INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

COMPUTER SCIENCE AND ENGINEERING

Course: CS341 (Operating System), Model Solution Mid-Semester Exam

1. [System Architecture, Structure, Service and Design: 5 Lectures (=28%)] 14 Marks[3+4+3+4]

a) [**3 Marks**] How does OS hide the peculiarities of specific hardware from the users?

*Ans: OS **abstract the hardware interface using drive**r and provide a **hardware abstract layer**. **I/O subsystem** hides the peculiarities of specific hardware devices from the user. Only the **device driver** knows the peculiarities of the specific device to whom it is assigned.*

b) [**4 Marks**] What are the advantages of using micro-kernel approach in designing an OS?

*Ans: The main advantages are that new services do not need to modify the kernel and it's **easier to port between hardware**. Also, micro-kernels **provide more security because of less time in privileged mode**. **Reliable because it small code easy to test properly**. With microkernel, one can provide support for dynamic loading/unloading (inserting/removing) user or device modules at runtime.*

c) [**3 Marks**] What are the OS service functions that are helpful to users? What are the OS service functions that help in ensuring efficient operation of system itself?

*Ans: OS provide these service functions that are helpful to users: **GUI, Program execution, Error Detection, File System Manipulation, and Communication** between processes or computers. OS provide these service functions* that help in ensuring efficient operation of system itself*: **Resource Sharing and Allocation, Accounting and Protection and Security.***

d) [**4 Marks**] What are the differences between user functions, library functions, APIs and system calls? (With examples).

*Ans : API act as system call interface and it invokes the intended system call in OS kernel and returns status of the system call and any return values. The caller/user **need know nothing (need not to know)** about how the system call is implemented, but just needs to obey API and understand what OS will do as a result call. Most details of OS interface hidden from programmer by API. **Library uses API to implement the library functions. User function use library function to implement big software/function.***

*Examples of system call: **sys_read(), sys_write().***

*Examples of APIs : read**(File *fd, buff, size); write(File *fd, buff, size);***

*Examples of library function: **printf/scanf** function of C*

2. [Process Scheduling and Scheduling Algorithms: 8 Lectures (=44%)] 22 Marks [3+2+6+6+5]

a. [**3 marks**] What are the differences between short term scheduler and long term scheduler?

*Ans: Short-term scheduler (or CPU scheduler) **selects which process should be executed next and allocates CPU**. Short-term scheduler is **invoked frequently (milliseconds) $\Rightarrow$ (must be fast).** But Long-term scheduler (or job scheduler) **selects which processes should be brought into the ready queue (from HDD to memory)**. Long-term scheduler is invoked infrequently (seconds, minutes) $\Rightarrow$ (may be slow). The long-term scheduler controls the **degree of multiprogramming.***

b. [**2 Marks**] Given a system with ***n*** processes (arbitrary execution time, pre-emption is not allowed), how many possible ways can those processes be scheduled on a processor?

Ans: ***n!***

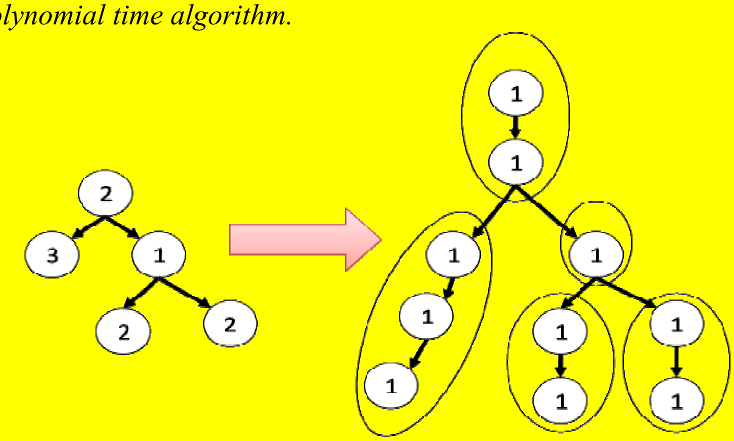
c. [**6 Marks**] $P_m|out\text{-}tree, p_j=1|C_{max}$ (non-preemptive scheduling ***n*** tasks (*with unit execution time of each task and tasks precedence is out-tree dependency*) on ***m*** homogenous processor system, C_{max}=makespan) can be solved in O(n) time by Hu's Highest Level First (or CF) Algorithm. If we relax $p_j=1$, then execution time of each task will be arbitrary.

Design an efficient approach to solve $P_m|out\text{-}tree|C_{max}$. Your approach may be heuristic or approximation or proper algorithm with time complexity, space complexity and approximation bound (if it is an approximation).

Ans: You may find many approaches to solve this problem. Let's discuss two basic approaches to solve this.

Approach 1

Assume each task can be spiltted into unit time subtasks with chain precedence and spitted subtasks of a task may be scheduled at different processor. The resultant transform task graph is out-tree, so the transformed graph can be scheduled on ***m*** *processor optimally using HLP/CP algorithm. As number of node in the transformed out tree is* $\sum t_i$ *and number of edge is* $(E+\sum t_i - n)$*, where* t_i *is execution time of* i_{th} *task. Time complexity of algorithm will be* $O(\sum t_i + E+\sum t_i - n)$ *which is not polynomial in term of n but polynomial in terms of* $\sum t_i$*. So it is Pseudo polynomial time algorithm.*



Approach 2

One can use the CP/HLF heuristic to solve by calculating critical/level value of node as maximum of sum task execution of path to leaf nodes. Create a list based on priority and execute a higher priority ready task on available processor. This approach will achieve an approximation ratio of 2- (1/m).

Proof*: At any point of time either (a task from CP will execute) or (no processor will be idle). Let L be the length of the CP path, m the number of processors, I the total idle time, C the makespan, and S the sum of all task weights*

$I \le (m-1)L$ *Says :Processors can be idle only when a red task is running*

$L \le C_{opt}$ *: The optimal makespan is longer than any path in the out-tree(valid for DAG also)*

$C_{opt} \ge S/m$ *: S/m is the makespan with zero idle time*

$m.C = I + S$ *: Total Time = Idle time + Busy time*

So $m.C \le (m-1).C_{opt} + m.C_{opt}) \implies C \le (2 - 1/m).C_{opt}$

d. [**6 Marks**] Grahm's list scheduling (non-preemptive scheduling of ***N*** independent tasks with arbitrary execution time on ***M*** identical processor) approach approximate $\mathbf{C_{max}}$ by a factor of 2. Design an efficient approach to scheduling the same task set on ***M*** uniform processors.

*Ans: Graham's list scheduling use non-premptive scheduling and create an arbitrary list to schedule the job on m-**identical processor** by selecting one job at a time and assigning to lowest loaded machine. This approach approximates C_{max} by factor of 2 and more precisely **2-1/m**. If we use **LPT (Longest Processing Time) rule** (priority to LPT job) then we can achieve an approximation ratio **3/2** and further upto **4/3-1/3m**.*

You may find many approaches to solve this problem. Let's discuss two basic approaches to solve this.

Approach1:

Sort the Jobs/Tasks in non-increasing order, Sort the processor based on speed ($s_1>s_2>\ldots>s_m$).

- *Take the longest job and assigned to the fastest processor*
- *Do while (there is a Job to schedule)*
    - *Take the next longest job and assigned to next fastest processor, if it is the slowest one then set next processor is the fasted one. {Processor are arranged in a Cycle: Next of slowest processor one (s_m) is the fastest processor (s_1) }*

Time Complexity of this approach is O(n), where n is number of Task. When we schedule a Job we consider only one processor (the next fastest), but this may not lead to a good solution. This approach doesn't quantify the speed and execution time of job.

Approximation Bound: This is approach is not based on proper LPT rule for uniform processors.

Approach2:

Sort the Jobs/Tasks in non-increasing order, Sort the processor based on speed ($s_1>s_2>\ldots>s_m$).

- *Do while (there is a Job to schedule)*
    - *Take the next longest job and assigned to processor, where expected finishing time of the Job will be minimum among all the processors. {In first step, longest job will go to fastest machine to reduce expected finishing time of the longest Job}. When we schedule a job, we consider all the processor.*

Time Complexity of this approach is O(n.m), where n is number of Task and m is number of processor. When we schedule a Job we consider all the processors. This approach doest quantify the speed and execution time of job at the time of scheduling.

Approximation bound:

LPT rule achieve [Sahani, Gonzalez, "Bound for LPT rule for Uniform Processor",SIAM J Comp. 1977] an approximation upper bound of 2 ($C_{max}/C_{maxopt}<2$) , but have lower bound of 3/2 ($C_{max}/C_{maxopt}<3/2-\varepsilon$). These bound can improved to (1.52, 1.584) [Dobson, "Scheduling Independent Task on Uniform Processor", SIAM J Comp. 1984].

*Details of [Sahani, Gonzalez] 2 approximation: (used f instead of C_{max}, f*is optimal schedule length)*

Consider an m-processor system with job set $S=(t_1\geq t_2\geq\ldots\geq t_n)$ and speed $s_1,s_2,\ldots,s_m$. If in the LPT schedule of S, the finish time f is determined by t_n, (i.e., if task n has the latest completion time), then $f/f^* \leq 1+(m-1)t_n/(Qf^*)$ where $Q=\sum s_i$. This above claim can be proved by the following analysis.

Let P_k determine the finishing time of LPT rule, where P_k is partition of tasks of S that got assigned to processor k. Let T_i be the sum of execution time of task got assigned to processor i prior to t_n's assignment. Then

$$T_1+T_2+\ldots+T_m=t_1+t_2+\ldots+t_{n-1}.$$

Since task n determine the finishing time of LPT, $f=T_k+t_n/s_k \geq f$ for $i\neq k$.

Hence $f.s_i-T_i\leq tn$ *and* $(f.\sum_{i\neq k} s_i - \sum_{i\neq k} T_i) \leq (m-1)t_n$.

Also $f\, s_k=T_k+t_n$

So $(f.\sum_{i\neq k} s_i - \sum_{i\neq k} T_i) \leq (m-1)t_n \Rightarrow f.\sum_{i\neq k} s_i \leq (m-1)t_n + \sum_{i\neq k} T_i \Rightarrow$ *Along with* $f\, s_k=T_k+t_n$

$\rightarrow \quad f.\sum_{i\neq k} s_i - f s_k \leq (m-1)t_n + \sum_{i\neq k} T_i + T_k + t_n$

$\rightarrow \quad f. .\sum s_i \leq mt_n + \sum T_i \quad \rightarrow f.Q \leq mt_n + \sum T_i \quad \rightarrow f \leq (Qmt_n + Q \sum T_i)$

Since $f^* \geq (\sum T_i / \sum s_i) = (\sum T_i / Q) \quad \rightarrow f^*Q \geq \sum T_i \quad \rightarrow 1/(f^*Q) \leq 1/(\sum T_i) \rightarrow 1/f^* \leq Q/\sum T_i$

$\rightarrow f/f^* \leq 1 + (m-1)t_n/(f^*Q)$

As $t_1 \geq t_2 \geq \ldots \geq t_n$ *and* $n > m$

$\rightarrow f/f^* \leq 1 + (m-1)t_n/(Q (\sum T_i / Q)) \leq 1 + (m-1)t_n/nt_n \leq 1 + (m-1)/n \leq 1 + (m-1)/(m+1) \leq 2m/(m+1)$

$$\rightarrow \mathbf{f/f^* \leq 2}$$

e. [**5 Marks**] Draw Gannt's chart (and calculate average flow time) to schedule the following tasks using FCFS, SJF, SRT, RR (q=2) and RR (q=1) on a processor.

Process	Arrival Time	Processing Time
A	0	3
B	1	6
C	4	4
D	6	2

Ans:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Flow Time is same as turn-around time for uniprocessor system. Flow time of a Job is difference between completion time and arrival time. So Average Flow Time = $(\sum(C_i - A_i))/N$

FCFS

A			*B*						*C*				*D*	

*Average Flow Time = ¼ * [(3-0) + (9-1) + (13-4) + (15-6)] =* ***7.25***

SJF (Shortest Job First) if we use pre-emption it will be SRT First

A			*B*						*D*		*C*			

At time 0: only one Job A, at time 3 only one job B, at time 9 two job C and D (D shorter than C), at time 11 only one Job C.

*Average Flow Time = ¼ * [(3-0) + (9-1) + (15-4) + (11-6)] =* ***6.75***

SRT (Shortest Remaining Time) First

A			*B*	*C*		*C or D*		*C or D*		*B*				

*Average Flow Time = ¼ * [(3-0) + (15-1) + (8-4) + (10-6)] =* ***6.25***

RR (Round Robbin with time quantum q =2)

A		*B*		*C*		*D*		*A*	*B*		*C*		*B*	

*Average Flow Time = ¼ * [(9-0) + (15-1) + (12-4) + (8-6)] =* ***8.25***

RR (Round Robbin with time quantum q =1)

A	*B*	*A*	*B*	*C*	*A*	*B*	*C*	*D*	*B*	*C*	*D*	*B*	*C*	*B*

*Average Flow Time = ¼ * [(6-0) + (15-1) + (14-4) + (12-6)] = 9.**00***

3. [Threading and Synchronization: 5 Lectures (=28%)] 14 Marks [3+3+4+4]

(a) [**3 Marks**] What are the differences between User Thread, Kernel Thread and Hardware Thread? How threading is beneficial in presence of non-blocking I/Os (async I/Os) and cache miss in processor.

*Ans: **User threads** - management done by user-level threads library, **Kernel threads** - Supported by the Kernel and **Hardware thread** supported by underlying hardware processing platform. User threads run on top of Kernel threads and Kernel threads run on top of Hardware threads using some mapping policy.*

User threads are much faster as compared to kernel thread as they don't need to run in kernel mode or secure mode. Kernel threads are safe and secure but in user threads user need to ensure safety by writing code properly.

OS schedule kernel threads not the hardware thread. When a thread of a process gets blocked by requesting an I/O whole process gets blocked. kthreads are especially good for apps that frequently block. Kthread require a full thread control block (TCB) for each thread to maintain information about threads, as a result there is significant overhead and increased in kernel complexity.

Hardware threads improve performance of system by running multiple kernel threads on multiple hardware thread.

OS switch to other threads of same process when a thread get block by requesting to asynchronous I/Os. Similarly in hardware when a thread gets blocked by cache miss the Hardware platform choose other threads to execute.

(b) [**3 Marks**] What are the difference between Binary semaphore and counting semaphore? And how semaphore without busy wait can be implemented?

*Ans: A semaphore is a synchronization tool that provides more sophisticated ways (than Mutex) for process to synchronize their activities. Semaphore is abstract data type with one integer variable two atomic function Wait () and Signal (), the integer variable associated with a semaphore represent the number of available resources. **If the range of integer variable is 0 and 1 then it is binary semaphore otherwise counting semaphore.** A binary semaphore is same as Mutex or Lock.*

Semaphore without busy waiting can be implemented by adding a process list/queue of all busy wait to the ADT of semaphore. When a process don't get a chance of resource he will wait in a queue till the resource is free and his turn (waiting number). Source code is given bellow for both Wait () and Signal ().

```
wait(semaphore *S) {
  S->value--;
  if (S->value < 0) {
    add this process to S->list;   block();
  }
}
```

```
signal(semaphore *S) {
  S->value++;
  if (S->value <= 0) {
    remove a process P from S->list;     wakeup(P);
  }
}
```

(c) [**4 Marks**] Atomic function/instruction **Test and Set (TAS)** used to implement lock and unlock function of Mutual Exclusion (Mutex). Show that the above (in the Text Box) given TAS solution guarantees Mutual exclusion of a shared variable for M processes on N processors. You don't need PoSet/ToSet/TimedEvent to prove.

```
Shared boolean  Lock=false;//free
do {    while(TAS(Lock)) DoNothing();
                  Critical_Section();
          Lock=false;
                    Reminder_Section();
}       //Text box for Question 3.c
```

Ans: TAS function/instruction run atomically. Expanded code of TAS function can be written as

Bool synchronized *TAS (Lock){ // Run atomically (or No interruption)*

*RV=Lock; Lock=**true**; **return** RV;*

}

If a process try for TAS and it return a zero then lock is available and it acquire. Otherwise he needs to wait. Also in case of Lock is true and writing a true to Lock will not change the state of the lock.

*Synchronized method: it is not possible for two invocations of synchronized methods on the same object to interleave, so these methods are **atomic**. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object. When a **synchronized method exits, it automatically establishes a happens-before (ToSet:Total Order Set) relationship** with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.*

*As TAS executes in **total order** and from the code only one can get grand lock, so it satisfies the mutual exclusion of CS of N processes on M Processors. Also in hardware if TAS get implemented using LL/SC, it guaranteed the Total Order Set relationship in multiprocessor platform.*

(d) [**4 Marks**] What are the assumption Peterson solution assume to prove deadlock free and starvation free? How Peterson solution for Mutual exclusion of two processes on single processor, can be extended to work for two processes on two processors.

*Ans: Peterson mutual execution solution assumes **load instruction and store instruction are atomic**.*

Although Peterson solution works for process synchronization in uni-processor system, this solution is not suitable (impractical) for multiprocessor system. This is because busy-waiting by processors can cause excessive/substantial traffic on interconnection/BUS, there by degrading the system performance.

*To overcome this problem, **many multiprocessor systems provide instructions to atomically read and write a single memory location (TAS/CAS/XCHG/FIA). The operations of reading the word and storing into it are guaranteed to be indivisible - no other processor can access the memory word until the instruction is completed. The CPU executing the TAS instruction locks the memory bus to forbid other CPUs from accessing memory until it is done.***

*So we need to use the **TAS/CAS/XCHG to ensure atomic operation in multiprocessor**.*