1. #include "threads/synch.h"
2. #include <stdio.h>
3. #include <string.h>
4. #include "threads/interrupt.h"
5. #include "threads/thread.h"

8. *down: wait for the value to become positive, then*
9. *decrement it.*
11. *up: increment the value (and wake up one waiting*
12. *thread, if any). \*/*
13. void
14. sema\_init (struct semaphore \*sema, unsigned value)
15. {
16. ASSERT (sema != NULL);
18. sema->value = value;
19. list\_init (&sema->waiters);
20. }
22. */\* Down operation on a semaphore.  Waits for SEMA's value*
23. *to become positive and then atomically decrements it.\*/*

26. void
27. sema\_down (struct semaphore \*sema)
28. {
29. enum intr\_level old\_level;
31. ASSERT (sema != NULL);
32. ASSERT (!intr\_context ());
34. old\_level = intr\_disable ();
35. while (sema->value == 0)
36. {
38. list\_insert\_ordered (&sema->waiters, &thread\_current ()->elem,
39. more\_prio, NULL);
40. thread\_block ();
41. }
42. sema->value--;
43. intr\_set\_level (old\_level);
44. }
46. bool
47. sema\_try\_down (struct semaphore \*sema)
48. {
49. enum intr\_level old\_level;
50. bool success;
52. ASSERT (sema != NULL);
54. old\_level = intr\_disable ();
55. if (sema->value > 0)
56. {
57. sema->value--;
58. success = true;
59. }
60. else
61. success = false;
62. intr\_set\_level (old\_level);
64. return success;
65. }
67. */\* Up operation on a semaphore.  Increments SEMA's value*
68. *and wakes up one thread of those waiting for SEMA, if any*
69. *\*/*
70. void
71. sema\_up (struct semaphore \*sema)
72. {
73. enum intr\_level old\_level;
74. bool yield = false;
76. ASSERT (sema != NULL);
78. old\_level = intr\_disable ();
79. if (!list\_empty (&sema->waiters))
80. {
81. struct thread \*t = list\_entry (list\_pop\_front (&sema->waiters),
82. struct thread, elem);
83. thread\_unblock (t);
85. if (t->priority > thread\_current ()->priority)
86. yield = true;
87. }
88. sema->value++;
89. intr\_set\_level (old\_level);
91. if (yield)
92. {
93. if (!intr\_context ())
94. thread\_yield ();
95. else
96. intr\_yield\_on\_return ();
97. }
98. }
100. static void sema\_test\_helper (void \*sema\_);
102. void
103. sema\_self\_test (void)
104. {
105. struct semaphore sema[2];
106. int i;
108. printf ("Testing semaphores...");
109. sema\_init (&sema[0], 0);
110. sema\_init (&sema[1], 0);
111. thread\_create ("sema-test", PRI\_DEFAULT, sema\_test\_helper, &sema);
112. for (i = 0; i < 10; i++)
113. {
114. sema\_up (&sema[0]);
115. sema\_down (&sema[1]);
116. }
117. printf ("done.\n");
118. }
120. static void
121. sema\_test\_helper (void \*sema\_)
122. {
123. struct semaphore \*sema = sema\_;
124. int i;
126. for (i = 0; i < 10; i++)
127. {
128. sema\_down (&sema[0]);
129. sema\_up (&sema[1]);
130. }
131. }
132. */\* Initializes LOCK.*
133. *\*/*
134. void
135. lock\_init (struct lock \*lock)
136. {
137. ASSERT (lock != NULL);
139. lock->holder = NULL;
140. sema\_init (&lock->semaphore, 1);
141. }
143. void
144. lock\_acquire (struct lock \*lock)
145. {
146. ASSERT (lock != NULL);
147. ASSERT (!intr\_context ());
148. ASSERT (!lock\_held\_by\_current\_thread (lock));
150. if (!lock\_try\_acquire (lock)) */\* Someone else holding the lock. \*/*
151. {
152. sema\_down (&lock->semaphore);
153. lock->holder = thread\_current ();
154. }
155. }
157. bool
158. lock\_try\_acquire (struct lock \*lock)
159. {
160. bool success;
162. ASSERT (lock != NULL);
163. ASSERT (!lock\_held\_by\_current\_thread (lock));
165. success = sema\_try\_down (&lock->semaphore);
166. if (success)
167. lock->holder = thread\_current ();
168. return success;
169. }
171. */\* Releases LOCK,*
172. *\*/*
173. void
174. lock\_release (struct lock \*lock)
175. {
176. ASSERT (lock != NULL);
177. ASSERT (lock\_held\_by\_current\_thread (lock));
179. lock->holder = NULL;
180. sema\_up (&lock->semaphore);
182. bool
183. lock\_held\_by\_current\_thread (const struct lock \*lock)
184. {
185. ASSERT (lock != NULL);
187. return lock->holder == thread\_current ();
188. }
189. */\* One semaphore in a list. \*/*
190. struct semaphore\_elem
191. {
192. struct list\_elem elem;              */\* List element. \*/*
193. struct semaphore semaphore;
194. };
196. */\* Initializes condition variable COND.*
197. *\*/*
198. void
199. cond\_init (struct condition \*cond)
200. {
201. ASSERT (cond != NULL);
203. list\_init (&cond->waiters);
204. }
206. */\* Atomically releases LOCK and waits for COND to be signaled by*
207. *some other piece of code*
208. *\*/*
209. void
210. cond\_wait (struct condition \*cond, struct lock \*lock)
211. {
212. struct semaphore\_elem waiter;
214. ASSERT (cond != NULL);
215. ASSERT (lock != NULL);
216. ASSERT (!intr\_context ());
217. ASSERT (lock\_held\_by\_current\_thread (lock));
219. sema\_init (&waiter.semaphore, 0);
220. list\_push\_back (&cond->waiters, &waiter.elem);
221. lock\_release (lock);
222. sema\_down (&waiter.semaphore);
223. lock\_acquire (lock);
224. }
226. void
227. cond\_signal (struct condition \*cond, struct lock \*lock UNUSED)
228. {
229. ASSERT (cond != NULL);
230. ASSERT (lock != NULL);
231. ASSERT (!intr\_context ());
232. ASSERT (lock\_held\_by\_current\_thread (lock));
234. if (!list\_empty (&cond->waiters))
235. sema\_up (&list\_entry (list\_pop\_front (&cond->waiters),
236. struct semaphore\_elem, elem)->semaphore);
237. }
239. void
240. cond\_broadcast (struct condition \*cond, struct lock \*lock)
241. {
242. ASSERT (cond != NULL);
243. ASSERT (lock != NULL);
245. while (!list\_empty (&cond->waiters))
246. cond\_signal (cond, lock);
247. }