Lecture 10

Threads & Concurrency Control

Threads · ULT vs KLT · Race Conditions · Dijkstra · CAS · Stage 2

SENG 21213 · Computer Architecture & Operating Systems

Learning Objectives — By the end of this lecture you should be able to:

- 1. Define a thread and distinguish process-level resource ownership from thread-level execution characteristics
- 2. Compare ULT and KLT on 8 attributes including blocking behaviour, multi-core use, and context-switch cost
- 3. Analyse the myglobal race condition, identify the non-atomic READ-MODIFY-WRITE, and apply a mutex fix
- 4. Describe the three types of process interaction and the hazards associated with each
- 5. State all six Dijkstra requirements for mutual exclusion and evaluate a given mechanism against them
- 6. Explain CAS and XCHG hardware mutual exclusion and identify their three drawbacks

Section 1 · Threads and Multithreading

📖 **Stallings Reference**

- Chapter 4: Threads
- Section 4.1 — Processes and Threads
- Section 4.2 — Types of Threads

Multithreading extends the process abstraction by allowing a single process to have multiple concurrent execution contexts (threads) that share the same address space and resources.

1.1 Process Characteristics — Two Parts

Characteristic	Managed At	What It Represents
Resource Ownership	Process level	Virtual address space (code, data, heap), open files, I/O devices, IPC channels, shared memory. One per process.
Scheduling / Execution	Thread level	The execution path: program counter, register set, stack. Multiple threads per process — each can be independently scheduled.

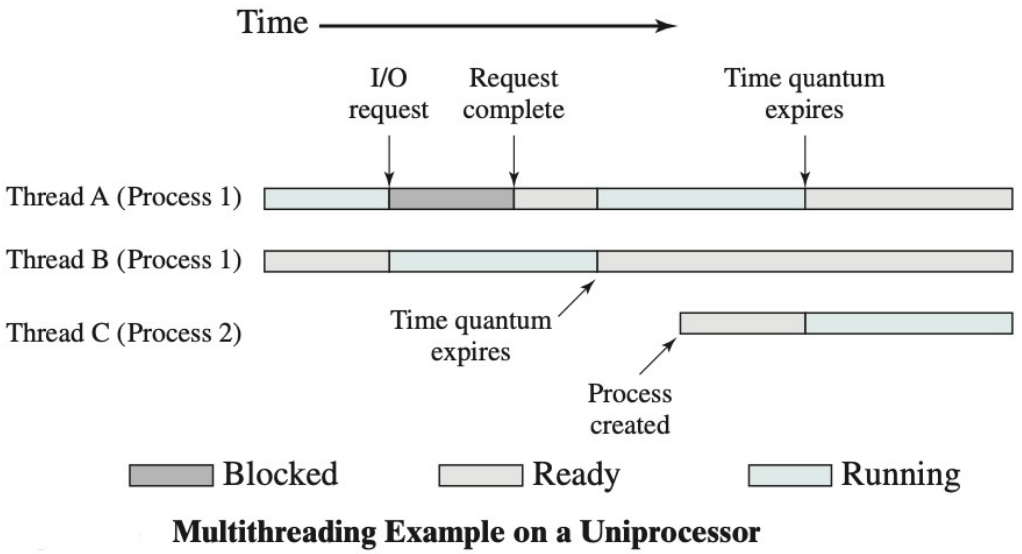
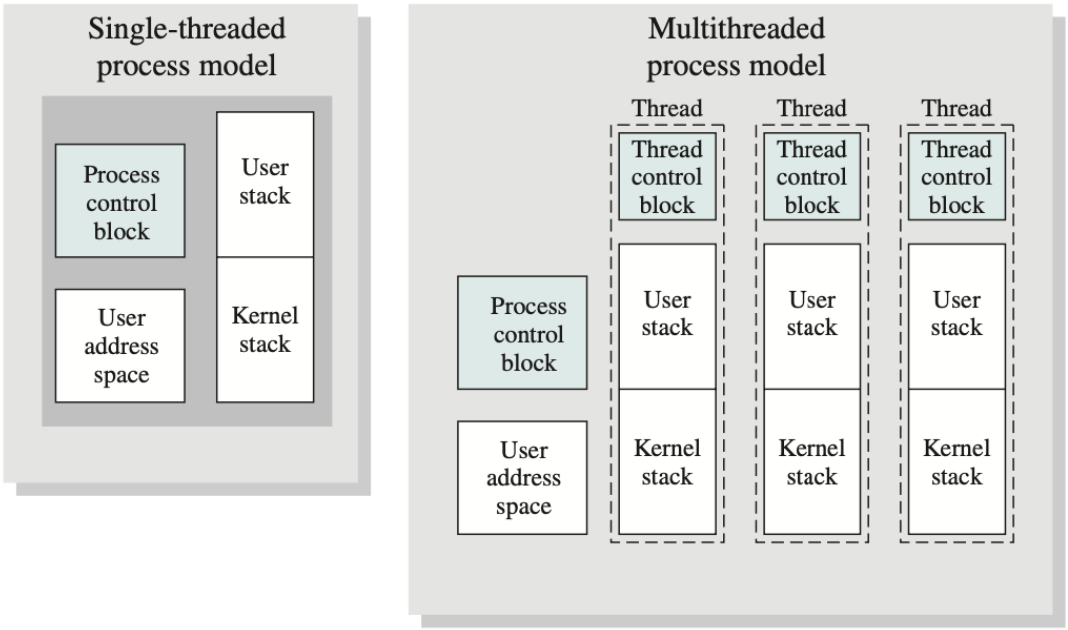


Figure: Single-threaded vs. multi-threaded process: shared code/data/heap; separate stacks and PCs per thread (Stallings Fig. 4.1)

1.2 Benefits of Multithreading

Benefit	Description	Example
Responsiveness	One thread blocks on slow I/O while another continues serving the user interface.	Web browser: one thread downloads page; another renders existing content; another responds to clicks.
Resource Sharing	Threads share process memory automatically — no IPC (pipes, sockets) needed for inter-thread communication.	Server: all worker threads share the connection pool and cache objects without copying.
Economy	Thread creation 10–100× cheaper than process creation. Context switch 5–10× cheaper for threads than processes.	Web server: spawning a thread per request is feasible; forking a process per request is too expensive.
Multiprocessor Utilisation	Multiple threads run in true parallel on separate cores — processes are the unit of parallelism.	A 4-thread matrix multiplication uses all 4 cores; a single-threaded version uses only 1.
Modularity	Decompose application into concurrent activities: I/O thread, compute thread, GUI thread.	Media player: decode thread, render thread, audio thread — each can be independently optimised.

1.3 Thread States and Operations



Single-Threaded and Multithreaded Process Models

Figure: User-Level Threads (ULT) vs. Kernel-Level Threads (KLT) implementation models (Stallings Fig. 4.5)

Operation	Description	State Transition
Spawn	Create a new thread with a given entry function and stack. New thread starts in Ready state.	None → Ready
Block	Thread calls a blocking operation (mutex_lock, semaphore wait, read). Releases CPU.	Running → Blocked
Unblock	Event the thread was waiting for occurred. Thread becomes eligible to run again.	Blocked → Ready
Finish	Thread returns from its entry function or calls pthread_exit(). Resources freed.	Running → Exit

1.4 ULT vs. KLT — Detailed Comparison

Attribute	ULT (User-Level Threads)	KLT (Kernel-Level Threads)
Who manages threads	Thread library in user space (no kernel involvement)	OS kernel — system calls for create, destroy, sync
Context switch cost	Very fast — no system call, no kernel mode transition	Slower — requires trap to kernel (~1 µs)
If one thread blocks on I/O	ENTIRE PROCESS blocks — kernel sees only one thread	Other threads in the process continue running
Multi-core utilisation	NO — kernel schedules the process on one core	YES — kernel can schedule threads on different cores

Attribute	ULT (User-Level Threads)	KLT (Kernel-Level Threads)
Signal delivery	Delivered to process; library routes to correct thread	Delivered directly to specific thread
Thread-local storage	Managed by library	Supported by kernel and compiler (TLS)
Portability	Can run on OS without kernel thread support	Tied to OS API (pthreads, Win32 threads)
Examples	Green threads (early Java, Ruby MRI), Go goroutines (hybrid)	Linux NPTL pthreads, Windows threads, macOS GCD

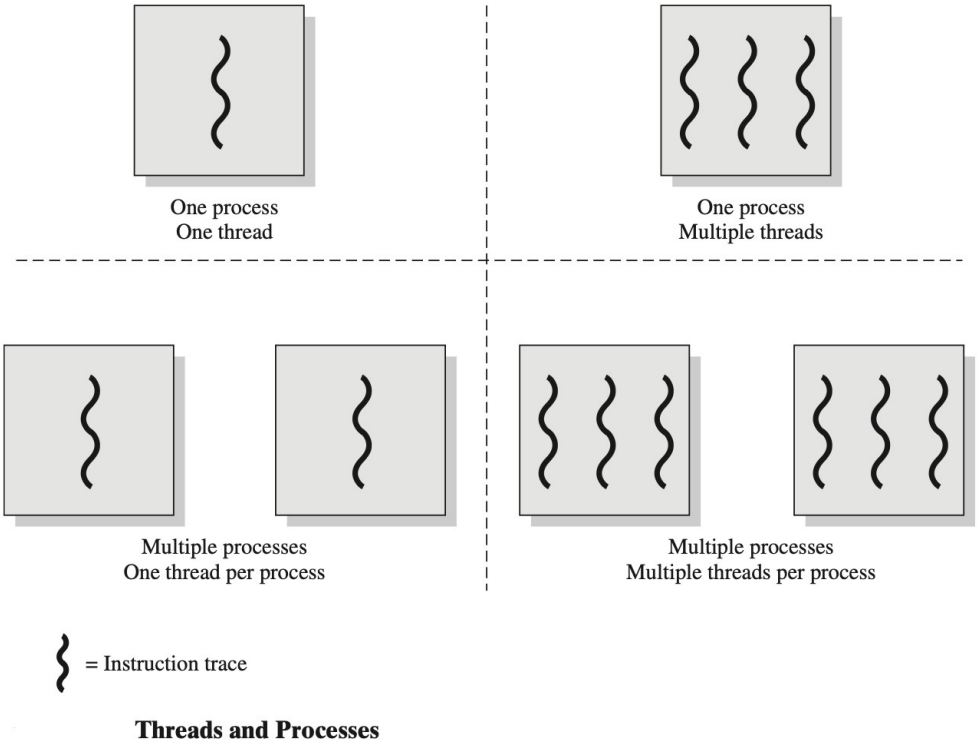


Figure: Multithreading models: many-to-one, one-to-one, many-to-many (Stallings Fig. 4.6)

Section 2 · Race Conditions and the Critical Section Problem

📖 Stallings Reference

- Chapter 5: Concurrency — Mutual Exclusion and Synchronisation
- Section 5.1 — Principles of Concurrency

2.1 The Echo Shared-Variable Problem

```
struct binary_semaphore {
       enum {zero, one} value;
       queueType queue;
};
void semWaitB(binary_semaphore s)
{
       if (s.value == one)
            s.value = zero;
       else {
                    /* place this process in s.queue */;
                    /* block this process */;
       }
}
void semSignalB(semaphore s)
{
       if (s.queue is empty())
            s.value = one;
       else {
                    /* remove a process P from s.queue */;
                    /* place process P on ready list */;
       }
}
```

Figure: Echo procedure with shared global variable chin — demonstrates why shared state requires protection (Stallings Fig. 5.1)

The echo() procedure stores a character in global variable chin, then outputs it. If two processes call echo() concurrently and process P1 is preempted AFTER storing to chin but BEFORE outputting, and P2 then calls echo() overwriting chin, P1 will output P2's character. P1's character is permanently lost.

✓ Solution

- Only ONE process/thread may execute inside echo() at any time — mutual exclusion on the critical section (the three statements that access/modify chin).

2.2 Race Conditions — Three Examples

★ Race Condition Definition

- A race condition occurs when the result of a computation depends on the relative ordering of execution of multiple threads or processes — and this ordering is non-deterministic (depends on scheduler timing).

Example	Scenario	Why It's a Race
Single Shared Variable	Two processes: P1 sets a=1; P2 sets a=2. Which wins?	Final value of a is undefined — depends entirely on scheduling order. Neither is 'correct'.
Dependent Variables	Invariant: b=1, c=2. P3: b←b+c. P4: c←b+c. In order P3→P4: b=3,c=5. In order P4→P3: b=4,c=3. Interleaved: b=2,c=3.	Three possible results from one program. Any interleaving of the four READ/WRITE operations gives a different result.
Invariant Violation	a==b must hold always. P1: a←a+1;	System invariant broken. Subsequent code relying

Example	Scenario	Why It's a Race
	b←b+1. P2: b←2*b; a←2*a. Interleaved: a=2→4, b=1→2→3. Now a=4 ≠ b=3.	on a==b produces wrong results or crashes.

2.3 Process Interaction Types

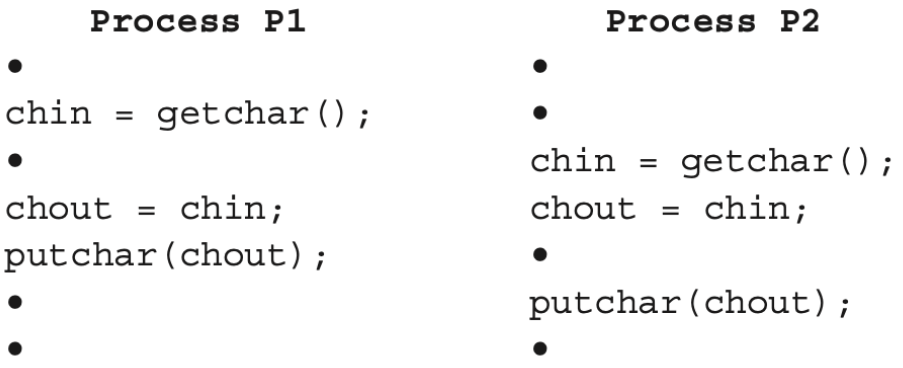


Figure: Three types of process interaction and the problems each can cause (Stallings Fig. 5.3)

Type	Description	Problems
Unaware of each other	Independent processes compete for the same resources (CPU, disk, printer) without knowing each other exist.	Mutual exclusion (access to non-sharable resources), Deadlock (each holds resource the other needs), Starvation (one never gets access)
Indirectly aware	Processes share access to a common object (buffer, database table, file) without explicitly knowing each other.	Mutual exclusion, Deadlock, Starvation, Data coherence (reading stale data)
Directly aware	Processes communicate explicitly by PID using IPC (pipes, message queues, signals, sockets). Designed to cooperate.	Deadlock (each waiting for a message from the other), Starvation (one message source dominates)

Section 3 · Mutual Exclusion — Requirements and Hardware

3.1 Dijkstra's Six Requirements

Any correct mutual exclusion mechanism must simultaneously satisfy ALL six requirements:

Requirement	Description
1. Mutual Exclusion	Only ONE process at a time may be in its critical section for a given shared resource.
2. Non-interference	A process halted outside its critical section must not interfere with other processes trying to enter.
3. No deadlock or	No set of processes should be indefinitely prevented from entering the critical section.

Requirement	Description
starvation	
4. Progress	If no process is in the critical section, any process requesting entry must eventually be allowed in, without delay.
5. No speed assumptions	The mechanism must work regardless of relative process execution speeds or number of processors.
6. Finite time inside	A process remains in its critical section for a finite, bounded time only.

3.2 Hardware Support — Compare-and-Swap (CAS)

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
     while (compare_and_swap(bolt, 0, 1) == 1)
         /* do nothing */;
     /* critical section */;
     bolt = 0;
     /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ... ,P(n));
}
```

(a) Compare and swap instruction

```
/* program mutualexclusion */
int const n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
      int keyi = 1;
      do exchange (&keyi, &bolt)
      while (keyi != 0);
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ..., P(n));
}
```

(b) Exchange instruction

Figure: Semaphore operation trace — sem_wait blocking and wakeup by sem_signal (Stallings Fig. 5.11)

```
/* Compare-and-Swap: atomic test-and-set operation */
/* Hardware executes all of this as ONE uninterruptible instruction */
int compare_and_swap(int *word, int testval, int newval) {
    int oldval = *word;
    if (oldval == testval) *word = newval;  /* atomic: bus is locked */
    return oldval;
}

/* Spinlock implemented with CAS: */
int lock = 0;  /* 0 = unlocked, 1 = locked */

void acquire_lock() {
    while (compare_and_swap(&lock, 0, 1) != 0)  /* spin */
        ;  /* yield(); in practice — avoids burning CPU */
}
void release_lock() { lock = 0; }  /* atomic store on most ISAs */

/* x86 CMPXCHG instruction implements CAS in hardware */
/* LOCK prefix makes the memory bus exclusive during the operation */
```

3.3 Hardware Support — Exchange Instruction

```
/* Exchange (XCHG) instruction: atomically swaps register and memory */
/* x86: XCHG has an implicit LOCK prefix — always atomic */
void exchange(int *reg, int *memory) {
    /* Hardware does this atomically — cannot be interrupted mid-swap */
    int temp = *memory;
    *memory = *reg;
    *reg = temp;
}

/* Spinlock with XCHG: */
int lock = 0;
void lock_acquire() {
    int key = 1;
    do { exchange(&key, &lock); } while (key == 1);  /* spin */
}
void lock_release() { lock = 0; }
```

3.4 The myglobal Race Condition — Pthreads

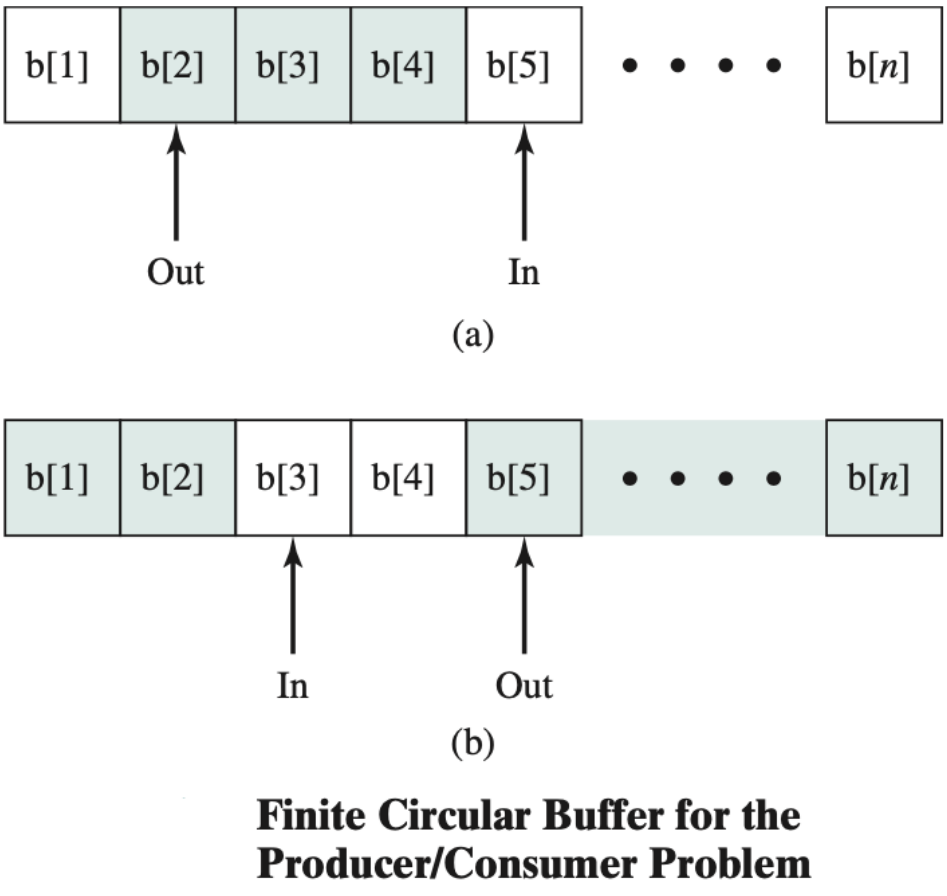


Figure: Race condition in producer-consumer shared buffer — concurrent access without mutual exclusion (Stallings Fig. 5.14)

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

int myglobal = 0;   /* SHARED — no protection! */
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *thread_fn(void *arg) {
    int i, j;
    for (i = 0; i < 20; i++) {
        /* UNSAFE version: */
        j = myglobal;   /* (1) READ  — another thread may preempt HERE */
        j = j + 1;      /* (2) INC local copy */
        sleep(1);       /* VERY likely to be preempted here */
        myglobal = j;   /* (3) WRITE — may overwrite other thread's write */
        /* Result: READ-MODIFY-WRITE is NOT atomic → lost updates */
        /* SAFE version (comment out unsafe; uncomment below): */
        /* pthread_mutex_lock(&lock);   */
        /* myglobal++;                  */
        /* pthread_mutex_unlock(&lock); */
    }
    return NULL;
}

int main() {
    pthread_t t1;
    pthread_create(&t1, NULL, thread_fn, NULL);
    thread_fn(NULL);            /* main thread also does 20 increments */
    pthread_join(t1, NULL);
    printf("myglobal = %d\n", myglobal);
    /* Expected: 40. Actual without mutex: often 20-39 (lost updates) */
    return 0;
}
```

⚠ Hardware Mutual Exclusion Drawbacks

- Busy waiting (spinning): Process consumes CPU cycles doing nothing useful while waiting for the lock. Only acceptable for very short critical sections (< a few µs). For longer waits, block the process instead.
- Starvation: When multiple processes are spinning, the selection of which one 'wins' the CAS is non-deterministic. A single process could theoretically wait indefinitely.
- Priority inversion: Low-priority process L holds lock; high-priority process H spins waiting for it. H occupies CPU preventing L from running and releasing the lock. Neither progresses. Fix: priority inheritance protocol.

Section 4 · OS Assignment — Stage 2

★ Milestone 2: Threads + Mutex + Semaphore

- thread.h — Define thread_t: { int tid; int state; uint32_t esp; uint32_t eip; pcb_t *parent; uint32_t stack[1024]; }

- thread.c — create_thread(void (*entry)(), pcb_t *parent): allocate thread_t on kernel heap, initialise stack frame (push entry address), add to parent's thread_list.
- mutex.c — typedef struct { int locked; thread_t *waitq; } mutex_t; void mutex_lock(mutex_t *m): if m->locked, add current thread to m->waitq and block(); else m->locked=1. void mutex_unlock(mutex_t *m): if waitq nonempty, wakeup(dequeue(waitq)); else m->locked=0.
- semaphore.c — typedef struct { int count; thread_t *waitq; } sem_t; sem_wait(s): s->count--; if s->count < 0 block(); sem_signal(s): s->count++; if s->count <= 0 wakeup(dequeue(waitq)).
- Test 1: Port myglobal race — run without mutex (wrong count), then with mutex (count=40).
- Test 2: Producer-consumer with bounded buffer (size 5) using semaphores n (items), e (empty slots), s (mutex).

Section 5 · Practice Exercises

✎ Exercise 10.1 [12 marks] — Threads

- (a) [4] A web server handles requests using either: (A) fork() — one process per request, or (B) pthread_create() — one thread per request. Compare these approaches on 4 attributes: (i) memory isolation, (ii) creation overhead, (iii) communication between handlers, (iv) one handler crashing affects others. Which is safer? Which is more scalable?
- (b) [4] A ULT library implements 4 threads within one process (4-to-1 model). Thread T2 calls read() on a network socket (blocking). (i) What happens to threads T1, T3, T4? (ii) Why does this happen? (iii) How does the KLT (1-to-1) model solve this? (iv) What trade-off does 1-to-1 introduce?
- (c) [4] Describe the stack management in a multi-threaded process. (i) How many stack segments exist? (ii) Does each thread have its own stack pointer (SP/ESP) register? (iii) What happens if two threads overflow their stacks simultaneously? (iv) In your Stage 2 kernel, threads share the same kernel stack or have separate stacks? Why does the answer matter for context switching?

✎ Exercise 10.2 [14 marks] — Race Conditions and Mutual Exclusion

- (a) [6] The following two threads execute concurrently, sharing int x = 0 and int y = 0:
- Thread 1: x = 1; r1 = y; Thread 2: y = 1; r2 = x;
- (i) List ALL possible values of (r1, r2) if operations can be reordered arbitrarily.
- (ii) Which value of (r1, r2) is impossible? Explain what ordering prevents it.
- (iii) Can (r1=0, r2=0) occur on a single-processor system with context switches only between complete statements? Justify.
- (iv) Can (r1=0, r2=0) occur on a multi-core system? What hardware phenomenon enables it? Name the CPU memory model property that prevents it.
- (b) [4] Apply Dijkstra's 6 requirements to evaluate Peterson's Algorithm for 2-process mutual exclusion. For each requirement, state whether Peterson's satisfies it and briefly explain.
- (c) [4] Evaluate the Compare-and-Swap spinlock against Dijkstra's 6 requirements. Identify which requirement(s) it FAILS and explain why. Propose a queue-based modification to fix the failed requirement(s).