****

1. Demonstrate fork() system call. Let the parent process display its pid, ppid and a message 'I'm the parent'. Also let the child display its pid, ppid and a message 'I'm the child'.

2. Let the parent fork and let the child execute ls command. Observe the result with and without having wait() system call in the parent.

3. Let the parent create 4 children. Make them execute ls, ls –l, pwd and date commands. (One child executes one command.)

4. Create a child through fork(). Let the child generate the fibonacci series (1, 1, 2, 3, 5, 8...) upto n numbers. The value of n has to passed as a command line argument.

5. Let the parent create a child using fork(). Let the parent geneate an odd series upto n numbers (1, 3, 5, 7, 9...) and let the child create an even series upto n numbers (0, 2, 4, 6, 8 ...). (i) Pass the value of n as a command line argument. (ii) Read the value of n from the user.

6. Achieve the same as in Q(5) by using exec() system calls.

7. Let the parent create 2 children which work on a common sample file. Let one child count the total number of lines in the file while the other counts the total number of characters in the same file. Provide the filename as a command line argument.

8. Let the parent create 2 children which work on a common sample file. Let one child convert all lowercase to uppercase in the file while the other counts the total number of character 'a' s in the same file. Provide the filename as a command line argument.

9. Let the parent create 2 children which work on a common sample file. Let one child count the total number of words in the file. Let the other child invert the case of alphabets in the file. Provide the filename as a command line argument.

10. Solve Q-3.5 from text (7th edition)

I ES – "A" & "B" Sections – (Sample representative questions for RTOS Jan 2009 lab.)

1. Fork a separate process and execute the ls command (fork and execlp system commands to be used)
2. Implement the producer-consumer problem using a Circular Buffer.
3. Demonstrate all the shared memory system calls by writing suitable program.

Note: create locking mechanism using either pthread mutex's or pthread semaphore's for the critical section code in the following problem wherever needed.

4. Create two threads in a main program, let the first thread execute a function to display a message namely "this is thread one" , similarly let the second thread displays "this is thread two'.
5. Create a common shared memory area where in one thread writes a string termed "Hello There"; the second thread reads the string and displays it on the screen. Also the second string converts all lower case to upper case and vice versa in the shared memory. Next the first thread will read this from the shared memory and will output the same to the screen.
6. Write a c program using fork system call which generates Fibonacci sequence in the child process. The number of sequence is to be provided in the command line, ex: if 5 is provided the first 5 numbers of the Fibonacci series will be output by the child process. Perform necessary error checking such that a nonnegative number is not to be taken.
7. Next create a common shared memory area where in one thread writes a sequence of numbers, the second thread reads the numbers, calculates their sum and displays the result to the screen.
8. Declare a matrix of size m x n , and populate it. (Fill it) Create m threads such that each thread calculates the ith row sum. Let the main thread display each row sum result as well as the total sum.
9. create and save a sample text (a file containing any text) in your current working directory . Next write a program which creates two threads such that one thread counts the total number of words present in the file and the other thread changes all upper case characters to lower case.
10. Modify the above program to create a third thread too; the third thread writes "three threads have visited this file" at the very end of the file.
11. Implement peterson's solution where two threads of a process update a common global variable in their respective critical sections. The first thread increment the variable value by 100 and second thread decrements it by 75.
12. Show the implementation of reader's writer's problem using mutex or semaphore. The reader just reads a global variable and will display it on the screen; the writer will increment the variable by a value 35. Create two readers and one writer.

13. modify the above Qn. Such that the readers read from a common file and the writer updates the value in file. Let the initial value in file be be 25.
14. Implement Dining Philosopher's problem either semaphores or mutex's.
15. Implement the producer-consumer problem of buffer 5, let the critical section be implemented using either semaphores or mutex's.
16. Implement the following scheduling algorithms
    a. FCFS
    b. SJF
    c. SJF with Preemption.
    d. Priority.
    e. Round Robin.

    Read the number of processes, arrival time of the process, their burst times , their priorities, time slice and any other details as per requirement. Output the ***AVG. waiting time*** and ***AVG. turn around time*** in each case.

17. Implement the Banker's algorithm. Read the number of processes, the Max Resource type vector, the Need matrix and the current allocation matrix. Show all the intermediate steps of resource allocation to the processes and determine whether the system is in safe state or not.

(Sample representative questions for RTOS 2009 Aug lab.)

Note:- Familiarity with manipulating and working with numbers, characters, character strings, arrays, matrices, files is expected.

1. Forking a child process and making the child process execute a program.

The program may be related to working with array, matrix, files or string manipulations. (eg:- finding row sum/column sum of a matrix, changing cases of a character string,

counting for character occurance in a string, finding the sum of digits of a given number, counting for words in a file, counting for white spaces(blanks, tabs or newlines) in a file etc.)

Hint:- Write the program(code of the child process) in a separate file, compile and create its exe. Invoke this file name in the ***exec*** command.

2. Creating one or more threads and making the different threads perform different computations. The threads may be working with numbers, arrays, matrices, files or performing string manipulations etc. **Hint:-** Use pthread library function for creating threads and running them.

3. Create a common shared memory area where in one process writes a string termed "Hello There"; the second process reads the string and displays it on the screen converting all characters to upper case. Hint:- Run the first process first and the second process next. **Hint:-** Use POSIX shared memory API to create and use shared memory.

4. Show the implementation of the ***first reader – writer*** problem using pthread mutex. Create 2 reader threads and one writer thread. One reader thread reads the file and displays the characters , each one as a lower case on the screen. The other reader thread too reads the file and displays the characters , each one as a upper case on the screen. The writer thread should append a string termed "hello" to the end of the file. Execute the program and observe the result.

5. Implement the producer-consumer problem using a circular buffer. Implement the critical section using pthread mutex's. Run the program and observe the result.

6. Write a program which is a modification of the above question by also providing a common counter for the producer and consumer, which tracks the total number of items in the buffer. Display it everytime the producer thread or the consumer executes.

7. Implement the FCFS scheduling algorithm. Read the number of processes and their burst times . Assume that they all arrive at time zero. Output the ***AVG. waiting time*** and ***AVG. turn around time*** of each process.

8. Implement the Round Robin scheduling algorithm. Read the number of processes, time slice and their burst times . Assume that they all arrive at time zero. Output the ***AVG. waiting time*** and ***AVG. turn around time*** of each process.

9. Implement the SJF scheduling algorithms. Read the number of processes and their burst times . Assume that they all arrive at time zero. Output the ***AVG. waiting time*** and ***AVG. turn around time*** of each process.

10. Implement the Priority scheduling algorithms. Read the number of processes, their priorities and their burst times . Assume that they all arrive at time zero. Output the ***AVG. waiting time*** and ***AVG. turn around time*** of each process.

****

All programs compile using:

gcc filename.c -o output

✅ Q1 — Demonstrate fork(): parent + child prints pid, ppid and message

```
/*
Q1. Demonstrate fork() system call.
Let the parent process display its pid, ppid and a message ‘I’m the parent’.
Also let the child display its pid, ppid and a message ‘I’m the child’.
*/

#include <stdio.h>
#include <unistd.h>

int main() {
   pid_t pid = fork();

   if (pid < 0) {
      printf("Fork failed\n");
   }
   else if (pid == 0) {
      // Child
      printf("Child Process:\nPID = %d\nPPID = %d\nI'm the child\n", getpid(), getppid());
   }
   else {
      // Parent
      printf("Parent Process:\nPID = %d\nPPID = %d\nI'm the parent\n", getpid(), getppid());
   }

   return 0;
}
```

✅ Q2 — Child executes ls, compare with/without wait()

```
/*
Q2. Let the parent fork and let the child execute ls command.
Observe the result with and without having wait() system call in the parent.
*/

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
  pid_t pid = fork();

  if (pid == 0) {
    execl("/bin/ls", "ls", NULL);
    printf("execl failed\n");
  }
  else {
    // Uncomment wait() to see synchronized behavior:
    // wait(NULL);
    printf("Parent continues...\n");
  }

  return 0;
}
```

✅ Q3 — Parent creates 4 children executing different commands

```
/*
Q3. Let the parent create 4 children.
Make them execute ls, ls –l, pwd and date commands.
(One child executes one command.)
```

```
*/

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
  char *cmds[4][3] = {
    {"/bin/ls", "ls", NULL},
    {"/bin/ls", "ls", "-l"},
    {"/bin/pwd", "pwd", NULL},
    {"/bin/date", "date", NULL}
  };

  for (int i = 0; i < 4; i++) {
    pid_t pid = fork();

    if (pid == 0) {
      execvp(cmds[i][0], cmds[i]);
      printf("exec failed\n");
      return 0;
    }
  }

  for (int i = 0; i < 4; i++) wait(NULL);

  return 0;
}
```

✅ **Q4 — Child generates Fibonacci up to n (command-line argument)**

```
/*
```

Q4. Create a child through fork().

Let the child generate the fibonacci series (1,1,2,3,5,8...) upto n numbers.

The value of n has to be passed as a command line argument.

*/

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf("Usage: ./a.out n\n");
    return 0;
  }

  int n = atoi(argv[1]);
  pid_t pid = fork();

  if (pid == 0) {
    int a = 1, b = 1, c;
    printf("Fibonacci Series:\n");
    if (n >= 1) printf("1 ");
    if (n >= 2) printf("1 ");
    for (int i = 3; i <= n; i++) {
      c = a + b;
      printf("%d ", c);
      a = b;
      b = c;
    }
    printf("\n");
  }
```

```
    return 0;
}
```

✅ **Q5 — Parent prints odd series, child prints even series (n from argv & user)**

Version (i) — n from command line

```
/*
Q5(i). Let the parent create a child using fork().
Parent generates odd series upto n numbers (1,3,5,7...).
Child generates even series upto n numbers (0,2,4,6...).
Pass n as command line argument.
*/

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: ./a.out n\n");
        return 0;
    }

    int n = atoi(argv[1]);
    pid_t pid = fork();

    if (pid == 0) {
        printf("Even series:\n");
        for (int i = 0; i < n; i++)
            printf("%d ", i * 2);
        printf("\n");
```

```
    }
    else {
        printf("Odd series:\n");
        for (int i = 0; i < n; i++)
            printf("%d ", 1 + (i * 2));
        printf("\n");
    }

    return 0;
}
```

Version (ii) — n from user input

```
/*
Q5(ii). Same as Q5(i), but read n from user instead of command line.
*/

#include <stdio.h>
#include <unistd.h>

int main() {
    int n;
    printf("Enter n: ");
    scanf("%d", &n);

    pid_t pid = fork();

    if (pid == 0) {
        printf("Even series:\n");
        for (int i = 0; i < n; i++)
            printf("%d ", i * 2);
        printf("\n");
```

```
    }
    else {
        printf("Odd series:\n");
        for (int i = 0; n > i; i++)
            printf("%d ", 1 + (i * 2));
        printf("\n");
    }

    return 0;
}
```

✅ Q6 — Same as Q5 but using exec()

You need two programs:
main program spawns two children, each executing a different program.

Program: parent.c

```
/*
Q6. Achieve same as Q5 using exec() system calls.
Parent creates two children.
One executes even.c, other executes odd.c
*/

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: ./parent n\n");
        return 0;
    }
```

```
    pid_t pid = fork();

    if (pid == 0) {
        execl("./even", "even", argv[1], NULL);
    }
    else {
        pid = fork();
        if (pid == 0) {
            execl("./odd", "odd", argv[1], NULL);
        }
    }

    wait(NULL);
    wait(NULL);
    return 0;
}
```

even.c

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

int main(int argc, char *argv[]) {
    int n = atoi(argv[1]);
    for (int i = 0; i < n; i++)
        printf("%d ", i * 2);
    printf("\n");
    return 0;
}
```

odd.c

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

```
int main(int argc, char *argv[]) {
   int n = atoi(argv[1]);
   for (int i = 0; i < n; i++)
      printf("%d ", 1 + (i * 2));
   printf("\n");
   return 0;
}
```

✅ **Q7 — One child counts lines, one counts characters**

```
/*
Q7. Parent creates 2 children working on same file.
One child counts total number of lines.
Other child counts total number of characters.
File name is command line argument.
*/

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
   if (argc != 2) {
      printf("Usage: ./a.out filename\n");
      return 0;
   }

   char *file = argv[1];
   pid_t pid = fork();

   if (pid == 0) {
      FILE *fp = fopen(file, "r");
```

```
    int lines = 0, ch;

    while ((ch = fgetc(fp)) != EOF)
      if (ch == '\n') lines++;

    printf("Total lines = %d\n", lines);
    fclose(fp);
  }
  else {
    pid = fork();
    if (pid == 0) {
      FILE *fp = fopen(file, "r");
      int count = 0;

      while (fgetc(fp) != EOF)
        count++;

      printf("Total characters = %d\n", count);
      fclose(fp);
    }
  }

  wait(NULL);
  wait(NULL);
  return 0;
}
```

✅ **Q8 — One child converts lowercase → uppercase, other counts 'a'**

```
/*
Q8. Parent creates 2 children for same file.
Child1 converts all lowercase to uppercase.
```

```
Child2 counts total number of 'a' characters.
Filename from command line.
*/

#include <stdio.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf("Usage: ./a.out filename\n");
    return 0;
  }

  char *file = argv[1];
  pid_t pid = fork();

  if (pid == 0) {
    FILE *fp = fopen(file, "r");
    FILE *tmp = fopen("upper_output.txt", "w");
    int ch;

    while ((ch = fgetc(fp)) != EOF)
      fputc(toupper(ch), tmp);

    printf("Converted to uppercase → upper_output.txt\n");
    fclose(fp);
    fclose(tmp);
  }
  else {
```

```
    pid = fork();

    if (pid == 0) {
        FILE *fp = fopen(file, "r");
        int count = 0, ch;

        while ((ch = fgetc(fp)) != EOF)
            if (ch == 'a' || ch == 'A') count++;

        printf("Total 'a' characters = %d\n", count);
        fclose(fp);
    }
  }

  wait(NULL);
  wait(NULL);
  return 0;
}
```

✅ **Q9 — One child counts words, other inverts case of alphabets**

```
/*
Q9. Parent creates 2 children for same file.
Child1 counts total number of words.
Child2 inverts the case of all alphabets.
Filename from command line.
*/

#include <stdio.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/wait.h>
```

```
int main(int argc, char *argv[]) {
   if (argc != 2) {
      printf("Usage: ./a.out filename\n");
      return 0;
   }

   char *file = argv[1];
   pid_t pid = fork();

   if (pid == 0) {
      FILE *fp = fopen(file, "r");
      int words = 0, inword = 0, ch;

      while ((ch = fgetc(fp)) != EOF) {
         if (isspace(ch))
            inword = 0;
         else if (!inword) {
            inword = 1;
            words++;
         }
      }

      printf("Total words = %d\n", words);
      fclose(fp);
   }
   else {
      pid = fork();
      if (pid == 0) {
         FILE *fp = fopen(file, "r");
         FILE *tmp = fopen("invert_output.txt", "w");
```

```
        int ch;

        while ((ch = fgetc(fp)) != EOF) {
          if (isupper(ch)) fputc(tolower(ch), tmp);
          else if (islower(ch)) fputc(toupper(ch), tmp);
          else fputc(ch, tmp);
        }

        printf("Case inverted → invert_output.txt\n");
        fclose(fp);
        fclose(tmp);
      }
    }

    wait(NULL);
    wait(NULL);
    return 0;
}
```

Compile each with gcc -pthread qX.c -o qX (unless noted otherwise). Read the comments at the top of each file for any special compile/run instructions.

Q1 — fork + execlp

```
/*
Q1.
Fork a separate process and execute the ls command (fork and execlp system commands to be used)
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include <sys/wait.h>


int main(void) {
   pid_t pid = fork();
   if (pid < 0) {
      perror("fork");
      return 1;
   }
   if (pid == 0) {
      // child
      printf("Child: executing ls via execlp()\n");
      execlp("ls", "ls", "-la", (char *)NULL); // replace child with ls
      perror("execlp"); // only runs on error
      _exit(1);
   } else {
      // parent
      int status;
      waitpid(pid, &status, 0);
      printf("Parent: child finished with status %d\n", WEXITSTATUS(status));
   }
   return 0;
}
```

Compile & run:

```
gcc q1.c -o q1
./q1
```

Q2 — Producer-Consumer using Circular Buffer (pthreads, condition variables)

```
/*
Q2.
Implement the producer-consumer problem using a Circular Buffer.
Uses pthreads, mutex and condition variables.
```

```
*/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define BUFSIZE 5
int buffer[BUFSIZE];
int in = 0, out = 0, count = 0;

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;

void *producer(void *arg) {
  int id = *(int *)arg;
  for (int i = 1; i <= 15; i++) {
    pthread_mutex_lock(&m);
    while (count == BUFSIZE) pthread_cond_wait(&not_full, &m);
    buffer[in] = i;
    in = (in + 1) % BUFSIZE;
    count++;
    printf("Producer %d produced %d (count=%d)\n", id, i, count);
    pthread_cond_signal(&not_empty);
    pthread_mutex_unlock(&m);
    usleep(100000);
  }
  return NULL;
}
```

```
void *consumer(void *arg) {
   int id = *(int *)arg;
   for (int i = 1; i <= 15; i++) {
      pthread_mutex_lock(&m);
      while (count == 0) pthread_cond_wait(&not_empty, &m);
      int val = buffer[out];
      out = (out + 1) % BUFSIZE;
      count--;
      printf("Consumer %d consumed %d (count=%d)\n", id, val, count);
      pthread_cond_signal(&not_full);
      pthread_mutex_unlock(&m);
      usleep(150000);
   }
   return NULL;
}

int main(void) {
   pthread_t p, c;
   int pid = 1, cid = 1;
   pthread_create(&p, NULL, producer, &pid);
   pthread_create(&c, NULL, consumer, &cid);
   pthread_join(p, NULL);
   pthread_join(c, NULL);
   return 0;
}
```

Compile:

```
gcc q2.c -pthread -o q2
./q2
```

Q3 — Demonstrate System V Shared Memory calls (shmget, shmat, shmdt, shmctl) + lock with pthread mutex

```
/*
Q3.
Demonstrate all the shared memory system calls by writing suitable program.
Note: create locking mechanism using either pthread mutex's or pthread semaphore's
for the critical section code in the following problem wherever needed.

This program:
 - creates a System V shared memory segment
 - attaches to it
 - writes a string to it (parent)
 - forks a child which reads it
 - demonstrates shmdt and shmctl (IPC_STAT and IPC_RMID)
 - uses a pthread mutex put inside the shared memory (with process-shared attribute)
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#include <unistd.h>
#include <sys/wait.h>
#include <pthread.h>
#include <errno.h>

#define SHM_KEY 0x1234
#define SHM_SIZE 4096

struct shm_area {
   pthread_mutexattr_t mattr; // store attributes (for demonstration)
   pthread_mutex_t mutex;     // process-shared mutex
```

```
    char message[256];
};

int main(void) {
    int shmid = shmget(SHM_KEY, SHM_SIZE, IPC_CREAT | 0666);
    if (shmid == -1) { perror("shmget"); return 1; }

    printf("shmget -> shmid=%d\n", shmid);

    struct shm_area *shm = (struct shm_area *) shmat(shmid, NULL, 0);
    if (shm == (void *)-1) { perror("shmat"); return 1; }

    // Initialize process-shared mutex in shared memory
    pthread_mutexattr_t mattr;
    if (pthread_mutexattr_init(&mattr) != 0) { perror("pthread_mutexattr_init"); }
    if (pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED) != 0) {
perror("pthread_mutexattr_setpshared"); }
    // copy attr into shm (not strictly necessary but shown)
    shm->mattr = mattr; // struct copy (safe enough here)
    if (pthread_mutex_init(&shm->mutex, &mattr) != 0) { perror("pthread_mutex_init"); }

    // Parent writes into shared memory
    pthread_mutex_lock(&shm->mutex);
    strncpy(shm->message, "Hello from parent via shared memory", sizeof(shm->message));
    pthread_mutex_unlock(&shm->mutex);

    pid_t pid = fork();
    if (pid < 0) { perror("fork"); }
    else if (pid == 0) {
        // child: attach and read
        struct shm_area *child_shm = (struct shm_area *) shmat(shmid, NULL, 0);
```

```
    if (child_shm == (void *)-1) { perror("child shmat"); _exit(1); }
    pthread_mutex_lock(&child_shm->mutex);
    printf("Child read message: \"%s\"\n", child_shm->message);
    pthread_mutex_unlock(&child_shm->mutex);

    // detach and exit
    if (shmdt(child_shm) == -1) perror("child shmdt");
    _exit(0);
  } else {
    wait(NULL);
    // demonstrate IPC_STAT via shmctl
    struct shmid_ds ds;
    if (shmctl(shmid, IPC_STAT, &ds) == -1) perror("shmctl IPC_STAT");
    else {
      printf("shm_segsz=%zu, shm_nattch=%ld\n", ds.shm_segsz, (long)ds.shm_nattch);
    }

    // detach from shared memory
    if (shmdt(shm) == -1) perror("shmdt");

    // remove the shared memory segment
    if (shmctl(shmid, IPC_RMID, NULL) == -1) perror("shmctl IPC_RMID");
    else printf("shared memory removed (IPC_RMID)\n");
  }

  return 0;
}
```

Compile:

```
gcc q3.c -pthread -o q3
./q3
```

Q4 — Two threads printing messages

```
/*
Q4.
Create two threads in a main program, let the first thread execute a function to display a message
namely "this is thread one" , similarly let the second thread displays "this is thread two".
*/

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

void *thread1(void *arg) {
   (void)arg;
   printf("this is thread one\n");
   return NULL;
}

void *thread2(void *arg) {
   (void)arg;
   printf("this is thread two\n");
   return NULL;
}

int main(void) {
   pthread_t t1, t2;
   pthread_create(&t1, NULL, thread1, NULL);
   pthread_create(&t2, NULL, thread2, NULL);

   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
   return 0;
```

}

Compile:

gcc q4.c -pthread -o q4

./q4

Q5 — Two threads with a shared buffer: write "Hello There", other reads and toggles case, then first reads modified text

/*

Q5.

Create a common shared memory area where in one thread writes a string termed "Hello There";

the second thread reads the string and displays it on the screen. Also the second string converts

all lower case to upper case and vice versa in the shared memory. Next the first thread will read

this from the shared memory and will output the same to the screen.

*/

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <pthread.h>

char shared_buf[256];
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t written = PTHREAD_COND_INITIALIZER;
pthread_cond_t converted = PTHREAD_COND_INITIALIZER;
int state = 0; // 0 = nothing, 1 = written, 2 = converted

void *writer(void *arg) {
   (void)arg;
   pthread_mutex_lock(&m);
   strncpy(shared_buf, "Hello There", sizeof(shared_buf));
```

```
    printf("Writer: wrote \"%s\"\n", shared_buf);
    state = 1;
    pthread_cond_signal(&written);

    // wait until conversion done
    while (state != 2) pthread_cond_wait(&converted, &m);
    printf("Writer: after conversion reads \"%s\"\n", shared_buf);
    pthread_mutex_unlock(&m);
    return NULL;
}

void *converter(void *arg) {
    (void)arg;
    pthread_mutex_lock(&m);
    while (state != 1) pthread_cond_wait(&written, &m);
    printf("Converter: read \"%s\"\n", shared_buf);

    for (size_t i = 0; i < strlen(shared_buf); ++i) {
        if (islower(shared_buf[i])) shared_buf[i] = toupper(shared_buf[i]);
        else if (isupper(shared_buf[i])) shared_buf[i] = tolower(shared_buf[i]);
    }
    printf("Converter: converted to \"%s\"\n", shared_buf);
    state = 2;
    pthread_cond_signal(&converted);
    pthread_mutex_unlock(&m);
    return NULL;
}

int main(void) {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, writer, NULL);
```

```
    pthread_create(&t2, NULL, converter, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

Compile:

```
gcc q5.c -pthread -o q5
./q5
```

Q6 — fork-based Fibonacci generator with argument validation

```
/*
Q6.
Write a c program using fork system call which generates Fibonacci sequence in the child process.
The number of sequence is to be provided in the command line, ex: if 5 is provided the first 5 numbers of
the Fibonacci series will be output by the child process. Perform necessary error checking such that a
nonnegative number is not to be taken.
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s n (n must be non-negative integer)\n", argv[0]);
        return 1;
    }
    char *end;
```

```
    long n = strtol(argv[1], &end, 10);
    if (*end != '\0' || n < 0) {
        fprintf(stderr, "Invalid n: must be non-negative integer\n");
        return 1;
    }
    pid_t pid = fork();
    if (pid < 0) { perror("fork"); return 1; }
    if (pid == 0) {
        // child prints first n fibonacci numbers (starting with 1,1,2,...)
        if (n == 0) { printf("(no numbers requested)\n"); _exit(0); }
        if (n >= 1) printf("1");
        if (n >= 2) printf(" 1");
        long a = 1, b = 1;
        for (long i = 3; i <= n; ++i) {
            long c = a + b;
            printf(" %ld", c);
            a = b; b = c;
        }
        printf("\n");
        _exit(0);
    } else {
        wait(NULL);
        return 0;
    }
}
```

Compile:

```
gcc q6.c -o q6
./q6 5
```

Q7 — One thread writes numbers into shared memory (System V), second thread reads and sums them

```
/*
Q7.
Next create a common shared memory area where in one thread writes a sequence of numbers,
the second thread reads the numbers, calculates their sum and displays the result to the screen.

This implementation uses a simple heap-allocated shared array protected by pthread mutex/cond
(i.e., shared among threads in same process). The phrase "shared memory area" in thread context
is satisfied by a common region (global array). For inter-process shared memory you'd use shmget/shmctl.
*/

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

#define MAX_N 100

int arr[MAX_N];
int n = 0;
int ready = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t produced = PTHREAD_COND_INITIALIZER;

void *producer(void *arg) {
   (void)arg;
   pthread_mutex_lock(&m);
   n = 10; // example: write 10 numbers (1..10)
   for (int i = 0; i < n; ++i) arr[i] = i + 1;
   printf("Producer wrote %d numbers\n", n);
```

```
    ready = 1;
    pthread_cond_signal(&produced);
    pthread_mutex_unlock(&m);
    return NULL;
}

void *consumer(void *arg) {
    (void)arg;
    pthread_mutex_lock(&m);
    while (!ready) pthread_cond_wait(&produced, &m);
    long sum = 0;
    for (int i = 0; i < n; ++i) sum += arr[i];
    printf("Consumer computed sum = %ld\n", sum);
    pthread_mutex_unlock(&m);
    return NULL;
}

int main(void) {
    pthread_t p, c;
    pthread_create(&p, NULL, producer, NULL);
    pthread_create(&c, NULL, consumer, NULL);
    pthread_join(p, NULL);
    pthread_join(c, NULL);
    return 0;
}
```

Compile & run:

```
gcc q7.c -pthread -o q7
./q7
```

Q8 — Row-sum with m threads (each computes a row sum)

```
/*
```

Q8.

Declare a matrix of size m x n , and populate it. (Fill it) Create m threads such that each thread calculates the ith row sum. Let the main thread display each row sum result as well as the total sum.

*/

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

typedef struct {
   int row;
   int cols;
   int *data; // pointer to row start
   long row_sum;
} row_arg;

void *row_worker(void *arg) {
   row_arg *r = (row_arg *)arg;
   long s = 0;
   for (int j = 0; j < r->cols; ++j) s += r->data[j];
   r->row_sum = s;
   return NULL;
}

int main(void) {
   int m = 4, n = 5; // example size; change as needed
   int *matrix = malloc(sizeof(int) * m * n);
   if (!matrix) { perror("malloc"); return 1; }

   // populate matrix with sample values (row-major): value = row*10 + col
```

```
    for (int i = 0; i < m; ++i)
        for (int j = 0; j < n; ++j)
            matrix[i*n + j] = i * 10 + j + 1;

    pthread_t *threads = malloc(sizeof(pthread_t) * m);
    row_arg *args = malloc(sizeof(row_arg) * m);

    for (int i = 0; i < m; ++i) {
        args[i].row = i;
        args[i].cols = n;
        args[i].data = &matrix[i*n];
        args[i].row_sum = 0;
        pthread_create(&threads[i], NULL, row_worker, &args[i]);
    }

    long total = 0;
    for (int i = 0; i < m; ++i) {
        pthread_join(threads[i], NULL);
        printf("Row %d sum = %ld\n", i, args[i].row_sum);
        total += args[i].row_sum;
    }
    printf("Total sum = %ld\n", total);

    free(matrix);
    free(threads);
    free(args);
    return 0;
}
```

Compile:

```
gcc q8.c -pthread -o q8
./q8
```

Q9 — Two threads operate on a sample text file: count words & change uppercase→lowercase (file in CWD)

```
/*
Q9.
create and save a sample text (a file containing any text) in your current working directory .
Next write a program which creates two threads such that one thread counts the total number of words
present in the file and the other thread changes all upper case characters to lower case.
*/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <ctype.h>
#include <string.h>

const char *FNAME = "sample_text.txt";
pthread_mutex_t file_mutex = PTHREAD_MUTEX_INITIALIZER;

void ensure_sample_file(void) {
  FILE *f = fopen(FNAME, "r");
  if (f) { fclose(f); return; }
  f = fopen(FNAME, "w");
  if (!f) { perror("create sample file"); exit(1); }
  fprintf(f, "Hello World!\nThis is Sample TEXT with UpperCase Words.\nAnother Line.\n");
  fclose(f);
}

void *count_words(void *arg) {
  (void)arg;
  pthread_mutex_lock(&file_mutex);
```

```
    FILE *f = fopen(FNAME, "r");
    if (!f) { perror("open file for counting"); pthread_mutex_unlock(&file_mutex); return NULL; }
    int words = 0;
    int c;
    int inword = 0;
    while ((c = fgetc(f)) != EOF) {
        if (isspace(c)) inword = 0;
        else if (!inword) { inword = 1; ++words; }
    }
    fclose(f);
    printf("Word count = %d\n", words);
    pthread_mutex_unlock(&file_mutex);
    return NULL;
}

void *upper_to_lower(void *arg) {
    (void)arg;
    pthread_mutex_lock(&file_mutex);
    FILE *f = fopen(FNAME, "r");
    if (!f) { perror("open file for read"); pthread_mutex_unlock(&file_mutex); return NULL; }
    // read entire file into memory
    fseek(f, 0, SEEK_END); long sz = ftell(f); fseek(f, 0, SEEK_SET);
    char *buf = malloc(sz + 1);
    if (!buf) { fclose(f); pthread_mutex_unlock(&file_mutex); return NULL; }
    fread(buf, 1, sz, f);
    buf[sz] = '\0';
    fclose(f);
    // convert uppercase to lowercase
    for (long i = 0; i < sz; ++i) if (isupper((unsigned char)buf[i])) buf[i] = tolower((unsigned
char)buf[i]);
    // write back
```

```
    FILE *out = fopen(FNAME, "w");
    if (!out) { perror("open file for write"); free(buf); pthread_mutex_unlock(&file_mutex); return NULL; }
    fwrite(buf, 1, sz, out);
    fclose(out);
    free(buf);
    printf("Converted uppercase -> lowercase in %s\n", FNAME);
    pthread_mutex_unlock(&file_mutex);
    return NULL;
}

int main(void) {
    ensure_sample_file();
    pthread_t t1, t2;
    pthread_create(&t1, NULL, count_words, NULL);
    pthread_create(&t2, NULL, upper_to_lower, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

Compile & run:

```
gcc q9.c -pthread -o q9
./q9
# check sample_text.txt contents afterwards
```

Q10 — Modify Q9 to add third thread that appends a message

```
/*
Q10.
Modify the above program (Q9) to create a third thread too; the third thread writes
"three threads have visited this file" at the very end of the file.
*/
```

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

const char *FNAME = "sample_text.txt";
pthread_mutex_t file_mutex = PTHREAD_MUTEX_INITIALIZER;

void ensure_sample_file(void) {
   FILE *f = fopen(FNAME, "r");
   if (f) { fclose(f); return; }
   f = fopen(FNAME, "w");
   if (!f) { perror("create sample file"); exit(1); }
   fprintf(f, "Hello World!\nThis is Sample TEXT with UpperCase Words.\nAnother Line.\n");
   fclose(f);
}

void *count_words(void *arg) {
   (void)arg;
   pthread_mutex_lock(&file_mutex);
   FILE *f = fopen(FNAME, "r");
   if (!f) { perror("open file for counting"); pthread_mutex_unlock(&file_mutex); return NULL; }
   int words = 0;
   int c, inword = 0;
   while ((c = fgetc(f)) != EOF) {
      if (isspace(c)) inword = 0;
      else if (!inword) { inword = 1; ++words; }
   }
   fclose(f);
   printf("Word count = %d\n", words);
```

```
    pthread_mutex_unlock(&file_mutex);
    return NULL;
}

void *upper_to_lower(void *arg) {
    (void)arg;
    pthread_mutex_lock(&file_mutex);
    FILE *f = fopen(FNAME, "r");
    if (!f) { perror("open file for read"); pthread_mutex_unlock(&file_mutex); return NULL; }
    fseek(f, 0, SEEK_END); long sz = ftell(f); fseek(f, 0, SEEK_SET);
    char *buf = malloc(sz + 1);
    if (!buf) { fclose(f); pthread_mutex_unlock(&file_mutex); return NULL; }
    fread(buf, 1, sz, f);
    buf[sz] = '\0';
    fclose(f);
    for (long i = 0; i < sz; ++i) if (isupper((unsigned char)buf[i])) buf[i] = tolower((unsigned
char)buf[i]);
    FILE *out = fopen(FNAME, "w");
    if (!out) { perror("open file for write"); free(buf); pthread_mutex_unlock(&file_mutex); return
NULL; }
    fwrite(buf, 1, sz, out);
    fclose(out);
    free(buf);
    printf("Converted uppercase -> lowercase in %s\n", FNAME);
    pthread_mutex_unlock(&file_mutex);
    return NULL;
}

void *append_message(void *arg) {
    (void)arg;
    pthread_mutex_lock(&file_mutex);
    FILE *f = fopen(FNAME, "a");
```

```
    if (!f) { perror("open file for append"); pthread_mutex_unlock(&file_mutex); return NULL; }
    fprintf(f, "\nthree threads have visited this file\n");
    fclose(f);
    printf("Appender: message appended\n");
    pthread_mutex_unlock(&file_mutex);
    return NULL;
}

int main(void) {
    ensure_sample_file();
    pthread_t t1, t2, t3;
    pthread_create(&t1, NULL, count_words, NULL);
    pthread_create(&t2, NULL, upper_to_lower, NULL);
    pthread_create(&t3, NULL, append_message, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);

    printf("Final file \"%s\" content:\n", FNAME);
    FILE *f = fopen(FNAME, "r");
    if (f) { int c; while ((c=fgetc(f))!=EOF) putchar(c); fclose(f); }
    return 0;
}
```

Compile & run:

```
gcc q10.c -pthread -o q10
./q10
# check sample_text.txt to verify appended message
```

Q11 — Peterson’s solution (two threads update a common global variable)

```
/*
Q11.
Implement peterson’s solution where two threads of a process update a common global variable
in their respective critical sections. The first thread increments the variable value by 100
and second thread decrements it by 75.
*/

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

volatile int turn;
volatile int flag[2];
volatile int shared = 0;

void peterson_enter(int self) {
   int other = 1 - self;
   flag[self] = 1;
   turn = other;
   // wait while other wants in and it's other's turn
   while (flag[other] && turn == other) { /* busy wait */ }
}

void peterson_leave(int self) {
   flag[self] = 0;
}

void *thread_inc(void *arg) {
   (void)arg;
   int id = 0;
```

```
    peterson_enter(id);
    // critical section
    shared += 100;
    printf("Thread 1: incremented by 100, shared = %d\n", shared);
    peterson_leave(id);
    return NULL;
}

void *thread_dec(void *arg) {
    (void)arg;
    int id = 1;
    peterson_enter(id);
    // critical section
    shared -= 75;
    printf("Thread 2: decremented by 75, shared = %d\n", shared);
    peterson_leave(id);
    return NULL;
}

int main(void) {
    pthread_t t1, t2;
    flag[0] = flag[1] = 0;
    turn = 0;
    shared = 0;
    pthread_create(&t1, NULL, thread_inc, NULL);
    pthread_create(&t2, NULL, thread_dec, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Final shared value = %d\n", shared);
    return 0;
}
```

Compile & run:

gcc Q11.c -pthread -o Q11

./Q11

Q12 — Reader–Writer (two readers, one writer) using mutex + semaphore

```
/*
Q12.
Show the implementation of reader's writer's problem using mutex or semaphore.
The reader just reads a global variable and will display it on the screen; the writer will
increment the variable by a value 35. Create two readers and one writer.
*/

#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

int shared = 0;
int readcount = 0;
pthread_mutex_t rc_mutex = PTHREAD_MUTEX_INITIALIZER; // protect readcount
sem_t rw_sem; // writers get exclusive access

void *reader(void *arg) {
   int id = *(int*)arg;
   // reader entry
   pthread_mutex_lock(&rc_mutex);
   readcount++;
   if (readcount == 1) sem_wait(&rw_sem); // first reader locks writer
   pthread_mutex_unlock(&rc_mutex);

   // critical: read
```

```
    printf("Reader %d reads shared = %d\n", id, shared);
    sleep(1);

    // reader exit
    pthread_mutex_lock(&rc_mutex);
    readcount--;
    if (readcount == 0) sem_post(&rw_sem); // last reader releases writer
    pthread_mutex_unlock(&rc_mutex);
    return NULL;
}

void *writer(void *arg) {
    (void)arg;
    sem_wait(&rw_sem); // exclusive access
    shared += 35;
    printf("Writer increments by 35: shared = %d\n", shared);
    sleep(1);
    sem_post(&rw_sem);
    return NULL;
}

int main(void) {
    pthread_t r1, r2, w;
    int id1 = 1, id2 = 2;
    sem_init(&rw_sem, 0, 1);
    shared = 100; // initial example value

    pthread_create(&r1, NULL, reader, &id1);
    pthread_create(&r2, NULL, reader, &id2);
    // ensure readers attempt to read while writer may also run
    sleep(0); // no-op but left for clarity
```

```
    pthread_create(&w, NULL, writer, NULL);

    pthread_join(r1, NULL);
    pthread_join(r2, NULL);
    pthread_join(w, NULL);
    sem_destroy(&rw_sem);
    return 0;
}
```

Compile & run:

```
gcc Q12.c -pthread -o Q12
./Q12
```

Q13 — Reader–Writer with file I/O (initial value in file = 25)

```
/*
Q13.
Modify the above Qn. Such that the readers read from a common file and the writer updates
the value in file. Let the initial value in file be 25.
*/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

const char *FNAME = "q13_value.txt";
pthread_mutex_t rc_mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t rw_sem;
int readcount = 0;

void ensure_file() {
```

```
    FILE *f = fopen(FNAME, "r");
    if (!f) {
        f = fopen(FNAME, "w");
        if (!f) { perror("fopen"); exit(1); }
        fprintf(f, "25\n");
        fclose(f);
    } else fclose(f);
}

int read_value_from_file() {
    FILE *f = fopen(FNAME, "r");
    if (!f) { perror("read fopen"); return -1; }
    int v;
    if (fscanf(f, "%d", &v) != 1) v = -1;
    fclose(f);
    return v;
}

void write_value_to_file(int v) {
    FILE *f = fopen(FNAME, "w");
    if (!f) { perror("write fopen"); return; }
    fprintf(f, "%d\n", v);
    fclose(f);
}

void *reader(void *arg) {
    int id = *(int*)arg;
    pthread_mutex_lock(&rc_mutex);
    readcount++;
    if (readcount == 1) sem_wait(&rw_sem);
    pthread_mutex_unlock(&rc_mutex);
```

```
    int v = read_value_from_file();
    printf("Reader %d read from file: %d\n", id, v);
    sleep(1);

    pthread_mutex_lock(&rc_mutex);
    readcount--;
    if (readcount == 0) sem_post(&rw_sem);
    pthread_mutex_unlock(&rc_mutex);
    return NULL;
}

void *writer(void *arg) {
    (void)arg;
    sem_wait(&rw_sem);
    int v = read_value_from_file();
    if (v >= 0) {
        v += 35;
        write_value_to_file(v);
        printf("Writer updated file value to %d\n", v);
    } else printf("Writer: error reading file\n");
    sleep(1);
    sem_post(&rw_sem);
    return NULL;
}

int main(void) {
    ensure_file();
    sem_init(&rw_sem, 0, 1);
    pthread_t r1, r2, w;
    int id1 = 1, id2 = 2;
```

```
    pthread_create(&r1, NULL, reader, &id1);
    pthread_create(&r2, NULL, reader, &id2);
    sleep(0);
    pthread_create(&w, NULL, writer, NULL);

    pthread_join(r1, NULL);
    pthread_join(r2, NULL);
    pthread_join(w, NULL);

    sem_destroy(&rw_sem);

    printf("Final file value: %d\n", read_value_from_file());
    return 0;
}
```

Compile & run:

```
gcc Q13.c -pthread -o Q13
./Q13
# check q13_value.txt
```

Q14 — Dining Philosophers (5 philosophers, semaphores)

```
/*
Q14.
Implement Dining Philosopher's problem either semaphores or mutex's.
This uses 5 philosophers, each alternates thinking/eating. Forks are semaphores.
*/

#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
```

```
#define N 5
sem_t forks[N];
pthread_t phil[N];

void thinking(int id) {
   printf("Philosopher %d thinking\n", id);
   usleep(100000 + (id * 10000));
}

void eating(int id) {
   printf("Philosopher %d eating\n", id);
   usleep(150000 + (id * 10000));
}

void *philosopher(void *arg) {
   int id = *(int*)arg;
   int left = id;
   int right = (id + 1) % N;

   for (int i = 0; i < 3; ++i) { // do 3 cycles
      thinking(id);
      // pick lower-numbered fork first to avoid deadlock, or use odd/even pickup
      if (id % 2 == 0) {
         sem_wait(&forks[left]);
         sem_wait(&forks[right]);
      } else {
         sem_wait(&forks[right]);
         sem_wait(&forks[left]);
      }
      eating(id);
```

```
    sem_post(&forks[left]);
    sem_post(&forks[right]);
  }
  printf("Philosopher %d done\n", id);
  return NULL;
}

int main(void) {
  int ids[N];
  for (int i = 0; i < N; ++i) sem_init(&forks[i], 0, 1);
  for (int i = 0; i < N; ++i) {
    ids[i] = i;
    pthread_create(&phil[i], NULL, philosopher, &ids[i]);
  }
  for (int i = 0; i < N; ++i) pthread_join(phil[i], NULL);
  for (int i = 0; i < N; ++i) sem_destroy(&forks[i]);
  return 0;
}
```

Compile & run:

```
gcc Q14.c -pthread -o Q14
./Q14
```

Q15 — Producer–Consumer (buffer size 5) using semaphores

```
/*
Q15.
Implement the producer-consumer problem of buffer 5, let the critical section be implemented
using either semaphores or mutex's.
This uses semaphores for empty, full and a mutex for buffer access.
*/

#include <stdio.h>
```

```
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define BUFSIZE 5
int buffer[BUFSIZE];
int in = 0, out = 0;

sem_t empty_slots;
sem_t filled_slots;
pthread_mutex_t buf_mutex = PTHREAD_MUTEX_INITIALIZER;

void *producer(void *arg) {
   int id = *(int*)arg;
   for (int i = 1; i <= 10; ++i) {
      sem_wait(&empty_slots);
      pthread_mutex_lock(&buf_mutex);
      buffer[in] = i;
      in = (in + 1) % BUFSIZE;
      printf("Producer %d produced %d\n", id, i);
      pthread_mutex_unlock(&buf_mutex);
      sem_post(&filled_slots);
      usleep(100000);
   }
   return NULL;
}

void *consumer(void *arg) {
   int id = *(int*)arg;
   for (int i = 1; i <= 10; ++i) {
      sem_wait(&filled_slots);
```

```
        pthread_mutex_lock(&buf_mutex);
        int val = buffer[out];
        out = (out + 1) % BUFSIZE;
        printf("Consumer %d consumed %d\n", id, val);
        pthread_mutex_unlock(&buf_mutex);
        sem_post(&empty_slots);
        usleep(150000);
    }
    return NULL;
}

int main(void) {
    sem_init(&empty_slots, 0, BUFSIZE);
    sem_init(&filled_slots, 0, 0);

    pthread_t p, c;
    int pid = 1, cid = 1;
    pthread_create(&p, NULL, producer, &pid);
    pthread_create(&c, NULL, consumer, &cid);

    pthread_join(p, NULL);
    pthread_join(c, NULL);

    sem_destroy(&empty_slots);
    sem_destroy(&filled_slots);
    return 0;
}
```

Compile & run:

```
gcc Q15.c -pthread -o Q15
./Q15
```

Q16 — CPU Scheduling algorithms (FCFS, SJF non-preemptive, SRTF preemptive, Priority non-preemptive, Round Robin)

```
/*
Q16.
Implement the following scheduling algorithms:
a. FCFS
b. SJF (non-preemptive)
c. SJF with Preemption (SRTF)
d. Priority (non-preemptive)
e. Round Robin

Read number of processes, arrival times, burst times, priorities, time slice (for RR).
Output average waiting time and average turnaround time for each algorithm.

Note: Inputs are read from stdin interactively. For simplicity this implementation expects small N.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

typedef struct {
  int pid;
  int at;
  int bt;
  int bt_copy;
  int pr;
  int ct;
  int tat;
  int wt;
```

```
} proc;

void print_avg(proc *p, int n, const char *name) {
   double avgwt = 0, avgtat = 0;
   for (int i = 0; i < n; ++i) {
      avgwt += p[i].wt;
      avgtat += p[i].tat;
   }
   avgwt /= n; avgtat /= n;
   printf("%s -> Avg Waiting Time = %.2f, Avg Turnaround Time = %.2f\n", name, avgwt, avgtat);
}

int cmp_at(const void *a, const void *b) {
   proc *p1 = (proc*)a; proc *p2 = (proc*)b;
   if (p1->at != p2->at) return p1->at - p2->at;
   return p1->pid - p2->pid;
}

/* FCFS */
void fcfs(proc *arr, int n) {
   proc *p = malloc(sizeof(proc)*n);
   memcpy(p, arr, sizeof(proc)*n);
   qsort(p, n, sizeof(proc), cmp_at);
   int time = 0;
   for (int i = 0; i < n; ++i) {
      if (time < p[i].at) time = p[i].at;
      time += p[i].bt;
      p[i].ct = time;
      p[i].tat = p[i].ct - p[i].at;
      p[i].wt = p[i].tat - p[i].bt;
   }
```

```
    print_avg(p, n, "FCFS");
    free(p);
}

/* SJF non-preemptive */
void sjf_np(proc *arr, int n) {
    proc *p = malloc(sizeof(proc)*n);
    memcpy(p, arr, sizeof(proc)*n);
    int time = 0, completed = 0;
    int finished[n]; memset(finished, 0, sizeof(finished));
    while (completed < n) {
        int idx = -1; int minbt = INT_MAX;
        for (int i = 0; i < n; ++i) {
            if (!finished[i] && p[i].at <= time) {
                if (p[i].bt < minbt) { minbt = p[i].bt; idx = i; }
            }
        }
        if (idx == -1) { time++; continue; }
        time += p[idx].bt;
        p[idx].ct = time;
        p[idx].tat = p[idx].ct - p[idx].at;
        p[idx].wt = p[idx].tat - p[idx].bt;
        finished[idx] = 1; completed++;
    }
    print_avg(p, n, "SJF (Non-preemptive)");
    free(p);
}

/* SRTF (preemptive SJF) */
void srtf(proc *arr, int n) {
    proc *p = malloc(sizeof(proc)*n);
```

```
    memcpy(p, arr, sizeof(proc)*n);
    for (int i=0;i<n;i++) p[i].bt_copy = p[i].bt;
    int completed = 0, time = 0;
    int prev = -1;
    while (completed < n) {
        int idx = -1, minrem = INT_MAX;
        for (int i = 0; i < n; ++i) {
            if (p[i].at <= time && p[i].bt_copy > 0) {
                if (p[i].bt_copy < minrem) { minrem = p[i].bt_copy; idx = i; }
            }
        }
        if (idx == -1) { time++; continue; }
        p[idx].bt_copy--;
        time++;
        if (p[idx].bt_copy == 0) {
            completed++;
            p[idx].ct = time;
            p[idx].tat = p[idx].ct - p[idx].at;
            p[idx].wt = p[idx].tat - p[idx].bt;
        }
    }
    print_avg(p, n, "SJF (Preemptive - SRTF)");
    free(p);
}

/* Priority (non-preemptive) - lower number = higher priority */
void priority_np(proc *arr, int n) {
    proc *p = malloc(sizeof(proc)*n);
    memcpy(p, arr, sizeof(proc)*n);
    int time = 0, completed = 0;
    int finished[n]; memset(finished, 0, sizeof(finished));
```

```
    while (completed < n) {
        int idx = -1, bestpr = INT_MAX;
        for (int i = 0; i < n; ++i) {
            if (!finished[i] && p[i].at <= time) {
                if (p[i].pr < bestpr) { bestpr = p[i].pr; idx = i; }
            }
        }
        if (idx == -1) { time++; continue; }
        time += p[idx].bt;
        p[idx].ct = time;
        p[idx].tat = p[idx].ct - p[idx].at;
        p[idx].wt = p[idx].tat - p[idx].bt;
        finished[idx] = 1; completed++;
    }
    print_avg(p, n, "Priority (Non-preemptive)");
    free(p);
}

/* Round Robin */
void rr(proc *arr, int n, int tq) {
    proc *p = malloc(sizeof(proc)*n);
    memcpy(p, arr, sizeof(proc)*n);
    for (int i=0;i<n;i++) p[i].bt_copy = p[i].bt;
    int time = 0;
    int done = 0;
    int queue[1000]; int qh=0, qt=0;
    int inqueue[n]; memset(inqueue,0,sizeof(inqueue));
    // enqueue processes as they arrive
    while (done < n) {
        // enqueue newly arrived
        for (int i=0;i<n;i++) {
```

```
        if (p[i].at <= time && !inqueue[i] && p[i].bt_copy>0) {
            queue[qt++] = i; inqueue[i]=1;
        }
    }
    if (qh==qt) { time++; continue; }
    int idx = queue[qh++]; // pop
    int exec = (p[idx].bt_copy > tq) ? tq : p[idx].bt_copy;
    p[idx].bt_copy -= exec;
    time += exec;
    // enqueue newly arrived during execution
    for (int i=0;i<n;i++) {
        if (p[i].at <= time && !inqueue[i] && p[i].bt_copy>0) {
            queue[qt++] = i; inqueue[i]=1;
        }
    }
    if (p[idx].bt_copy > 0) {
        queue[qt++] = idx; // requeue
    } else {
        done++;
        p[idx].ct = time;
        p[idx].tat = p[idx].ct - p[idx].at;
        p[idx].wt = p[idx].tat - p[idx].bt;
    }
  }
  print_avg(p, n, "Round Robin");
  free(p);
}

int main(void) {
  int n;
  printf("Enter number of processes: ");
```

```
    if (scanf("%d", &n) != 1 || n <= 0) return 0;
    proc *p = malloc(sizeof(proc) * n);
    for (int i = 0; i < n; ++i) {
        p[i].pid = i;
        printf("Process %d - Arrival time, Burst time, Priority: ", i);
        scanf("%d %d %d", &p[i].at, &p[i].bt, &p[i].pr);
        p[i].bt_copy = p[i].bt;
    }
    int tq;
    printf("Enter time quantum for Round Robin: ");
    scanf("%d", &tq);

    fcfs(p, n);
    sjf_np(p, n);
    srtf(p, n);
    priority_np(p, n);
    rr(p, n, tq);

    free(p);
    return 0;
}
```

Compile & run:

```
gcc Q16.c -o Q16
./Q16
# Follow prompts to input process data.
```

Example input style:

```
Enter number of processes: 3
Process 0 - Arrival time, Burst time, Priority: 0 5 2
Process 1 - Arrival time, Burst time, Priority: 1 3 1
Process 2 - Arrival time, Burst time, Priority: 2 8 3
Enter time quantum for Round Robin: 2
```

Q17 — Banker’s Algorithm (safety algorithm, show steps)

```
/*
Q17.
Implement the Banker’s algorithm. Read the number of processes, the Max Resource type vector,
the Need matrix and the current allocation matrix. Show all the intermediate steps of resource allocation
to the processes and determine whether the system is in safe state or not.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
   int n, m;
   printf("Enter number of processes: ");
   scanf("%d", &n);
   printf("Enter number of resource types: ");
   scanf("%d", &m);

   int alloc[n][m];
   int max[n][m];
   int need[n][m];
   int avail[m];

   printf("Enter Allocation matrix (n x m):\n");
   for (int i=0;i<n;i++)
      for (int j=0;j<m;j++) scanf("%d", &alloc[i][j]);

   printf("Enter Max matrix (n x m):\n");
```

```
for (int i=0;i<n;i++)
   for (int j=0;j<m;j++) scanf("%d", &max[i][j]);

printf("Enter Available vector (m):\n");
for (int j=0;j<m;j++) scanf("%d", &avail[j]);

// compute Need = Max - Alloc
for (int i=0;i<n;i++)
   for (int j=0;j<m;j++) need[i][j] = max[i][j] - alloc[i][j];

printf("\nNeed matrix:\n");
for (int i=0;i<n;i++) {
   for (int j=0;j<m;j++) printf("%d ", need[i][j]);
   printf("\n");
}

// Safety algorithm
int finish[n]; memset(finish, 0, sizeof(finish));
int work[m];
for (int j=0;j<m;j++) work[j] = avail[j];
int safe_seq[n];
int idx = 0;

printf("\nRunning Safety Algorithm steps:\n");
int found;
do {
   found = 0;
   for (int i = 0; i < n; ++i) {
      if (!finish[i]) {
         int can = 1;
         for (int j = 0; j < m; ++j) {
```

```
            if (need[i][j] > work[j]) { can = 0; break; }
        }
        if (can) {
            printf("Process P%d can be satisfied -> adding to safe sequence. Work before: ", i);
            for (int j=0;j<m;j++) printf("%d ", work[j]);
            printf("\n");
            // simulate allocation finishing
            for (int j = 0; j < m; ++j) work[j] += alloc[i][j];
            finish[i] = 1;
            safe_seq[idx++] = i;
            printf("After P%d finishes, Work becomes: ", i);
            for (int j=0;j<m;j++) printf("%d ", work[j]);
            printf("\n\n");
            found = 1;
        }
      }
   }
} while (found);

int all_finished = 1;
for (int i=0;i<n;i++) if (!finish[i]) { all_finished = 0; break; }

if (all_finished) {
   printf("System is in SAFE state.\nSafe sequence: ");
   for (int i=0;i<idx;i++) printf("P%d ", safe_seq[i]);
   printf("\n");
} else {
   printf("System is NOT in safe state. Processes not satisfied:\n");
   for (int i=0;i<n;i++) if (!finish[i]) printf("P%d ", i);
   printf("\n");
}
```

```
  // OPTIONAL: show sample resource request handling
  char choose;
  printf("Do you want to test a resource request? (y/n): ");
  scanf(" %c", &choose);
  if (choose == 'y' || choose == 'Y') {
    int proc; printf("Enter process id making request (0..%d): ", n-1);
    scanf("%d", &proc);
    int req[m];
    printf("Enter request vector (%d elements):\n", m);
    for (int j=0;j<m;j++) scanf("%d", &req[j]);
    // check request <= need
    int ok = 1;
    for (int j=0;j<m;j++) if (req[j] > need[proc][j]) ok = 0;
    if (!ok) {
      printf("Error: request exceeds process's declared need.\n");
    } else {
      // check request <= available (avail==work initially)
      int ok2 = 1;
      for (int j=0;j<m;j++) if (req[j] > avail[j]) ok2 = 0;
      if (!ok2) {
        printf("Request cannot be granted immediately (not enough available resources).\n");
      } else {
        // pretend to allocate and run safety
        int temp_avail[m]; int temp_alloc[n][m]; int temp_need[n][m];
        for (int j=0;j<m;j++) temp_avail[j] = avail[j] - req[j];
        for (int i=0;i<n;i++) for (int j=0;j<m;j++) { temp_alloc[i][j] = alloc[i][j]; temp_need[i][j] =
need[i][j]; }
        for (int j=0;j<m;j++) { temp_alloc[proc][j] += req[j]; temp_need[proc][j] -= req[j]; }
        // run safety on temp matrices
        int finish2[n]; memset(finish2,0,sizeof(finish2));
```

```
            int work2[m]; for (int j=0;j<m;j++) work2[j] = temp_avail[j];
            int changed;
            do {
              changed = 0;
              for (int i=0;i<n;i++) if (!finish2[i]) {
                int can = 1;
                for (int j=0;j<m;j++) if (temp_need[i][j] > work2[j]) { can = 0; break; }
                if (can) {
                  for (int j=0;j<m;j++) work2[j] += temp_alloc[i][j];
                  finish2[i] = 1; changed = 1;
                }
              }
            } while (changed);
            int safe = 1;
            for (int i=0;i<n;i++) if (!finish2[i]) safe = 0;
            if (safe) printf("Request can be GRANTED safely.\n");
            else printf("Request SHOULD NOT be granted (unsafe).\n");
          }
        }
      }

    return 0;
}
```

Compile & run:

gcc Q17.c -o Q17

./Q17

follow prompts to input matrices/vectors

Save each snippet into separate files as named in the headers (e.g., Q1_parent.c, child_prog.c, Q2.c, ...). When a program uses pthreads, compile with -pthread. For POSIX shared memory, no extra link flags are usually needed.

Q1 — Fork a child and make child execute a separate program (exec)

Two files: child_prog.c (the program the child will run) and Q1_parent.c (forks and execs child).

child_prog.c — example: counts whitespaces in a file (or string)

```
/*
child_prog.c
Child program: example program that reads a filename (or reads stdin) and prints
the number of whitespace characters (spaces, tabs, newlines) and total characters.

Compile:
gcc child_prog.c -o childprog

Usage:
./childprog filename
*/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Usage: %s filename\n", argv[0]);
    return 1;
  }
  const char *fname = argv[1];
  FILE *f = fopen(fname, "r");
  if (!f) { perror("fopen"); return 1; }
  int c; long total = 0, whites = 0;
  while ((c = fgetc(f)) != EOF) {
    total++;
    if (c == ' ' || c == '\t' || c == '\n') whites++;
  }
  fclose(f);
```

```
   printf("File: %s\nTotal characters: %ld\nWhitespace chars (space/tab/newline): %ld\n",
fname, total, whites);
   return 0;
}
```

Q1_parent.c — fork & execl to run childprog

```
/*
Q1_parent.c
1. Fork a child process and make the child execute the program childprog (using execl/execlp).
  The child program (childprog) should be compiled separately (see child_prog.c).
Compile:
gcc Q1_parent.c -o Q1_parent
Run:
./Q1_parent filename
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
   if (argc != 2) {
      fprintf(stderr, "Usage: %s filename\n", argv[0]);
      return 1;
   }
   pid_t pid = fork();
   if (pid < 0) { perror("fork"); return 1; }
   if (pid == 0) {
      // child: execute childprog (must be in same dir and executable)
      execl("./childprog", "childprog", argv[1], (char *)NULL);
      perror("execl");
      _exit(1);
```

```
    } else {
        int status;
        waitpid(pid, &status, 0);
        printf("Parent: child finished with status %d\n", WEXITSTATUS(status));
    }
    return 0;
}
```

Q2 — Create threads performing different computations (pthread)

```
/*
Q2.c
2. Creating one or more threads and making the different threads perform different computations.
   Here: Thread A computes sum of an integer array, Thread B computes product of another array,
   Thread C reverses and prints a string.
Compile:
gcc Q2.c -pthread -o Q2
Run:
./Q2
*/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

void *sum_thread(void *arg) {
    int *arr = arg;
    int n = arr[0];
    long sum = 0;
    for (int i = 1; i <= n; ++i) sum += arr[i];
    printf("Sum thread: sum = %ld\n", sum);
```

```
    return NULL;
}

void *prod_thread(void *arg) {
    int *arr = arg;
    int n = arr[0];
    long prod = 1;
    for (int i = 1; i <= n; ++i) prod *= arr[i];
    printf("Product thread: prod = %ld\n", prod);
    return NULL;
}

void *rev_thread(void *arg) {
    char *s = arg;
    int L = strlen(s);
    printf("Reverse thread: ");
    for (int i = L-1; i >= 0; --i) putchar(s[i]);
    putchar('\n');
    return NULL;
}

int main(void) {
    pthread_t t1, t2, t3;
    // sample data
    int arr1[] = {5, 1,2,3,4,5}; // arr1[0]=n, then elements
    int arr2[] = {3, 2,3,4};
    char str[] = "HelloThreads";

    pthread_create(&t1, NULL, sum_thread, arr1);
    pthread_create(&t2, NULL, prod_thread, arr2);
    pthread_create(&t3, NULL, rev_thread, str);
```

```
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    return 0;
}
```

Q3 — POSIX shared memory: one process writes "Hello There"; second reads and converts to UPPER

Two programs: Q3_writer.c and Q3_reader.c. Use shm_open, mmap, shm_unlink.

Q3_writer.c

```
/*
Q3_writer.c
3. Create a POSIX shared memory area where one process writes "Hello There".
Compile:
gcc Q3_writer.c -o Q3_writer -lrt   (on some systems -lrt not needed; it's fine without too)
Run first:
./Q3_writer
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>

const char *SHM_NAME = "/exam_shm_example";
int main(void) {
    int fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
    if (fd == -1) { perror("shm_open"); return 1; }
```

```
    size_t size = 256;
    if (ftruncate(fd, size) == -1) { perror("ftruncate"); return 1; }

    void *ptr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (ptr == MAP_FAILED) { perror("mmap"); return 1; }

    const char *msg = "Hello There";
    strncpy((char*)ptr, msg, size-1);
    ((char*)ptr)[size-1] = '\0';
    printf("Writer: wrote \"%s\" into shared memory %s\n", msg, SHM_NAME);

    // keep writer simple: detach and leave shm for reader to open (do not shm_unlink here)
    munmap(ptr, size);
    close(fd);
    return 0;
}
```

Q3_reader.c

```
/*
Q3_reader.c
3. Reader process: opens the same POSIX shm, reads string and prints uppercase version.
Run after writer:
./Q3_reader
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <ctype.h>
```

```
const char *SHM_NAME = "/exam_shm_example";
int main(void) {
  int fd = shm_open(SHM_NAME, O_RDONLY, 0666);
  if (fd == -1) { perror("shm_open"); return 1; }
  size_t size = 256;
  void *ptr = mmap(NULL, size, PROT_READ, MAP_SHARED, fd, 0);
  if (ptr == MAP_FAILED) { perror("mmap"); return 1; }
  char buf[256];
  strncpy(buf, (char*)ptr, sizeof(buf)-1);
  buf[sizeof(buf)-1] = '\0';
  printf("Reader: original: \"%s\"\n", buf);
  printf("Reader: upper-case: \"");
  for (size_t i=0;i<strlen(buf);++i) putchar(toupper((unsigned char)buf[i]));
  printf("\"\n");
  munmap(ptr, size);
  close(fd);
  // Optionally unlink shared memory
  // shm_unlink(SHM_NAME);
  return 0;
}
```

Q4 — Reader-Writer with pthread mutex: two readers + one writer operate on a file

```
/*
Q4.c
4. Reader-writer using pthread mutex. 2 reader threads and 1 writer thread.
  - Reader1: reads file and prints characters in lowercase
  - Reader2: reads file and prints characters in UPPERCASE
  - Writer: appends "hello" at end of file
Instructions:
Create a sample file 'sample_rw.txt' (the code will create it if missing).
Compile:
```

```
gcc Q4.c -pthread -o Q4
Run:
./Q4
*/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <ctype.h>
#include <string.h>

const char *FNAME = "sample_rw.txt";
pthread_mutex_t file_mutex = PTHREAD_MUTEX_INITIALIZER;

void ensure_file() {
  FILE *f = fopen(FNAME, "r");
  if (!f) {
    f = fopen(FNAME, "w");
    fprintf(f, "This Is A Sample File.\nLine Two: MORE Text!\n");
    fclose(f);
  } else fclose(f);
}

void *reader_lower(void *arg) {
  (void)arg;
  pthread_mutex_lock(&file_mutex);
  FILE *f = fopen(FNAME, "r");
  if (!f) { perror("fopen"); pthread_mutex_unlock(&file_mutex); return NULL; }
  int c;
  printf("Reader-Lower: ");
  while ((c = fgetc(f)) != EOF) putchar(tolower((unsigned char)c));
  printf("\n");
```

```
    fclose(f);
    pthread_mutex_unlock(&file_mutex);
    return NULL;
}

void *reader_upper(void *arg) {
    (void)arg;
    pthread_mutex_lock(&file_mutex);
    FILE *f = fopen(FNAME, "r");
    if (!f) { perror("fopen"); pthread_mutex_unlock(&file_mutex); return NULL; }
    int c;
    printf("Reader-Upper: ");
    while ((c = fgetc(f)) != EOF) putchar(toupper((unsigned char)c));
    printf("\n");
    fclose(f);
    pthread_mutex_unlock(&file_mutex);
    return NULL;
}

void *writer_append(void *arg) {
    (void)arg;
    pthread_mutex_lock(&file_mutex);
    FILE *f = fopen(FNAME, "a");
    if (!f) { perror("fopen"); pthread_mutex_unlock(&file_mutex); return NULL; }
    fprintf(f, "hello");
    printf("Writer: appended \"hello\" to file\n");
    fclose(f);
    pthread_mutex_unlock(&file_mutex);
    return NULL;
}
```

```
int main(void) {
   ensure_file();
   pthread_t r1, r2, w;
   pthread_create(&r1, NULL, reader_lower, NULL);
   pthread_create(&r2, NULL, reader_upper, NULL);
   // small sleep to increase chance readers run before writer, but mutex makes access safe
   usleep(50000);
   pthread_create(&w, NULL, writer_append, NULL);
   pthread_join(r1, NULL);
   pthread_join(r2, NULL);
   pthread_join(w, NULL);
   printf("Final file contents:\n");
   FILE *f = fopen(FNAME, "r");
   if (f) { int c; while ((c=fgetc(f))!=EOF) putchar(c); fclose(f); }
   return 0;
}
```

Q5 — Producer–Consumer using circular buffer, critical section with pthread mutex + cond vars

```
/*
Q5.c
5. Producer-consumer using circular buffer (size 5). Mutex protects critical section; use condition variables.
Compile:
gcc Q5.c -pthread -o Q5
Run:
./Q5
*/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
```

```
#define BUFSZ 5
int buffer[BUFSZ];
int in = 0, out = 0, count = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;

void *producer(void *arg) {
  int id = *(int*)arg;
  for (int i=1;i<=15;i++) {
    pthread_mutex_lock(&mutex);
    while (count == BUFSZ) pthread_cond_wait(&not_full, &mutex);
    buffer[in] = i;
    in = (in + 1) % BUFSZ;
    count++;
    printf("Producer %d produced %d (count=%d)\n", id, i, count);
    pthread_cond_signal(&not_empty);
    pthread_mutex_unlock(&mutex);
    usleep(80000);
  }
  return NULL;
}

void *consumer(void *arg) {
  int id = *(int*)arg;
  for (int i=1;i<=15;i++) {
    pthread_mutex_lock(&mutex);
    while (count == 0) pthread_cond_wait(&not_empty, &mutex);
    int val = buffer[out];
    out = (out + 1) % BUFSZ;
```

```
        count--;
        printf("Consumer %d consumed %d (count=%d)\n", id, val, count);
        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&mutex);
        usleep(120000);
    }
    return NULL;
}

int main(void) {
    pthread_t p, c;
    int pid = 1, cid = 1;
    pthread_create(&p, NULL, producer, &pid);
    pthread_create(&c, NULL, consumer, &cid);
    pthread_join(p, NULL);
    pthread_join(c, NULL);
    return 0;
}
```

Q6 — Producer-consumer with shared counter (tracks total items in buffer)

```
/*
Q6.c
6. Producer-consumer as Q5 but also maintain a common counter variable that tracks total items
   currently in the buffer. Display it every time producer/consumer runs.
Compile:
gcc Q6.c -pthread -o Q6
Run:
./Q6
*/
#include <stdio.h>
```

```
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define BUFSZ 5
int buffer[BUFSZ];
int in = 0, out = 0, count = 0;
int shared_counter = 0; // tracks total items in buffer (duplicate of count but as global shared counter)
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;

void *producer(void *arg) {
  int id = *(int*)arg;
  for (int i=1;i<=15;i++) {
    pthread_mutex_lock(&mutex);
    while (count == BUFSZ) pthread_cond_wait(&not_full, &mutex);
    buffer[in] = i; in = (in+1)%BUFSZ; count++; shared_counter = count;
    printf("Producer %d produced %d (shared_counter=%d)\n", id, i, shared_counter);
    pthread_cond_signal(&not_empty);
    pthread_mutex_unlock(&mutex);
    usleep(80000);
  }
  return NULL;
}

void *consumer(void *arg) {
  int id = *(int*)arg;
  for (int i=1;i<=15;i++) {
    pthread_mutex_lock(&mutex);
```

```
        while (count == 0) pthread_cond_wait(&not_empty, &mutex);
        int val = buffer[out]; out = (out+1)%BUFSZ; count--; shared_counter = count;
        printf("Consumer %d consumed %d (shared_counter=%d)\n", id, val, shared_counter);
        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&mutex);
        usleep(120000);
    }
    return NULL;
}

int main(void) {
    pthread_t p, c;
    int pid = 1, cid = 1;
    pthread_create(&p, NULL, producer, &pid);
    pthread_create(&c, NULL, consumer, &cid);
    pthread_join(p, NULL);
    pthread_join(c, NULL);
    return 0;
}
```

Q7 — FCFS scheduling (all arrivals at 0)

```
/*
Q7.c
7. FCFS scheduling. Read number of processes and their burst times. Assume arrival = 0 for all.
   Output average waiting time and average turnaround time.
Compile:
gcc Q7.c -o Q7
Run:
./Q7
*/
#include <stdio.h>
```

```c
#include <stdlib.h>

int main(void) {
   int n;
   printf("Enter number of processes: ");
   if (scanf("%d", &n)!=1 || n<=0) return 0;
   int *bt = malloc(sizeof(int)*n);
   for (int i=0;i<n;i++) {
      printf("Burst time for P%d: ", i);
      scanf("%d", &bt[i]);
   }
   int *ct = malloc(sizeof(int)*n);
   int time = 0;
   for (int i=0;i<n;i++) {
      time += bt[i];
      ct[i] = time;
   }
   double total_wt=0, total_tat=0;
   for (int i=0;i<n;i++) {
      int tat = ct[i]; // arrival 0 => tat = completion time
      int wt = tat - bt[i];
      total_wt += wt;
      total_tat += tat;
      printf("P%d: BT=%d WT=%d TAT=%d\n", i, bt[i], wt, tat);
   }
   printf("Average Waiting Time = %.2f\n", total_wt/n);
   printf("Average Turnaround Time = %.2f\n", total_tat/n);
   free(bt); free(ct);
   return 0;
}
```

Q8 — Round Robin (arrival = 0)

```
/*
Q8.c
8. Round Robin scheduling. Read number of processes, time slice, and burst times (arrival=0).
  Output average waiting time and average turnaround time.
Compile:
gcc Q8.c -o Q8
Run:
./Q8
*/
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int n, tq;
  printf("Enter number of processes: ");
  if (scanf("%d", &n)!=1 || n<=0) return 0;
  printf("Enter time quantum: ");
  scanf("%d", &tq);
  int *bt = malloc(sizeof(int)*n);
  int *rem = malloc(sizeof(int)*n);
  for (int i=0;i<n;i++) {
    printf("Burst time for P%d: ", i);
    scanf("%d", &bt[i]);
    rem[i] = bt[i];
  }

  int time = 0, done = 0;
  int *ct = malloc(sizeof(int)*n);
  for (int i=0;i<n;i++) ct[i]=0;
```

```
  while (done < n) {
    int progressed = 0;
    for (int i=0;i<n;i++) {
      if (rem[i] > 0) {
        progressed = 1;
        int exec = rem[i] > tq ? tq : rem[i];
        rem[i] -= exec;
        time += exec;
        if (rem[i] == 0) { ct[i] = time; done++; }
      }
    }
    if (!progressed) break;
  }
  double total_wt=0,total_tat=0;
  for (int i=0;i<n;i++) {
    int tat = ct[i];
    int wt = tat - bt[i];
    printf("P%d: BT=%d WT=%d TAT=%d\n", i, bt[i], wt, tat);
    total_wt += wt; total_tat += tat;
  }
  printf("Avg WT = %.2f, Avg TAT = %.2f\n", total_wt/n, total_tat/n);
  free(bt); free(rem); free(ct);
  return 0;
}
```

Q9 — SJF (non-preemptive), arrival = 0

```
/*
Q9.c
9. SJF scheduling (non-preemptive). Read n and burst times (arrival=0). Output AVG WT and AVG TAT.
Compile:
```

```
gcc Q9.c -o Q9
Run:
./Q9
*/
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int n;
   printf("Enter number of processes: ");
   if (scanf("%d", &n)!=1 || n<=0) return 0;
   int *bt = malloc(sizeof(int)*n);
   for (int i=0;i<n;i++) { printf("Burst time for P%d: ", i); scanf("%d", &bt[i]); }
   // sort by burst time (keep pid mapping)
   int *idx = malloc(sizeof(int)*n);
   for (int i=0;i<n;i++) idx[i]=i;
   for (int i=0;i<n-1;i++) for (int j=i+1;j<n;j++)
      if (bt[idx[i]] > bt[idx[j]]) { int t=idx[i]; idx[i]=idx[j]; idx[j]=t; }

   int time = 0;
   int *ct = malloc(sizeof(int)*n);
   for (int k=0;k<n;k++) {
      int i = idx[k];
      time += bt[i];
      ct[i] = time;
   }
   double totwt=0, tottat=0;
   for (int i=0;i<n;i++) {
      int tat = ct[i];
      int wt = tat - bt[i];
      printf("P%d: BT=%d WT=%d TAT=%d\n", i, bt[i], wt, tat);
```

```
    totwt += wt; tottat += tat;
  }
  printf("Avg WT=%.2f Avg TAT=%.2f\n", totwt/n, tottat/n);
  free(bt); free(idx); free(ct);
  return 0;
}
```

Q10 — Priority (non-preemptive), arrival = 0

```
/*
Q10.c
10. Priority scheduling (non-preemptive). Read number of processes, priority and burst times (arrival=0).
  Lower priority number => higher priority.
Compile:
gcc Q10.c -o Q10
Run:
./Q10
*/
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int n;
  printf("Enter number of processes: ");
  if (scanf("%d",&n)!=1 || n<=0) return 0;
  int *bt = malloc(sizeof(int)*n);
  int *pr = malloc(sizeof(int)*n);
  for (int i=0;i<n;i++) {
    printf("Enter priority (lower value = higher priority) and burst for P%d: ", i);
    scanf("%d %d", &pr[i], &bt[i]);
  }
```

```
    // sort by priority
    int *idx = malloc(sizeof(int)*n);
    for (int i=0;i<n;i++) idx[i]=i;
    for (int i=0;i<n-1;i++) for (int j=i+1;j<n;j++)
        if (pr[idx[i]] > pr[idx[j]]) { int t = idx[i]; idx[i]=idx[j]; idx[j]=t; }

    int time = 0;
    int *ct = malloc(sizeof(int)*n);
    for (int k=0;k<n;k++) {
        int i = idx[k];
        time += bt[i];
        ct[i] = time;
    }
    double tw=0, tt=0;
    for (int i=0;i<n;i++) {
        int tat = ct[i], wt = tat - bt[i];
        printf("P%d: Priority=%d BT=%d WT=%d TAT=%d\n", i, pr[i], bt[i], wt, tat);
        tw += wt; tt += tat;
    }
    printf("Avg WT=%.2f Avg TAT=%.2f\n", tw/n, tt/n);
    free(bt); free(pr); free(idx); free(ct);
    return 0;
}
```

✅ **PROGRAM 1 — Producer–Consumer (Semaphores + Circular Buffer + User Input)**

File: prod_cons_sem.c

```
/*
---------------------------------------------------------------------------
Producer–Consumer Problem using Semaphores + Circular Buffer
---------------------------------------------------------------------------
User Input:
   - Buffer size
   - Number of items to produce/consume

Shows:
   - Step-by-step production/consumption
   - Semaphore operations (full, empty)
   - Mutex-protected critical section
---------------------------------------------------------------------------
*/

#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

int *buffer, in = 0, out = 0;
int BUFFER_SIZE, TOTAL_ITEMS;

sem_t empty, full;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *producer(void *arg) {
   for (int item = 1; item <= TOTAL_ITEMS; item++) {
```

```
        sem_wait(&empty);   // wait for empty slot
        pthread_mutex_lock(&mutex);

        buffer[in] = item;
        printf("[Producer] Produced item %d at index %d\n", item, in);
        in = (in + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);
        sem_post(&full);    // signal filled slot

        usleep(100000);
    }
    return NULL;
}

void *consumer(void *arg) {
    for (int i = 1; i <= TOTAL_ITEMS; i++) {

        sem_wait(&full);    // wait for item
        pthread_mutex_lock(&mutex);

        int item = buffer[out];
        printf("    [Consumer] Consumed item %d from index %d\n", item, out);
        out = (out + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);
        sem_post(&empty);   // signal empty slot

        usleep(150000);
    }
    return NULL;
```

```
}

int main() {
    printf("Enter buffer size: ");
    scanf("%d", &BUFFER_SIZE);

    printf("Enter total number of items to produce/consume: ");
    scanf("%d", &TOTAL_ITEMS);

    buffer = (int*)malloc(sizeof(int) * BUFFER_SIZE);

    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    pthread_t prod, cons;
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    printf("\nAll items successfully produced and consumed.\n");

    return 0;
}
```

✅ **PROGRAM 2 — Producer–Consumer (Mutex + Condition Variables + User Input)**

File: prod_cons_cond.c

```
/*
--------------------------------------------------------------------------
Producer–Consumer using pthread mutex + condition variables
```

```
--------------------------------------------------------------------------------
User Input:
   - Buffer size
   - Total items

Shows:
   - Wait/signal events
   - Critical section
--------------------------------------------------------------------------------
*/

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int *buffer, in = 0, out = 0, count = 0;
int BUFFER_SIZE, TOTAL_ITEMS;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;

void *producer(void *arg) {
   for (int item = 1; item <= TOTAL_ITEMS; item++) {
      pthread_mutex_lock(&mutex);

      while (count == BUFFER_SIZE) {
         printf("[Producer] Buffer full — waiting...\n");
         pthread_cond_wait(&not_full, &mutex);
      }
```

```
        buffer[in] = item;
        printf("[Producer] Produced %d at index %d (count=%d)\n", item, in, count+1);
        in = (in + 1) % BUFFER_SIZE;
        count++;

        pthread_cond_signal(&not_empty);
        pthread_mutex_unlock(&mutex);

        usleep(120000);
    }
    return NULL;
}

void *consumer(void *arg) {
    for (int i = 1; i <= TOTAL_ITEMS; i++) {
        pthread_mutex_lock(&mutex);

        while (count == 0) {
            printf("    [Consumer] Buffer empty — waiting...\n");
            pthread_cond_wait(&not_empty, &mutex);
        }

        int item = buffer[out];
        printf("    [Consumer] Consumed %d from index %d (count=%d)\n", item, out, count-1);
        out = (out + 1) % BUFFER_SIZE;
        count--;

        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&mutex);

        usleep(150000);
```

```
    }
    return NULL;
}

int main() {
    printf("Enter buffer size: ");
    scanf("%d", &BUFFER_SIZE);

    printf("Enter total items: ");
    scanf("%d", &TOTAL_ITEMS);

    buffer = malloc(sizeof(int) * BUFFER_SIZE);

    pthread_t p, c;
    pthread_create(&p, NULL, producer, NULL);
    pthread_create(&c, NULL, consumer, NULL);

    pthread_join(p, NULL);
    pthread_join(c, NULL);

    return 0;
}
```

✅ **PROGRAM 3 — Producer–Consumer with Counter (step by step)**

File: prod_cons_counter.c

```
/*
-------------------------------------------------------------------------
Producer–Consumer with shared counter showing number of items in buffer.
-------------------------------------------------------------------------
*/
```

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

int *buffer, in=0, out=0, count=0;
int BUFFER_SIZE, TOTAL_ITEMS;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;

void *producer(void *arg) {
  for(int i=1;i<=TOTAL_ITEMS;i++){
    pthread_mutex_lock(&mutex);
    while(count==BUFFER_SIZE) pthread_cond_wait(&not_full,&mutex);

    buffer[in]=i;
    count++;
    printf("[Producer] %d produced | buffer count = %d\n", i, count);
    in=(in+1)%BUFFER_SIZE;

    pthread_cond_signal(&not_empty);
    pthread_mutex_unlock(&mutex);

    usleep(100000);
  }
  return NULL;
}

void *consumer(void *arg) {
  for(int i=1;i<=TOTAL_ITEMS;i++){
```

```
        pthread_mutex_lock(&mutex);
        while(count==0) pthread_cond_wait(&not_empty,&mutex);

        int item=buffer[out];
        count--;
        printf("    [Consumer] %d consumed | buffer count = %d\n", item, count);
        out=(out+1)%BUFFER_SIZE;

        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&mutex);

        usleep(150000);
    }
    return NULL;
}

int main(){
    printf("Enter buffer size: ");
    scanf("%d",&BUFFER_SIZE);

    printf("Enter total items: ");
    scanf("%d",&TOTAL_ITEMS);

    buffer = malloc(sizeof(int)*BUFFER_SIZE);

    pthread_t p,c;
    pthread_create(&p,NULL,producer,NULL);
    pthread_create(&c,NULL,consumer,NULL);

    pthread_join(p,NULL);
    pthread_join(c,NULL);
```

```
   return 0;
}
```

4) Reader–Writer (first readers–writers problem — readers priority)

File: reader_writer_steps.c
Compile: gcc reader_writer_steps.c -pthread -o reader_writer_steps
Run: ./reader_writer_steps

```
/*
reader_writer_steps.c
First Readers-Writers problem (readers priority).
Two readers and one writer (user can set number of read/write rounds).
Shows step-by-step entry/exit and synchronization.

Mechanism:
 - pthread_mutex_t rc_mutex protects readcount
 - sem_t wrt ensures exclusive writer access
*/

#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

int shared = 100;
int readcount = 0;
pthread_mutex_t rc_mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t wrt;

int ROUNDS;

void *reader(void *arg){
```

```
    int id = *(int*)arg;
    for(int r=0;r<ROUNDS;r++){
        // ENTRY
        pthread_mutex_lock(&rc_mutex);
        readcount++;
        if(readcount == 1){
            printf("[Reader %d] first reader, waiting for writer lock...\n", id);
            sem_wait(&wrt); // first reader blocks writers
            printf("[Reader %d] acquired writer lock for readers\n", id);
        }
        pthread_mutex_unlock(&rc_mutex);

        // CRITICAL (reading)
        printf("[Reader %d] reading shared = %d (round %d)\n", id, shared, r+1);
        usleep(150000);

        // EXIT
        pthread_mutex_lock(&rc_mutex);
        readcount--;
        if(readcount == 0){
            printf("[Reader %d] last reader, releasing writer lock\n", id);
            sem_post(&wrt);
        }
        pthread_mutex_unlock(&rc_mutex);

        usleep(100000);
    }
    return NULL;
}

void *writer(void *arg){
```

```
    int id = *(int*)arg;
    for(int r=0;r<ROUNDS;r++){
        printf("    [Writer] trying to acquire writer lock (round %d)\n", r+1);
        sem_wait(&wrt);
        // CRITICAL (writing)
        printf("    [Writer] writing: incrementing shared by 35\n");
        shared += 35;
        usleep(200000);
        printf("    [Writer] new shared = %d\n", shared);
        sem_post(&wrt);
        usleep(200000);
    }
    return NULL;
}

int main(){
    printf("Readers-Writers (readers priority) demonstration\n");
    printf("Enter number of rounds each reader/writer should perform: ");
    scanf("%d", &ROUNDS);

    sem_init(&wrt, 0, 1);

    pthread_t r1, r2, w;
    int id1 = 1, id2 = 2, wid = 1;
    pthread_create(&r1, NULL, reader, &id1);
    usleep(20000); // staggered start to show interleaving
    pthread_create(&r2, NULL, reader, &id2);
    usleep(20000);
    pthread_create(&w, NULL, writer, &wid);

    pthread_join(r1, NULL);
```

```
    pthread_join(r2, NULL);
    pthread_join(w, NULL);

    sem_destroy(&wrt);
    printf("Demonstration complete. Final shared = %d\n", shared);
    return 0;
}
```

---

5) Reader–Writer with File (readers print lowercase & uppercase; writer appends "hello")

File: reader_writer_file_steps.c
Compile: gcc reader_writer_file_steps.c -pthread -o reader_writer_file_steps
Run: ./reader_writer_file_steps sample.txt
(If sample.txt doesn't exist, program creates it.)

```
/*
reader_writer_file_steps.c
Two reader threads and one writer thread working on a common file.
 - Reader1: prints file contents as lowercase
 - Reader2: prints file contents as UPPERCASE
 - Writer: appends the string "hello" to the file

Synchronization:
 - pthread_mutex_t file_mutex protects file access (simple solution)
 - Readers and writer coordinate via the same mutex (serializes for simplicity)
This version shows step-by-step file operations.
*/

#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <unistd.h>
```

```
pthread_mutex_t file_mutex = PTHREAD_MUTEX_INITIALIZER;
const char *FNAME = "sample_file.txt";
char filename[256];

void ensure_file(){
   FILE *f = fopen(filename, "r");
   if(!f){
      f = fopen(filename, "w");
      fprintf(f, "This is Sample File.\nLine Two: RTOS Lab.\n");
      fclose(f);
      printf("Created sample file '%s' with initial text.\n", filename);
   } else fclose(f);
}

void *reader_lower(void *arg){
   (void)arg;
   pthread_mutex_lock(&file_mutex);
   printf("[Reader-lower] Opening file for reading\n");
   FILE *f = fopen(filename, "r");
   if(!f){ perror("fopen"); pthread_mutex_unlock(&file_mutex); return NULL; }
   printf("[Reader-lower] Contents (lowercase):\n");
   int c;
   while((c = fgetc(f)) != EOF) putchar(tolower((unsigned char)c));
   printf("\n[Reader-lower] Done reading.\n");
   fclose(f);
   pthread_mutex_unlock(&file_mutex);
   return NULL;
}

void *reader_upper(void *arg){
```

```
    (void)arg;
    pthread_mutex_lock(&file_mutex);
    printf("[Reader-UPPER] Opening file for reading\n");
    FILE *f = fopen(filename, "r");
    if(!f){ perror("fopen"); pthread_mutex_unlock(&file_mutex); return NULL; }
    printf("[Reader-UPPER] Contents (UPPERCASE):\n");
    int c;
    while((c = fgetc(f)) != EOF) putchar(toupper((unsigned char)c));
    printf("\n[Reader-UPPER] Done reading.\n");
    fclose(f);
    pthread_mutex_unlock(&file_mutex);
    return NULL;
}

void *writer_append(void *arg){
    (void)arg;
    pthread_mutex_lock(&file_mutex);
    printf("    [Writer] Opening file for appending\n");
    FILE *f = fopen(filename, "a");
    if(!f){ perror("fopen"); pthread_mutex_unlock(&file_mutex); return NULL; }
    fprintf(f, "hello");
    printf("    [Writer] Appended 'hello' to file\n");
    fclose(f);
    pthread_mutex_unlock(&file_mutex);
    return NULL;
}

int main(int argc, char **argv){
    if(argc<2) { printf("Usage: %s filename\nUsing default sample_file.txt\n", argv[0]);
strcpy(filename, FNAME); }
    else strcpy(filename, argv[1]);
```

```
    ensure_file();

    pthread_t r1, r2, w;
    pthread_create(&r1, NULL, reader_lower, NULL);
    usleep(30000);
    pthread_create(&r2, NULL, reader_upper, NULL);
    usleep(30000);
    pthread_create(&w, NULL, writer_append, NULL);

    pthread_join(r1, NULL);
    pthread_join(r2, NULL);
    pthread_join(w, NULL);

    printf("Final file contents (%s):\n", filename);
    FILE *f = fopen(filename, "r"); int ch;
    while((ch=fgetc(f))!=EOF) putchar(ch);
    fclose(f);
    return 0;
}
```

---

6) Dining Philosophers — show pick/release steps, allow N philosophers (user input)

File: dining_philosophers_steps.c
Compile: gcc dining_philosophers_steps.c -pthread -o dining_philosophers_steps
Run: ./dining_philosophers_steps — program asks for number of philosophers and cycles.

```
/*
dining_philosophers_steps.c
Dining philosophers with user-specified count (default 5).
Demonstrates forks as semaphores and uses odd/even pick-up to avoid deadlock.
Shows each philosopher thinking/picking forks/eating/releasing steps.
*/
```

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <unistd.h>

int N = 5;
sem_t *forks;
int CYCLES = 3;

void *philosopher(void *arg){
   int id = *(int*)arg;
   int left = id;
   int right = (id + 1) % N;
   for(int c=0;c<CYCLES;c++){
      printf("[P%d] thinking (cycle %d)\n", id, c+1);
      usleep(100000 + (id*10000));
      if(id % 2 == 0){
         printf("[P%d] picking left fork %d\n", id, left);
         sem_wait(&forks[left]);
         printf("[P%d] picking right fork %d\n", id, right);
         sem_wait(&forks[right]);
      } else {
         printf("[P%d] picking right fork %d\n", id, right);
         sem_wait(&forks[right]);
         printf("[P%d] picking left fork %d\n", id, left);
         sem_wait(&forks[left]);
      }
      printf("[P%d] eating (cycle %d)\n", id, c+1);
      usleep(150000 + (id*10000));
      sem_post(&forks[left]);
```

```
        sem_post(&forks[right]);
        printf("[P%d] released forks %d and %d\n", id, left, right);
        usleep(50000);
    }
    printf("[P%d] finished all cycles\n", id);
    return NULL;
}

int main(){
    printf("Dining Philosophers Demo\nEnter number of philosophers (default 5): ");
    char line[32]; if(fgets(line, sizeof(line), stdin) && sscanf(line, "%d", &N)!=1) N=5;
    printf("Enter cycles per philosopher (default 3): ");
    if(fgets(line, sizeof(line), stdin) && sscanf(line, "%d", &CYCLES)!=1) CYCLES=3;

    forks = malloc(sizeof(sem_t) * N);
    for(int i=0;i<N;i++) sem_init(&forks[i], 0, 1);

    pthread_t *ph = malloc(sizeof(pthread_t) * N);
    int *ids = malloc(sizeof(int) * N);
    for(int i=0;i<N;i++){ ids[i]=i; pthread_create(&ph[i], NULL, philosopher, &ids[i]); }

    for(int i=0;i<N;i++) pthread_join(ph[i], NULL);

    for(int i=0;i<N;i++) sem_destroy(&forks[i]);
    free(forks); free(ph); free(ids);
    printf("Dining philosophers demo complete.\n");
    return 0;
}
```

---

7) Peterson's Solution (show entry/exit and final value)

File: peterson_steps.c
Compile: gcc peterson_steps.c -pthread -o peterson_steps
Run: ./peterson_steps — program asks how many times each thread should enter critical section.

```
/*
peterson_steps.c
Peterson's solution for 2 threads. Each thread enters critical section multiple times.
Thread0 increments by 100 each time; Thread1 decrements by 75 each time.
Shows entry-wait, critical section, exit steps.
*/

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

volatile int flag[2] = {0,0};
volatile int turn = 0;
int shared = 0;
int ROUNDS = 3;

void enter_cs(int self){
   int other = 1 - self;
   flag[self] = 1;
   turn = other;
   // busy wait
   while(flag[other] && turn == other) { /* spin */ }
}

void leave_cs(int self){
   flag[self] = 0;
}
```

```
void *t_inc(void *arg){
  for(int i=0;i<ROUNDS;i++){
    printf("[T0] wants to enter CS (round %d)\n", i+1);
    enter_cs(0);
    printf("[T0] entered CS: shared (before) = %d\n", shared);
    shared += 100;
    usleep(100000);
    printf("[T0] leaving CS: shared (after) = %d\n", shared);
    leave_cs(0);
    usleep(120000);
  }
  return NULL;
}

void *t_dec(void *arg){
  for(int i=0;i<ROUNDS;i++){
    printf("    [T1] wants to enter CS (round %d)\n", i+1);
    enter_cs(1);
    printf("    [T1] entered CS: shared (before) = %d\n", shared);
    shared -= 75;
    usleep(100000);
    printf("    [T1] leaving CS: shared (after) = %d\n", shared);
    leave_cs(1);
    usleep(120000);
  }
  return NULL;
}

int main(){
  printf("Peterson's Solution Demo\nEnter rounds per thread (default 3): ");
  char buf[32]; if(fgets(buf,sizeof(buf),stdin) && sscanf(buf,"%d",&ROUNDS)!=1) ROUNDS=3;
```

```
    pthread_t a,b;
    pthread_create(&a,NULL,t_inc,NULL);
    pthread_create(&b,NULL,t_dec,NULL);
    pthread_join(a,NULL);
    pthread_join(b,NULL);

    printf("Final shared value = %d\n", shared);
    return 0;
}
```

---

8) Sleeping Barber (complete, step-by-step)

File: sleeping_barber_steps.c
Compile: gcc sleeping_barber_steps.c -pthread -o sleeping_barber_steps
Run: ./sleeping_barber_steps — program asks for number of chairs and number of customers.

```
/*
sleeping_barber_steps.c
Sleeping Barber problem with user-specified chairs and customers.
Semaphores used:
 - customers: counts waiting customers
 - barber: barber ready to cut
 - mutex: to protect waiting count
Shows steps: customers arrive, sit/leave, barber cuts, etc.
*/

#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <stdlib.h>

int CHAIRS = 3;
```

```
int NUM_CUSTOMERS = 10;
int waiting = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t customers; // counts waiting customers
sem_t barber;    // barber ready to cut

void *barber_func(void *a){
  while(1){
    sem_wait(&customers); // wait for a customer
    pthread_mutex_lock(&mutex);
    waiting--;
    printf("[Barber] Calls next customer. Waiting now = %d\n", waiting);
    sem_post(&barber); // signal barber ready for that customer
    pthread_mutex_unlock(&mutex);

    printf("[Barber] Cutting hair...\n");
    sleep(1); // cutting time
  }
  return NULL;
}

void *customer_func(void *arg){
  int id = *(int*)arg;
  printf("[Customer %d] Arrives\n", id);

  pthread_mutex_lock(&mutex);
  if(waiting < CHAIRS){
    waiting++;
    printf("[Customer %d] Sits in waiting room. waiting=%d\n", id, waiting);
    sem_post(&customers); // notify barber
    pthread_mutex_unlock(&mutex);
```

```
    sem_wait(&barber); // wait until barber is ready for this customer
    printf("[Customer %d] Getting haircut\n", id);
    // after haircut customer leaves
  } else {
    printf("[Customer %d] No chair available, leaves\n", id);
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}

int main(){
  printf("Sleeping Barber Demo\nEnter number of chairs: "); char buf[16];
  if(fgets(buf,sizeof(buf),stdin) && sscanf(buf,"%d",&CHAIRS)!=1) CHAIRS=3;
  printf("Enter number of customers: ");
  if(fgets(buf,sizeof(buf),stdin) && sscanf(buf,"%d",&NUM_CUSTOMERS)!=1)
NUM_CUSTOMERS=10;

  sem_init(&customers, 0, 0);
  sem_init(&barber, 0, 0);

  pthread_t b; pthread_create(&b,NULL,barber_func,NULL);
  pthread_t *cust = malloc(sizeof(pthread_t) * NUM_CUSTOMERS);
  int *ids = malloc(sizeof(int) * NUM_CUSTOMERS);

  for(int i=0;i<NUM_CUSTOMERS;i++){
    ids[i]=i+1;
    pthread_create(&cust[i], NULL, customer_func, &ids[i]);
    usleep(200000); // customers arrive spaced out
  }
```

```
    for(int i=0;i<NUM_CUSTOMERS;i++) pthread_join(cust[i], NULL);
    printf("All customers processed (some may have left). Program will keep barber thread alive; terminating main.\n");
    return 0;
}
```

---

9) Cigarette Smokers Problem (full working version)

File: cigarette_smokers.c
Compile: gcc cigarette_smokers.c -pthread -o cigarette_smokers
Run: ./cigarette_smokers — program runs a short demo then exits.

```
/*
cigarette_smokers.c
Complete working solution for the Cigarette Smokers problem.
 - Agent places two distinct items on table (tobacco, paper, match)
 - Three smokers, each has infinite of one item:
    Smoker 0: has tobacco, needs paper+match
    Smoker 1: has paper, needs tobacco+match
    Smoker 2: has match, needs tobacco+paper

Semaphores:
 - agent_sem to serialize agent placing items (not strictly needed)
 - mutex to protect table state
 - semaphores smoker_sem[3] to wake the appropriate smoker

This demo runs a limited number of iterations and then exits.
*/

#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
```

```
#define ITER 8

sem_t mutex;
sem_t agent_sem;
sem_t tobacco, paper, match;
sem_t smoker_sem[3]; // smoker semaphores

void *agent(void *a){
  for(int i=0;i<ITER;i++){
    sem_wait(&agent_sem);
    int choice = rand() % 3;
    if(choice == 0){
      // place tobacco + paper -> smoker with match (2) should act
      printf("[Agent] places TOBACCO + PAPER\n");
      sem_post(&tobacco);
      sem_post(&paper);
    } else if(choice == 1){
      // tobacco + match -> smoker with paper (1)
      printf("[Agent] places TOBACCO + MATCH\n");
      sem_post(&tobacco);
      sem_post(&match);
    } else {
      // paper + match -> smoker with tobacco (0)
      printf("[Agent] places PAPER + MATCH\n");
      sem_post(&paper);
      sem_post(&match);
    }
    usleep(200000);
  }
  return NULL;
}
```

```
// helper: when two items are present check who should be woken
void *watcher(void *a){
   // This watcher design uses semaphores for pair detection:
   // When tobacco & paper available -> signal smoker_with_match (2)
   // When tobacco & match available -> signal smoker_with_paper (1)
   // When paper & match available -> signal smoker_with_tobacco (0)
   // We'll implement simple polling with sem_trywait to detect pairs.
   int iterations = 0;
   while(iterations < ITER){
      // try to detect pairs by testing combinations
      if(sem_trywait(&tobacco) == 0){
         if(sem_trywait(&paper) == 0){
            // tobacco+paper -> smoker 2
            printf("[Watcher] detected TOBACCO + PAPER -> waking smoker 2\n");
            sem_post(&smoker_sem[2]);
            iterations++;
         } else {
            // not paper, restore tobacco
            sem_post(&tobacco);
         }
      }
      if(sem_trywait(&tobacco) == 0){
         if(sem_trywait(&match) == 0){
            printf("[Watcher] detected TOBACCO + MATCH -> waking smoker 1\n");
            sem_post(&smoker_sem[1]);
            iterations++;
         } else {
            sem_post(&tobacco);
         }
      }
```

```
      if(sem_trywait(&paper) == 0){
        if(sem_trywait(&match) == 0){
          printf("[Watcher] detected PAPER + MATCH -> waking smoker 0\n");
          sem_post(&smoker_sem[0]);
          iterations++;
        } else {
          sem_post(&paper);
        }
      }
      usleep(100000); // give time for agent
    }
    return NULL;
}

void *smoker(void *arg){
    int id = *(int*)arg;
    const char *name = (id==0)?"Smoker-Tobacco":(id==1)?"Smoker-Paper":"Smoker-Match";
    for(int i=0;i<ITER/2;i++){
      sem_wait(&smoker_sem[id]);
      // take items and smoke
      printf("[%s] got items and is smoking\n", name);
      usleep(300000);
      // done smoking, signal agent that more items can be placed
      sem_post(&agent_sem);
    }
    return NULL;
}

int main(){
    srand(1);
    sem_init(&mutex,0,1);
```

```
    sem_init(&agent_sem,0,1);
    sem_init(&tobacco,0,0);
    sem_init(&paper,0,0);
    sem_init(&match,0,0);
    for(int i=0;i<3;i++) sem_init(&smoker_sem[i],0,0);

    pthread_t ag, wt, s0, s1, s2;
    int id0=0,id1=1,id2=2;
    pthread_create(&ag,NULL,agent,NULL);
    pthread_create(&wt,NULL,watcher,NULL);
    pthread_create(&s0,NULL,smoker,&id0);
    pthread_create(&s1,NULL,smoker,&id1);
    pthread_create(&s2,NULL,smoker,&id2);

    pthread_join(ag,NULL);
    pthread_join(wt,NULL);
    pthread_join(s0,NULL);
    pthread_join(s1,NULL);
    pthread_join(s2,NULL);

    printf("Cigarette smokers demo finished.\n");
    return 0;
}
```

1) Banker's Algorithm — banker_steps.c

Compile:

gcc banker_steps.c -o banker_steps

Run:

./banker_steps

Code:

```
/*
banker_steps.c
Interactive Banker's Algorithm (safety algorithm + optional request test)
- Reads: number of processes (n) and resource types (m)
- Reads Allocation matrix (n x m)
- Reads Max matrix (n x m)
- Reads Available vector (m)
- Computes Need = Max - Allocation, prints matrices
- Runs Safety Algorithm and prints each step: which process can run, Work vector updates
- Optionally allows testing a new Request from a process and shows whether request can be granted safely.

Author: exam-friendly version (step-by-step)
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void print_vec(const char *label, int *v, int m){
   printf("%s: [", label);
   for(int j=0;j<m;j++) printf("%d%s", v[j], j==m-1? "": ", ");
   printf("]\n");
}

void print_matrix(const char *label, int **mat, int n, int m){
   printf("%s:\n", label);
   for(int i=0;i<n;i++){
      printf("P%d: ", i);
      for(int j=0;j<m;j++) printf("%d%s", mat[i][j], j==m-1? "": " ");
      printf("\n");
```

```
    }
}

int main(){
    int n, m;
    printf("BANKER'S ALGORITHM (step-by-step)\n");
    printf("Enter number of processes (n): ");
    if(scanf("%d", &n) != 1) return 0;
    printf("Enter number of resource types (m): ");
    if(scanf("%d", &m) != 1) return 0;

    // allocate matrices
    int **alloc = malloc(n * sizeof(int*));
    int **max = malloc(n * sizeof(int*));
    int **need = malloc(n * sizeof(int*));
    for(int i=0;i<n;i++){
        alloc[i] = calloc(m, sizeof(int));
        max[i] = calloc(m, sizeof(int));
        need[i] = calloc(m, sizeof(int));
    }
    int *avail = calloc(m, sizeof(int));

    printf("Enter Allocation matrix (n x m). For each process enter %d integers:\n", m);
    for(int i=0;i<n;i++){
        printf("Allocation for P%d: ", i);
        for(int j=0;j<m;j++) scanf("%d", &alloc[i][j]);
    }

    printf("Enter Max matrix (n x m). For each process enter %d integers:\n", m);
    for(int i=0;i<n;i++){
        printf("Max for P%d: ", i);
```

```
   for(int j=0;j<m;j++) scanf("%d", &max[i][j]);
}

printf("Enter Available vector (%d integers): ", m);
for(int j=0;j<m;j++) scanf("%d", &avail[j]);

// compute need
for(int i=0;i<n;i++)
   for(int j=0;j<m;j++)
      need[i][j] = max[i][j] - alloc[i][j];

printf("\n--- Input Summary ---\n");
print_matrix("Allocation", alloc, n, m);
print_matrix("Max", max, n, m);
print_matrix("Need", need, n, m);
print_vec("Available", avail, m);
printf("----------------------\n\n");

// Safety algorithm
int *work = malloc(m * sizeof(int));
int *finish = calloc(n, sizeof(int));
for(int j=0;j<m;j++) work[j] = avail[j];

printf("Running Safety Algorithm:\n");
int safe_seq[n];
int found_any = 1;
int seq_index = 0;

// iterate until cannot find any new process
while(found_any){
   found_any = 0;
```

```
  for(int i=0;i<n;i++){
    if(finish[i]) continue;
    int can = 1;
    for(int j=0;j<m;j++){
      if(need[i][j] > work[j]) { can = 0; break; }
    }
    if(can){
      // print the step
      printf("-> Process P%d can be satisfied. Work before: ", i);
      print_vec("", work, m);
      // simulate completion
      for(int j=0;j<m;j++) work[j] += alloc[i][j];
      finish[i] = 1;
      safe_seq[seq_index++] = i;
      found_any = 1;
      printf("   After P%d finishes, Work becomes: ", i);
      print_vec("", work, m);
    }
  }
}

int all_finished = 1;
for(int i=0;i<n;i++) if(!finish[i]) all_finished = 0;

if(all_finished){
  printf("\nSystem IS IN A SAFE STATE.\nSafe sequence: ");
  for(int i=0;i<seq_index;i++){
    if(i) printf(" -> ");
    printf("P%d", safe_seq[i]);
  }
  printf("\n");
```

```
} else {
   printf("\nSystem is NOT in a safe state. Processes not finished: ");
   for(int i=0;i<n;i++) if(!finish[i]) printf("P%d ", i);
   printf("\n");
}

// Offer to test a request
char choice;
printf("\nDo you want to test a resource request? (y/n): ");
scanf(" %c", &choice);
if(choice == 'y' || choice == 'Y'){
   int pid;
   int *req = malloc(m * sizeof(int));
   printf("Enter process id making request (0..%d): ", n-1);
   scanf("%d", &pid);
   printf("Enter request vector (%d ints): ", m);
   for(int j=0;j<m;j++) scanf("%d", &req[j]);

   // check req <= need
   int ok = 1;
   for(int j=0;j<m;j++) if(req[j] > need[pid][j]) ok = 0;
   if(!ok){
      printf("Request INVALID: requested more than declared need.\n");
   } else {
      // check req <= avail
      int ok2 = 1;
      for(int j=0;j<m;j++) if(req[j] > avail[j]) ok2 = 0;
      if(!ok2){
         printf("Request cannot be granted now (not enough available resources).\n");
      } else {
         // pretend allocate: avail' = avail - req; alloc'[pid]+=req; need'[pid]-=req
```

```
printf("Pretend allocation: simulate safety check after allocation...\n");
int *work2 = malloc(m * sizeof(int));
int **alloc2 = malloc(n * sizeof(int*));
int **need2 = malloc(n * sizeof(int*));
int *finish2 = calloc(n, sizeof(int));
for(int i=0;i<n;i++){ alloc2[i] = malloc(m*sizeof(int)); need2[i] = malloc(m*sizeof(int)); }
for(int j=0;j<m;j++) work2[j] = avail[j] - req[j];
for(int i=0;i<n;i++){
  for(int j=0;j<m;j++){
    alloc2[i][j] = alloc[i][j];
    need2[i][j] = need[i][j];
  }
}
for(int j=0;j<m;j++){
  alloc2[pid][j] += req[j];
  need2[pid][j] -= req[j];
}
printf("Work initially (after pretend allocation): ");
print_vec("", work2, m);
// run safety
int seq2[n], idx2=0;
int progress = 1;
while(progress){
  progress = 0;
  for(int i=0;i<n;i++){
    if(finish2[i]) continue;
    int can=1;
    for(int j=0;j<m;j++) if(need2[i][j] > work2[j]) { can=0; break; }
    if(can){
      printf(" -> After allocation, P%d can be satisfied. Work before: ", i);
      print_vec("", work2, m);
```

```
                for(int j=0;j<m;j++) work2[j] += alloc2[i][j];
                finish2[i]=1; seq2[idx2++]=i;
                printf("    After P%d finishes, Work becomes: ", i);
                print_vec("", work2, m);
                progress=1;
              }
            }
          }
          int safe2 = 1;
          for(int i=0;i<n;i++) if(!finish2[i]) safe2 = 0;
          if(safe2){
            printf("Request CAN BE GRANTED: system remains SAFE. Safe sequence: ");
            for(int i=0;i<idx2;i++){
              if(i) printf(" -> ");
              printf("P%d", seq2[i]);
            }
            printf("\n");
          } else {
            printf("Request CANNOT BE GRANTED: system would be UNSAFE.\n");
          }
          // free temp
          for(int i=0;i<n;i++){ free(alloc2[i]); free(need2[i]); }
          free(alloc2); free(need2); free(work2); free(finish2);
        }
      }
      free(req);
    }

    // free memory
    for(int i=0;i<n;i++){ free(alloc[i]); free(max[i]); free(need[i]); }
    free(alloc); free(max); free(need); free(avail); free(work); free(finish);
```

```
    return 0;
}
```

---

2) Deadlock Detection using Resource Allocation Graph (RAG) — rag_deadlock_steps.c

This program gives you two input modes:

- **Mode A:** Build a RAG from processes and resources (you enter processes, resources, and edges).
- **Mode B:** Simpler general directed graph cycle detection — useful if you convert your RAG to a process-only wait-for graph (WFG).

It prints DFS stack steps and reports whether a cycle (deadlock) exists and where it was detected.

Compile:

gcc rag_deadlock_steps.c -o rag_deadlock_steps

Run:

./rag_deadlock_steps

Code:

```
/*
rag_deadlock_steps.c
Deadlock detection in Resource Allocation Graph (RAG) using DFS cycle detection.
Two modes:
 1) RAG mode - user provides processes (P), resources (R) and edges:
     - request edge: P -> R
     - allocation edge: R -> P
   The program builds a directed graph of nodes (P0..Pn-1, R0..Rm-1) and detects cycles.
 2) General directed graph mode - input nodes and edges and detect cycle (useful for Wait-For Graph).

Output:
 - Prints each DFS visit, recursion stack, and when a back-edge (cycle) is found prints the cycle trace.
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXN 200

int **adj; // adjacency matrix dynamic
int N;    // total nodes

void print_stack(int stack[], int top){
   printf("    DFS stack [bottom->top]: ");
   for(int i=0;i<=top;i++) printf("%d ", stack[i]);
   printf("\n");
}

int dfs_cycle(int u, int visited[], int recStack[], int stack[], int *top){
   visited[u] = 1;
   recStack[u] = 1;
   stack[++(*top)] = u;
   printf("Visiting node %d. Mark visited. ", u);
   print_stack(stack, *top);

   for(int v=0; v<N; v++){
      if(!adj[u][v]) continue;
      printf("  Edge %d -> %d\n", u, v);
      if(!visited[v]){
         printf("    Node %d not visited, recursing...\n", v);
         if(dfs_cycle(v, visited, recStack, stack, top)) return 1;
      } else if(recStack[v]){
         // back edge found -> cycle
```

```
        printf("    Back-edge detected from %d to %d (node in recursion stack). Cycle found!\n", u,
v);
        // print cycle by scanning stack
        printf("    Cycle trace: ");
        int started = 0;
        for(int i=0;i<=*top;i++){
          if(stack[i] == v) started = 1;
          if(started) printf("%d ", stack[i]);
        }
        printf("%d\n", v); // close cycle
        return 1;
      } else {
        printf("    Node %d already visited and not in recursion stack (ignore)\n", v);
      }
    }

    // backtrack
    recStack[u] = 0;
    printf("Backtracking from node %d. Pop stack.\n", u);
    (*top)--;
    return 0;
}

int detect_cycle(){
    int *visited = calloc(N, sizeof(int));
    int *recStack = calloc(N, sizeof(int));
    int *stack = calloc(N, sizeof(int));
    int top = -1;
    for(int i=0;i<N;i++){
      if(!visited[i]){
        if(dfs_cycle(i, visited, recStack, stack, &top)){
```

```
            free(visited); free(recStack); free(stack);
            return 1;
          }
        }
      }
    free(visited); free(recStack); free(stack);
    return 0;
}

int main(){
    printf("Deadlock detection via Resource Allocation Graph (RAG)\n");
    printf("Choose mode:\n 1 - RAG (process/resource)\n 2 - General directed graph\nChoice: ");
    int mode; if(scanf("%d", &mode) != 1) return 0;

    if(mode == 1){
        int P, R;
        printf("Enter number of processes (P): ");
        scanf("%d", &P);
        printf("Enter number of resources (R): ");
        scanf("%d", &R);
        N = P + R;
        adj = malloc(N * sizeof(int*));
        for(int i=0;i<N;i++){ adj[i] = calloc(N, sizeof(int)); }

        printf("We label processes P0..P%d (nodes 0..%d)\n", P-1, P-1);
        printf("Resources R0..R%d are nodes %d..%d\n", R-1, P, P+R-1);

        printf("\nEnter number of edges in the graph: ");
        int E; scanf("%d", &E);
        printf("For each edge enter: type u v\n type=1 -> request edge P->R (enter P-index R-index)\n
type=2 -> allocation edge R->P (enter R-index P-index)\n");
```

```
    for(int e=0;e<E;e++){
        int type, a, b;
        printf("Edge %d: ", e+1); scanf("%d %d %d", &type, &a, &b);
        if(type == 1){
            // P->R : node a -> node P + b
            int u = a; int v = P + b;
            adj[u][v] = 1;
        } else {
            // R->P : node P + a -> node b
            int u = P + a; int v = b;
            adj[u][v] = 1;
        }
    }

    printf("\nAdjacency list (as edges):\n");
    for(int i=0;i<N;i++){
        for(int j=0;j<N;j++) if(adj[i][j]) printf("%d -> %d\n", i, j);
    }
    printf("\nRunning cycle detection (DFS) with detailed steps:\n");
    int has_cycle = detect_cycle();
    if(has_cycle) printf("\nDeadlock (cycle) detected in RAG.\n");
    else printf("\nNo cycle detected. No deadlock.\n");

    for(int i=0;i<N;i++) free(adj[i]);
    free(adj);
} else {
    // general directed graph
    printf("Enter number of nodes: ");
    scanf("%d", &N);
    adj = malloc(N * sizeof(int*));
    for(int i=0;i<N;i++){ adj[i] = calloc(N, sizeof(int)); }
```

```
    printf("Enter number of directed edges: ");
    int E; scanf("%d",&E);
    printf("Enter edges as pairs: u v (0-based node indices)\n");
    for(int i=0;i<E;i++){ int u,v; scanf("%d %d",&u,&v); adj[u][v]=1; }

    printf("\nEdges entered:\n");
    for(int i=0;i<N;i++) for(int j=0;j<N;j++) if(adj[i][j]) printf("%d -> %d\n", i, j);

    printf("\nRunning DFS-based cycle detection (detailed):\n");
    int has_cycle = detect_cycle();
    if(has_cycle) printf("\nCycle detected in graph (deadlock possible).\n");
    else printf("\nNo cycle detected in graph.\n");

    for(int i=0;i<N;i++) free(adj[i]); free(adj);
  }

  return 0;
}
```

✅ **1) FCFS Scheduling (non-preemptive)**

File: fcfs_steps.c
Compile:

```
gcc fcfs_steps.c -o fcfs_steps
```

```
/*
fcfs_steps.c
FCFS Scheduling — Non-Preemptive
Input:
  n
  AT BT (for each process)
Output:
```

```
   - Step-by-step CPU timeline
   - Completion, Turnaround, Waiting Time for each process
   - Average WT, Average TAT
*/

#include <stdio.h>

struct Process {
   int id, at, bt;
   int ct, tat, wt;
};

int main(){
   int n;
   printf("Enter number of processes: ");
   scanf("%d",&n);

   struct Process p[n];
   for(int i=0;i<n;i++){
      p[i].id=i;
      printf("Enter AT and BT for P%d: ",i);
      scanf("%d %d",&p[i].at,&p[i].bt);
   }

   // Sort by Arrival Time
   for(int i=0;i<n-1;i++)
      for(int j=i+1;j<n;j++)
         if(p[i].at > p[j].at){
            struct Process t=p[i]; p[i]=p[j]; p[j]=t;
         }
```

```
    int time = 0;
    printf("\n--- CPU Timeline (FCFS) ---\n");
    for(int i=0;i<n;i++){
        if(time < p[i].at){
            printf("Time %d → %d : CPU IDLE\n", time, p[i].at);
            time = p[i].at;
        }
        printf("Time %d: P%d START\n", time, p[i].id);
        time += p[i].bt;
        p[i].ct = time;
        printf("Time %d: P%d FINISH\n", time, p[i].id);
    }

    double totalWT=0, totalTAT=0;
    printf("\n--- Results ---\n");
    for(int i=0;i<n;i++){
        p[i].tat = p[i].ct - p[i].at;
        p[i].wt = p[i].tat - p[i].bt;
        totalWT += p[i].wt;
        totalTAT += p[i].tat;

        printf("P%d: CT=%d TAT=%d WT=%d\n", p[i].id, p[i].ct, p[i].tat, p[i].wt);
    }

    printf("Average WT = %.2f\n", totalWT/n);
    printf("Average TAT = %.2f\n", totalTAT/n);
    return 0;
}
```

✅ **2) SJF Scheduling (Non-Preemptive)**

File: sjf_nonpreemptive_steps.c
Compile:

gcc sjf_nonpreemptive_steps.c -o sjf_np

```
/*
sjf_nonpreemptive_steps.c
Shortest Job First — NON-PREEMPTIVE

Algorithm:
  - At any time when CPU becomes free, pick the process with smallest BT among arrived ones.
*/

#include <stdio.h>

struct Process {
  int id, at, bt;
  int ct, tat, wt, done;
};

int main(){
  int n;
  printf("Enter number of processes: ");
  scanf("%d",&n);

  struct Process p[n];
  for(int i=0;i<n;i++){
    p[i].id=i;
    p[i].done=0;
    printf("Enter AT and BT for P%d: ",i);
    scanf("%d %d",&p[i].at,&p[i].bt);
  }
```

```
int completed=0, time=0;

printf("\n--- CPU Timeline (SJF Non-Preemptive) ---\n");
while(completed < n){
   int idx = -1;
   int bestBT = 1e9;

   for(int i=0;i<n;i++){
      if(!p[i].done && p[i].at <= time && p[i].bt < bestBT){
         bestBT = p[i].bt;
         idx = i;
      }
   }

   if(idx == -1){
      printf("Time %d: CPU IDLE\n", time);
      time++;
      continue;
   }

   printf("Time %d: P%d START\n", time, p[idx].id);
   time += p[idx].bt;
   p[idx].ct = time;
   p[idx].done = 1;
   completed++;
   printf("Time %d: P%d FINISH\n", time, p[idx].id);
}

double totalWT=0,totalTAT=0;
printf("\n--- Results ---\n");
for(int i=0;i<n;i++){
```

```
    p[i].tat = p[i].ct - p[i].at;
    p[i].wt = p[i].tat - p[i].bt;
    totalWT += p[i].wt;
    totalTAT += p[i].tat;
    printf("P%d: CT=%d TAT=%d WT=%d\n", i, p[i].ct, p[i].tat, p[i].wt);
  }

  printf("Average WT = %.2f\n", totalWT/n);
  printf("Average TAT = %.2f\n", totalTAT/n);
  return 0;
}
```

✅ **3) SRTF / SJF Preemptive (line-by-line events)**

File: srtf_steps.c
Compile:

```
gcc srtf_steps.c -o srtf_steps
```

```
/*
srtf_steps.c
Shortest Remaining Time First — PREEMPTIVE
Shows:
  - Every time a process is preempted
  - Start, resume, finish events
*/

#include <stdio.h>

struct Process {
  int id, at, bt, rem;
  int ct, tat, wt;
};
```

```
int main(){
   int n;
   printf("Enter number of processes: ");
   scanf("%d",&n);

   struct Process p[n];
   for(int i=0;i<n;i++){
      p[i].id=i;
      printf("Enter AT and BT for P%d: ",i);
      scanf("%d %d",&p[i].at,&p[i].bt);
      p[i].rem = p[i].bt;
   }

   int time=0, completed=0, last=-1;

   printf("\n--- CPU Timeline (SRTF) ---\n");
   while(completed < n){
      int idx=-1;
      int best=1e9;

      for(int i=0;i<n;i++){
         if(p[i].at <= time && p[i].rem > 0 && p[i].rem < best){
            best = p[i].rem;
            idx = i;
         }
      }

      if(idx == -1){
         printf("Time %d: CPU IDLE\n", time);
         time++;
         continue;
```

```
        }

        if(idx != last)
            printf("Time %d: CPU switches to P%d\n", time, idx);

        p[idx].rem--;
        time++;

        if(p[idx].rem == 0){
            p[idx].ct = time;
            completed++;
            printf("Time %d: P%d FINISHED\n", time, idx);
        }

        last = idx;
    }

    double totalWT=0,totalTAT=0;
    printf("\n--- Results ---\n");
    for(int i=0;i<n;i++){
        p[i].tat = p[i].ct - p[i].at;
        p[i].wt = p[i].tat - p[i].bt;
        totalWT += p[i].wt;
        totalTAT += p[i].tat;
        printf("P%d: CT=%d TAT=%d WT=%d\n", i, p[i].ct, p[i].tat, p[i].wt);
    }

    printf("Average WT = %.2f\n", totalWT/n);
    printf("Average TAT = %.2f\n", totalTAT/n);
    return 0;
}
```

✅ **4) Priority Scheduling (Non-Preemptive)**

File: priority_nonpreemptive_steps.c
Compile:

gcc priority_nonpreemptive_steps.c -o priority_np

```
/*
priority_nonpreemptive_steps.c
Lower number = Higher priority
*/

#include <stdio.h>

struct Process {
  int id, at, bt, pr;
  int ct, tat, wt, done;
};

int main(){
  int n;
  printf("Enter number of processes: "); scanf("%d",&n);

  struct Process p[n];
  for(int i=0;i<n;i++){
    p[i].id=i; p[i].done=0;
    printf("Enter AT, BT, Priority for P%d: ",i);
    scanf("%d %d %d",&p[i].at,&p[i].bt,&p[i].pr);
  }

  int time=0, completed=0;

  printf("\n--- CPU Timeline (Priority Non-Preemptive) ---\n");
```

```
while(completed < n){
   int idx=-1;
   int bestPR=1e9;

   for(int i=0;i<n;i++){
      if(!p[i].done && p[i].at <= time){
         if(p[i].pr < bestPR){
            bestPR = p[i].pr;
            idx = i;
         }
      }
   }

   if(idx == -1){
      printf("Time %d: CPU IDLE\n", time);
      time++;
      continue;
   }

   printf("Time %d: P%d START (priority=%d)\n", time, idx, p[idx].pr);
   time += p[idx].bt;
   p[idx].ct = time;
   p[idx].done = 1;
   printf("Time %d: P%d FINISH\n", time, idx);
   completed++;
}

double totalWT=0,totalTAT=0;
printf("\n--- Results ---\n");
for(int i=0;i<n;i++){
   p[i].tat = p[i].ct - p[i].at;
```

```
        p[i].wt = p[i].tat - p[i].bt;
        totalWT += p[i].wt;
        totalTAT += p[i].tat;
        printf("P%d: Priority=%d CT=%d TAT=%d WT=%d\n",
            i, p[i].pr, p[i].ct, p[i].tat, p[i].wt);
    }

    printf("Average WT = %.2f\n", totalWT/n);
    printf("Average TAT = %.2f\n", totalTAT/n);

    return 0;
}
```

✅ **5) Priority Scheduling (Preemptive)**

File: priority_preemptive_steps.c
Compile:

```
gcc priority_preemptive_steps.c -o priority_pre
```

```
/*
priority_preemptive_steps.c
Lower number = Higher priority
Switch whenever a new higher-priority process arrives.
*/

#include <stdio.h>

struct Process {
    int id, at, bt, pr;
    int rem;
    int ct, tat, wt;
};
```

```
int main(){
   int n;
   printf("Enter number of processes: "); scanf("%d",&n);

   struct Process p[n];
   for(int i=0;i<n;i++){
      p[i].id=i;
      printf("Enter AT, BT, Priority for P%d: ",i);
      scanf("%d %d %d",&p[i].at,&p[i].bt,&p[i].pr);
      p[i].rem = p[i].bt;
   }

   int time=0, completed=0, last=-1;

   printf("\n--- CPU Timeline (Priority Preemptive) ---\n");

   while(completed < n){
      int idx=-1;
      int bestPR=1e9;

      for(int i=0;i<n;i++){
         if(p[i].at <= time && p[i].rem > 0 && p[i].pr < bestPR){
            bestPR = p[i].pr;
            idx = i;
         }
      }

      if(idx == -1){
         printf("Time %d: CPU IDLE\n", time);
         time++;
         continue;
```

```
        }

        if(idx != last)
            printf("Time %d: Switch to P%d (priority=%d)\n", time, idx, p[idx].pr);

        p[idx].rem--;
        time++;

        if(p[idx].rem == 0){
            p[idx].ct = time;
            completed++;
            printf("Time %d: P%d FINISHED\n", time, idx);
        }

        last = idx;
    }

    double totalWT=0,totalTAT=0;
    printf("\n--- Results ---\n");
    for(int i=0;i<n;i++){
        p[i].tat = p[i].ct - p[i].at;
        p[i].wt = p[i].tat - p[i].bt;
        printf("P%d: Priority=%d CT=%d TAT=%d WT=%d\n",
               i, p[i].pr, p[i].ct, p[i].tat, p[i].wt);
        totalWT += p[i].wt;
        totalTAT += p[i].tat;
    }

    printf("Avg WT = %.2f\nAvg TAT = %.2f\n", totalWT/n, totalTAT/n);
    return 0;
}
```

✅ **6) Round Robin Scheduling (with timeline + preemption)**

File: rr_steps.c
Compile:

gcc rr_steps.c -o rr_steps

```
/*
rr_steps.c
Round Robin Scheduling with time quantum (preemptive)
Shows:
  - Every CPU switch
  - Remaining burst times
  - Completion time
*/

#include <stdio.h>

struct Process {
  int id, at, bt;
  int rem;
  int ct, tat, wt;
};

int main(){
  int n, q;
  printf("Enter number of processes: "); scanf("%d",&n);
  printf("Enter time quantum: "); scanf("%d",&q);

  struct Process p[n];
  for(int i=0;i<n;i++){
    p[i].id=i;
    printf("Enter AT, BT for P%d: ",i);
```

```
    scanf("%d %d",&p[i].at,&p[i].bt);
    p[i].rem = p[i].bt;
}

int time=0, completed=0;
int queue[1000], front=0, rear=0;
int visited[n]; for(int i=0;i<n;i++) visited[i]=0;

printf("\n--- CPU Timeline (Round Robin) ---\n");

// load initial processes
while(completed < n){
    for(int i=0;i<n;i++){
        if(p[i].at == time && !visited[i]){
            queue[rear++] = i;
            visited[i] = 1;
            printf("Time %d: P%d ARRIVED\n", time, i);
        }
    }

    if(front == rear){
        printf("Time %d: CPU IDLE\n", time);
        time++;
        continue;
    }

    int idx = queue[front++];
    printf("Time %d: CPU runs P%d (rem=%d)\n", time, idx, p[idx].rem);

    int run = (p[idx].rem < q)? p[idx].rem : q;
    p[idx].rem -= run;
```

```
    time += run;

    // new arrivals during this run
    for(int t=time-run+1; t<=time; t++){
        for(int i=0;i<n;i++){
            if(p[i].at == t && !visited[i]){
                queue[rear++] = i;
                visited[i] = 1;
                printf("Time %d: P%d ARRIVED\n", t, i);
            }
        }
    }

    if(p[idx].rem == 0){
        p[idx].ct = time;
        completed++;
        printf("Time %d: P%d FINISHED\n", time, idx);
    } else {
        queue[rear++] = idx;
        printf("Time %d: P%d PREEMPTED (rem=%d)\n", time, idx, p[idx].rem);
    }
}

double totalWT=0,totalTAT=0;
printf("\n--- Results ---\n");
for(int i=0;i<n;i++){
    p[i].tat = p[i].ct - p[i].at;
    p[i].wt = p[i].tat - p[i].bt;
    printf("P%d: CT=%d TAT=%d WT=%d\n", i, p[i].ct, p[i].tat, p[i].wt);
    totalWT += p[i].wt;
    totalTAT += p[i].tat;
```

```
    }

    printf("Average WT = %.2f\nAverage TAT = %.2f\n", totalWT/n, totalTAT/n);

    return 0;
}
```

⭐ PROGRAM 1 — POSIX Shared Memory Writer

File: shm_writer.c

This program creates shared memory and writes a user-provided string into it.

Compile:

gcc shm_writer.c -o shm_writer -lrt

Run:

./shm_writer

```
/*
shm_writer.c
Creates a POSIX shared memory region and writes a user-given string.
Another program (reader) will open and read this memory.

Functions used:
    shm_open, ftruncate, mmap, close
*/

#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <string.h>

int main() {
```

```
    char input[100];
    printf("Enter a string to write into shared memory: ");
    scanf(" %[^\n]", input);

    int fd = shm_open("/myshm", O_CREAT | O_RDWR, 0666);
    ftruncate(fd, 1024);

    char *ptr = mmap(0, 1024, PROT_WRITE, MAP_SHARED, fd, 0);

    strcpy(ptr, input);

    printf("Writer: Stored '%s' into shared memory.\n", input);

    close(fd);
    return 0;
}
```

⭐ **PROGRAM 2 — POSIX Shared Memory Reader + Case Converter**

File: shm_reader.c

Reads the string from shared memory and converts it to uppercase.

Compile:

```
gcc shm_reader.c -o shm_reader -lrt
```

Run:

```
./shm_reader
```

```
/*
shm_reader.c
Opens existing shared memory, reads string, converts to uppercase.
*/

#include <stdio.h>
#include <fcntl.h>
```

```
#include <sys/mman.h>
#include <unistd.h>
#include <ctype.h>

int main() {
   int fd = shm_open("/myshm", O_RDONLY, 0666);
   if (fd < 0) {
      printf("Shared memory not found. Run writer first!\n");
      return 1;
   }

   char *ptr = mmap(0, 1024, PROT_READ, MAP_SHARED, fd, 0);

   printf("Reader: Original string from shared memory: %s\n", ptr);

   printf("Reader: Uppercase version: ");
   for(int i=0; ptr[i] != '\0'; i++)
      putchar(toupper(ptr[i]));
   printf("\n");

   close(fd);
   shm_unlink("/myshm");
   return 0;
}
```

⭐ **PROGRAM 3 — Thread Example: Two Threads Printing Messages**

File: two_threads.c

```
/*
two_threads.c
Creates two threads that print messages independently.
*/
```

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

void *thread1(void *arg) {
   printf("This is thread ONE\n");
   return NULL;
}

void *thread2(void *arg) {
   printf("This is thread TWO\n");
   return NULL;
}

int main() {
   pthread_t t1, t2;

   pthread_create(&t1, NULL, thread1, NULL);
   pthread_create(&t2, NULL, thread2, NULL);

   pthread_join(t1, NULL);
   pthread_join(t2, NULL);

   return 0;
}
```

⭐ **PROGRAM 4 — Thread: Matrix Row Sum**

File: row_sum_threads.c

```
/*
row_sum_threads.c
User enters matrix. Each thread computes ONE ROW SUM.
```

```
*/

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

int m, n;
int mat[20][20];
int result[20];

void *compute_row(void *arg) {
   int r = *(int*)arg;
   int sum = 0;

   for(int j = 0; j < n; j++)
      sum += mat[r][j];

   result[r] = sum;
   printf("Thread for Row %d: Sum = %d\n", r, sum);

   return NULL;
}

int main() {
   printf("Enter rows and columns (m n): ");
   scanf("%d %d", &m, &n);

   printf("Enter matrix values:\n");
   for(int i=0;i<m;i++)
      for(int j=0;j<n;j++)
         scanf("%d", &mat[i][j]);
```

```
    pthread_t tid[m];
    int rows[m];

    for(int i=0;i<m;i++){
        rows[i]=i;
        pthread_create(&tid[i], NULL, compute_row, &rows[i]);
    }

    for(int i=0;i<m;i++)
        pthread_join(tid[i], NULL);

    int total = 0;
    for(int i=0;i<m;i++)
        total += result[i];

    printf("\nTotal of all row sums = %d\n", total);
    return 0;
}
```

⭐ **PROGRAM 5 — Thread: Word Count + Lowercase Converter in File**

File: file_threads.c

```
/*
file_threads.c
Thread 1: Counts words in a file
Thread 2: Converts entire file to lowercase
*/

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

```
char filename[100];

void *count_words(void *arg) {
   FILE *fp = fopen(filename, "r");
   if(!fp){ printf("File not found!\n"); return NULL; }

   int words = 0, in_word = 0;
   char ch;

   while((ch = fgetc(fp)) != EOF){
      if(isspace(ch)) in_word = 0;
      else if(!in_word){
         in_word = 1;
         words++;
      }
   }

   printf("Thread 1: Total words = %d\n", words);
   fclose(fp);
   return NULL;
}

void *lowercase_convert(void *arg) {
   FILE *fp = fopen(filename, "r+");
   if(!fp){ printf("File not found!\n"); return NULL; }

   char ch; long pos;
   while((ch = fgetc(fp)) != EOF){
      pos = ftell(fp);
      fseek(fp, pos-1, SEEK_SET);
      fputc(tolower(ch), fp);
```

```
    }

    fclose(fp);
    printf("Thread 2: File converted to lowercase.\n");
    return NULL;
}

int main() {
    printf("Enter filename: ");
    scanf("%s", filename);

    pthread_t t1,t2;
    pthread_create(&t1, NULL, count_words, NULL);
    pthread_create(&t2, NULL, lowercase_convert, NULL);

    pthread_join(t1,NULL);
    pthread_join(t2,NULL);

    return 0;
}
```

⭐ **PROGRAM 6 — Thread: Shared Memory Transformation**

File: thread_shared_transform.c

```
/*
Thread 1 writes "Hello There"
Thread 2 reads it and toggles case (a→A, A→a)
Thread 1 prints modified result
*/

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

```
#include <string.h>
#include <ctype.h>

char shared[100];

void *writer(void *arg) {
   strcpy(shared, "Hello There");
   printf("Writer Thread: Stored '%s'\n", shared);
   return NULL;
}

void *reader(void *arg) {
   for(int i=0; shared[i] != '\0'; i++){
      if(islower(shared[i])) shared[i] = toupper(shared[i]);
      else shared[i] = tolower(shared[i]);
   }
   printf("Reader Thread: Modified string = %s\n", shared);
   return NULL;
}

int main(){
   pthread_t w, r;

   pthread_create(&w, NULL, writer, NULL);
   pthread_join(w, NULL);

   pthread_create(&r, NULL, reader, NULL);
   pthread_join(r, NULL);

   printf("Main: Final result = %s\n", shared);
```

```
    return 0;
}
```

⭐ PROGRAM 7 — Producer–Consumer Using Threads (Mutex + Condition Variables)

This is an easier version for thread-level producer–consumer.

File: thread_pc.c

```
/*
thread_pc.c
Thread-based Producer–Consumer using mutex + condition variables
*/

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int buffer[5];
int in=0, out=0, count=0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;

void *producer(void *arg){
    for(int i=1;i<=10;i++){
        pthread_mutex_lock(&mutex);
        while(count==5) pthread_cond_wait(&not_full, &mutex);

        buffer[in] = i;
        printf("[Producer] %d produced\n", i);
        in = (in+1)%5;
        count++;
```

```
        pthread_cond_signal(&not_empty);
        pthread_mutex_unlock(&mutex);
        usleep(100000);
    }
    return NULL;
}

void *consumer(void *arg){
    for(int i=1;i<=10;i++){
        pthread_mutex_lock(&mutex);
        while(count==0) pthread_cond_wait(&not_empty, &mutex);

        int item = buffer[out];
        printf("    [Consumer] %d consumed\n", item);
        out = (out+1)%5;
        count--;

        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&mutex);
        usleep(150000);
    }
    return NULL;
}

int main(){
    pthread_t p,c;
    pthread_create(&p,NULL,producer,NULL);
    pthread_create(&c,NULL,consumer,NULL);

    pthread_join(p,NULL);
```

```
    pthread_join(c,NULL);

    return 0;
}
```

All programs compile cleanly with:

gcc program_name.c -o program_name

⭐ **PROGRAM 1 — Simple fork() showing parent/child PIDs**

File: fork_basic.c

```
/*
fork_basic.c
Demonstrate fork(): parent and child display PID, PPID and a message.
*/

#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        printf("Fork failed!\n");
        return 1;
    }
    else if (pid == 0) {
        printf("Child Process:\n");
        printf("PID = %d, PPID = %d\n", getpid(), getppid());
        printf("I am the child.\n");
    }
    else {
```

```
    printf("Parent Process:\n");
    printf("PID = %d, PPID = %d\n", getpid(), getppid());
    printf("I am the parent.\n");
  }

  return 0;
}
```

⭐ **PROGRAM 2 — fork + exec ls (with & without wait())**

File: fork_exec_ls.c

```
/*
fork_exec_ls.c
Parent forks, child executes "ls".
Shows behavior with and without wait().
*/

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
  pid_t pid = fork();

  if(pid == 0) {
    printf("Child: Executing 'ls'\n");
    execlp("ls", "ls", NULL);
    printf("EXEC FAILED\n");
  }
  else {
    // Toggle wait() for exam demonstration:
    printf("Parent: Waiting for child...\n");
```

```
    wait(NULL);
    printf("Parent: Child finished.\n");
  }

  return 0;
}
```

To observe behavior *without wait()* → comment out wait.

⭐ **PROGRAM 3 — Parent creates 4 children running different commands**

File: fork_4_commands.c

```
/*
fork_4_commands.c
Parent creates 4 children. Each child executes:
 C1 -> ls
 C2 -> ls -l
 C3 -> pwd
 C4 -> date
*/

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
  char *cmds[][3] = {
    {"ls", NULL, NULL},
    {"ls", "-l", NULL},
    {"pwd", NULL, NULL},
    {"date", NULL, NULL}
  };
```

```
    for(int i=0;i<4;i++){
        pid_t pid = fork();
        if(pid == 0){
            printf("Child %d executing command...\n", i+1);
            execlp(cmds[i][0], cmds[i][0], cmds[i][1], cmds[i][2], NULL);
            printf("Exec failed!\n");
            return 1;
        }
    }

    for(int i=0;i<4;i++)
        wait(NULL);

    printf("Parent: All children finished.\n");
    return 0;
}
```

⭐ **PROGRAM 4 — fork() child → Fibonacci up to n (user input)**

File: fork_fibonacci.c

```
/*
fork_fibonacci.c
Parent forks child; child prints Fibonacci series up to n (from user input).
*/

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int n;
    printf("Enter n for Fibonacci: ");
```

```
    scanf("%d", &n);

    if(n <= 0){
        printf("Invalid n.\n");
        return 1;
    }

    pid_t pid = fork();

    if(pid == 0){
        printf("Child: Fibonacci series: ");
        int a=1,b=1;
        if(n>=1) printf("%d ",a);
        if(n>=2) printf("%d ",b);

        for(int i=3;i<=n;i++){
            int c=a+b;
            printf("%d ",c);
            a=b; b=c;
        }
        printf("\n");
    }
    else {
        wait(NULL);
        printf("Parent: Child completed Fibonacci.\n");
    }

    return 0;
}
```

⭐ **PROGRAM 5 — Parent odd numbers, Child even numbers (with user input)**

File: fork_odd_even.c

```
/*
fork_odd_even.c
Parent prints odd numbers, child prints even numbers up to n.
*/

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(){
   int n;
   printf("Enter n: ");
   scanf("%d",&n);

   pid_t pid = fork();

   if(pid == 0){
      printf("Child (Even numbers): ");
      for(int i=0;i<=n;i+=2) printf("%d ",i);
      printf("\n");
   }
   else {
      printf("Parent (Odd numbers): ");
      for(int i=1;i<=n;i+=2) printf("%d ",i);
      printf("\n");
      wait(NULL);
   }

   return 0;
}
```

⭐ **PROGRAM 6 — Same (odd/even) but using exec()**

File: odd_child.c (for child via exec)

```
// odd_child.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){
   int n = atoi(argv[1]);
   printf("Child (Even numbers via exec): ");
   for(int i=0;i<=n;i+=2) printf("%d ", i);
   printf("\n");
   return 0;
}
```

File: fork_exec_odd_even.c

```
/*
Parent does odd numbers.
Child execs odd_child.c to do even numbers.
Compile odd_child.c -> ./child_even
*/

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(){
   int n;
   printf("Enter n: ");
   scanf("%d",&n);

   pid_t pid=fork();
```

```
  if(pid==0){
    execl("./child_even","child_even",
        (char[20]){0}, NULL); // Suppose compiled to child_even
  }
  else{
    printf("Parent (Odd numbers): ");
    for(int i=1;i<=n;i+=2) printf("%d ", i);
    printf("\n");
    wait(NULL);
  }
  return 0;
}
```

⭐ **PROGRAM 7 — File Operations: Child 1 counts lines, Child 2 counts characters**

File: fork_file_line_char.c

```
/*
fork_file_line_char.c
Child 1 -> count lines
Child 2 -> count characters
*/

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc,char *argv[]){
  if(argc != 2){
    printf("Usage: %s <filename>\n", argv[0]);
    return 1;
  }
```

```
    char *file = argv[1];

    pid_t c1 = fork();
    if(c1 == 0){
        FILE *fp = fopen(file, "r");
        int lines=0; char ch;
        while((ch = fgetc(fp)) != EOF)
            if(ch=='\n') lines++;
        printf("Child 1: Total lines = %d\n", lines);
        fclose(fp);
        return 0;
    }

    pid_t c2 = fork();
    if(c2 == 0){
        FILE *fp = fopen(file, "r");
        int chars=0;
        while(fgetc(fp) != EOF) chars++;
        printf("Child 2: Total characters = %d\n", chars);
        fclose(fp);
        return 0;
    }

    wait(NULL);
    wait(NULL);
    printf("Parent: Both children completed.\n");
    return 0;
}
```

⭐ **PROGRAM 8 — Lowercase conversion & counting 'a'**

File: fork_file_lower_a.c

```
/*
fork_file_lower_a.c
Child 1 -> convert to uppercase
Child 2 -> count 'a'
*/

#include <stdio.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/wait.h>

int main(int argc,char *argv[]){
  if(argc!=2){
    printf("Usage: %s <file>\n", argv[0]);
    return 1;
  }

  char *file = argv[1];

  pid_t c1 = fork();
  if(c1 == 0){
    FILE *fp = fopen(file,"r+");
    char ch; long pos;
    while((ch=fgetc(fp))!=EOF){
      pos=ftell(fp);
      fseek(fp,pos-1,SEEK_SET);
      fputc(toupper(ch),fp);
    }
    printf("Child 1: Converted to uppercase.\n");
    fclose(fp);
```

```
    return 0;
  }

  pid_t c2 = fork();
  if(c2 == 0){
    FILE *fp = fopen(file,"r");
    int count=0, ch;
    while((ch=fgetc(fp))!=EOF)
      if(ch=='a' || ch=='A') count++;
    printf("Child 2: Count of 'a' = %d\n", count);
    fclose(fp);
    return 0;
  }

  wait(NULL);
  wait(NULL);
  printf("Parent: Finished operations.\n");
  return 0;
}
```

⭐ **PROGRAM 9 — Count words + invert case in same file**

File: fork_file_words_invert.c

```
/*
fork_file_words_invert.c
Child 1: count words
Child 2: invert case of every alphabet
*/

#include <stdio.h>
#include <unistd.h>
#include <ctype.h>
```

```
#include <sys/wait.h>

int main(int argc,char *argv[]){
  if(argc != 2){
    printf("Usage: %s <file>\n", argv[0]);
    return 1;
  }

  char *file = argv[1];

  pid_t c1 = fork();
  if(c1 == 0){
    FILE *fp = fopen(file,"r");
    int words=0, in_word=0;
    char ch;
    while((ch=fgetc(fp))!=EOF){
      if(isspace(ch)) in_word=0;
      else if(!in_word){
        in_word=1;
        words++;
      }
    }
    printf("Child 1: Total words = %d\n", words);
    fclose(fp);
    return 0;
  }

  pid_t c2 = fork();
  if(c2 == 0){
    FILE *fp = fopen(file,"r+");
    char ch; long pos;
```

```
        while((ch=fgetc(fp))!=EOF){
            pos = ftell(fp);
            fseek(fp, pos-1, SEEK_SET);

            if(isupper(ch)) fputc(tolower(ch),fp);
            else if(islower(ch)) fputc(toupper(ch),fp);
            else fputc(ch,fp);
        }
        printf("Child 2: Case inverted.\n");
        fclose(fp);
        return 0;
    }

    wait(NULL);
    wait(NULL);
    printf("Parent: All file operations complete.\n");

    return 0;
}
```