1. a) False
   b) True
   c) True
   d) True
   e) True
   f) False

Correct Solution

a) False
b) True
c) True
d) False

(I am not too sure. But this may be due to the system setting up limited direct execution (baby proofing the CPU) before resuming the program)

e) False

(I am not too sure. But this may be because the lock spins using CPU cycle until spinlock is available for the process. Using this, the process may be in running state and not the blocked state)

f) False

Notes

- **User Mode**
  - Is restricted
  - Executing code has no ability to *directly* access hardware or reference memory [1]
  - Crashes are always recoverable [1]
  - Is where most of the code on our computer / applications are executed [3]
- **Kernel Mode**
  - Is previleged (non-restricted)
  - Executing code has complete and unrestricted access to the underlying hardware [3]

  - Is generally reserved for the lowest-level, most trusted functions of the operating system [1]
  - Is fatal to crash; it will halt the entire PC (i.e the blue screen of death) [3]

- **Interrupt**
  - Are signals sent to the CPU by external devices, normally I/O devices. [2]
  - Tells the CPU to stop its current activities and execute the appropriate part of the operating system (**Interrupt Handler**). [2]
  - Has three different types [2]

    1) **Hardware Interupts**
       * Are generated by hardware devices to signal that they need some attention from the OS.
       * May be due to receiving some data

         Examples

         · Keystrokes on the keyboard
         · Receiving data on the ethernet card

       * May be due to completing a task which the operating system previous requested

         Examples

         Transfering data between the hard drive and memory

    2) **Software Interupts**

       * Are generated by programs when a system call is requested

    3) **Traps**

       * Are generated by the CPU itself
       * Indicate that some error or condition occured for which assistance from the operating system is needed

- **Content Switch**
  - Is switching from running a user level process to the OS kernel and often to other user processes before the current process is resumed
  - Happens during a timer interrupt or system call
  - Saves the following states for a process during a context switch
    * Stack Pointer

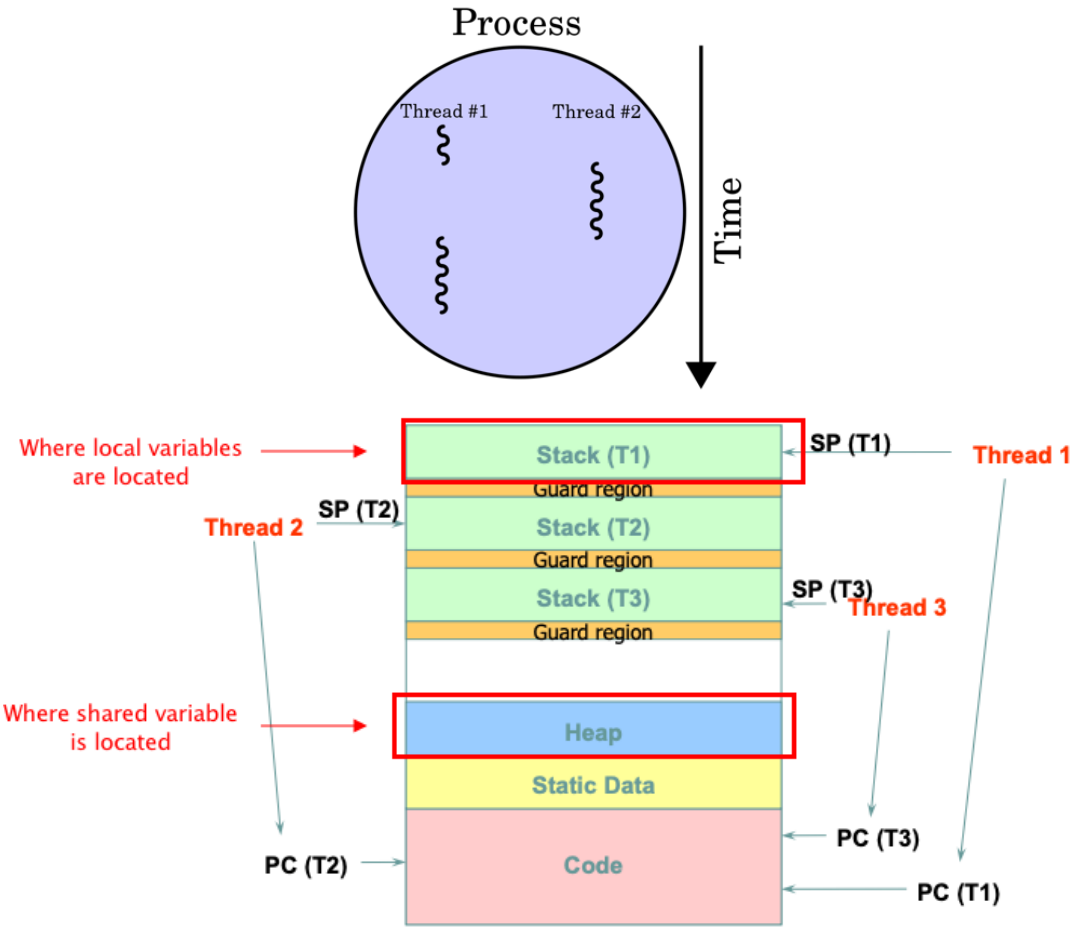
    - Program Counter
    - User Registers
    - Kernel State
  - May hinder performance

- **System Call**

  Example

  - `yield()`
    - Is a system call
    - Causes the calling thread to relinquish the CPU
    - Places the current thread at the end of the run queue
    - Schedules another thread to run

- **Thread**
  - Is a lightweight process that can be managed independently by a schdeduler [4]
  - Improves the application performance using parallelism. (e.g peach)



  - A thread is bound to a single process
  - A process can have multiple threads

  - Has two types
    - **User-level Threads:**
      - Are implemented by users and kernel is not aware of the existence of these threads
      - Are represented by a program counter(PC), stack, registers and a small process control block
      - Are small and much faster than kernel level threads
    - **Kernel-level Threads:**
      - Are handled by the operating system directly
      - Thread management is done by the kernel
      - Are slower than user-level threads

- **Process**
  - Is a program in execution
  - Is named by it's process ID or PID
  - Can be described by the following states at any point in time
    - Address Space
    - CPU Registers
    - Program Counter
    - Stack Pointer
    - I/O Information

    (wait. this is PCB)
  - Exists in one of many different **process states**, including
    1. Running
    2. Ready to Run
    3. Blocked

    - Different events (Getting Scheduled, descheduled, or waiting for I/O) transitions one of these states to the other

- **Signals**
  - Provides a way to communicate with the process
  - Can cause job to stop, continue, or terminate
  - Can be delivered to an application
    - Stops the application from whatever its doing
    - Runs Signal handler (some code in application to handle the signal)
    - When finished, the process resumes previous behavior

- **Spinlock**
  - Is the simplest lock to build
  - Uses a lock variable

  - 0 - (available/unlock/free)
  - 1 - (acquired/locked/held)
- Has two operations
  1. `acquire()`

```
boolean test_and_set(boolean *lock)
{
      boolean old = *lock;
      *lock = True;
      return old;
}
boolean lock;

void acquire(boolean *lock) {
       while(test_and_set(lock));
}
```

  2. `release()`

```
void release(boolean *lock) {
       *lock = false;
}
```

- Allows a single thread to enter critical section at a time
- Spins using CPU cycles until the lock becomes available.
- May spin forever

- **Scheduling policies**
  - Are algorithms for allocating CPU resources to concurrent tasks deployed on (i.e., allocated to) a processor (i.e., computing resource) or a shared pool of processors [5]
  - Are sometimes called **Discipline**
  - Covers the following algorithms in textbook
    - **First In First Out**
    - **Shortest Job First**
    - **Shortest Time-to-completion First**
    - **Round Robin**
      - Runs job for a **time slice** or **quantum**
      - Each job gets equal share of CPU time

- Is clock-driven [6]
- Is starvation-free [7]
- Must have the length of a time slice (**quantum**) as multiple of timer-interrupt period

```
void release(boolean *lock) {
        *lock = false;
}
```

* **Multi-level Feedback Queue**

References

1) Coding Horror, Understanding User and Kernel Mode, link
2) Kansas State University, Basics of How Operating Systems Work, link
3) Kansas State University, Glossary, link
4) Tutorials Point, User-level threads and Kernel-level threads, link
5) Science Direct, Scheduling Policy, link
6) Guru 99: What is CPU Scheduling?, link
7) Wikipedia: Round-robin Scheduling, link

2. a) Is a simple form of lock. It allows a single thread to enter ciritical section at a time. It spins using CPU cycles until lock becomes available. It uses variable `lock` with two values (0 for available, 1 for acquired) with operations `acquire()` and `release()`.

   b) Is a fraction of total turnaround time for a process. Is used by round-robin scheduling algorithm. A process under RR has equal parts of these. Furthermore, scheduling quantum assigned to a process to be multiples of timer-interrupt period

   Correct Solution

   In a preemptive scheduler, this is the time alloted to a process before context-switching to another process

   c) Is the programmatic way in which user requests for previleged service in operating system. On system call, the current process states (program counter, CPU register, kernal state) are saved, enters kernel mode, performs previleged operations such as reading the disk, executes return-from-trap instructions, and returns back to user mode and resumes the program with the attained result

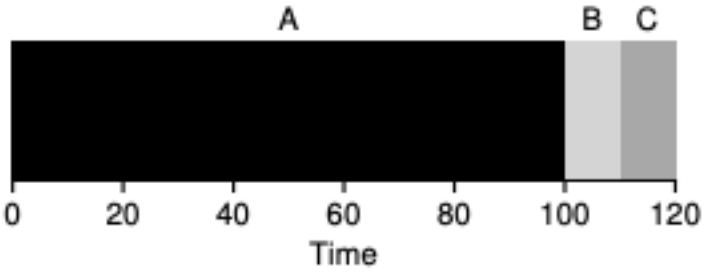
Notes

- **Response Time**
  - **Formula** $T_{response} = T_{firstrun} - T_{arrival}$
  - measures the interactive performance between users and the system
- **Turnaround Time**
  - **Formula** $T_{turnaround} = T_{completion} - T_{arrival}$
  - measures the amount of time taken to complete a process
- **System Call**
  - Is the programmatic way in which a computer program requests a previleged service from the kernel of the operating system
  - i.e. Reading from disk
  - Steps
    1) Setup **trap tables** on boot
    2) Execute system call
    3) Save *Program Counter*, *CPU registers*, *kernal stack* (so process can resume after **return-from-trap** or **context switch**)
    4) Switch from **user mode** to **kernel mode**
    5) Perform previleged operations
    6) Finish and execute **return-from-trap** instruction
    7) Return from **kernel mode** to **user mode** and resume user program

3. a) `(b)` is the only scheduling algorithm that causes starvation

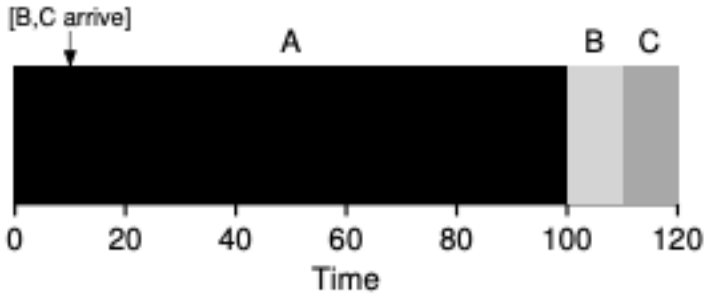
<u>Notes</u>

- **Starvation**
  - Is the problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time [1]
- **Convoy Effect**
  - Is the problem where number of relatively-short potential consumers of a resource get queued behind a heavy weight consumer

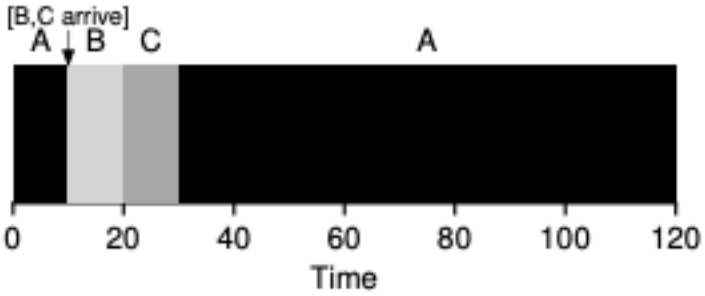


- **First In First Out**
  - Is the most basic scheduling algorithm
  - Is vulnerable to **convoy effect**

  - No **starvation** as long as every process eventually completes
- **Shortest Job First**
  - Improves average **turnaround time** given processes of uneven length
  - Is a general scheduling principle useful in situation where turnaround time per process matters
  - Is vulnerable to **convoy effect**



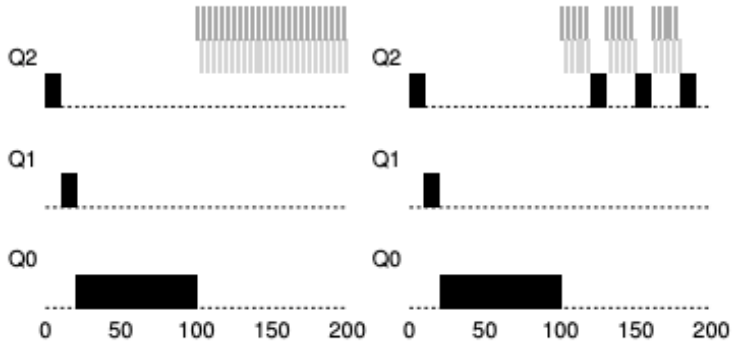
  - Is vulnerable to **starvation**
    - When only short-term jobs come in while a long term job is in queue
- **Shortest Time-to-completion First**
  - Addresses **convoy effect** in **Shortest Job First**
  - Determines which of the remaining+new jobs has least time left, and schedule accordingly at any time



  - Is vulnerable to **starvation**
    - When only short-term jobs come in while a long term job is in queue
- **Round Robin**
  - Has good **response time** but terrible **turnaround time**
  - Runs job for a **time slice** or **quantum**
  - Each job gets equal share of CPU time
  - Is clock-driven [6]
  - Is starvation-free [7]
  - Must have the length of a time slice (**quantum**) as multiple of timer-interrupt period

```
void release(boolean *lock) {
        *lock = false;
}
```

- **Multi-level Feedback Queue**
  - Is the most well known approaches to shceduling
  - Optimizes **turnaround time**, and minimizes **response time**
  - Observees the execution of a job and priortizes accordingly without prior knowledge
  - Rules
    - **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't)
    - **Rule 2:** If Priority(A) = Priority(B). A & B run in round-robin fashion using the time slice (quantum length) of the given queue
    - **Rule 3:** When a job enters the system, it is placed at the highest priority(the top most queue)
    - **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (it moves down on queue)
    - **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

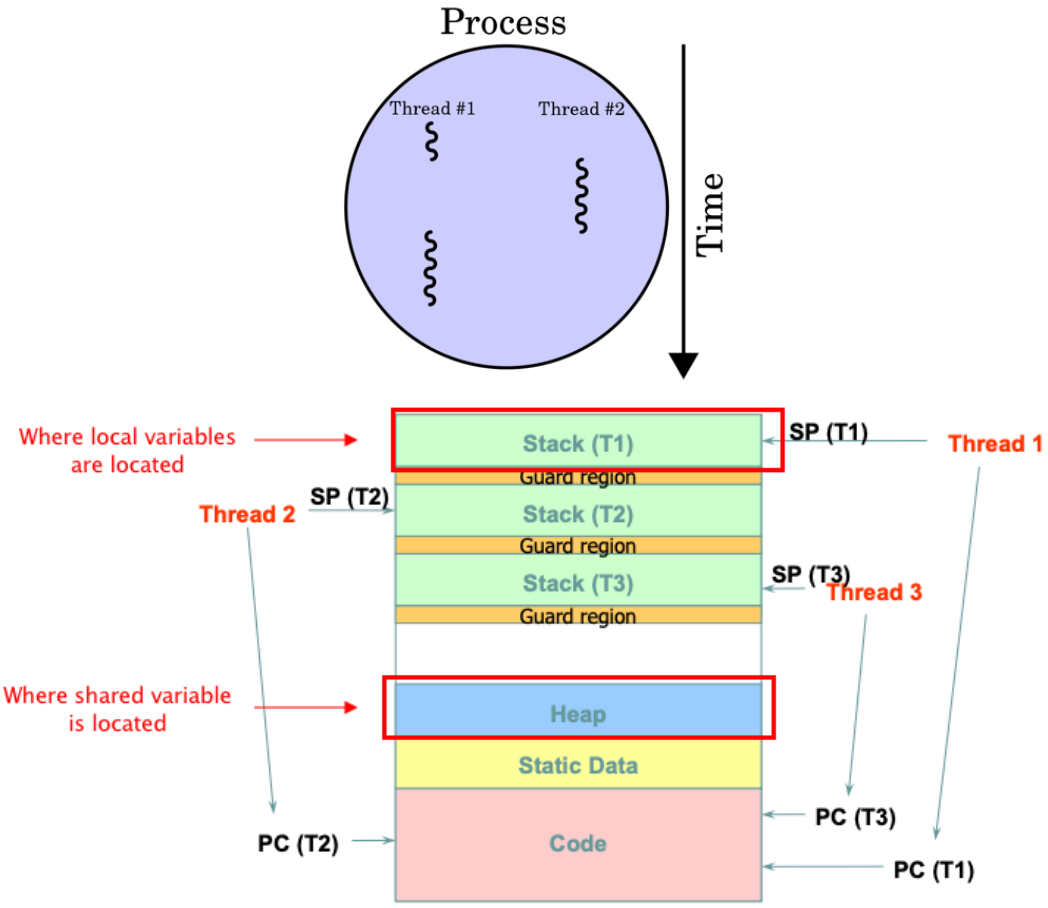


b) Nothing. `pthread_cond_signal` unblocks at least one of the thread that are blocked on the specified condition variable `cond`. Since there are no other threads waiting on `cv1`, there are no threads to awake.

Notes

- **Critical Section**
  - Is a piece of code that accesses a *shared* resource, usually a variable or data structure
- **Thread**
  - Is a lightweight process that can be managed independently by a schdeduler [4]

- Improves the application performance using parallelism. (e.g peach)



  - A thread is bound to a single process
  - A process can have multiple threads
  - Has two types
    - **User-level Threads:**
      - Are implemented by users and kernel is not aware of the existence of these threads
      - Are represented by a program counter(PC), stack, registers and a small process control block
      - Are small and much faster than kernel level threads
    - **Kernel-level Threads:**
      - Are handled by the operating system directly
      - Thread management is done by the kernel
      - Are slower than user-level threads

- **Thread API**
  - `pthread_cond_wait`
    - Puts the calling thread to sleep (a blocked state)
    - Waits for some other thread to signal it

Example

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

Puts calling thread cond to sleep

- `pthread_cond_signal`
  - Is used to unblocks at least one of the threads that are blocked on the specified condition variable cond

Example

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

Wakes a thread that's been put to sleep on cond variable

c) Interrupt is a signals usually sent by external devices, normally I/O devices. It tells the CPU to stop the current process and execute appropriate part of the operating system.

System call is a programmatic way which a computer requests for previleged service from the kernel of the Operating System

Correct Solution

Interrupt is a signals are sent by hardware (keyboardm mouse, etc.), or software (page fault, protection violation, system call)

System call is a strictly subset of software interrupts where a user program asks the OS to perform some previleged functionality on its behalf.

Notes

- **Interrupt**
  - Are signals sent to the CPU by external devices, normally I/O devices.

  - Tells the CPU to stop current process and execute appropriate part of the operating system
- **System Call**
  - Is the programmatic way in which a computer program requests a previleged service from the kernel of the operating system

d) No. I don't agree. I tried to find the reasons regarding inner workings of kernel threads and user threads without success :(

e) There are 3 additional processes in totoal. `fork()` produces a child process of `run_itself` and one `exec()` produces a process for the command `ls -l` and the other `exec()` produces a process for command `cat /proc/cpuinfo`

Correct Solution

Only 1 process, because the exec system call does not spawn any new processes

4. a) I am not too sure on this one. My guess is that account one enters the critical section and while it's in critical section, account 2 switches context and goes to blocked state.

Then, account 1 need to wait for account 2. Thus, deadlock.

Correct Solution

There are two scenarios.

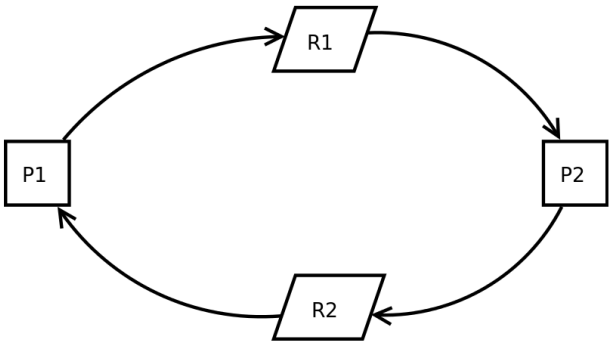
1. When two accounts transfer at the same time.
   - `user_1` executes `transfer_amount(user_2, user_1)`
   - `user_2` executes `transfer_amount(user_1, user_2)`
   - Each will acquire their own lock at `pthread_mutex_lock(a1->lock)`
   - But because they are already locked `pthread_mutex_lock(a2->lock)` can't execute, thus deadlock
2. When performing self transfer
   - `user_1` executes `transfer_amount(user_1, user_1)`
   - `user_1` will acquire own lock at `pthread_mutex_lock(a1->lock)`
   - But because user_1 is already locked, `pthread_mutex_lock(a2->lock)` can't execute, thus deadlock

Notes

- **Semaphores**
  -
- **Livelock**

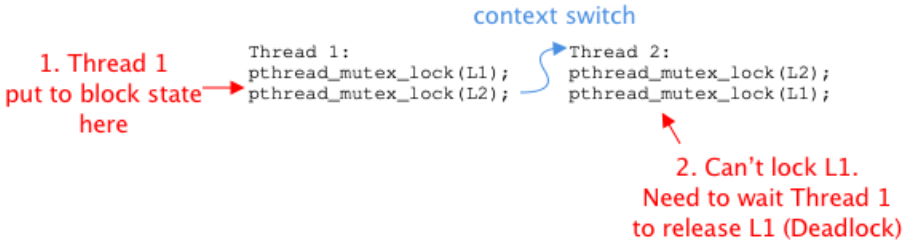
  - Two or more threads reapeatedly attempting this code over and over (e.g acquiring lock), but progress is not being made (e.g acquiring lock)
  - Solution: Add a random delay before trying again (decrease odd of livelock)

- **Deadlock**



  - Is a state in which each member of a group is waiting for another member including itself, to take action (e.g. releasing lock)
  - Conditions for Deadlock (All four must be met)
    * **Mutual Exclusion**
      · Occurs when threads claim exclusive control of resources that they require (e.g. thread grabing a lock)
    * **Hold-and-wait**
      · Occurs when threads hold resources allocated to them (e.g locks that they have already acquired) while waiting for additional resources (e.g. locks that they wish to acquire)
    * **No Preemption**
      · Occurs when resource cannot be forcibly removed from threads that are holding them
    * **Circular Wait**
      · Occurs when there exists a circular chain of threads such that each threads hold one or more resources (e.g. locks) that are being requested by the next thread in the chain.

    Example



  - Preventions

* **Circular Wait**
  - Write code such that circular wait is never induced
  - Is the most practical prevention technique
  - Requires deep understanding of the code base
  - **Total Ordering** (Most starightforward)

    Example

    Given two locks in the system (`L1` and `L2`), always acuiqure `L1` before `L2`

  - **Partial Ordering** (Applied to complex systems)

    Example

    Memory mapping code in Linux (has then different groups).

    (Simple) `i_mutex` before `i_mmap_mutex`

    (More complex) `i_mmap_mutex` before `private_lock` before `swap_lock` before `mapping->tree_lock`

* **Hold-and-wait**
  - Can be avoided by acquiring all locks at once
  - Can be problematic
  - Must know which lock must be held and acquire ahead of time
  - Is likely to decrease concurrency (since all need to be acquired over their needs)

    Example

```
1  pthread_mutex_lock(prevention);   // begin acquisition
2  pthread_mutex_lock(L1);
3  pthread_mutex_lock(L2);
4  ...
5  pthread_mutex_unlock(prevention); // end
```

Lock all in order that doesn't cause deadlock

unlock in the same order

* **No Preemption**
  - Can be avoided by adding code that force unlock if not available

Thread 1

```
1  top:
2    pthread_mutex_lock(L1);
3    if (pthread_mutex_trylock(L2) != 0) {
4      pthread_mutex_unlock(L1);
5      goto top;
6    }
```

1. Check if L2 is locked or not available

2. Unlock L1 forcibly (to avoid deadlock)

Thread 2

```
1  top:
2    pthread_mutex_lock(L2);
3    if (pthread_mutex_trylock(L1) != 0) {
4      pthread_mutex_unlock(L2);
5      goto top;
6    }
```

- `pthread_mutex_trylock` tries to lock the speicied mutex.
- `pthread_mutex_trylock` returns 0 if lock is available
- `pthread_mutex_trylock` returns the following error if occupied

  `EBUSY` - Mutex is already locked

  `EINVAL` - Is not initialized mutex

  `EFAULT` - Is in valid pointer
- May result in **live lock**

* **Mutual Exclusion**
  - Idea: Avoid the mutual exclusion at all
  - Use **lock-free/wait-free** approach: building data structures in a manner that does not require explicit locking using hardware instructions

  Example

```
1  int CompareAndSwap(int *address, int expected, int new) {
2    if (*address == expected) {
3      *address = new;
4      return 1; // success
5    }
6    return 0; // failure
7  }
```

– Avoidance

* **Banker's Algorithm**

b) i. No. In this case, only one account can enter at a time globally.

ii. Only one trasaction can be performed globally. Concurrency is lost.

c)

```
typedef struct acct {
    float balance;
    pthread_mutex *lock;
    pthread_cond_t *cond;
} account;

void transfer_amount (account *a1, account *a2, float amount) {

    // make sure mutex is done in order (see tip on page 392)
    if (a1->lock > a2->lock) {
        pthread_mutex_lock(a1->lock);
        pthread_mutex_lock(a2->lock);
    } else {
        pthread_mutex_lock(a2->lock);
        pthread_mutex_lock(a1->lock);
    }

    a1->balance -= amount;
    a2->balance += amount;

    // put account to sleep as long as balance is negative
    while (a1->balance < 0) {
        Pthread_cond_wait(a1->cond, a1->lock);
    }

    // unlock in total order
    if (a1->lock > a2->lock) {
        pthread_mutex_unlock(a2->lock);
        pthread_mutex_unlock(a1->lock);
    } else {
        pthread_mutex_unlock(a1->lock);
        pthread_mutex_unlock(a2->lock);
    }


}
```

Correct Solution

```
    typedef struct acct {
        float balance;
        pthread_mutex *lock;
        pthread_cond_t *cond;
    } account;

    void transfer_amount (account *a1, account *a2, float amount) {

        pthread_mutex_lock(a1->lock);

        while(a1->balance - amount < 0) {
            pthread_cond_wait(a1->cond, a1->lock);
        }
```

```
14
15          a1->balance -= amount;
16
17          pthread_mutex_unlock(a1->lock);
18
19
20          pthread_mutex_lock(a2->lock);
21
22          a2->balance += amount;
23
24          pthread_cond_broadcast(a2->cond);
25
26          pthread_mutex_unlock(a2->lock);
27
28
29      }
```

Notes

- **Thread**
  - **syntax (creation):**

```
1     int pthread_create(pthread_t *thread,
2                        const pthread_attr_t *attr,
3                        void * (*start_routine)(void*),
4                        void * arg)
```

    * `thread`
      · is a pointer to a structure of type `pthread_t`
    * `attr`
      · is used to specify any attributes this thread might have
      · is initialized with a separate call `pthread_attr_init()`
      · set default by passing `NULL`
    * `(start_routine)`
      · means which function this thread should start running in?
      · setting void pointer `(void *)` as an argument to function `start_routine` allows us to pass in any type of argument
      · setting void pointer `(void *)` as return type allows us to return any type of result
    * `args`
      · is where to pass the arguments for the function pointer ((start_routine))

    Example

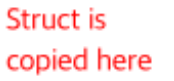
```c
#include <stdio.h>
#include <pthread.h>

typedef struct {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *args = (myarg_t *) arg;
    printf("%d %d\n", args->a, args->b);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    myarg_t args = { 10, 20 };

    int rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```



- **Condition Variable**
  - is an explicit queue that threads can put themselves on when some state of execution is not as desired (so it can be put to sleep)
  - when states are changed, one or more of the waiting threads can be awaken and be allowed to continue (done by **signaling** the condition)
  - queue is **FIFO**
  - `wait()` call is used to put thread to sleep
  - `singal()` call is used to awake thread from sleep
  - **Syntax (initialization):**

    `pthread_cont_t c = PTHREAD_COND_INITIALIZER`

    Example

```
1  int done  = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

– **Syntax (Wait):**

`Pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)`

Example

```
1  int done  = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

Put thread to sleep until done

– **Syntax (Signal):**

`Pthread_cond_signal(pthread_cond_t *c)`
Example

```
1  int done  = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

5. a) The algorithm will favor I/O bound process.

   CPU bound processes are ones that implement an algorithm that requires a large amount of processing time (e.g. algorithm that takes a long time to compute)

   I/O bound processes are interactive process, and it mostly waiting for the completion of input and output.

   A short run time of I/O bound process is expected, which means it won't consume a lot of processor time.

   It follows from above that the algorithm will favour I/O bound process over CPU-bound process.

   b) This algorithm can permanently starve CPU-bound process.

   This will happen when a constant stream of I/O bound processes occur over CPU bound process.

   The CPU bound process will never get its time to comlpete its process.

Notes

- **CPU-bound process** [1]
  - CPU Bound processes are ones that are implementing algorithms with a large number of calculations
  - Programs such as simulations may be CPU bound for most of the life of the process.
  - Users do not typically expect an immediate response from the computer when running CPU bound programs.
  - They should be given a lower priority by the scheduler.
- **I/O-bound process** [1]
  - Processes that are mostly waiting for the completion of input or output (I/O) are I/O Bound.
  - Interactive processes, such as office applications are mostly I/O bound the entire life of the process. Some processes may be I/O bound for only a few short periods of time.
  - The expected short run time of I/O bound processes means that they will not stay the running the process for very long.
  - They should be given high priority by the scheduler.

References

1) Kansas State University, The Process Scheduler, link