strukture PCB. U strukturi PCB postoje polja za čuvanje svih tih registara; pomeraji ovih polja u odnosu na početak strukture PCB označeni su simboličkim konstantama `offsPSW`, `offsSP`, `offsR0`, ..., `offsR31`.

Rešenje

```
yield: ; Save current context
        push r0
        load r0,#cur[sp]
        store r1,#offsR1[r0] ; save r1
        pop r1 ; save r0 through r1
        store r1,#offsR0[r0]
        store r2,#offsR2[r0] ; save other regs
        store r3,#offsR3[r0]
        ...
        store r31,#offsR31[r0]
        store psw,#offsPSW[r0] ; save psw
        store sp,#offsSP[r0] ; save sp

        ; Restore new context
        load r0,#nxt[sp]
        load sp,#offsSP[r0] ; restore sp
        load psw,#offsPSW[r0] ; restore psw
        load r31,#offsR31[r0] ; restore r31
        ... ; restore other regs
        load r1,#offsR1[r0]
        load r0,#offsR0[r0] ; restore r0
        ; Return
        ret
```

3. zadatak, prvi kolokvijum, april 2014.

Na asembleru datog procesora napisati kod operacije

```
void yield (PCB* cur, PCB* nxt);
```

poput one date na predavanjima, a koja u PCB na koji ukazuje prvi argument čuva kontekst izvršavanja koje se napušta i restaurira kontekst koji je sačuvan u PCB na koga ukazuje drugi argument.

Procesor je RISC, sa *load/store* arhitekturom i ima sledeće registre dostupne korisničkim procesima: 32 registra opšte namene (R0..R31), SP, PSW i PC. Za podršku efikasnoj promeni konteksta, ovaj procesor ima sledeće:

- registar `base` kome se može pristupati samo u privilegovanom režimu rada procesora (nije dostupan korisničkim procesima i nije deo konteksta procesa); sve instrukcije i načini adresiranja koriste ovaj registar kao i svaki drugi registar opšte namene;
- instrukciju `saveregs` koja prepisuje sadržaj redom svih registara dostupnih korisničkim procesima, osim PC, u memoriju počev od adrese na koju ukazuje sadržaj registra `base`; ova instrukcija je dostupna samo u privilegovanom režimu rada procesora;
- instrukciju `loadregs` koja učitava redom sve registre dostupne korisničkim procesima, osim PC, iz memorije počev od adrese na koju ukazuje sadržaj registra `base`; ova instrukcija je dostupna samo u privilegovanom režimu rada procesora.

U strukturi PCB postoji polje za čuvanje svih registara dostupnih procesima; pomeraj ovog polja u odnosu na početak strukture PCB označen je simboličkom konstantom `offsContext`.

Rešenje

```
yield: ; Save current context
        load base,#cur[sp]
        add base,base,#offsContext
        saveregs
        ; Restore new context
        load base,#nxt[sp]
        add base,base,#offsContext
        loadregs
        ; Return
        ret
```

3. zadatak, prvi kolokvijum, septembar 2014.

Neki troadresni RISC procesor sa load/store arhitekturom, poput onog opisanog na predavanjima, poseduje 32 registra opšte namene, označenih sa $R0..R31$, statusnu reč PSW i pokazivač steka SP, koji su dostupni instrukcijama koje se izvršavaju u korisničkom režimu rada procesora, kao i dva posebna registra Rx i Rp koji su dostupni samo u privilegovanom (sistemskom) režimu rada procesora. Registar Rx se može koristiti kao i bilo koji registar $R0..R31$, i operativni sistem ga može koristiti proizvoljno za sopstvene potrebe (npr. prilikom promene konteksta). Registar Rp se može samo čitati, jer je ožičen tako da njegova vrednost predstavlja jedinstveni identifikator svakog pojedinačnog procesora u multiprocesorskom sistemu. Svi registri su 32-bitni, a adresibilna jedinica je bajt.

Za multiprocesorski sistem sa ovim procesorom pravi se operativni sistem. U kernelu tog sistema postoje sledeće definisane konstante i strukture podataka:

```
const int NumOfProcessors = ... // Number of processors
struct PCB;  // Process Control Block
PCB* runningProcesses[NumOfProcessors]; // Running processes
```

U strukturi PCB postoje polja za čuvanje vrednosti svih registara $R0..R31$, PSW i SP. Pomeraji ovih polja u odnosu na početak strukture PCB simbolički označavaju sa `offs_r0` itd. Niz `runningProcesses`, u svakom svom elementu n, sadrži pokazivač na PCB onog procesa koji se trenutno izvršava na procesoru broj n (n=0..*NumOfProcesses*-1). Na raspolaganju je operacija `schedule()` (bez argumenata), koja vrši odabir narednog procesa za izvršavanje na procesoru na kome se izvršava i koja upisuje adresu PCB tog procesa u odgovarajući element niza `runningProcesses`.

1. Na asembleru datog procesora napisati operaciju `dispatch()` (bez argumenata) koja čuva kontekst tekućeg izvršavanja i restaurira kontekst izvršavanja procesa kome treba dati procesor. U asembleru datog procesora može se koristiti identifikator statički alociranog podatka iz C programa, pri čemu se takva upotreba prevodi u konstantu sa vrednošću adrese tog podatka. Ova operacija se izvršava u kodu kernela, u sistemskom režimu. U ovu operaciju ulazi se iz prekidne rutine koja obrađuje sistemski poziv, pa su prekidi već maskirani (ne treba ih maskirati i demaskirati).
2. Da li je u ovu operaciju neophodno ubaciti kod za međusobno isključenje konkurentnog izvršavanja od strane različitih procesora tehnikom uposlenog čekanja (*spin lock*)? Obrazložiti.

Rešenje

1.

```
dispatch: ; Save the current context
          LOAD Rx,Rp    ; Rx:=# of current processor
          SHL Rx,2      ; Rx:=Rx*4
          LOAD Rx,#runningProcesses[Rx]; Rx:=&running process' PCB
          STORE #offs_r0[Rx],R0 ; store R0
          STORE #offs_r1[Rx],R1 ; store R1
          ...
          STORE #offs_psw[Rx],PSW ; store PSW
```

```
STORE #offs_sp[Rx],SP ; store SP
; Call scheduler
CALL schedule
; Restore the next context
LOAD Rx,Rp    ; Rx:=# of current processor
SHL Rx,2      ; Rx:=Rx*4
LOAD Rx,#runningProcesses[Rx]; Rx:=&next process' PCB
LOAD R0,#offs_r0[Rx]    ; restore R0
LOAD R1,#offs_r1[Rx]    ; restore R1
...
LOAD PSW,#offs_psw[Rx] ; restore PSW
LOAD SP,#offs_sp[Rx]    ; restore SP
RTS                  ; return from subroutine
```

2. Nije potrebno, jer svaki procesor jedini pristupa samo svom odgovarajućem elementu `n` niza `runningProcesses`, pa nema potencijalnog konflikta između procesora (nema deljenih podataka u memoriji kojima pristupaju različiti procesori u ovoj operaciji).

3. zadatak, prvi kolokvijum, mart 2013.

U nekom operativnom sistemu svi sistemski pozivi izvršavaju se kao softverski prekid koji skače na prekidnu rutinu označenu kao `sys_call`, dok se sama identifikacija sistemskog poziva i njegovi parametri prenose kroz registre procesora. Jezgro tog operativnog sistema je višenitno – poseduje više internih kernel niti koje obavljaju različite poslove: izvršavaju uporedne I/O operacije, vrše druge interne poslove jezgra, pa čak postoje i niti koje obavljaju sve potrebne radnje prilikom promene konteksta korisničkih procesa (osim samog čuvanja i restauracije konteksta procesora), kao što su smeštanje PCB korisničkog procesa koji je do tada bio tekući u odgovarajući red (spremnih ili blokiranih, u zavisnosti od situacije), izbor novog tekućeg procesa iz skupa spremnih, promenu memorijskog konteksta, obradu samog konkretnog sistemskog poziva, itd. Prilikom obrade sistemskog poziva `sys_call`, prema tome, treba samo oduzeti procesor tekućem korisničkom procesu i dodeliti ga tekućoj kernel niti.

Na PCB tekućeg korisničkog procesa ukazuje globalni pokazivač `runningUserProcess`, a na PCB tekuće interne niti jezgra ukazuje globalni pokazivač `runningKernelThread`. I interne niti jezgra vrše promenu konteksta između sebe na isti način, pozivom softverskog prekida koji ima istu internu strukturu kao i rutina `sys_call`.

Prilikom obrade prekida, procesor čuva ceo kontekst svog izvršavanja, odnosno sve programski dostupne registre (uključujući i PC, SP i PSW) u memoriji. Međutim, kako se ne bi dogodilo prekoračenje steka korisničkog procesa ili stranična greška prilikom tog postupka, procesor ove registre čuva u strukuturu u memoriji čija se adresa nalazi u posebnom namenskom registru procesora SPX. Prilikom povratka iz prekidne rutine, procesor sam restaurira kontekst izvršavanja iz iste ove strukture na koju tada ukazuje SPX. Ovaj registar dostupan je samo iz privilegovanog sistemskog režima rada procesora u koji se prelazi softverskim prekidom.

I korisnički procesi i interne niti jezgra u svojim PCB strukturama imaju polje pod nazivom context u kojima se čuva kontekst procesora. Pomeraj ove strukture u odnosu na početak PCB dat je simboličkom vrednošću `offsContext`. Procesor je RISC, sa *load/store* arhitekturom.

Upotreba imena globalne promenljive iz C koda u asembleru datog procesora predstavlja memorijsku adresu te globalne statičke promenljive. Deo memorije koju koristi jezgro (kod i podaci) preslikava se u virtuelni adresni prostor svih korisničkih procesa, tako da prilikom obrade sistemskog poziva i prelaska u kod kernela nema promene memorijskog konteksta.

Na asembleru datog procesora napisati prekidnu rutinu `sys_call`. Obrazložiti rešenje!

Rešenje

Kada se pri obradi prekida kontekst procesora prepisuje u memoriju, bitno je da u registru SPX bude adresa polja *context* iz PCB-a tekuće niti koja gubi procesor. Isto važi pri svakoj promeni konteksta, pa i pri napuštanju kernel niti. Vrednost samog SPX se ne mora čuvati, jer se jednostavno restaurira dodavanjem pomeraja polja *context* na vrednost adrese PCB-a. Dakle, jedino što je prilikom promene konteksta potrebno uraditi jeste postaviti SPX na vrednost adrese polja *context* niti koja dobija procesor. Tako rutina `sys_call` izgleda ovako:

```
sys_call: load spx, [runningKernelThread]
          add spx, #offsContext
          iret
```

Povratak iz rutine će restaurirati kontekst kernel niti iz strukture context PCB-a niti koja je dobila procesor, jer na tu strukturu tada ukazuje SPX. Potpuno analogno izgleda i rutina kojom se kontrola iz kernel niti predaje korisničkoj niti, samo što je umesto `runningKernelThread` upotrebljen `runningUserProcess`.

3. zadatak, prvi kolokvijum, april 2013.

U nekom operativnom sistemu svi sistemski pozivi izvršavaju se kao softverski prekid koji skače na prekidnu rutinu označenu kao `sys_call`, dok se sama identifikacija sistemskog poziva i njegovi parametri prenose kroz registre procesora. Prilikom obrade sistemskog poziva `sys_call` treba uraditi redom sledeće:

1. Sačuvati kontekst tekućeg korisničkog procesa u njegov PCB. Procesor je RISC, sa *load/store* arhitekturom i ima sledeće programski dostupne registre: 32 registra opšte namene (R0..R31), SP, PSW i PC. Prilikom obrade prekida, procesor prelazi u sistemski (zaštićeni, kernel) režim, a na steku čuva samo PSW i PC. Na PCB tekućeg korisničkog procesa ukazuje globalni pokazivač koji se nalazi na adresi simbolički označenoj sa `running`. Ostale registre (R0..R31 i SP) treba sačuvati u odgovarajuća polja strukture PCB. U strukturi PCB postoje polja za čuvanje svih tih registara; pomeraji ovih polja u odnosu na početak strukture PCB označeni su simboličkim konstantama `offsSP`, `offsR0`, ..., `offsR31`.
2. Preći na stek kernela u kome se izvršava ceo kernel kod. Na vrh tog steka (zapravo dno, pošto je stek prazan inicijalno, kao i posle svakog izlaska iz kernel koda) ukazuje pokazivač na lokaciji simbolički označenoj sa `kernelStack`.
3. Dozvoliti spoljašnje maskirajuće prekide instrukcijom `inte`, pošto je kod kernela i svih prekidnih rutina napravljen tako da se pojava prekida samo pamti u softverskim indikatorima, a kernel kod obrađuje prekid odloženo, kada je to bezbedno i odgovarajuće.
4. Pozvati potprogram na adresi simbolički označenoj sa `kernel`. Ovaj potprogram (bez argumenata) obavlja sve potrebne poslove, kao što su obrada sistemskog poziva i raspoređivanje (nakon čega `running` pokazuje na PCB novog tekućeg procesa).
5. Po povratku iz potprograma `kernel`, maskirati prekide instrukcijom `intd`, povratiti kontekst novog tekućeg procesa iz njegovog PCB-a, i vratiti se u korisnički režim i proces iz sistemskog poziva. Instrukcija `iret` prebacuje procesor u korisnički režim i vraća sačuvane registre PC i PSW sa tekućeg steka. Na asembleru datog procesora napisati prekidnu rutinu `sys_call`.

Rešenje

```
sys_call: ; Save current context
          push r0
          load r0,[running]
          store r1,#offsR1[r0] ; save r1
          pop r1 ; save r0 through r1
          store r1,#offsR0[r0]
          store r2,#offsR2[r0] ; save other regs
          store r3,#offsR3[r0]
          ...
          store r31,#offsR31[r0]
          store sp,#offsSP[r0] ; save sp

          ; Switch to kernel code
          load sp,[kernelStack] ; switch to kernel stack
          inte ; enable interrupts
          call kernel ; go to kernel code
          intd ; disable interrupts

          ; Restore new context
          load r0,[running]
          load sp,#offsSP[r0] ; restore sp
```

```
load r31,#offsR31[r0] ; restore r31
... ; restore other regs
load r1,#offsR1[r0]
load r0,#offsR0[r0] ; restore r0
; Return
iret
```

3. zadatak, prvi kolokvijum, septembar 2011.

U implementaciji jezgra nekog jednoprocesorskog *time-sharing* operativnog sistema, radi pojednostavljenja celog mehanizma promene konteksta, primenjeno je sledeće neobično rešenje. Promena konteksta vrši se isključivo kao posledica prekida, na samo jednom mestu u kodu koji se izvršava na prekid. Prekidi dolaze od raznih uređaja, u najmanju ruku od vremenskog brojača, jer on u svakom slučaju generiše prekid u konačnom vremenu, bilo zbog toga što je tekućoj niti isteklo dodeljeno procesorsko vreme, ili iz nekog drugog razloga zbog koga je vremenski brojač od strane sistema bio startovan da meri interval i generiše prekid kada on istekne. Zbog toga tekuća nit nikada ne gubi procesor sinhrono, čak ni kada poziva blokirajuću operaciju npr. na semaforu. Umesto toga, ukoliko je potrebno da se nit blokira u nekom blokirajućem pozivu, nit se samo „označi“ blokiranom i nastavlja sa izvršavanjem (uposlenim čekanjem) sve dok ne stigne sledeći prekid. Kada takav prekid stigne, kod koji se poziva proverava stanje niti koja je prekinuta i ako je ona označen kao blokirana, vrši samu promenu konteksta.

Na primer, implementacija operacije `wait()` na semaforu u ovom sistemu izgleda ovako:

```
class Thread {
  int isBlocked;
  jmpbuf context;
  static Thread* running;
  ...;
};

void Semaphore::wait () {
  lock();  // Disable interrupts
  if (--val<0) {
    Thread::running->isBlocked = 1;// Mark as blocked,
    queue->put(Thread::running);   // put it in the semaphore queue,
    unlock();                      // enable interrupts,
    while (Thread::running->isBlocked); // and then busy-wait
  }                               // until it is preempted, suspended, and
                                  // then resumed later
  else unlock();  // Enable interrupts
}
```

Korišćenjem standardnih bibliotečnih operacija `setjmp()` i `longjmp()`, realizovati prekidnu rutinu `yield()` koja vrši opisanu promenu konteksta na prekid. Klasa `Schedule` koja realizuje red spremnih niti sa raspoređivanjem ima interfejs kao u školskom jezgru.

Rešenje

```
interrupt void yield () {
  if (setjmp(Thread::running->context)==0) {

    if (!Thread::running->isBlocked) Scheduler::put(Thread::running);
    Thread::running = Scheduler::get();
    Thread::running->isBlocked = 0;

    longjmp(Thread::running->context,1);

  }
}
```

4. zadatak, prvi kolokvijum, april 2010.

U nekom operativnom sistemu svi sistemski pozivi izvršavaju se kao softverski prekid koji skače na prekidnu rutinu označenu kao `sys_call`, dok se sama identifikacija sistemskog poziva i njegovi parametri prenose kroz registre procesora. Jezgro tog operativnog sistema je višenitno – poseduje više internih kernel niti koje obavljaju različite poslove: izvršavaju uporedne I/O operacije, vrše druge interne poslove jezgra, pa čak postoje i niti koje obavljaju sve potrebne radnje prilikom promene konteksta korisničkih procesa (osim samog čuvanja i restauracije konteksta procesora), kao što su smeštanje PCB korisničkog procesa koji je do tada bio tekući u odgovarajući red (spremnih ili blokiranih, u zavisnosti od situacije), izbor novog tekućeg procesa iz skupa spremnih, promenu memorijskog konteksta, obradu samog konkretnog sistemskog poziva, itd. Prilikom obrade sistemskog poziva `sys_call`, prema tome, treba samo oduzeti procesor tekućem korisničkom procesu i dodeliti ga tekućoj kernel niti.

Na PCB tekućeg korisničkog proces a ukazuje globalni pokazivač `runningUserProces`, a na PCB tekuće interne niti jezgra ukazuje globalni pokazivač `runningKernelThread`. I interne niti jezgra vrše promenu konteksta između sebe na isti način, pozivom softverskog prekida koji ima istu internu strukturu kao i rutina `sys_call`. I korisnički iprocesi i interne niti jezgra u svojim PCB strukturama imaju polje u kojima se čuva SP i čiji je pomeraj u odnosu na početak PCB dat simboličkom vrednošću `offsSP`.

Procesor je RISC, sa *load/store* arhitekturom. Od registara opšte namene poseduje registarski fajl sa registrima R0..R63, SP, PSW i PC. Prilikom prihvatanja prekida i pre skoka u prekidnu rutinu, hardver procesora na steku čuva sve programski dostupne registre, osim SP.

Napomene: Upotreba imena globalne promenljive iz C koda u asembleru datog procesora predstavlja memorijsku adresu te globalne statičke promenljive. Deo memorije koju koristi jezgro (kod i podaci) preslikava se u virtuelni adresni prostor svih korisničkih procesa, tako da prilikom obrade sistemskog poziva i prela ska u kod kernela nema promene memorijskog konteksta.

Na asembleru datog procesora napisati prekidnu rutinu `sys_call`.

Rešenje

```
sys_call: load r0, [runningUserProcess]
          store sp, offsSP[r0]
          load r0, [runningKernelThread]
          load sp, offsSP[r0]
          iret
```

Sistemski pozivi

1. zadatak, drugi kolokvijum, maj 2022.

Koristeći samo sistemske pozive *fork*, *wait* i *exit*, kao i funkciju *printf*, napisati C program koji pronalazi maksimalnu vrednost svih elemenata ogromne celobrojne matrice dimenzije M puta N tako što uporedo pronalazi maksimum u svakoj vrsti matrice (maksimum svake vrste pronalazi u po jednom od pokrenutih procesa-dece), a onda pronalazi maksimum tih maksimuma. U slučaju greške, ovaj program treba da ispiše poruku o grešci i vrati status -1, a u slučaju uspeha treba da ispiše pronađeni maksimum i vrati status 0. Pretpostaviti da je matrica već nekako inicijalizovana.

```
const int M = ..., N = ...;
extern int mat[M][N];
```

Rešenje

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

const int M = ..., N = ...;
extern int mat[M][N];

int max(int i) {
    int m = mat[i][0];
    for (int j = 1; j < N; j++) {
        if (mat[i][j] > m) {
            m = mat[i][j];
        }
    }
    return m;
}

int main() {
    for (int i = 0; i < M; i++) {
        pid_t pid = fork();
        if (pid < 0) {
            printf("Error: Cannot create a child process.\n");
            exit(-1);
        }
        if (pid == 0) {
            exit(max(i));
        }
    }
    int max;
    wait(&max);
    for (int i = 1; i < M; i++) {
        int m;
        wait(&m);
        if (m > max) {
            max = m;
        }
    }
    printf("Max: %d\n", max);
    exit(0);
}
```

1. zadatak, drugi kolokvijum, jun 2022.

Koristeći samo sistemske pozive *fork*, *wait*/*waitpid* i *exit*, kao i funkciju *printf*, napisati C program koji pronalazi maksimalnu vrednost svih elemenata ogromne matrice dimenzije M puta N elemenata tipa `double` tako što uporedo pronalazi maksimum u svakoj vrsti matrice (maksimum svake vrste pronalazi u po jednom od pokrenutih procesadece), a onda pronalazi maksimum tih maksimuma. U slučaju greške, ovaj program treba da ispiše poruku o grešci i vrati status -1, a u slučaju uspeha treba da ispiše pronađeni maksimum i vrati status 0.

Pretpostaviti da je matrica već nekako inicijalizovana.

```
const int M = ..., N = ...;
extern double mat[M][N];
```

Rešenje

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

#define handle_error(msg) do { \
    printf(msg); \
    exit(-1); \
} while (0)

const int M = ..., N = ...;
extern double mat[M][N];
pid_t pids[M];

int max(int i) {
    double m = mat[i][0];
    int mj = 0;
    for (int j = 1; j < N; j++) {
        if (mat[i][j] > m) {
            m = mat[i][j];
            mj = j;
        }
    }
    return mj;
}

int main() {
    for (int i = 0; i < M; i++) {
        pid_t pid = pids[i] = fork();
        if (pid < 0) {
            handle_error("Error: Cannot create a child process.\n");
        }
        if (pid == 0) {
            exit(max(i));
        }
    }
    double max;
    int mj;
    if (waitpid(pids[0], &mj) < 0) {
        handle_error("Error waiting a child process.\n");
    }
    max = mat[0][mj];
    for (int i = 1; i < M; i++) {
```

```
        if (waitpid(pids[i], &mj) < 0) {
            handle_error("Error waiting a child process.\n");
        }
        if (mat[i][mj] > max) {
            max = mat[i][mj];
        }
    }
    printf("Max: %f\n", max);
    exit(0);
}
```

1. zadatak, kolokvijum, oktobar 2020.

U sistemu Windows postoji sistemski poziv

```
int spawnvp (int mode, char *path, const char **argv);
```

koji radi isto što i kombinacija poziva *fork* i *exec* na sistemima nalik sistemu Unix: kreira proces dete pozivajućeg procesa nad programom u izvršivom fajlu zadatom putanjom u argumentu `path`. Parametar `argv` predstavlja niz pokazivača na nizove znakova (*null- terminated strings*) koji predstavljaju argumenate poziva programa koji treba izvršiti. Po konvenciji, prvi argument pokazuje na naziv fajla samog programa koji se izvršava. Ovaj niz pokazivača mora da se završi elementom sa vrednošću `NULL`. Ovaj sistemski poziv vraća 0 u slučaju uspeha, a vrednost manju od 0 u slučaju greške. Parametar `mode` definiše modalitet kreiranja procesa deteta sledećim simboličkim konstantama:

P_OVERLAY	Overlays the parent process with child, which destroys the parent. This has the same effect as the exec functions.
P_WAIT	Suspends the parent process until the child process has finished executing (synchronous spawn).
P_NOWAIT	Continues to execute the parent process concurrently with child process (asynchronous spawn).
P_DETACH	The child is run in the background without access to the console (asynchronous spawn).

Korišćenjem ovog sistemskog poziva realizovati sledeću funkciju:

```
int multispawn (int number, const char* path, const char* args[]);
```

Ova funkcija treba da kreira zadati broj (`number`) procesa dece, i svaki od tih procesa dece treba da izvršava isti program zadat u fajlu sa stazom `path`, sa samo po jednim argumentom. Argumenti tih procesa dece zadati su redom u prvih `number` elemenata niza `args`. Ova funkcija treba da vrati broj uspešno kreiranih procesa dece i da odmah vrati kontrolu pozivaocu, ne čekajući da se ti procesi deca završe.

Rešenje

```
#include <process.h>

int multispawn (int number, const char* path, const char* args[]){
  int ret = 0;
  // Prepare the arguments for the children:
  const char* childArgs[3];
  childArgs[0] = path;
  childArgs[2] = NULL;
  for (int i=0; i<number; i++) {
    childArgs[1] = args[i];
    // Create a child:
    int status = spawnvp(P_NOWAIT,path,childArgs);
    if (status>=0) ret++;
  }
  return ret;
}
```

1. zadatak, kolokvijum, jul 2020.

Dat je neki veliki jednodimenzioni niz `data` veličine `N`, čiji su elementi tipa `Data` (deklaracije dole). Jedan element niza obrađuje procedura `processData`. Ovaj niz potrebno je obraditi pomoću `n` uporednih niti (procedura `parallelProcessing`), gde je `n` zadati parametar, tako što svaka od tih `n` uporednih niti iterativno obrađuje približno isti broj elemenata ovog velikog niza. Drugim rečima, niz treba particionisati na `n` disjunktnih podnizova, što približnije jednakih, i te particije obrađivati uporedo, kako bi se niz obradio paralelno na višeprocesorskom sistemu.

Koristiti sistemski poziv `thread_create` koji kreira nit nad funkcijom na koju ukazuje prvi parametar, i pozivu te funkcije dostavlja drugi parametar ovog sistemskog poziva. Ova funkcija vraća identifikator kreirane niti (pozitivna vrednost), odnosno status greške (negativna vrednost). Ignorisati moguće greške.

```
const int N = ...;
class Data;
Data data[N];
void processData(Data*);
void parallelProcessing (int n);

int thread_create (void (*thread_body)(void*), void* arg);
```

Rešenje

```
typedef struct ChunkDesc {
  int offset, size;
} ChunkDesc;

void dataProcessor (void* chunk) {
  ChunkDesc* cnk = (ChunkDesc*)chunk;
  int end = cnk->offset+cnk->size;
  for (int i=cnk->offset; i<end; i++)
      processData(&data[i]);
  delete cnk;
}

void parallelProcessing (int n) {
   int chunkSz = N/n;
   int offset = 0;
   for (int i=0; i<n; i++) {
     int myChunkSz = chunkSz;
     if (i<N%n) myChunkSz++;
     ChunkDesc* chunk = new ChunkDesc;
     chunk->offset = offset; chunk->size = myChunkSz;
     offset+= myChunkSz;
     thread_create(dataProcessor,chunk);
   }
}
```

4. zadatak, kolokvijum, avgust 2020.

Na raspolaganju su sledeći standardni POSIX sistemski pozivi:

- `int close (int fd)`: zatvara fajl sa zadatim deskriptorom;
- `int write (int fd, const void *buffer, size_t size)`: upisuje dati sadržaj u fajl sa zadatim deskriptorom; ako je fajl otvoren sa postavljenim O_APPEND, onda se pre upisa tekuća pozicija implicitno pomera na kraj fajla i upis uvek vrši iza kraja fajla, proširujući sadržaj;
- `int open(const char *pathname, int flags, mode_t mode)`: otvara fajl sa zadatom putanjom; argument flags mora da uključi jedno od sledećih prava pristupa: *O_RDONLY*, *O_WRONLY*, ili *O_RDWR*. Ako je uključen i fleg *O_CREAT*, fajl će biti kreiran ukoliko ne postoji; ako je pritom uključen i *O_EXCL*, a fajl

već postoji, funkcija će vratiti grešku (-1), a kod greške postavljen u globalnoj sistemskoj promenljivoj *errno* biće jednak *EEXIST*. Ako je uključen *O_TRUNC*, i ako fajl već postoji, njegov sadržaj se pri otvaranju briše. Argument *mode* definiše prava pristupa za fajl koji se kreira samo u slučaju da je uključen *O_CREAT* (tada je i obavezan), i to na sledeći način:

S_IRWXU	0x0700	user (file owner) has read, write and execute permission
S_IRUSR	0x0400	user has read permission
S_IWUSR	0x0200	user has write permission
S_IXUSR	0x0100	user has execute permission
S_IRWXG	0x0070	group has read, write and execute permission
S_IRGRP	0x0040	group has read permission
S_IWGRP	0x0020	group has write permission
S_IXGRP	0x0010	group has execute permission
S_IRWXO	0x0007	others have read, write and execute permission
S_IROTH	0x0004	others have read permission
S_IWOTH	0x0002	others have write permission
S_IXOTH	0x0001	others have execute permission

U nekom programu koristi se u uređeno binarno stablo sa čvorom tipa strukture `Node` sa celim brojevima kao sadržajem čvora (polje `contents` tipa `int`), tako da su u levom podstablu (polje `left` tipa `Node*`) svakog čvora brojevi koji su manji, a u desnom (polje `right`) oni koji su veći ili jednaki od sadržaja datog čvora. Implementirati funkciju koja u fajl sa datim imenom upisuje niz celih brojeva iz ovakvog stabla sa datim korenom, uređen neopadajuće (funkcija vraća 0 u slučaju uspeha, vrednost manju od 0 neku grešku). Ako fajl ne postoji, treba ga kreirati sa pravima na čitanje i upis za vlasnika, a samo na čitanje za sve ostale.

```
int save (const char* fname, Node* root);
```

Rešenje

```
int write (int fd, Node* root) {
  if (!root) return 0;
  int ret = write(root->left);
    if (ret<0) return ret;
  ret = write(fd,&(root->contents),sizeof(int));
    if (ret<0) return ret;
  ret = write(root->right);
    return ret;
}

int save (const char* fname, Node* root) {
  int fd = open(fname,O_WRONLY|O_CREAT|O_TRUNC,
                      S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
  if (fd<0) return fd;
  int ret = write(fd,root);
  close(fd);
  return ret;
}
```

2. zadatak, kolokvijum, septembar 2020.

Neki operativni sistem podržava sledeće sistemske pozive za upravljanje nitima:

- `fork()`: kreira novu nit-dete kao klon roditelja, poput Unix sistemskog poziva *fork* (samo što kreira nit, a ne proces); u kontekstu roditelja vraća ID kreirane niti-deteta, a u kontekstu deteta vraća 0;
- `exit()`: gasi pozivajuću nit;
- `wait(int pid)`: suspenduje pozivajuću nit sve dok se nit dete te niti sa datim ID ne završi (ne ugasi se); argument 0 suspenduje pozivajuću nit sve dok se svi njeni potomci ne završe.

Korišćenjem ovih sistemskih poziva, realizovati operaciju `cobegin()` koja prima tri argumenta: niz pokazivača na funkcije, niz argumenata tipa `void*` i ceo broj n, a koja pokreće uporedno izvršavanje n niti nad funkcijama iz datog

niza, pri čemu funkciju iz i-tog elementa niza pokazivača na funkcije poziva sa i-tim elementom u nizu argumenata, a vraća kontrolu pozivaocu tek kada se sve te niti (odnosno funkcije) završe. Obratiti pažnju na to da je nit koja poziva ovu funkciju `cobegin` možda ranije pokrenula neke druge niti koje ne treba čekati pre izlaska iz `cobegin`. Ignorisati sve moguće greške.

```
typedef void (*PF)(void*);

void cobegin (PF f[], void* af[], int n);
```

Rešenje

```
typedef void (*PF)(void*);

void cobegin(PF f[], void* af[], int n) {
    int* ids = new int[n];
    for (int i = 0; i < n; i++)
        if ((ids[i] = fork()) == 0) {
            pf[i](af[i]);
            exit();
        }
    for (int i = 0; i < n; i++)
        wait(ids[i]);
    delete ids;
}
```

3. zadatak, prvi kolokvijum, mart 2019.

U nekom operativnom sistemu postoje sledeći sistemski pozivi:

- `fork()`, `execlp(const char*)`: kao u sistemu Unix i njemu sličnim;
- `int wait(int pid, unsigned timeout)`: suspenduje pozivajući roditeljski proces dok se ne završi proces-dete sa zadatim PID, ali ga suspenduje najduže onoliko koliko je zadato drugim argumentom (vreme čekanja u milisekundama). Ako je vrednost prvog argumenta NULL, pozivajući proces se suspenduje dok se ne završe sva njegova deca (ili ne istekne vreme čekanja); ako je vrednost drugog argumenta 0, poziv odmah vraća rezultat, bez čekanja. Vraćena vrednost 0 označava da su svi procesi koji su se čekali završili (pre isteka vremena čekanja); vraćena vrednost veća od 0 znači da je vreme čekanja isteklo pre nego što je neki od procesa koji su se čekali završili.
- `int kill(int pid)`: gasi proces sa zadatim PID.

Svi ovi sistemski pozivi vraćaju negativan kod greške u slučaju neuspeha.

Na jeziku C napisati program koji se poziva sa jednim argumentom koji predstavlja stazu do *exe* fajla. Ovaj program treba da kreira N procesa koji izvršavaju program u *exe* fajlu zadatom argumentom (N je konstanta definisana u programu), a potom da čeka da se svi ti procesi-deca završe u roku od 5 sekundi. Sve procese-decu koji se nisu završili u tom roku treba da ugasi. Obraditi sve greške u sistemskim pozivima ispisom odgovarajuće poruke.

Rešenje

```
const int N = ...;
const unsigned timeout = 5000;
int pid[N];

int main (int argc, const* char argv[]) {
  if (argc<2) {
    printf("Error: Missing argument for the program to run.\n");
    exit(-1);
  }

  int ret = 0;  // Status to return on exit
```

```
  // Create children:
  for (int i=0; i<N; i++) {
    pid[i] = fork();
    if (pid[i] < 0) {
      printf("Error: Failed to create a child process number %d.\n",i);
      ret = -2;
    } else
    if (pid[i] == 0) {
      execlp(argv[1]);
      printf("Error: Failed to execute the program for the child process
number %d.\n",i);
      exit(-1);  // Terminate the child, not the parent
    }
  }

  // Wait for all children:
  int stat = wait(NULL,timeout);
  if (stat<0) {
    printf("Error: Failed to wait for the children processes.\n");
    ret -= 4;
  }

  if (stat == 0) {
    printf("All children completed in time.\n",i);
    exit(0);
  }

  // Kill all incomplete children:
  for (int i=0; i<N; i++) {
    stat = wait(pid[i],0);
    if (stat == 0) continue;  // Child completed
    if (stat<0) {
      printf("Error: Failed to wait for the child process number %d.\n",i);
      if (ret > -8) ret -= 8;
    }
    stat = kill(pid[i]);
    if (stat<0) {
      printf("Error: Failed to kill the child process number %d.\n",i);
      if (ret > -16) ret -= 16;
    }
  }

  exit(ret);
}
```

3. zadatak, prvi kolokvijum, maj 2019.

U nekom operativnom sistemu postoje sledeći sistemski pozivi:

- `int fork()`: kao u sistemu Unix i njemu sličnim, samo što ne kreira proces, nego nit; u kontekstu niti-roditelja vraća identifikator niti-deteta (PID), a u kontekstu niti-deteta vraća 0;
- `exit()`: završava pozivajuću nit;
- `wait(int pid)`: suspenduje pozivajuću roditeljsku nit dok se ne završi nit-dete sa zadatim PID.

Svi ovi sistemski pozivi vraćaju negativan kod greške u slučaju neuspeha.

Data je globalna matrica `mat` celih brojeva (`int`), dimenzija $M \times N$ (M i N su konstante). Potrebno je napisati

potprogram `par_sum` koji treba da izračuna ukupan zbir svih elemenata matrice `mat`, ali na sledeći način: najpre treba da izračuna zbir svake, i-te vrste ove matrice, uporedo, u M uporednih niti (zbir jedne vrste izračunava jedna nit), i da taj zbir smesti u i-ti element jednog pomoćnog niza; potom treba da sabere ovako izračunate zbirove vrsta i vrati rezultat. Ignorisati eventualne greške i prekoračenje.

```
const int M = ..., N = ...;
int mat[M][N];
int par_sum ();
```

Rešenje

```
const int M = ..., N = ...;
int mat[M][N];
int sums[M];
int pid[M];

void sum (int row) {
  int s = 0;
  for (int i=0; i<N; i++)
    s += mat[row][i];
  sums[row] = s;
}

int par_sum () {
  for (int i=0; i<M; i++) {
    pid[i] = fork();
    if (pid[i]==0) {
      sum(i);
      exit();
    }
  }

  int s = 0;
  for (int i=0; i<M; i++) {
    wait(pid[i]);
    s += sums[i];
  }
  return s;
}
```

3. zadatak, prvi kolokvijum, jun 2019.

U nekom operativnom sistemu postoje sledeći sistemski pozivi:

- `int thread_create(void(*)(void*),void*)`: kreira nit nad funkcijom na koju ukazuje prvi argument; ta funkcija prima jedan argument tipa `void*` i ne vraća rezultat; novokreirana nit poziva tu funkciju sa stvarnim argumentom jednakim drugom argumentu ovog sistemskog poziva; sistemski poziv vraća PID kreirane niti;
- `void wait(int pid)`: suspenduje pozivajuću roditeljsku nit dok se ne završi nit-dete sa zadatim PID; ako je vrednost argumenta 0, pozivajuća nit se suspenduje dok se ne završe sve niti-deca.

Korišćenjem bibliotečne funkcije `memcpy`, implementirati funkciju `par_memcpy` istog potpisa koja će iskoristiti paralelizam na multiprocesorskom sistemu i velike segmente memorije kopirati paralelno. Ova funkcija će to uraditi tako što će kreirati potreban broj uporednih niti, a svaka nit će kopirati po jedan blok (particiju traženog segmenta memorije) veličine `BLOCK_SIZE` (poslednja nit možda manje od toga). Ignorisati eventualne greške i prekoračenja.

```
void* memcpy (void* destination, const void* source, size_t num);
void* par_memcpy (void* destination, const void* source, size_t num);
size_t BLOCK_SIZE = ...;
```

Rešenje

```
struct copy_task {
  void* dst;
  const void* src;
  size_t num;
};

void copy_block (void* task) {
  copy_task* t = (copy_task*)task;
  memcpy(t->dst,t->src,t->num);
}

void* par_memcpy (void* dest, const void* source, size_t num) {
  if (num<=0) return dest;
  int tail = num%BLOCK_SIZE;
  int numOfBlocks = num/BLOCK_SIZE + (tail?1:0);
  copy_task* tasks = new copy_task[numOfBlocks];
  for (int i=0; i<numOfBlocks; i++) {
    tasks[i].dst = (char*)dest + i*BLOCK_SIZE;
    tasks[i].src = (const char*)source + i*BLOCK_SIZE;
    tasks[i].num = (i<numOfBlocks-1 || tail==0) ? BLOCK_SIZE : tail;
    thread_create(&copy_block,&tasks[i]);
  }
  wait(NULL);
  return dest;
}
```

3. zadatak, prvi kolokvijum, mart 2018.

Korišćenjem sistemskih poziva `fork`, `execlp` i `wait`, napisati program koji implementira jedan krajnje jednostavan interpreter komandne linije (engl. *command line interpreter*). Ovaj interpreter treba da učitava niz stringova razdvojenih belinama (razmacima ili novim redovima) sa standardnog ulaza, sve dok ne učita string „q“ koji prekida njegov rad. Kada učita svaki string, interpreter treba da pokrene proces nad programom zadatim stazom u tom stringu, sačeka njegov završetak, i pređe na sledeći string. Ukoliko kreiranje procesa nije uspelo, treba da ispiše poruku o grešci i pređe na sledeći. Pretpostavlja se da svaki string ima najviše 32 znaka. Podsetnik na bibliotečne funkcije i sistemske pozive koje se mogu koristiti:

- `scanf`: učitava sa standardnog ulaza; ukoliko se u formatizacionom specifikatoru, iza znaka `%`, napiše ceo broj, on označava maksimalan broj znakova koji će se učitati sa stadardnog ulaza; na primer, `%32s` učitava string, ali ne duži od 32 znaka (i dodaje `'\0'` na kraj);
- `void wait(int pid)`: suspenduje pozivajući roditeljski proces dok se ne završi proces-dete sa zadatim PID; ako je vrednost argumenta 0, pozivajući proces se suspenduje dok se ne završe sve procesi-deca;
- `int strcmp(char*,char*)`: poredi dva data stringa i vraća 0 ako su jednaki.

Rešenje

```
#include <stdio.h>

char command[33];

void main () {

  while (true) {

    scanf("%32s",&command);
  if (strcmp(command,"q")==0) break;
```

```
    int pid = fork();
    if (pid<0)
      printf("Error executing %s.\n",command);
    else
    if (pid>0)
      wait(pid);
    else
      execlp(command);
  }
}
```

3. zadatak, prvi kolokvijum, april 2018.

U nastavku je data donekle izmenjena i pojednostavljena specifikacija sistemskog poziva iz standardnog *POSIX thread* API (*Pthreads*):

`int pthread_create(pthread_t *thread, void *(*routine)(void *), void *arg)`: kreira novu nit nad funkcijom na koju ukazuje routine, dostavljajući joj argument `arg`; identifikator novokreirane niti smešta u lokaciju na koju ukazuje thread; vraća negativnu vrednost u slučaju greške, a 0 u slučaju uspeha.

Dat je potpis funkcije fun, pri čemu su `A`, `B`, `C` i `D` neki tipovi:

```
extern D fun (A a, B b, C c);
```

Potrebno je realizovati potprogram `fun_async` čiji će poziv izvršiti asinhroni poziv funkcije `fun`, tj. inicirati izvršavanje funkcije `fun` u posebnoj niti i potom odmah vratiti kontrolu pozivaocu. Po završetku funkcije `fun`, ta nit treba da pozove povratnu funkciju na koju ukazuje argument `cb` (engl. *callback*) i da joj dostavi vraćenu vrednost iz funkcije `fun`. Ignorisati greške.

```
typedef void (*CallbackD)(D);
void fun_async (A a, B b, C c, CallbackD cb);
```

Rešenje

```
extern D fun (A a, B b, C c);
typedef void (*CallbackD)(D);

struct fun_params { A a; B b; C c; CallbackD cb; };

void fun_async (A a, B b, C c, CallbackD cb) {
  fun_params* params = new fun_params;
  params->a = a; params->b = b; params->c = c;
  params->cb = cb;
  pthread_t pid;
  pthread_create(&pid,&fun_wrapper,params);
}

void fun_wrapper (void* params) {
  fun_params* p = (fun_params*)params;
  D d = fun(p->a,p->b,p->c);
  CallbackD callback = p->cb;
  delete p;
  callback(d);
}
```

3. zadatak, prvi kolokvijum, jun 2018.

U nastavku je data donekle izmenjena i pojednostavljena specifikacija dva sistemska poziva iz standardnog *POSIX thread* API (*Pthreads*). Obe ove funkcije vraćaju negativnu vrednost u slučaju greške, a 0 u slučaju uspeha.

- `int pthread_create(pthread_t *thread, void *(*start_routine)(void *), void *arg)`: kreira novu nit nad funkcijom na koju ukazuje `start_routine`, dostavljajući joj argument `arg`; identifikator novokreirane niti smešta u lokaciju na koju ukazuje `thread`;
- `int pthread_join(pthread_t thread, void **retval)`: čeka da se nit identifikovana sa `thread` završi (ukoliko se već završila, odmah vraća kontrolu); ako `retval` nije `NULL`, kopira povratnu vrednost funkcije `start_routine` koju je ta nit izvršavala u lokaciju na koju ukazuje `retval`.

Data je globalna kvadratna matrica `mat` celih brojeva (`int`), dimenzija $N \times N$. Potrebno je napisati potprogram `par_sum` koji treba da izračuna ukupan zbir svih elemenata matrice `mat`, ali na sledeći način: najpre treba da izračuna zbir svake, i-te vrste ove matrice, uporedo, u N uporednih niti (zbir jedne vrste izračunava jedna nit), i da taj zbir smesti u i-ti element jednog pomoćnog niza; potom treba da sabere ovako izračunate zbirove vrsta i vrati rezultat. Ignorisati eventualne greške i prekoračenje.

```
const int N = ...;
int mat[N][N];
int par_sum ();
```

Rešenje

```
const int N = ...;
int mat[N][N];
int sums[N];
pthread_t pid[N];
typedef int Row[N];

void sum (Row* row) {
  int s = 0;
  for (int i=0; i<N; i++)
    s += (*row)[i];
  sums[row-&mat[0]] = s;
}

// Wrapper, for type-casting only:
void* _sum (void* row) {
  sum((Row*)row);
  return NULL;
}

int par_sum () {
  for (int i=0; i<N; i++)
    pthread_create(&pid[i],&_sum,&a[i]);
  int s = 0;
  for (int i=0; i<N; i++) {
    pthread_join(pid[i],NULL);
    s += sums[i];
  }
  return s;
}
```

3. zadatak, prvi kolokvijum, mart 2017.

U nekom operativnom sistemu postoje sledeći sistemski pozivi:

- `int thread_create(void(*)(void*),void*)`: kreira nit nad funkcijom na koju ukazuje prvi argument; ta funkcija prima jedan argument tipa `void*` i ne vraća rezultat; novokreirana nit poziva tu funkciju sa stvarnim argumentom jednakim drugom argumentu ovog sistemskog poziva; sistemski poziv vraća PID kreirane niti;
- `void wait(int pid)`: suspenduje pozivajuću roditeljsku nit dok se ne završi nit-dete sa zadatim PID; ako je vrednost argumenta 0, pozivajuća nit se suspenduje dok se ne završe sve niti-deca;

- `char getc(FILE* stream)`: iz fajla na čiji deskriptor ukazuje prvi argument učitava i vraća jedan znak; svaki poziv učitava naredni znak iz fajla; ukoliko je učitavanje stiglo do kraja fajla, funkcija vraća vrednost `EOF`.

U nekom programu već je otvoreno N ulaznih fajlova na čije deskriptore ukazuju elementi niza `streams`:

```
const int N = ..., M = ...;
FILE* streams [N];
char text [N][M];
```

Na jeziku C napisati deo programa koji u N uporednih niti-dece učitava najviše po M znakova u svaki red tabele `text`; svaka nit učitava po jedan red tabele iz različitog fajla, u i-ti red iz i- tog fajla. Kada se učitavanje u potpunosti završi, program treba da ispiše sve redove tabele `text` na standardni izlaz.

Rešenje

```
#include <stdio.h>
const int N = ..., M = ...;
FILE* streams[N];
char text[N][M];

void read_line (void* ptr) {
  int i = (FILE**)ptr-streams;
  FILE* stream = streams[i];

  char c = getc(stream);
  int j = 0;
  while ((c!=EOF) && (j<M)) {
    text[i][j] = c;
    j++;
    c = getc(stream);
  }

  if (j<M)
    text[i][j] = '\0';
  else
    text[i][M-1] = '\0';
}

void read_text () {
  int i;
  for (i=0; i<N; i++) {
    text[i][0] = '\0';
    thread_create(read_line,&streams[i]);
  }

  wait(0);

  for (i=0; i<N; i++)
    printf("%s\n",text[i]);
}
```

3. zadatak, prvi kolokvijum, mart 2016.

U nastavku je data donekle izmenjena i pojednostavljena specifikacija dva sistemska poziva iz standardnog *POSIX thread* API (Pthreads). Obe ove funkcije vraćaju negativnu vrednost u slučaju greške, a 0 u slučaju uspeha.

- `int pthread_create(pthread_t *thread, void *(*start_routine)(void *) ,void *arg)`: kreira novu nit nad funkcijom na koju ukazuje `start_routine`, dostavljajući joj argument `arg`; identifikator novokreirane niti smešta u lokaciju na koju ukazuje `thread`;

- `int pthread_join(pthread_t thread, void **retval)`: čeka da se nit identifikovana sa `thread` završi (ukoliko se već završila, odmah vraća kontrolu); ako `retval` nije `NULL`, kopira povratnu vrednost funkcije `start_routine` koju je ta nit izvršavala u lokaciju na koju ukazuje `retval`.

Data je struktura `Node` koja predstavlja jedan čvor binarnog stabla. Korišćenjem datih sistemskih poziva implementirati funkciju `createSubtree` koja rekurzivno kreira binarno stablo dubine `n` i vraća pokazivač na njegov koreni čvor, tako što u istoj niti kreira levo podstablo, a u novokreiranoj niti uporedo kreira desno podstablo. Ignorisati eventualne greške.

```
struct Node {
  Node () : leftChild(NULL), rightChild(NULL) {}
  Node *leftChild, *rightChild;
  ...
};

Node* createSubtree (int n);
```

Rešenje

```
// Wrapper, for type-casting only:
inline void* _createSubtree (void* n) {
  return createSubtree(*(int*)n);
}

Node* createSubtree (int n) {
  if (n<=0) return 0;
  Node* node = new Node();
  if (n<=1) return node;

  int n1 = n-1;
  // Create a new thread to create the right subtree:
  pthread_t pid;
  pthread_create(&pid,&_createSubtree,&n1);
  // Create the left subtree in this thread:
  node->leftChild = createSubtree(n1);
  // Join with the right-subtree child thread:
  pthread_join(pid,&node->rightChild);
  return node;
}
```

3. zadatak, prvi kolokvijum, maj 2016.

U sistemu UNIX i sličnim sistemima sistemski poziv

```
int execvp(const char * filename, const char * const args[]);
```

radi isto što i poziv `execlp`, s tim što drugi argument predstavlja niz pokazivača na nizove znakova (*null-terminated strings*) koji predstavljaju listu argumenata poziva programa koji treba izvršiti. Po konvenciji, prvi argument pokazuje na naziv fajla samog programa koji se izvršava. Ovaj niz pokazivača mora da se završi elementom sa vrednošću NULL. Ovaj poziv vraća 0 u slučaju uspeha, a vrednost manju od 0 u slučaju greške.

Korišćenjem ovog sistemskog poziva, kao i sistemskog poziva `fork()`, realizovati sledeću funkciju:

```
int multiexec (int number, const char* filename, const char* const args[]);
```

Ova funkcija treba da kreira zadati broj (`number`) procesa-dece, i svaki od tih procesa-dece treba da izvršava isti program zadat u fajlu sa imenom `filename`, sa samo po jednim argumentom. Argumenti tih procesa-dece zadati su redom u prvih `number` elemenata niza args. Ova funkcija treba da vrati broj uspešno kreiranih procesa-dece (bez obzira na to da li su ti procesi uspešno izvršili `execvp`) i da odmah vrati kontrolu pozivaocu, ne čekajući da se ti procesi završe.

Rešenje

```
int multiexec (int number, const char* filename, const char* const args[]){
  int ret = 0;
  // Prepare the arguments for the children:
  const char* childArgs[3];
  childArgs[0] = filename;
  childArgs[2] = NULL;
  for (int i=0; i<number; i++) {
    childArgs[1] = args[i];
    // Create a child:
    int status = fork();
    if (status<0) continue; // fork failed
    if (status==0) // Child's context
      if (execvp(filename,childArgs)<0) exit();
    else // Parent's context
      ret++;
  }
  return ret;
}
```

3. zadatak, prvi kolokvijum, septembar 2015.

Neki operativni sistem podržava sledeće sistemske pozive za upravljanje nitima:

- `fork()`: kreira novu nit-dete kao klon roditelja, poput Unix sistemskog poziva *fork* (samo što kreira nit, a ne proces); u kontekstu roditelja vraća ID kreirane niti-deteta, a u kontekstu deteta vraća 0;
- `exit()`: gasi pozivajuću nit;
- `wait(int pid)`: suspenduje pozivajuću nit sve dok se nit dete te niti sa datim ID ne završi (ne ugasi se); argument 0 suspenduje pozivajuću nit sve dok se sva njena deca ne završe.

Korišćenjem ovih sistemskih poziva, realizovati operaciju `cobegin()` koja prima dva argumenta, pokazivače na dve funkcije, i koja pokreće uporedno izvršavanje dve niti nad te dve funkcije, a vraća kontrolu pozivaocu tek kada se obe te niti (odnosno funkcije) završe.

```
void cobegin (void (*f)(), void (*g)());
```

Rešenje

```
void cobegin (void (*f)(), void (*g)()) {
  int id1 = 0, id2 = 0;
  if (id1 = fork())
    if (id2 = fork()) {
      wait(id1);
      wait(id2);
      return;
    } else {
      g();
      exit(0);
    }
  else {
    f();
    exit(0);
  }
}
```

4. zadatak, prvi kolokvijum, mart 2014.

U sistemu UNIX i sličnim sistemima sistemski poziv

```
int execvp(const char * filename, char * const args[]);
```

radi isto što i poziv **execlp**, s tim što drugi argument predstavlja niz pokazivača na nizove znakova (*null-terminated strings*) koji predstavljaju listu argumenata poziva programa koji treba izvršiti. Po konvenciji, prvi argument pokazuje na naziv fajla samog programa koji se izvršava. Ovaj niz pokazivača mora da se završi elementom sa vrednošću NULL.

U nastavku je dat jedan neispravan program. Namera programera je bila da sastavi program koji prima jedan celobrojni argument (po konvenciji, u programu se on vidi kao drugi argument, pored naziva programa), i da, samo ukoliko je vrednost celobrojnog argumenta veća od 0, kreira proces-dete nad istim programom i sa za jedan manjom vrednošću svog celobrojnog argumenta, potom ispiše svoju vrednost argumenta i sačeka da se proces-dete završi, a onda se i sam završi. Proces-dete radi to isto (i tako rekurzivno). Napisati ispravnu implementaciju ovog programa.

```
void main (int argc, char* const argv[]){
  if (argc<2) return; // Exception!

  // Get the argument value:
  int myArg = 0;
  sscanf(argv[1],"%d",&myArg);
  if (myArg<=0) return;

  // Prepare the arguments for the child:
  char childArg[10];  // The value of the second argument
  sprintf(childArg,"%d",myArg-1);
  char* childArgs[3];
  childArgs[0] = argv[0];
  childArgs[1] = childArg;
  childArgs[2] = NULL;

  // Create a child:
  execvp(argv[0],childArgs);
  printf("%s\n",myArg);
  wait(NULL);
}
```

Rešenje

```
void main (int argc, char* const argv[]){
  if (argc<2) return; // Exception!

  // Get the argument value:
  int myArg = 0;
  sscanf(argv[1],"%d",&myArg);
  if (myArg<=0) return;

  // Prepare the arguments for the child:
  char childArg[10];  // The value of the second argument
  sprintf(childArg,"%d",myArg-1);
  char* childArgs[3];
  childArgs[0] = argv[0];
  childArgs[1] = childArg;
  childArgs[2] = NULL;

  // Create a child:
  int status = fork();
  if (status<0) exit(); // Exception!
  if (status==0) // Child's context
    execvp(argv[0],childArgs);
```

```
  else {  // Parent's context
    printf("%d\n",myArg);
    wait(NULL);
  }
}
```

4. zadatak, prvi kolokvijum, mart 2013.

U nekom operativnom sistemu postoje sledeći sistemski pozivi:

- `int thread_create(void (*)(int), int)`: kreira nit nad funkcijom na koju ukazuje prvi argument; ta funkcija prima jedan celobrojni argument i ne vraća rezultat; novokreirana nit poziva tu funkciju sa stvarnim argumentom jednakim drugom argumentu ovog sistemskog poziva; sistemski poziv vraća PID kreirane niti;
- `void wait(int pid)`: suspenduje pozivajuću roditeljsku nit dok se ne završi nit-dete sa zadatim PID.

Data je celobrojna konstanta `N` i funkcija `f` koja prima dva celobrojna argumenta:

```
int f(int,int);
```

Na jeziku C napisati program koji u *N*N* uporednih niti-dece izračunava vrednost funkcije `f(i,j)`, `i=0,...,N-1`, `j=0,...,N-1`, svaku vrednost u po jednoj niti. Kada sve niti-deca završe, program treba da ispiše sve dobijene vrednosti redom na standardni izlaz.

Rešenje

```
const int N = ...;
int f(int,int);
#include <iostream.h>

struct f_params {
  int i, j, r, pid;
};

f_params params[N*N];

void f_wrapper (int index) {
  params[index].r=f(params[index].i,params[index].j);
}

void main () {
  for (int i=0; i<N; i++)
    for (int j=0; j<N; j++) {
      integer ind = i*N+j;
      params[ind].i=i;
      params[ind].j=j;
      params[ind].pid=thread_create(f_wrapper,ind);
    }
  for (int k=0; k<N*N; k++) {
    wait(params[k].pid);
    cout<<params[k].r<<' ';
  }
}
```

4. zadatak, prvi kolokvijum, april 2013.

U nekom operativnom sistemu postoji sledeći sistemski poziv:

```
int thread_create(void (*)(void*), void*);
```

koji kreira nit nad funkcijom na koju ukazuje prvi argument. Ta funkcija prima jedan netipizirani pokazivač i ne vraća rezultat. Novokreirana nit poziva tu funkciju sa stvarnim argumentom jednakim drugom argumentu ovog sistemskog poziva. Sistemski poziv vraća PID kreirane niti.

Korišćenjem ovog sistemskog poziva realizovati klasu `Thread` poput one u školskom jezgru, sa sledećim interfejsom:

```
class Thread {
public:
  void start ();
  virtual ~Thread ();
protected:
  Thread ();
  virtual void run () {}
};
```

Destruktor ove klase treba da sačeka da se nit predstavljena ovim objektom završi. U tu svrhu postoji sistemski poziv `wait(int pid)` koji suspenduje pozivajuću nit dok se nit sa datim PID ne završi.

Rešenje

```
class Thread {
public:
  void start ();
  virtual ~Thread ();
protected:
  Thread () : myPID(0) {}
  virtual void run () {}
private:
  static void thread (void*);
  int myPID;
};

void Thread::thread (void* p) {
  Thread* thr = (Thread*)p;
  if(thr) thr->run();
}

void Thread::start () {
  if (myPID==0) myPID = thread_create(thread,this);
}

Thread::~Thread () {
  if (myPID>0) wait(myPID);
}
```

4. zadatak, prvi kolokvijum, septembar 2013.

U nekom operativnom sistemu postoji sistemski poziv

```
int thread_create(void (*)(void*), void*)
```

koji kreira nit nad funkcijom na koju ukazuje prvi argument. Ta funkcija prima jedan argument tipa `void*` i ne vraća rezultat. Novokreirana nit poziva tu funkciju sa stvarnim argumentom jednakim drugom argumentu ovog sistemskog poziva. Stablo u kome svaki čvor sadrži jedan podatak tipa `Data` i proizvoljno mnogo potomaka realizovano je pomoću sledeće strukture čvora:

```
struct Node {
  Data data;
  Node* children_head;
  Node* next_sibling;
```

```
};

extern void process(Data*);
```

Potomci datog čvora uvezani su u jednostruko ulančanu listu na čiji prvi element ukazuje polje `children_head`. Na sledećeg u listi ukazuje polje `next_sibling`. Procedura `process` vrši obradu datog podatka u jednom čvoru.

Na jeziku C napisati proceduru `visit_node` koja, kada se pozove za koren stabla (ili bilo kog podstabla), obilazi to stablo rekurzivno na sledeći način: najpre obradi podatak u tom čvoru pozivom procedure `process`, a zatim kreira po jednu nit koja obilazi podstablo svakog podčvora počev od drugog u listi (ako postoji), a u istoj niti obilazi podstablo prvog podčvora (ako postoji).

Rešenje

```
void visit_node (void* n) {
  if (n==0) return;
  Node* nd = (Node*)n;
  process(&nd->data);
  if (nd->children_head) {
    for (Node* p=nd->children_head->next_sibling; p; p=p->next_sibling)
      thread_create(&visit_node,p);
    visit_node(nd->children_head);
  }
}
```

4. zadatak, prvi kolokvijum, mart 2012.

Dat je sledeći kod koji koristi školsko jezgro.

```
class TreeNode {
public:
  TreeNode* getLeftChild();
  TreeNode* getRightChild();
  void process();
  ...
};

class TreeVisitor : public Thread {
public:
  TreeVisitor (TreeNode* root) : myRoot(root) {}
protected:
  virtual void run();
  void visit(TreeNode*);
private:
  TreeNode* myRoot;
};

void TreeVisitor::run () {
  visit(myRoot);
}

void TreeVisitor::visit (TreeNode* node) {
  if (node==0) return;
  TreeNode* rn = node->getRightChild();
  if (rn) (new TreeVisitor(rn))->start();
  node->process();
  visit(node->getLeftChild());
}
```

```
void userMain () {
  TreeNode* root = ...;
  Thread* thr = new TreeVisitor(root);
  thr->start();
}
```

Napisati C kod koji radi isto, odnosno obilazi dato stablo rekurzivnim kreiranjem niti koje obilaze podstabla, samo korišćenjem nekog sistema u kome se niti kreiraju i odmah pokreću kao parametrizovani pozivi funkcije sledećim sistemskim pozivom:

```
void create_thread (void(*thread_body)(void*), void* param);
```

Klasa `TreeNode` implementirana je sledećim C kodom:

```
struct TreeNode;
TreeNode* getLeftChild (TreeNode*);
TreeNode* getRightChild (TreeNode*);
void process (TreeNode*);
```

Rešenje

```
void visit (void* nd) {
  TreeNode* node = (TreeNode*)nd;
  if (node==0) return;
  TreeNode* rn = getRightChild(node);
  if (rn) create_thread(&visit,rn);
  process(node);
  visit(getLeftChild(node));
}

void main () {
  TreeNode* root = ...;
  create_thread(&visit,root);
}
```

4. zadatak, prvi kolokvijum, maj 2012.

Klasa `Node`, čija je delimična definicija data dole, predstavlja čvor binarnog stabla. Funkcije `getLeftChild()` i `getRightChild()` vraćaju levo, odnosno desno podstablo datog čvora (tačnije, njegov levi i desni čvor-potomak).

```
class Node {
public:
  Node* getLeftChild();
  Node* getRightChild();
  ...
};
```

Potrebno je prebrojati čvorove u stablu rekurzivnim obilaskom datog binarnog stabla korišćenjem uporednih niti na sledeći način. Ako neka nit trenutno obilazi neki čvor, onda ona treba da napravi novu nit-potomka koja će obići desno podstablo tog čvora, a sama ta nit će nastaviti sa obilaskom levog podstabla tog čvora, i tako rekurzivno.

Realizovati globalnu operaciju `size(Node* root)` koja prebrojava čvorove i vraća broj čvorova u stablu sa datim korenim čvorom na opisani način. Niti treba kreirati sistemskim pozivom `fork()` za koga treba pretpostaviti da ima istu sintaksu i značenje povratne vrednosti kao i istoimeni Unix sistemski poziv, samo što umesto procesa kreira nit u istom adresnom prostoru roditelja: novokreirana nit ima istu poziciju u izvršavanju kao i roditeljska nit, deli isti adresni prostor, samo što poseduje sopstveni kontrolni stek koji inicijalno predstavlja identičnu kopiju roditeljskog steka u trenutku poziva `fork()`. Sistemski poziv `wait(null)` suspenduje pozivajuću nit sve dok se ne završe sve niti-potomci, a poziv `exit()` gasi pozivajuću nit.

Rešenje

```
// Helper recursive function: traverse the tree, compute its size,
// and store its size in sz (sz is external to the thread's stack):

void size_ (Node* node, int& sz) {
  Node* ln = node->getLeftChild();
  int lsz = 0; // left subtree size
  Node* rn = node->getRightChild();
  int rsz = 0; // right subtree size

  if (rn) {
    if (fork()==0) {
      size_(rn,&rsz);
      exit();
    }
  }

  if (ln) size_(ln,&lsz);

  wait(null); // wait for all descendants to complete
  *sz = lsz+rsz+1;  // compute the cumulative size and return to the caller
}

int size (Node* node) {
  if (node==0) return 0;
  int sz = 0;
  size_(node,&sz);
  return sz;
}
```

4. zadatak, prvi kolokvijum, septembar 2012.

U nekom sistemu niti se kreiraju sistemskim pozivom `fork()` koji ima istu sintaksu i značenje povratne vrednosti kao i istoimeni Unix sistemski poziv, samo što umesto procesa kreira nit u istom adresnom prostoru roditelja: novokreirana nit ima istu poziciju u izvršavanju kao i roditeljska nit, deli isti adresni prostor, samo što poseduje sopstveni kontrolni stek koji inicijalno predstavlja identičnu kopiju roditeljskog steka u trenutku poziva `fork()`. Sistemski poziv `exit()` gasi pozivajuću nit.

Korišćenjem ovih sistemskih poziva, realizovati funkciju:

```
int create_thread (void (*fun)(void*), void* p);
```

koja kreira nit nad potprogramom na koji ukazuje prvi argument, tako što taj potprogram poziva u kontekstu novokreirane niti sa datim argumentom `p`, i završava kreiranu nit nakon završetka tog potprograma. Ukoliko sistemski poziv `fork()` vrati grešku, isti kod greške treba vratiti i pozivaocu funkcije `create_thread()`. Ukoliko je kreiranje niti uspešno, pozivaocu funkcije `create_thread()` treba vratiti rezultat poziva `fork()` (ID kreirane niti).

Rešenje

```
int create_thread (void (*fun)(void*), void* p) {
  int ret = fork();
  if (ret<0) return ret;  // Error
  if (ret>0) return ret;  // Parent thread context
  // Child context (ret==0):
  (*fun)(p);
  exit();
}
```

4. zadatak, prvi kolokvijum, maj 2011.

U nekom operativnom sistemu sistemski poziv `clone()` pravi novu nit (*thread*) kao klon roditeljske niti, sa istovetnim kontekstom izvršavanja i u istom adresnom prostoru kao što je i roditeljska nit – isto kao i `fork()`, samo što pravi nit u istom adresnom prostoru, a ne proces. Ovaj poziv vraća 0 u kontekstu niti-deteta, a vrednost >0 koja predstavlja ID kreirane niti u kontekstu niti-roditelja. Vraćena vrednost <0 označava neuspešan poziv. Sistemski poziv `terminate(int)` gasi nit sa datim ID. Korišćenjem ovog sistemskog poziva realizovati klasu `Thread` sa istim interfejsom kao u školskom jezgru (kreiranje niti nad virtuelnom funkcijom `run()` i pokretanje niti pozivom funkcije `start()`).

Rešenje

```
class Thread {
public:
  int start ();
protected:
  Thread () : myID(0) {}
  virtual void run() {}
private:
  int myID;
};

int Thread::start () {
  int id = clone();
  if (id<0) return id;  // Failure
  if (id>0) {  // Successful start in the parent's context
    myID=id;
    return 0;
  }
  // Child context:
  run();
  while (myID==0); // Busy-wait until the parent writes myID
  terminate(myID);
}
```

4. zadatak, prvi kolokvijum, septembar 2011.

Klasa `Node` čija je delimična definicija data dole predstavlja čvor binarnog stabla. Funkcije `getLeftChild()` i `getRightChild()` vraćaju levo, odnosno desno podstablo datog čvora (tačnije, njegov levi i desni čvor-potomak). Funkcija `visit()` obavlja nekakvu obradu čvora.

```
class Node {
public:
  Node* getLeftChild();
  Node* getRightChild();
  void visit();
  ...
};
```

Potrebno je obilaziti dato binarno stablo korišćenjem uporednih niti na sledeći način. Ako neka nit trenutno obilazi neki čvor, onda ona treba da napravi novu nit-potomka koja će obići desno podstablo tog čvora, a sama ta nit će obraditi dati čvor i nastaviti sa obilaskom levog podstabla tog čvora, i tako rekurzivno.

Realizovati globalnu operaciju `visit(Node* root)` koja vrši opisani obilazak stabla sa datim korenim čvorom. Niti treba kreirati sistemskim pozivom `fork()` za koga treba pretpostaviti da ima istu sintaksu i semantiku kao i istoimeni Unix sistemski poziv, samo što umesto procesa kreira nit u istom adresnom prostoru roditelja.

Rešenje

```
int visit (Node* node) {
  if (node==0) return;

  Node* ln = node->getLeftChild();
  Node* rn = node->getRightChild();
  if (rn) {
    if (fork()==0) {
      visit(rn);
      exit();
      // it is assumed that exit() kills only this thread,
      // not the entire process (all threads)
    }
  }

  node->visit();

  if (ln) visit(ln);
}
```

4. zadatak, prvi kolokvijum, april 2009.

U jezgru nekog operativnog sistema definisani su sledeći makroi, strukture i funkcije:

```
#define saveAll ...     // sve registre osim pc i sp čuva na vrhu steka (menja sp)
#define restoreAll ...  // sve registre osim pc i sp restaurira sa steka (menja sp)
#define getSP ...       // vraća vrednost sp registra
#define setSP(expr) ... // izračunava izraz i upisuje ga u registar sp
void copy(int* dst, int* src, int n); // kopira niz od n reči sa src na dst
struct PCB {
  int sp;   // sačuvana vrednost sp registra
  ...       // NIJE DOZVOLJENO PRAVITI PRETPOSTAVKE U VEZI SA OSTALIM POLJIMA PCB STRUKTURE
};
PCB* running; // The running process
```

Korišćenjem ovih makroa, funkcije i struktura, napisati kod funkcija `setjump()` i `longjump()` koje će se koristiti u funkciji `dispatch()` na sledeći način:

```
void dispatch() {
    lock();
    if (setjump() == 0) {
        Scheduler::put(running);
        running = Scheduler::get();
        longjump();
    } else unlock();
}
```

Poznato je da se radi o RISC procesoru koji pored SP i PC registara ima još 16 opštenamenskih registara. Stek raste ka nižim adresama i SP pokazuje na poslednju zauzetu lokaciju na vrhu steka. Povratnu vrednost funkcije uvek vraćaju kroz registar R0, a prevodilac generiše kod tako da sadržaj svih ostalih registara ostane neizmenjen i nakon poziva funkcije (pod uslovom da se ne menja stek). Parametri se funkcijama prenose preko steka, a za čuvanje međurezultata dati kompajler koristi najmanji mogući broj registara. Ako taj broj premašuje broj registara, onda za tu namenu koristi i lokacije na steku. Smatrati da su svi podaci i sve adrese veličine tipa `int` i da je adresibilna jedinica jedna reč veličine tipa `int`. U rešenju nije dozvoljeno korišćenje asemblerskog koda.

Rešenje

```
#define saveAll ... //sve registre osim pc i sp čuva na vrhu steka
#define restoreAll ... //sve registre osim pc i sp restaurira sa steka
#define getSP ... //vraća vrednost sp registra
#define setSP(expr) ... //izračunava izraz i upisuje ga u registar sp
void copy(int* dst, int* src, int n); //kopira niz od n reči sa src
                                      //na dst
struct PCB {
  int* sp; // sačuvana vrednost sp registra
  ...     // NIJE DOZVOLJENO PRAVITI PRETPOSTAVKE U VEZI SA OSTALIM POLJIMA
          //PCB STRUKTURE
};
PCB* running; // The running process

int setjump(){ // R0 is used to access fields of PCB pointed by running,
// since its content will be overwritten
// with the return result at the end(return 0)
// compiler will not save it on the stack
  saveAll;
  running->sp = getSP;
  setSP(running->sp-1);
  copy((int*)running->sp-1, (int*)running->sp+16, 1);
  return 0;
}

int longjump(){
  setSP(running->sp);
  restoreAll;
  return 1;
}
```

5. zadatak, prvi kolokvijum, april 2009.

Dat je sledeći program:

```
#include <stdio.h>
#define N 3

void main() {
    int i, f = 0, s = 0;
    for (i = 0; i < N; i++) {
        f = f || fork();
        s += i;
    }
    if (f) exit(0);
    printf("%d ", s);
}
```

Pod pretpostavkom da su svi sistemski pozivi uspeli, da je funkcija `printf` bezbedna za poziv u konkurentnim procesima (svaki poziv se izvršava izolovano i bez interakcije sa pozivima iz drugih procesa, tj., uporedne pozive ove funkcije iz različitih procesa sistem sekvencijalizuje), i da uvek koristi isti standardni izlaz, prikazati ispis koji se dobije pokretanjem datog programa? Odgovor obrazložiti.

Rešenje

```
#include <stdio.h>
#define N 3
```

```
void main () {
  int i, f = 0, s = 0;
  for (i=0; i<N; i++) {
    f = f || fork();
    s += i;
  }
  if (f) exit(0);
  printf("%d ",s);
}
```

U tri prolaza petlje kreira se 8 procesa: u prvom prolazu od 1 nastaju 2, u drugom od svakog od ovih nastaju po 2, što je ukupno 4 i u trećem prolazu od svakog od ova 4 nastaju po 2, što je ukupno 8. Svaki od njih će računati sumu s u svom adresnom prostoru, koja će imati vrednost 0+1+2 = 3.

Međutim, kako pri pozivu `fork()` samo jedan proces dobije povratnu vrednost 0, jedan od prva dva procesa će u f izračunati 1, a drugi 0. Zbog logičke funkcije „ili“, svi procesi koji nastanu od procesa u kojem je `f = 1` će takođe imati `f = 1` i zbog toga nikada neće stići do ispisa. Onaj koji ima vrednost `f = 0` se na isti način deli na dva procesa, gde f u jednom procesu ima vrednost 1, a u drugom 0. Kao i u prethodnom slučaju, jedini procesi koji imaju šansu da dođu do ispisa su oni koji nastaju od onog koji u f ima vrednost 0. Dalje posmatramo proces koji u f ima vrednost 0. Taj proces se još jednom deli na dva od kojih samo jedan u f ima vrednost 0 i samo taj proces će pri izvršavanju doći do ispisa, što znači da će biti ispisana samo jedna suma, 3.

Rezonovanje bi se moglo nastaviti na isti način dalje, i upotrebom indukcije bi se moglo pokazati da će za proizvoljnu vrednost N biti ispisana samo jedna suma brojeva od 1 do N.

4. zadatak, prvi kolokvijum, april 2008.

Dat je sledeći program:

```
#include <stdio.h>
#define N 3
int pid[N];

void main () {
  int i;
  for (i=0; i<N; i++) pid[i]=0;
  for (i=0; i<N; i++) {
    if (pid[i]==0) pid[i]=fork(); // if1
    if (pid[i]==0)                // if2
      for (j=i+1; j<N; j++) pid[j]=1;
  }
}
```

Pod pretpostavkom da su svi sistemski pozivi uspeli, koliko će ukupno procesa biti kreirano (direktno ili indirektno) od strane jednog procesa početno kreiranog nad ovim programom, uključujući i taj jedan početni proces? Odgovor obrazložiti.

Rešenje

Odgovor: 4

Početni proces kreira tri procesa-potomka, za svaku iteraciju (spoljne) `for` petlje `(i=0, 1, 2)` . U svakoj od tih iteracija, naredba `if` označena sa `if1` izvršava svoju *then* granu, jer je `pid[i]` uvek 0 (tako je inicijalizovan). U tom procesu *then* grana naredbe if2 se ne izvršava, jer je `pid[i]` tada različit od 0 (pošto je `fork()` vratio ID kreiranog procesa). U svakom od kreiranih procesa-potomaka, `fork()` vraća 0, pa je u naredbi `if2 pid[i]==0`, odnosno izvršava se ugnežđena `for` petlja (po `j`) koja u sve elemente niza `pid` iza `i`-tog upisuje 1. Zbog toga se u ovim procesima-potomcima, u svim narednim iteracijama glavne `for` petlje (po `i`), neće izvršiti `fork()` u naredbi `if1`, pa oni više neće kreirati svoje potomke.

5. zadatak, prvi kolokvijum, april 2007.

Dat je sledeći program:

```
#include <stdio.h>
#define N 3
int pid[N];

void main () {
  int i;
  for (i=0; i<N; i++) pid[i]=0;
  for (i=0; i<N; i++) pid[i]=fork();
  for (i=0; i<N; i++) printf("%d ",pid[i]);
}
```

Pod pretpostavkom da su svi sistemski pozivi uspeli, koliko ukupno nula ispisuju svi procesi kreirani od strane jednog procesa nad ovim programom? Odgovor obrazložiti.

Rešenje

```
#include <stdio.h>
#define N 3
int pid[N];

void main () {
  int i;
  for (i=0; i<N; i++) pid[i]=0;
  for (i=0; i<N; i++) pid[i]=fork();
  for (i=0; i<N; i++) printf("%d ",pid[i]);
}
```

U tri prolaza petlje u sredini kreira se 8 procesa: u prvom prolazu od 1 nastaju 2, u drugom od svakog od ovih nastaju po 2, što je ukupno 4 i u trećem prolazu od svakog od ova 4 nastaju po 2, što je ukupno 8. Svaki od njih će imati sopstveni niz pid i ispisaće 3 broja. To je ukupno 24 ispisana broja.

Posle svakog poziva `fork()` ostaju po dva identična procesa (roditelj i novokreirani potomak) koji se razlikuju samo po rezultatu `fork()` funkcije. U prvom prolazu kroz petlju ostaju dva procesa, jedan sa `pid[0]=0` i drugi sa `pid[0]<>0`. Pošto su u kontrolnim strukturama ova dva procesa identicna, to znači da će od svakog od njih nastati isti broj novih procesa, odnosno da će polovina svih procesa imati `pid[0]=0`, a druga `<>0`. Dalje, problem možemo raščlaniti na dva koji imaju za 1 manju dimenziju problema (niz sa elementima `pid[1]` i `pid[2]`). To znači da će, iz istog razloga kao i u prethodnom slučaju, polovina procesa imati `pid[1]=0`, a druga `<>0`. Ovakvim razmatranjem ili direktnim brojanjem dolazi se da je polovina ispisanih brojeva jednaka nuli, odnosno $24/2 = 12$ nula.

6. zadatak, prvi kolokvijum, april 2006.

U nekom operativnom sistemu postoji sistemska usluga kreiranja niti koja se iz jezika C/C++ poziva pomoću bibliotečne funkcije `create_thread()`:

```
typedef unsigned long PID;  // Process ID
PID create_thread (void (*body)(void*), void* arg);
```

Ovaj sistemski poziv kreira nit nad C funkcijom na koju ukazuje prvi argument `body`, pri čemu se pozivu te funkcije kao argument dostavlja pokazivač `arg`.

Za ovaj operativni sistem realizuje se biblioteka za konkurentno izvršavanje C++ programa koja koncept niti podržava klasom `Thread`. Ova klasa realizuje koncept niti kao u školskom jezgru i ima sledeći interfejs:

```
class Thread {
public:
  void start ();  // Starts the thread
protected:
```

```
  Thread ();
  virtual void run() {}  // The body of the thread
private:
  PID pid;  // PID of system thread
};
```

tako da se korisničke niti kreiraju kao u sledećem primeru:

```
class Robot : public Thread {
...
  virtual void run ();
};

Robot* r = new Robot;
r->start();
```

Preslikavanje niti iz korisničke aplikacije na niti operativnog sistema je jedan-na-jedan. Korišćenjem navedenog sistemskog poziva, realizovati operaciju `Thread::start()` (uz odgovarajuće dodatne delove koda, ako je potrebno).

Rešenje

U `protected` sekciju klase `Thread` potrebno je dodati sledeći metod:

```
class Thread{
  ...
protected:
  static void starter(void*);
  ...
}

void Thread::starter(void* toStart){
  Thread* t = (Thread*)toStart;
  if (t) t->run();
}
```

U tom slučaju implementacija metoda start izgleda:

```
void Thread::start(){
  pid = create_thread(&starter, this);
}
```

Interfejs niti

2. zadatak, kolokvijum, avgust 2021.

Školsko jezgro proširuje se nestatičkom funkcijom `Thread::join()` koju sme da pozove samo roditeljska nit date niti da bi sačekala da se data nit-dete završi; ukoliko ovu operaciju pozove neka druga nit koja nije roditelj date niti, ova funkcija vraća grešku (`-1`). Kada jezgro pravi novu nit, poziva funkciju `Thread::wrapper` koja obavlja sve potrebne radnje pre poziva funkcije `Thread::run` (u kontekstu napravljene niti) i nakon povratka iz nje:

```
void Thread::wrapper (Thread* toRun) {
  ...
  unlock();
  toRun->run();
  lock();
  ...
}
```

Precizno navesti sve izmene i dopune koje je potrebno napraviti u klasi `Thread` i implementirati operaciju `Thread::join`. Uzeti u obzir to da nit-roditelj može pozvati `join` više puta, pri čemu se samo pri prvom pozivu eventualno može zaustaviti dok nit-dete ne završi, svi pozivi nakon toga su neblokirajući. Problem gašenja niti i brisanja objekta klase `Thread`, kao i sinhronizacije potrebne za to ne treba rešavati u ovom zadatku.

Rešenje

U klasi `Thread` treba uraditi sledeće izmene:

- Dodati privatan član: `Thread* parent`; ukazuje na roditeljsku nit; inicijalizuje se na `Thread::runningThread` u konstruktoru klase Thread;
- Dodati privatne članove tipa `bool`, inicijalizovane na false u konstruktoru klase `Thread`: complete koji ukazuje na to da je ova nit završila i isParentWaiting koji ukazuje na to da roditeljska nit čeka na završetak ove niti;
- U funkciji `Thread::wrapper` uraditi sledeće dopune:

```
void Thread::wrapper (Thread* toRun) {
  ...
  unlock();
  toRun->run();
  lock();
  toRun->complete = true;
  if (toRun->isParentWaiting) {
    toRun->isParentWaiting = false;
    Scheduler::put(toRun->parent);
  }
  ...
}
```

- Dodati javnu fukciju članicu:

```
int Thread::join () {
  if (!this || this->parent!=Thread::runningThread) return -1;
  if (this->complete) return 0;
  lock();
  this->isParentWaiting = true;
  if (setjmp(Thread::runningThread->context)==0) {
    Thread::runningThread = Scheduler::get();
    longjmp(Thread::runningThread->context,1);
  }
  unlock();
```

```
  return 0;
}
```

3. zadatak, prvi kolokvijum, april 2017.

U standardnom jeziku C++ postoje, između ostalog, sledeći elementi podrške za niti:

- `std::thread`: klasa kojom se predstavljaju niti; nova nit se pokreće odmah po kreiranju nekog objekta ove klase;
- `std::thread::thread`: konstruktor klase `std::thread` koji kreira nit nad funkcijom na koju ukazuje prvi argument; ta funkcija prima jedan argument i ne vraća rezultat; novokreirana nit poziva datu funkciju sa stvarnim argumentom jednakim drugom argumentu konstruktora;
- `std::thread::join`: suspenduje pozivajuću nit dok se ne završi nit čija se ova funkcija poziva;
- `thread_local`: specifikator životnog veka i načina alokacije koji se navodi u deklaraciji objekta; objekti ove kategorije životnog veka alociraju se kad god se kreira nova nit i nestaju sa završetkom te niti; svaka kreirana nit, uključujući i „glavnu“ nit kreiranu nad pozivom funkcije main, ima svoju sopstvenu kopiju (instancu) ovog objekta, različitu od instanci tog objekta u drugim nitima.

Precizno navesti šta sve može da ispiše sledeći C++ program (izlazne operacije ispisa na standardni izlaz su atomične):

```
thread_local int i=0;
void f (int id) {
  i=id;
  ++i;
  std::cout<<i;
}
void main() {
  i=9;
  std::thread t1(f,1);
  std::thread t2(f,2);
  std::thread t3(f,3);
  t1.join();
  t2.join();
  t3.join();
  std::cout<<i<<std::endl;
}
```

Rešenje

2349, ili 2439, ili 3249, ili 3429, ili 4239, ili 4329 i ništa više osim toga.

3. zadatak, prvi kolokvijum, jun 2017.

Data je klasa `Thread` koja implementira niti kao u školskom jezgru, ali samo sa jednom mogućnošću konstrukcije objekata te klase, kako je dato u njenom interfejsu dole (samo objektno orijentisani API za kreiranje niti).

1. Realizovati klasu `ThreadFnCaller` koja omogućava kreiranje niti nad zadatom globalnom funkcijom koja prima jedan argument tipa `void*` i ne vraća rezultat, tako da ta nit izvršava zadatu funkciju sa zadatim argumentom.
2. Napisati fragment koda koji kreira `N` niti nad funkcijom `fn`, pri čemu i-ta nit izvršava tu funkciju sa argumentom koji je dat u i-tom elementu niza `args`.

```
class Thread {
public:
  Thread ();
  void start ();
  virtual void run () {}
};
```

Rešenje

1.

```
class ThreadFnCaller : public Thread {
public:
  ThreadFnCaller (void (*fn)(void*), void* arg) : myFn(fn), myArg(arg) {}
  virtual void run () { myFn(myArg); }
private:
  void (*myFn)(void*);
  void* myArg;
};
```

2.

```
for (int i=0; i<N; i++) (new ThreadFnCaller(fn,args[i]))->start();
```

3. zadatak, prvi kolokvijum, septembar 2016.

Dat je neki veliki jednodimenzioni niz `data` veličine `N`, čiji su elementi tipa `Data` (deklaracije dole). Jedan element niza obrađuje procedura `processData`. Ovaj niz potrebno je obraditi pomoću n uporednih niti (procedura `parallelProcessing`), gde je `n` zadati parametar, tako što svaka od tih uporednih niti iterativno obrađuje približno isti broj elemenata ovog velikog niza. Drugim rečima, niz treba particionisati na `n` disjunktnih podnizova, što približnije jednakih, i te particije obrađivati uporedo. Implementaciju dati u školskom jezgru.

```
const int N = ...;
class Data;
Data data[N];
void processData(Data*);
void parallelProcessing (int n);
```

Rešenje

```
class DataProcessor : public Thread {
public:
  DataProcessor (int offs, int sz) : offset(offs), size(sz) {}
  virtual void run ();
private:
  int offset, size;
};

void DataProcessor::run () {
  int end = this->offset+this->size;
  for (int i=this->offset; i<end; i++)
      processData(&data[i]);
}

void parallelProcessing (int n) {
   int chunk = N/n;
   int offset = 0;
   for (int i=0; i<n; i++) {
     int myChunk = chunk;
     if (i<N%n) myChunk++;
     new DataProcessor(offset,myChunk)->start();
     offset+=myChunk;
   }
}
```

3. zadatak, prvi kolokvijum, mart 2015.

U nekom sistemu sistemski poziv `fork()` kreira nit – klon pozivajuće niti, sa iskopiranim celokupnim stekom, slično istoimenom sistemskom pozivu na sistemu Unix (osim što se ovde radi o nitima, a ne procesima).

Dole je dat program čija je zamisao da izvršava dve uporedne niti, jednu koja učitava znak po znak sa stadardnog ulaza i taj znak prenosi kroz promenljivu c drugoj niti, koja taj primljeni znak ispisuje na standardni izlaz, i tako neograničeno.

1. Precizno objasniti problem koji ovaj program ima i ispraviti taj problem.
2. Prepraviti samo funkciju `pipe()` tako da se umesto jednog para niti koje vrše razmenu znakova. formiraju dva para takvih niti; svaki par niti predstavlja odvojeni „tok“. pa je potrebno definisati dva para promenljivih `c` i `flag` (npr. `c1`, `c2`, `flag1` i `flag2`).

```
#include <iostream>

void writer (char* c, int* flag) {
  while (1) {
    while (*flag==1);
    cin>>(*c);
    *flag = 1;
  }
}

void reader (char* c, int* flag) {
  while (1) {
    while (*flag==0);
    cout<<(*c);
    *flag = 0;
  }
}

void pipe () {
  char c;
  int flag = 0;
  if (fork())
    writer(&c,&flag);
  else
    reader(&c,&flag);
}

void main () {
  pipe();
}
```

Rešenje

1. Problem je što su promenljive `flag` i `c` koje bi trebalo da budu deljene (zajedničke) između niti, jer preko njih treba da razmenjuju podatke i sinhronizuju se, definisane kao automatske (alociraju se na steku), što znači da zapravo neće biti deljene, već će svaka nit imati svoju instancu ovih promenljivih (na svom steku), pošto svaka nit ima svoj zaseban stek, pa razmene zapravo neće ni biti. Rešenje je prosto deklarisati ih kao statičke (`static`).
2.

```
void pipe () {
  static char c1, c2;
  static int flag1 = 0, flag2 = 0;

  if (fork())
```

```
    writer(&c1,&flag1);
  else
    if (fork())
      reader(&c1,&flag1);
    else
      if (fork())
        writer(&c2,&flag2);
      else
        reader(&c2,&flag2);
}
```

3. zadatak, prvi kolokvijum, maj 2015.

U nekom sistemu sistemski poziv `fork()` kreira nit – klon pozivajuće niti, sa iskopiranim celokupnim stekom, slično istoimenom sistemskom pozivu na sistemu Unix (osim što se ovde radi o nitima, a ne procesima). Dole je dat (neispravan) program čija je zamisao da izvršava dva para uporednih niti, pri čemu u svakom paru jedna nit učitava znak po znak sa standardnog ulaza i taj znak prenosi kroz promenljivu `c[0]/c[1]` drugoj niti u istom paru, koja taj primljeni znak ispisuje na standardni izlaz, i tako neograničeno.

1. Precizno objasniti problem koji ovaj program ima (zašto je neispravan).
2. Prepraviti samo funkciju `pipe()` tako da se umesto dva para niti koje vrše razmenu znakova, formira `N` parova takvih niti; svaki par niti i predstavlja odvojeni „tok“ koji treba da razmenjuje podatke preko `c[i]` i `flag[i]`.

```
#include <iostream>

void writer (char* c, int* flag) {
  while (1) {
    while (*flag==1);
    cin>>(*c);
    *flag = 1;
  }
}

void reader (char* c, int* flag) {
  while (1) {
    while (*flag==0);
    cout<<(*c);
    *flag = 0;
  }
}

const int N = ...; // N>2
char c[N];
int flag[N];

void main () {
  for (int i=0; i<N; i++) flag[i]=0;
  pipe();
}

void pipe () {
  if (fork())
    writer(&c[0],&flag[0]);
  else
    reader(&c[0],&flag[0]);
  if (fork())
    writer(&c[1],&flag[1]);
  else
```

```
    reader(&c[1],&flag[1]);
}
```

Rešenje

1. Problem je u neispravnim kontrolnim strukturama u telu funkcije pipe. Početna (roditeljska) nit izvršava then granu prve `if` naredbe, u kojoj se poziva funkcija `writer` i koja se onda neograničeno (beskonačno) izvršava, pa ta prva nit nikada ne izlazi iz ove funkcije. Nit-dete, kreirana u prvom `fork` pozivu, izvršava else granu iste prve `if` naredbe, u kojoj se poziva funkcija reader koja se takođe neograničeno izvršava. Prema tome, ni jedna od ove dve niti neće nikada doći do druge `if` naredbe, pa se drugi par niti nikada neće kreirati.
2.

```
void pipe () {
  for (int i=0; i<N; i++)
    if (fork())
      writer(&c[i],&flag[i]);
    else
      if (fork())
        reader(&c[i],&flag[i]);
}
```

4. zadatak, prvi kolokvijum, septembar 2014.

Korišćenjem niti u školskom jezgru realizovati klasu `Search` koja ima dole dati interfejs. Objekat ove klase obavlja pretragu datog niza celih brojeva a u opsegu indeksa počev od `i` zaključno sa `j`, tražeći u njemu vrednost `x`, ali tako što pretragu obavlja konkurentno i binarno-rekurzivno: jedna već kreirana nit nastavlja pretragu jedne polovine datog niza, a kreira novu nit koja vrši istu takvu pretragu druge polovine niza, i tako dalje rekurzivno. Ako se na nekom elementu n pronađe vrednost x, treba ispisati „Found at n!“, a ako se tu ne nađe ta vrednost, treba ispisati „Not found at n!“.

```
class Search : public Thread {
public:
  Search (int a[], int i, int j, int x);
};
```

Primer korišćenja ove klase je sledeći:

```
int array[N];
Thread *t = new Search(array,0,N-1,5);
t->start();
```

Rešenje

```
class Search : public Thread {
public:
  Search (int a[], int i, int j, int x) : array(a), ii(i), jj(j), xx(x) {}
protected:
  virtual void run () { find(ii,jj); }
  void find (int i, int j);
private:
  int *array, ii, jj, xx;
};

void Search::find (int i, int j) {
  if (array==0 || i<0 || j<0 || j<i) return;
  if (i==j) {
    if(array[i]==x)
      printf("Found at %d!\n",i);
    else
```

```
      printf("Not found at %d!\n",i);
    return;
  }
  int k = (i+j)/2;
  Thread* t = new Search(array,k+1,j,x);
  t->start();
  find(i,k);
}
```

3. zadatak, prvi kolokvijum, septembar 2013.

1. Na asembleru nekog RISC procesora sa *load/store* arhitekturom, poput onog opisanog na predavanjima, napisati prevod sledeće rekurzivne C funkcije koja izračunava najveći zajednički delilac (engl. *greatest common divisor*, GCD) dva data nenegativna cela broja x i y Euklidovim algoritmom, pri čemu je uvek $x \geq y \geq 0$ (pretpostaviti da su argumenti uvek ovakvi i ispravni). Funkcija `remainder(x,y)` je implementirana i vraća ostatak pri deljenju x sa y.

```
unsigned int gcd (unsigned int x, unsigned int y) {
  if (y==0) return x;
  else return gcd(y,remainder(x,y));
}
```

2. Da li je ova funkcija bezbedna za uporedne pozive iz konkurentnih niti? Obrazložiti.

Rešenje

1.

```
gcd:      LOAD R0,#x[SP] ; R0:=x
          LOAD R1,#y[SP] ; R1:=y
          CMP R1,#0 ; if (y==0)
          JMPNE gcd_else
gcd_then: POP PC ; return x (already in R0)
gcd_else: PUSH R1 ; push y for call of remainder
          PUSH R0 ; push x for call of remainder
          CALL remainder
          POP R1 ; remove x from the stack
          POP R1 ; remove y from the stack
          PUSH R0 ; push remainder(...) for call of gcd
          PUSH R1 ; push y for call of gcd
          CALL gcd
          POP R1 ; stack cleanup
          POP R1
          POP PC ; return (the result is already in R0)
```

2. Jeste bezbedna (ovakvi potprogrami se ponekad nazivaju *reentrant*), pošto uopšte ne pristupa statičkim (globalnim) podacima, već sve podatke ima ili na steku, ili u registrima, što je deo konteksta niti.

2. zadatak, drugi kolokvijum, maj 2012.

Data je bibliotečna funkcija namenjena podršci optimističkom pristupu međusobnom isključenju bez eksplicitnog zaključavanja:

```
int cmpxchg(void** ptr, void* oldValue, void* newValue);
```

Ova funkcija kao prvi argument (`ptr`) prima adresu lokacije u kojoj se nalazi neki pokazivač bilo kog tipa (`void*`). Ona atomično poredi vrednost pokazivača koji je dostavljen kao drugi argument (`oldValue`) sa vrednošću na lokaciji na koju ukazuje `ptr`, i ako su te vrednosti iste, u lokaciju na koju ukazuje `ptr` upisuje vrednost datu trećim argumentom (`newValue`) i vraća 1; u suprotnom, ako su ove vrednosti različite, ne radi ništa, već samo vraća 0. Atomičnost je obezbeđena implementacijom pomoću odgovarajuće mašinske instrukcije.

Struktura `Record` predstavlja zapis (jedan element) jednostruko ulančane liste. U toj strukturi polje `next` ukazuje na sledeći element u listi.

Korišćenjem date operacije `cmpxchg()` implementirati funkciju:

```
void insert (Record** head, Record* e);
```

Ova funkcija prima argument koji predstavlja adresu pokazivača na prvi element liste (adresu lokacije u kojoj je glava liste), a kao drugi argument dobija pokazivač na novi element u listi koga treba da umetne na početak date liste. Lista je deljena između više procesa, pa ova funkcija treba da bude sigurna za uporedne pozive iz više procesa, s tim da međusobno isključenje treba obezbediti optimističkom strategijom bez eksplicitnog zaključavanja. Zapis na koga ukazuje drugi element (`e`) je privatan samo za pozivajući proces (drugi procesi mu ne pristupaju pre umetanja u listu).

Rešenje

```
void insert (Record** head, Record* e) {
  do {
    Record* oldHead = *head;
    Record* newHead = oldHead;
    e->next = newHead;
    newHead = e;
  } while (cmpxchg(head,oldHead,newHead)==0);
}
```

4. zadatak, prvi kolokvijum, april 2011.

Klasa `Node` čija je delimična definicija data dole predstavlja čvor binarnog stabla. Funkcije `getLeftChild()` i `getRightChild()` vraćaju levo, odnosno desno podstablo datog čvora (tačnije, njegov levi i desni čvor-potomak). Funkcija `visit()` obavlja nekakvu obradu čvora.

```
class Node {
public:
  Node* getLeftChild();
  Node* getRightChild();
  void visit();
  ...
};
```

Potrebno je obilaziti dato binarno stablo korišćenjem uporednih niti na sledeći način. Ako neka nit trenutno obilazi neki čvor, onda ona treba da napravi novu nit-potomka koja će obići desno podstablo tog čvora, a sama ta nit će obraditi dati čvor i nastaviti sa obilaskom levog podstabla tog čvora, i tako rekurzivno.

Realizovati klasu `TreeVisitor` izvedenu iz klase `Thread` iz školskog jezgra koja realizuje opisani obilazak binarnog stabla. Ova klasa treba da se koristi na sledeći način:

```
Node* tree = ...;  // The root node of a binary tree
TreeVisitor* tv = new TreeVisitor(tree);
tv->start();
```

Rešenje

```
class TreeVisitor : public Thread {
public:
  TreeVisitor (Node* root) : myRoot(root) {}
protected:
  virtual void run();
  void visit(Node*);
private:
  Node* myRoot;
};
```

```
void TreeVisitor::run () {
  visit(myRoot);
}

void TreeVisitor::visit (Node* node) {
  if (node==0) return;

  Node* rn = node->getRightChild();
  if (rn) (new TreeVisitor(rn))->start();

  node->visit();

  visit(node->getLeftChild());
}
```

5. zadatak, prvi kolokvijum, april 2010.

Na narednoj strani prikazan je najvažniji deo programa koji pronalazi put kroz lavirint (engl. *maze*) poput onog datog na slici dole. Objašnjenje nekih pomoćnih operacija je sledeće:

- `Direction operator-(Position pos2, Position pos1)`: Vraća smer u kome se polje označeno pozicijom pos2 u kvadratnoj koordinatnoj mreži može dostići iz jednog od četiri susedna polja na poziciji `pos1`. Na primer, ako je pos2 desni sused od `pos1`, vratiće E; ako mu je donji sused, vratiće S; ako mu je levi, vratiće W; ako je gornji, vratiće N.
- `int Maze::isExit(Position)`: Vraća 1 ako je polje na datoj poziciji u lavirintu izlaz, odnosno 0 ako nije.
- `Position* Maze::getFirstOption(Position p, Direction d), getSecondOption, getThirdOption`: Za dato polje `p` u lavirintu u koje se stiglo napredovanjem u datom smeru `d`, vraćaju slobodna susedna polja u koja se može ići bez povratka u polje iz koga se stiglo. Ako nema ni jednog mogućeg smera (slobodnog susednog polja osim onoga iz koga se stiglo u `p`), sve tri operacije vratiće 0; ako je moguće nastaviti samo u jednom smeru, samo će `getFirstOption` vratiti poziciju susednog slobodnog polja, ostale će vratiti 0; ako je moguće nastaviti u dva smera, prve dve operacije će vratiti pozicije susednih slobodnih polja, treća će vratiti 0.

Tragač za izlazom pokrenut je na sledeći način za lavirint prikazan na donjoj slici:

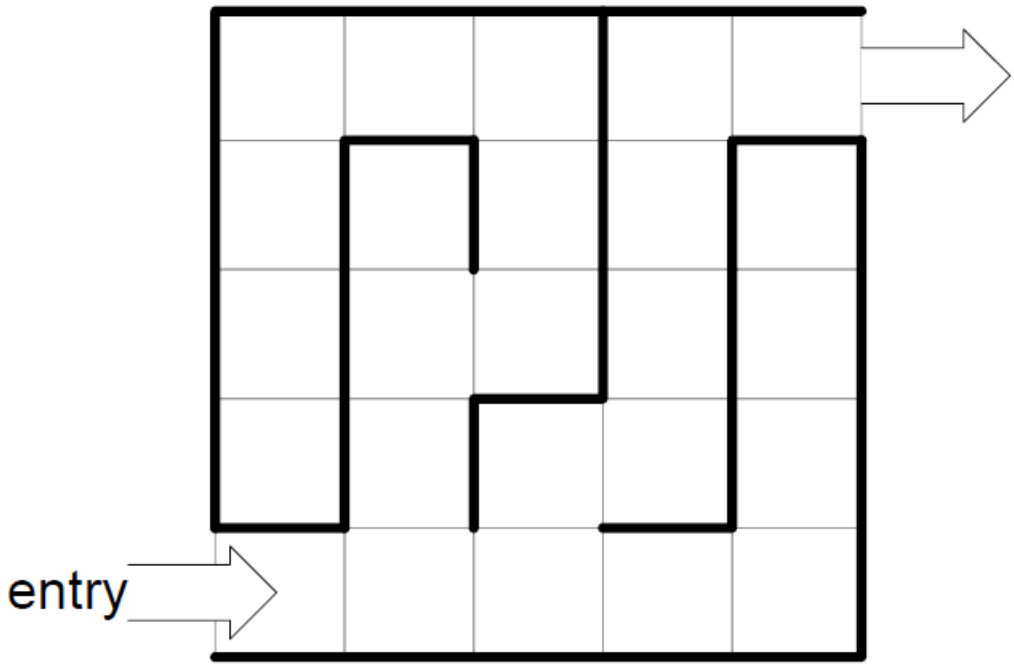
```
MazeExitSeeker* seeker = new MazeExitSeeker(theMaze,entry,E);
seeker->start();
```

1. Precizno objasniti kako ovaj program pretražuje lavirint.
2. Koliko ukupno niti će biti pokrenuto za ovaj lavirint, računajući i početnu nit (`seeker`) prikazanu gore?
3. Uz pretpostavku da svaka nit ima svoj logički izlazni uređaj `cout` (nema preplitanja ispisa iz različitih niti), koliko puta će ukupno biti ispisana rečenica „I have reached a dead end and I am giving up.”? entry

```
enum Direction {N, E, S, W};

class Position {
public:
  friend Direction operator- (Position pos2, Position pos1);
  ...
};


class Maze {
public:
  int isExit(Position);
  Position* getFirstOption(Position,Direction);
  Position* getSecondOption(Position,Direction);
  Position* getThirdOption(Position,Direction);
```



Slika 3: Lavirint

```
  ...
};

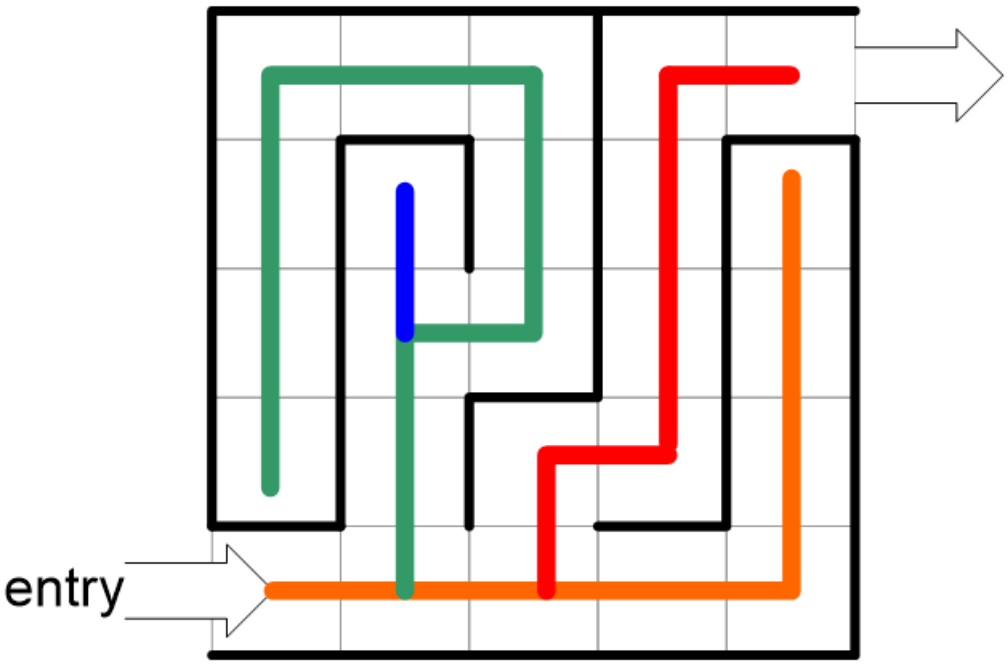

class MazeExitSeeker : public Thread {
public:
  MazeExitSeeker (Maze* m, Position start, Direction dir) :
      myMaze(m), myPos(start), myDir(dir) {}
protected:
  virtual void run();
private:
  Maze* myMaze;
  Position myPos;
  Direction myDir;
};


void MazeExitSeeker::run () {
  cout<<"I'm starting my search from "<<myPos<<" towards "<<myDir<<.\n"
  while (1) {
    if (myMaze->isExit(myPos)) {
      cout<<"I have found the exit!\n"
      return;
    }
    Position* nextPos1 = myMaze->getFirstOption(myPos,myDir);
    Position* nextPos2 = myMaze->getSecondOption(myPos,myDir);
    Position* nextPos3 = myMaze->getThirdOption(myPos,myDir);
    if (nextPos1==0) {
      cout<<"I have reached a dead end and I am giving up.\n"
      return;
```

```
    }
    if (nextPos2!=0)
      (new MazeExitSeeker(myMaze,*nextPos2,*nextPos2-myPos))->start();
    if (nextPos3!=0)
      (new MazeExitSeeker(myMaze,*nextPos3,*nextPos3-myPos))->start();
    myDir=*nextPos1-myPos;
    myPos=*nextPos1;
  }
}
```

Rešenje

1. Svaka nit napreduje kroz hodnike lavirinta i u svakom koraku, na datom polju u kvadratnom koordinatnom sistemu, radi sledeće. Najpre pogleda da li je pronašla izlaz i ako jeste, ispisuje to i završava se. Zatim od lavirinta dobije najviše tri susedna slobodna polja u koja može da pređe (osim onoga iz koga je stigla). Ako ni jedno od takvih ne postoji, ta nit je udarila u slepi hodnik, pa ispisuje rečenicu „I have reached a dead end and I am giving up“ i gasi se. Inače, ta nit nastavlja da ide prvom dobijenom opcijom, tako da joj je sledeće polje to susedno polje dobijeno kao prva opcija, u tom smeru. Ako postoje druga i treća opcija, onda ova nit kreira nove niti, po jednu za svaku od tih raspoloživih opcija (drugu i treću), tako da te niti nastavljaju od tih susednih polja, u odgovarajućim smerovima.
2. Četiri niti. Na slici je prikazan jedan mogući raspored puteva kojim tragaju te niti, uz pretpostavku da roditeljska nit nastavlja desnim od mogućih puteva, a potomci ostalim.
3. Tri puta.



Slika 4: Rešenje

Promena konteksta

1. zadatak, kolokvijum, septembar 2020.

Dostupni su sledeći delovi školskog jezgra, kao i funkcije iz standardne biblioteke:

- `Thread::running`: statički član – pokazivač na objekat klase `Thread` koji predstavlja tekuću nit;
- `Thread::context`: nestatički član tipa `jmp_buf` u kom se čuva procesorski kontekst niti;
- `const size_t STACK_SIZE`: veličina prostora za stek niti (u jednicama `sizeof(char)`);
- `Thread::stack`: pokazivač tipa `void*` koji ukazuje na adresu početka dela memorije u kom je alociran stek niti;
- `jmp_buf::sp`: polje za sačuvanu vrednost registra SP; stek raste ka nižim adresama, a SP ukazuje na prvu slobodnu lokaciju veličine `sizeof(int)`;
- `Scheduler::put(Thread*)`: statička članica kojom se u red spremnih stavlja data nit;
- `void* malloc(size_t sz)`: standardna funkcija koja alocira prostor veličine `sz` (u jedinicama `sizeof(char)`); funkcija `free(void*)` oslobađa ovako alociran prostor;
- `memcpy(vodi* dst, const void* src, size_t size)`: kopira memorijski sadržaj veličine `size` sa mesta na koje ukazuje `src` na mesto na koje ukazuje `dst`;
- `setjmp()`, `longjmp()`: standardne bibliotečne funkcije.

Pomoću ovih elemenata implementirati sistemski poziv školskog jezgra – funkciju:

```
Thread* t_fork();
```

kojom se kreira nova nit kao „klon“ pozivajuće, roditeljske niti, sa istim početnim kontekstom, ali sa sopstvenom kontrolom toka, po uzoru na standardni sistemski poziv `fork()` za procese. U slučaju uspeha, u kontekstu roditeljske niti ova funkcija treba da vrati pokazivač na objekat niti deteta, a u kontekstu niti deteta treba da vrati 0; u slučaju greške, treba da podigne izuzetak tipa `ThreadCreationException`.

Rešenje

```
Thread* t_fork() {
    lock();
    // Allocate a new stack:
    void* stck = malloc(STACK_SIZE);
    if (!stck) throw ThreadCreationException();
    // and copy its contents from the parent's stack:
    memcpy(stck, Thread::running->stack, STACK_SIZE);
    // Create a new Thread object:
    Thread* newThr = new Thread();
    if (!newThr) {
        free(stck);
        throw ThreadCreationException();
    }
    newThr->stack = stck;
    if (setjmp(newThr->context) == 0) {
        // Parent thread:
        // and set its stack pointer:
        newThr->context->sp = newThr->context->sp - Thread::running->stack + stck;
        // Put the new thread to the ready list and return:
        Scheduler::put(newThr);
        unlock();
        return newThr;
    } else {
        // Child thread:
        unlock();
        return 0;
    }
}
```

2. zadatak, prvi kolokvijum, jun 2019.

Raspoređivanje i promena konteksta u školskom jezgru implementirani su na sledeći način:

- Svi objekti klase `Thread` smešteni su u statički vektor `Thread::allThreads` tipa `Thread[NumOfThreads]`.
- Nestatički podatak član `Thread::isRunnable` govori o tome da li je odgovarajuća nit spremna za izvršavanje ili se baš ona izvršava, ili nije (jer je suspendovana).
- Statički podatak član `Thread::running` tipa `int` ukazuje na onaj element niza `allThreads` koji predstavlja nit koja se trenutno izvršava (tekuća nit).
- Za izvršavanje se bira prva sledeća spremna nit u nizu `allThreads` iza one tekuće, i tako u krug.
- Nestatički podatak član `Thread::context` tipa `jmp_buf` čuva procesorski kontekst niti.
- Implementirana je funkcija `yield(jmp_buf oldC, jmp_buf newC)` koja čuva kontekst procesora u prvi argument i restaurira kontekst iz drugog argumenta.

Korišćenjem date funkcije `yield`, implementirati funkciju `dispatch` koja treba da preda procesor niti koja je data kao argument, pod uslovom da je taj argument dat i da je ta nit spremna. Ako je data nit spremna, funkcija treba da vrati 0; u suprotnom, tekuća nit treba da nastavi izvršavanje, a ova funkcija da vrati -1. Ako nije zadata nit, ova funkcija treba da preda procesor sledećoj spremnoj niti i da vrati 0.

```
int dispatch (Thread* newT = 0);
```

Rešenje

```
int dispatch (Thread* newT = 0) {
  lock();
  if (newT && !newT->isRunnable) {
    unlock();
    return -1;
  }
  int oldR = Thread::running;
  jmp_buf oldC = Thread::allThreads[oldR].context;
  if (newT)
    Thread::running = newT - Thread::allThreads;
  else
    do
      Thread::running = (Thread::running+1)%NumOfThreads;
    while (!Thread::allThreads[Thread::running].isRunnable &&
           Thread::running!=oldR);
  if (Thread::running!=oldR) {
    jmp_buf newC = Thread::allThreads[Thread::running].context;
    yield(oldC,newC);
  }
  unlock();
  return 0;
}
```

2. zadatak, prvi kolokvijum, april 2017.

U nekom asimetričnom multiprocesorskom operativnom sistemu jedan od procesora posvećen je samo obavljanju ulazno-izlaznih operacija koje su zahtevali korisnički procesi. Operacije sa svakim pojedinčanim ulazno-izlaznim uređajem obavlja po jedna interna kernel nit predstavljena objektom klase `IOThread`. Ove niti izvršavaju se samo na ovom posvećenom procesoru i nemaju nikakve veze sa ostalim nitima kernela niti korisničkim procesima (osim preuzimanja zahteva koje su oni postavili); na ovom procesoru izvršavaju se samo ove niti.

Svaka od tih niti ima sledeći generički oblik: ona uzme jedan zahtev za ulazno-izlaznom operacijom iz reda zahteva postavljenih za taj uređaj, pokrene operaciju sa uređajem na način specifičan za taj uređaj, a onda se suspenduje pozivom operacije `IOThread::suspend` dok uređaj ne signalizira spremnost za novu operaciju spoljašnjim prekidom. Prekidne rutine svih tih uređaja ne vrše promenu konteksta (preotimanje procesora), već samo postavljaju polje

`IOThread::isReady` svoje niti na 1, čime ta nit ponovno postaje spremna i može da zada novu operaciju. Zbog svega ovoga nije potrebno raditi nikakvo maskiranje prekida niti međusobno isključenje sa ostalim procesorima.

Sve ove niti predstavljene su objektima klase `IOThread` u statičkom nizu `IOThread::allThreads`, a njihov broj je konstantan i iznosi `IOThread::NumberOfThreads`. Polje `IOThread::running` je pokazivač na nit koja se trenutno izvršava na ovom posvećenom procesoru. Na raspolaganju je funkcija:

```
void IOThread::yield (IOThread* oldThread, IOThread* newThread);
```

koja čuva kontekst tekuće niti u za to predviđeno polje objekta na koga ukazuje prvi argument i restaurira kontekst niti iz polja objekta na koga ukazuje drugi argument. Implementirati operaciju `IOThread::suspend`. Za izvršavanje je dovoljno uzeti prvu (ili bilo koju drugu) spremnu nit iz niza `IOThread::allThreads`. Obratiti pažnju na to da je moguće da nijedna nit nije spremna za izvršavanje, jer su sve suspendovane i čekaju na završetak svojih operacija i prekide od svojih uređaja koji će ih ponovo učiniti spremnim; u tom slučaju procesor treba da uposleno čeka dok neka nit ne postane spremna.

Rešenje

```
void IOThread::suspend () {
  IOThread::running->isReady = 0;
  int newRunning = -1;
  while (newRunning==-1) {
    for (int i=0; i<IOThread::NumberOfThreads; i++)
      if (IOThread::allThreads[i].isReady) {
        newRunning = i;
        break;
      }
  }
  IOThread* oldThread = IOThread::running;
  IOThread* newThread = &IOThread::allThreads[newRunning];
  IOThread::running = newThread;
  yield(oldThread,newThread);
}
```

2. zadatak, prvi kolokvijum, jun 2017.

U školskom jezgru promena konteksta implementirana je korišćenjem date funkcije `yield()`, u sistemskom pozivu `dispatch()` i na svim ostalim mestima na sličan način kao što je dato.

Potrebno je implementirati sistemski poziv (statičku operaciju):

```
void Thread::wait(Thread* forChild=0);
```

kojim pozivajuća nit čeka (suspenduje se ako je potrebno) dok se ne završi nit-dete na koje ukazuje argument, odnosno sve niti-deca koje je ova pozivajuča nit do tada kreirala, ako je ovaj argument *null*. Za te potrebe treba implementirati i sledeće nestatičke funkcije-članice:

- `void Thread::created(Thread* parent)`: poziva je jezgro interno za datu novokreiranu nit (`this`), kada je ta nit kreirana, sa argumentom `parent` koji ukazuje na roditeljsku nit u čijem kontekstu je ova nova nit-dete kreirana;
- `void Thread::completed()`: poziva je jezgro za datu nit (`this`), kada se ta nit završila.

Ukoliko proširujete klasu `Thread` novim članovima, precizno navedite kako.

```
void yield (jmp_buf oldBuf, jmp_buf newBuf) {
  if (setjmp(oldBuf)==0) longjmp(newBuf,1);
}

void Thread::dispatch () {
  lock();
  jmp_buf oldBuf = Thread::running->context;
```

```
  Scheduler::put(Thread::running);
  Thread::running = Scheduler::get();
  jmp_buf newBuf = Thread::running->context;
  yield(oldBuf, newBuf);
  unlock();
}
```

Rešenje

U klasu `Thread` dodati su sledeći privatni, nestatički podaci-članovi sa datim inicijalnim vrednostima:

```
Thread* Thread::parent = 0;
bool Thread::isActive = false;
bool Thread::isWaitingForAllChildren = false;
unsigned long Thread::activeChildrenCounter = 0;
Thread* Thread::isWaitingForChild = 0;
```

```
void Thread::created (Thread* par) {
  this->isActive = true;
  this->parent = par;
  if (par) this->parent->activeChildrenCounter++;
}

void Thread::completed () {
  this->isActive = false;
  if (!this->parent) return;
  this->parent->activeChildrenCounter--;
  if ((this->parent->isWaitingForAllChildren &&
       this->parent->activeChildrenCounter==0) ||
      this->parent->isWaitingForChild==this) {
    this->parent->isWaitingForAllChildren = false;
    this->parent->isWaitingForChild = 0;
    Scheduler::put(this->parent);
  }
}

void Thread::wait (Thread* forChild=0) {
  lock();
  jmp_buf old = Thread::running->context;

  if (forChild==0)
    if (Thread::running->activeChildrenCounter>0)
      Thread::running->isWaitingForAllChildren = true;
    else
      Scheduler::put(Thread::running);
  else
    if (forChild->parent==Thread::running && forChild->isActive)
      Thread::running->isWaitingForChild = forChild;
    else
      Scheduler::put(Thread::running);

  Thread::running = Scheduler::get();
  jmp_buf new = Thread::running->context;
  yield(old,new);
  unlock();
}
```

2. zadatak, prvi kolokvijum, maj 2016.

U školskom jezgru promena konteksta implementirana je korišćenjem date funkcije `yield()`, u sistemskom pozivu `dispatch()` i na svim ostalim mestima na sličan način kao što je dato.

Potrebno je implementirati sistemski poziv (statičku operaciju) `Thread::wait()` kojim pozivajuća nit čeka (suspenduje se ako je potrebno) dok se ne završe sve niti-deca koje je ova pozivajuća nit do tada kreirala. Za te potrebe treba implementirati i sledeće nestatičke funkcije-članice:

- `void Thread::created(Thread* parent)`: poziva je jezgro interno za datu novokreiranu nit (`this`), kada je ta nit kreirana, sa argumentom `parent` koji ukazuje na roditeljsku nit u čijem kontekstu je ova nova nit-dete kreirana;
- `void Thread::completed()`: poziva je jezgro za datu nit (`this`), kada se ta nit završila.

Ukoliko proširujete klasu `Thread` novim članovima, precizno navedite kako.

```
void yield (jmp_buf old, jmp_buf new) {
  if (setjmp(old)==0) longjmp(new,1);
}

void Thread::dispatch () {
  lock();
  jmp_buf old = Thread::running->context;
  Scheduler::put(Thread::running);
  Thread::running = Scheduler::get();
  jmp_buf new = Thread::running->context;
  yield(old,new);
  unlock();
}
```

Rešenje

U klasu `Thread` dodati su sledeći privatni, nestatički podaci-članovi sa datim inicijalnim vrednostima:

```
Thread* Thread::parent = 0;
bool Thread::isWaitingForChildren = false;
unsigned long Thread::activeChildrenCounter = 0;
```

```
void Thread::created (Thread* parent) {
  this->parent = parent;
  if (parent) parent->activeChildrenCounter++;
}

void Thread::completed () {
  if (this->myParent && --this->myParent->activeChildrenCounter==0 &&
      this->myParent->isWaitingForChildren) {
    this->myParent->isWaitingForChildren = false;
    Scheduler::put(this->myParent);
  }
}

void Thread::wait () {
  lock();
  jmp_buf old = Thread::running->context;
  if (Thread::running->activeChildrenCounter>0)
    Thread::running->isWaitingForChildren = true;
  else
    Scheduler::put(Thread::running);
  Thread::running = Scheduler::get();
  jmp_buf new = Thread::running->context;
```

```
  yield(old,new);
  unlock();
}
```

2. zadatak, prvi kolokvijum, septembar 2015.

Školsko jezgro proširuje se konceptom tzv. asinhronih signala, koji podržavaju mnogi operativni sistemi, sa sledećim značenjem.

Kernel može „poslati“ signal nekoj korisničkoj niti. Signal je celobrojna konstanta u opsegu `1..SIGS-1`. Čim ta nit ponovo dobije procesor, umesto da odmah nastavi izvršavanje tamo gde je bila prekinuta, odnosno izgubila procesor, najpre će skočiti u proceduru za obradu tog signala, tzv. *signal handler*, pa kada se iz te procedure vrati, nastaviće dalje izvršavanje tamo gde je ono bilo prekinuto.

Na procedure za obradu signala pokazuju pokazivači u tabeli koja se nalazi u PCB svake niti, poput vektor tabele, u nizu `Thread::sigHandlers[SIGS]`. Za svaku nit, signalu n odgovara procedura za obradu na koju ukazuje pokazivač u ulazu `Thread::sigHandlers[n]` (ulaz 0 se ne koristi). Kada „šalje“ signal nekoj niti, kernel u PCB te niti postavi vrednost signala n u polje `Thread::signal`; vrednost različita od 0 u ovom polju ukazuje na postojanje poslatog signala, dok 0 znači da signala nema.

Dole je data implementacija sistemskog poziva `dispatch()` u školskom jezgru. Modifikovati ovu implementaciju tako da podrži obradu poslatog signala datoj niti. Smatrati da su na isti ovakav način realizovani i svi ostali sistemski pozivi i preuzimanja procesora.

```
void yield (jmp_buf old, jmp_buf new) {
  if (setjmp(old)==0)
    longjmp(new,1);
}

void dispatch () {
  lock();
  jmp_buf old = Thread::running->context;
  Scheduler::put(Thread::running);
  Thread::running = Scheduler::get();
  jmp_buf new = Thread::running->context;
  yield(old,new);
  unlock();
}
```

Rešenje

```
void yield (jmp_buf old, jmp_buf new) {
  if (setjmp(old)==0)
    longjmp(new,1);
}

void dispatch () {
  lock();
  jmp_buf old = Thread::running->context;
  Scheduler::put(Thread::running);
  Thread::running = Scheduler::get();
  jmp_buf new = Thread::running->context;
  yield(old,new);
  unlock();
  if (Thread::running->signal) {
    Thread::running->sigHandlers[Thread::running->signal]();
    Thread::running->signal = 0;
  }
}
```

4. zadatak, prvi kolokvijum, april 2014.

U nekom sistemu svi sistemski pozivi vrše se softverskim prekidom broj 44h, pri čemu u registru `r0` sistem očekuje identifikator sistemskog poziva. U ovom sistemu postoji sistemski poziv za kreiranje i pokretanje niti nad potprogramom koji prihvata jedan argument tipa pokazivača. Ovaj poziv u registrima očekuje sledeće argumente:

- u `r0` treba da bude identifikator ovog poziva, 0 u ovom slučaju;
- u `r1` treba da bude adresa potprograma nad kojim se kreira nit;
- u `r2` treba da bude argument potprograma nad kojim se kreira nit.

Svaki sistemski poziv vraća rezultat u registru `r0`, a za ovaj sistemski poziv rezultat je identifikator niti u jezgru (ceo broj veći od 0), odnosno kod greške (ceo broj manji od 0). U asemblerskim blokovima unutar C/C++ koda može se koristiti simbolička konstanta sa nazivom formalnog argumenta funkcije unutar koje se nalazi dati asemblerski kod; ova simbolička konstanta ima vrednost odgovarajućeg pomeraja lokacije u kojoj se nalazi dati formalni argument u odnosu na vrh steka (kao što je pokazano u primerima na predavanjima). C/C++ funkcije vraćaju vrednost u registru `r0` ukoliko je tip povratne vrednosti odgovarajući.

1. Implementirati funkciju

   ```
   int create_thread (void (*f)(void*), void* arg);
   ```

   koja vrši opisani sistemski poziv (kreira nit nad funkcijom `f` sa argumentom `arg`) i koja može da se poziva iz C koda sa zadatim argumentima. Ova funkcija vraća identifikator kreirane niti koji se može koristiti u ostalim sistemskim pozivima da identifikuje tu nit u jezgru, odnosno kod greške.

2. Korišćenjem prethodne funkcije, implementirati funkciju `start` klase `Thread`. Ova klasa obezbeđuje koncept niti u objektnom duhu, poput onog u školskom jezgru (kreira nit nad virtuelnom funkcijom `run`).

   ```
   class Thread {
   public:
     Thread () : pid(0) {}
     int start ();
     virtual void run () {}
   private:
     int pid;  // Thread ID
   };
   ```

Rešenje

```
int create_thread (void (*f)(void*), void* arg) {
  asm {
    load r0,#0
    load r1,#f[sp]
    load r2,#arg[sp]
    int 44h
  }
}

void wrapper (void* t) {
  if(t)((Thread*)t)->run();
}

int Thread::start () {
  if(pid)
    return pid;
  else
    return pid = create_thread(&wrapper,this);
}
```

3. zadatak, prvi kolokvijum, maj 2012.

Korišćenjem standardnih bibliotečnih funkcija `setjmp()` i `longjmp()`, realizovati operaciju

```
yield(jmp_buf old, jmp_buf new);
```

koja čuva čuva kontekst niti čiji je `jmp_buf` dat kao prvi argument, oduzima joj procesor i restaurira kontekst niti čiji je `jmp_buf` dat kao drugi argument, kojoj predaje procesor.

Koristeći ovu operaciju `yield()`, realizovati operaciju `dispatch()` koja ima isti efekat kao i ona data u školskom jezgru.

Rešenje

```
void yield (jmp_buf old, jmp_buf new) {
  if (setjmp(old)==0)
    longjmp(new,1);
}

void dispatch () {
  lock();
  jmp_buf old = Thread::running->context;
  Scheduler::put(Thread::running);
  Thread::running = Scheduler::get();
  jmp_buf new = Thread::running->context;
  yield(old,new);
  unlock();
}
```

3. zadatak, prvi kolokvijum, septembar 2012.

U nekom sistemu realizovana je operacija

```
void yield (jmp_buf old, jmp_buf new);
```

koja čuva kontekst niti čiji je `jmp_buf` dat kao prvi argument, oduzima joj procesor i restaurira kontekst niti čiji je `jmp_buf` dat kao drugi argument, kojoj predaje procesor.

Koristeći ovu operaciju `yield()`, realizovati operacije:

- `Thread::suspend()`: (statička) suspenduje (blokira) izvršavanje pozivajuće niti sve dok je neka druga nit ne „probudi“ pomoću `resume()`;
- `Thread::resume()`: (nestatička) „budi“ (deblokira) nit za koju je pozvana; vrši i promenu konteksta predajući procesor niti koja je na redu za izvršavanje.

Pretpostaviti da je lista spremnih procesa uvek neprazna prilikom suspenzije niti i da je nit za koju se poziva `resume` sigurno suspendovana (ignorisati mogućnost greške). Klase `Thread` i `Scheduler` su u preostalim delovima implementirane kao u školskom jezgru.

Rešenje

```
void Thread::suspend () {
  lock();
  jmp_buf old = Thread::running->context;
  Thread::running = Scheduler::get();
  jmp_buf new = Thread::running->context;
  yield(old,new);
  unlock();
}

void Thread::resume () {
  lock();
```

```
  Scheduler::put(this);
  jmp_buf old = Thread::running->context;
  Scheduler::put(Thread::running);
  Thread::running = Scheduler::get();
  jmp_buf new = Thread::running->context;
  yield(old,new);
  unlock();
}
```

3. zadatak, prvi kolokvijum, april 2010.

U donju tabelu upisati "Da" ukoliko operativni sistem datu operaciju treba, a "Ne" ukoliko ne treba da izvrši prilikom promene konteksta procesa, odnosno niti.

Operacija	Promena konteksta procesa	Promena konteksta niti
Restauracija programski dostupnih registara procesora za podatke		
Invalidacija TLB-a čiji ključevi ne sadrže identifikaciju procesa		
Invalidacija TLB-a čiji ključevi sadrže identifikaciju procesa		
Restauracija registra koji čuva identifikaciju korisničkog procesa		
Restauracija pokazivača na tabelu preslikavanja stranica (PMTP)		
Restauracija procesorske statusne reči (PSW)		
Invalidacija procesorskog keša koji kao ključeve čuva virtuelne adrese		
Invalidacija procesorskog keša koji kao ključeve čuva fizičke adrese		
Zatvaranje otvorenih fajlova		
Restauracija pokazivača steka (SP)		

Rešenje

Operacija	Promena konteksta procesa	Promena konteksta niti
Restauracija programski dostupnih registara procesora za podatke	Da	Da
Invalidacija TLB-a čiji ključevi ne sadrže identifikaciju procesa	Da	Ne
Invalidacija TLB-a čiji ključevi sadrže identifikaciju procesa	Ne	Ne
Restauracija registra koji čuva identifikaciju korisničkog procesa	Da	Ne
Restauracija pokazivača na tabelu preslikavanja stranica (PMTP)	Da	Ne
Restauracija procesorske statusne reči (PSW)	Da	Da
Invalidacija procesorskog keša koji kao ključeve čuva virtuelne adrese	Da	Ne
Invalidacija procesorskog keša koji kao ključeve čuva fizičke adrese	Ne	Ne
Zatvaranje otvorenih fajlova	Ne	Ne
Restauracija pokazivača steka (SP)	Da	Da

5. zadatak, prvi kolokvijum, april 2008.

U jezgru nekog jednoprocesorskog operativnog sistema definisani su sledeći elementi:

```
const int IVTSize = ...; // Number of interrupt entries
typedef int InerruptID;  // IVT Entry Number

class Thread {
public: ...
  jmp_buf context;
  Thread* next; // Pointer to the next thread in a (waiting or ready) list
};

Thread* running; // The running thread
Thread* readyFirst; // The head of the ready list
Thread* readyLast; // The tail of the ready list
```

Implementirane su i standardne funkcije `setjmp()` i `longjmp()` koje čuvaju, odnosno restauriraju sve programski dostupne registre (uključujući i PC i SP). Pretpostavlja se da lista spremnih niti nikad nije prazna – uvek je u njoj bar jedna podrazumevana nit koja izvršava praznu petlju i nikad se ne suspenduje i ne gasi, tzv. besposlena (engl. *idle*) nit koja u svakoj svojoj iteraciji vrši eksplicitno preuzimanje (koje ne treba realizovati). Korišćenjem ovih elemenata, implementirati sledeće funkcije:

- `void suspend(InterruptID)`: suspenduje (blokira) izvršavanje tekuće niti sve dok ga ne „probudi“ prekid sa datim brojem;
- `void resume(InterruptID)`: „budi“ (deblokira) izvršavanje jedne niti suspendovane na prekidu sa datim brojem.

Pretpostavlja se da su svi ulazi IVT tabele u opsegu `0..IVTSize-1` već inicijalizovani tako da ukazuju na po jednu jednostavnu prekidnu rutinu, pri čemu sve one prosto pozivaju operaciju `resume()` sa argumentom koji predstavlja broj tog prekida i ne rade ništa više, a razlikuju se samo po tom broju. Prekidi se mogu maskirati i demaskirati operacijama `lock()` i `unlock()`, respektivno.

Rešenje

```
// Auxiliary list operations:

void put(Thread* t, Thread*& head, Thread*& tail) {
  if (t==0) return; // Error
  t ->next=0;
  if (tail) tail->next=t;
  else head=t;
  tail=t;
};
Thread* get(Thread*& head, Thread*& tail) {
  Thread* t = head;
  if (t==0) return;
  t ->next=0;
  head=head->next;
  if (head==0) tail=0;
}

class InterruptWaitingQueue {
public:
  static InterruptWaitingQueue* Instance();
  void put(InterruptID,Thread*);
  Thread* get(InterruptID);
private:
  struct InterruptWaitingQueueEntry {
    Thread* head;
    Thread* tail;
    InterruptWaitingQueueEntry():head(0),tail(0) {}
  };
  InterruptWaitingQueueEntry queue[IVTSize];
};

InterruptWaitingQueue* InterruptWaitingQueue::Instance() {
  static InterruptWaitingQueue instance;
  return &instance;
}
void InterruptWaitingQueue::put(InterruptID intID, Thread* t) {
  put(t,queue[intID].head,queue[intID].tail);
}

Thread* InterruptWaitingQueue::get(InterruptID intID) {
```

```
  return get(queue[intID].head,queue[intID].tail);
}

void suspend (InterruptID intID) {
  if (intID>=IVTSize) return; // Error;
  lock();
  if (setjmp(running->context)==0) {
    // Suspend the running thread:
    InterruptWaitingQueue::Instance()->put(running);
    // Get a new running thread:
    running=get(readyFirst,readyLast);
    longjmp(running->context);
  };
  unlock();
}

void resume (InterruptID intID) {
  if (intID>=IVTSize) return; // Error;
  Thread* t = InterruptWaitingQueue::Instance()->get(intID);
  if (t==0) return; // No effect
  put(t,readyFirst,readyLast);
}
```

4. zadatak, prvi kolokvijum, april 2007.

U jezgru nekog operativnog sistema definisane su sledeće strukture podataka:

```
typedef unsigned int PID;
#define MaxProc ... // Max number of processes
struct PCB {
  int sp;     // Saved stack pointer
  PID next;         // Next PCB in the Ready or a waiting list
  ...
};
PCB* allProcesses[MaxProc]; // Maps a PID to a PCB*
PID running; // The running process
PID ready;   // Head of Ready list
```

Korišćenjem ovih struktura, po uzoru na funkciju datu na predavanjima, na jeziku C i asembleru zamišljenog procesora napisati telo funkcije `yield(PCB* current, PCB* next)` koja vrši promenu konteksta sa procesa `current` na proces `next`; ova funkcija služi samo za realizaciju drugih operacija jezgra i poziva se samo iz koda jezgra. Korišćenjem te funkcije, na jeziku C napisati tela sledećih funkcija:

- `suspend()`: suspenduje (blokira) izvršavanje pozivajućeg procesa sve dok ga neki drugi proces ne „probudi“ pomoću `resume()`;
- `resume(PID pid)`: „budi“ (deblokira) izvršavanje procesa koji je zadat argumentom.

Rešenje

```
typedef unsigned int PID;
#define MaxProc ... // Max number of processes
struct PCB {
  int sp;     // Saved stack pointer
  PID next;         // Next PCB in the Ready or a waiting list
  ...
};
PCB* allProcesses[MaxProc]; // Maps a PID to a PCB*
PID running; // The running process
```

```
PID ready;   // Head of Ready list

void yield(PCB* current, PCB* next){
  asm{
    push R0
    ...
    push Rn

    load R0, #current[BP]
    load #sp[R0], SP //#sp je pomjeraj odgovarajuceg polja strukture
    load R0, #next[BP]   // i u ovom slucaju je #sp=0
    load SP, #sp[R0]

    pop Rn
    ...
    pop R0
  }
  return;
}
```

Drugi način za `yield` (Neki kompajleri posjeduju registarske pseudovarijable, pa je registrima moguće pristupiti i direktno iz C odnosno C++ koda. Jedan od takvih je TC.):

```
void yield(PCB* current, PCB* next){
  asm{
    push R0
    ...
    push Rn
  }
  current->sp = _SP;    //_SP je pseudovarijabla za SP
  _SP = next->sp;

  asm{
    pop Rn
    ...
    pop R0
  }
  return;
}


void suspend(){
  PCB* old = allProcesses[running];
  running = ready;
  ready = allProcesses[ready]->next;
  PCB* new = allProcesses[running];
  allProcesses[running]->next=0;
  yield(old, new);
  return;
}

resume(PID pid){
  int ind=(running != pid);
  for(PID i = 0;i < MaxProc;i++) if (allProcesses[i].next == pid) ind=0;
  if (ind){
    allProcesses[pid]->next = ready;
    ready = pid;
```

```
  }
}
```

5. zadatak, prvi kolokvijum, april 2006.

U jezgru nekog operativnog sistema definisane su sledeće strukture podataka:

```
typedef unsigned int PID;
#define MaxProc ... // Max number of processes
struct PCB {
  PID pid; // Process ID
  jmp_buf context;
  PCB* next;        // Next PCB in the Ready or a waiting list
  ...
};
PCB* allProcesses[MaxProc]; // Maps a PID to a PCB*
PCB* running; // The running process
PCB* ready;   // Head of Ready list
```

Korišćenjem bibliotečnih funkcija `setjmp()` i `longjmp()`, na jeziku C napisati tela sledećih funkcija:

- `suspend()`: suspenduje (blokira) izvršavanje pozivajućeg procesa sve dok ga neki drugi proces ne „probudi“ pomoću `resume()`;
- `resume(PID pid)`: „budi“ (deblokira) izvršavanje procesa koji je zadat argumentom.

Rešenje

Potrebno je dodati sledeću deklaraciju, koja će predstavljati pokazivač na listu suspendovanih (blokiranih) procesa.

```
PCB* blocked;
```

Implementacije traženih funkcija:

```
void suspend () {
  lock();
  if (setjmp(running->context)==0) {
    // stavlja running proces u listu blokiranih
    running->next = blocked;
    blocked = running;

    // uzima prvi spreman proces i dodeljuje mu procesor
    running = ready;
    ready = ready->next;

    longjmp(running->context,1);
  } else {
     unlock ();
     return;
  }
}

void resume(PID pid){
  if (blocked == 0) return;

  lock();

  // pronalazi PCB procesa koji treba deblokirati
  PCB *prev = 0, *curr = blocked;
  while (curr != 0 && curr->pid != pid){
     prev = curr;
```

```
    curr = curr->next;
  }

  if (curr == 0) {
    unlock();
    return;
  }

  // vraća nađeni PCB u listu spremnih
  if (prev == 0) blocked = blocked->next;
  else prev->next = curr->next;

  curr->next = ready;
  ready = curr;

  unlock();
}
```

1. zadatak, drugi kolokvijum, maj 2006.

U nekom računarskom sistemu postoje spoljašnji maskirajući prekidi koje generišu ulazno/izlazni uređaji. Ovi prekidi maskiraju se operacijom `int_mask()`, a demaskiraju operacijom `int_unmask()`. Po prihvatanju prekida, procesor automatski maskira prekide. Potrebno je realizovati koncept spoljašnjeg događaja (*external event*) koji se manifestuje pojavom spoljašnjeg maskirajućeg prekida. Ovaj koncept treba realizovati klasom `ExternalEvent`, po uzoru na realizaciju semafora pomoću operacija `setjmp()` i `longjmp()` datu na predavanjima. Pretpostaviti da operacije `setjmp()` i `longjmp()` čuvaju celokupan kontekst izvršavanja, tj. sve programski dostupne registre. Interfejs klase `ExternalEvent` treba da izgleda ovako:

```
const int NumOfInterruptEntries = ... ; // Number of interrupt entries

class ExternalEvent {
public:
  ExternalEvent(int interruptNo);
  void wait();

  static void interruptOccurrence(int interruptNo);
};
```

Konstruktor ove klase kreira jednu instancu događaja koji se signalizira prekidom sa brojem ulaza zadatim parametrom konstruktora. Operacija `wait()` blokira pozivajuću nit sve dok se prekid ne dogodi. Pojava prekida na ulazu N deblokira jednu nit blokiranu na događaju koji je vezan za prekid sa tim brojem ulaza N. Pretpostaviti da u sistemu već postoje prekidne rutine vezane za spoljašnje maskirajuće prekide tako da prekidna rutina za prekid na ulazu N poziva funkciju `ExternalEvent::interruptOccurrence(int)` sa argumentom koji predstavlja taj broj ulaza.

Rešenje

```
// ExternalEvent.h
void int_mask();
void int_unmask();

const int NumOfInterruptEntries = ... ; // Number of interrupt entries

class ExternalEvent {
public:
  ExternalEvent(int interruptNo);
  void wait();

  static void interruptOccurrence(int interruptNo);
```

```
protected:
  void signal ();

  void block ();
  void deblock ();

  ~ExternalEvent();
  static ExternalEvent* events[NumOfInterruptEntries];

private:
  int val, intNo;
  Queue blocked;
};

// ExternalEvent.cpp

ExternalEvent::ExternalEvent(int interruptNo) : val(0),
intNo(interruptNo){
   if (intNo>=0 && intNo<NumOfInterruptEntries) {
      // Razmisliti zašto je ovo neophodno.
      // Uputstvo: šta ako je pokazivač veličine više reči koje se upisuju neatomično u narednoj naredbi?
      int_mask();
      // add the event to the event list
      events[intNo]=this;
      int_unmask();
   }
}

ExternalEvent::~ExternalEvent(){
  // Isto kao i gore!
  int_mask();
  events[intNo]=0;
  int_unmask();
}

void ExternalEvent::interruptOccurrence(int interruptNo){
   if (interruptNo intNo>=0 && interruptNo intNo<NumOfInterruptEntries
  && events[interruptNo])
     events[interruptNo]->signal();
}

void ExternalEvent::block () {
  if (setjmp(Thread::runningThread->context)==0) {
    // Blocking:
    blocked->put(Thread::runningThread);
    Thread::runningThread = Scheduler::get();
    longjmp(Thread::runningThread->context);
  } else return;
}

void ExternalEvent::deblock () {
  // Deblocking:
  Thread* t = blocked->get();
  Scheduler::put(t);
}
```

```
void ExternalEvent::wait () {
  int_mask();
  if (--val<0) block();
  int_unmask();
}

void ExternalEvent::signal () {
  if (val++<0) deblock();
}
```

Sinhronizacija procesa (implementacija)

2. zadatak, drugi kolokvijum, jun 2022.

U školsko jezgro dodaje je se koncept *uslova* (engl. *condition*) za uslovnu sinhronizaciju, podržan klasom `Condition` čiji je interfejs dat dole. Uslov može biti ispunjen ili neispunjen; inicijalna vrednost zadaje se konstruktorom. Niti koje smeju da nastave izvršavanje iza neke tačke samo ako je uslov ispunjen treba da pozovu operaciju `wait`, koja ih suspenduje ako uslov nije ispunjen. Bilo koja nit koja ispuni uslov to objavljuje pozivom operacije `set`; sve niti koje čekaju na taj uslov tada nastavljaju izvršavanje. Kada neka nit promeni uslov tako da on više nije ispunjen, treba da pozove operaciju `clear`. Implementirati klasu `Condition`.

```
class Condition {
public:
    Condition(bool init = false);
    void set();
    void clear();
    void wait();
};
```

Rešenje

```
class Condition {
public:
    Condition(bool init = false) : cond(init) {}
    void set();
    void clear() {
        lock();
        cond = false;
        unlock();
    }
    void wait();
private:
    bool cond;
    Queue blocked;
};
void Condition::wait() {
    lock();
    if (!cond) {
        if (setjmp(Thread::runningThread->context) == 0) {
            blocked.put(Thread::runningThread);
            Thread::runningThread = Scheduler::get();
            longjmp(Thread::runningThread->context, 1);
        }
    }
    unlock();
}
void Condition::set() {
    lock();
    cond = true;
    for (Thread* t = blocked.get(); t; t = blocked.get()) {
        Scheduler::put(t);
    }
    unlock();
}
```

2. zadatak, kolokvijum, jun 2021.

U Školskom jezgru implementirana je statička operacija klase `Thread`

```
void Thread::yield (Thread* oldRunning, Thread* newRunning);
```

koja obavlja promenu konteksta oduzimajući procesor (tekućoj) niti na koju pokazuje parametar `oldRunning` i predajući procesor niti na koju pokazuje parametar `newRunning`. Korišćenjem ove operacije implementirati operacije `wait` i `signal` klase `Semaphore`, s tim da se uvek, pa i kod neblokirajućih poziva obavlja promena konteksta ukoliko raspoređivač odabere neku drugu nit za izvršavanje. U redu spremnih uvek se nalazi barem neka nit, makar nit *idle* koja ne radi ništa korisno (samo troši procesorsko vreme instrukcijama bez efekta).

Rešenje

```
void Semaphore::wait() {
    lock(lck);
    Thread* oldRunning = Thread::runningThread;
    if (--val < 0)
        this->blocked.put(oldRunning);
    else
        Scheduler::put(oldRunning);
    Thread* newRunning = Thread::runningThread = Scheduler::get();
    if (oldRunning != newRunning)
        Thread::yield(oldRunning, newRunning);
    unlock(lck);
}

void Semaphore::signal() {
    lock(lck);
    if (++val <= 0)
        Scheduler::put(this->blocked.get());
    Thread* oldRunning = Thread::runningThread;
    Scheduler::put(oldRunning);
    Thread* newRunning = Thread::runningThread = Scheduler::get();
    if (oldRunning != newRunning)
        Thread::yield(oldRunning, newRunning);
    unlock(lck);
}
```

1. zadatak, kolokvijum, avgust 2020.

Školsko jezgro proširuje se konceptom koji služi za alokaciju i dealokaciju nedeljivog resursa u sistemu, ponaša se kao binarni semafor, a implementiran je klasom `Resource` čiji je interfejs dat u nastavku. Kada želi da zauzme neki deljeni resurs, nit treba da pozove operaciju `acquire` objekta ove klase koji služi za međusobno isključenje pristupa tom resursu, a kada ga oslobodi, treba da pozove operaciju `release`. Da bi se sprečila pojava mrtve blokade (*deadlock*), koristi se algoritam „čekaj ili umri“ (*wait-die*) koji funkcioniše na sledeći način. Svakoj niti pri samom kreiranju pridružuje se jedinstvena „vremenska marka“ (ceo broj dostupan u atributu `Thread::timestamp`), tako da starije niti imaju manju vrednost ove marke. Kada nit *Ta* zahteva resurs koji drži zauzeta nit *Tb*, onda:

- Ako je *Ta* starija nego *Tb* (*Ta*<*Tb*), nit *Ta* čeka suspendovana dok ne dobije resurs i tada operacija `acquire` vraća 1 (istu vrednost ova operacija vraća i kada je resurs slobodan, onda ga nit odmah dobija);
- Ako je *Ta* mlađa nego *Tb* (*Ta*>*Tb*), niti Ta se odbija zahtev tako što operacija `acquire` vraća 0, a nit onda tu situaciju obrađuje na odgovarajući način (pokušava ponovo ili odustaje).

Implementirati u potpunosti klasu `Resource`.

```
class Resource {
public:
  Resource ();
  int acquire();
  void release();
};
```

Rešenje

Pokazana je jedna jednostavna implementacija koja zadovoljava uslove, koristi jedostavan FIFO red zahteva, ali koja ima značajan stepen odbijanja zahteva (red zahteva je opadajući po vremenskoj marki). Moguće su i složenije varijante sa boljim učinkom (manje odbijenih zahteva), ali treba paziti na moguće izgladnjivanje.

```
typedef unsigned long Time;
const Time MAXTIME = (Time)-1;

class Resource {
public:
  Resource () : holder(0), last(MAXTIME) {}
  int acquire();
  void release();
protected:
  void block (); // Implementacija ista kao i za Semaphore
  void deblock (); // Implementacija ista kao i za Semaphore
private:
  Thread* holder;
  Time last;
  ThreadQueue blocked;
};

void Resource::acquire () {
  lock();
  if (!holder) {
    holder = Thread::running;
    last = holder->timestamp;
    unlock();
    return 1;
  } else
  if (Thread::running->timestamp>=last) {
    unlock();
    return 0;
  } else {
    last = Thread::running->timestamp;
    block();
    unlock();
    return 1;
  }
}

void Resource::release () {
  lock();
  Thread* thr = blocked.first();
  if (thr) {
    holder = thr;
    deblock();
  } else {
    holder = 0; last = MAXTIME;
  }
  unlock();
}
```

1. zadatak, drugi kolokvijum, maj 2019.

Školsko jezgro proširuje se konceptom *mutex* koji predstavlja binarni semafor namenjen za međusobno isključenje pristupa kritičnim sekcijama ili deljenim resursima, ali uz mogućnost da nit može ugnežđeno „zauzimati“ (operacija

wait) i „oslobađati“ (operacija *signal*) isti *mutex* na sledeći način:

- operaciju *signal* može da pozove samo nit koja je već zauzela *mutex* operacijom *wait*; u suprotnom, operacija *signal* vraća grešku;
- nit koja je već zauzela *mutex* operacijom *wait*, može ponovo izvršiti *wait* na njemu više puta, ali bez mrtve blokade sama sa sobom (samo prvi poziv *wait* slobodnog *mutex*-a ima efekta, ostali pozivi iz iste niti nemaju efekta); ova ista nit mora pozvati operaciju *signal* isti broj puta (upareno), tako da samo (poslednji) poziv operacije *signal* koji je uparen sa prvim efektivnim pozivom *wait* ima efekat oslobađanja *mutex*-a; u slučaju neuparenog poziva *signal* vratiti grešku.

Operacije *wait* i *signal* vraćaju celobrojnu vrednost, 0 u slučaju uspeha, negativnu vrednost u slučaju greške. Prikazati implementaciju klase `Mutex`, po uzoru na klasu `Semaphore` prikazanu na predavanjima (ne treba implementirati red čekanja niti, pretpostaviti da ta klasa postoji).

Rešenje

```
class Mutex {
public:
  Mutex () : lck(0), holder(0), nestingCnt(0) {}
  int wait();
  int signal();

protected:
  void block (); // Implementacija ista kao i za Semaphore
  void deblock (); // Implementacija ista kao i za Semaphore
private:
  unsigned lck, nestingCnt;
  ThreadQueue blocked;
  Thread* holder;
};

int Mutex::wait () {
  if (holder==Thread::running) {
    nestingCnt++;
    return 0;
  }
  lock(&lck);
  if (holder) block();
  holder = Thread::running;
  nestingCnt++;
  unlock(&lck);
  return 0;
}

int Mutex::signal () {
  if (holder!=Thread::running || nestingCnt==0) return -1; // Error
  lock(&lck);
  if (--nestingCnt==0) {
    holder = 0;
    if (!blocked.isEmpty()) deblock();
  }
  unlock(&lck);
  return 0;
}
```

1. zadatak, drugi kolokvijum, april 2017.

Neki multiprocesorski operativni sistem sa preotimanjem (*preemptive*) dozvoljava preuzimanje tokom izvršavanja koda kernela, osim u kritičnim sekcijama, kada se pristupa deljenim strukturama podataka jezgra kojima mogu da pristupaju različiti procesori. Svaku takvu deljenu strukturu „štiti“ jedna celobrojna promenljiva koja služi za međusobno isključenje pristupa sa različitih procesora pomoću tehnike *spin lock*, dok se zabrana preuzimanja na datom procesoru obezbeđuje maskiranjem prekida. Međusobno isključenje pristupa kritičnoj sekciji, odnosno deljenoj strukturi, obavlja se pozivima operacija čiji je argument pokazivač na celobrojnu promenljivu koja „štiti“ datu deljenu strukturu:

```
void lock (short* lck);
void unlock (short* lck);
```

na ulazu, odnosno izlazu iz kritične sekcije. Pošto postoji potreba da kernel kod istovremeno pristupa različitim deljenim strukturama, kritične sekcije se mogu ugnježdavati. Zato demaskiranje prekida treba uraditi tek kada se izađe iz krajnje spoljašnje kritične sekcije (a ne pri svakom pozivu `unlock`).

Podaci koji su svojstveni svakom procesoru (svaki procesor ima svoju instancu ovih podataka) grupisani su u strukturu tipa `ProcessorPrivate`. (Ovu strukturu možete proširivati po potrebi.) Na raspolaganju su i sledeće funkcije ili makroi:

- `disable_interrupts()`: onemogućava spoljašnje prekide na ovom procesoru;
- `enable_interrupts()`: dozvoljava spoljašnje prekide na ovom procesoru;
- `test_and_set(short* lck)`: „obavija“ instrukciju procesora tipa test-and-set;
- `this_processor()`: vraća celobrojni identifikator tekućeg procesora (na kome se kod izvršava);
- `ProcessorPrivate processor_private[NumOfProcessors]`: niz struktura sa „privatnim“ podacima svakog procesora.

Implementirati operacije `lock` i `unlock`.

Rešenje

```
struct ProcessorPrivate {
  int lock_count; // Counts the number of locks (nested critical sections)
  ...
  ProcessorPrivate () : lock_count(0),... {...}
};

void lock (short* lck) {
  disable_interrupts(); // Multiple calls of disable_intterupt are harmless
  processor_private[this_processor()].lock_count++;
  while (test_and_set(lck));
}

void unlock (short* lck) {
  *lck=0;
  if (--processor_private[this_processor()].lock_count==0)
    enable_interrupts();
}
```

1. zadatak, drugi kolokvijum, jun 2017.

Školsko jezgro treba proširiti konceptom *barijere* („ograda“, „rampa“, engl. *barrier*) čiji je interfejs dat dole. Semantika ovog koncepta i operacija nad njim je sledeća:

- Barijera može biti u jednom od dva stanja: *otvorena* ili *zatvorena*. Barijera se inicijalizuje zadatim stanjem (argument konstruktora). Nit u čijem kontekstu se kreira barijera (tj. izvršava konstruktor) je njen *vlasnik*.
- `void Barrier::pass()`: ukoliko pozivajuća nit nije vlasnik ove barijere, ova operacija nema efekta – nit nastavlja svoje izvršavanje; ukoliko je pozivajuća nit vlasnik barijere, a barijera zatvorena, nit se suspenduje (blokira) sve dok se barijera ne otvori; ako je barijera otvorena, nit nastavlja izvršavanje bez blokade;

- `void Barrier::close()`: zatvara barijeru;
- `void Barrier::open()`: otvara barijeru i eventualno deblokira nit koja je na barijeri blokirana.

Implementirati ovu klasu `Barrier`. Ukoliko proširujete klasu `Thread`, precizno napisati kojim članovima.

```
class Barrier {
public:
  Barrier (int open=1);
  void open();
  void close();
  void pass();
};
```

Rešenje

U klasu `Thread` treba dodati sledeći član:

```
Barrier* blockedOnBarrier(0);
```

```
class Barrier {
public:
  Barrier (int open=1) : isOpen(open), owner(Thread::running) {}
  void open();
  void close();
  void pass();
private:
  int isOpen;
  Thread* owner;
};

void Barrier::pass () {
  if (this->owner!=Thread::running) return;
  lock();
  if (!this->isOpen) {
    Thread::running->blockedOnBarrier = this;
    if (setjmp(Thread::running->context)==0) {
       Thread::running = Scheduler->get();
       longjmp(Thread::running->context,1);
    }
  }
  unlock();
}

void Barrier::open () {
  lock();
  this->isOpen = 1;
  if (this->owner->blockedOnBarrier==this) {
    this->owner->blockedOnBarrier = 0;
    Scheduler::put(this->owner);
  }
  unlock();
}

inline void Barrier::close () {
  this->isOpen = 0;
}
```

1. zadatak, drugi kolokvijum, maj 2016.

Školsko jezgro proširuje se konceptom *mutex* koji predstavlja binarni semafor, poput događaja (*event*), sa istom semantikom operacija *wait* i *signal* kao kod događaja, ali sa sledećim dodatnim ograničenjima koja su u skladu sa namenom upotrebe samo za međusobno isključenje kritične sekcije:

- inicijalna vrednost je uvek 1;
- operaciju *signal* može da pozove samo nit koja je zatvorila ulaz u kritičnu sekciju, odnosno koja je pozvala operaciju *wait*; u suprotnom, operacija *signal* vraća grešku;
- nit koja je već zatvorila *mutex* operacijom *wait*, ne može ponovo izvršiti *wait* na njemu.

Operacije *wait* i *signal* vraćaju celobrojnu vrednost, 0 u slučaju uspeha, negativnu vrednost u slučaju greške. Prikazati implementaciju klase `Mutex`, po uzoru na klasu `Semaphore` prikazanu na predavanjima (ne treba implementirati red čekanja niti, pretpostaviti da ta klasa postoji).

Rešenje

```
class Mutex {
public:
  Mutex () : val(1), lck(0), holder(0) {}
  int wait();
  int signal();

protected:
  void block (); // Implementacija ista kao i za Semaphore
  void deblock (); // Implementacija ista kao i za Semaphore
private:
  int val, lck;
  ThreadQueue blocked;
  Thread* holder;
};

int Mutex::wait () {
  if (holder==Thread::running) return -1; // Error
  lock(&lck);
  if (--val<0) block();
  holder = Thread::running;
  unlock(&lck);
  return 0;
}

int Mutex::signal () {
  if (holder!=Thread::running) return -1; // Error
  lock(&lck);
  holder = 0;
  if (val<0) {
    val++;
    deblock();
  } else // No blocked threads, val>=0
    val = 1;
  unlock(&lck);
  return 0;
}
```

2. zadatak, prvi kolokvijum, septembar 2016.

Školsko jezgro proširuje se konceptom tzv. *tačke spajanja* ili *susreta* (engl. *join*, *rendez-vous*), kao jednostavne sinhronizacione primitive sa sledećim značenjem.

Program može kreirati objekat klase `Join`, čiji je interfejs dat dole, zadajući mu kao parametre konstruktora (pokazivače na) dve niti koje će se sinhronizovati (susretati, „spajati“) na ovom objektu. Iz konteksta ove dve niti (i samo njih) može se pozvati operacija `wait` ovog objekta. Tom operacijom se ove dve niti „spajaju“ (sinhronizuju, susreću), odnosno međusobno sačekuju, tako da ni jedna od njih neće nastaviti svoje izvšavanje dalje ako ona druga nije stigla do iste te tačke spajanja (tj. pozvala ovu operaciju `wait` na istom ovom objektu klase `Join`). Drugim rečima, niti nastavljaju izvršavanje iza poziva `wait` samo kada su obe stigle do te tačke. Dole je data implementacija sistemskog poziva `dispatch()` u školskom jezgru, kao primer kako su implementirana sva mesta promene konteksta. Implementirati u celini klasu `Join`, bez ikakve izmene ili dopune klase `Thread`. Operacija `wait` treba da vrati sledeće:

- -1 u slučaju da je ovu operaciju pozvala neka druga nit, osim one dve koje su definisane konstruktorom (ili bilo kakve druge greške);
- 0 u slučaju da je pozivajuća nit prva stigla do tačke spajanja, a 1 u suprotnom.

```
void dispatch () {
  lock();
  jmp_buf old = Thread::running->context;
  Scheduler::put(Thread::running);
  Thread::running = Scheduler::get();
  jmp_buf new = Thread::running->context;
  if (setjmp(old)==0) longjmp(new,1);
  unlock();
}

class Join {
public:
  Join (Thread* thread1, Thread* thread2);
  int wait ();
};
```

Rešenje

```
class Join {
public:
  Join (Thread* thread1, Thread* thread2);
  int wait ();
private:
  Thread* thread[2];
  int arrived[2];
};

Join::Join (Thread* t1, Thread* t2) {
  this->thread[0] = t1; this->thread[1] = t2;
  this->arrived[0] = this->arrived[1] = 0;
}

int Join::wait () {
  if (Thread::running!=this->thread[0] && Thread::running!=this->thread[1])
    return -1; // Exception
  lock();
  int ret;
  int myself = (Thread::running==this->thread[0])?0:1;
  int other = 1-myself;
  this->arrived[myself] = 1;
  if (this->arrived[other]) {
    ret = 1;
    Scheduler::put(this->thread[other]);
    this->arrived[0] = this->arrived[1] = 0;
  } else {
```

```
    ret = 0;
    jmp_buf old = Thread::running->context;
    Thread::running = Scheduler::get();
    jmp_buf new = Thread::running->context;
    if (setjmp(old)==0) longjmp(new,1);
  }
  unlock();
  return ret;
}
```

1. zadatak, drugi kolokvijum, septembar 2015.

U školskom jezgru modifikuje se (uopštava) koncept brojačkog semafora podržan klasom `Semaphore` na sledeći način. Operacija `wait()` prima jedan nenegativan celobrojni argument n sa podrazumevanom vrednošću 1 i sa sledećim značenjem. Trenutna nenegativna vrednost semafora v predstavlja broj raspoloživih „žetona“. Operacijom *wait* pozivaju ća nit „traži“ n „žetona“ (kod standardnog brojačkog semafora n je uvek podrazumevano 1). Ako je trenutna vrednost v semafora veća ili jednaka argumentu n operacije *wait*, ta vrednost v će biti umanjena za n, a nit će nastaviti izvršavanje bez blokade, jer je „dobila“ svih traženih n „žetona“. U suprotnom, ova nit će „uzeti“ v „žetona“, i čekaće blokirana na semaforu dok se operacijama *signal* ne pojavi još n-v žetona; kada se to dogodi, nit može nastaviti sa izvršavanjem (jer je dobila svih traženih n „žetona“). Operacija *signal* „obezbeđuje“ uvek jedan „žeton“, kao i kod standardnog brojačkog semafora.

```
class Semaphore {
public:
  Semaphore (unsigned int init=1);
  void wait(unsigned int n=1);
  void signal();
  int val ();
};
```

Za podršku ovoj implementaciji, u klasi `Thread` postoji nenegativno celobrojno polje `waiting` koje pokazuje na koliko još preostalih „žetona“ čeka data nit, ukoliko je blokirana na nekom semaforu. Osim toga, klasa `Queue` kojom se implementira FIFO red čekanja na semaforu ima operaciju `first()` koja vraća prvu nit u tom redu, ako je ima (0 ako je red prazan), bez izbacivanja te niti iz reda. Pomoćne operacije `block()` i `deblock()` klase `Semaphore` su iste kao i u postojećoj implementaciji školskog jezgra.

Dati izmenjenu implementaciju operacija `wait()` i `signal()`.

Rešenje

```
void Semaphore::wait (unsigned int n = 1) {
  lock(lck);
  val -= n;
  if (val<0) {
    Thread::running->waiting = -val;
    val = 0;
    block();
  }
  unlock(lck);
}

void Semaphore::signal () {
  lock(lck);
  Thread* thr = blocked.first();
  if (thr) {  // There are blocked threads, val==0
    thr->waiting--;
    if (thr->waiting==0)
      deblock();
```

```
  } else // No blocked threads, val>=0
    val++;
  unlock(lck);
}
```

1. zadatak, drugi kolokvijum, april 2014.

Modifikovati operaciju `wait` klase `Semaphore` u školskom jezgru tako da ima izmenjenu deklaraciju datu dole sa sledećim dodatnim mogućnostima i izmenjenim ponašanjem:

- ukoliko je vrednost argumenta `toBlock` različita od 0, operacija se ponaša na standardan način i vraća 1 ako se pozivajuća nit blokirala (suspendovala), a 0 ako nije;
- ukoliko je vrednost argumenta `toBlock` jednaka 0, ako je vrednost semafora nula (ili manja od 0), pozivajuća nit se neće blokirati, vrednost semafora se neće promeniti, a operacija će odmah vratiti -1; inače, ukoliko je vrednost semafora veća od nule, operacija se ponaša na standardan način (i vraća 0 pošto se nit nije blokirala).

```
int Semaphore::wait (int toBlock);
```

Rešenje

```
int Semaphore::wait (int toBlock) {
  lock(lck);
  int ret = 0;
  if (!toBlock && val<=0)
    ret = -1;
  else
    if (--val<0) {
      ret = 1;
      block();
    }
  unlock(lck);
  return ret;
}
```

1. zadatak, drugi kolokvijum, septembar 2014.

U nekom operativnom sistemu postoje samo standardni brojački semafori podržani klasom `Semaphore` čiji je interfejs isti kao u školskom jezgru:

```
class Semaphore {
public:
  Semaphore (int init=1);
  void wait();
  void signal();
  int val ();
};
```

Pomoću ovih semafora implementirati koncept događaja (binarnog semafora), podržanog klasom `Event` sa sledećim interfejsom:

```
class Event {
public:
  Event ();
  void wait();
  void signal();
  int val ();
};
```

Rešenje

```
class Event {
public:
  Event () : mySem(0), mutex(1) {}
  void wait() { mySem.wait(); }
  void signal();
  int val () { return mySem.val(); }
private:
  Semaphore mySem, mutex;
};

inline void Event::signal () {
  mutex.wait();
  if (mySem.val()<1) mySem.signal();
  mutex.signal();
}
```

2. zadatak, drugi kolokvijum, april 2013.

Data je sledeća implementacija operacije `Semaphore::lock()` koja obezbeđuje zaključavanje semafora (međusobno isključenje operacija na semaforu) u kodu školskog jezgra za višeprocesorski sistem. Operacija `swap()` implementirana je pomoću odgovarajuće atomične instrukcije procesora. Objasniti zašto ova implementacija operacije `lock()` nije dobra, a onda je popraviti.

```
extern void swap (int*, int*);

void Semaphore::lock() {
  int zero = 0;
  mask_interrupts();
  while (!this->lck) {}
  swap(&zero,&(this->lck));
}
```

Rešenje

Data implementacija nije dobra jer je moguće utrkivanje (*race condition*) između izvršavanja istog ovog koda za isti semafor na dva (ili više) procesora. I jedan i drugi procesor mogu da izvrše petlju `while` i iz nje izađu, uporedno pročitavši iz atributa `lck` vrednost 1, a potom izvrše `swap` i izađu iz operacije `lock`, odnosno uđu u operaciju semafora. Ispravljena verzija je sledeća (ima i nešto malo efikasnijih, kako je prikazano na predavanjima):

```
void Semaphore::lock() {
    int zero = 0;
    mask_interrupts();
    while (!zero) swap(&zero, &(this->lck));
}
```

2. zadatak, drugi kolokvijum, septembar 2013.

U nekom operativnom sistemu svi sistemski pozivi izvršavaju se kao softverski prekid koji skače na prekidnu rutinu označenu kao `sys_call`, dok se sama identifikacija sistemskog poziva i njegovi parametri prenose kroz registre procesora. Jezgro tog operativnog sistema je višenitno – poseduje više internih kernel niti koje obavljaju različite poslove: izvršavaju uporedne I/O operacije, vrše druge interne poslove jezgra itd. Prilikom obrade sistemskog poziva `sys_call`, prema tome, prekidna rutina samo oduzima procesor tekućem korisničkom procesu uz čuvanje njegovog konteksta i dodeljuje procesor tekućoj kernel niti.

Jedna od tih niti zadužena je za samu obradu zahteva koju je korisnički proces postavio u sistemskom pozivu. Ona preuzima parametre poziva iz sačuvanih registara i, na osnovu vrste sistemskog poziva, poziva odgovarajuću proceduru kernela za taj sistemski poziv.

Ovde se posmatraju sistemski pozivi za operacije na binarnim semaforima (događajima). Unutar kernela, binarni semafori su realizovani klasom `Event`, poput klase `Semaphore` date u školskom jezgru. Realizovati operacije `wait` i `signal` ove klase, koje poziva gore pomenuta kernel nit kada obrađuje te sistemske pozive. Na raspolaganju su operacije `lock()` i `unlock()` koje obezbeđuju međusobno isključenje unutar jezgra, kao i klasa `Scheduler` koja implementira raspoređivanje, kao u školskom jezgru. Na korisnički proces koji je izvršio sistemski poziv ukazuje pokazivač `runningUserThread` unutar kernela.

Rešenje

```
void Event::wait () {
  lock(lck);
  if (--val<0) {
    blocked.put(runningUserThread);
    runningUserThread = Scheduler::get();
  }
  unlock(lck);
}

void Event::signal () {
  lock(lck);
  if (val++<0)
    Scheduler::put(blocked.get());
  else val = 1;
  unlock(lck);
}
```

2. zadatak, drugi kolokvijum, septembar 2012.

U nekom višeprocesorskom sistemu ne postoji stanje suspendovanih (blokiranih) niti, već su sve aktivne niti uvek spremne, dok se operacije čekanja na semaforu i drugim sinhronizacionim primitivama realizuju uposlenim čekanjem. Jezgro sistema povremeno (na prekid od tajmera) jednostavno preotima neki procesor od tekuće niti i predaje ga nekoj drugoj aktivnoj niti. U sistemu su implementirane operacije

```
void lock (int lck);
void unlock (int lck);
```

koje realizuju međusobno isključenje nad deljenom strukturom podataka zaštićenom celobrojnim ključem `lck` maskiranjem prekida i mehanizmom *spin-lock* za višeprocesorski pristup.

Realizovati klasu `Semaphore` koja apstrahuje standardni brojački semafor, sa uposlenim čekanjem.

Rešenje

```
class Semaphore {
public:
  Semaphore (int init=1) : v(init), lck(0) {}
  void wait ();
  void signal ();
private:
  int v, lck;
};

void Semaphore::wait () {
  int done = 0;
  while (!done) {
    lock(lck);
    if (v>0) v--, done=1;
    unlock(lck);
  }
```

```
}

void Semaphore::signal () {
  lock(lck);
  v++;
  unlock(lck);
}
```

2. zadatak, drugi kolokvijum, maj 2011.

U klasi `Semaphore` postoji privatni podatak član `isLocked` tipa `int` koji služi da obezbedi međusobno isključenje pristupa strukturi semafora u višeprocesorskom operativnom sistemu. Ako je njegova vrednost 1, kod koji se izvršava na nekom procesoru je zaključao semafor za svoj isključivi pristup; ako je vrednost 0, pristup semaforu je slobodan. Korišćenjem operacije `test_and_set()` koja je implementirana korišćenjem odgovarajuće atomične instrukcije procesora, realozovati operaciju `Semaphore::lock()` koja treba da obezbedi međusobno isključenje pristupa strukturi semafora u višeprocesorskom operativnom sistemu, ali tako da bude efikasnija od dole date implementacije tako što ne poziva operaciju `test_and_set` ako je semafor već zaključan od strane drugog procesora, pošto ta operacija ima veće režijske troškove na magistrali računara.

```
void Semaphore::lock () {
  while (test_and_set(this->isLocked));
}
```

Rešenje

```
void Semaphore::lock () {
  for (int acquired = 0; !acquired;) {
    while (this->isLocked);
    acquired = !test_and_set(this->isLocked);
  }
}
```

2. zadatak, drugi kolokvijum, septembar 2011.

Izmeniti datu implementaciju operacije *wait* na semaforu u školskom jezru tako da, pre nego što odmah blokira pozivajuću nit ukoliko je semafor zatvoren, najpre pokuša da uposleno sačeka, ali ograničeno, ponavljajući petlju čekanja najviše `SemWaitLimit` puta. Ostatak klase `Semaphore` se ne menja.

```
void Semaphore::wait () {
  lock(lck);
  if (--val<0)
    block();
  unlock(lck);
}
```

Rešenje

```
void Semaphore::wait () {
  for (int i=SemWaitLimit; val<=0 && i>0; i--);
  lock(lck);
  if (--val<0)
    block();
  unlock(lck);
}
```

1. zadatak, drugi kolokvijum, maj 2010.

U nekom operativnom sistemu svi sistemski pozivi izvršavaju se kao softverski prekid koji skače na prekidnu rutinu označenu kao `sys_call`, dok se sama identifikacija sistemskog poziva i njegovi parametri prenose kroz registre

procesora. Jezgro tog operativnog sistema je višenitno – poseduje više internih kernel niti koje obavljaju različite poslove: izvršavaju uporedne I/O operacije, vrše druge interne poslove jezgra itd. Prema tome, sve potrebne radnje prilikom promene konteksta korisničkih procesa (osim samog čuvanja i restauracije konteksta procesora), kao što su smeštanje PCB korisničkog procesa koji je do tada bio tekući u odgovarajući red (spremnih ili blokiranih, u zavisnosti od situacije), izbor novog tekućeg procesa iz skupa spremnih, promena memorijskog konteksta, obrada samog konkretnog sistemskog poziva, itd. obavljaju se u kontekstu internih kernel niti. Prilikom obrade sistemskog poziva u prekidnoj rutini `sys_call`, prema tome, samo se oduzima procesor tekućem korisničkom procesu i dodeljuje se tekućoj kernel niti.

Na PCB tekućeg korisničkog procesa ukazuje globalni pokazivač `runningUserProcess`, a na PCB tekuće interne niti jezgra ukazuje globalni pokazivač `runningKernelThread`. I interne niti jezgra vrše promenu konteksta između sebe na isti način, pozivom softverskog prekida koji ima istu internu strukturu kao i rutina `sys_call`.

Posebna interna kernel nit zadužena je za obradu sistemskog poziva izdatog od strane korisničkog procesa. Ova nit najpre određuje o kom sistemskom pozivu se radi i preuzima njegove parametre, a onda poziva odgovarajuće operacije jezgra koje obavljaju zahtevanu sistemsku uslugu, recimo operacije `wait` ili `signal` na semaforu.

Po uzoru na implementaciju klase `Semaphore` u školskom jezgru, implementirati ovu klasu za potrebe opisanog jezgra. Međusobno isključenje koda operacija `wait` i `signal` obezbeđuju uobičajene operacije `lock` i `unlock`. Operacije `lock` i `unlock` su implementirane (ne treba ih implementirati).

Rešenje

```
void Semaphore::wait () {
  lock(lck);
  if (--val<0) {
    blocked.put(runningUserProcess);
    runningUserProcess = UserProcessScheduler::get();
  }
  unlock(lck);
}

void Semaphore::signal () {
  lock(lck);
  if (val++<0)
    UserProcessScheduler::put(blocked.get());
  unlock(lck);
}
```

Pomoćne operacije `block()` i `unblock()` više nisu potrebne (izbacuju se). Ostatak definicije klase `Semaphore` ostaje isti.

1. zadatak, drugi kolokvijum, maj 2009.

Realizaciju semafora pomoću C++ klase `Semaphore` date na predavanjima modifikovati tako da se promena konteksta može dogoditi i kod neblokirajućih poziva `wait` i `signal` na semaforu (tj. pri svakom pozivu `signal`, kao i pri pozivu `wait` i kada ne treba blokirati pozivajući proces).

Rešenje

```
void Semaphore::wait () {
  lock();
  if (setjmp(Thread::runningThread->context)==0) {
    if (--val<0)
      blocked.put(Thread::runningThread);
    else
      Scheduler::put(Thread::runningThread);
    Thread::runningThread = Scheduler::get();
    longjmp(Thread::runningThread->context,1);
```

```
  }
  unlock();
}

void Semaphore::signal () {
  lock();
  if (val++<0)
    Scheduler::put(blocked.get());
  if (setjmp(Thread::runningThread->context)==0) {
    Scheduler::put(Thread::runningThread);
    Thread::runningThread = Scheduler::get();
    longjmp(Thread::runningThread->context,1);
  }
  unlock();
}
```

Pomoćne operacije `block()` i `unblock()` više nisu potrebne (izbacuju se). Ostatak definicije klase `Semaphore` ostaje isti.

1. zadatak, drugi kolokvijum, maj 2008.

U jezgru nekog multiprocesorskog operativnog sistema realizovane su sledeće operacije koje se koriste za obezbeđenje atomičnosti operacija nad semaforom:

```
void Semaphore::lock (int* lck) {
  disable_interrupts();
  while test_and_set(lck);
}

void Semaphore::unlock (int* lck) {
  *lck=0;
  enable_interrupts();
}
```

Koncept semafora realizovan je klasom `Semaphore`, čiji su fragmenti prikazani u nastavku:

```
class Semaphore {
...
private:
  static int commonLock;  // initialized to 0
  int myLock;  // initialized to 0
...
};
```

Komentarisati razliku između sledeće dve realizacije operacija nad semaforom, odnosno njihovog ulaznog i izlaznog protokola sa ciljem obezbeđenja atomičnosti:

```
lock(&commonLock);          lock(&myLock);
...                         ...
unlock(&commonLock);        unlock(&myLock);
```

Posebno komentarisati sledeće aspekte:

- da li su obe varijante sasvim korektne i pod kojim uslovima?
- stepen paralelizma koji ove varijante omogućavaju.

Rešenje

Prva varijanta zaključava semafor (kao deljeni resurs) maskiranjem prekida i uposlenim čekanjem (operacijom tipa *test and set*) nad jednom zajedničkom varijablom (`commonLock`) koja se koristi za sve semafore (instance klase `Semaphore`), dok druga varijanta radi to isto samo korišćenjem posebne varijable za svaki semafor (`myLock`). Prva

varijanta ne dozvoljava ulaz u kod operacije nad semaforom nekom procesoru ako je bilo koji drugi procesor ušao u kod operacije nad bilo kojim drugim semaforom, dok su kod druge varijante moguća paralelna izvršavanja koda operacija nad različitim semaforima. Odatle slede zaključci:

- Ukoliko kod operacija nad semaforom ne uzrokuje konflikte na deljenim strukturama koje se koriste u izvršavanjima na različitim procesorima, onda su obe varijante korektne. U suprotnom, ukoliko ovaj kod uzrokuje ovakve konflikte, npr. korišćenjem istog reda spremnih procesa za različite procesore bez obezbeđenja međusobnog isključenja nad tom strukturom, onda je samo prva varijanta korektna (pod uslovom da ne pravi konflikte sa drugim uslugama operativnog sistema), a druge ne.
- Druga varijanta omogućava znatno veći stepen paralelizma, jer ne zaustavlja napredovanje operacije nad jednim semaforom na jednom procesoru zbog toga što je drugi procesor ušao u operaciju nad drugim semaforom, kako to čini prva varijanta.

1. zadatak, drugi kolokvijum, maj 2007.

Na raspolaganju su sledeće funkcije:

- `swap(int*,int*)` koja, korišćenjem odgovarajuće mašinske instrukcije, atomično zamenjuje vrednosti dve memorijske reči na adresama zadatim argumentima (C/C++ tip int je uvek veličine mašinske reči).
- `set_interrupts(int)` koja maskira sve nemaskirajuće prekide ako je argument jednak 0, odnosno demaskira ih ako je argument jednak 1.

Korišćenjem ovih funkcija realizovati standardni brojački semafor (klasu `Semaphore`) sa operacijama `wait()` i `signal()`, pri čemu se koristi uposleno čekanje umesto suspenzije (blokiranja) pozivajućeg procesa. Realizacija treba da bude prilagođena multiprocesorskom sistemu sa preuzimanjem (*preemptive*).

Rešenje

```
class Semaphore {
public:
  Semaphore (unsigned int value=1);
  void wait ();
  void signal();
protected:
  inline void lock();    // inline nije semantički bitan
  inline void unlock();  // inline nije semantički bitan
private:
  int lock;
  unsigned int val;
};

Semaphore::Semaphore (unsigned int v) : lock(0), val(v) {}

void Semaphore::lock () {
  set_interrupts(0);
  for (int lck=1; lck;) swap(&lck,&lock);
}

void Semaphore::unlock () {
  lock=0;
  set_interrupts(1);
}

void Semaphore::wait () {
  for (int done==0; !done; ) {
    while(val==0);
    lock();
      if (val==0) { unlock(); continue; }
```

```
      done=1;
      val--;
    unlock();
  }
}

void Semaphore::signal () {
  lock();
  val++;
  unlock();
}
```

Sinhronizacija procesa (interfejs)

3. zadatak, drugi kolokvijum, jun 2022.

Više uporednih niti-pisaca upisuje izračunate celobrojne dvodimenzionalne koordinate (x, y) na koje treba pomeriti robota u deljeni objekat klase `SharedCoord` čiji je interfejs dat dole; svaka ovakva nit nezavisno upisuje svoj par izračunatih koordinata pozivom operacije `write` ove klase. Jedna nit-čitalac periodično očitava par koordinata iz tog deljenog objekta i pomera robota na očitane koordinate; ova nit to radi pozivom operacije `read` ove klase, nezavisno od pisaca, svojim tempom, tako da svaki put čita poslednje upisane koordinate (može pročitati više puta isti par koordinata ili neke izračunate koordinate i preskočiti).

Implementirati klasu `SharedCoord` uz neophodnu sinhronizaciju korišćenjem najmanjeg broja semafora školskog jezgra.

```
class SharedCoord {
public:
    SharedCoord();
    void read(int& x, int& y);
    void write(int x, int y);
};
```

Rešenje

```
class SharedCoord {
public:
    SharedCoord();
    void read(int& x, int& y);
    void write(int x, int y);
private:
    Semaphore mutex;
    int x, y;
};
inline SharedCoord() : mutex(1) {}
inline void SharedCoord::read(int& x_, int& y_) {
    mutex.wait();
    x_ = this->x;
    y_ = this->y;
    mutex.signal();
}
inline void SharedCoord::write(int x_, int y_) {
    mutex.wait();
    this->x = x_;
    this->y = y_;
    mutex.signal();
}
```

2. zadatak, kolokvijum, jul 2021.

Korišćenjem sistemskog poziva *fork* i POSIX semafora za sinhronizaciju napisati program koji, kad se nad njim pokrene proces, pokrene jedno svoje dete, a potom ta dva procesa strogo naizmenično na svoj standardni izlaz ispisuju po jedno „A“ (proces-roditelj), odnosno „B“ (proces-dete) tačno po 10 puta, nakon čega se završavaju. Za sistemski poziv *fork* pretpostavlja se sledeće:

- proces-dete nasleđuje standardni izlaz od roditelja, a operativni sistem obezbeđuje međusobno isključenje ispisa na isti standardni izlaz;
- proces-dete nasleđuje deskriptore semafora koje je proces-roditelj otvorio, tako da isti deskriptori ukazuju na isti semafor u sistemu koji je deljen između roditelja i deteta; oba procesa moraju da zatvore semafor kada im više nije potreban.

Rešenje

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <unistd.h> // for fork()
#include <stdio.h> // for printf

sem_t *sem1, *sem2;

int main() {
    sem1 = sem_open("/sem1", O_CREAT | O_EXCL, O_RDWR, 1);
    if (sem1 == NULL) return -1;
    sem2 = sem_open("/sem2", O_CREAT | O_EXCL, O_RDWR, 0);
    if (sem2 == NULL) {
        sem_close(sem1);
        return -1;
    }
    pid_t pid = fork();
    if (pid < 0) return -1;
    for (int i = 0; i < 10; i++) {
        if (pid > 0) {
            if (sem_wait(sem1) < 0) break;
            printf("A");
            fflush(stdout);
            if (sem_post(sem2) < 0) break;
        } else {
            if (sem_wait(sem2) < 0) break;
            printf("B");
            fflush(stdout);
            if (sem_post(sem1) < 0) break;
        }
    }
    sem_close(sem1);
    sem_unlink("/sem1");
    sem_close(sem2);
    sem_unlink("/sem2");
    return 0;
}
```

2. zadatak, kolokvijum, jun 2020.

Korišćenjem školskog jezgra i date klase `RowAdder`, implementirati funkciju `mat_add` koja sabira elemente svake vrste matrice `mat` i rezultat tog sabiranja smešta u odgovarajući element niza `res`, i to radi uporedo tako da kontrolu vraća pozivaocu tek kada su uporedna sabiranja svih vrsta matrice završena.

```
class RowAdder : public Thread {
public:
  RowAdder (int arr[], int cnt, int* res, Semaphore* sem)
    : a(arr), n(cnt), r(res), s(sem) {}
protected:
  virtual void run ();
private:
  int *a, n, *r;
  Semaphore* s;
};

void RowAdder::run () {
```

```
  *r = 0;
  for (int i=0; i<n; i++) *r += a[i];
  if (s) s->signal();
}

const int M = ..., N = ...;
int mat[M][N];
int res[M];

void mat_add ();
```

Rešenje

```
void mat_add () {
  Thread* thr[M];
  Semaphore* sem[M];

  for (int i=0; i<M; i++) {
    sem[i] = new Semaphore(0);
    thr[i] = new RowAdder(mat[i],N,&res[i],sem[i]);
    thr[i]->start();
  }

  for (int i=0; i<M; i++) {
    sem[i]->wait();
    delete sem[i];
    delete thr[i];
  }
}
```

1. zadatak, drugi kolokvijum, jun 2019.

U nekom sistemu postoje sledeći sistemski pozivi:

- `sem_t* sem_create(unsigned init_value)`: kreira semafor sa zadatom inicijalnom vrednošću;
- `int sem_wait(sem_t* semaphore)`

  `int sem_signal(sem_t* semaphore)`: standardne operacije *wait* i *signal* na zadatom semaforu;
- `int thread_create(void(*)(void*),void*)`: kreira nit nad funkcijom na koju ukazuje prvi argument; ta funkcija prima jedan argument tipa `void*` i ne vraća rezultat; novokreirana nit poziva tu funkciju sa stvarnim argumentom jednakim drugom argumentu ovog sistemskog poziva; sistemski poziv vraća PID kreirane niti.

Sve funkcije vraćaju negativnu vrednost ili *null* pokazivač u slučaju greške. Korišćenjem ovih sistemskih poziva, uz pomoću semafora za sinhronizaciju, napisati program koji kreira N niti koje će pozivati funkciju `process` bez argumenata jedna nakon druge, strogo u poretku numeracije tih niti (nit i+1 treba da pozove funkciju `process` kada nit i završi sa tim pozivom). Ignorisati sve greške.

Rešenje

```
const int N = ...;
struct sem_t;
sem_t* semaphores[N];

void initSems () {
  semaphores[0] = sem_create(1);
  for (int i=1; i<N; i++)
    semaphores[i] = sem_create(0);
}
```

```
void threadBody (void* ps) {
  sem_t** psem = (sem_t**)ps;
  sem_wait(*psem++);
  process();
  if (psem < semaphores+N)
    sem_signal(*psem);
}

int main () {
  initSems();
  for (int i=0; i<N; i++)
    thread_create(threadBody,semaphores+i);
}
```

1. zadatak, drugi kolokvijum, april 2018.

U nastavku je dat parcijalni opis neznatno izmenjenog (pojednostavljenog) POSIX API.

```
#include <fcntl.h>
/* For O_* constants */
#include <sys/stat.h>
sem_t *sem_open(const char *name, int oflag, unsigned int value);
#include <semaphore.h>
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_close(sem_t *sem);
```

`sem_open()` creates a new POSIX semaphore or opens an existing semaphore. The semaphore is identified by name of the form `/somename`; that is, a null-terminated string of up to `NAME_MAX-4` (i.e., 251) characters consisting of an initial slash, followed by one or more characters, none of which are slashes. Two processes can operate on the same named semaphore by passing the same name to `sem_open()`.

The `oflag` argument specifies flags that control the operation of the call. (Definitions of the flags values can be obtained by including `<fcntl.h>`.) If `O_CREAT` is specified in `oflag`, then the semaphore is created if it does not already exist. If both `O_CREAT` and `O_EXCL` are specified in `oflag`, then an error is returned if a semaphore with the given name already exists. If `O_CREAT` is specified in `oflag`, then the value argument must be specified. The value argument specifies the initial value for the new semaphore.

If `O_CREAT` is specified, and a semaphore with the given name already exists, then value is ignored. After the semaphore has been opened, it can be operated on using `sem_post()` and `sem_wait()`. When a process has finished using the semaphore, it can use `sem_close()` to close the semaphore.

On success, `sem_open()` returns the address of the new semaphore; this address is used when calling other semaphore-related functions.

`sem_post()` increments (unlocks) the semaphore pointed to by `sem`. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a `sem_wait()` call will be woken up and proceed to lock the semaphore.

`sem_wait()` decrements (locks) the semaphore pointed to by `sem`. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero).

```
#include <fcntl.h>
pid_t getpid(void);
```

`getpid()` shall return the process ID of the calling process.

Korišćenjem ovih sistemskih poziva, implementirati apstrakciju `Mutex` koja predstavlja sinhronizacionu primitivu za međusobno isključenje, koja se može koristiti samo u te svrhe, tj. uz ograničenje da samo proces koji je zaključao

kritičnu sekciju može da je otključa: ukoliko to nije zadovoljeno, operacije `exit` treba da vrati grešku (negativnu vrednost). Ova apstrakcija treba da ima sledeći objektno orijentisani interfejs:

```
class Mutex {
public:
  Mutex (const char *name);
  int entry ();
  int exit();
};
```

Rešenje

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

class Mutex {
public:
  Mutex (const char *name);
 ~Mutex ();
  int entry ();
  int exit();
private:
  sem_t* mutex;
  pid_t pid;
};

Mutex::Mutex (const char *name) {
  this->mutex = sem_open(name,O_CREAT,1);
}

int Mutex::entry () {
  if (!this->mutex) return -1;
  int ret = sem_wait(this->mutex);
  if (ret<0) return ret;
  this->pid = getpid(); // Must not be done before sem_wait
  return 0;
}

int Mutex::exit () {
  if (!this->mutex || this->pid!=getpid()) return -1; // error
  return sem_post(this->mutex);
}

Mutex::~Mutex () {
  sem_close(this->mutex);
}
```

1. zadatak, drugi kolokvijum, jun 2018.

U klasu `Semaphore` u školskom jezgru dodata je statička funkcija članica

```
int Semaphore::waitOr(Semaphore*,Semaphore*);
```

koja čeka na dva semafora po *ili* uslovu, tj. dok bilo koji od dva semafora ne postane veći od 0. Ukoliko je taj uslov ispunjen, ova operacija smanjuje samo jedan od njih za 1 (nikada oba, čak i ako su oba veća od 0), i vraća 1 ili 2, ukazujući na to koji od njih je umanjila za 1 (prvi ili drugi).

Potrebno je implementirati ograničeni bafer čiji je interfejs dat dole. Ovom baferu dostavljaju se uporedo, od strane

dve vrste proizvođača, elementi tipa `int` operacijom `put1`, odnosno elementi tipa `double` operacijom `put2`. Ove elemente treba smeštati u dva različita interna bafera (za svaki tip elementa po jedan). Sa druge strane, potrošač operacijom `get` treba da dobije bilo koji od dve vrste elemenata, koji god je na raspolaganju. Ova operacija `s` treba da postavi uzetu vrednost u jedan od dva objekta na koji ukazuju argumenti, i da vrati vrednost 1 ili 2, u zavisnosti koji od ta dva objekta je postavio vrednost.

```
class BoundedBuffer {
public:
  BoundedBuffer ();

  void put1 (int);
  void put2 (double);
  int get (int*, double*);
};
```

Rešenje

```
const int N = ...;  // Capacity of the buffer

class BoundedBuffer {
public:

  BoundedBuffer ();

  void put1 (int);
  void put2 (double);
  int get (int*, double*);

private:
  Semaphore mutex;
  Semaphore spaceAvailable1, itemAvailable1;
  Semaphore spaceAvailable2, itemAvailable2;

  int buffer1[N];
  int head1, tail1;

  double buffer2[N];
  int head2, tail2;
};

BoundedBuffer::BoundedBuffer () :
  mutex(1),
  spaceAvailable1(N), itemAvailable1(0),
  head1(0), tail1(0),
  spaceAvailable2(N), itemAvailable2(0),
  head2(0), tail2(0) {}

void BoundedBuffer::put1 (int d) {
  spaceAvailable1.wait();
  mutex.wait();
  buffer1[tail1] = d;
  tail1 = (tail1+1)%N;
  mutex.signal();
  itemAvailable1.signal();
}

// Similar for put2
```

```
int BoundedBuffer::get (int* pi, double* pd) {
  int s = Semaphore::waitOr(&itemAvailable1,&itemAvailable2);
  mutex.wait();
  if (s==1) {
    *pi = buffer1[head1];
    head1 = (head1+1)%N;
    mutex.signal();
    spaceAvailable1.signal();
  } else {
    *pd = buffer2[head2];
    head2 = (head2+1)%N;
    mutex.signal();
    spaceAvailable2.signal();
  }
  return s;
}
```

1. zadatak, treći kolokvijum, jun 2016.

U nekom sistemu izvršena je adaptacija interfejsa znakovno orijentisanog sekvencijalnog izlaznog uređaja na blokovski orijentisani sekvencijalni uređaj pomoću ograničenog bafera. Proizvođači, koji su uporedne niti, upisuju u bafer sekvencijalno bajt po bajt, operacijom `put`. Potrošači, takođe uporedne niti, uzimaju po ceo blok podataka veličine `BlockSize` iz bafera operacijom `read` i prenose ih dalje na izlazni uređaj.

Implementirati klasu `Buffer` čiji je interfejs dat dole. Sinhronizaciju vršiti pomoću semafora školskog jezgra, koji su, radi pogodnosti upotrebe za ovu namenu, prošireni operacijom `signal(unsigned)` koja atomično inkrementira vrednost semafora za vrednost zadatog argumenta (koji može biti proizvoljan prirodan broj, a ne samo jedan, kako je podrazumevano).

```
typedef unsigned short Byte;
const int BlockSize = ...;
const int NumOfBlocks = ...;
const int BufferSize = NumOfBlocks*BlockSize;

class Buffer {
public:
  Buffer ();
  void put (Byte b);
  void read (Byte block[]);
};
```

Rešenje

```
typedef unsigned short Byte;
const int BlockSize = ...;
const int NumOfBlocks = ...;
const int BufferSize = NumOfBlocks*BlockSize;
class Buffer {
public:
  Buffer ();
  void put (Byte b);
  void read (Byte block[]);
private:
  Byte buffer[BufferSize];
  int rdCursor, wrCursor;
  Semaphore mutex, spaceAvailable, itemAvailable;
};
```

```
Buffer::Buffer () : rdCursor(0), wrCursor(0),
  mutex(1), spaceAvailable(BufferSize), itemAvailable(0) {}
void Buffer::put (Byte b) {
  spaceAvailable.wait();
  mutex.wait();
    this->buffer[this->wrCursor] = b;
    this->wrCursor = (this->wrCursor+1)%BufferSize;
    bool toSignal = (this->wrCursor%BlockSize == 0);
  mutex.signal();
  if (toSignal) itemAvailable.signal();
}
void Buffer::read (Byte block[]) {
  itemAvailable.wait();
  mutex.wait();
    for (int i=0; i<BlockSize; i++)
      block[i] = this->buffer[this->rdCursor+i];
    this->rdCursor = (this->rdCursor+BlockSize)%BufferSize;
  mutex.signal();
  spaceAvailable.signal(BlockSize);
}
```

1. zadatak, drugi kolokvijum, septembar 2016.

Školsko jezgro treba proširiti podrškom za slanje i prijem poruka između niti, implementacijom sledeće dve operacije klase `Thread`:

- `void Thread::send(char* message)`: pozivajuća nit šalje poruku datoj niti (`this`); ukoliko je ovoj niti već stigla neka poruka koju ona nije preuzela, pozivajuća nit se suspenduje dok se prethodna poruka ne preuzme i tek onda ostavlja poruku i nastavlja izvršavanje;
- `static char* Thread::receive()`: pozivajuća nit preuzima poruku koja joj je poslata; ukoliko poruke nema, pozivajuća nit se suspenduje dok poruka ne stigne.

Rešenje

U klasu `Thread` treba dodati sledeće članove:

```
char* Thread::message(0);
Semaphore msgEmpty(1), msgAvailable(0);
```

```
void Thread::send (char* msg) {
  this->msgEmpty.wait();
  this->message = msg;
  this->msgAvailable.signal();
}

char* Thread::receive () {
  Thread::running->msgAvailable.wait();
  char* msg = Thread::running->message;
  Thread::running->msgEmpty.signal();
  return msg;
}
```

1. zadatak, drugi kolokvijum, maj 2015.

U nastavku je dat parcijalni opis neznatno izmenjenog (pojednostavljenog) POSIX API za semafore. Napisati delove koda dva procesa koji međusobno isključuju izvršavanja svojih kritičnih sekcija korišćenjem ovih semafora.

```
#include <fcntl.h>
/* For O_* constants */
```

```
#include <sys/stat.h>
sem_t *sem_open(const char *name, int oflag, unsigned int value);
#include <semaphore.h>
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_close(sem_t *sem);
```

sem_open() creates a new POSIX semaphore or opens an existing semaphore. The semaphore is identified by name of the form /somename; that is, a null-terminated string of up to NAME_MAX-4 (i.e., 251) characters consisting of an initial slash, followed by one or more characters, none of which are slashes. Two processes can operate on the same named semaphore by passing the same name to sem_open().

The oflag argument specifies flags that control the operation of the call. (Definitions of the flags values can be obtained by including <fcntl.h>.) If O_CREAT is specified in oflag, then the semaphore is created if it does not already exist. If both O_CREAT and O_EXCL are specified in oflag, then an error is returned if a semaphore with the given name already exists.

If O_CREAT is specified in oflag, then the value argument must be specified. The value argument specifies the initial value for the new semaphore. If O_CREAT is specified, and a semaphore with the given name already exists, then value is ignored. After the semaphore has been opened, it can be operated on using sem_post() and sem_wait(). When a process has finished using the semaphore, it can use sem_close() to close the semaphore.

On success, sem_open() returns the address of the new semaphore; this address is used when calling other semaphore-related functions.

sem_post() increments (unlocks) the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem_wait() call will be woken up and proceed to lock the semaphore.

sem_wait() decrements (locks) the semaphore pointed to by sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero).

Rešenje

Deo koda oba procesa izgleda isto:

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

// Initialization:
const char* mutexName = "/myprogram_mutex";
sem_t* mutex = sem_open(mutexName,O_CREAT,1);

...
// Use for mutual exlusion
sem_wait(mutex);
// Critical section
sem_post(mutex);
...

// Release the semaphore when it is no longer needed:
sem_close(mutex);
```

1. zadatak, drugi kolokvijum, april 2013.

U nastavku je dat kod uporednih procesa koji simuliraju policijski helikopter i automobil u poteri za beguncem, objašnjen na predavanjima. Korišćenjem standardnih brojačkih semafora obezbediti potrebnu sinhronizaciju koja

eliminiše problem utrkivanja (*race condition*) koji postoji u ovom kodu, objašnjen na predavanjima, ali tako da se tokom izvršavanja potprograma `moveTo` ne sprečava napredovanje procesa `Helicopter`.

```
type Coord = record {
  x : integer;
  y : integer;
};

var sharedCoord : Coord;

process Helicopter
var nextCoord : Coord;
begin
  loop
    computeNextCoord(nextCoord);
    sharedCoord := nextCoord;
  end;
end;

process PoliceCar
begin
  loop
    moveTo(sharedCoord);
  end;
end;
```

Rešenje

```
type Coord = record {
  x : integer;
  y : integer;
};

var sharedCoord : Coord;
    mutex : Semaphore = 1;

process Helicopter
var nextCoord : Coord;
begin
  loop
    computeNextCoord(nextCoord);
    mutex.wait();
    sharedCoord := nextCoord;
    mutex.signal();
  end;
end;

process PoliceCar
var nextCoord : Coord;
begin
  loop
    mutex.wait();
    nextCoord := sharedCoord;
    mutex.signal();
    moveTo(nextCoord);
  end;
end;
```

1. zadatak, drugi kolokvijum, septembar 2013.

U nekom operativnom sistemu postoje sledeći sistemski pozivi koji se odnose na klasične brojačke semafore za sinhronizaciju između uporednih procesa:

- `sem_t* sem_open(const char* name, unsigned int value)`: Otvara semafor sa datim simboličkim imenom name za korišćenje u pozivajućem procesu, ukoliko je semafor sa tim imenom već kreiran (od strane istog ili nekog drugog procesa). Ukoliko nije, kreira takav semafor sa datom početnom vrednošću `value`. Različiti procesi tako mogu da dele isti semafor. Struktura `sem_t` predstavlja ručku do semafora (deskriptor). U slučaju greške, vraća 0.
- `int sem_wait(sem_t* sem)`: Operacija *wait* na datom semaforu. Vraća 0 u slučaju uspeha, -1 u slučaju greške.
- `int sem_post(sem_t* sem)`: Operacija *signal* na datom semaforu. Vraća 0 u slučaju uspeha, -1 u slučaju greške.
- `int sem_unlink(sem_t* sem)`: Oslobađa semafor od strane pozivajućeg procesa. Semafor se dealocira kada ga oslobode svi procesi koji ga koriste. Vraća 0 u slučaju uspeha, -1 u slučaju greške.

Sve navedene deklaracije nalaze se u `semaphore.h`.

Napisati program koji (u funkciji `main`) obezbeđuje međusobno isključenje kritične sekcije u odnosu na druge procese koji su na isti način konstruisani.

Rešenje

```
#include <semaphore.h>

void main () {
  sem_t* mutex = sem_open("mutex",1);
  if (mutex==0) return; // Exception!

  <noncritical_section>

  int status = sem_wait(mutex);
  if (status<0) ... // Exception!
  <critical_section>
  int status = sem_post(mutex);
  if (status<0) ... // Exception!

  <noncritical_section>
  sem_unlink(mutex);
}
```

1. zadatak, drugi kolokvijum, maj 2012.

Dva uporedna kooperativna procesa `A` i `B` razmenjuju podatke tako što proces `A` izračunava vrednost deljene promenljive a pozivom svoje privatne procedure `compute_a`. Tu vrednost promenljive `a` koristi proces `B` tako što na osnovu nje izračunava vrednost deljene promenljive `b` pozivom svoje privatne procedure `compute_b`. Tu vrednost promenljive `b` potom koristi proces `A` za izračunavanje nove vrednosti promenljive `a` i tako ciklično. Implementirati ove procese korišćenjem uposlenog čekanja za uslovnu sinhronizaciju. Inicijalno, proces `A` može odmah da izračuna početnu vrednost za `a` na osnovu statički inicijalizovane vrednosti za `b`, dok proces `B` ne može da izračuna prvu vrednost `b` dok ne dobije prvu izračunatu vrednost `a`.

Rešenje

```
shared var
  a, b : integer := 0;
  sa, sb : boolean := 0;

process a;
```

```
begin
  loop
    compute_a(b);
    sa:=1;
    while (sb==0) do null; sb:=0;
  end;
end;

process b;
begin
  loop
    while (sa==0) do null; sa:=0;
    compute_b(a);
    sb:=1;
  end;
end;
```

1. zadatak, drugi kolokvijum, septembar 2012.

Dva uporedna kooperativna procesa A i B treba da ulaze u kritičnu sekciju strogo naizmenično. Korišćenjem standardnih brojačkih semafora prikazati potrebnu sinhronizaciju.

Rešenje

```
shared var
  sa : Semaphore:=1,
  sb : Semaphore:=0;

process A;
begin
  loop
    wait(sa);
    <critical-section>
    signal(sb);
    <non-critical-section>
  end;
end;

process B;
begin
  loop
    wait(sb);
    <critical-section>
    signal(sa);
    <non-critical-section>
  end;
end;
```

1. zadatak, drugi kolokvijum, maj 2011.

Dva uporedna kooperativna procesa A i B razmenjuju podatke tako što proces A izračunava vrednost deljene promenljive `a` pozivom svoje privatne procedure `compute_a`. Tu vrednost promenljive `a` koristi proces B tako što na osnovu nje izračunava vrednost deljene promenljive `b` pozivom svoje privatne procedure `compute_b`. Tu vrednost promenljive `b` potom koristi proces A za izračunavanje nove vrednosti promenljive `a` i tako ciklično. Korišćenjem standardnih brojačkih semafora napisati kod procesa A i B sa svim neophodnim deklaracijama. Inicijalno, proces A može odmah da izračuna početnu vrednost za `a` na osnovu statički inicijalizovane vrednosti za `b`, dok proces B ne može da izračuna prvu vrednost `b` dok ne dobije prvu izračunatu vrednost `a`.

Rešenje

```
shared var
  a, b : integer := 0;
  sa, sb : semaphore := 0;

process a;
begin
  loop
    compute_a(b);
    sa.signal();
    sb.wait();
  end;
end;

process b;
begin
  loop
    sa.wait();
    compute_b(a);
    sb.signal();
  end;
end;
```

1. zadatak, drugi kolokvijum, septembar 2011.

Korišćenjem standardnih brojačkih semafora rešiti problem utrkivanja (engl. *race condition*) u kodu datog programa koji simulira rad policijskog helikoptera i automobila, a koji je objašnjen na predavanjima.

```
type Coord = record {
  x : integer;
  y : integer;
};

var sharedCoord : Coord;

process Helicopter
var nextCoord : Coord;
begin
  loop
    computeNextCoord(nextCoord);
    sharedCoord := nextCoord;
  end;
end;

process PoliceCar
begin
  loop
    moveTo(sharedCoord);
  end;
end;
```

Rešenje

```
type Coord = record {
  x : integer;
  y : integer;
};
```

```
var sharedCoord : Coord;
    readyToWrite : Semaphore = 1;
    readyToRead : Semaphore = 0;

process Helicopter
var nextCoord : Coord;
begin
  loop
    computeNextCoord(nextCoord);
    readyToWrite.wait;
    sharedCoord := nextCoord;
    readyToRead.signal;
  end;
end;

process PoliceCar
var nextCoord : Coord;
begin
  loop
    readyToRead.wait;
    nextCoord:= sharedCoord;
    readyToWrite.signal;
    moveTo(nextCoord);
  end;
end;
```

2. zadatak, drugi kolokvijum, maj 2010.

Kreirano je više procesa istog tipa P koji imaju istu sledeću strukturu. U kritičnoj sekciji A nalaze se ugnežđene dve kritične sekcije B i C, s tim da se sekcije B i C izvršavaju jedna posle druge (nisu ugnežđene). Potrebno je obezbediti sledeću sinhronizaciju: u kritičnoj sekciji A može se u jednom trenutku nalaziti najviše N ovih procesa tipa P, dok se u sekciji B, odnosno C može nalaziti najviše jedan proces. (Svaka od sekcija B i C je međusobno isključiva, ali se B i C ne isključuju međusobno – jedan proces može biti u B dok je drugi u C.) Korišćenjem standardnih brojačkih semafora obezbediti ovu sinhronizaciju: prikazati strukturu tipa procesa P, uz odgovarajuće definicije i inicijalizacije potrebnih semafora.

Rešenje

```
shared var mutexA : semaphore:=N;
           mutexB, mutexC : semaphore:=1;

type P = process begin
  ...
  wait(mutexA);
    <critical section A>
    ...
    wait(mutexB);
      <critical section B>
    signal(mutexB);
    ...
    wait(mutexC);
      <critical section C>
    signal(mutexC);
    ...
  signal(mutexA);
  ...
```

```
end;
```

2. zadatak, drugi kolokvijum, maj 2009.

Dat je kod dva tipa procesa A i B:

```
shared var count : integer := 0;
           mutex : semaphore:=1;
           gate : semaphore:=1;

type A = process begin
  ...
  wait(mutex);
    count:=count+1;
    if (count=1) then wait(gate);
  signal(mutex);
  ... (*  Critical section A *)
  wait(mutex);
    count:=count-1;
    if (count=0) then signal(gate);
  signal(mutex);
  ...
end;

type B = process begin
  ...
  wait(gate);
  ... (*  Critical section B *)
  signal(gate);
  ...
end;
```

Ukoliko postoji proizvoljno, ali konačno mnogo aktivnih procesa tipa A i procesa tipa B, precizno odgovoriti i objasniti koliko *istovremeno* može biti procesa u označenim kritičnim sekcijama.

Rešenje

Ili proizvoljno mnogo procesa tipa A u svojim kritičnim sekcijama i ni jedan proces tipa B u svojoj, ili samo jedan proces tipa B i ni jedan proces tipa A.

Semafor `mutex` služi da obezbedi međusobno isključenje pristupa deljenoj promeljivoj `count` od strane uporednih procesa tipa A. Promenjliva `count` je brojač procesa tipa A koji su ušli u svoju kritičnu sekciju. Kada prvi ovakav proces ulazi u svoju kritičnu sekciju, izvršiće `wait` na semaforu `gate` i ili proći taj semafor (i zatvoriti ga), ili se blokirati na njemu. U prvom slučaju, svi naredni procesi tipa A slobodno ulaze u svoju kritičnu sekciju. U drugom slučaju, svi naredni procesi tipa A koji žele da uđu u svoju kritičnu sekciju će se blokirati na semaforu `mutex`. Poslednji proces tipa A koji izlazi iz svoje kritične sekcije izvršiće `signal` na semaforu `gate`. Kako samo jedan proces može proći operaciju `wait` na semaforu `gate` bez blokiranja, to znači da će ili prvi proces tipa A ili samo jedan proces tipa B to moći da uradi i tako uđe u svoju kritičnu sekciju. Odatle sledi dati odgovor.

2. zadatak, drugi kolokvijum, maj 2006.

Korišćenjem koncepta spoljašnjeg događaja iz zadatka 1, realizovati telo niti koja ciklično izvršava sledeći posao:

- uzima jedan zahtev iz liste postavljenih zahteva za ulaznom operacijom prenosa bloka podataka sa nekog ulaznog uređaja u memoriju pomoću DMA kontrolera, odnosno čeka blokirana ako takvog zahteva nema
- pokreće operaciju prenosa bloka podataka na zadato mesto u memoriji pomoću DMA kontrolera
- čeka (suspendovana) da DMA kontroler završi prenos, što DMA kontroler signalizira prekidom na ulazu N
- kada se završi prenos, označava zahtev ispunjenim i signalizira semafor koji je zadat u zahtevu, kako bi se nit koja je postavila zahtev deblokirala.

Zahtev za ulaznom operacijom i red zahteva izgledaju ovako:

```
struct IORequest {
  IORequest* next; // Next in the request queue
  void* buffer;  // Input buffer in memory
  long size;  // Size of the data block
  int isCompleted; // Is this request completed?
  Semaphore* toSignal;  // The semaphore to signal on completion
};

extern IORequest* requestQueue;
extern Semaphore mutex;
extern Semaphore requestQueSize;
```

Za pristup do reda zahteva treba obezbediti međusobno isključenje pomoću klasičnog brojačkog semafora `mutex`. Brojački semafor `requestQueSize` ima vrednost koja je jednaka broju zahteva u redu zahteva requestQueue. Prenosa bloka podataka sa DMA kontrolera pokreće se operacijom `void startDMA(void* buffer, long size)`.

Rešenje

```
ExternalEvent dmaEvent(N);

while (1) {
  requestQueSize.wait();
  mutex.wait();
  IORequest* r = requestQueue;
  requestQueue = requestQueue->next;
  mutex.signal();

  startDMA(r->buffer, r->size);
  dmaEvent.wait();

  r ->isCompleted = 1;
  r ->toSignal->signal();
}
```

Baferi, proizvođač/potrošač

3. zadatak, drugi kolokvijum, maj 2022.

Više procesa proizvođača i potrošača međusobno razmenjuju znakove (`char`) preko ograničenog bafera tako što svi ti procesi dele jedan zajednički logički segment memorije veličine `sizeof(bbuffer)` koji su alocirali sistemskim pozivom `mmap` kao *bss* segment (inicijalizovan nulama pri alokaciji). Svaki od tih procesa adresu tog segmenta konvertuje u pokazivač na tip `struct bbuffer` i dalje radi operacije sa ograničenim baferom pozivajući sledeće operacije iz biblioteke `bbuf` čije su deklaracije (`bbuf.h`) date dole:

- `bbuf_init`: svaki proces koji želi da koristi bafer mora najpre da pozove ovu operaciju kako bi ona otvorila potrebne semafore; ukoliko neki od sistemskih poziva vezanih za semafore nije uspeo, sve one već otvorene semafore treba osloboditi i vratiti negativnu vrednost (greška); ukoliko je inicijalizacija uspela, treba vratiti 0;
- `bbuf_close`: svaki proces koji koristi bafer treba da pozove ovu operaciju kada završi sa korišćenjem bafera;
- `bbuf_append`, `bbuf_take`: operacije stavljanja i uzimanja elementa (`char`) u bafer.

Sve ove operacije podrazumevaju da je argument ispravan pokazivač (ne treba ga proveravati). Sve operacije osim `bbuf_init` takođe podrazumevaju da je inicijalizacija pre toga uspešno završena – proces ih ne sme pozivati ako nije, pa ne treba proveravati ispravnost stanja bafera.

Implementirati biblioteku `bbuf` (`bbuf.cc`): dati definiciju strukture `bbuffer` i implementirati sve ove operacije korišćenjem POSIX semafora.

```
struct bbuffer;
int bbuf_init(struct bbuffer*);
void bbuf_append(struct bbuffer*, char);
char bbuf_take(struct bbuffer*);
void bbuf_close(struct bbuffer*);
```

Rešenje

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

const size_t BBSIZE = ...;
static sem_t *mutex, *space_available, *item_available;

struct bbuffer {
    char buf[BBSIZE];
    int head, tail;
};

int bbuf_init(struct bbuffer* bb) {
    mutex = sem_open("/bounded_buf_mx", O_CREAT, O_RDWR, 1);
    space_available = sem_open("/bounded_buf_sa", O_CREAT, O_RDWR, BBSIZE);
    item_available = sem_open("/bounded_buf_ia", O_CREAT, O_RDWR, 0);
    if (!mutex || !space_available || !item_available) {
        if (mutex) {
            sem_close(mutex);
        }
        if (space_available) {
            sem_close(space_available);
        }
        if (item_available) {
            sem_close(item_available);
        }
        return -1;
    }
```

```
    return 0;
}

void bbuf_close(struct bbuffer* bb) {
    if (mutex) {
        sem_close(mutex);
    }
    if (space_available) {
        sem_close(space_available);
    }
    if (item_available) {
        sem_close(item_available);
    }
}

void bbuf_append(struct bbuffer* bb, char c) {
    sem_wait(space_available);
    sem_wait(mutex);
    bb->buf[bb->tail] = c;
    bb->tail = (bb->tail+1)%BBSIZE;
    sem_post(mutex);
    sem_post(item_available);
}

char bbuf_take(struct bbuffer* bb) {
    sem_wait(item_available);
    sem_wait(mutex);
    char c = bb->buf[bb->head];
    bb->head = (bb->head+1)%BBSIZE;
    sem_post(mutex);
    sem_post(space_available);
    return c;
}
```

3. zadatak, kolokvijum, jul 2020.

Realizovati u potpunosti klasu `DoubleBuffer` čiji je interfejs dat. Ova klasa implementira dvostruki bafer. Proizvođač stavlja u „izlazni“ bafer blokove veličine `chunkSizeProd` znakova pozivom operacije `put()`; potrošač uzima iz „ulaznog“ bafera blokove veličine `chunkSizeCons` znakova pozivom operacije `get()`. Kada obojica završe sa svojim baferom, baferi zamenjuju uloge. Proizvođač i potrošač su uporedne niti (ne treba ih realizovati), dok je sva potrebna sinhronizacija unutar klase `DoubleBuffer`. Pretpostaviti da je zadata veličina oba bafera u znakovima (argument `size` konstruktora) celobrojan umnožak obe zadate veličine bloka (argumenti `chunkSize`). Za sinhronizaciju koristiti semafore školskog jezgra.

```
class DoubleBuffer {
public:
  DoubleBuffer (size_t size, size_t chunkSizeProd, size_t chunkSizeCons);
  void put (const char* buffer);
  void get (char* buffer);
private:
  ...
};
```

Rešenje

```
class DoubleBuffer {
public:
```

```
  DoubleBuffer (size_t size, size_t chunkSizeProd, size_t chunkSizeCons);
  void put (const char* buffer);
  void get (char* buffer);

private:
  Semaphore inputBufReady, outputBufReady;
  char* buffer[2];
  size_t size, chunkP, chunkC, head, tail, slots, items;
  int inputBuf, outputBuf;
};
DoubleBuffer::DoubleBuffer (size_t sz, size_t cp, size_t cc)
  : inputBufReady(1), outputBufReady(0) {
  buffer[0] = new char[sz];
  buffer[1] = new char[sz];
  size = sz;
  chunkP = ((cp>0)?cp:1);
  chunkC = ((cc>0)?cc:1);
  head = tail = 0;
  slots = size; items = 0;
  inputBuf = 0; outputBuf = 1;
}
void DoubleBuffer::put (const char* buf) {
  if (slots==0) {
    inputBufReady.wait();
    outputBuf = !outputBuf;
    slots = size;
    tail = 0;
  }
  for (size_t i=0; i<chunkP; i++) {
    buffer[outputBuf][tail++] = buf[i++];
    slots--;
  }
  if (slots==0)
    outputBufReady.signal();
}
void DoubleBuffer::get (char* buf) {
  if (items==0) {
    outputBufReady.wait();
    inputBuf = !inputBuf;
    items = size;
    head = 0;
  }
  for (size_t i=0; i<chunkC; i++) {
    buf[i++]= buffer[inputBuf][head++];
    items--;
  }
  if (items==0)
    inputBufReady.signal();
}
```

1. zadatak, treći kolokvijum, septembar 2013.

Realizovati u potpunosti klasu `DoubleBuffer` čiji je interfejs dat. Ova klasa implementira dvostruki bafer. Proizvođač stavlja u bafer blokove veličine `chunkSize` znakova pozivom operacije `put()`; znakovi se uzimaju iz bafera pozivaoca na koji ukazuje argument buffer, a stavljaju u trenutni „izlazni“ bafer od dva interna bafera veličine `size` znakova. Potrošač uzima znak po znak iz trenutnog „ulaznog“ bafera pozivom operacije `get()`. Kada obojica završe sa svojim baferom, baferi zamenjuju uloge. Proizvođač i potrošač su uporedne niti (ne treba ih realizovati), dok je sva potrebna

sinhronizacija unutar klase `DoubleBuffer`. Pretpostaviti da je zadata veličina bafera u znakovima (argument `size` konstruktora) celobrojan umnožak zadate veličine bloka (argument `chunkSize`). Za sinhronizaciju koristiti semafore.

```
class DoubleBuffer {
public:
  DoubleBuffer (int size, int chunkSize);
  void put (char* buffer);
  char get ();
private:
  ...
};
```

Rešenje

```
class DoubleBuffer {
public:
  DoubleBuffer (int size, int chunkSize);
  void put (char* buffer);
  char get ();
private:
  Semaphore inputBufReady, outputBufReady;
  char* buffer[2];
  int size, chunk, head, tail, slots, items, inputBuf, outputBuf;
};

DoubleBuffer::DoubleBuffer (int sz, int cs)
  : inputBufReady(1), outputBufReady(0) {
  buffer[0] = new char[sz];
  buffer[1] = new char[sz];
  size = sz;
  chunk = ((cs>0)?cs:1);
  head = tail = 0;
  slots = size; items = 0;
  inputBuf = 0; outputBuf = 1;
}

void DoubleBuffer::put (char* buf) {
  if (slots==0) {
    inputBufReady.wait();
    outputBuf = !outputBuf;
    slots = size;
    tail = 0;
  }
  for (int i=0; i<chunk; i++) {
    buffer[outputBuf][tail++] = buf[i++];
    slots--;
  }
  if (slots==0)
    outputBufReady.signal();
}

char DoubleBuffer::get () {
  if (items==0) {
    outputBufReady.wait();
    inputBuf = !inputBuf;
    items = size;
    head = 0;
```

```
  }
  char ret = buffer[inputBuf][head++];
  items--;
  if (items==0)
    inputBufReady.signal();
  return ret;
}
```

1. zadatak, treći kolokvijum, septembar 2012.

Realizovati u potpunosti klasu `DoubleBuffer` čiji je interfejs dat. Ova klasa implementira dvostruki bafer. Proizvođač stavlja u bafer znak po znak pozivom operacije `put()`; znak se stavlja u trenutni „izlazni“ bafer od dva interna bafera veličine `size` znakova. Potrošač uzima blokove veličine `chunkSize` znakova iz trenutnog „ulaznog“ bafera pozivom operacije `get()`; znakovi se prepisuju u bafer pozivaoca na koji ukazuje argument `buffer`. Kada obojica završe sa svojim baferom, baferi zamenjuju uloge. Proizvođač i potrošač su uporedne niti (ne treba ih realizovati), dok je sva potrebna sinhronizacija unutar klase `DoubleBuffer`. Pretpostaviti da je zadata veličina bafera u znakovima (argument `size` konstruktora) celobrojan umnožak zadate veličine bloka (argument `chunkSize`). Za sinhronizaciju koristiti semafore.

```
class DoubleBuffer {
public:
  DoubleBuffer (int size, int chunkSize);
  void put (char);
  void get (char* buffer);
private:
  ...
};
```

Rešenje

```
class DoubleBuffer {
public:
  DoubleBuffer (int size, int chunkSize);
  void put (char);
  void get (char* buffer);
private:
  Semaphore inputBufReady, outputBufReady;
  char* buffer[2];
  int size, chunk, head, tail, slots, items, inputBuf, outputBuf;
};

DoubleBuffer::DoubleBuffer (int sz, int cs)
  : inputBufReady(1), outputBufReady(0) {
  buffer[0] = new char[sz];
  buffer[1] = new char[sz];
  size = sz;
  chunk = ((cs>0)?cs:1);
  head=tail=0;
  slots=size; items=0;
  inputBuf=0; outputBuf=1;
}

void DoubleBuffer::put (char c) {
  if (slots==0) {
    inputBufReady.wait();
    outputBuf=!outputBuf;
    slots=size;
```

```
    tail=0;
  }
  buffer[outputBuf][tail++]=c;
  slots--;
  if (slots==0)
    outputBufReady.signal();
}

void DoubleBuffer::get (char* buf) {
  if (items==0) {
    outputBufReady.wait();
    inputBuf=!inputBuf;
    items=size;
    head=0;
  }
  for (int i=0; i<chunk; i++) {
    buf[i++] = buffer[inputBuf][head++];
    items--;
  }
  if (items==0)
    inputBufReady.signal();
}
```

1. zadatak, treći kolokvijum, jun 2011.

Realizovati u potpunosti klasu `DoubleBuffer` čiji je interfejs dat. Ova klasa implementira dvostruki bafer. Proizvođač stavlja element tipa `Data*` pozivom operacije `put()`, čime se element stavlja u trenutni „izlazni“ bafer od dva interna bafera, dok potrošač uzima elemente iz trenutnog „ulaznog“ bafera pozivom operacije `get()`. Kada obojica završe sa svojim baferom, baferi zamenjuju uloge. Proizvođai i potrošai su uporedne niti (ne treba ih realizovati), dok je sva potrebna sinhronizacija unutar klase `DoubleBuffer`. Za sinhronizaciju koristiti semafore.

```
class Data;

class DoubleBuffer {
public:
  DoubleBuffer (int size);
  void put (Data*);
  Data* get ();
private:
  ...
};
```

Rešenje

```
class Data;

class DoubleBuffer {
public:
  DoubleBuffer (int size);
  void put (Data*);
  Data* get ();
private:
  Semaphore inputBufReady, outputBufReady;
  Data** buffer[2];
  int size, head, tail, slots, items, inputBuf, outputBuf;
};
```

```
DoubleBuffer::DoubleBuffer (int sz) : inputBufReady(1), outputBufReady(0) {
  buffer[0] = new (Data*)[sz];
  buffer[1] = new (Data*)[sz];
  size = sz;
  head=tail=0;
  slots=size; items=0;
  inputBuf=0; outputBuf=1;
}

void DoubleBuffer::put (Data* d) {
  if (slots==0) {
    inputBufReady.wait();
    outputBuf=!outputBuf;
    slots=size;
    tail=0;
  }
  buffer[outputBuf][tail++]=d;
  slots--;
  if (slots==0)
    outputBufReady.signal();
}

Data* DoubleBuffer::get () {
  if (items==0) {
    outputBufReady.wait();
    inputBuf=!inputBuf;
    items=size;
    head=0;
  }
  Data* d = buffer[inputBuf][head++];
  items--;
  if (items==0)
    inputBufReady.signal();
  return d;
}
```

2. zadatak, drugi kolokvijum, maj 2008.

Jedan proces-proizvođač proizvodi elemente i stavlja ih u ograničeni bafer, dok dva procesa- potrošača uzimaju elemente iz tog bafera. Potrebno je obezbediti odgovarajuću sinhronizaciju između ovih procesa tako da potrošači naizmenično uzimaju po jedan element iz bafera. Realizovati ograničeni bafer koji obezbeđuje ovu sinhronizaciju pomoću klasičnih brojačkih semafora.

Rešenje

```
class Data;
const int N = ...;

class BoundedBuffer {
public:
  BoundedBuffer();
  void put (Data*);
  Data* get (int consumerID);  // consumerID should be 1 or 2
private:
  Data* buf[N];
  int head, tail;
  Semaphore mutex, spaceAvailable, itemAvailable, gate1, gate2;
```

```
};

BoundedBuffer::BoundedBuffer () :
  head(0), tail(0),
  mutex(1), spaceAvailable(N), itemAvailable(0),
  gate1(1), gate2(0) {}

void BoundedBuffer::put (Data* d) {
    spaceAvailable.wait();
    mutex.wait();
    buf[tail]=d;
    tail=(tail+1)%N;
    mutex.signal();
    itemAvailable.signal();
}

Data* BoundedBuffer::get (int myID) {
    if (myID==1)
      gate1.wait();
    else if (myID==2)
      gate2.wait();
    else return 0; // Error

    itemAvailable.wait();
    mutex.wait();
    Data* d = buf[head];
    head=(head+1)%N;
    mutex.signal();
    spaceAvailable.signal();
    if (myID==1)
      gate2.signal();
    else if (myID==2)
      gate1.signal();
    return d;
}
```

Prikazano rešenje je napisano uz pretpostavku da se jednoj niti daje prednost, tj. unapred je određena nit koja mora prva da uzme podatak iz bafera. To rešenje je u potpunosti ispravno i prihvatljivo, ali je moguće dati i za nijansu bolje rešenje. U nastavku je prikazano rešenje koje ne favorizuje nijednu nit, tj. ona nit koja prva zatraži podatak, dobija ga, a dalje se uzimanje podataka iz bafera obavlja naizmenično. Izmene u odnosu na prethodno rešenje su posebno naznačene.

```
class Data;
const int N = ...;

class BoundedBuffer
{
public:
  BoundedBuffer();
  void put (Data*);
  Data* get (int consumerID);  // consumerID should be 1 or 2
private:
  Data* buf[N];
  int head, tail, **firstTime**; // IZMENA
  Semaphore mutex, spaceAvailable, itemAvailable, gate1, gate2;
};
```

```
BoundedBuffer::BoundedBuffer () :
  head(0), tail(0), **firstTime (1)**, // IZMENA
  mutex(1), spaceAvailable(N), itemAvailable(0),
  gate1(0), gate2(0) {}

void BoundedBuffer::put (Data* d) {
    spaceAvailable.wait();
    mutex.wait();
    buf[tail]=d;
    tail=(tail+1)%N;
    mutex.signal();
    itemAvailable.signal();
}

Data* BoundedBuffer::get (int myID) {
    // IZMENA:
    if (firstTime) { //dobro zbog brzine, da se izbegne nepotrebna
                     // sinhronizacija posle prvog puta
      mutex.wait();
      if (firstTime) { //iako je vec provereno, mora se ponovo proveriti
                       // nedeljivo sa promenom
        if (myID==1)
          gate1.signal();
        else if (myID==2)
          gate2.signal();
        firstTime = 0;
      }
      mutex.signal()
    } //firstTime
    // KRAJ IZMENE

    if (myID==1)
      gate1.wait();
    else if (myID==2)
      gate2.wait();
    else return 0; // Error
    itemAvailable.wait();
    mutex.wait();
    Data* d = buf[head];
    head=(head+1)%N;
    mutex.signal();
    spaceAvailable.signal();
    if (myID==1)
      gate2.signal();
    else if (myID==2)
      gate1.signal();
    return d;
}
```

2. zadatak, drugi kolokvijum, maj 2007.

Dat je sledeći interfejs klase koja realizuje ograničeni bafer (*bounded buffer*) kao posrednik između samo jednog proizvođača (*producer*) i samo jednog potrošača (*consumer*) koji su uporedne niti:

```
class BoundedBuffer {
public:
```

```
  BoundedBuffer(int capacity);
  void put (char* package, int size);
  void get (char* package, int size);
};
```

U bafer se smeštaju znakovi (`char`). Operacija put smešta u bafer dati niz znakova date dužine. Operacija get iz bafera uzima `size` znakova i smešta ih u niz na koji ukazuje pokazivač `package`. Ovi nizovi (paketi koji se prenose) su proizvoljne i promenljive dužine (ali svakako manje od kapaciteta bafera). Implementirati ovu klasu korišćenjem standardnih brojačkih semafora.

Rešenje

```
class BoundedBuffer {
public:
  BoundedBuffer(int capacity);
  void put (char* package, int size);
  void get (char* package, int size);
private:
  char* buf;
  int capacity, head, tail;
  Semaphore spaceAvailable, itemAvailable;
};

BoundedBuffer::BoundedBuffer (int cap) :
  buf(new char[cap]), capacity(cap), head(0), tail(0),
  spaceAvailable(cap), itemAvailable(0) {}

void BoundedBuffer::put (char* p, int n) {
  for (int i=0; i<n; i++) {
    spaceAvailable.wait();
    buf[tail]=p[i];
    tail=(tail+1)%capacity;
    itemAvailable.signal();
  }
}
```

Operacija `get` analogno.

Ulaz/izlaz

1. zadatak, treći kolokvijum, jun 2022.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima kontrolera tastature preko koga stižu znakovi otkucani na tastaturi:

```
typedef volatile unsigned REG;
REG* ioStatus =...;     // status register
char* ioData =...;      // data register
```

Kontroler tastature poseduje interni memorijski modul koji služi za prihvatanje jednog ili više znakova otkucanih na tastaturi (hardverski bafer). Kada se ovom baferu pojave znakovi (jedan ili više), kontroler generiše prekid. Tada se iz registra za podatke mogu čitati pristigli znakovi sukcesivnim operacijama čitanja, jedan po jedan, sve dok je bit spremnosti (*ready*) u razredu 0 statusnog registra postavljen na 1. Naredni nalet pristiglih znakova će ponovo generisati prekid.

Za smeštanje znakova učitanih sa tastature kernel koristi svoj (softverski) ograničeni bafer veličine 256 znakova. U ovaj bafer upisuju se znakovi učitani sa kontrolera tastature sve dok u njemu ima mesta; znakovi koji ne mogu da stanu u bafer se jednostavno odbacuju. Iz ovog bafera znakove uzimaju različiti uporedni tokovi kontrole (procesi) pozivom operacije `getc`; ukoliko u baferu nema znakova, pozivajući tok kontrole treba da se suspenduje dok znakova ne bude. Sinhronizacija se može obavljati semaforima čiji je interfejs isti kao u školskom jezgru. Pretpostaviti sledeće:

- prekidna rutina izvršava se međusobno isključivo sa operacijama na semaforu;
- operacija `signal` na semaforu može se pozivati iz prekidne rutine, jer ona ne radi nikakvu promenu konteksta;
- procesor maskira prekide pri obradi prekida; prekid sa kontrolera tastature se može eksplicitno maskirati pozivom `kbint_mask()`, a demaskirati pozivom `kbint_unmask()`.

Implementirati opisani podsistem: bafer kernela (operaciju `getc` i sve druge potrebne operacije) i prekidnu rutinu kontrolera tastature.

Rešenje

```
class KeyboardBuffer {
public:
    KeyboardBuffer() : head(0), tail(0), count(0), mutex(1), itemAvailable(0) {}
    char getc();
    void putc(char c);
private:
    static const size_t KB_SIZE = 256;
    char buffer[KB_SIZE];
    size_t head, tail, count;
    Semaphore mutex, itemAvailable;
};
static KeyboardBuffer* instance() {
    static KeyboardBuffer _instance;
    return &_instance;
}
char KeyboardBuffer::getc() {
    itemAvailable.wait();
    mutex.wait();
    kbint_mask();
    char c = buffer[head];
    head = (head + 1) % KB_SIZE;
    count--;
    kbint_unmask();
    mutex.signal();
    return c;
}
```

```
void KeyboardBuffer::putc(char c) {
    if (count < KB_SIZE) {
        buffer[tail] = c;
        tail = (tail + 1) % KB_SIZE;
        count++;
        itemAvailable.signal();
    }
}
interrupt void keyb_int() {
    while (*ioStatus & 1) {
        KeyboardBuffer::instance()->putc(*ioData);
    }
}
```

3. zadatak, kolokvijum, jul 2021.

U računaru postoji ulazni uređaj koji prihvata pakete sa neke komunikacione veze. Paket je veličine `PACKET_SIZE` (u `sizeof(char)`). Kada paket stigne u interni hardverski bafer uređaja, uređaj generiše prekid na koji je vezana rutina `packetArrived` kernela. Kernel treba da prebaci paket u svoj interni memorijski kružni bafer `packetBuffer` koji ima mesta za `BUFFER_SIZE` paketa. Ostatak kernela te pakete iz kružnog bafera kernela isporučuje dalje kome treba (procesima). Kada prebaci paket u svoj kružni bafer, kernel treba to da signalizira uređaju upisom konstante `IO_COMPLETE` u upravljački registar uređaja, kako bi ovaj znao da može da prihvati naredni paket u svoj interni bafer. Ukoliko je kružni bafer kernela pun, kernel treba uređaju da javi da paket mora da se odbaci upisom konstante `IO_REJECT` u upravljački registar uređaja. Interni hardverski bafer uređaja je veličine jednog paketa i preslikan je u memorijski prostor počev od adrese `ioData`. Date su deklaracije pokazivača preko kojih se može pristupiti registrima kontrolera ulaznog uređaja, kao i deklaracije za kružni bafer:

```
typedef volatile unsigned int REG;
REG* ioCtrl = ...; // Input device control register
volatile char* ioData = ...; // Input device packet buffer

extern const size_t PACKET_SIZE, BUFFER_SIZE;
extern char packetBuffer[BUFFER_SIZE * PACKET_SIZE];
extern char *pbHead, *pbTail;

interrupt void packetArrived();
```

Glava `pbHead` ukazuje na prvi znak paketa u kružnom baferu koji je na redu za prenos u ostatak kernela, a rep `pbTail` ukazuje na prvi znak prvog slobodnog mesta za prijem novog paketa u kružni bafer. Tokom izvršavanja prekidnih rutina drugi prekidi su maskirani i nema uporednog izvršavanja ostalog koda kernela.

Realizovati prekidnu rutinu `packetArrived`.

Rešenje

```
interrupt void packetArrived() {
    if (pbTail == pbHead) { // Buffer full, reject packet
        *ioCtrl = IO_REJECT;
    } else {
        for (int i = 0; i < PACKET_SIZE; i++) pbTail[i] = ioData[i];
        *ioControl = IO_COMPLETE;
        if ((pbTail - packetBuffer)/PACKET_SIZE == BUFFER_SIZE - 1) {
            pbTail = packetBuffer;
        } else {
            pbTail = pbTail + PACKET_SIZE;
        }
    }
}
```

3. zadatak, kolokvijum, avgust 2021.

U računaru postoji ulazni uređaj koji prihvata pakete sa neke komunikacione veze. Paket je veličine `PACKET_SIZE` (u `sizeof(char)`). Kada paket stigne u interni hardverski bafer uređaja, uređaj generiše prekid na koji je vezana rutina `packetArrived` kernela. Kernel treba da prebaci paket u svoj interni memorijski kružni bafer `packetBuffer` koji ima mesta za
`BUFFER_SIZE` paketa. Ovo prebacivanje kernel obavlja DMA prenosom. Ostatak kernela te pakete iz kružnog bafera kernela isporučuje dalje kome treba (procesima). Kada prebaci paket u svoj kružni bafer, kernel treba to da signalizira uređaju upisom konstante `IO_COMPLETE` u upravljački registar uređaja, kako bi ovaj znao da može da prihvati naredni paket u svoj interni bafer. Ukoliko je kružni bafer kernela pun ili pri DMA prenosu dođe do greške, kernel treba uređaju da javi da paket mora da se odbaci upisom konstante `IO_REJECT` u upravljački registar uređaja. Date su deklaracije pokazivača preko kojih se može pristupiti registrima DMA kontrolera i kontrolera ulaznog uređaja, kao i deklaracije za kružni bafer:

```
typedef unsigned int REG;
REG* dmaCtrl =...;   // DMA control register
REG* dmaStatus =...; // DMA status register
REG* dmaAddress =...;   // DMA block address register
REG* dmaCount =...;   // DMA block size register
REG* ioCtrl =...;   // Input device control register
extern const size_t PACKET_SIZE, BUFFER_SIZE;
extern char packetBuffer[BUFFER_SIZE * PACKET_SIZE];
extern char *pbHead, *pbTail;

interrupt void packetArrived ();
interrupt void dmaComplete ();
```

Greška u DMA prenosu signalizira se postavljanjem bita u statusnom registru; ovaj bit maskira se konstantom `DMA_ERROR`. DMA prenos pokreće se upisom konstante `DMA_START` u upravljački registar DMA kontrolera. Kada DMA kontroler završi prenos, generiše prekid na koji je vezana rutina `dmaComplete` kernela. Po završetku prenosa ne treba ništa upisivati u upravljački registar DMA kontrolera. Glava `pbHead` ukazuje na prvi znak paketa u kružnom baferu koji je na redu za prenos u ostatak kernela, a rep `pbTail` ukazuje na prvi znak prvog slobodnog mesta za prijem novog paketa u kružni bafer. Tokom izvršavanja prekidnih rutina drugi prekidi su maskirani i nema uporednog izvršavanja ostalog koda kernela. Realizovati prekidne rutine `packetArrived` i `dmaComplete`.

Rešenje

```
interrupt void packetArrived () {
  if (pbTail == pbHead) {  // Buffer full, reject packet
    *ioCtrl = IO_REJECT;
  } else {
    *dmaAddress = pbTail;
    *dmaCount = PACKET_SIZE;
    *dmaCtrl = DMA_START;
  }
}

interrupt void dmaComplete () {
  if (*dmaStatus & DMA_ERROR) {
    *ioControl = IO_REJECT;
  } else {
    *ioControl = IO_COMPLETE;
    if ((pbTail-packetBuffer)/PACKET_SIZE == BUFFER_SIZE-1)
      pbTail = packetBuffer;
    else
      pbTail = pbTail+PACKET_SIZE;
  }
}
```

3. zadatak, kolokvijum, oktobar 2020.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima jednog DMA kontrolera:

```
typedef unsigned int REG;
REG* dmaCtrl =...;  // DMA control register
REG* dmaStatus =...; // DMA status register
REG* dmaAddress =...;  // DMA block address register
REG* dmaCount =...;  // DMA block size register
```

U upravljačkom registru najniži bit je bit *Start* kojim se pokreće prenos jednog bloka preko DMA, a u statusnom registru najniži bit je bit završetka prenosa (*TransferComplete*), a bit do njega bit greške (*Error*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`).

Zahtevi za ulaznim operacijama na nekom uređaju sa kog se prenos blokova vrši preko ovog DMA kontrolera vezani su u jednostruko ulančanu listu. Zahtev ima sledeću strukturu:

```
struct IORequest {
  REG* buffer; // Data buffer (block)
  unsigned int size; // Buffer (blok) size
  int status; // Status of operation
  IORequest* next; // Next in the list
};
```

Na prvi zahtev u listi pokazuje globalni pokazivač `ioHead`. Kada u praznu listu kernel stavi prvi zahtev, pozvaće operaciju `transfer` koja treba da pokrene prenos za taj prvi zahtev i pokrene DMA kontroler upisom u bit *Start*. Kada se završi prenos zadat jednim zahtevom, potrebno je u polje `status` date strukture preneti status završene operacije (0 – ispravno završeno do kraja, -1 – greška), izbaciti obrađeni zahtev iz liste i pokrenuti prenos za sledeći zapis u listi, bez zaustavljanja DMA kontrolera. Ako zahteva u listi više nema, tek onda treba zaustaviti DMA kontroler resetovanjem bita *Start*. Kada bude stavljao novi zahtev u listu, kernel će proveriti i videti da je ona bila prazna, pa ponovo pozvati operaciju `transfer` itd.

Potrebno je napisati kod operacije `transfer()`, zajedno sa odgovarajućom prekidnom rutinom `dmaInterrupt()` za prekid od DMA kontrolera.

```
void transfer ();
interrupt void dmaInterrupt ();
```

Rešenje

```
extern IORequest* ioHead;
IORequest* pending = 0;

int startDMA () {
  pending = ioHead;
  if (pending==0) return 0;
  ioHead = ioHead->next;
  *dmaAddress = pending->buffer;
  *dmaCount = pending->size;
  return 1;
}

void transfer() {
  startDMA();
  *dmaCtrl = 1; // Start I/O
}

interrupt void dmaInterrupt () {
  if (pending==0) return; // Exception
  if (*dmaStatus&2) // Error in I/O
    pending->status = -1;
```

```
  else
    pending->status = 0;
  if (startDMA()==0) { // No more requests
    *dmaCtrl = 0;
    return;
  }
}
```

3. zadatak, kolokvijum, avgust 2020.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima kontrolera jednog ulaznog uređaja:

```
typedef unsigned int REG;
REG* ioCtrl =...;   // control register
REG* ioStatus =...; // status register
REG* ioData =...;   // data register
const unsigned BlockSize = ...; // block size
```

Uređaj je blokovski, što znači da jedna ulazna operacija uvek prenosi jedan blok iste veličine `BlockSize` reči, ali jednu po jednu reč preko registra za podatke. U upravljačkom registru najniži bit je bit *Start* kojim se pokreće prenos. Za svaki nov prenos jednog bloka potrebno je upisati 1 u bit *Start*, a na kraju prenosa bloka u taj bit upisati 0. U statusnom registru najniži bit je bit *Ready* koji signalizira spremnost jedne reči u registru za podatke, a bit do njega bit greške (*Error*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`). U slučaju greške u prenosu, uređaj generiše isti prekid kao i u slučaju spremnosti za prenos novog podatka.

Zahtevi za ulaznim operacijama prenosa blokova podataka na ovom uređaju vezani su u jednostruko ulančanu listu. Zahtev ima sledeću strukturu:

```
struct IORequest {
  REG* buffer; // Data buffer (block)
  int status; // Status of operation
  IORequest* next; // Next in the list
};
```

Na prvi zahtev u listi pokazuje globali pokazivač `ioHead`. Kada u praznu listu kernel stavi prvi zahtev, pozvaće operaciju `transfer()` koja treba da pokrene prenos za taj prvi zahtev. Kada se završi prenos bloka zadat jednim zahtevom, potrebno je u polje `status` date strukture preneti status završene operacije (0 – ispravno završeno do kraja, -1 – greška), izbaciti obrađeni zahtev iz liste i pokrenuti prenos za sledeći zapis u listi. Ako zahteva u listi više nema, ne treba uraditi više ništa (kada bude stavljao novi zahtev u listu, kernel će proveriti i videti da je ona bila prazna, pa pozvati ponovo operaciju `transfer()` itd.)

Potrebno je napisati kod operacije `transfer()`, zajedno sa odgovarajućom prekidnom rutinom `ioInterrupt()` za prekid od uređaja, pri čemu prenos treba vršiti tehnikom programiranog ulaza-izlaza uz korišćenje prekida.

```
void transfer ();
interrupt void ioInterrupt ();
```

Rešenje

```
static REG* ptr = 0;  // pointer to current data item
static int count = 0; // counter

void startIO () { // Helper: start a new transfer
  if (ioHead==0) return;
  ptr = ioHead->buffer;
  count = BlockSize;
  *ioCtrl = 1; // Start I/O
}

void transfer () {
```

```
  startIO();
}

interrupt void ioInterrupt () {
  if (*ioStatus&2)
    ioHead->status = -1;  // Error in I/O
  else {  // Transfer the next data item
    *ptr++ = *ioData;
    if (--count)
      return;
    else
      ioHead->status = 0;  // Transfer completed successfully
  }
  *ioCtrl = 0; // Stop the transfer
  ioHead = ioHead->next; // Remove the request from the list
  startIO();  // Start a new transfer
}
```

1. zadatak, prvi kolokvijum, mart 2019.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima kontrolera jednog ulaznog uređaja:

```
typedef unsigned long REG;
REG* ioCtrl =...;     // control register
REG* ioStatus =...;   // status register
REG* ioData =...;     // data register
REG* ioBlock =...;    // block address register
const unsigned int BLOCK_SIZE = ...;
const REG START_READING = ...;
```

Sa ovog ulaznog uređaja se jednom operacijom prenosi čitav blok podataka veličine `BLOCK_SIZE`. Adresa bloka na uređaju koji treba pročitati zadaje se upisom u registar na adresi `ioBlock`. Da bi se uređaju zahtevao prenos jednog bloka podataka, potrebno je najpre zadati adresu (broj) traženog bloka na uređaju, a potom u upravljački registar upisati vrednost predstavljenu simboličkom konstantom `START_READING`. Kontroler uređaja poseduje interni memorijski modul koji služi za prihvatanje celog pročitanog bloka podataka (bafer). Kada završi dohvatanje traženog bloka u ovaj svoj interni bafer, kontroler uređaja postavlja bit u razredu 0 svog statusnog registra na 1; ovaj signal *ne* generiše prekid procesoru. U slučaju greške, bit 1 postavlja takođe na 1. Kada je pročitan blok spreman u baferu, može se pročitati jedna po jedna reč tog bafera sukcesivnim čitanjem istog registra za podatke na adresi `ioData` (ovo obezbeđuje interni hardver kontrolera: svako naredno čitanje sa ove adrese inkrementira interni adresni registar za adresiranje unutar bafera; upis `START_READING` u upravljački registar zapravo resetuje ovaj adresni registar). Zbog toga se sukcesivne reči mogu čitati sa ove adrese bez ikakvog čekanja (bez sinhronizacije), u sukcesivnim ciklusima čitanja.

Zahtevi za ulaznim operacijama na ovom uređaju vezani su u jednostruko ulančanu listu. Zahtev ima sledeću strukturu:

```
struct IORequest {
  REG* buffer; // Data buffer (data block) in memory
  REG block;   // The block number (address)
  int status;  // Status of operation
  IORequest* next; // Next in the list
};
```

Na prvi zahtev u listi pokazuje globalni pokazivač `ioHead`, a na poslednji `ioTail`. Operacijom `transfer()` kernel stavlja jedan zahtev u ovu listu i po potrebi pokreće transfer na uređaju. Kada se završi prenos zadat jednim zahtevom, potrebno je u polje `status` date strukture preneti status završene operacije (0 – ispravno završeno do kraja, -1 – greška) i pokrenuti prenos za sledeći zapis u listi, a zahtev izbaciti iz liste. Napisati kod operacije `void transfer (IORequest* request)`.

Rešenje

```
void performIO () {
  while (ioHead!=0) {

    IORequest* ioPending = ioHead; // Take the first request,
    ioHead = ioHead->next; // remove it from the list,
    if (ioHead==0) ioTail = 0;

    *ioBlock = ioPending->block; // and read it from I/O
    *ioCtrl = START_READING;

    while (*ioStatus&1==0);  // Wait for completion

    if (*ioStatus&2) // Error in I/O
      ioPending->status = -1;
    else {
      ioPending->status = 0;
      for (int i=0; i<BLOCK_SIZE; i++)
        ioPending->buffer[i] = *ioData;
    }
  }
}

void transfer (IORequest* req) {
  req->next = 0;
  if (!ioHead) {
    ioHead = ioTail = req;
    performIO();
  } else
    ioTail = ioTail->next = req;
}
```

1. zadatak, prvi kolokvijum, maj 2019.

Na jedan isti kontroler vezana su dva ulazna uređaja. Kontroler može da obavlja uporedne prenose sa ova dva uređaja preko dva logička „kanala“. Date su deklaracije pokazivača preko kojih se može pristupiti registrima ovog kontrolera:

```
typedef volatile unsigned int REG;
REG* ioCtrl =...;   // control register
REG* ioStatus =...; // status register
REG* ioData =...;   // two data registers
```

U (samo jednom) upravljačkom registru dva najniža bita su biti *Start* kojim se pokreće prenos na prvom, odnosno drugom kanalu. Upravljački registar vezan je tako da se može i čitati. U (samo jednom) statusnom registru dva najniža bita su biti spremnosti (*Ready*), a dva bita do njih su biti greške (*Error*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Kada je uređaj na jednom kanalu obavio prenos jednog podatka i spreman je za sledeći, odgovarajući bit *Ready* se postavlja na 1, a kontroler *ne* generiše prekid. U slučaju greške u prenosu, uređaj postavlja *Ready* bit kao i u slučaju završetka prenosa jednog podatka. Na dve susedne adrese počev od adrese `ioData` nalaze se dva registra za podatke, za dva kanala.

Zahtevi za ulaznim operacijama prenosa blokova podataka vezani su u jednostruko ulančanu listu. Zahtev se može opslužiti na bilo kom slobodnom kanalu. Zahtev ima sledeću strukturu:

```
struct IORequest {
  REG* buffer; // Data buffer (block)
  unsigned int size; // Buffer (blok) size
  int status; // Status of operation
```

```
  IORequest* next; // Next in the list
};
```

Polja u ovoj strukturi mogu se koristiti kao promenljive tokom prenosa (ne mora se očuvati njihova početna vrednost nakon prenosa). Na prvi zahtev u listi pokazuje globalni pokazivač `ioHead`. Kada u praznu listu kernel stavi prvi zahtev ili nekoliko zahteva odjednom, pozvaće operaciju `transfer()` koja treba da pokrene prenos za prvi zahtev (ili prva dva). Kada se završi prenos zadat jednim zahtevom, potrebno je u polje `status` date strukture preneti status završene operacije (0 – ispravno završeno do kraja, -1 – greška) i pokrenuti prenos na kanalu koji je završio prenos za sledeći zapis u listi. Ako zahteva u listi više nema, ne treba uraditi više ništa (kada bude stavljao novi zahtev u listu, kernel će proveriti i videti da je ona bila prazna, pa pozvati ponovo operaciju `transfer()` itd.). Zahtev koji je opslužen ne treba više da bude u listi (ali ne treba dealocirati/brisati samu strukturu zahteva).

Potrebno je napisati kod operacije `transfer()`, pri čemu prenos treba vršiti tehnikom programiranog ulaza-izlaza prozivanjem.

```
void transfer ();
```

Rešenje

```
static IORequest* pending[2] = {0,0}; // Pending requests for two channels

void startIO (int i) { // Helper: start a new transfer with channel i
  if (ioHead==0) {
    *ioCtrl &= ~(1<<i); // Stop channel i
    pending[i] = 0;
    return;
  }
  pending[i] = iohead;
  ioHead = ioHead->next; // Remove the request from the list
  pending[i]->next = 0;
  // Start I/O with channel i:
  *ioCtrl |= (1<<i);
  return;
}

void handleIO (int i) { // Helper: handle I/O with channel i
  if (!((*ioStatus)&(1<<i))) return;
  if ((*ioStatus)&(1<<(2+i)))
    pending[i]->status = -1;  // Error in I/O
  else {  // Transfer the next data item
    *(pending[i]->buffer)++ = ioData[i];
    if (--pending[i]->size)
      return;
    else {
      pending[i]->status = 0;  // Transfer completed successfully
  }
  startIO(i); // Initiate the next transfer with this channel
}

void transfer () {
  if (!ioHead) return;
  startIO(0);
  startIO(1);
  while (pending[0] || pending[1]) {
    if (pending[0]) handleIO(0);
    if (pending[1]) handleIO(1);
  }
}
```

1. zadatak, prvi kolokvijum, jun 2019.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva ulazno/izlazna uređaja:

```
typedef volatile unsigned int REG;
REG* io1Ctrl =...;   // Device 1 control register
REG* io1Status =...; // Device 1 status register
REG* io1Data =...;   // Device 1 data register
REG* io2Ctrl =...;   // Device 2 control register
REG* io2Status =...; // Device 2 status register
REG* io2Data =...;   // Device 2 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće uređaj, a u statusnim registrima najniži bit je bit spremnosti (*Ready*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Potrebno je napisati funkciju `transfer` koja najpre vrši ulaz bloka podataka zadate veličine sa prvog uređaja korišćenjem tehnike prozivanja (*polling*), a potom izlaz tog istog učitanog bloka podataka na drugi uređaj korišćenjem prekida, i vraća kontrolu pozivaocu tek kada se oba prenosa završe.

Rešenje

```
static REG* ioPtr = 0;
static int ioCount = 0;
static int ioCompleted = 0;

void transfer (int count) {
  REG* buffer = new REG[count];

  // I/O 1
  ioPtr = buffer;
  ioCount = count;
  *io1Ctrl = 1; // Start I/O 1
  while (ioCount>0) {
    while (!(*io1Status&1)); // busy wait
    *ioPtr++ = *io1Data;
    ioCount--;
  }
  *io1Ctrl = 0; // Stop I/O 1

  // I/O 2:
  ioPtr = buffer;
  ioCount = count;
  ioCompleted = 0;
  *io2Ctrl = 1; // Start I/O 2

  // Wait for I/O 2 completion:
  while (!ioCompleted);

  delete [] buffer;
}

interrupt void io2Interrupt() {
  *io2Data = *ioPtr++;
  if (--ioCount == 0) {
    ioCompleted = 1;
    *io2Ctrl = 0; // Stop I/O 2
  }
}
```

1. zadatak, prvi kolokvijum, mart 2018.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva ulazna uređaja:

```
typedef unsigned REG;
REG* io1Ctrl =...;   // Device 1 control register
REG* io1Data =...;   // Device 1 data register
REG* io1Stat =...;   // Device 1 status register
REG* io2Ctrl =...;   // Device 2 control register
REG* io2Data =...;   // Device 2 data register
REG* io2Stat =...;   // Device 2 status register
REG* timer =...;     // Timer
```

U upravljačkim registrima bit 0 je bit *Start*. Spremnost novog ulaznog podatka prvi uređaj ne signalizira nikakvim signalom, ali je sigurno da je on spreman najkasnije 50 ms nakon preuzimanja prethodnog podatka (čitanja podatka iz registra), odnosno od postavljanja bita *Start*, ako se radi o prvom podatku. Spremnost novog ulaznog podatka drugi uređaj signalizira bitom spremnosti u razredu 0 svog statusnog registra, čije postavljanje ne generiše prekid procesoru.

Na magistralu računara vezan je i registar posebnog uređaja, vremenskog brojača (tajmera). Upisom celobrojne vrednosti n u ovaj registar vremenski brojač počinje merenje vremena od n milisekundi, nakon isteka tog vremena, generiše prekid procesoru.

Potrebno je napisati kod, uključujući i prekidnu rutinu od vremenskog brojača, koji vrši prenos po jednog bloka podataka sa svakog od dva uređaja uporedo. Prenos se obavlja pozivom sledeće funkcije iz koje se vraća kada su oba prenosa završena:

```
void transfer (unsigned* blk1, int count1, unsigned* blk2, int count2);
```

Rešenje

```
static unsigned *io1Ptr = 0, *io2Ptr = 0;
static int io1Count = 0, io2Count = 0;
static const unsigned timeout = 50;

void transfer (unsigned* blk1, int count1, unsigned* blk2, int count2) {
  // I/O 1:
  io1Ptr = blk1;
  io1Count = count1;
  *io1Ctrl = 1; // Start I/O 1
  *timer = timeout; // Start timer

  // I/O 2:
  io2Ptr = blk2;
  *io2Ctrl = 1; // Start I/O 2
  for (io2Count=count2; io2Count>0; io2Count--) {
    while (*io2Stat&1 == 0); // Busy-wait for I/O 2 to be ready
    *io2Ptr++ = *io2Data;
  }
  *io2Ctrl = 0; // Stop I/O 2

  // Busy wait for I/O 1 completion:
  while (io1Count);
}

interrupt void timerInterrupt () {
  *io1Ptr++ = *io1Data;
  if (--io1Count)
    *timer = timeout; // Restart timer
  else
```

```
    *io1Ctrl = 0; // Stop I/O 1
}
```

1. zadatak, prvi kolokvijum, april 2018.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima četiri uređaja; prva dva uređaja su ulazni, a druga dva su izlazni:

```
typedef unsigned int REG;
REG* io1Ctrl =...;   // Device 1 control register
REG* io1Status =...; // Device 1 status register
REG* io1Data =...;   // Device 1 data register
REG* io2Ctrl =...;   // Device 2 control register
REG* io2Status =...; // Device 2 status register
REG* io2Data =...;   // Device 2 data register
REG* io3Ctrl =...;   // Device 3 control register
REG* io3Status =...; // Device 3 status register
REG* io3Data =...;   // Device 3 data register
REG* io4Ctrl =...;   // Device 4 control register
REG* io4Status =...; // Device 4 status register
REG* io4Data =...;   // Device 4 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće uređaj, a u statusnim registrima najniži bit je bit spremnosti (*Ready*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Samo uređaj broj 2 generiše prekid kada je spreman.

Potrebno je napisati proceduru `transfer` i odgovarajuću prekidnu rutinu za prekid sa uređaja broj 2 koja učitava po jednu reč sa uređaja 1 i tu reč odmah prebacuje na uređaj 3, odnosno učitava po jednu reč sa uređaja 2 i tu reč odmah prebacuje na uređaj 4. Ova dva prenosa se obavljaju uporedo, sve dok se sa bilo kog ulaznog uređaja ne učita vrednost 0. Prenose sa uređajima 1, 3 i 4 vršiti programiranim ulazom/izlazom sa prozivanjem (*polling*), a sa uređajem 2 korišćenjem prekida. Smatrati da je periferijski uređaj 4 veoma brz, tj. da vrlo brzo (reda vremena izvršavanja nekoliko desetina instrukcija procesora) od prijema podatka biva spreman za prijem novog podatka.

Rešenje

```
const REG ESC = 0;
bool complete = false;

void transfer () {
  REG data = ESC;

  *io1Ctrl = 1;
  *io2Ctrl = 1;
  *io3Ctrl = 1;
  *io4Ctrl = 1;

  while (!complete) {
    while (*io1Status&1==0);
    data = *io1Data;

  if (data==ESC) break;

    while (*io3Status&1==0);
    *io3Data = data;
  }

  complete = true;
  *io1Ctrl = 0;
```

```
  *io2Ctrl = 0;
  *io3Ctrl = 0;
  *io4Ctrl = 0;
}

interrupt void device2 () {
  if (complete) return;
  REG data = *io2Data;
  if (data!=ESC) {
    while (*io4Status&1==0);
    *io4Data = data;
  } else {
    complete = true;
    *io1Ctrl = 0;
    *io2Ctrl = 0;
    *io3Ctrl = 0;
    *io4Ctrl = 0;
  }
}
```

1. zadatak, prvi kolokvijum, jun 2018.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima tri uređaja; prvi uređaja je ulazni, a druga dva su izlazni:

```
typedef unsigned int REG;
REG* io1Ctrl =...;   // Device 1 control register
REG* io1Status =...; // Device 1 status register
REG* io1Data =...;   // Device 1 data register
REG* io2Ctrl =...;   // Device 2 control register
REG* io2Status =...; // Device 2 status register
REG* io2Data =...;   // Device 2 data register
REG* io3Ctrl =...;   // Device 3 control register
REG* io3Status =...; // Device 3 status register
REG* io3Data =...;   // Device 3 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće uređaj, a u statusnim registrima najniži bit je bit spremnosti (*Ready*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Potrebno je napisati potprogram `transfer` koji učitava po jednu reo sa ulaznog uređaja i odmah tu učitanu reč šalje na bilo koji od ova dva izlazna uređaja, koji god od njih je pre spreman. Ovaj prenos se obavlja sve dok se sa ulaznog uređaja ne učita vrednost 0. Sve prenose vršiti programiranim ulazom/izlazom sa prozivanjem (*polling*).

Rešenje

```
const REG ESC = 0;

void transfer () {
  REG data = ESC;

  *io1Ctrl = 1;
  *io2Ctrl = 1;
  *io3Ctrl = 1;

  while (1) {
    while (*io1Status==0);
    data = *io1Data;

  if (data==ESC) break;
```

```
    ind done = 0;
    while (!done) {
     if (*io2Status!=0)
       *io2Data = data, done = 1;
     else
     if (*io3Status!=0)
       *io3Data = data, done = 1;
    }
  }

  *io1Ctrl = 0;
  *io2Ctrl = 0;
  *io3Ctrl = 0;
}
```

1. zadatak, prvi kolokvijum, mart 2017.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima kontrolera jednog izlaznog uređaja:

```
typedef unsigned int REG;
REG* ioCtrl =...;     // control register
REG* ioStatus =...;   // status register
REG* ioData =...;     // data register
const unsigned int BLOCK_SIZE = ...;
const REG START_SENDING = ..., END_SENDING = ...;
```

Na ovaj izlazni uređaj se jednom operacijom prenosi čitav blok podataka veličine `BLOCK_SIZE`. Da bi se uređaju zadao prenos jednog bloka podataka, potrebno je najpre u upravljački registar upisati vrednost predstavljenu simboličkom konstantom `START_SENDING`. Potom treba jednu po jednu rem upisivati na istu adresu registra za podatke (`ioData`). Kontroler uređaja poseduje memorijski modul koji služi za prihvatanje celog bloka podataka (bafer), tako da se reči upisane redom na adresu registra za podatke upisuju redom u ovaj bafer (ovo obezbeđuje interni hardver kontrolera: svaki naredni upis na ovu adresu inkrementira interni adresni registar za adresiranje unutar bafera; upis `START_SENDING` u upravljački registar zapravo resetuje ovaj adresni registar). Zbog toga se sukcesivne reči mogu upisivati na ovu adresu bez ikakvog čekanja (bez sinhronizacije), u sukcesivnim ciklusima upisa. Kada se završi upis celog bloka podataka, potrebno je u upravljački registar upisati vrednost predstavljenu simboličkom konstantom `END_SENDING`. Ovim se kontroleru nalaže da započne izlaznu operaciju zadatog bloka podataka. Kada završi izlaznu operaciju prenosa celog bloka podataka, kontroler periferije postavlja bit u razredu 0 statusnog registra, ali ne generiše prekid procesoru. U statusnom registru bit 1 je tada bit greške u prenosu (*Error*).

Zahtevi za izlaznim operacijama na ovom uređaju vezani su u jednostruko ulančanu listu. Zahtev ima sledeću strukturu:

```
struct IORequest {
  REG* buffer; // Data buffer (data block)
  int status;  // Status of operation
  IORequest* next; // Next in the list
};
```

Na prvi zahtev u listi pokazuje globalni pokazivač `ioHead`, a na poslednji `ioTail`. Operacijom `transfer()` kernel stavlja jedan zahtev u ovu listu i po potrebi pokreće transfer na uređaju. Kada se završi prenos zadat jednim zahtevom, potrebno je u polje `status` date strukture preneti status završene operacije (0 – ispravno završeno do kraja, -1 – greška) i pokrenuti prenos za sledeći zapis u listi, a zahtev izbaciti iz liste.

Napisati kod operacije `void transfer (IORequest* request)`.

Rešenje

```
void performIO () {
  while (ioHead!=0) {
```

```
    IORequest* ioPending = ioHead; // Take the first request,
    ioHead = ioHead->next; // remove it from the list,
    if (ioHead==0) ioTail = 0;


 *ioCtrl = START_SENDING; // and send it to I/O
    for (int i=0; i<BLOCK_SIZE; i++)
      *ioData = ioPending->buffer[i];
    *ioCtrl = END_SENDING;

    while (*ioStatus&1 == 0);  // Wait for completion

    if (*ioStatus&2) // Error in I/O
      ioPending->status = -1;
    else
      ioPending->status = 0;
  }
}

void transfer (IORequest* req) {
  req->next = 0;
  if (!ioHead) {
    ioHead = ioTail = req;
    performIO();
  } else
    ioTail = ioTail->next = req;
}
```

1. zadatak, prvi kolokvijum, april 2017.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva izlazna uređaja:

```
typedef unsigned int REG;
REG* io1Ctrl =...;   // Device 1 control register
REG* io1Data =...;   // Device 1 data register
REG* io2Ctrl =...;   // Device 2 control register
REG* io2Data =...;   // Device 2 data register
REG* timer =...;     // Timer
```

U upravljačkim registrima bit 0 je bit *Start*. Prenos svakog pojedinačnog podatka na svaki od ovih uređaja zahteva se upisom podatka u registar podataka tog uređaja. Spremnost registra za podatke za prihvat novog izlaznog podatka prvi uređaj ne signalizira nikakvim signalom, ali je sigurno da je on spreman za novi najkasnije 50 ms nakon zadatog prethodnog zahteva (upisa podatka u registar). Spremnost registra za podatke za prihvat novog izlaznog podatka drugi uređaj signalizira prekidom.

Na magistralu računara vezan je i registar posebnog uređaja, vremenskog brojača. Upisom celobrojne vrednosti n u ovaj registar vremenski brojač počinje merenje vremena od n milisekundi, nakon isteka tog vremena, generiše prekid procesoru.

Potrebno je napisati kod, uključujući i obe prekidne rutine (od drugog uređaja i tajmera), koji vrši prenos po jednog bloka podataka na svaki od dva uređaja uporedo. Prenos se obavlja pozivom sledeće funkcije iz koje se vraća kada su oba prenosa završena:

```
void transfer (unsigned* blk1, int count1, unsigned* blk2, int count2);
```

Rešenje

```
static unsigned *io1Ptr = 0, *io2Ptr = 0;
static int io1Count = 0, io2Count = 0;
static unsigned timeout = 50;
void transfer (unsigned* blk1, int count1, unsigned* blk2, int count2) {
  // I/O 1:
  io1Ptr = blk1;
  io1Count = count1;
  *io1Ctrl = 1; // Start I/O 1
  *io1Data = *io1Ptr++;
  *timer = timeout; // Start timer
  // I/O 2:
  io2Ptr = blk2;
  io2Count = count2;
  *io2Ctrl = 1; // Start I/O 2
  *io2Data = *io2Ptr++;

  // Busy wait for I/O completion:
  while (io1Count || io2Count);
}
interrupt void io2Interrupt() {
  if (--io2Count)
    *io2Data = *io1Ptr++; // New output request
  else
    *io2Ctrl = 0; // Stop I/O 2
}
interrupt void timerInterrupt () {
  if (--io1Count) {
    *io1Data = *io1Ptr++; // New output request
    *timer = timeout; // Restart timer
  } else
    *io1Ctrl = 0; // Stop I/O 1
}
```

1. zadatak, prvi kolokvijum, jun 2017.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima jednog kontrolera mrežne kartice, kao i jednog DMA kontrolera koji je direktno spregnut sa tom mrežnom karticom i može prenositi paket veličine `PKT_SIZE` primljen sa mreže u memoriju:

```
typedef volatile unsigned int REG;
REG* ioNetCtrl =...;    // Network device control register
REG* ioNetStatus =...;  // Network device status register
REG* ioNetData =...;    // Network device data register
REG* dmaCtrl =...;      // DMA control register
REG* dmaStatus =...;    // DMA status register
REG* dmaAddr =...;      // DMA buffer address register
REG* dmaCount =...;     // DMA buffer size register
const int BUF_SIZE = ..., PKT_SIZE = ...;
char buffer[BUF_SIZE][PKT_SIZE];
int bufHead, bufTail;
```

Kada stigne jedan paket sa mreže, mrežna kartica generiše prekid procesoru. Tada treba pokrenuti transfer tog paketa sa mrežne kartice u memoriju, korišćenjem DMA kontrolera. DMA kontroler se pokreće upisom konstante `DMA_START` u njegov upravljači registar. Paket veličine `PKT_SIZE` treba smestiti na mesto na koje ukazuje indeks `bufTail` u kružnom baferu `buffer`. Ako je bafer pun, što se vidi tako da je `bufTail==bufHead`, mrežnoj kartici treba javiti da primljeni paket treba da odbaci (tj. da prijemnoj strani treba da javi da prijem nije uspeo), što

se postiže upisom konstante `PKT_REJECT` u upravljački registar mrežne kartice. Napisati kod prekidnih rutina za prekide sa mrežne kartice i DMA kontrolera.

Rešenje

```
interrupt void intNet () {
  if (bufTail==bufHead) {
    // Buffer full, reject the packet:
    *ioNetCtrl = PKT_REJECT;
    return;
  }
  *dmaAddr = &buffer[bufTail];
  *dmaCount = PKT_SIZE;
  *dmaCtrl = DMA_START;
}


interrupt void intDMA () {
  ++bufTail %= BUF_SIZE;
}
```

1. zadatak, prvi kolokvijum, mart 2016.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva DMA kontrolera:

```
typedef unsigned int REG;
REG* dma1Ctrl =...;     // DMA1 control register
REG* dma1Status =...;   // DMA1 status register
REG* dma2Ctrl =...;     // DMA2 control register
REG* dma2Status =...;   // DMA2 status register
```

Inicijalno, svaki DMA kontroler je u *pasivnom* stanju. Prvi upis u upravljački registar u pasivnom stanju tumači se kao upis adrese bloka podataka za prenos. Naredni upis u isti registar tumači se kao upis veličine bloka podataka za prenos. Naredni upis nenulte vrednosti u ovaj registar tumači se kao pokretanje prenosa, čime se kontroler prebacuje u *aktivno* stanje. Kada završi prenos, bilo ispravno ili u slu čaju greške, DMA kontroler zaustavlja prenos, generiše prekid procesoru i prebacuje se ponovo u pasivno stanje. U statusnom registru najniži bit je bit završetka transfera (*TransferComplete*), a bit do njega bit greške (*Error*). Završetak prenosa sa bilo kog DMA kontrolera postavlja njegov bit *TransferComplete* i generiše isti zahtev za prekid procesoru (signali završetka operacije sa dva DMA kontrolera vezani su na ulazni zahtev za prekid preko OR logičkog kola). Zahtevi za ulaznim operacijama na nekom uređaju sa kog se prenos blokova vrši preko bilo kog od ova DMA kontrolera vezani su u jednostruko ulančanu listu. Zahtev ima sledeću strukturu:

```
struct IORequest {
  REG* buffer; // Data buffer (block)
  unsigned int size; // Buffer (blok) size
  int status; // Status of operation
  IORequest* next; // Next in the list
};
```

Na prvi zahtev u listi pokazuje globali pokazivač `ioHead`, a na poslednji `ioTail`. Operacijom `transfer()` kernel stavlja jedan zahtev u ovu listu i po potrebi pokreće transfer na bilo kom trenutno slobodnom DMA kontroleru. Kada se završi prenos zadat jednim zahtevom na jednom DMA kontroleru, potrebno je u polje status date strukture preneti status završene operacije (0 – ispravno završeno do kraja, -1 – greška) i pokrenuti prenos za sledeći zapis u listi na tom DMA kontroleru, a zahtev izbaciti iz liste. Obratiti pažnju na to da oba DMA kontrolera mogu završiti prenos i generisati prekid istovremeno. Potrebno je napisati kod operacije `transfer()`, zajedno sa odgovarajućom prekidnom rutinom `dmaInterrupt()` za prekid od DMA kontrolera.

```
void transfer (IORequest* request);
interrupt void dmaInterrupt ();
```

Rešenje

```
IORequest *dma1Pending = 0, *dma2Pending = 0; // Currently pending requests

void startDMA1 () {
  if (ioHead!=0 && dma1Pending==0) {
    dma1Pending = ioHead; // Take the first request,
    ioHead = ioHead->next; // remove it from the list
    if (ioHead==0) ioTail = 0;
    *dma1Ctrl = dma1Pending->buffer; // and assign it to DMA1
    *dma1Ctrl = dma1Pending->size;
    *dma1Ctrl = 1; // Start I/O
  }
}

void startDMA2 () {
  if (ioHead!=0 && dma2Pending==0) {
    dma2Pending = ioHead; // Take the first request,
    ioHead = ioHead->next; // remove it from the list
    if (ioHead==0) ioTail = 0;
    *dma2Ctrl = dma2Pending->buffer; // and assign it to DMA2
    *dma2Ctrl = dma2Pending->size;
    *dma2Ctrl = 1; // Start I/O
  }
}

void transfer (IORequest* req) {
  req->next = 0;
  if (!ioHead) {
    ioHead = ioTail = req;
    startDMA1();
    startDMA2();
  } else
    ioTail = ioTail->next = req;
}

interrupt void dmaInterrupt () {
  if (*dma1Status&1) { // DMA1 transfer complete
    if (dma1Pending==0) return; // Exception
    if (*dma1Status&2) // Error in I/O
      dma1Pending->status = -1;
    else
      dma1Pending->status = 0;
    dma1Pending = 0;
    startDMA1();
  }
  if (*dma2Status&1) { // DMA2 transfer complete
    if (dma2Pending==0) return; // Exception
    if (*dma2Status&2) // Error in I/O
      dma2Pending->status = -1;
    else
      dma2Pending->status = 0;
    dma2Pending = 0;
    startDMA2();
  }
}
```

1. zadatak, prvi kolokvijum, maj 2016.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima kontrolera jednog izlaznog uređaja:

```
typedef unsigned int REG;
REG* ioCtrl =...;     // control register
REG* ioStatus =...;   // status register
REG* ioData =...;     // data register
const unsigned int BLOCK_SIZE = ...;
const REG START_SENDING = ..., END_SENDING = ...;
```

Na ovaj izlazni uređaj se jednom operacijom prenosi čitav blok podataka veličine `BLOCK_SIZE`. Da bi se uređaju zadao prenos jednog bloka podataka, potrebno je najpre u upravljački registar upisati vrednost predstavljenu simboličkom konstantom `START_SENDING`. Potom treba jednu po jednu rem upisivati na istu adresu registra za podatke (`ioData`). Kontroler uređaja poseduje memorijski modul koji služi za prihvatanje celog bloka podataka (bafer), tako da se reči upisane redom na adresu registra za podatke upisuju redom u ovaj bafer (ovo obezbeđuje interni hardver kontrolera: svaki naredni upis na ovu adresu inkrementira interni adresni registar za adresiranje unutar bafera; upis `START_SENDING` u upravljački registar zapravo resetuje ovaj adresni registar). Zbog toga se sukcesivne reči mogu upisivati na ovu adresu bez ikakvog čekanja (bez sinhronizacije), u sukcesivnim ciklusima upisa. Kada se završi upis celog bloka podataka, potrebno je u upravljački registar upisati vrednost predstavljenu simboličkom konstantom `END_SENDING`. Ovim se kontroleru nalaže da započne izlaznu operaciju zadatog bloka podataka.

Kada završi izlaznu operaciju prenosa celog bloka podataka, kontroler periferije generiše prekid procesoru. U statusnom registru najniži bit je tada bit greške u prenosu (*Error*).

Zahtevi za izlaznim operacijama na ovom uređaju vezani su u jednostruko ulančanu listu. Zahtev ima sledeću strukturu:

```
struct IORequest {
  REG* buffer; // Data buffer (data block)
  int status;  // Status of operation
  IORequest* next; // Next in the list
};
```

Na prvi zahtev u listi pokazuje globalni pokazivač `ioHead`, a na poslednji `ioTail`. Operacijom `transfer()` kernel stavlja jedan zahtev u ovu listu i po potrebi pokreće transfer na uređaju. Kada se završi prenos zadat jednim zahtevom, potrebno je u polje status date strukture preneti status završene operacije (0 – ispravno završeno do kraja, -1 – greška) i pokrenuti prenos za sledeći zapis u listi, a zahtev izbaciti iz liste. Potrebno je napisati kod operacije `transfer()`, zajedno sa odgovarajućom prekidnom rutinom `ioInterrupt()` za prekid od kontrolera uređaja.

```
void transfer (IORequest* request);
interrupt void ioInterrupt ();
```

Rešenje

```
IORequest* ioPending = 0; // Currently pending request

void performIO () {
  if (ioHead!=0 && ioPending==0) {
    ioPending = ioHead; // Take the first request,
    ioHead = ioHead->next; // remove it from the list,
    if (ioHead==0) ioTail = 0;
    *ioCtrl = START_SENDING; // and send it to I/O
    for (int i=0; i<BLOCK_SIZE; i++)
      *ioData = ioPending->buffer[i];
    *ioCtrl = END_SENDING;
  }
}

void transfer (IORequest* req) {
  req->next = 0;
```

```
  if (!ioHead) {
    ioHead = ioTail = req;
    performIO();
  } else
    ioTail = ioTail->next = req;
}

interrupt void ioInterrupt () {
  if (ioPending==0) return; // Exception
  if (*ioStatus&1) // Error in I/O
    ioPending->status = -1;
  else
    ioPending->status = 0;
  ioPending = 0;
  performIO();
}
```

1. zadatak, prvi kolokvijum, septembar 2016.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima jednog kontrolera izlaznog uređaja:

```
typedef unsigned int REG;
REG* ioCtrl =...;    // Output device control register
REG* ioData =...;    // Output device data register
const int BUFSIZE = ..., DELAY = ...;
REG buffer[BUFSIZE];
```

U upravljačkom registru najniži bit je bit *Start* kojim se pokreće uređaj. Statusni registar nije programski dostupan, niti kontroler signalizira spremnost na bilo koji način (npr. prekidom), ali je poznato da će, nakon zadavanja prenosa jednog podatka, kontroler sigurno završiti operaciju i biti spreman za novu nakon `DELAY` vremena.

Na jeziku C napisati kod operacije `transfer()` koja vrši prenos bloka podataka iz bafera `buffer` na dati izlazni uređaj. Ova operacija poziva se iz konteksta kernel niti. U kernelu je dostupna operacija `sleep(int)` koja uspavljuje (suspenduje) pozivajuću nit na zadato vreme. Sve eventualne greške ignorisati.

Rešenje

```
void transfer () {
  // Start the controller:
  *ioCtrl = 1;

  // Perform the output transfer:
  for (int i=0; i<BUFSIZE; i++) {
      sleep(DELAY); // Sleep-wait
      *ioData = buffer[i]; // Write data
  }

  // Stop the controller:
  *ioCtrl = 0;
}
```

1. zadatak, prvi kolokvijum, mart 2015.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima kontrolera jednog ulazno-izlaznog uređaja:

```
typedef unsigned int REG;
REG* ioCtrl =...;   // control register
REG* ioStatus =...; // status register
REG* ioData =...;   // data register
```

U upravljačkom registru najniži bit je bit *Start* kojim se pokreće prenos. a bit do njega definiše smer prenosa podataka (0-ulaz. 1-izlaz). U statusnom registru najniži bit je bit spremnosti (*Ready*), a bit do njega bit greške (*Error*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`). U slučaju greške u prenosu. uređaj generiše isti prekid kao i u slučaju spremnosti za prenos novog podatka.

Zahtevi za ulaznim i izlaznim operacijama prenosa blokova podataka na ovom ure đaju vezani su u jednostruko ulančanu listu. Zahtev ima sledeću strukturu:

```
struct IORequest {
  REG* buffer; // Data buffer (block)
  unsigned int size; // Buffer (blok) size
  int dir; // Transfer direction: 0-input, 1-output
  int status; // Status of operation
  IORequest* next; // Next in the list
};
```

Na prvi zahtev u listi pokazuje globali pokazivač `ioHead`. Kada u praznu listu kernel stavi prvi zahtev, pozvaće operaciju `transfer()` koja treba da pokrene prenos za taj prvi zahtev. Kada se završi prenos zadat jednim zahtevom. potrebno je u polje status date strukture preneti status završene operacije (0 – ispravno završeno do kraja. -1 – greška). izbaciti obrađeni zahtev iz liste i pokrenuti prenos za sledeći zapis u listi. Ako zahteva u listi više nema, ne treba uraditi više ništa (kada bude stavljao novi zahtev u listu, kernel će proveriti i videti da je ona bila prazna, pa pozvati ponovo operaciju `transfer()` itd.)

Potrebno je napisati kod operacije `transfer()`. zajedno sa odgovarajućom prekidnom rutinom `ioInterrupt()` za prekid od uređaja, pri čemu prenos treba vršiti tehnikom programiranog ulaza-izlaza uz korišćenje prekida.

```
void transfer ();
interrupt void ioInterrupt ();
```

Rešenje

```
static int dir = 0;   // current transfer direction
static REG* ptr = 0;  // pointer to current data item
static int count = 0; // counter

void startIO () { // Helper: start a new transfer
  ptr = ioHead->buffer;
  count = ioHead->size;
  dir = ioHead->dir;
  *ioCtrl = 1 | (dir<<1); // Start I/O
}

void transfer () {
  if (ioHead==0) return; // Exception - no requests
  startIO();
}

interrupt void ioInterrupt () {
  if (*ioStatus&2)
    ioHead->status = -1;  // Error in I/O
  else {  // Transfer the next data item
    if (dir)
      *ioData = *ptr++;
    else
      *ptr++ = *ioData;
    if (--count)
      return;
    else
      ioHead->status = 0;  // Transfer completed successfully
```

```
  }

  ioHead = ioHead->next; // Remove the request from the list
  if (ioHead==0)
    *ioCtrl = 0; // No more requests
  else
    startIO();  // Start a new transfer
}
```

1. zadatak, prvi kolokvijum, maj 2015.

Na jedan isti ulazno-izlazni kontroler vezana su dva ulazno-izlazna uređaja. Kontroler može da obavlja uporedne prenose sa ova dva uređaja preko dva logička „kanala“. Date su deklaracije pokazivača preko kojih se može pristupiti registrima ovog kontrolera:

```
typedef unsigned int REG;
REG* ioCtrl =...;   // control register
REG* ioStatus =...; // status register
REG* ioData =...;   // two data registers
```

U (samo jednom) upravljačkom registru dva najniža bita su biti *Start* kojim se pokreće prenos na prvom, odnosno drugom kanalu, a dva naredna bita definišu smer prenosa podataka na ova dva kanala (0-ulaz, 1-izlaz). Upravljački registar vezan je tako da se može i čitati. U (samo jednom) statusnom registru dva najniža bita su biti spremnosti (*Ready*), a dva bita do njih su biti greške (*Error*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Kada je uređaj na jednom kanalu obavio prenos jednog podatka i spreman je za sledeći, odgovarajući bit *Ready* se postavlja na 1, a kontroler generiše prekid (isti prekid za oba kanala). U slučaju greške u prenosu, uređaj generiše isti prekid kao i u slučaju završetka prenosa jednog podatka. Na dve susedne adrese počev od adrese `ioData` nalaze se dva registra za podatke, za dva kanala.

Zahtevi za ulaznim i izlaznim operacijama prenosa blokova podataka vezani su u jednostruko ulančanu listu. Zahtev se može opslužiti na bilo kom slobodnom kanalu. Zahtev ima sledeću strukturu:

```
struct IORequest {
  REG* buffer; // Data buffer (block)
  unsigned int size; // Buffer (blok) size
  int dir; // Transfer direction: 0-input, 1-output
  int status; // Status of operation
  IORequest* next; // Next in the list
};
```

Polja u ovoj strukturi mogu se koristiti kao promenljive tokom prenosa (ne mora se očuvati njihova početna vrednost nakon prenosa). Na prvi zahtev u listi pokazuje globali pokazivač ioHead. Kada u praznu listu kernel stavi prvi zahtev ili nekoliko zahteva odjednom, pozvaće operaciju `transfer()` koja treba da pokrene prenos za prvi zahtev (ili prva dva). Kada se završi prenos zadat jednim zahtevom, potrebno je u polje `status` date strukture preneti status završene operacije (0 – ispravno završeno do kraja, -1 – greška) i pokrenuti prenos na kanalu koji je završio prenos za sledeći zapis u listi. Ako zahteva u listi više nema, ne treba uraditi više ništa (kada bude stavljao novi zahtev u listu, kernel će proveriti i videti da je ona bila prazna, pa pozvati ponovo operaciju `transfer()` itd.). Zahtev koji je opslužen ne treba više da bude u listi (ali ne treba brisati samu strukturu zahteva).

Potrebno je napisati kod operacije `transfer()`, zajedno sa odgovarajućom prekidnom rutinom `ioInterrupt()` za prekid od kontrolera, pri čemu prenos treba vršiti tehnikom programiranog ulaza-izlaza uz korišćenje prekida.

```
void transfer ();
interrupt void ioInterrupt ();
```

Rešenje

```
static IORequest* pending[2] = {0,0}; // Pending requests for two channels

void startIO (int i) { // Helper: start a new transfer with channel i
```

```
  if (ioHead==0) {
    *ioCtrl &= ~(1<<i); // Stop channel i
    return;
  }
  pending[i] = iohead;
  ioHead = ioHead->next; // Remove the request from the list
  // Start I/O with channel i:
  if (pending[i]->dir)
    *ioCtrl |= 1<<(2+i);
  else
    *ioCtrl &= ~(1<<(2+i));
  *ioCtrl |= (1<<i);
}

void handleInterrupt (int i) { // Helper: handle interrupt from channel i
  if ((*ioStatus)&(1<<(2+i)))
    pending[i]->status = -1;  // Error in I/O
  else {  // Transfer the next data item
    if (pending[i]->dir)
      ioData[i] = *(pending[i]->buffer)++;
    else
      *(pending[i]->buffer)++ = ioData[i];
    if (--pending[i]->size)
      return;
    else
      pending[i]->status = 0;  // Transfer completed successfully
  }
  startIO(i); // Initiate the next transfer with this channel
}

void transfer () {
  startIO(0);
  startIO(1);
}

interrupt void ioInterrupt () {
  if (*ioStatus & 1)
    handleInterrupt(0);
  else
    handleInterrupt(1);
}
```

1. zadatak, prvi kolokvijum, septembar 2015.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima jednog kontrolera ulaznog uređaja i jednog DMA kontrolera koji vrši prenos na jedan izlazni uređaj:

```
typedef unsigned int REG;
REG* ioCtrl =...;    // Input device control register
REG* ioStatus =...;  // Input device status register
REG* ioData =...;    // Input device data register
REG* dmaCtrl =...;   // DMA control register
REG* dmaStatus =...; // DMA status register
REG* dmaAddr =...;   // DMA buffer address register
REG* dmaCount =...;  // DMA buffer size register
const int BUFSIZE;
REG buffer[2][BUFSIZE];
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće uređaj, a u statusnim registrima najniži bit je bit spremnosti (*Ready*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`).

Na jeziku C napisati kod operacije `transfer()` i prekidne rutine `dmaInterrupt()` za prekid sa DMA koje vrše prenos blokova podataka sa datog ulaznog uređaja tehnikom prozivanja (*polling*) na dati izlazni uređaj korišćenjem DMA na sledeći način. Data su dva memorijska bafera za smeštanje blokova podataka, svaki veličine `BUFSIZE` (nizovi `buffer[2]`). Prvi blok podataka treba učitavati u prvi niz (`buffer[0]`), reč po reč, tehnikom prozivanja. Kada se ovaj bafer napuni, treba pokrenuti izlaznu operaciju za podatke iz ovog bafera preko DMA kontrolera, a ulaznu operaciju nastaviti uporednim učitavanjem u drugi niz (`buffer[1]`), čime ova dva bafera zamenjuju uloge. Kada se i ovo učitavanje završi, treba ponovo učitavati u prvi, a izlaz vršiti iz drugog bafera, odnosno zameniti uloge bafera itd. u nedogled. Pretpostaviti da izlazni prenos jednog celog bloka podataka preko DMA traje znatno kraće nego ulazni prenos jednog bloka, tako da će izlaz iz jednog bafera sigurno biti završen pre nego što se uporedni ulaz u drugi bafer završi. Sve eventualne greške ignorisati.

Rešenje

```
void transfer () {
  int curBuffer = 0;
  // Start input controller:
  *ioCtrl = 1;

  while () {

    // Perform input transfer:
    for (int i=0; i<BUFSIZE; i++) {
      while (!(*ioStatus&1)); // Busy wait
      buffer[curBuffer][i] = *ioData; // Read data
    }

    // Start DMA output:
    *dmaAddr = buffer[curBuffer];
    *dmaCount = BUFSIZE;
    *dmaCtrl = 1;
    // Swap the buffers:
    curBuffer = 1-curBuffer;

  }
}

interrupt void dmaInterrupt () {
  *dmaCtrl = 0;
}
```

1. zadatak, prvi kolokvijum, mart 2014.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima jednog DMA kontrolera:

```
typedef unsigned int REG;
REG* dmaCtrl =...;   // DMA control register
REG* dmaStatus =...; // DMA status register
REG* dmaAddress =...;   // DMA block address register
REG* dmaCount =...;   // DMA block size register
```

U upravljačkom registru najniži bit je bit *Start* kojim se pokreće prenos jednog bloka preko DMA, a u statusnom registru najniži bit je bit završetka prenosa (*TransferComplete*), a bit do njega bit greške (*Error*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`).

Zahtevi za ulaznim operacijama na nekom uređaju sa kog se prenos blokova vrši preko ovog DMA kontrolera vezani su u jednostruko ulančanu listu. Zahtev ima sledeću strukturu:

```
struct IORequest {
  REG* buffer; // Data buffer (block)
  unsigned int size; // Buffer (blok) size
  int status; // Status of operation
  IORequest* next; // Next in the list
};
```

Na prvi zahtev u listi pokazuje globalni pokazivač `ioHead`. Kada u praznu listu kernel stavi prvi zahtev, pozvaće operaciju `transfer()` koja treba da pokrene prenos za taj prvi zahtev. Kada se završi prenos zadat jednim zahtevom, potrebno je u polje `status` date strukture preneti status završene operacije (0 – ispravno završeno do kraja, -1 – greška), izbaciti obrađeni zahtev iz liste i pokrenuti prenos za sledeći zapis u listi. Ako zahteva u listi više nema, ne treba uraditi više ništa (kada bude stavljao novi zahtev u listu, kernel će proveriti i videti da je ona bila prazna, pa pozvati ponovo operaciju `transfer()` itd.) Potrebno je napisati kod operacije `transfer()`, zajedno sa odgovarajućom prekidnom rutinom `dmaInterrupt()` za prekid od DMA kontrolera.

```
void transfer ();
interrupt void dmaInterrupt ();
```

Rešenje

```
void transfer () {
  if (ioHead==0) return; // Exception
  *dmaAddress = ioHead->buffer;
  *dmaCount = ioHead->size;
  *dmaCtrl = 1; // Start I/O
}

interrupt void dmaInterrupt () {
  if (ioHead==0) return; // Exception
  if (*dmaStatus&2) // Error in I/O
    ioHead->status = -1;
  else
    ioHead->status = 0;
  ioHead = ioHead->next; // Remove the request from the list
  if (ioHead==0) { // No more requests
    *dmaCtrl = 0;
    return;
  }
  // Start a new transfer
  *dmaAddress = ioHead->buffer;
  *dmaCount = ioHead->size;
  *dmaCtrl = 1; // Start I/O
}
```

1. zadatak, prvi kolokvijum, april 2014.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva DMA kontrolera:

```
typedef unsigned int REG;
REG* dma1Ctrl =...;     // DMA1 control register
REG* dma1Status =...;   // DMA1 status register
REG* dma1Address =...;  // DMA1 block address register
REG* dma1Count =...;    // DMA1 block size register
REG* dma2Ctrl =...;     // DMA2 control register
REG* dma2Status =...;   // DMA2 status register
REG* dma2Address =...;  // DMA2 block address register
REG* dma2Count =...;    // DMA2 block size register
```

U upravljačkom registru najniži bit je bit *Start* kojim se pokreće prenos jednog bloka preko DMA, a u statusnom

registru najniži bit je bit završetka prenosa (*TransferComplete*), a bit do njega bit greške (*Error*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Kada DMA konotroler završi zadati prenos, on se automatski zaustavlja (nije ga potrebno zaustavljati upisom u upravljački registar). Završetak prenosa sa bilo kog DMA kontrolera generiše isti zahtev za prekid procesoru (signali završetka operacije sa dva DMA kontrolera vezani su na ulazni zahtev za prekid preko OR logičkog kola).

Zahtevi za ulaznim operacijama na nekom uređaju sa kog se prenos blokova vrši preko bilo kog od ova DMA kontrolera vezani su u jednostruko ulančanu listu. Zahtev ima sledeću strukturu:

```
struct IORequest {
  REG* buffer; // Data buffer (block)
  unsigned int size; // Buffer (blok) size
  int status; // Status of operation
  IORequest* next; // Next in the list
};
```

Na prvi zahtev u listi pokazuje globalni pokazivač `ioHead`. Kada kernel u listu stavi novi zahtev, pozvaće operaciju `transfer()` koja treba da pokrene prenos za taj zahtev na bilo kom trenutno slobodnom DMA kontroleru (u slučaju da su oba kontrolera zauzeta ne treba ništa uraditi). Zahtev koji se dodeli nekom od DMA kontrolera na obradu izbacuje se iz liste. Kada se završi prenos zadat jednim zahtevom na jednom DMA kontroleru, potrebno je u polje `status` date strukture preneti status završene operacije (0 – ispravno završeno do kraja, -1 – greška) i pokrenuti prenos za sledeći zapis u listi na tom DMA kontroleru, i potom izbaciti zahtev iz liste. Obratiti pažnju na to da oba DMA kontrolera mogu završiti prenos i generisati prekid istovremeno. Ako zahteva u listi više nema, ne treba uraditi više ništa (kada bude stavljao novi zahtev u listu, kernel će proveriti i videti da je ona bila prazna, pa pozvati ponovo operaciju `transfer()` itd.) Potrebno je napisati kod operacije `transfer()`, zajedno sa odgovarajućom prekidnom rutinom `dmaInterrupt()` za prekid od DMA kontrolera.

```
void transfer ();
interrupt void dmaInterrupt ();
```

Rešenje

```
IORequest *dma1Pending = 0, *dma2Pending = 0; // Currently pending requests

void startDMA1 () {
  if (ioHead!=0 && dma1Pending==0) {
    dma1Pending = ioHead; // Take the first request,
    ioHead = ioHead->next; // remove it from the list
    *dma1Address = dma1Pending->buffer; // and assign it to DMA1
    *dma1Count = dma1Pending->size;
    *dma1Ctrl = 1; // Start I/O
  }
}

void startDMA2 () {
  if (ioHead!=0 && dma2Pending == 0) {
    dma2Pending = ioHead; // Take the first request,
    ioHead = ioHead->next; // remove it from the list
    *dma2Address = dma2Pending->buffer; // and assign it to DMA2
    *dma2Count = dma2Pending->size;
    *dma2Ctrl = 1; // Start I/O
  }
}

void transfer () {
  startDMA1();
  startDMA2();
}
```

```
interrupt void dmaInterrupt () {
  if (dma1Status&1) { // DMA1 completed
    if (dma1Pending==0) return; // Exception
    if (*dma1Status&2) // Error in I/O
      dma1Pending->status = -1;
    else
      dma1Pending->status = 0;
    dma1Pending = 0;
    startDMA1();
  }
  if (dma2Status&1) { // DMA1 completed
    if (dma2Pending==0) return; // Exception
    if (*dma2Status&2) // Error in I/O
      dma2Pending->status = -1;
    else
      dma2Pending->status = 0;
    dma2Pending = 0;
    startDMA2();
  }
}
```

1. zadatak, prvi kolokvijum, septembar 2014.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima jednog ulaznog uređaja i registru posebnog uređaja – vremenskog brojača:

```
typedef unsigned int REG;
REG* ioCtrl =...;   // Device control register
REG* ioData =...;   // Device data register
REG* timer = ...;   // Timer
```

Učitavanje svakog pojedinačnog podatka sa ovog ulaznog uređaja zahteva se posebnim upisom vredosti 1 u najniži bit upravljačkog registra ovog uređaja. Spremnost ulaznog podatka u registru za podatke uređaj ne signalizira nikakvim signalom, ali je sigurno da je ulazni podatak spreman u registru podataka najkasnije 50 ms nakon zadatog zahteva (upisa u kontrolni registar).

Na magistralu računara vezan je i registar posebnog uređaja, vremenskog brojača. Upisom celobrojne vrednosti n u ovaj registar, vremenski brojač počinje merenje vremena od n milisekundi, nakon isteka tog vremena, generiše prekid procesoru.

Na jeziku C napisati kod operacije `transfer()` zajedno sa odgovarajućom prekidnom rutinom za prekid od vremenskog brojača `timerInterrupt()`, koja obavlja učitavanje bloka podataka zadate dužine na zadatu adresu u memoriji sa datog ulaznog uređaja.

```
void transfer (REG* buffer, unsigned int count);
interrupt void timerInterrupt ();
```

Rešenje

```
static const unsigned int timeout = 50; // 50 ms
static int completed = 0;
static REG* ptr = 0;
static unsigned int count = 0;

void transfer (REG* buffer, unsigned int cnt) {
  // Initialize transfer
  completed = 0;
  ptr = buffer;
  count = cnt;
```

```
  // Start transfer:
  *ioCtrl = 1; // Input request
  *timer = timeout; // Start timer

  while (!completed); // Busy wait for transfer completion
}

interrupt void timerInterrupt () {
  *ptr++ = *ioData; // Read data
  if (--count) {
    *ioCtrl = 1; // New input request
    *timer = timeout; // Restart timer
  } else // Completed
    completed = 1;
}
```

1. zadatak, prvi kolokvijum, mart 2013.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva uazno/izlazna uređaja:

```
typedef unsigned int REG;
REG* io1Ctrl =...;   // Device 1 control register
REG* io1Status =...; // Device 1 status register
REG* io1Data =...;   // Device 1 data register
REG* io2Ctrl =...;   // Device 2 control register
REG* io2Status =...; // Device 2 status register
REG* io2Data =...;   // Device 2 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće uređaj, a u statusnim registrima najniži bit je bit spremnosti (*Ready*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`).

Potrebno je napisati kod koji najpre vrši ulaz bloka podataka zadate veličine sa prvog uređaja korišćenjem prekida, a potom izlaz tog istog učitanog bloka podataka na drugi uređaj tehnikom prozivanja (*polling*).

Rešenje

```
static REG* ioPtr = 0;
static int ioCount = 0;
static int ioCompleted = 0;

void transfer (int count) {
  REG* buffer = new REG[count];

  // I/O 1:
  ioPtr = buffer;
  ioCount = count;
  ioCompleted = 0;
  *io1Ctrl = 1; // Start I/O 1

  // Wait for I/O 1 completion:
  while (!ioCompleted);

  // I/O 2
  ioPtr = buffer;
  ioCount = count;
  *io2Ctrl = 1; // Start I/O 2
  while (ioCount>0) {
```

```
    while (!(*io2Status&1)); // busy wait
    *io2Data = *ioPtr++;
    ioCount--;
  }
  *io2Ctrl = 0; // Stop I/O 2

  delete [] buffer;
}

interrupt void io1Interrupt() {
  *ioPtr++ = *io1Data;
  if (--ioCount == 0) {
    ioCompleted = 1;
    *io1Ctrl = 0; // Stop I/O 1
  }
}
```

1. zadatak, prvi kolokvijum, april 2013.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima tri uređaja; prva dva uređaja su ulazni, a treći je izlazni:

```
typedef unsigned int REG;
REG* io1Ctrl =...;   // Device 1 control register
REG* io1Status =...; // Device 1 status register
REG* io1Data =...;   // Device 1 data register
REG* io2Ctrl =...;   // Device 2 control register
REG* io2Status =...; // Device 2 status register
REG* io2Data =...;   // Device 2 data register
REG* io3Ctrl =...;   // Device 3 control register
REG* io3Status =...; // Device 3 status register
REG* io3Data =...;   // Device 3 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće uređaj, a u statusnim registrima najniži bit je bit spremnosti (*Ready*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`).

Potrebno je napisati program koji učitava po jednu reč sa bilo kog od dva ulazna uređaja (na kom god se pre pojavi spreman podatak) i odmah tu učitanu reč šalje na izlazni uređaj. Ovaj prenos se obavlja sve dok se sa ulaznog uređaja ne učita vrednost 0. Sve prenose vršiti programiranim ulazom/izlazom sa prozivanjem (*polling*).

Rešenje

```
const REG ESC = 0;

void main () {
  REG data = ESC;

  *io1Ctrl = 1;
  *io2Ctrl = 1;
  *io3Ctrl = 1;

  while (1) {
    while (*io1Status==0 && *io2Status==0);
    if (*io1Status) data = *io1Data;
    else data = *io2Data;
  if (data==ESC) break;
    while (*io3Status==0);
    *io3Data = data;
```

```
  }

  *io1Ctrl = 0;
  *io2Ctrl = 0;
  *io3Ctrl = 0;
}
```

1. zadatak, prvi kolokvijum, septembar 2013.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima jednog ulaznog uređaja:

```
typedef unsigned int REG;
REG* ioCtrl =...;   // Device control register
REG* ioStatus =...; // Device status register
REG* ioData =...;   // Device data register
```

U upravljačkom registru najniži bit je bit *Start* kojim se pokreće uređaj, a u statusnom registru najniži bit je bit spremnosti (*Ready*), a bit do njega bit greške (*Error*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`).

Zahtevi za ulaznim operacijama na tom uređaju vezani su u jednostruko ulančanu listu. Zahtev ima sledeću strukturu:

```
struct IORequest {
  REG* buffer; // Data buffer (block)
  unsigned int size; // Buffer (blok) size
  void (*callBack)(IORequest*); // Call-back function
  int status; // Status of operation
  IORequest* next; // Next in the list
};
```

Kada se završi prenos zadat jednim zahtevom, potrebno je pozvati funkciju na koju ukazuje pokazivač `callBack` u tom zahtevu, sa argumentom koji ukazuje na taj zahtev. Ovu funkciju implementira onaj ko je zahtev postavio i služi da mu signalizira da je zahtev obrađen. U polju `status` date strukture treba preneti status završene operacije (0 – ispravno završeno do kraja, -1 – greška). Obrađeni zahtev ne treba brisati iz liste (to je odgovornost onog ko je zahtev postavio).

Potrebno je napisati kod operacije `transfer()`, zajedno sa odgovarajućom prekidnom rutinom za prekid od uređaja `ioInterrupt()`, koja obavlja sve prenose zadate zahtevima u listi na čiji prvi zapis ukazuje argument `ioHead` tehnikom programiranog ulaza korišćenjem prekida.

```
void transfer (IORequest* ioHead);
interrupt void ioInterrupt ();
```

Rešenje

```
static int completed = 0;
static int status = 0;
static REG* ptr = 0;
static unsigned int count = 0;

void transfer (IORequest* ioHead) {
  while (ioHead) {
    completed = 0;  // initialize transfer
    status = 0;
    ptr = ioHead->buffer;
    count = ioHead->size;
    *ioCtrl = 1; // Start I/O
    while (!completed); // wait for I/O to complete
    ioHead->status=status;  // set status
    ioHead->callBack(ioHead);  // signal completion
```

```
    ioHead = ioHead->next;  // take next
  }
}

interrupt void ioInterrupt () {
  if (*ioStatus&2) { // Error in I/O
    completed = 1;
    status = -1;
    *ioCtrl = 0; // Stop I/O 2
    return;
  }
  *ptr++ = *ioData; // input data
  if (--count == 0) { // transfer completed
    completed=1;  // signal completion
    *ioCtrl = 0;    // stop I/O
  }
}
```

1. zadatak, prvi kolokvijum, mart 2012.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva izlazna uređaja:

```
typedef unsigned int REG;
REG* io1Ctrl =...;   // Device 1 control register
REG* io1Status =...; // Device 1 status register
REG* io1Data =...;   // Device 1 data register
REG* io2Ctrl =...;   // Device 2 control register
REG* io2Status =...; // Device 2 status register
REG* io2Data =...;   // Device 2 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće uređaj, a u statusnim registrima najniži bit je bit spremnosti (*Ready*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Potrebno je napisati kod koji vrši prenos po jednog bloka podataka na svaki od dva uređaja uporedo, na prvi uređaj tehnikom prozivanja (*polling*), a na drugi korišćenjem prekida. Transfer se obavlja pozivom sledeće funkcije iz koje se vraća kada su oba prenosa završena:

```
void transfer (unsigned* blk1, int count1, unsigned* blk2, int count2);
```

Rešenje

```
static unsigned* io2Ptr = 0;
static int io2Count = 0;
static int io2Completed = 0;

void transfer (unsigned* blk1, int count1, unsigned* blk2, int count2) {
  // I/O 2:
  io2Ptr = blk2;
  io2Count = count2;
  io2Completed = 0;
  *io2Ctrl = 1; // Start I/O 2

  // I/O 1
  *io1Ctrl = 1; // Start I/O 1
  while (count1>0) {
    while (!(*io1Status&1)); // busy wait
    *io1Data = *blk1++;
    count1--;
  }
```

```
  *io1Ctrl = 0; // Stop I/O 1

  // Wait for I/O 2 completion:
  while (!io2Completed);
}

interrupt void io2Interrupt() {
  *io2Data = *io2Ptr++;
  if (--io2Count == 0) {
    io2Completed = 1;
    *io2Ctrl = 0; // Stop I/O 2
  }
}
```

1. zadatak, prvi kolokvijum, maj 2012.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima jednog DMA uređaja:

```
typedef unsigned int REG;
REG* dmaCtrl =...; // control register
REG* dmaStatus =...; // status register
REG* dmaBlkAddr =...; // data block address register
REG* dmaBlkSize =...; // data block size register
```

U upravljačkom registru najniži bit je bit *Start* kojim se pokreće prenos preko DMA, a u statusnom registru najniži bit je bit spremnosti (*Ready*) čija vrednost 1 ukazuje da je DMA spreman za novi prenos podatka (inicijalno je tako postavljen). Postavljanje bita spremnosti kada DMA završi zadati prenos generiše signal zahteva za prekid procesoru. Zahtevi za izlaznim operacijama na nekom izlaznom uređaju vezani su u jednostruko ulančanu listu. Zahtev ima sledeću strukturu:

```
struct OutputRequest {
  char* buffer; // Buffer with data (block)
  unsigned int size; // Buffer (blok) size
  void (*callBack)(OutputRequest*); // Call-back function
  OutputRequest* next; // Next in the list
};
```

Kada se završi prenos zadat jednim zahtevom, potrebno je pozvati funkciju na koju ukazuje pokazivač `callBack` u tom zahtevu, sa argumentom koji ukazuje na taj zahtev. Ovu funkciju implementira onaj ko je zahtev postavio i služi da mu signalizira da je zahtev obrađen. Obrađeni zahtev ne treba brisati iz liste (to je odgovornost onog ko je zahtev postavio).

Potrebno je napisati kod operacije `transfer()`, zajedno sa odgovarajućom prekidnom rutinom `dmaInterrupt()`, koja obavlja sve prenose zadate zahtevima u listi na čiji prvi zapis ukazuje argument `ioHead`.

```
void transfer (OutputRequest* ioHead);
interrupt void dmaInterrupt ();
```

Rešenje

```
static int dmaCompleted = 0;

void transfer (OutputRequest* ioHead) {
  while (ioHead) {
    dmaCompleted = 0;  // initialize transfer
    *dmaBlkAddress = ioHead->buffer;
    *dmaBlkSize = ioHead->size;
    *dmaCtrl = 1;  // start transfer
    while (!dmaCompleted); // wait for DMA to complete
    ioHead->callBack(ioHead);  // signal completion
```

```
    ioHead = ioHead->next;  // take next
  }
}

interrupt void dmaInterrupt () {
  dmaCompleted = 1;
  *dmaCtrl = 0;
}
```

1. zadatak, prvi kolokvijum, septembar 2012.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva ulazna uređaja:

```
typedef unsigned int REG;
REG* io1Ctrl =...;   // Device 1 control register
REG* io1Status =...; // Device 1 status register
REG* io1Data =...;   // Device 1 data register
REG* io2Ctrl =...;   // Device 2 control register
REG* io2Status =...; // Device 2 status register
REG* io2Data =...;   // Device 2 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće uređaj, a u statusnim registrima najniži bit je bit spremnosti (*Ready*), a bit do njega bit greške (*Error*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`).

Potrebno je napisati kod koji vrši prenos po jednog bloka podataka sa svakog od dva uređaja uporedo, sa prvog uređaja tehnikom prozivanja (*polling*), a sa drugog korišćenjem prekida. U slučaju greške na bilo kom uređaju, prenos sa tog uređaja treba prekinuti, a sa drugog uređaja treba nastaviti (osim ukoliko je i kod njega došlo do greške). Transfer se obavlja pozivom sledeće funkcije iz koje se vraća kada su oba prenosa završena:

```
int transfer (unsigned* blk1, int count1, unsigned* blk2, int count2);
```

Povratna vrednost ove funkcije treba da bude ceo broj čija binarna predstava u najniža dva bita ima sledeće značenje: bit 0 ukazuje na grešku u prenosu sa prvog uređaja (0-ispravno, 1- greška), a bit 1 na grešku u prenosu sa drugog uređaja.

Rešenje

```
static unsigned* io2Ptr = 0;
static int io2Count = 0;
static int io2Completed = 0;

int transfer (unsigned* blk1, int count1, unsigned* blk2, int count2) {
  // I/O 2:
  io2Ptr = blk2;
  io2Count = count2;
  io2Completed = 0;
  int status = 0;
  *io2Ctrl = 1; // Start I/O 2

  // I/O 1
  *io1Ctrl = 1; // Start I/O 1
  while (count1>0) {
    while (!(*io1Status&1)); // Busy wait
    if (*io1Status&2) { // Error in I/O 1
      status |= 1;
      break;
    }
    *blk1++ = *io1Data;
```

```
    count1--;
  }
  *io1Ctrl = 0; // Stop I/O 1

  // Wait for I/O 2 completion:
  while (!io2Completed);
  if(io2Completed<0)status |= 2;
  return status;
}

interrupt void io2Interrupt() {
  if (*io2Status&2) { // Error in I/O 2
    io2Completed = -1;
    *io2Ctrl = 0; // Stop I/O 2
    return;
  }
  *io2Ptr++ = *io2Data;
  if (--io2Count == 0) {
    io2Completed = 1;
    *io2Ctrl = 0; // Stop I/O 2
  }
}
```

1. zadatak, prvi kolokvijum, april 2011.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva izlazna uređaja:

```
typedef unsigned int REG;
REG* io1 Ctrl =...;   // Device 1 control register
REG* io1 Status =...; // Device 1 status register
REG* io1 Data =...;   // Device 1 data register
REG* io2Ctrl =...;   // Device 2 control register
REG* io2Status =...; // Device 2 status register
REG* io2Data =...;   // Device 2 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće uređaj, a u statusnim registrima najniži bit je bit spremnosti (*Ready*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Biti spremnosti statusnih registara oba uređaja vezani su preko logičkog ILI kola na isti signal zahteva za prekid, tako da se isti prekid generiše ako je bilo koji (ili oba) uređaja postavio svoj bit spremnosti.

Potrebno je napisati kod koji može da izvrši prenos datog bloka podataka korišćenjem oba uređaja uporedo, tako da se na svaki uređaj prenese po pola datog bloka (ili približno pola, ako je broj reči neparan). Ovaj prenos pokreće se pozivom dole date funkcije `transfer()`.

```
int flag1 = 0, flag2 = 0; // I/O completed
unsigned int index1, count1;
unsigned int index2, count2;
REG* buf1;
REG* buf2;

void transfer (REG* buffer, unsigned int count) {
  count1 = count/2;
  count2 = count1+count%2;
  buf1 = buffer;
  buf2 = buffer+count1;
  index1 = index2 = 0;
  flag1 = flag2 = 0;
  *io1Ctrl = 1;
  *io1Ctr2 = 1;
```

```
}

int isIOCompleted () {
  return flag1 && flag2;
}

interrupt void ioInterrupt();
```

Napisati kod prekidne rutine `ioInterrupt()`.

Rešenje

```
interrupt void ioInterrupt() {
  if (*ioStatus1) {
    *io1Data = buf1[index1++];
    if (--count1 == 0) {
      flag1 = 1;
      *iosCtrl1 = 0;
    }
  }
  if (*ioStatus2) {
    *io2Data = buf2[index2++];
    if (--count2 == 0) {
      flag2 = 1;
      *iosCtrl2 = 0;
    }
  }
}
```

1. zadatak, prvi kolokvijum, maj 2011.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva ulazna uređaja:

```
typedef unsigned int REG;
REG* io1 Ctrl =...;   // Device 1 control register
REG* io1 Status =...; // Device 1 status register
REG* io1 Data =...;   // Device 1 data register
REG* io2Ctrl =...;   // Device 2 control register
REG* io2Status =...; // Device 2 status register
REG* io2Data =...;   // Device 2 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće uređaj, a u statusnim registrima najniži bit je bit spremnosti (*Ready*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`).

Potrebno je napisati kod funkcije `transfer()` koji vrši prenos datog bloka podataka korišćenjem oba uređaja uporedo, tako da se sa svakog uređaja prenese po pola datog bloka (ili približno pola, ako je broj reči neparan), programiranim ulazom/izlazom sa prozivanjem (*polling*), uz maksimalni paralelizam rada uređaja i prenosa delova bloka.

```
void transfer (REG* buffer, unsigned int count) ;
```

Rešenje

```
void transfer (REG* buffer, unsigned int count) {
  unsigned int index1, count1;
  unsigned int index2, count2;
  REG* buf1;
  REG* buf2;

  count1 = count/2;
```

```
  count2 = count1+count%2;
  buf1 = buffer;
  buf2 = buffer+count1;
  index1 = index2 = 0;
  *io1Ctrl = 1;
  *io1Ctr2 = 1;

  while ((count1!=0) || (count2!=0)) {
    if ((count1!=0) && *ioStatus1) {
      buf1[index1++] = *io1Data;
      if (--count1 == 0) *iosCtrl1 = 0;
    }
    if ((count2!=0) && *ioStatus2) {
      buf2[index2++] = *io2Data;
      if (--count2 ==   0) *iosCtrl2 = 0;
    }
  }
}
```

1. zadatak, prvi kolokvijum, septembar 2011.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva uređaja; prvi uređaj je ulazni, drugi izlazni:

```
typedef unsigned int REG;
REG* io1 Ctrl =...;   // Device 1 control register
REG* io1 Status =...; // Device 1 status register
REG* io1 Data =...;   // Device 1 data register
REG* io2Ctrl =...;   // Device 2 control register
REG* io2Status =...; // Device 2 status register
REG* io2Data =...;   // Device 2 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće uređaj, a u statusnim registrima najniži bit je bit spremnosti (*Ready*). Svi registri su veličine jedne mašinske reči (tip `unsigned int`).

Potrebno je napisati program koji učitava po jednu reč sa ulaznog urešaja i odmah tu učitanu reč šalje na izlazni uređaj. Ovaj prenos se ponavlja sve dok se sa ulaznog uređaja ne učita vrednost 0. Oba prenosa vršiti programiranim ulazom/izlazom za prozivanjem (*polling*).

Rešenje

```
const REG ESC = 0;

void main () {
  *io1Ctrl = 1;
  *io2Ctrl = 1;

  while (1) {
    while (*io1Status==0);
    REG data = *io1Data;
  if (data==ESC) break;
    while (*io2Status==0);
    *io2Data = data;
  }

  *io1Ctrl = 0;
  *io2Ctrl = 0;
}
```

1. zadatak, prvi kolokvijum, april 2010.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima dva DMA kontrolera:

```
typedef unsigned int REG;
REG* dma1Ctrl =...; // DMA1 control register
REG* dma1Status =...; // DMA1 status register
REG* dma1Addr =...; // DMA1 block address register
REG* dma1Count =...; // DMA1 block size register
REG* dma2Ctrl =...; // DMA2 control register
REG* dma2Status =...; // DMA2 status register
REG* dma2Addr =...; // DMA2 block address register
REG* dma2Count =...; // DMA2 block size register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće DMA kontroler, a u statusnim registrima najniži bit je bit spremnosti (*Ready*) čija vrednost 1 ukazuje da je DMA kontroler završio prenos. Svaki DMA kontroler upravlja svojom izlaznom periferijom sa kojom direktno komunicira, tako da procesor ne treba (i ne može) direktno da pristupa kontrolerima periferija. Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Jezgro operativnog sistema postavlja zahteve za prenos blokova podataka iz memorije na bilo koju od dve periferije uvezivanjem sledećih struktura sa zahtevima u listu:

```
struct DMAReq {
  DMAReq* next; // Next request in the list
  unsigned int addr; // Block address
  unsigned int size; // Block size
};
DMAReq* head;
DMAReq* tail;
```

Na jeziku C napisati potprogram koji uzima jedan po jedan postavljeni zahtev iz liste i zadaje ih bilo kom slobodnom DMA kontroleru. Čekanje da se pojavi prvi zahtev u praznoj listi i da se oslobodi DMA kontroler treba obavljati uposleno, odnosno prozivanjem (*busy waiting*, *polling*).

Rešenje

```
void DMADriver() {
  while (1){
    DMAReq* req;
    // Busy wait for a request:
    while (head==0);
    // Take a request:
    req = head;
    head=head->next;
    if (head==0) tail==0;
    req->next=0;
    // Busy wait for a free DMA:
    while ((*dma1Status&1)==0 && (*dma2Status&1)==0);
    // Start DMA:
    if (*dma1Status&1) {
      *dma1Addr=req->addr;
      *dma1Count=req->size;
      *dma1Ctrl=1;
    } else
    if (*dma2Status&1) {
      *dma2Addr=req->addr;
      *dma2Count=req->size;
      *dma2Ctrl=1;
    }
    free(req);
```

```
  }
}
```

2. zadatak, prvi kolokvijum, april 2009.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima nekih periferijskih uređaja, pošto su oni inicijalizovani adresama tih registara:

```
typedef unsigned int REG;
REG* ioCtrl1 =...;    // PER1 control register
REG* ioStatus1 =...;  // PER1 status register
REG* ioData1 =...;    // PER1 data register
REG* ioCtrl2 =...;    // PER2 control register
REG* ioStatus2 =...;  // PER2 status register
REG* ioData2 =...;    // PER2 data register
REG* ioCtrl3 =...;    // PER3 control register
REG* ioStatus3 =...;  // PER3 status register
REG* ioData3 =...;    // PER3 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće periferija, a u statusnim registrima najniži bit je bit spremnosti (*Ready*) čija vrednost 1 ukazuje da je kontroler periferije spreman za prenos podatka preko svog registra za podatke. PER1 i PER2 su ulazne, a PER3 je izlazna periferija. Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Date su deklaracije bloka podataka koji treba prenositi:

```
const int N = ...;
unsigned int buf[N];
```

Na jeziku C napisati program koji uporedo sa PER1 i PER2 učitava blok od ukupno `N` reči, i na PER3 upisuje taj isti blok od `N` reči, korišćenjem mehanizma prozivanja (engl. *polling*), bez nametanja naizmeničnosti prenosa po jednog podatka (npr. da se uvek jedan podatak prebaci sa PER1, pa onda sa PER2 i tako naizmenično), već podatak sa datom periferijom treba preneti kad god je ona spremna, ne čekajući ostale. Broj podataka koji se čitaju sa pojedinačnih periferija PER1 i PER2 nije unapred poznat, već zavisi od brzine rada periferija.

Rešenje

```
void main () {
  int i12=0, i3=0;
  *ioCtrl1=1; *ioCtrl2=1; *ioCtrl3=1;  // Start
  while (i3<N) {
    if (i12<N && *ioStatus1&1) buf[i12++]=*ioData1;
    if (i12<N && *ioStatus2&1) buf[i12++]=*ioData2;
    if (i3<i12 && *ioStatus3&1) *ioData3=buf[i3++];
  }
  *ioCtrl1=0; *ioCtrl2=0;  *ioCtrl3=0; // Stop them all
}
```

2. zadatak, prvi kolokvijum, april 2008.

Date su deklaracije pokazivača preko kojih se može pristupiti registrima nekih periferijskih uređaja, pošto su oni inicijalizovani adresama tih registara:

```
typedef unsigned int REG;
REG* ioCtrl1 =...; // PER1 control register
REG* ioStatus1 =...; // PER1 status register
REG* ioData1 =...; // PER1 data register
REG* ioCtrl2 =...; // PER2 control register
REG* ioStatus2 =...; // PER2 status register
REG* ioData2 =...; // PER2 data register
```

U upravljačkim registrima najniži bit je bit *Start* kojim se pokreće periferija, a u statusnim registrima najniži bit je bit spremnosti (*Ready*) čija vrednost 1 ukazuje da je kontroler periferije spreman za prenos podatka preko svog registra za podatke. PER1 je ulazna, a PER2 izlazna periferija. Svi registri su veličine jedne mašinske reči (tip `unsigned int`). Date su deklaracije dva bloka podataka koje treba preneti:

```
const int N = ...;
unsigned int buf1[N], buf2[N];
```

Na jeziku C napisati program koji uporedo sa PER1 učitava blok od N reči, a na PER2 upisuje blok od N reči, korišćenjem mehanizma prozivanja (engl. *polling*), bez nametanja naizmeničnosti prenosa po jednog podatka (npr. da se uvek jedan podatak prebaci sa PER1, pa onda sa PER2 i tako naizmenično), već podatak sa datom periferijom treba preneti kad god je ona spremna, ne čekajući na onu drugu.

Rešenje

```
void main () {
  int i1=0, i2=0;
  *ioCtrl1=1; *ioCtrl2=1;  // Start them both
  while (i1<N || i2<N) {
    if (i1<N && *ioStatus1&1) buf1[i1++]=*ioData1;
    if (i2<N && *ioStatus2&1) *ioData2=buf2[i2++];
  }
  *ioCtrl1=0; *ioCtrl2=0;  // Stop them both
}
```

1. zadatak, prvi kolokvijum, april 2007.

Predložiti interfejs `DMACtrl` i specifikovati ga apstraktnom klasom na jeziku C++. Ovaj interfejs treba da delovima operativnog sistema koji upravljaju uređajima obezbedi uniforman pristup različitim vrstama DMA kontrolera, tako što im na isti način zadaje operacije prenosa bloka podataka iz ili u memoriju, kao i način obaveštavanja o završenom prenosu i eventualnoj grešci. Obaveštavanje o završenoj operaciji može da bude prozivanjem (*polling*) ili prekidom sa zadatim ulazom.

Rešenje

```
enum Status{OK, Error};

class DMA {
public:
//ulazni parametri: adresa bloka u memoriji, adresa registra podataka
//uređaja, adresa statusnog registra, maska za statusni registar, smjer
//operacije i velicina bloka
 virtual void start(void* mem, void* devData, void*  devStatus, int
StatusMask, int memToIO, int vel);

//rezultat: logicka vrijednost koja pokazuje da li je operacija zavrsena
 virtual int is_finished();

//zadaje kontroleru broj prekidne rutine preko koje traba da obavjesti
//sistem o kraju operacije
 virtual void setIntNo(int interruptNo);

//vraca status poslednje zavrsene operacije
 virtual Status getStatus()
}
```

2. zadatak, prvi kolokvijum, april 2006.

Na nekom računaru sa multiprogramskim operativnim sistemom neki kontroler periferije nema vezan svoj signal završetka operacije na ulaze za prekid procesora, ali je poznato vreme završetka operacije zadate tom uređaju u najgorem slučaju (maksimalno vreme završetka operacije). Za obradu ulazno/izlaznih operacija zadatih tom uređaju brine se poseban proces tog operativnog sistema. Kakvu uslugu treba operativni sistem da obezbedi, a ovaj proces da koristi, da bi se izbegla tehnika kojom se može detektovati završetak operacije čitanjem statusnog registra i ispitivanjem bita završetka operacije (*polling*), tj. da bi se izbeglo uposleno čekanje (*busy waiting*)? Precizno opisati kako ovaj proces treba da koristi ovu uslugu.

Rešenje

Operativni sistem treba da nudi uslugu da neki proces može da se uspava i da se probuti posle tačno odredjenog vremena (ili malo kasnije od tog vremena). Npr operativni sistem treba da obezbedi primitivu `void sleep(int time)`, koja obezbedjuje da se proces uspava time milisekunti (odnosno stavi u red uspavanih, engl. *Sleep*), a da se posle toga može probuditi (odnosno proces vrati u red spremnih, engl. *Ready*). Proces za obradu ulazno/izlazne operacije treba da radi sledeće:

```
const int maxTime = ...; // response time (worst case)

while (!end){
 initialize_IO_operation();
 sleep(maxTime);
 process_IO_result();
}
```

Ulaz/izlaz (blokovski uređaji)

3. zadatak, kolokvijum, jun 2021.

U nekom sistemu implementira se keš blokova sa blokovskih uređaja („diskova“) kodom koji je dat u nastavku. Za svaki uređaj sa datim identifikatorom pravi se jedan objekat klase `BlockIOCache`, inicijalizovan tim identifikatorom, koji predstavlja keš blokova sa tog uređaja. Keš je kapaciteta `CACHESIZE` blokova veličine `BLKSIZE`. Keš je interno organizovan kao heš mapa `map` sa `MAPSIZE` ulaza. Svaki ulaz niza `map` sadrži glavu liste keširanih blokova koji se preslikavaju u taj ulaz. Funkcija `hash` je heš funkcija koja preslikava broj bloka u ulaz u nizu `map`. Glava liste, kao i pokazivač na sledeći element u listi čuvaju se kao indeksi elementa niza `entries` koji sadrži keširane blokove; vrednost -1 označava kraj (*null*). Svaki element niza `entries` je struktura tipa `CacheEntry` u kojoj je polje `blkNo` broj bloka koji je keširan u tom elementu, polje `next` ukazuje na sledeći element liste, a polje `buf` je sadržaj samog bloka.

Na početku složene operacije sa uređajem, kod koji koristi keš najpre traži da je potrebni blok učitan pozivom funkcije `getBlock` koja vraća pokazivač na niz bajtova u baferu – učitanom bloku. Pošto više ovakvih složenih operacija može biti ugnježdeno, blok iz keša može biti izbačen (zamenjen drugim) samo ako ga više niko ne koristi, što se realizuje brojanjem referenci u polju `refCounter` strukture `CacheEntry`.

Član `freeHead` je glava liste slobodnih elemenata u nizu `entries` (-1 ako slobodnih nema). Funkcija `evict` izbacuje jedan keširani blok iz punog keša, ukoliko takav može da nađe, oslobađa njegov ulaz i stavlja ga u listu slobodnih.

Implementirati funkciju `getBlock` koja treba da obezbedi da je traženi blok u kešu, odnosno učita ga ako nije. Ako nema mesta u kešu jer nijedan blok ne može da se izbaci, treba vratiti *null*. Ostale članice date klase su implementirane, a na raspolaganju je i funkcija koja učitava blok na dati uređaj:

```
void ioRead(int device, BlkNo blk, Byte* buffer);

typedef unsigned char Byte; // Unit of memory
typedef long long BlkNo; // Device block number
const unsigned BLKSIZE = ...; // Block size in Bytes
class BlockIOCache {
public:
    BlockIOCache(int device);
    Byte* getBlock(BlkNo blk);
    ...
protected:
    static int hash(BlkNo);
    void evict();
private:
    static const unsigned CACHESIZE = ...; // Cache size in blocks
    static const unsigned MAPSIZE = ...; // Hash map size in entries
    struct CacheEntry {
        BlkNo blkNo;
        int next;
        int refCounter;
        Byte buf[BLKSIZE];
    };
    int dev;
    int map[MAPSIZE]; // Hash map
    CacheEntry entries[CACHESIZE]; // Cache
    int freeHead; // Free entries list head
};
```

Rešenje

```
Byte* BlockIOCache::getBlock(BlkNo blk) {
    // Find the requested block in the cache and return it if present:
    int entry = hash(blk);
```

```
    for (int i = map[entry]; i != -1; i = entries[i].next) {
        if (entries[i].blkNo==blk) {
            entries[i].refCounter++;
            return entries[i].buf;
        }
    }
    // The block is not in the cache, find a free slot to load it:
    if (freeHead == -1) evict();
    if (freeHead == -1) return 0; // Error: cannot find space
    int free = freeHead;
    freeHead = entries[freeHead].next;
    // Load the requested block:
    entries[free].blkNo = blk;
    entries[free].refCounter = 1;
    entries[free].next = map[entry];
    map[entry] = free;
    ioRead(dev, blk, entries[free].buf);
    return entries[free].buf;
}
```

1. zadatak, treći kolokvijum, jun 2019.

U nekom sistemu implementira se keš blokova za ulazne, blokovski orijentisane uređaje sa direktnim pristupom; pošto su uređaji ulazni, blokovi se mogu samo čitati. Za svaki takav uređaj sa datim identifikatorom pravi se jedan objekat klase `BlockIOCache`, inicijalizovan tim identifikatorom, koji predstavlja keš blokova sa tog uređaja. Keš čuva najviše `CACHESIZE` blokova veličine `BLKSIZE` u nizu `cache`. Svaki ulaz i u nizu `cacheMap` sadrži broj bloka na uređaju koji se nalazi u ulazu i keša. Kada učitava blokove, keš najpre redom popunjava svoje ulaze, dok ima neiskorišćenih ulaza; podatak član `numOfBlocks` govori o tome koliko je ulaza zauzeto (redom, prvih, od ulaza broj 0). Kada popuni sve ulaze, traženi blok koji nije u kešu učitava se na mesto bloka koji je najdavnije učitan (zamenjuje se taj blok učitanim blokom).

Na raspolaganju je operacija `ioRead` koja sa datog uređaja učitava blok sa zadatim brojem u bafer zadat poslednjim argumentom. Implementirati funkciju `BlockIOCache::read` koja treba da iz bloka za datim brojem `blk` niz bajtova počev od pozicije `offset` i dužine `sz` prepiše u bafer `buffer` koji je alocirao pozivalac. Pretpostaviti da su ovi argumenti ispravni i konzistentni (pozivalac je proverio njihovu ispravnost) i da se eventualne greške u operaciji `ioRead` obrađuju u njoj ili negde drugde (ignorisati ih). (Definicija klase `BlockIOCache` nije kompletna i može se dopunjavati po potrebi.)

```
typedef char byte; // Unit of memory
typedef long long BlkNo; // Device block number
void ioRead (int device, BlkNo blk, byte* buffer);

class BlockIOCache {
public:
  BlockIOCache (int device) : dev(device), numOfBlocks(0) {}
  void read (BlkNo blk, byte* buffer, size_t offset, size_t sz);

private:
  static const unsigned BLKSIZE = ...; // Block size in Bytes
  static const unsigned CACHESIZE = ...; // Cache size in blocks

  int dev;
  byte cache [CACHESIZE][BLKSIZE]; // Cache
  BlkNo cacheMap [CACHESIZE]; // Contents of the cache (block numbers)
  int numOfBlocks; // Number of used entries (blocks in the cache)
};
```

Rešenje

U klasi `BlockIOCache` potreban je još sledeći nestatički podatak član:

```
int BlockIOCache::toReplace = 0;

void BlockIOCache::read(BlkNo blk, byte* buffer, size_t offset, size_t sz)
{
  // Search for the requested block in the cache:
  int entry = -1;
  for (int i=0; i<this->numOfBlocks && entry<0; i++)
    if (this->cacheMap[i]==blk) entry = i; // Block found
  if (entry<0) {
    // The block is not in the cache, load it to the cache:
    if (this->numOfBlocks<CACHESIZE)
      entry = this->numOfBlocks++;  // Load it to a free slot
    else {
      entry = this->toReplace++; // Replace the least recently loaded block
      this->toReplace %= CACHESIZE;
    }
    this->cacheMap[entry] = blk;
    ioRead(this->dev,blk,this->cache[entry]);
  }
  // Copy the extract to the buffer and return:
  for (size_t j=0; j<sz && offset+j<BLKSIZE; j++)
    buffer[j] = this->cache[entry][offset+j];
}
```

1. zadatak, treći kolokvijum, jun 2018.

Dat je neki sekvencijalni, znakovno orijentisani ulazni uređaj (ulazni tok) sa koga se znak učitava sledećom funkcijom:

```
char getchar();
```

Od ovog uređaja napraviti apstrakciju sekvencijalnog, blokovski orijentisanog ulaznog uređaja, sa koga se blok veličine `BlockSize` učitava na zadatu adresu funkcijom:

```
void readBlock(char* addr);
```

Ignorisati sve greške.

Rešenje

```
void readBlock (char* addr) {
  for (int i=0; i<BlockSize; i++)
    addr[i] = getchar();
}
```

1. zadatak, treći kolokvijum, jun 2017.

U nekom sistemu implementira se keš blokova sa blokovskih uređaja („diskova“) kodom koji je dat u nastavku. Za svaki uređaj sa datim identifikatorom pravi se jedan objekat klase `BlockIOCache`, inicijalizovan tim identifikatorom, koji predstavlja keš blokova sa tog uređaja. Keš čuva najviše `CACHESIZE` blokova veličine `BLKSIZE` u nizu `cache`. Svaki ulaz i u nizu cacheMap sadrži broj bloka na disku koji se nalazi u ulazu i keša. Flegovi u ulazu i niza `flags` ukazuju na to da je u ulazu i validan sadržaj (da ulaz nije prazan, odnosno već je bio korišćen i u njega je učitan sadržaj nekog bloka) – fleg `F_VALID`, odnosno da je sadržaj „zaprljan“, tj. menjan i treba ga upisati na disk – fleg `F_DIRTY`.

Kod koji koristi ovaj keš koristi ga na sledeći način za neku složenu operaciju sa uređajem:

```
Byte* buffer = diskCache->getBlock(blk);
...
```

```
buffer[...] = ...;
...
diskCache->writeBlock(buffer);
...
diskCache->releaseBlock(buffer);
```

Na početku ovakve složene operacije sa uređajem, ovaj kod najpre traži da keš obezbedi da je potrebni blok učitan pozivom funkcije `getDiskBlock` koja vraća pokazivač na niz bajtova u baferu – učitanom bloku. Potom ovaj kod može da čita ili upisuje u sadržaj tog bafera. Ako je upisivao, mora da pozove funkciju `writeBlock`, kako bi keš znao da dati blok treba da upiše na uređaj. Kada završi celu operaciju, kod poziva funkciju `releaseBlock`, kako bi kešu stavio do znanja da on više ne koristi taj blok. Pošto više ovakvih složenih operacija može biti ugnježđeno, blok iz keša može biti izbačen (zamenjen drugim) samo ako ga više niko ne koristi, što se realizuje brojanjem referenci u nizu `refCounter`.

Implementirati funkcije `getBlock` i `releaseBlock`. Funkcija `getBlock` treba da obezbedi da je traženi blok u kešu, odnosno učita ga ako nije; ako ga treba učitati, to može biti urađeno u bilo koji ulaz keša koji sadrži blok koji nije validan ili nije više korišćen; ako takvog nema, treba vratiti 0 (greška, nema mesta u kešu). Na raspolaganju su sledeće dve funkcije koje vrše učitavanje, odnosno upis (respektivno) bloka na dati uređaj:

```
void ioRead(int device, BlkNo blk, Byte* buffer);
void ioWrite(int device, BlkNo blk, Byte* buffer);
typedef char Byte; // Unit of memory
typedef long long BlkNo; // Device block number

class BlockIOCache {
public:
  BlockIOCache (int device);

  Byte* getBlock (BlkNo blk);
  void writeBlock (Byte* buffer);
  void releaseBlock (Byte* buffer);

private:
  static const unsigned BLKSIZE = ...; // Block size in Bytes
  static const unsigned CACHESIZE = ...; // Cache size in blocks
  static const short F_VALID = 0x01; // Valid bit mask
  static const short F_DIRTY = 0x02; // Dirty bit mask

  int dev;
  Byte cache [CACHESIZE][BLKSIZE]; // Cache
  BlkNo cacheMap [CACHESIZE]; // Contents of the cache (block numbers)
  unisgned refCounter [CACHESIZE]; // Reference counters
  short flags [CACHESIZE]; // Valid, Dirty flags
};

BlockIOCache::BlockIOCache (int device) : dev(device) {
  for (int i=0; i<CACHESIZE; i++)
    this->cacheMap[i] = this->refCounter[i] = this->flags[i] = 0;
}

void BlockIOCache::writeBlock (Byte* buffer) {
  int i = (buffer-this->cache[0])/BLKSIZE;
  if (i<0 || i>=CACHESIZE) return; // Exception
  this->flags[i] |= F_DIRTY;
}
```

Rešenje

```
Byte* BlockIOCache::getBlock (BlkNo blk) {
  // Find the requested block in the cache and return it if present:
  int i, free = -1;
  for (i=0; i<CACHESIZE; i++) {
    if (this->flags[i]&F_VALID && this->cacheMap[i]==blk) {
      this->refCounter[i]++;
      return &this->cache[i];
    }
    if ((this->flags[i]&F_VALID==0 || this->refCounter[i]==0)
        && free==-1) free=i;
  }
  // The block is not in the cache, load it to the 'free' slot:
  if (free==-1) return 0; // A problem: there is no free space in the cache
  // Load the requested block:
  this->cacheMap[free] = blk;
  this->flags[free] = F_VALID;
  this->refCounter[free] = 1;
  ioRead(this->dev,blk,this->cache[free]);
  return this->cache[free];
}

void BlockIOCache::releaseBlock (Byte* buffer) {
  int i = (buffer-this->cache[0])/BLKSIZE;
  if (i<0 || i>=CACHESIZE) return; // Exception
  // If the block is dirty, write it to the device:
  if (this->flags[i]&F_VALID && this->flags[i]&F_DIRTY)
    ioWrite(this->dev,this->cacheMap[i],this->cache[i]);
  this->flags[i] &= ~F_DIRTY;
  this->refCounter[i]--;
}
```

1. zadatak, treći kolokvijum, septembar 2016.

Dat je proceduralni interfejs prema nekom blokovski orijentisanom ulaznom uređaju sa direktnim pristupom:

```
extern const int BlockSize;
extern int BlockIOHandle;
long getSize(BlockIOHandle handle);
int readBlock(BlockIOHandle handle, long blockNo, char* addr);
```

Uređaj se identifikuje „ručkom“ tipa `BlockIOHandle`, a blok je veličine `BlockSize` znakova. Operacija `getSize` vraća ukupnu veličinu sadržaja (podataka) na uređaju (u znakovima), a operacija `readBlock` učitava blok sa zadatim brojem u bafer na zadatoj adresi u memoriji i vraća 0 u slučaju uspeha. Obe operacije vraćaju negativnu vrednost u slučaju greške, uključujući i pokušaj čitanja bloka preko granice veličine sadržaja.

Korišćenjem ovog interfejsa implementirati sledeći objektno orijentisani interfejs prema ovom uređaju, koji od njega čini apstrakciju ulaznog toka, odnosno znakovno orijentisanog ulaznog uređaja sa direktnim pristupom:

```
class IOStream {
public:
  IOStream (BlockIOHandle d);
  int seek (long offset);
  int getChar (char& c);
};
```

Operacija `seek` postavlja poziciju „kurzora“ za čitanje na zadatu poziciju (pomeraj počev od znaka na poziciji 0), a operacije `getChar` čita sledeći znak sa tekuće pozicije kurzora u izlazni argument c i pomera kurzor za jedno mesto

unapred. U slučaju bilo kakve greške, uključujući i pomeranje kurzora preko veličine sadržaja ili čitanje znaka kada je kurzor stigao do kraja sadržaja, operacije treba da vrate negativnu vrednost, a nulu u slučaju uspeha.

Rešenje

```
class IOStream {
public:
  IOStream (BlockIOHandle d) : dev(d), cursor(-1), curBlock(-1)
    { seek(0); }

  int seek (long offset);
  int getChar (char& c);
protected:
  int loadBlock(); // Helper; loads the block corresponding to the cursor
private:
  BlockIOHandle dev;
  char buffer[BlockSize];
  long curBlock, cursor;
};

int IOStream::loadBlock () {
  long blockNo = cursor/BlockSize;
  if (curBlock==blockNo) return 0;
  if (readBlock(dev,blockNo,buffer)<0)) {
    cursor = -1;
    return -1;
  }
  curBlock = blockNo;
  return 0;
}

int IOStream::seek (int offset) {
  cursor = offset;
  if (cursor<0 || cursor>=getSize(dev)) {
    cursor = -1;
    return -1;
  }
  return 0;
}

int IOStream::getchar (char& c) {
  if (cursor<0 || cursor>=getSize(dev)) {
    cursor = -1;
    return -1;
  }
  if (loadBlock()<0) return -1;
  c = buffer[(cursor++)%BlockSize];
  return 0;
}
```

1. zadatak, treći kolokvijum, jun 2015.

U nekom sistemu svaki drajver blokovski orijentisanog uređaja („diska“) registruje sledeću strukturu (tabelu) koja sadrži pokazivače na funkcije koje implementiraju odgovarajuće operacije sa tim uređajem:

```
typedef ... Byte; // Unit of memory
typedef ... BlkNo; // Disk block number
```

```
typedef int (*DiskOperation)(BlkNo block, Byte* buffer);

struct DiskOperationsTable {
  int isValid;
  DiskOperation readBlock, writeBlock;
  DiskOperationsTable () : isValid(0), readBlock(0), writeBlock(0) {}
};
```

Sistem organizuje tabelu registrovanih drajvera za priključene uređaje kao niz ovih struktura, s tim da polje `isValid==1` označava da je dati element niza zauzet (validan, postavljen, odnosno disk je registrovan), a 0 da je ulaz slobodan (disk nije registrovan):

```
const int MaxNumOfDisks; // Maximal number of registered disk devices
DiskOperationsTable disks[MaxNumOfDisks];
```

Sistem preslikava simbolička imena dodeljena priključenim uređajima, u obliku slova abecede, brojevima ulaza u tabeli `disks` (u opsegu od 0 do `MaxNumOfDisks-1`).

1. Realizovati funkcije:

```
int readBlock(int diskNo, BlkNo block, Byte* buffer);
int writeBlock(int diskNo, BlkNo block, Byte* buffer);
```

koje treba da pozovu odgovarajuću implementaciju operacije drajvera (polimorfno, dinamičkim vezivanjem) za zadati uređaj. (Ove funkcije poziva interna kernel nit kada opslužuje zahteve za operacijama sa diskovima, da bi inicijalizovala prenos na odgovarajućem uređaju.)

2. Realizovati funkciju koja registruje operacije drajvera za dati disk:

```
int registerDriver(int diskNo, DiskOperation read, DiskOperation write);
```

U slučaju greške, sve ovde navedene funkcije vraćaju negativnu vrednost, a u slučaju uspeha vraćaju 0.

Rešenje

```
int readBlock(int diskNo, BlkNo block, Byte* buffer) {
  if (diskNo<0 || diskNo>=MaxNumOfDisks) return -1; // Error
  if (disks[diskNo].isValid==0) return -2; // Error
  if (disks[diskNo].readBlock==0) return -3; // Error
  return (disks[diskNo].readBlock)(block,buffer);
}

int writeBlock(int diskNo, BlkNo block, Byte* buffer) {
  if (diskNo<0 || diskNo>=MaxNumOfDisks) return -1; // Error
  if (disks[diskNo].isValid==0) return -2; // Error
  if (disks[diskNo].writeBlock==0) return -3; // Error
  return (disks[diskNo].writeBlock)(block,buffer);
}

int registerDriver(int diskNo, DiskOperation read, DiskOperation write) {
  if (diskNo<0 || diskNo>=MaxNumOfDisks) return -1; // Error
  if (disks[diskNo].isValid) return -2; // Error
  disks[diskNo].isValid = 1;
  disks[diskNo].readBlock = read;
  disks[diskNo].writeBlock = write;
}
```

1. zadatak, treći kolokvijum, septembar 2015.

Na neki sekvencijalni, blokovski orijentisani izlazni uređaj identifikovan „ručkom“ `handle` se blok znakova veličine `BlockSize` ispisuje sa zadate adrese u memoriji funkcijom:

```
void writeBlock(IOHandle handle, char* addr);
```

Od ovog uređaja napraviti apstrakciju sekvencijalnog, znakovno orijentisanog izlaznog uređaja (izlazni tok), odnosno realizovati funkciju koja ispisuje znak po znak na taj uređaj:

```
void putchar(IOHandle handle, char);
```

Ignorisati sve greške.

Rešenje

```
void putchar (IOHandle handle, char c) {
 static char buffer[BlockSize];
 static int cursor = 0;
 buffer[cursor++] = c;
 if (cursor==BlockSize) {
   writeBlock(handle,buffer);
   cursor = 0;
 }
}
```

1. zadatak, treći kolokvijum, jun 2014.

U nekom sistemu svaki drajver blokovski orijentisanog uređaja („diska“) registruje sledeću strukturu (tabelu) koja sadrži pokazivače na funkcije koje implementiraju odgovarajuće operacije sa tim uređajem:

```
typedef ... Byte; // Unit of memory
typedef ... BlkNo; // Disk block number

struct DiskOperationsTable {
  int (*readBlock)(BlkNo block, Byte* buffer);
  int (*writeBlock)(BlkNo block, Byte* buffer);
};
```

Sistem organizuje tabelu registrovanih drajvera za priključene uređaje kao niz pokazivača na ove strukture:

```
const int MaxNumOfDisks; // Maximal number of registered disk devices

DiskOperationsTable* disks[MaxNumOfDisks];
```

Sistem preslikava simbolička imena dodeljena priključenim uređajima, u obliku slova abecede, brojevima ulaza u tabeli `disks` (u opsegu od 0 do `MaxNumOfDisks-1`). Ukoliko uređaj sa datim simboličkim imenom nije priključen, odgovarajući ulaz u ovoj tabeli je *null*.

Realizovati funkcije:

```
int readBlock(int diskNo, BlkNo block, Byte* buffer);
int writeBlock(int diskNo, BlkNo block, Byte* buffer);
```

koje treba da pozovu odgovarajuću implementaciju operacije drajvera (polimorfno, dinamičkim vezivanjem) za zadati uređaj. (Ove funkcije poziva interna kernel nit kada opslužuje zahteve za operacijama sa diskovima, da bi inicijalizovala prenos na odgovarajućem uređaju.) U slučaju greške, sve ovde navedene funkcije vraćaju negativnu vrednost, a u slučaju uspeha vraćaju 0.

Rešenje

```
int readBlock(int diskNo, BlkNo block, Byte* buffer) {
  if (diskNo<0 || diskNo>=MaxNumOfDisks) return -1; // Error
```

```
  if (disks[diskNo]==NULL) return -1; // Error
  return (disks[diskNo]->readBlock)(block,buffer);
}

int writeBlock(int diskNo, BlkNo block, Byte* buffer) {
  if (diskNo<0 || diskNo>=MaxNumOfDisks) return -1; // Error
  if (disks[diskNo]==NULL) return -1; // Error
  return (disks[diskNo]->writeBlock)(block,buffer);
}
```

1. zadatak, treći kolokvijum, septembar 2014.

Dat je neki sekvencijalni, blokovski orijentisani ulazni uređaj sa koga se blok znakova veličine `BlockSize` učitava na zadatu adresu funkcijom:

```
void readBlock(char* addr);
```

Od ovog uređaja napraviti apstrakciju sekvencijalnog, znakovno orijentisanog ulaznog uređaja (ulazni tok), odnosno realizovati funkciju koja učitava znak po znak sa tog uređaja:

```
char getchar();
```

Ignorisati sve greške.

Rešenje

```
char getchar () {
 static char buffer[BlockSize];
 static int cursor = BlockSize;
 if (cursor==BlockSize) {
   readBlock(buffer);
   cursor = 0;
 }
 return buffer[cursor++];
}
```

1. zadatak, treći kolokvijum, jun 2013.

Neki sistem organizuje keš blokova sa diska na sledeći način. Keš čuva najviše `CACHESIZE` blokova veličine `BLKSIZE` u nizu `diskCache`. Svaki ulaz i u nizu `diskCacheMap` sadrži broj bloka na disku koji se nalazi u ulazu i keša. Vrednost 0 u nekom ulazu niza `diskCacheMap` označava da je taj ulaz prazan (u njega nije učitan blok). Data je sledeća implementacija keša:

```
typedef ... Byte; // Unit of memory
typedef ... BlkNo; // Disk block number
const int BLKSIZE = ...; // Disk block size in Bytes
const int CACHESIZE = ...; // Disk cache size in blocks

Byte diskCache [CACHESIZE][BLKSIZE]; // Disk cache
BlkNo diskCacheMap [CACHESIZE]; // The contents of disk cache. 0 for empty

Byte* getDiskBlock (BlkNo blk) {
  // Search for the requested block in the cache and return it if found:
  int hash = blk%CACHESIZE;
  int cursor = hash;
  for (int i=0; i<CACHESIZE; i++) {
    cursor = (hash+i)%CACHESIZE;
    if (diskCacheMap[cursor]==blk) return diskCache[cursor];
    if (diskCacheMap[cursor]==0) break;
  }
```

```
  // Not found.
  if (diskCacheMap[cursor]!=0) cursor=hash; // Cache full
  // If there is a block to evict, write it to the disk:
  if (diskCacheMap[cursor]!=0)
    diskWrite(diskCacheMap[cursor],diskCache[cursor]);
  // Load the requested block:
  diskCacheMap[cursor] = blk;
  diskRead(blk,diskCache[cursor]);
  return diskCache[cursor];
}
```

Operacije `diskRead` i `diskWrite` vrše sinhrono čitanje, odnosno upis datog bloka sa diska.

Funkcija `getDiskBlock` vraća pokazivač na deo memorije u kome se nalazi učitan traženi blok diska, pri čemu se taj blok učitava u keš (uz prethodno snimanje eventualno izbačenog bloka) ukoliko taj blok veo nije u kešu. Ovu funkciju koristi ostatak sistema za pristup blokovima diska. Ova operacija implementira keš kao heš (*hash*) tabelu, s tim da koliziju rešava otvorenim adresiranjem.

Modifikovati funkciju `getDiskBlock` tako da se svaki blok na disku uvek smešta u isti ulaz u kešu, određen istom navedenom heš funkcijom.

Rešenje

```
Byte* getDiskBlock (BlkNo blk) {
  // Find the requested block in the cache and return it if present:
  int hash = blk%CACHESIZE;
  if (diskCacheMap[hash]==blk) return diskCache[hash];
  // The block is not in the cache.
  // If there is a block to evict, write it to the disk:
  if (diskCacheMap[hash]!=0)
    diskWrite(diskCacheMap[hash],diskCache[hash]);
  // Load the requested block:
  diskCacheMap[hash] = blk;
  diskRead(blk,diskCache[hash]);
  return diskCache[hash];
}
```

1. zadatak, treći kolokvijum, jun 2012.

Neki sistem organizuje keš blokova sa diska na sledeći način. Keš čuva najviše `CACHESIZE` blokova veličine `BLKSIZE` u nizu `diskCache`. Svaki ulaz i u nizu `diskCacheMap` sadrži broj bloka na disku koji se nalazi u ulazu i keša. Vrednost 0 u nekom ulazu niza `diskCacheMap` označava da je taj ulaz prazan (u njega nije učitan blok). Data je sledeća implementacija keša:

```
typedef ... Byte; // Unit of memory
typedef ... BlkNo; // Disk block number
const int BLKSIZE = ...; // Disk block size in Bytes
const int CACHESIZE = ...; // Disk cache size in blocks

BYTE diskCache [CACHESIZE][BLKSIZE]; // Disk cache
BlkNo diskCacheMap [CACHESIZE]; // The contents of disk cache. 0 for empty

int diskCacheCursor = 0; // FIFO cursor for eviction/loading

Byte* getDiskBlock (BlkNo blk) {
  // Search for the requested block in the cache and return it if found:
  for (int i=0; i<CACHESIZE; i++) {
    if (diskCacheMap[i]==blk) return diskCache[i];
    if (diskCacheMap[i]==0) break;
```

```
  }
  // Not found.
  // If there is a block to evict, write it to the disk:
  if (diskCacheMap[diskCacheCursor]!=0)
    diskWrite(diskCacheMap[diskCacheCursor],diskCache[diskCacheCursor]);
  // Load the requested block:
  diskCacheMap[diskCacheCursor] = blk;
  diskRead(blk,diskCache[diskCacheCursor]);
  Byte* ret = diskCache[diskCacheCursor];
  diskCacheCursor = (diskCacheCursor+1)%CACHESIZE;
  return ret;
}
```

Operacije `diskRead` i `diskWrite` vrše sinhrono čitanje, odnosno upis datog bloka sa diska:

```
void diskRead(BlkNo block, Byte* toBuffer);
void diskWrite(BlkNo block, Byte* fromBuffer);
```

Ukoliko je keš pun, a treba učitati novi blok, iz keša se izbacuje blok iz ulaza na koji ukazuje kurzor `diskCacheCursor` koji se pomera u krug, tako da je izbacivanje (engl. *eviction*) po FIFO (*first-in-first-out*) principu.

Funkcija `getDiskBlock` vraća pokazivač na deo memorije u kome se nalazi učitan traženi blok diska, pri čemu se taj blok učitava u keš (uz prethodno snimanje eventualno izbačenog bloka) ukoliko taj blok veo nije u kešu. Ovu funkciju koristi ostatak sistema za pristup blokovima diska.

Ova implementacija keša je nedovoljno efikasna jer se neki blok sa diska može smestiti u bilo koji ulaz keša i zato pronalaženje bloka u punom kešu često zahteva obilaženje velikog dela niza `diskCacheMap`. Zbog toga treba promeniti implementaciju ovog keša tako da se `diskCacheMap` organizuje kao heš (*hash*) tabela. Disk blok broj `b` se smešta u ulaz `hash(b)=b mod CACHESIZE`, ukoliko je taj ulaz slobodan. U slučaju da nije (tj. u slučaju kolizije), taj blok se smešta u prvi naredni slobodan blok (u krug, po modulu `CACHESIZE`; rešavanje kolizije otvorenim adresiranjem).

Dati ovako izmenjenu funkciju `getDiskBlock`.

Rešenje

```
Byte* getDiskBlock (BlkNo blk) {
  // Search for the requested block in the cache and return it if found:
  int hash = blk%CACHESIZE;
  int cursor = hash;
  for (int i=0; i<CACHESIZE; i++) {
    cursor = (hash+i)%CACHESIZE;
    if (diskCacheMap[cursor]==blk) return diskCache[cursor];
    if (diskCacheMap[cursor]==0) break;
  }
  // Not found.
  if (diskCacheMap[cursor]!=0) cursor=hash; // Cache full
  // If there is a block to evict, write it to the disk:
  if (diskCacheMap[cursor]!=0)
    diskWrite(diskCacheMap[cursor],diskCache[cursor]);
  // Load the requested block:
  diskCacheMap[cursor] = blk;
  diskRead(blk,diskCache[cursor]);
  return diskCache[cursor];
}
```

1. zadatak, treći kolokvijum, septembar 2011.

Dat je interfejs klase koja apstrahuje jedan sekvencijalni, blokovski, ulazni uređaj sa blokom veličine `BlockSize` znakova:

```
class BlockDevice {
public:
  void open ();  // Open the device channel
  void close (); // Close the device channel
  void loadBlock (char* buffer); // Read next block to the given buffer
};
```

Implementirati klasu `CharDevice` sa dole datim interfejsom koja adaptira interfejs datog blokovskog ulaznog uređaja u interfejs sekvencijalnog, ali znakovnog ulaznog uređaja. Konstruktor i destruktor ove klase treba da otvaraju, odnosno zatvaraju dati blokovski uređaj.

```
class CharDevice {
public:
  CharDevice (BlockDevice* bd);
 ~CharDevice ();
  char getChar ();
};
```

Korišćenjem klase `CharDevice` ilustrovati primerom učitavanje niza znakova sa ulaznog uređaja sve dok se ne učita znak `'\0'`.

Rešenje

```
class CharDevice {
public:
  CharDevice (BlockDevice* bd) : myDev(bd), cursor(BlockSize) {
    if (myDev) myDev->open();
  }

 ~CharDevice () { if (myDev) myDev->close(); }

  char getChar ();

protected:
  void load ();

private:
  BlockDevice* myDev;
  char buffer[BlockSize];
  int cursor;
};

void CharDevice::load () {
  if (myDev && cursor==BlockSize) {
    myDev->loadBlock(buffer);
    cursor=0;
  }
}

char CharDevice::getChar () {
  load();
  if (cursor<BlockSize) return buffer[cursor++];
  else return '\0'; // Exception;
}

void main () {
  BlockDevice bd = ...;
  CharDevice input(bd);
```

```
  for (char c=input.getchar(); c!='\0'; c=input.getchar())
    ...
}
```

Komandna linija

2. zadatak, treći kolokvijum, jun 2017.

Sistemi bazirani na sistemu Unix podržavaju strukture direktorijuma tipa acikličnog usmerenog grafa (DAG) pomoću dve vrste referenci na fajl kao objekat u fajl sistemu:

- *soft* (*symbolic*) *link*: „meka (ili simboličkа) veza“ predstavlja fajl posebnog tipa čiji sadržaj čuva proizvoljnu (apsolutnu ili relativnu) putanju do nekog drugog fajla, poput „prečice“ (*shortcut*); svaku operaciju (komandu) nad ovakvom vezom sistem preusmerava na referencirani fajl, osim komande `rm` za brisanje – brisanje simboličke veze briše samu vezu, ne i referencirani fajl; ukoliko referencirani fajl nestane ili se premesti, veza ostaje neažurna, „viseća“;
- *hard link*: „tvrda veza“ predstavlja jedan ulaz u direktorijumu koji referencira određeni fajl kao objekat (tj. njegov *inode*); na jedan fajl može ukazivati više tvrdih veza; komanda `rm` uklanja tvrdu vezu, a sam fajl se implicitno briše iz sistema kada nestane poslednja tvrda veza na njega.

Iz komandne linije mogu se izvršiti sledeće sistemske komande:

`ls`	prikazuje sadržaj tekućeg direktorijuma (*list*)
`cd <dir>`	menja tekući direktorijum (*change directory*)
`ln <src> <dst>`	za postojeći fajl sa zadatom stazom `<src>` kreira novu tvrdu vezu sa datom stazom `<dst>` (*link*)
`ln -s <src> <dst>`	za postojeći fajl sa zadatom stazom `<src>` kreira novu meku vezu sa datom stazom `<dst>`
`rm <file>`	briše ulaz sa zadatim imenom iz tekućeg direktorijuma; ukoliko je to poslednja tvrda veza na fajl, briše se i sam fajl.

Sve staze mogu biti apsolutne ili relativne. Zabeležena je sledeća sesija jednog korisnika:

```
$ cd /home/docs
$ ls
bar foo txt
$ cd /home/pics
$ ls
jane john chld
$ ln /home/docs/bar foo
$ rm john
$ cd /home/docs
$ rm bar
$ cd /home/pics
$ ls
```

1. Napisati izlaz poslednje komande.
2. Nakon prikazane sekvence zadate su sledeće komande:

```
$ ln -s /home/docs/foo bar
$ rm /home/docs/foo
$ rm foo
$ cd /home/docs
$ ls
```

Napisati izlaz poslednje komande.

Rešenje

1. jane chld foo
2. txt

2. zadatak, treći kolokvijum, septembar 2012.

Neki fajl sistem podržava strukture direktorijuma tipa acikličnog usmerenog grafa (DAG). Iz komandne linije mogu se izvršiti sledeće sistemske komande:

`dir`	prikazuje sadržaj tekućeg direktorijuma
`cd <dir>`	menja tekući direktorijum
`alias <file> <newname>`	za postojeći fajl sa zadatom stazom `<file>` kreira novi ulaz u tekućem direktorijum pod datim novim imenom `<newname>` (*link*)
`rem <file>`	briše ulaz sa zadatim imenom iz tekućeg direktorijuma; ukoliko je to poslednja referenca na fajl, briše se i sam fajl.

Zabeležena je sledeća sesija jednog korisnika:

```
$ cd /home/docs
$ dir
bar <file>
foo <file>
txt <dir>
$ cd /home/pics
$ dir
jane <file>
john <file>
chld <dir>
$ alias /home/docs/bar foo
$ rem john
$ cd /home/docs
$ rem bar
$ cd /home/pics
$ dir
```

1. Napisati izlaz poslednje komande.
2. Nakon prikazane sekvence zadate su sledeće komande:

   ```
   $ rem foo
   $ cd /home/docs
   $ dir
   ```

   Napisati izlaz poslednje komande.

Rešenje

1. ```
   jane <file>
   chld <dir>
   foo <file>
   ```
2. ```
   foo <file>
   txt <dir>
   ```

Fajl sistem (interfejs)

2. zadatak, treći kolokvijum, jun 2022.

Neprazno binarno stablo čiji su čvorovi tipa `Node` zapisano je u binarni fajl na sledeći način (prikazan je pojednostavljen kod bez obrade grešaka): za svaki čvor najpre je zapisan njegov sadržaj tipa `NodeData`, zatim jedan `int` indikator koji kaže da li taj čvor ima svoje levo podstablo, zatim jedan `int` indikator koji kaže da li taj čvor ima svoje desno podstablo, a onda isto tako redom celo levo podstablo ako ga ima, pa celo desno podstablo ako ga ima. Korišćenjem istog POSIX API za fajlove napisati kod funkcije `readTree` za učitavanje i izgradnju stabla iz takvog fajla. Prvi parametar je staza do fajla, a drugi parametar je adresa pokazivača u koji treba upisati adresu korenog čvora formiranog i učitanog stabla. Obraditi greške vraćanjem negativne vrednosti. POSIX API funkcija

```
ssize_t read(int fd, void *buf, size_t count);
```

vraća broj stvarno učitanih bajtova (može biti manji od zahtevanog ako se stigne do kraja fajla) ili negativnu vrednost u slučaju greške.

```
#include <fcntl.h>

struct Node {
    NodeData data;
    Node *left, *right;
};
void writeSubtree(int fd, Node* node) {
    write(fd, node->data, sizeof(NodeData));
    int ind = (node->left != 0);
    write(fd, ind, sizeof(int));
    ind = (node->right != 0);
    write(fd, ind, sizeof(int));
    if (node->left) {
        writeSubtree(fd, node->left);
    }
    if (node->right) {
        writeSubtree(fd, node->right);
    }
}
void writeTree(const char* pathname, Node* root) {
    int fd = open(*pathname, O_WRONLY | O_CREATE | O_TRUNC);
    int r = writeSubtree(fd, root);
    close(fd);
}
int readTree(const char* pathname, Node** root);
```

Rešenje

```
#include <fcntl.h>
int readSubtree(int fd, Node** node) {
    static NodeData data;
    static int left, right;
    ssize_t cnt = read(fd, &data, sizeof(NodeData));
    if (cnt<sizeof(NodeData)) {
        return -2;
    }
    cnt = read(fd, &left, sizeof(int));
    if (cnt<sizeof(int)) {
        return -2;
    }
    cnt = read(fd, &right, sizeof(int));
    if (cnt < sizeof(int)) {
```

```
        return -2;
    }
    *node = new Node();
    if (*node == 0) {
        return -3;
    }
    (*node)->data = data;
    (*node)->left = (*node)->right = 0;
    int r = 0;
    if (left) {
        r = readSubtree(fd, &((*node)->left));
        if (r < 0) {
            return r;
        }
    }
    if (right) {
        r = readSubtree(fd, &((*node)->right));
        if (r < 0) {
            return r;
        }
    }
    return 0;
}
int readTree(const char* pathname, Node** root) {
    int fd = open(*pathname, O_RDONLY);
    if (fd < 0) {
        return -1;
    }
    int r = readSubtree(fd, root);
    close(fd);
    return r;
}
```

4. zadatak, kolokvijum, jun 2021.

Na raspolaganju je POSIX API za fajlove:

```
int open(const char* path, int flags);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
```

Tip `ssize_t` je označen celobrojni tip, isti kao `size_t`, samo što može da prihvati i vrednost -1. Funkcija `read` vraća stvarno učitan broj bajtova (0 ili više); ako je taj broj manji od `count`, stiglo se do kraja fajla. Funkcije `open`, `close` i `read` vraćaju -1 u slučaju greške.

Korišćenjem ovog interfejsa realizovati apstrakciju `IFStream` kao klasu čiji je interfejs dat dole. Ova klasa apstrahuje ulazni znakovni tok vezan za fajl. Staza do fajla zadaje se parametrom konstruktora. Zatvaranje fajla treba obezbediti destruktorom. Operacija `getc` vraća jedan znak učitan sa toka. Kada se došlo do kraja fajla, operacija `eof` vraća `true`. Ukoliko je u nekoj ranijoj operaciji sa tokom, uključujući i otvaranje, došlo do greške, funkcije `err` i `eof` vraćaju `true`. Ukoliko je tok u stanju greške ili se stiglo do njegovog kraja, operacija `getc` treba da vraća `'\-1'`. Učitavanje treba da bude što je moguće efikasnije u smislu da se sa uređaja učitava najmanji mogući broj blokova za sekvencijalni pristup. Veličina bloka u bajtovima je `BLOCK_SIZE`, a veličina znakova u bajtovima je `CHAR_SIZE`.

```
class IFStream {
public:
    IFStream(const char* path);
    ~IFStream();
    bool eof() const;
    bool err() const;
```

```
    char getc();
private:
    static const size_t BLOCK_SIZE = ..., CHAR_SIZE = 2;
};
```

Ova apstrakcija može se koristiti ovako:

```
IFStream str("myfile.txt");
while (!str.err() && !str.eof()) {
    char c = str.getc();
    ...
}
```

Rešenje

```
#include <sys/stat.h>
#include <fcntl.h>

class IFStream {
public:
    IFStream(const char* path);
    ~IFStream() {
        if (fd >= 0) close(fd);
    }
    bool eof() const { return isEOF; }
    bool err() const { return isErr; }
    char getc();
protected:
    inline void fetch();
private:
    static const size_t BLOCK_SIZE = ..., CHAR_SIZE = 2;
    int fd;
    bool isEOF, isErr;
    char buffer[(BLOCK_SIZE + CHAR_SIZE - 1)/CHAR_SIZE];
    ssize_t curPos, size;
};

IFStream::IFStream(const char* path) : isEOF(false), isErr(false),
                                       curPos(-1), size(-1) {
    fd = open(path, O_RDONLY);
    if (fd == -1) {
        isErr = isEOF = true;
        return;
    }
    fetch();
}

inline void IFStream::fetch() {
    size = read(fd, buffer, BLOCK_SIZE);
    if (size == -1) isErr = true;
    if (size <= 0) isEOF = true;
    curPos = 0;
}

char IFStream::getc() {
    if (!isErr && !isEOF && curPos < size) {
        char c = buffer[curPos++];
        if (curPos >= size)
```

```
        if (size < BLOCK_SIZE) isEOF = true;
        else fetch();
        return c;
    }
    return '\-1';
}
```

4. zadatak, kolokvijum, jul 2020.

Dva procesa, pošiljalac i primalac, komuniciraju slanjem, odnosno prijemom „poruke“ preko fajla kao deljenog objekta. Kada pripremi poruku određene veličine u svom baferu, pošiljalac kreira fajl sa nazivom „*buffer.bin*“ u roditeljskom direktorijumu svog tekućeg direktorijuma, upiše poruku u taj fajl i zatvori fajl. Kada primalac „vidi“ ovaj fajl, on iz njega pročita poslatu poruku, a potom obriše taj fajl. Zbog ovakve sinhronizacije, pošiljalac treba da funkcioniše ovako: ukoliko pri pokušaju kreiranja navedenog fajla zaključi da taj fajl već postoji (jer ga primalac joc uvek nije obrisao), pošiljalac se uspavljuje 5 sekundi, ostavljajući priliku primaocu da pročita i obriše fajl, a onda pokušava ponovo. Oba procesa izvršavaju se u ime istog korisnika. Napisati kod funkcije

```
int send(const void *buffer, size_t size);
```

procesa pošiljaoca koju on poziva kada želi da pošalje poruku datu u baferu zadate veličine. Ukoliko dođe do bilo koje greške koju ne može da obradi, ova funkcija treba da vrati -1, a u slučaju uspeha vraća 0. Na raspolaganju su sledeći standardni POSIX sistemski pozivi:

- `unsigned int sleep(unsigned int sleep_time_in_seconds)`: uspavljuje (suspenduje) pozivajući proces na zadato vreme (u sekundama);
- `int close (int fd)`: zatvara fajl sa zadatim deskriptorom;
- `int write (int fd, const void *buffer, size_t size)`: upisuje dati sadržaj u fajl sa zadatim deskriptorom;
- `int open(const char *pathname, int flags, mode_t mode)`: otvara fajl sa zadatom putanjom; argument *flags* mora da uključi jedno od sledećih prava pristupa: *O_RDONLY*, *O_WRONLY*, ili *O_RDWR*. Ako je uključen i fleg *O_CREAT*, fajl će biti kreiran ukoliko ne postoji; ako je pritom uključen i *O_EXCL*, a fajl već postoji, funkcija će vratiti grešku (-1), a kod greške postavljen u globalnoj sistemskoj promenljivoj *errno* biće jednak *EEXIST*. Argument mode definiše prava pristupa za fajl koji se kreira samo u slučaju da je uključen *O_CREAT*, i to na sledeći način:

S_IRWXU	0x0700	user (file owner) has read, write and execute permission
S_IRUSR	0x0400	user has read permission
S_IWUSR	0x0200	user has write permission
S_IXUSR	0x0100	user has execute permission
S_IRWXG	0x0070	group has read, write and execute permission
S_IRGRP	0x0040	group has read permission
S_IWGRP	0x0020	group has write permission
S_IXGRP	0x0010	group has execute permission
S_IRWXO	0x0007	others have read, write and execute permission
S_IROTH	0x0004	others have read permission
S_IWOTH	0x0002	others have write permission
S_IXOTH	0x0001	others have execute permission

Rešenje

```
const char* fname = "../buffer.bin";
const unsigned int SLEEP_TIME = 5;

int send (const void* buffer, size_t size) {
  int fd;
  while (1) {
    fd = open(fname,O_WRONLY|O_CREAT|O_EXCL,S_IRUSR|S_IWUSR);
    if (fd>=0) break;
    if (errno!=EEXIST) return -1;
```

```
    sleep(SLEEP_TIME);
  }
  ret = write(fd,buffer,size) < 0 ? -1:0;
  ret |= close(fd);
  return ret;
}
```

2. zadatak, treći kolokvijum, jun 2019.

U nekom binarnom fajlu zapisano je ogromno binarno stablo; stablo je potpuno balansirano, ima n nivoa i tačno $2^n - 1$ čvorova, tj. svaki čvor osim lista ima tačno dva deteta. U svakom čvoru nalazi se jedna jedinstvena vrednost tipa `int`, a stablo je sortirano (levi potomci su manji, a desni veći od svakog čvora). Stablo je zapisano u sadržaju fajla kao niz, tako da svaki element niza sadrži samo vrednost u odgovarajućem čvoru, dok su deca čvora koji odgovara elementu i implicitno određena i nalaze se u elementima $2i + 1$ i $2i + 2$ tog niza; koren je u elementu 0.

Korišćenjem dole datih sistemskih poziva standardnog C fajl interfejsa, implementirati funkciju

```
int binary_search (const char* filename, unsigned n, int x);
```

koja u binarnom fajlu sa datim imenom, u kome je zapisano stablo sa n nivoa u opisanom formatu, binarnom pretragom traži vrednost datu x i vraća 1 ako je pronađe, odnosno 0 ako je ne pronađe. Ignorisati sve potencijalne greške u sistemskim pozivima. Sledeće funkcije deklarisane su u zaglavlju `cstdio`:

- `std::FILE* std::fopen(const char* filename, const char* mode)`; otvara fajl sa zadatim imenom u zadatom modalitetu; za čitanje je modalitet „r“, za upis „w“, a za otvaranje fajla u binarnom režimu treba dodati sufiks „b“ na modalitet;
- `int std::fclose(FILE*)`; zatvara dati fajl;
- `std::size_t fread(void* buffer, std::size_t size, std::size_t count, std::FILE* stream)`; iz datog fajla, počev od tekuće pozicije kurzora, u zadati bafer učitava najviše `count` objekata, svaki veličine `size` i pomera kurzor fajla iza pročitanog sadržaja; vraća broj stvarno pročitanih objekata (manje od traženog u slučaju greške ili nailaska na kraj fajla);
- `int fseek(std::FILE* stream, long offset, int origin)`; pomera kurzor datog fajla na pomeraj (poziciju) zadatu parametrom `offset` (u bajtovima, odnosno jedinicama koje vraća operator `sizeof`), u odnosu na položaj zadat parametrom `origin`; za pomeraj u odnosu na početak fajla, ovaj parametar treba da bude jednak konstanti `SEEK_SET`.

Rešenje

```
#include <cstdio>
using namespace std;

int binary_search (const char* filename, unsigned n, int x) {
  FILE* f = fopen(filename, "rb");
  long nodeIndex = 0;
  int nodeValue;
  for (unsigned i=0; i<n; i++) {
    fseek(f,nodeIndex*sizeof(nodeValue),SEEK_SET);
    fread(&nodeValue,sizeof(nodeValue),1,f);
    if (x==nodeValue) { fclose(f); return 1; }
    if (x<nodeValue)
      nodeIndex = 2*nodeIndex + 1;
    else
      nodeIndex = 2*nodeIndex + 2;
  }
  fclose(f);
  return 0;
}
```

2. zadatak, treći kolokvijum, jun 2018.

Dole je dat izvod iz dokumentacije za API za fajlove u GNU sistemima. Date deklaracije su u `<unistd.h>` i `<fctl.h>`. Korišćenjem samo dole datih funkcija, realizovati sledeću funkciju koja prepisuje ceo sadržaj datog ulaznog fajla proizvoljne veličine u dati izlazni fajl, korišćenjem svog bafera određene veličine. U slučaju bilo kakve greške, funkcija treba da vrati negativnu vrednost, u suprotnom treba da vrati 0.

```
int fcopy (const char *filenamefrom, const char *filenameto);
```

```
int open (const char *filename, int flags);
```

Creates and returns a new file descriptor for the file named by ***filename****. Initially, the file position indicator for the file is at the beginning of the file. The* ***flags*** *argument controls how the file is to be opened. This is a bit mask; you create the value by the bitwise OR of the appropriate parameters (using the '|' operator in C) – the following macros:*

- `O_RDONLY`: *Open the file for read access.*
- `O_WRONLY`: *Open the file for write access.*
- `O_RDWR`: *Open the file for both reading and writing.*
- `O_CREAT`: *If set, the file will be created if it doesn't already exist.*
- `O_APPEND`: *The bit that enables append mode for the file. If set, then all write operations write the data at the end of the file, extending it, regardless of the current file position.*
- `O_TRUNC`: *Truncate the file to zero length.*

The normal return value from ***open*** *is a non-negative integer file descriptor. In the case of an error, a value of -1 is returned instead.*

```
int close (int filedes);
```

Closes the file descriptor ***filedes****. The normal return value is 0. In the case of an error, a value of -1 is returned.*

```
ssize_t
```

This data type is used to represent the sizes of blocks that can be read or written in a single operation. It is similar to ***size_t****, but must be a signed type.*

```
ssize_t read (int filedes, void *buffer, size_t size);
```

Reads up to ***size*** *bytes from the file with descriptor* ***filedes****, storing the results in the* ***buffer****. (This is not necessarily a character string, and no terminating null character is added.)*

The return value is the number of bytes actually read. This might be less than ***size****; for example, if there aren't that many bytes left in the file or if there aren't that many bytes immediately available. The exact behavior depends on what kind of file it is. Note that reading less than* ***size*** *bytes is not an error.*

A value of zero indicates end-of-file (except if the value of the ***size*** *argument is also zero). This is not considered an error. If you keep calling read while at end-of-file, it will keep returning zero and doing nothing else.*

If ***read*** *returns at least one character, there is no way you can tell whether end-of-file was reached. But if you did reach the end, the next read will return zero. In case of an error,* ***read*** *returns -1.*

```
ssize_t write (int filedes, const void *buffer, size_t size);
```

Writes up to ***size*** *bytes from* ***buffer*** *to the file with descriptor* ***filedes****. The data in* ***buffer*** *is not necessarily a character string and a null character is output like any other character. The return value is the number of bytes actually written. This may be* ***size****, but can always be smaller. Your program should always call write in a loop, iterating until all the data is written. In the case of an error,* ***write*** *returns -1.*

Rešenje

```
#include <unistd.h>
#include <fctl.h>

int fcopy (const char *filenamefrom, const char *filenameto) {
  static const int BufferSize = 1024;
  char buffer[BufferSize];
```

```
  int ffrom = open(filenamefrom,O_RDONLY);
  if (ffrom<0) return ffrom;

  int fto = open(filenameto,O_WRONLY|O_CREAT|O_APPEND|O_TRUNC);
  if (fto<0) {
    close(ffrom);
    return fto;
  }

  do {
    ssize_t numRead = read(ffrom,buffer,BufferSize);
    if (numRead<0) {
      close(ffrom);
      close(fto);
      return numRead;
    }
    ssize_t numWritten = 0;
    while (numWritten<numRead) {
      ssize_t nw = write(fto,&buffer[numWritten],numRead-numWritten);
      if (nw<0) {
        close(ffrom);
        close(fto);
        return nw;
      }
      numWritten += nw;
    }
  while (numRead!=0);

  close(ffrom);
  close(fto);
  return 0;
}
```

2. zadatak, treći kolokvijum, jun 2016.

U implementaciji nekog fajl sistema svaki čvor u hijerarhijskoj strukturi direktorijuma i fajlova predstavljen je objektom klase `Node`. Operacija te klase:

```
Node* Node::getSubnode(const char* pStart, const char* pEnd);
```

vraća podčvor datog čvora `this` koji ima simboličko ime zadato nizom znakova koji počinje znakom na koga ukazuje `pStart`, a završava znakom ispred znaka na koga ukazuje `pEnd` (`pEnd` može ukazivati na `'\0'` ili `'/'`). Ukoliko dati čvor `this` nije direktorijum, ili u njemu ne postoji podčvor sa datim simboličkim imenom, ova funkcija vraća 0.

Za svaki proces se u polju `curDir` strukture PCB čuva pokazivač na čvor (tipa `Node*`) koji predstavlja tekući direktorijum datog procesa. Koreni direktorijum cele hijerarhije dostupan je kao statički pokazivač `Node::rootNode` tipa `Node*`.

Znak za razdvajanje (delimiter) u stazama, kao i znak za koreni direktorijum je kosa crta `'/'`. Implementirati sledeću funkciju koja se koristi u implementaciji ovog fajl sistema:

```
Node* Node::getNode (PCB* pcb, const char* path);
```

Ova funkcija vraća čvor određen stazom koja je zadataka drugim argumentom, pri čemu ta staza može biti zadata kao apsolutna (počinje znakom `'/'`), ili kao relativna (ne počinje znakom `'/'`) u odnosu na tekući direktorijum procesa čiji je PCB dat kao prvi argument.

Rešenje

```
Node* Node::getNode (PCB* pcb, const char* path) {
  static const char delimiter = '/';
  if (pcb==0 || path==0) return 0; // Exception!
  Node* node = 0;
  const char* pStart = path;
  if (*pStart==delimiter) {
    node = rootNode;
    pStart++;
  } else {
    node = pcb->curDir;
  }
  while (node && *pStart) {
    const char* pEnd = pStart+1;
    while (*pEnd && *pEnd!=delimiter) pEnd++;
    node = node->getSubnode(pStart,pEnd);
    pStart = (*pEnd)?(pEnd+1):pEnd;
  };
  return node;
}
```

2. zadatak, treći kolokvijum, septembar 2016.

Dole je dat izvod iz dokumentacije za API prema znakovnim tokovima (*character streams*) vezanim za fajlove u GNU sistemima. Sve date deklaracije su u `<stdio.h>`. Korišćenjem samo dole datih funkcija, realizovati sledeću funkciju koja prepisuje ceo sadržaj datog ulaznog fajla u dati izlazni fajl, ali u obrnutom redosledu (poretku) znakova. U slučaju bilo kakve greške, funkcija treba da vrati negativnu vrednost, u suprotnom treba da vrati 0.

```
int fcopyreverse (const char *filenamefrom, const char *filenameto);
```

```
FILE* fopen (const char *filename, const char *opentype);
```

Opening a file with the ***fopen*** *function creates a new stream and establishes a connection between the stream and a file. This may involve creating a new file. The* ***fopen*** *function opens a stream for I/O to the file* ***filename****, and returns a pointer to the stream. If the open fails,* ***fopen*** *returns a null pointer. The* ***opentype*** *argument is a string that controls how the file is opened and specifies attributes of the resulting stream. It must begin with one of the following sequences of characters:*

- `'r'` - *Open an existing file for reading only.*
- `'w'` - *Open the file for writing only. If the file already exists, it is truncated to zero length. Otherwise a new file is created.*
- `'a'` - *Open a file for append access; that is, writing at the end of file only. If the file already exists, its initial contents are unchanged and output to the stream is appended to the end of the file. Otherwise, a new, empty file is created.*
- `'r+'` - *Open an existing file for both reading and writing. The initial contents of the file are unchanged and the initial file position is at the beginning of the file.*
- `'w+'` - *Open a file for both reading and writing. If the file already exists, it is truncated to zero length. Otherwise, a new file is created.*
- `'a+'` - *Open or create file for both reading and appending. If the file exists, its initial contents are unchanged. Otherwise, a new file is created. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.*

```
int fclose (FILE *stream);
```

This function causes ***stream*** *to be closed and the connection to the corresponding file to be broken. Any buffered output is written and any buffered input is discarded. The* ***fclose*** *function returns a value of 0 if the file was closed successfully, and* ***EOF*** *if an error was detected. It is important to check for errors when you call* ***fclose*** *to close an output stream, because real, everyday errors can be detected at this time. For example, when* ***fclose*** *writes the*

remaining buffered output, it might get an error because the disk is full. Even if you know the buffer is empty, errors can still occur when closing a file if you are using NFS.

```
int fseek (FILE *stream, long int offset, int whence);
```

*The **fseek** function is used to change the file position of the stream **stream**. The value of **whence** must be one of the constants **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**, to indicate whether the **offset** is relative to the beginning of the file, the current file position, or the end of the file, respectively. This function returns a value of zero if the operation was successful, and a nonzero value to indicate failure. A successful call also clears the end-of-file indicator of stream and discards any characters that were "pushed back" by the use of **ungetc**. **fseek** either flushes any buffered output before setting the file position or else remembers it so it will be written later in its proper place in the file.*

```
int fgetc (FILE *stream);
```

*This function reads the next character as an **unsigned char** from the stream **stream** and returns its value, converted to an **int**. If an end-of-file condition or read error occurs, **EOF** is returned instead.*

```
int fputc (int c, FILE *stream);
```

*The **fputc** function converts the character **c** to type **unsigned char**, and writes it to the **stream** stream. **EOF** is returned if a write error occurs; otherwise the character **c** is returned.*

Rešenje

```
#include <stdio.h>

int fcopyreverse (const char* filenamefrom, const char* filenameto) {
  File* from = fopen(filenamefrom,"r");
  if (from==NULL) return -1;  // Error with opening input file
  File* to = fopen(filenameto,"w");
  if (to==NULL) return -2;  // Error with opening output file
  int ret = 0;
  ret = fseek(from,-1,SEEK_END);
  while (ret==0) {
    int c = fgetc(from);
    if (c==EOF) return -3; // Error with reading input file
    ret = fputc(c,to);
    if (ret==EOF) return -4; // Error with writing to output file
    ret = fseek(from,-2,SEEK_CUR);
  }
  ret = fclose(from);
  if (ret==EOF) return -5; // Error with closing input file
  ret = fclose(to);
  if (ret==EOF) return -6; // Error with closing output file
  return ret;
}
```

2. zadatak, treći kolokvijum, jun 2015.

Neki fajl sistem primenjuje deljene (*shared*) i ekskluzivne (*exclusive*) ključeve za pristup fajlu. Za operacije čitanja (*read*, *r*) i izvršavanja (*execute*, *x*) fajla potreban je deljeni ključ, a za operaciju upisa (*write*, *w*) ekskluzivni ključ.

Tokom operacije otvaranja fajla, operativni sistem proverava da li je operacija koju je zahtevao proces dozvoljena (u smislu ključa) i zaključava fajl odgovarajućim ključem ukoliko jeste dozvoljena. Ovo se obavlja u sledećoj funkciji `lock`:

```
int lock(FCB* file, unsigned int op);
const int OP_RD = 4;
const int OP_WR = 2;
const int OP_EX = 1;
```

Zahtevana operacija se identifikuje jednim od tri bita *rwx* u argumentu `op`, s tim što je uvek tačno jedan bit postavljen na 1. Za potrebe maskiranja tih bita definisane su konstante `OP_RD`, `OP_WR` i `OP_EX`. Funkcija treba da vrati 1 ako je operacija dozvoljena, a 0 ako nije.

Pre poziva ove funkcije, FCB traženog fajla je već učitan u memoriju i na njega ukazuje prvi argument. U toj strukturi, pored ostalog, postoje i celobrojna polja `sharedLock` i `exclLock` za deljeni i ekskluzivni ključ nad datim fajlom (1-zaključan, 0-otključan). Realizovati funkciju `lock`.

Rešenje

```
int lock(FCB* f, unsigned int op) {
  if (f==0) return -1; // Exception!
  if ((op & OP_WR) && (!f->sharedLock && !f->exclLock))
    return f->exclLock = 1;
  if ((op & (OP_RD|OP_EX)) && !f->exclLock)
    return f->sharedLock = 1;
  return 0;
}
```

2. zadatak, treći kolokvijum, septembar 2015.

U nekom fajl sistemu postoje, između ostalih, i sledeći sistemski pozivi za osnovne operacije sa fajlom:

```
int fgetsize (int fhandle, unsigned long& size);
int fresize (int fhandle, unsigned long newsize);
int fmoveto (int fhandle, unsigned long offset);
int fwrite (int fhandle, byte* buffer, unsigned long size);
```

Operacija `fgetsize` u izlazni argument `size` upisuje trenutnu veličinu sadržaja fajla, a operacija `fresize` menja veličinu sadržaja fajla na novu zadatu veličinu, koja može biti i veća i manja od trenutne; ukoliko je manja, sadržaj na kraju se odseca, a ukoliko je veća, sadržaj koji se dodaje je nedefinisan. Operacija `fmoveto` pomera kurzor na zadatu poziciju bajta (numeracija bajtova sadržaja fajla počinje od 0). Sve ove operacije u slučaju greške vraćaju negativnu vrednost sa kodom greške.

Realizovati operaciju koja proširuje sadržaj fajla datim sadržajem (dodaje ga na kraj):

```
int append (int fhandle, byte* buffer, unsigned long size);
```

Rešenje

```
int append (int fhandle, byte* buffer, unsigned long sz) {
  int ret = 0;
  unsigned long oldSize = 0;
  ret = fgetsize(fhandle,oldSize);
  if (ret<0) return ret;
  unsigned long newSize = oldSize + sz;
  ret = fresize(fhandle,newSize);
  if (ret<0) return ret;
  ret = fmoveto(fhandle,oldSize);
  if (ret<0) return ret;
  ret = fwrite(fhandle,buffer,sz);
  return ret;
}
```

2. zadatak, treći kolokvijum, jun 2014.

U nekom fajl sistemu primenjuje se zaštita pristupa fajlovima kao u sistemu Unix. U strukturi FCB postoje sledeća polja:

- `unsigned long int owner`: identifikator korisnika koji je vlasnik fajla;

- `unsigned long int group`: identifikator grupe korisnika kojoj je fajl pridružen;
- `unsigned int protection`: biti prava pristupa (relevantno je samo 9 najnižih bita, pri čemu najviša 3 su dodeljena vlasniku, naredna 3 grupi, a najniža 3 ostalima).

U strukturi UCB (*user control block*) koja predstavlja jednog registrovanog korisnika sistema takođe postoji polje `group` koje predstavlja identifikator grupe kojoj je taj korisnik pridružen. Prava pristupa za grupu kojoj je pridružen fajl odnose se na korisnike koji su pridruženi istoj toj grupi.

Na raspolaganju je i sledeća funkcija koja vraća pokazivač na odgovarajuću strukturu UCB za korisnika koji je identifikovan datim identifikatorom:

```
UCB* getUCB (unsigned long int uid);
```

Realizovati funkciju koja ispituje da li je tražena operacija dozvoljena datom korisniku za dati fajl. Operacija se identifikuje jednim od tri najniža bita *rwx*, sa značenjem kao u bitima prava pristupa za fajl, s tim što je uvek samo jedan bit postavljen na 1. Funkcija treba da vrati 1 ako je operacija dozvoljena, a 0 ako nije.

```
int isAllowed(FCB* file, unsigned long int uid, unsigned int op);
```

Rešenje

```
int isAllowed(FCB* f, unsigned long int uid, unsigned int op) {
  if (f==0) return 0; // Exception!
  unsigned int prot = f->protection;
  if (uid==f->owner)
    prot >>= 6;
  else {
    UCB* usr = getUCB(uid);
    if (usr==0) return 0; // Exception!
    if (usr->group==f->group) prot >>= 3;
  } else
    prot &= 7;
  return (op&prot)?1:0;
}
```

2. zadatak, treći kolokvijum, septembar 2014.

U nekom fajl sistemu postoje sledeći sistemski pozivi za osnovne operacije sa fajlom:

```
int open (const char *pathname, int flags, mode_t mode);
int close (int fhandle);
int read (int fhandle, byte* buffer, unsigned long size);
int write (int fhandle, byte* buffer, unsigned long size);
```

Sve ove operacije u slučaju greške vraćaju negativnu vrednost sa kodom greške. U slučaju uspeha, operacija otvaranja fajla vraća „ručku“ fajla (engl. *file handle*), a ostale vraćaju 0. Realizovati objektno orijentisani „omotač“ ovog interfejsa, odnosno implementirati apstrakciju fajla kao klasu sa sledećim interfejsom:

```
class File {
public:
  File (const char *pathname, int flags, mode_t mode) throw Exception;
 ~File () throw Exception;

  void read (byte* buffer, unsigned long size) throw Exception;
  void write (byte* buffer, unsigned long size) throw Exception;
};
```

Prilikom kreiranja objekta ove klase treba implicitno otvoriti fajl, a prilikom uništavanja objekta treba implicitno zatvoriti fajl. U slu čaju greške, sve ove operacije treba da podignu izuzetak definisanog tipa `Exception`. Instance ovog tipa (klase) mogu se inicijalizovati celobrojnim kodom greške koju vraćaju sistemski pozivi.

Rešenje

```
class File {
public:
  File (const char *pathname, int flags, mode_t mode) throw Exception;
 ~File () throw Exception;

  void read (byte* buffer, unsigned long size) throw Exception;
  void write (byte* buffer, unsigned long size) throw Exception;

private:
  int fh;
};

File::File (const char *p, int f, mode_t m) throw Exception : fh(0) {
  int s = open(p,f,m);
  if (s<0) throw Exception(s);
  fh=s;
}

File::~File () throw Exception {
  int s = close(fh);
  if (s<0) throw Exception(s);
}

void File::read (byte* b, unsigned long sz) throw Exception {
  int s = read(fh,b,sz);
  if (s<0) throw Exception(s);
}

void File::write (byte* b, unsigned long sz) throw Exception {
  int s = write(fh,b,sz);
  if (s<0) throw Exception(s);
}
```

2. zadatak, treći kolokvijum, jun 2013.

U nekom interfejsu fajl sistema definisano je nekoliko funkcija za rad sa direktorijumima. Dat je izvod iz dokumentacije jedne od njih:

```
struct dirent * readdir (DIR *dirstream);
```

This function reads the next entry from the directory ***`dirstream`****... If there are no more entries in the directory or an error is detected,* ***`readdir`*** *returns a null pointer.*

Prevod: Ova funkcija čita sledeći ulaz u direktorijumu `dirstream`... Ako više nema ulaza u direktorijumu ili je došlo do greške, `readdir` vraća *null* pokazivač.

Napomena: simbol `.` označava tekući direktorijum.

Šta radi sledeći program?

```
#include <stddef.h>
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

int main (void)
{
  DIR *dp;
```

```
  struct dirent *ep;

  dp = opendir ("./");
  if (dp != NULL)
    {
      while (ep = readdir (dp))
        puts (ep->d_name);
      closedir (dp);
    }
  else
    puts ("Couldn't open the directory.");

  return 0;
}
```

Rešenje

Na standardni izlaz ispisuje sadržaj tekućeg direktorijuma, tačnije, nazive ulaza u tekućem direktorijumu (po jedan u redu).

2. zadatak, treći kolokvijum, septembar 2013.

Dat je izvod iz originalne dokumentacije za Unix/Linux sistemski poziv za otvaranje fajla:

```
int open(const char *pathname, int flags, mode_t mode);
```

The argument flags must include one of the following access modes: ***O_RDONLY***, ***O_WRONLY***, *or* ***O_RDWR***. *These request opening the file read-only, write-only, or read/write, respectively. In addition, zero or more file creation flags and file status flags can be bitwise-or'd in* ***flags***. *The file creation flags are* ***O_CLOEXEC***, ***O_CREAT***, ***O_DIRECTORY***, ***O_EXCL***, ***O_NOCTTY***, ***O_NOFOLLOW***, ***O_TRUNC***, *and* ***O_TTY_INIT***...

- ***O_CREAT*** *If the file does not exist it will be created. The owner (user ID) of the file is set to the effective user ID of the process. The group ownership (group ID) is set to the effective group ID of the process...*

mode *specifies the permissions to use in case a new file is created. This argument must be supplied when* ***O_CREAT*** *is specified in* ***flags***.

The following symbolic constants are provided for ***mode****:*

S_IRWXU	0x0700	user (file owner) has read, write and execute permission
S_IRUSR	0x0400	user has read permission
S_IWUSR	0x0200	user has write permission
S_IXUSR	0x0100	user has execute permission
S_IRWXG	0x0070	group has read, write and execute permission
S_IRGRP	0x0040	group has read permission
S_IWGRP	0x0020	group has write permission
S_IXGRP	0x0010	group has execute permission
S_IRWXO	0x0007	others have read, write and execute permission
S_IROTH	0x0004	others have read permission
S_IWOTH	0x0002	others have write permission
S_IXOTH	0x0001	others have execute permission

1. Napisati kod za sistemski poziv koji će kreirati fajl „test.txt“ u tekućem direktorijumu pozivajućeg procesa (označava se sa .) i otvara ga za čitanje i upis, tako da vlasnik tog fajla ima prava na čitanje i upis, članovi grupe prava samo na čitanje, a ostali nikakva prava.
2. Ukoliko takav identičan poziv izvrši neki drugi proces (nakon što je poziv iz a) uspešno izvršen) koji se izvršava u ime drugog korisnika koji nije član iste grupe kao korisnik u čije ime je izvršen prvi poziv, da li će taj drugi poziv uspeti? Obrazložiti.

Rešenje

1. `open("./test.txt", O_CREAT|O_RDWR, S_IRUSR|S_IWUSR|S_IRGRP);`
2. Neće. Taj korisnik (u čije ime se izvršava drugi proces) je pripadnik „ostalih“, pošto nije ni vlasnik fajla niti pripadnik iste grupe, pa nema nikakva prava nad fajlom, a traži otvaranje tog (sada postojećeg) fajla sa mogućnošću čitanja i upisa.

2. zadatak, treći kolokvijum, jun 2012.

U nekom fajl sistemu u sistemskom pozivu za otvaranje fajla proces navodi da li će fajl samo čitati ili ga i na bilo koji način menjati. U zavisnosti od toga, taj sistemski poziv zaključava fajl sa jednim od dve vrste ključa. Ako se fajl otvara samo za čitanje, fajl se zaključava deljenim ključem; ako se fajl otvara za izmenu, zaključava se ekskluzivnim ključem. Ukoliko poziv ne može da se izvrši zbog toga što ključ ne može da se dobije, poziv se otkazuje bez izmena u fajl sistemu i vraća se greška.

Procesi A, B, C i D izvršavaju sistemske pozive otvaranja i zatvaranja istog fajla u sledećem redosledu (neki proces izvršava poziv zatvaranja fajla samo ako ga je uspešno otvorio):

1. A: `open(READ)`
2. B: `open(WRITE)`
3. C: `open(READ)`
4. A: `close`
5. C: `close`
6. D: `open(WRITE)`

Koje od ovih operacija će se izvršiti uspešno, a koje neuspešno? Precizno obrazložiti odgovor!

Rešenje

Neuspešna je operacija 2), ostale su uspešne.

2. zadatak, treći kolokvijum, jun 2011.

U nekom operativnom sistemu struktura PCB sadrži polje `open_files` koje je pokazivač na tabelu otvorenih fajlova tog procesa. U toj tabeli (zapravo nizu), svaki ulaz je struktura koja predstavlja deskriptor jednog otvorenog fajla. U toj strukturi postoji celobrojno polje `access` čija vrednost 1 označava da je dati proces otvorio dati fajl sa pravom na upis, a vrednost 0 označava da je dati proces otvorio dati fajl samo sa pravom na čitanje. Promenljiva tipa `FHANDLE` predstavlja indeks u tabeli otvorenih fajlova datog procesa u čijem je kontekstu otvaren fajl. Implementirati operaciju `check_access()` čiji je potpis dat, a koju poziva fajl podsistem u svakom sistemskom pozivu za pristup sadržaju fajla radi provere prava datog procesa na izvršavanje date operacije nad sadržajem fajla. Argument `p` je pokazivač na PCB procesa koji je izdao sistemski poziv, argument `f` je identifikator fajla, a argument `write` ima vrednost 0 ako sistemski poziv zahteva samo čitanje, odnosno vrednost 1 ako sistemski poziv zahteva upis u dati fajl.

```
int check_access (PCB* p, FHANDLE f, int write);
```

Rešenje

```
int check_access (PCB* p, FHANDLE f, int write) {
  return p->open_files[FHANDLE].access || !write;
}
```

2. zadatak, treći kolokvijum, septembar 2011.

Dat je deo API za neki fajl sistem:

```
typedef int FHANDLE; // File handle
FHANDLE fopen (char* fname, int accessFlags);
int fclose (FHANDLE);
int fread (FHANDLE, unsigned int offset, unsigned int size, void* buffer);
int fwrite (FHANDLE, unsigned int offset, unsigned int size, void* buffer);
unsigned int fsize (FHANDLE); // Returns the file size
```

Sve funkcije osim `fsize` vraćaju pozitivnu vrednost u slučaju uspeha, a negativnu vrednost sa kodom greške u slučaju neuspeha. Koncept kurzora nije ugrađen u ovaj fajl sistem, već funkcije za čitanje i upis u fajl zadaju pomeraj (*offset*) od koga se čita/upisuje.

Korišćenjem ovog API realizovati klasu koja apstrahuje fajl sa ugrađenim kurzorom i sledećim interfejsom:

```
class File {
public:
  File (char* fname, int accessFlags);
 ~File ();
  int read (void* buffer, unsigned int size);
  int write (void* buffer, unsigned int size);
  int seek (unsigned int offset); // Move the cursor to the given offset
};
```

Rešenje

```
class File {
public:
  File (char* fname, int accessFlags);
 ~File ();
  int read (void* buffer, int size);
  int write (void* buffer, int size);
  int seek (unsigned int offset); // Move the cursor to the given offset
private:
  FHANDLE fh;
  unsigned int cursor;
};

File::File (char* fname, int af) : cursor(0) {
  fh=fopen(fname,af);
}

File::~File () { fclose(fh); }

int File::read (void* buffer, unsigned int size) {
  int ret = fread(fh,cursor,size,buffer);
  seek(cursor+size);
  return ret;
}

int File::write (void* buffer, unsigned int size) {
  int ret = fwrite(fh,cursor,size,buffer);
  seek(cursor+size);
  return ret;
}

int File::seek (unsigned int offset) {
  unsigned int size = fsize(fh);
  if (offset>=size) cursor=size;
  else cursor=offset;
  return 1;
}
```

Fajl sistem (implementacija)

3. zadatak, treći kolokvijum, jun 2022.

Sistemski pozivi za pristup sadržaju binarnog fajla uobičajeno omogućavaju čitanje ili upis size susednih bajtova počev od pozicije *offset* u odnosu na početak sadržaja fajla (prvi bajt je na poziciji 0). Da bi ostvario ovakav pristup, neki kernel koristi klasu `FLogicalAccess` čiji je interfejs dat dole. Ova klasa koristi se tako što se za svaki ovakav pristup instancira objekat ove klase, inicijalizuje se pozivom operacije `reset` sa zadatim parametrima, a onda se iterira sve dok operacija `end` ne vrati 1. U svakoj iteraciji se pristupa odgovarajućim bajtovima jednog logičkog bloka fajla, tako što se može dobiti sledeća informacija:

- `getBlock`: vraća redni broj logičkog bloka u kom se pristupa u toj iteraciji;
- `getRelOffset`: vraća redni broj bajta u tekućem bloku počev od kog se pristupa;
- `getRelSize`: broj bajtova u tekućem bloku kojima se pristupa u toj iteraciji.

```
class FLogicalAccess {
public:
    FLogicalAccess ();
    void reset (size_t offset, size_t size);
    int end() const;
    void next();
    size_t getBlock() const;
    size_t getRelOffset() const;
    size_t getRelSize() const;
};
```

Primer upotrebe ove klase je sledeći:

```
FLogicalAccess fla;
for (fla.reset(offset,size); !fla.end(); fla.next()) {
    // Access fla.getRelSize() bytes
    // starting from the offset fla.getRelOffset()
    // in the logical block fla.getBlock()
}
```

Na primer, ako je veličina bloka 8 bajtova, a inicijalno je zadato: $offset = 11$ i $size = 15$, onda će u prvoj iteraciji biti: `getBlock()` = 1, `getRelOffset()` = 3, `getRelSize()` = 5, u drugoj će ove vrednosti redom biti 2, 0, 8, a u trećoj 3, 0, 2; posle toga će se izaći iz petlje zbog `end()` = 1.

Implementirati u potpunosti klasu `FLogicalAccess`. Veličina bloka je `BLK_SIZE`.

Rešenje

```
class FLogicalAccess {
public:
    FLogicalAccess() {
        reset(0, 0);
    }
    void reset(size_t offset, size_t size);
    int end() const {
        return isEnd;
    }
    void next();
    size_t getBlock() const {
        return blk;
    }
    size_t getRelOffset() const {
        return relOfs;
    }
    size_t getRelSize() const {
```

```
        return relSz;
    }
private:
    size_t size;
    size_t blk, relOfs, relSz;
    int isEnd;
};
inline void FLogicalAccess::reset(size_t offset, size_t sz) {
    blk = offset / BLK_SIZE;
    relOfs = offset % BLK_SIZE;
    relSz = (sz > BLK_SIZE - relOfs) ? (BLK_SIZE - relOfs) : sz;
    isEnd = (sz == 0);
    size -= relSz;
}
inline void FLogicalAccess::next() {
    if (size > 0) {
        blk++;
        relOfs = 0;
        relSz = (size > BLK_SIZE) ? BLK_SIZE : size;
        isEnd = 0;
        size -= relSz;
    } else {
        reset(0, 0);
    }
}
```

4. zadatak, kolokvijum, jul 2021.

Neki fajl sistem primenjuje kombinovanu tehniku indeksirane alokacije sadržaja fajla. U FCB fajla polje `singleIndex` predstavlja direktni indeks kao niz od `SingleIndexSize` elemenata, pri čemu svaki element sadrži broj fizičkog bloka sa sadržajem fajla (ili 0, ako je neiskorišćen). Ako veličina sadržaja fajla preraste veličinu podržanu ovim indeksom, naredni blokovi sadržaja fajla (preko ove veličine) indeksiraju se indeksom u dva nivoa. Za te potrebe, polje `dblIndex` u FCB sadrži broj bloka na disku u kom je indeks drugog nivoa. Taj blok sa indeksom drugog nivoa sadrži `DblIndexSize` ulaza sa brojevima blokova sa sadržajem fajla, odnosno 0 ako ulaz nije iskorišćen. Polje `size` u FCB sadrži veličinu fajla u bajtovima. Veličina bloka u bajtovima je `BLOCK_SIZE`. Svi ulazi u indeksu prvog nivoa i indeksu drugog nivoa (ako postoji) su sigurno postavljeni na validne vrednosti (broj bloka sa sadržajem ili 0). Realizovati funkciju `extendFile` koja proširuje sadržaj fajla čiji je FCB dat za jedan blok:

```
int extendFile(FCB* fcb);
```

Ova funkcija treba da vrati 0 u slučaju uspeha, a negativnu vrednost u slučaju greške. Ona ne treba da menja polje `size` u FCB (to će raditi pozivalac). Na raspolaganju je funkcija koja vraća pokazivač na sadržaj bloka na disku sa datim brojem, učitanog u keš:

```
void* getBlock(PBlock blkNo);
```

kao i funkcija koja zauzima jedan slobodan blok i vraća njegov broj ili 0 ako takvog nema:

```
PBlock allocateBlock();
```

Rešenje

```
int extendFile(FCB* fcb) {
    size_t entry = (fcb->size + BLOCK_SIZE - 1)/BLOCK_SIZE;
    if (entry < SingleIndexSize) {
        PBlock newBlock = allocateBlock();
        if (!newBlock) return -1; // Error: no free space for the new block
        fcb->singleIndex[entry] = newBlock;
        return 0;
```

```
    }
    entry -= SingleIndexSize;
    if (entry >= DblIndexSize) return -2; // Error: file size limit exceeded
    if (!fcb->dblIndex) {
        fcb->dblIndex = allocateBlock();
        if (!fcb->dblIndex) return -3; // Error: no space for index block
    }
    PBlock newBlock = allocateBlock();
    if (!newBlock) return -1; // Error: no free space for the new block
    PBlock* dblIx = (PBlock*) getBlock(fcb->dblIndex);
    dblIx[entry] = newBlock;
    return 0;
}
```

4. zadatak, kolokvijum, avgust 2021.

U nekom FAT fajl sistemu FCB-ovi su smešteni na jednom delu particije, a sadržaj fajlova na drugom, disjunktnom delu particije. Za taj fajl sistem pravi se sistemski program koji radi kompakciju prostora na disku tako što premešta blokove sa sadržajem fajla (ne i one sa FCB-om) tako da budu susedni na particiji. FAT tabela je veličine `FAT_SIZE` i njeni ulazi predstavljaju sve blokove na particiji. Ulaz FAT-a sadrži broj sledećeg bloka u ulančanoj listi blokova sa sadržajem fajla, odnosno u ulančanoj listi slobodnih blokova čija je glava u globalnoj promenljivoj `freeBlkHead`; nula u ulazu označava kraj liste blokova sa sadržajem, a negativna vrednost u ulazu označava slobodan blok (sledeći u lancu slobodnih je dat suprotnom vrednošću te negativne, `-1` označava kraj liste). FCB svakog fajla smešten je u jedan blok na prvom delu particije, a ulaz FAT-a koji odgovara tom bloku sadrži broj prvog bloka sa sadržajem tog fajla. FAT je cela keširana u memoriji.

Implementirati funkciju `compact` koja za dati fajl radi kompakciju premeštanjem blokova sa sadržajem tog fajla tako da budu smešteni redom počev od bloka sa brojem `startBlk`. Parametar `fcbBlk` određuje broj bloka u kom je FCB datog fajla. Prostor počev od bloka broj `startBlk` nije oslobođen (u njemu se možda nalaze blokovi sa sadržajem fajlova), ali do kraja particije svakako ima dovoljno blokova za smeštanje sadržaja datog fajla. Na raspolaganju su operacije za čitanje i upis bloka sa datim brojem za dati bafer u memoriji veličine `BLK_SIZE` znakova. Ignorisati greške.

```
int FAT[FAT_SIZE];
void readBlock (int blk, char* buffer);
void writeBlock(int blk, const char* buffer);
void compactFile (int fcbBlk, int startBlk);
```

Rešenje

```
inline void swap (int* x, int* y) { int temp = *x; *x = *y; *y = temp; }

void moveBlk (int blk) {  // Moves the given block to the first free block
  char buffer[BLK_SIZE];
  readBlock(blk,buffer);
  writeBlock(freeBlkHead,buffer);
  swap(&FAT[freeBlkHead],&FAT[blk]);
  for (int i=0; i<FAT_SIZE; i++)
    if (FAT[i]==blk) { FAT[i]=freeBlkHead; break; }
  freeBlkHead = blk;
}

void compactFile (int fcbBlk, int startBlk) {
  for (int b = FAT[fcbBlk]; b!=0; b=FAT[b]) {
    if (FAT[startBlk]>=0) moveBlk(startBlk);
    moveBlk(b);
    startBlk++;
```

```
  }
}
```

4. zadatak, kolokvijum, oktobar 2020.

Da bi smanjio internu fragmentaciju, a povećao efikasnost, neki fajl sistem za alokaciju sadržaja fajla koristi klastere (*cluster*) različite veličine. Klaster uvek sadrži susedne blokove na disku (blokove sa susednim brojevima). Za sadržaj fajla pre njegovog kraja koriste se klasteri veličine `CLUSTER_SIZE` blokova, gde je `CLUSTER_SIZE` predefinisana konstanta, mali prirodan broj veći od 1. Za sam kraj sadržaja fajla koristi se samo onoliko susednih blokova na disku koliko je potrebno, odnosno klaster veličine manje od ili jednake `CLUSTER_SIZE` blokova.

Sistem primenjuje indeksiranu alokaciju sadržaja fajla, pri čemu je indeks u samom FCB. Polje `index` strukture FCB predstavlja indeks kao niz, pri čemu svaki element i sadrži broj prvog fizičkog bloka (početak klastera) sa sadržajem fajla (ili 0, ako je neiskorišćen) za klaster sa rednim brojem i (počev od 0). Polje `size` u FCB sadrži veličinu sadržaja fajla u bajtovima. Veličina bloka na disku u bajtovima je `BLOCK_SIZE`. FCB je uvek učitan u memoriju za otvoren fajl.

Realizovati funkciju `getPBlock`:

```
size_t getPBlock (FCB* fcb, size_t bt);
```

Ova funkcija treba da vrati broj fizičkog bloka za bajt sa zadatim logičkim rednim brojem `bt` unutar sadržaja fajla (0 u slučaju da `bt` prekoračuje veličinu sadržaja fajla).

Rešenje

```
size_t getPBlock (FCB* fcb, size_t bt) {
  if (bt>=fcb->size) return 0;
  size_t lBlk = bt/BLOCK_SIZE;
  size_t cluster = lBlk/CLUSTER_SIZE;
  size_t pBlk = fcb->index[cluster] + lBlk%CLUSTER_SIZE;
  return pBlk;
}
```

4. zadatak, kolokvijum, jun 2020.

U implementaciji nekog fajl sistema evidencija slobodnih blokova na particiji vodi se pomoću bit-vektora koji se kešira u memoriji u nizu `blocks` veličine `NumOfBytes` (konstanta `NumOfBlocks` predstavlja broj blokova na particiji). Svaki element ovog niza je veličine jednog bajta, a svaki bit odgovara jednom bloku na disku (1-zauzet, 0-slobodan). Pretpostaviti da je `NumOfBlocks` umnožak broja 8 (`BITS_IN_BYTE`). Date su pomoćne funkcije i funkcija `allocateBlock` koja vrši alokaciju slobodnog bloka tražeći prvi slobodan blok počev od onog datog argumentom. Blok 0 je uvek zauzet.

```
typedef unsigned char byte;
typedef unsigned long ulong;
const byte BITS_IN_BYTE = 8;
const ulong NumOfBlocks = ...;
const ulong NumOfBytes = NumOfBlocks/BITS_IN_BYTE;
byte blocks[NumOfBytes];

void blockToBit(ulong blkNo, ulong& bt, byte& mask) {
  bt = blkNo/BITS_IN_BYTE;
  mask = 1<<(blkNo%BITS_IN_BYTE);
}

void bitToBlk(ulong& blkNo, ulong bt, byte mask) {
  blkNo = bt*BITS_IN_BYTE;
  for (; !(mask&1); mask>>=1) blkNo++;
}
```

```
ulong allocateBlock (ulong startingFrom) {
  ulong bt = 0; byte msk = 0;
  for (ulong blk = startingFrom; blk<NumOfBlocks; blk++) {
    blockToBit(blk,bt,msk);
    if ((blocks[bt]&msk)==0) {
      blocks[bt] |= msk;
      return blk;
    }
  }
  for (ulong blk =1; blk<startingFrom; blk++) {
    blockToBit(blk,bt,msk);
    if ((blocks[bt]&msk)==0) {
      blocks[bt] |= msk;
      return blk;
    }
  }
  return 0;
}
```

Modifikovati funkciju `allocateBlock` tako da alocira blok koji je najbliži onom koji je tad argumentom, i to najbliži sa bilo koje strane.

Rešenje

```
#define max(a,b) ((a)>=(b)?(a):(b))

ulong allocateBlock (ulong startingFrom) {
  if (startingFrom>=NumOfBlocks) return 0;
  ulong bt = 0, blk = 0; byte msk = 0;
  ulong limit = max(NumOfBlocks-startingFrom,startingFrom);
  for (ulong i=0; i<limit; i++) {
    if (i<NumOfBlocks-startingFrom) {
      blk = startingFrom+i;
      blockToBit(blk,bt,msk);
      if ((blocks[bt]&msk)==0) {
        blocks[bt] |= msk;
        return blk;
      }
    }
    if (i<startingFrom && i>0) {
      blk = startingFrom-i;
      blockToBit(blk,bt,msk);
      if ((blocks[bt]&msk)==0) {
        blocks[bt] |= msk;
        return blk;
      }
    }
  }
  return 0;
}
```

4. zadatak, kolokvijum, septembar 2020.

Neki fajl sistem primenjuje kombinovanu tehniku indeksirane alokacije sadržaja fajla. U FCB fajla polje `singleIndex` predstavlja direktni indeks kao niz od `SingleIndexSize` elemenata, pri čemu svaki element sadrži broj fizičkog bloka sa sadržajem fajla (ili 0, ako je neiskorišćen). Ako veličina sadržaja fajla preraste veličinu podržanu ovim indeksom, naredni blokovi sadržaja fajla (preko ove veličine) indeksiraju se indeksom u dva nivoa. Za te potrebe,

polje `dblIndex` u FCB sadrži broj bloka na disku u kom je indeks drugog nivoa. Taj blok sa indeksom drugog nivoa sadrži `DblIndexSize` ulaza sa brojevima blokova sa sadržajem fajla, odnosno 0 ako ulaz nije iskorišćen. Polje `size` u FCB sadrži veličinu fajla u bajtovima. Veličina bloka je `BlockSize`. Svi ulazi u indeksu prvog nivoa i indeksu drugog nivoa (ako postoji) su sigurno postavljeni na validne vrednosti (broj bloka sa sadržajem ili 0).

Realizovati funkciju `truncateFile` koja treba da obriše sadržaj fajla čiji je FCB dat:

```
void truncateFile (FCB* fcb);
```

Na raspolaganju je funkcija koja vraća pokazivač na sadržaj bloka na disku sa datim brojem, učitanog u keš:

```
void* getBlock (PBlock blkNo);
```

kao i funkcija koja blok sa datim brojem označava slobodnim:

```
void freeBlock (PBlock blkNo);
```

Rešenje

```
void truncateFile(FCB* fcb) {
    for (size_t i = 0; i < SingleIndexSize; i++)
        if (fcb->singleIndex[i]) {
            freeBlock(fcb->singleIndex[i]);
            fcb->singleIndex[i] = 0;
        }
    if (fcb->dblIndex) {
        PBlock* dblIx = (PBlock*)getBlock(fcb->dblIndex);
        for (size_t i = 0; i < DblIndexSize; i++)
            if (dblIx[i]) freeBlock(dblIx[i]);
        freeBlock(fcb->dblIndex);
        fcb->dblIndex = 0;
    }
    fcb->size = 0;
}
```

3. zadatak, treći kolokvijum, jun 2019.

Za fajl opisan u zadatku 2. poziva se opisani potprogram za argument n=12 i vrednost x koja ne postoji u stablu, a koja je veća od vrednosti u svim čvorovima stabla. Veličina bloka na disku je 512B, a veličina jednog podatka tipa `int` je 32 bita. Fajl se alocira indeksiranim metodom, s tim da je indeks prvog nivoa u FCB koji se smatra učitanim u memoriju (učitava se prilikom otvaranja fajla).

1. Ako je indeks u samo jednom nivou, koliko blokova sa sadržajem ukupno dohvata ovaj program?
2. Ako je indeks u dva nivoa, s tim da je indeks prvog nivoa u FCB i učitan je u memoriju, a jedan indeks drugog nivoa zauzima jedan blok na disku i ima ulaze veličine 64 bita, koliko ukupno blokova dohvata (učitava) ovaj program, računajući i pristupe indeksima drugog nivoa?

Izvesti rezultat i obrazložiti postupak.

Rešenje

Za dati slučaj, program pristupa elementima niza sa sledećim indeksima ($ni := 2ni + 2$):

0, 2, 6, 14, 30, 62, 126, 254, 510, 1022, 2046, 4094.

Svaki element niza zauzima $4=2^2$ bajta, a blok je veličine $512\text{B}=2^9\text{B}$, pa jedan blok sadrži 2^7=128 elemenata niza. Zato prvih 7 adresiranih elemenata niza pripada istom bloku sa sadržajem fajla (bloku broj 0), dok su svi ostali elementi u različitim blokovima. Navedeni elementi pripadaju redom sledećim logičkim blokovima sadržaja fajla:

0, 0, 0, 0, 0, 0, 0, 1, 3, 7, 15, 31.

1. Za slučaj indeksa u jednom nivou, dohvata se ukupno 1+5=6 blokova sa sadržajem (indeksni blok je već učitan).

2. Za slučaj indeksa u dva nivoa, jedan ulaz u indeksu prvog nivoa, odnosno jedan indeksni blok drugog nivoa, pokriva 2^6=64 blokova sa sadržajem, pa su svi adresirani elementi pokriveni jednim indeksnim blokom drugog nivoa. Zato se ukupno dohvata jedan indeksni blok drugog nivoa i 6 blokova sa sadržajem, ukupno 7.

3. zadatak, treći kolokvijum, jun 2018.

Neki fajl sistem koristi indeksiranu alokaciju sadržaja fajla sa jednostepenim indeksom u jednom bloku. Struktura FCB keširana je u memoriji, kao i indeksni blok. U strukturi FCB, pored ostalih, postoje sledeća polja:

- `unsigned long size`: veličina sadržaja fajla u bajtovima;
- `unsigned long* index`: pokazivač na (keširan) indeks (ulazi su tipa `unsigned long`).

Osim toga, definisane su i sledeće konstante i funkcija:

- `unsigned long BlockSize`: veličina bloka na disku u bajtovima;
- `unsigned long MaxFileSize`: maksimalna dozvoljena veličina sadržaja fajla u bajtovima; jednaka je maksimalnom broju ulaza u indeksu pomnoženom veličinom bloka `BlockSize`;
- `unsigned long allocateBlock()`: alocira jedan slobodan blok na disku i vraća njegov broj; ukoliko slobodnog bloka nema, vraća 0.

Realizovati funkciju

```
unsigned long extend (FCB* fcb, unsigned extension);
```

koja proširuje sadržaj fajla na čiji FCB ukazuje prvi argument za broj bajtova dat drugim argumentom. Ukoliko traženo proširenje premašuje maksimalnu veličinu fajla ili na disku nema dovoljno slobodnog prostora, sadržaj fajla treba proširiti koliko je moguće. Ova funkcija vraća broj bajtova sa koliko je stvarno uspela da proširi sadržaj (jednako ili manje od traženog).

Rešenje

```
unsigned long extend (FCB* fcb, unsigned extension) {
  if (fcb==0) return 0;

  unsigned long oldSize = fcb->size;
  unsigned oldNumOfBlocks = oldSize/BlockSize + oldSize%BlockSize?1:0;

  unsigned long newSize = oldSize + extension;
  if (newSize>MaxFileSize) newSize = MaxFileSize;
  unsigned newNumOfBlocks = newSize/BlockSize + newSize%BlockSize?1:0;

  unsigned i;
  for (i=oldNumOfBlocks; i<newNumOfBlocks; i++) {
    unsigned long block = allocateBlock();
    if (block==0) break;
    fcb->index[i] = block;
  }

  if (i<newNumOfBlocks)
    newSize = i*BlockSize;

  fcb->size = newSize;
  return newSize-oldSize;
```

3. zadatak, treći kolokvijum, jun 2017.

U implementaciji nekog FAT fajl sistema cela FAT keširana je u memoriji u strukturi:

```
extern unsigned long fat[];
```

Za ulančavanje se kao *null* vrednost u ulazu u FAT koristi 0 (blok broj 0 je rezervisan). U FCB fajla polje `head` sadrži redni broj prvog bloka sa sadržajem fajla i polje `size` koje sadrži veličinu sadržaja fajla u bajtovima. Slobodni blokovi su označeni ulazima u FAT sa vrednošću -1. Implementirati sledeću funkciju koja treba da obriše sadržaj fajla:

```
void truncate (FCB* fcb);
```

Rešenje

```
void truncate (FCB* fcb) {
  if (fcb==0) return;
  unsigned long cur=fcb->head, next;
  while (cur) {
    next = fat[cur];
    fat[cur] = -1;
    cur = next;
  }
  fcb->head = 0;
  fcb->size = 0;
}
```

3. zadatak, treći kolokvijum, jun 2016.

U implementaciji nekog FAT fajl sistema cela FAT keširana je u memoriji u strukturi:

```
extern unsigned long fat[];
unsigned long freeHead;
```

Za ulančavanje se kao *null* vrednost u ulazu u FAT koristi 0 (blok broj 0 je rezervisan). U FCB fajla polje `head` sadrži redni broj prvog bloka sa sadržajem fajla. Slobodni blokovi su takođe ulančani u FAT, a broj prvog slobodnog bloka u lancu sadrži globalna promenljiva `freeHead`.

Implementirati sledeću funkciju:

```
unsigned long append (FCB* fcb);
```

koja treba da proširi sadržaj fajla sa datim FCB za jedan blok (da doda jedan blok na kraj). Funkcija vraća 0 ako slobodnih blokova nema ili u slučaju bilo koje druge greške, odnosno broj alociranog bloka ako je operacija uspela. Ne treba vršiti nikakve optimizacije u smislu alokacije najbližeg bloka već alociranim blokovima datog fajla.

Rešenje

```
unsigned long append (FCB* fcb) {
  if (fcb==0 || freeHead==0) return 0;
  unsigned long last=0, next=fcb->head, ret=freeHead;
  while (next) last=next, next=fat[next];
  if (last)
    fat[last] = freeHead;
  else
    fcb->head = freeHead;
  unsigned long oldHead = freeHead;
  freeHead = fat[freeHead];
  fat[oldHead] = 0;
  return ret;
}
```

3. zadatak, treći kolokvijum, septembar 2016.

Neki fajl sistem primenjuje indeksiranu alokaciju fajlova, sa indeksima u dva nivoa. U FCB fajla nalazi se polje `index` tipa `unsigned long`, koje predstavlja broj bloka na disku u kome se nalazi indeks prvog nivoa. Indeksi

oba nivoa su iste veličine `IndexSize` ulaza tipa `unsigned long`. Nepopunjeni ulazi u indeksima imaju vrednost 0. Operacija `getPBlock` po potrebi u keš učitava blok sa diska sa zadatim brojem i vraća pokazivač na mesto u kešu u koje je taj blok učitan; u slučaju greške vraća 0.

```
typedef unsigned long ulong;
extern const ulong IndexSize;
void* getPBlock(ulong pBlockNo);
```

Realizovati funkciju `getFileBlock()` datu dole, koja se koristi u implementaciji fajl sistema i koja treba da vrati adresu na učitan logički blok fajla sa zadatim (logičkim) brojem; u slučaju greške treba da vrati 0.

```
void* getFileBlock (FCB* fcb, ulong lBlockNo);
```

Rešenje

```
void* getFileBlock (FCB* fcb, ulong lBlockNo) {
  if (fcb==0) return 0; // Exception
  ulong entry0 = lBlockNo/IndexSize;
  if (entry0>=IndexSize) return 0; // Exception
  ulong entry1 = lBlockNo%IndexSize;
  ulong index = fcb->index;
  if (index==0) return 0; // Exception
  ulong* index0 = (ulong*)getPBlock(index);
  if (index0==0) return 0; // Exception
  index = index0[entry0];
  if (index==0) return 0; // Exception
  ulong* index1 = (ulong*)getPBlock(index);
  if (index1==0) return 0; // Exception
  return getBlock(index1[entry1]);
}
```

3. zadatak, treći kolokvijum, jun 2015.

U implementaciji nekog fajl sistema evidencija slobodnih blokova na disku vodi se pomoću bit-vektora koji se kešira u memoriji u nizu `blocks` veličine `NumOfBlocks/BITS_IN_BYTE` (konstanta `NumOfBlocks` predstavlja broj blokova na disku). Svaki element ovog niza je veličine jednog bajta, a svaki bit odgovara jednom bloku na disku (1-zauzet, 0-slobodan). Pretpostaviti da je `NumOfBlocks` umnožak broja 8 (`BITS_IN_BYTE`).

```
typedef unsigned char byte;
const unsigned int BITS_IN_BYTE = 8;
const unsigned long NumOfBlocks = ...;
byte blocks[NumOfBlocks/BITS_IN_BYTE];
```

1. Implementirati sledeće dve funkcije:

```
void blockToBit(unsigned long blkNo, unsigned long& bt, byte& mask);
void bitToBlk(unsigned long& blkNo, unsigned long bt, byte mask);
```

Funkcija `blockToBit` prima kao ulazni parametar redni broj bloka `blkNo` i na osnovu njega izračunava i upisuje u izlazne parametre broj bajta (`bt`) u bit-vektoru i jednu jedinu jedinicu u onaj razred parametra `msk` koji odgovara bitu u tom bajtu za dati blok. Funkcija bitToBlock radi obrnutu konverziju: za ulazni parametar koji je redni broj bajta u bit-vektoru (`bt`) i najniži razred u parametru `msk` koji je postavljen na 1, izračunava i upisuje u izlazni parametar `blk` redni broj bloka koji odgovara tom bajtu i bitu.

2. Korišćenjem funkcija pod a), implementirati funkcije `allocateBlock` i `freeBlock`. Funkcija `allocateBlock` treba da pronađe prvi slobodan blok i označi ga zauzetim. Ako takav blok nađe, vraća broj tog bloka, a ako slobodnog bloka nema, vraća 0 (blok broj 0 je rezervisan i nikada se ne koristi u fajl sistemu). Kako bi se optimizovao pristup fajlovima, slobodan blok se traži u blizini bloka sa datim rednim brojem `startingFrom`, na sledeći način:

- slobodan blok treba tražiti počev od datog bloka naviše (i alocirati prvi slobodan na koji se naiđe), pri tom može (ali ne mora) da se alocira i bilo koji slobodan blok koji je u istom bajtu bit-vektora kao i dati blok, bez obzira da li je ispred ili iza datog bloka (i on se smatra dovoljno bliskim).
- ukoliko se na ovaj način ne pronađe, počinje se pretraga od prvog bloka (blok broj 0 je rezervisan) pa sve do zadatog bloka `startingFrom`.

Funkcija `freeBlock` označava dati blok slobodnim.

```
unsigned long allocateBlock (unsigned long startingFrom);
void freeBlock (unsigned long blk);
```

Rešenje

1.

```
void blockToBit(unsigned long blkNo, unsigned long& bt, byte& mask) {
  bt = blkNo/BITS_IN_BYTE;
  mask = 1<<(blkNo%BITS_IN_BYTE);
}

void bitToBlk(unsigned long& blkNo, unsigned long bt, byte mask) {
  blkNo = bt*BITS_IN_BYTE;
  for (; !(mask&1); mask>>=1) blkNo++;
}
```

2.

```
void freeBlock (unsigned long blk) {
  if (blk>=NumOfBlocks) return;
  unsigned long bt = 0; byte msk = 0;
  blockToBit(blk,bt,msk);
  blocks[bt] &= ~msk;
}
unsigned long allocateBlock (unsigned long startingFrom) {
  unsigned long bt = 0; byte msk = 0;
  for (unsigned long blk = startingFrom; blk<NumOfBlocks; blk++) {
    blockToBit(blk,bt,msk);
    if ((blocks[bt]&msk)==0) {
      blocks[bt] |= msk;
      return blk;
    }
  }

  for (unsigned long blk =1; blk<startingFrom; blk++) {
    blockToBit(blk,bt,msk);
    if ((blocks[bt]&msk)==0) {
      blocks[bt] |= msk;
      return blk;
    }
  }
  return 0;
}
```

Nešto efikasnija implementacija:

```
unsigned long allocateBlock (unsigned long startingFrom) {
  if (startingFrom>=NumOfBlocks) return 0;
  unsigned long bt = 0; byte msk = 0;
  blockToBit(startingFrom,bt,msk);
  for (; bt<NumOfBlocks/BITS_IN_BYTE; bt++) {
```

```
    if (blocks[bt]==~0) continue; // All blocks in this byte are occupied
    unsigned long blk = 0;
    bitToBlock(blk,bt,~blocks[bt]);
    blockToBit(blk,bt,msk)
    blocks[bt] |= msk;
    return blk;
  }

  blockToBit(1,bt,msk); // blk 0 is reserved
  for (; bt<NumOfBlocks/BITS_IN_BYTE; bt++) {
    if (blocks[bt]==~0) continue; // All blocks in this byte are occupied
    unsigned long blk = 0;
    bitToBlock(blk,bt,~blocks[bt]);
    blockToBit(blk,bt,msk)
    blocks[bt] |= msk;
    return blk;
  }

  return 0;
}
```

3. zadatak, treći kolokvijum, septembar 2015.

Neki fajl sistem primenjuje FAT za alokaciju sadržaja fajla. FAT je cela keširana u memoriji, na nju ukazuje pokazivač `fat`, i ima `FATSIZE` ulaza tipa `unsigned`. Prilikom ulančavanja blokova sa sadržajem fajla, *null* vrednost se označava vrednošću 0 u odgovarajućem ulazu u FAT, dok se slobodni blokovi ne ulančavaju posebno, već su njima odgovarajući ulazi u FAT označeni vrednostima `~0U` (sve jedinice binarno); blokovi broj 0 i broj `~0U` se ne koriste u fajl sistemu. U FCB polje head tipa unsigned sadrži broj prvog bloka sa sadržajem fajla (0 ako je sadržaj prazan).

Realizovati funkciju `extendFile()` datu dole, koja se koristi u implementaciji fajl sistema i koja treba da proširi sadržaj fajla za by blokova (da ih alocira i doda na kraj sadržaja fajla). Ova funkcija treba da vrati broj blokova kojim je stvarno proširen sadržaj fajla i koji može biti jednak `by`, ukoliko je u fajl sistemu bilo dovoljno slobodnog mesta, odnosno manji od te vrednosti (uključujući i 0), ukoliko nije bilo dovoljno slobodnih blokova.

```
unsigned extendFile (FCB* fcb, unsigned by);
```

Rešenje

```
unsigned extendFile (FCB* fcb, unsigned by) {
  if (fcb==0) return 0; // Exception
  // Find the first free block:
  for (unsigned free = 1; free<FATSIZE && (fat[free]!=~0U); free++);
  if (free==FATSIZE) return 0; // No free space
  // Find the file's tail block:
  unsigned tail = fcb->head;
  if (tail)
    while (fat[tail]) tail = fat[tail];
  // Extend the file:
  unsigned extendedBy = 0;
  while (1) {
    if (tail)
      fat[tail] = free;
    else
      fcb->head = free;
    tail = free;
    fat[tail] = 0;
    extendedBy++;
    if (extendedBy==by) return extendedBy;
```

```
    // Find the next free block:
    for (free++; free<FATSIZE && (fat[free]!=~0U); free++);
    if (free==FATSIZE) return extendedBy; // No more free space
  }
  return extendedBy;
}
```

3. zadatak, treći kolokvijum, jun 2014.

U implementaciji nekog fajl sistema evidencija slobodnih blokova na disku vodi se na sledeći način. Indeks (spisak) slobodnih blokova je neograničen i zapisuje se u samim slobodnim blokovima, ulančanim u jednostruku listu. Prema tome, prvi slobodan blok sadrži spisak najviše N slobodnih blokova (*ne* uključujući njega samog, tj. on nije na spisku), dok poslednji ulaz u tom spisku u tom bloku sadrži broj sledećeg bloka u listi, u kome se nalazi nastavak spiska slobodnih blokova itd. Ukoliko je neki slobodni blok bio na spisku, a više nije, njegov ulaz u indeksu slobodnih blokova ima vrednost 0 (blok broj 0 nikada nije slobodan). Kada se zahteva jedan slobodan blok, treba jednostavno uzeti prvi slobodan blok sa spiska. Pri tome, ukoliko je ceo spisak sadržan u prvom bloku u listi ispražnjen, treba alocirati upravo taj prvi blok iz liste i izbaciti ga iz liste.

Na prvi blok u listi ukazuje globalna promenljiva `freeBlocksHead`. Na raspolaganju je i funkcija `getBlock` koja vraća pokazivač na deo memorije u kešu u koju je učitan traženi blok sa diska. (Napomena: broj N je određen veličinom bloka i veličinom tipa `BlockNo`.)

```
typedef ... BlockNo; // Disk block number
extern int blockSize; // Disk block size
void* getBlock (BlockNo block);
extern BlockNo freeBlocksHead;
```

Realizovati funkciju `getFreeBlock()` koja treba da alocira i vrati jedan slobodni blok:

```
BlockNo getFreeBlock ();
```

Rešenje

```
BlockNo getFreeBlock () {
  static const int numOfEntries = blockSize/sizeof(BlockNo);

  BlockNo ret = 0;
  if (freeBlocksHead==0) return ret; // No free blocks!
  BlockNo* index = (BlockNo*)getBlock(freeBlocksHead); // Get first block
  for (int i=0; i<numOfEntries-1; i++) {
    if (index[i]==0) continue;
    ret = index[i]; // Found another block
    index[i]=0;
    return ret;
  }
  // No other blocks found in the first block, return this first block:
  ret = freeBlocksHead;
  freeBlocksHead = index[numOfEntries-1];
  return ret;
}
```

3. zadatak, treći kolokvijum, septembar 2014.

Neki fajl sistem primenjuje kombinovanu tehniku indeksirane alokacije sadržaja fajla. U FCB fajla polje `singleIndex` predstavlja direktni indeks kao niz od `SingleIndexSize` elemenata, pri čemu svaki element sadrži broj fizičkog bloka sa sadržajem fajla (ili *null*, ako je neiskorišćen). Ako veličina sadržaja fajla preraste veličinu podržanu ovim indeksom, naredni blokovi sadržaja fajla (preko ove veličine) indeksiraju se indeksom u dva nivoa. Za te potrebe, polje `dblIndex` u FCB sadrži `DblIndex0Size` elemenata, pri čemu svaki element sadrži broj bloka sa indeksom

drugog nivoa (ili *null*, ako je neiskorišćen). Blokovi sa indeksom drugog nivoa sadrže najviše `DblIndex1Size` ulaza sa brojevima blokova sa sadržajem fajla. Veličina bloka je `BlockSize`.

Realizovati funkciju `getFilePBlockNo()` datu dole, koja se koristi u implementaciji modula za organizaciju fajla i koja treba da vrati broj fizičkog bloka na disku za dati fajl i dati redni broj bajta sadržaja tog fajla (počev od 0). Dati redni broj bajta je vei proveren pre poziva ove funkcije, tako da sigurno adresira bajt unutar stvarne veličine sadržaja fajla (ne prekoračuje ga).

```
PBlock getFilePBlockNo (FCB* fcb, unsigned long byte);
```

Na raspolaganju je funkcija koja vraća pokazivač na sadržaj bloka na disku sa datim brojem, učitanog u keš:

```
void* getDiskBlock (PBlock blkNo);
```

Rešenje

```
PBlock getFilePBlockNo (FCB* fcb, unsigned long bt) {
  if (fcb==0) return -1; // Exception
  unsigned long lblk = bt/BlockSize;  // Logical block number
  if (lblk<SingleIndexSize) return fcb->singleIndex[lblk];
  lblk -= SingleIndexSize;
  unsigned long dblIndex0Entry = lblk/DblIndex1Size;
  unsigned long dblIndex1Entry = lblk%DblIndex1Size;
  if (dblIndex0Entry>=DblIndex0Size)
    return -1; // Exception: file size overflow
  PBlock dblIndex1PBlkNo = fcb->dblIndex[dblIndex0Entry];
  PBlock* dblIndex1 = (PBlock*)getDiskBlock(dblIndex1PBlkNo);
  if (dblIndex1==0) return -1; // Exception
  return dblIndex1[dblIndex1Entry];
}
```

3. zadatak, treći kolokvijum, jun 2013.

U implementaciji nekog fajl sistema definisani su celobrojni tip `Byte` koji predstavlja bajt, kao i celobrojni tip `BlkNo` koji predstavlja broj logičkog bloka sadržaja fajla (numeracija počev od 0). U strukturi FCB celobrojno polje `cur` predstavlja kurzor za čitanje i upis (u bajtovima, počev od 0), a polje `size` stvarnu veličinu u bajtovima. Konstanta `BLKSIZE` predstavlja veličinu bloka u bajtovima. Na raspolaganju je funkcija koja učitava logički blok datog fajla i vraća pokazivač na taj učitani blok u kešu (vraća 0 u slučaju greške):

```
Byte* readFileBlock (FCB* file, BlkNo blockNo);
```

Realizovati funkciju `fread()` koja za dati fajl učitava `n` bajtova u dati bafer, počev od kurzora, i vraća broj stvarno učitanih bajtova, a kurzor pomera na kraj učitane sekvence. U slučaju prekoračenja veličine sadržaja fajla ili druge greške treba vratiti broj stvarno učitanih bajtova:

```
int fread (FCB* file, Byte* buffer, int n);
```

Rešenje

```
int fread (FCB* f, Byte* buf, int n) {
  if (f==0) return 0;
  if (f->cur+n>f->size) n = f->size - f->cur;
  int read = 0; // Number of bytes read
  while (n>0) {
    BlkNo blkNo = f->cur/BLKSIZE;
    int byte = f->cur%BLKSIZE;
    Byte* block = readFileBlock(f,blkNo);
    if (block==0) return read; // Exception
    while (byte<BLKSIZE && n>0) {
      buf[read++]=block[byte++];
      n--; f->cur++;
```

```
    }
  }
  return read;
}
```

3. zadatak, treći kolokvijum, septembar 2013.

Neki fajl sistem primenjuje kombinovanu tehniku indeksirane alokacije sadržaja fajla. U FCB fajla nalaze se dva ulaza koji predstavljaju indeks nultog nivoa (direktni pokazivači na dva prva bloka sadržaja fajla) i još četiri ulaza koji ukazuju na blokove sa indeksima prvog nivoa.

Na slici je prikazan deo FCB nekog fajla. Blok je veličine 512B, a broj bloka (svaki ulaz u indeksnom bloku) zauzima 4 bajta. Bajtovi sadržaja fajla se broje počev od 0.

1400
1401
pokazivač na tabelu 2
NULL
NULL

(a) FCB Index

NULL
2500
2501
2502
2503
...

(b) Druga tabela

U kom bloku na disku se nalazi bajt sadržaja ovog fajla sa datim rednim brojem (pored konačnog odgovora, dati i celu računicu):

1. 1000 (decimalno)?
2. 2570 (decimalno)?
3. Kolika je maksimalna veličina fajla koju dozvoljava ovaj sistem?

Rešenje

1. 1401
2. 2503
3. Jedan indeksni blok sadrži najviše 512B:4B = 128 ulaza. Maksimalna veličina fajla je: $2 \cdot 512\text{B} + 4 \cdot 128 \cdot 512\text{B} = 257\text{KB}$.

3. zadatak, treći kolokvijum, jun 2012.

Neki fajl sistem primenjuje kombinovanu tehniku indeksirane alokacije sadržaja fajla. U FCB fajla nalaze se dva ulaza koji predstavljaju indeks nultog nivoa (direktni pokazivači na dva prva bloka sadržaja fajla) i još dva ulaza koji ukazuju na blokove sa indeksima prvog nivoa.

Na slici je prikazan deo FCB nekog fajla. Blok je veličine 1KB, a broj bloka (svaki ulaz u indeksnom bloku) zauzima 4 bajta. Bajtovi sadržaja fajla se broje počev od 0.

1000
1001
pokazivač na tabelu 2
NULL

(c) FCB Index

4600
4601
4602
4603
...

(d) Druga tabela

U kom bloku na disku se nalazi bajt sadržaja ovog fajla sa datim rednim brojem (pored konačnog odgovora, dati i celu računicu):

1. 2000 (decimalno)?
2. 3570 (decimalno)?
3. Kolika je maksimalna veličina fajla koju dozvoljava ovaj sistem?

Rešenje

1. 1001
2. 4601
3. Jedan indeksni blok sadrži najviše 1024B:4B = 256 ulaza. Maksimalna veličina fajla je: $2 \cdot 1\text{KB} + 2 \cdot 256 \cdot 1\text{KB} = 514\text{KB}$.

3. zadatak, treći kolokvijum, septembar 2012.

Neki fajl sistem primenjuje indeksirani pristup alokaciji prostora za sadržaj fajla, s tim da je indeks neograničen i organizovan kao jednostruko ulančana lista indeksnih blokova. Na prvi indeksni blok u listi ukazuje polje `index` u FCB. Svaki indeksni blok sadrži `NumOfEntries` ulaza tipa `BlkNo` koji ukazuju na blokove sa sadržajem fajla i još jedan ulaz (iza ovih) istog tipa koji ukazuje na sledeći indeksni blok u listi (vrednost 0 označava *null* ulaz). Na raspolaganju je funkcija za pristup blokovima diska kroz keš, koja vraća pokazivač na deo memorije u kome se nalazi traženi blok diska učitan u keš (vraća 0 u slučaju greške):

```
Byte* getDiskBlock (BlkNo block);
```

Realizovati funkciju `getFileBlock()` koja za dati fajl dohvata logički blok sa datim brojem. U slučaju prekoračenja veličine sadržaja fajla ili druge greške treba vratiti 0.

```
Byte* getFileBlock (FCB* file, unsigned int block);
```

Rešenje

```
Byte* getFileBlock (FCB* file, unsigned int block) {
  if (file==0 || file->index==0) return 0;
  BlkNo* index = (BlkNo*)getDiskBlock(file->index);
  if (index==0) return 0;
  while (block>=NumOfEntries) {
    block -= NumOfEntries;
    if (index[NumOfEntries]==0) return 0;
    index = (BlkNo*)getDiskBlock(index[NumOfEntries]);
    if (index==0) return 0;
  }
  if (index[block]==0) return 0;
  return getDiskBlock(index[block]);
}
```

3. zadatak, treći kolokvijum, jun 2011.

Neki fajl sistem koristi FAT, uz dodatni mehanizam detekcije i oporavka od korupcije ulančanih lista na sledeći način. Kada se FAT kešira u memoriji, vidi se kao niz struktura tipa `FATEntry`. Ova struktura ima dva celobrojna polja. Polje `next` je broj ulaza u FAT u kome se nalazi sledeći element u ulančanoj listi; vrednost 0 označava kraj liste. Polje `fid` ove strukture sadrži identifikator fajla kome pripada taj element liste. U strukturi FCB celobrojno polje `id` predstavlja identifikator datog fajla, a polje `head` sadrži redni broj ulaza u FAT koji je prvi element u ulančanoj listi datog fajla.

Implementirati funkciju čiji je potpis dat dole. Ona treba da proveri konzistentnost ulančane liste datog fajla, proverom da li svi elementi liste pripadaju baš tom fajlu. Ukoliko je sve u redu, ova funkcija treba da vrati 1. Ukoliko naiđe na element u listi koji je pogrešno ulančan, odnosno ne pripada tom fajlu (tako što polje `fid` ne odgovara identifikatoru tog fajla), taj pogrešno ulančani ostatak liste treba da „odseče“ postavljanjem terminatora liste (vrednost 0 u polje `next`) u poslednji ispravan element liste (ili glavu liste, ako je prvi element pogrešan) i da vrati 0.

```
FATEntry FAT[...];

int check_consistency (FCB* file);
```

Rešenje

```
int check_consistency (FCB* file) {
  if (file==0) return 0;
  FID fid = file->id;
  int cur=file->head, prev=0;
  for (; cur!=0; prev=cur, cur=FAT[cur].next)
    if (FAT[cur].fid!=fid) {
      if (prev) FAT[prev].next=0;
      else file->head=0;
      return 0;
    }
  return 1;
}
```

3. zadatak, treći kolokvijum, septembar 2011.

Neki fajl sistem koristi indeksirani pristup alokaciji fajlova sa indeksima u dva nivoa. Polje `index` u strukturi FCB sadrži broj indeksnog bloka prvog nivoa. Broj bloka 0 u indeksu označava *null* - nealociran ulaz (fizički blok broj 0 se nikada ne koristi za fajlove). Konstanta `INDEXSIZE` definiše broj ulaza u jednom indeksu (u jednom bloku), a tip `BLKNO` predstavlja broj bloka (logičkog ili fizičkog). Realizovati funkciju koja se koristi kod direktnog pristupa fajlu:

```
void* f_getblk(FCB* file, BLKNO logical);
```

koja vraća pokazivač na dohvaćeni i keširani blok sa podacima fajla sa datim logičkim brojem. Na raspolaganju je funkcija:

```
void* f_getblk(BLKNO physical);
```

koja vraća pokazivač na keširani blok sa datim fizičkim brojem i učitava ga u keš ako je potrebno.

Rešenje

```
void* f_getblk(FCB* file, BLKNO lb) {
  if (file==0) return 0; // Exception: null FCB

  unsigned long int entry = lb/INDEXSIZE;
  if (entry>=INDEXSIZE) return 0; // Exception: lb too large!

  BLKNO* index = (BLKNO*)f_getblk(file->index);
  if (index==0) return 0; // Exception: error accessing index!

  BLKNO index2blk = index[entry];
  if (index2blk==0) return 0;  // Access over the file's boundary

  index = (BLKNO*)f_getblk(index2blk);
  if (index==0) return 0; // Exception: error accessing index!

  entry = lb%INDEXSIZE;
  return f_getblk(index[entry]);
}
```

Uvod u operativne sisteme

1. zadatak, prvi kolokvijum, april 2009.

Pored svake dole navedene tvrdnje napisati da li je ona tačna ili netačna:

1. Multiprogramski sistem bez deljenja vremena stvara privid da svaki korisnik ima svoj računar uspešnije nego sistem sa deljenim vremenom.
2. Poslovi koji rade isključivo izračunavanje (koriste samo CPU), brže će biti završeni (počev od trenutka kada su aktivirani) na sistemu sa deljenim vremenom nego na sistemu sa paketnom obradom.
3. Smanjenje kvanta vremena koje se dodeljuje procesima u sistemu sa deljenim vremenom može da poveća ukupno vreme zadržavanja procesa u sistemu.
4. U monoprogramskom multiprocesnom sistemu, u jednom trenutku u memoriji ne može biti više od jednog procesa.

Rešenje

1. Netačno.
2. Netačno.
3. Tačno.
4. Netačno.

1. zadatak, prvi kolokvijum, april 2008.

Posmatra se izvršavanje nekoliko neinteraktivnih poslova sa ulazno/izlaznim operacijama na paketnom operativnom sistemu i na operativnom sistemu sa multiprogramiranjem, pod pretpostavkom identičnih hardverskih platformi i uporednom izvršavanju na multiprogramskom sistemu. Pored svake dole navedene tvrdnje napisati da li je ona tačna ili netačna:

1. Vreme od početka do završetka izvršavanja posla kraće je na multiprogramskom sistemu.
2. Iskorišćenje procesora bolje je na multiprogramskom sistemu.
3. Propusnost sistema (broj završenih poslova u jedinici vremena, *throughput*) bolja je na multiprogramskom sistemu.
4. Režijsko vreme (beskorisno za poslove) manje je na multiprogramskom sistemu.

Rešenje

1. Netačno.
2. Tačno.
3. Tačno.
4. Netačno.

1. zadatak, prvi kolokvijum, april 2006.

Posmatra se izvršavanje nekoliko neinteraktivnih poslova sa ulazno/izlaznim operacijama na paketnom operativnom sistemu i na operativnom sistemu sa multiprogramiranjem, pod pretpostavkom identičnih hardverskih platformi i uporednom izvršavanju na multiprogramskom sistemu. Pored svake dole navedene tvrdnje napisati da li je ona tačna ili netačna:

1. Vreme od početka do završetka izvršavanja posla kraće je na multiprogramskom sistemu.
2. Iskorišćenje procesora bolje je na multiprogramskom sistemu.
3. Propusnost sistema (broj završenih poslova u jedinici vremena, *throughput*) bolja je na multiprogramskom sistemu.
4. Režijsko vreme (beskorisno za poslove) manje je na multiprogramskom sistemu.

Rešenje

1. NE - Objašnjenje: kod multiprogramskog okruženja POJEDINAČNI posao će trajati duže zbog konkurentnog izvršavanja i režijskih operacija koje se događaju u toku izvršenja posla.

2. DA - Objašnjenje: Dok jedan posao čeka na I/O, drugi se izvršava, tj. koristi processor, tako da se iskorišćenje procesora povećava.
3. DA - Objašnjenje: Pošto se poslovi izvrsavaju u paraleli (dok jedan ceka na I/O, a drugi se izvrsava), onda se ukupno vreme odziva skraćuje, pa se samim tim broj završenih poslova u jedinici vremena (*throughput*) povećava.
4. NE - Objašnjenje: Jedino reijsko vreme kod paketnog OSa je dovlačenje posla u radnu memoriju, dok kod multiprogramskog OSa postoje režijske operacije i prilikom svake zamene poslova.