Operating Systems & Systems Programming

Sample Exam Papers — 2 Mock Tests

Module: 06-38059 · Second Year · University of Birmingham
School of Computer Science

Two original mock papers following the 2024/25 exam format. Each paper contains **three questions**, must be answered in **two hours**, and is marked out of 60 (rescaled to 100).

How to use these papers:

1. Sit each paper under exam conditions (phone away, 2-hour timer).
2. Do **not** look at the model answers until you finish.
3. Self-mark using the marking outline at the back.
4. Note your weakest topics; re-read those sections of the revision guide.

Sample Paper 1

Time allowed: 2 hours · [Answer all questions]

Calculators allowed (non-alphabetic; hex allowed)

> Note: Each question is marked out of 20. The paper will be marked out of 60, which will be rescaled to a mark out of 100.

Question 1

The following C code maintains a dynamic list of names (for example, a contact list).

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #define MAX_NAME 32
5
6   typedef struct {
7       char *names;       // flat block: MAX_NAME bytes per slot
8       int   count;
9       int   capacity;
10  } NameList;
11
12  NameList *createList(int initialCapacity) {
13      NameList *list = malloc(sizeof(NameList));
14      list->names    = malloc(initialCapacity * MAX_NAME);
15      list->capacity = initialCapacity;
16      list->count    = 0;
17      return list;
18  }
19
20  void addName(NameList *list, char *name) {
21      if (list->count == list->capacity) {
22          list->capacity = list->capacity * 2;
23          list->names    = realloc(list->names, list->capacity * MAX_NAME);
24      }
25      strcpy(list->names + (list->count * MAX_NAME), name);
26      list->count++;
27  }
28
29  char *getName(NameList *list, int index) {
30      return list->names + (index * MAX_NAME);
31  }
32
33  void destroyList(NameList *list) {
34      free(list);
35  }
```

(a) Explain the difference between malloc and calloc. Which would be more appropriate on line 14, and why? **[3 marks]**

(b) The code above has at least **four** bugs. For each bug, state the line number, explain the bug clearly, and give a corrected version of the affected line(s). **[10 marks]**

(c) Write a function with the signature int findName(NameList *list, char *name) that returns the index of name in the list (0-based), or -1 if not found. Use linear search. **[7 marks]**

Question 2

A web-analytics service maintains a per-URL counter of visits. Many web-server threads call recordVisit concurrently.

```c
#include <pthread.h>
#include <string.h>
#define MAX_PAGES 100

typedef struct {
    char url[256];
    long visits;
} PageCounter;

PageCounter pages[MAX_PAGES];
int numPages = 0;

int registerPage(const char *url) {
    if (numPages >= MAX_PAGES) return -1;
    strcpy(pages[numPages].url, url);
    pages[numPages].visits = 0;
    return numPages++;
}

void recordVisit(const char *url) {
    for (int i = 0; i < numPages; i++) {
        if (strcmp(pages[i].url, url) == 0) {
            pages[i].visits++;
            return;
        }
    }
}

long getVisits(const char *url) {
    for (int i = 0; i < numPages; i++) {
        if (strcmp(pages[i].url, url) == 0) {
            return pages[i].visits;
        }
    }
    return -1;
}
```

(a) Identify **two** distinct race conditions in this code. For each, describe a specific interleaving that causes incorrect behaviour. **[3 marks]**

(b) Modify the code to be thread-safe while **maximising parallelism**. Use pthreads (pthread_mutex_t and/or reader-writer locks). Briefly justify the placement of every lock and unlock. **[10 marks]**

(c) Why would wrapping the entire body of every function in a single global mutex be considered "thread-safe but slow"? Refer to getVisits in your answer. **[3 marks]**

(d) Suppose recordVisit is extended to append a line to a log file (a slow disk operation that takes around 10 ms). Should the mutex be released *before* or *after* the disk write? Justify with a concrete scenario showing the consequence of the wrong choice. **[4 marks]**

Question 3

Consider an air-traffic-control server running the following three processes:

- **Radar tracker** — receives a radar pulse every 100 ms; must finish processing each pulse within 50 ms or the pulse is too late. Modest CPU and memory.
- **Flight planner** — computes optimal trajectories for newly filed flights. CPU-intensive (a single computation can take many seconds), uses approximately 2 GB of memory per running computation.
- **Operator console** — GUI used by the human controllers; redraws twice per second. Small memory, must remain visually responsive.

The server has 4 GB of physical memory and one CPU core.

(a) If you used a **non-preemptive** scheduler, could the radar tracker still be guaranteed to meet its 50 ms deadline? Justify your answer with a worst-case scenario. **[4 marks]**

(b) What problem would a strict **priority scheduler** (radar > console > planner) cause? How would you fix it? **[4 marks]**

(c) Describe a suitable scheduling strategy for this system. Refer to each of the three processes by name and **justify** your choice. **[6 marks]**

(d) Two flight-planner computations can be running concurrently, each "wanting" 2 GB, so the total demand may exceed the 4 GB of physical memory. Describe a suitable memory-management strategy and justify why it is appropriate for this scenario. **[6 marks]**

— End of Sample Paper 1 —

Sample Paper 2

Time allowed: 2 hours · [Answer all questions]

Calculators allowed (non-alphabetic; hex allowed)

> Note: Each question is marked out of 20. The paper will be marked out of 60, which will be rescaled to a mark out of 100.

Question 1

A library system stores books in a singly-linked list ordered by ISBN.

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

typedef struct book {
    char isbn[14];
    char *title;
    struct book *next;
} book_t;

book_t *insertBook(book_t *head, char *isbn, char *title) {
    book_t *newBook = malloc(sizeof(book_t));
    strcpy(newBook->isbn, isbn);
    newBook->title = title;

    if (head == NULL || strcmp(newBook->isbn, head->isbn) < 0) {
        newBook->next = head;
        return newBook;
    }
    book_t *cur = head;
    while (strcmp(cur->next->isbn, newBook->isbn) < 0) {
        cur = cur->next;
    }
    newBook->next = cur->next;
    cur->next = newBook;
    return head;
}

book_t *removeBook(book_t *head, char *isbn) {
    book_t *cur = head;
    while (strcmp(cur->isbn, isbn) != 0) {
        cur = cur->next;
    }
    free(cur->title);
    free(cur);
    return head;
}
```

(a) Explain the difference between passing book_t *head and book_t **head to insertBook. Why might the second be preferable for a function that may insert at the front of the list? **[4 marks]**

(b) The code has at least **five** bugs. For each, state the line number, explain the bug, and give a corrected version. **[12 marks]**

(c) Write a function with the signature int countBooks(book_t *head) that returns the number of books in the list. **[4 marks]**

Question 2

A shared printer queue accepts jobs from many client threads. A single printer thread takes jobs from the queue one at a time.

```c
#include <pthread.h>
#define QUEUE_SIZE 50

typedef struct {
    char filename[128];
    int  pages;
} PrintJob;

PrintJob queue[QUEUE_SIZE];
int head = 0, tail = 0, jobCount = 0;

void submitJob(PrintJob job) {
    while (jobCount == QUEUE_SIZE) { /* busy wait */ }
    queue[tail] = job;
    tail = (tail + 1) % QUEUE_SIZE;
    jobCount++;
}

PrintJob takeJob(void) {
    while (jobCount == 0) { /* busy wait */ }
    PrintJob j = queue[head];
    head = (head + 1) % QUEUE_SIZE;
    jobCount--;
    return j;
}
```

(a) What is a race condition? Why is busy-waiting on an unprotected shared variable such as jobCount particularly unsafe (give two different reasons)? **[4 marks]**

(b) Rewrite both functions using a pthread_mutex_t and condition variables (or counting semaphores) so that they are correct **and** efficient (no busy-waiting). Briefly explain your logic. **[12 marks]**

(c) Explain the difference between a mutex and a counting semaphore. Give one specific situation in which a semaphore would be preferable to a mutex. **[4 marks]**

Question 3

Consider a smartphone running the following four processes:

- **Phone app** — handles incoming calls. When a call arrives, the system must display the call screen within 200 ms. Calls are infrequent but critical.
- **Camera app** — runs in the foreground, captures and processes video frames at 60 frames per second. CPU-intensive, modest memory.
- **Email sync** — runs in the background; periodically downloads new mail (intermittent network I/O with large bursts).
- **Background music** — streams audio from a file. Steady, low CPU; the audio must never glitch.

(a) Why are kernel critical sections required to be short on a mobile device? Give **two** specific consequences of a long critical section in this scenario. **[4 marks]**

(b) Should the phone app's interrupt handler run at the same priority as a normal user process? Justify your answer. **[4 marks]**

(c) Describe a suitable scheduling strategy that satisfies all four processes' requirements. Refer to each process by name and use the appropriate scheduling-class terminology where relevant. **[6 marks]**

(d) Phones have very limited RAM. When the user switches to the camera and memory pressure becomes high, which app(s) should be considered for swap-out and why? Justify with reference to memory-management concepts from the module. **[6 marks]**

— End of Sample Paper 2 —

Model Answers

Detailed walkthroughs for self-marking

Note: Don't peek before finishing both papers. Where these answers say *"see §X of the revision guide"*, that's a hint that the theory you needed is in the main 7-Day Revision Guide PDF.

Sample Paper 1 — Model Answers

Q1 — NameList code review

(a) malloc vs calloc (3 marks)

malloc(size) allocates size bytes of uninitialised memory — the contents are whatever was previously there (garbage). **calloc(n, size)** allocates n × size bytes and zero-initialises them. On line 14, **calloc would be safer**: the block holds fixed-size string slots, and zero-filling guarantees every uninitialised slot reads as an empty string (its first byte is '\0'), preventing accidental garbage reads. (Either answer with proper justification scores full marks.)

(b) Four bugs and fixes (10 marks)

Bug 1 — Line 13: No NULL check after malloc(sizeof(NameList)). If allocation fails, the next line dereferences NULL. **Fix:**

```
NameList *list = malloc(sizeof(NameList));
if (list == NULL) return NULL;
```

Bug 2 — Line 14: Same — no NULL check after the second malloc. Worse, if it fails the NameList itself is leaked. **Fix:**

```
list->names = malloc(initialCapacity * MAX_NAME);
if (list->names == NULL) { free(list); return NULL; }
```

Bug 3 — Line 23: realloc returns the new pointer (which may be a *different* address) or NULL on failure. Assigning it back into list->names directly destroys the original pointer on failure, leaking the old buffer. **Fix:**

```
char *tmp = realloc(list->names, list->capacity * MAX_NAME);
if (tmp == NULL) { /* keep old buffer, report failure */ return; }
list->names = tmp;
```

Bug 4 — Line 25 (strcpy): If name is longer than MAX_NAME - 1 characters, strcpy overruns the slot and corrupts the next slot — a classic **buffer overflow**. **Fix:** use strncpy with explicit null termination:

```
char *slot = list->names + (list->count * MAX_NAME);
strncpy(slot, name, MAX_NAME - 1);
slot[MAX_NAME - 1] = '\0';
```

Bug 5 — Line 34 (destroyList): Only frees the wrapper struct, leaking list->names. **Fix:**

```
void destroyList(NameList *list) {
    if (list == NULL) return;
    free(list->names);
    free(list);
}
```

Note: Any four of the five above earn full marks. Marks are split: ~1 mark for spotting the bug, ~1.5 marks for the explanation and fix.

(c) findName function (7 marks)

```c
int findName(NameList *list, char *name) {
    if (list == NULL || name == NULL) return -1;
    for (int i = 0; i < list->count; i++) {
        char *slot = list->names + (i * MAX_NAME);
        if (strcmp(slot, name) == 0) {
            return i;
        }
    }
    return -1;
}
```

Mark scheme: +2 for the loop bound (i < count, not capacity); +2 for the correct slot offset arithmetic; +1 for using strcmp (not ==); +1 for returning -1 when not found; +1 for defensive NULL check or commented assumption.

Q2 — Concurrent visit counter

(a) Two race conditions (3 marks)

Race 1: recordVisit ↔ recordVisit. Two threads both execute pages[i].visits++ simultaneously. The increment is not atomic — it reads the value, adds one, writes back. If both threads read the same old value v, both write v+1, so one increment is **lost** (the count is too low).

Race 2: registerPage ↔ recordVisit (or getVisits). Between writing the URL (line 15) and incrementing numPages (line 17), another thread can iterate up to the new numPages-1 and read a half-constructed entry. Or: two threads both call registerPage, both read the same numPages, both write to the same slot, then both increment — leaving a duplicate slot and one lost registration.

(b) Thread-safe with maximum parallelism (10 marks)

Strategy: **one reader-writer lock**. Reads (getVisits) can run concurrently with each other. Writes that change the array structure (registerPage) need exclusive access. Increments (recordVisit) only modify a single counter, so they take the read lock for the lookup and use an atomic operation for the increment — but a simpler, fully correct answer uses the write lock for recordVisit too.

```c
#include <pthread.h>

pthread_rwlock_t lock = PTHREAD_RWLOCK_INITIALIZER;

int registerPage(const char *url) {
    pthread_rwlock_wrlock(&lock);
    if (numPages >= MAX_PAGES) {
        pthread_rwlock_unlock(&lock);
        return -1;
    }
    strcpy(pages[numPages].url, url);
    pages[numPages].visits = 0;
    int idx = numPages++;
    pthread_rwlock_unlock(&lock);
    return idx;
}

void recordVisit(const char *url) {
    pthread_rwlock_wrlock(&lock);            // we modify .visits
    for (int i = 0; i < numPages; i++) {
        if (strcmp(pages[i].url, url) == 0) {
            pages[i].visits++;
            pthread_rwlock_unlock(&lock);    // unlock BEFORE return
            return;
        }
    }
    pthread_rwlock_unlock(&lock);
}

long getVisits(const char *url) {
    pthread_rwlock_rdlock(&lock);            // many readers allowed
    for (int i = 0; i < numPages; i++) {
        if (strcmp(pages[i].url, url) == 0) {
            long v = pages[i].visits;
            pthread_rwlock_unlock(&lock);
            return v;
        }
    }
    pthread_rwlock_unlock(&lock);
    return -1;
}
```

Mark scheme: +3 for choosing rwlock (or justified mutex+atomic) to allow concurrent reads; +3 for correct lock/unlock around *every* path including every return; +2 for separating read vs write semantics correctly; +2 for an explanation that mentions "maximise parallelism by letting getVisits run

concurrently".

(c) Why one global mutex is slow (3 marks)

A single global mutex **serialises every call**. If 100 web-server threads all call getVisits for read-only statistics, they queue up behind one another even though they do not write anything — they only read independent fields. The hardware can serve all 100 in parallel, but the global mutex forces them into single-file. The contention point becomes the lock itself.

(d) Release the mutex before the disk write (4 marks)

Release before the 10 ms disk write. **Scenario:** with 100 threads per second, holding the mutex during the write means each thread waits behind the previous one's 10 ms write — throughput collapses to 100 ops/s regardless of CPU count. Releasing first means the next thread can immediately increment its counter and queue its own write in parallel.

Caveat: writes to the same log file must still be serialised (the file handle is a shared resource). A separate, finer-grained mutex on the log handle is the textbook answer.

Q3 — Air-traffic-control scheduling

(a) Non-preemptive scheduling (4 marks)

No, the deadline cannot be guaranteed. Under non-preemptive scheduling, once a process starts running on the CPU, the scheduler cannot interrupt it. If the flight planner begins a CPU burst (which can run for many seconds), and a radar pulse arrives during that burst, the radar tracker must wait until the planner voluntarily yields or blocks on I/O. Many seconds >> 50 ms, so the pulse misses its deadline. Preemption is essential here.

(b) Strict priority — and its fix (4 marks)

Strict priority causes **starvation** of the flight planner. While anything higher-priority is runnable (the radar is constantly running every 100 ms; the console refreshes twice a second), the planner never gets the CPU and its results are never delivered.

Fix: aging. The scheduler gradually raises the priority of any process that has been waiting for too long, so eventually the planner's priority exceeds the radar's and it gets a turn. (Alternative correct fix: use a multilevel queue with guaranteed time share for the planner.)

(c) Suitable scheduling strategy (6 marks)

Preemptive priority scheduling with aging, ideally inside a multilevel queue. Mapping:

- **Radar tracker** → highest priority class (real-time). Preempts everything; needs the 50 ms guarantee.
- **Operator console** → medium priority (interactive). Short bursts twice a second; round-robin within the interactive class is fine.
- **Flight planner** → low priority (batch/background). Gets CPU only when neither higher-priority process is runnable, plus the aging top-up so it eventually finishes.

Justification: matches each process's actual need (hard deadline for radar; responsiveness for console; throughput for planner) and prevents starvation via aging.

(d) Memory-management strategy (6 marks)

Demand paging with virtual memory. Each planner computation has a 2 GB *virtual* address space, but only its **working set** of pages is in RAM at any moment. With two planners running and 4 GB RAM, the OS swaps less-recently-used pages of one planner to disk while the other is computing, and vice versa.

Key constraints to mention:

- Use a **working-set model** to size each process's resident pages — give the planner enough frames that it does not **thrash** (constantly fault).
- The radar tracker's pages should be **pinned** (locked in RAM, never swapped) to honour the real-time deadline — a page fault on radar code = miss.
- Console is small; cheap to leave resident.

Justification: demand paging gives each planner the illusion of 2 GB without requiring the OS to physically own 4+ GB; pinning radar code keeps the deadline; working-set sizing prevents thrashing.

Sample Paper 2 — Model Answers

Q1 — Book linked list

(a) head vs head pointer-to-pointer (4 marks)

Passing book_t *head gives the function a **copy** of the pointer. If the function reassigns head (because it inserts at the front and the new node becomes the new head), the caller does not see the change — it must use the return value.

Passing book_t **head gives the function the **address** of the caller's head-pointer variable. The function can write through it (*head = newBook) so the caller automatically sees the updated head — no need to remember to assign the return value, eliminating a class of bugs where the caller writes insertBook(head, ...) instead of head = insertBook(head, ...).

(b) Five bugs and fixes (12 marks)

Bug 1 — Line 11: No NULL check after malloc. **Fix:** if (newBook == NULL) return head;

Bug 2 — Line 12 (strcpy): isbn field is char[14]; if the caller passes a longer string this overruns into title. **Fix:** strncpy(newBook->isbn, isbn, 13); newBook->isbn[13] = '\0';

Bug 3 — Line 13: Stores the caller's title pointer directly. If the caller later frees it, or it points to a stack buffer, the book is left dangling. **Fix:** duplicate it: newBook->title = strdup(title);

Bug 4 — Line 20: The condition strcmp(cur->next->isbn, ...) dereferences cur->next without checking it is not NULL. If the new ISBN is larger than every existing ISBN, the walk falls off the end and dereferences NULL. **Fix:**

```
while (cur->next != NULL &&
       strcmp(cur->next->isbn, newBook->isbn) < 0) {
    cur = cur->next;
}
```

Bug 5 — removeBook, line 30: No null check on cur. If the ISBN is not in the list, the loop walks off the end and dereferences NULL. **Fix:**

```
while (cur != NULL && strcmp(cur->isbn, isbn) != 0) {
    cur = cur->next;
}
if (cur == NULL) return head;       // not found
```

Bug 6 — removeBook: Frees the node but never **unlinks** it from the list. The previous node's next still points at freed memory — classic use-after-free on the next operation. **Fix:** walk with two pointers (prev and cur) and set prev->next = cur->next; before freeing. Also handle the head case (cur == head) by returning cur->next.

(c) countBooks function (4 marks)

```
int countBooks(book_t *head) {
    int n = 0;
    for (book_t *cur = head; cur != NULL; cur = cur->next) {
        n++;
    }
    return n;
}
```

Mark scheme: +2 for the NULL-terminated walk; +1 for the counter; +1 for accepting an empty list (returns 0 cleanly).

Q2 — Printer queue

(a) Race condition and busy-wait risks (4 marks)

A **race condition** occurs when two or more threads access shared mutable state and the final outcome depends on the order their instructions interleave. Busy-waiting on an unprotected jobCount is unsafe because:

- **Lost-wakeup / torn read:** the reader may see a partially-updated value of jobCount (an old half mixed with a new half on architectures where int writes are not atomic), or pass the == 0 check between the writer's update of jobCount and the index variables.
- **CPU waste:** the spinning thread occupies a CPU core doing no useful work; on a single-core system it may even prevent the other thread from running and actually changing jobCount — a **livelock**.

(b) Correct and efficient version (12 marks)

```c
#include <pthread.h>

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

void submitJob(PrintJob job) {
    pthread_mutex_lock(&m);
    while (jobCount == QUEUE_SIZE) {            // wait in a while, not if
        pthread_cond_wait(&not_full, &m);
    }
    queue[tail] = job;
    tail = (tail + 1) % QUEUE_SIZE;
    jobCount++;
    pthread_cond_signal(&not_empty);            // wake one taker
    pthread_mutex_unlock(&m);
}

PrintJob takeJob(void) {
    pthread_mutex_lock(&m);
    while (jobCount == 0) {                     // wait in a while
        pthread_cond_wait(&not_empty, &m);
    }
    PrintJob j = queue[head];
    head = (head + 1) % QUEUE_SIZE;
    jobCount--;
    pthread_cond_signal(&not_full);             // wake one submitter
    pthread_mutex_unlock(&m);
    return j;
}
```

Mark scheme: +2 for the mutex protecting the shared state; +3 for the two condition variables (full/empty) — one is not enough; +3 for waiting in a while loop (not if — protects against spurious wakeups); +2 for signalling the correct cond from each function; +2 for unlocking before return in takeJob.

Alternative: counting semaphores. Use sem_init(&empty, 0, QUEUE_SIZE), sem_init(&full, 0, 0), plus a binary mutex for the head/tail update — see §7 of the revision guide.

(c) Mutex vs semaphore (4 marks)

A **mutex** is a binary lock with the constraint that only the thread that locked it may unlock it (ownership). A **counting semaphore** holds an integer count and supports two operations: wait (decrement, block if zero) and signal (increment, wake one waiter). Semaphores have no ownership and the count can be any non-negative number.

When to prefer a semaphore: when modelling a pool of N identical resources (e.g. "five database connections available"). A mutex can only express 0 or 1; a semaphore with initial value 5 lets up to five

threads proceed concurrently and the sixth blocks. Producer/consumer with bounded buffers (above) is the canonical example.

Q3 — Smartphone scheduling and memory

(a) Why kernel critical sections must be short (4 marks)

Kernel critical sections often **disable interrupts** (or hold a spinlock that delays interrupt handlers) to prevent races inside the kernel. While they are executing:

- **Latency consequence:** if the phone-call interrupt fires while a long critical section is held, the call screen does not appear within 200 ms — the user perceives a frozen phone.
- **Glitching consequence:** the audio driver's interrupt-driven buffer refill can be delayed past the audio device's deadline, causing a click/glitch in the background music. Frame-deadline misses in the camera pipeline are similar.

(b) Interrupt handler priority (4 marks)

No. Interrupt handlers run in *interrupt context* and must complete as quickly as possible — they are above user-process priority entirely, not scheduled by the same mechanism. Treating them like a normal process would:

- Miss the 200 ms call-screen deadline if any user process were ahead.
- Allow the camera (CPU-heavy) to delay servicing the call, which is unacceptable.

Typical kernel design: a short **top half** in interrupt context (acknowledge the hardware, queue the work), then a **bottom half** in process context (do the actual processing) — see §10 of the revision guide.

(c) Suitable scheduling strategy (6 marks)

Multilevel queue + preemptive priority + round-robin within each level:

- **Interrupt-context (top half) — phone call**: above all scheduling.
- **Real-time class — background music** (and the audio bottom half): must meet audio-buffer deadlines; given a small, guaranteed CPU share.
- **Foreground/interactive class — camera**: high priority, round-robin with the music's deadline-driven slot; preempts background work.
- **Background class — email sync**: runs only when nothing higher is runnable. Aging ensures it eventually completes.

Justification: matches each process's actual requirement (hard real-time for audio glitches; foreground responsiveness for camera; opportunistic work for email); prevents starvation of email via aging.

(d) Memory-pressure swap-out strategy (6 marks)

Pick the least-needed process and swap it out. Justification for each:

- **Email sync** — first candidate. It runs intermittently; if its memory (downloaded message buffers) is swapped to disk and reloaded the next time it wakes up, the user notices no difference. Background processes that have been idle longest are obvious swap targets.
- **Background music** — do **not** swap its decoded-audio working set; a page fault during playback would cause an audible glitch. Its working set is small anyway.
- **Phone app code** — **pin** its handler pages in memory. A page fault while answering a call would blow the 200 ms deadline; the kernel must guarantee it never pages.
- **Camera** — currently in use, so its working set is hot. Swapping it out destroys the very thing the user is interacting with.

Concepts to mention: *working-set model* (give each app the pages it actively uses), *demand paging* (only the pages currently touched are resident), *pinning* for real-time pages, and a *LRU-style replacement policy* (least recently used pages of email sync are the natural eviction victims) — see §11 of the revision guide.

Self-Marking Summary

After taking each paper, tot up your marks. The total is out of 60, then rescale: your_score / 60 × 100. Birmingham CS grade boundaries (approximate):

70+ First · **60–69** 2:1 · **50–59** 2:2 · **40–49** Third · **<40** Fail

If you scored under 50: identify which sub-parts cost you the most marks. Almost always it is one of: (a) missing a NULL check, (b) not unlocking before every return, (c) giving a generic answer without referencing the scenario by name. Re-read §1 (Exam Strategy) of the main revision guide and retry that question type.

If you scored 50–65: you are in the safe-pass range. Push for the harder marks: writing complete corrected code in (b)-style parts and justifying each design choice in Q3.

If you scored 65+: you are on track for a 2:1 or higher. Time yourself strictly on the second paper to make sure speed is not your weak point.