Chapter 04 Figures and codes

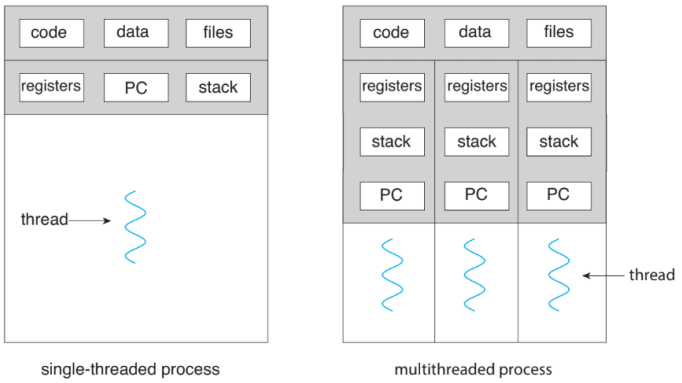

Figure 4.1 Single-threaded and multithreaded processes.

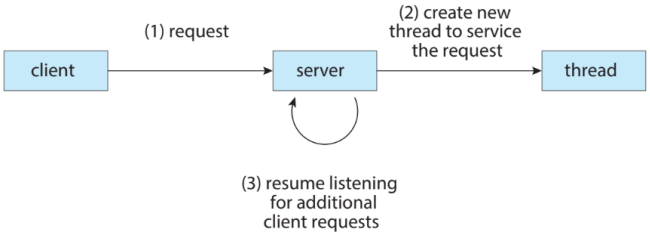

Figure 4.2 Multithreaded server architecture.

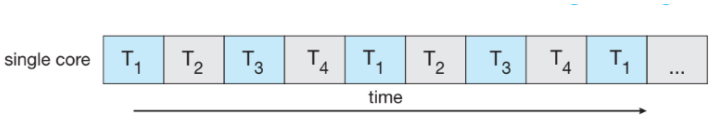

Figure 4.3 Concurrent execution on a single-core system.

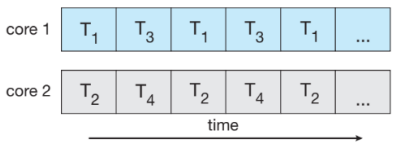

Figure 4.4 Parallel execution on a multicore system.

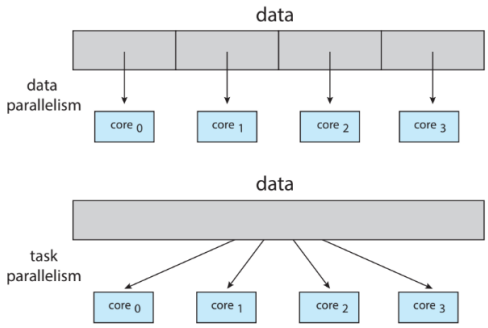

Figure 4.5 Data and task parallelism.

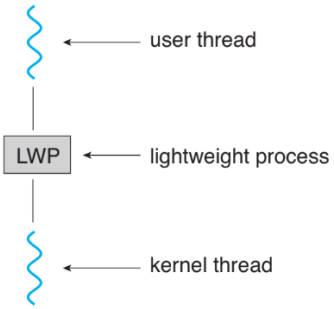

Figure 4.20 Lightweight process (LWP).

Chapter 06 Figures and codes

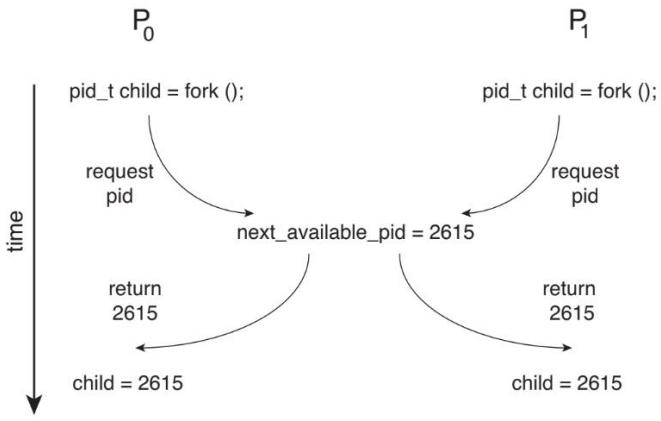

Figure 6.2 Race condition when assigning a pid.

Chapter 05 Figures and codes

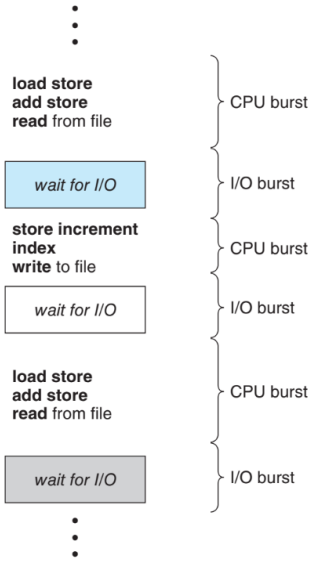

Figure 5.1 Alternating sequence of CPU and I/O bursts.

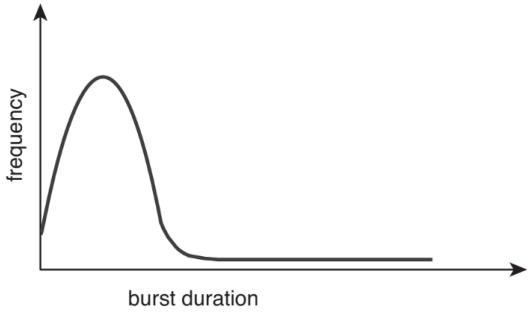

Figure 5.2 Histogram of CPU-burst durations.

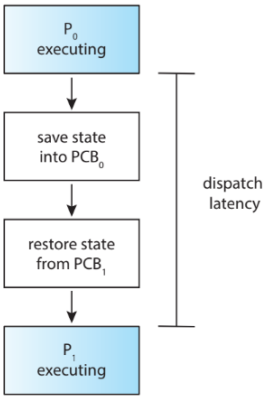

Figure 5.3 The role of the dispatcher.

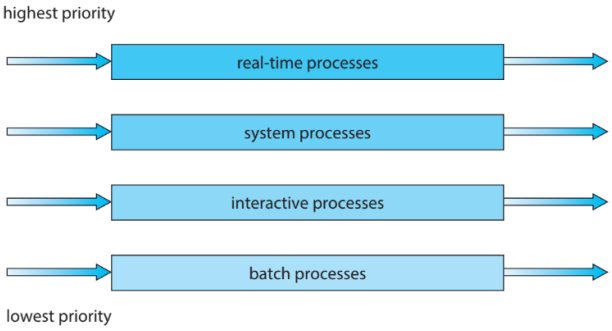

Figure 5.8 Multilevel queue scheduling.

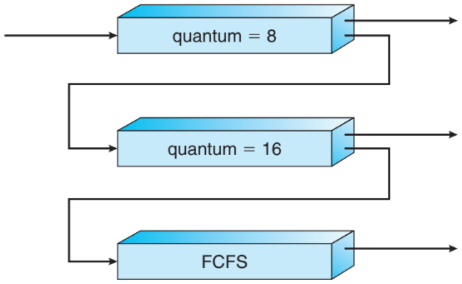

Figure 5.9 Multilevel feedback queues.

Figure:25

```c
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[])
{
    int i, policy;
    pthread_t tid[NUM THREADS];
    pthread_attr_t attr;

    //get the default attributes
    pthread_attr_init(&attr);

    // get the current scheduling policy
    if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
        fprintf(stderr, "Unable to get policy.\n");
    else
    {
        if (policy == SCHED OTHER)
            printf("SCHED OTHER\n");
        else if (policy == SCHED RR)
            printf("SCHED RR\n");
        else if (policy == SCHED FIFO)
            printf("SCHED FIFO\n");
    }

    //set the scheduling policy - FIFO, RR, or OTHER
    if (pthread_attr_setschedpolicy(&attr, SCHED FIFO) != 0)
        fprintf(stderr, "Unable to set policy.\n");

    //create the threads
    for (i = 0; i < NUM THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);

    //now join on each thread
    for (i = 0; i < NUM THREADS; i++)
        pthread_join(tid[i], NULL);
}
// Each thread will begin control in this function
void *runner(void *param)
{
    //  do some work ...
    pthread_exit(0);
}
```

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int sum = 0; // this data is shared by the thread(s)
void *runner(void *parameters)
{ // The thread will begin control in this funtion
    int i, upper = *((int *)parameters);
    if (upper > 0)

        for (i = 1; i <= upper; i++)
            sum += i;
    pthread_exit(0);
}

int main(int argc, char *argv[])
{
    pthread_t threadID;        // thread identifier
    pthread_attr_t attributes; // set attributes for the thread
    int num = 1000;

    pthread_attr_init(&attributes); // get the default attributes
    pthread_create(&threadID, &attributes, runner, (void *)&num); // create the thread
    pthread_join(threadID, NULL); // now wait for the thread to exit
    printf("sum=%d\n", sum);
    exit(0);
}
```

```c
#include <stdio.h>
#include <pthread.h>

static volatile int counter = 0;
void *mythread(void *arg)
{
    printf("%s: begin\n", (char *)arg);
    int i;
    // int counter = 0;
    for (i = 0; i < 1e7; i++)
    {
        counter = counter + 1;

        printf("%s: done. Counter = %d\n", (char *)arg, counter);
        return NULL;
    }
}
int main(int argc, char *argv[])
{
    pthread_t p1, p2;

    printf("main: begin (counter = %d)\n", counter);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 4
#define ARRAY_SIZE 1000000

int global_array[ARRAY_SIZE]; // Shared array

// Function to initialize the array with random values
void initialize_array()
{
    for (int i = 0; i < ARRAY_SIZE; ++i)
    {
        global_array[i] = rand() % 1000;
    }
}
// Function to find the sum of elements in a portion of the array
void *sum_array(void *arg)
{
    int thread_id = *((int *)arg);
    int start = thread_id * (ARRAY_SIZE / NUM_THREADS);
    int end = start + (ARRAY_SIZE / NUM_THREADS);
    int sum = 0;

    // Calculate the sum of elements in the assigned portion of the array
    for (int i = start; i < end; ++i)
    {
        sum += global_array[i];
    }
    return (void *)(long)sum; // Return the sum as a void pointer
}

int main()
{
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    void *thread_results[NUM_THREADS];
    long total_sum = 0;

    // Initialize the array with random values
    initialize_array();

    // Create threads to compute the sum of array elements
    for (int i = 0; i < NUM_THREADS; ++i)
    {
        thread_args[i] = i;
        pthread_create(&threads[i], NULL, sum_array, (void *)&thread_args[i]);
    }
    // Join threads and collect results
    for (int i = 0; i < NUM_THREADS; ++i)
    {
        pthread_join(threads[i], &thread_results[i]);
        total_sum += (long)thread_results[i]; // Accumulate the partial sums
```

```
    }
    printf("Total sum of array elements: %ld\n", total_sum);

return 0;
}
```