On Testing Multithreaded Programs

Lukáš Petrlík

Department of Computer Science and Engineering
Faculty of Applied Sciences
ZČU Plzeň

DSS 2004

0. Overview

1. Definition of Basic Terms
2. Issues in Serial Testing
3. Testing Concurrent Programs
4. Test Adequacy Criteria
5. Improving Test Yield
6. Research Plan

1. What is Software Testing

Basic Terms:

- **error** – a mistake made by a developer
- **fault** – errors lead to one or more faults
  - faults are located in the the program
  - fault is the difference between the incorrect program and a correct version
- **failure** – difference between the behavior of the incorrect program and a correct version
  - a fault may lead to zero or more failures

Software Testing and Debugging

Testing (*dynamic testing*) is the process of executing a program with the intent of exhibiting faults.

Monitoring is the process of gathering information about a program's execution.

Debugging is the activity of (1) determining the location and nature of the fault, (2) fixing the fault.

Testing and debugging takes about 50% of development time!

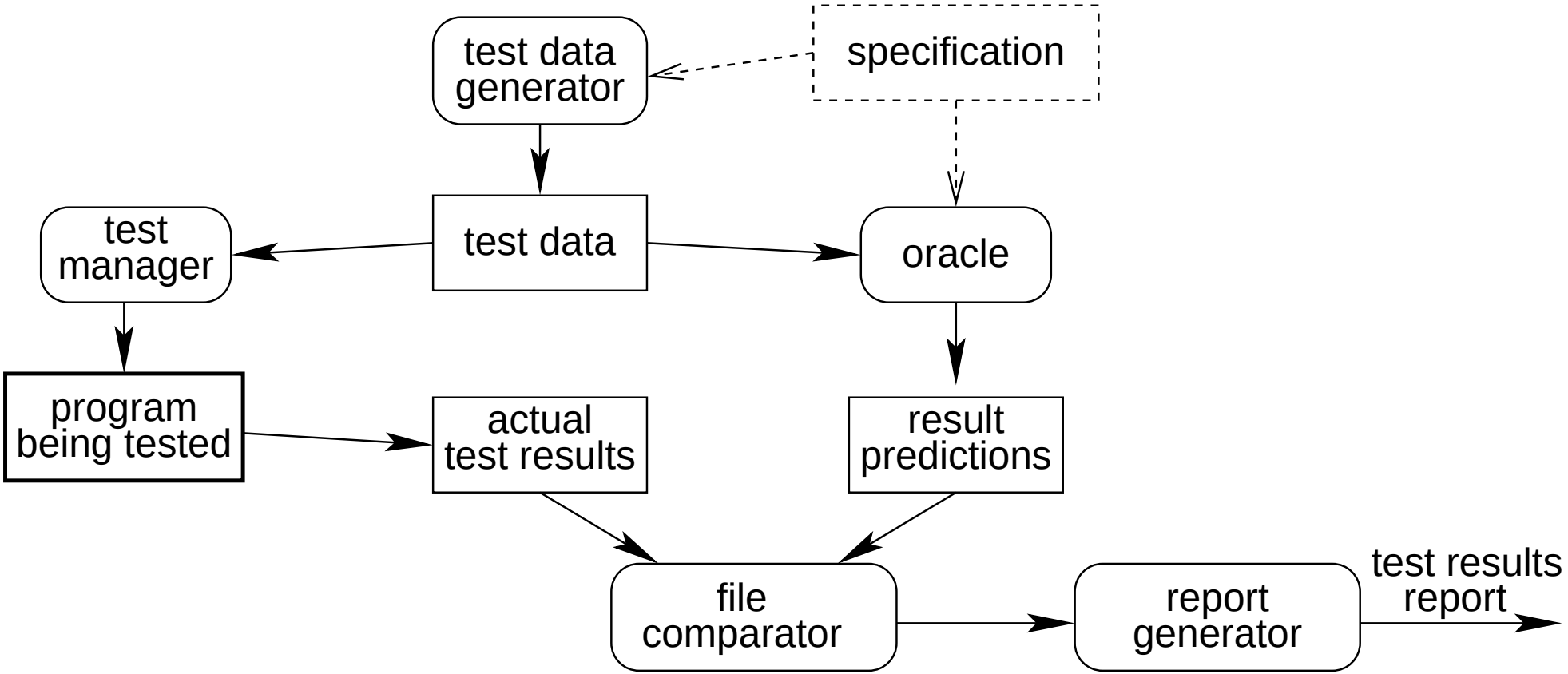
Testing must cost as little as possible $\Rightarrow$ tests should be automatic.

Automated Software Testing

The players:

- **test manager** – controls the test executions
- **test data generator** – generates input data
- **oracle** – generates expected output
- **file comparator** – compares expected and actual output
- **report generator** – reports results

Automated Software Testing (cont.)



2. Issues in Serial Testing

How to design test cases?

- **black-box testing**
  - program to be tested is considered to be a black box
  - test data derived from the specifications
  - exhaustive testing (i.e. use every possible input) is infeasible
  - we are limited to trying a small subset of possible inputs $\Rightarrow$ well selected test cases should (1) maximize yield, (2) cover larger set of test cases

- **white-box testing**
  - test data derived from both the specification and structure of the program

Test Coverage

- **exhaustive path testing** (i.e. executing all possible paths) is infeasible

- **statement coverage**
  - example: **if** (a>1) $\wedge$ (b=0) **then** x := x/a;
  - test case (a=2, b=0) covers both "**if**" and assignment
  - would not exhibit faults in program logic, e.g. "or" instead of "and"

- **branch coverage** – each branch is/is not taken
  - some faults in program logic still not uncovered
  - e.g. test cases (a=2, b=0) and (a=2, b=1)
  - would not exhibit e.g. "a>1" instead of "a>=1"

Test Coverage (cont.)

- **multiple condition coverage**
  - each condition takes both true and false at least once
- **all-du-path coverage**
  - at least one path for each definition-use
- other types of code coverage exist

Measuring Test Coverage

- tools to measure test coverage exist, e.g. GCT (Generic Coverage Tool)
  - instrumentation of code
  - translate instrumented code
  - read and interpret the results
- tests made without measuring typically cover about 55% of the program (Grady 1993)

Measuring Test Coverage (cont.)

Sample GCT output (taken from GCT tutorial):

```
”lc.c”, line 137: operator > might be >=. (L==R)
”lc.c”, line 137: condition 1 (argc, 1) was taken TRUE 43, FALSE 0 times
”lc.c”, line 139: condition 1 (<...>[...], 1) was taken TRUE 0, FALSE 14
”lc.c”, line 153: if was taken TRUE 0, FALSE 29 times.
”lc.c”, line 162: loop zero times: 0, one time: 19, many times: 10.
”lc.c”, line 172: if was taken TRUE 0, FALSE 43 times.
```

3. Testing Concurrent Programs

- testing and debugging concurrent programs is more difficult
  - concurrent faults not present in serial programs
  - two executions of a program with the same data do not have to be identical
  - testing: concurrent faults are hard to exhibit
  - debugging: concurrent faults are hard to reproduce

Testing Concurrent Programs (cont.)

Concurrent faults:

- **safety faults** – race conditions, can be avoided by proper synchronization
  - common cause: the programmer forgets to lock
  - hard to detect
- **liveness faults**, e.g. starvation and deadlock
  - common cause: the programmer forgets to unlock (leads to a halt)
  - deadlock is easy to detect: create resource allocation graph, deadlock manifests as a cycle in the graph

Race Conditions

- formally specified first by Netzer (1992)

- there are two kinds of race conditions
  - **general races** – failures in programs intended to be deterministic
    - cause nondeterministic execution
  - **data races** – failures in nondeterministic programs
    - cause nondeterministic execution of critical sections accessing and updating shared data
  - data races are easier to detect (they are a local property, can be detected from the execution trace)

Data Race Detection

Types of race detection:

- static analysis – applied at compile time
  - undecidable for unrestricted programs
- post mortem analysis – collect trace information, analyze
  - problem – traces tend to be huge, we cannot store everything
- on-the-fly analysis – trace is analyzed as it is generated
  - traces can include shared memory access
  - problem – slows the application down significantly

The probe effect

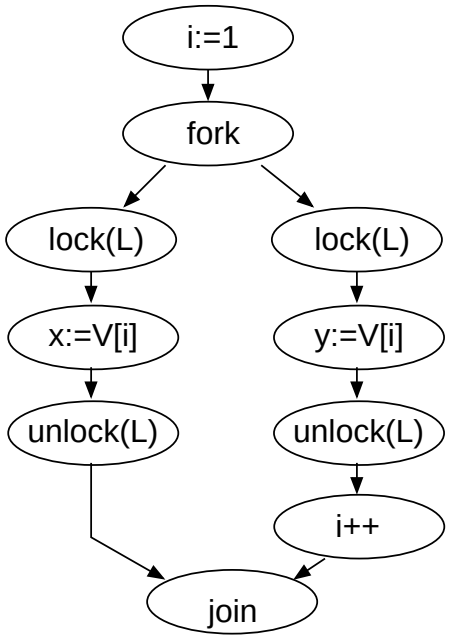
- any SW technique for collecting data perturbs the application
- more data $\Rightarrow$ large instrumentation overhead $\Rightarrow$ collected data no longer match the original program (e.g. the concurrent failure disappears)

Finding Races in Execution Traces

- most techniques based on state enumeration
- execution traces $\equiv$ branch-free programs
  - branches are ”hard wired” then trace is generated
  - monotonic synchronization constructs = branch-free program either always terminates or always deadlocks in the same state
  - if synchronization constructs are monotonic, exhibited races can be found in P
  - single semaphore – P
  - other primitives – NP hard or co-NP-hard

Example: Anne Dinning and Edith Schonberg

Dynamic on-the-fly algorithm to detect race conditions in programs with fork/join and locks



Example (cont.)

- Dinning-Schonberg algorithm reports a race if:
  - shared variable is accessed in two unordered blocks
  - at least one of the accesses is a write
  - intersection of lock covers is empty

Limitation of the Dinning-Schonberg algorithm:

- limited to programs with critical sections synchronized using lock and unlock (not applicable to other synchronization primitives)

- will not find race conditions in programs with internal non-determinism (races in execution paths not taken)

4. Test Adequacy Criteria

- traditional test coverage methods can be used for serializations, but will not uncover concurrent faults
- branch coverage?
  - threads may or may not be independent
  - we cannot cover branches in one thread independently of the other
- all-du-path coverage?
  - we obtain partial paths (definition $\rightarrow$ post, wait $\rightarrow$ use)
  - some work done (e.g. Yang and Pollock 1998), but has (inherent?) problems

Test Adequacy Criteria (cont.)

- liveness
  - 0, 1, more threads suspended? (e.g. Long 2003)
  - state transition coverage? (Taylor, Levine, and Kelly 1992)
    - includes control and synchronization, omits data values etc.
    - state explosion problem, hides sequential activities in each state
  - interaction coverage? (Katayama, Furukawa, and Ushijima 1997 for Ada task-types)

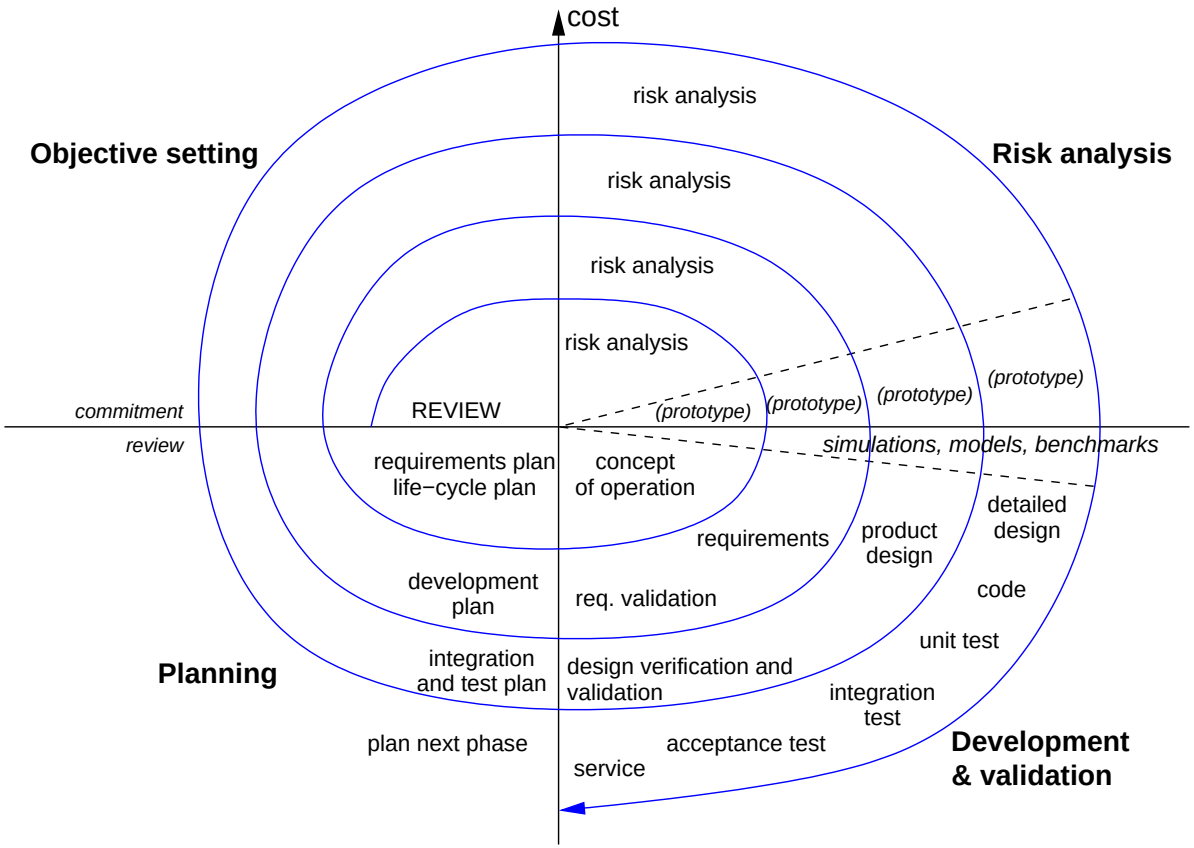
- race coverage?
  - contention on variables
  - GCT can measure whether two methods were executed at the same time, ConTest measures whether a context switch occurred when executing a method

Improving Test Yield

- probe effect (Gait 1986)
  - inserted delays can hide/exhibit failures
  - ”noise maker” – enforces different interleavings of events on each execution
  - delays should be inserted to useful places (synchronization events, communication events, thread creation), delay duration needs to change
  - problem: exponential number of interleavings, heuristics needed to find the most useful ones
- existing utilities: ConAn, ConTest

5. Research Plan

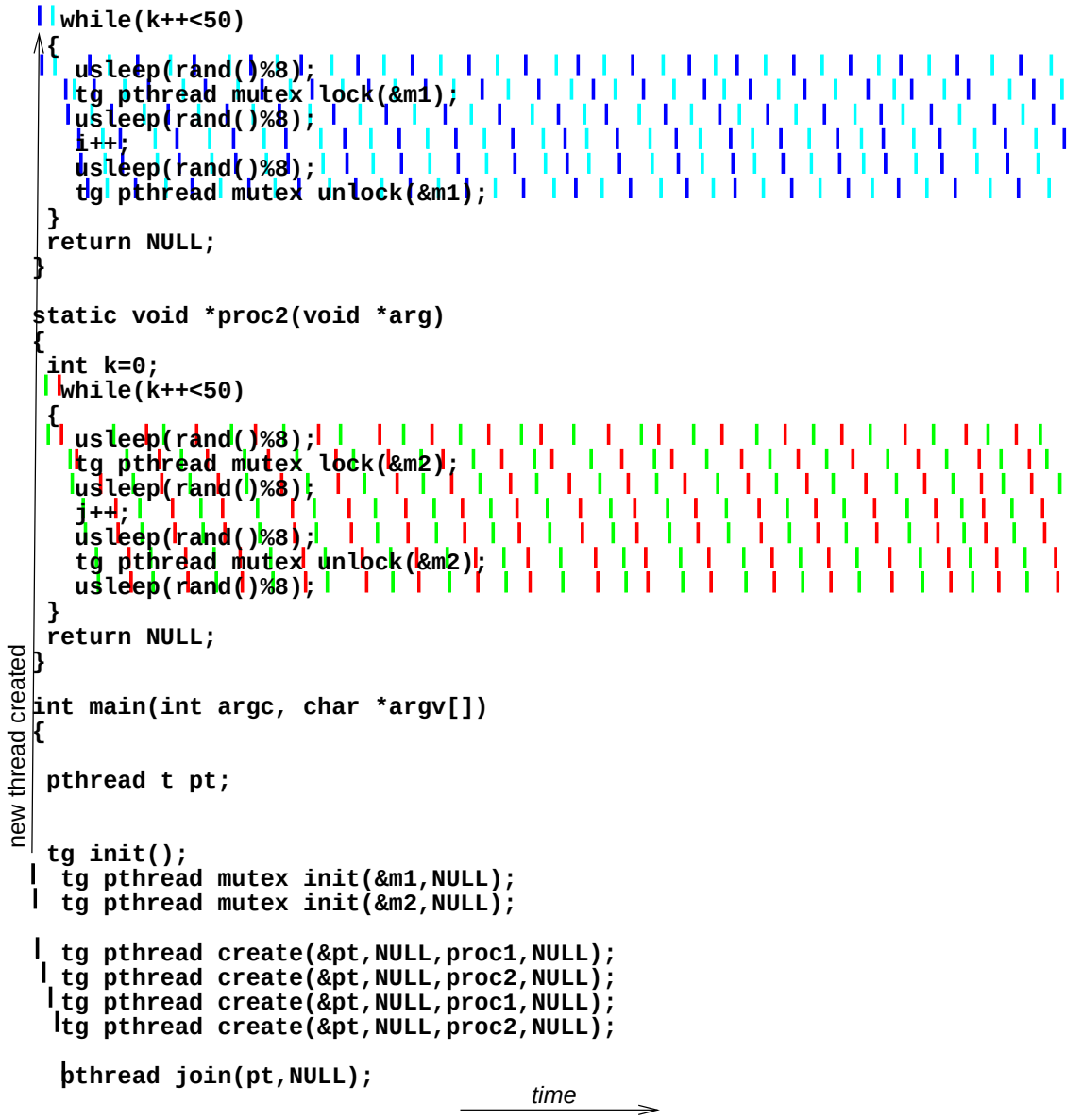
Goal of the project: support for testing multithreaded applications.



Steps for the First Cycle

- try to understand the problem
  - create a benchmark (set of test cases)
  - develop support for visualization
  - look for patterns
- develop prototypes
  - prototype instrumentation injecting ”noise”
  - look for the ”alternatives” (utilize the visualization)
- find the solution at the high level of granularity
  - refine for lower level of granularity (e.g. adapt to race coverage)
- automate instrumentation, possibly for C# or Java

Tracing & Visualization of Trace Logs



Tracing & Visualization of Trace Logs

```
sim_logger(SIM_LOGENTER, "gmi_chwrite(\"%s\", %s, %u) \"%s\"\n",
  name, (data.type == GMI_EOF)? "EOF" : "data",
  pktno, sim_getattr7name());
sim_logger(SIM_LOGVALUE,
  "gmi_chwrite: data = { type=%u, tag=%u, nelem=%u, elsize=%u }",
  data.type, data.tag, data.nelem, data.elsize);
chan = sim_getobject(name, SIM_OBJ_K);
if (chan==NULL || chan==(void*)1 || chan==(void*)2)
{
  sim_logger(SIM_LOGERR,
    "gmi_chwrite failed: access to nonexistent channel %s",
    name);
  sim_setevent(SIM_SETEV_EXIT, GMI_EV_LIBARG,
    "gmi_chwrite: bad argument (name)");
}

if (data.type == GMI_EOF) /* EOF requires special handling. */
{
sim_chcond *tc;
  sim_barsync(chan+1, chan->writers);
  sim_mutex_lock(&(chan->mutex));
 /* This should be done only once. */
 if (chan->iseof == 0)
 {
   chan->iseof = 1;
   for (tc = chan->cond; tc != NULL; tc = tc->prev)
   {  sim_cond_signal(&(tc->cond));}
 }
 sim_mutex_unlock(&(chan->mutex));
 return;
}

ent = sim_malloc(sizeof(sim_chunk));
ent->pktno = pktno;
/* Make a deep copy of the data. */
ent->d.data = data;
ent->d.data = sim_malloc(data.nelem * data.elsize);
memcpy(ent->d.data, data.data, data.nelem * data.elsize);
sim_mutex_lock(&(chan->mutex));
/* Ignore data if EOF. */
if (chan->iseof != 0)
{
  sim_logger(SIM_LOGERR,
    "gmi_chwrite: write ignored in EOF state");
  sim_mutex_unlock(&(chan->mutex));
  return;
}
/* Insert new entry into channel's list of chunks. */
ent->prev = NULL;
ent->next = chan->entries;
if (chan->entries != NULL)
{
  assert(chan->entries->prev == NULL);
  chan->entries->prev = ent;
}
chan->entries = ent;
if (chan->cond != NULL) /* Someone may wait for us. */
{
sim_chcond *tc;
  for (tc=chan->cond; tc!=NULL && tc->pktno!=pktno; tc=tc->prev)
  ;
  if (tc != NULL)
  {
    sim_logger(SIM_LOGINTRN,
      "gmi_chwrite: waking up thread %u", tc->thid);
    sim_cond_signal(&(tc->cond));
  }
}
```

Histogram

```
  sim_logger(SIM_LOGENTER, "gmi_chwrite(\"%s\", %s, %u) \"%s\"",
  name, (data.type == GMI_EOF)? "EOF" : "data",
  pktno, sim_getattr(name));
  sim_logger(SIM_LOGVALUE,
   "gmi_chwrite: data = { type=%u, tag=%u, nelem=%u, elsize=%u }",
   data.type, data.tag, data.nelem, data.elsize);
  chan = sim_getobject(name, SIM_OBJ_K);
  if (chan == NULL || chan == (void*)1 || chan == (void*)2)
  {
    sim_logger(SIM_LOGERR,
     "gmi_chwrite failed: access to nonexistent channel %s",
     name);
    sim_setevent(SIM_SETEV_EXIT, GMI_EV_LIBARG,
     "gmi_chwrite: bad argument (name)");
  }
  if (data.type == GMI_EOF) /* EOF requires special handling. */
  {
  sim_chcond *tc;
    sim_barsync(chan+1, chan->writers);
    sim_mutex_lock(&(chan->mutex));
   /* This should be done only once. */
   if (chan->iseof == 0)
   {
    chan->iseof = 1;
     for (tc = chan->cond; tc != NULL; tc = tc->prev)
     {  sim_cond_signal(&(tc->cond));}
   }
    sim_mutex_unlock(&(chan->mutex));
    return;
  }
  ent = sim_malloc(sizeof(sim_chunk));
  ent->pktno = pktno;
  /* Make a deep copy of the data. */
  ent->d = data;
  ent->d.data = sim_malloc(data.nelem * data.elsize);
  memcpy(ent->d.data, data.data, data.nelem * data.elsize);
  sim_mutex_lock(&(chan->mutex));
  /* Ignore data if EOF. */
  if (chan->iseof != 0)
  {
    sim_logger(SIM_LOGERR,
     "gmi_chwrite: write ignored in EOF state");
    sim_mutex_unlock(&(chan->mutex));
    return;
  }
  /* Insert new entry into channel's list of chunks. */
  ent->prev = NULL;
  ent->next = chan->entries;
  if (chan->entries != NULL)
  {
    assert(chan->entries->prev == NULL);
    chan->entries->prev = ent;
  }
  chan->entries = ent;
  if (chan->cond != NULL) /* Someone may wait for us. */
  {
  sim_chcond *tc;
    for (tc=chan->cond; tc!=NULL && tc->pktno!=pktno; tc=tc->prev)
    if (tc != NULL)
    {
      sim_logger(SIM_LOGINTRN,
       "gmi_chwrite: waking up thread %u", tc->thid);
      sim_cond_signal(&(tc->cond));
    }
  }
  sim_stat_data1("gmi_chwrite", chan->stat, data);
  sim_mutex_unlock(&(chan->mutex));
  sim_yield();
}
```

Switch Occurrences (Slow CPU)

```
        chan = simobject(name, SIM_OBJ_K);
        if (chan == NULL || chan == (void*)1 || chan == (void*)2)
        {
                sim logger(SIM_LOGERR,
                           "gmi_chwrite failed: access to nonexistent channel %s",
                           name);
                sim setevent(SIM_SETEV_EXIT, GMI_EV_LIBARG,
                             "gmi_chwrite: bad argument (name)");
        }
        if (data.type == GMI_EOF) /* EOF requires special handling. */
        {
                sim chcond *tc;
                sim barsync(chan+1, chan->writers);
                sim mutex_lock(&(chan->mutex));
                /* This should be done only once. */
                if (chan->iseof == 0)
                {
                        chan->iseof = 1;
                        for (tc = chan->cond; tc != NULL; tc = tc->prev)
                                { sim_cond_signal(&(tc->cond));}
                }
                sim_mutex_unlock(&(chan->mutex));
                return;
        }
        ent = sim_malloc(sizeof(sim_chunk));
        ent->pktno = pktno;
        /* Make a deep copy of the data. */
        ent->d = data;
        ent->d.data = sim_malloc(data.nelem * data.elsize);
        memcpy(ent->d.data, data.data, data.nelem * data.elsize);
        sim_mutex_lock(&(chan->mutex));
        /* Ignore data if EOF. */
        if (chan->iseof != 0)
        {
                sim_logger(SIM_LOGERR,
                           "gmi_chwrite: write ignored in EOF state");
                sim_mutex_unlock(&(chan->mutex));
                return;
        }
        /* Insert new entry into channel's list of chunks. */
        ent->prev = NULL;
        ent->next = chan->entries;
        if (chan->entries != NULL)
        {
                assert(chan->entries->prev == NULL);
                chan->entries->prev = ent;
        }
        chan->entries = ent;
        if (chan->cond != NULL) /* Someone may wait for us. */
        {
                sim_chcond *tc;
                for (tc=chan->cond; tc!=NULL && tc->pktno!=pktno; tc=tc->prev)
                if (tc != NULL)
                {
                        sim_logger(SIM_LOGINTRN,
                                   "gmi_chwrite: waking up thread %u", tc->thid);
                        sim_cond_signal(&(tc->cond));
                }
        }
        sim_stat_data1("gmi_chwrite", chan->stat, data);
        sim_mutex_unlock(&(chan->mutex));
        sim_yield();
}
```

Switch Occurrences (Fast CPU)

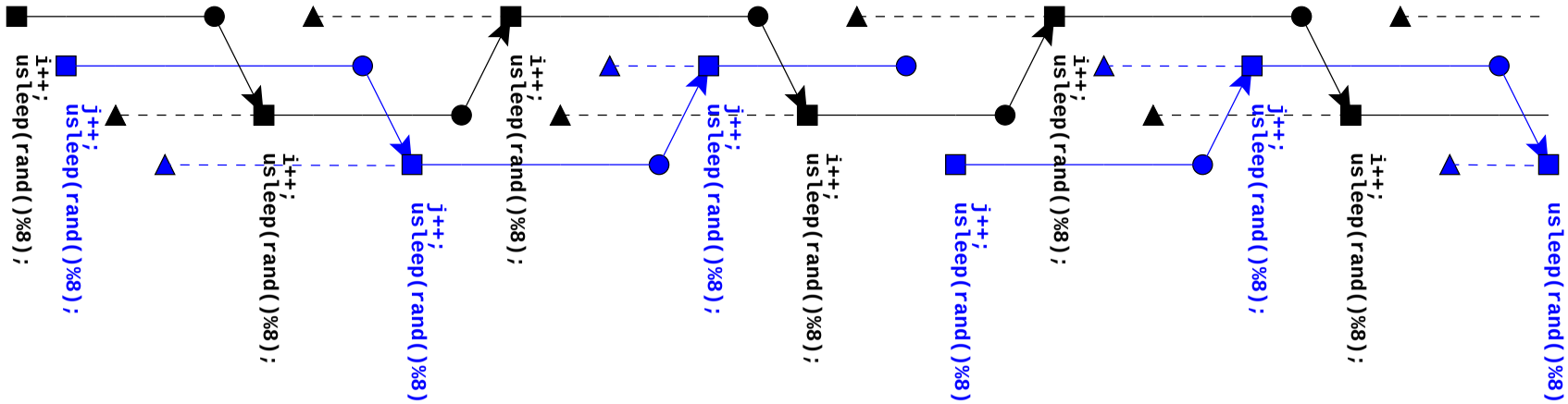
```
        chan = sim_getobject(name, SIM_OBJ_K);
        if (chan == NULL || chan == (void*)1 || chan == (void*)2)
        {
                sim_logger(SIM_LOGERR,
                           "gmi_chwrite failed: access to nonexistent channel %s",
                           name);
                sim_setevent(SIM_SETEV_EXIT, GMI_EV_LIBARG,
                             "gmi_chwrite: bad argument (name)");
        }
        if (data.type == GMI_EOF) /* EOF requires special handling. */
        {
                sim_chcond *tc;
                sim_barsync(chan+1, chan->writers);
                sim_mutex_lock(&(chan->mutex));
                /* This should be done only once. */
                if (chan->iseof == 0)
                {
                        chan->iseof = 1;
                        for (tc = chan->cond; tc != NULL; tc = tc->prev)
                                { sim_cond_signal(&(tc->cond));}
                }
                sim_mutex_unlock(&(chan->mutex));
                return;
        }
        ent = sim_malloc(sizeof(sim_chunk));
        ent->pktno = pktno;
        /* Make a deep copy of the data. */
        ent->d = data;
        ent->d.data = sim_malloc(data.nelem * data.elsize);
        memcpy(ent->d.data, data.data, data.nelem * data.elsize);
        sim_mutex_lock(&(chan->mutex));
        /* Ignore data if EOF. */
        if (chan->iseof != 0)
        {
                sim_logger(SIM_LOGERR,
                           "gmi_chwrite: write ignored in EOF state");
                sim_mutex_unlock(&(chan->mutex));
                return;
        }
        /* Insert new entry into channel's list of chunks. */
        ent->prev = NULL;
        ent->next = chan->entries;
        if (chan->entries != NULL)
        {
                assert(chan->entries->prev == NULL);
                chan->entries->prev = ent;
        }
        chan->entries = ent;
        if (chan->cond != NULL) /* Someone may wait for us. */
        {
                sim_chcond *tc;
                for (tc=chan->cond; tc!=NULL && tc->pktno!=pktno; tc=tc->prev)
                if (tc != NULL)
                {
                        sim_logger(SIM_LOGINTRN,
                                   "gmi_chwrite: waking up thread %u", tc->thid);
                        sim_cond_signal(&(tc->cond));
                }
        }
        sim_stat_data1("gmi_chwrite", chan->stat, data);
        sim_mutex_unlock(&(chan->mutex));
        sim_yield();
}
```

Switch Occurrences (multiprocessor)

```
        chan = sim_getobject(name, SIM_OBJ_K);
        if (chan == NULL || chan == (void*)1 || chan == (void*)2)
        {
                sim_logger(SIM_LOGERR,
                           "gmi_chwrite failed: access to nonexistent channel %s",
                           name);
                sim_setevent(SIM_SETEV_EXIT, GMI_EV_LIBARG,
                             "gmi_chwrite: bad argument (name)");
        }
        if (data.type == GMI_EOF) /* EOF requires special handling. */
        {
                sim_chcond *tc;
                sim_barsync(chan+1, chan->writers);
                sim_mutex_lock(&(chan->mutex));
                /* This should be done only once. */
                if (chan->iseof == 0)
                {
                        chan->iseof = 1;
                        for (tc = chan->cond; tc != NULL; tc = tc->prev)
                                { sim_cond_signal(&(tc->cond));}
                }
                sim_mutex_unlock(&(chan->mutex));
                return;
        }
        ent = sim_malloc(sizeof(sim_chunk));
        ent->pktno = pktno;
        /* Make a deep copy of the data. */
        ent->d = data;
        ent->d.data = sim_malloc(data.nelem * data.elsize);
        memcpy(ent->d.data, data.data, data.nelem * data.elsize);
        sim_mutex_lock(&(chan->mutex));
        /* Ignore data if EOF. */
        if (chan->iseof != 0)
        {
                sim_logger(SIM_LOGERR,
                           "gmi_chwrite: write ignored in EOF state");
                sim_mutex_unlock(&(chan->mutex));
                return;
        }
        /* Insert new entry into channel's list of chunks. */
        ent->prev = NULL;
        ent->next = chan->entries;
        if (chan->entries != NULL)
        {
                assert(chan->entries->prev == NULL);
                chan->entries->prev = ent;
        }
        chan->entries = ent;
        if (chan->cond != NULL) /* Someone may wait for us. */
        {
                sim_chcond *tc;
                for (tc=chan->cond; tc!=NULL && tc->pktno!=pktno; tc=tc->prev)
                if (tc != NULL)
                {
                        sim_logger(SIM_LOGINTRN,
                                   "gmi_chwrite: waking up thread %u", tc->thid);
                        sim_cond_signal(&(tc->cond));
                }
        }
        sim_stat_data1("gmi_chwrite", chan->stat, data);
        sim_mutex_unlock(&(chan->mutex));
        sim_yield();
}
```

Visualization of Event Ordering

■ – pthread_mutex_lock()
● – pthread_mutex_unlock()
▲ – thread suspended while trying to lock



- i++ (resp. j++) are protected by mutexes m1 (resp. m2)

Lessons learned so far

- complete traces tend to be huge, really :-)
- faster single-CPU computers $\Rightarrow$ less probable to exhibit a concurrent fault
- multiprocessors $\Rightarrow$ more likely to uncover faults