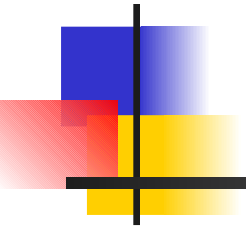
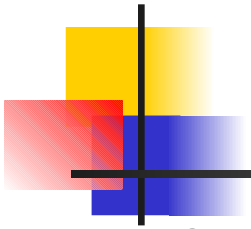
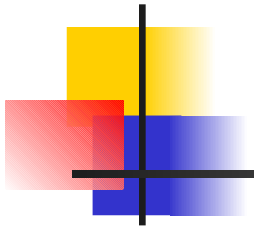


Scheduling Strategies



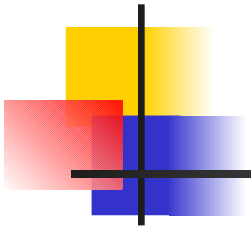
Introduction

- Scheduling is used to determine which process is given control over the CPU. It can be divided into three categories:
  - long term scheduler (or **job scheduler**) – determines which jobs or processes may compete for systems resources. The main objective is to provide the medium term scheduler with the appropriate number of jobs. Too few jobs and the CPU sits idle, too many jobs and system efficiency may be degraded.
  - medium term scheduler (or **swapper**) – swaps processes in and out of memory. This includes the memory management system that will be presented in another lecture.
  - short-term scheduler (or **dispatcher**) – allocates the CPU to a process that is loaded into main memory and ready to run. Typically the dispatcher allocates the CPU for a fixed maximum amount of time. A process releases the CPU after this time and returns to the pool of processes from which the dispatcher selects the process to execute. On multithreaded systems the short-term scheduler can also schedule threads. Thread scheduling occurs within process scheduling, as threads are finer-grained processes.



Process states

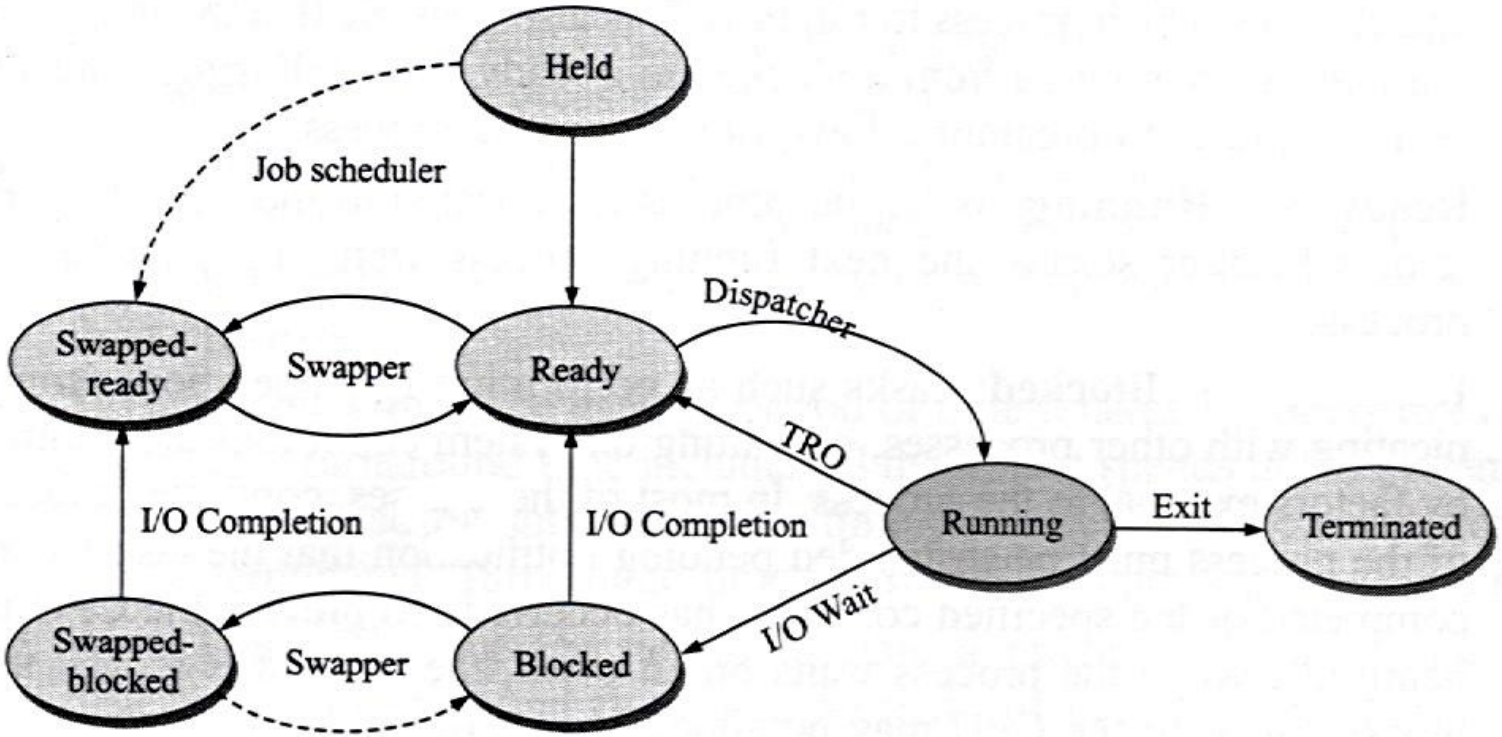
- According to the short-term scheduler, a process can exists in one of the following four states:
  - **blocked**: a process that is waiting for some event to occur before it can continue to execute. Most frequently, this event is completion of an I/O operating. Blocked processes to not require the services of a CPU as their execution cannot proceed until the blocking event completes. Within rt.cpp/ rt.h a process can be explicitly blocked by suspending it, i.e. p1.Suspend().
  - **ready**: a process that is not allocated to a CPU but is ready to run. It will execute as soon as it is allocated by the dispatcher.
  - **running**: a process that is currently executing on a CPU. In a multiprocessor system if the system has *n* CPUs that at most *n* processes can be in the running state.
  - **terminated**: a process that has halted its execution but a record of the process still remains is still maintained by the operating system. In UNIX these are referred to as a zombie process. The OS retains information on a terminated processes for a variety of reasons, e.g. and I/O operating pending completion.

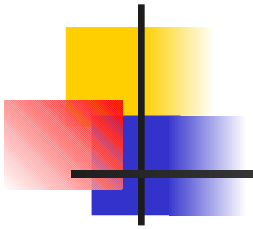


Process states – medium/long term scheduler

- The medium term scheduler adds two additional process states
  - **swapped-blocked**: a process that is waiting for some event to occur and has been swapped out to secondary storage
  - **swapped-ready**: a process that is ready to run but is swapped out to secondary storage. A swapped-ready process may not be allocated to the CPU since it is not in main memory.
- The long-term scheduler adds the following state:
  - **held**: a process that has been created but will not be considered for loading into memory or for execution. Typically only new processes can become a held processes. Once a process leaves the held state, it does not return to it.

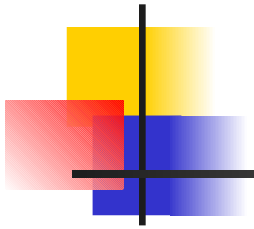
Seven state model and state transitions





Scheduling criteria I

- The goal is to identify the process whose selection will result in the best possible system performance.
- Performance measurement is subjective and comprises a mixed number of criteria of varying importance.
- The scheduling policy determines the importance of each of the criteria.
- Some of these criteria include:
  - **CPU utilisation:** the % of time that a CPU is executing a process. The load on the system affects the level of utilisation that can be achieved. High utilisation is more easily achieved on a heavily loaded system. CPU utilisation is more important on large, time-shared, expensive systems rather than single-user desktop systems.
  - **balanced utilisation:** % of time all resources are utilised. Instead of just CPU utilisation, utilisation of memory, I/O devices, and other system resources are also considered.
  - **throughput:** the number of processes the system can execute in a period of time. Evaluation of throughput must consider the average length of a process. On systems with long processes, throughput will be less than on systems with short processes.



Scheduling criteria II

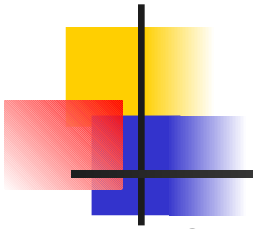
- ...cont
  - **turnaround time**: the average period of time it takes a process to execute. This includes all the time it spends on the system and can be computed by subtracting the time the process was created from the time it terminated. Turnaround time is inversely proportional to throughput.
  - **wait time**: the average period of time a process spends waiting.
  - **response time**: on interactive systems, the average time it takes the system to start responding to user inputs.
  - **predictability**: lack of variability in other measures of performance
  - **fairness**: the degree to which all processes are given equal opportunities to execute. In particular, do processes should not be subject to starvation.
  - **priorities**: give preferential treatment to processes with higher priorities.

Scheduling algorithms - FCFS

- First come first served (FCFS): This is the simplest algorithm. Jobs are scheduled in the order that they are received.
- FCFS is non-preemptive.
- Implementation is easy accomplished by implementing a queue of the processes to be scheduled or by storing the time the process was received and selecting the process with the earliest time. FCFS tend to favour CPU-bound processes.
- Consider a system with a CPU-bound process and a number of I/O-bound processes.
  - The I/O bound processes will tend to execute briefly, then block for I/O. A CPU-bound process in the ready state should not have to wait long before being made runnable. The system will frequently find itself with all the I/O bound processes blocked and the CPU-bound processes running.
  - As the I/O operations completes, the ready queue fills up with the I/O bound processes. Their wait time increases while I/O device utilisation plummets. Under some circumstances, CPU utilisation can also suffer.
  - In the situation described above, once a CPU-bound process does issue an I/O request, the CPU can return to process all the I/O bound processes. If their processing completes before the CPU-bound process's I/O completes, the CPU sits idle. An algorithm that provides a better mix of CPU and I/O bound processing usually performs better than FCFS.

Scheduling algorithms - SJF

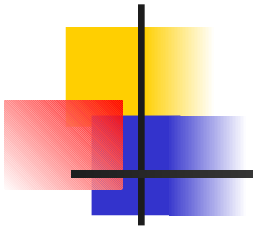
- Shortest Job First (SJF) is also a non-preemptive algorithm.
- It selects the job with the shortest expected processing time. In case of a tie, FCFS can be used. SJF was originally implemented for batch processing as it relies on a time estimate supplied with the batch job.
- SJF has been modified for short-term scheduling and used in an interactive environment. The algorithm bases its decision on the expected processor burst time. Expected burst times are computed as a simple average of the process's previous burst times or as an exponential average. #
- The exponential average for a burst can be computed using the equation
  - $e_{n+1} = at_n + (1 - a)\, e_n$
  - Where $\mathbf{e_n}$ presents the *n*th expected burst, $\mathbf{t_n}$ represents the length of the burst at time *n*, and **a** is a constant that controls the relative weight given to the most recent burst (if a=0, the most recent burst is ignored; with a=1 the most recent burst is the only one considered).
- In extreme cases SJF may cause starvation if there is a constant arrival of small jobs.



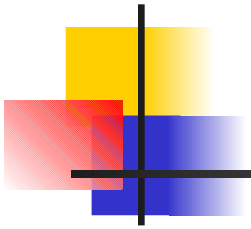
Scheduling algorithms - SRT

- Shortest Remaining Time (SRT)
- SRT is a preemptive version of SJF.
- Any time a new process enters the pool of processes to be scheduled, the scheduler compares the expected value for its remaining processing time with that of the process currently scheduled.
- If the new process's time is less, the currently scheduled process is preempted.
- Like SJF, SRT favours short jobs, and long jobs can be a victim of starvation.

Scheduling algorithms - RR



- Round Robin (RR)
- RR is a preemptive algorithm that selects the process that has been waiting the longest.
- After a specified time quantum, the running process is preempted and a new selection is made. Interrupts from an interval timer ensures that processing is suspended and control is transferred to the scheduling algorithm at the conclusion of the time quantum.
- RR is used extensively in time-shared systems.
- Use of small time quantums allows RR to provide good response time.
- However short time quantums increase the number of context switches between processes, thus decreasing utilisation.



Scheduling algorithms – Priority Based

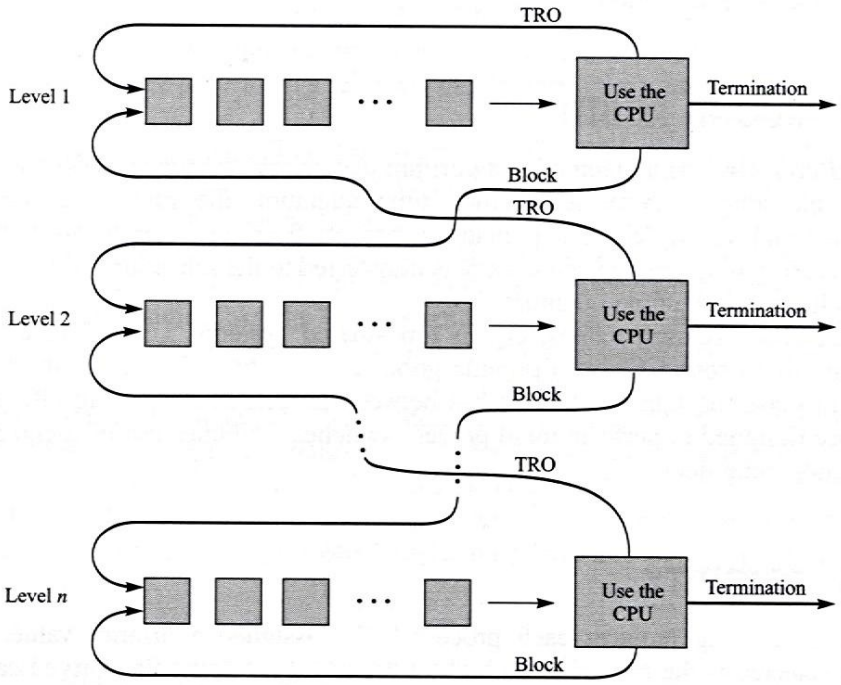
- We have previously discussed a type of priority based scheduling on OS9.
- Each process is assigned a priority level. FCFS can be used in case of a tie.
- Priority scheduling may be preemptive or non-preemptive.
- Priority assignment mechanisms vary widely and include basing the priority on
  - a process characteristic (memory usage, I/O, frequency etc.)
  - the user executing the process
  - usage cost (CPU time for higher-priority jobs cost more)
  - user- or administrator-assignable parameter.
- Some of the mechanisms yield dynamically changing priorities (e.g. amount of time running), while others are static (the priority associated with a user).
- Although it may be intuitively appealing to have the priority value directly proportional to its actual selection priority, on some systems the processes with the lowest-priority values are afforded the highest-selection priority.

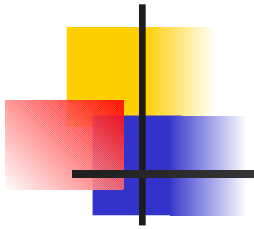
Scheduling algorithms – MFQ - I

- The major disadvantage of the five preceding algorithms is they basically treat all jobs the same. This results in each algorithm favouring certain kinds of processes.
- An algorithm using Multi-level Feedback Queues (MFQ) addresses this deficiency by customising the scheduling of a process based on the process's performance characteristics. An MFQ implements two or more scheduling queues.
  - An entering process is inserted into the top-level queue. When selected, processes in the queue are allocated a relatively small time-slice.
  - Upon expiration of the time slice, the process is moved to the next lower queue. Time slices associated with the queues increase as the level lowers. If a process blocks before using its entire timeslice, it is either moved to the next higher-level queue or to the top-level queue.
  - The process selected by MFQ is the next process in the highest-level nonempty queue.
  - Typically, selection within a queue is FCFS.
  - If a process enters a high-level queue while a process from a lower-level queue is executing, the low-level queue process may be preempted.

Scheduling algorithms – MFQ - II

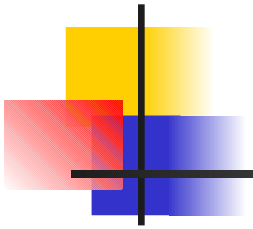
- I/O bound jobs remain in the higher-level queues where they receive higher priority over more CPU-bound processes in lower-level queues. However the more CPU-bound jobs are granted longer time-slices.
- As with SJF and SRT, it is possible for a CPU-bound job to suffer from starvation. A possible remedy is to elevate a process to the next higher-level queue if it has remained idle in a queue for a certain amount of time.





Scheduling algorithms – MFQ - III

- The number of variables in a MFQ system makes it both flexible and complex. The major options can be summarised as follows:
  - the number of queues
  - the time-slice associated with each queue.
  - the selection of algorithms used to select a process within each queue.
  - the condition(s) that will cause a process to sink to a lower-level queue.
  - the condition(s) that will cause a process to rise to a higher-level queue.
  - whether arrival of a process into a higher-level queue will preempt a process from a lower-level queues.
  - the mechanism for determining which queue a new process enters into.



Real-time scheduling - I

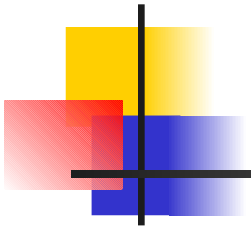
- Time plays an important role for real-time systems. An example is the embedded system in a portable MP3 player. Bit information that come out of the storage drive must be converted to sound. If the calculations take too long then the sound produced may sound peculiar. This is the case, having the right answer at the wrong time is the wrong answer.
- In hard and soft real-time systems real-time behaviour is achieved by dividing the program into a number of processes, each of whose behaviour is predictable and known in advance. These processes are generally short lived and can run to completion in well under a second. When an external event is detected, it is the job of the scheduler to schedule the processes in such a way that all deadlines are met.
- The events that a real-time system may have to respond to can be further categorised as periodic (occurring at regular intervals) or aperiodic (occurring unpredictable). A system may have to respond to multiple periodic event streams. Depending on how much time each event requires for processing, it may not even be possible to handle them all.

Real-time scheduling - II

- For example, if there are m periodic events and event i occurs with period P_i and requires C_i seconds of CPU time to handle each event, then the load can only be handled if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- A real-time system that meets this criteria is said to be schedulable.
- Consider a real-time system with three periodic events, with periods 100, 200 and 500 ms.
  - If these events require 50, 30 and 100msec of CPU time per event, respectively, the system is schedulable because 0.5+0.15+0.2<1.
  - If a fourth event with a period of 1 second is added the system will remain schedulable as long as this event does not need more than 150 ms of CPU time per event.
  - Here we have assumed that the context-switching overhead is so small that it can be ignored (this assumption may not always be true).



Real-time scheduling - III

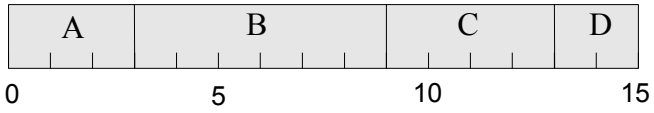
- Real-time scheduling algorithms can be static or dynamic.
- The former make their scheduling decisions before the system starts running. The latter make their scheduling decisions at run-time.
- Static scheduling only works when there is perfect information available in advance about the work needed to be done and the deadlines that have to be met.
- Dynamic scheduling algorithms do not have these restrictions.

Example - FCFS

Consider the process listed in the table below.

Process	*Arrival time*	*Processing time*
A	0.000	3
B	1.001	6
C	4.001	4
D	6.001	2

FCFS. The processes are executed in the order they are received.

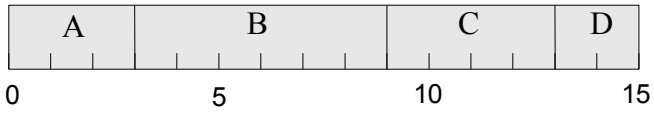


Example - SJF

Consider the process listed in the table below.

Process	*Arrival time*	*Processing time*
A	0.000	3
B	1.001	6
C	4.001	4
D	6.001	2

SJF. Process A starts executing since it is the only choice at time 0. At time 3, B is the only process in the queue. At time 9 when B completes, process D runs because it is shorter than process C.

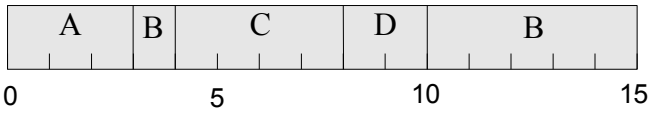


Example - SRT

Consider the process listed in the table below.

Process	*Arrival time*	*Processing time*
A	0.000	3
B	1.001	6
C	4.001	4
D	6.001	2

SRT. Process A starts executing since it is the only choice at time 0. It remains running when process B arrives because its remaining time is less. At time 3, process B is the only process in the queue. At time 4.001, process C arrives and starts running because its remaining time (4) is less than process B's remaining time (4.999). At time 6.001, process C remains running because its remaining time (1.999) is less than process D's remaining time (2). When process C terminates, process D runs because its remaining time is less than that of process B,

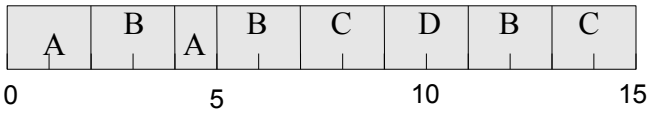


Example – RR (quantum=2)

Consider the process listed in the table below.

Process	*Arrival time*	*Processing time*
A	0.000	3
B	1.001	6
C	4.001	4
D	6.001	2

RR (quantum 2). When process A's first quantum expires, process B runs. At time 4, process A restarts and process B returns to the ready queue. At time 4.001, process C enters the ready queue after process B. At time 5, process A terminates and process B runs. At time 6.001, process D enters the ready queue behind process C. Starting at time 7, process C, D, B, and C run in sequence.

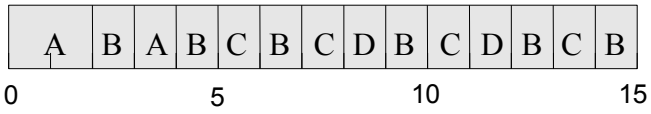


Example – RR (quantum=1)

Consider the process listed in the table below.

Process	*Arrival time*	*Processing time*
A	0.000	3
B	1.001	6
C	4.001	4
D	6.001	2

RR (quantum 1). Process A runs for two time-slices since process B does not arrive until 1.001. Process B runs at time 4 since process C does not arrive until 4.001. Process B runs again at time 6 since process D does not arrive until 6.001. Process D enters the ready queue behind process C. From time 7 onwards, execution cycles through processes C, D, and B.



Scheduling in Windows

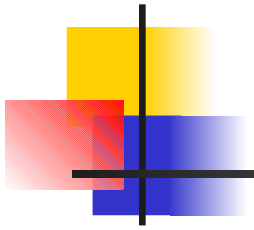
- Windows 2000/XP uses a priority-based, preemptive scheduling algorithm. The dispatcher uses a 32-level priority based scheme. Priorities are divided in two classes:

•the *variable class* contains threads having priorities ranging from 1-15

•the *real-time class* contains threads with priorities ranging from 16-31

•note that there is also a thread running at priority 0 that is used for memory management.

- The dispatcher uses a queue for each scheduling priority and traverses the queue from highest to lowers until it finds a thread that is ready to run. If no ready thread is found, the dispatcher will execute a special thread called the ***idle thread***. Processes typically belong to the NORMAL_PRIORITY_CLASS.
- When a thread's time-quantum runs out, that thread is interrupted. If the thread is in the variable priority class then its priority is lowered.
- In order to cater for interactive programs, *e.g.* multimedia type programs, Windows 2000 /XP has a special scheduling rule for processes in the NORMAL_PRIORITY_CLASS.
  - It distinguishes between the foreground process, *i.e.* the process that has focus, and the background processes.
  - Windows 2000 increases the scheduling quantum for the foreground process by some factor, typically 3. This means that the foreground process is given three times longer to run before a time-slicing preemption occurs.



Scheduling in Windows

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1