Cs703 Current Midterm papers solved by Naina_Mailk

Q: 1 what are the Characteristics of Real-Time Operating system?

Characteristics of Real-Time Operating Systems

• Deterministic

✓ Operations are performed at fixed, predetermined times or within predetermined time intervals

✓ Concerned with how long the operating system delays before acknowledging an interrupt and there is sufficient capacity to handle all the requests within the required time

• Responsiveness

✓ How long, after acknowledgment, it takes the operating system to service the interrupt

o Includes amount of time to begin execution of the interrupt

o Includes the amount of time to perform the interrupt

o Effect of interrupt nesting

• User control

✓ User specifies priority

✓ Specify paging

✓ What processes must always reside in main memory

✓ Disks algorithms to use

✓ Rights of processes

• Reliability

✓ Degradation of performance may have catastrophic consequences

• Fail-soft operation

✓ Ability of a system to fail in such a way as to preserve as much capability and data as possible

✓ Stability

Q: 2Compare the kernel thread and user level thread?

Kernel threads

• OS manages threads *and* processes

✓ all thread operations are implemented in the kernel

✓ OS schedules all of the threads in a system

o if one thread in a process blocks (e.g., on I/O), the OS knows about it, and can run other threads from that process

o possible to overlap I/O and computation inside a process

• Kernel threads are cheaper than processes

✓ less state to allocate and initialize

• But, they‘re still pretty expensive for fine-grained use (e.g., orders of magnitude more expensive than a procedure call)

- ✓ thread operations are all system callsise
  - o context switch
  - o argument checks
- ✓ must maintain kernel state for each thread

User-level threads

• To make threads cheap and fast, they need to be implemented at the user level
- ✓ managed entirely by user-level library, e.g., libpthreads.a

• User-level threads are small and fast
- ✓ each thread is represented simply by a PC, registers, a stack, and a small thread control block (TCB)
- ✓ creating a thread, switching between threads, and synchronizing threads are done via procedure calls
  - o no kernel involvement is necessary!
- ✓ user-level thread operations can be 10-100x faster than kernel threads as a result

Q: 3 Briefly discuss the disadvantages of "Threads" paradigm?

Answer: Disadvantages of Threads paradigm are as follows:
1. It makes the overall code of the program, non-deterministic. It introduces Heisenbugs in the program. A Heisenbug is a type of bug that disappears or alters its behavior when you attempt to debug it.
2. In large, complex real life programs/problems, the programmer usually does not visualize properly that which synchronization mechanism should be deployed here in a particular situation. Hence, introducing wrong synchronization primitives in different scenarios, resulting in lots of undetectable bugs in the code.
3. As threads grow in numbers, for each thread kernel has to allocate memory internally for its stack. Hence the kernel's memory space is used for the purpose, it grows and soon reaches a limit beyond which it cannot expand. Secondly, when there are too many threads created, the benefit of parallelism achieved by overlapping I/O and CPU activities is destroyed / nullified.
4. As the number of threads grow, the overall thread management, done by a main controller thread, becomes a cumbersome job.
5. Context Switching overhead increases to a great extent, with the increase of no. of threads. Similarly thread scheduling and synchronization overhead increases with the increase in number of threads.

Q: 4 what are the desirable properties of a good CPU scheduling algorithm? Evaluate the FCFS scheduling algorithm on the basis of the established desirable properties?

Goals of "the perfect scheduler"

• Minimize latency: metric = response time (user time scales ~50-150millisec) or job completion time

• Maximize throughput: Maximize #of jobs per unit time.

• Maximize utilization: keep CPU and I/O devices busy. Recurring theme with OS scheduling

• Fairness: everyone gets to make progress, no one starves

FCFS scheduling algorithm:

Although the FCFS is very simple as it run job order as it arrives.
But it does not satisfy the above 3 goals of the scheduler.
- ✓ As the short jobs time is increase when these are beyond the long job so response time is not fast.
- ✓ FCFS not use the maximum num of the jobs per unit.

- ✓ As the FCFS is non-preemptive so that it is not using the I/O devices and only keep busy the CPU.
- ✓ In the fairness point of view it is fair because it give chance to everyone to progress.

Q: 5 Differentiates between distributed system and parallel system?

Distributed Systems

Distribute the computation among several physical processors.

• Loosely coupled system

✓ Each processor has its own local memory; processors communicate with one another through various communications lines, such as high-speed buses or telephone lines.

• Advantages of distributed systems:

✓ Resources Sharing, Computation speed up – load sharing, Reliability, Communications **Parallel Systems**

Multiprocessor systems with more than on CPU in close communication.

• Tightly coupled system

✓ Processors share memory and a clock; communication usually takes place through the shared memory.

• Advantages of parallel system

✓ Increased throughput, Economical, Increased reliability, graceful degradation, fail-soft systems

Q: 6Explain the reference counting and sweep in the garbage collection?

Garbage Collection

• How does the memory manager know when memory can be freed?

• Need to make certain assumptions about pointers

✓ Memory manager can distinguish pointers from non-pointers

✓ All pointers point to the start of a block

✓ Cannot hide pointers (e.g., by coercing them to an int, and then back again)

Reference counting

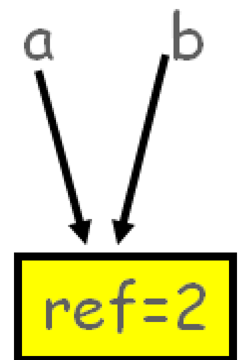
• Algorithm: counter pointers to object

✓ each object has ―ref count‖ of pointers to it

✓ increment when pointer set to it

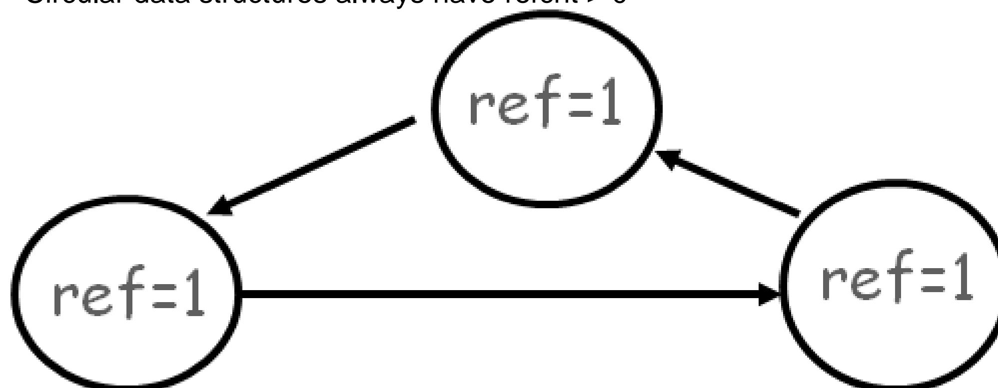
✓ decremented when pointer killed

```
void foo(bar c) {
   bar a, b;
   a = c;      <........................ c->refcnt++;
   b = a;      <........................ a->refcnt++;
   a = 0;      <........................ a->refcnt--;
   return;     <........................ b->refcnt--;
}
```



Problems

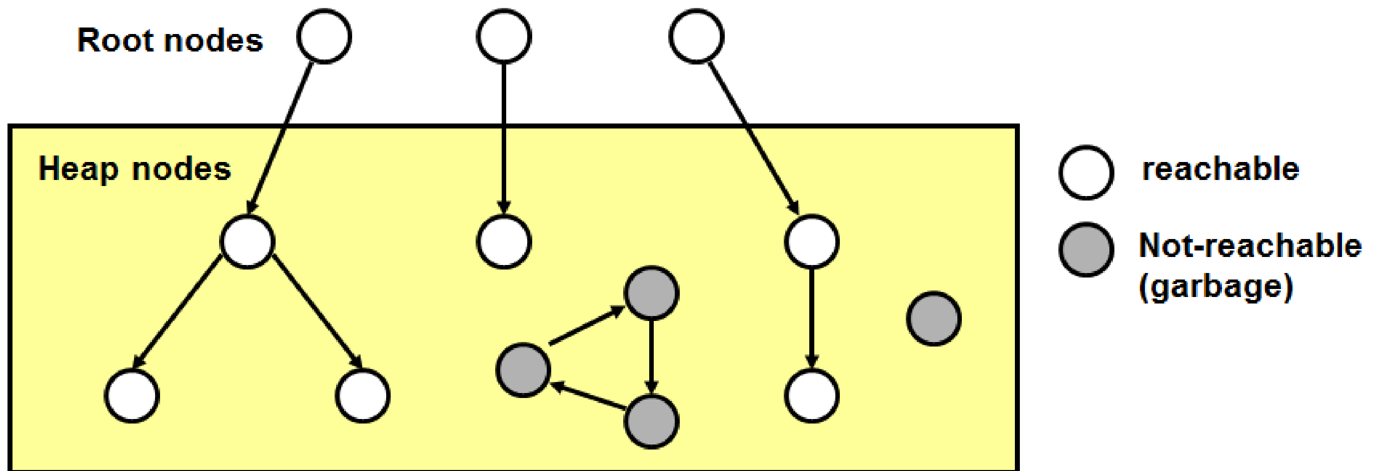
• Circular data structures always have refcnt > 0



Memory as a Graph

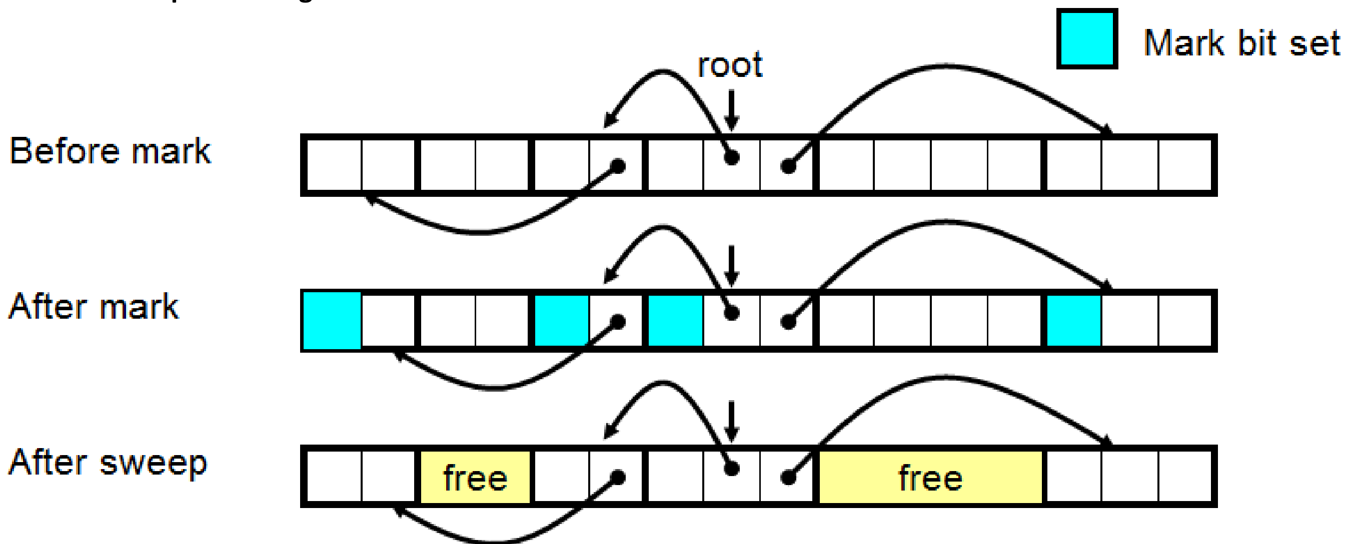
• We view memory as a directed graph

✓ Each block is a node in the graph
✓ Each pointer is an edge in the graph
✓ Locations not in the heap that contain pointers into the heap are called *root* nodes (e.g. registers, locations on the stack, global variables)



Assumptions
• Instructions used by the Garbage Collector
✓ is_ptr(p): determines whether p is a pointer
✓ length(b): returns the length of block b, not including the header
✓ get_roots(): returns all the roots

Mark and Sweep Collecting



Mark and Sweep
• Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // do nothing if not pointer
   if (markBitSet(p)) return        // check if already marked
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)   // mark all children
     mark(p[i]);
   return;
}
```

• Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
   }
}
```

Q: 7Short and long time slice in the RR?

RR Time slice tradeoffs

• Performance depends on length of the time-slice

✓ Context switching isn‘t a free operation.

✓ If time-slice is set too high (attempting to amortize context switch cost), you get FCFS. (i.e. processes will finish or block before their slice is up anyway)

✓ If it‘s set too low you‘re spending all of your time context switching between threads.

✓ Time-slice frequently set to ~50-100 milliseconds

✓ Context switches typically cost 0.5-1 millisecond

Q: 8Write a C program to create a child thread which show the massage "Hello World"?

The Pthreads "hello, world" Program

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *thread(void *vargp);

int main() {
  pthread_t tid;

  Pthread_create(&tid, NULL, thread, NULL);
  Pthread_join(tid, NULL);
  exit(0);
}

/* thread routine */
void *thread(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}
```

Thread attributes (usually NULL)

*Thread arguments (void *p)*

*return value (void **p)*

Q: 9Shortcoming of the SJF algorithm?

Answer:

As this algorithm is optimal in response time, average, latency but this big disadvantage of this algorithm is that we cannot predicate the future about the size of the job because we want to run the short size job first. e.g.

- We believe on the user that he will tell us the size and if user gives the wrong information to check it. So it will kill the job
- Secondly, if we are imagine the past for future.(if the last job was length 10 then next will also) but this is not a good solution for SJF as the size can change.

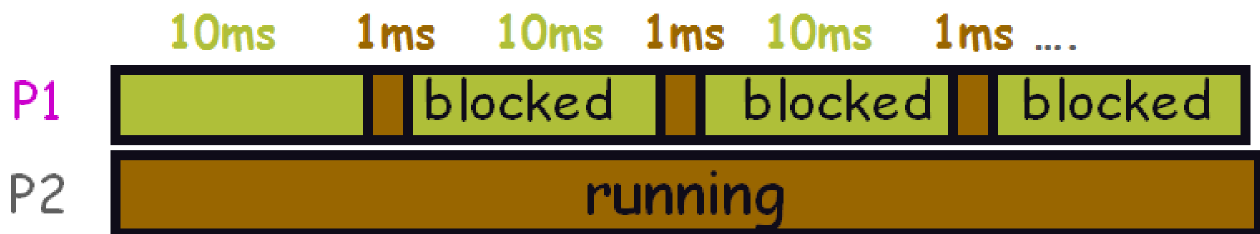
Q: 10Non-premeetive version of SJF?/ Non preemitive of SJF is not as efficient as the preemitive of the SJF discuss effects of short time slice to long in RR?

• STCF (or shortest-job-first)
- ✓ run whatever job has least amount of stuff to do
- ✓ can be pre-emptive or non-pre-emptive

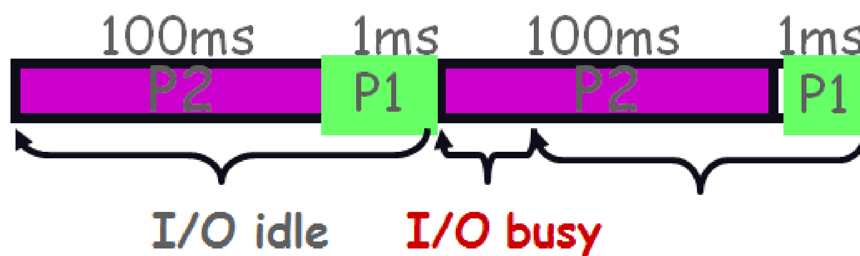
Provably optimal: moving shorter job before longer job improves waiting time of short job more than harms waiting time for long job.
As we know that ~STCF Offers better I/O utilization then the CPU and the I/O bounds are pre-emitive and CPU bounds are no-pre-emitive.

Non-preemitive jobs cannot be interrupts in the process while the preemitive can.

~STCF vs RR
• Two processes P1, P2



RR with 100ms time slice: I/O idle ~90%



- ✓ 1ms time slice? RR would switch to P1 9 times for no reason (since it would still be blocked on I/O)

• ~STCF Offers better I/O utilization

Q: 11 Features of real time OS?

Features of Real-Time Operating Systems
• Fast process or thread switch
• Small size
• Ability to respond to external interrupts quickly
• Multitasking with inter-process communication tools such as semaphores, signals, and events
• Use of special sequential files that can accumulate data at a fast rate
• Preemptive scheduling base on priority
• Minimization of intervals during which interrupts are disabled
• Delay tasks for fixed amount of time
• Special alarms and timeouts

Q: 12 Advantages and disadvantages of the monolithic design why layering is best for construction of the OS?

Monolithic design:
Early structure: Monolithic

• Traditionally, OS's (like UNIX, DOS) were built as a monolithic entity:

• Major **Advantages:**
✓ Cost of module interaction is low
• Disadvantages
✓ Hard to understand
✓ Hard to modify
✓ Unreliable
✓ Hard to maintain
• What id alternative?
✓ Find ways to organize the OS in order to simplify its design and implementation.

Layering
• The traditional approach is layering
✓ Implement OS as a set of layers
✓ Each layer presents an enhanced virtual machine to the layer above.
• The first description of the system was Djakarta's THE system.
✓ Layer 5: job managers
✓ Layer 4: device managers
✓ Layer 3: console manager
✓ Layer 2: pager manager
✓ Layer 1: Kernel
✓ Layer 0: Hardware

Q: 13 Static libraries and to short come of the static libraries?

Static Libraries:

Static libraries are simple collection of the ordinary object file. Usually end with ".a" suffix. Static libraries are not used as often they once were because of the advantages of the shared libraries Static libraries allows user to link to the program without having to recompile its code that's mean saving of time. Static libraries is used by the developers.

Shared Libraries

• Static libraries have the following disadvantages:
- ✓ Potential for duplicating lots of common code in the executable files on a filesystem.
- o e.g., every C program needs the standard C library
- ✓ Potential for duplicating lots of code in the virtual memory space of many processes.
- ✓ Minor bug fixes of system libraries require each application to explicitly relink

Q: 14 Difference between the semaphores and condition variables?

Semaphores

• semaphore = a synchronization primitive
- ✓ higher level than locks
- ✓ invented by Dijkstra in 1968, as part of the THE OS

• A semaphore is:
- ✓ a variable that is manipulated atomically through two operations, signal and wait
- ✓ wait(semaphore): decrement, block until semaphore is open also called P(), after Dutch word for test, also called down()
- ✓ signal(semaphore): increment, allow another to enter also called V(), after Dutch word for increment, also called up()

Condition variables

A simple example:

```
AddToQueue() {
lock.Acquire(); // lock before using shared data
put item on queue; // ok to access shared data
lock.Release(); // unlock after done with shared data
}
RemoveFromQueue() {
lock.Acquire(); // lock before using shared data
if something on queue // ok to access shared data
remove it;
lock.Release(); // unlock after done with shared data
return item;
}
```

Semaphores VS condition variables

1. Up differ from signal in that:

In condition variables, signal has no effect if no thread is waiting on the condition.
In semaphores, up has the same effect whether or not a thread is waiting.

2. Down differ from wait in that:

Down checks the condition and blocks only it necessary.
Wait is explicit it does not check the condition ever.

Q: 14 Fault tolerance and robustness?

Both describe the consistency of an application's behavior, but "robustness" describes an application's response to its input, while "fault-tolerance" describes an application's response to its environment. An app is robust when it can work consistently with inconsistent data. For example: a maps application is robust when it can parse addresses in various formats with various misspellings and return a useful location. A music player is robust when it can continue decoding an MP3 after encountering a malformed frame. An image editor is robust when it can modify an image with embedded EXIF metadata it might not recognize -- especially if it can make changes to the image without wrecking the EXIF data. An app is fault-tolerant when it can work consistently in an inconsistent environment. A database application is fault-tolerant when it can access an alternate shard when the primary is unavailable. A web application is fault-tolerant when it can continue handling requests from cache even when an API host is unreachable. A storage subsystem is fault-tolerant when it can return results calculated from parity when a disk member is offline.