

INF-2201: OP fundamentals
Project 3 preemptive scheduling
Lise Ekhorn Johansen
UiT id: ljo177@uit.no
Github username: Lise-johansen
GitHub classroom: project-3-Lise-johansen
13.03.2023

1. Introduction

This project aims to transform the non-preemptive kernel into a preemptive kernel, which involves switching the system call mechanism from a simple function call to an interrupt-based one. Additionally, synchronization primitives such as semaphores, barriers, and conditional variables will also be implemented. Additionally, an unfair philosopher dining problem has been presented, and the goal is to make it fairer.

2. Design

2.1 Interrupt handler

First, enter the critical section, and disable all other interrupts. It then sends an end-of-interrupt signal (EOI) to the PIC to clear the interrupt. Then a context switch is performed for the thread/process to run. Finally, the critical section state is left to enable other interrupts.

2.2 Synchronization primitive mechanic

There are four synchronization mechanics, lock, condition variable, semaphore, and barrier. Each mechanism is designed using structures and is responsible for managing resources or coordinating the execution of multiple threads.

2.2.1 Design of locks

Only one thread holds the lock at a time, while other threads are blocked from accessing.

2.2.2 Design of condition variables

A waiting queue is associated with the conditional variable and threads are blocked and unblocked in the queue based on the state of the condition.

2.2.3 Design of semaphore

Semaphore is designed as a counter that keeps track of shared resources by counting how many threads are currently accessing or are done with a shared resource.

The semaphore counter will increase when a thread is done using a shared resource, and the counter will decrease when a thread wants to access a shared resource.

2.2.3 Design of barrier

The barrier can be compared to a bouncer waiting for all threads to arrive before unblocking them. If not all threads have arrived, the arriving threads are blocked until all threads have arrived.

3. Implementation

3.1 Interrupt handler

After the EOI, switch the kernel stack, call on the scheduler entry, and switch back to the user stack. And the fake interrupt handler is implemented as a system call.

3.2 Synchronization primitives

All the following implementations follow this pattern: enter critically, do the work, and leave critical.

3.2.1 Semaphore

A semaphore has to support two behaviors. The pseudocode 1 and 2 show the implementation of the two behaviors.

In pseudocode 1 the value of the semaphore is incremented. If the value is positive the threads in the waiting queue are unblocked.

To signal that a thread what to access a shared resource, the value is decremented by one. If the value becomes negative the currently running thread is blocked until something wakes the thread up again (i.e. value becomes positive).

```
1 void semaphore_up()
2 {
3     value++;
4
5     if (value < 0)
6     {
7         unblock(waiting);
8     }
9 }
10
```

Pseudocode 1: Demonstrates the implementation of a counting semaphore, which this initialized through an "init" function that set all starting values. This code represents when a thread is done with the shared resource.

```
11 void semaphore_down()
12 {
13     value--;
14
15     if (value < 0)
16     {
17         block(waiting);
18     }
19 }
20
```

Pseudocode 2: This code checks if there are any available shared resources, and if not, blocks the currently running thread and waits for the running threads to release the resources they are using (i.e. done running). When a resource is available, the blocked thread is unblocked and continues execution.

3.2.2 Condition variables

First, unblock the lock and block the thread, and when the thread is unblocked again the lock can be reacquired, which is the implementation of the condition variable.

Pseudocode 3 is the implementation of the remaining behaviors needed for condition variables. Both unblock thread(s) from the waiting queue when it is not empty. But one uses a while to loop through and unblock all threads, and the other uses an if statement to unblock only one thread from the waiting queue.

```
3 void condition_something()
4 {
5     when(waiting queue is not NULL)
6     {
7         unblock(waiting);
8     }
9 }
```

Pseudocode 3: Collected logic for implementation. All the condition variables function is made through an init function, that initializes a waiting queue to empty.

3.2.3 Barrier

Pseudocode 4 illustrates the implementation of the barrier function. Every time a thread reaches the barrier, it increments a counter by one. When the counter equals the total number of threads required to reach the barrier, all the threads in the waiting queue are unblocked. If the counter is less than the total number of threads required to reach the barrier, the thread is blocked by adding it to the waiting queue. Important to initialize all initial values to empty or zero. Also, reset the counter to zero if the counter equals the required number of threads.

```
2
3 void barrier()
4 {
5     n_equal++;
6
7     if(n_equal == n_count)
8     {
9         n_equal = 0;
10
11         while(waiting is not empty)
12         {
13             unblock(waiting);
14         }
15     }
16     else
17     {
18         block(waiting);
19     }
20 }
```

Pseudocode 4: Illustrate the implementation of a barrier, all initializations happen in an init function.

4. Experiments and Result

4.1 Setup of experiments: fair

Test to see the fairness of the philosopher problem is done by looking at light (representing the philosophers) and see if it has a fair distribution of activity.

4.2 Result: fair

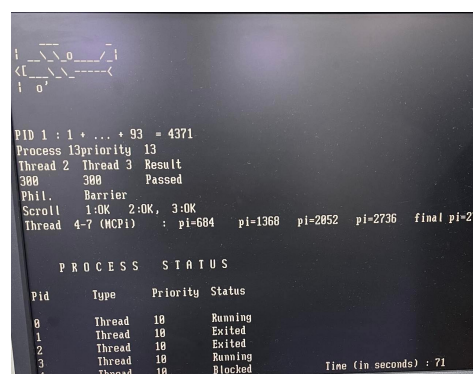
A philosopher's activities look to be evenly distributed in terms of frequency and duration. More on a fair result in the discussion.

4.3 Test for the correctness of the OS

To evaluate correctness, a single test is run by booting the OS image on a PC without an existing OS. Success is confirmed if the output matches the expected output and all tests pass.

4.4 result OS

Picture 1 shows the output when the OS image is run. Where the result matches the expected output.



```

  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000

PID 1 : 1 + ... + 93 = 4371
Process 13 priority 13
Thread 2 Thread 3 Result
300 300 Passed
Phil. Barrier
Scroll 1:0K 2:0K 3:0K
Thread 4-7 (MCPI) : pi=604 pi=1368 pi=2852 pi=2736 final pi=2736

PROCESS STATUS
Pid Type Priority Status
0 Thread 10 Running
1 Thread 10 Exited
2 Thread 10 Exited
3 Thread 10 Running
4 Thread 10 Blocked
Time (in seconds) : 71
```

Picture 1: Shows the results of the OS test. Test for lock (pi), condition variable (thread 1 and 2), and barrier have passed. The philosopher problem has successfully utilized the semaphores.

5. Discussion

5.1 Implementation: Lock

The lock used in this project is from project 2, with one modification. The lock acquire function was implemented without entering and leaving a critical section, making it

possible to call it within another function that enters and leaves a critical section. Making the lock acquire function mutual exclusion.

5.1.2 Semaphore

One assumption regarding the implementation of the semaphores is that the threads are a non-busy wait.

5.1.3 Philosophers dining problem

Initialize a lock and a philosopher as condition variables. From the perspective of philosopher 1, a condition is set that only allows philosopher 1 to eat if it is not bigger than the other philosophers. If the condition is not met, the philosopher is put in the waiting queue, releases the lock, and signals to the others that the lock is released. If the condition is met, philosopher 1 gets to eat when, and when it is finished it releases the lock and signals to the others that the lock is released.

5.2 Philosophers' dining problem: fair

Fairness in this project means that every philosopher eats the same amount.

Fairness is ensured through the use of a condition variable that places the philosophers in the waiting queue when it has eaten more than the others. Fairness is also ensured through a set eating time. This set eating time guarantees that no philosopher eats for a longer duration than the others.

Starvation is prevented by limiting how much one philosopher can eat. When one philosopher has eaten more than the other, it releases the lock (forks/table) and exists the critical section allowing the other the acquire the lock.

Deadlocks are preventers through the use of a lock, ensuring that only one philosopher can access the critical section and eat at a time.

Can improve the implementation by dividing a number by the time an iteration loop takes, instead of using a magical number for eating time.

5.3 Completion of project

Based on the experiment's results, a working preemptive kernel was made. And an implementation of a philosopher dining problem was made fairer. Can say it is fairer on the foundation of both a theoretical reason, as the use of conditional and locks guarantee that no philosopher can eat more or longer than the others. And an observational reason, as there are no discernible biases towards any of the philosophers when the code is running.

6. Conclusion

A working preemptive kernel was implemented, that passed all tests. Additionally, a fairer implementation of the philosopher dining problem using condition variables was implemented.

Bibliography

[1] educative.com, "What are conditional Variables in OS", Ahmad. Amazon, Read: 24.02.2023

URL: [What are Conditional Variables in OS? \(educative.io\)](https://educative.io/what-are-conditional-variables-in-os)

[2] Scaler.com, "Semaphores in OS", Mature. Taneesha, Published: 11.02.2022, Read: 24.03.2023

URL: [Semaphore in OS \(Operating System\) | Scaler Topics](https://Scaler.com/topics/semaphore-in-os)