UCLA COMPUTER SCIENCE FIELD NOTES

CS 33

Introduction to Computer Organization

Bits, x86-64, procedures, linking, caches, virtual memory, processes, and systems behavior

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	**Bits, integers, and floating point**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
2	**x86-64 instructions and addressing**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
3	**Procedures, stack frames, and ABI**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
4	**Linking, relocation, and loading**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
5	**Caches and locality**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
6	**Virtual memory and address translation**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
7	**Processes, exceptions, and signals**	Explain the model, apply it to a concrete case, and defend the relevant correctness or design decision.
8	**I/O, concurrency, and performance reasoning**	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/2024/COMSCI33. 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 Bits, integers, and floating point

The idea

Unsigned and two's-complement representations make arithmetic a finite modular system. Floating point stores sign, exponent, and significand, so many decimal values are approximate and exceptional values need explicit rules.

Rebuild the chapter from first principles

The claim I need to own is this: Unsigned and two's-complement representations make arithmetic a finite modular system. Floating point stores sign, exponent, and significand, so many decimal values are approximate and exceptional values need explicit rules. 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 bits, integers, and floating point. 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 bits, integers, and floating point 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 x86-64 instructions and addressing. 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.** Unsigned and two's-complement representations make arithmetic a finite modular system.
- **Mechanism.** Floating point stores sign, exponent, and significand, so many decimal values are approximate and exceptional values need explicit rules.
- **Boundary.** The useful test is whether I can apply bits, integers, and floating point to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when bits, integers, and floating point is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I convert through powers of two and track bit width; pretending an int is an unbounded mathematical integer hides overflow.

The easy version

Unsigned and two's-complement representations make arithmetic a finite modular system.

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: Represent -13 in 8-bit two's complement and explain overflow. The conclusion is 13 is 00001101; invert and add one to get 11110011. Overflow occurs when a signed result leaves -128 through 127, detectable when same-sign operands produce

an opposite-sign result.

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. Represent -13 in 8-bit two's complement and explain overflow.
Solution. 13 is 00001101; invert and add one to get 11110011. Overflow occurs when a signed result leaves -128 through 127, detectable when same-sign operands produce an opposite-sign result.

Practice 2. Give a short oral explanation of bits, integers, and floating point 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 bits, integers, and floating point 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 bits, integers, and floating point 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 x86-64 instructions and addressing

The idea

Assembly exposes data movement, arithmetic, condition codes, jumps, and effective addresses. The instruction pointer and registers form visible machine state; memory operands compute base plus index times scale plus displacement.

Rebuild the chapter from first principles

The claim I need to own is this: Assembly exposes data movement, arithmetic, condition codes, jumps, and effective addresses. The instruction pointer and registers form visible machine state; memory operands compute base plus index times scale plus displacement. 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 x86-64 instructions and addressing. 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 x86-64 instructions and addressing 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 bits, integers, and floating point to procedures, stack frames, and abi. 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.** Assembly exposes data movement, arithmetic, condition codes, jumps, and effective addresses.
- **Mechanism.** The instruction pointer and registers form visible machine state; memory operands compute base plus index times scale plus displacement.
- **Boundary.** The useful test is whether I can apply x86-64 instructions and addressing to a new case and explain the assumption that makes the answer valid.
- **Decision test.** I should be able to say when x86-64 instructions and addressing is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I translate one instruction at a time into a state change instead of reading assembly like compressed C++.

The easy version

Assembly exposes data movement, arithmetic, condition codes, jumps, and effective addresses.

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 address does 8(%rbx,%rcx,4) denote? The conclusion is It is rbx + 4*rcx + 8. The syntax computes an address; the instruction determines whether that address is read, written, or formed with lea.

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 address does 8(%rbx,%rcx,4) denote?
Solution. It is rbx + 4*rcx + 8. The syntax computes an address; the instruction determines whether that address is read, written, or formed with lea.

Practice 2. Give a short oral explanation of x86-64 instructions and addressing 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 x86-64 instructions and addressing 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 x86-64 instructions and addressing 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 Procedures, stack frames, and ABI

The idea

The calling convention assigns argument, return, caller-saved, and callee-saved registers. Calls push a return address; stack frames hold spills, locals, and saved state while maintaining alignment.

Rebuild the chapter from first principles

The claim I need to own is this: The calling convention assigns argument, return, caller-saved, and callee-saved registers. Calls push a return address; stack frames hold spills, locals, and saved state while maintaining alignment. 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 procedures, stack frames, and abi. 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 procedures, stack frames, and abi 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 x86-64 instructions and addressing to linking, relocation, and loading. 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 calling convention assigns argument, return, caller-saved, and callee-saved registers.
- **Mechanism.** Calls push a return address; stack frames hold spills, locals, and saved state while maintaining alignment.
- **Boundary.** The useful test is whether I can apply procedures, stack frames, and abi to a new case and explain the assumption that makes the answer valid.

- **Decision test.** I should be able to say when procedures, stack frames, and abi is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I draw the stack at the call boundary and label who promises to preserve each register.

The easy version

The calling convention assigns argument, return, caller-saved, and callee-saved registers.

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 must a callee restore callee-saved registers? The conclusion is The ABI lets callers assume those values survive calls. A callee may use them only after saving their old values and restoring them before return.

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 must a callee restore callee-saved registers?
Solution. The ABI lets callers assume those values survive calls. A callee may use them only after saving their old values and restoring them before return.

Practice 2. Give a short oral explanation of procedures, stack frames, and abi 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 procedures, stack frames, and abi 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 procedures, stack frames, and abi 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 Linking, relocation, and loading

The idea

Object files contain code, data, symbols, and relocation records. The linker resolves references and lays out an executable; the loader maps segments and begins execution. Static and dynamic linking trade size, sharing, deployment, and indirection.

Rebuild the chapter from first principles

The claim I need to own is this: Object files contain code, data, symbols, and relocation records. The linker resolves references and lays out an executable; the loader maps segments and begins execution. Static and dynamic linking trade size, sharing, deployment, and indirection. 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 linking, relocation, and loading. 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 linking, relocation, and loading 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 procedures, stack frames, and abi to caches and locality. 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.** Object files contain code, data, symbols, and relocation records.
- **Mechanism.** The linker resolves references and lays out an executable; the loader maps segments and begins execution.
- **Boundary.** Static and dynamic linking trade size, sharing, deployment, and indirection.
- **Decision test.** I should be able to say when linking, relocation, and loading is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

Undefined reference is a link-time name problem, not a syntax problem; I ask which object should define the missing symbol.

The easy version

Object files contain code, data, symbols, and relocation records.

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: Distinguish declaration, definition, and

relocation. The conclusion is A declaration states a name and type; a definition allocates code or storage. Relocation records tell the linker where final addresses must be patched once layout is known.

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. Distinguish declaration, definition, and relocation.
Solution. A declaration states a name and type; a definition allocates code or storage. Relocation records tell the linker where final addresses must be patched once layout is known.

Practice 2. Give a short oral explanation of linking, relocation, and loading 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 linking, relocation, and loading 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 linking, relocation, and loading 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 Caches and locality

The idea

Caches exploit temporal and spatial locality using blocks organized into sets with tags, valid bits, and replacement state. Address bits divide into offset, index, and tag. Misses may be compulsory, conflict, or capacity misses.

Rebuild the chapter from first principles

The claim I need to own is this: Caches exploit temporal and spatial locality using blocks organized into sets with tags, valid bits, and replacement state. Address bits divide into offset, index, and tag. Misses may be compulsory, conflict, or capacity misses. 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 caches and locality. 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 caches and locality 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 linking, relocation, and loading to virtual memory and address translation. 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.** Caches exploit temporal and spatial locality using blocks organized into sets with tags, valid bits, and replacement state.
- **Mechanism.** Address bits divide into offset, index, and tag.
- **Boundary.** Misses may be compulsory, conflict, or capacity misses.
- **Decision test.** I should be able to say when caches and locality is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I compute the address split before simulating accesses; otherwise cache questions become guesswork.

The easy version

Caches exploit temporal and spatial locality using blocks organized into sets with tags, valid bits, and replacement state.

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: A 4 KiB direct-mapped cache has 64-byte blocks. Find offset and index bits. The conclusion is 64 bytes requires 6 offset bits. There are 4096/64=64 sets, requiring 6 index bits; remaining address bits are tag.

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. A 4 KiB direct-mapped cache has 64-byte blocks. Find offset and index bits.
Solution. 64 bytes requires 6 offset bits. There are 4096/64=64 sets, requiring 6 index bits; remaining address bits are tag.

Practice 2. Give a short oral explanation of caches and locality 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 caches and locality 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 caches and locality 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 Virtual memory and address translation

The idea

Virtual addresses isolate processes and let the OS manage sparse address spaces. Page tables map virtual pages to physical frames; a TLB caches translations; faults transfer control to the kernel when a mapping is absent or protected.

Rebuild the chapter from first principles

The claim I need to own is this: Virtual addresses isolate processes and let the OS manage sparse address spaces. Page tables map virtual pages to physical frames; a TLB caches translations; faults transfer control to the kernel when a mapping is absent or protected. 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 and address translation. 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 and address translation 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 caches and locality to processes, exceptions, and signals. 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.** Virtual addresses isolate processes and let the OS manage sparse address spaces.
- **Mechanism.** Page tables map virtual pages to physical frames; a TLB caches translations; faults transfer control to the kernel when a mapping is absent or protected.
- **Boundary.** The useful test is whether I can apply virtual memory and address translation to a new case and explain the assumption that makes the answer valid.

- **Decision test.** I should be able to say when virtual memory and address translation is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I separate cache lookup from address translation. They cooperate but solve different problems.

The easy version

Virtual addresses isolate processes and let the OS manage sparse address spaces.

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: For 4 KiB pages, split a 32-bit virtual address. The conclusion is The page offset is 12 bits because 4096=2^12. The remaining 20 bits form the virtual page number before any multilevel page-table split.

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. For 4 KiB pages, split a 32-bit virtual address.
Solution. The page offset is 12 bits because 4096=2^12. The remaining 20 bits form the virtual page number before any multilevel page-table split.

Practice 2. Give a short oral explanation of virtual memory and address translation 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 and address translation 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 virtual memory and address translation 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 Processes, exceptions, and signals

The idea

Exceptions are controlled transfers caused by faults, traps, interrupts, or aborts. A process combines an address space with execution state and OS resources. Context switches save one execution and restore another; signals provide asynchronous user-space notification.

Rebuild the chapter from first principles

The claim I need to own is this: Exceptions are controlled transfers caused by faults, traps, interrupts, or aborts. A process combines an address space with execution state and OS resources. Context switches save one execution and restore another; signals provide asynchronous user-space notification. 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, exceptions, and signals. 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, exceptions, and signals 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 and address translation to i/o, concurrency, and performance reasoning. 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.** Exceptions are controlled transfers caused by faults, traps, interrupts, or aborts.
- **Mechanism.** A process combines an address space with execution state and OS resources.
- **Boundary.** Context switches save one execution and restore another; signals provide asynchronous user-space notification.
- **Decision test.** I should be able to say when processes, exceptions, and signals 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: a process is not just a running program file; it is a protected live execution with kernel-managed state.

The easy version

Exceptions are controlled transfers caused by faults, traps, interrupts, or aborts.

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: Differentiate a system call from a hardware interrupt. The conclusion is A system call is a deliberate synchronous trap from a process into the kernel. A hardware interrupt is asynchronous to the current instruction stream and comes from a device or timer.

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. Differentiate a system call from a hardware interrupt.
Solution. A system call is a deliberate synchronous trap from a process into the kernel. A hardware interrupt is asynchronous to the current instruction stream and comes from a device or timer.

Practice 2. Give a short oral explanation of processes, exceptions, and signals 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, exceptions, and signals 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, exceptions, and signals 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 I/O, concurrency, and performance reasoning

The idea

File descriptors give a uniform interface to files, pipes, sockets, and devices. Partial reads, buffering, and interleaving matter. Performance combines instruction count, cycles per instruction, clock rate, locality, and parallel overlap rather than one isolated number.

Rebuild the chapter from first principles

The claim I need to own is this: File descriptors give a uniform interface to files, pipes, sockets, and devices. Partial reads, buffering, and interleaving matter. Performance combines instruction count, cycles per instruction, clock rate, locality, and parallel overlap rather than one isolated number. 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, concurrency, and performance reasoning. 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, concurrency, and performance reasoning 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, exceptions, and signals 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.** File descriptors give a uniform interface to files, pipes, sockets, and devices.
- **Mechanism.** Partial reads, buffering, and interleaving matter.
- **Boundary.** Performance combines instruction count, cycles per instruction, clock rate, locality, and parallel overlap rather than one isolated number.
- **Decision test.** I should be able to say when i/o, concurrency, and performance reasoning is the right model, when its assumptions fail, and what evidence would change my choice.

How I actually think about it

I assume I/O can be partial and concurrency can reorder events unless an interface promises otherwise.

The easy version

File descriptors give a uniform interface to files, pipes, sockets, and devices.

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 is one read call not guaranteed to fill a buffer from a pipe? The conclusion is A pipe may currently contain fewer bytes than requested. Correct code loops until its protocol is complete, EOF occurs, or an error is handled.

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 is one read call not guaranteed to fill a buffer from a pipe?
Solution. A pipe may currently contain fewer bytes than requested. Correct code loops until its protocol is complete, EOF occurs, or an error is handled.

Practice 2. Give a short oral explanation of i/o, concurrency, and performance reasoning 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, concurrency, and performance reasoning 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, concurrency, and performance reasoning 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.