CPU Scheduling

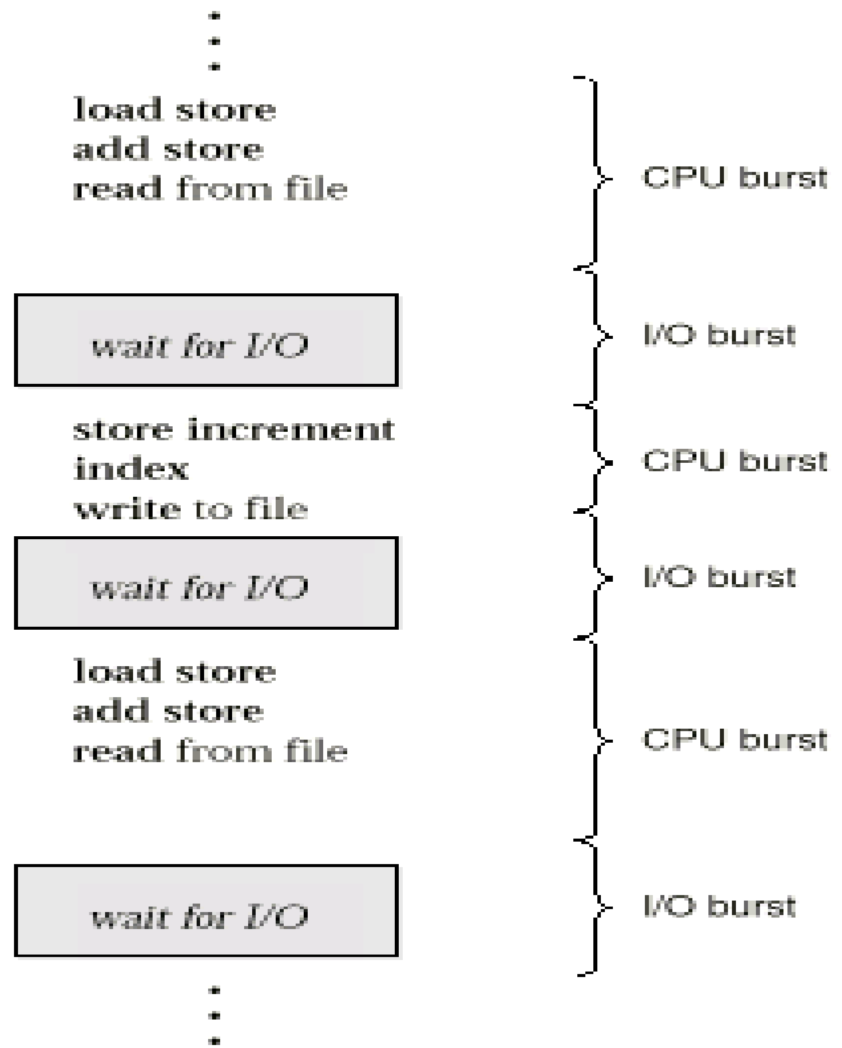
Introduction

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time Scheduling
- Algorithm Evaluation

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

Alternating Sequence of CPU And I/O Bursts



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*.
- All other scheduling is *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.

Scheduling Criteria

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

Optimization Criteria

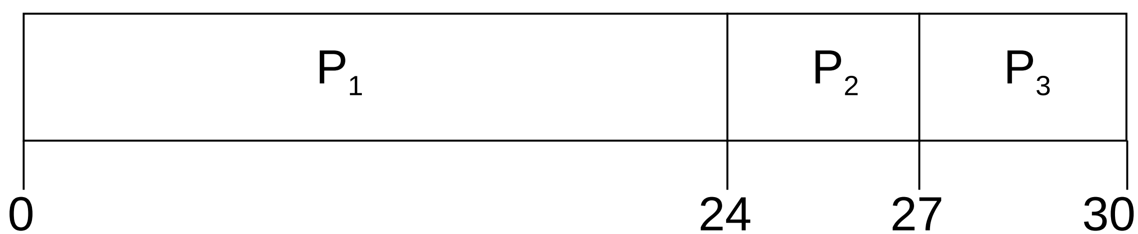
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First-Come, First-Served (FCFS) Scheduling

- Example:

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:



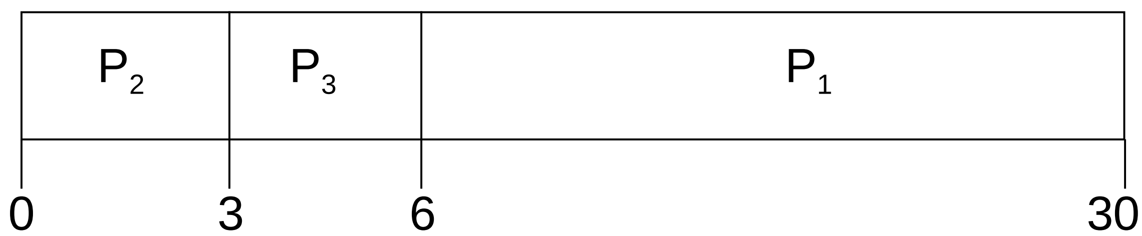
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

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

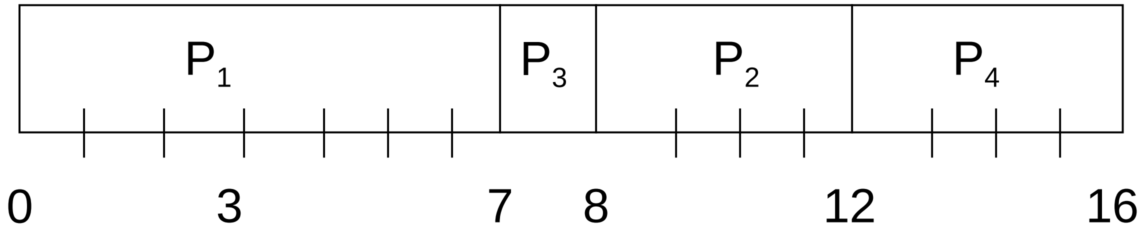
Shortest-Job-First (SJF) Scheduling

- 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-First (SRTF).
- 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	7
P_2	2	4
P_3	4	1
P_4	5	4

- SJF (non-preemptive)

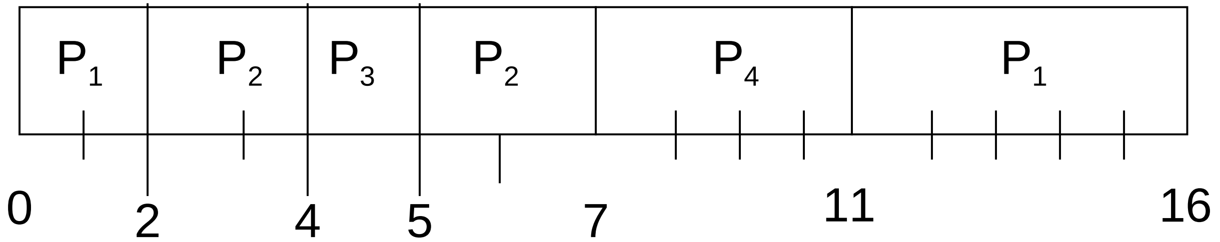


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF(SRTF)

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- SJF (preemptive)



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

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.
- 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 $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high.

Example: RR with Time Quantum = 20

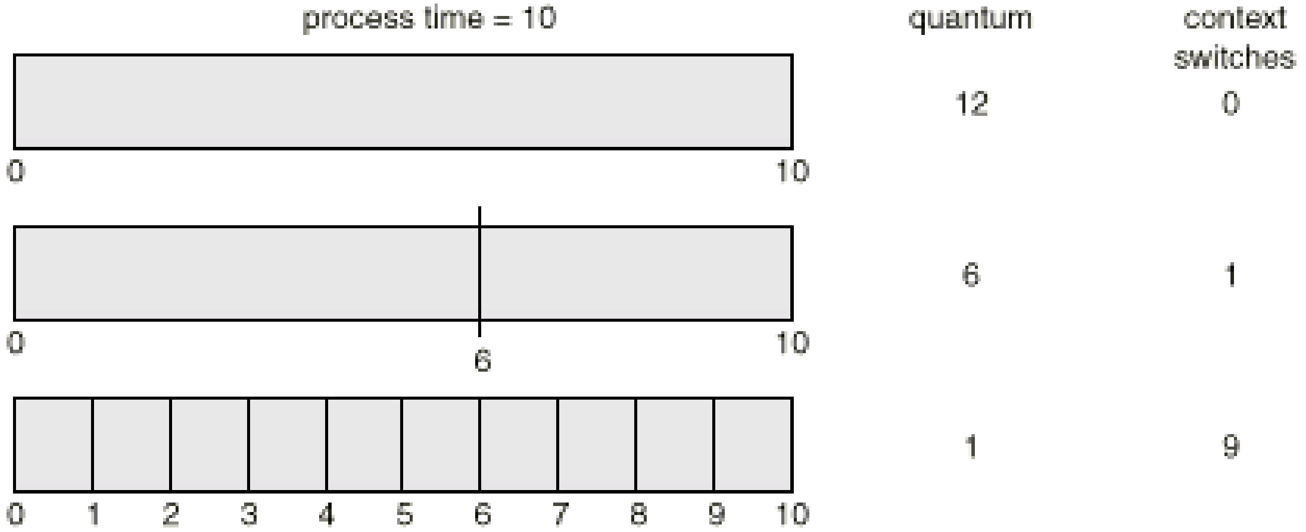
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

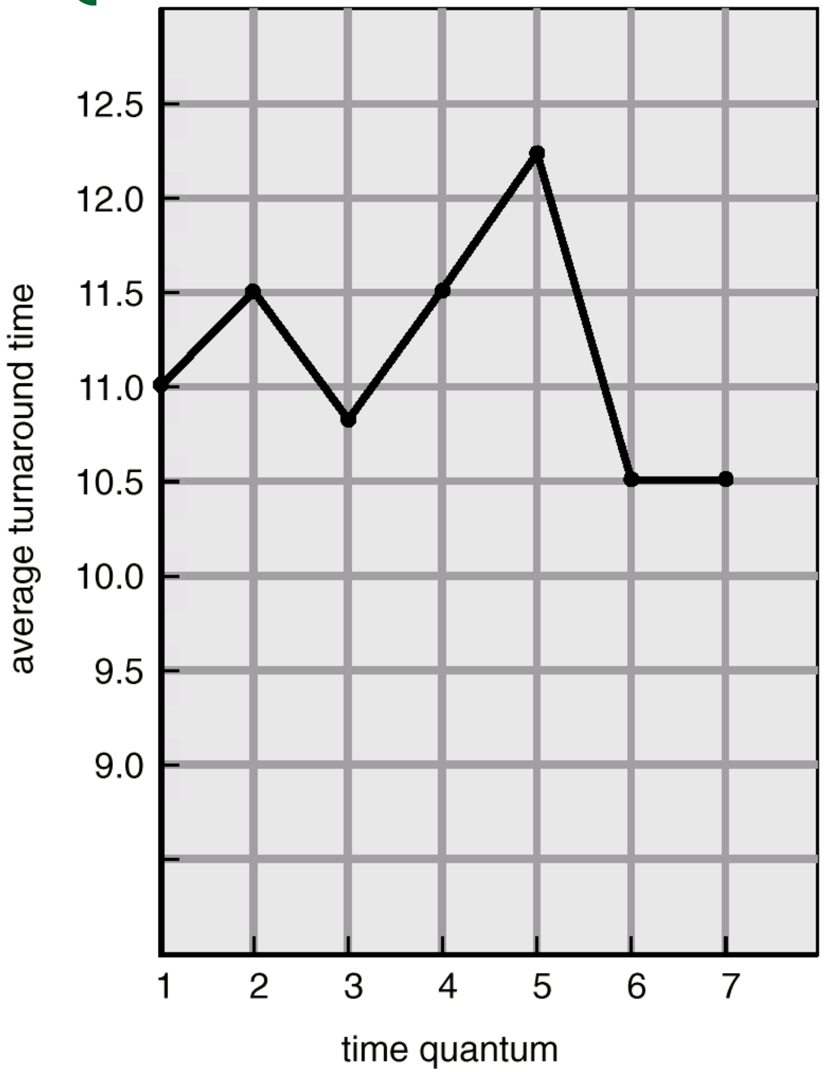
0	20	37	57	77	97	117	121	134	154	162
P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3	

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

How a Smaller Time Quantum Increases Context Switches



Turnaround Time Varies With The Time Quantum

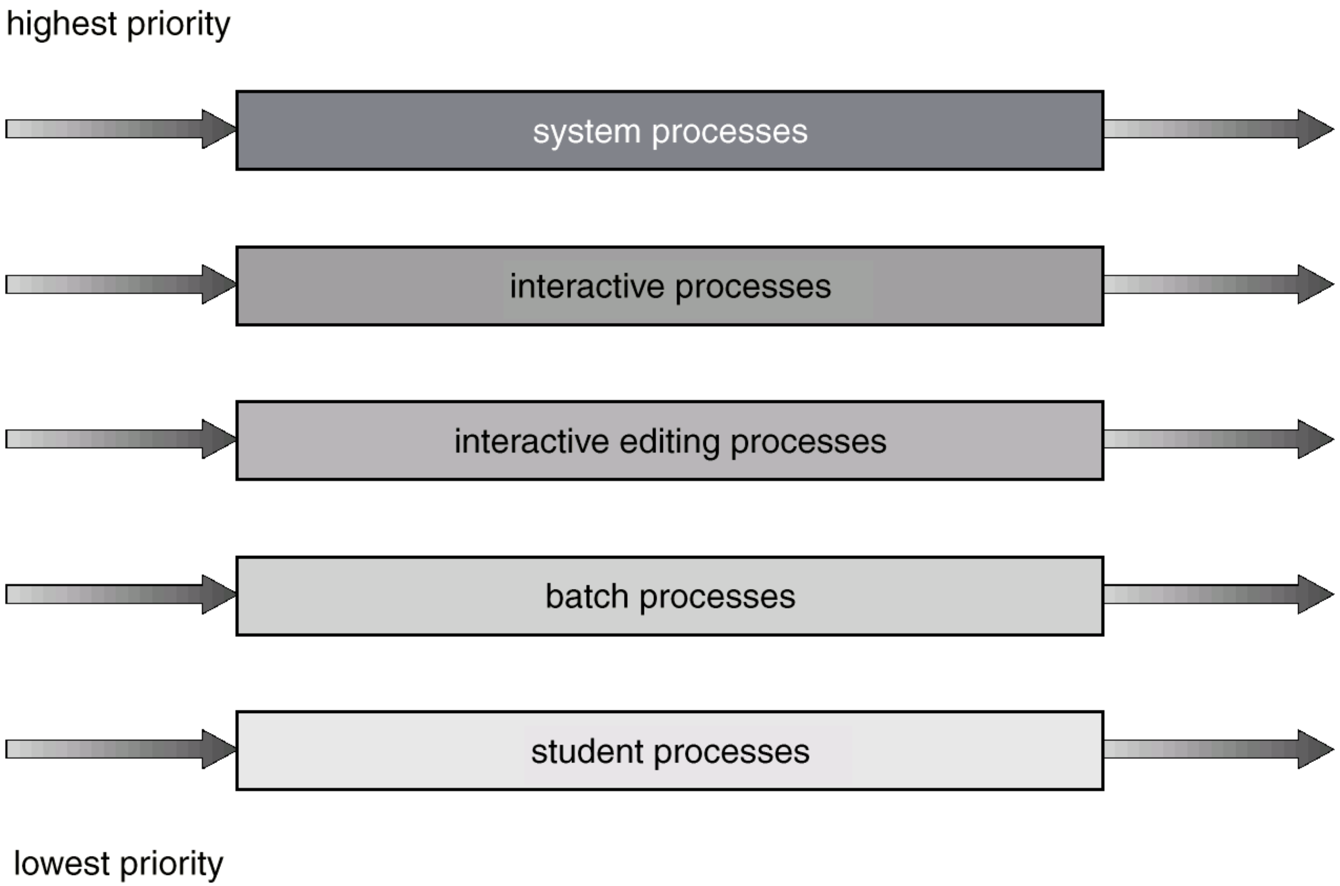


process	time
P_1	6
P_2	3
P_3	1
P_4	7

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. Possibility 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

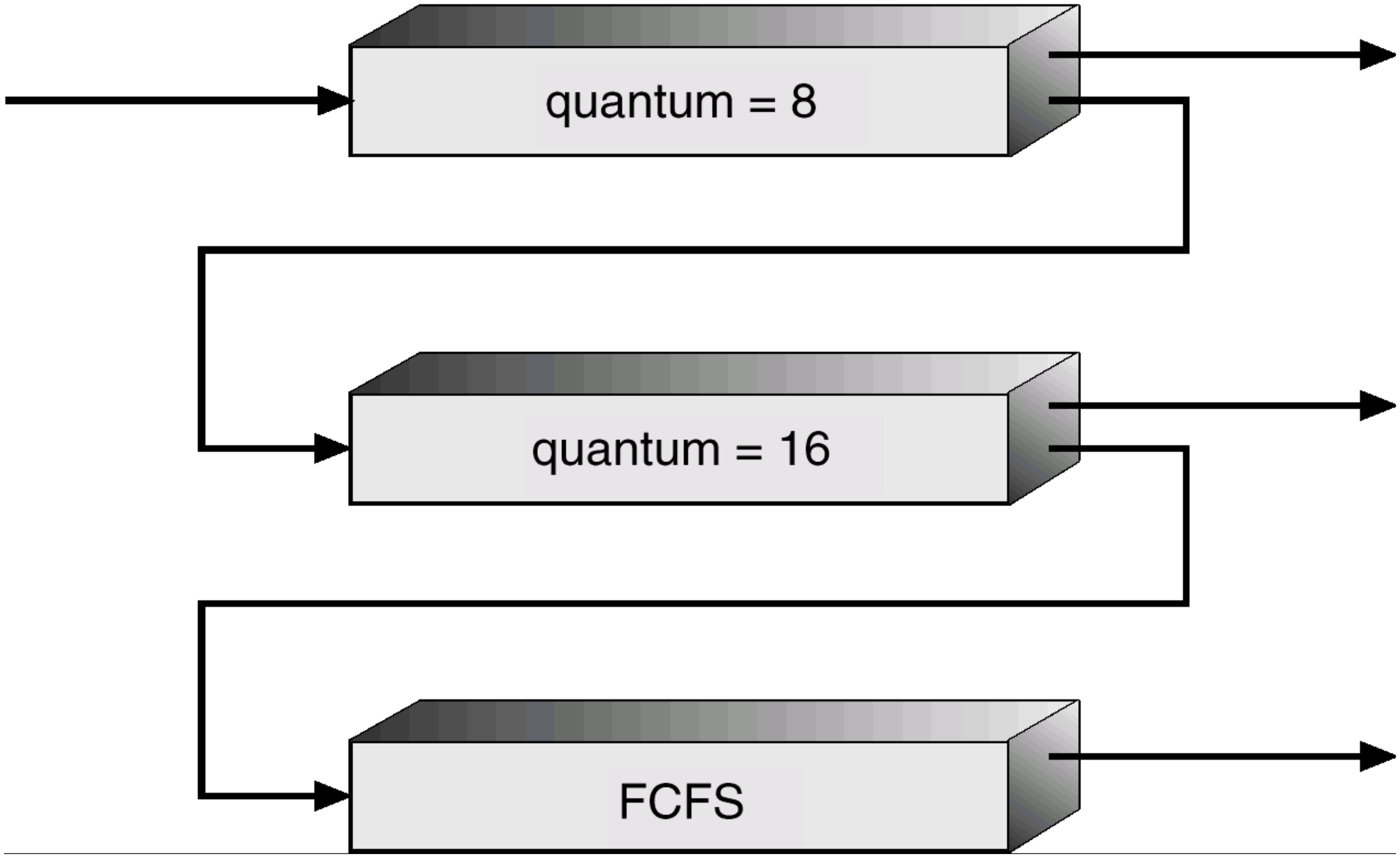
Multilevel Queue Scheduling



Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented 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

Multilevel Feedback Queues



Example of Multilevel Feedback Queue

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

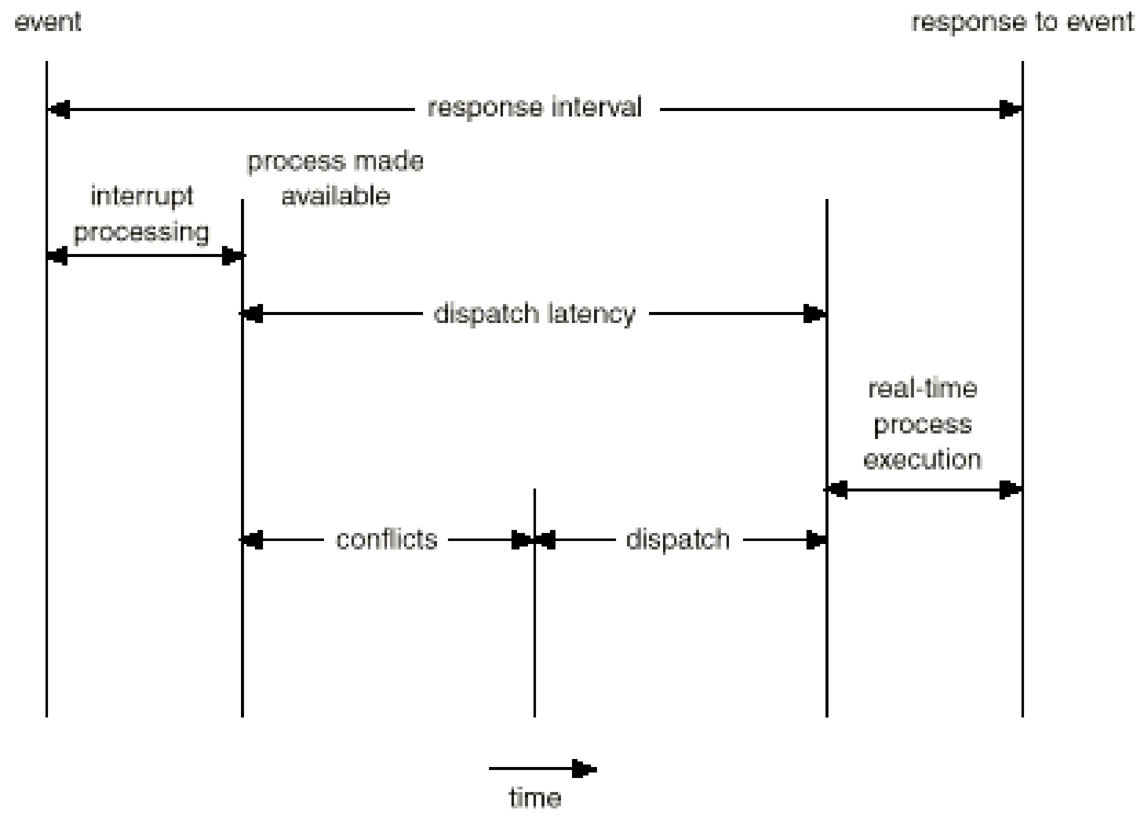
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available.
- *Homogeneous processors* within a multiprocessor.
- *Load sharing*
- *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing.

Real-Time Scheduling

- *Hard real-time* systems – required to complete a critical task within a guaranteed amount of time.
- *Soft real-time* computing – requires that critical processes receive priority over less fortunate ones.

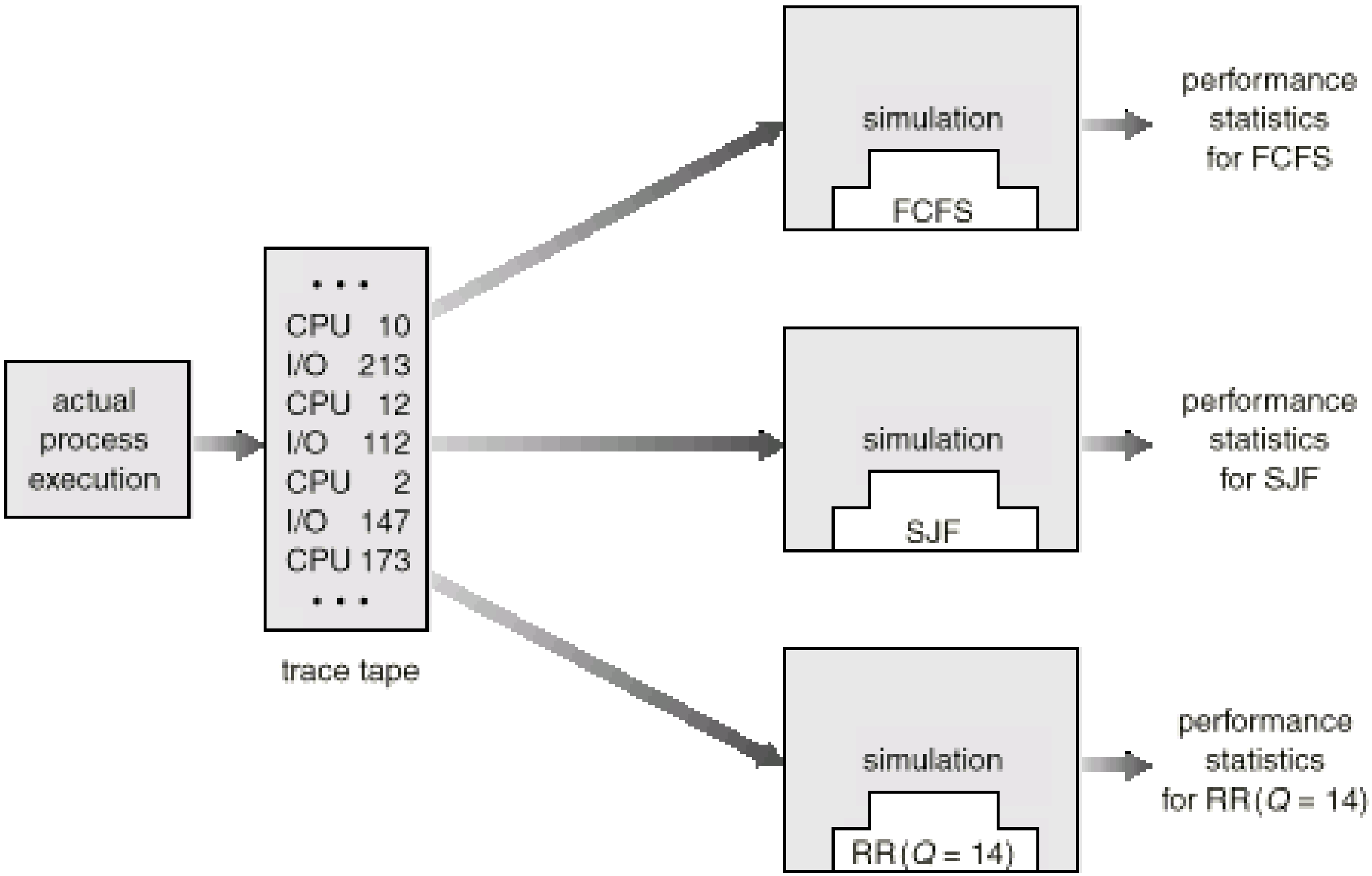
Dispatch Latency



Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload.
- Queuing models
- Implementation

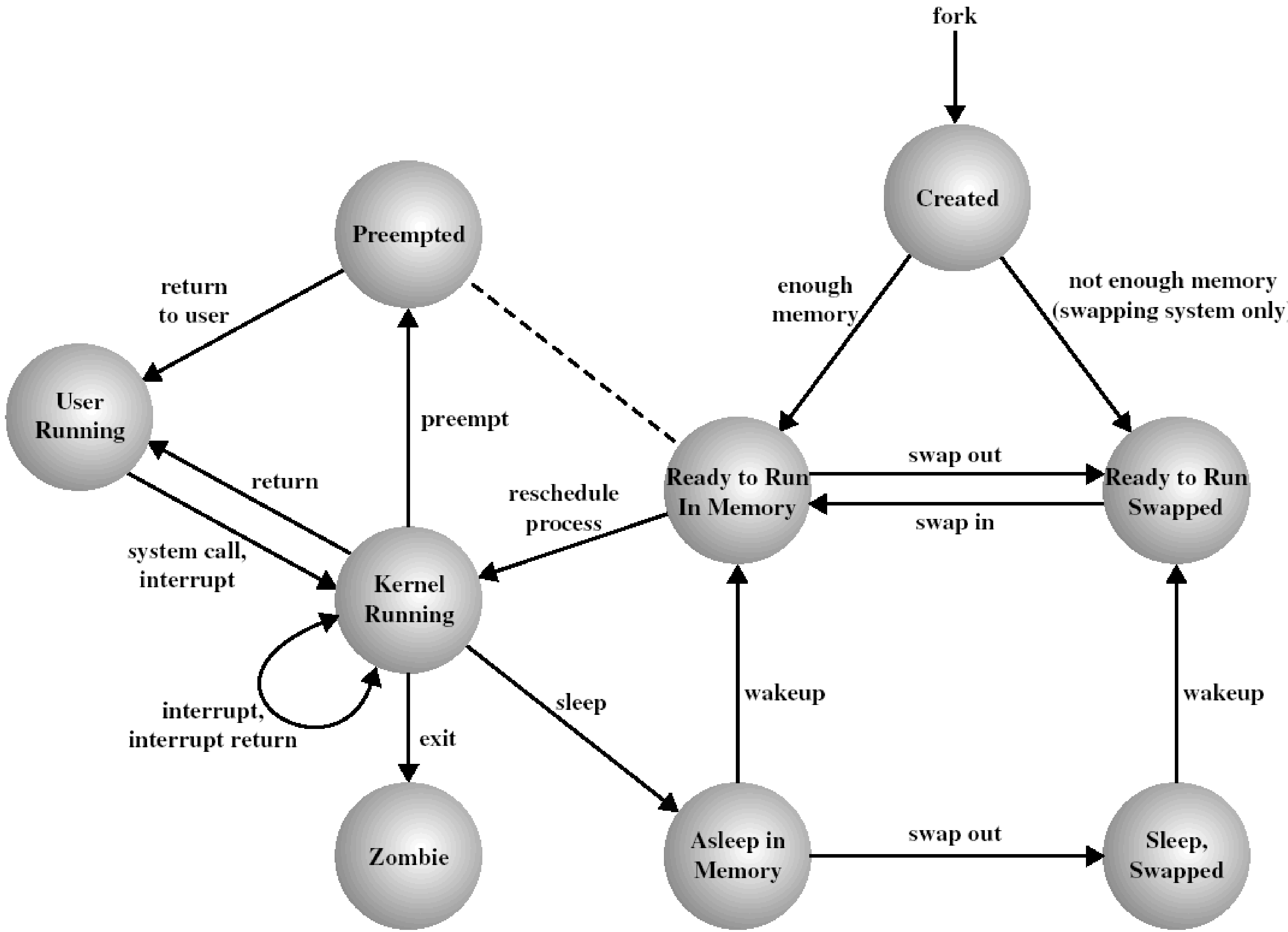
Evaluation of CPU



Case Study
Unix Process Management

UNIX Process States

User Running	Executing in user mode.
Kernel Running	Executing in kernel mode.
Ready to Run, in Memory	Ready to run as soon as the kernel schedules it.
Asleep in Memory	Unable to execute until an event occurs; process is in main memory (a blocked state).
Ready to Run, Swapped	Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
Sleeping, Swapped	The process is awaiting an event and has been swapped to secondary storage (a blocked state).
Preempted	Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
Created	Process is newly created and not yet ready to run.
Zombie	Process no longer exists, but it leaves a record for its parent process to collect.


fork
Created
enough memory
not enough memory (swapping system only)
Preempted
return to user
User Running
preempt
Ready to Run In Memory
swap out
Ready to Run Swapped
swap in
return
reschedule process
system call, interrupt
Kernel Running
interrupt, interrupt return
exit
sleep
wakeup
wakeup
Zombie
Asleep in Memory
swap out
Sleep, Swapped

Zombies

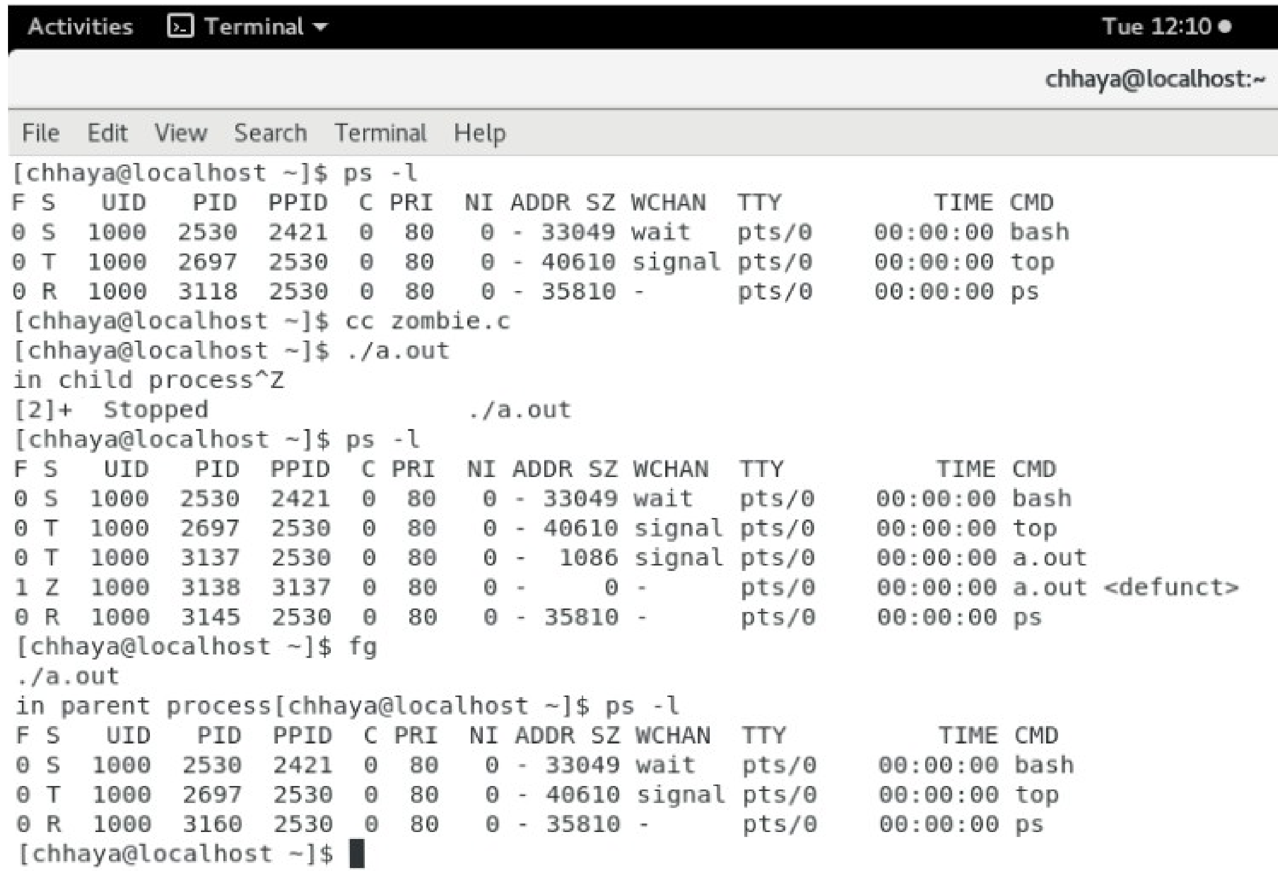
- A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process.
- A child process always first becomes a zombie before being removed from the process table.
- The parent process reads the exit status of the child process which reaps off the child process entry from the process table.
- zombie is not really a process as it has terminated but the system retains an entry in the process table for the non-existing child process.
- A zombie is put to rest when the parent finally executes a wait().

A C program to demonstrate Zombie Process.

```
// Child becomes Zombie as parent is sleeping
// when child process exits.
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // Fork returns process id
    // in parent process
    pid_t child_pid = fork();

    // Parent process
    if (child_pid > 0)
        sleep(50);

    // Child process
    else
        exit(0);

    return 0;
}
```

A C program to demonstrate Zombie Process.

```c
// Child becomes Zombie as parent is sleeping
// when child process exits.
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // Fork returns process id
    // in parent process
    pid_t child_pid = fork();

    // Parent process
    if (child_pid > 0)
        sleep(50);

    // Child process
    else
        exit(0);

    return 0;
}
```



```
Activities   Terminal ▾                                   Tue 12:10 ●
                                                  chhaya@localhost:~
File  Edit  View  Search  Terminal  Help
[chhaya@localhost ~]$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2530  2421  0  80   0 - 33049 wait   pts/0    00:00:00 bash
0 T  1000  2697  2530  0  80   0 - 40610 signal pts/0    00:00:00 top
0 R  1000  3118  2530  0  80   0 - 35810 -      pts/0    00:00:00 ps
[chhaya@localhost ~]$ cc zombie.c
[chhaya@localhost ~]$ ./a.out
in child process^Z
[2]+  Stopped                 ./a.out
[chhaya@localhost ~]$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2530  2421  0  80   0 - 33049 wait   pts/0    00:00:00 bash
0 T  1000  2697  2530  0  80   0 - 40610 signal pts/0    00:00:00 top
0 T  1000  3137  2530  0  80   0 -  1086 signal pts/0    00:00:00 a.out
1 Z  1000  3138  3137  0  80   0 -     0 -      pts/0    00:00:00 a.out <defunct>
0 R  1000  3145  2530  0  80   0 - 35810 -      pts/0    00:00:00 ps
[chhaya@localhost ~]$ fg
./a.out
in parent process[chhaya@localhost ~]$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  2530  2421  0  80   0 - 33049 wait   pts/0    00:00:00 bash
0 T  1000  2697  2530  0  80   0 - 40610 signal pts/0    00:00:00 top
0 R  1000  3160  2530  0  80   0 - 35810 -      pts/0    00:00:00 ps
[chhaya@localhost ~]$
```

Orphans

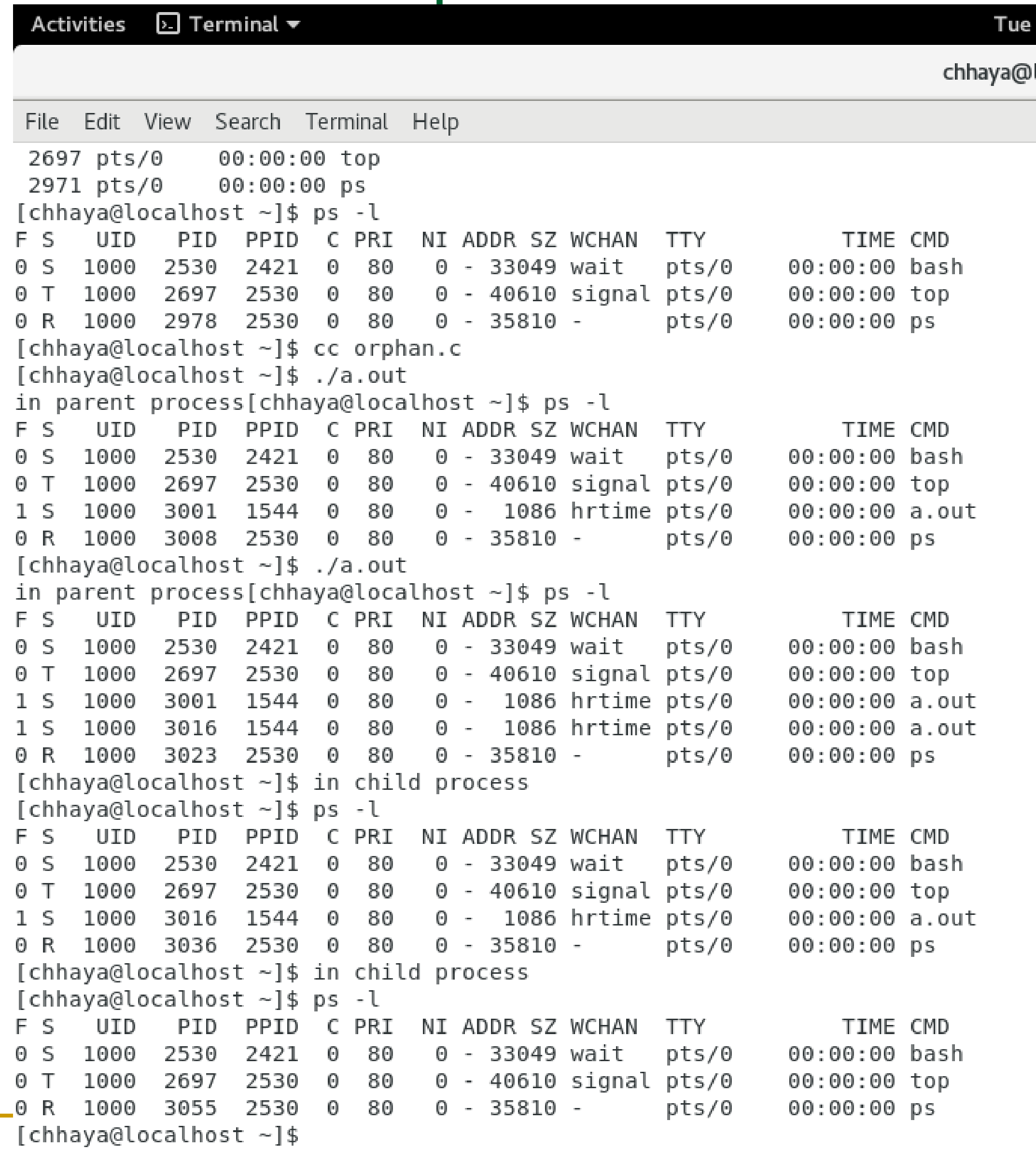
- A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.
- When a parent terminates, orphans and zombies are adopted by the init process (process-id -1) of the system.

A C program to demonstrate Orphan Process.

```
// Parent process finishes execution while the
// child process is running. The child process
// becomes orphan.
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    // Create a child process
    int pid = fork();

    if (pid > 0)
        printf("in parent process");

    // Note that pid is 0 in child process
    // and negative if fork() fails
    else if (pid == 0)
    {
        sleep(30);
        printf("in child process");
    }
return 0;
}
```

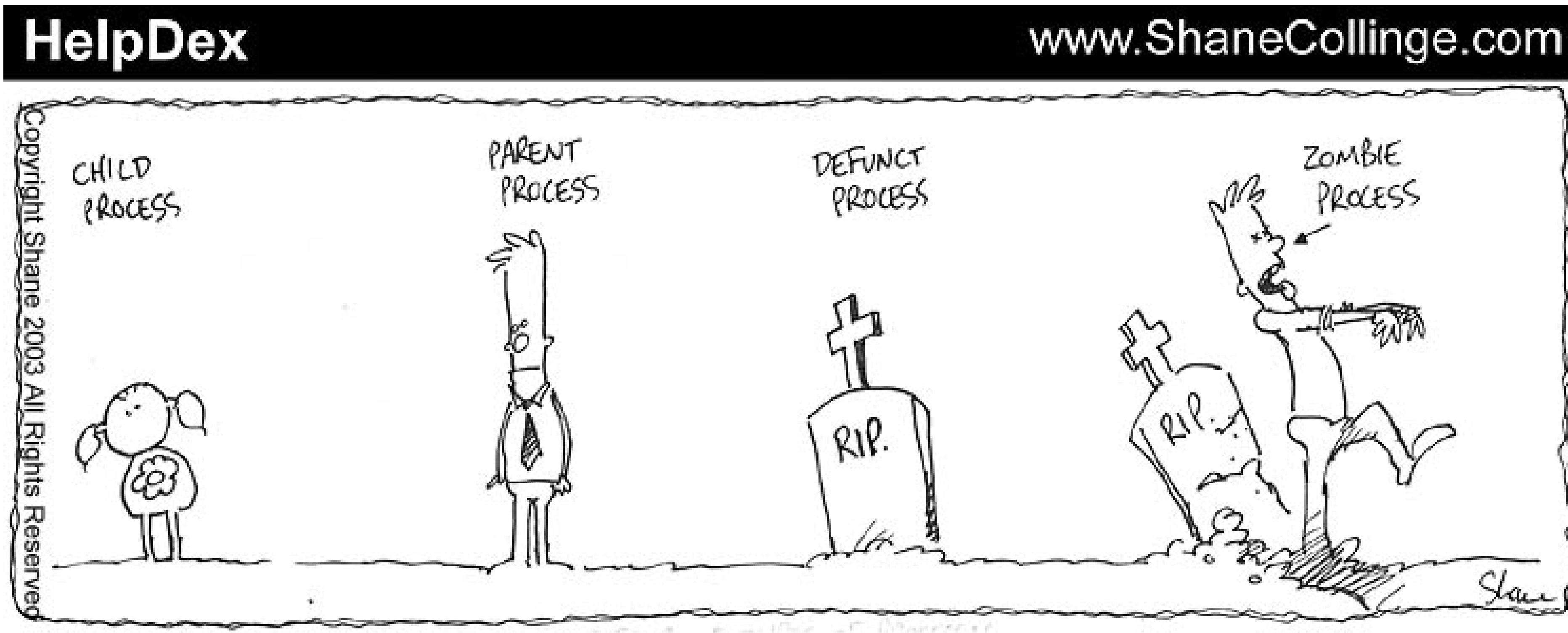
A C program to demonstrate Orphan Process.

```c
// Parent process finishes execution while the
// child process is running. The child process
// becomes orphan.
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
   // Create a child process
   int pid = fork();

   if (pid > 0)
      printf("in parent process");

   // Note that pid is 0 in child process
   // and negative if fork() fails
   else if (pid == 0)
   {
      sleep(30);
      printf("in child process");
   }
return 0;
}
```



Daemons

- Daemons are server processes that run continuously.
- Most of the time, they are initialized at system startup and then wait in the background until their service is required.
- A typical example is the networking daemon, xinetd, which is started in almost every boot procedure. After the system is booted, the network daemon just sits and waits until a client program, such as an FTP client, needs to connect.


HelpDex
www.ShaneCollinge.com
CHILD PROCESS
PARENT PROCESS
DEFUNCT PROCESS
R.I.P.
ZOMBIE PROCESS
R.I.P.
Copyright Shane 2003 All Rights Reserve