INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Stamp / Signature of the Invigilator

MID SEMESTER EXAMINATION	SPRING SEMESTER

Roll Number										Section		Name	
Subject Number	C	S	3	1	2	0	2	Subject Name					*Operating Systems*
Department / Center of the student												Additional Sheets	

Important Instructions and Guidelines for Students

1. You must occupy your seat as per the Examination Schedule/Sitting Plan.
2. Do not keep mobile phones or any similar electronic gadgets with you even in the switched off mode.
3. Loose papers, class notes, books or any such materials must not be in your possession, even if they are irrelevant to the subject you are taking examination.
4. Data book, codes, graph papers, relevant standard tables/charts or any other materials are allowed only when instructed by the paper-setter.
5. Use of instrument box, pencil box and non-programmable calculator is allowed during the examination. However, exchange of these items of any other papers (including question papers) is not permitted.
6. Write on both sides of the answer script and do not tear off any page. **Use last page(s) of the answer script for rough work.** Report to the Invigilator if the answer script has torn or distorted page(s).
7. It is your responsibility to ensure that you have signed the Attendance Sheet. Keep your Admit Card/Identity Card on the desk for checking by the invigilator.
8. You may leave the examination hall for wash room or for drinking water for a very short period. Record your absence from the Examination Hall in the register provided. Smoking and the consumption of any kind of beverages are strictly prohibited inside the Examination Hall.
9. Do not leave the Examination Hall without submitting your answer script to the invigilator. **In any case, you are not allowed to take away the answer script with you.** After the completion of the examination, do not leave the seat until the invigilators collect all the answer scripts.
10. During the examination, either inside or outside the Examination Hall, gathering information from any kind of sources or exchanging Information with others or any such attempt will be treated as '**unfair means**'. Do not adopt unfair means and do not indulge in unseemly behavior.

Violation of any of the above instructions may lead to severe punishment.

Signature of the Student

To be filled in by the examiner

Question Number	1	2	3	4	5	6	7	8	9	10	Total
Marks obtained											

Marks Obtained (in words)	Signature of the Examiner	Signature of the Scrutineer

CS31202 Operating Systems

Mid-Semester Examination

23-Feb-2026, 03:00pm–05:00pm

Maximum Marks: 80

Instructions to students

- Please write in the spaces provided in the question paper itself. Be brief and precise.
- For rough work and leftover answers (if needed), you can use the extra blank pages provided at the end. You are given sufficient extra space, so try to avoid taking supplementary sheets from the invigilators.
- If you use extra page(s) for any leftover answer, please give a pointer (number of the extra page) from the given space where you begin the answer. Without this, we will not look into the extra pages.
- Do not write anything on this page. Questions start from the next page (Page 3).

1. [*Scheduling on multi-core platforms*]

Foonix Operating System (FooOS) adopts first-come-first-served (FCFS) process scheduling, that is, the ready queues are FIFO queues. Four processes are available at time $t = 0$, and line up in the ready queue as P1, P2, P3, P4 (P1 at the front, P4 at the back). Each process needs three CPU bursts and two IO bursts. The burst times are given in the table below. The time unit is millisecond (ms) throughout this exercise. Upon the completion of an IO burst, a process joins the back of its ready queue. In case of ties (in IO completion times), lower-numbered processes have higher priorities.

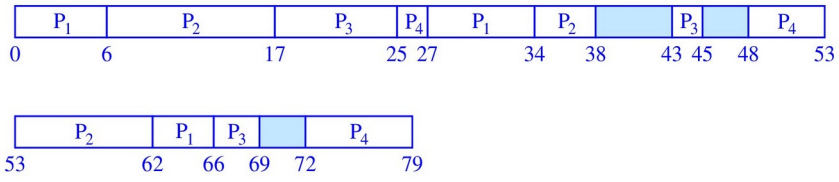
Process	CPU Burst 1	IO Burst 1	CPU Burst 2	IO Burst 2	CPU Burst 3
P1	6	16	7	20	4
P2	11	15	4	15	9
P3	8	18	2	14	3
P4	2	21	5	19	7

(a) FooOS is installed on a Bar Lab Machine which has only one processor (core). Determine how the processes are scheduled, and draw the Gantt chart for this schedule. **[7]**

First, show your calculations in the following format.

Process	CPU Burst 1 duration	First IO completion time, and the ready queue at that time	CPU Burst 2 duration	Second IO completion time, and the ready queue at that time	CPU Burst 3 duration
P1	0 – 6	6 + 16 = 22 P4, P1	27 – 34	34 + 20 = 54 P1	62 – 66
P2	6 – 17	17 + 15 = 32 P2	34 – 38	38 + 15 = 53 P2	53 – 62
P3	17 – 25	25 + 18 = 43 P3	43 – 45	45 + 14 = 59 P1, P3	66 – 69
P4	25 – 27	27 + 21 = 48 P4	48 – 53	53 + 19 = 72 P4	72 – 79

Then, draw the Gantt chart for this schedule. In the Gantt chart, shade the CPU-idle durations.



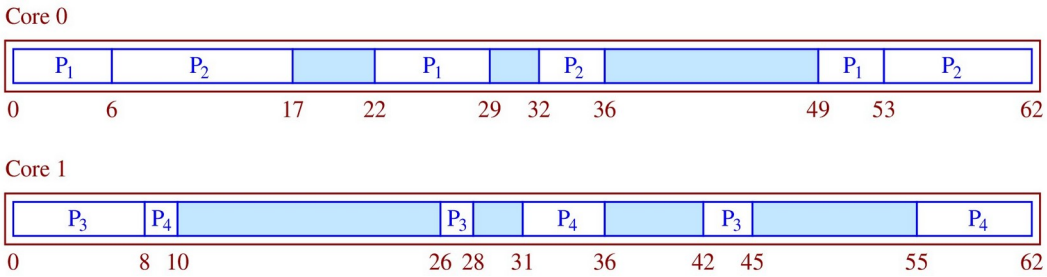
(b) A machine in Car Lab also runs on FooOS. This machine has two independent processing cores (Core 0 and Core 1) that run in parallel. FooOS uses two ready queues, one for each core. A process once assigned to a core will run on that core until completion. Consider the same processes as in Part (a) (the burst-time table is reproduced below for your convenience). Suppose that FooOS maps P1, P2 to Core 0, and P3, P4 to Core 1. At $t = 0$, P1 is at the front of the ready queue for Core 0, and P3 is at the front of the ready queue for Core 1. Compute the schedule in this case. **[7]**

Process	CPU Burst 1	IO Burst 1	CPU Burst 2	IO Burst 2	CPU Burst 3
P1	6	16	7	20	4
P2	11	15	4	15	9
P3	8	18	2	14	3
P4	2	21	5	19	7

Show your calculations in the following table.

Process	CPU Burst 1 duration	First IO completion time, and the ready queue at that time	CPU Burst 2 duration	Second IO completion time, and the ready queue at that time	CPU Burst 3 duration
Core 0					
P1	0 – 6	6 + 16 = 22 P1	22 – 29	29 + 20 = 49 P1	49 – 53
P2	6 – 17	17 + 15 = 32 P2	32 – 36	36 + 15 = 51 P2	53 – 62
Core 1					
P3	0 – 8	8 + 18 = 26 P3	26 – 28	28 + 14 = 42 P3	42 – 45
P4	8 – 10	10 + 21 = 31 P4	31 – 36	36 + 19 = 55 P4	55 – 62

Then, draw the Gantt chart for this schedule. In the Gantt chart, shade the CPU-idle durations for each core.

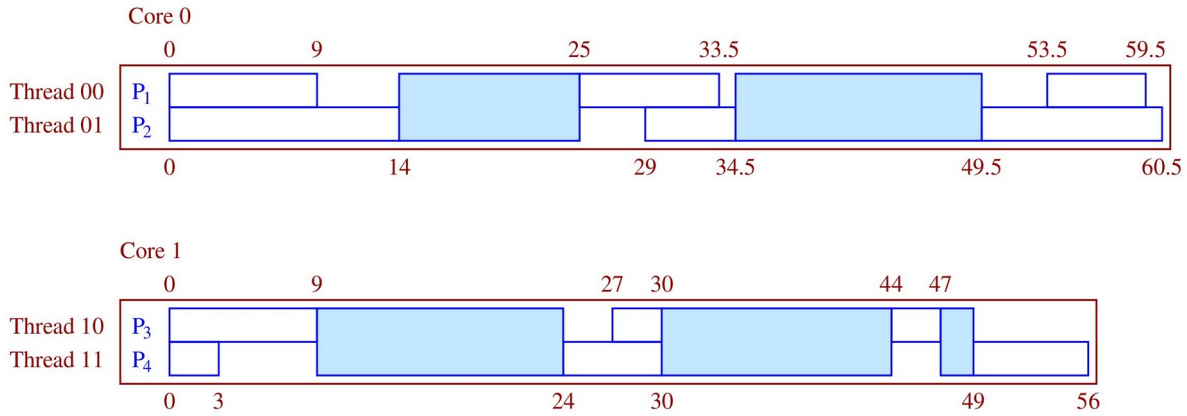


(c) A machine in Dar Lab also uses FooOS, and is again equipped with two processors. But each of these processors supports chip multithreading (also called hyperthreading), with two hyperthreaded cores per processor. The two (hardware) threads on the same processor do not run in absolute parallelism. The CPU switches between the two threads in order to utilize long-latency events (like memory stall) of the processes running on them. Because the memory-stall and the computation times of the two threads are not perfectly complementarily aligned, and also because of overheads in repopulating the instruction cache, the burst time increases by 50% when both the threads are running together. The two threads need not run together for their entire CPU bursts. The 50% increase applies only to the overlapped duration. For the non-overlapped parts, the original burst times are taken. As an example, suppose that two processes complete their CPU bursts of durations 8 ms and 11 ms with a partial overlap of 5 ms. These times correspond to the *absolute times* of the processes when they exclusively use the processor. Then, the processes will actually take $5 \times 1.5 + (8 - 5) = 10.5$ ms and $5 \times 1.5 + (11 - 5) = 13.5$ ms in order to complete their respective bursts.

Consider the four processes as in the previous two parts. The given CPU burst times are their absolute running times when the processes use a processor exclusively. Overlapping computations on the hyperthreaded cores experience 50% increase in actual running times on the overlapped parts only, as explained above. Let us number the hyperthreaded cores of Processor/Core 0 as 00 and 01, and those of Processor/Core 1 as 10 and 11. The OS maintains four ready queues (one for each hyperthreaded core), and schedules P1, P2, P3, P4 to the hyperthreaded cores 00, 01, 10, and 11, respectively. Once again, there is no migration of a process from its assigned hyperthreaded core. Since we now have only one process for each hyperthreaded core, a process completing an IO burst is immediately ready to run on its assigned hyperthreaded core.

Draw the Gantt chart for this schedule on the Dar-Lab machine. Shade the CPU-idle durations. A CPU (processor) is now called idle when neither of its two hyperthreaded cores is running any process. The burst-time table with absolute CPU burst times is given again below. **[7]**

Process	CPU Burst 1	IO Burst 1	CPU Burst 2	IO Burst 2	CPU Burst 3
P1	6	16	7	20	4
P2	11	15	4	15	9
P3	8	18	2	14	3
P4	2	21	5	19	7



2. [*Synchronization primitives*]

You want some public service. You are given the next available token. Tokens are served in the ascending order. You wait until your turn comes, that is, until you own the smallest token yet to be served. This sequencing of the requests is as fair as it could be. Now, suppose that the service-seekers are processes, collecting the token and waiting for the turn of the token constitute the entry section, and the service is accessing the critical section. We plan to apply the idea of tokens, to solve the critical-section problem. This token-based solution clearly supports progress and bounded wait. Only mutual exclusion needs to be enforced.

Let n processes $P_0, P_1, \ldots, P_{n-1}$ plan to access their critical sections (each possibly multiple times, perhaps in an infinite loop). The protocol described below uses a shared int array `token[]` of size n (initialized to 0). If P_i is not interested in entering its critical section, we have `token[`i`]` = 0. Otherwise, `token[`i`]` stores the token (a positive integer) that P_i gets in the entry section by computing the maximum in the `token[]` array. P_i makes a busy wait until its token becomes the minimum non-zero token. The code of the entry section is at lines 1–12, and the exit-section code is at line 13. All variables used in the code (other than the `token[]` array) are private to P_i.

Code for Process P_i

```
    while (1) {
        /* Select next token */
1       max = 0;
2       for ( j = 0; j < n; ++j ) {
3           v = token[j];
4           if (max < v) max = v;
        }
5       max = max + 1;
6       token[i] = max;

        /* Wait until token[i] is the minimum non-zero token */
7       do {
8           myturn = 1;
9           for ( j = 0; j < n; ++j ) {
10              if ( (j != i) && (token[j] != 0) && (token[j] < max) ) {  /* smaller non-zero token found */
11                  myturn = 0; break;
                }
            }
12      } while ( !myturn );

        CRITICAL SECTION

13      token[i] = 0;  /* the computed token will no longer be needed */

        REMAINDER SECTION
    }
```

In all the parts of this exercise, we make the following two assumptions.

- The OS uses round robin scheduling, so each process periodically gets chances to run.
- Neither the hardware nor the compiler swaps instructions in the codes.

(a) As per this protocol, two (or more) processes (or the same process) may get the same token at different instants. For example, if all the processes are in their remainder sections, the `token[]` array is filled with 0's alone, so the next process to enter its critical section computes `max` = 0, and gets the token 1. Demonstrate how two (or more) processes trying to access their critical sections at the same time (concurrently or in parallel) may compute the same token. **[3]**

Suppose that two processes P_i and P_j want to enter their critical sections at the same time (concurrently or in parallel). They compute the same value of `max`. This may happen because each of the processes reads the old value (0) as the token of the other process before the other process writes its token in the shared array. Clearly, this situation can be generalized to any number of processes, all of which are in their max-computation loops at the same time.

(b) To address the problem of Part (a), assign a priority to the processes (with the same token). If *token*[*i*] and *token*[*j*] are equal, and $j < i$, then P_j is given priority over P_i. In view of this, we change the condition of line 10 as follows.

```
( (j != i) && (token[j] != 0) && ( (token[j] < max) || ((token[j] == max) && (j < i)) ) )
```

Explain how, even with this modification, the protocol does not guarantee mutual exclusion. **[5]**

Again suppose that two processes P_i and P_j want to enter their critical sections at the same time (concurrently or in parallel). They compute the same value of `max`, and will therefore end up with the same token. Let $i < j$. Before P_i gets a chance to update `token[i]`, it is preempted. P_j continues to run, and eventually its token becomes the minimum in the `token[]` array. Since P_i has still not updated `token[i]`, P_j sees `token[i] == 0` (Line 10). Failing to find any process which should be served earlier, P_j enters its critical section. While P_j is still in its critical section, P_i is rescheduled to run. It writes its token, and moves forward. It sees `token[j] == token[i]`. But since $j > i$, P_i understands that P_j is of priority lower than P_i. Therefore P_i too enters its critical section.

(c) Propose a hardware-based solution that ensures that every process gets a unique token. The solution keeps a global count `last_token` (initialized to 0), and instead of computing the maximum current token, a requesting process will get the token `last_token` + 1, and at the same time, `last_token` will be updated to this incremented value. Under this implementation, no token is repeated (even at separate times), but this does not affect the working of the protocol. Write a function `gettoken()` that uses the compare-and-swap primitive. Justify whether this solution guarantees mutual exclusion (by eliminating the problem raised in Part (b))? **[4]**

```
int gettoken ( )
{

    /* This function uses atomic increment */

    int temp, retval;

    do {
       temp = last_token;
       retval = temp + 1;
    } while (compare_and_swap(&last_token, temp, retval) != temp);

    return retval;

}
```

Is the problem raised in Part (b) solved? Justify. **[1]**

No. A process now increments `last_token` atomically, but may still be preempted before `token[i]` is updated.

(d) In this part, suppose that you do not have an option to use any hardware-based solution, that is, you want to patch the given code (the code based on computing the maximum of the `token`[] array, not the code using `last_token`). Of course, two (or more) processes may now compute the same token at the same time, so we use the revised priority model introduced in Part (b). Introduce a shared boolean array `computing`[]. Each P_i now keeps `computing`[i] busy so long as it is computing its token. Rewrite the code for Process P_i (as on page 6, but with the modification of Part (b) in effect) using the shared array `computing`[] so that mutual exclusion is guaranteed. **[6]**

Revised code for P_i

```
while (1) {
    /* ENTRY SECTION */
    computing[i] = 1;   /* Going to compute my token */
    for ( j = 0; j < n; ++j ) {
        v = token[j];
        if (max < v) max = v;
    }
    max = max + 1;
    token[i] = max;
    computing[i] = 0;   /* Token computation done */

    /* Wait until token[i] is the minimum non-zero token */
    do {
        myturn = 1;
        for ( j = 0; j < n; ++j ) {
            while (computing[j] == 1) ; /* wait until token computation finishes */
            if ( (j != i) && (token[j] != 0) && ((token[j] < max) || ((token[j] == max) && (j < i))) ) {
                myturn = 0; break;
            }
        }
    } while ( !myturn );

    CRITICAL SECTION

    token[i] = 0;    /* EXIT SECTION */

    REMAINDER SECTION
}
```

Note: This algorithm is known as *Lamport's Bakery Algorithm*. Here, no process is allowed to enter its critical section if some token computation is in progress. Because of round robin scheduling, each token computation will eventually finish. So the wait at this stage is bounded. The only problem with this protocol arises when there is an overflow during the increment of `max`.

3. [*Synchronization example*]

Consider a river with a left bank and a right bank. Missionaries and Cannibals continuously arrive at the left bank, and wish to cross to the right bank. A small boat with exactly two seats is available to transport passengers across the river. The boat is initially located at the left bank. For each trip, it carries exactly two passengers to the right bank, allows both passengers to disembark there, and then returns empty to the left bank to serve the next crossing. The only permitted boarding combinations are (Missionary, Missionary) and (Missionary, Cannibal). Only when no Missionary is available on the left bank, the combination (Cannibal, Cannibal) is allowed.

Missionaries and Cannibals arrive at the left bank continuously. Upon arrival, each Missionary executes the procedure `Missionary()`, and each Cannibal executes the procedure `Cannibal()`. The boat, which repeatedly transports passengers across the river, executes the procedure `Boat()` in an infinite loop. Each passenger will cross the river only once. Inside `Missionary()` and `Cannibal()`, the first arriving passenger increments the appropriate waiting count, and blocks itself. When a second passenger arrives, she checks whether a compatible passenger is waiting. If so, she wakes up the companion passenger, and notifies the boat that two valid passengers are ready to cross. The two passengers then board the boat, after which the boat begins its journey. Both passengers remain blocked while the boat is in transit. Upon reaching the right bank, the boat signals the passengers, one by one, that the journey has completed. The passengers then disembark from the boat.

Using semaphores, implement the synchronization logic inside `Missionary()`, `Cannibal()`, and `Boat()` such that (a) only valid passenger pairs are allowed to board the boat, (b) the boat departs only after exactly two valid passengers have boarded, and (c) both riding passengers are correctly signaled, and disembark upon reaching the right bank.

Consider the incomplete solutions for `Missionary()`, `Cannibal()`, and `Boat()`, given below. The solution involves six semaphores and two shared variables. The shared variables `mWaiting` and `cWaiting` respectively denote the number of Missionaries and the number of Cannibals waiting on the left bank. **[20]**

Shared Variables (integers)

```
mWaiting = 0 ;        // Number of missionaries waiting on the left bank
cWaiting = 0 ;        // Number of cannibals waiting on the left bank
```

Semaphores

```
mutex = 1 ;           // Ensures mutual exclusion while accessing mWaiting and/or cWaiting
mSem = 0 ;            // Blocks a missionary until a valid companion is available
cSem = 0 ;            // Blocks a cannibal until a valid companion is available
boatReady = 0 ;       // Boat waits on this until two valid passengers are ready
boardDone = 0 ;       // Boat waits until both passengers complete boarding

tripDone = ____0____  // Passengers wait on this until the boat completes the trip
```

Procedures

```
Missionary()
{
    wait(mutex);                                  // Enter critical section to update shared variables
    mWaiting++;                                       // Increment count of waiting missionaries

    if ( ____mWaiting >= 2____ ) {                // Two missionaries are available

        ____signal(mSem)____ ;                    // Wake up first waiting missionary

        ____signal(mSem)____ ;                    // Wake up second waiting missionary
        mWaiting -= 2;                                // Reserve two missionaries for next ride

        ____signal(boatReady)____ ;               // Notify the boat that a valid pair is ready
    }
    else if (cWaiting >= 1) {                         // One cannibal is waiting
```

```
        signal(mSem);                                   // Wake up the missionary

                    signal(cSem)
        ____________________________________________ ;  // Wake up a waiting cannibal
        mWaiting--;                                     // Reserve one missionary for next ride
        cWaiting--;                                     // Reserve one cannibal for next ride

                  signal(boatReady)
        ____________________________________________ ;  // Notify the boat that a valid pair is ready
    }
    signal(mutex);                                      // Exit critical section

                     wait(mSem)
    ____________________________________________ ;      // Wait until missionary has been selected
                                                        // as part of a valid pair
    board();                                            // Board the boat

                    signal(boardDone)
    ____________________________________________ ;      // Signal completion of boarding of missionary
    wait(tripDone);                                     // Wait until the boat reaches the right bank
    deboard();                                          // Disembark from the boat
}

Cannibal ()
{
    wait(mutex);                                // Enter critical section to update shared variables
    cWaiting++;                                         // Increment count of waiting cannibals

    if ( ____(mWaiting == 0) && (cWaiting >= 2)____ ) { // No missionaries are waiting
        signal(cSem);                                   // Wake up first waiting cannibal
        signal(cSem);                                   // Wake up second waiting cannibal

                   cWaiting -= 2
        ____________________________________________ ;  // Reserve two cannibals for next ride
        signal(boatReady);                              // Notify the boat that a valid pair is ready
    }
    else if (mWaiting >= 1) {                           // (at least) one missionary is waiting

                    signal(mSem)
        ____________________________________________ ;  // Wake up a missionary


                    signal(cSem)
        ____________________________________________ ;  // Wake up the cannibal
        mWaiting--;                                     // Reserve one missionary for next ride
        cWaiting--;                                     // Reserve one cannibal for next ride

                  signal(boatReady)
        ____________________________________________ ;  // Notify the boat that a valid pair is ready
    }
    signal(mutex);                                      // Exit critical section

                     wait(cSem)
    ____________________________________________ ;      // Wait until cannibal has been selected as
                                                        // part of a valid pair
    board();                                            // Board the boat

                    signal(boardDone)
    ____________________________________________ ;      // Signal completion of boarding of cannibal


                    wait(tripDone)
    ____________________________________________ ;      // Wait until the boat reaches the right bank
    deboard();                                          // Disembark from the boat
}

Boat ()
{
    while (true) {

                   wait(boatReady)
        ____________________________________________ ;  // Wait until two valid passengers are ready

        wait(boardDone);                                // Wait for the first passenger to board
        wait(boardDone);                                // Wait for the second passenger to board
        crossRiver();                                   // Transport passengers to the right bank

                  signal(tripDone)
        ____________________________________________ ;  // Signal first passenger to disembark


                  signal(tripDone)
        ____________________________________________ ;  // Signal second passenger to disembark
        returnBoat();                                   // Return the boat empty to the left bank
    }
}
```

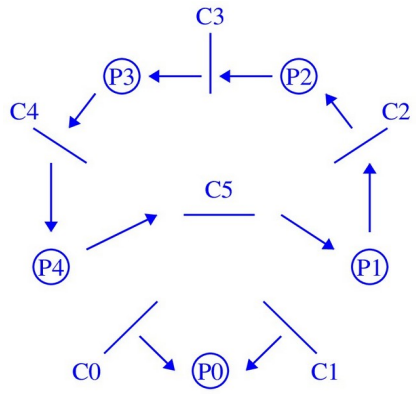
4. [*Deadlock*]

(a) Consider the dining philosopher's problem with five philosophers, and with the same setting as in the original problem statement. We add a single extra chopstick (or fork if they are eating spaghetti) at the center of the table. Any philosopher can get it in order to complete its requirement for eating (having two chopsticks). We call the three chopsticks for each philosopher left (L), right (R), and center (C). In view of this revised situation, the code for grabbing two chopsticks is rewritten for each philosopher as follows. The function `trygrab()` is non-blocking. If the requested chopstick is available, it is taken, and the function returns `TRUE`. If the requested chopstick is not available, the function returns `FALSE` without waiting. The `wait()` call however blocks until the specified chopstick is available.

Each chopstick is a semaphore initialized to 1. No process can wait on two different unavailable semaphores together (the first wait blocks a process, and prevents it from making a second request). So a philosopher has to sequentially choose the chopsticks (S and T) to wait upon. Like `wait(S)`, the call `trygrab(S)` is atomic but returns immediately without putting the requesting process in the waiting queue associated with the semaphore `S`, in case `S` is unavailable.

```
grab_chopsticks ( i ) {     // all variables are private to Philosopher i, all semaphores are shared
   ngrabbed = 0;                                                  // Number of chopsticks grabbed so far
   if (trygrab(L)) { ++ngrabbed; gotL = TRUE; } else { gotL = FALSE; }   // first try left chopstick
   if (trygrab(R)) { ++ngrabbed; gotR = TRUE; } else { gotR = FALSE; }   // then try right chopstick
   if (ngrabbed < 2) {        // either left chopstick or right chopstick or both is/are unavailable
       if (trygrab(C)) { ++ngrabbed; gotC = TRUE; } else { gotC = FALSE; }   // try center chopstick
   } else { gotC = FALSE; }
   if (ngrabbed == 2) { return (gotL,gotR,gotC); }       // which chopsticks to release after eating
   choose any one of the chopsticks not grabbed, call it S;     // S is L, R, or C with gotS = FALSE
   wait(S); ++ngrabbed; gotS = TRUE;          // wait until the chopstick S is available and grabbed
   if (ngrabbed == 2) { return (gotL,gotR,gotC); }
   choose any one of the chopsticks not grabbed, call it T;                  // gotT is still FALSE
   wait(T); gotT = TRUE;               // wait until the second chopstick T is available and grabbed
   return (gotL,gotR,gotC);
}
```

Prove/Disprove: This algorithm may lead to a deadlock. **[7]**



Deadlock is possible.
Suppose that P0 grabs C0 and C1, and starts eating.
Now, the problem reduces to one with four philosophers.
The central chopstick plays the role of the one between P1 and P4.
P2 grabs C2, P3 grabs C3, P4 grabs C4 (left available for each)
P1 grabs C5 (both left and right unavailable).
For each of P1, P2, P3, P4, the second choice is no longer available.
P1 decides to wait on C2, P2 on C3, P3 on C4, and P4 on C5.
This is a case of circular wait, and leads to a deadlock.

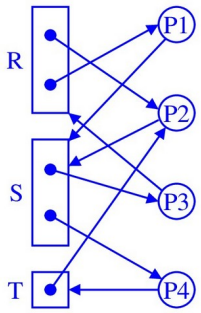
(b) There are three resources R, S, and T. Resources R and S have two instances each, whereas resource T has only one instance. Four processes P1, P2, P3, and P4 make the following requests.

P1: request(R); request(S);
P2: request(R); request(T); request(S);
P3: request(S); request(R);
P4: request(S); request(T);

For each process, the requests must follow the sequence as specified above. However, the scheduler may interleave the requests from different processes in any arbitrary order (without violating the sequence for each process). Suppose that the scheduler attends the requests in the following sequence.

(R from P2), (S from P3), (R from P1), (T from P2), (S from P4), (T from P4), (S from P1), (R from P3), (S from P2).

Draw the resource allocation graph after this sequence, and argue from that graph that a deadlock has occurred. **[7]**



The resource allocation graph contains the cycle
P1 - S - P4 - T - P2 - S - P3 - R - P1.
All the four processes are waiting to get an additional resource. There is no process to release any resource. So this is a case of deadlock.

(c) There are five resources R1, R2, R3, R4, and R5 (each with one instance only). Three processes P1, P2, and P3 request, (use,) and release these resources in the following sequences.

P1: request(R4), request(R3), release(R3); request(R5); release(R4); request(R1); release(R1); release(R5);
P2: request(R2); request(R5); release(R2); release(R5); request(R3); release(R3); request(R4); release(R4);
P3: request(R2); request(R3); release(R2); request(R1); release(R1); request(R5); release(R5); release(R3);

The requests and releases by each process must follow its sequence given above. However, the scheduler (assumed to be round robin) may interleave the instructions in any arbitrary manner (without disturbing the per-process sequences). Prove that no matter what the scheduler does, deadlock is not possible. (**Hint:** Order the resources.) **[6]**

Order the resources as R2 < R4 < R3 < R5 < R1 or as R4 < R2 < R3 < R5 < R1. From the request sequences, it follows that no process holds a higher-order resource and asks for a lower-order one. Thus the condition of circular wait cannot hold, and no deadlock can ensue.

Extra page for leftover answers (if any) and/or rough work

Extra page for leftover answers (if any) and/or rough work

Extra page for rough work

Extra page for rough work