

EE445M Lab 3 Report

Yen-Kai Huang
Siavash Zanganeh Kamali

March 8, 2014

1 Objective

The goal of this lab is to extend the RTOS we built in Lab1 and Lab2 to include blocking and priority scheduling. We will also extend the RTOS to include two real-time periodic tasks, two aperiodic button tasks, and minimally invasive tools to carry out performance measures. The interpreter will be extended to allow recording debugging/performance data and downloading them to the PC screen through UART.

2 Software Design

In this lab we implemented priority scheduler, and blocking semaphore. Implementing priority scheduler requires rewriting all the functions that deal with the thread link-list structures (**AddThread**, **wake-up background thread**, **Signal**, **Kill**, **Sleep**, **Block**)

```
1 /***** System Performance Measurement *****/
2
3 #ifdef __Performance_Measurement__
4
5
6     #ifdef __Jitter_Measurement__
7         // Measure jitter of periodic tasks
8         #define JITTERSIZE 64
9
10        unsigned long Period1;
11        long MaxJitter1;           // largest time jitter between
12        long MinJitter1;          // smallest time jitter between
13        unsigned long const JitterSize1=JITTERSIZE;
14        unsigned long JitterHistogram1[JITTERSIZE]={0,};
15
16        unsigned long Period2;
17        long MaxJitter2;           // largest time jitter between
18        long MinJitter2;          // smallest time jitter between
19        unsigned long const JitterSize2=JITTERSIZE;
20        unsigned long JitterHistogram2[JITTERSIZE]={0,};
21    #endif
22
23    #ifdef __Critical_Interval_Measurement__
24
25        void DisableInterrupts_check(void);
26        void EnableInterrupts_check(void);
```

```

27 long StartCritical_check(void);
28 void EndCritical_check(long sr);
29 #endif
30
31 #ifdef __Profiling__
32
33 #define PROFILELENGTH 100
34
35 #define Profile_PendSV_Trigger 1
36 #define Profile_SysTick_Starts 2
37 #define Profile_SysTick_End 3
38 #define Profile_Timer1_Starts 4
39 #define Profile_Timer1_End 5
40 #define Profile_Timer2_Starts 6
41 #define Profile_Timer2_End 7
42 #define Profile_Timer3_Starts 8
43 #define Profile_Timer3_End 9
44 #define Profile_Timer4_Starts 10
45 #define Profile_Timer4_End 11
46
47 typedef struct {
48     unsigned char value;
49     unsigned long time;
50 } TimeStamp;
51
52 TimeStamp Profile[PROFILELENGTH];
53 TimeStamp *ProfilePt = Profile;
54
55 void timestamp(unsigned char value);
56
57 #endif
58 #endif
59
60 /***** Datastructure for OS Round-robin scheduler *****/
61
62 #define NUM_PRIORITY 8
63 #define STACKSIZE_MAX 200
64 #define STACKSIZE_MIN 64
65
66 typedef struct tcb_s {
67     long *sp;
68     struct tcb_s *next;
69     struct tcb_s *prev;
70     long *stack;
71     unsigned int id;
72     unsigned long sleepCount;
73     unsigned char priority;
74 } TCB;
75
76 static unsigned long uniqid = 0;
77
78 TCB *RunPt, *NextPt, *SleepPt;
79 TCB *PriPt[NUM_PRIORITY];
80
81 static unsigned long Timer=0;
82

```

```

83 /*****
84
85
86 void OS_Init(void) { volatile unsigned long delay;
87     OS_DisableInterrupts();
88     NVIC_ST_CTRL_R = 0;
89     NVIC_ST_CURRENT_R = 0;
90     NVIC_SYS_PRI3_R =(NVIC_SYS_PRI3_R&0x00FFFFFF)|0xC0000000;
91     NVIC_SYS_PRI3_R =(NVIC_SYS_PRI3_R&0xFF00FFFF)|0x00E00000;
92
93     SYSCTL_RCGC1_R |= SYSCTL_RCGC1_TIMER1;    delay = SYSCTL_RCGC1_R;
94     TIMER1_CTL_R &= ~TIMER_CTL_TAEN;
95     TIMER1_CFG_R = TIMER_CFG_32_BIT_TIMER;
96     TIMER1_TAMR_R = TIMER_TAMR_TAMR_PERIOD;
97     TIMER1_TAILR_R = 800000-1;
98     TIMER1_ICR_R = TIMER_ICR_TATOCINT;
99     TIMER1_IMR_R |= TIMER_IMR_TATOIM;
100    NVIC_PRI5_R = (NVIC_PRI5_R&0xFFFF00FF)|0x00008000;
101    NVIC_ENO_R = NVIC_ENO_INT21;
102    TIMER1_CTL_R |= TIMER_CTL_TAEN;
103
104    SYSCTL_RCGC1_R |= SYSCTL_RCGC1_TIMER2;
105    delay = SYSCTL_RCGC1_R;
106    TIMER2_CTL_R &= ~TIMER_CTL_TAEN;
107    TIMER2_CFG_R = TIMER_CFG_32_BIT_TIMER;
108    TIMER2_TAMR_R = TIMER_TAMR_TAMR_PERIOD;
109    TIMER2_TAILR_R = 0xFFFFFFFF;
110    TIMER2_CTL_R = TIMER_CTL_TAEN;
111
112    SYSCTL_RCGC2_R |= SYSCTL_RCGC2_GPIOF;
113
114    GPIO_PORTF_LOCK_R = GPIO_LOCK_KEY;
115    GPIO_PORTF_CR_R |= (SW1|SW2);
116    GPIO_PORTF_LOCK_R = 0;
117
118    GPIO_PORTF_DIR_R &= ~(SW1|SW2);
119    GPIO_PORTF_AFSEL_R &= ~(SW1|SW2);
120    GPIO_PORTF_DEN_R |= (SW1|SW2);
121    GPIO_PORTF_PCTL_R &= ~0x000F000F;
122    GPIO_PORTF_AMSEL_R = 0;
123    GPIO_PORTF_PUR_R |= (SW1|SW2);
124    GPIO_PORTF_IS_R &= ~(SW1|SW2);
125    GPIO_PORTF_IBE_R |= (SW1|SW2);
126    GPIO_PORTF_IEV_R &= ~(SW1|SW2);
127    GPIO_PORTF_ICR_R = (SW1|SW2);
128    GPIO_PORTF_IM_R |= (SW1|SW2);
129
130    Heap_Init();
131
132    OS_AddThread(dummyThread,32,NUM_PRIORITY-1);
133 }
134
135 void OS_InitSemaphore(Sema4Type *semaPt, long value){
136     semaPt->Value = value;
137     semaPt->BlocketListPt = 0;
138 }

```

```

139
140 void OS_Wait(Sema4Type *semaPt){
141     #ifdef __Critical_Interval_Measurement__
142     long sr = StartCritical_check();
143     #else
144     long sr = StartCritical();
145     #endif
146
147     semaPt->Value -= 1;
148
149     if(semaPt->Value < 0) {
150         if (RunPt->next == RunPt) {
151             PriPt[RunPt->priority] = NULL;
152         } else {
153             DLUNLINK(RunPt);
154             PriPt[RunPt->priority] = RunPt->next;
155         }
156
157         if(semaPt->BlocketListPt) {
158             DLLINKLAST(RunPt, semaPt->BlocketListPt);
159         } else {
160             RunPt->next = RunPt->prev = RunPt;
161             semaPt->BlocketListPt = RunPt;
162         }
163         updateNextPt(RunPt->priority);
164         triggerPendSV();
165     }
166
167     #ifdef __Critical_Interval_Measurement__
168     EndCritical_check(sr);
169     #else
170     EndCritical(sr);
171     #endif
172 }
173
174 void OS_Signal(Sema4Type *semaPt){
175     #ifdef __Critical_Interval_Measurement__
176     long sr = StartCritical_check();
177     #else
178     long sr = StartCritical();
179     #endif
180
181     TCB *current = semaPt->BlocketListPt;
182     semaPt->Value += 1;
183
184     if(semaPt->Value <= 0) {
185         if (current->next == current) {
186             semaPt->BlocketListPt = NULL;
187         } else {
188             DLUNLINK(current);
189             semaPt->BlocketListPt = current->next;
190         }
191
192         if(PriPt[current->priority]) {
193             DLLINK(current, PriPt[current->priority]);
194         } else {

```

```

195     current->next = current->prev = current;
196     PriPt[current->priority] = current;
197 }
198 updateNextPt(0);
199 triggerPendSV();
200 }
201
202 #ifdef __Critical_Interval_Measurement__
203 EndCritical_check(sr);
204 #else
205 EndCritical(sr);
206 #endif
207 }
208
209 void OS_bWait(Sema4Type *semaPt){
210     OS_Wait(semaPt);
211 }
212
213 void OS_bSignal(Sema4Type *semaPt){
214     #ifdef __Critical_Interval_Measurement__
215     long sr = StartCritical_check();
216     #else
217     long sr = StartCritical();
218     #endif
219
220     TCB *current = semaPt->BlocketListPt;
221     if(semaPt->Value < 1) semaPt->Value += 1;
222     if(semaPt->Value <= 0) {
223         if (current->next == current) {
224             semaPt->BlocketListPt = NULL;
225         } else {
226             DLUNLINK(current);
227             semaPt->BlocketListPt = current->next;
228         }
229
230         if(PriPt[current->priority]) {
231             DLLINK(current, PriPt[current->priority]);
232         } else {
233             current->next = current->prev = current;
234             PriPt[current->priority] = current;
235         }
236         updateNextPt(0);
237         triggerPendSV();
238     }
239
240     #ifdef __Critical_Interval_Measurement__
241     EndCritical_check(sr);
242     #else
243     EndCritical(sr);
244     #endif
245 }
246
247 int OS_AddThread(void(*task)(void),
248                 unsigned long stackSize,
249                 unsigned long priority) {
250     TCB *tcb; long sr;

```

```

251
252 if (priority >= NUM_PRIORITY) priority = NUM_PRIORITY-1;
253
254 stackSize = (stackSize < STACKSIZE_MIN) ? STACKSIZE_MIN : ( (
    stackSize > STACKSIZE_MAX) ? STACKSIZE_MAX : stackSize ) ;
255
256 #ifdef __Critical_Interval_Measurement__
257 sr = StartCritical_check();
258 #else
259 sr = StartCritical();
260 #endif
261
262
263
264 if (!(tcb = (TCB *) Heap_Malloc(sizeof(TCB)))){
265     return NULL;
266 };
267 if (!(tcb->stack = (long *) Heap_Malloc (sizeof(long)*stackSize))){
268     Heap_Free (tcb);
269     return NULL;
270 };
271 SetInitialStack(tcb, stackSize);
272
273     if(PriPt[priority]) {
274         DLLINKLAST(tcb, PriPt[priority]);
275     } else {
276         tcb->next = tcb->prev = tcb;
277         PriPt[priority] = tcb;
278     }
279
280 tcb->stack[stackSize-2] = (long) (task);
281 tcb->sleepCount = 0;
282 tcb->priority = priority;
283 tcb->id = OS_Id();
284 updateNextPt(0);
285
286 if(!RunPt) RunPt = NextPt;
287
288 updateNextPt(0);
289 triggerPendSV();
290 #ifdef __Critical_Interval_Measurement__
291 EndCritical_check(sr);
292 #else
293 EndCritical(sr);
294 #endif
295
296 return 1;
297 }
298
299 unsigned long OS_Id(void) {
300     return uniqid++;
301 }
302
303 #define IDLE          0
304 #define ONE_IN_USE 1
305 int OS_AddPeriodicThread(void(*task)(void), unsigned long period,

```

```

306     unsigned long priority){
307     long sr; volatile unsigned long delay;
308     static char currentTimer = 0;
309     if (priority > 5) {
310         priority = 5;
311     }
312
313     #ifdef __Critical_Interval_Measurement__
314     sr = StartCritical_check();
315     #else
316     sr = StartCritical();
317     #endif
318
319     switch(currentTimer) {
320     case IDLE:
321         SYSCTL_RCGC1_R |= SYSCTL_RCGC1_TIMER3;
322
323         delay = SYSCTL_RCGC1_R;
324         delay = SYSCTL_RCGC1_R;
325         PeriodicTask1 = task;
326         TIMER3_CTL_R &= ~TIMER_CTL_TAEN;
327         TIMER3_CFG_R = TIMER_CFG_32_BIT_TIMER;
328         TIMER3_TAMR_R = TIMER_TAMR_TAMR_PERIOD;
329         TIMER3_TAILR_R = period-1;
330         TIMER3_ICR_R = TIMER_ICR_TATOCINT;
331         TIMER3_IMR_R |= TIMER_IMR_TATOIM;
332         NVIC_PRI8_R = (NVIC_PRI8_R&0x00FFFFFF)|((priority)<<29);
333         NVIC_EN1_R |= NVIC_EN1_INT35;
334         TIMER3_CTL_R |= TIMER_CTL_TAEN;
335         currentTimer++;
336
337         #ifdef __Jitter_Measurement__
338         Period1 = period;
339         #endif
340
341         break;
342     case ONE_IN_USE:
343         SYSCTL_RCGCTIMER_R |= SYSCTL_RCGCTIMER_R4;
344         delay = SYSCTL_RCGC1_R;
345         delay = SYSCTL_RCGC1_R;
346         PeriodicTask2 = task;
347         TIMER4_CTL_R &= ~TIMER_CTL_TAEN;
348         TIMER4_CFG_R = TIMER_CFG_32_BIT_TIMER;
349         TIMER4_TAMR_R = TIMER_TAMR_TAMR_PERIOD;
350         TIMER4_TAILR_R = period-1;
351         TIMER4_ICR_R = TIMER_ICR_TATOCINT;
352         TIMER4_IMR_R |= TIMER_IMR_TATOIM;
353         NVIC_PRI17_R = (NVIC_PRI17_R&0xFF00FFFF)|((priority)<<21);
354         NVIC_EN2_R |= NVIC_EN2_INT70;
355         TIMER4_CTL_R |= TIMER_CTL_TAEN;
356         currentTimer++;
357
358         #ifdef __Jitter_Measurement__
359         Period2 = period;
360         #endif

```

```

361     break;
362     default:
363         break;
364 }
365
366 #ifdef __Critical_Interval_Measurement__
367 EndCritical_check(sr);
368 #else
369 EndCritical(sr);
370 #endif
371
372 return 1;
373 }
374 #undef IDLE
375 #undef ONE_IN_USE
376
377 #define IDLE      0
378 #define ONE_IN_USE 1
379 #define BOTH_IN_USE 2
380
381 static unsigned char buttonuse = IDLE;
382 static unsigned long sw1priority;
383 static unsigned long sw2priority;
384
385 int OS_AddSW1Task(void(*task)(void), unsigned long priority){
386     SW1Task = task;
387     sw1priority = (priority & 0x07)<<21;
388
389     if(buttonuse == IDLE) {
390         NVIC_ENO_R |= NVIC_ENO_INT30;
391     }
392
393     NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|sw1priority;
394
395     buttonuse += 1;
396
397     return 1;
398 }
399
400 int OS_AddSW2Task(void(*task)(void), unsigned long priority){
401     SW2Task = task;
402     sw2priority = (priority & 0x07)<<21;
403
404     if(buttonuse == IDLE) {
405         NVIC_ENO_R |= NVIC_ENO_INT30;
406     }
407
408     NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|sw2priority;
409
410     buttonuse += 1;
411
412     return 1;
413 }
414
415 int OS_RemoveSW1Task(void) {
416     if(SW1Task == dummy) {

```



```

417     return 0;
418 }
419
420 if(--buttonuse) {      NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|
sw2priority;    } else {
421     NVIC_ENO_R &= ~NVIC_ENO_INT30;    }
422
423 SW1Task = dummy;
424 return 1;
425 }
426
427 int OS_RemoveSW2Task(void) {
428     if(SW2Task == dummy) {
429         return 0;
430     }
431
432     if(--buttonuse) {
433         NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|sw1priority;
434     } else {
435         NVIC_ENO_R &= ~NVIC_ENO_INT30;
436     }
437
438     SW2Task = dummy;
439     return 1;
440 }
441
442 #ifdef DEPRECATE
443 int OS_AddButtonTask(void(*task)(void), unsigned long priority) {
444     OS_AddSW1Task(task, priority);
445 }
446 #endif
447
448 #undef IDLE
449 #undef ONE_IN_USE
450 #undef BOTH_IN_USE
451
452 int OS_AddDownTask(void(*task)(void), unsigned long priority);
453
454 void OS_Sleep(unsigned long sleepTime){
455     TCB *tcb = RunPt;
456
457     #ifdef __Critical_Interval_Measurement__
458     DisableInterrupts_check();
459     #else
460     DisableInterrupts();
461     #endif
462
463     if (RunPt->next == RunPt) {
464         PriPt[RunPt->priority] = NULL;
465         updateNextPt(RunPt->priority);
466     } else {
467         DLUNLINK(RunPt);
468         PriPt[RunPt->priority] = RunPt->next;
469         NextPt = RunPt->next;
470     }
471 }

```

```

472     if(SleepPt) {
473         DLLINK(tcb, SleepPt);
474     } else {
475         tcb->next = tcb->prev = tcb;
476         SleepPt = tcb;
477     }
478     tcb->sleepCount = sleepTime;
479
480     triggerPendSV();
481
482     #ifdef __Critical_Interval_Measurement__
483     EnableInterrupts_check();
484     #else
485     EnableInterrupts();
486     #endif
487 }
488
489 void OS_Kill(void){
490
491     #ifdef __Critical_Interval_Measurement__
492     DisableInterrupts_check();
493     #else
494     DisableInterrupts();
495     #endif
496
497     Heap_Free(RunPt->stack);
498     if (RunPt->next == RunPt) {
499         PriPt[RunPt->priority] = NULL;
500     } else {
501         DLUNLINK(RunPt);
502         PriPt[RunPt->priority] = RunPt->next;
503     }
504     Heap_Free(RunPt);
505     updateNextPt(0);
506     triggerPendSV();
507
508     #ifdef __Critical_Interval_Measurement__
509     EnableInterrupts_check();
510     #else
511     EnableInterrupts();
512     #endif
513
514 }
515
516
517 void OS_Suspend(void){
518     NVIC_ST_CURRENT_R = 0;          NVIC_INT_CTRL_R |=
519     NVIC_INT_CTRL_PENDSTSET; }
520
521 void OS_Fifo_Init(unsigned long size) {
522     OSFifo_Init();
523     OS_InitSemaphore(&Sema4fifo, 0);
524 }
525
526 int OS_Fifo_Put(unsigned long data) {
527     if(OSFifo_Put(data) == FIFOSUCCESS) {

```

```

527     OS_Signal(&Sema4fifo);
528     return FIFOSUCCESS;
529 } else {
530     return FIFOFAIL;
531 }
532 }
533
534 unsigned long OS_Fifo_Get(void) {
535     unsigned long data;
536     OS_Wait(&Sema4fifo);
537     OSFifo_Get(&data);
538     return data;
539 }
540
541 long OS_Fifo_Size(void) {
542     return OSFifo_Size();
543 }
544
545 void OS_MailBox_Init(void) {
546     OS_InitSemaphore(&Sema4MailboxEmpty, 1);
547     OS_InitSemaphore(&Sema4MailboxFull, 0);
548 }
549
550 void OS_MailBox_Send(unsigned long data) {
551     OS_bSignal(&Sema4MailboxFull);
552     OS_bWait(&Sema4MailboxEmpty);
553     Mailbox = data;
554 }
555
556 unsigned long OS_MailBox_Recv(void) {
557     OS_bSignal(&Sema4MailboxEmpty);
558     OS_bWait(&Sema4MailboxFull);
559     return Mailbox;
560 }
561
562 unsigned long OS_Time(void){
563     return TIMER2_TAV_R;
564 }
565
566 unsigned long OS_TimeDifference(unsigned long stop, unsigned long start
567 ) {
568     return (start-stop);
569 }
570
571 void OS_ClearMsTime(void){
572     Timer = 0;
573 }
574
575 unsigned long OS_MsTime(void){
576     return Timer;
577 }
578
579 void OS_Launch(unsigned long theTimeSlice){
580     NVIC_ST_RELOAD_R = theTimeSlice - 1;    NVIC_ST_CTRL_R = 0x00000007;
        StartOS();
}

```

```

581
582 void SysTick_Handler(void) { long sr;
583     #ifdef __Critical_Interval_Measurement__
584     sr = StartCritical_check();
585     #else
586     sr = StartCritical();
587     #endif
588
589     #ifdef __Profiling__
590     timestamp(Profile_SysTick_Starts);
591     #endif
592
593     NextPt = RunPt->next;
594     NVIC_INT_CTRL_R |= NVIC_INT_CTRL_PEND_SV;
595     #ifdef __Profiling__
596     timestamp(Profile_PendSV_Trigger);
597     #endif
598
599     #ifdef __Profiling__
600     timestamp(Profile_SysTick_End);
601     #endif
602
603     #ifdef __Critical_Interval_Measurement__
604     EndCritical_check(sr);
605     #else
606     EndCritical(sr);
607     #endif
608
609 }
610
611
612 static void triggerPendSV (void) {
613     #ifdef __Profiling__
614     timestamp(Profile_PendSV_Trigger);
615     #endif
616
617     NVIC_ST_CURRENT_R = 0;
618     NVIC_INT_CTRL_R |= NVIC_INT_CTRL_PEND_SV;
619 }
620
621 static void updateNextPt (int startLevel) {
622     int i;
623     for (i=startLevel; i<NUM_PRIORITY; i++) {
624         if (PriPt[i]){
625             NextPt = PriPt[i];
626             break;
627         }
628     }
629 }
630
631
632 void Timer1A_Handler(void){
633     TCB *current;
634     int minPriority = NUM_PRIORITY;
635
636     #ifdef __Profiling__

```

```

637 timestamp(Profile_Timer1_Starts);
638 #endif
639
640 TIMER1_ICR_R = TIMER_ICR_TATOCINT;
641 ++Timer;
642
643 if(!SleepPt) {
644     #ifdef __Profiling__
645     timestamp(Profile_Timer1_End);
646     #endif
647
648     return;
649 }
650 current = SleepPt;
651 do {
652     TCB *currentNext = current->next;
653     if(!(--(current->sleepCount))) {
654         minPriority = (current->priority < minPriority) ? (current->
priority) : minPriority;
655         if (current->next == current) {
656             SleepPt = NULL;
657             if(PriPt[current->priority]) {
658                 DLLINK(current, PriPt[current->priority]);
659             } else {
660                 current->next = current->prev = current;
661                 PriPt[current->priority] = current;
662             }
663             break;
664         } else {
665             DLUNLINK (current);
666             SleepPt = current->next;
667             if(PriPt[current->priority]) {
668                 DLLINK(current, PriPt[current->priority]);
669             } else {
670                 current->next = current->prev = current;
671                 PriPt[current->priority] = current;
672             }
673         }
674     }
675     current = currentNext;
676 } while (current != SleepPt);
677 if (minPriority < RunPt->priority ) {
678     updateNextPt(0);
679     triggerPendSV();
680 }
681
682 #ifdef __Profiling__
683 timestamp(Profile_Timer1_End);
684 #endif
685 }
686
687
688
689
690 #ifdef __Jitter_Measurement__
691 unsigned static long LastTime1; #define IDLE 0

```

```

692 #define STARTED 1
693 unsigned char MeasureState1 = IDLE;
694 #endif
695
696 void Timer3A_Handler(void){
697 #ifdef __Jitter_Measurement__
698     long jitter;
699     int index;
700     unsigned long thisTime = OS_Time();
701 #endif
702
703     #ifdef __Profiling__
704     timestamp(Profile_Timer3_Starts);
705     #endif
706
707     TIMER3_ICR_R = TIMER_ICR_TATOCINT;
708
709     (*PeriodicTask1)();
710
711     #ifdef __Jitter_Measurement__
712     if(MeasureState1 == STARTED) {
713         jitter = (long)(OS_TimeDifference(thisTime,LastTime1)/80)-(long)(
714         Period1/80);      MaxJitter1 = (jitter > MaxJitter1)? jitter:
715         MaxJitter1;
716         MinJitter1 = (jitter < MinJitter1)? jitter: MinJitter1;
717         index = jitter+JITTERSIZE/2;      if(index < 0) index = 0;
718         if(index>=JitterSize1) index = JITTERSIZE-1;
719         JitterHistogram1[index]++;
720     } else {      MeasureState1 = STARTED;
721     }
722     LastTime1 = thisTime;
723     #endif
724
725     #ifdef __Profiling__
726     timestamp(Profile_Timer3_End);
727     #endif
728 }
729
730 #ifdef __Jitter_Measurement__
731 unsigned static long LastTime2;
732 unsigned char MeasureState2 = IDLE;
733 #endif
734
735 void Timer4A_Handler(void){
736 #ifdef __Jitter_Measurement__
737     long jitter;
738     int index;
739     unsigned long thisTime = OS_Time();
740 #endif
741
742     #ifdef __Profiling__
743     timestamp(Profile_Timer4_Starts);
744     #endif
745
746     TIMER4_ICR_R = TIMER_ICR_TATOCINT;

```

```

746
747     (*PeriodicTask2)();
748
749 #ifdef __Jitter_Measurement__
750     if(MeasureState2 == STARTED) {
751         jitter = (long)(OS_TimeDifference(thisTime,LastTime2)/80)-(long)(
Period2/80);        MaxJitter2 = (jitter > MaxJitter2)? jitter:
MaxJitter2;
752         MinJitter2 = (jitter < MinJitter2)? jitter: MinJitter2;
753         index = jitter+JITTERSIZE/2;        if(index < 0) index = 0;
754         if(index>=JitterSize2) index = JITTERSIZE-1;
755         JitterHistogram2[index]++;
756     } else {        MeasureState2 = STARTED;
757     }
758     LastTime2 = thisTime;
759 #endif
760
761 #ifdef __Profiling__
762     timestamp(Profile_Timer4_End);
763 #endif
764 }
765
766 #ifdef __Jitter_Measurement__
767 void Jitter(void) {
768     printf("Timer1 jitter = %ld\r\nTimer2 jitter = %ld\r\n", MaxJitter1 -
MinJitter1, MaxJitter2 - MinJitter2);
769 }
770 #endif
771
772 #undef IDLE
773 #undef STARTED
774
775 void static DebounceTask1(void){
776     OS_Sleep(1);
777     LastPF4 = PF4;
778     GPIO_PORTF_ICR_R = SW1;
779     GPIO_PORTF_IM_R |= SW1;
780
781     OS_Kill();
782 }
783
784 void static DebounceTask2(void){
785     OS_Sleep(1);
786     LastPF0 = PF0;
787     GPIO_PORTF_ICR_R = SW2;
788     GPIO_PORTF_IM_R |= SW2;
789
790     OS_Kill();
791 }
792
793 void GPIOPortF_Handler(void){
794     unsigned long vector = GPIO_PORTF_MIS_R;
795     if(vector & SW1) {
796         if(LastPF4){
797             (*SW1Task)();
798         }

```

```

799
800     OS_AddThread(&DebounceTask1, 32, 0);
801     GPIO_PORTF_IM_R &= ~SW1;
802
803 } else {
804     if(LastPF0){
805         (*SW2Task)();
806     }
807
808     OS_AddThread(&DebounceTask2, 32, 0);
809     GPIO_PORTF_IM_R &= ~SW2;
810 }
811 }
812
813 /***** Performance Measurement *****/
814 #ifdef __Critical_Interval_Measurement__
815
816 unsigned long MaxCriticalInterval;
817 unsigned long TotalCriticalInterval;
818 unsigned long StartCriticalTime;
819
820 unsigned int CS_Duration;
821
822 long StartCritical_check(void) {
823     long sr = StartCritical();
824
825     StartCriticalTime = OS_Time();
826
827     return sr;
828 }
829
830 void EndCritical_check(long sr) {
831     unsigned long endCriticalTime = OS_Time();
832     long criticalInterval = (long)(OS_TimeDifference(endCriticalTime,
833         StartCriticalTime)/80);
834     TotalCriticalInterval += criticalInterval;
835     MaxCriticalInterval = (criticalInterval > MaxCriticalInterval)?
836         criticalInterval : MaxCriticalInterval;
837
838     EndCritical(sr);
839 }
840
841 void EnableInterrupts_check(void) {
842     EnableInterrupts();
843     StartCriticalTime = OS_Time();
844 }
845
846 void DisableInterrupts_check(void) {
847     unsigned long endCriticalTime = OS_Time();
848     long criticalInterval = (long)(OS_TimeDifference(endCriticalTime,
849         StartCriticalTime)/80);
850     TotalCriticalInterval += criticalInterval;
851     MaxCriticalInterval = (criticalInterval > MaxCriticalInterval)?
852         criticalInterval : MaxCriticalInterval;
853
854     DisableInterrupts();

```



```

851 }
852 }
853
854 Sema4Type Sema4CriticalIntervalReady;
855
856 static void CriticalIntervalThread(void) {
857     OS_Sleep(CS_Duration);    OS_Signal(&Sema4CriticalIntervalReady);
858     OS_Kill();
859     while(1);
860 }
861
862 void OS_CriticalInterval_Start(int duration) {
863     CS_Duration = duration;
864     StartCriticalTime = MaxCriticalInterval = TotalCriticalInterval = 0;
865
866     OS_InitSemaphore(&Sema4CriticalIntervalReady, 0);
867     OS_AddThread(CriticalIntervalThread, 32, 0);
868
869     OS_Wait(&Sema4CriticalIntervalReady);
870     printf("Maximum time in critical section = %ld\r\n" \
871           "Percentage of time in critical section = %ld/%d = %ld.%.2ld%%\r\n", \
872           MaxCriticalInterval, TotalCriticalInterval, duration*10000,
873           TotalCriticalInterval/(duration*100), TotalCriticalInterval%(
874           duration*100));
875 }
876
877 #endif
878
879 #ifdef __Profiling__
880 Sema4Type Sema4ProfilingDone;
881
882 unsigned char Profiling;
883 unsigned long ProfileStartTime;
884
885 void timestamp(unsigned char value) {
886     if(!Profiling) return;
887
888     if(ProfilePt >= &Profile[PROFILELENGTH]) {
889         Profiling = 0;
890         OS_Signal(&Sema4ProfilingDone);
891
892         return;
893     }
894
895     ProfilePt->value = value;
896     ProfilePt->time = ProfileStartTime - OS_Time();
897     ProfilePt++;
898
899 }
900
901 void Profile_Dump(void) {
902     TimeStamp *pt;
903
904

```

```

905     for(pt = Profile; pt < ProfilePt; ++pt) {
906         printf("[%07lu] Thread-%d\r\n", pt->time, pt->value);
907     }
908 }
909
910 void OS_Profile_Start(void) {
911     Profiling = 1;
912     ProfilePt = Profile;
913     ProfileStartTime = OS_Time();
914
915     OS_InitSemaphore(&Sema4ProfilingDone, 0);
916     OS_Wait(&Sema4ProfilingDone);
917
918     Profile_Dump();
919 }
920
921 #endif

```

Code/OS.c

The change does not involve osasm.c

```

1  ;/* OSasm.s: low-level OS commands, written in assembly */
2  ;// Real Time Operating System
3
4  ; This example accompanies the book
5  ; "Embedded Systems: Real Time Interfacing to the Arm Cortex M3",
6  ; ISBN: 978-1463590154, Jonathan Valvano, copyright (c) 2011
7  ;
8  ; Programs 6.4 through 6.12, section 6.2
9  ;
10 ;Copyright 2011 by Jonathan W. Valvano, valvano@mail.utexas.edu
11 ; You may use, edit, run or distribute this file
12 ; as long as the above copyright notice remains
13 ; THIS SOFTWARE IS PROVIDED "AS IS". NO WARRANTIES, WHETHER EXPRESS,
14 ; IMPLIED
15 ; OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
16 ; MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS
17 ; SOFTWARE.
18 ; VALVANO SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL,
19 ; INCIDENTAL,
20 ; OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
21 ; For more information about my classes, my research, and my books, see
22 ; http://users.ece.utexas.edu/~valvano/
23 ; */
24
25     AREA |.text|, CODE, READONLY, ALIGN=2
26     THUMB
27     REQUIRE8
28     PRESERVE8
29
30     EXTERN  RunPt          ; currently running thread
31     EXTERN  NextPt
32     EXPORT  OS_DisableInterrupts
33     EXPORT  OS_EnableInterrupts
34     EXPORT  StartOS

```

```

33      EXPORT    PendSV_Handler
34
35 PendSV_Handler          ; 1) Saves R0-R3,R12,LR,PC,PSR
36     CPSID      I        ; 2) Prevent interrupt during switch
37     PUSH       {R4-R11} ; 3) Save remaining regs r4-11
38     LDR        R0, =RunPt ; 4) R0=pointer to RunPt, old thread
39     LDR        R1, [R0]   ; 5) R1 = RunPt
40     STR        SP, [R1]   ; 5) Save SP into TCB
41     LDR        R1, =NextPt
42     LDR        R1, [R1]   ; 6) R1 = NextPt
43     STR        R1, [R0]   ; 6) RunPt = NextPt
44     LDR        SP, [R1]   ; 7) new thread SP;
45                               ; 7) SP = RunPt->sp;
46     POP        {R4-R11}   ; 8) restore regs r4-11
47     CPSIE      I        ; 9) tasks run with interrupts enabled
48     BX         LR        ; 10) restore R0-R3,R12,LR,PC,PSR
49
50
51 StartOS
52     LDR        R0, =RunPt ; currently running thread
53     LDR        R2, [R0]   ; R2 = value of RunPt
54     LDR        SP, [R2]   ; new thread SP; SP = RunPt->
55     stackPointer;
56     POP        {R4-R11}   ; restore regs r4-11
57     POP        {R0-R3}   ; restore regs r0-3
58     POP        {R12}
59     POP        {LR}      ; discard LR from initial stack
60     POP        {LR}      ; start location
61     POP        {R1}      ; discard PSR
62     CPSIE      I        ; Enable interrupts at processor level
63     BX         LR        ; start first thread
64
65     ALIGN
66     END

```

Code/osasm.s

Then we test the blocking semaphore with this program.

```

1 Sema4Type Readyc;          // set in background
2 int Lost;
3 void BackgroundThread1c(void){
4     Count1++;
5     OS_Signal(&Readyc);
6 }
7 void Thread5c(void){
8     for(;;){
9         OS_Wait(&Readyc);
10        Count5++; // Count2 + Count5 should equal Count1
11        Lost = Count1-Count5-Count2;
12    }
13 }
14 void Thread2c(void){
15     OS_InitSemaphore(&Readyc,0);
16     Count1 = 0; // number of times signal is called
17     Count2 = 0;
18     Count5 = 0; // Count2 + Count5 should equal Count1

```

```

19 NumCreated += OS_AddThread(&Thread5c,128,3);
20 OS_AddPeriodicThread(&BackgroundThread1c,TIME_1MS,0);
21 for(;;){
22     OS_Wait(&ReadyC);
23     Count2++;    // Count2 + Count5 should equal Count1
24 }
25 }
26
27 int Testmain3(void){
28     Count4 = 0;
29     OS_Init();
30
31     NumCreated = 0 ;
32     OS_AddSW1Task(&BackgroundThread5c,2);
33     NumCreated += OS_AddThread(&Thread2c,128,2);
34     NumCreated += OS_AddThread(&Thread3c,128,3);
35     NumCreated += OS_AddThread(&Thread4c,128,3);
36     OS_Launch(TIME_2MS);
37     return 0;
38 }

```

Code/Testmain3_blocking.c

We also measured the jitter with this program.

```

1 extern void Jitter(void);    // prints jitter information (write this)
2 void Thread7(void){
3     OS_bWait(&Sema4UART);
4     UART_OutString("\n\rEE345M/EE380L, Lab 3 Preparation 2\n\r");
5     OS_bSignal(&Sema4UART);
6
7     OS_Sleep(1000);    // 10 seconds
8
9     OS_bWait(&Sema4UART);
10    Jitter();    // print jitter information
11    UART_OutString("\n\r\n\r");
12    OS_bSignal(&Sema4UART);
13
14    OS_Kill();
15 }
16
17 #define workA 500    // {5,50,500 us} work in Task A
18 #define counts1us 80    // number of OS_Time counts per 1us
19 void TaskA(void){
20     PE1 = 0x02;
21     CountA++;
22     PseudoWork(workA*counts1us); // do work (100ns time resolution)
23     PE1 = 0x00;
24 }
25 #define workB 250    // 250 us work in Task B
26 void TaskB(void){
27     PE2 = 0x04;
28     CountB++;
29     PseudoWork(workB*counts1us); // do work (100ns time resolution)
30     PE2 = 0x00;
31 }
32

```

```

33 int main(void) {
34     PLL_Init();
35     OS_InitSemaphore(&Sema4UART, 1);
36
37     Debug_PortE_Init();
38     OS_Init();
39     NumCreated = 0 ;
40     NumCreated += OS_AddThread(&Interpreter, 128, 1);
41     NumCreated += OS_AddThread(&Thread6, 128, 2);
42     NumCreated += OS_AddThread(&Thread7, 128, 1);
43
44     OS_AddPeriodicThread(&TaskA, TIME_1MS, 0);
45     OS_AddPeriodicThread(&TaskB, 2*TIME_1MS, 1);
46
47     OS_Launch(TIME_2MS);
48     return 0;
49 }

```

Code/Jitter_measurement.c

3 Measurement

(a) Plot of the logic analyzerrunning the blocking/sleeping/killing/round-robin system

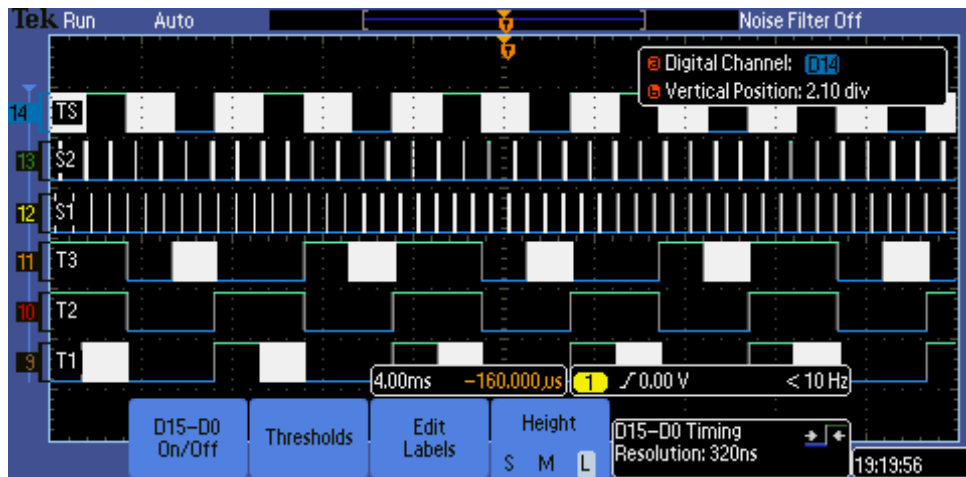
We imitate a round-robin system by setting all the priorities of threads (T1, T2, T3) to an equal value. Background thread S1 and S2 are signalling until a counter counts to a threshold value. And foreground thread ST signals until it kills itself. See Figure-1.

(b) Plot of the scope window running the blocking/sleeping/killing/priority system

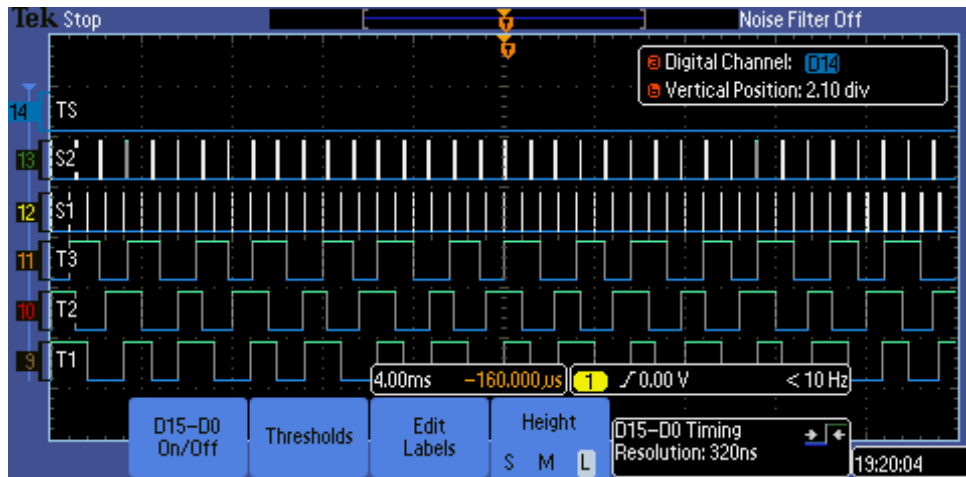
Figure-2 shows the scope picture of running a priority scheduler on the Producer-Consumer main. The labels show the name of the threads. Among those, DAS is a periodic background thread, all the other threads are foreground threads. Button is an aperiodic thread caused by pressing the switch. The Interpreter thread is a foreground thread that waits on user input, therefore also aperiodic.

(c) Table like Table 3.1 each showing performance measurements versus sizes of the Fifo and timeslices.

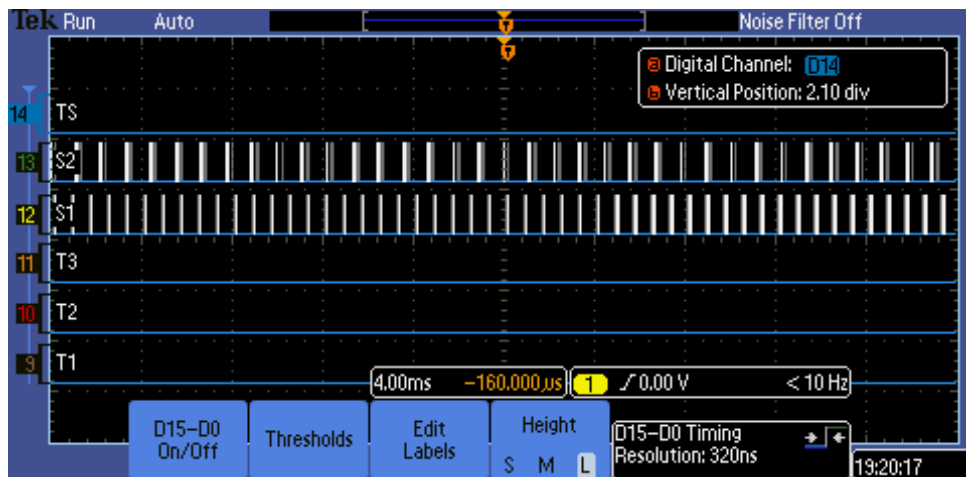
See Table-1.



(a) Initial execution

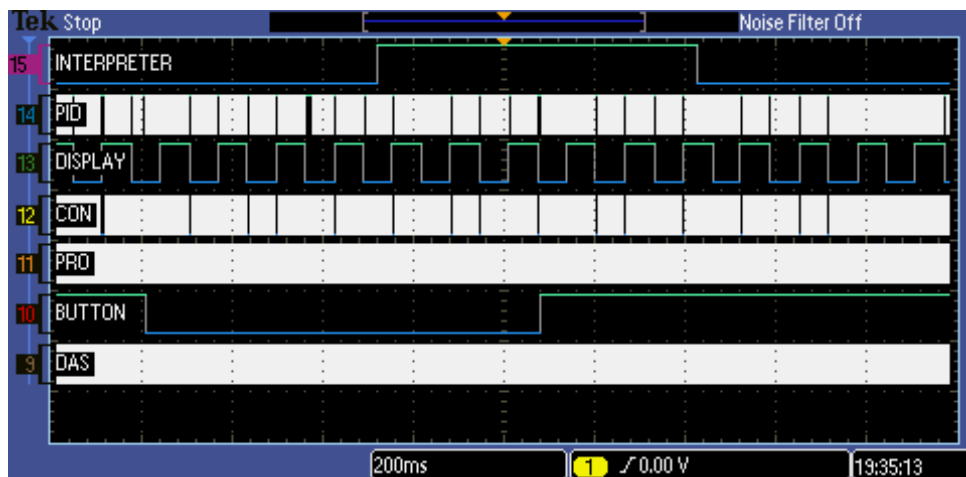


(b) After ST kills itself

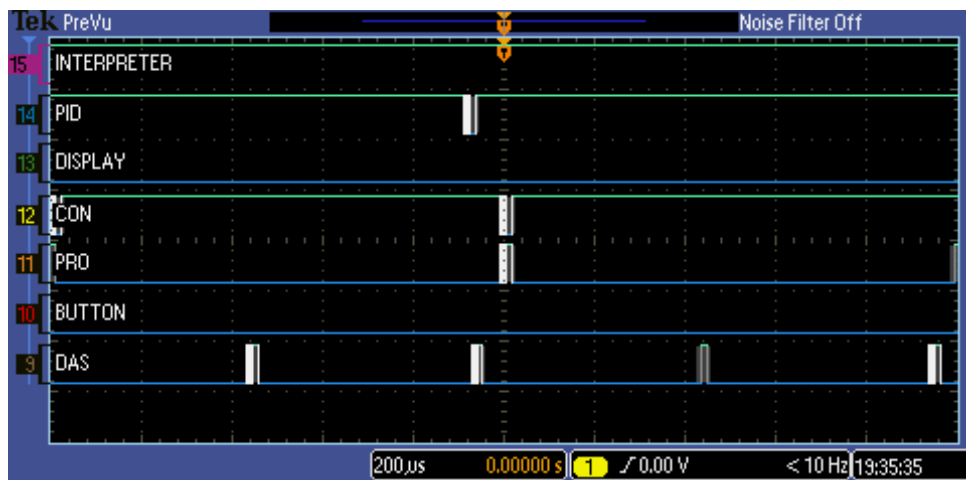


(c) After the signalling stops

Figure 1: Scope showing Round-robin scheduler running



(a) Initial execution



(b) Zoomed-in picture

Figure 2: Scope showing Priority scheduler running

Performance Measurement				
Spinlock/Non-Cooperative				
FIFOSize	TIMESLICE(ms)	DataLost	Jitter(μs)	PIDWork
4	2	1895	9	1842
8	2	0	9	1743
32	2	0	9	1743
32	1	0	10	1745
32	10	0	9	1778
Spinlock/Cooperative				
FIFOSize	TIMESLICE(ms)	DataLost	Jitter(μs)	PIDWork
4	2	1916	10	2656
8	2	0	9	2507
32	2	0	10	2507
32	1	0	10	2507
32	10	0	9	2510
Block/Priority				
FIFOSize	TIMESLICE(ms)	DataLost	Jitter(μs)	PIDWork
4	2	1473	1	5388
8	2	986	1	5316
32	2	0	1	5153
32	1	0	1	5153
32	10	0	1	5153

Table 1. Performance Measurements

4 Analysis and Discussion

How would your implementation of OS_AddPeriodicThread be different if there were 10 background threads? (Preparation 1)

Instead of using different timers for each background thread, use one background thread that runs all the other threads. For each background thread, there should be a data structure that holds a pointer to the background function, the period of the thread, and the current time remaining to next execution. (In units of the period of the timer interrupt) The interrupt will pass through each thread, decrements their counter, execute if the counter is 0, and then sets their timer to the reload value again. This can work for any number of periodic tasks.

How would your implementation of blocking semaphores be different if there were 100 foreground threads? (Preparation 4)

Since we are using linking and unlinking, our implementation is functional if there were 100 foreground threads. The scheduler is as fast as a scheduler without blocking semaphores. The only limitation will be the heap space in our dynamic memory allocation module.

How would your implementation of the priority scheduler be different if there were 100 foreground threads? (Preparation 5)

Since we're using different linked list for every priority, our implementation is compatible with 100 foreground threads. However, if the number of priority levels increases, a better method of searching should be implemented rather than a simple linear search to find the available highest priority.

What happens to your OS if all the threads are blocked? If your OS would crash, describe exactly what the OS does? What happens to your OS if all the threads are sleeping? If your OS would crash, describe exactly what the OS does? If you answered crash to either or both, explain how you might change your OS to prevent the crash.

If last active thread blocks, the scheduler assigns the last thread to be blocked as the next thread to be run, and therefore the OS continues running the foreground thread that was supposed to be blocked. This is an unexpected behaviour and will probably cause a crash in the system.

If last active thread sleeps, the scheduler assigns the last thread to be slept as the next thread to be run, and therefore the OS continues running the foreground threads that were supposed to be asleep. This can cause undefined behaviour and eventually crash the system.

Currently, our OS_Init always adds a dummy thread at the lowest priority that will always remain active which prevents crashes. a more robust method will be to make the microcontroller sleeps in case of no active foreground thread and waits for interrupts to occur.

What happens to your OS if one of the foreground threads returns? e.g., what if you added this foreground

```
void BadThread(void){ int i;
    for(i=0; i<100; i++){};
    return;
}
```

What should your OS have done in this case? Do not implement it, rather, with one sentence, say what the OS should have done? Hint: I asked this question on an exam.

OS should have detected that a foreground thread has tried to return and therefore killed the faulty thread.

What are the advantages of spinlock semaphores over blocking semaphores? What are the advantages of blocking semaphores over spinlock?

Spinlock semaphores are easier to implement and do not slow down the process of choosing next thread in the scheduler. However, blocking semaphores can be a lot faster, more CPU efficient, and can provide the means to implement bounded waiting.

Consider the case where thread T_1 interrupts thread T_2 , and we are experimentally verifying the system operates without critical sections. Let n be the number of times T_1 interrupts T_2 . Let m be the total number of interruptible locations within T_2 . Assume the time during which T_1 triggers is random with respect to the place (between which two instructions of T_2) it gets interrupted. In other words, there are m equally-likely places within T_2 for the T_1 interrupt to occur. What is the probability after n interrupts that a particular place in T_2 was never selected? Furthermore, what is the probability that all locations were interrupted at least once?

Probability that a line was never selected is $(m - 1/m)^n$
 Probability that all locations were interrupted once

$$p = \begin{cases} 0 & \text{if } n < m \\ (1 - (\frac{m-1}{m})^n)^m & \text{if } n \geq m \end{cases}$$