Debugging (2/2)

Martin Kellogg

Reading Quiz: debugging 2

Q1: What is the inspiration for the name of the “wynot” tool?

A. The “wynot” tool helps answer the question “why not?”
B. It is named after the Pokémon “Wynaut”
C. “wynot” is an abbreviation for “**W**orked **Y**esterday, **NO**t **T**oday”
D. None of these are the inspiration for the tool’s name

Q2: Which web browser was an experimental subject in the article?

A. Mozilla/Netscape
B. Chromium/Google Chrome
C. Internet Explorer

Reading Quiz: debugging 2

Q1: What is the inspiration for the name of the “wynot” tool?

A. The “wynot” tool helps answer the question “why not?”

B. It is named after the Pokémon “Wynaut”

C. “wynot” is an abbreviation for “**W**orked **Y**esterday, **NO**t **T**oday”

D. None of these are the inspiration for the tool’s name

Q2: Which web browser was an experimental subject in the article?

A. Mozilla/Netscape

B. Chromium/Google Chrome

C. Internet Explorer

Reading Quiz: debugging 2

Q1: What is the inspiration for the name of the “wynot” tool?

A. The “wynot” tool helps answer the question “why not?”

B. It is named after the Pokémon “Wynaut”

C. “wynot” is an abbreviation for “**W**orked **Y**esterday, **NO**t **T**oday”

D. None of these are the inspiration for the tool’s name

Q2: Which web browser was an experimental subject in the article?

A. Mozilla/Netscape

B. Chromium/Google Chrome

C. Internet Explorer

Debugging strategies

Debugging strategies

- “printf” debugging: using print statements to find a bug
  - and its larger-scale cousin: **logging**

Debugging strategies

- “printf” debugging: using print statements to find a bug
  - and its larger-scale cousin: **logging**
- delta debugging
  - a **formalization** of the scientific approach to debugging

Debugging strategies

- “printf” debugging: using print statements to find a bug
  - and its larger-scale cousin: **logging**
- delta debugging
  - a **formalization** of the scientific approach to debugging
- debuggers: **inspecting program state** while it is running
  - we’ll talk a little about how they work

Debugging (Part 2/2)

Today's agenda:

- Debugging
  - **printf debugging and logging**
  - delta debugging
  - debuggers

“printf” debugging

- probably your most common debugging strategy already!

“printf” debugging

- probably your most common debugging strategy already!
- key idea: **instrument** the program so that it prints the values of key variables at a particular point

“printf” debugging

- probably your most common debugging strategy already!
- key idea: **instrument** the program so that it prints the values of key variables at a particular point
- advantages:
  - easy and natural

“printf” debugging

- probably your most common debugging strategy already!
- key idea: **instrument** the program so that it prints the values of key variables at a particular point
- advantages:
  - easy and natural
- disadvantages:
  - must recompile, rerun program each time you want to test something else
  - sometimes considered “unprofessional”

“printf” debugging

- probably your most common debugging strategy already!
- key idea: **instrument** the program so that it prints the values of key variables at a part
- advantages:
  - easy and natural
- disadvantages:
  - must recompile, re
    something else
  - sometimes considered “unprofessional”

This is a **misconception**: professional engineers commonly use printf debugging. But printf debugging should be just one tool in your toolbox of debugging strategies!

Logging

Definition: ***logging*** is the process of recording information about the program's internal state as it runs via a printf-like interface

Logging

Definition: ***logging*** is the process of recording information about the program's internal state as it runs via a printf-like interface

- logging is a key technology for **monitoring** modern systems
  - e.g., via tools like Log4j, slf4j, etc.

Logging

Definition: ***logging*** is the process of recording information about the program's internal state as it runs via a printf-like interface

- logging is a key technology for **monitoring** modern systems
  - e.g., via tools like Log4j, slf4j, etc.
- logs also play a major role in debugging **large-scale failures** of important distributed systems

Logging

Definition: ***logging*** is the process of recording information about the program's internal state as it runs via a printf-like interface

- logging is a key technology for **monitoring** modern systems
  - e.g., via tools like Log4j, slf4j, etc.
- logs also play a major role in debugging **large-scale failures** of important distributed systems
  - we'll discuss this more when we talk about **post-mortems** in our DevOps lectures, near the end of the semester

Logging: levels

Typical example of a (Java) logging statement:

```
log.debug("myVariable=%s", myVariable);
```

Logging: levels

Typical example of a (Java) logging statement:

```
log.debug("myVariable=%s", myVariable);
```

the log itself is usually a static field; the logging framework instantiates it, etc.

Logging: levels

Typical example of a (Java) logging statement:

```
log.debug("myVariable=%s", myVariable);
```

"debug" means if debug-level logging isn't enabled in the framework, this becomes a no-op

Logging: levels

Typical example of a (Java) logging statement:

```
log.debug("myVariable=%s", myVariable);
```

"debug" means if debug-level logging isn't enabled in the framework, this becomes a no-op

levels:
error ⊆ warning ⊆ info ⊆ debug

developer chooses one level, all lower level messages are also logged

Logging: levels

Typical example of a (Java) logging statement:

```
log.debug("myVariable=%s", myVariable);
```

printf-like syntax isn't just for show: goal here is lazy evaluation, so that if debug logging isn't enabled, this string is never constructed

Logging: levels

Typical example of a (Java) logging statement:

```
log.debug("myVariable=%s", myVariable);
```

arguments to printf passed by reference, so if debug-level logging is off, this argument's toString() method is never called

Logging: advice

Logging: advice

- **Do** log lots of information at debug or info level, so that if something is wrong with your service you can quickly get lots of information that you can use to debug it.

Logging: advice

- **Do** log lots of information at debug or info level, so that if something is wrong with your service you can quickly get lots of information that you can use to debug it.
- **Don't** log sensitive data (e.g., credit card numbers in plaintext!)
  - this is a surprisingly common and important problem - developers have a tendency to log anything that might be useful when debugging a failure later!

Debugging (Part 2/2)

Today's agenda:

- Debugging
  - printf debugging and logging
  - **delta debugging**
  - debuggers

Delta debugging: summary

Delta debugging: summary

- ***Delta debugging*** is an automated debugging approach that finds a minimal “interesting” subset of a given set.

Delta debugging: summary

- ***Delta debugging*** is an automated debugging approach that finds a minimal “interesting” subset of a given set.
- Delta debugging is based on **divide-and-conquer** and relies heavily on critical assumptions (**monotonicity**, **unambiguity**, and **consistency**).

Delta debugging: summary

- ***Delta debugging*** is an automated debugging approach that finds a minimal “interesting” subset of a given set.
- Delta debugging is based on **divide-and-conquer** and relies heavily on critical assumptions (**monotonicity**, **unambiguity**, and **consistency**).
- It can be used to find which code changes cause a bug, to minimize failure-inducing inputs, and even to find harmful thread schedules.

Delta debugging: motivation

- Three Problems: One Common Approach
  - Simplifying Failure-Inducing Input
  - Isolating Failure-Inducing Thread Schedules
  - Identifying Failure-Inducing Code Changes

Delta debugging: motivation: inputs

- Having a **test input** may not be enough

Delta debugging: motivation: inputs

- Having a **test input** may not be enough
  - Even if you know the suspicious code, the input may be **too large** to step through

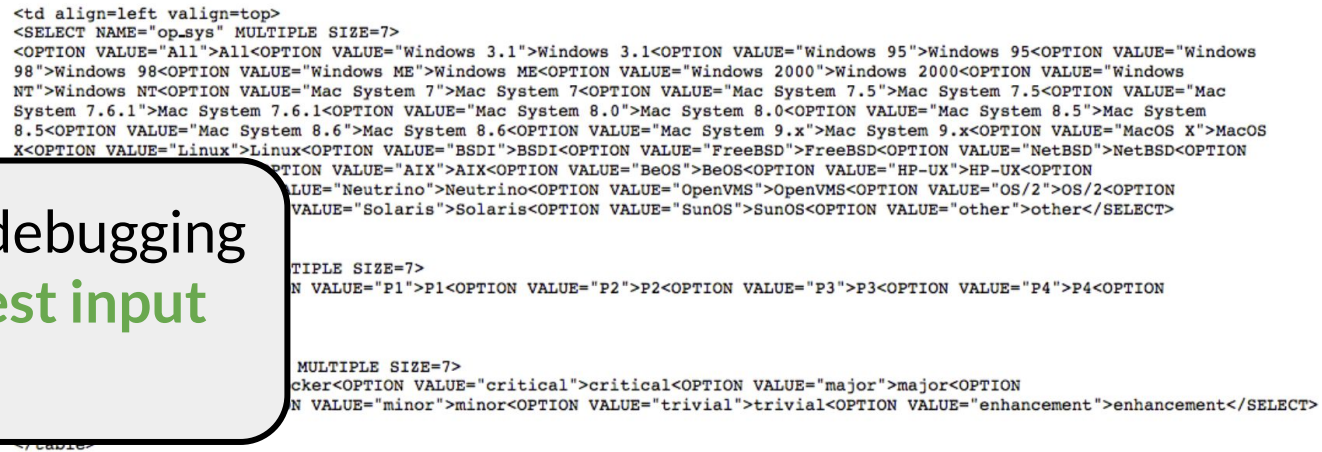
Delta debugging: motivation: inputs

- Having a **test input** may not be enough
  - Even if you know the suspicious code, the input may be **too large** to step through
- This HTML input makes a version of Mozilla crash. Which portion is relevant?

```
<td align=left valign=top>
<SELECT NAME="op_sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">Windows 3.1<OPTION VALUE="Windows 95">Windows 95<OPTION VALUE="Windows
98">Windows 98<OPTION VALUE="Windows ME">Windows ME<OPTION VALUE="Windows 2000">Windows 2000<OPTION VALUE="Windows
NT">Windows NT<OPTION VALUE="Mac System 7">Mac System 7<OPTION VALUE="Mac System 7.5">Mac System 7.5<OPTION VALUE="Mac
System 7.6.1">Mac System 7.6.1<OPTION VALUE="Mac System 8.0">Mac System 8.0<OPTION VALUE="Mac System 8.5">Mac System
8.5<OPTION VALUE="Mac System 8.6">Mac System 8.6<OPTION VALUE="Mac System 9.x">Mac System 9.x<OPTION VALUE="MacOS X">MacOS
X<OPTION VALUE="Linux">Linux<OPTION VALUE="BSDI">BSDI<OPTION VALUE="FreeBSD">FreeBSD<OPTION VALUE="NetBSD">NetBSD<OPTION
VALUE="OpenBSD">OpenBSD<OPTION VALUE="AIX">AIX<OPTION VALUE="BeOS">BeOS<OPTION VALUE="HP-UX">HP-UX<OPTION
VALUE="IRIX">IRIX<OPTION VALUE="Neutrino">Neutrino<OPTION VALUE="OpenVMS">OpenVMS<OPTION VALUE="OS/2">OS/2<OPTION
VALUE="OSF/1">OSF/1<OPTION VALUE="Solaris">Solaris<OPTION VALUE="SunOS">SunOS<OPTION VALUE="other">other</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<OPTION VALUE="--">--<OPTION VALUE="P1">P1<OPTION VALUE="P2">P2<OPTION VALUE="P3">P3<OPTION VALUE="P4">P4<OPTION
VALUE="P5">P5</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="bug_severity" MULTIPLE SIZE=7>
<OPTION VALUE="blocker">blocker<OPTION VALUE="critical">critical<OPTION VALUE="major">major<OPTION
VALUE="normal">normal<OPTION VALUE="minor">minor<OPTION VALUE="trivial">trivial<OPTION VALUE="enhancement">enhancement</SELECT>
</tr>
</table>
```

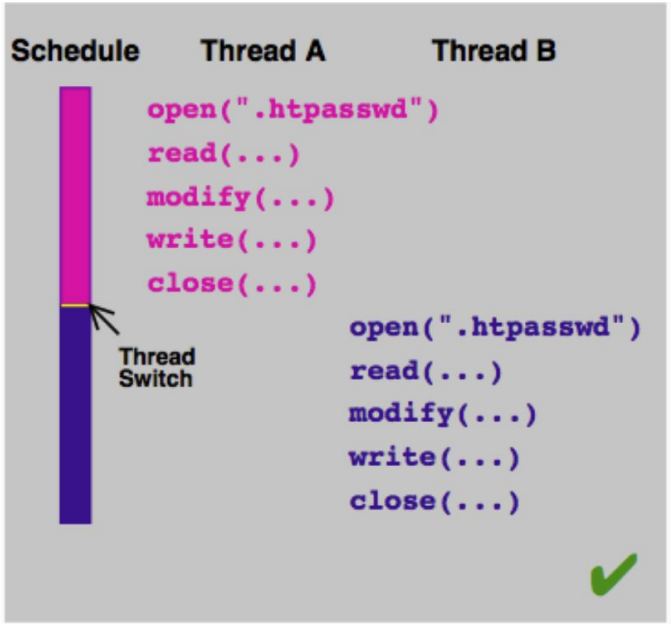
Delta debugging: motivation: inputs

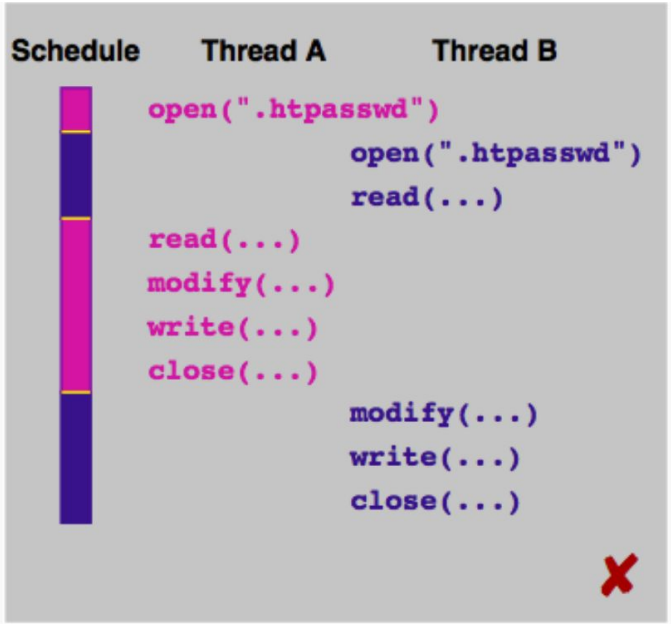
- Having a **test input** may not be enough
  - Even if you know the suspicious code, the input may be **too large** to step through
- This HTML input makes a version of Mozilla crash. Which portion is relevant?



Implication: delta debugging will be useful for **test input minimization**

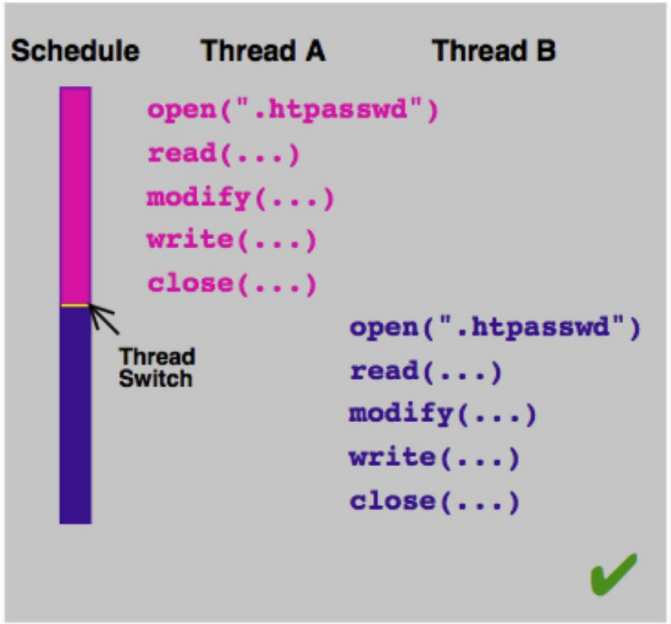
Delta debugging: motivation: thread schedules

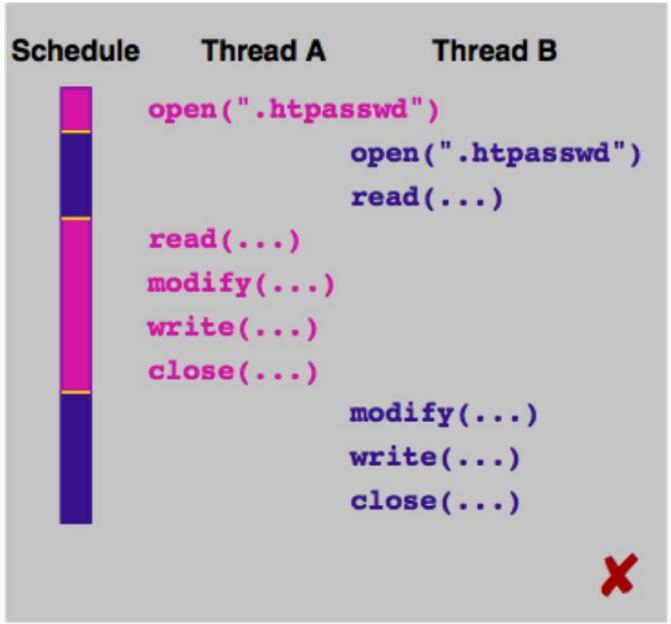




Delta debugging: motivation: thread schedules

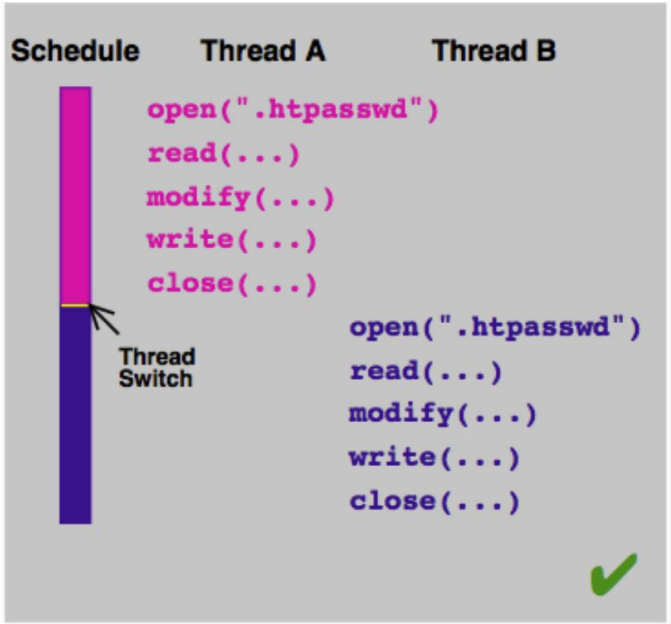
- Multithreaded programs can be **nondeterministic**

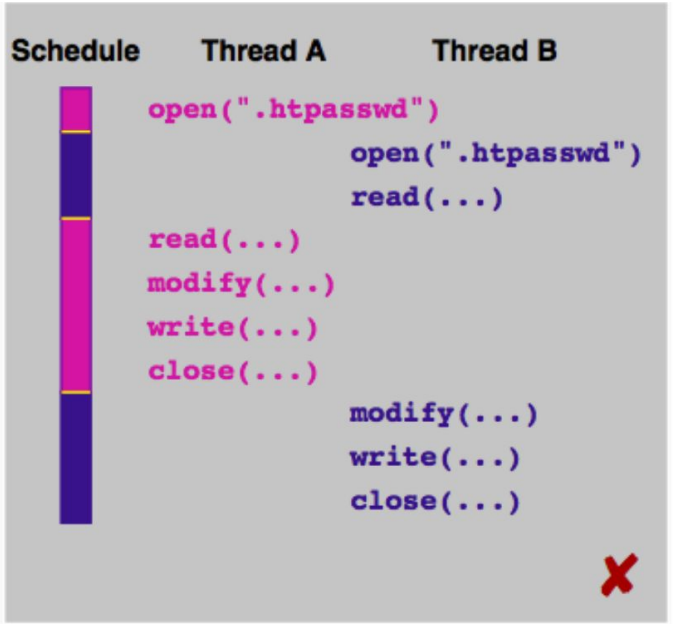




Delta debugging: motivation: thread schedules

- Multithreaded programs can be **nondeterministic**
  - Can we find simple, bug-inducing thread schedules?





Delta debugging: motivation: code changes

Delta debugging: motivation: code changes

- A new version of GDB has a UI bug

Delta debugging: motivation: code changes

- A new version of GDB has a UI bug
  - The old version does not have that bug (it is a **regression**)

Delta debugging: motivation: code changes

- A new version of GDB has a UI bug
  - The old version does not have that bug (it is a **regression**)
- 178,000 lines of code have been modified between the two versions

Delta debugging: motivation: code changes

- A new version of GDB has a UI bug
  - The old version does not have that bug (it is a **regression**)
- 178,000 lines of code have been modified between the two versions
  - Where is the bug?
    - … and **which commit** is responsible for introducing it?

Delta debugging: motivation: code changes

- A new version of GDB has a UI bug
  - The old version does not have that bug (it is a **regression**)
- 178,000 lines of code have been modified between the two versions
  - Where is the bug?
    - … and **which commit** is responsible for introducing it?
  - These days: **continuous integration testing** helps
    - … but does not totally solve this. Why?

Delta debugging: differences

Definition: With respect to debugging, a ***difference*** is a change in the program configuration or state that may lead to alternate observations

Delta debugging: differences

Definition: With respect to debugging, a ***difference*** is a change in the program configuration or state that may lead to alternate observations

- Difference in the **input**: different character or bit in the input stream

Delta debugging: differences

Definition: With respect to debugging, a ***difference*** is a change in the program configuration or state that may lead to alternate observations

- Difference in the **input**: different character or bit in the input stream
- Difference in **thread schedule**: difference in the time before a given thread preemption is performed

Delta debugging: differences

Definition: With respect to debugging, a ***difference*** is a change in the program configuration or state that may lead to alternate observations

- Difference in the **input**: different character or bit in the input stream
- Difference in **thread schedule**: difference in the time before a given thread preemption is performed
- Difference in **code**: different statements or expressions in two versions of a program

Delta debugging: differences

Definition: With respect to debugging, a ***difference*** is a change in the program configuration or state that may lead to alternate observations

- Difference in the **input**: different character or bit in the input stream
- Difference in **thread schedule**: difference in the time before a given thread preemption is performed
- Difference in **code**: different statements or expressions in two versions of a program
- Difference in **program state**: different values of internal variables

Delta debugging: unified solution

- Define the ***Abstract Debugging Problem*** as:

Delta debugging: unified solution

- Define the ***Abstract Debugging Problem*** as:
  - Find which part of something (= which difference, which input, which change) determines the failure

Delta debugging: unified solution

- Define the ***Abstract Debugging Problem*** as:
  - Find which part of something (= which difference, which input, which change) determines the failure
  - “Find the **smallest subset** of a given set that is still ***interesting***”

Delta debugging: unified solution

- Define the ***Abstract Debugging Problem*** as:
  - Find which part of something (= which difference, which input, which change) determines the failure
  - “Find the **smallest subset** of a given set that is still ***interesting***”
- Abstract solution: **divide-and-conquer**

Delta debugging: unified solution

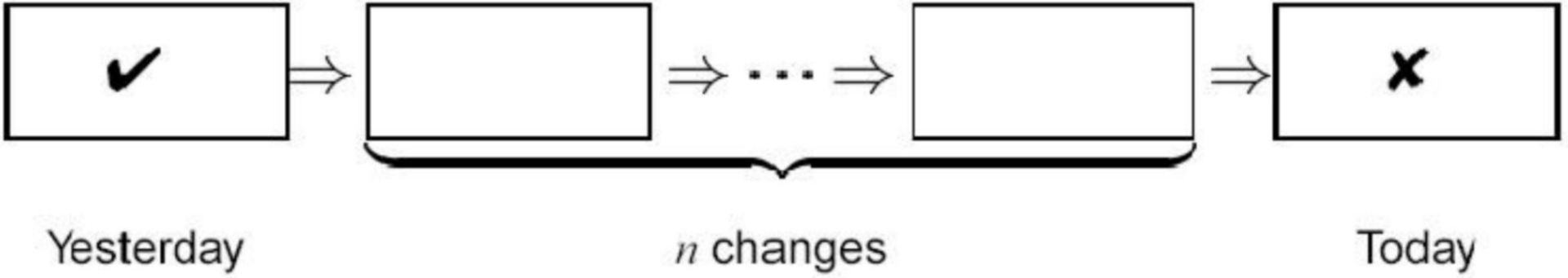
- Define the ***Abstract Debugging Problem*** as:
  - Find which part of something (= which difference, which input, which change) determines the failure
  - “Find the **smallest subset** of a given set that is still ***interesting***”
- Abstract solution: **divide-and-conquer**
  - **key idea**: split up the set into two subsets, check which of the two is still “**interesting**”

Delta debugging: unified solution

- Define the ***Abstract Debugging Problem*** as:
  - Find which part of something (= which difference, which input, which change) determines the failure
  - “Find the **smallest subset** of a given set that is still ***interesting***”
- Abstract solution: **divide-and-conquer**
  - **key idea**: split up the set into two subsets, check which of the two is still “**interesting**”
  - can be applied to working and failing inputs, code versions, thread schedules, program states, etc.

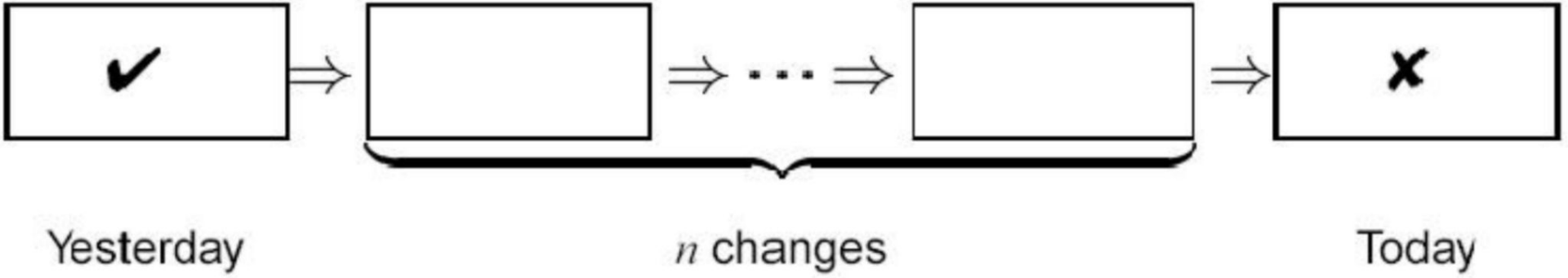
Delta debugging: unified solution

“Yesterday, my program worked. Today, it does not.”



Delta debugging: unified solution

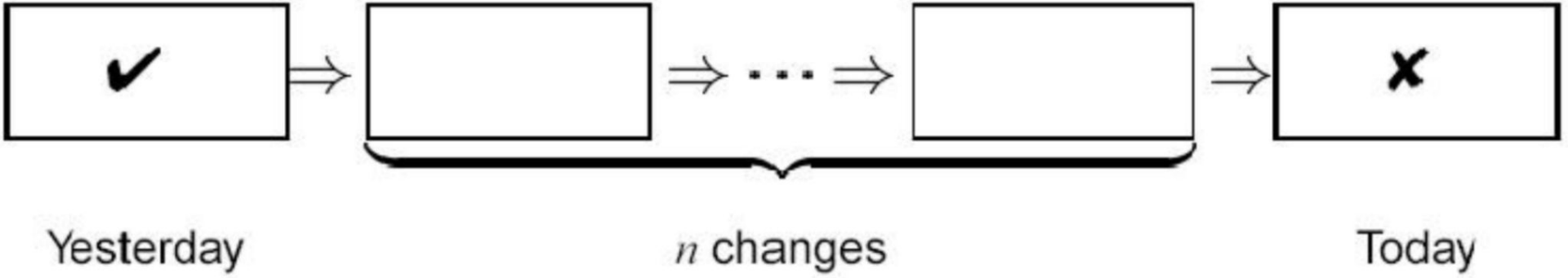
“Yesterday, my program worked. Today, it does not.”



- We will iteratively:

Delta debugging: unified solution

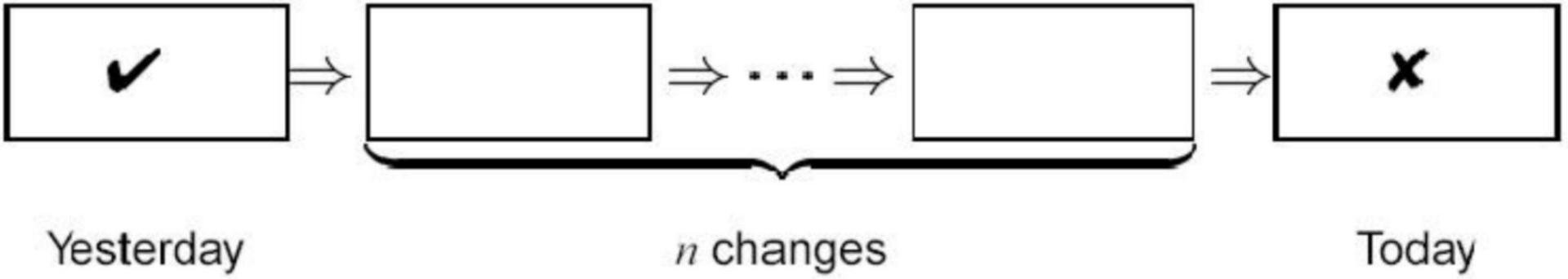
“Yesterday, my program worked. Today, it does not.”



- We will iteratively:
  - **hypothesize** that a small subset is interesting

Delta debugging: unified solution

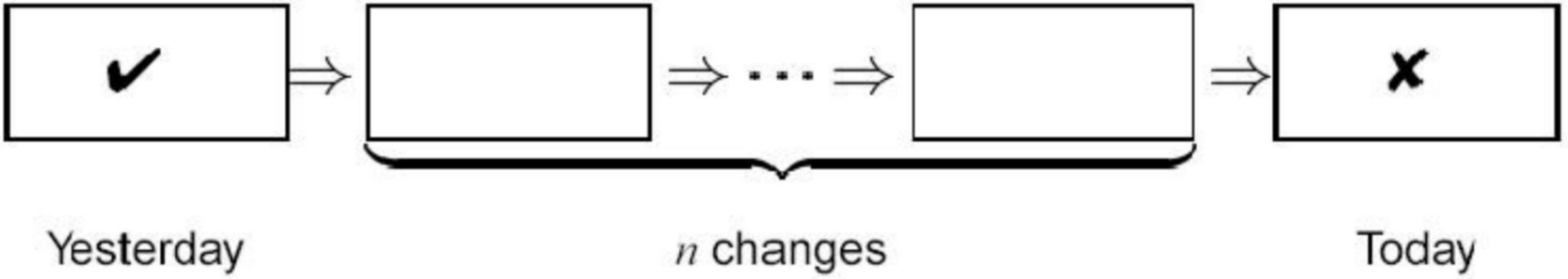
“Yesterday, my program worked. Today, it does not.”



- We will iteratively:
  - **hypothesize** that a small subset is interesting
    - e.g., the subset of changes {1, 3, 8} causes the bug

Delta debugging: unified solution

“Yesterday, my program worked. Today, it does not.”



- We will iteratively:
  - **hypothesize** that a small subset is interesting
    - e.g., the subset of changes {1, 3, 8} causes the bug
  - run tests to **falsify** our hypothesis

Delta debugging: algorithm

Delta debugging: algorithm

- Given:

Delta debugging: algorithm

- Given:
  - a set $C = \{c_1, \ldots, c_n\}$ (of changes)

Delta debugging: algorithm

- Given:
  - a set ***C*** = $\{c_1, \ldots, c_n\}$ (of changes)
  - a function ***Interesting*** : $C \rightarrow \{\text{True, False}\}$

Delta debugging: algorithm

- Given:
  - a set ***C*** = $\{c_1, \ldots, c_n\}$ (of changes)
  - a function ***Interesting*** : C $\rightarrow$ {True, False}
  - Interesting(C) = Yes , Interesting({}) = No

Delta debugging: algorithm

- Given:
  - a set ***C*** = $\{c_1, \ldots, c_n\}$ (of changes)
  - a function ***Interesting*** : C → {True, False}
  - Interesting(C) = Yes , Interesting({}) = No
  - Interesting is monotonic, unambiguous and consistent (more on these later)

Delta debugging: algorithm

- Given:
  - a set ***C*** = {c_1 , … , c_n } (of changes)
  - a function ***Interesting*** : C → {True, False}
  - Interesting(C) = Yes , Interesting({}) = No
  - Interesting is monotonic, unambiguous and consistent (more on these later)
- The **delta debugging algorithm** returns a **minimal Interesting subset** M of C:

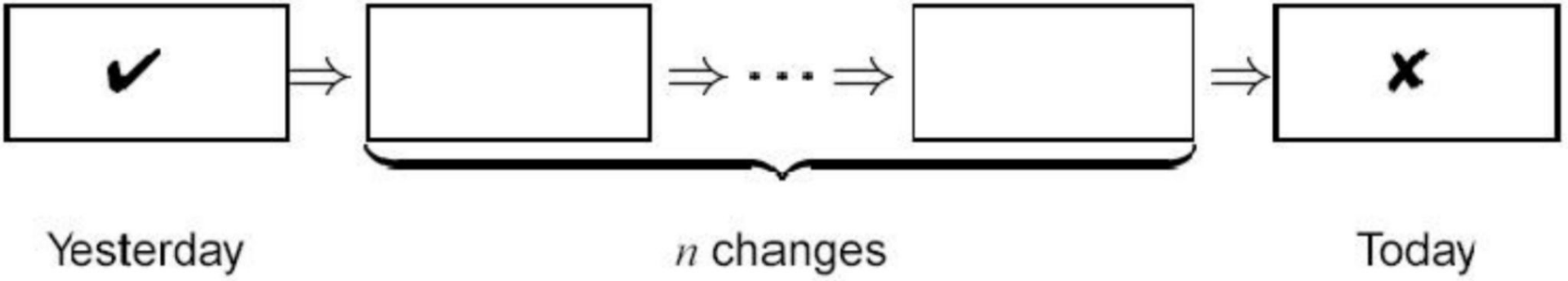
Delta debugging: algorithm

- Given:
  - a set ***C*** = {c_1 , … , c_n } (of changes)
  - a function ***Interesting*** : C → {True, False}
  - Interesting(C) = Yes , Interesting({}) = No
  - Interesting is monotonic, unambiguous and consistent (more on these later)
- The **delta debugging algorithm** returns a **minimal Interesting subset** M of C:
  - Interesting(M) = Yes

Delta debugging: algorithm

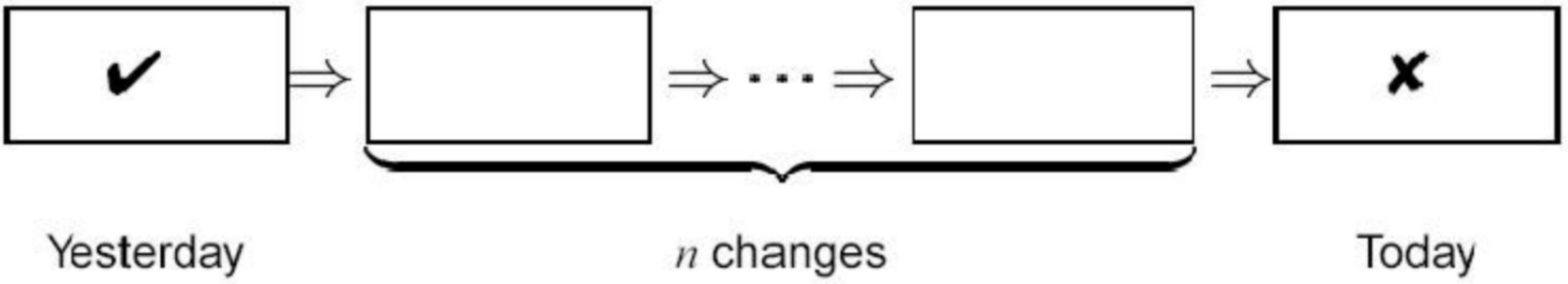
- Given:
  - a set ***C*** = {c_1 , … , c_n } (of changes)
  - a function ***Interesting*** : C → {True, False}
  - Interesting(C) = Yes , Interesting({}) = No
  - Interesting is monotonic, unambiguous and consistent (more on these later)
- The **delta debugging algorithm** returns a **minimal Interesting subset** M of C:
  - Interesting(M) = Yes
  - Forall m ⊂ M, Interesting(M - m) = No

Delta debugging: example



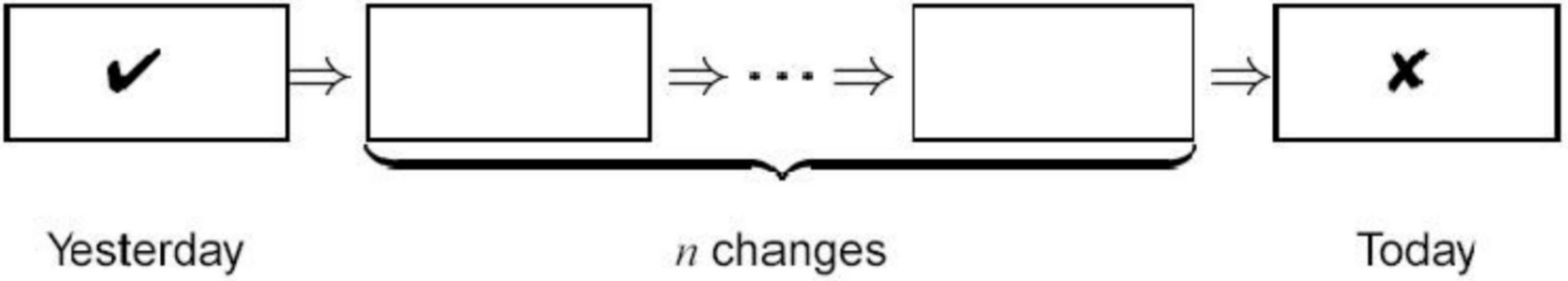
- C =
- Interesting(X) =

Delta debugging: example



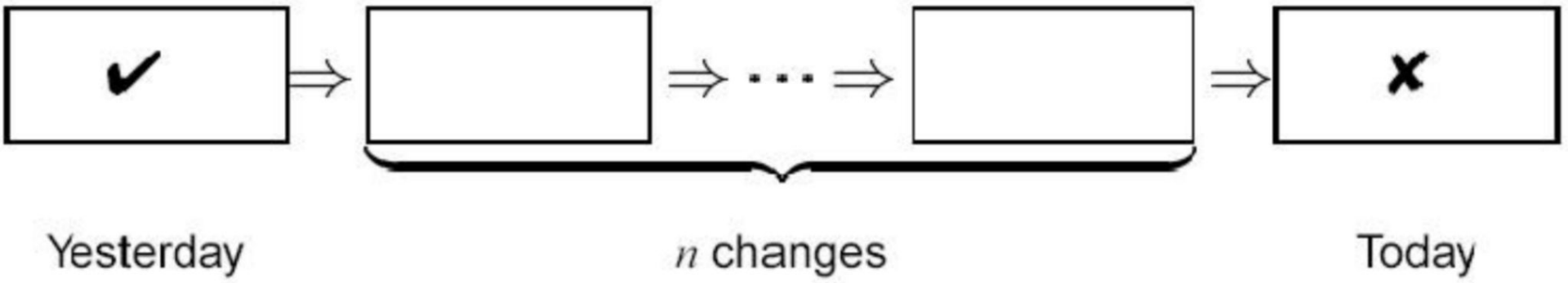
- C = set of *n* changes
- Interesting(X) =

Delta debugging: example



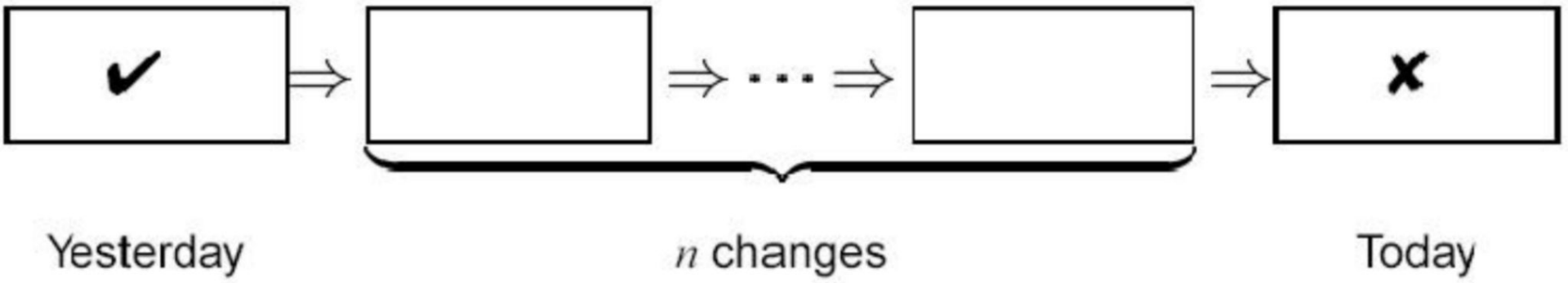
- C = set of *n* changes
- Interesting(X) = apply the changes in in X to Yesterday's version and compile. Run the tests on the result.

Delta debugging: example



- C = set of *n* changes
- Interesting(X) = apply the changes in in X to Yesterday's version and compile. Run the tests on the result.
  - If the tests **fail**, Interesting(X) = **True**.

Delta debugging: example



- C = set of *n* changes
- Interesting(X) = apply the changes in in X to Yesterday's version and compile. Run the tests on the result.
  - If the tests **fail**, Interesting(X) = **True**.
  - If the tests **pass**, Interesting(X) = **False**.

Delta debugging: algorithm: naive

- We could just **try all subsets** of C to find the smallest one that is Interesting

Delta debugging: algorithm: naive

- We could just **try all subsets** of C to find the smallest one that is Interesting
  - **Problem**: if |C| = N, this takes 2^N time

Delta debugging: algorithm: naive

- We could just **try all subsets** of C to find the smallest one that is Interesting
  - **Problem**: if |C| = N, this takes 2^N time
  - Recall: real-world software is **unimaginably huge**

Delta debugging: algorithm: naive

- We could just **try all subsets** of C to find the smallest one that is Interesting
  - **Problem**: if |C| = N, this takes 2^N time
  - Recall: real-world software is **unimaginably huge**
- We want a **polynomial-time** solution
  - Ideally one that is more like log(N)
  - Or we'll loop for what feels like forever

Delta debugging: algorithm candidate

Precondition: Interesting({c_1 … c_n }) = True

DD({c, , …, c_n }) =

if n = 1 then return {c_1 }

let P_1 = {c_1 , … $c_{n/2}$}

let P_2 = {$c_{n/2+1}$, …, c_n }

if **Interesting**(P_1) is True:

then return DD(P_1)

else return DD(P_2)

Delta debugging: algorithm candidate

Precondition: Interesting($\{c_1 \ldots c_n\}$) = True

DD($\{c, , \ldots, c_n\}$) =

if n = 1 then return $\{c_1\}$

let $P_1 = \{c_1, \ldots c_{n/2}\}$

let $P_2 = \{c_{n/2+1}, \ldots, c_n\}$

if **Interesting**(P_1) is True:

then return DD(P_1)

else return DD(P_2)

This is just **binary search**! It won't work if you need a big subset to be Interesting

Delta debugging: algorithm: assumptions

Delta debugging: algorithm: assumptions

- **Any subset** of changes may be Interesting
  - Not just singleton subsets of size 1 (cf. binary search)

Delta debugging: algorithm: assumptions

- **Any subset** of changes may be Interesting
  - Not just singleton subsets of size 1 (cf. binary search)
- Interesting is **Monotonic**
  - Interesting(X)→ Interesting(X ∪ {c})

Delta debugging: algorithm: assumptions

- **Any subset** of changes may be Interesting
  - Not just singleton subsets of size 1 (cf. binary search)
- Interesting is **Monotonic**
  - Interesting(X)→ Interesting(X ∪ {c})
- Interesting is **Unambiguous**
  - Interesting(X) & Interesting(Y) → Interesting(X ∩ Y)

Delta debugging: algorithm: assumptions

- **Any subset** of changes may be Interesting
  - Not just singleton subsets of size 1 (cf. binary search)
- Interesting is **Monotonic**
  - Interesting(X)→ Interesting(X ∪ {c})
- Interesting is **Unambiguous**
  - Interesting(X) & Interesting(Y) → Interesting(X ∩ Y)
- Interesting is **Consistent**
  - Interesting(X) = True xor Interesting(X) = False
  - (Some formulations also allow: Interesting(X) = Unknown)

Delta debugging: algorithm: insights

- Basic Binary Search:
  - Divide C into P_1 and P_2
  - If Interesting(P_1) = True then recurse on P_1
  - If Interesting(P_2) = True then recurse on P_2

Delta debugging: algorithm: insights

- Basic Binary Search:
  - Divide C into P_1 and P_2
  - If Interesting(P_1) = True then recurse on P_1
  - If Interesting(P_2) = True then recurse on P_2
- At most one case can apply (by **Unambiguous**)

Delta debugging: algorithm: insights

- Basic Binary Search:
  - Divide C into P_1 and P_2
  - If Interesting(P_1) = True then recurse on P_1
  - If Interesting(P_2) = True then recurse on P_2
- At most one case can apply (by **Unambiguous**)

Unambiguous =
Interesting(X) & Interesting(Y) → Interesting(X ∩ Y)

Delta debugging: algorithm: insights

- Basic Binary Search:
  - Divide C into P_1 and P_2
  - If Interesting(P_1) = True then recurse on P_1
  - If Interesting(P_2) = True then recurse on P_2
- At most one case can apply (by **Unambiguous**)
- By **Consistency**, the only other possibility is:

Delta debugging: algorithm: insights

- Basic Binary Search:
  - Divide C into P_1 and P_2
  - If Interesting(P_1) = True the
  - If Interesting(P_2) = True the
- At most one case can apply (by **Unambiguous**)
- By **Consistency**, the only other possibility is:

Consistency =
Interesting(X) = True xor
Interesting(X) = False

Delta debugging: algorithm: insights

- Basic Binary Search:
  - Divide C into P_1 and P_2
  - If Interesting(P_1) = True then recurse on P_1
  - If Interesting(P_2) = True then recurse on P_2
- At most one case can apply (by **Unambiguous**)
- By **Consistency**, the only other possibility is:
  - (Interesting(P1) = False) *and* (Interesting(P2) = False)

Delta debugging: algorithm: insights

- Basic Binary Search:
  - Divide C into P_1 and P_2
  - If Interesting(P_1) = True then recurse on P_1
  - If Interesting(P_2) = True then recurse on P_2
- At most one case can apply (by **Unambiguous**)
- By **Consistency**, the only other possibility is:
  - (Interesting(P1) = False) *and* (Interesting(P2) = False)
  - What happens in such a case?

Delta debugging: algorithm: interference

- By **Monotonicity**
  - If Interesting(P_1) = False and Interesting(P_2) = False

Delta debugging: algorithm: interference

- By **Monotonicity**
  - If Interesting(P_1) = False and Interesting(P_2) = False

Monotonicity =
Interesting(X)⟶
Interesting(X ∪ {c})

Delta debugging: algorithm: interference

- By **Monotonicity**
  - If Interesting(P_1) = False and Interesting(P_2) = False
  - Then no subset of P_1 alone or subset of P_2 alone is Interesting

Delta debugging: algorithm: interference

- By **Monotonicity**
  - If Interesting(P_1) = False and Interesting(P_2) = False
  - Then no subset of P_1 alone or subset of P_2 alone is Interesting
- So the Interesting subset must use a **combination** of elements from P_1 and P_2

Delta debugging: algorithm: interference

- By **Monotonicity**
  - If Interesting(P_1) = False and Interesting(P_2) = False
  - Then no subset of P_1 alone or subset of P_2 alone is Interesting
- So the Interesting subset must use a **combination** of elements from P_1 and P_2
- In Delta Debugging, this is called ***interference***

Delta debugging: algorithm: interference

- Why is this true?

Delta debugging: algorithm: interference

- Why is this true?
  - Consider P_1
    - Find a minimal subset D_2 of P_2
    - Such that Interesting($P_1 \cup D_2$) = True

Delta debugging: algorithm: interference

- Why is this true?
  - Consider P_1
    - Find a minimal subset D_2 of P_2
    - Such that Interesting($P_1 \cup D_2$) = True
  - Consider P_2
    - Find a minimal subset D_1 of P_1
    - Such that Interesting($P_2 \cup D_1$) = True

Delta debugging: algorithm: interference

- Why is this true?
  - Consider P_1
    - Find a minimal subset D_2 of P_2
    - Such that Interesting($P_1 \cup D_2$) = True
  - Consider P_2
    - Find a minimal subset D_1 of P_1
    - Such that Interesting($P_2 \cup D_1$) = True
  - Then by **Unambiguous**
    - Interesting($(P_1 \cup D_2) \cap (P_2 \cup D_1)$) = Interesting($D_1 \cup D_2$) is also minimal

Delta debugging: algorithm: interference

- Why is this true?
  - Consider P_1
    - Find a minimal subset D_2 of P_2
    - Such that Interesting($P_1 \cup D_2$) = True
  - Consider P_2
    - Find a minimal subset D_1 of P_1
    - Such that Interesting($P_2 \cup D_1$) = True
  - Then by **Unambiguous**
    - Interesting($(P_1 \cup D_2) \cap (P_2 \cup D_1)$) = Interesting($D_1 \cup D_2$) is also minimal

Key point: combination of elements from both

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let’s use DD to find it

1 2 3 4 5 6 7 8 = Interesting

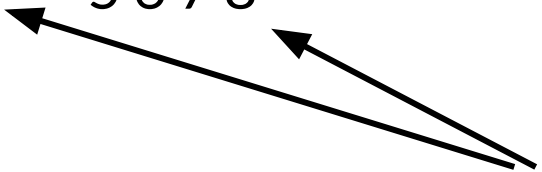
Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let's use DD to find it

1 2 3 4 5 6 7 8 = Interesting

1 2 3 4

5 6 7 8



First step: partition C = {1, …, 8}
into P_1 = {1, …, 4} and P_2 = {5, …, 8}

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let's use DD to find it

1 2 3 4 5 6 7 8 = Interesting
1 2 3 4 = ???
5 6 7 8 = ???

Next step: test P_1 and P_2

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let's use DD to find it

1 2 3 4 5 6 7 8 = Interesting
1 2 3 4 = False
5 6 7 8 = False

Interference! Sub-step: find minimal subset D_1 of P_1 such that Interesting(D_1 + P_2)

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let's use DD to find it

1 2 3 4	5 6 7 8	= Interesting
1 2 3 4		= False
	5 6 7 8	= False
1 2	5 6 7 8	= ???

Interference! Sub-step: find minimal subset D_1 of P_1 such that Interesting(D_1 + P_2)

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let's use DD to find it

1 2 3 4	5 6 7 8	= Interesting
1 2 3 4		= False
	5 6 7 8	= False
1 2	5 6 7 8	= False

Interference! Sub-step: find minimal subset D_1 of P_1 such that Interesting(D_1 + P_2)

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let's use DD to find it

1 2 3 4	5 6 7 8	= Interesting	
1 2 3 4		= False	
	5 6 7 8	= False	
1 2	5 6 7 8	= False	
3 4	5 6 7 8	= ???	

Interference! Sub-step: find minimal subset D_1 of P_1 such that Interesting(D_1 + P_2)

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let's use DD to find it

1 2 3 4 5 6 7 8	= Interesting
1 2 3 4	= False
5 6 7 8	= False
1 2 5 6 7 8	= False
3 4 5 6 7 8	= True
3 5 6 7 8	= True

$D_1 = \{ 3 \}$

Now we need to find D_2

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let's use DD to find it

1 2 3 4	5 6 7 8	= Interesting	$D_1 = \{ 3 \}$
1 2 3 4		= False	
	5 6 7 8	= False	Now we need to find D_2
1 2	5 6 7 8	= False	
3 4	5 6 7 8	= True	
3	5 6 7 8	= True	
1 2 3 4	5 6	= True	

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let's use DD to find it

1 2 3 4	5 6 7 8	= Interesting
1 2 3 4		= False
	5 6 7 8	= False
1 2	5 6 7 8	= False
3 4	5 6 7 8	= True
3	5 6 7 8	= True
1 2 3 4	5	= False

$D_1 = \{ 3 \}$

Now we need to find D_2

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let's use DD to find it

1 2 3 4	5 6 7 8	= Interesting	$D_1 = \{ 3 \}$
1 2 3 4		= False	
	5 6 7 8	= False	$D_2 = \{ 6 \}$
1 2	5 6 7 8	= False	
3 4	5 6 7 8	= True	
3	5 6 7 8	= True	
1 2 3 4	6	= True	

Delta debugging: algorithm: example

- Suppose {3,6} Is Smallest Interesting Subset of {1, …, 8}
- Let's use DD to find it

1 2 3 4	5 6 7 8	= Interesting
1 2 3 4		= False
	5 6 7 8	= False
1 2	5 6 7 8	= False
3 4	5 6 7 8	= True
3	5 6 7 8	= True
1 2 3 4	6	= True

$D_1 = \{ 3 \}$

$D_2 = \{ 6 \}$

So, final answer =
$D_1 \cup D_2 = \{ 3, 6 \}$

Delta debugging: final algorithm

Precondition: Interesting($\{c_1 \ldots c_n\}$) = True

DD(P, $\{c, , \ldots, c_n\}$) =

if n = 1 then return $\{c_1\}$

let $P_1 = \{c_1, \ldots c_{n/2}\}$

let $P_2 = \{c_{n/2+1}, \ldots, c_n\}$

if **Interesting**($P_1 \cup P$) is True then return DD(P, P_1)

else if **Interesting**($P_2 \cup P$) is True then return DD(P, P_2)

else return **DD**($P \cup P_2, P_1$) $\cup$ **DD**($P \cup P_1, P_2$)

Delta debugging: algorithmic complexity

Delta debugging: algorithmic complexity

- If a single change induces the failure:
  - DD is **logarithmic**: 2 * log |C|
  - Why?

Delta debugging: algorithmic complexity

- If a single change induces the failure:
  - DD is **logarithmic**: 2 * log |C|
  - Why?
- Otherwise, DD is **linear**
  - Assuming constant time per Interesting() check
  - Is this realistic?

Delta debugging: algorithmic complexity

- If a single change induces the failure:
  - DD is **logarithmic**: 2 * log |C|
  - Why?
- Otherwise, DD is **linear**
  - Assuming constant time per Interesting() check
  - Is this realistic?
- If Interesting can return “Unknown”
  - DD is **quadratic**: $|C|^2 + 3|C|$
  - If all tests are Unknown except last one (unlikely)

Assumptions restated on this slide for convenience

Delta debugging: questioning assumptions

- **All three** assumptions are **questionable**
- Interesting is **Monotonic**
  - Interesting(X)→ Interesting(X ∪ {c})
- Interesting is **Unambiguous**
  - Interesting(X) & Interesting(Y) → Interesting(X ∩ Y)
- Interesting is **Consistent**
  - Interesting(X) = True xor Interesting(X) = False
  - (Some formulations also allow: Interesting(X) = Unknown)

Assumptions restated on this slide for convenience

Delta debugging: questioning assumptions

- **All three** assumptions are **questionable**
- ~~Interesting is **Monotonic**~~
  - Interesting(X)→ Interesting(X ∪ {c})
- Interesting is **Unambiguous**
  - Interesting(X) & Interes
- Interesting is **Consistent**
  - Interesting(X) = True xc
  - (Some formulations als

Monotonicity is rare in the real world. But DD still finds *an* interesting subset if Interesting is not monotonic (might not be minimal)

Assumptions restated on this slide for convenience

Delta debugging: questioning assumptions

- **All three** assumptions are **questionable**
- Interesting is **Monotonic**
  - Interesting(X)→ Interesting(X ∪ {c})
- ~~Interesting is **Unambiguous**~~
  - Interesting(X) & Interes
- Interesting is **Consistent**
  - Interesting(X) = True xc
  - (Some formulations als

Ambiguity will cause DD to fail. Hint: try tracing DD on Interesting ({2, 8}) = True, but Interesting({2, 8} intersect {3, 6}) = False

Assumptions restated on this slide for convenience

Delta debugging: questioning assumptions

- **All three** assumptions are
- Interesting is **Monotonic**
  - Interesting(X)→ Intere
- Interesting is **Unambiguou**
  - Interesting(X) & Interes
- ~~Interesting is **Consistent**~~
  - Interesting(X) = True xor Interesting(X) = False
  - (Some formulations also allow: Interesting(X) = Unknown)

The world is **often inconsistent**. Example: we are minimizing changes to a program to find patches that makes it crash. Some subsets may not build or run!

Delta debugging: in the real world

- `git bisect` implements a DD-like algorithm (look it up!)
- for thread schedules: DejaVu tool by IBM, CHESS by Microsoft, etc.
- Eclipse plugins for code changes (“DDinput”, “DDchange”)
- you can also do delta debugging **by hand** (I do this often for programs that cause compiler bugs!)

Debugging (Part 2/2)

Today's agenda:

- Debugging
  - printf debugging and logging
  - delta debugging
  - **debuggers**

Debuggers

Debuggers

Definition: a ***debugger*** is “a software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables.” [definition from Microsoft Developer Network]

Debuggers

Definition: a ***debugger*** is “a software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables.” [definition from Microsoft Developer Network]

- Can operate on source code or assembly code

Debuggers

Definition: a ***debugger*** is “a software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables.” [definition from Microsoft Developer Network]

- Can operate on source code or assembly code
- **Inspect** the values of registers, memory

Debuggers

Definition: a ***debugger*** is “a software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables.” [definition from Microsoft Developer Network]

- Can operate on source code or assembly code
- **Inspect** the values of registers, memory
- **Key Features** (we’ll explain all of them): attach to process, single-stepping, breakpoints, conditional breakpoints, watchpoints

Debuggers: how do they work

Debuggers: how do they work: signals

Debuggers: how do they work: signals

- A ***signal*** is an **asynchronous** notification sent to a process about an event:
  - User pressed Ctrl-C (or did kill %pid)
    - Or asked the Windows Task Manager to terminate it
  - Exceptions (divide by zero, null pointer)
  - From the OS (SIGPIPE)

Debuggers: how do they work: signals

- A ***signal*** is an **asynchronous** notification sent to a process about an event:
  - User pressed Ctrl-C (or did kill %pid)
    - Or asked the Windows Task Manager to terminate it
  - Exceptions (divide by zero, null pointer)
  - From the OS (SIGPIPE)
- You can install a **signal handler** – a procedure that will be executed when the signal occurs.

Debuggers: how do they work: signals

- A ***signal*** is an **asynchronous** notification sent to a process about an event:
  - User pressed Ctrl-C (or did kill %pid)
    - Or asked the Windows Task Manager to terminate it
  - Exceptions (divide by zero, null pointer)
  - From the OS (SIGPIPE)
- You can install a **signal handler** – a procedure that will be executed when the signal occurs.
  - Signal handlers are vulnerable to **race conditions**. Why?

Debuggers: how do they work: attaching

- Attaching a debugger to a process requires **operating system** support

Debuggers: how do they work: attaching

- Attaching a debugger to a process requires **operating system** support
- There is a **special system call** that allows one process to act as a debugger for a target

Debuggers: how do they work: attaching

- Attaching a debugger to a process requires **operating system** support
- There is a **special system call** that allows one process to act as a debugger for a target
  - What are the **security** concerns?

Debuggers: how do they work: attaching

- Attaching a debugger to a process requires **operating system** support
- There is a **special system call** that allows one process to act as a debugger for a target
  - What are the **security** concerns?
- Once this is done, the debugger can basically "**catch signals**" delivered to the target
  - this isn't exactly what happens, but it's a good explanation …

Debuggers: how do they work: breakpoints

- We now have all the ingredients for a “**classic**” debugger (like gdb): **breakpoints** and **interactive debugging**. How it works:

Debuggers: how do they work: breakpoints

- We now have all the ingredients for a "**classic**" debugger (like gdb): **breakpoints** and **interactive debugging**. How it works:

A ***breakpoint*** is a user-specified program statement on which the debugger should stop the program and begin an interactive debugging session

Debuggers: how do they work: breakpoints

- We now have all the ingredients for a “**classic**” debugger (like gdb): **breakpoints** and **interactive debugging**. How it works:
  - Attach to target

Debuggers: how do they work: breakpoints

- We now have all the ingredients for a “**classic**” debugger (like gdb): **breakpoints** and **interactive debugging**. How it works:
  - Attach to target
  - Set up signal handler

Debuggers: how do they work: breakpoints

- We now have all the ingredients for a "**classic**" debugger (like gdb): **breakpoints** and **interactive debugging**. How it works:
  - Attach to target
  - Set up signal handler
  - Add in **exception causing instructions** at desired breakpoints

Debuggers: how do they work: breakpoints

- We now have all the ingredients for a "**classic**" debugger (like gdb): **breakpoints** and **interactive debugging**. How it works:
  - Attach to target
  - Set up signal handler
  - Add in **exception causing instructions** at desired breakpoints
  - **Inspect** globals, do other debugger things, etc.

Debuggers: how do they work: breakpoints

```
#define BREAKPOINT *(0)=0
int global = 11;
int debugger_signal_handler() {
  printf("debugger prompt: \n");
  // debugger code goes here!
}
void main() {
  signal(SIGSEGV, debugger_signal_handler) ;
  global = 33;
  BREAKPOINT;
  global = 55;
  printf("Outside, global = %d\n", global);
 }
```

All code added by the debugger in **purple**

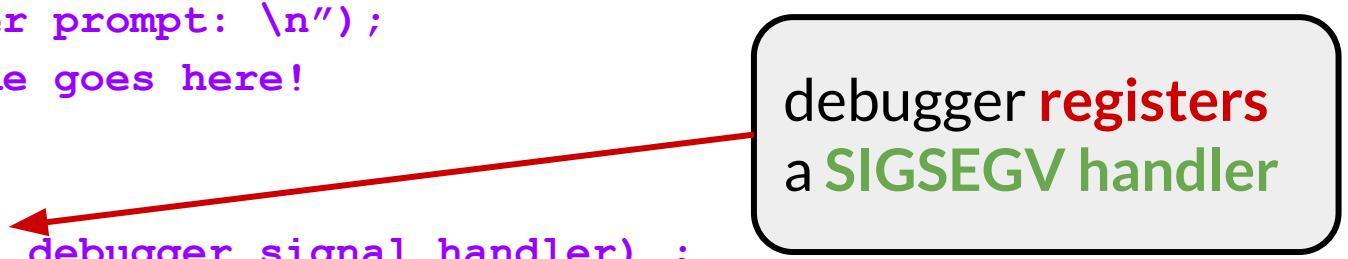
Debuggers: how do they work: breakpoints

```
#define BREAKPOINT *(0)=0
int global = 11;
int debugger_signal_handler() {
  printf("debugger prompt: \n");
  // debugger code goes here!
}
void main() {
  signal(SIGSEGV, debugger_signal_handler) ;
  global = 33;
  BREAKPOINT;
  global = 55;
  printf("Outside, global = %d\n", global);
 }
```

"**BREAKPOINT**" macro is guaranteed to cause SIGSEGV

Debuggers: how do they work: breakpoints

```
#define BREAKPOINT *(0)=0
int global = 11;
int debugger_signal_handler() {
  printf("debugger prompt: \n");
  // debugger code goes here!
}
void main() {
  signal(SIGSEGV, debugger_signal_handler) ;
  global = 33;
  BREAKPOINT;
  global = 55;
  printf("Outside, global = %d\n", global);
 }
```



Debuggers: how do they work: breakpoints

```
#define BREAKPOINT *(0)=0
int global = 11;
int debugger_signal_handler() {
  printf("debugger prompt: \n");
  // debugger code goes here!
}
void main() {
  signal(SIGSEGV, debugger_signal_handler) ;
  global = 33;
  BREAKPOINT;
  global = 55;
  printf("Outside, global = %d\n", global);
 }
```

Debuggers: how do they work: breakpoints

```
#define BREAKPOINT *(0)=0
int global = 11;
int debugger_signal_handler() {
  printf("debugger prompt: \n");
  // debugger code goes here!
}
void main() {
  signal(SIGSEGV, debugger_signal_handler) ;
  global = 33;
  BREAKPOINT;
  global = 55;
  printf("Outside, global = %d\n", global);
 }
```

at the user-specified breakpoint, the debugger **forces** a SIGSEGV (which its handler will intercept)

Debuggers: advanced breakpoints

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***
  - Special register: if PC value = HBP register value, signal

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***
  - Special register: if PC value = HBP register value, signal
  - Faster than software, works on ROMs, only limited number of breakpoints, etc.

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***
  - Special register: if PC value = HBP register value, signal
  - Faster than software, works on ROMs, only limited number of breakpoints, etc.
- Feature: ***conditional breakpoint***: “break at instruction X if **some_var = some_value**”

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***
  - Special register: if PC value = HBP register value, signal
  - Faster than software, works on ROMs, only limited number of breakpoints, etc.
- Feature: ***conditional breakpoint***: “break at instruction X if **some_var = some_value**”
- As before, but signal handler checks if **some_var = some_value**

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***
  - Special register: if PC value = HBP register value, signal
  - Faster than software, works on ROMs, only limited number of breakpoints, etc.
- Feature: ***conditional breakpoint***: "break at instruction X if **some_var = some_value**"
- As before, but signal handler checks if **some_var = some_value**
  - If so, present interactive debugging prompt

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***
  - Special register: if PC value = HBP register value, signal
  - Faster than software, works on ROMs, only limited number of breakpoints, etc.
- Feature: ***conditional breakpoint***: "break at instruction X if **some_var = some_value**"
- As before, but signal handler checks if **some_var = some_value**
  - If so, present interactive debugging prompt
  - If not, return to program immediately

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***
  - Special register: if PC value = HBP register value, signal
  - Faster than software, works on ROMs, only limited number of breakpoints, etc.
- Feature: ***conditional breakpoint***: “break at instruction X if **some_var = some_value**”
- As before, but signal handler checks if **some_var = some_value**
  - If so, present interactive debugging prompt
  - If not, return to program immediately
  - Is this fast or slow?

Debuggers: advanced breakpoints

- Optimization: ***hardware breakpoints***
  - Special register: if PC value = HBP register value, signal
  - Faster than software, works on ROMs, only limited number of breakpoints, etc.
- Feature: ***conditional breakpoint***: "break at instruction X if **some_var = some_value**"
- As before, but signal handler checks if **some_var = some_value**
  - If so, present interactive debugging prompt
  - If not, return to program immediately
  - Is this fast or **slow**?

Debuggers: single-stepping

Debuggers: single-stepping

- Debuggers also allow you to advance through code **one instruction at a time** (this is called ***single-stepping***)

Debuggers: single-stepping

- Debuggers also allow you to advance through code **one instruction at a time** (this is called ***single-stepping***)
- To implement this, put a breakpoint at the first instruction (= at program start)

Debuggers: single-stepping

- Debuggers also allow you to advance through code **one instruction at a time** (this is called ***single-stepping***)
- To implement this, put a breakpoint at the first instruction (= at program start)
- The “**single step**” or “**next**” interactive command is equal to:

Debuggers: single-stepping

- Debuggers also allow you to advance through code **one instruction at a time** (this is called ***single-stepping***)
- To implement this, put a breakpoint at the first instruction (= at program start)
- The "**single step**" or "**next**" interactive command is equal to:
  - Put a breakpoint at the next instruction
  - Resume execution
  - (No, really.)

Debuggers: watchpoints

- You want to know when a variable **changes**

Debuggers: watchpoints

- You want to know when a variable **changes**
- A ***watchpoint*** is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Debuggers: watchpoints

- You want to know when a variable **changes**
- A ***watchpoint*** is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**
- How could we implement this?

Debuggers: watchpoints

A ***watchpoint*** is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Debuggers: watchpoints

A *watchpoint* is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Software Watchpoints:

Debuggers: watchpoints

A ***watchpoint*** is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Software Watchpoints:

- Put a breakpoint at **every instruction** (ouch!)

Debuggers: watchpoints

A *watchpoint* is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Software Watchpoints:

- Put a breakpoint at **every instruction** (ouch!)
- Check the current value of **L** against a stored value

Debuggers: watchpoints

A *watchpoint* is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Software Watchpoints:

- Put a breakpoint at **every instruction** (ouch!)
- Check the current value of **L** against a stored value
- If different, give interactive debugging prompt

Debuggers: watchpoints

A *watchpoint* is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Software Watchpoints:

- Put a breakpoint at **every instruction** (ouch!)
- Check the current value of **L** against a stored value
- If different, give interactive debugging prompt
- If not, set next breakpoint and continue (single-step)

Debuggers: watchpoints

A *watchpoint* is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Software Watchpoints:

- Put a breakpoint at **every instruction** (ouch!)
- Check the current value of **L** against a stored value
- If different, give interactive debugging prompt
- If not, set next breakpoint and continue (single-step)

Hardware Watchpoints:

Debuggers: watchpoints

A *watchpoint* is like a breakpoint, but it stops execution after **any instruction** changes the value at location **L**

Software Watchpoints:

- Put a breakpoint at **every instruction** (ouch!)
- Check the current value of **L** against a stored value
- If different, give interactive debugging prompt
- If not, set next breakpoint and continue (single-step)

Hardware Watchpoints:

- Special register holds **L**: if the value at address **L** ever changes, the CPU raises an exception

Related tool: profilers

Note: from here on, this is **NOT** fair game for the midterm/final - we did not get to these slides in class!

NOT FAIR GAME FOR EXAMS

Related tool: profilers

Definition: A ***profiler*** is a performance analysis tool that measures the frequency and duration of function calls as a program runs.

NOT FAIR GAME FOR EXAMS

Related tool: profilers

Definition: A ***profiler*** is a performance analysis tool that measures the frequency and duration of function calls as a program runs.

- **Interpreted languages** provide special hooks for profiling

NOT FAIR GAME FOR EXAMS

Related tool: profilers

Definition: A ***profiler*** is a performance analysis tool that measures the frequency and duration of function calls as a program runs.

- **Interpreted languages** provide special hooks for profiling
  - You **register a function** that will get called whenever the target program calls a method, loads a class, allocates an object, etc. (cf. signal handlers)

NOT FAIR GAME FOR EXAMS

Related tool: profilers

Definition: A ***profiler*** is a performance analysis tool that measures the frequency and duration of function calls as a program runs.

- **Interpreted languages** provide special hooks for profiling
  - You **register a function** that will get called whenever the target program calls a method, loads a class, allocates an object, etc. (cf. signal handlers)
- Alternative: use signals directly (called ***sampling***)

NOT FAIR GAME FOR EXAMS

Related tool: profilers

Definition: A ***profiler*** is a performance analysis tool that measures the frequency and duration of function calls as a program runs.

- **Interpreted languages** provide special hooks for profiling
  - You **register a function** that will get called whenever the target program calls a method, loads a class, allocates an object, etc. (cf. signal handlers)
- Alternative: use signals directly (called ***sampling***)
  - Ask the OS to **send you a signal** every X seconds (see `alarm(2)`)

NOT FAIR GAME FOR EXAMS

Related tool: profilers

Definition: A ***profiler*** is a performance analysis tool that measures the frequency and duration of function calls as a program runs.

- **Interpreted languages** provide special hooks for profiling
  - You **register a function** that will get called whenever the target program calls a method, loads a class, allocates an object, etc. (cf. signal handlers)
- Alternative: use signals directly (called ***sampling***)
  - Ask the OS to **send you a signal** every X seconds (see `alarm(2)`)
  - In the signal handler you determine the value of the target **program counter** and append it to a growing list file

NOT FAIR GAME FOR EXAMS

Related tool: profilers

Definition: A ***profiler*** is a performa
frequency and duration of functio

- **Interpreted languages** provid
  - You **register a function** tha
    program calls a method, lo
    (cf. signal handlers)
- Alternative: use signals directly (called ***sampling***)
  - Ask the OS to **send you a signal** every X seconds (see `alarm(2)`)
  - In the signal handler you determine the value of the target **program counter** and append it to a growing list file

This explanation of **sampling** leaves out some things:

- need to map PC values back to procedure names
- need to sum up map results
- sampling is cheap but can miss periodic behavior

Debugging: takeaways

- Debugging is a lot easier when you treat it as a science, rather than an art
- printf debugging and logging are good for determining what causes failures after the fact
- delta debugging is a semi-automated approach to formalizing the abstract debugging problem
  - useful way of thinking about how to debug anything
  - try `git bisect`
- debuggers are fantastic when you want to understand a program's internal state