

Gebze Technical University

Department of Computer Engineering

CSE312 Spring 2024

Operating Systems

Homework #01

Mehmet Mahir Kayadelen, Student no. 210104004252

At first I implemented the 4 system calls.

-Fork

-Exit

-Waitpid

-Execve

1-Fork

```
uint32_t TaskManager::Fork(CPUState* cpustate) {
    Task* cur_Task = tasks[currentTask];

    Task* child = new Task(gdt, 0, this);

    // copy parent stack to the child stack
    for(int i = 0; i < sizeof(cur_Task->stack); i++){
        child->stack[i] = cur_Task->stack[i];
    }

    // calculating offset of current esp(cpustate)
    uint32_t parentEspRelativeToStackStart = (uint32_t)cpustate - (uint32_t)cur_Task->stack;

    // calculating child's current esp(cpustate) with calculating parent esp offset
    child->cpustate = (CPUState *) (child->stack + parentEspRelativeToStackStart);

    uint32_t parentEbpRelativeToStackStart = (uint32_t)cpustate->ebp - (uint32_t)cur_Task->stack ;
    child->cpustate->ebp = (uint32_t) (child->stack + parentEbpRelativeToStackStart);

    child->pPid = cur_Task->pId;
    child->cpustate->ecx = 0;

    AddTask(child);

    printf("\n");

    return child->pId;
}
```

The current task being executed is retrieved from the tasks array using currentTask as the index. This current task is stored in the pointer cur_Task.

A new task object (child) is created. This task represents the forked (child) task.

The constructor for the Task class is called with gdt, 0, and this as arguments. gdt represents the Global Descriptor Table, and this is a pointer to the TaskManager instance.

The entire stack of the parent task (cur_Task) is copied to the child task's stack.

Loop iterates through each byte of the parent's stack and copies it to the corresponding location in the child's stack.

The offset of the parent's stack pointer (esp) relative to the start of the parent's stack is calculated. This is done by subtracting the base address of the parent's stack from the address of the cpustate (which includes the esp).

The child's stack pointer (esp) is calculated by adding the previously computed offset (parentEspRelativeToStackStart) to the base address of the child's stack.

This effectively positions the child's esp at the same relative position within the child's stack as the parent's esp within the parent's stack.

Similar to the esp, the offset of the parent's base pointer (ebp) relative to the start of the parent's stack is calculated

The child's base pointer (ebp) is then set to the same relative position within the child's stack as the parent's ebp within the parent's stack.

The child's parent process ID (pPid) is set to the current task's process ID (pId).

The ecx register in the child's CPU state is set to 0. This is done to differentiate between the parent and child processes after a fork, as the child process returns 0 from the fork call.

```
uint8_t myos::fork(){
    uint8_t childPid;
    asm("int $0x80" : "=c" (childPid) : "a" (SYSCALLS::FORK));
    return childPid;
}
```

The fork function uses inline assembly to invoke a system call for forking a new process. It uses the int \$0x80 instruction to make the system call, passes the syscall number for fork via the eax register, and stores the result (child process ID) in the ecx register. The child process ID is then returned as an 8-bit unsigned integer.

2-Exit

```
bool TaskManager::Exit(){
    tasks[currentTask]->taskState = FINISHED;

    return true;
}
```

This system call basically changes current task's taskState into FINISHED and returns.

3-Waitpid

```
bool TaskManager::Waitpid(uint32_t esp){  
  
    CPUState *cpustate = (CPUState *) esp;  
    uint32_t pid = cpustate->ebx;  
  
    if(tasks[currentTask]->pId == pid || pid == 0){ // for self waiting  
        return false;  
    }  
  
    int index = -1; // default value of non found  
    for(int i =0 ; i < numTasks;i++){ // iterates through tasks and finds if waiting Process exist  
        if(tasks[i]->pId == pid){  
            index = i;  
            break;  
        }  
    }  
  
    if(index == -1) // process doesnt exist  
        return false;  
  
    if(numTasks<= index || tasks[index]->taskState == FINISHED){  
        printf("returned \n");  
        return false;  
    }  
  
    tasks[currentTask]->cpustate=cpustate;  
    tasks[currentTask]->waitPid = pid;  
    tasks[currentTask]->taskState = WAITING;  
    return true;  
}
```

It takes a uint32_t argument esp, which represents the stack pointer of the CPU state.

The function returns a bool indicating whether the wait operation was successful or not.

The esp argument is cast to a CPUState pointer, cpustate. This allows access to the CPU state (registers and other information) at the time of the call.

Checks if the current task's PID is the same as the PID to wait for or if the PID is 0. If either condition is true, the function returns false, as a task cannot wait for itself or for a PID of 0

A variable index is initialized to -1 to indicate that the target task has not been found.

The loop iterates over all tasks to find a task with the specified PID.

If a task with the specified PID is found, index is set to the task's index and the loop is exited.

If index remains -1, it means no task with the specified PID was found, so the function returns false.

The CPU state of the current task is updated to the cpustate obtained earlier.

The current task's waitPid is set to the PID it is waiting for.

The current task's state is set to WAITING.

```
void myos::waitpid(common::uint8_t wPid){
    asm("int $0x80" : : "a" (SYSCALLS::WAITPID), "b" (wPid));
}
```

The waitpid function uses inline assembly to invoke a system call for waiting on a specific process. It sets up the necessary registers (eax for the syscall number and ebx for the PID) and then triggers interrupt 0x80 to make the system call. This function does not return a value.

4-Execve

```
int TaskManager::Execve(void (*entrypoint)()){
    tasks[currentTask]->cpustate = (CPUState*) (tasks[currentTask]->stack + 4096 - sizeof(CPUState));
    tasks[currentTask]->cpustate->eip = (uint32_t) entrypoint; // move instruction pointer to the function
    tasks[currentTask]->cpustate->cs = gdt->CodeSegmentSelector(); // for kernel
    tasks[currentTask]->cpustate->ebp = 0; // for make sure there will be no problem
    tasks[currentTask]->cpustate->eflags = 0x202;

    return 0;
}
```

It takes a pointer to a function (entrypoint) as an argument. This pointer represents the new entry point (the starting address of the function to execute) for the current task.

The CPU state (cpustate) of the current task is set to point to a location within the task's stack. This location is calculated by taking the base address of the task's stack, adding 4096 (indicating the size of the stack), and then subtracting the size of a CPUState structure. This sets up the cpustate at the top of the task's stack.

The instruction pointer (eip) in the cpustate is set to the address of the entrypoint function.

This means when the task is scheduled to run, it will start executing from the entrypoint function.

The code segment register (cs) in the cpustate is set to the code segment selector obtained from the Global Descriptor Table (gdt). This ensures that the task will use the correct code segment for execution, set up for kernel mode.

The base pointer register (ebp) in the cpustate is set to 0.

This is done to ensure there are no issues with the base pointer, as it may have been used previously by the task.

```
int myos::execve(void entrypoint()){
    int result;

    asm("int $0x80" : "=c" (result) : "a" (SYSCALLS::EXECVE), "b" ((uint32_t)entrypoint));

    return result;
}
```

The execve function uses inline assembly to invoke a system call for executing a new program. It sets up the necessary registers (eax for the syscall number and ebx for the entry point address) and triggers interrupt 0x80 to make the system call. The result of the system call is stored in the result variable and returned by the function. This function allows the current process to start executing a new function specified by the entrypoint argument.

I made improvements and configurations in the scheduler function, here is the full schedule function.

```
99 CPUState* TaskManager::Schedule(CPUState* cpustate)
100 {
101     if(numTasks <= 0)
102         return cpustate;
103
104     // idleprocess
105
106     static int idleTime = 0;
107
108     if(idleTask->taskState == RUNNING)
109     {
110         idleTask->cpustate = cpustate;
111     }
112     else if(idleTask->taskState == READY)
113     {
114         idleTask->taskState = RUNNING;
115         if(currentTask >= 0)
116             tasks[currentTask]->cpustate = cpustate;
117     }
118
119     if(++idleTime % IDLE != 0)
120     {
121         return idleTask->cpustate;
122     }
123     else
124     {
125         idleTime = 0;
126         idleTask->taskState = READY;
127     }
128
129     // idleprocess
130
131
132     // idleprocess
133
134     int start = currentTask == -1 ? 0 : currentTask;
135
136     do{
137         if(++currentTask >= numTasks)
138             currentTask %= numTasks;
139         if(tasks[currentTask]->taskState == WAITING )
140         {
141             if(tasks[tasks[currentTask]->waitPid -1]->taskState == FINISHED)
142                 tasks[currentTask]->taskState = READY;
143         }
144     }
145     while(tasks[currentTask]->taskState != READY);
146
147     if(tasks[start]->taskState == RUNNING)
148         tasks[start]->taskState = READY;
149
150     tasks[currentTask]->taskState = RUNNING;
151     printTasks();
152     return tasks[currentTask]->cpustate;
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
```

idleprocess is a void function without a function. I used it to slow down the ProcessTable while printing to increase readability.

The variable start is set to 0 if currentTask is -1 (no current task); otherwise, it is set to currentTask

The currentTask index is incremented. If it exceeds the number of tasks, it wraps around to 0 using the modulo operation.

If the task state is WAITING, it checks if the task it is waiting for has finished. If so, it sets the current task's state to READY.

This loop continues until a task with the READY state is found.

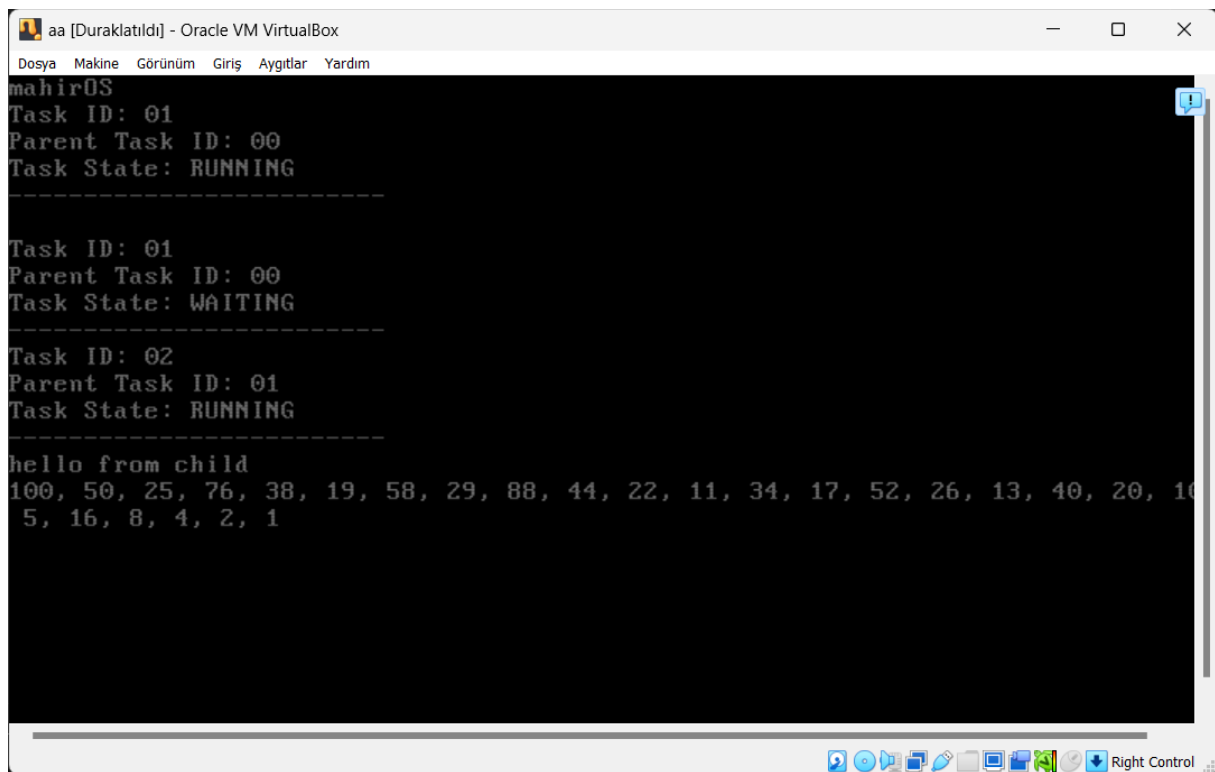
If the task that was initially running (start) is still in the RUNNING state, its state is changed to READY.

The newly selected task's state is set to RUNNING.

The function `printTasks()` is called, to print the status of all tasks in each Schedule.

The function returns the CPU state of the newly selected task.

Test:



The screenshot shows a terminal window titled "aa [Duraklatıldı] - Oracle VM VirtualBox". The terminal output is as follows:

```
mahirOS
Task ID: 01
Parent Task ID: 00
Task State: RUNNING
-----
Task ID: 01
Parent Task ID: 00
Task State: WAITING
-----
Task ID: 02
Parent Task ID: 01
Task State: RUNNING
-----
hello from child
100, 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10
5, 16, 8, 4, 2, 1
```

The terminal window has a menu bar with "Dosya", "Makine", "Görünüm", "Giriş", "Aygıtlar", and "Yardım". The bottom of the window shows a taskbar with various icons and a "Right Control" button.

```
aa [Duraklatıldı] - Oracle VM VirtualBox
Dosya Makine Görünüm Giriş Aygıtlar Yardım
Its parent time!

Task ID: 01
Parent Task ID: 00
Task State: WAITING
-----
Task ID: 02
Parent Task ID: 01
Task State: FINISHED
-----
Task ID: 03
Parent Task ID: 01
Task State: RUNNING
-----
hello from child
100, 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10
5, 16, 8, 4, 2, 1
```

```
aa [Duraklatıldı] - Oracle VM VirtualBox
Dosya Makine Görünüm Giriş Aygıtlar Yardım
-----
Task ID: 03
Parent Task ID: 01
Task State: FINISHED
-----
Task ID: 04
Parent Task ID: 01
Task State: FINISHED
-----
Task ID: 05
Parent Task ID: 01
Task State: RUNNING
-----
hello from child
392146832
```



```
aa [Duraklatıldı] - Oracle VM VirtualBox
Dosya Makine Görünüm Giriş Aygıtlar Yardım
Task State: FINISHED
-----
Task ID: 05
Parent Task ID: 01
Task State: FINISHED
-----
Task ID: 06
Parent Task ID: 01
Task State: RUNNING
-----
hello from child
392146832
```

```
aa [Duraklatıldı] - Oracle VM VirtualBox
Dosya Makine Görünüm Giriş Aygıtlar Yardım
Task State: FINISHED
-----
Task ID: 05
Parent Task ID: 01
Task State: FINISHED
-----
Task ID: 06
Parent Task ID: 01
Task State: FINISHED
-----
Its parent time!
```

My init function forks a total of 6 times. In the first 3 forks, child processes run collatz, and in the next 3 forks, child processes run the long program. The parent process waits for the child processes to finish and then continues. I print the Process Table in each context switch operation. After the 3rd context switch, the output of function is not visible well because the Process Table output expands too much. I used 6 forks and last child runs program with execve system call.

Here is my test task(init)

The screenshot shows the Visual Studio Code editor interface with the following details:

- Explorer Panel:** Displays the file structure of the project. The 'src' directory is expanded, showing files like `kernel.cpp`, `multitasking.cpp`, `syscalls.cpp`, and `loader.s`. The `kernel.cpp` file is selected.
- Editor Area:** Contains two open files, both named `kernel.cpp`.
 - Left Editor (src > kernel.cpp > init()):** Shows the `init()` function starting at line 249. It defines a variable `a`, forks a child process, and prints "hello from child\n".
 - Right Editor (src > kernel.cpp > init()):** Shows the `init()` function starting at line 250. It defines a variable `d`, forks a child process, and prints "hello from child\n".
- Terminal Panel:** Located at the bottom, it shows the output of the program, including "hello from child\n" and "Its parent time\n".
- StatusBar:** At the bottom right, it indicates the current position is line 291, column 31, with 4 spaces, UTF-8 encoding, LF line endings, and C++ language.