CS 241
a.k.a. System Programming

Spring 2012, University of Illinos
as taught by Brigthen Godfrey

notes by Michele R. Esposito

Please, let report any error or type-o at micheleresposito@gmail.com

Contents

1 C Programming **3**
- 1.1 What is POSIX . . . 3
- 1.2 Explain the difference between a library function and a system call 3
- 1.3 Give an example of a POSIX system call used in CS241 lectures or reading. . . . 4
- 1.4 What is the * operator? What does it do? What is the & operator? What does it do? . . . 4
- 1.5 How do you define a function pointer? . . . 4
- 1.6 What is the difference between a global and a static variable? . . . 4

2 Memory **4**
- 2.1 Key concepts . . . 4
- 2.2 What is the difference between physical and virtual memory? . . . 5
- 2.3 What are the different mem allocation algorithms, and what are the advantages to each? . . . 5
- 2.4 How are VA transelated to physical addresses in multi-level page table? . . . 6
- 2.5 How does page size and the number of levels of page tables affect the number of entries in a page table? . . . 6
- 2.6 What is the difference between internal and external fragmentation? 7
- 2.7 What is the btw MMU and TLB? . . . 7
- 2.8 What are different page replacement policies and teh advantages of each? . . . 7
- 2.9 How the buddy system works and the run time for different operations . . . 8
- 2.10 What is thrashing? When does it occur? . . . 8
- 2.11 What causes a SEGFAULT and what happens when one occurs? 8
- 2.12 When is a process swapped out to disk? . . . 9
- 2.13 Three benefits of virtual memory . . . 9
- 2.14 Name one advantage of segmentation over paging, and one advantage of paging over segmentation . . . 9
- 2.15 Assuming a 32-bit address space and 4 KB pages, what is the virtual page number and offset for VA 0xd34f6a5 . . . 9
- 2.16 Give an example of a page fault that is an error, and an example of a page fault that is not an error . . . 9
- 2.17 Why are pages set to read-only in the copy-on-write technique? . 9
- 2.18 Suppose a 64-bit AS and 16KB pages. How big is the page table of a single process? . . . 9
- 2.19 Which scheme is better OPT or LRU? . . . 10
- 2.20 Why does LRU not suffer from Belady's anomaly? . . . 10
- 2.21 How does the VM subsystem know the exact location where a particular page is stored on disk, if its swapped out of memory? . 10
- 2.22 What is the working set of a process? . . . 10
- 2.23 Compare different memory allocation lists . . . 10

3 Thread and Processes **10**
3.1 Which resources are shared between threads of the same process? Which are not shared? 10
3.2 X is a global variable and initially x=0. what are the possible values for X after two threads both try to increment x? 11
3.3 What happens when a thread calls exit() 11
3.4 What happens to a process's resources when it terminates normally? 11
3.5 Explain the actions needed to perform a process context switch . 11
3.6 What are the advantages and disadvantages of kernel-level threads over user-level threads? 12
3.7 Compare the use of fork() to the use of pthread-create() 13
3.8 In a multiprocessor system, what conditions will cause threads within a process to block? 13
3.9 Explain how a process can become orphaned and what the OS does with it. How about zombies? 13
3.10 Describe how to use the POSIX call wait() 13
3.11 Explain what happens when a process calls exec() 13
3.12 Explain how re-entrant functions are used in C. 14
3.13 What are the maximum number of threads that can be run concurrently? 14
3.14 If a process spawns a number of threads, in what order will these threads run? 14
3.15 Explain how to use pthread-detach() and pthread-join() and why these are used. 14
3.16 Explain how a shell process can execute a different program . . . 14
3.17 Explain how one process can wait on the return value of another process 15
3.18 Describe the transitions between running, ready and blocked in the 5 state model. 15

4 Scheduling **15**
4.1 Which policies have the possibility of resulting in the starvation of processes? 16
4.2 Which scheduling algorithm results results in the smallest AWT? 16
4.3 What scheduling algorithm has the longest average response time? 16
4.4 Define turnaround time, waiting time and response time in the context of scheduling algorithms 16
4.5 What is the Convoy Effect? 16
4.6 Why do processes need to be scheduled? 17
4.7 What is starvation? 17
4.8 What is response time? what other metrics do we use to evaluate scheduling algorithms 17
4.9 Which scheduling algorithm minimizes avg initial response time? Waiting time? Turnaround time? 17
4.10 Why are SJF and Preemptive SJF hard to implement in real systems? 18
4.11 What does it mean to preempt a process? 18
4.12 What does it mean for a scheduling algorithm to be preemptive? 18
4.13 Describe the Round-Robin scheduling algorithm. Explain the performance advantages and disadvantages 18

4.14 Describe the First Come First Serve (FCFS) scheduling algorithm. Explain the performance advantages and disadvantages. . 18
4.15 Describe the Pre-emptive and Non-preemptive SJF scheduling algorithms. Explain the performance advantages and disadvantages 19
4.16 Describe the Preemptive Priority-based scheduling algorithm. Explain the performance advantages and disadvantages. 20
4.17 How does the length of the time quantum affect Round-Robin scheduling? 20
4.18 Define fairness in terms of scheduling algorithms. What are the fairness properties of each of the scheduling disciplines discussed in class? 20
4.19 Which scheduling algorithms guarantee progress? 20
4.20 A process was switched from running to ready state. Describe hte characteristics of the scheduling algorithm being used 20

5 Synchronization **21**

6 Semaphores / Mutexes **22**
6.1 Semaphores for mutual exclusion 22
6.2 When do you have to use a semaphore or mutex? 23
6.3 What is mutual exclusion? 23
6.4 How can you use a mutex lock to ensure that concurrent code operates correctly? 23

7 Processes and Deadlock **24**

8 Interprocess communication **26**

9 Networking **28**

10 File system and I/O **29**

Study Guide Midterm

CS 241
Spring 2012
Created by Michele Esposito

1 C Programming

1.1 What is POSIX

POSIX is the UNIX Interface Standard. It is a family of standards specified by the IEEE for maintaining compability between OSs. POSIX defines the application programming interface (API) along with command lines shells and utility interfaces, for software compatible with variants of UNIX and other operating systems. It includes:

- Process: creation and control
- Signals
- Floating point exceptions
- Segmentations
- C Library
- Timers
- Priority scheduling
- Shared Memory
- Semaphores
- Thread creation, control and cleanup
- Thread scheduling
- and much more ....

1.2 Explain the difference between a library function and a system call

System Calls A syscall is a request to the OS to perform some activity. They are expensive, as the system needs to perform the following operations:

- The hardware save its state
- The OS code takes control of the CPU, privileges are updated
- The OS examines the call parameters
- The OS performs the requested function
- The OS saves its state (and call results)

- The OS returns control of the CPU to the caller

A call to a library function can be anything like $printf$, that does not necessarily require a syscall. This is just like a normal C function, only comes from an external library.

1.3 Give an example of a POSIX system call used in CS241 lectures or reading.

fprintf, fclosef, pthread-create. There are a loot!

1.4 What is the * operator? What does it do? What is the & operator? What does it do?

The * operator is the dereference operator. When you use it, it looks up the memory inside the address of your variable.
The & is the address operator. When you use it tell you the address of your variable.

1.5 How do you define a function pointer?

A function pointer is a pointer which points to the address of a function.

1.6 What is the difference between a global and a static variable?

Static variables are local in scope, but their life is for the entire program. Globals instead have global scope.

2 Memory

2.1 Key concepts

- **Virtual memory**
  - Memory addresses used by an application
  - Unrelated to physical address
  - May not even be stored in physical memory
- **Physical memory**
  - The RAM in my computer
- **Memory Management Unit (MMU)**
  - Hardware which translates virtual to physical addresses every time any program accesses any memory
- **Page**
  - Unit in which OS allocates memory to applications

  - MMU also works in units of page

- **Page table**
  - DS used by MMU to remember virtual-to-physical mapping
  - One per process
  - Created by OS, stored in memory (at least top level)

- **Translation Lookaside Buffer (TLB)**
  - Cache of virtual-to-physical mappings
  - Faster than extra memory references needed to look up in page table
  - Must be flushed when switching between apps

- **Multilevel page table**
  - Top level page table points to other page tables rather than individual pages

- **Segmentation fault**
  - Program accesses memory outside the segment that it is allowed to access
    * defer. NULL pointer
    * write past end of heap ...

- **Page faults**
  - Happen when the VP is not currently mapped to a valid physical page
  - Seg fault is one kind of page fault

2.2 What is the difference between physical and virtual memory?

Physical memory is the actual memory on the machine, which is made of the HD, RAM and caches. Virtual memory instead is an illusion created so that the program can think it has an infinite amount of resources to draw from. it just adds a very important level of abstraction about dealing with memory.

2.3 What are the different mem allocation algorithms, and what are the advantages to each?

- **Implicit Free List**
  - Idea: Each block contains a header w some extra information
  - Allocated bit indicates free or not.
  - Size field indicates entire size of block (w header)
  - No explicit structure tracking location of free/allocated blocks
  - $O(n)$ worst case to allocate a free block.

- $O(n)$ to coalesce.
- Summary
  - **Implementation**: very simple.
  - **Allocate:** linear-time worst case.
  - **Free:** Linear-time worst case even with coalescing.
  - **Memory usage**: depends on placement policy.

- **Explicit Free Lists**
  - Use doubly liked listed, with boundary tags for coalescing, to keep track of free blocks.
  - Efficiency depends on insertion policy.

- **Buddy Allocators**
  - Special case of segregated fits
  - Divide into lists of powers of two.
  - Advantages: minimizes external fragmentation
  - Disadvantages: internal fragmentation when not 2^n sized request.

2.4 How are VA transelated to physical addresses in multi-level page table?

Starting from the VA, we have the following steps:

1. The CPU sends the VA to the MMU
2. The MMU decomposes the address, and sends the VPN to the TLB.
3. If TLB hit, then PA is send to the memory, data is returned
4. If TLB miss, then:
   (a) MMU looks up index in primary PT, to get secondary page table
   (b) MMU tries to access secondary PT.
   (c) MMU looks up index in secondary PT to get physical frame #.

2.5 How does page size and the number of levels of page tables affect the number of entries in a page table?

The more levels you have, the more number of entries you are allowed, as you have another level of indirection, which allows you to have more and more page. However, too many levels are a problem. The page size does not really effect.

2.6 What is the difference between internal and external fragmentation?

- External fragmentation
  - Unused chunks of memory between allocated chunks
  - Can't use for large contiguous allocations
- Internal fragmentation
  - Unused memory within allocated regions
  - Because we have allocated more than the requested

2.7 What is the btw MMU and TLB?

The MMU is a hardware unit that transelates the $VA \rightarrow PA$. However, this process is expensive, and it takes a long time to complete. A TLB, transelation lookaside buffer, is a cache for the MMU. Like that, the MMU will only need to compute the address once, and then it will be saved in the system.

2.8 What are different page replacement policies and teh advantages of each?

The goal of the page replacement algorithm is to evict the best page possible, which are those that will never be used again in the system.
Algorithms

- **OPT a.k.a. MIN**
  - Evict page that won't be used for the longest time in the future.
  - This requires to see the future, thus it cannot be implemented.
  - It is only a benchmark.
- **Random:** Throw out a random page. Not the best scheme ...
- **FIFO:** Throw out pages in the order that they were allocated.
  - Maintain a list of allocated pages.
  - When the length of hte list grows, pop first page off list and allocate it.
  - Advantages: Maybe the page allocated very long ago isn't used anymore.
  - Disadvantages:
    - Doesn't consider locality.
    - Suffers Belady's anomaly: Performance of a n app might get worste as the size of physical memory increases!
- **Least Recently used (LRU)**
  - Evict the page that was used the longest time ago.
    - Keep track of when pages are referenced to make a better decision

    - Use past behaviour to predict future behavior
  - Implementation
    - Every time a page is accessed, record a timestamps of the access time
    - When choosing a page to evict, scan over all pages and throw out page with oldest timestamps
  - Issues:
    - Not Optimal
    - Requires lot of space for time stamp.
  - Approx. LRU
    - Use a counter to keep track of the LRU.

2.9 How the buddy system works and the run time for different operations

The buddy allocator is a special type of segregated list.

- Basic idea:
  - Limited to power-of-two sizes
  - Can only coalesce with "buddy",who is other half of next-higher power of two
- Clever use of low address bits to find buddies.
- Advantage: low external fragmentation
- Disadvantage: Internal fragmentation when not 2^n sized requested.

Allocation is very fast, freeing not as much, as coalescing requires $O(N)$ time, but its still fast.

2.10 What is thrashing? When does it occur?

As systems become more loaded, spends more of its time paging, and eventually, no useful work will get done, as the system is overcommitted. Thus, the page replacement algorithm doesn't matter. Solutions?

- Change the priorities to " slow down" processes that are thrashing

2.11 What causes a SEGFAULT and what happens when one occurs?

A SEGFAULT is an attempt to access memory that the CPU cannot physically address. It occurs when the hardware notifies an OS about a memory access violation. The OS kernel then sends a signal to the process which caused the exception. By default, the process receiving the signal dumps core and terminates.

2.12 When is a process swapped out to disk?

When memory is full and the process is not running. By swapping that process to memory, new memory is available for the process to use.

2.13 Three benefits of virtual memory

1. **Protections:** processes cannot overwrite important areas of memory, or access them w/o authorization.
2. **Best memory usage:** Process can ask for more memory, and they are also allowed to get more and more space.
3. **Shared memory:** Processes can share memory, so that there is no need of copying the same memory twice.

2.14 Name one advantage of segmentation over paging, and one advantage of paging over segmentation

Segmentation is easy to implement, and fast to access as you only need to keep track of view variables.
Paging allows you better management of memory, less external fragmentation, better protection and better memory sharing.

2.15 Assuming a 32-bit address space and 4 KB pages, what is the virtual page number and offset for VA 0xd34f6a5

I would probably say it depends on architecture. However, $d34f6$ will be the VPN, and $a5$ will be the offset.

2.16 Give an example of a page fault that is an error, and an example of a page fault that is not an error

- A SEGFAULT would cause a page fault, that is an error, as you are trying to access memory that either doesn't exists or is protected
- A page that is not in memory, or that has not been paged.

2.17 Why are pages set to read-only in the copy-on-write technique?

Because the process cannot write on the memory, as it is shared between two processes. Instead, the process is only allowed to take a look at it. Whenever it is going to need to write on it, it is then going to be copied into new memory.

2.18 Suppose a 64-bit AS and 16KB pages. How big is the page table of a single process?

You would have $2^6 4(2^{-13})2^3 = 2^{54}$ bytes. Huge table! Does not work. By having a multi-level page table, we can have the first level point at the addresses that we are actually using within our AS.

2.19 Which scheme is better OPT or LRU?

OPT is better, because you are always evicting the page that won't be used for the longest time in the future. However, OPT cannot be implemented because you cannot forsee hte future, thus the correct answer is LRU.

2.20 Why does LRU not suffer from Belady's anomaly?

With LRU you are evicting the page that you have not used nearby. Thus, if your memory is larger, you are going to be able to keep more pages in, which is only goign to be an advantage for this algorithm.

2.21 How does the VM subsystem know the exact location where a particular page is stored on disk, if its swapped out of memory?

2.22 What is the working set of a process?

The *working set* of a program is a collection of those pages in its VA space that have been recently referenced. it includes both shared and private data. As the working set increases, memory demand increases. A process has an associated minimum working set size for the process. The working set size increase, memory demand increases.

2.23 Compare different memory allocation lists

- **Implicit**
  - Advantage: Simple implementation, useful for some specific applications
  - Disadvantage: $O(N)$ coalescing time, $O(N)$ allocation worst case.
- **Explicit**
  - Advantage: Fast memory allocation, small internal fragmentation.
  - Disadvantage: External fragmentation, can take $O(N)$ on allocation.
- **Segregated and buddy system**
  - Advantage: Fast allocation, no external fragmentation
  - Disadvantage: Internal fragmentation when not in powers of two.

3 Thread and Processes

3.1 Which resources are shared between threads of the same process? Which are not shared?

- **Shared:**
  - Address space.
  - Global variables.

  - Open Files
  - Heap
  - Process informations
  - Data segment
  - Code segment
- **Not shared:**
  - Program counter

3.2 X is a global variable and initially x=0. what are the possible values for X after two threads both try to increment x?

It is a free for all. There can be all sort of things happening. The value could be from 0 to 2.

3.3 What happens when a thread calls exit()

The kernel will clean all the memory associated with that thread, effectively killing it. An argument is returned along with the exit call.

3.4 What happens to a process's resources when it terminates normally?

The OS will clean them up, freeing all of its memory associated.

3.5 Explain the actions needed to perform a process context switch

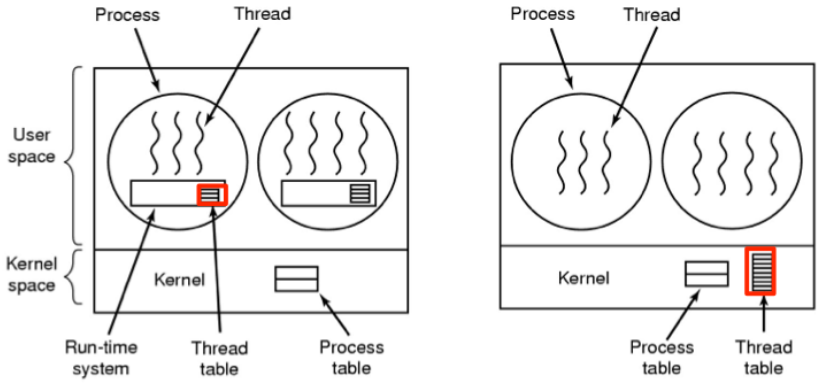
The OS needs to to the following for a process context switch:

- Switch to kernel
- Save the state of the process
- Change the Page Table Base pointer.
- Load the other process from memory and its associated state

The OS needs to to the following for a process context switch:

- Save the state of the process
- Switch to kernel
- Load the table of the other OS, load its stack, PC, registers and few other variables
- Go back to that process.

3.6 What are the advantages and disadvantages of kernel-level threads over user-level threads?



- **User Lever**
  - Advantages
    - Fast Context Switching: keep the OS out of it!
      - User level thread libraries do not require system calls
      - Have to use thread-yield:
      - save the thread information into the thread table
      - Call the thread scheduler to pick another thread to run
      - Saving local thread state scheduling are local procedures
      - Customized scheduling
  - Disadvantages
    - What happens if one thread makes a blocking I/O call?
      - Change the system to be non-blocking
      - Always check to see if a syscall will block
    - What happens if one thread never yields?
      - Introduce clocked interrupts
    - Multi-threaded programs frequently make system calls
      - Causes a trap into the kernel anyway!
- **Kernel-level Threads**
  - Advantages
    - Kernel schedules threads in addition to processes
    - Multiple threads of a process can run simultaneously
      - Now what happens if one thread blocks on I/O?
      - Kernel-level threads can make blocking I/O calls w/o blocking other threads of same process

    - Good for multi core architectures.
  - Disadvantages
    - Overhead in the kernel, extra DS, scheduling...
    - Thread creation is expensive.
    - Which thread should receive a signal?

3.7 Compare the use of fork() to the use of pthread-create()

pthread-create() creates a thread. The thread will start from the function that is passed withing the argument. Instead, fork will create a process, which will start from the next line, and return 0 to the child pid.

3.8 In a multiprocessor system, what conditions will cause threads within a process to block?

- Blocking call to I/O.
- Waiting for other threads.
- Shared memory might cause a block as well

3.9 Explain how a process can become orphaned and what the OS does with it. How about zombies?

A process becomes orphan when his parent is killed, via a SEGFAULT or exit call. In case of an error, the OS is going to kill it. Otherwise, the process is going to keep running normally. A zombie is going to get killed by the OS whenever the parent is going to terminate.

3.10 Describe how to use the POSIX call wait()

POSIX wait call is made for processes. It makes a parent process wait until its child process terminates. Useful for several reasons. Also, by waiting, the parent will make reap on his child, destroying all the traces of that process.

3.11 Explain what happens when a process calls exec()

- Load and runs in current process:
  - Executable filename
  - With argument list argv
  - And environment variable list envp
- Does not return(unless error)
- Overwrites code, data, and stack
  - Keeps PID, open files and signal context.

3.12 Explain how re-entrant functions are used in C.

A re-entrant function is one that can be interrupted in the middle of its execution and then safely called again before its previous invocations complete executing. They are used if you have code that will have HW interrupt or signal. They are so called thread safe functions.

3.13 What are the maximum number of threads that can be run concurrently?

It depends by the number of cores on your machine.

3.14 If a process spawns a number of threads, in what order will these threads run?

There is no way of knowing. It depends by the scheduling of the system, in case of kernel-threads. If you are implementing user threads, then you might be able to know.

3.15 Explain how to use pthread-detach() and pthread-join() and why these are used.

detach() is used if you have a joinable thread and you want to make it detachable. Then, ti meas you do not care anymore about that thread to return, and you will not wait anymore for it.
join() instead is used to make a thread wait for other threads, if they were specified to be joinable. Then, you will get the return value of the other thread, and you will be able to know when its going to return.

3.16 Explain how a shell process can execute a different program

It can do so by using fork(), followed by execve. Here is an example:

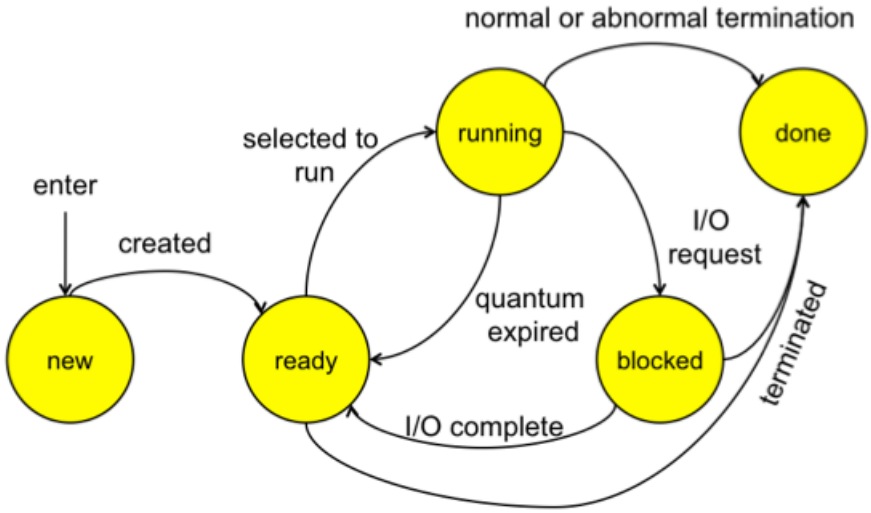
```
/* start fork */
pid_t pid = fork();
int child_status;

if ( pid == 0 ) // child
{
  int ret = execvp(arg1, cmd);
  if ( ret < 1 ) puts ("the comand does not exits");
  _exit(0);
}
else if ( pid < 0 )
{
  puts("Failed to fork");
  exit(1);
}
```

```
else // parent
  wait( & child_status );
```

3.17 Explain how one process can wait on the return value of another process

By using wait() look at the example above.

3.18 Describe the transitions between running, ready and blocked in the 5 state model.



4 Scheduling

High-level objectives

Objective	
Fairness	Equitable share of resources
Priority	Allocate to most important first
Efficiency	Make best use of equipment
Encourage good behavior	Can't take advantage of the system
Support heavy loads	Degrade gracefully
Adapting to different environments	interactive, real-time, multi-media

Quantitative objectives

Objective	
Fairness	Processes get close to equal share of the CPU
Efficiency	Keep resources as busy as possible
Throughput	Number of processes that complete per unit time
Waiting time	Time a process spends waiting in kernel's ready queue
Turnaround time	time from process start to its completion
Response time	Amount of time from when a request was first submitted until first response is produced

4.1 Which policies have the possibility of resulting in the starvation of processes?

Shortest job first and priority based algorithms can result in starvation of the process. The reason why is that you risk of always having other processes coming before, cutting the running time of the processes.

4.2 Which scheduling algorithm results results in the smallest AWT?

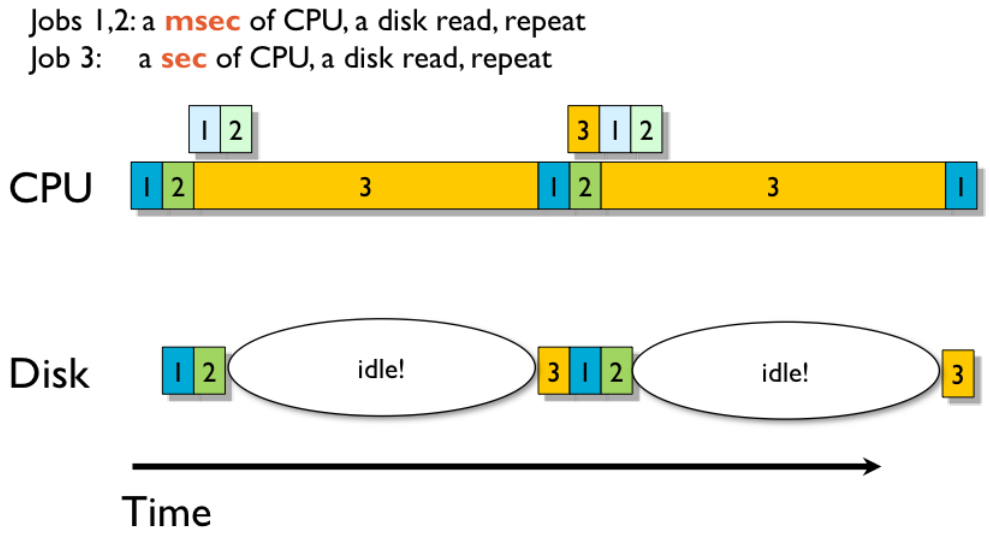
Preemptive SJF has provably the best AWT. in fact, in PSJF, the shortest processes always preempt the longer ones, minimizing the waiting time.

4.3 What scheduling algorithm has the longest average response time?

4.4 Define turnaround time, waiting time and response time in the context of scheduling algorithms

4.5 What is the Convoy Effect?

Is when all I/O processes are waiting for a long CPU process or vice versa. Thus, not smart utilization of resources.



4.6 Why do processes need to be scheduled?

Scheduling is deciding which process/thread should occupy each resource at each moment. We need it because when we have a pool of processes to choose from in ready state, we need to make a decision on which process is going to run, for how long and with what resource

4.7 What is starvation?

It means that the longest or low priority jobs may never be ended up scheduling, as the scheduler never pick them.

4.8 What is response time? what other metrics do we use to evaluate scheduling algorithms

- **Response time**
  - Amount of time from when a request was first submitted until first response is produced
- **Throughput**
  - Number of processes that complete per unit time
- **Waiting time**
  - Time a process spends waiting in kernel's ready queue
- **Turnaround time**
  - Time from process start to its completion

4.9 Which scheduling algorithm minimizes avg initial response time? Waiting time? Turnaround time?

- **Initial response time**
  - Round-robin, as every process will get scheduled in an average time quantum.
- **Waiting time**
  - Shortest Job First has provably the best waiting time.
- **Turnaround time**
  - Probably, SJF and Round robin.

This table reassures all the qualities.

Scheduling algorithm	CPU overhead	Throughput	Turnaround time	Response time
First in First out	Low	Low	High	Low
Shortest Job First	Medium	High	Medium	Medium
Priority based	Medium	Low	High	High
Round-Robin	High	Medium	Medium	High
Multilevel Queue	High	High	Medium	Medium

4.10 Why are SJF and Preemptive SJF hard to implement in real systems?

Because you are required to know the running time of the process, which is impossible because of the Halting problem. Systems that implement this type of scheduling look at

4.11 What does it mean to preempt a process?

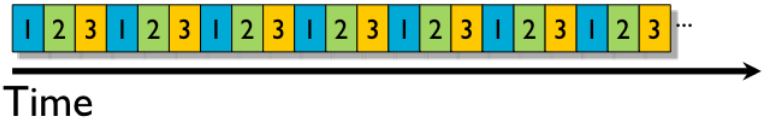
It is the act of temporarily interrupting a task, without requiring its cooperation, with the intention of resuming the task at a later time. Such a change is called context switch.

4.12 What does it mean for a scheduling algorithm to be preemptive?

It means that it will block a thread after a fixed time quantum expires. This is done to ensure that no process will suffer of starvation.

4.13 Describe the Round-Robin scheduling algorithm. Explain the performance advantages and disadvantages

Round Robin Select process/thread from ready queue in a round-robin fashion.



Time

- **Advantages**
  - Jobs get fair share of CPU
  - Shortest jobs finish relatively quickly
- **Disadvantages**
  - Might want some jobs to have greater share
  - Context switch overhead
  - Poor average waiting time with similar job lengths
  - Performance depends on length of time quantum

4.14 Describe the First Come First Serve (FCFS) scheduling algorithm. Explain the performance advantages and disadvantages.

- Process that requests the CPU first is allocated the CPU first
- Non-preemptive
- Implementation

  - FIFO queues
  - A new process enters the tail of the queue
  - The scheduler selects next process to run from the head of the queue
- **Advantages:**
  - Simple implementation.
  - Good for some specific implementation
- **Problems:**
  - Non-preemptive
  - Non optimal AWT
  - Cannot utilize resources in parallel
  - Result:
    * Waiting time depends on arrival order
    * Potentially long wait for jobs that arrive later
    * Convoy effect, low CPU and I/O device utilization

4.15 Describe the Pre-emptive and Non-preemptive SJF scheduling algorithms. Explain the performance advantages and disadvantages

- Job with shortest CPU time goes first
  - Often used in batch systems
- it is either preemptive or non-preemptive
- Non preemptive
  - **Advantages**
    * Low average waiting time
    * Helps keep I/O devices busy
  - **Disadvantages**
    * Not practical: cannot predict future CPU burst time
    * Starvation: Long jobs may never be scheduled.
- Preemptive SJF
  - Algorithm
    * Job with least remaining time to completion runs
    * So, a new job that is shorter than remainder of running job preempts it
  - **Advantages**
    * Similar to non-preemptive SJF
    * Provably minimal average waiting time
  - **Disadvantages**
    * Starvations again: long job keeps getting preempted by shorter ones.

4.16 Describe the Preemptive Priority-based scheduling algorithm. Explain the performance advantages and disadvantages.

- Rationale
  - Higher priority jobs are more mission-critical
- Each job is assigned a priority
- Select highest priority runnable job
  - FCFS or Round Robin to break ties
- **Disadvantages**
  - May not give the best AWT
  - Starvation of lower priority processes.

4.17 How does the length of the time quantum affect Round-Robin scheduling?

It changes a lot regarding the performance of your scheduler, as changes AWT, number of context switches and much more.

- Choice depends on
  - Priorities, architecture
- Typical quantum: 10-100 ms
  - Large enough that overhead is small percentage
  - Small enough to give illusion of concurrency

4.18 Define fairness in terms of scheduling algorithms. What are the fairness properties of each of the scheduling disciplines discussed in class?

By fairness, we mean that each process gets an equitable share of resources. In a more quantitative lever, we mean that the processes get close to an equal share of the CPU.

4.19 Which scheduling algorithms guarantee progress?

Round Robin, First Comes First Serves are guaranteed to never make your process starve.

4.20 A process was switched from running to ready state. Describe hte characteristics of the scheduling algorithm being used

The process must be preemptive. In fact, if such a transition happens, it implies that the job must have ran out of its time quantum.

5 Synchronization

1. **What is the readers-writers problem?**
   It is a generalization of the mutual exclusion problem. You have *readers* which can only read the object, *writer* threads that can only modify the thread. This problem occurs frequently in real time systems.

2. **What is the producers-consumers problem?**
   You have some consumer and some producers, which share a buffer. The point is if the buffer is full, then producers must wait. Else, if buffer is empty, consumers must wait. So, the point is to find a balance in between.

3. **What is the dining philosopher problem?**
   You have n dining philosophers, and only n forks. Each philosopher needs two forks to eat. The point of the problem is to have them eat without encounter a deadlock.

4. **Recognize a correct solution to the readers-writers problem, the producers-consumers problem, and the dining philosopher. Be able to identify and explain an error in a specific implementation of any of the classic synchronization problems.**

   - **Readers-writers:**
     use conditional variable to signal whether its safe or not. Otherwise, just wait for your turn.
   - **Producer- consumer:**
     You can use a mutex to signal the shared buffer, thus insert or remove item, and a semaphore to signal the number of empty slots, as well as the number of filled slots.
   - **Dining philosophers:**
     You can make the philosophers request resources in order. This way, you will be guaranteed to be able to always avoid deadlocks.

5. **What happens when readers are prioritized over writers in the classic readers writer problem? How about if writers are prioritized over readers?**

   - Favor readers: No reader waits unless a writer is already in critical section. A reader that arrives after a waiting writer gets priority over a writer.
   - Favor writers: Once a writer is ready to write, it performs its write as soon as possible. A reader that arrives after a writer must wait, even if the writer is also waiting.
   - **Both problem suffer of starvation!**

6. **What is required so that deadlock and starvation do not occur in the dining philosophers problem? Give examples of solutions.**

- You can make the philosophers request resources in order. This way, you will be guaranteed to be able to always avoid deadlocks.
- No hold and wait: do not allow the philosophers to hold the resources while waiting.

7. **What is the difference between starvation, deadlock, race conditions and critical sections? Describe each.**

   - Starvation: A thread is always waiting for some other event to finish.
   - Deadlock: Each process is waiting for another one in a circular manner
   - Race conditions: Occurs when separate processes or threads of execution depend on some shared state.
   - Critical sections: Operations upon shared states. They must be ensured to be mutual exclusive for the system to work correctly.

8. **What would happen if a systems hardware synchronization primitive were replaced with a software function?**
   I guess it would still work. Nevertheless, you could not achieve the same performances as in with hardware, and you would lack security features that hardware provides.

9. **Which type of variables must be protected against concurrent readers and writers in any combination?**
   Shared variables. That is, variables on the heap or global status variables.

6 Semaphores / Mutexes

6.1 Semaphores for mutual exclusion

Some terminology:

- **Binary semaphore**
  - Semaphore whose value is always 0 or 1
- **Mutex**
  - binary semaphore used for mutual exclusion
  - Used for exclusive access to a shared resource (critical section)
    - *wait* operation: "locking" the mutex
    - *post* operation: "unlocking" or "releasing" the mutex
    - *Holding* a mutex: locked and not yet unlocked
- **Semaphore**
  - Generalization of mutexes: count nubmer of available "resources"
  - Wait for an available resource (–), notify availability (++)
  - Example: wait for free buffer spae, signal more buffer space

- **Condition variables**
  - Represent an arbitrary event
  - Operations: wait foe even, signal occurrence of event
  - Tied to a mutex for mutual exclusion

Semaphores vs. condition variables

Semaphore	Condition Variable
Integer value (≥ 0)	No integer value
Wait doesn't always block	Wait always blocks
Signal either un-blocks thread —— increments counter	Signal either un-block thread or is lost
if signal releases thread, both may continue concurrently	if signal releases thread, only one continues

6.2 When do you have to use a semaphore or mutex?

When memory does not have a single clear cut owner, or when the two threads need to work together. In this cases, you cannot have separate memory, therefore you need a way of synchronizing the work of the two thread.

6.3 What is mutual exclusion?

Mutual exclusion refers to the problem of ensuring that no two processes or threads can be in their critical section at the same time.

6.4 How can you use a mutex lock to ensure that concurrent code operates correctly?

```
#include<semaphore.h>
...
int cnt=0;
sem_t cnt_mutex;

int main(void)
{
  ...
  /* initialize mutex */
  result = sem_init( & cnt_mutex, 0, 1);
  if ( result < 0)
    exit(-1);
  ...

  /* Clean up the semaphore that we're done with */
  result = sem_destroy( & cnt_mutex);
  assert( result == 0 );
}

void * worker ( coid * ptr)
{
  int i;
```

```
  for ( i=0; i < ITERATIONS_PER_THREAD; i ++) {
    sem_wait( & cnt_mutex);
    cnt++;
    sem_post( & cnt_mutex);
    }
}
```

1. **Understand the common semaphore and mutex functions (sem_wait(), sem_post(), etc).**

   - *_wait(): you ask you semaphore to wait until post operation, which is exit from the critical section. It locks the mutex.
   - *_post(): you increment the mutex, thus you release it from the blocked state.

2. **What are proper and improper code replacements for a test_and_set() operation?**
   you have to use atomic wait and post operations. This is an example:

   ```
   boolean test_and_set(boolean* lock) atomic {
     boolean initial = *lock;
     *lock = true;
     return initial;
   }
   ```

3. **How does the internal counter of a POSIX semaphore work? What does it mean if the value of the semaphore is 1?**
   If the value of a semaphore is 1, then we will keep waiting, as we are stuck in a while(1) loop.

4. **How can the reader-writer problem be solved using only POSIX mutexes?**
   You can make either writer or reader wait until a certain condition.

5. **Using only one mutex, is it possible to create a semaphore? If so, how? If not, why?**
   Yes, you can. You can associate a counter with that mutex, thus creating a semaphore. Though, you have to make sure that your mutex will not always block and that your counter has mutual exclusion.

6. **What are condition variables? Understand how they can be used in code.**
   Conditional variables represent an arbitrary event. The operations are wait for event, signal occurrence of event. It is strictly tied to a mutex for mutual exclusion.

7 Processes and Deadlock

1. **Define deadlock.**
   There exists a cycle of processes such that each process cannot proceed until the next process takes some specific action.

2. **Define circular wait, mutual exclusion, hold and wait, and no preemption. How are these related to deadlock?**

   - Circular wait: Every process is waiting for another in a circle fashion. Look definition of deadlock.
   - Mutual exclusion: Shared memory that two threads cannot access at the same time. Deadlocks work on mutual exclusion.
   - Hold and wait: A thread gets right on an object and keeps waiting to have other right to get the rest of work. Can cause a deadlock.
   - No preemption: A system will not stop a process from running. With preemptive, you will automatically resolve deadlocks, as you are giving up resources.

3. **How would the implementation of a web server using threads differ from one using processes?**
   It is a lot faster. You really do not need to run a process per incoming connection. However, you might have slight more complications due to the fact that you will need some good semaphores ad mutexes.

4. **What can happen if synchronization in a multiple-threaded program is not programmed carefully?**
   Race conditions will be broken. Everything could happen, is a free for all. Wrong data, segfauls, broken process flow.... Anything really.

5. **Why might an operating system use a resource allocation graph?**

   Because you will eb able to see which process allocated what, thus ensure to keep mutual exclusion.

6. **What are the conditions of a deadlock? How could you guarantee that each one of these conditions can be prevented?**

   - Mutual exclusion: give non-exclusive access only, read-only access.
   - hold-and-wait: Make a process request all the resources at one time.
   - No preemption condition: Allow preemption! It will break circular wait now.
   - Circular wait condition: Number resources and make processes ask for the minimum resource.

7. **What does waitpid() do?**
   Makes a process wait for another one, and then rips it whenever it finishes.

8. **What is the difference between Deadlock Prevention, Deadlock Detection & Recovery, and Deadlock Avoidance? What deadlock handling mechanism would you use?**

   - Prevention: Design system so that deadlock is impossible
   - Avoidance: Steer around deadlock with smart scheduling

- Detection & recovery: Check for deadlock periodically, recover by killing a deadlocked processes and releasing its resources.

The mechanism very much depends on what your goals are. If they are rare, and not worth the overhead, then even none of them.

9. **What are the components of a resource allocation graph?**
   Process, resource, request, ownership.

10. **What problem does the Bankers Algorithm solve? Given a set of processes how would you use the Bankers Algorithm?**
    Thanks to the banker algorithm, we are able to check if a given state of a system is safe. The steps are:
    Upon a resource allocation request:

    (a) Pretend that we approve the request.

    (b) Call function: would we then be safe?

    (c) If save, then approve. Else, reject.

11. **What is a safe state and how can you determine if a system is in a safe state?**
    There exists and execution order that can finish. In general, its hard to predict. Can do so by trying out.

8 Interprocess communication

1. **What is the difference between a FIFO and a pipe?**
   FIFOs are named pipes. They are special pipes that persist even after all the processes have closed them. They are actually implemented as a file. Memory persists.

2. **How would you redirect standard out to a file?**

```
int bak, new;
fflush(stdout);
bak = dup(1);
new = open("/dev/null", O_WRONLY);
dup2(new, 1);
close(new);
/* your code here ... */
fflush(stdout);
dup2(bak, 1);
close(bak);
```

3. **What happens when two processes read and write to a memory mapped file?**
   If its shared memory, then the mutual exclusion must be provided by the user. Otherwise the mutual excursion is provided by the OS.

4. **Explain how two processes can share memory using shmem.**

(a) Create identifier ("key") for a shared memory segment.

```
key_t ftok(const char *pathname, int proj_id);
k = ftok(    /my/file    , 0xaa);
```

(b) Create shared memory segment

```
int shmget( key_+t key, size_t size, int shmflg);
id = shmget(key, size, 0644 | IPC_CREAT);
```

(c) Access to shared memory requires an attach

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
shared_memory = (char *) shmat(id, (void *) 0, 0);
```

5. **Explain how a process can set which signals are caught or ignored using a signal set.**
   Using a custom signal handler. Any process can modify its signal handler simply by changing the signal mask of the given process, and specifying a new handler function. Here is an example:

```
struct sigaction sa;
sa.sa_handler = handle;  /* the handler function!! */
sa.sa_flags = 0;
sigemptyset(&sa.sa_mask);

sigaction(SIGING, &sa, NULL);
```

6. **How can one process send a signal to another?**
   Using the simple syscall kill(pid,signal). Works like in the shell.

7. **Describe the purpose of a POSIX signal.**
   Communicating between processes, invalid memory accesses, program termination.... anything that involves certain specific actions.

8. **Some signals cannot caught or ignored. Which signals are they and why shouldnt they be allowed to be caught?**
   SIGSEGV should not be ignored, as well as SIGKILL and SIGALRM.

9. **What does kill -¡parameter¿ pid do?**
   Sends signal number ¡parameter¿ to the given pid.

10. **How is the function sigwait() used?**
    the *sigwait()* function shall select a pending signal from set, atomically clear it from the system's set of pending signals, and return that signal number in the location referenced by sig.

11. **How does the function alarm() work?**
    Sends a SIGALRM after the time specified in the parameter.

9 Networking

1. **When do you use the close() system call with sockets?**
   You use close when you want to tear down a connection on both side. After it, you cannot write again to that socket.

2. **Discuss how a multithreaded web server running on a single processor system could be optimized using the process scheduling methods discussed in class. Which do you recommend?**
   I would probably use a round robin. In fact, By using RR you ensure that every client gets a fair share of the server, and most important, that everybody gets an answer in a quick time.

3. **How do select() and poll() work? What problem do they it solve?**
   You use them to wait for input. Select waits for readable/ writable file descriptors, whereas polls quereis the file descriptors for the events that you are asking.

4. **Describe the Posix accept() function.**
   Accept blocks waiting for a new connection. It returns the value of a socket description that the connection will be moved, so that you can still communicate over the previous socket.

5. **How does HTTP work?**
   The Hypertext Transfer Protocol is a communication protocol between client/server. It uses TCP and the purpose is transferring files and stuff. You can transfer a single or multiple file at once depending on the type of connection requested.

6. **Describe the services provided by TCP.**
   TCP provides reliable connection oriented transmission. It provides reliable transport, flow control and congestion control. However, it does not provide timing, throughput guarantees and security.

7. **How does TCP connection establishment work?**
   You need to cave a TCP connection request, and a TCP connection response. That happens through the SYN packet.

8. **Describe the services provided by UDP.**
   Provides unreliable data transfer. Does not provide: connection setup, reliability, flow control, congestion control, timing, throughput guarantee or security.

9. **How do sockets support the client-server model?**
   We use two ways to initialize a socket. The first one, the server listen for an incoming connection, in a so called "server socket". Then, the client connects to the socket and the server accepts the incoming connection.

10. **Which is better, UDP or TCP? Which one would you use?**
    I would use TCP. In fact, it would provide me with a reliable connection and a higher lever service.

11. **How does the Domain Name System (DNS) work?**
    The DNS system is a layer of indirecting between the IP address and the browser. It requests a recursive query of the different domains of several layers.

12. **How does DNS use caching?**
    You use caching as it greatly improves the time of response. Therefore, you have a caching local server to answer all the local requests, so that you don't have to go query the foreign server.

13. **How is DNS related to IP?**
    DNS translate into IP addresses.

10 File system and I/O

1. **Given a description of the block size and i-node structure, what is the maximum size of a file?**
   block size * inode_num + offset

2. **How many i-node operations are required to fetch a file at /path/-to/file?**
   The number of directories from the root to the file.

3. **What information is stored in an i-node? What information isnt?**
   The inode number is just the "index" of the inode, so that you can go and access the rest of the system from your disk. Also, an inode stores all the information about a regular file, directory or other file system, except its data and name. That is:

   - Size of the file in bytes.
   - Device ID (this identifies the device containing the file).
   - The Used ID of the file's owner.
   - The Group ID of the file.
   - The file mode which determines the file type and how the file's owner, its group, and others can access the file.
   - Link count telling how many hard links point to the idone.
   - Timestamps for may different things.

4. **What data structure best describes an i-node?**
   A bi-directional tree. That is, a tree with nodes going both up and down.

5. **What are the advantages and disadvantages of an i-node based file system?**
   The advantages are that is extensively used in many different places. However, the inode structure presents a simple way and flexible way of using file systems.
   There are a number of disadvantages of an extensible inode table. First, the "negative" inode space introduces quite a bit of complexity to the inode allocation ad read/write functions. Second, it is not easily extensible

to file stems that implement the proposed 64-bit block number extension. Also, it takes aways the beginning portion of the disk.

6. **Given the description of an i-node file system, how many i-node accesses are required to read the entire contents of a file of a given size? How many blocks does this file consume on disk?**
   it depends on the size of the block and many different things. However, we could derive a simple formula to fet all this values.

7. **What is an advantage of a soft link over a hard link?**
   A soft link does not change the inode structure, bit it binds to the name of another folder. Like that, it is clear what is the target of the node. Instead with a hard link, you don't really know which one is the link and which one is not. Once a soft link gets deleted, only the certain link is deleted. Also, soft links can link to external files or paths that doesn't exist.

8. **What is I/O polling? What are advantages and disadvantages?**
   In polling, the CPU issues I/O command, writes instructions to the device's registers and waits for completion. However, like this you have a busy waiting, which is the disadvantage, as the CPU always needs to keep polling. Also, it is expensive for large transfers. I is better for small, dedicated systems with infrequent I/O.

9. **Describe disk I/O access using DMA.** Direct memory access is typically done with interrupt-driven I/O. The CPU asks DMA controller to perform device-to-memory transfer. The DMA issues I/O command and transfers new item into memory. The CPU module is interrupted after completion.

10. **How are file descriptors shared between threads in a single process? How are they shared between after a process executes a fork()?**
    You can share different file descriptors using both caching, and shared memory.

11. **When the size of a block changes in an i-node based file system, how does this change the maximum size of a file?**
    You just need to change the metadata in the files.

12. **How do polling and interrupt driven I/O differ? What are the advantages and disadvantages of each?**

    - Polling: Expensive for large transfers. Its better for small, dedicated systems with infrequent I/O.
    - Interrupt-driven: Overcomes CPU busy waiting. I/O module interrupts when ready: event driven.

13. **How does the page-out process work?**
    You select the pages to evict, those that have not been evicted yet, and then you send them out to disk, so that you have more RAM, which you can use for whatever you need.

14. **Understand how hard-links result in different file names affecting the same i-node.**
    They do so because having multiple links to the same location, it is just like having two paths to the same folder.

15. **If an i-node based file system has a certain number of direct and single-indirect blocks, how large is the file?**
    number of blocks

16. **Where does fstat() look to find the information that it returns?**
    In the metadata file on the inodes.

17. **How does a file system use caching?**
    Most filesystems cache significant amounts of disk in memory. Like that, they will not have to go look many times inside the disk to see what's up with the inode structure below.