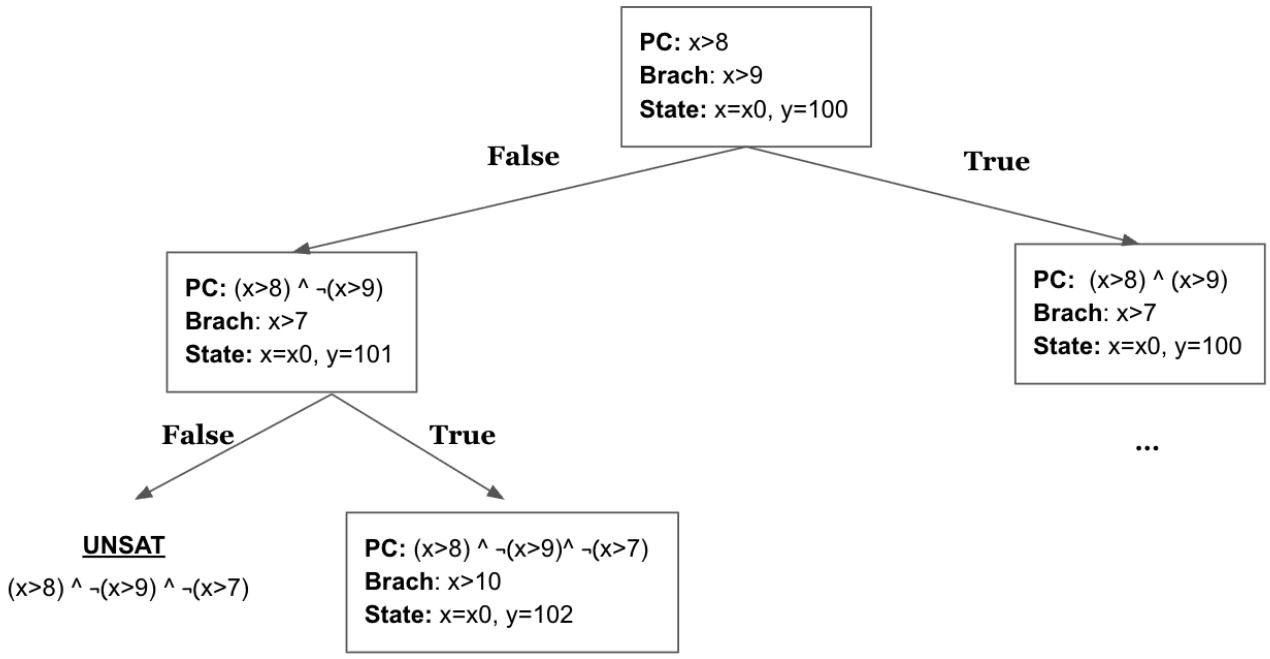
Trustworthy Software Systems / Spring '26 / Final Exam (Max: 110/100)
Name, student-id _____

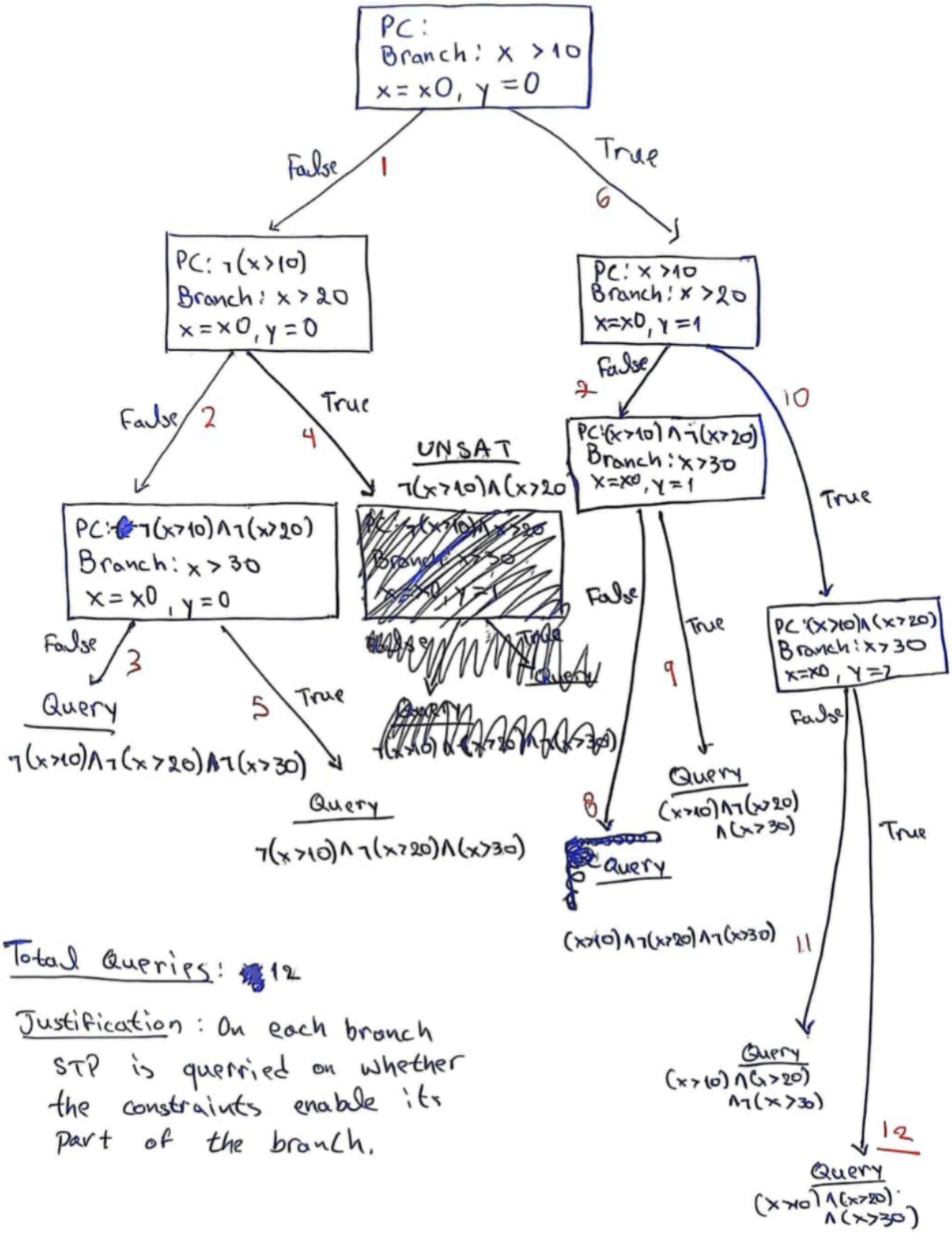
1) [35 pts] Symbolic and Concolic Execution Engines.

The following program is run using the KLEE symbolic execution engine.

```
int main(void) {
  int x,  y = 0;
  make_symbolic(&x);
  if (x > 10)
    y += 1;
  if (x > 20)
    y += 1;
  if (x > 30)
    y += 1;
  return y;
}
```

a) [10 pts] Assuming no query caching (i.e., solver result reuse), and assuming that KLEE forks and checks both branch directions for feasibility at every symbolic conditional, how many solver queries will be issued? Explain your answer by drawing the tree of executions, starting with the initial call to main, showing all the forks explored by KLEE along with the respective calls to the solver. (You may find it useful to construct the tree by adding a new node on each fork, and, for each node, maintain the path condition, the current branch, and the state. See the example below, from a *hypothetical* program. The next page is empty for you to construct something similar. Use ¬ for "not" and ∧ for "and.")

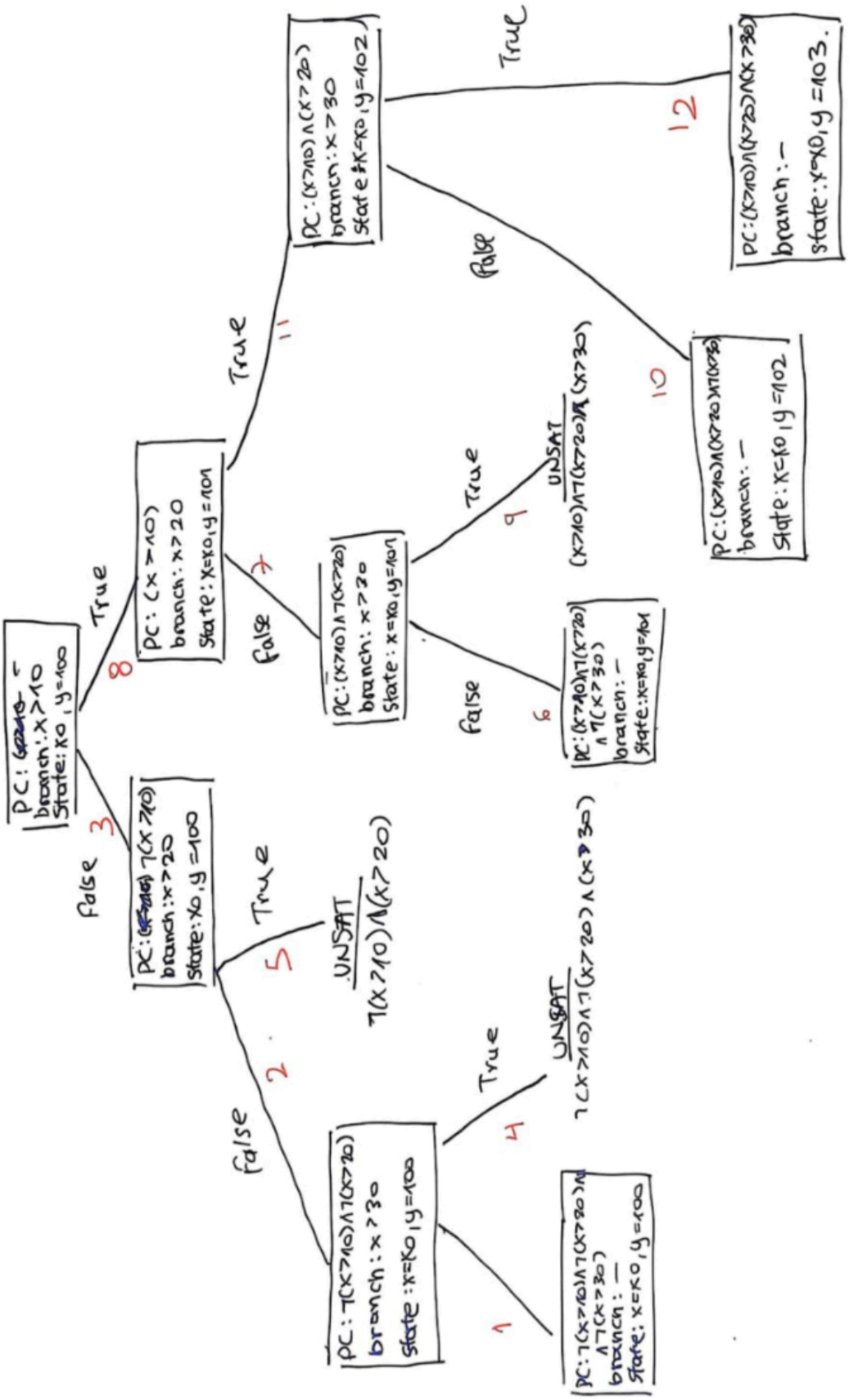




Total Queries: 12

Justification: On each branch STP is querried on whether the constraints enable its part of the branch.

Solution to 1-a) by Christos Komis



Solution to 1-a) by Anastasia Marinakou

The following program is run using the SAGE concolic execution engine with seed inputs x=1, y=2 for bar.

```
1.  int foo(int a, int b) {
2.    return a - b;
3.  }
4.
5.  void bar(int x, int y) {
6.
7.    if (x > y)
8.      x = 3;
9.    else
10.     y = 3;
11.
12.   y = y * 2;
13.
14.   x = foo(x, y);
15.
16.   if (x < 0)
17.     y = y - 1;
18.
19.   assert(x + y == 0);
20. }
```

b) [10 pts] Complete the table below with the values of the variables x and y for the concrete and symbolic execution of the program at each line indicated in the left-most column. After the second row, also write in the right-most column the full path condition under which the program has executed. (Use ¬ for "not" and ∧ for "and.")

Line	Concrete Execution	Symbolic Execution	Path Condition
6	x=1, y=2	x=x0, y=y0	N/A
11	x=1, y=3	x=x0, y=3	¬(x0 > y0)
13	x=1, y=6	x=x0, y=6	¬(x0 > y0)
15	x=-5, y=6	x=x0-6, y=6	¬(x0 > y0)
18	x=-5, y=5	x=x0-6, y=5	¬(x0 > y0) ^ (x0 – 6 < 0)

c) [5 pts] After completing the execution in part (b), what path constraint will SAGE solve to generate the next concrete test input? Explain the reasoning behind generating this specific constraint, and also provide a valid solution.

- Path Constraint:

>

¬(¬(x0 > y0)) ⇒ x0 > y0 anchored at y0=2, which was the value when the constrained was colltected. That is: x0 > y0 ^ y0 == 2.

- Explanation:

>

The main originality of our search algorithm is in the way children are expanded as shown in Figure 4. Given an `input` (line 1), the function `ExpandExecution` symbolically executes the program under test with that `input` and generates a *path constraint* `PC` (line 4) as defined earlier. `PC` is a conjunction of `|PC|` constraints, each corresponding to a conditional statement in the program and expressed using symbolic variables representing values of input parameters (see [16, 7]). Then, our algorithm attempts to *expand every* constraint in the path constraint (at a position `j` greater or equal to a parameter called `input.bound` which is initially `0`). This is done by checking whether the conjunction of the part of the path constraint prior to the `j`th constraint `PC[0..(j-1)]` and of the negation of the `j`th constraint `not(PC[j])` is satisfiable. If so, a solution `I` to this new path constraint is used to update the previous solution `input` while values of input parameters not involved in the path constraint are preserved (this update is denoted by `input + I` on line 7). The resulting new input value is saved for future evaluation (line 9).

```
1  ExpandExecution(input) {
2    childInputs = {};
3    // symbolically execute (program,input)
4    PC = ComputePathConstraint(input);
5    for (j=input.bound; j < |PC|; j++) {
6      if((PC[0..(j-1)] and not(PC[j]))
                    has a solution I){
7        newInput = input + I;
8        newInput.bound = j;
9        childInputs = childInputs + newInput;
10   }
11   return childInputs;
12 }
```

Figure 4. Computing new children.

- Solution:

> x0 > y0, y0 = 2. We choose: x=3, y=2.

See above: "Our algorithm attempts to expand every constraint in the path constraint (at a position j greater or equal to a parameter called *input.bound,* which is initially 0)" and "If so, a solution I to this new path constraint is used to update the previous solution input while values of input parameters not involved in the path constraint are preserved."

d) [10 pts] We now execute the program a second time, taking as inputs the new concrete values for x and y that you generated in (c). Please fill the table below again with the respective concrete and symbolic values at each line, as well as the respective path conditions.

Line	Concrete Execution	Symbolic Execution	Path Condition
6	x=3, y=2	x=x0, y=y0	N/A
11	x=3, y=2	x=3, y=y0	(x0 > y0)
13	x=3, y=4	x=3, y=2*y0	(x0 > y0)
15	x=-1, y=4	x=3-2*y0, y=2*y0	(x0 > y0)
18	x=-1, y=3	x=3-2*y0, y=2*y0-1	(x0 > y0) ^ (3 - 2*y0 < 0)

> At the end of this second execution, does the assertion in line 19 fail? Yes: 3-2*y0 + 2*y0-1 = 2 ≠ 0.

2) [30 pts] Exokernels, Monolithic Kernels, and Web-Server Optimizations.

You are designing an HTTP web server for serving static content. The host machine runs on the Linux kernel, and uses a magnetic hard disk drive (HDD). Your server handles client connections over TCP. (I hope at this point in your DIT life, you do know that the HTTP protocol runs on top of TCP/IP.)

For each HTTP GET request it receives on a connection, the server locates the requested file, reads its contents, and transmits them back to the client over that same TCP connection. Requested files not in the host's kernel buffer cache need to be read from the magnetic disk. Furthermore, since each connection is over the TCP protocol, assume that the server must also send the client a transport-level acknowledgment (ACK) packet for each packet it receives, and it must compute transport-level checksums for the outgoing packets. Finally, because static HTML pages reference embedded images and scripts, related objects are typically requested sequentially.

a) [24 pts] Read the above paragraph carefully, and suggest four distinct optimizations that an Exokernel design (with a custom library OS) makes possible from user-space, to reduce redundant work, latency, or I/O for this server. For each optimization: (1) briefly describe the optimization; (2) describe the redundant cost/inefficiency it removes; and (3) explain how the Exokernel design enables it, and contrast it versus a Monolithic kernel.

a-i) [6 pts]

>Optimization (brief description):
Piggyback the TCP ACK on the HTTP response — withhold the standalone ACK, and carry it on the first response segment, since a reply always immediately follows the request.

>Cost (brief description):
One extra packet (a standalone ACK) on the wire per request — wasted bandwidth plus the per-packet send/receive processing (an interrupt and packet handling on each side).

>Exokernel design vs. Monolithic:
The libOS implements TCP itself, in the same address space as the HTTP logic, so app-level knowledge ("a reply is imminent") directly controls ACK behavior. A monolithic kernel hides TCP and can only guess with a generic timer (delayed ACK); if the app produces the response within the timer window, the ACK does get piggybacked, so the exokernel's real advantage here is determinism — it never waits on the timer and never risks emitting the redundant standalone ACK.

a-ii) [6 pts]

>Optimization (brief description):
Zero-copy send.

>Cost (brief description):
The 2+ memory copies (cache → app buffer on read, app buffer → socket buffer on write).

>Exokernel design vs. Monolithic:
Physical memory pages and the network device are exposed as protected, bindable resources, so the libOS can hand cache pages directly to the NIC for DMA. On a monolithic kernel, the generic read()/write() socket path copies through userspace; it took a bespoke special-case syscall (sendfile()/splice()) to recover zero-copy — the monolithic kernel retrofits one special case, whereas the exokernel gets zero-copy generically from exposing the resources.

a-iii) [6 pts]

>Optimization (brief description):
Cached checksums.

>Cost (brief description):
Redundant CPU work — recomputing the checksum over the same (unchanging) file data.

>Exokernel design vs. Monolithic:
The libOS owns both the file cache and TCP/IP packet generation, so it can precompute and cache the data's partial (ones-complement) checksum beside the file and, at send time, fold in the small, changing header contribution. A monolithic kernel's socket API gives the app no way to supply or cache checksum state. Correctness invariant: the cached value is valid only while the file content and the packet segmentation are unchanged — it must be invalidated/recomputed if the file is modified, or if the bytes are re-segmented into different packets (a different MSS/path-MTU, or a byte-range / partial request).

a-iv) [6 pts]

>Optimization (brief description):
Application-aware disk layout.

>Cost (brief description):
Expensive random HDD seeks on a cache miss.

>Exokernel design vs. Monolithic:
The file system is a library OS, so the server controls disk block allocation and can co-locate related files — a generic allocator uses fixed, content-agnostic placement and doesn't know the relationships.

b) [6 pts] Explain the common principle underlying all the above optimizations.

>
Each optimization exploits application-level knowledge that a fixed OS abstraction discards at its boundary (the end-to-end argument). The exokernel's bet — securely expose low-level resources and let the library OS own the policy — is precisely what makes the server optimizable along any of these dimensions: the server, not the kernel, owns policy across storage, caching, networking, and memory.

Disclaimer: Note that some of the above cost/optimization discussed have already been implemented in modern monolithic systems in the in-kernel TCP/IP stack, or on the network devices stack. However, the question was specific in its "assuming…" parts.

3) [20 pts] Single-Address-Space Operating Systems (OSes).

µFork supports POSIX-like fork() in a Single-Address-Space OS. To do so, on fork(), it copies the parent's memory into a different region of the same global address space and uses CHERI capabilities to preserve isolation. Consider a parent copied from parent region P to child region C.

a) [5 pts] Explain why a naive, byte-for-byte copy of the parent's memory is insufficient to correctly resume the child, and compare and contrast this with the classic POSIX-like fork().

›

The copy is byte-identical but lives at different addresses. Any absolute pointer inside C still holds an address in P, so when the child dereferences it, it reads or writes the parent's memory instead of its own copy — breaking both correctness (it mutates the parent) and isolation. (Position-independent/offset-relative references would survive; absolute ones do not.)
The child gets a separate virtual address space with the same virtual addresses as the parent. The pointer's value is unchanged, but the child's page table maps that virtual address to the child's own physical pages (typically copy-on-write). The mapping differs, not the pointer, so nothing needs rewriting.

b) [5 pts] Why would solving the problem implied in (a) with ordinary, per-process page tables undermine the Single-Address-Space OS design?

›

It reintroduces exactly what the SASOS removed: separate address spaces, address-space switches, TLB flushes/pressure, and per-process page-table management. You'd lose the fast context switches and lightweight process model that motivated the design — defeating the point.

c) [5 pts] What are the two core primitives that CHERI contributes to µfork to make its fork() POSIX-like?

›

- Finding & relocating pointers: In CHERI, a pointer is a capability carrying base, bounds, and permissions, plus a hardware tag bit marking the word as a valid capability. The tag lets µFork scan the copied image and identify exactly which words are pointers (vs data) — impossible with raw integers. It then rebases each capability that pointed into region P by the offset (C minus P) so it points into C, preserving its bounds/permissions. Crucially, it rebases only capabilities into the copied private region; capabilities to shared/global objects (libraries, shared memory, kernel services) are left untouched.

- Isolation in one address space: Capabilities are unforgeable (you can't manufacture one with integer arithmetic — any non-capability store clears the tag) and bounded, so even though parent, child, and kernel share one address space, a µprocess can only touch memory it holds a capability for. That yields process-like isolation without separate page tables.

d) [5 pts] Describe two scenarios in which a Single-Address-Space OS could be useful, and explain why.

›

Serverless / FaaS (cloud functions). Functions are short-lived, single-purpose, and packed at high-density on a host. A SASOS makes sandbox creation and switching extremely cheap (µFork forks in ~54 µs, ~3.7× faster, +24% FaaS throughput) and boots in milliseconds with a tiny memory footprint, so the host can run far more instances and cut cold-start latency. The standard "fork a pre-initialized Zygote per invocation" trick becomes viable precisely because µFork restores fork() on a SASOS.

IPC intensive workloads is another valid option…

4) [15 pts] Distributed Consensus, Fault Tolerance, and State Machine Replication.

Consider an implementation of the Paxos algorithm where a leader waits for less than a majority of acceptors to answer "OK" to a "Prepare" or an "Accept" message, before proceeding to execute the next steps.

a) [5 pts] Explain how such an implementation can break the Paxos safety (agreement) property.

>

A majority of acceptors may already have chosen a different value. The leader may get responses from acceptors from outside the majority, and may proceed to propose and then commit a new value (which breaks the property that only one value can be chosen as a result of running a Paxos instance).

Now, consider the canonical implementation of the Paxos algorithm.

b) [5pts] is it possible for two Paxos leaders (for the same Paxos instance) to both see "prepare_ok" messages from a majority of nodes? If yes, describe a scenario in which this happens.

>

Suppose there are four nodes, N1, N2, N3, and N4. Assume N1 sends out "Prepare" messages with "n = 10," and gets replies from all nodes. Then N1 crashes. After a while, node N2 heartbeat times out and decides to become a leader for the same Paxos instance, and sends out "Prepare" messages with n = 11. N2 will likely receive responses from N2, N3, and N4: That is, a majority of nodes.

c) [5pts] You are designing a distributed, fault-tolerant locking service used by thousands of applications to coordinate access to shared, global resources (e.g., database replicas). Your service will provide applications with the following two interfaces:

- **Acquire(resource_id): To request exclusive rights on a specific resource**
- **Release(resource_id): To release their exclusive rights previously held on a specific resource**

Explain how Paxos can be used to implement this service. Bear in mind that such a design, with a distributed locking service at its core, is how Google handles the coordination of its planetary-scale applications.

>

See "The Chubby lock service for loosely-coupled distributed systems" by Mike Burrows, Google Inc.

5) [10 pts] Interface Commutativity

Consider the following interface specification.

```
/*
 * Resets the global value of `GLOBAL` to 0 if it is not already 0.
 */
void reset(void);

/*
 * Returns the global value of `GLOBAL`.
 * The value returned will be the value set by the last `reset` or `set` call.
 */
int get(void);

/*
 * Sets the global value of `GLOBAL` to `x` if it is not already `x`.
 */
void set(int x);
```

For the following six pairs of actions, assuming that any value returned from any of these actions can later be inspected, answer the following questions.

i) <reset(), reset()>
ii) <reset(), get()>
iii) <reset(), set(x)>
iv) <get(), get()>
v) <get(), set(x)>
vi) <set(x), set(y)>

a) [5 pts] Which pairs of actions are SIM-commutative in every reachable state? And Why?

>

<reset(), reset()>: Neither returns a value; both orders leave GLOBAL = 0. Nothing observable has changed.
<get(), get()>: Neither changes state; both reads return the same current value v in either order.

b) [5 pts] Which pairs of actions are not SIM-commutative in at least one reachable state? And Why?

>

<reset(), get()>: Assume GLOBAL ≠ 0; reset -> get: returns 0; get -> reset: returns GLOBAL ≠ 0.
<reset(), set(x)>: Assume GLOBAL=5, x ≠ 0; reset() -> set(x): final state GLOBAL = x ≠ 0.
; set(x) -> reset(): final state GLOBAL = 0.
<get(), set(x)>: Assume GLOBAL=5 and x ≠ 0; and x ≠ 5; get() -> set(x): returns 5; set(x) -> get(): returns x ≠ 5.
<set(x), set(y)>: Assume x ≠ y; set(x) -> set(y): Final state y; set(y) -> set(x): Final state x.