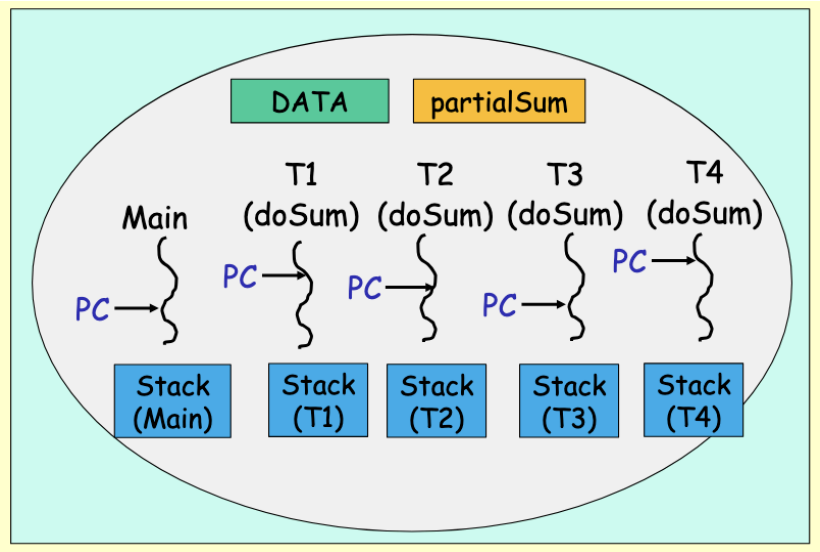
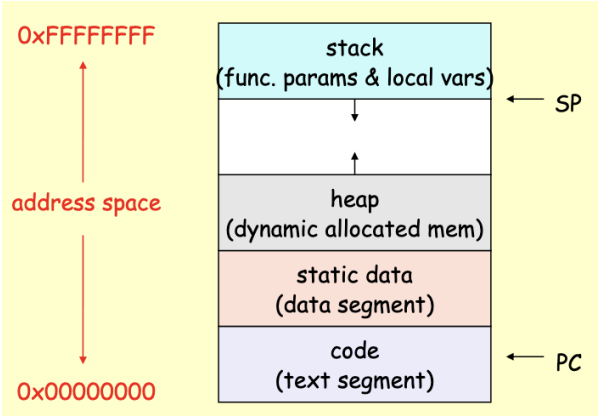
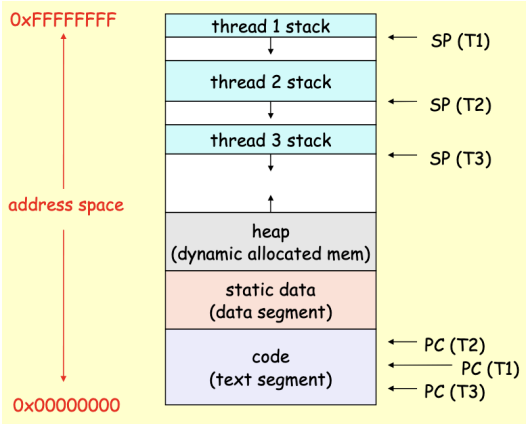
Topic 4 - Threading

What is a thread?

- A “unit of execution” (hardware execution state: **PC, general purpose registers, stack, and SP**)
- Each thread has a separate hardware execution state (like above), but share the same PCB.
- Observe that threads share everything (static & dynamic data)



Single-Threaded Processes Address Space	Multi-Threaded Processes Address Space
	

Process vs. Threads

- Process: defines the address space and general process resources (such as open files, sockets, etc.)
    - Can have multiple threads executing within them
    - Processes are just containers in which threads execute
- Thread: defines a sequential unit of execution within a process
    - All threads within a process share the same resources: code, data & heap
    - Bound to a single process
    - A unit of scheduling

Thread Methods

Function	Description
pthread_create	Creates a new thread
pthread_exit	Terminates the calling thread
pthread_join	Waits for a specific thread to terminate

Thread Creation Example

```
pthread_t threads[3];
int arg1 = 66;
int arg2 = 77;
double arg3 = 88.8;

pthread_create(&threads[0], NULL, func1, (void *)&arg1);

pthread_create(&threads[1], NULL, func1, (void *)&arg2);

pthread_create(&threads[2], NULL, func2, (void *)&arg3);

for(int i = 0; i < 3; i++){
      pthread_join(threads[i], NULL);
      printf("Thread %d terminated\n", i);
}
```

On the exam you can write it as:

```
threads[i] = CreateThread(func, params);
```

Usefulness?

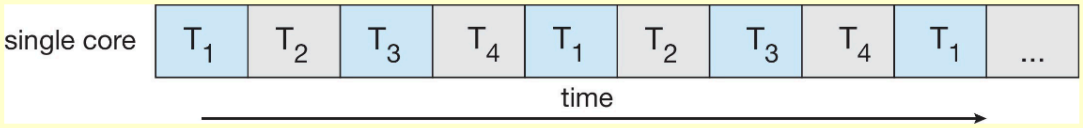
Multi-threading is still quite useful in a uniprocessor system, despite only one thread being able to execute in the CPU at a time. Example:

- Handling concurrent events (e.g., web servers, web browsers, edits, etc.)
  - One thread to handle each incoming request (Web server)
  - One thread downloading a page, one thread refreshing the GUI (Web browser)
  - One thread reading user input, one thread refreshing the GUI, one thread saving the file to the disk in the background (Editor)

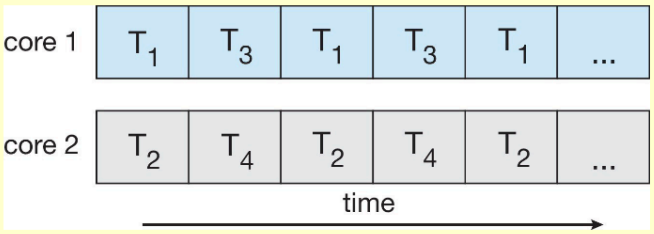
Concurrency vs Parallelism

Concurrency	Paralellism
Ability of a system to handle multiple tasks or threads, often by interleaving their execution	Ability of the system to execute multiple threads or tasks simultaneously
Does NOT necessarily mean that threads are executed at the same time; it can involve a single-core CPU switching between threads quickly	Parallelism requires multiple execution units, such as multi-core CPUs, where each task/thread can run on a different core
is about structure (dealing with many things at once)	is about execution (doing many things at once)

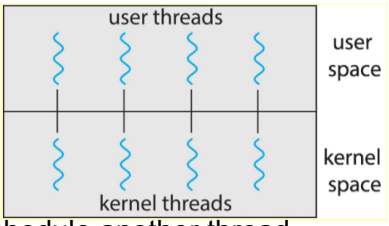
Concurrent execution of 4 threads T1, T2, T3 and T4 on a single-core system



Parallel execution of 4 threads T1, T2, T3 and T4 on a dual-core system



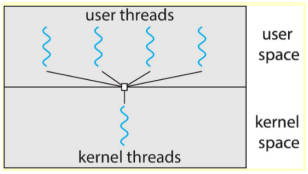
Threading Models



One-to-One

Each user-level thread maps to a kernel thread

- Creating a user-level thread creates a kernel thread
- Thread creation and management require system calls
- Advantage: When a thread blocks for I/O, the kernel can schedule another thread, leading to true concurrency. The kernel can also schedule multiple threads in parallel



Many-to-One

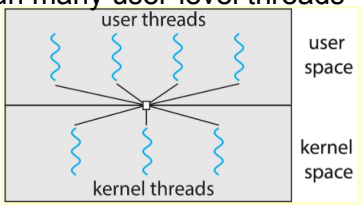
Many user-level threads are mapped to a single kernel thread

- A user-level library manages thread creation/management
- But: When a user-level thread blocks for I/O, all the other user-level threads get blocked with it. The kernel also can NOT schedule multiple user-level threads in parallel

Many-to-Many

Many user-level threads are mapped to many kernel threads

- OS allocates a sufficient number of kernel threads to run many user-level threads concurrently and in parallel
- Difficult to implement and not common



TLDR

Threads allow different parts of the program to be run, similar to fork, but without creating a new PCB. Threads share the PCB but have their own **PC, general-purpose registers, stack, and SP,** whereas with a fork, a new PCB is created. Threads are created using pthread_create: `pthread_create(&threads[0], NULL, func, args...);`. To start the thread, you use pthread_join: `pthread_join(threads[i], NULL);` The process terminates if all threads of the process terminate by calling pthread_exit(), or if one thread invokes exit(0);

Thread Practice Problems

1. Briefly explain how a thread differs from a process.

A process is a currently executing program, with its own PCB, that defines the address space and general process resources. A thread exists within a process and is a sequential unit of execution within a process, sharing the PCB. The only thing it doesn't share is its **PC, general-purpose registers, stack, and SP.**

2. What state (information) does a thread share with other threads in the same process, and what state (information) is private/specific to a thread? Be explicit in your answer.

A thread shares with other threads in the same process the same resources (code, data, and heap), but what they don't share (i.e., what is private to a thread) is the thread's state (PC, SP, stack, general-purpose registers).

3. What resources does the OS allocate for a thread during thread creation? How do they differ from those allocated when a process is created?

The OS allocates for a thread during thread creation its stack, PC, SP, and general-purpose registers in the same address space as the parent process. This differs from the resources allocated when a process is created, because when this occurs, an entirely new address space is allocated, new execution state (PC, registers, stack, and SP), and a new PCB (threads share the same PCB as the parent process).

4. What are the advantages of multi-threading? Give an example when multi-threading is preferable to a single-threaded process.

An advantage of multi-threading is that you can break a problem into smaller parts for faster execution. An example of this would be matrix multiplication. You can have numerous threads execute different parts of the problem "at the same time" for faster execution, but in a single-threaded process, you would just have one thread to handle the entire multiplication. This becomes tedious when the matrix becomes REALLY large, and will cause the process to last a very long time.

5. List 1 item that is shared by all threads, and 1 item that is private to each thread.

One item that is shared by all threads is the address space, and one item that is private to each thread is its program counters (PC).

6. What's the purpose of the "pthread_detach()" function?

The purpose of the pthread_detach() function is to make it so that the thread won't go into a zombie mode and wait until another thread terminates it with pthread_join, and that the thread can be terminated automatically with automatic resource cleanup upon termination.

7. List 2 conditions under which a multi-threaded process terminates.

Two conditions under which a multi-threaded process terminates are if all threads terminate or if one thread terminates using the exit() system call.

8. Define concurrency and parallelism, and briefly explain the difference between the two.
Concurrency is when a system can handle the execution of multiple tasks by interleaving their execution. Parallelism is when a system can handle execution multiple tasks simultaneously. The difference is that concurrency doesn't execute the tasks at the "same time", but rather appears to execute them at the same time through rapid context switching. Parallelism is accomplished by using multi-core CPUs, where tasks are scheduled on different cores in order to execute at the same time.

9. If you have a concurrent system , i.e., a multi-tasking or multi-threading OS, does the underlying hardware have to have multiple CPU cores? Why or why not?
The underlying hardware does **not** need to have multiple CPU cores. This is because tasks are able to be interleaved with one another, so a multi-core CPU is not required as those are for parallelism.

10. To achieve parallelism, does the underlying hardware have to have multiple CPU cores? Why or why not?
The underlying hardware **must** have multiple CPU cores. This is because if there aren't multiple CPU cores, the tasks will end up being interleaved, which is concurrency. For tasks to execute simultaneously, we need multiple cores so that we can execute each task on their own core. This is not possible on a single core CPU.

11. Give one reason why you will want to implement a multi-threaded application in a single-core machine.
You may want to implement a multi-threaded application on a single-core machine to break up parts of a problem into smaller parts for faster execution.

12. Consider an OS where the threads are implemented by a user-level thread library and all threads of a process are mapped to just one kernel thread (many-to-one mapping). Consider a process that has two threads, T1 and T2, and T1 is running. True/False: When T1 blocks while executing an I/O system call, the OS can switch to T2 and start running it. Justify your answer.

When T1 blocks will executing an I/O system call, the OS **cannot** (False) switch to T2 and start running it! This is because in a many-to-one mapping, if a thread is blocked while executing an I/O system call, all other threads are also blocked.

13. Consider an OS where each user-level thread has a corresponding kernel-level thread (one-to-one mapping) as in Linux. Consider a process that has two threads, T1 and T2, and T1 is running. True/False: When T1 blocks while executing an I/O system call, the OS can switch to T2 and start running it. Justify your answer.

When T1 blocks while executing an I/O system call, the OS **can** switch to T2 and start running it. This is because in a one-to-one mapping, each user-level thread is mapped to a kernel-level thread. This means that if one thread gets blocked, all other threads will not be affected, allowing the OS to switch to them and start running!

14. Consider a process with 4 threads in Linux. When a thread invokes "fork()" system call to create a new process, how many threads will the new process have? Briefly explain.

When a thread invokes the fork() system call to create a new process, the new process will have one thread. This is because this newly forked process will not clone the threads of the previous process.

15. Consider a process with 4 threads in Linux. When a thread invokes “ exec()” system call to load a new executable on top of the current process, how many threads will the new process have? Briefly explain.

When a thread invokes the exec() system call to load a new executable on top of the current process, this new process will have just one thread that executes what was passed in the system call, and the previous process and all other threads will be terminated.

16. What's the output of the following code when run in a Unix system? Assume you have 2 functions: (1) CreateThread(F, arg) creates a thread that starts running at the function “F” and takes the argument “arg” and returns the id of the created thread, (2) JoinThread(int threadId) is used to wait for the termination of a thread.

```
int g = 0;
int tids[2];
void T(int arg){
    if (arg == 3) {g++; printf("[MainThread] g: %d\n", g);}
    if (arg == 1) {g++; printf("[Thread1] g: %d\n", g);}
    if (arg == 0){
        JoinThread(tids[1]);
        g++;
        printf("[Thread0] g: %d\n", g);
    } //end-if
} //end-T

main(){
    tids[1] = CreateThread(T, 1);
    tids[0] = CreateThread(T, 0);
    JoinThread(tids[0]);
    T(3);
} //end-main
```

[Thread1] g: 1
[Thread0] g: 2
[MainThread] g: 3

18. Consider parallel summation: You are given an array A[0..N-1], and you are asked to compute the sum of the elements of this array in parallel as follows: The main thread will create

4 threads, each thread will compute the sum of N/4 elements of the array and terminate. The main thread will then wait for the termination of the worker threads, compute the overall sum, print it on the screen and terminate. Assume you have 2 functions: (1) CreateThread(F, arg) creates a thread that starts running at the function "F" and takes the argument "arg" and returns the id of the created thread, (2) JoinThread(int threadId) is used to wait for the termination of a thread. Assume that array A is global and N is divisible by 4.

```
#include <stdio.h>
#define N ...
int A[N];
// Array to be summed
int partialSum[4];
void sum(int id){
     int s = 0;
     int start = id * (N/4);
     int end = start + (N/4);
     for(int i = start; i < end; i++){
          s += A[i];
     }
     partialSum[id] = s;
}

int main(){
     int threads[4];
     for(int i = 0; i < 4; i++){
          threads[i] = CreateThread(sum, i);
     }
     for(int i = 0; i < 4; i++){
          JoinThread(threads[i]);
     }
     int s = 0;
     for(int i = 0; i < 4; i++){
          s += partialSum[i];
     }
     printf("The sum is %d", s);
     exit(0);
}
```

19. Write a multi-threaded program to find minimum element of an array A[0..N-1] with two threads. The first thread should find the minimum of A[0..N/2] and the second thread should find the minimum of A[N/2+1..N-1]. Each thread should start immediately upon being created and

each must execute the same function to find the minimum. After threads have been terminated, the minimum element of the array should be printed by the main thread. Assume that array A is global and N is divisible by 2.

```
#include <stdio.h>
#define N …
int A[N]; // Array for which the minimum is to be computed

int smallest[2];
void findMin(int id){
      int start = id * (N/2);
      int end = start + (N/2);
      int small = A[start];
      for(int i = start + 1; i < end; i++){
            if(A[i] < small){
                  small = A[i];
            }
      }

      smallest[id] = small;
}

int main(){
      int threads[2];
      for(int i = 0; i < 2; i++){
            threads[i] = CreateThread(findMin, i);
            JoinThread(threads[i]);
      }
      if(smallest[0] < smallest[1]){
            printf("The smallest number is %d", smallest[0]);
      }else{
            printf("The smallest number is %d", smallest[1]);
      }
      exit(0);
}
```

20. You are given an unsorted array A[0..N-1] and a "key", and you are asked to find the first index where key is stored in parallel as follows: The main thread will create 4 threads, each thread will search N/4 of the elements and terminate. The main thread will then wait for the termination of the worker threads, compute the overall search result, print it on the screen and

terminate. If the key does not exist in the array, then print “Key does not exist in the array”. Assume you have 2 functions: (1) CreateThread(F, arg) creates a thread that starts running at the function “F” and takes the argument “arg” and returns the id of the created thread, (2) JoinThread(int threadId) is used to wait for the termination of a thread. Assume that array A and “key” are both global and N is divisible by 4.

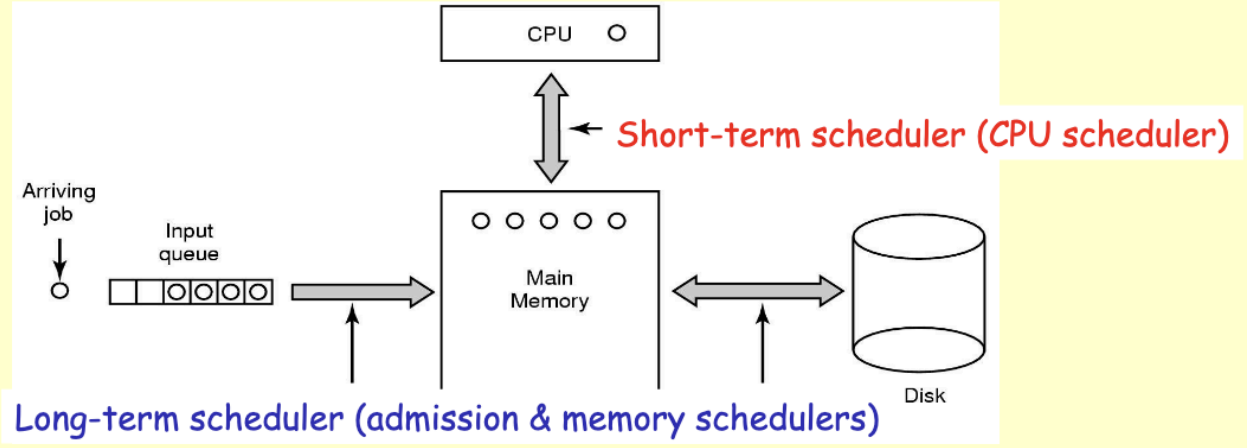
```
#include <stdio.h>
#define N ...

int A[N];
// Array to be searched
int key;
// Key to be searched

int index[4];
void findKey(int id){
	int start = id * (N/4);
	int end = start + (N/4);

	for(int i = start; i < end; i++){
		if(A[i] == key){
			index[i] = i;
			return;
		}
	}
}
int main(){
	for(int i = 0; i < 4; i++)
		index[i] = N;

	int threads[4];
	for(int i = 0; i < 4; i++){
		threads[i] = CreateThread(findKey, i);
	}
	for(int i = 0; i < 4; i++){
		JoinThread(threads[i]);
	}
	int smallIndex = N;
	for(int i = 0; i < 4; i++)
		if(smallIndex < index[i] && index[i] != N)
			smallIndex = index[i];
```

```
	if(smallIndex != N)
		printf(“The key was found at index: %d”, smallIndex);
	else
		printf(“The key does not exist!”);
	exit(0);
}
```

Topic 5 - Process Scheduling

Time Scales of Scheduling

- Long-term: determining the level of multi-tasking (multiprogramming)
    - How many jobs are loaded into primary memory
    - The act of loading in a new job (or loading one out) is swapping
    - Only available in batch systems
- Short-term: Which job to run next to result in “good service”
    - Happens frequently, want to minimize context-switch overhead



Short-Term (CPU) Scheduling

Among all **tasks** that are **ready to run** in a **multi-tasking system,** which one do i run next and for how long? (This is called CPU Scheduling)

Non-Preemptive Scheduling
If scheduling takes place only when the currently running thread

- Waits for an I/O
- Or terminates (exit)

The task voluntarily releases the CPU. This is called a non-preemptive scheduling algorithm

Preemptive Scheduling
If scheduling also takes place when:

- A new process is admitted to the system (new arrival)
- A task completes its I/O and is added back to the ready queue
- Via an interrupt (typically the timer interrupt)

The scheduler interrupts the current task and forces a context switch. This is called a preemptive scheduling algorithm!

Scheduling Goals

Goals depend on the type of system:

- Batch systems: strive to maximize job throughput and minimize turnaround time
- Interactive systems: minimize the response time of interactive jobs (such as editors or web browsers)
- Real-time systems: try to meet task deadlines

Avoid Process Starvation:

- Occurs when a process is not able to make progress
- A poor scheduling policy can cause starvation

Scheduling Algorithms

First-come, first-served (**FCFS**)

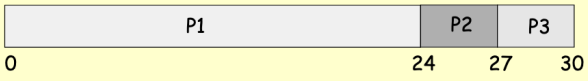
- Jobs are scheduled in the order that they arrive
- Typically non-preemptive
- Jobs are treated equally, no starvation

Problems:

- The average waiting time can be large
    - Small jobs are left waiting behind long ones (convoy effect)
    - Results in long turnaround time, long response times

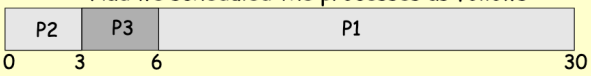
Example:

Process	Arrival Time	Burst Time
1	0	24
2	0	3
3	0	3



Waiting time for P1: 0, P2: 24, P3: 27 ␣ avg: (24+27)/3 = 17

Had we scheduled the processes as follows:



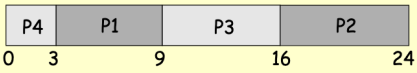
Waiting time for P1: 6, P2: 0, P3: 3 ␣ avg: (6+3)/3 = 3

Shortest-Job First (**SJF**)

- Choose the job with the smallest expected CPU burst. **FCFS scheduling used to break ties**
- Can be preemptive or non-preemptive
    - The choice occurs when a new process with a smaller CPU burst than the currently executing one becomes available in the ready queue
    - The preemptive version will preempt the currently executing process and dispatch a new process, BUT the non-preemptive version will NOT
    - Preemptive version is sometimes called shortest-remaining-time-first scheduling

Example non-preemptive:

Process	Arrival Time	Burst Time
1	0	6
2	0	8
3	0	7
4	0	3



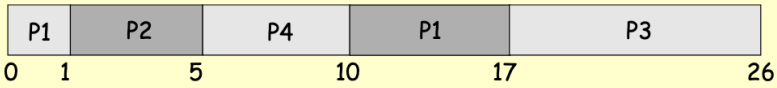
Process	Waiting Time
P4	0
P1	3
P3	9
P2	16

Average waiting time: [(9-6)+(24-8)+(16-7)+(3-3)]/4 = 7

can prove that this has optimal min. average waiting time

Example preemptive:

Process	Arrival Time	Burst Time
1	0	8
2	1	4
3	2	9
4	3	5



Average waiting time = [(17-8)+(5-1-4)+(26-2-9)+(10-3-5)]/4 = 26/4 = 6.5

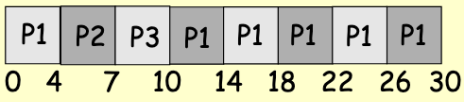
Round Robin Scheduling

- Designed especially for time-sharing systems
- Similar to FCFS, but preemption is added
- A time quantum is defined, typically 10ms
    - A process runs a full quantum or until it blocks
- The scheduler picks the first process from the queue, sets a timer interrupt to fire after 1 quantum, and dispatches the process
- One of 2 things can happen
    - The process issues an I/O before its quantum expires and voluntarily releases the CPU
    - The quantum expires (the timer goes off) A context switch occurs, and the process is put at the tail of the queue and the head process is dispatched

Example:

Process	Arrival Time	Burst Time
1	0	24
2	0	3
3	0	3

Assume that the time quantum is 4



Average waiting time: (6+4+7)/3 = 5.66

Priority Scheduling

Assign priorities to processes

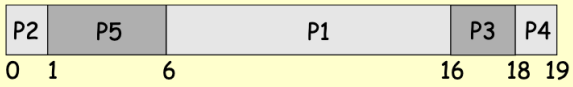
- Choose the process with the highest priority to run next
- If tie, use another scheduling algorithm to break
- To implement SJF, priority = expected length of the CPU burst

Can be preemptive or non-preemptive

- The choice occurs when a new process with a higher priority than the currently executing one becomes available in the ready queue
- Preemptive version will preempt the currently executing process and dispatch the new process, BUT the non-preemptive version would not

Non-Preemptive Priority Scheduling Example:

Process	Arrival Time	Burst Time	Priority
1	0	10	3
2	0	1	1
3	0	2	3
4	0	1	4
5	0	5	2



Average waiting time: [(16-10)+(1-1)+(18-2)+(19-1)+(6-5)]/5 = 8.2

Priority Scheduling with Round Robin

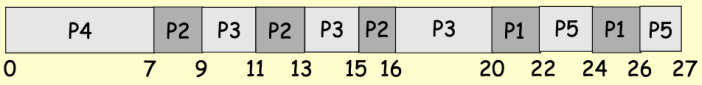
Run the process with the highest priority

- Processes with the same priority run round-robin with a time quantum

Example:

Process	Arrival Time	Burst Time	Priority
1	0	4	3
2	0	5	2
3	0	8	2
4	0	7	1
5	0	3	3

Assume that the time quantum is 2



Average waiting time: (20+11+12+0+24)/5 = 13.4

FCFSA

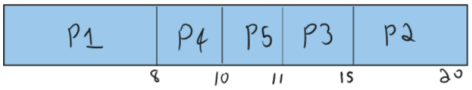
FCFS

Process	Arrival Time	Burst Time	Finish Time	Turnaround (FT-AT)	Waiting Time (TA-Burst)
1	0	8	8	8	0
2	2	5	13	11	6
3	3	4	17	14	10
4	8	2	19	11	9
5	10	1	20	10	9

Shortest Job First (SJF)

SJF

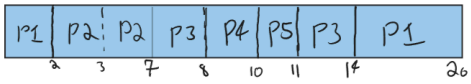
Process	Arrival Time	Burst Time	Finish Time	Turnaround (FT-AT)	Waiting Time (TA-Burst)
1	0	8	8	8	0
2	2	5	20	18	13
3	3	4	15	12	8
4	8	2	10	2	0
5	10	1	11	1	0



Shortest Remaining Time First (SRTF)

SRTF

Process	Arrival Time	Burst Time	Finish Time	Turnaround (FT-AT)	Waiting Time (TA-Burst)
1	0	8	20	20	12
2	2	5	7	5	0
3	3	4	14	11	7
4	8	2	10	2	0
5	10	1	11	1	0

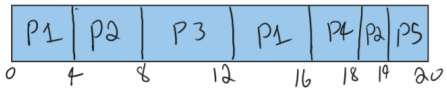


Round Robin (RR)

RR

Process	Arrival Time	Burst Time	Finish Time	Turnaround (FT-AT)	Waiting Time (TA-Burst)
1	0	8	16	16	8
2	2	5	19	17	2
3	3	4	12	9	5
4	8	2	18	10	8
5	10	1	20	10	9

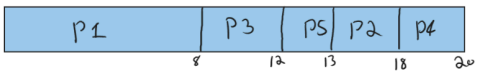
Quantum = 4



Priority Scheduling (Non-Preemptive)

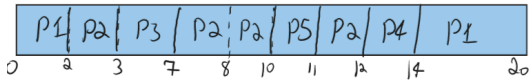
Priority (Non-Preemptive)

Process	Arrival Time	Burst Time	Priority	Finish Time	Turnaround (FT-AT)	Waiting Time (TA-Burst)
1	0	8	4	8	8	0
2	2	5	2	18	16	11
3	3	4	1	12	9	5
4	8	2	3	20	12	10
5	10	1	1	13	3	2



Priority Scheduling (Preemptive)

Process	Arrival Time	Burst Time	Priority	Finish Time	Turnaround (FT-AT)	Waiting Time (TA-Burst)
1	0	8	4	20	20	12
2	2	5	2	12	10	5
3	3	4	1	7	4	0
4	8	2	3	14	6	4
5	10	1	1	11	1	0



Multi-level Feedback Queues

Scheduling algorithms can be combined in practice

- Have multiple queues
- Pick a different algorithm for each queue

Multi-level feedback queues (MLFQ)

- Multiple queues representing different job types
    - System, interactive (foreground), CPU-bound, batch (background) etc.
- Queues have priorities
    - Schedule jobs within a queue using RR
- Jobs move between queues based on execution history
    - Feedback: switch from CPU-bound to interactive behavior

Practice Questions - Scheduling

1. Briefly describe the difference between the long-term scheduler and the short-term (CPU) scheduler.

The difference between the long-term scheduler and the short-term (CPU) scheduler is that the short-term scheduler focuses on scheduling tasks into the **CPU** to result in “good service”. The long-term scheduler involves the admission of jobs into the system by selecting which jobs are loaded into **main memory**. The main difference is that long-term schedules into main memory and short-term schedules into the CPU.

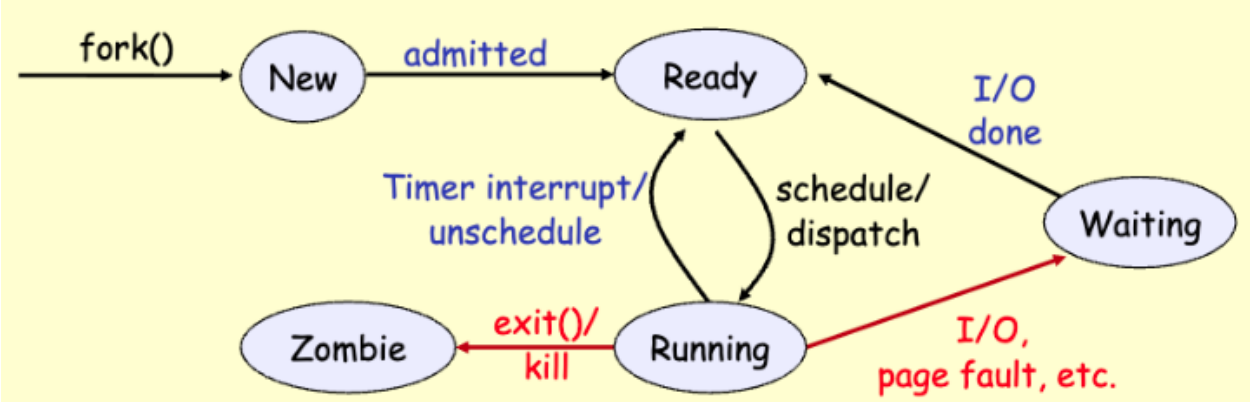
2. List 4 events that might occur to cause the kernel to context switch from one user process to another.

One event that might occur to cause the kernel to context switch from one user process to another is that the process encounters some I/O and is moved to the waiting queue. Another event is that the process terminates while in the running queue, so the next process must be loaded from the ready queue into the running queue. Another event could be the result of a preemptive priority scheduling algorithm, where one process has a higher priority than the current one, then the current process will be preempted, causing a context switch. Finally, the process’s time quantum can expire in, for example, round robin scheduling, causing the current process to be preempted and the next process in the ready queue to be scheduled.

3. What's the difference between preemptive and non-preemptive scheduling? Briefly explain. Which one is more suitable for a timesharing system?

Preemptive scheduling is when you move the currently executing process from running Q to ready Q based on some scheduling algorithm. Non-preemptive scheduling is when you **do not** forcibly release a process from the running state. Timesharing is more suitable in preemptive scheduling since you need quick response times in timesharing systems, and for this, you need rapid context switching between unfinished processes.

4. Draw the process state diagram. Clearly mark each transition between different states.



5. What state is saved on a context switch between threads?

The current state of the thread is saved on a context switch between threads. This includes the values of general-purpose registers, PC, SP, etc.

6. Selecting a proper time quantum is important in round robin scheduling. What happens if the quantum is too big? What happens if it is too small? Briefly explain.

If the quantum is too big, then it simply just becomes a first-come first-served scheduling algorithm. This is because this algorithm uses a circular queue, adding to the end on preemption and new arrivals. If the quantum is too big, then the tasks will just be executed in the order that they came! If the quantum is too small, then you will have costly rapid context switching.

7. What's starvation as it applies to process scheduling? In which scheduling discipline is it likely to occur? Give an example.

Starvation as it applies to process scheduling is when a process is blocked from being executed for a long period of time by the execution of other processes (CPU busy). This is likely to happen in Shortest Remaining Time First. An example using SRTF is lets say there a process executing with 10 units of time left. Then lets say 100 jobs enter the ready queue with a bust time of 8 for each of them. Then the current job will be preempted and can't continue execution until those 100 jobs finish executing.

8. A friend of yours is to design an OS that would run batch jobs, and s/he decides to employ the Round Robin Scheduling algorithm. Is your friend making a good choice? Why or why not?

No because batch jobs are ran in the background and don't necessarily require fast response time. Plus if you use round robin, your jobs are likely to be constantly interrupted. A better choice may be First Come First Serve (FCFS)

9. Is it possible for a process to move from the Ready state to the Waiting (blocked) state? Why or why not? Briefly justify your answer.

It is NOT possible for a process to move from the ready state to the waiting state! This is because the process itself must be running inorder to encounter an interrupt or get blocked by I/O, something that can't happen in the ready state.

10. Consider a system that wants to maximize throughput. What scheduling algorithm would you use in such a system? Justify your answer.

A system that want to maximize throughput should use a scheduling algorithm such as Shortest Job First. By throughput we mean finish as many jobs as possible as fast as possible, so we can use a scheduler that prioritzes the shortest jobs to accomplish this so that more jobs are completed in a specified time frame.

11. In Round-Robin scheduling, new processes are placed at the end of the queue, rather than at the beginning. Suggest a reason for this.

New processes are placed at the end of the queue rather than at the beginning because if they were placed are the beginning than earlier processes not yet scheduled would be starved.

12. Explain why Round-Robin scheduling tends to favor CPU bound processes over I/O bound ones.

Round-Robin tends to favor CPU boud over I/O bound processes because a process's execution won't be blocked before the time quantum expires. It will continue using the CPU until the time quantum expires, and then preempt. In I/O bound it can be blocked before the time quantum expires, causing less CPU utilization.

13. CPU scheduling quanta have remained about the same over the past 20 years, but processors are now over 1,000 times faster. Why has CPU scheduling quanta NOT changed?

CPU scheduling quanta has NOT changed because it has been widely researched that the current scheduling quanta is sufficient for users.

14. Briefly explain how the shortest job first scheduling works.

The Shortest Job First (SJF) Scheduling Algorithm works by dispatching processes to the running queue based on their predicted burst time. Processes with shorter burst times are prioritized, and processes are not preempted when a shorter job enters the ready queue.

15. Briefly explain multi-level feedback queuing scheduling algorithm.

A multi-level feedback queueing algorithm is one where you have multiple queues, each with different priorities and their own scheduling algorithm. Jobs are scheduled to a specific queue using Round Robin. Jobs can switch between queues based on their behavior, for example using a certain queue because the job is CPU bound rather than I/O bound.

16. Consider a set of "n" tasks with known burst times r1, r2, .., rn to be run on a uniprocessor machine. Which scheduling algorithm will result in the maximum throughput, i.e., minimum average turnaround time. Justify your answer.

The Shortest Job First (SJF) algorithm would be best as it would result in maximum throughput. This is because it would execute the shortest jobs first, making it so that more jobs are executed in a given window of time.

17. True/False: Shortest Remaining Time First scheduling may cause starvation. Justify your answer.

True, Shortest Remaing Time First (SRTF) may cause starvation. A long task can be continuously blocked by shorter tasks, blocking the execution of the longer task. This is known as starvation!

18. True/False: Preemptive Priority scheduling may cause starvation. Justify your answer.

True, Preemptive Priority Scheduling may cause starvation. This is because a task with a low priority can be continuously blocked by higher priority task, making it so that the low priority task cannot finish execution.

19. True/False: Round Robin scheduling is better than FCFS in terms of response time. Justify your answer.

True because in Round Robin (RR) each task gets access to the CPU little by little, but with FCFS everyone has to wait for their turn. So if an important task has 20 tasks ahead of it in FCFS, it will have poor response time compared to RR.

20. Consider the following 5 processes in a system.

Process	Arrival Time	Burst Time
P1	4	5
P2	6	4
P3	0	3
P4	6	2
P5	5	4

Draw a Gantt chart illustrating the execution of these processes using FCFS. Compute the average waiting time and average turnaround time.

Topic 6 - Process Synchronization

Process Synchronization

If a shared resource is being modified by more than one thread concurrently, then unexpected results may occur!

- The problem is that concurrent threads access a shared resource without any synchronization, and this creates a race condition (output depends on timing)
- We need mechanisms for controlling access to shared resources when working with concurrency
- Synchronization is necessary for any shared data structure

What resources are shared among threads?

- Local variables and **NOT** shared (data on the threads stack)
- Global variables are shared
- Dynamically allocated objects are shared
    - Stored in the heap

We want mutual exclusion when accessing shared resources

- When one thread is using a shared resource, other threads must be excluded from accessing the same resource until the first thread is done with it

The part of the code where the shared resource is accessed is called the **critical section**

- Only one thread can execute in the critical section at a time, all other threads are forced to wait on entry

Structure

<Entry Code>

Critical Section =>

<Exit Code>

Remainder Section

```
int deposit(account, amount){
        <Entry Code>
        balance = get_balance(account);
        balance += amount;
        put_balance(account, balance);
        <Exit Code>
        return balance;
} //end-deposit
```

Critical Section

- At most one thread is in the critical section at once
- If thread T is outside the critical section, then T cannot prevent thread S from enterting the critical section
- If thread T is waiting to enter the critical section, then T will eventually enter the critical section (no starvation)

Mechanisms for Synchronization in Concurrent Programs

Mutex Locks (spinlocks, blocking locks)

- A basic primitive for ensuring mutual exclusion in multi-threaded code

Semaphores

- A higher-level synchronization primitive used for both mutual exclusion and coordination between threads or processes

Mutex Locks

- A synchronization primitive used to protect critical sections (CS), ensuring only one thread can access a shared resource at a time with the following two operations:
    - acquire()/lock(): a thread calls this before entering the CS
    - release()/unlock(): a thread calls this after leaving the critical section
    - Threads must pair up these calls acquire() -> release() / **lock() -> unlock()**

Example:

```
int lock = 0; // unlocked
int deposit(account, amount){
      acquire(lock);
      balance = get_balance(account);
      balance += amount;
      put_balance(account, balance);
      release(lock);
      return balance;
}
```

Spinlocks: Pros & Cons

Pros:

- Can be implemented in user-space (no need to trap to the kernel)
- Useful in multi-processor environments

Cons:

- Horribly wasteful in single CPU environment
- If a thread is spinning on a lock, it is simply wasting CPU cycles

Mutex Locks: Ideal Implementation

When acquire() is called:

- Spin for a short amount of time to get the lock
- If you get the lock during this time, enter the CS
- If you don't get the lock, trap to the kernel to release the CPU and sleep on the mutex's waiting queue

When release() is called:

- If there is a thread sleeping on the mutex's waiting queue, trap to the kernel and wake one of these threads
- Otherwise simply set the mutex to 0 (unlocked)

POSIX pthread_mutex functions work exactly this way!

Pthread Mutex Syntax

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

- **pthread_mutex_init(&mutex, NULL);**

int pthread_mutex_lock(pthread_mutex_t *mp); // function to lock

- **pthread_mutex_lock(&mutex);**

int pthread_mutex_unlock(pthread_mutex_t *mp); //function to unlock

- **pthread_mutex_unlock(&mutex);**

Recursive Locking

06-ProcessSynchronization/ex6.c

```c
pthread_mutex_t mutex;

void functionB() {
    pthread_mutex_lock(&mutex);
    printf("Inside functionB\n");
    pthread_mutex_unlock(&mutex);
}

void functionA() {
    pthread_mutex_lock(&mutex);
    printf("Inside functionA\n");
    functionB();  // Calls another
    pthread_mutex_unlock(&mutex);
}
```

Semaphores

- A synchronization primitive that is used for both mutual exclusion and coordination between threads or processes
- An integer variable that is manipulated atomically through two operations: wait(down, P) and post(up, V)
    - wait(semaphore): decrements the semaphore value
        - Blocks the thread if semaphore value is non-positive until becomes open
    - post(semaphore): increments the semaphore value
        - Unblocks a waiting thread (if any)

There are two types of semaphores

- Binary Semaphore (a.k.a mutex semaphore)
    - Takes the values of 0 or 1
    - Use as a blocking mutex lock: Initialize to 1 to allow mutual exclusion
    - Use for thread/process coordination: Initialize to 0 to act as a signaling mechanism. This use is NOT possible with mutex locks
- Counting Semaphore
    - Takes arbitrary values
    - Represents a resource with many units available
    - Counter is initialize to N, where N is the number of available units
    - Allows thread/processes to proceed as long as more units are available
    - Each wait() decrements the counter, and each post() increments it

Initialized as: sem_t sem;
Functions:
int sem_init(sem_t *sem, int pshared, unsinged int value); //assign initial value

- Initialize 0: sem_init(&sem, 0, 0);
- Initialize 1: sem_init(&sem, 0, 1);
- Initialize N: sem_init(&sem, 0 N); where N >= 0

int sem_wait(sem_t *sem);

int sem_post(sem_t *sem);

int sem_destroy(sem_t *sem);

Use Case 1: Mutual Exclusion

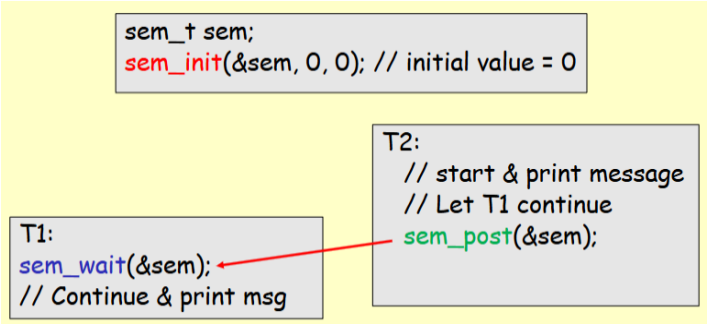
- Declare a binary semaphore & initialize it to 1
- Call sem_wait() before entering the critical section
- Call sem_post() when leaving the critical section

```
sem_t sem;
sem_init(&sem, 0, 1);
sem_wait(&sem);
//Critical section code
sem_post(&sem);
```

Use Case 2: Provide Coordination

Assume you have two threads T1 & T2

- We want T2 to run first & print a message
- Then we want T1 to continue execution & print a message
- To do this: declare a **binary semaphore** & initialize it to 0
- T1 calls sem_wait() to stop at the rendezvous point
- T2 calls sem_post() to let T1 continue



Use Case 3: Allow a certain number of threads inside the critical section

Assume you have an application where you have 3 resources of a certain kind

- You want to have a critical section where there could be up to three threads inside the CS at the same time
- 4th thread trying to enter the critical section must block at the entry until one of the threads inside the CS leaves
- To do this: create a counting semaphore with an initial value of 3
- Threads simply do sem_wait() at Entry, sem_post() at Exit
- This is the molecule example!

Deadlocks & Starvation

Semaphores can lead to deadlock if not used properly

Deadlocks - two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Starvation - indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended

Semaphores vs Mutex Locks

Aspect	Mutex Locks	Semaphores
Purpose	Mutual exclusion mechanism	Signaling mechanism
Counter Value	Always binary (unlocked or locked)	Can be 0, 1 (binary semaphore), or greater than 1 (counting semaphore)
Ownership	Owned by the thread that locks it	Not owned by a specific thread
Blocking	Lock operation blocks if already locked	Wait operation blocks if value = 0
Complexity	Simpler: single thread access only	Can manage multiple threads simultaneously with counting semaphores
Use Cases	Protecting critical sections	Synchronization and coordination
Deadlock Potential	Can occur if unlock is not managed properly	Can occur if used improperly

Practice Problems: Process Synchronization

1. What's a Critical Section? What is mutual exclusion? Write a simple code to show how you would achieve mutual exclusion in a critical section using a mutex lock.

The critical section is a part of the code where at most on thread may be executing at any given time. Mutual exclusion is when one thread is using a shared resource, all other threads must be **excluded** from using that same resource.

2. What is atomic instruction? In Intel architecture, what's the name of the atomic instruction used to implement a mutex lock?

An atomic instruction is one that **cannot** get interrupted. In intel architecture, the name of the atomic instruction used to implement a mutex lock is Compare & Change.

3. What's Test-and-Set instruction? What is it used for? Briefly describe.

Tests the value of a memory location and sets it to a new value in atomically. First loads 0 unlocked, then attempts to lock by setting to 1, but only if the tests passes and its 0, otherwise the lock was already set. It is used to implement locks and achieve mutual exclusion.

4. Briefly explain why spinlocks are not appropriate for single processor systems yet are often used in multiprocessor systems.

Constantly wasting CPU cycles so that a thread can check if another thread is done with a lock. With multiprocessor systems, you can make this checking happen on one core alone while have execution continue on another core.

5. What's race condition? Briefly explain.

A race condition is when multiple threads try to access the same resource at the same time. This causes the output to be nondeterministic and uncertain, as the results depends on timing of threads.

6. List 4 requirements of a critical section.

1. Mutual Exclusion
2. Progress (no dead locks)
3. No starvation (Bounded Waiting)
4. Performance (small overhead)

7. List 2 reasons why it is a bad idea to implement locks by disabling interrupts?

One reason is this will affect important system activities. Another reason is that in multi-core processors, threads running on another core would still be able to run, potentially breaking mutual exclusion.

8. What's a deadlock? How does it occur? Briefly explain.

If a thread fails to unlock a thread it creates a deadlock so all other processes are stuck in a waiting cycle and are starved since that process will never release

9. What's a condition variable. Why and where do you need them. Briefly explain.

A condition variable is a tool that allows threads to sleep until a certain condition is true. This is used in multi threaded applications to synchronize threads together, and this is important because sometimes a thread cannot continue until something else occurs.

10. Compare and contrast a mutex lock to a binary semaphore in terms of purpose and ownership.

With mutex's the thread has ownership over the lock, and with binary semaphores the thread does not have ownership over it. Both accomplish mutual exclusion, but mutex locks are safer because since there is no ownership in semaphores, another thread not in the CS can unlock the semaphore.

11. Is it possible for one thread to lock a mutex lock and another thread to unlock it? Why or why not. Briefly explain.

No

12. Is recursive locking possible with mutex locks? Is it possible with binary semaphores? Why or why not. Briefly explain.

Yes this is possible with mutex locks as long as the same thread that does the locking does the unlocking. This is not possible with binary semaphores because they do not have the same ownership properties.

13. What's the purpose of recursive locking and how do you do it? Briefly explain.

Recursive locking allows a thread that currently holds the lock to acquire it again without being blocked. Without recursive locking the thread would deadlock itself, which is why we have recursive locking! You do this by using a special type of lock called a recursive mutex lock, and the lock accomplishes this by keeping a count of how many times the lock was used and only releasing once the count goes back to 0 (unlock).

14. Briefly explain what priority inversion is and how unbounded priority inversion occurs.

15. Briefly explain how priority ceiling protocol solves the priority inversion problem.

16. Briefly explain how priority inheritance protocol solves the priority inversion problem.