CS2106

Tutorial04

Multilevel Feedback Queue (MLFQ)

Basic Rules

- If Priority(A) > Priority(B), A runs
- If Priority(A) == Priority(B), A and B runs in RR

Priority Setting/Changing rules:

- New job Highest priority
- If a job fully utilized its time slice priority reduced
- If a job give up / blocks before it finishes the time slice priority retained

Q1. MLFQ

Consider the standard 3 levels MLFQ scheduling algorithm with the following parameters:

Time quantum for all priority levels is 2 time units (TUs).

Interval between timer interrupt is 1 TU.

1(a)

Interval between timer interrupt is 1 TU. Time quantum is 2 time units (TUs).

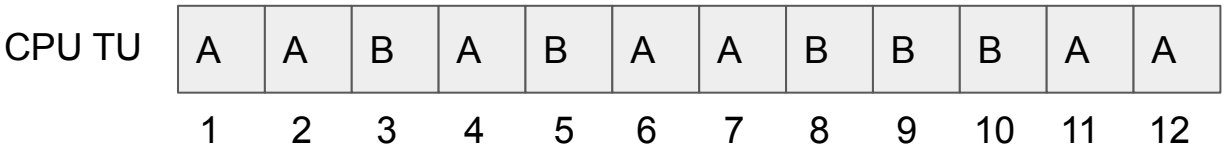
Task A
Behavior: **CPU 3TUs, *I/O 1TU*, CPU 4TUs**

Task B
Behavior: **CPU 1TU, *I/O 1TU*, CPU 1TU, *I/O 2TU*, CPU 3TU**

1(a)

Task A
Behavior: **CPU 3TUs, *I/O 1TU*, CPU 4TUs**

Task B
Behavior: **CPU 1TU, *I/O 1TU*, CPU 1TU, *I/O 2TU*, CPU 3TU**



Q2 MLFQ Discussion

As discussed in the lecture, the simple MLFQ has a few shortcomings. Describe the scheduling behavior for the following two cases.

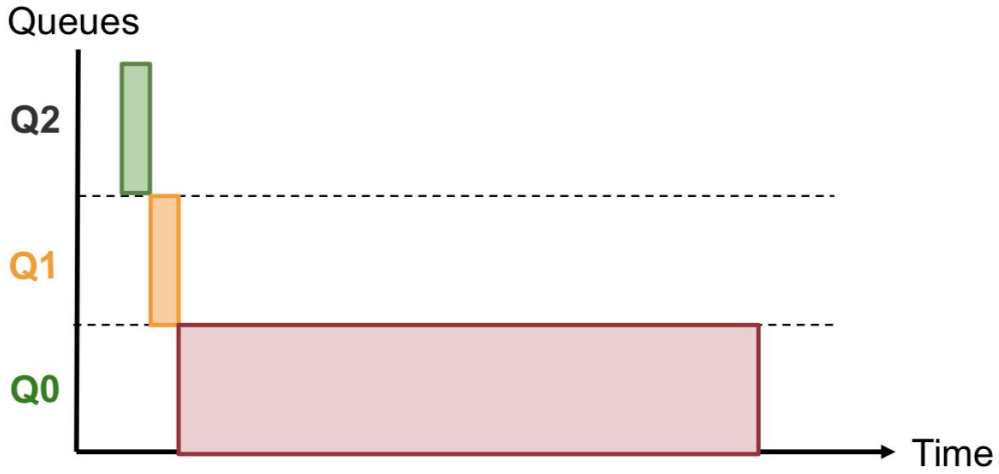
a. (Change of heart) A process with a lengthy CPU-intensive phase followed by I/O-intensive phase.

b. (Gaming the system) A process repeatedly gives up CPU just before the time quantum lapses.

2(a)

The process can sink to the lowest priority during the CPU intensive phase.

With the low priority, the process may not receive CPU time in a timely fashion during the I/O phase which degrades the responsiveness.



2(b)

If a process give up / blocks before the time quantum lapses, it will retain its priority.

Since all process enter the system with the highest priority, a process can keep its high priority indefinitely by using this trick and receive **disproportionately more** CPU time than other processes.

2(i, ii)

The following are two simple tweaks. For each of the rules, identify which case (a or b above) it is designed to solve, then briefly describe the new scheduling behavior.

I. (Rule – Accounting matters) The CPU usage of a process is now accumulated across time quanta. Once the CPU usage exceeds a single time quantum, the priority of the task will be decremented.

II. (Rule – Timely boost) All processes in the system will be moved to the highest priority level periodically.

2(i)

This tweak is to fix case (b).

The trick in (b) works because the scheduler is “memory-less”, i.e. the CPU usage is counted from fresh every time a process receives a time quantum.

If the CPU usage is accumulated, then a CPU intensive process will still **overshoot the allowed time quantum** and get a demotion in priority. This will prevent the process from hogging the CPU.

2(ii)

This tweak is for case (a).

By periodically boosting the priority of all processes (essentially treat all process as “new” and hence have highest priority), a process with different behavior phases may get a chance to be treated correctly even after it has sank to the lowest priority.

Q3. Shared Memory

Compile and run the code in shm.c.

Try running the program multiple times with increasingly higher values for n and nChild (n = 10000 and nChild = 100 should do the trick). Explain the results.

Q3 (Good Behaviour)

Time	Value of X	P1	P2
1	345	`Load X → Reg1`	
2	345	`Add 1 to Reg 1`	
3	1345	`Store Reg1 → X`	
4	1345		`Load X → Reg1'`
5	1345		`Add 1to Reg1`
6	**2345**		`Store Reg1' → X`

Q3 (Bad Behaviour)

Time	Value of X	P1	P2
1	345	`Load X → Reg1`	
2	345	`Add 1 to Reg1`	
3	345		`Load X → Reg1'`
4	345		`Add 1 to Reg1'`
5	1345	`Store Reg1 → X`	
6	**1345**		`Store Reg1' → X`

Q3

When incrementing the value, one process must 1) read the value from the memory, 2) modify it, and 3) write the value back to the memory. If two or more processes have their executions interleaved, it is very likely that one process will overwrite the value written by another process. In this case, it is impossible to have a deterministic value.

The reason why the number of increments and/or the number of processes must be large enough is to ensure that the processes do run at the same time in the system. Note that it takes some time to setup the recently created process, and if this time exceeds the time required for the increment, then the processes won't try to simultaneously read and write at the shared memory.

Q4. Shared Memory

Modify the code in shm.c such that the output is deterministic regardless of how large n and nChild are. We want the processes to take turns when modifying the result value that resides in the shared memory.

We will add another field in the shared memory, called the order value, that specifies which process' turn it is to increment the shared result. If the order value is 0, then the parent should increase the shared result, if the order value is 1, then the first child should increase it, if the order value is 2, then the second child and so on. Each process has an associated pOrder and checks whether the value order is equal to its pOrder; if it is, then it proceeds to increment the shared result. Otherwise, it waits until the order value is equal to its pOrder.

Q4

```
int main(int argc, char *argv[]) {
    int i, shmid, n = 100, childPid, nChild = 1, pOrder = 0; int *shm;
    if (argc > 1) n = atoi(argv[1]);
    if (argc > 2) nChild = atoi(argv[2]);

    // create Share Memory Region
    shmid = shmget(IPC_PRIVATE, 2*sizeof(int), IPC_CREAT | 0600);
    if (shmid == -1) {
        printf("Cannot create shared memory!\n"); exit(1);
    } else printf("Shared Memory Id = %d\n", shmid);

    // attach the shared memory region to this process
    shm = (int*)shmat(shmid, NULL, 0);
    if (shm == (int*) -1) {
        printf("Cannot attach shared memory!\n"); exit(1);
    }
    ...
```

Q4

```
    shm[0] = 0; shm[1] = 0; // initialize shared memory to 0
    for (i = 0; i < nChild; i++) {
        childPid = fork();
        if (childPid == 0) {
            pOrder = i + 1; break;
        }
    }
    // only increment the shared value if it's the process' turn
    while (shm[0] != pOrder);
    for (i = 0; i < n; i ++) shm[1]++;
    shm[0] = pOrder + 1;
    if (childPid != 0) {
        for (i = 0; i < nChild; i++) wait(NULL);
        printf("The value in the shared memory is: %d\n", shm[1]);
        // detach and destroy
        shmdt((char*)shm); shmctl( shmid, IPC_RMID, 0);
    } return 0;
}
```

Q4

Why is shm faster than shm_protected? Why is running shm_protected with large values for nChild particularly slow?

Q4

To guarantee that two processes won't try to modify the shared memory location at the same time, we must serialize the access to it, i.e., make it impossible for two processes to access it at the same time. Thus, the execution time is longer.

The technique we use to prevent simultaneous access is called busy waiting. Each process is continuously checking whether its time to modify the variable has come. This requires the process to run on the CPU and consume CPU cycles.

At the moment when it's Child X's time to increment the value, there are nChild – X processes doing busy waiting – each of these processes will get scheduled to run on the CPU but their execution will not make any progress. Because of this, your program will take a long time to complete for large values of nChild.

Q5. Thread vs. Process

For each of the following scenarios, discuss whether multithread or multiprocess is the best implementation model.

If you think that both models have merits for a particular scenario, briefly describe additional criteria you would use to choose between the models.

- Implementing a command line shell.
- Implementing the “tabbed browsing” in a web browser, i.e., each tab visits an independent webpage.
- Implementing a complex multi-player game with dynamic environments and sophisticated AI.

Q5

General key points:

Memory: How much memory is needed and how much of the memory is shared? Threads share the same memory space. So, if heavy memory space usage is needed (imagine each task use up almost the entire memory space), then multi process has to be used since each process has independent memory space. On the other hand, if the tasks share large amount of data, then process is more expensive.

Q5

Overhead: Since thread creation is “cheap”, we need to justify the overhead required by spawning a child process.

Protection: If the child process can potentially hang, deface the memory, then process model provides a safe “sandbox” for each task due to the independent memory space.

Hardware platform: Whether the processor is capable to exploit multiple threads / processes. Modern CPUs (e.g. intel’s hyperthreading processors) explicitly support multiple threads execution. On the other hand, systems with multiple separate processors can execute multiple processes more efficiently.

Q5

Operating System: Some OSes have features that favor thread model over process model and vice versa. e.g. Linux supports process “cloning” (instead of using “fork”) which can reduce the overhead drastically; “Copy on write” mechanism reduces memory duplication overhead; “Process migration” mechanism allows a process to move from one computer to another, etc.

Q5

- Process model. Key points: Memory + Protection.
- Arguable. Thread model (used by older Firefox) has lower overhead (much of the webpage rendering code is shared between the tabs) and suitable for less memory intensive webpages. Process model (used by most browsers nowadays) provides protection and can support multiple resource heavy webpages.
- Thread model. Games can be time sensitive and the process model requires overheads that are not necessary in this context.