Minor

● Graded

Student

Saharsh Laud

Total Points

20.5 / 60 pts

Question 1

Question 1 — **4** / 10 pts

1.1 **Part a** — **1** / 2 pts

- ✔ **+ 1 pt** No

- **+ 1 pt** Timer Interrupt
- **+ 0.5 pts** Context switch/scheduling
- **+ 0 pts** Incorrect

1.2 **Part b** — **0** / 4 pts

- **+ 2 pts** Alarm based system
- **+ 1 pt** Advantage of the scheme
- **+ 1 pt** Explanation of the scheme
- ✔ **+ 0 pts** Incorrect

1.3 **Part c** — **3** / 4 pts

- ✔ **+ 1 pt** **Inter-Processor Interrupts** : Identify the use of IPIs as the synchronization
- ✔ **+ 1 pt** **Master Core Role**: how the core with timer access serves as a coordinator
- ✔ **+ 1 pt** **Context Switch Triggering** :how IPIs trigger context switches on cores without timer access
- **+ 1 pt** **Scheduling Process:** how the system maintains fair scheduling across all cores
- **+ 0 pts** *Incorrect*

Question 2

Question 2 **3.5** / 10 pts

2.1 **Part a** **1** / 3 pts

✔ **+ 1 pt** Load both the DLLs (new & old) in **memory** rather than patching/overwriting existing DLL in-memory.

+ 1 pt Utilize concept of **stub function/dispatch table/indirection table** to identify which DLL's function is to be called.

+ 1 pt Use of **function pointers** in the described concept

+ 0 pts *Incorrect*

1 internally, stub function would allow the kernel to identify which DLL's function call is it be made

2.2 **Part b** **1** / 2 pts

+ 0.5 pts Use **reference counter** to count the specific version of DLL, to indicate which threads are using.

✔ **+ 1 pt** Explanation for **counter value** = 0 -> i.e., remove the old DLL.

+ 0.5 pts Method for updation of the counter value in the stub function.

+ 0 pts *Incorrect*

2 How would the kernel recognize the old DLL is no longer in use? It can't be removed if in active use by any of the threads of current process!

2.3 **Part c** **0** / 2 pts

+ 2 pts Resource information is stored in the task_struct of a process. file_struct for open files, mm_struct for memory mapping and socket handles for n/w connections. If the concept of transferring from task_struct is explained (i.e., no need to do any transfer, just pass the pointers, because new DLL and old DLL will share the same task_struct.)

✔ **+ 0 pts** incorrect

+ 1 pt If some components of the complete rubric are missing. Please tally your answer with the provided rubric to see what is missing.

2.4 **Part d** **1.5** / 3 pts

+ 1.5 pts Concept of dispatch table/indirection table/stub functions is explained

+ 1.5 pts concept of keeping the old and new DLL function entries in the dispatch/indirection table is explained

+ 3 pts Concept of stub function/wrapper function, need to keep old and new DLL function pointer in stub explained and function adapters(functions that can change the input signatures based on the context of the request and direct to the right DLL function explained)

+ 3 pts Concept of Dynamic Binary patching explained appropriately

+ 0 pts incorrect

+ 1 pt concept of keeping old DLL and new DLL alive is explained

✔ **+ 1.5 pts** partial explanation of exception handler mechanism

Question 3

Question 3 — **4** / 10 pts

3.1 **Part a** — **1** / 3 pts

+ 1 pt Yes, Share all memory regions except stack and TLS

+ 2 pts Justification: Fast inter-thread communication, easy **context switch** between threads

+ 0 pts Not correct

+ 1 pt Point adjustment

3.2 **Part b** — **3** / 3 pts

✔ **+ 3 pts** Each thread has its own private stack. These stacks are arranged in the shared virtual memory space one after the other such that they **do not overlap or interfere** with each other.

+ 0 pts Not correct

3.3 **Part c** — **0** / 4 pts

+ 1 pt Segment registers should be in the context of a thread. During thread context switch, the OS will load stack segment selector of newly scheduled thread into the segment register.

+ 2 pts Each thread's stack segment has a base (start of the stack) and a limit (max size). For a thread, Base + offset should lie in the segment (otherwise segmentation fault).

+ 1 pt Thread should not overflow its heap allocation as it can overwrite another thread's stack, basically entire space needs to be protected.

✔ **+ 0 pts** Not correct

Question 4

Question 4 — **2** / 10 pts

4.1 **Part a** — **1** / 3 pts

+ 1.5 pts Binary trees have a large depth, requiring multiple nodes along the search path to be locked during delete or update operations. This increases contention and limits parallelism, reducing system efficiency.

*Locking entire path from root to target node lock more nodes

*Implementing the only parent node and this node lock system is complex.

+ 1.5 pts Each cache line will contain multiple nodes of a binary tree. If a thread updates one of the node in a cache line, then it will lead to false sharing.

+ 1.5 pts Many levels (more no. of memory access)

+ 0 pts Incorrect

---

points adjustment

✔ **+ 1 pt** 1

+ 0.5 pts 0.5

4.2 **Part b** — **0** / 2 pts

+ 0.5 pts Parent nodes store bit vectors to track free entries in child nodes.

+ 1.5 pts not need to access all child node; find the child in parent node that have unallocated pid and go to only that child node

+ 1.5 pts Uses hardware instruction (bsf) for fast bit scanning, making operations efficient.

✔ **+ 0 pts** Incorrect

4.3 **Part c** — **1** / 5 pts

+ 0 pts Incorrect/Not Attempted

+ 1 pt TLB must be flushed whenever virtual to physical address mappings change

+ 1.5 pts Flushing is required when there is a context switch from a user process -> user process or kernel process -> user process

+ 1.5 pts Flushing is not required when switching from a kernel process -> kernel process or user process -> kernel process

+ 1 pt This is because kernel virtual memory mappings are the same for all processes

💬 **+ 1 pt** Point adjustment

Question 5

Question 5 — **6** / 10 pts

5.1 **Part a** — **0** / 3 pts

+ 1.5 pts Only Single thread that calls fork is "forked"

+ 1.5 pts Parent's entire virtual memory space is copied (except the stacks and execution context of other threads).

✔ **+ 0 pts** Wrong/Not Related to Question

It doesnot create a new thread inside the same thread group

5.2 **Part b** — **6** / 7 pts

✔ **+ 1.5 pts** Fork Bomb(We will have 2^n processes)

✔ **+ 2 pts** Before any write all processes will share the same physical memory pages

✔ **+ 0.5 pts** When any process writes to these shared COW pages,
Page fault occurs and Reverse mapping is done to determine all processes sharing the page

✔ **+ 2 pts** Kernel copies page and assigns new physical frame to writing proces, However other processes continue to share same old physical frame

+ 1 pt If Memory amplification exceeds physical RAM and swap space then the system will crash

+ 0 pts Wrong/unrelated

Question 6

Question 6 — **1** / 10 pts

6.1 **Part a** — **1** / 5 pts

✔ **+ 1 pt** When the function returns, it stores data in stack to pre designated memory.

+ 1 pt It calls the exit system call and enters the ZOMBIE state.

+ 1 pt The SIGCHLD signal is sent to the parent.

+ 1 pt The parent calls the wait system call, reads the return value from the pre□designated memory location, and collects the exit status of the child.

+ 1 pt Finally, the child's state is completely erased from the OS

+ 0 pts Click here to replace this description.

6.2 **Part b** — **0** / 5 pts

+ 3 pts Preventing Starvation

+ 2 pts Deferred tasks from Kernel Threads

✔ **+ 0 pts** Incorrect

Operating Systems Midterm Exam

2024-25 Sem II

Name: SAHARSH LAUD

Entry number: 2024 MCS 2002

- [ ] COL 331
- [x] COL 633
- [ ] ELL 405
- [ ] ELL 783

Total: 60 marks

- Each question is for 10 marks.
- Prove and justify all your answers.

1. Answer the following questions:
   a. Consider a thread running on a core. It does not make any system calls, nor does it suffer from any exceptions. Will it continue to run forever without giving other threads a chance, if it has an infinite loop? [2]

No, It will not keep running forever. A timer interrupt will be generated by CPU and the OS will come into action. The OS can either pause the process for this thread or terminate it thus ending the execution and scheduler will schedule another process in place of it. Timer interrupts are generated in jiffies.

   b. Consider a modern tickless or jiffy-less kernel. It does not use the solution that you proposed in 1.a. Additionally, there is no periodic event source. We still want the system to be responsive and have frequent context switches. No thread should monopolize a core. How will this scheme work? What are the advantages of such a scheme? [4]

SAMARTH LAUD
2024MCS2002

We can make use of exceptions & system calls so whenever a process requires os interruption for running on a core it can do so.
We can also use the concept of aging of a process by factoring a age number with a thread and if a younger thread wants to execute the os will execute it preempting the older thread. The OS can check for age of a thread and process all threads in order of ages.
This ensures context switches without timer interrupts.
The advantage is that there is no need for extra timer chip and we can maintain a sorted agewise list of processes. If some process needs immediate attention it can raise interrupt.

c. Now, consider a design where there are 8 cores. 7 of the 8 cores are not connected to any kind of timer. Only one of the cores can access any device that it wants. How can it coordinate the execution of the entire system? We clearly do not want any thread to run forever regardless of the core and monopolize it. [HINT: This will require concepts from Chapter 4] [4]

We can use concept of Inter-process Interrupt here. The 8 cores would have a local interrupt controller (Ex: LAPIC).
The core with timer interrupt will send this interrupt to all remaining 7 cores. The local interrupt controllers will service this interrupt and will interrupt the current executing thread. The OS then takes action corresponding to each cores interrupt.
This way even if a thread tries to monopolize the timer interrupt from the 8th core can interrupt this thread in any of the 7 cores (or the 8th core itself) through interprocess interrupt and we would have the desired effects. The interrupt handler for particular core would handle 8th cores interrupt sent.

SAHARSH LAUD
2024 MCS1002

2. Consider DLLs or shared objects. We wish to create a hot-swappable DLL, which can be replaced at run time. This means that a multi-threaded process can simply decide to upgrade its DLL while it is running. Any thread can issue a request to update the DLL. At that point of time, other threads could be executing functions in the old version. Regardless of this, the update process should happen seamlessly and cause minimal disruption.

   a. How do we ensure that the disruption is minimized? Note that there will be a short period of time in which both the DLLs are active (new and old). [3]

If there is some function in new DLL which is not supported by old version we can use Dynamic Library Translation and process it using existing function modules in older version or simply ignore it. The control will go to the line following the updated part present in new DLL & not in old one. So threads running on OLD DLL's won't notice the difference

   b. When can the old DLL's code and data be unmapped? (and how?) [2]

After all the threads executing the old DLL have finished execution we can unmap its old code and data with the updated version of the DLL. We can update the central image of the DLL which is shared by all threads and the threads can update their copies of the old DLL including code & data.

   c. How can resources acquired by the old DLL be transferred to the new DLL? This includes open files, mapped virtual memory regions, network connections, etc. [Hint: Think about the *task_struct*] [2]

The resources of the old DLL can be transferred by updating the task_struct values and adding the old values to the task_structs of threads utilizing the new DLL version This way the task_struct structure can hand details of resources occupied by old DLL as well new DLL.

SAHARSH LAUD
2024MCS2002

d. How can a system be created to use the older version of a function, if it has changed its signature in the new DLL? [3]

Method of Dynamic Library Translation can be used to use existing functions in old DLL & give similar results. Also it can maintain version numbering and use the versioning of the OLD DLL when the thread makes a call to execute it. This way older DLL function can be used through its version number.

3. Answer the following questions:

a. Do all the threads use exactly the same virtual memory space? Justify your answer. [3]

The kernel threads share the same memory space ie. all virtual addresses for different kernel threads are mapped to the same physical addresses. user threads for same process share the same virtual memory space. But for threads in different user processes the virtual memory space is different.

b. If they use the same virtual memory space, how are the stacks arranged in this shared space? [3]

(Thread local storage)

In the shared virtual memory space stacks are arranged based on different thread ids. The stacks are placed so that there is no overlap between stacks of different threads in the same shared memory space. Virtual space of one thread is isolated from another but exist in same space as a whole.

c. Design a mechanism to stop one thread from accessing the stack of another thread if all the threads share the same virtual address space. Can segmentation be somehow used? Justify your answer. [4]

We can segment the stack space corresponding to one thread ~~and~~ through segment registers and add segment id corresponding to a thread. If a thread with different segment id tries to access a particular thread's stack, the segment id mismatch would cause an exception thereby preventing unauthorized access.

AMARSH LAUD
2024MCS2002

4. Answer the following questions.

a. What is wrong with a binary tree (from an OS perspective)? List at least two reasons. Focus on concurrency aspects also. [3]

Most processes have similar prefixes which can be used while storing pid's. Binary tree doesn't use this fact unlike Radix tree.
Binary tree doesn't store any info about subtree in its node so even if pid is not free we need to search entire subtree unlike ~~augmented tree~~ idr tree.
Time To Find First 0/1 is more in Binary tree as compared to augmented tree

b. What is the advantage that one gets if a parent node stores information about its children in an IDR tree? [2]

The advantage is that if a particular pid is not available in subtree the parent nodes info can tell us this.
Also this helps in maintaining hierarchical structure and use common prefixes to reduce pid search time & search time for lowest free process id.

c. In which case is it necessary to flush the TLB and reload the page table? And why? Why is it not necessary in the other cases? [5]

If a particular requested page's entry is not present in TLB its looked in page table. If its not in Page table then there is a Page Fault and the page needs to be fetched from the swap space since its absent from main memory. Since Page was brought from swap space the spatial & temporal localities have changed and we need to Flush the TLB and reload Page table.
In other cases the page may not be in TLB but in Page Table entry ie. present in main memory. In such a case we just update TLB entries as TLB is still warm. So no need to flush TLB & reload Page Table.
It can also happen during context switch that TLB flushed & Page Table Reloaded.

SAHARSH LAUD
2024 MCS2002

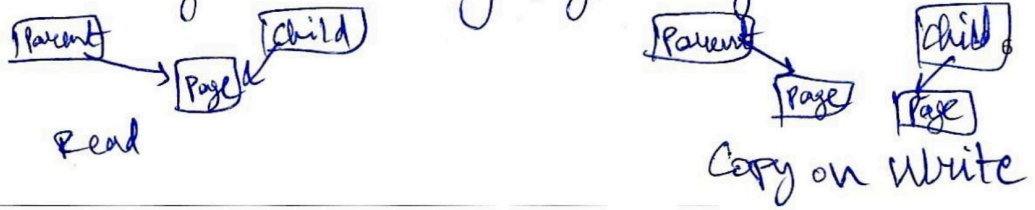
5. Answer the following questions:

a. What happens when a thread in a thread group is forked? [3]

when thread is forked it creates a new thread within same thread group with older thread as its parent since threads are light weight processes. The new thread shares memory space with all existing threads and is assigned thread group id of the thread group.

b. If a process repeatedly calls *fork n* times, what happens? [7]
Focus on the COW memory regions, actions taken when there is a write by a process and the potential to destabilize the system.

If a process repeatedly calls fork n times then 2^n processes would be created in total minus one original parent so $2^n - 1$ children forked. If there is a Read only operation by for child then the entire page ~~virtual~~ is as it is copied and both parent & child refer to same page since there is no chance of conflict.
But if there is a write then Copy on write mechanism creates separate copies for both the parent & child process and they make changes on their own copies of memory regions.
Now there can be destability if some page is Readonly before the fork() - If this page is written to it can lead to errors. So there is a PS2 bit and after forking it initially all Pages & processes are in Read mode write permission is disabled. If someone wants to write its a page fault. If PS2 bit is set then OS will know it's a Readonly page and cannot be written to so it makes separate copy of this page to write to. and original Read only page stays intact



SAMARTH LAUD

2024MCS2002

6. Answer the following questions:

a. Consider a work item that comprises a function pointer and its arguments. We wish to execute this item on a separate thread and later read its return value from a pre-designated memory location. What exactly happens when the function returns? How is the return value written to a pre-designated memory location, and how does the thread terminate? [5]

when The thread in work item inside work queue. ~~The~~ work item returns its function value is stored in predisignated memory pool for the work queue. The work item is executed on a separate thread inside work queue's work pool while other processes continue execution. After the function value is returned its stored in work queues memory stack (predesignated) and when all work items inside a work pool finish execution the return value can be fetched from work queues stack and the thread is terminated.

b. What is the rationale for running some *softirq* threads in process context? [5]

The rationale for running some softirq threads in process context is that sometimes a process might have functions or operations that need some immediate action but remaining operations can be completed after sometime not immediately.
Thus softirq (~~SOF~~) are run in process context so that the immediate action is performed and the background operations can be handled by the softirqs so that the process doesnot halt its execution.
Its like giving some immediate response to the processes needs and then running softirqs to process the lower lesser priority operations in background without hampering process execution to a large time.
Software Interrupt Requests work as lower levels of interrupt handler.

---------- Rough sheet ----------