Lecture 3: CPU Scheduling

Contents

- What is process
- Context Switch
- Processes hierarchy
- Process creation and termination
- CPU Scheduling
- Scheduling Criteria & Optimization
- Basic Scheduling Approaches
- Priority Scheduling
- Queuing and Queues Organization
- Scheduling Examples in Real OS
- Deadline Real-Time CPU Scheduling

What is a process?

Textbooks use the terms *job* and *process* almost interchangeably

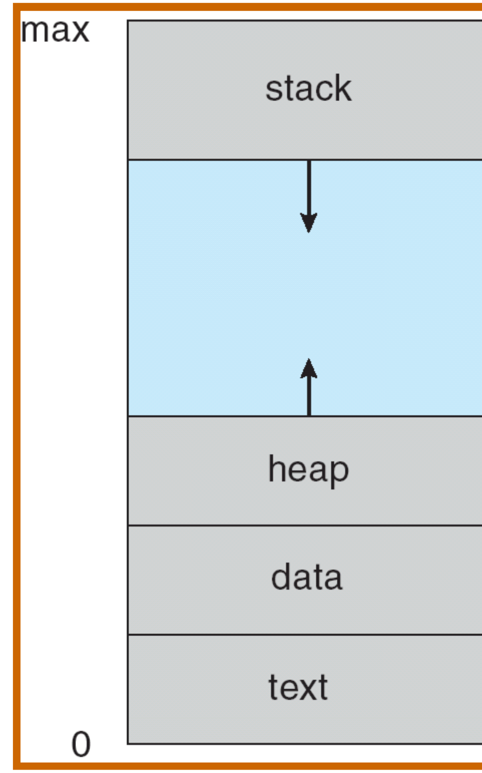
Process – a program in execution; process execution must progress in sequential fashion

A process includes:

- program counter
- stack
- data section.

Information associated with each process:

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information (“process environment”)

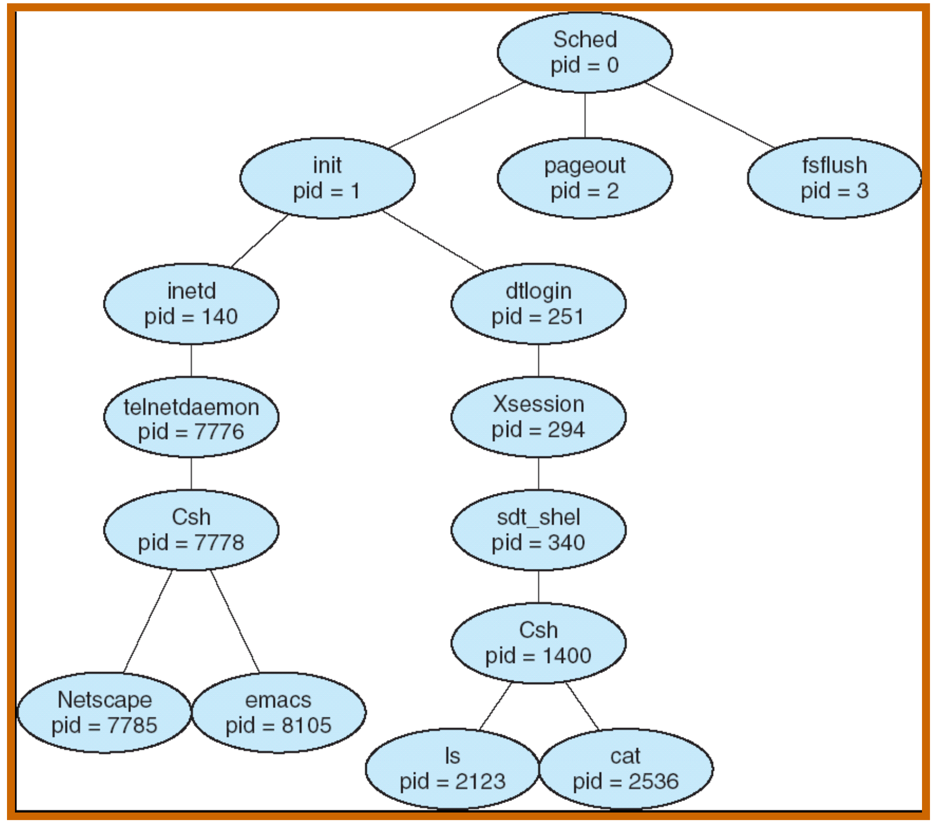


C Program Forking Separate Process

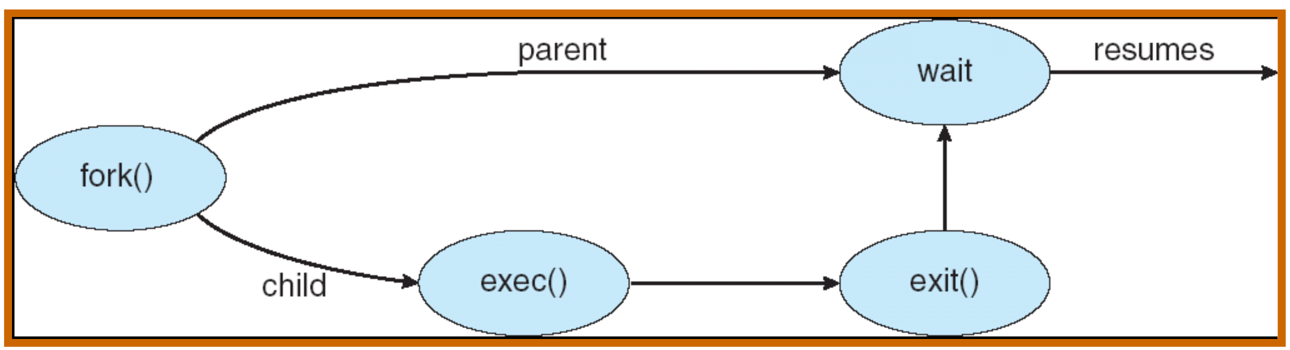
```
int main()
{
   Pid_t  pid;
   /* fork another process */
   pid = fork();
   if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
   }
   else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
   }
   else { /* parent process */
        /* parent will wait for the child to complete */
        wait (NULL);
        printf ("Child Complete");
        exit(0);
   }
}
```

Process Creation Illustrated

Tree of processes



POSIX parent process waiting for its child to finish



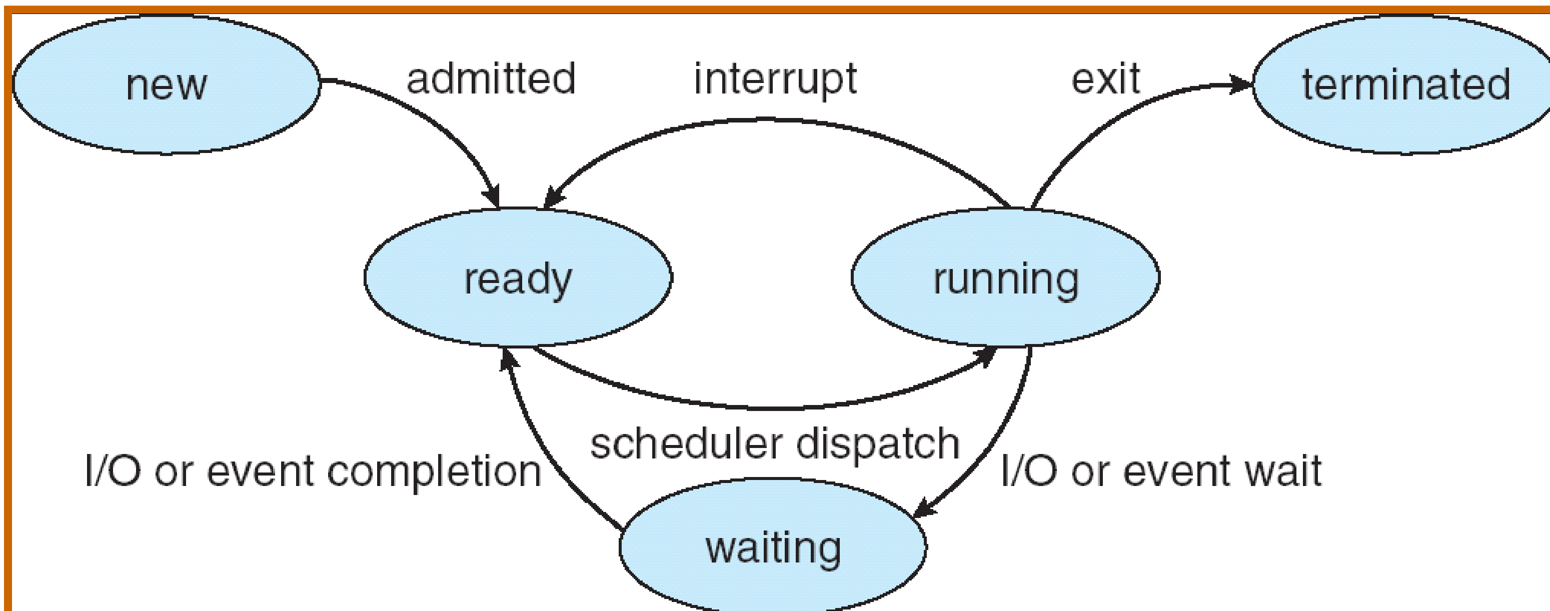
Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system

- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow children to continue if the parent terminates – the problem of '*zombie*'
    - All children terminated - *cascading termination*

Process State

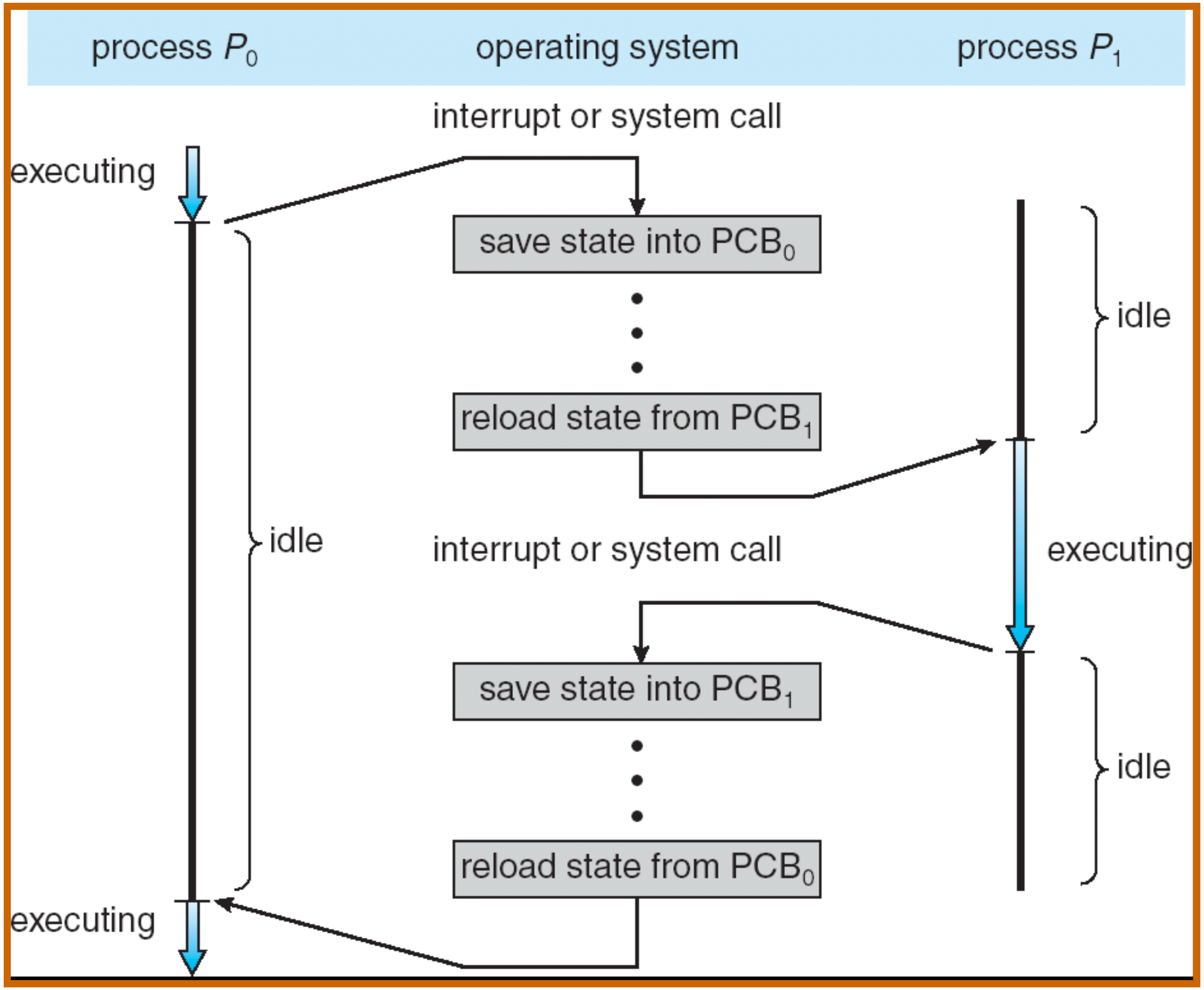
- As a process executes, it changes its ***state***
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a CPU
  - **terminated**: The process has finished execution



Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is *overhead*; the system does no useful work while switching
- Time dependent on hardware support
  - Hardware designers try to support routine context-switch actions like saving/restoring all CPU registers by one pair of machine instructions

CPU Switch From Process to Process



Context switch is similar to handling an interrupt

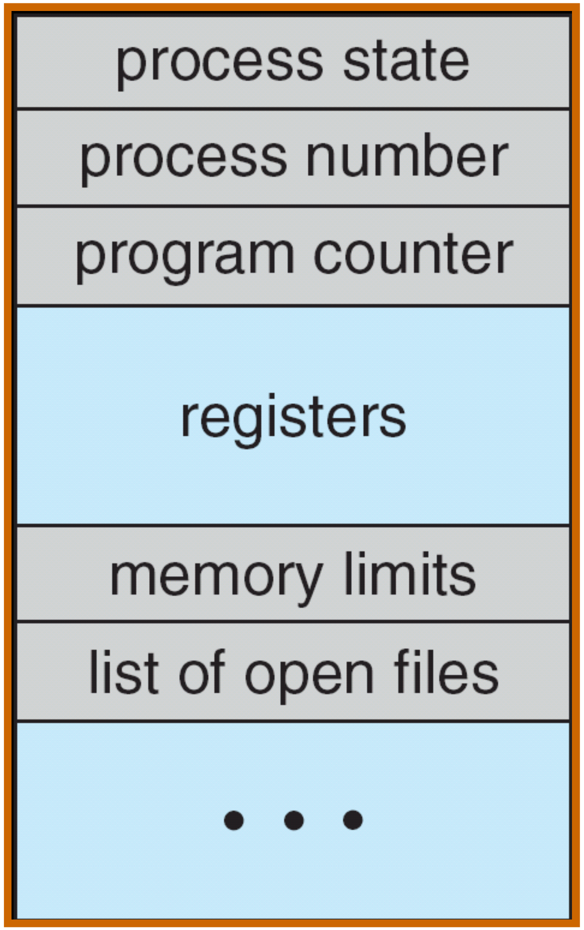
Context switch steps:
1.Save current process to PCB
2.Decide which process to run
3.Reload of new process from PCB

Context switch should be fast, because it is overhead.

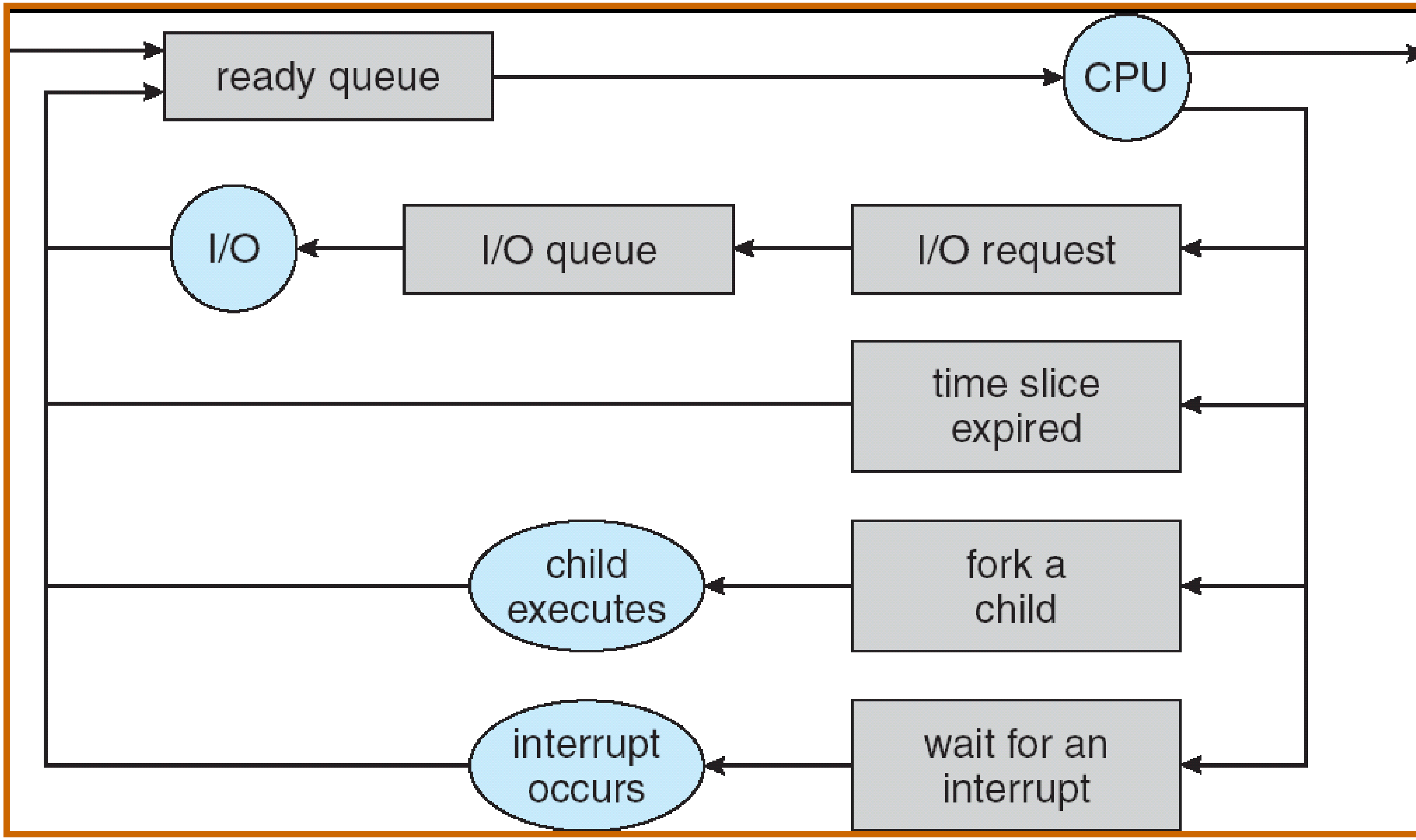
Process Control Block (PCB)

Information associated with each process

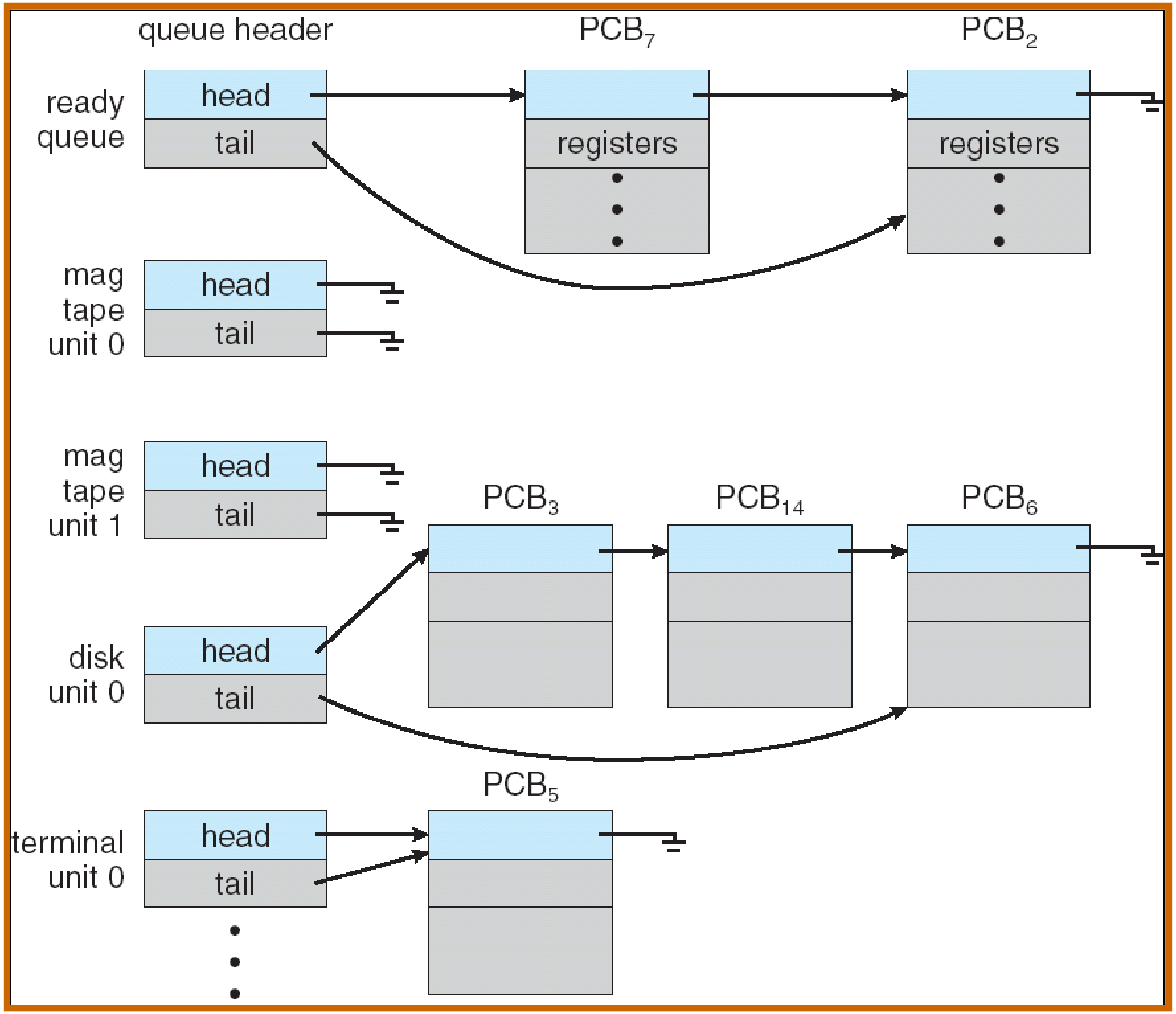
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information ("process environment")



Simplified Model of Process Scheduling



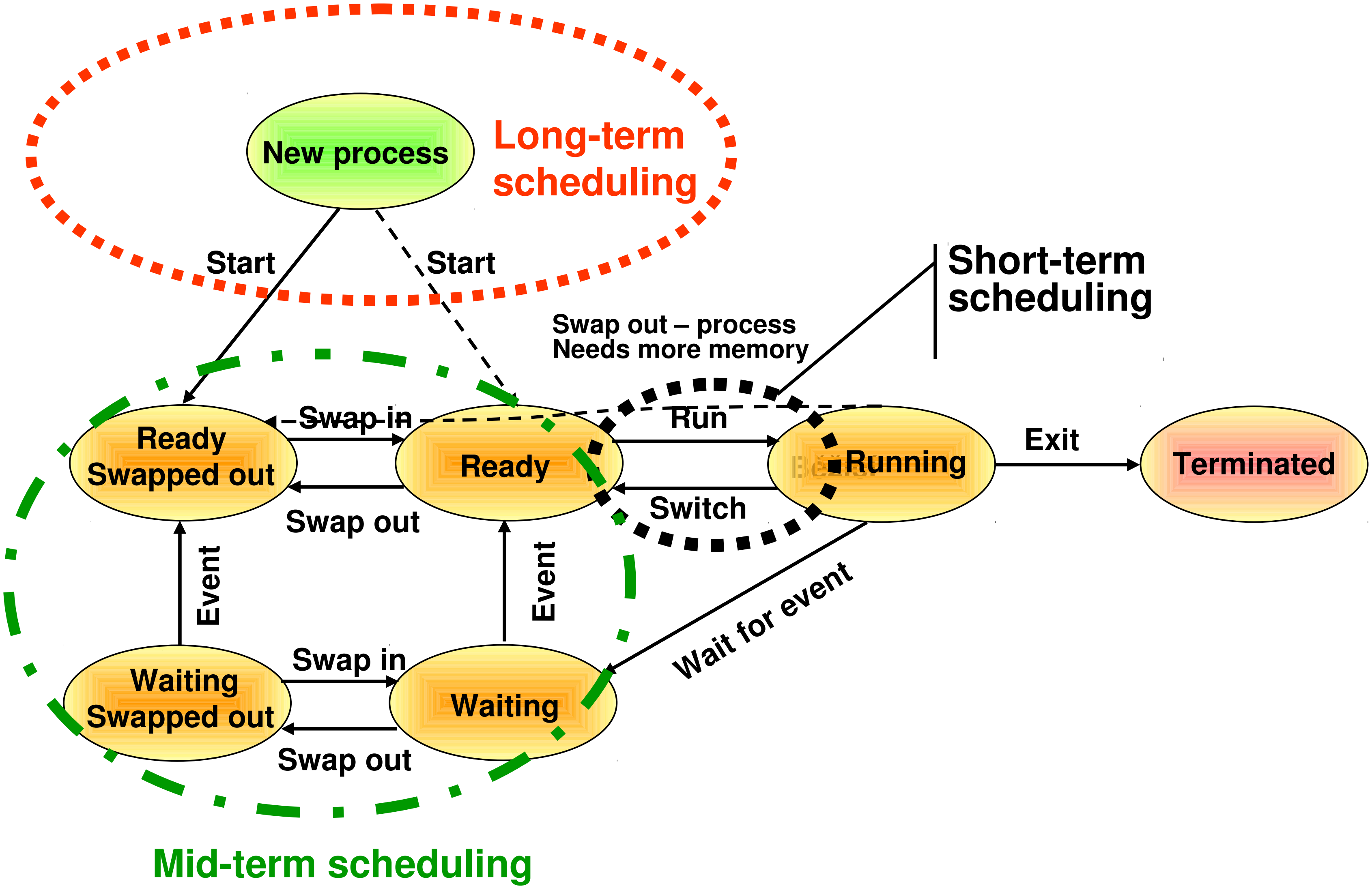
Ready Queue and Various I/O Device Queues



Schedulers

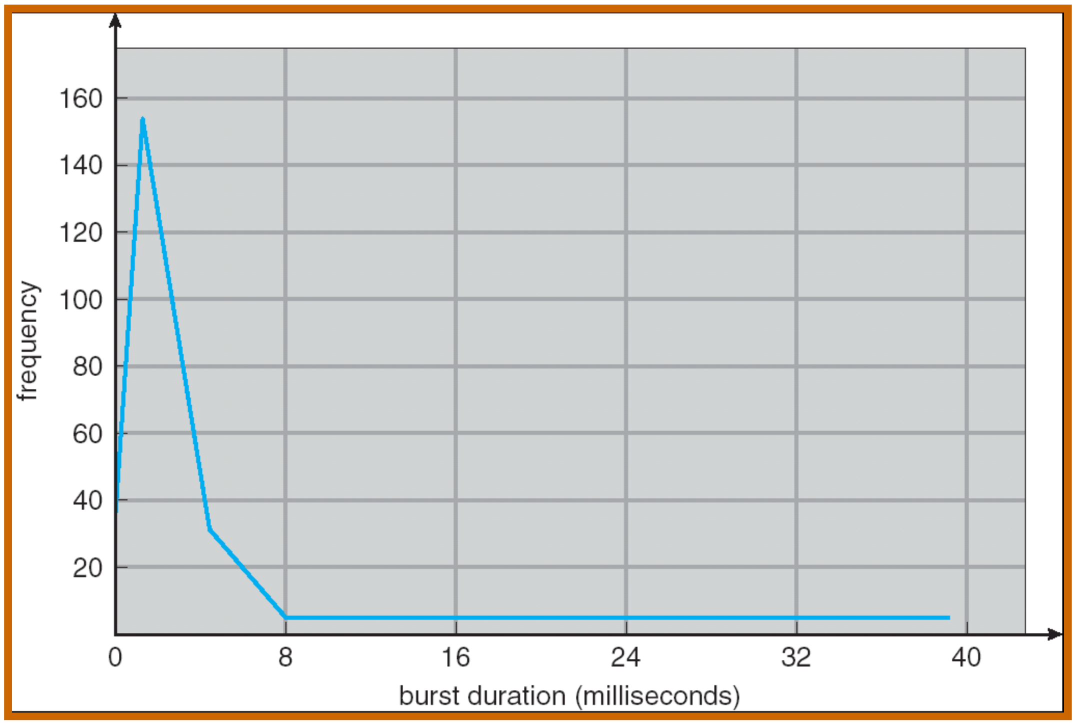
- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the *degree of multiprogramming*
- **Mid-term scheduler** (or tactic scheduler) – selects which process swap out to free memory or swap in if the memory is free
  - Partially belongs to memory manager
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)

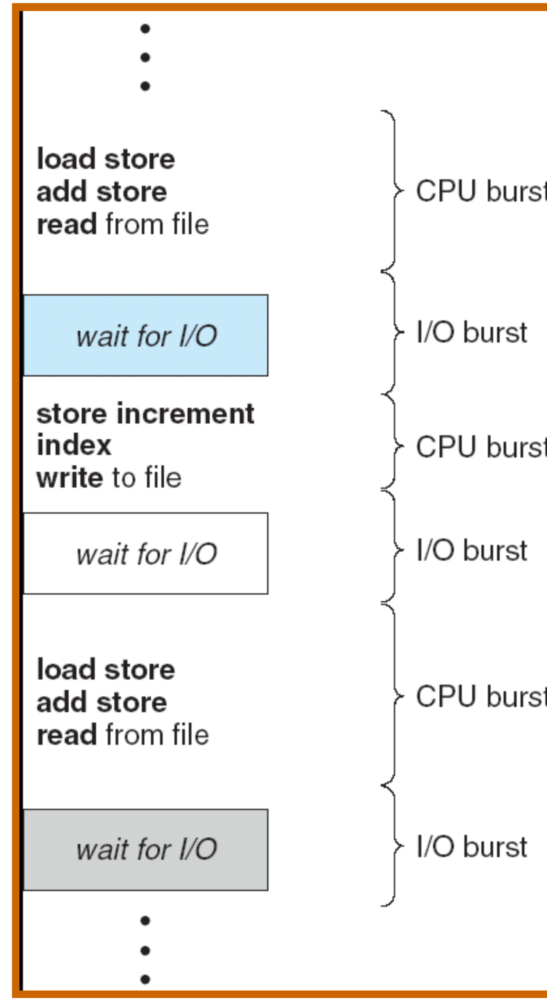
Process states with swapping



Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
- CPU burst distribution





CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is *nonpreemptive*
- 2 and 3 scheduling are *preemptive*

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term 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 another running – overhead

Scheduling Criteria & Optimization

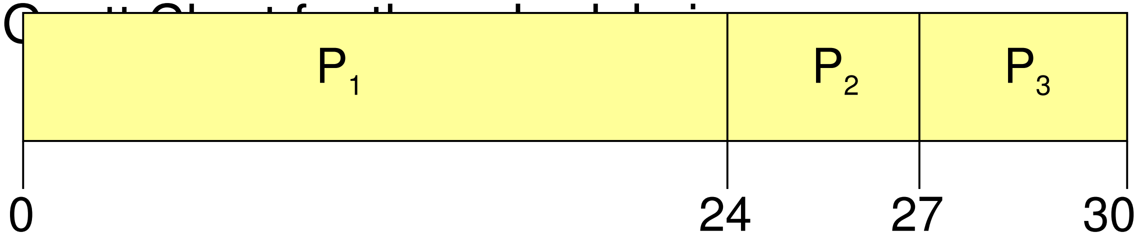
- CPU utilization – keep the CPU as busy as possible
  - Maximize CPU utilization
- Throughput – # of processes that complete their execution per time unit
  - Maximize throughput
- Turnaround time – amount of time to execute a particular process
  - Minimize turnaround time
- Waiting time – amount of time a process has been waiting in the ready queue
  - Minimize waiting time
- Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing and interactive environment)
  - Minimize response time

First-Come, First-Served (FCFS) Scheduling

- Most simple nonpreemptive scheduling.

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



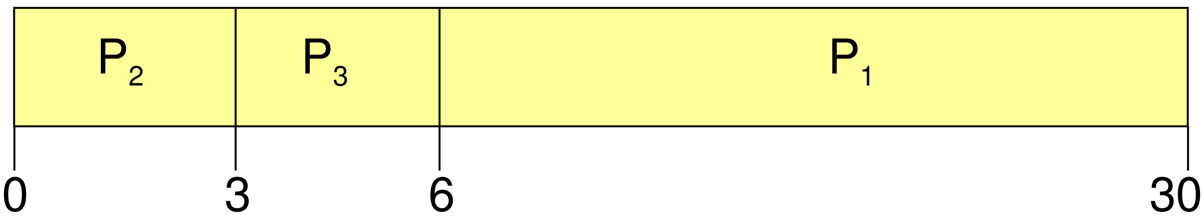
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- *Convoy effect* short process behind long process

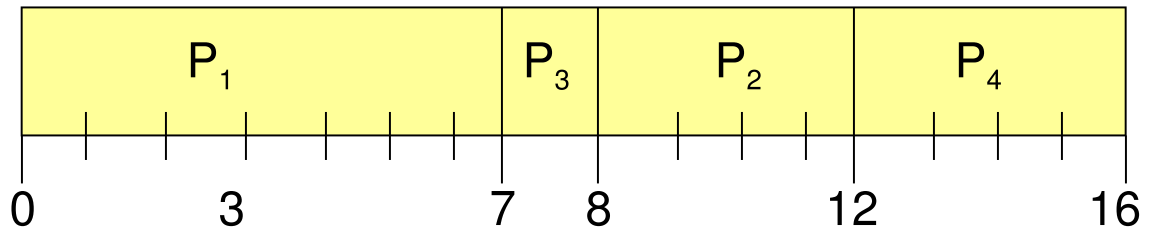
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time (SRT)
- SJF is optimal – gives minimum average waiting time for a given set of processes

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

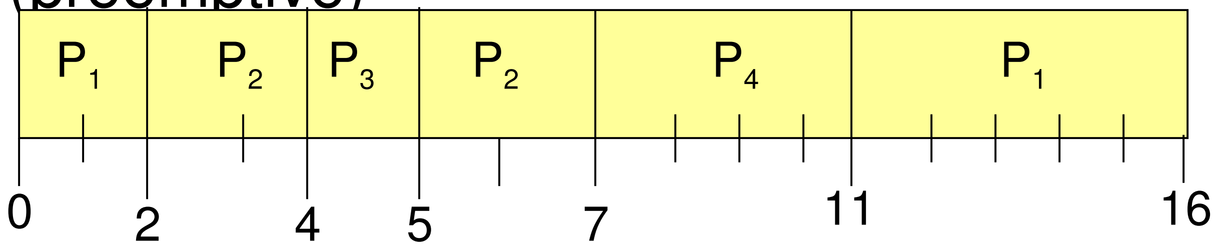


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)

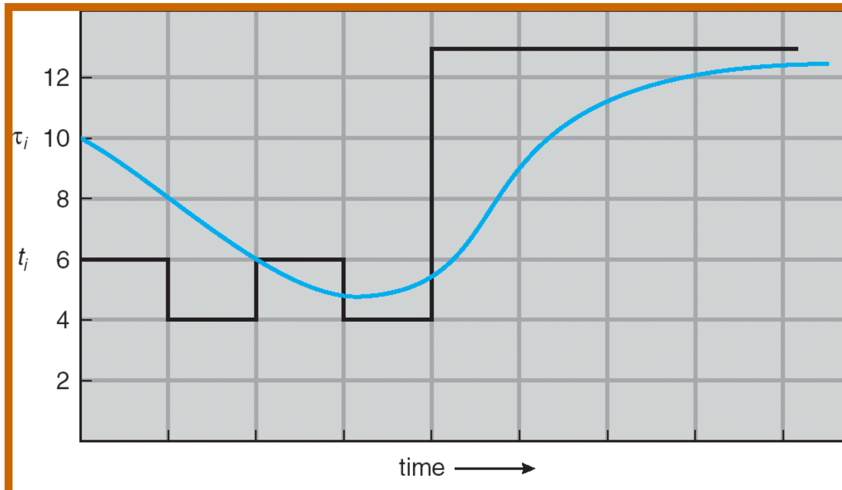


- Average waiting time = (9 + 1 + 0 +2)/4 = 3

Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

1. $t_n =$ actual lenght of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define : $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots$$
$$+ (1 - \alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute (When MIT shut down in 1973 their IBM 7094 - the biggest computer - they found process with low priority waiting from 1967)
- Solution: Aging – as time progresses increase the priority of the process

Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Performance
  - q large $\Rightarrow$ FCFS
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

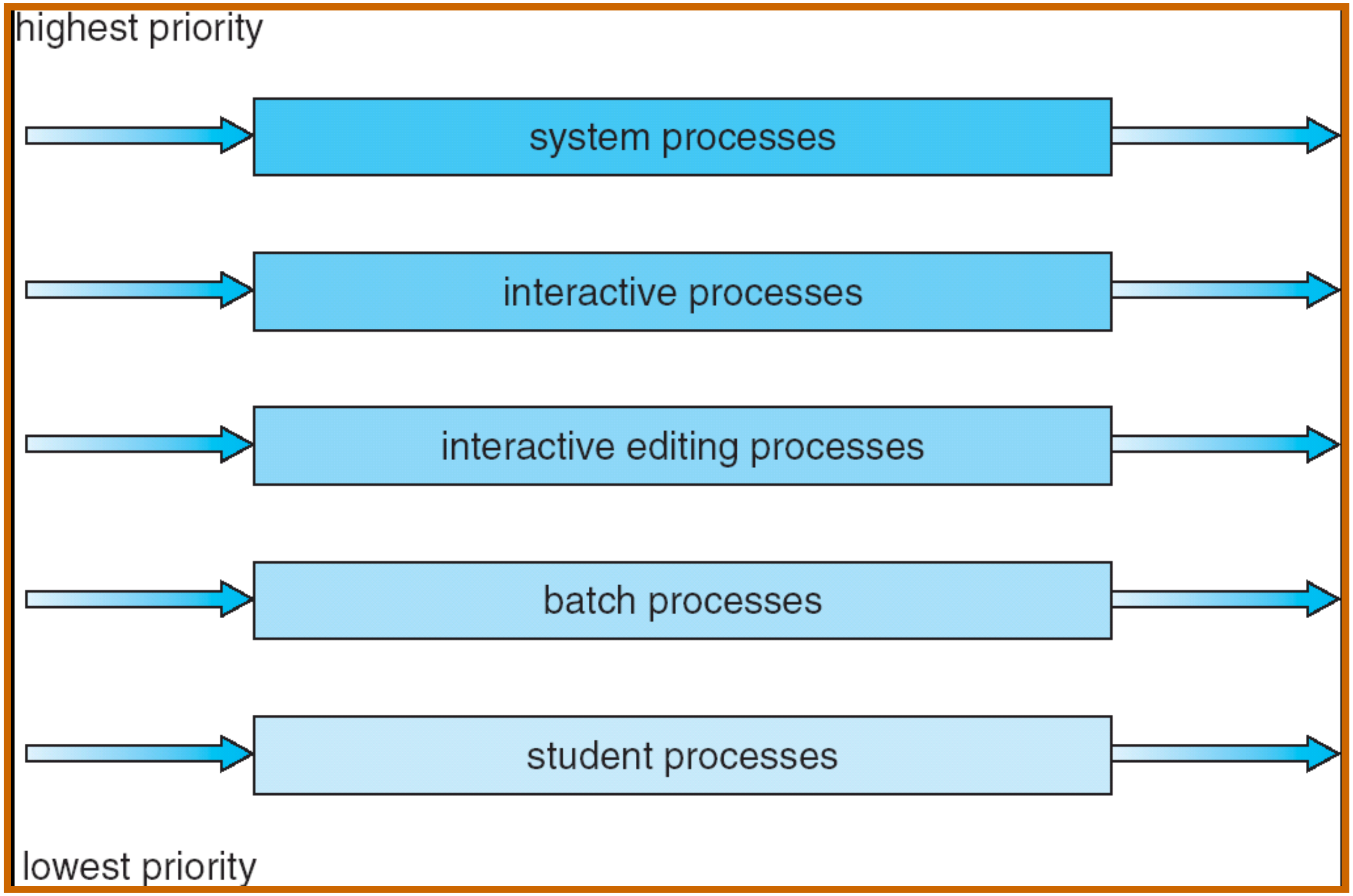
0 20 37 57 77 97 117 121 134 154 162

- Typically, higher average turnaround than SJF, but better *response*

Multilevel Queue

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Danger of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

Multilevel Queue Scheduling

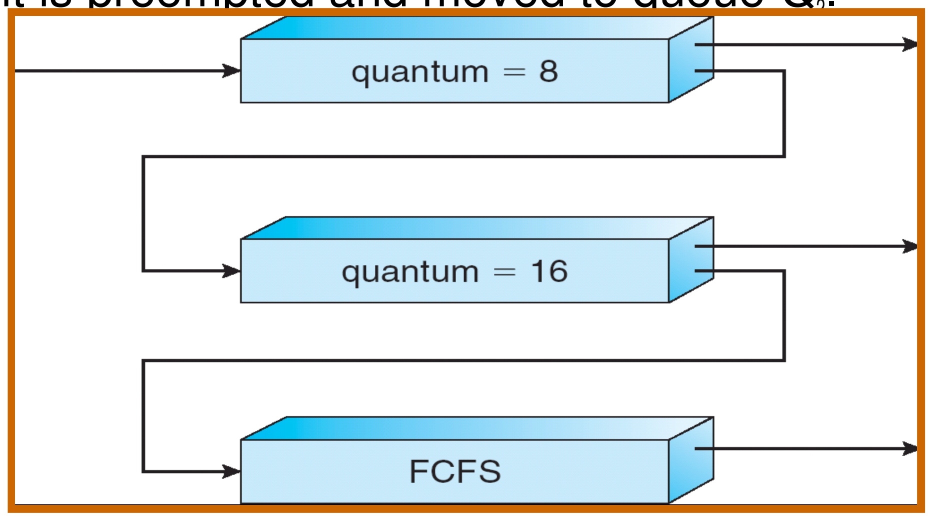


Multilevel Feedback Queue

- A process can move between the various queues; aging can be treated this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0. When it gains CPU, job receives 8 milliseconds. If it exhausts 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 the job receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.



Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
  - Multiple-Processor Scheduling has to decide not only which process to execute but also <u>where</u> (i.e. on which CPU) to execute it
- *Homogeneous processors* within a multiprocessor
- *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing
- *Symmetric multiprocessing (SMP)* – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- *Processor affinity* – process has affinity for the processor on which it has been recently running
  - Reason: Some data might be still in cache
  - *Soft affinity* is usually used – the process can migrate among CPUs

Windows XP Priorities

Priority classes (assigned to each process)

	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

- Relative priority "normal" is a <u>base</u> priority for each class – starting priority of the thread
- When the thread exhausts its quantum, the priority is lowered
- When the thread comes from a wait-state, the priority is increased depending on the reason for waiting
  - A thread released from waiting for keyboard gets more boost than a thread having been waiting for disk I/O

Linux Scheduling

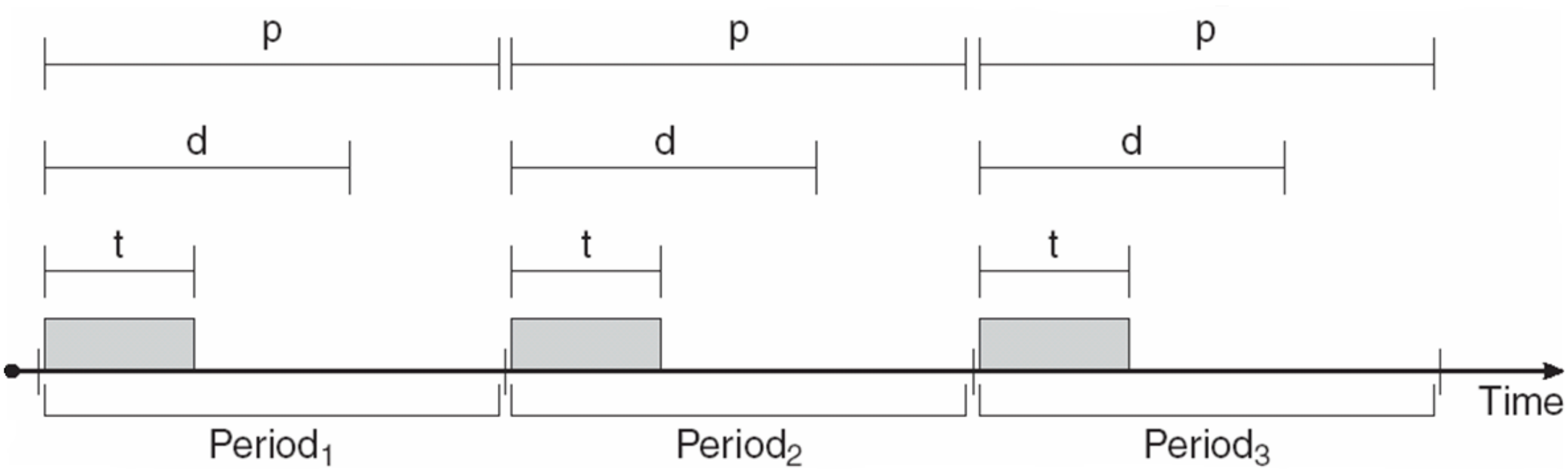
- Two algorithms: time-sharing and real-time
- Time-sharing
  - Prioritized credit-based – process with most credits is scheduled next
  - Credit subtracted when timer interrupt occurs
  - When credit = 0, another process chosen
  - When all processes have credit = 0, recrediting occurs
    - Based on factors including priority and history
- Real-time
  - Soft real-time
  - POSIX.1b compliant – two classes
    - FCFS and RR
    - Highest priority process always runs first

Real-Time Systems

- A real-time system requires that results be not only correct but in time
  - produced within a specified deadline period
- An embedded system is a computing device that is part of a larger system
  - automobile, airliner, dishwasher, ...
- A safety-critical system is a real-time system with catastrophic results in case of failure
  - e.g., airplanes, racket, railway traffic control system
- A hard real-time system guarantees that real-time tasks be completed within their required deadlines
  - mainly single-purpose systems
- A soft real-time system provides priority of real-time tasks over non real-time tasks
  - a “standard” computing system with a real-time part that takes precedence

Real-Time CPU Scheduling

- Periodic processes require the CPU at specified intervals (periods)
- *p* is the duration of the period
- *d* is the deadline by when the process must be serviced (must finish within *d*) – often equal to *p*
- *t* is the processing time



Scheduling of two and more tasks

Can be scheduled if $r = \sum_{i=1}^{N} \frac{t_i}{p_i} \leq 1$ (N = number of processes)

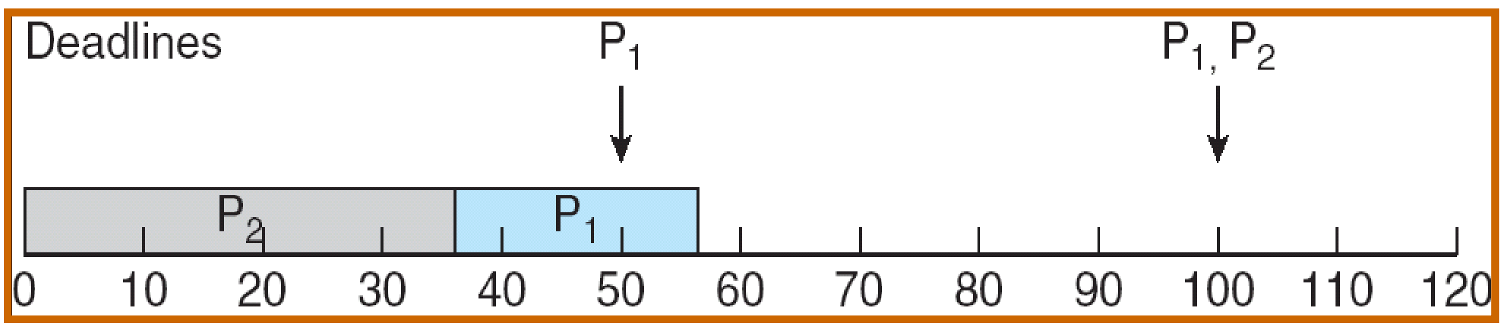
r – CPU utilization

Process P_1: service time = 20, period = 50, deadline = 50

Process P_2: service time = 35, period = 100, deadline = 100

$$r = \frac{20}{50} + \frac{35}{100} = 0.75 < 1 \Rightarrow \text{schedulable}$$

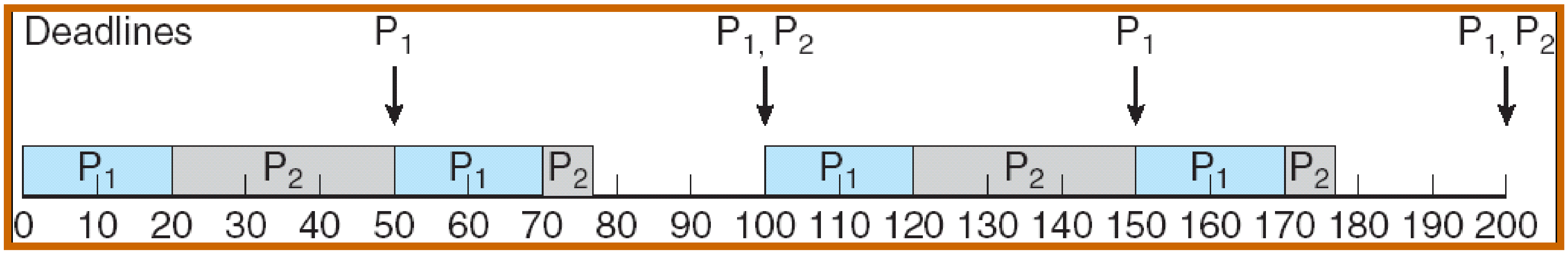
When P_2 has a higher priority than P_1, a failure occurs:



Rate Monotonic Scheduling (RMS)

- A process priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority

- P_1 is assigned a higher priority than P_2.

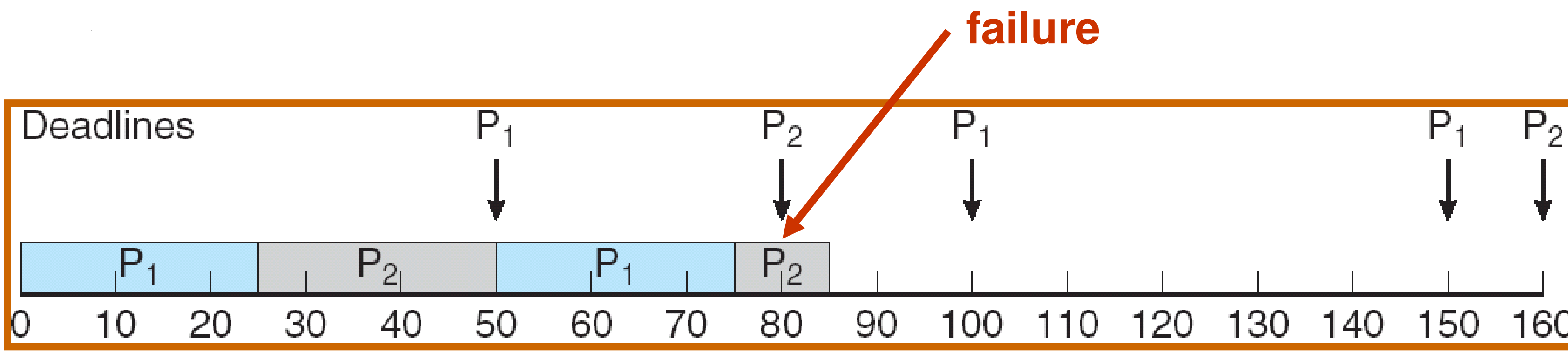


Process P_1: service time = 20, period = 50, deadline = 50

Process P_2: service time = 35, period = 100, deadline = 100

works well

Missed Deadlines with RMS



Process P_1: service time = 25, period = 50, deadline = 50
Process P_2: service time = 35, period = 80, deadline = 80

$$r = \frac{25}{50} + \frac{35}{80} = 0{,}9375 < 1 \Rightarrow \text{schedulable}$$

RMS is guaranteed to work if

$$r = \sum_{i=1}^{N} \frac{t_i}{p_i} \le N\left(\sqrt[N]{2} - 1\right);$$

N = number of processes

$$\lim_{N \to \infty} N\left(\sqrt[N]{2} - 1\right) = \ln 2 \approx 0.693147$$

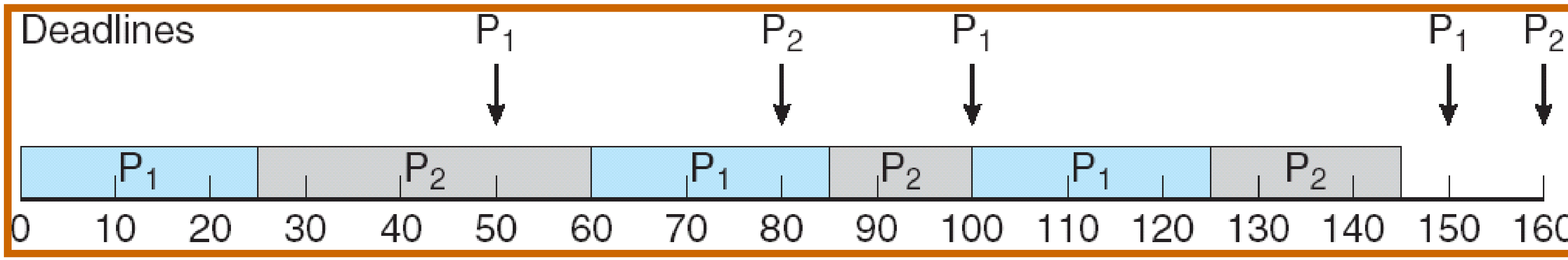
sufficient condition

N	$N\left(\sqrt[N]{2} - 1\right)$
2	0,828427
3	0,779763
4	0,756828
5	0,743491
10	0,717734
20	0,705298

Earliest Deadline First (EDF) Scheduling

- Priorities are assigned according to deadlines:

  the earlier the deadline, the higher the priority;
  the later the deadline, the lower the priority.



Process P_1: service time = 25, period = 50, deadline = 50

Process P_2: service time = 35, period = 80, deadline = 80

Works well even for the case when RMS failed

PREEMPTION may occur

RMS and EDF Comparison

- **RMS:**
  - Deeply elaborated algorithm
  - Deadline guaranteed if the condition $r \leq N\left(\sqrt[N]{2}-1\right)$ is satisfied (sufficient condition)
  - Used in many RT OS

- **EDF:**
  - Periodic processes deadlines kept even at 100% CPU load
  - Consequences of the overload are unknown and unpredictable
  - When the deadlines and periods are not equal, the behaviour is unknown

End of Lecture 3