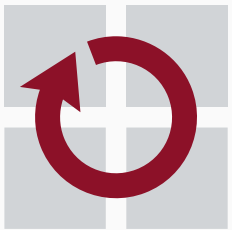
Übungen zu Systemprogrammierung 1

Ü2 – Sortieren und Tooling

Sommersemester 2025

Luis Gerhorst, Thomas Preisner, Tobias Häberlein, Jürgen Kleinöder

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg



Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

Agenda

2.1 Git – Teil 2

2.2 Dyn. Speicherverwaltung – Teil 2

2.3 Fehlerbehandlung

2.4 Generisches Sortieren

2.5 Ein- und Ausgabe

2.6 valgrind: Debuggen von Speicherfehlern

2.7 Aufgabe 2: wsort

2.8 Gelerntes anwenden

Versionierungsschema

- Git gibt jedem Commit eindeutigen Commit-Hash
- spezielle Revisionsschlüsselwörter
  - HEAD: aktuelle Version (lokal)
- `git show <commit-hash>`: zeige Commit an
  `git show`: zeige HEAD
- Online-Ressourcen
  - `https://learngitbranching.js.org/`
  - `https://education.github.com/git-cheat-sheet-education.pdf`
  - `https://www.w3schools.com/git/default.asp`

Gruppenarbeit

- Ab dieser Abgabe: Zusammenarbeit mit Partner
- Anlegen des Repositorys mit:

```
/proj/i4sp1/bin/mkrepo wsort <partner...>
```

- Nur einmal notwendig
- Nutzung desselben Repositorys von allen Teammitgliedern

Gemeinsames Arbeiten mit Git

- Änderungen absprechen, gleichzeitiges Arbeiten vermeiden
- Hochladen mit `git push`
- Herunterladen mit `git pull`
- Gleichzeitige Änderungen führen zu (lösbaren) “Fehlern”
  - **push:** Gleichzeitige Änderungen
    → `failed to push some refs`
    → `git pull`
  - **pull:** Änderungen an der gleichen Stelle
    → `Need to specify how to reconcile divergent branches.`
    → *Merge-Konflikt* auflösen (siehe Appendix)

Agenda

2.1 Git – Teil 2

2.2 Dyn. Speicherverwaltung – Teil 2

2.3 Fehlerbehandlung

2.4 Generisches Sortieren

2.5 Ein- und Ausgabe

2.6 valgrind: Debuggen von Speicherfehlern

2.7 Aufgabe 2: wsort

2.8 Gelerntes anwenden

Dynamische Speicherverwaltung – Teil 2

Verändern der Größe von Feldern, welche durch `malloc(3p)`/`calloc(3p)` erzeugt worden sind:

```
int *numbers = malloc(n * sizeof(int));
if (numbers == NULL) {
  // Fehlerbehandlung
}
... // Speicherbedarf gestiegen
int *neu = realloc(numbers, (n+10) * sizeof(*neu /* == int */));
if (neu == NULL) {
  // Fehlerbehandlung
  free(numbers);
  return -1; // oder: exit(EXIT_FAILURE), dann free() nicht benötigt
}
numbers = neu;
```

Puhh, kompliziert!

- Nur in seltenen Spezialfällen nötig (e.g. `wsort`)
- Empfehlung: Vermeiden wenn es geht
  $\Rightarrow$ Speicherleaks unwahrscheinlicher

Agenda

2.1 Git – Teil 2

2.2 Dyn. Speicherverwaltung – Teil 2

2.3 Fehlerbehandlung

2.4 Generisches Sortieren

2.5 Ein- und Ausgabe

2.6 valgrind: Debuggen von Speicherfehlern

2.7 Aufgabe 2: wsort

2.8 Gelerntes anwenden

Fehlerbehandlung

- Fehler können aus verschiedensten Gründen auftreten:
  - Systemressourcen erschöpft (`malloc(3p)` schlägt fehl; Festplatte voll)
  - Fehlerhafte Benutzereingaben (`fopen(3p)` schlägt fehl)
  - Transiente Fehler: z. B. nicht erreichbarer Server
  - …
- Gute Software **erkennt Fehler**, führt eine **angebrachte Behandlung** durch und gibt eine **aussagekräftige Fehlermeldung** aus
- Kann ein Programm trotz eines Fehlers sinnvoll weiterlaufen?
  - Beispiel 1: Benutzer gibt ungültige URL in den Browser ein
    - Fehlerbehandlung: Fehlermeldung anzeigen
    - Programm (Browser) läuft weiter
  - Beispiel 2: Kopierprogramm: Öffnen der Quelldatei schlägt fehl
    - Fehlerbehandlung: Fehlermeldung anzeigen
    - Programm beenden da kopieren nicht möglich

Fehlerbehandlung: Beendigung des Programms

- Fehler verhindert sinnvolles Weiterarbeiten?

⇒ Programm beenden (`exit(3p)`), Programmabbruch anzeigen

- Signalisierung des Fehlers an Aufrufer des Programms über Exitstatus
  - Exitstatus `==` `0`: zeigt erfolgreiche Programmausführung an
  - Exitstatus `!=` `0`: zeigt Fehler bei der Ausführung an
    - Bedeutung des entsprechenden Wertes ist nicht standardisiert
    - Manchmal enthält Man-Page Informationen über Bedeutung des Exitstatus
- POSIX bietet vordefinierte Makros für den Exitstatus an:
  - `EXIT_SUCCESS`
  - `EXIT_FAILURE`

  ⇒ Beispielnutzung: `exit(EXIT_FAILURE);`
- Exitstatus des letzten Befehls ist in der Shell-Variable `$?` gespeichert

Erkennung und Ausgabe von Fehlern

- Fehler treten häufig in Funktionen der C-Bibliothek auf
  - erkennbar i. d. R. am Rückgabewert (Man-Page, **RETURN VALUES**)
- Die Fehlerursache wird über die globale Variable `errno` übermittelt
  - Der Wert `errno = 0` ist reserviert, alles andere ist ein Fehlercode
  - Bibliotheksfunktionen setzen `errno` im Fehlerfall (sonst nicht zwingend)
  - Bekanntmachung im Programm durch Einbinden von `errno.h`
- Fehlercodes als lesbare Strings ausgegeben mit `perror(3p)`

```c
char *mem = malloc(...);      // malloc gibt im Fehlerfall
if (mem == NULL) {            // NULL zurück
    perror("malloc");        // Ausgabe der Fehlerursache
    exit(EXIT_FAILURE);      // Programm mit Fehlercode beenden
}
```

  - `perror(3p)` nur verwenden, wenn die `errno` gesetzt wurde
  - `errno` ist nur **direkt** nach dem Funktionsaufruf gültig
  - mögliche Ausgabe:

```
malloc: Cannot allocate memory
```

Fehlerbehandlung in SP

- Korrekte Fehlerbehandlung steht in SP im Fokus!
- **Alle** Funktionen müssen auf mögliche Fehler geprüft werden
  - Außer Funktionen die nicht fehlschlagen können (Man-Page, **ERRORS**)
  - Passende Fehlermeldung
    - `errno` gesetzt: Grund mit `perror(3p)` ausgeben
    - Sonst: Eigene Meldung mit `fprintf(3p)` auf **`stderr`** ausgeben
    - Ausnahme: Bibliotheken erzeugen *keine* Fehlermeldungen, sondern geben Fehlercode zurück
  - Passende Fehlerbehandlung
    - Permanenter Fehler: `exit(3p)` mit `EXIT_FAILURE`
    - Sonst: Passend weiter arbeiten (`return`, `continue`, etc.)
    - Ausnahme: Bibliotheken (z.B. lilo) beenden das Programm *nicht*, sondern geben Fehlercode zurück
- Fehlerbehandlung gibt Punkte
  - Man-Pages der verwendeten Funktionen lesen
  - Passende Fehlerbehandlung einbauen, meist `perror(3p)` plus `exit(3p)`

Agenda

2.1 Git – Teil 2

2.2 Dyn. Speicherverwaltung – Teil 2

2.3 Fehlerbehandlung

2.4 Generisches Sortieren

2.5 Ein- und Ausgabe

2.6 valgrind: Debuggen von Speicherfehlern

2.7 Aufgabe 2: wsort

2.8 Gelerntes anwenden

Generisches Sortieren mit qsort

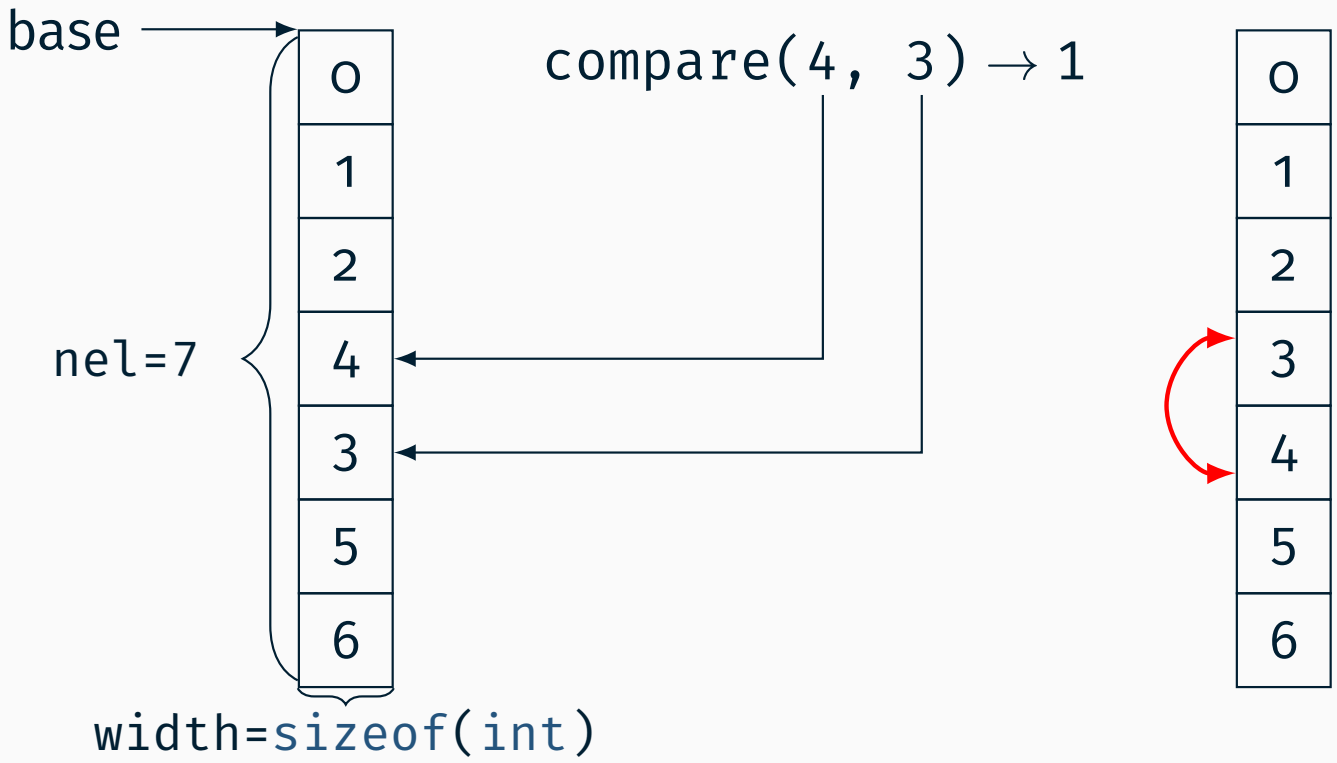
- Vergleich nahezu beliebiger Daten
  - alle Daten müssen die gleiche Größe haben
- `qsort` weiß nicht, was es sortiert (wie der Vergleich zu bewerkstelligen ist)
  - Aufrufer stellt Routine zum Vergleich zweier Elemente zur Verfügung
  - Fachbegriff für dieses Programmierschema: *Callback*
- Prototyp aus `stdlib.h`:

```
void qsort(void *base,
           size_t nel,
           size_t width,
           int (*compare) (const void *, const void *));
```

  - `base`: Zeiger auf das erste Element des zu sortierenden Feldes
  - `nel`: Anzahl der Elemente im zu sortierenden Feld
  - `width`: Größe eines Elements
  - `compare`: Vergleichsfunktion

Arbeitsweise von qsort

- `qsort` vergleicht je zwei Elemente mit Hilfe der Vergleichsfunktion



  - Die Funktion vergleicht die beiden Elemente und liefert:
    - `< 0` falls Element 1 kleiner gewertet wird als Element 2
    - `0` falls Element 1 und Element 2 gleich gewertet werden
    - `> 0` falls Element 1 größer gewertet wird als Element 2

Vergleichsfunktion

```
void qsort(..., int (*compare) (const void *, const void *));
```

- Die Vergleichsfunktion erhält Zeiger auf Feldelemente („Zeiger in das Array“)
- `qsort(3p)` kennt den tatsächlichen Datentyp nicht
  → Prototyp ist generisch mit `void`-Zeigern parametrisiert
- `const`-Zusicherung: Die Vergleichsfunktion darf das Array nicht verändern

Beispiel: Vergleichsfunktion für Array aus `int`

```
int compare(const void *a, const void *b) {
  const int *ia = (const int *) a;
  const int *ib = (const int *) b;
  if (*ia < *ib) {
    return -1;
  } else if (*ia == *ib) {
    return 0;
  } else {
    return +1;
  }
}
```

Vergleichsfunktion: Sortieren eines Arrays aus int*

Ziel: Sortieren eines Arrays aus `int*` anhand der Werte der `int`s

Lösung

```
int compare(const void *a, const void *b) {
  int * const *ia = (int * const *) a;
  int * const *ib = (int * const *) b;
  if (**ia < **ib) {
    return -1;
  } else if (**ia == **ib) {
    return 0;
  } else {
    return +1;
  }
}
```

Website cdecl.org erklärt C Typdeklarationen in natürlicher Sprache.

Agenda

2.1 Git – Teil 2

2.2 Dyn. Speicherverwaltung – Teil 2

2.3 Fehlerbehandlung

2.4 Generisches Sortieren

2.5 Ein- und Ausgabe

2.6 valgrind: Debuggen von Speicherfehlern

2.7 Aufgabe 2: wsort

2.8 Gelerntes anwenden

Zeilenweise Ein- und Ausgabe

- Zeilenweises Lesen

```
char *fgets(char *s, int n, FILE *fp);
```

  - liest Zeichen von Dateikanal `fp` in das Feld `s` bis entweder `n-1` Zeichen gelesen wurden oder `\n` gelesen oder EOF erreicht wurde
  - `s` wird mit `\0` abgeschlossen (`\n` wird nicht entfernt)
  - gibt bei EOF oder Fehler `NULL` zurück, sonst `s`; setzt **`errno`**
  - für `fp` kann **`stdin`** eingesetzt werden, um von der Standardeingabe zu lesen

- Zeilenweises Schreiben

```
int fputs(char *s, FILE *fp);
```

  - schreibt die Zeichen im Feld `s` auf Dateikanal `fp`
  - für `fp` kann auch **`stdout`** oder **`stderr`** eingesetzt werden
  - als Ergebnis wird die Anzahl der geschriebenen Zeichen geliefert
  - gibt EOF bei Fehler zurück

Ein- und Ausgabe: Fehlerbehandlung

- Fehlerbehandlung
  - Funktion wie `fgets(3p)` oder `printf(3p)` aufrufen
  - Falls Rückgabewert Fehler oder EOF anzeigt
    - Mit `ferror(3p)` oder `feof(3p)` prüfen um zu unterscheiden
    - Falls Fehler, passend behandeln (`perror(3p)`, etc.)
- Falls kein `close(3p)`/`fclose(3p)` vorhanden (z. B. beim Schreiben nach **`stdout`**)
  - Vor Beenden des Programms Aufruf von `fflush(3p)` nötig!
  - Funktionen wie `printf(3p)` schreiben nicht sofort, sondern sind gepuffert (Zeilenweise bei **`stdout`**, Blockweise für Dateien)
  - Ohne manuelles „Spülen“ wird Fehler nicht sichtbar

Ein- und Ausgabe: Fehlerbehandlung in SP

- Korrekte Fehlerbehandlung bei Ein-/Ausgabe ist wichtig
  - Lesen
    - Uninitialisierte Variablen nach Lesefehler
    - Mögliche Endlosschleife bei EOF
  - Schreiben
    - Schreibfehler werden ignoriert
    - Bei voller Festplatte wird die Datei nicht (komplett) geschrieben
- Fehlerbehandlung ist bei allen Ein-/Ausgaben nötig, die zur Grundfunktionalität des Programms gehören
  - Gilt für *alle* Ein-/Ausgabe-Funktionen, inklusive `printf(3p)`, `close(3p)`, `fclose(3p)` (Details siehe Man-Pages)
  - Grundfunktionalität geht aus der Aufgabe hervor
  - Keine Fehlerbehandlung nötig bei
    - Debug-/Info-Ausgaben
    - Fehlerbehandlung selbst

Agenda

2.1 Git – Teil 2

2.2 Dyn. Speicherverwaltung – Teil 2

2.3 Fehlerbehandlung

2.4 Generisches Sortieren

2.5 Ein- und Ausgabe

2.6 valgrind: Debuggen von Speicherfehlern

2.7 Aufgabe 2: wsort

2.8 Gelerntes anwenden

valgrind

- Baukasten von Debugging- und Profiling-Werkzeugen
- Für uns relevant: *memcheck*
  - Erkennt Speicherzugriff-Probleme:
    - Nutzung von nicht-initialisiertem Speicher
    - Zugriff auf freigegebenen Speicher
    - Zugriff über das Ende von allozierten Speicherbereichen
    - Aber: Prüft nur den konkret ausgeführten Code-Pfad
- Programm sollte Debug-Symbole enthalten
  - mit GCC-Flag `-g` übersetzen
- **Laufzeitprüfung**: Kann nur Anwesenheit von Fehlern zeigen, nicht aber deren Abwesenheit.

valgrind: Zugriffe auf nicht allozierten Speicher finden

```
=711= Invalid read of size 4
=711=    at 0x804841B: main (test.c:19)
=711= Address 0x0 is not stack'd, malloc'd or (recently) free'd
=711=
=711= Process terminating with default action of signal 11 (SIGSEGV)
=711= Access not within mapped region at address 0x0
```

- In Zeile 19 wird lesend auf die Adresse `0x0` zugegriffen
  `NULL`-Pointer wurde dereferenziert
- Der Prozess wird auf Grund einer Speicherzugriffsverletzung (SIGSEGV) beendet

```
=787= Invalid write of size 1
=787=    at 0x48DC9EC: memcpy (mc_replace_strmem.c:497)
=787=    by 0x80485A2: test_malloc (test.c:57)
=787=    by 0x80484A8: main (test.c:22)
=787=  Address 0x6d1f02d is 0 bytes after a block of size 5 alloc'd
```

- In Zeile 57 wird `memcpy` aufgerufen, welches ein Byte an eine *ungültige* Adresse schreibt

valgrind: Auffinden von nicht freigegebenem Speicher

```
=787= HEAP SUMMARY:
=787=     in use at exit: 5 bytes in 1 blocks
=787=   total heap usage: 1 allocs, 0 frees, 5 bytes allocated
```

- Bei Programmende ist noch ein Speicherbereich (Block) belegt
- Während der Programmausführung wurde einmal `malloc()` und keinmal `free()` aufgerufen
- Mit Hilfe der Optionen `–leak-check=full –show-reachable=yes` wird angezeigt, wo der Speicher angelegt wurde, der nicht freigegeben wurde.

```
=799= 5 bytes in 1 blocks are definitely lost in loss record 1
=799=    at 0x48DAF50: malloc (vg_replace_malloc.c:236)
=799=    by 0x8048576: test_malloc (test.c:52)
=799=    by 0x80484A8: main (test.c:22)
```

- In Zeile 52 wurde der Speicher angefordert
- Im Quellcode Stellen identifizieren, an denen `free()`-Aufrufe fehlen

valgrind: Auffinden uninitialisierten Speichers

```
=799= Use of uninitialised value of size 4
=799=    at 0x4964316: _itoa_word (_itoa.c:195)
=799=    by 0x4967C59: vfprintf (vfprintf.c:1616)
=799=    by 0x496F3DF: printf (printf.c:35)
=799=    by 0x8048562: test_int (test.c:48)
=799=    by 0x8048484: main (test.c:15)
```

- In Zeile 48 wird auf uninitialisierten Speicher zugegriffen
- Mit Hilfe der Option `–track-origins=yes` wird angezeigt, wo der uninitialisierte Speicher angelegt wurde

```
=683= Use of uninitialised value of size 4
=683=    at 0x4964316: _itoa_word (_itoa.c:195)
=683=    by 0x4967C59: vfprintf (vfprintf.c:1616)
=683=    by 0x496F3DF: printf (printf.c:35)
=683=    by 0x8048562: test_int (test.c:48)
=683=    by 0x8048484: main (test.c:15)
=683=  Uninitialised value was created by a stack allocation
=683=    at 0x804846A: main (test.c:10)
```

`valgrind`: Auffinden uninitialisierten Speichers

- Spezialfall: Zugriff auf uninitialisierten Speicher bei Bedingungsprüfungen

```
=683= Conditional jump or move depends on uninitialised value(s)
=683=    at 0x48DC0E7: __GI_strlen (mc_replace_strmem.c:284)
=683=    by 0x496886E: vfprintf (vfprintf.c:1617)
=683=    by 0x496F3DF: printf (printf.c:35)
=683=    by 0x8048562: test_int (test.c:48)
=683=    by 0x8048484: main (test.c:15)
```

Agenda

2.1 Git – Teil 2

2.2 Dyn. Speicherverwaltung – Teil 2

2.3 Fehlerbehandlung

2.4 Generisches Sortieren

2.5 Ein- und Ausgabe

2.6 valgrind: Debuggen von Speicherfehlern

2.7 Aufgabe 2: wsort

2.8 Gelerntes anwenden

Aufgabe 2: wsort

- Lernziele
  - Einlesen von der Standardeingabe (**`stdin`**)
  - Umgang mit dynamischer Speicherverwaltung (`realloc(3p)`)
  - Verwendung von Debug-Werkzeugen
- Ausprobieren eures Programmes
  - Beispiel-Eingabedateien im CIP unter /proj/i4sp1/pub/aufgabe2/
  - Vergleichen der Ausgabe mit vorgegebenem Binary
    - Hier am Beispiel der `wlist0` (alternativ: `kompare`, `meld`)

```
$ ./wsort < /proj/i4sp1/pub/aufgabe2/wlist0 > wlist0.mine
$ /proj/i4sp1/pub/aufgabe2/wsort < \
            /proj/i4sp1/pub/aufgabe2/wlist0 > wlist0.spteam
$ diff -s -u wlist0.mine wlist0.spteam
```

Agenda

2.1 Git – Teil 2

2.2 Dyn. Speicherverwaltung – Teil 2

2.3 Fehlerbehandlung

2.4 Generisches Sortieren

2.5 Ein- und Ausgabe

2.6 valgrind: Debuggen von Speicherfehlern

2.7 Aufgabe 2: wsort

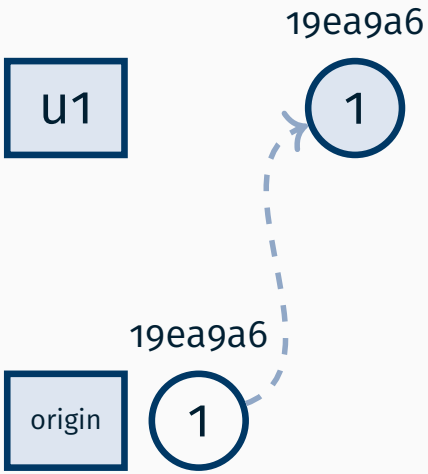
2.8 Gelerntes anwenden

Aktive Mitarbeit!

„Aufgabenstellung“

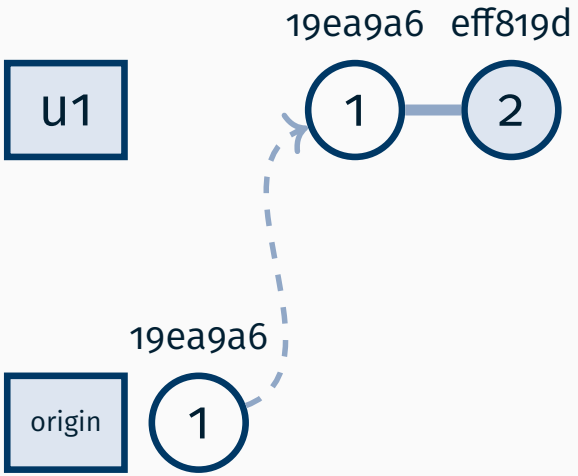
- `isort` Programm, welches ein Array von Zufallszahlen sortiert

Gemeinsames Arbeiten mit Git im Detail



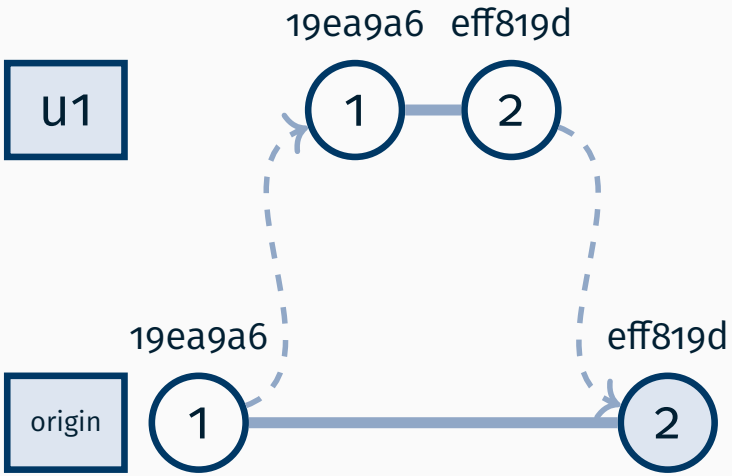

```
~/nutzer1 $ git remote -v
origin gitgitlab.cs.fau.de:i4sp/ss23/test.git (fetch)
origin gitgitlab.cs.fau.de:i4sp/ss23/test.git (push)
```

Gemeinsames Arbeiten mit Git im Detail



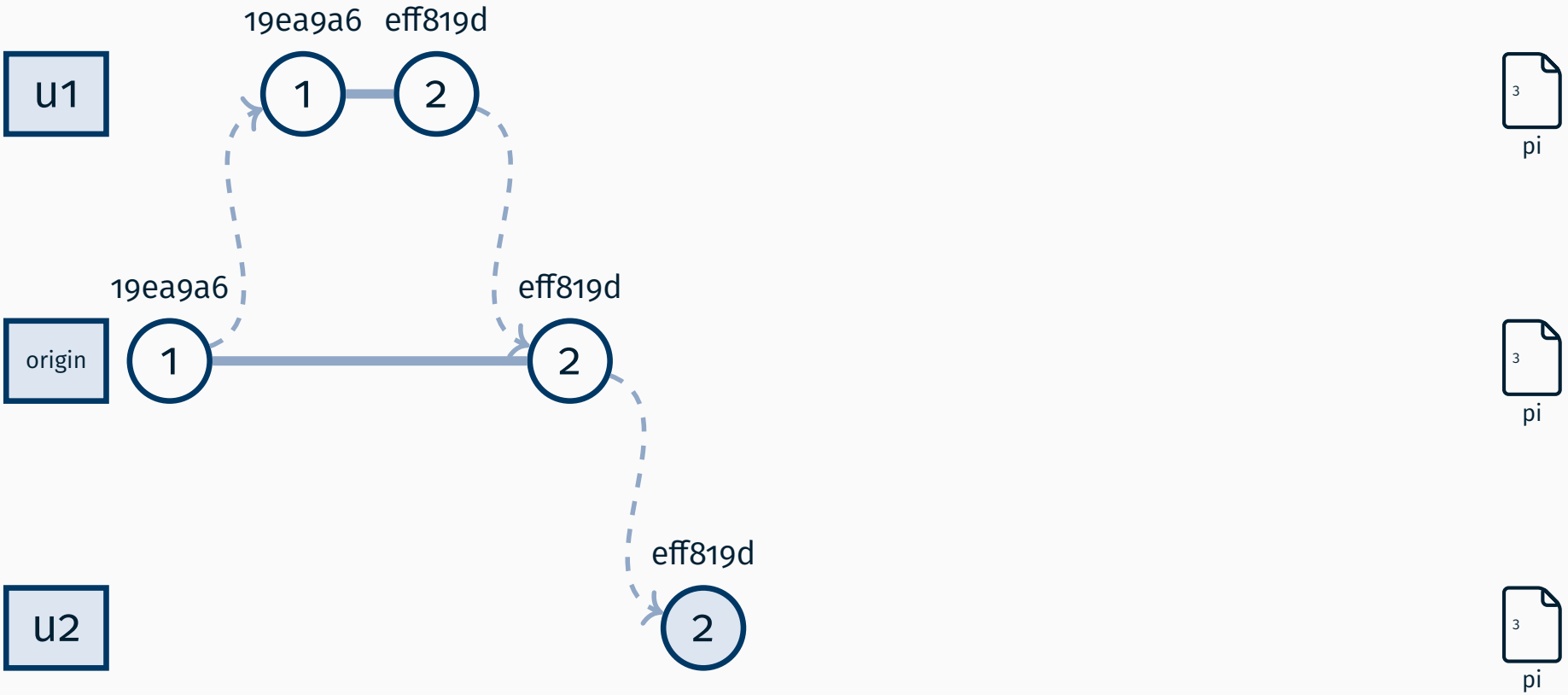
```
~/nutzer1 $ echo "3" > pi
~/nutzer1 $ git add pi
~/nutzer1 $ git commit -m "pi ist 3"
[main eff819d] pi ist 3
 1 file changed, 1 insertion(+)
 create mode 100644 pi
```

Gemeinsames Arbeiten mit Git im Detail



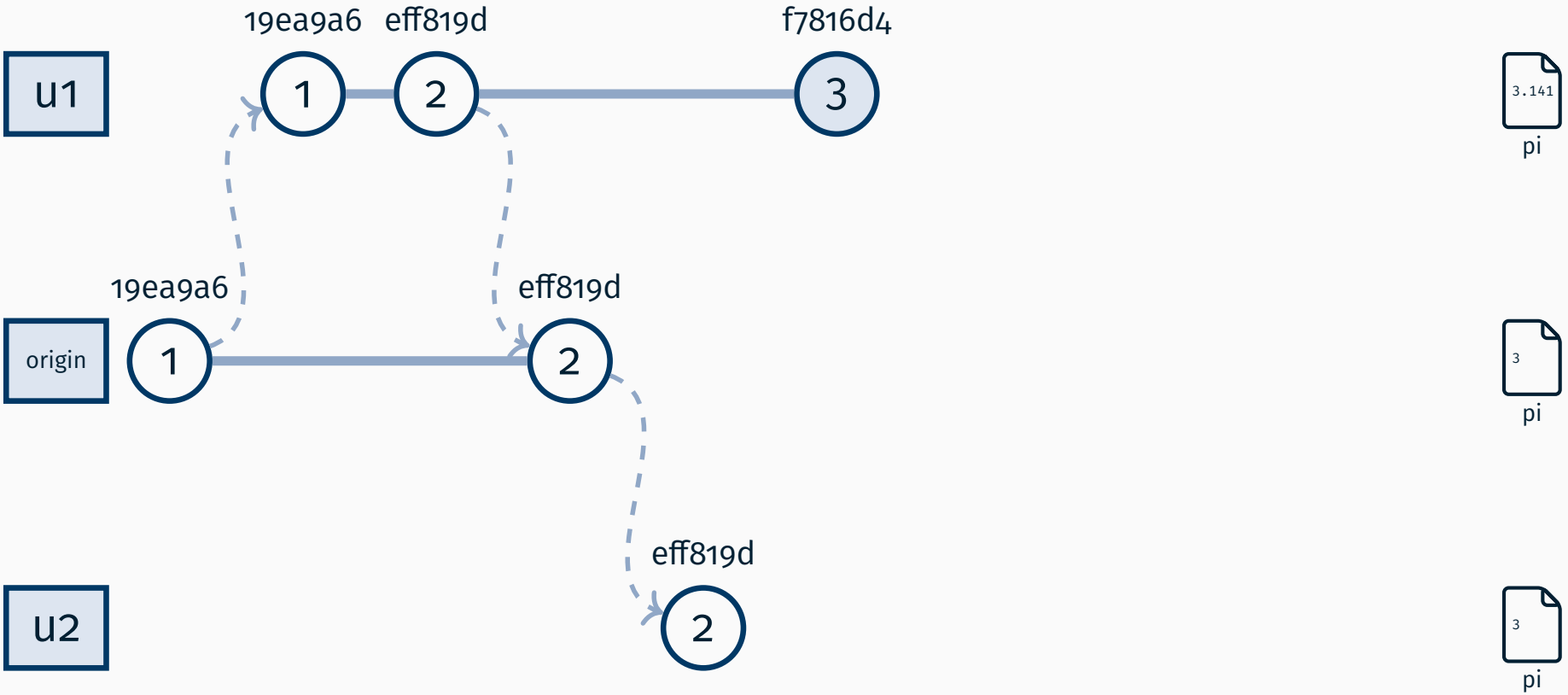
```
~/nutzer1 $ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 261 bytes | 261.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
To gitlab.cs.fau.de:i4sp/ss23/test.git
   19ea9a6..eff819d  main -> main
```

Gemeinsames Arbeiten mit Git im Detail



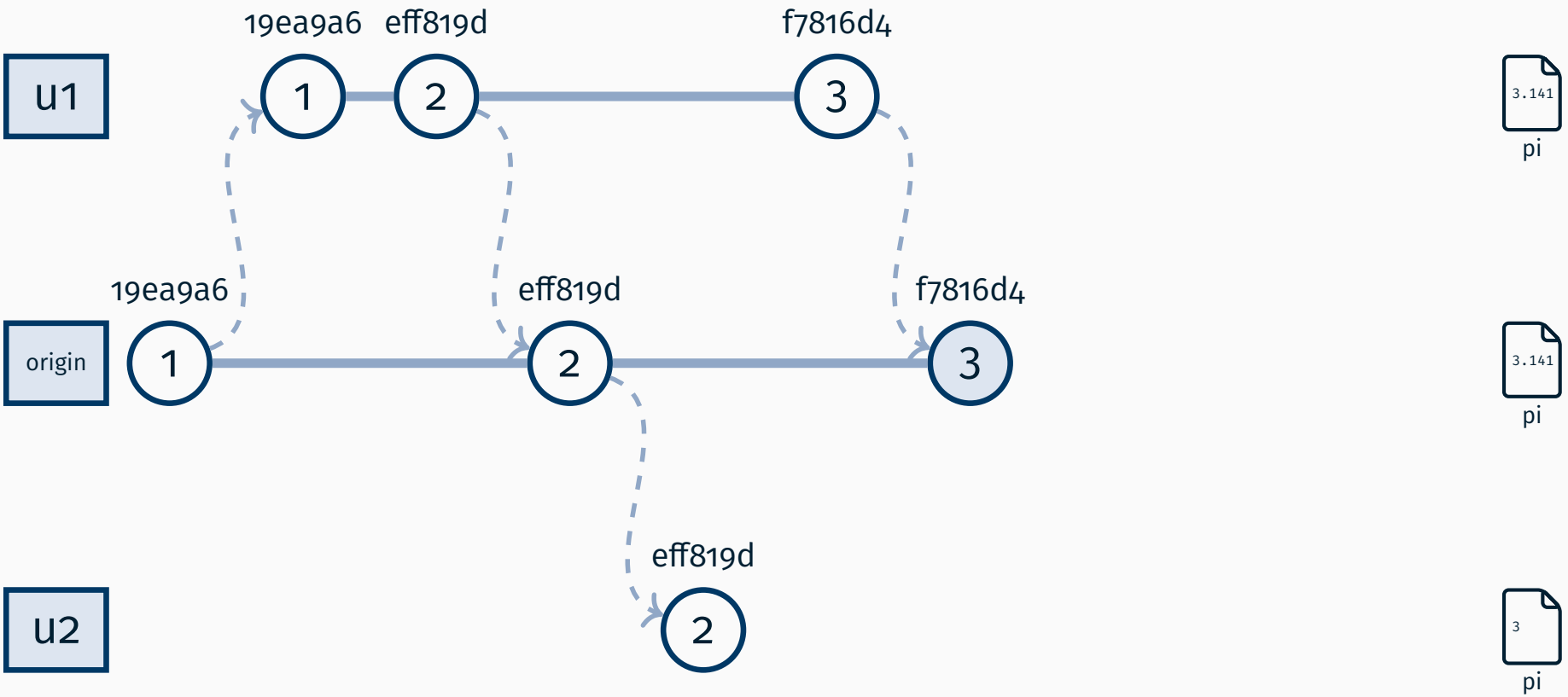

```
~ $ git clone gitgitlab.cs.fau.de:i4sp/ss23/test.git nutzer2
Cloning into 'nutzer2'...
(...)
~ $ cd nutzer2
~/nutzer2 $ cat pi
3
```

Gemeinsames Arbeiten mit Git im Detail



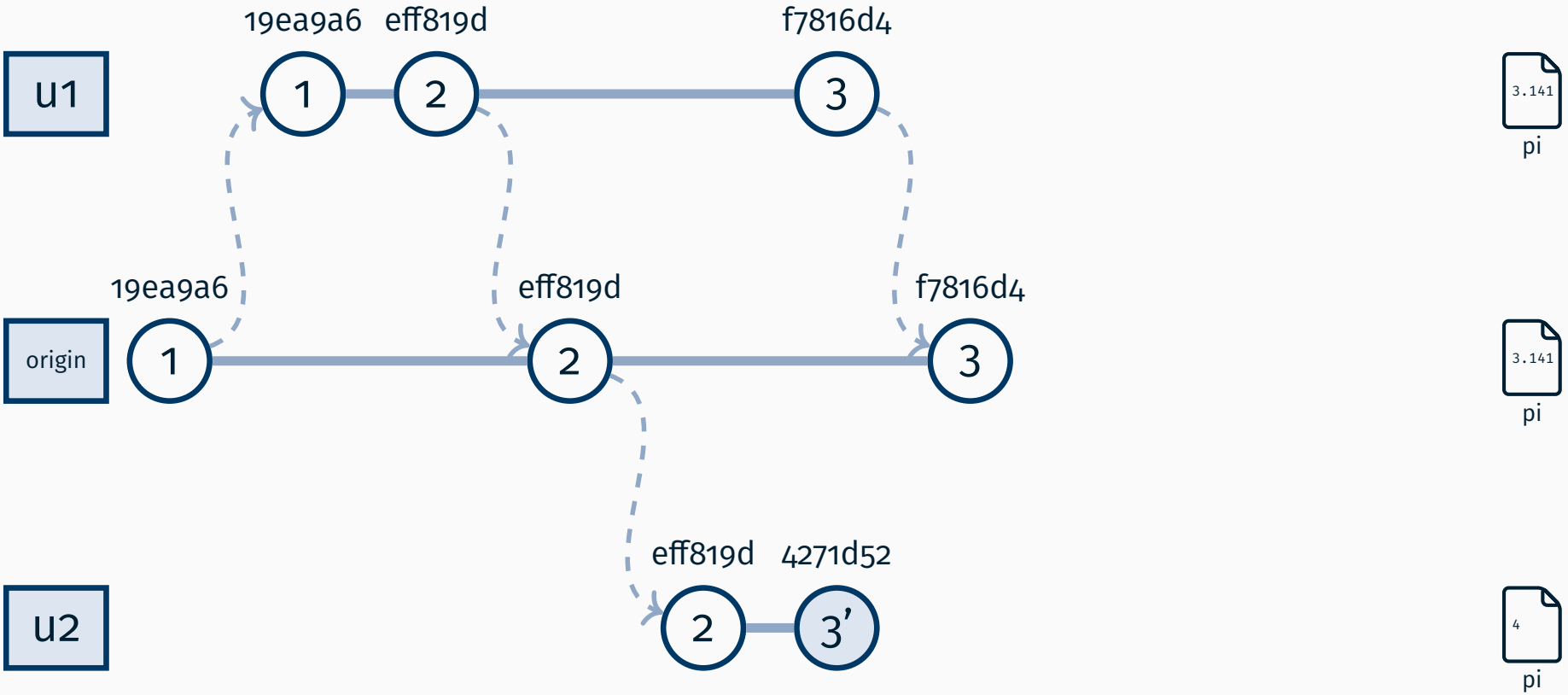

```
~/nutzer1 $ echo "3.141" > pi
~/nutzer1 $ git add pi
~/nutzer1 $ git commit -m "pi ist nicht 3"
[main f7816d4] pi ist nicht 3
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Gemeinsames Arbeiten mit Git im Detail



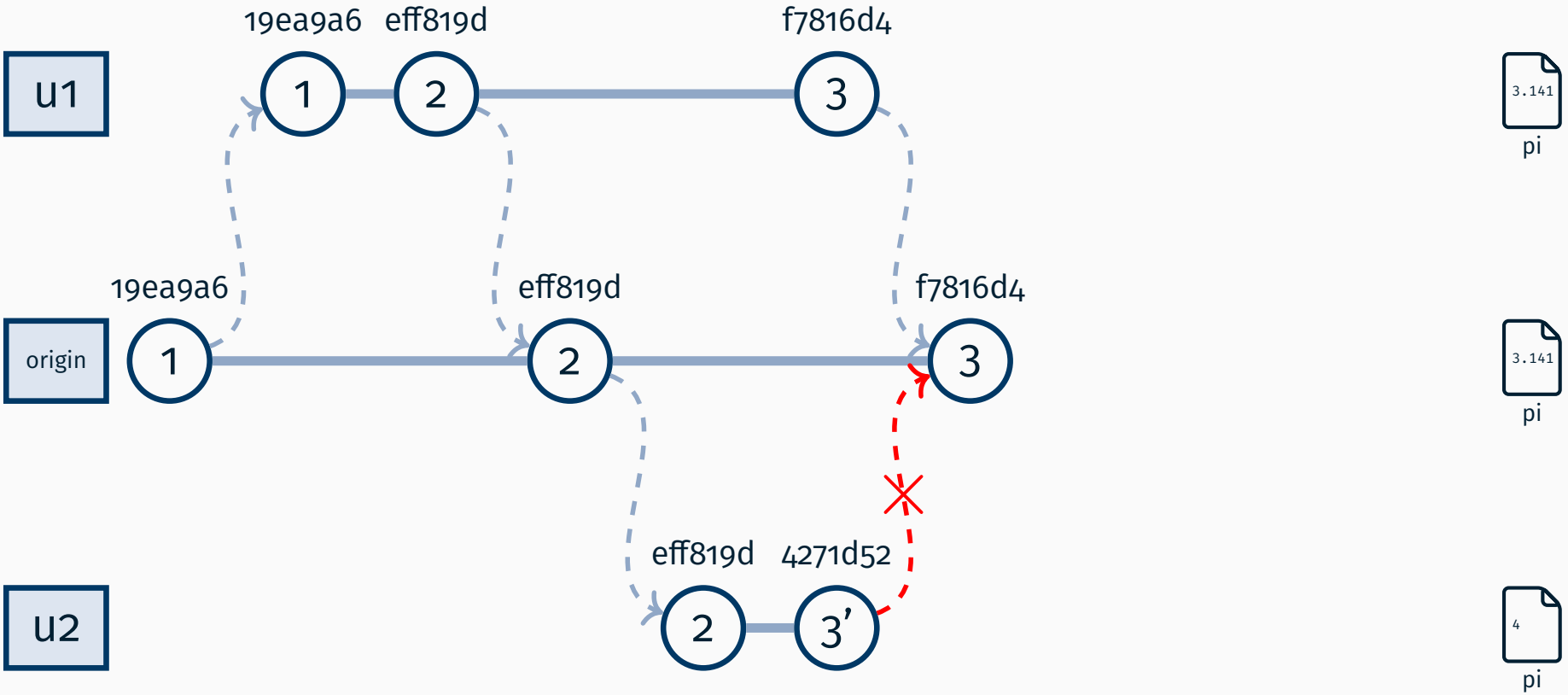

```
~/nutzer1 $ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 261 bytes | 261.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
To gitlab.cs.fau.de:i4sp/ss23/test.git
   eff819d..f7816d4  main -> main
```

Gemeinsames Arbeiten mit Git im Detail



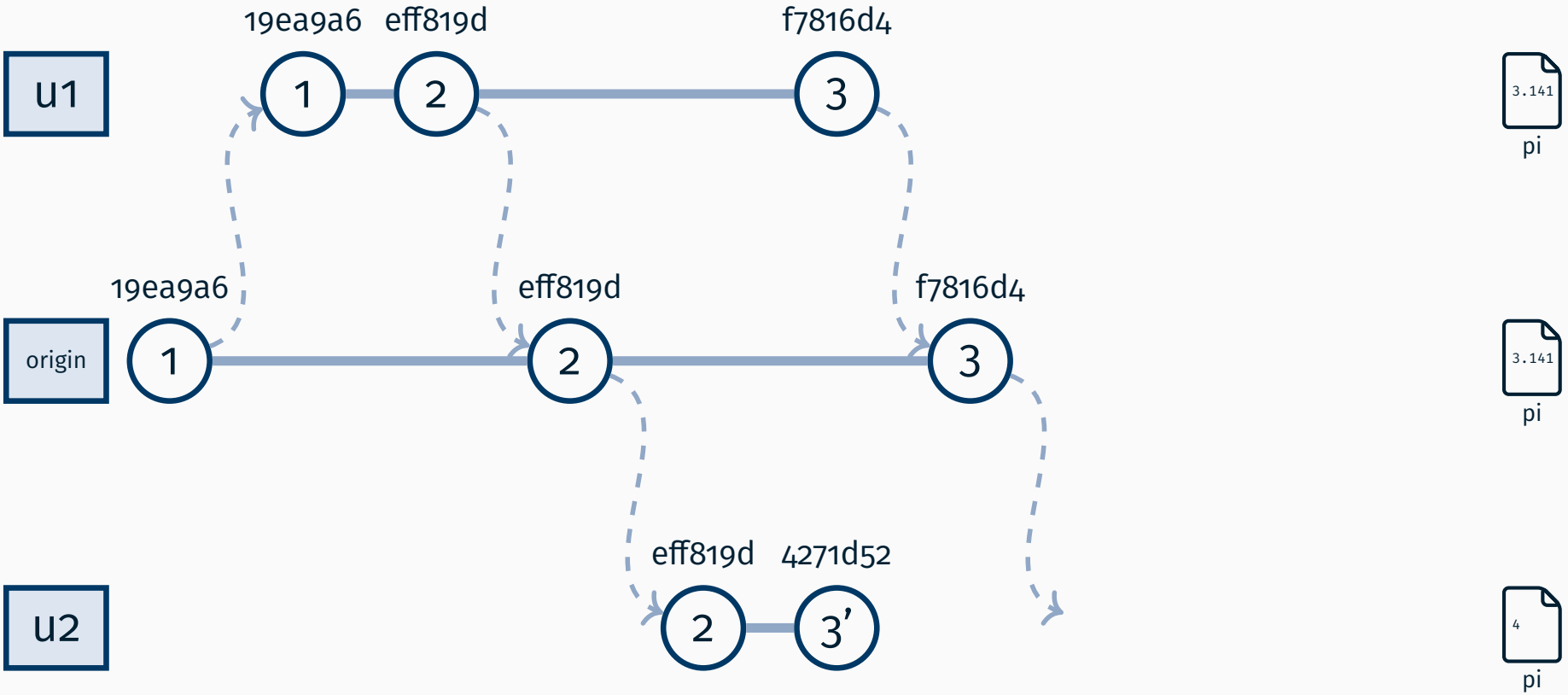

```
~/nutzer2 $ echo 4 > pi
~/nutzer2 $ git add pi
~/nutzer2 $ git commit -m "pi ist 4"
[main 4271d52] pi ist 4
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Gemeinsames Arbeiten mit Git im Detail



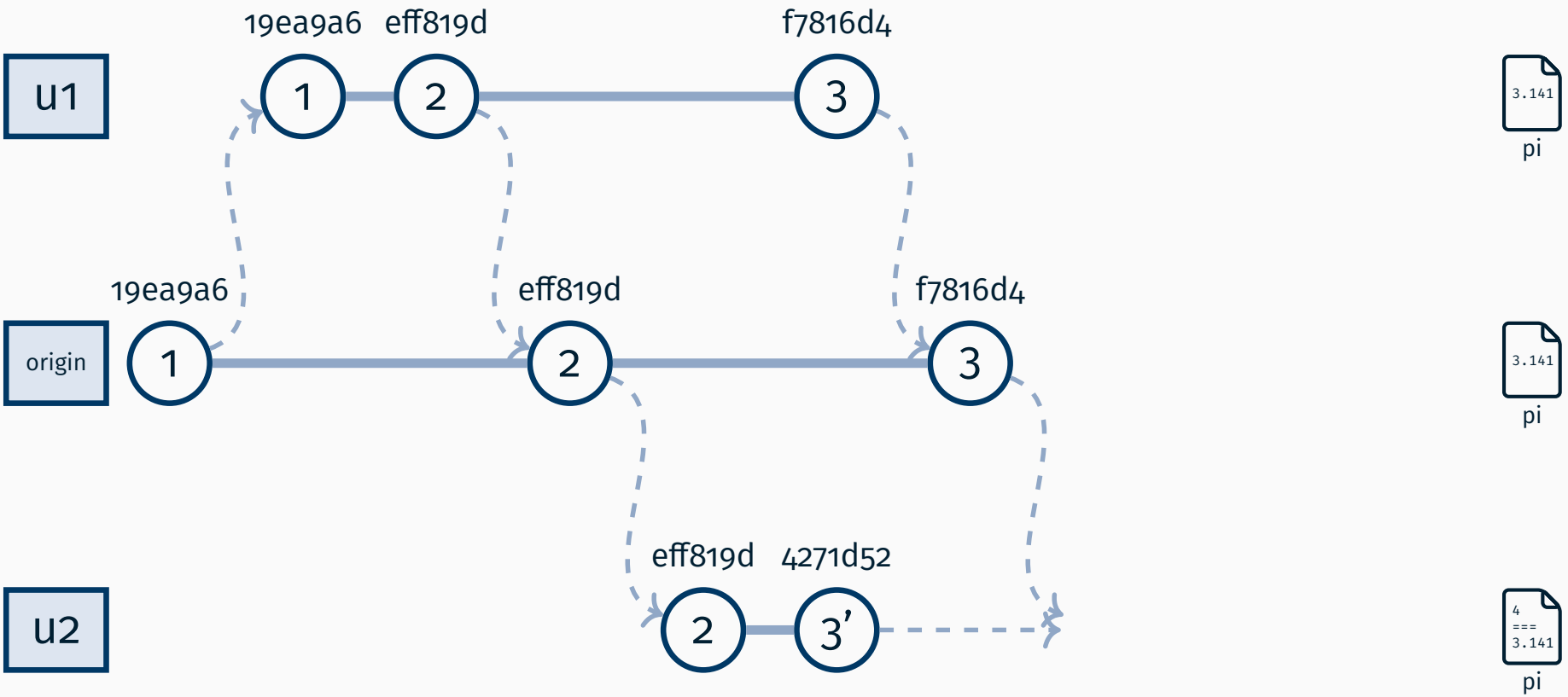

```
~/nutzer2 $ git push
To gitlab.cs.fau.de:i4sp/ss23/test.git
  [rejected]        main -> main (fetch first)
error: failed to push some refs to 'gitlab.cs.fau.de:i4sp/ss23/test.git'
```

Gemeinsames Arbeiten mit Git im Detail



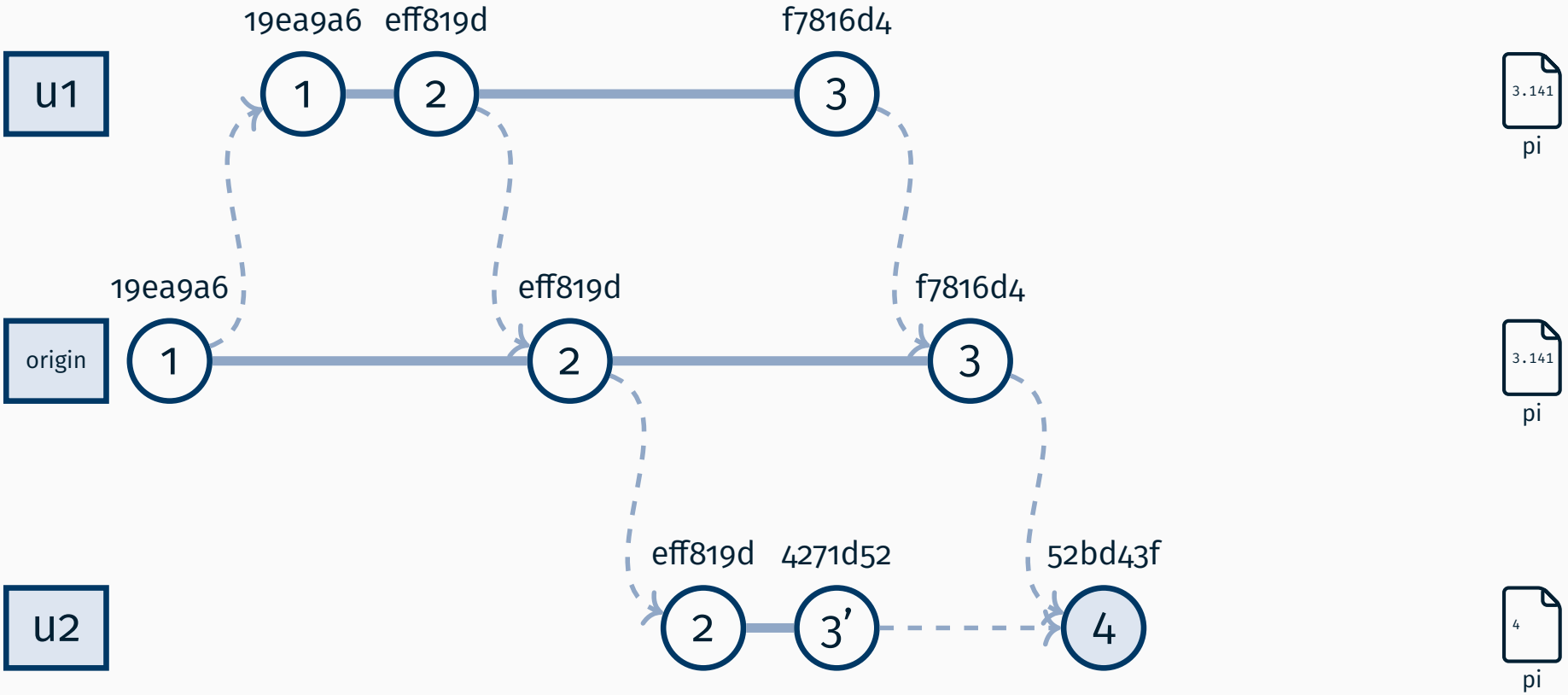

```
~/nutzer2 $ git pull
(...)
From gitlab.cs.fau.de:i4sp/ss23/test
   eff819d..f7816d4  main       -> origin/main
fatal: Need to specify how to reconcile divergent branches.
```

Gemeinsames Arbeiten mit Git im Detail



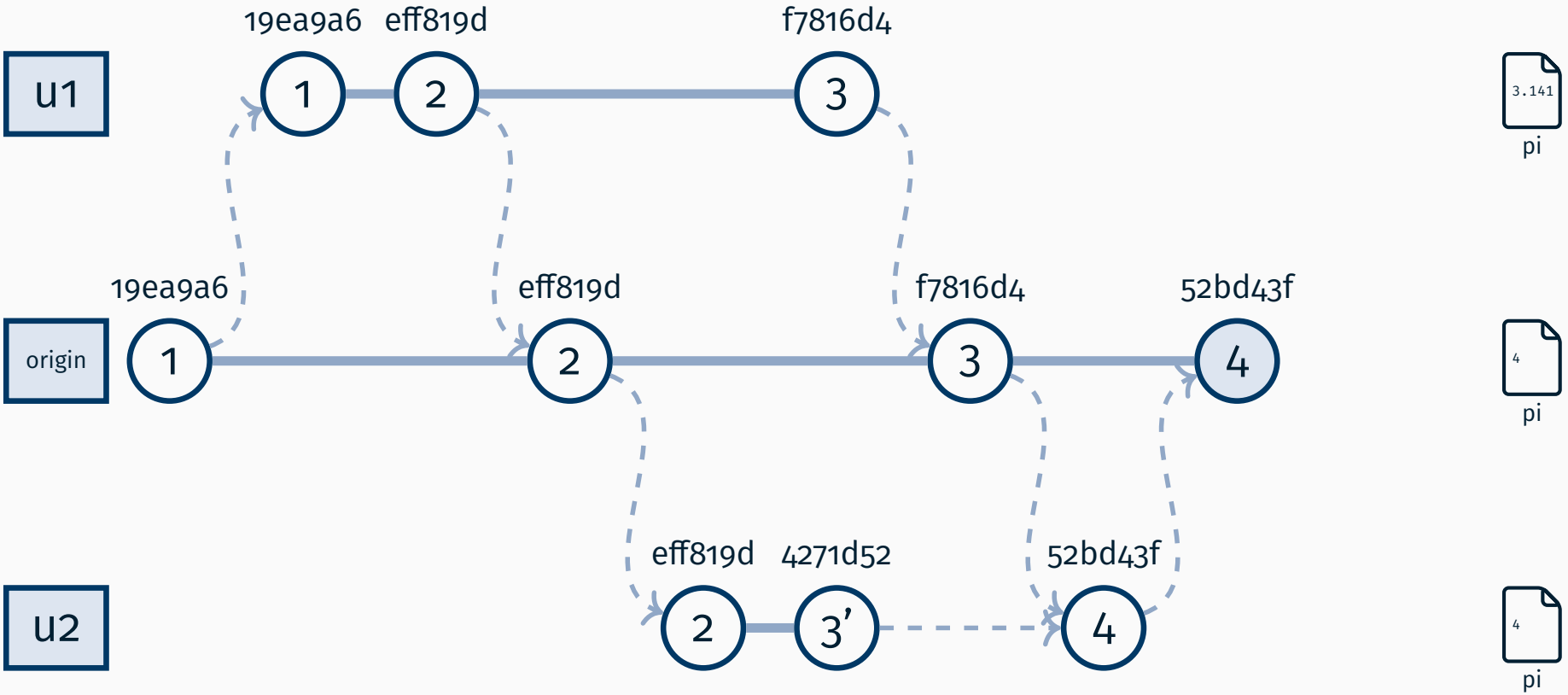

```
~/nutzer2 $ git merge origin/HEAD
~/nutzer2 $ cat pi
<<<<<<< HEAD
4
=======
3.141
>>>>>>> origin/HEAD
```

Gemeinsames Arbeiten mit Git im Detail



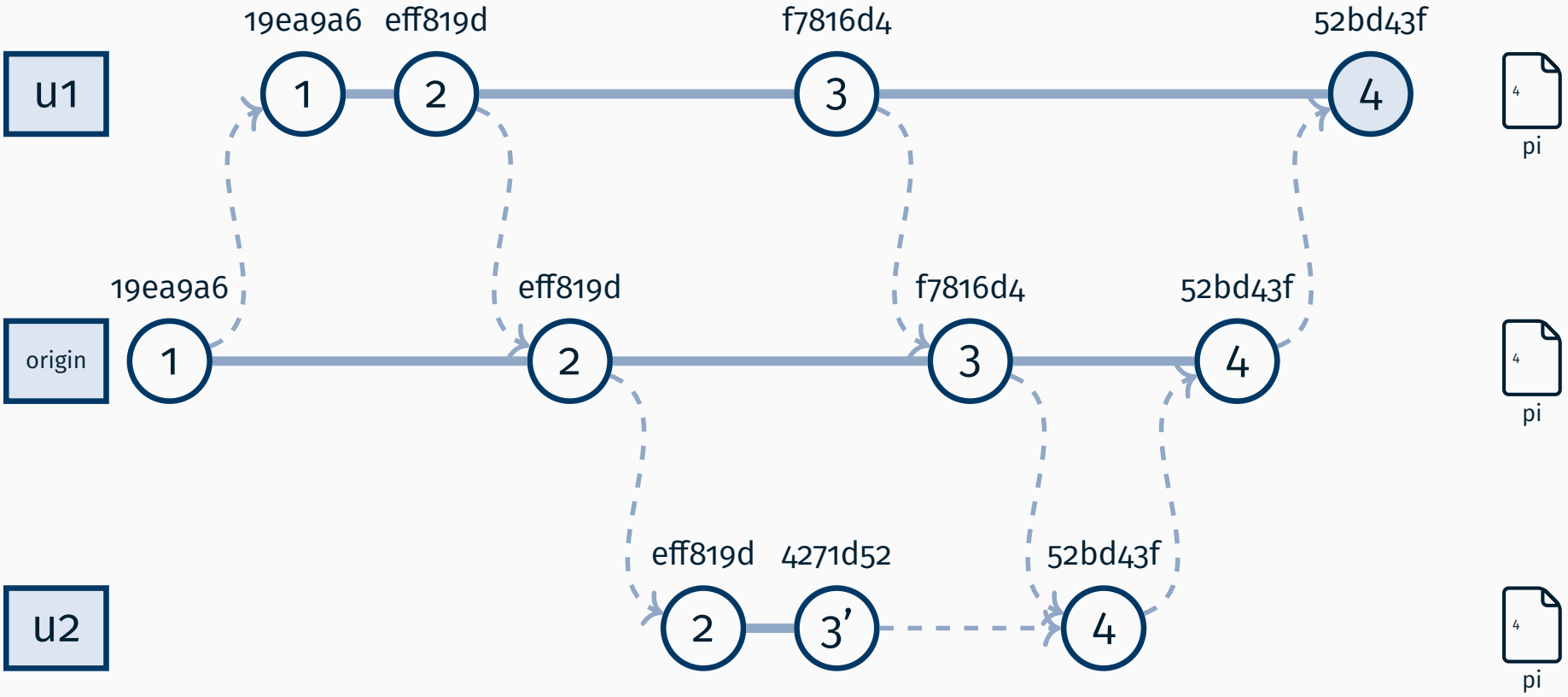

```
~/nutzer2 $ echo "pi=4" > pi
~/nutzer2 $ git add pi
~/nutzer2 $ git commit -m "merge pi"
[main 52bd43f] merge pi
```

Gemeinsames Arbeiten mit Git im Detail




```
~/nutzer2 $ git push
(...)
To gitlab.cs.fau.de:i4sp/ss23/test.git
   f7816d4..52bd43f  main -> main
```

Gemeinsames Arbeiten mit Git im Detail




```
~/nutzer1 $ git pull
(...)
From gitlab.cs.fau.de:i4sp/ss23/test
   f7816d4..52bd43f  main       -> origin/main
Updating f7816d4..52bd43f
Fast-forward
 pi | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```