UCLA COMPUTER SCIENCE FIELD NOTES

CS 111

Operating Systems Principles

Kernels, processes, threads, synchronization, scheduling, virtual memory, files, distributed systems, and security

How these notes are written. I wrote each chapter the way I wish the material had been explained the first time: the real idea, the point where I usually get stuck, the simplest mental model that still stays honest, and a worked decision instead of a polished answer appearing from nowhere. Every chapter closes with practice and a fully written solution. If a sentence sounds impressive but does not help me solve or explain something, it does not belong here.

Daniel Mastick

Curriculum map, working notes, practice, and solutions

Course map

Week	Core thread	What should be solid by the end
1	**Kernel structure, boot, and system calls**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
2	**Processes, threads, and context**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
3	**Synchronization and deadlock**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
4	**CPU scheduling**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
5	**Virtual memory**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
6	**File systems and storage**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
7	**I/O, devices, and interrupts**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
8	**Distributed systems, protection, and security**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
9	**Integration workshop**	Connect at least three chapters in one end-to-end problem and explain where their contracts meet.
10	**Synthesis and project review**	Audit assumptions and explain a complete solution from interface to failure handling.

Scope anchor: https://catalog.registrar.ucla.edu/course/2025/COMSCI111. The sequence is aligned to the official catalog description and expanded into a practical ten-week study plan.

How I would use this packet

Treat each numbered section as one chapter. Read the formal explanation, pause at the student interpretation, and see whether the easy version still says the same thing. Rework the example without looking, then cover the solution in the chapter practice box. The cumulative set at the end is deliberately mixed: knowledge becomes durable when I have to choose the tool instead of being told which chapter I am in.

Contents

CHAPTER 1 Kernel structure, boot, and system calls

The idea

The OS virtualizes hardware and arbitrates resources through protected kernel code. Boot establishes execution, memory, devices, and the first process. System calls cross a privilege boundary through a controlled trap and validated arguments.

Rebuild the chapter from first principles

The claim I need to own is this: The OS virtualizes hardware and arbitrates resources through protected kernel code. Boot establishes execution, memory, devices, and the first process. System calls cross a privilege boundary through a controlled trap and validated arguments. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of kernel structure, boot, and system calls. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach kernel structure, boot, and system calls on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from the course foundations to processes, threads, and context. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** The OS virtualizes hardware and arbitrates resources through protected kernel code.
- **Mechanism.** Boot establishes execution, memory, devices, and the first process.
- **Boundary.** System calls cross a privilege boundary through a controlled trap and validated arguments.
- **Decision test.** I should be able to say when kernel structure, boot, and system calls is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I view the kernel as a concurrent server for untrusted processes.

The easy version

The OS virtualizes hardware and arbitrates resources through protected kernel code.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Why copy user pointers carefully? The conclusion is A user address may be invalid or change concurrently. The kernel must validate access, handle faults safely, and avoid trusting user memory after a check without a protected copy.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Why copy user pointers carefully?
Solution. A user address may be invalid or change concurrently. The kernel must validate access, handle faults safely, and avoid trusting user memory after a check without a protected copy.

Practice 2. Give a short oral explanation of kernel structure, boot, and system calls that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct kernel structure, boot, and system calls from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a kernel structure, boot, and system calls problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 2 Processes, threads, and context

The idea

A process owns an address space and resources; threads share them while keeping separate registers and stacks. Creation, exit, wait, signals, and context switches define lifecycle and coordination.

Rebuild the chapter from first principles

The claim I need to own is this: A process owns an address space and resources; threads share them while keeping separate registers and stacks. Creation, exit, wait, signals, and context switches define lifecycle and coordination. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the

definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of processes, threads, and context. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach processes, threads, and context on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from kernel structure, boot, and system calls to synchronization and deadlock. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method

valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** A process owns an address space and resources; threads share them while keeping separate registers and stacks.
- **Mechanism.** Creation, exit, wait, signals, and context switches define lifecycle and coordination.
- **Boundary.** The useful test is whether I can apply processes, threads, and context to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when processes, threads, and context is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I list what is shared and what is private before reasoning about threads.

The easy version

A process owns an address space and resources; threads share them while keeping separate registers and stacks.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: What changes in a context switch? The conclusion is Save the current thread's registers and scheduling state, choose another runnable thread, switch address-space context if needed, restore registers, and resume.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. What changes in a context switch?
Solution. Save the current thread's registers and scheduling state, choose another runnable thread, switch address-space context if needed, restore registers, and resume.

Practice 2. Give a short oral explanation of processes, threads, and context that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct processes, threads, and context from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a processes, threads, and context problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 3 Synchronization and deadlock

The idea

Atomicity, mutual exclusion, visibility, and ordering protect invariants under interleaving. Locks, condition variables, semaphores, monitors, and atomic operations serve different protocols. Deadlock needs mutual exclusion, hold-and-wait, no preemption, and a cycle.

Rebuild the chapter from first principles

The claim I need to own is this: Atomicity, mutual exclusion, visibility, and ordering protect invariants under interleaving. Locks, condition variables, semaphores, monitors, and atomic operations serve different protocols. Deadlock needs mutual exclusion, hold-and-wait, no preemption, and a cycle. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of synchronization and deadlock. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach synchronization and deadlock on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from processes, threads, and context to cpu scheduling. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Atomicity, mutual exclusion, visibility, and ordering protect invariants under interleaving.
- **Mechanism.** Locks, condition variables, semaphores, monitors, and atomic operations serve different protocols.
- **Boundary.** Deadlock needs mutual exclusion, hold-and-wait, no preemption, and a cycle.
- **Decision test.** I should be able to say when synchronization and deadlock is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

The lock protects an invariant, not a line of code; I write that invariant beside it.

The easy version

Atomicity, mutual exclusion, visibility, and ordering protect invariants under interleaving.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Why use a while loop around condition wait? The conclusion is Wakeups can be spurious or another thread can consume the condition first. Recheck the predicate while holding the lock before proceeding.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Why use a while loop around condition wait?
Solution. Wakeups can be spurious or another thread can consume the condition first. Recheck the predicate while holding the lock before proceeding.

Practice 2. Give a short oral explanation of synchronization and deadlock that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct synchronization and deadlock from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a synchronization and deadlock problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 4 CPU scheduling

The idea

Schedulers balance response time, throughput, fairness, deadlines, and overhead. FCFS, shortest-job, round-robin, priority, multilevel feedback, and real-time policies encode different assumptions; starvation and priority inversion require explicit remedies.

Rebuild the chapter from first principles

The claim I need to own is this: Schedulers balance response time, throughput, fairness, deadlines, and overhead. FCFS, shortest-job, round-robin, priority, multilevel feedback, and real-time policies encode different assumptions; starvation and priority inversion require explicit remedies. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of cpu scheduling. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach cpu scheduling on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from synchronization and deadlock to virtual memory. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Schedulers balance response time, throughput, fairness, deadlines, and overhead.
- **Mechanism.** FCFS, shortest-job, round-robin, priority, multilevel feedback, and real-time policies encode different assumptions; starvation and priority inversion require explicit remedies.
- **Boundary.** The useful test is whether I can apply cpu scheduling to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when cpu scheduling is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

My shorthand for this chapter is: my shorthand for this chapter is: there is no universally best scheduler because the word best hides the workload and metric.

The easy version

Schedulers balance response time, throughput, fairness, deadlines, and overhead.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: How does aging prevent starvation? The conclusion is Increase a waiting task's effective priority over time so continued postponement eventually makes it schedulable.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. How does aging prevent starvation?
Solution. Increase a waiting task's effective priority over time so continued postponement eventually makes it schedulable.

Practice 2. Give a short oral explanation of cpu scheduling that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct cpu scheduling from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a cpu scheduling problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 5 Virtual memory

The idea

Paging maps virtual pages to frames, enabling isolation, sparse spaces, sharing, and demand loading. TLBs cache translations; replacement and working sets govern page faults; copy-on-write delays duplication.

Rebuild the chapter from first principles

The claim I need to own is this: Paging maps virtual pages to frames, enabling isolation, sparse spaces, sharing, and demand loading. TLBs cache translations; replacement and working sets govern page faults; copy-on-write delays duplication. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of virtual memory. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach virtual memory on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from cpu scheduling to file systems and storage. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Paging maps virtual pages to frames, enabling isolation, sparse spaces, sharing, and demand loading.
- **Mechanism.** TLBs cache translations; replacement and working sets govern page faults; copy-on-write delays duplication.
- **Boundary.** The useful test is whether I can apply virtual memory to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when virtual memory is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I separate address translation from page replacement: one locates data, the other decides what remains resident.

The easy version

Paging maps virtual pages to frames, enabling isolation, sparse spaces, sharing, and demand loading.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Why can too many processes cause thrashing? The conclusion is Their combined working sets exceed physical memory, so useful pages are repeatedly evicted and faulted back. CPU utilization falls while paging I/O dominates.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Why can too many processes cause thrashing?
Solution. Their combined working sets exceed physical memory, so useful pages are repeatedly evicted and faulted back. CPU utilization falls while paging I/O dominates.

Practice 2. Give a short oral explanation of virtual memory that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct virtual memory from a blank page. Include the central defi-

nition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a virtual memory problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 6 File systems and storage

The idea

Files name byte sequences or records; directories map names; inodes or equivalent metadata locate blocks. Allocation, free-space tracking, caching, journaling, copy-on-write, and crash ordering determine performance and robustness.

Rebuild the chapter from first principles

The claim I need to own is this: Files name byte sequences or records; directories map names; inodes or equivalent metadata locate blocks. Allocation, free-space tracking, caching, journaling, copy-on-write, and crash ordering determine performance and robustness. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of file systems and storage. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach file systems and storage on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from virtual memory to i/o, devices, and interrupts. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Files name byte sequences or records; directories map names; inodes or equivalent metadata locate blocks.
- **Mechanism.** Allocation, free-space tracking, caching, journaling, copy-on-write, and crash ordering determine performance and robustness.
- **Boundary.** The useful test is whether I can apply file systems and storage to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when file systems and storage is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I trace both namespace lookup and block lookup; a path and its file contents are different structures.

The easy version

Files name byte sequences or records; directories map names; inodes or equivalent metadata locate blocks.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Why journal metadata before applying it? The conclusion is A committed log record lets recovery replay an intended atomic metadata update after a crash, avoiding a partially updated on-disk structure.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Why journal metadata before applying it?
Solution. A committed log record lets recovery replay an intended atomic metadata update after a crash, avoiding a partially updated on-disk structure.

Practice 2. Give a short oral explanation of file systems and storage that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct file systems and storage from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a file systems and storage problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 7 I/O, devices, and interrupts

The idea

Drivers translate generic OS operations into device commands. Interrupts signal asynchronous events; DMA transfers data without per-byte CPU copying. Buffering, queues, polling, and back-pressure connect device rate to software rate.

Rebuild the chapter from first principles

The claim I need to own is this: Drivers translate generic OS operations into device commands. Interrupts signal asynchronous events; DMA transfers data without per-byte CPU copying. Buffering, queues, polling, and backpressure connect device rate to software rate. I do not count that as learned just because the sentence looks familiar. I should be able to name the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of i/o, devices, and interrupts. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach i/o, devices, and interrupts on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from file systems and storage to distributed systems, protection, and security. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** Drivers translate generic OS operations into device commands.
- **Mechanism.** Interrupts signal asynchronous events; DMA transfers data without per-byte CPU copying.
- **Boundary.** Buffering, queues, polling, and backpressure connect device rate to software rate.
- **Decision test.** I should be able to say when i/o, devices, and interrupts is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I ask which side owns each buffer and what happens when the producer outruns the consumer.

The easy version

Drivers translate generic OS operations into device commands.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: Compare polling and interrupts. The conclusion is Polling repeatedly checks state and can waste CPU but offers predictable control at high rates. Interrupts avoid idle checks but add handling and context overhead.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. Compare polling and interrupts.
Solution. Polling repeatedly checks state and can waste CPU but offers predictable control at high rates. Interrupts avoid idle checks but add handling and context overhead.

Practice 2. Give a short oral explanation of i/o, devices, and interrupts that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.
Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct i/o, devices, and interrupts from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a i/o, devices, and interrupts problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

CHAPTER 8 Distributed systems, protection, and security

The idea

RPC makes remote calls resemble local ones while retaining partial failure, delay, duplication, and versioning. Distributed files and transactions need naming, consistency, idempotency, and recovery. Protection uses identities, capabilities or ACLs, isolation, least privilege, and audit.

Rebuild the chapter from first principles

The claim I need to own is this: RPC makes remote calls resemble local ones while retaining partial failure, delay, duplication, and versioning. Distributed files and transactions need naming, consistency, idempotency, and recovery. Protection uses identities, capabilities or ACLs, isolation, least privilege, and audit. I do not count that as learned just because the sentence looks familiar. I should be able to name

the objects, state what changes and what stays fixed, recover the rule from the definitions, and explain why the result is allowed.

The move

My working sequence

1. **Translate.** Rewrite the prompt in the language of distributed systems, protection, and security. List the known quantities, the unknown, the units or types, and any hidden constraint.
2. **Choose.** State the theorem, model, algorithm, accounting rule, or physical law before substituting values. This is where I make the assumption visible.
3. **Execute.** Work one justified step at a time and keep intermediate state readable enough that I can audit it later.
4. **Verify.** Check a boundary case, sign, unit, invariant, order of magnitude, or alternative derivation. Then translate the result back into an ordinary sentence.

Details I would refuse to skip

- **Definitions:** I should be able to define every technical noun in the chapter without using that same noun in the definition.
- **Assumptions:** I write the condition that makes the method legal beside the method itself. A correct procedure under the wrong assumptions is still a wrong answer.
- **Representation:** I move freely among words, equations or code, a diagram, and a small numerical example. If one representation disagrees with another, that disagreement is useful evidence.
- **Failure modes:** I keep the nearest counterexample in mind: the boundary where the rule changes, the invariant breaks, or the model stops matching reality.
- **Communication:** My final answer names what the result means and what it does not establish. I do not let notation do the explanatory work for me.

Sanity check

Closed-note check. Can I teach distributed systems, protection, and security on a blank page, derive or reconstruct its main rule, work a fresh example, and diagnose a deliberately broken solution? If I can only recognize the finished work, I am not done.

How this chapter connects

I read this chapter as the bridge from i/o, devices, and interrupts to the cumulative course model. The earlier material supplies the language and constraints; this chapter adds a new reasoning move; the later material asks me to use that move without being told its name. That connection matters because exams and real projects mix chapters on purpose.

What I need to be able to do

For this chapter, I should be able to define the objects and state involved, state the governing invariant or contract, trace a small example without hand-waving, and explain which assumption makes the method valid. I should also be able to compare this model with the nearest alternative and say what failure would make me switch approaches.

Deep notes

- **Model.** RPC makes remote calls resemble local ones while retaining partial failure, delay, duplication, and versioning.
- **Mechanism.** Distributed files and transactions need naming, consistency, idempotency, and recovery.
- **Boundary.** Protection uses identities, capabilities or ACLs, isolation, least privilege, and audit.
- **Decision test.** I should be able to say when distributed systems, protection, and security is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

My shorthand for this chapter is: my shorthand for this chapter is: the network creates outcomes a local call does not have: maybe the request happened even though the reply vanished.

The easy version

RPC makes remote calls resemble local ones while retaining partial failure, delay, duplication, and versioning.

Worked example

A useful way to work this chapter is to make the smallest concrete instance, write the state or contract explicitly, and then scale the reasoning only after that instance behaves correctly. The worked question below makes that reasoning concrete: How should a payment RPC handle retry? The conclusion is Attach a unique idempotency key, durably record the result, and return the same result for duplicates. A timeout alone cannot reveal whether the first request committed.

Common miss

The common mistake is to memorize the visible procedure while dropping its precondition. I write the assumption beside the method and test one boundary case before trusting the result.

Solved chapter practice

Practice. How should a payment RPC handle retry?
Solution. Attach a unique idempotency key, durably record the result, and return the same result for duplicates. A timeout alone cannot reveal whether the first request committed.

Practice 2. Give a short oral explanation of distributed systems, protection, and security that includes its model, one assumption, one failure mode, and one comparison with a neighboring method.

Solution. A strong answer begins with the model in the blue box, names the assumption that licenses the method, uses the common miss above as the failure mode, and chooses the comparison according to the decision test in the deep notes. The goal is not one memorized sentence; it is a four-part explanation I can reproduce on a new example.

Mastery practice A. Reconstruct distributed systems, protection, and security from a blank page. Include the central definition, the governing rule, one required assumption, and one boundary case.
Solution. Start from the blue idea box, but rewrite it in my own words. Identify the objects and state, name the rule before using it, copy the relevant assumption from the details audit, and choose a boundary where that assumption becomes equality or fails. A complete solution explains why each part belongs; a list of vocabulary is not enough.

Mastery practice B. A classmate gets a polished-looking answer to a distributed systems, protection, and security problem but never states a model or checks the result. Write the shortest useful review of that solution.
Solution. I would ask four concrete questions: What exactly are the inputs and desired output? Which rule licenses the main step? Which assumption could invalidate it? Which independent check supports the conclusion? If the answer cannot survive those four questions, I treat the numerical or verbal result as unverified.

Final study check

I should be able to close this packet and reconstruct the core models, not merely recognize their names. My final check is to explain one complete problem aloud: what the inputs mean, which invariant or contract controls the work, why the method is legal, what it costs, how it can fail, and what evidence would convince another engineer.