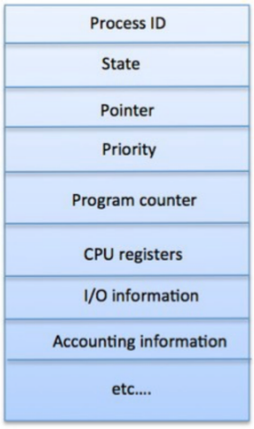
Processes / Threads

Each process has their own PCB, threads have different PC and CPU reg, and forks duplicate the parent PCB with their own PID.
Processes run in multiple memory spaces, and in memory, they have a text, stack, data and heap section.
The PCB is a data structure in the OS kernel containing the info needed to manage the scheduling of a process



Two methods of inter-process communication = Shared memory and messages back and forth

Thread is a path of execution within a process. Threads within the same process run in a shared memory space. They are not independant of one another so they share their code section, data section, and OS resources, but they have their own individual program counter, register set and stack space. Different types of threads: User-Level and Kernal-Level. Each thread belongs to only one process and a thread cannot exist outside of a process.

User-Level threads exist only at user-level. That is, all of the common thread operations (creation, deletion, synchronization) are implemented in user code, typically a user lib. The operating system is not involved at all, and, is completely oblivious to the existence of user-level threads. The benefit of user-level threads is a closer coupling of thread interaction.

Kernel-level threads are implemented in the OS kernel. System calls are necessary for thread creation, deletion and so on. Once created, kernel-level threads have similar interaction capabilities as user-level threads. A significant difference is how the threads execute. U-L share a single process, when the process blocks, none of the threads created as part of the process can execute. Threads supported by the kernel share the CPU. The OS maintained enough state for each thread in a set of threads such that if one thread blocks, the other threads can continue to execute.

USER LEVEL THREAD	KERNEL LEVEL THREAD
User thread are implemented by users.	kernel threads are implemented by OS.
OS doesn't recognized user level threads.	Kernel threads are recognized by OS.
Implementation of User threads is easy.	Implementation of Kernel thread is complicated.
Context switch time is less.	Context switch time is more.
Context switch requires no hardware support.	Hardware support is needed.
If one user level thread perform blocking operation then entire process will be blocked.	If one kernel thread perform blocking operation then another thread can continue execution.
Example : Java thread, POSIX threads.	Example : Window Solaris.

- write() - write to the console
- fork() - creates a new process by duplicating the calling process. The new process (the child) is an exact duplicate of the calling process (parent). The child has a unique PID, does not inherit the parent's memory locks or semaphore adjustments or outstanding asynchronous I/O operations for the parent. Return the PID of the child to the parent, and the child gets 0 if it is successful and -1 is returned to the parent if it fails (errno). Both child and parent are executable simultaneously.
- _exit() - exits the process "immediately" and any file descriptors belonging to that process are closed. This function does not return anything.
- exec() - replaces the current image with a new process image. It loads the program into the current process space and runs it from the entry point.
- wait() - waits for a single object. Returns the number of bytes written
- read() - reads a file. Returns the number of bytes read.
- clone() - duplicates the process into a child and the child inherits even more from the parent (like memory space) and returns the PID for child

*** *can pass a system call by 1) registers 2) address of parameter block and 3) push onto stack*

- Context Switching
  - The process of saving the context of one process and loading the context of another process is known as Context Switching aka like loading and unloading the process form running to ready state.
  - Happens when a high priority process comes to ready state, an interrupt occurs, user and kernel mode switch, or a preemptive CPU scheduling used.

3) Services from the OS:
- User interface, program execution, I/O operations
- memory management and file system
- interprocess communication and networking

2) What is the purpose of system programs?
- The purpose of system programs is to provide a level of abstraction to make it easier for user programs to access capabilities provided by the OS and the hardware. System programs are implemented to run in the OS and have a variety of functions. For example, there are system programs to provide information about processes or manage I/O devices.

1) What is the purpose of system calls?
- System calls allow program to invoke routines that will be executed by OS.
- often the case that more privileged mode required to perform certain operations.
- operations performed by system routine can involve other OS functionality, requiring certain degree of protection against user-level programming mistakes.
- some system calls deal with low-level interfaces and is necessary to provide users with a higher-level abstract interface (i.e., system call) to deal with the programming complexity.

- Advantages of Thread over Process:
  - Responsiveness
    - If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
  - Faster context switch
    - Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU
  - Effective utilization of multiprocessor system
    - If we have multiple threads in a single process, then we can schedule multiple threads on multiple processor. This makes process execution faster.
  - Resource sharing
    - Resources like code, data and files can be shared among all threads within a process (Each stack and register needs to be individual for each thread--they cannot be shared)
  - Communication
    - Communication between multiple threads is easier, as the threads shares common address space, while in processes we have to follow specific communication techniques
  - Enhanced throughput of the system
    - If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus the throughput of the system is increased.

Scheduling

- **Preemptive** = allocates the CPU to the processes for a limited amount of time. Preemptive allows a currently running "program"/process to be stopped, so that the OS can decide which of the current ready processes to execute. Aka a program does not necessarily execute to completion, but can be stopped and another process executed by the OS. This allows the OS to share the machine resources across multiple processes in such a way that they each can make progress. This can allow better efficiencies and performance in the execution of those processes collectively.

- **Non-preemptive** = allocates the CPU to the process until it finishes/terminates or switches to a different state. Makes the scheduling decision about a program at the beginning of the program's execution. Once that program starts executing, it cannot be stopped to schedule another (interrupts can still occur). So, another program cannot run until this program is complete.

Turnaround Time = (Completion Time) - (Arrival Time) / Completion Time = (Burst Time) + (Arrival Time)

Arrival Time = Will be given or need to determine based on the arrival times from the previous processes and add those together.

- **First Come First Serve** (FCFS)
  - Simplest algo. Schedules according to arrival times of processes. First come first serve algo process that requests the CPU first, is allocated first. It is implemented using the FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. FCFS = non-preemptive.

- **Shortest Job First** (SJF)
  - Provess which have the shortest burst time are scheduling first. If two processes have the same bust time, then FCFS is used to break the tie. Non-preemptive.

- **Longest Job First** (LJF)
  - It is simular to SJF but in this algo the priority is given to the process having the longest burst time. Starts off as non-preemptive but as the processes begin executing, it cannot be interrupted before it completes execution.

- **Shortest Remaining Time First** (SRTF)
  - Preemptive mode of SJF algo which jobs are schedule according to shortest remaining time.

- **Longest Remaining Time First** (LRTF)
  - Preemptive mode of LJF algo which jobs are schedule according to longest remaining time.

- **Round Robin Scheduling**
  - Each process is assigned a fixed time (time slice/time quantum -- think proj1) in a cyclic way. It is designed especially for the time-sharing system. The ready queue is like a circular queue. The CPU scheduler goes around the ready queue, allocating the CPI to each process for a time interval of us to 1-time quantum. To implement the RR, we keep track of the ready queue as a FIFO queue of the processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1-time slice and dispatches the process. The process may have a CPI burst of less than 1 time slice and in this case, the process itself will release the CPU voluntarily, and then the scheduler will then proceed to the next process in the ready queue. Otherwise, the time will go off for that process which will cause an interrupt to the operating system. A context switch (switching of the processes) will be executed, and the process will be put at the tail of the ready queue to wait its next turn.

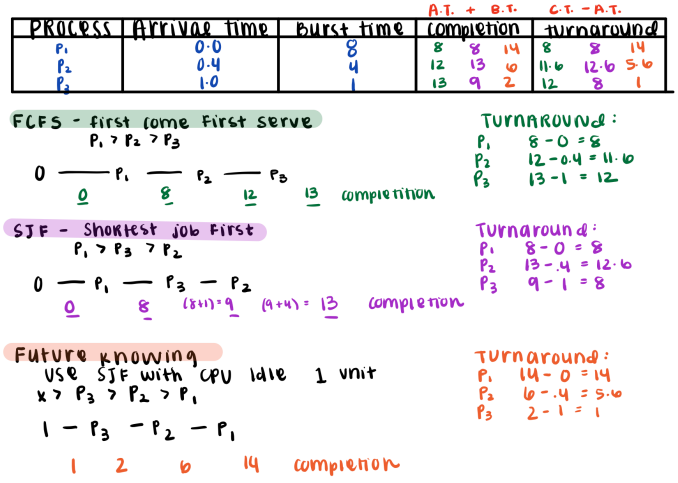
- **Priority Based Scheduling**
  - Processes are scheduled according to their priorities (highest priority processes is scheduled first). If priorities of two processes match, then schedule according to time. Starvation is possible with this algo

- **Highest Response Ratio Next** (HRRN)
  - Process with the highest response ratio is scheduled. This algo avoids starvation.

- **Multilevel Queue Scheduling**
  - According to the priority of process, processes are placed in the different queues. Generally high priority process are placed in the top level queue. Only after completion of processes from top level queue, lower level queued processes are scheduled. It can suffer from starvation

- **Multilevel Feedback Queue Scheduling**
  - It allows the process to move in between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a processes uses too much CPU time, it is moved to a lower-priority queue.

***Recall**: Atomic operations in concurrent programming are program operations that run completely independently of any other processes.*



- CPU utilization vs. Response time
  - CPU utilization increases if context switching overheads are minimized, and the context switching could be lowered if they perform less frequently, but if they are less frequent, then the response time increases (not as responsive).

- Average Turnaround Time vs. Max Waiting time
  - Avg. TAT is minimized by executing the shortest tasks first, but then this could starve the tasks with that are longer, so this would increase their wait time

- I/O dev$ice utilization vs. CPU utilization
  - CPU utilization is maximized by running lengthly CPU bound tasks without performing any context switches. I/O device utilization is maximized by scheduling I/O bound jobs as soon as they become ready by having more context switching

1) Suppose an algo favors those processes that have used the least processor time in the recent past. Why will this algo favor I/O bound programs and yet not permanently starve CPU-bound programs?

- I/O bound program spend the majority of their total execution time doing I/O activity, and relatively less time executing on the CPU.
- CPU bound programs will tend to spend a relatively higher proportion of time on the processor, while I/O bound programs have a smaller proportion of time on the processor.
- Therefore, it is likely that I/O bound programs will result in smaller processor times in the recent past and will be favored b y the short term CPU scheduling algo
- The reason no starvation will occur permanently is because when I/O is occurring, the process doing I/O is not wanting to use the CPU because it is likely waiting for the I/O to complete, and this means the CPU bound processes will have a chance to run during this time.

Concurrency and Synchronization / Deadlocks

What is a **critical section**? A critical section is a group of instructions or a region of code that needs to be executed atomically (atomicity is unbreakability like an uninterrupted operation) such as accessing a resource. The problem needs to synchronize the different processes.

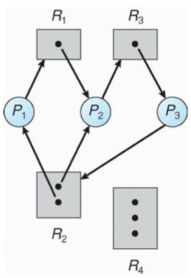
- What are solutions (3) to the critical section problem?
  - **Mutual exclusion** - Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.
  - **Progress** - Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.
  - **led waiting** - Bounded waiting means that each process must have a limited waiting time. It should not wait sly to access the critical section.

What is the multi-resource deadlock problem?
Deadlock can arise if four conditions hold simultaneously:
1) **Mutual exclusion** - only one process at a time can use a resource.
2) **Hold and wait** - a process holding at least one resource is waiting to acquire additional resources held by other processes
3) **No preemption** - a resource can be released only voluntarily by the process holding it, after that process has completed its task using the resource.
4) **Circular wait** - there exists a set {P1, P2, P3, ...} of waiting processes such that P1 is waiting for a resource that is held by P2, and P2 is waiting for a resource that is held by P3, ... , Pn-1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P1

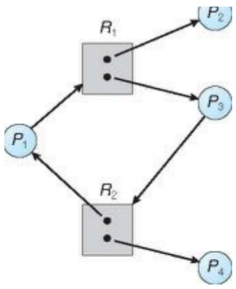
If a resource allocation graph contains no cycles => there can be no deadlock
If a resource allocation graph contains a cycle => There is only one instance per resource type = Deadlock, or if there are several instances per resource type = Possible Deadlock.

Difference between busy waiting and blocking:
Busy waiting - a process keeps testing for some condition. It is constantly using the CPU, sitting in a tight loop
Blocking - a process gives up the CPU and is awakened later when the condition that is being waited for has become true

Deadlock



No Deadlock



Deadlock avoidance - Requires that the system has some additional *pre*-information available. Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. The deadlock-avoidance algorithm dynamically examines the resource allocation state to ensure that there can never be a circular-wait condition. Resource allocation state is defined by the number of available and allocated resources, and the maximum possible demands of the processes. Avoidance achieved by ensuring that system will never enter an unsafe state

If a system is in a **safe state** -> no deadlocks
A state is **safe** if the system can allocate all resources requested by all processes without entering a deadlock state.
If a system is in **unsafe state** -> possibility of deadlock (no guaranteed)

Avoidance algorithms: Avoidance algorithms prevent deadlocks from ever happening (never allowing an unsafe state to be entered)
Single instances of a resource type - Use a resource allocation graph to evaluate
Multiple instances of a resource type - Must run an algo on the resource allocation graph
Banker's algorithm
Requirements:
Each process must *a priori* claim maximum use
When a process requests a resource it may have to wait
When a process gets all its resources it must return them in a finite amount of time
This algo is a bookkeeping method for tracking and assigning resources
Let n = number of processes, and m = number of resource types
Available - Vector length of m, if Available[j] = k, there are k instances of resource type Rj available
Max - n x m matrix. If Max[i, j].= k, then process Pi may request at most k instances of resource type Rj
Allocation - n x m matrix. If Allocation[i, j] = k then Pi is currently allocated k instances of Rj
Need - n x m matrix. Need[i, j] = k, then Pi may need k more instances of Rj to complete its task
Need[i, j] = Max[i, j] - Allocation[i, j]

Deadlock prevention - Idea is to restrain the ways request can be made
Mutual exclusion -
-Not required for sharable resources
-Must hold for non-sharable resources

Hold and wait -
-Must guarantee that whenever a process requests a resource, it does not hold any other resources (strict)
-Requires process to request and be allocated ALL of its resources before it begins execution
-Allows process to request resources only when the process has none allocated to it
-Low resource utilization and starvation is possible

No preemption -
-If a process that is holding some resources request another resource that cannot be immediately allocated to it, then all resources currently being held by that process are released
-Preempt resources are added to the list of resources for which the process is waiting
-Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Circular wait -
-Impose a total ordering of all resource types, and require that each process requests resources in an including order of enumeration

- Know about synchronization constructs and how they are used:
  - Synchronization constructs are used to ensure the consistency of shared data and to coordinate parallel execution among threads.
  - **Mutex**
    - Mutual exclusion object that synchronizes access to a resource. Makes sure that one thread can acquire the Mutex as a time to enter the critical section. This thread only releases the Mutex when it exits the critical section.
    - A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa. At any point of time, only one thread can work with the entire buffer. The concept can be generalized using semaphore.
    - Mutex is a locking mechanism used to synchronize access to a resource. Only one task can acquire the mutex. There is ownership associated with mutex, and only one owner can release the lock (mutex).
    - A thread can need more than one mutex (lock) since it can need more than one resource. If any lock is not available then the thread will wait/block on the lock.
    - A mutex can be locked or unlocked, and a recursive mutex can be locked more than once and a count is needed to be associated with it to keep track of how many times it is locked/unlocked. The programmer will need to make sure to unlock the mutex as many times as it is locked to ensure this works properly. If a mutex that is not a recursive mutex is locked more than once, then deadlock occurs because it enters a waiting list of that mutex, and this happens because no other thread can unlock the mutex.
    - The end goal for a mutex is atomic access

Example :

```
wait (mutex);
...
Critical Section
...
signal (mutex);
```

```
void barrier() {
  wait(barrier_sem); //decrement semaphore to 0 and run if first thread, if not, wait for previous thre
ad to signal
  count = count + 1;
  if (count == N)
    then {
      count = 0;
      signal(barrier_sem); // reset the semaphore to initial value
  } else {
    signal(barrier_sem); // indicate this thread has reached the barrier, signal next thread to execute
    while (count > 0) ;
  }
}
```

Is this the correct solution? If so I believe that it is reusable as all of the threads will be running after the barrier and the semaphore and count values will be 1 and 0 respectively.

  - **Semaphore** (binary and counting)
    - A semaphore is a generalized mutex.
    - Semaphore is a signaling mechanism ("I am done, so you can carry on" kind of signal) and a thread that is waiting on a semaphore can be signaled by another thread. This is different than a mutex as the mutex can be signaled only by the thread that called the wait function.
    - A semaphore uses two atomic operations, wait and signal for process synchronization.
    - The wait operation decrements the value of its argument S, it if is positive. If S is negative or zero, then no operation is performed.

```
wait(S)
{
    while (S<=0);
    S--;
}
```

The signal operation increments the value of its argument S

```
signal(S)
{

S++;
}
```

    - There are only two types of semaphores: counting and binary
      - Counting Semaphores - integer value semaphores and have unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. Think of it like a bouncer at a club, once it becomes full, sends a signal that no one else is let in, but as people leave, it is in charge of execute what happens next/who is next let in
      - Binary Semaphores - are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0.

*** *Mutex is different from a semaphore* *** Mutex = locking mechanism while a semaphore is a signaling mechanism. A binary semaphore can be used as a Mutex but a Mutex can never be used as a semaphore.

  - **Condition variables** (as in monitors)
    - Condition variables are what waits for threads.
    - User mode objects that cannot be shared across processes
    - Enable threads to automatically release a lock and enter a sleeping state

Dining philosopher Solutions:
- Create a rule to avoid deadlock = When the philosopher requests their chopstick, only accept (0) request if there are two chopsticks or more remaining after, deny if there are not (1) This works because as long as one philosopher is eating, deadlock is not possible because once they are done eating, they will return their chopsticks and it would be the next philosopher's turn. Starvation is possible.
- Semaphore solution: There are 3 states of philosopher: THINKING, HUNGRY and EATING. Here there are two semaphores: Mutex and a semaphore for the array for philosopher. Mutex is used such that no two philosopher may access the pickup or putdown at the same time. The array is used to control the behavior of each philosopher. (Note semaphores here can result in deadlock)

Deadlock's detection and recovery

Detection:

-Allow system to enter deadlock state

Detection algorithm determines if the resource allocation graph is in a deadlock state and if it is, a recovery scheme is used to get out of it.

Assume that it is possible to recover.

It's possible that this might require a rollback to a prior consistent state.

Single instance - run an algorithm to check for cycle in the Resource allocation graph. Cycle = deadlock.

Multiple instance - Detection of the cycle is necessary but not sufficient condition for deadlock detection, the system may or may not be in deadlock varies according to different situations

Recovery:

A traditional operating system doesn't deal with deadlock recovery as it is time and space consuming process. Real-time OS use recovery by killing all the process involved in the deadlock one by one.

After killing each process, check for deadlock again, keep repeating the process till system recovers from deadlock.

Resource preemption

Resources are preempted from the processes involved in the deadlock, preempted resources are allocated to other processes so that there is a possibility of recovering the system from deadlock. In this case, the system goes into starvation.

Interrupts:

When an interrupt occurs, the OS evaluates the type of interrupt from the interrupt table (it comes as a key) and finds the pointer and sees what is running and if it is possible to interrupt it.

Exception = TRAP = software interrupt

IRQ = interrupt request = hardware interrupt

Timer interrupts at a constant interval

Race conditions:

- They are undesirable so it is the reason we need semaphores.
- They try to perform two or more operations at the same time

which is bad bc things needs to be done in a specific sequence.

Assume working with deadlocks (according to the bankers algo)

Increase Available (more resources added) = Safely.

More resources means it will be easier for the processes to get their resources.

Decrease Available (resource permanently removed) = Not safe.

Increases the amount of unsafe conditions.

If this occurs at the beginning then make sure that the max # of instances = max requirement of a process, otherwise it will not start. If this occurs during execution, a possibility that the max condition will be violated.

Increase Max for one process (process needs or wants more resources than allowed) = Unsafe

If a process increases the max number of instances of a resources during execution, this too could violate the beginning condition that there are enough resources to meet maximum demands.

Decrease Max for one process (process decides it doesn't need that many resources) = Safe

Still meets the starting condition for max available resources

Increase the number of processes = Unsafe

If the number of max resources requires (including the newly added processes) is within the max resources provided, this change would be ok. However, if that isn't true, it might enter unsafe state

Decrease number of processes = Safe

Because decreasing the number of processes reduces the total maximum demand

Explain the difference between multiprogramming, multitasking and multiprocessing

Multiprogramming is a term used to mean an ability to run multiple programs (jobs) at the same time on a computer system.

The term continues with respect to the "degree of multiprogramming" and how program-level resources (e.g., amount of memory, amount of disk, and so on) are allocated to the job. These are decisions that are made at the time of "job" entry (i.e., when a program is actually loaded and begun), including whether there are enough resources available to start the job at all. (**Multitasking** is what is actually doing the switching of the processes, it is an extension of M.P.)

Multiprocessing refers to the OS's ability to manage multiple processes running on the computer system at the same time. The main focus is on process-level support and mechanisms for handling process state transitions, sharing of I/O devices, and so on. Here there is more concern on making sure the computer system resources can be used correctly and effectively given the processes that are current running on the machine.