## 2 Process Management

- **Process**
  Process is an instance of a program in execution
  A program is passive; a process active.
  Multiple instances of the same program are different processes
  Process has three main components. You can see the activeness of a process with respect to these three components.

  1. Address space (Figure 6) The memory that the process can access.
     Consists of various pieces: the program code, static variables, heap, stack, etc.

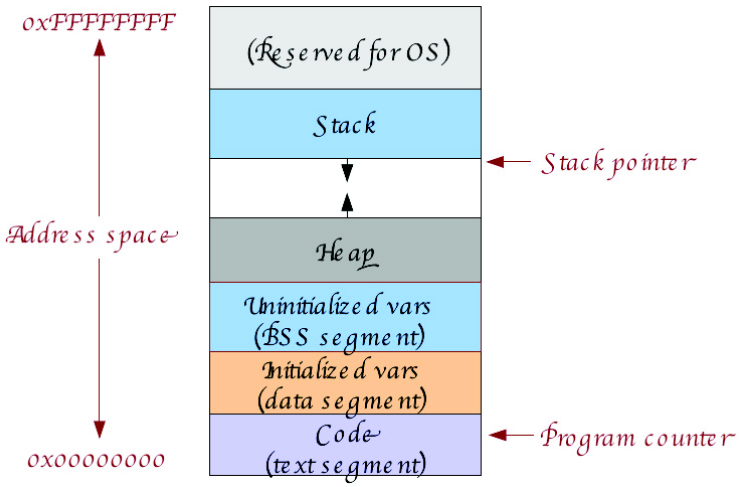



**Figure 6. Address space**

     The address space is how the process sees its own memory. Not necessarily how the memory is laid out in physical RAM.

  2. CPU state
     The CPU registers associated with the running process Includes general purpose registers, program counter, stack pointer, etc.
     We can see an example where the CPU state is used. Let take there are two process P1 and P2. Both the processes are using a statement, i.e. "a=b+c". This statement can be divided into three instructions, i.e. "mov b AX; add ax c; mov AX a;". Although both the processes using the same statement, still the memory required for a, b , and c are different from each other. This is because these variables are stored corresponding to their process address space. But the registers which are used in both the process are belongs to CPU. So, whenever CPU switches from one process to another process all the registers information is pushed into the stack area of the process address space. Similarly, when the process again resumes, the register information stored in the stack is popped and stored in the actual CPU registers.

  3. OS resources
     Various OS state associated with the process Examples: open files, network sockets, etc.

  OS represents each process using **Process Control Block(PCB)**. One PCB per process. OS maintains a list of PCB's for all processes.

- **Contents of PCB**

- Pointer to next process
- Process id
- parent process id
- Process state
- CPU state: CPU register contents
- Scheduling info (Priority info, pointer to scheduling queue)
- Memory info (Pointers to different memory areas)
- Open file information
- Accounting info(CPU time,...)

- **How to create a process?**

  Give an example of fork() system call. And also show the PID and PPID. Each process identified by a unique, positive integer id (process id).
  As a process executes, it changes its state.
  When fork is invoked, child is an exact copy of parent. When fork is called all pages are shared between parent and child, just by copying the parent's page tables.
  But, when data in any of the shared pages change, OS intercepts and makes a copy of the page. Thus, parent and child will have different copies of this page.
  This is because, a large portion of executables are not used. Copying each page from parent and child would incur significant disk swapping. So, Postpone coping of pages as much as possible will optimizing the performance.

- **Process States (5-state model)**(Figure 7)
  **new**: The process is being created
  **ready**: The process is waiting to be assigned to a CPU
  **running**: Instructions are being executed
  **waiting**: The process is waiting for some event (needed for its progress) to occur
  **terminated**: The process has finished execution

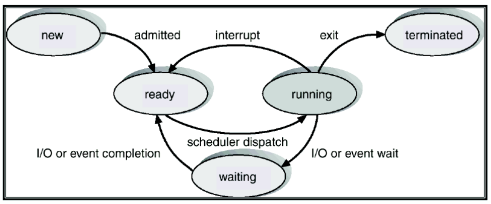



**Figure 7. Process state**

- Process scheduling

  CPU is allotted to the process, and process runs. (Figure 9)

  A newly arrived process is put in the ready queue. Processes waits in ready queue for allocating the CPU. Once the CPU is assigned to a process, then that process will execute. While executing the process, any one of the following events can occur.

  - The process could issue an I/O request and then it would be placed in an I/O queue.
  - The process could create new sub process and will wait for its termination.
  - The process could be removed forcibly from the CPU, as a result of interrupt and put back in the ready queue.

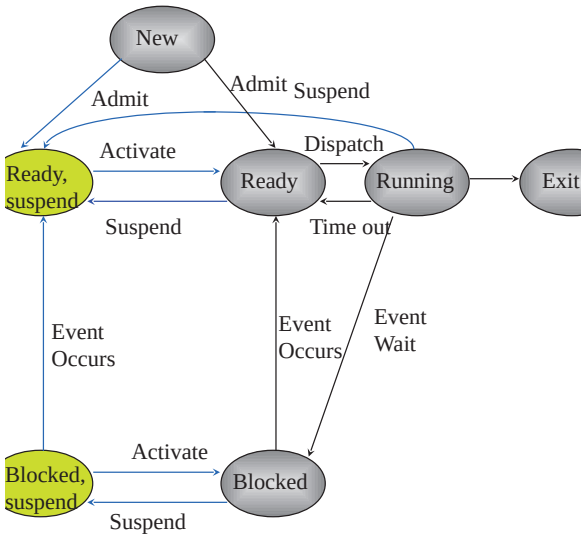



**Figure 8. Process state**

The process scheduling needs two components.

- Scheduling queues : The scheduling queues are basically of two types.(Figure 10)
  - **Ready queue** = contains those processes that are ready to run. Those process are present in the main memory.
  - **I/O queue (waiting state )** = holds those processes waiting for I/O service.
    (In addition to that system maintains a **job queue** which keeps all the PCBs present in the system)

- Scheduler : Taking the responsibility of switches process (PCB) from one queue to another. It also picks a process from the ready queue according to some scheduling policy and assign it to CPU.
  - Long term scheduler
    - It is also called job scheduler
    - Selects which processes should be brought into the ready queue
    - It is used to move the process from new to ready state and from waiting to ready state.
    - The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. If the average rate of number of processes arriving in memory is equal to that of departing the system then the long term scheduler may need to be invoked only when a process departs the system. It may also be very important that long term scheduler should take a careful selection of processes i.e. processes should be combination of CPU and I/O bound types. If all processes are I/O bound, the ready queue will always be empty and the short term scheduler will have nothing to do. If all processes are CPU bound. no process will be waiting for I/O operation and again the system will be unbalanced. Therefore, the long term scheduler provides good performance by selecting combination of CPU bound and I/O bound process.
    - Controls the degree of multiprogramming (no. of jobs in memory)
    - Invoked infrequently (seconds, minutes)
  - Short term scheduler
    - It is also called CPU scheduler
    - Selects which process should be executed next and allocates CPU (moves process from ready state to running state)

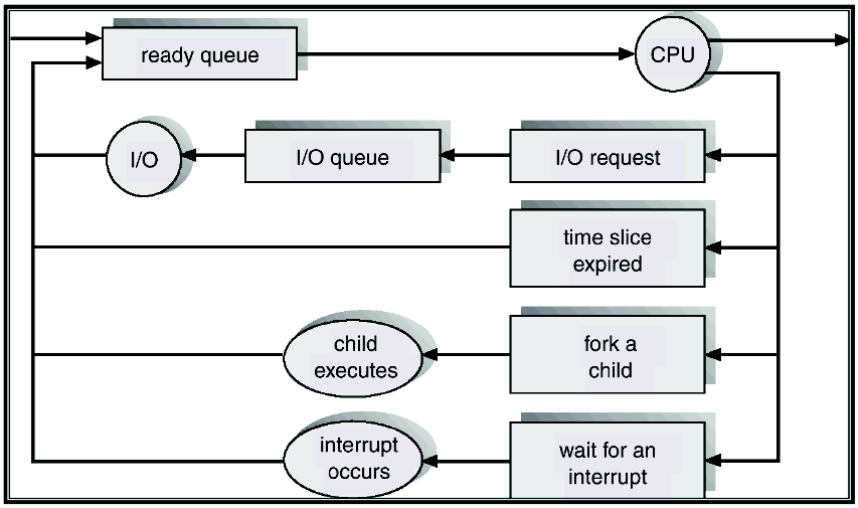



**Figure 9. Scheduling**

  - Invoked very frequently (milliseconds), must be fast
* Medium term scheduler(Figure 11)
  - What if all processes do not fit in memory(with respect to ready queue)
  - Copy the process image to some pre-designated area in the disk (swap out). Bring in again later and add to ready queue later when more memory is available. Temporarily removes processes from main memory and places them on secondary memory (such as a disk drive) or vice versa.
  - Allows the long-term scheduler to admit more processes than actually fit in memory but too many processes can increase disk activity (paging), so there is some "optimum" level of multi-programming.
  - Conditions for medium term scheduler
    1. Running process may become suspended if it makes an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out.
    2. a process which has not been active for some time
    3. a process which has a low priority
    4. a process which is page faulting frequently
    5. a process which is taking up a large amount of memory in order to free up main memory for other processes

- How does the scheduler gets scheduled?(Suppose we have only one CPU)
  - As part of execution of an ISR (ex. timer interrupt in a time-sharing system)
  - Called directly by an I/O routine/event handler after blocking the process making the I/O or event request
- Context switching

  This is a mechanism to change the control of the CPU from one process to another process.(Figure 12)

  Context-switch time is an overhead; the system does not do any useful work while switching.

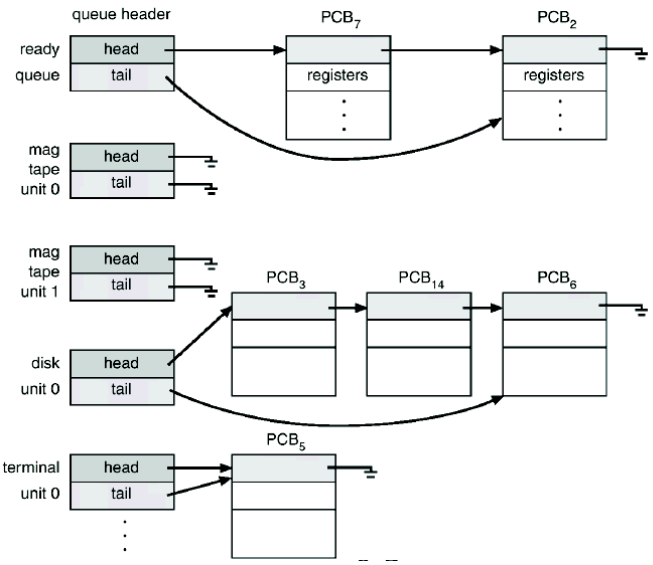



**Figure 10. Scheduling queue**

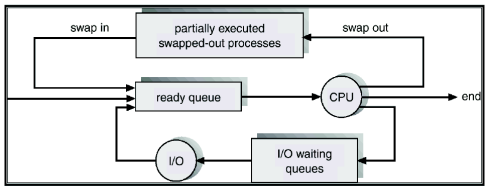



**Figure 11. Medium term scheduler**

## 3 Thread

- Why do we need a thread?
  Creating a new process is often very expensive.
  Processes don't (directly) share memory. Because each process has its own address space.

- What is a thread? (Figure 13)
  A thread is a light weight process. A thread defines a single sequential execution stream within a process (PC, stack, registers).
  Each thread has its own stack, PC, CPU registers, etc.
  All threads within a process share the same address space and OS resources.
  Threads share memory. They can communicate directly.

  Each thread has a thread control block (TCB). TCB contains processor state(Registers), thread state, and pointer to corresponding PCB.
  Context switch between two threads in the same process does not need to change address space.
  Context switch between two threads in different processes must change address space.

- Thread system classification (Figure 14)

- Different types of thread and their functionality.
  Basically there are two types :

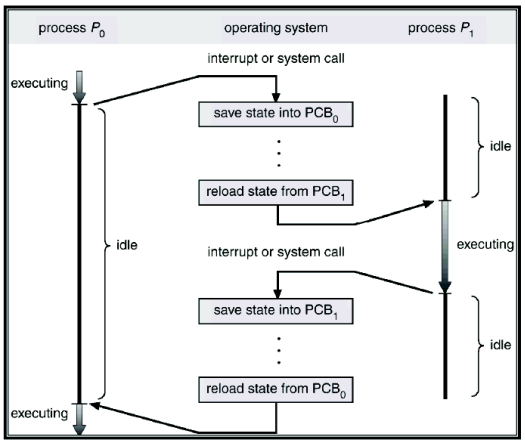



**Figure 12. Context switching**

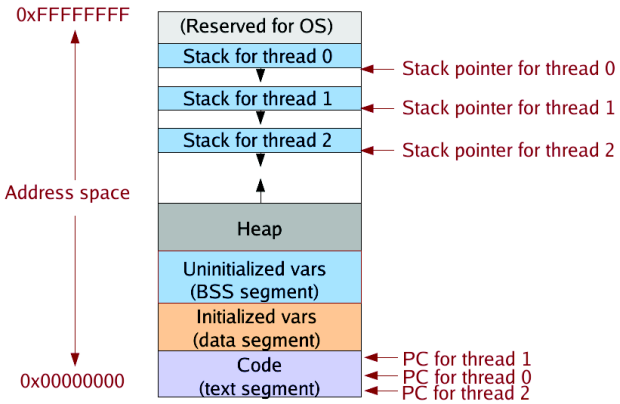



**Figure 13. Thread**

  - user level thread
  - kernel level thread.

- User level thread (Figure 15)
  Some OS only supports user level thread. In this case, OS does not need to know anything about multiple threads in a process. So, managing multiple threads only requires switching the CPU state (PC, registers, etc.). And this can be done directly by a user program without the help of OS. The programmer uses a **thread library** to manage threads.

  - No context switching is needed.
  - Each process has one or more threads.
  - Each process may maintains a ready queue.
  - User level threads are more faster than other level threads.
  - OS finds all the treads belongs to one process is a single one.

  - The main disadvantages of user level thread are:

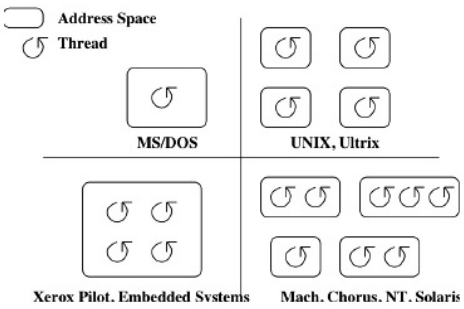



**Figure 14. Thread system classification**

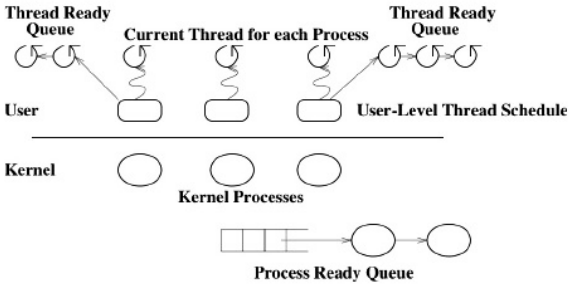



**Figure 15. User level thread**

  - ∗ if one thread wants for any I/O, then all the threads belongs to same process will automatically blocked.
  - ∗ Since the OS does not know about the existence of the user-level threads, it may make poor scheduling decisions.
- How to create a user-level thread? (Figure 16)
  Thread library maintains a TCB for each thread in the application, just a linked list. Allocate a separate stack for each thread (usually with malloc).

- Kernel level thread
  - System call is used to create kernel thread.
  - A kernel thread is also known as a lightweight process. The kernel does thread creation, termination, joining, and scheduling in kernel space.
  - Kernel threads are usually slower than user threads due to system overhead.
  - Switching between kernel threads of the same process requires a small context switch.
    - ∗ The values of registers, program counter, and stack pointer must be changed.
    - ∗ Memory management information does not need to be changed since the threads share an address space.
  - The kernel must manage and schedule threads (as well as processes), but it can use the same process scheduling algorithms.
  - Switching between kernel threads is slightly faster than switching between processes.
  - In a multiprocessor environment, the kernel may schedule threads on different processors.

- Thread model (Figure 17)

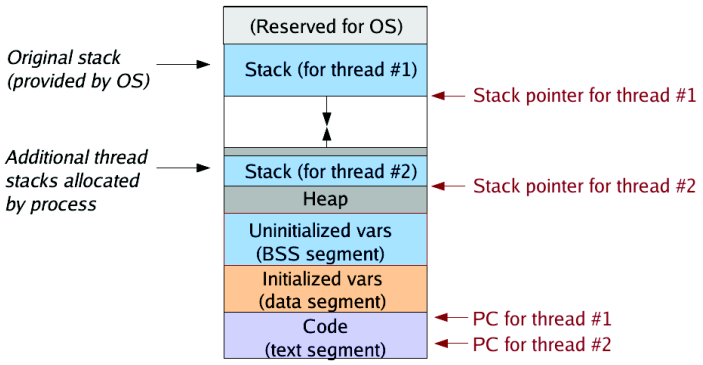
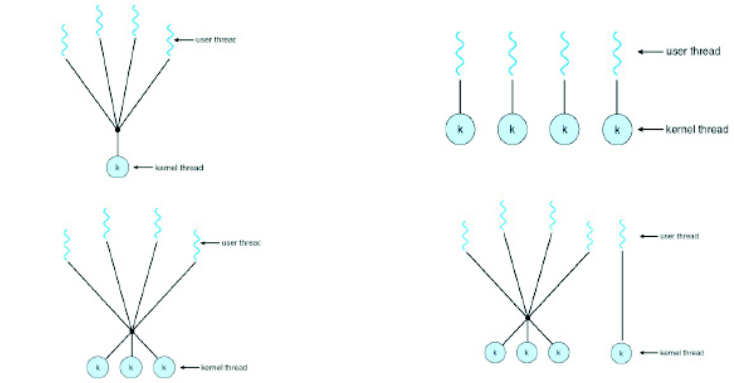



**Figure 16. User Level Thread**




Many-to-one, one-to-one, many-to-many and two-level

**Figure 17. Thread model**

## 4 CPU scheduling

- What is CPU scheduling?
  - Determining which processes run when there are multiple runnable processes.
  - As a whole we can say that there is a pool of runnable processes contending for the CPU and compete for resources. The job of the scheduler is to distribute the scarce resource of the CPU to the different processes "fairly" in such a way that the overall performance will be maximized.
- What is it goal?
  - The overall performance of the system should be maximum.
    - CPU utilization, throughput, etc. should be maximum.
- How do processes behave?
  - A process will run for a while (**CPU burst**), perform some IO (**IO burst**), then run for a while more (the next CPU burst). These things will continue till the completion of the process.
- Types of Process

  - IO Bound processes:
    * processes that perform lots of IO operations. Each IO operation is followed by a short CPU burst to process the IO.
  - CPU bound processes:
    * processes that perform lots of computation and do little IO.

- When do scheduling decisions take place? / When does CPU choose which process to run?

  1. When process switches from running to waiting.
     - IO request
     - wait for child to terminate
     - wait for synchronization operation to complete
  2. When process switches from running to ready
     - completion of interrupt handler
       * Ex: timer interrupt in time sharing / interactive systems. If scheduler switches processes in this case, it has preempted the running process. Another common case interrupt handler is the IO completion handler.
  3. When process switches from waiting to ready state
     - Ex: (on completion of IO or acquisition of a lock, for example).
  4. When a process terminates.

  Scheduling done only under 1 and 4 condition is **non-preemptive**. All other scheduling is **preemptive**.

- **Dispatcher**

  Dispatcher module gives control of the CPU to the process selected by the scheduler; this involves:

  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

  **Dispatch latency** - time it takes for the dispatcher to stop one process and start

- How to evaluate scheduling algorithm?
  There are many scheduling criteria:

  - **CPU Utilization :** The percentage of time that the CPU is busy.
  - **Throughput :** The number of processes completing per unit of time.
  - **Turnaround time :** The interval from the time of submission of a process to the time of its completion.
  - **Waiting time :** The total amount of time that a process is in the ready queue.
  - **Response time :** The time between the submission of a request and the first response to the request.

- Scheduling Algorithm Goals

  - Maximize CPU utilization and throughput
  - Minimize turnaround time, waiting time, response time

- Scheduling Algorithm

  - First Come First Serve (FCFS)

    - Maintains one ready queue. OS runs the process at head of queue, new processes come in at the end of the queue.
    - A process does not give up CPU until it either terminates or performs IO. (non-preemptive)
    - **Problem**: Average waiting time is more (If initial processes have more CPU burst)
- Shortest-Job-First (SJF)
    - Process has less CPU burst will execute first.
    - Eliminate Waiting and Turnaround time.
    - This algorithm is optimal with respect to average waiting time.
    - **Problem**: how does scheduler figure out how long will it take the process to run?
        - Scheduler, must use the past to predict the future. We can use weighted average technique.
          $s_{n+1} = w\ T_n + (1 - w)s_n$
          , where
          $\mathbf{s}_{n+1}$ be the predicted value of next CPU burst
          **w** is the weightage, 0 <= w <= 1
          $\mathbf{T}_n$ be the measured burst time of the $n^{th}$ burst.
          $\mathbf{s}_0$ is defined as some default constant or system average.
    - Preemptive vs. Non-preemptive SJF scheduler.
        - Preemptive SJF: If the new process has less CPU burst time as compare to remaining CPU burst time of the current process then the CPU preempts the running process and executes the new process.
        - Non-Preemptive SJF: Scheduling decision only takes place when running process voluntarily gives up CPU. In effect, it allows every running process to finish its CPU burst.
- Priority Scheduling
    - Each process is given a priority, then CPU executes process with highest priority.
    - Problem: **starvation** or blocking of low-priority processes.
        - Can use **aging** to prevent starvation - As the time progress, increase the priority of a process which stays at ready state but not run.
- Round-robin scheduling
    - Give response to users in a reasonable time.
    - Similar to FCFS but with preemption.
    - Have a time quantum or time slice. Implementing round-robin requires timer interrupts.
    - Let the first process in the queue run until it expires its quantum (i.e. runs for as long as the time quantum), then run the next process in the queue.
    - It gives good response time, but can give bad waiting time.
    - Problem: with a small quantum - context switch overhead.
- Multilevel Queue Scheduling
    - Classify processes into separate categories and give a queue to each category.
- Multilevel Feedback Queue Scheduling
    - Like multilevel scheduling, except processes can move between queues as their priority changes.
    - Can be used to give IO bound and interactive processes CPU priority over CPU bound processes.
    - Can also prevent starvation by increasing the priority of processes that have been idle for a long time.