제공된 강의 자료와 과거 시험 스타일(개념적 이해, 정확한 용어 사용, 코드 분석 강조)을 바탕으로 작성된 **포괄적인 스터디 가이드**입니다.

이 가이드는 **"자체적으로 완결된(internally complete)"** 구조를 가지고 있어, 교과서 없이도 시험 문제 해결에 필요한 핵심 메커니즘을 이해할 수 있도록 구성되었습니다.

1. 동시성(Concurrency): 추상화 (The Abstraction)

프로세스(Process) vs. 스레드(Thread)

- **프로세스 (고전적 관점):** 실행의 단일 지점. 하나의 PC(프로그램 카운터), 하나의 스택, 하나의 주소 공간을 가짐.
- **스레드 (새로운 관점):** 단일 프로세스 내의 여러 실행 지점.
  - **공유 상태 (Shared State):** 프로세스 내의 모든 스레드는 **주소 공간(Address Space)**을 공유함 (코드, 힙, 전역 데이터, 열린 파일 등).
  - **개인 상태 (Private State):** 각 스레드는 고유의 **TCB (Thread Control Block)**를 가짐:
    - **PC:** *이* 스레드가 현재 어디를 실행 중인가?
    - **레지스터 (Registers):** 연산을 위한 범용 레지스터.
    - **스택 (Stack):** 지역 변수, 리턴 주소, 함수 파라미터.
- **문맥 교환 (Context Switching):**
  - **프로세스 스위치:** 비용이 높음(Expensive). 페이지 테이블 교체 필요 (TLB 플러시 발생).
  - **스레드 스위치:** 비용이 저렴함(Cheap). 가상 메모리 매핑이 유지됨. 레지스터와 스택 포인터만 저장/복원하면 됨.

왜 멀티스레딩인가?

1. **병렬성 (Parallelism):** 멀티코어 CPU 활용. 단일 스레드 프로그램은 오직 1개의 코어만 사용 가능.
2. **지연 시간 은폐 (Latency Hiding):** 한 스레드가 I/O(예: 디스크 읽기)로 인해 차단(block)되면, OS 스케줄러가 *같은* 프로그램 내의 다른 스레드로 전환하여 CPU를 계속 활용할 수 있음.

2. 혼돈의 메커니즘: 경쟁 상태 (Race Conditions)

이 부분은 시험의 **코드 분석 문제**에서 가장 빈번하게 출제되는 주제입니다.

어셈블리 레벨의 관점 (The Assembly Level View)

`counter = counter + 1` 과 같은 고수준 코드는 **원자적(atomic)이지 않습니다**. 이는 다음 세 가지 명령어로 컴파일됩니다:

1. **Load:** 메모리(`counter` 의 주소)에서 값을 가져와 레지스터(예: `%eax`)에 넣음.
2. **Update:** 레지스터의 값에 1을 더함.
3. **Store:** 레지스터의 값을 다시 메모리에 저장함.

경쟁 상태 (The Race Condition)

경쟁 상태는 실행 타이밍(스케줄링 운)에 따라 결과가 달라지는 현상을 말합니다.

- *시나리오:* 스레드 A가 50을 로드함. (인터럽트 발생). 스레드 B가 50을 로드하고, 51로 증가시킨 뒤, 51을 저장함. 스레드 A가 다시 실행(레지스터에는 여전히 50이 있음)되어 51로 증가시키고 51을 저장함.
- *결과:* 카운터 값은 52가 되어야 하는데 51이 됨. 업데이트 하나가 "손실(lost)"됨.

핵심 용어 (시험 빈칸 채우기 대비)

- **임계 영역 (Critical Section):** 공유 자원(변수/구조체)에 접근하는 코드 조각으로, 한 번에 하나 이상의 스레드가 실행해서는 안 되는 영역.
- **상호 배제 (Mutual Exclusion):** 한 스레드가 임계 영역에 있으면, 다른 스레드들은 진입이 배제되어야 한다는 속성.
- **비결정적 (Indeterminate):** 실행할 때마다 결과가 달라지는 프로그램; 결정론적(deterministic)이지 않음.
- **원자성 (Atomicity):** "전부 아니면 전무(All or nothing)". 중간 상태에서 중단될 수 없는 연산.

3. Pthreads API (코딩 및 문법)

스레드 생성 (Thread Creation)

```
pthread_create(&thread, NULL, func, arg);
```

- **`func :`** `void *` 를 인자로 받고 `void *` 를 반환하는 함수 포인터.
- **`arg :`** 함수에 전달되는 단일 인자. 여러 값을 전달하려면 `struct` 에 담아서 포인터를 전달해야 함.

스레드 종료 대기 (Thread Completion)

```
pthread_join(thread, &ret_val);
```

- 특정 스레드가 끝날 때까지 기다림.
- **치명적인 시험 함정 (CRITICAL EXAM TRAP):** 스레드의 **스택**에 할당된 변수의 포인터를 반환하지 말 것.
  - *나쁜 예:* `int x; return &x;` (스레드 종료 시 스택이 파괴됨).
  - *좋은 예:* `int *x = malloc(sizeof(int)); return x;` (힙은 유지됨).

4. 동기화 원시/기법 (Synchronization Primitives)

시험에서는 이러한 도구들을 사용하여 로직을 구현하도록 요구합니다.

A. 락 (Locks / Mutex)

상호 배제(Mutual Exclusion)를 제공.

- **사용법:**

```
pthread_mutex_lock(&lock);
// 임계 영역 (공유 상태 업데이트)
pthread_mutex_unlock(&lock);
```

- **상태:** 자유(Free) 또는 점유(Held). 점유된 상태라면 `lock()` 호출 시 호출자는 자유 상태가 될 때까지 차단(block)됨.
- **규칙:** 락은 항상 초기화해야 함. 항상 언락(unlock)해야 함 (에러 발생 경로에서도).

B. 조건 변수 (Condition Variables, CV)

순서(Ordering)를 제공 (상태 변경 대기). 스레드가 특정 조건이 참이 될 때까지 기다려야 할 때 사용 (예: "버퍼가 비어있지 않음").

- **API:**
  - `pthread_cond_wait(cond, mutex)` : 스레드를 잠재우고(sleep) **동시에(atomically)** 락을 해제함. 깨어날 때 락을 다시 획득함.
  - `pthread_cond_signal(cond)` : 대기 중인 스레드 **하나**를 깨움.
  - `pthread_cond_broadcast(cond)` : 대기 중인 **모든** 스레드를 깨움.
- **"Mesa 시맨틱" 패턴 (암기 필수):**
  `if` 문이 아니라 항상 **`while`** 루프를 사용하라.

```
pthread_mutex_lock(&lock);
while (ready == 0) { // 조건 확인
    pthread_cond_wait(&cond, &lock); // 준비 안 됐으면 sleep
}
// 작업 수행
pthread_mutex_unlock(&lock);
```

  - *왜 `while` 인가?* **허위 기상(Spurious Wakeups)** 때문. 시그널이 없어도 OS가 스레드를 깨울 수 있으며, 시그널과 기상 사이에 다른 스레드가 자원을 채갈 수도 있음. 반드시 조건을 다시 확인해야 함.

C. 세마포어 (Semaphores / Dijkstra)

원자적으로 조작되는 정수 값.

- **`sem_wait()` (P):** 값을 감소시킴. 값이 0보다 작으면 블록(block).
- **`sem_post()` (V):** 값을 증가시킴. 대기 중인 스레드 하나를 깨움.
- **용도:** 락(초기값 1)으로 쓰거나 자원 카운터(초기값 N)로 사용 가능.

5. 스레드 구현 모델 (Thread Implementation Models)

OS는 이 스레드들을 어떻게 관리하는가?

1. 커널 수준 스레드 (1:1 모델)

- **구조:** 각 사용자 스레드가 하나의 커널 스레드에 직접 매핑됨.
- **장점:** 진정한 병렬성(다른 코어에서 실행 가능). 하나의 스레드가 블록(I/O)되어도 다른 스레드는 계속 실행.
- **단점:** 높은 오버헤드 (스레드 생성 시 시스템 콜 필요).
- **예시:** Linux, Windows, Mac OS X.

2. 사용자 수준 스레드 (N:1 모델)

- **구조:** 사용자 공간의 라이브러리가 스레드를 관리. OS는 하나의 프로세스로만 인식.
- **장점:** 매우 빠른 스위칭 (함수 호출 수준의 오버헤드). OS 지원 불필요.
- **단점:** 진정한 병렬성 없음 (OS는 프로세스를 1개의 CPU에 스케줄링). **블로킹 문제:** 한 사용자 스레드가 블로킹 시스템 콜(예: read)을 하면 *전체 프로세스*가 블록됨.
- **예시:** 초기 Java ("Green Threads").

3. 다대다 (M:N 모델)

- **구조:** 사용자 스레드를 커널 스레드 풀에 멀티플렉싱함.
- **장점:** 이론적으로 두 모델의 장점을 모두 가짐.
- **단점:** 구현이 복잡함. 스케줄러 활성화(Scheduler Activations, 커널과 사용자 공간 간의 조정)가 필요.

리눅스 구현 세부사항

- 리눅스는 본질적으로 프로세스와 스레드를 구분하지 않음. 둘 다 "태스크(Task)"임 (`struct task_struct`).
- **`clone()` 시스템 콜:** `fork()`와 유사하지만 공유 범위를 세밀하게 조정 가능.
  - `fork()`: 모든 것을 복사 (Copy-on-Write).
  - `pthread_create()`: `clone()`을 사용하여 주소 공간(`CLONE_VM`), 파일 디스크립터(`CLONE_FILES`) 등을 공유.

6. 일반적인 동시성 버그 및 시험 논리

1. 순서 위반 (Ordering Violations)

- *문제:* 스레드 A는 스레드 B가 이미 실행되었을 것(예: 변수 초기화)이라 기대하지만, B가 아직 실행되지 않음.
- *해결:* 조건 변수(CV)나 세마포어를 사용하여 순서를 강제.

2. 원자성 위반 (Atomicity Violations)

- *문제:* 스레드 A가 `if (ptr != NULL)` 을 확인하고 `*ptr` 을 출력하려 함. 그 사이에 스레드 B가 `ptr = NULL` 로 설정함.
- *해결:* 확인(Check)과 사용(Use)을 감싸는 락(Lock)을 추가.

3. 교착 상태 (Deadlock)

- *정의:* 스레드들이 서로를 무한히 기다리며 멈춰 있는 상태.
- *조건:*
  i. **상호 배제 (Mutual Exclusion):** 자원이 독점적으로 점유됨.
  ii. **점유 대기 (Hold and Wait):** 자원을 가진 상태에서 다른 자원을 기다림.
  iii. **비선점 (No Preemption):** 자원을 강제로 뺏을 수 없음.
  iv. **환형 대기 (Circular Wait):** A는 B를, B는 A를 기다림.
- *예방:* 락의 순서 정하기 (Lock Ordering, 항상 락 A를 획득한 후 락 B를 획득).

4. 우선순위 역전 (Priority Inversion)

- *시나리오:* 낮은 우선순위 스레드가 락을 잡고 있음. 높은 우선순위 스레드가 그 락을 기다림. 중간 우선순위 스레드가 낮은 우선순위 스레드를 선점(preempt)함. 결과적으로 높은 우선순위 스레드가 중간 우선순위 스레드를 기다리는 꼴이 됨.
- *해결:* **우선순위 상속 (Priority Inheritance)** (낮은 우선순위 스레드의 우선순위를 임시로 높여줌).

7. 시험 대비 "코드 추론" 체크리스트

"우유 구매(Got Milk)" 문제나 "카페 주문" 스타일의 문제를 분석할 때:

1. **원자성 확인:** 체크(`if milk == 0`)와 행동(`buy milk`) *사이*에 문맥 교환이 일어날 수 있는가? 그렇다면 두 사람이 우유를 살 수 있다.
2. **기아 상태(Starvation) 확인:** VIP가 항상 줄을 먼저 선다면, 일반 손님(Regulars)은 영원히 실행되지 못할 수 있는가? 일반 손님을 처리하기 위한 메커니즘(카운터나 "공정성" 스위치)이 필요함.
3. **스핀 대기(Spin Waiting) 확인:** `while (flag == 0);` 은 나쁜 코드임. CPU를 낭비함. `cond_wait` 를 사용해야 함.
4. **전역 vs 지역 확인:**
    - **스택**에 있는 변수는 해당 스레드에 비공개(private)임.
    - **전역/정적(Global/Static)** 또는 **힙(Heap)**에 있는 변수는 공유됨. 이곳이 경쟁 상태가 발생하는 지점임.