Guidedog Discussions GDS 0020

Parallel site builds within memory

STATE
committed
TYPE
Standards Track
AUTHORS
Vikrant Rathore
CREATED
2026-11-27
UPDATED
2026-11-27
DISCUSSION
No external discussion URL assigned
LABELS
build

Record status

The catalog defines the review process and the meaning of each state. This numbered record forms part of the catalog.

Contents

1. Abstract 1
2. Motivation and scope 2
3. Specification 2
4. Rationale and alternatives 7
5. Security and operational considerations 7
6. Backwards compatibility 7
7. Acceptance criteria and implementation status 8
8. Open questions 8
9. Review history 9
10. References 9

1. Abstract

`guidedog ssg build` builds every edition of every mount, one after another, and then assembles the site. We build the editions at the same time instead, each in a `guidedog build` child process, as many at once as the machine's memory and processors allow. A *job* is one edition build. Before a job starts, the build reserves a *weight* for it: the peak memory the job reached the last time it ran, with a margin. The weights of the running jobs never exceed nine tenths of the memory the machine had available when the build began, unless one job runs alone. A job whose weight exceeds that capacity runs alone, as it does today. We model the schedule as a transition system and prove in Lean that the bound holds, that every run ends, and that a parallel build ends with the outcome of the sequential build: the same jobs succeed, and the same job's failure is reported. We also prove that two jobs never write the same file, and that a file several jobs write is always read whole. From an empty build folder the Guidedog website builds in 12.8 s instead of 56 to 66 s, and publishes the same 3173 files; the measurements are in "Acceptance criteria".

2. Motivation and scope

The cost. The Guidedog website has five mounts: the home page and the manual in seven languages each, the manual as seven PDF books, the discussions, and the discussion records as PDF books. That is 23 project builds. `build_site` in `crates/gd-ssg/src/mounts.rs` runs them in a loop, one at a time. From an empty build folder the site takes 56 s of wall time and 109 s of processor time on Linux 7.0 (x86-64, 32 processors, 128 GB), so on average fewer than two processors work. The seven books take about 7 s each, 49 s together; the rest is short. Built at the same time as seven `guidedog build` processes, the books take 9.4 s, and each process peaks at 1.87 to 2.00 GB of resident memory, 13.6 GB together. Other projects show the same shape: the Sibuna website builds one source in nine languages. Release 1.0.0-beta.15 did not make these builds slower; we measured beta.13, beta.14 and beta.15 on the same sources and found the same times. The builds were always sequential.

The danger. Running everything at once is not safe. Seven books need 13.6 GB, which a 16 GB continuous-integration machine barely has, and a laptop with 8 GB does not. When memory runs out, the system kills a process or swaps until the build is slower than before. So the number of jobs that run at once must follow the memory, not only the processors.

Two facts that shape the design. The memory budget of a build (`--budget`, GDS 0004) bounds the buffers the host holds, not the memory of a conversion: the manual's PDF build peaks at 1.9 GB under the default budget of 1 GiB. So a weight cannot be derived from the budget; it must be measured. And the threads of one process cannot be measured apart: the peak memory of a process is one number for all its threads. A child process has a peak of its own.

Scope. This record covers `guidedog ssg build` and what it needs: the scheduler in `gd_host`, a lock that every project build takes on its records folder, the measurement of a build's peak memory, and the reading of the memory available, including a container's limit. It covers Linux, macOS and Windows, the systems Guidedog is released for, with the same behaviour on all three: each reports the memory available and a process's peak. Where a report fails, or on another system, the build is parallel when the user states the memory available (`GUIDEDOG_AVAILABLE_MIB`) and sequential otherwise. The builds inside one project (the books of one `pdf_documents` setting, the pages of one HTML build) are out of scope.

3. Specification

The interface.

```
guidedog ssg build [PROJECT] [--out DIR] [--jobs N|auto] [--no-pdf] [--preview]
guidedog build ... [--peak-memory-file=FILE]
```

`--jobs auto`, the default, runs at most as many jobs at once as the machine has processors. `--jobs 1` builds one edition after another inside the `guidedog` process, exactly as before this record. `--peak-memory-file=FILE` makes a project build write its process's peak resident memory, in bytes, to `FILE` when it ends, whether it succeeded or failed; a system that does not report it writes nothing.

`gd_host` gains three items.

```
pub struct Schedule { /* the jobs' phases, weights, capacity and slots */ }
impl Schedule {
    pub fn new(weights: Vec<u64>, capacity: u64, slots: usize) -> Schedule;
    pub fn may_start(&self, job: usize, room: impl Fn(u64) -> bool) -> bool;
    pub fn next(&self, room: impl Fn(u64) -> bool) -> Option<usize>;
    pub fn start(&mut self, job: usize);
    pub fn finish(&mut self, job: usize, succeeded: bool);
    pub fn is_ended(&self) -> bool;
    pub fn first_failure(&self) -> Option<usize>;
}
pub enum JobPhase { Waiting, Running, Succeeded, Failed }
pub fn peak_memory() -> Option<u64>;
```

`may_start` is the guard of `start` below, `next` picks among the jobs it allows, and `start` panics on a job it does not allow. `gd_plugins` gains `build_command`, the `guidedog build` command for a `BuildRequest`, which `build_with_program` already built inside. `gd_ssg::build_site` takes a `SiteHost` (a build in this process, and a

place for the children's reports) instead of a build callback; a closure is still a `SiteHost`. It returns the site's folder with a sentence on how the editions were built.

`available_memory` and `peak_memory` ask each system in its own way:

Table 41 Where the figures come from

System	Memory available	Peak of a process
Linux	`MemAvailable` in `/proc/meminfo`, lowered to the room the process's control group (cgroup v2) has left	`VmHWM` in `/proc/self/status`
macOS	the free, speculative and inactive pages that `/usr/bin/vm_stat` reports, times its page size	`ru_maxrss` of `getrusage`, which macOS gives in bytes
Windows	`ullAvailPhys` of `GlobalMemoryStatusEx`	`PeakWorkingSetSize` of `GetProcessMemoryInfo`

The calls of macOS and Windows are in `gd_host`'s module for foreign calls, each with its safety argument; `vm_stat` runs by its full path, so `PATH` cannot replace it. The build reads the memory available once a round of starts, since on macOS it runs a program.

When the build is parallel. The build runs jobs in child processes when all of these hold, and builds sequentially, as before, otherwise:

1. `--jobs` is not 1;
2. the build knows the `guidedog` program (`CommandContext::host_program`);
3. the memory available is known;
4. the plan has two jobs or more;
5. no job reads below the build's work folder `<build_dir>/ssg-projects`, and no job walks a folder that contains it. A job walks its source folder, `templates_path`, `html_static_path`, `html_extra_path`, `locale_dirs` and `pdf_font_paths`; none of them may lie inside the work folder or contain it. It reads its include roots, documented crates and local intersphinx inventories file by file; none of them may lie inside the work folder.

When condition 5 fails, the build says which mount reads what, and builds sequentially.

The jobs and their files. The plan lists the jobs in the order the sequential build runs them: mounts in the order of `conf.toml`, and the editions of a mount in their order. A job is a mount name m and a language l. Its child writes, below the work folder, only to m/l (its output folder), $m/.l.$ guidedog (the records folder that `guidedog build` keeps beside its output), $m/.l.$ log (its report, in a file the parent creates) and $m/.l.$ peak (its peak). Outside the work folder a child writes compiled catalogs (`.mo`) beside the `.po` files of its source, and Typst packages into the user's cache (see "Security").

Every project build locks its records folder. `gd_project::build` takes `gd_plugins::files::lock_exclusive` on `<records>/build.lock` before it writes anything and holds it until its commit has ended (GDS 0019). A second build of the same output is refused with `build.busy` (exit 4) instead of mixing its files with the first. This is not only for sites: two terminals that build one project into one folder were never protected. The builds of one project's default layout share one records folder (`_build/.doctrees`), so `guidedog build html` and `guidedog build pdf` of one project no longer run at once; they share the environment, and running them at once was never safe.

Weights and capacity. Let A_0 be the memory available when the build begins. The capacity is the budget that GDS 0004 calls fitting:

$$C = 16\,\mathrm{MiB} \cdot \left\lfloor \frac{9A_0/10}{16\,\mathrm{MiB}} \right\rfloor$$

The work folder keeps the peak of each job in `memory.toml`, a table from the job's key m/l to bytes. The parent writes it once the jobs have ended, whatever their outcome, under the site lock, from the peaks the children wrote; it keeps the jobs of the current plan and drops the others. A job that succeeded records its new peak. A job that failed or was stopped may have ended before its peak, so it keeps the larger of its old and new peaks; otherwise one interrupted build would make the next one start every job at once. A record that cannot be written stays as it was, and the build's outcome is reported as if the record did not exist. Only a parallel build measures peaks. Mount names hold only letters, digits, `-` and `_`, so no mount folder

is named like the work folder's own files `memory.toml` and `site.lock`. A job's weight is its recorded peak p_j with a quarter more, rounded up to 64 MiB:

$$w_j = 64\,\mathrm{MiB} \cdot \left\lceil \frac{p_j + p_j/4}{64\,\mathrm{MiB}} \right\rceil$$

A job without a record weighs as much as the heaviest recorded job of the same builder in this site, or, when there is none, 1 GiB for HTML and 3 GiB for PDF. The slots N are `--jobs`, or the processors, and at most the number of jobs. Each child still reads on as many threads as it would alone; the system shares the processors among them.

The model. A plan has n jobs $0, \ldots, n-1$, weights w, a capacity C, slots N, and an outcome $\mathit{fails} : \mathbb{N} \to \{0, 1\}$. The outcome is a property of the job alone, which assumption I below makes precise. A state gives each job a phase:

$$\mathit{phase} : \mathbb{N} \to \{\text{waiting}, \text{running}, \text{succeeded}, \text{failed}\}$$

The load and the count of a state are sums over the plan:

$$\mathit{load} = \sum_{k<n,\ \mathit{phase}\ (k) = \text{running}} w_k \quad \mathit{count} = |\{k < n : \mathit{phase}\ (k) = \text{running}\}|$$

In the initial state every job waits. The environment answers one question, $\mathit{room}\ (\sigma, j)$: may job j start now? The code asks whether nine tenths of the memory available now still hold w_j; when the memory available cannot be read, it consents, and the capacity alone decides. Nothing we prove depends on the answer; it can only hold a job back.

Table 42 The transitions

Step	Guard	Effect	Code
`start`	$j < n$, $\mathit{phase}\ (j) = \text{waiting}$, $\forall k.\ \mathit{phase}\ (k) = \text{failed} \Rightarrow j < k$, and $\mathit{count} = 0 \vee (\mathit{load} + w_j \leq C \wedge \mathit{count} < N \wedge \mathit{room}\ (\sigma, j))$	$\mathit{phase}\ (j) :=$ running	`Schedule::next` and `start`; the child is spawned
`succeed`	$\mathit{phase}\ (j) = \text{running}$ and $\mathit{fails}\ (j) = 0$	$\mathit{phase}\ (j) :=$ succeeded	the child exits with status 0; `finish(j, true)`
`fail`	$\mathit{phase}\ (j) = \text{running}$ and $\mathit{fails}\ (j) = 1$	$\mathit{phase}\ (j) :=$ failed	the child exits otherwise, or cannot start; `finish(j, false)`

The guard of `start` says three things. A job above a failed job never starts, as the sequential build stops at its first failure. A job starts at once when nothing runs, whatever its weight: a job heavier than the capacity runs alone, as it would run sequentially. Otherwise a job starts only when its weight fits beside the running weights and a slot is free. The model does not say which of the jobs that may start the code starts; the code starts the heaviest, and the earliest of equal weight, so that light jobs fill the gaps at the end. What we prove holds for any choice.

A build has *ended* when no step is possible. We write Reach for the states reachable from the initial state.

The invariant.

(J1) $\mathit{load} \leq C \vee \mathit{count} = 1$

(J2) $\mathit{count} \leq \max(N, 1)$

(J3) $k \geq n \Rightarrow \mathit{phase}\ (k) = \text{waiting}$

(J4) $\mathit{phase}\ (k) = \text{succeeded} \Rightarrow \mathit{fails}\ (k) = 0$

(J5) $\mathit{phase}\ (k) = \text{failed} \Rightarrow \mathit{fails}\ (k) = 1$

Lemma 1 (the invariant is inductive). J1 to J5 hold initially and every step keeps them. In Lean: `inv_init`, `inv_start`, `inv_end`, `inv_step`, `inv_reachable`.

Proof. Initially no job runs, so $\mathit{load} = \mathit{count} = 0$, and no job has ended. Changing the phase of one job $j < n$ changes the sums by the term of j alone (`total_update`). A `start` of j adds w_j to the load and 1 to the count (`load_start`, `count_start`). If nothing ran, the count becomes 1, which is J1's second alternative and keeps

J2. Otherwise the guard gives $load + w_j \le C$ and $count < N$, which keep J1 and J2. An end of j takes w_j and 1 away (`load_end`, `count_end`). If $load \le C$ held, it still holds. If $count = 1$ held, the count becomes 0, and a state where nothing runs has no load (`load_of_count_zero`). J3 holds because `start` requires $j < n$ and only running jobs end. J4 and J5 hold because each end step's guard reads $fails\ (j)$. □

Theorem 1 (memory).

$$\sigma \in Reach \Rightarrow load \le C \vee count = 1$$

This is J1. In Lean: `memory_bound`. *Corollary 1a.* When every weight is at most C, $load \le C$ in every reachable state (`memory_bound_fitting`). Proof: when one job j runs alone, the load is $w_j \le C$. □

Theorem 2 (slots). $\sigma \in Reach \Rightarrow count \le \max(N, 1)$. This is J2. In Lean: `slots_bound`.

Theorem 3 (termination). Every run has at most $2n$ steps. Proof: give a waiting job rank 0, a running job rank 1, and an ended job rank 2. Every step raises the sum of the ranks by one (`progress_step`), the sum starts at 0, and it is at most $2n$ (`progress_le`). □ In Lean: `runs_are_short`.

Theorem 4 (all succeed). In an ended state where no job failed, every job of the plan succeeded. In Lean: `ended_all_succeeded`. Proof: in an ended state nothing runs, since an end step would be possible (`ended_nothing_runs`). So the count is 0, and a waiting job j could start unless a job $k < j$ failed (`ended_waiting`). No job failed, so no job waits. □

Theorem 5 (the first failure). In a reachable ended state, if f is the first job that failed, every job before f succeeded, $fails\ (f) = 1$, and $fails\ (k) = 0$ for every $k < f$.

$$\wedge\ Ended\ (\sigma) \wedge phase\ (f) = \text{failed} \wedge (\forall k < f.\ \ phase\ (k) \neq \text{failed}) \Rightarrow (\forall k < f.\ \ phase\ (k) = \text{succeeded}) \wedge fails\ (f) = 1 \wedge \forall k < f.\ \ f$$

Proof: a job $k < f$ that waits would need a failed job before it, which would come before f. It does not run, and it did not fail, so it succeeded; J4 and J5 give the outcomes. □ In Lean: `ended_first_failure`. So f is the first job of the plan that fails, which is the job whose failure the sequential build reports, and the parallel build reports it. *Corollary 5a.* When no job of the plan fails, every job succeeds (`ended_none_fails`).

Theorem 6 (no job waits forever for memory). When nothing runs, a waiting job of the plan with no failed job before it can start, whatever its weight and whatever the environment says. In Lean: `idle_start`. With Theorem 3, every run ends after at most $2n$ steps, provided every child ends (assumption T), and Theorems 4 and 5 give the outcome.

Theorem 7 (the jobs' files are different files). Let two jobs have mount names m_1, m_2 without `/`, and languages l_1, l_2 that are not empty and hold neither `.` nor `/`. Let $fold$ lower ASCII capitals. If $fold\ (m_1) \neq fold\ (m_2)$ or $fold\ (l_1) \neq fold\ (l_2)$, then no file name of one job, folded, equals a file name of the other, folded. The names are m/l and $m/.l.s$ for any suffix s. `conf.toml` already ensures the premises: mount names are unique ignoring ASCII case and have no `/`; the languages of one mount are unique ignoring ASCII case, with only letters, digits, _ and `-`. So the jobs' files differ on every file system, also one that ignores case. Proof: the first `/` splits a name in one way only when the parts before it lack `/` (`split_unique`); so the folded mounts are equal and the folded names are. A name l starts with a byte that is not `.`, and $.l.s$ starts with `.`, so the names are of one kind; for $.l.s$ the first `.` after the leading one splits it, and the folded languages are equal (`names_fold`). □ In Lean: `jobs_disjoint`, and `jobs_disjoint_exact` without folding. The names of one job differ from each other too, since their suffixes do (`names_differ`). Two different jobs of a plan have different mounts or different languages, so this applies to every pair. The lock on the records folder turns a violation of this theorem, should the premises ever fail, into a refusal (GDS 0019, Theorem 1).

Theorem 8 (a shared file is read whole). Several jobs may write one compiled catalog: the HTML and the PDF edition of one language. `gd_host::publish` writes it to a stage file whose name nobody else uses (`create_new`) and renames it over the catalog. We model writers that each write their own value through a stage file of their own. A reader of the catalog sees its old contents, or one writer's contents, whole. When the writers have the same settings they write the same bytes, and once one has renamed, the catalog holds them for good. In Lean: `reader_sees_whole_file`, `renamed_holds_a_value`, `same_value`. No build reads a `.mo` file whose `.po` exists, so the build's outputs do not depend on which writer renamed last.

Theorem 9 (the parallel build builds the sequential site). Under assumptions I and T, a parallel build that ends publishes the site the sequential build publishes, and a parallel build that fails reports the problem the sequential build reports. Proof. By Theorem 7 and assumption I, no job writes a file another job reads or writes, except the catalogs, and by Theorem 8 and the remark after it these do not change what a build

reads. The parent fixes the publication clock once and passes it to every job, as the sequential build does. So each job's output folder and outcome are those of the same job run alone, which are those of the sequential build. By Theorems 4 and 5 the parallel build ends with the same jobs succeeded, or with the first failing job of the plan failed, and it reports that job's problem. The site is assembled from the output folders in plan order, by the same code as before. □ This theorem combines the others with an assumption about the real system; it is not a Lean theorem. A failure that is not a property of the job breaks its premise: a child that cannot start, or a job refused with `build.busy` because a stray build holds its folder. The parallel build then reports the first failure in plan order among the jobs it ran, which may differ from what the sequential build would report; it never publishes a site. The reports of the jobs are printed in plan order, each when it and every job before it have ended, so the terminal shows what the sequential build would show, later and in larger pieces. After a failure, the reports of jobs that were already running follow the failed job's report; the sequential build would not have run them, and we show what they found rather than hide it. When the person presses Ctrl+C, the children fail too, and the build reports `host.cancelled` rather than their failures.
The memory the jobs use. Let u_j be the peak memory of job j. Under assumption W, $u_j \le w_j$, and by Corollary 1a the running jobs use at most C, nine tenths of A_0, at every moment. When a job's weight exceeds C, it runs alone (Theorem 1), and the build uses what the sequential build uses for it. The parent itself was running when A_0 was read, so its memory then is outside A_0; while jobs run it grows only by the log it copies, one at a time, and it assembles the site after the last job has ended. W is an assumption about the jobs, not a theorem: a project can grow. Three things limit the damage when W fails: the margin of a quarter, the check of the memory available now before each start (*room*), and the record of the new peak, which corrects the next build.
What ties the model to the system.

Table 43 Assumptions

No.	Assumption	Source
A1	The memory available estimates what new processes can use without swapping: `MemAvailable` on Linux, the free, speculative and inactive pages on macOS (inactive pages are reclaimed before swapping; purgeable pages are counted among the others, so we leave them out), `ullAvailPhys` on Windows.	[proc5], [vmstat], [memstatus]
A2	In a control group, `memory.max` minus `memory.current` bounds what the group may still use; a group's limit is the smallest along its path to the root.	[cgroup2]
A3	The process's peak resident memory is `VmHWM` on Linux, `ru_maxrss` (in bytes) on macOS, and the peak working set on Windows.	[proc5], [getrusage], [memcounters]
A4	`rename` replaces its destination in one step; `create_new` fails when the name exists.	GDS 0019, A6; [open2]
I	A job's outcome and output depend only on what it reads; it reads its source, its settings, the user's Typst cache and its own folders, and no job writes another job's inputs except the catalogs. Condition 5 checks the configured paths; documents read only below the readable roots (GDS 0018).	this record; `gd_conf::readable_roots`
T	Every child process ends.	this record; Ctrl+C reaches the children, which share the terminal's process group
W	A job's peak memory is at most its weight.	measured, not proved; see above

What the model leaves out. It has no other programs: they share the machine's memory, and only the check of the memory available now sees them. It has one parent; the site lock (GDS 0019) keeps a second site build out. It does not model the time a job takes, so it proves no bound on wall time. Graham's bound for list scheduling [graham69] says that greedy start of jobs on N processors finishes within $2 - 1/N$ times the optimum when only processors limit the jobs; memory makes the problem a kind of bin packing, and we claim no bound.

4. Rationale and alternatives

Threads in one process. The obvious design runs `build_project` on threads of the `guidedog` process, which saves loading fonts once per job. We rejected it for five reasons. The peak memory of one thread's build cannot be measured, so no weight could be learned. Typst keeps a memo cache for the whole process, and each book build empties it (`comemo::evict(0)`) to free its memory; on threads, one book's end would empty the cache another book is using. Graphviz runs under one lock for the process (`GV_LOCK`), so diagrams of all jobs would wait for each other. The host's console, its budget questions and its crash report are one per process. A crash would end every job. A child process has none of these problems, and `gd_plugins::build_with_program` already builds through one.

A fixed number of jobs. `make -j N` and `cargo -j N` limit jobs by count. Seven books on eight processors need 14 GB; a count cannot know that.

Enforce the weight. `RLIMIT_AS` or a control group per job would stop a job at its weight instead of trusting a record. A limit on address space fails builds that reserve address space they never touch, a control group needs privileges, and neither exists on macOS and Windows. We prefer to measure and leave a margin.

Start a job only while the memory available now has room. GNU make's `--load-average` starts no job while the load is high [gnumake]. The memory available now alone would start too many jobs at once: a job that just started has not yet used its memory. We use it beside the weights, not instead of them.

Kill the running jobs at the first failure. The sequential build stops at the first failure; a parallel build could kill the jobs still running. A job killed while it commits its output relies on the journal of its records folder, and a job before the failed one must end to know which failure comes first (Theorem 5). We let running jobs end and start no job above the failed one.

Weights from the size of the sources. Memory depends on fonts, pictures, books and languages, and the measured peak captures all of them.

Order. The code starts the heaviest job that may start, which tends to finish sooner when heavy jobs come late in the plan; the theorems allow any order.

5. Security and operational considerations

Memory. The capacity leaves a tenth of the memory available to the system and other programs. The memory available is read again before each start, so a program that takes memory during the build holds jobs back. `GUIDEDOG_AVAILABLE_MIB` states the memory available where the system's report misleads, as it already does for budget suggestions (GDS 0004).

Typst packages. A child that needs an `@preview` package missing from the user's cache downloads it. Typst unpacks it into a folder of a random name beside its place and renames that folder; a second process that loses the race finds the package in place. On Windows the losing rename reports another error, and that job fails; the next build finds the package cached. An external `typst` program uses temporary files named after its process.

Children. A child runs the same `guidedog` program as the parent, with the same trust and confinement; `--untrusted` is not a `ssg build` option. The site lock is held while the children start; GDS 0019 proves that a child's copy of its descriptor does not keep it after the parent unlocks. Ctrl+C reaches every child, and the parent starts no further job. If the parent dies without its children, they finish their builds; a new site build that reaches one of their output folders meanwhile is refused with `build.busy`.

Reports. A child writes its report to its log file, with colours when the parent's terminal has them; the parent copies each log to its error stream in plan order.

6. Backwards compatibility

The site, the files of the work folder, the problem codes and the exit statuses are unchanged. `ssg build` gains `--jobs` and is parallel by default where condition 1 to 5 hold. A project build gains `--peak-memory-file` and its lock: a second build of one output folder is now refused with `build.busy` (exit 4) while the first runs. In a parallel build a child does not ask to raise its budget, since it has no terminal; it stops at its budget, as `--memory=ram` does. `--jobs 1` keeps the old behaviour, questions included. `available_memory` may report less than before inside a container with a memory limit, and now reports on macOS and Windows, where it reported nothing; budget suggestions there are capped to the memory available, as on Linux.

7. Acceptance criteria and implementation status

Implemented for 1.0.0-beta.16.

- `gd_host::Schedule` implements the model. `schedule::tests::the_schedule_keeps_the_invariant_in_every_interleaving` enumerates every reachable state of every plan of up to four jobs with weights 0, 1, 2, 3 and 5, capacity 4, every pattern of failures and one to eight slots, more than 100000 states, and checks J1 to J5, Corollary 1a, Theorems 3 to 6, that `next` picks the heaviest job that may start, and that no job starts above a failed one. Each of three faults put into the guard (one byte more capacity, no failure guard, no start when idle) makes it fail.
- `tests/integration/tests/projects/ssg.rs` builds a site of five editions (two HTML mounts in two languages, one edition from a source of its own, and a PDF book) with `--jobs 1` and with `--jobs 4`, finds the same files with the same bytes and the reports in plan order, and finds every edition's peak in `memory.toml`. A second test breaks two editions and finds the first in plan order reported, and no site published.
- `records_lock::tests::a_second_build_of_one_output_is_busy` finds `build.busy` while another build holds the records folder.
- `cargo xtask proofs` checks 30 theorems in `docs/gds/proofs`, the 17 of `Gds/R0020` among them, with no axiom beyond `propext`, `Classical.choice` and `Quot.sound`.
- `budget::memory_tests` reads `MemAvailable`, `VmHWM` and the control group limits from sample files.

Measurements. On Linux 7.0 (x86-64, 32 processors, 128 GB), the Guidedog website from an empty work folder, with the release build of this change:

Table 44 The Guidedog website, 23 editions

Build	Wall time	Most editions at once	Memory
`--jobs 1` (sequential)	66.3 s	1	4.27 GB peak of the one process
parallel, 107 GB available	12.8 s	23	capacity 96.2 GiB; each book peaks at 1.92 to 2.05 GB
parallel, `GUIDEDOG_AVAILABLE_MIB=6000`	33.5 s	6	capacity 5.3 GiB; at most 3.92 GiB in all `guidedog` processes together
parallel, `GUIDEDOG_AVAILABLE_MIB=2000`	59.1 s	14	capacity 1.8 GiB; each book (2.5 GiB weight) alone; at most 1.94 GiB together

The memory column of the parallel rows sums the resident memory of every `guidedog` process, sampled every 0.2 s. In the last row the sum exceeds the capacity only while one book runs alone, as Theorem 1 allows, and never two books ran at once. The four builds published the same 3173 files with the same bytes. The sequential build took 56 s on another run of the same machine; its processor time is 108 s, the parallel build's 148 s, which includes loading fonts once per book.

We ran everything on Linux x86-64. The macOS and Windows calls have not run yet: `gd-host` and `gd-plugins` pass clippy for `aarch64-apple-darwin` and `x86_64-pc-windows-msvc`, `gd-host`, `gd-plugins`, `gd-project` and `gd-ssg` pass it for `x86_64-pc-windows-gnu` with MinGW, and `vm_stat`'s report is parsed by a test from a sample. The release workflow builds and tests natively on macOS and Windows; its runs are the first of this code there, and we correct what they find before the release is tagged.

8. Open questions

- Whether macOS should count compressed memory that the system could release, which `vm_stat` reports apart; we leave it out and so err on the side of fewer jobs.
- Whether the books of one `pdf_documents` setting should be built in parallel too; the discussion records are 19 books in one job.
- Whether the verification workflow should cache `memory.toml` with the build folder, so that its builds start from measured weights.

9. Review history

- 2026-11-27: Written after the maintainer reported that site, manual and PDF builds felt slow after 1.0.0-beta.15. Measurement showed no regression and a sequential site build that uses fewer than two of 32 processors. The maintainer asked for a record with equations, axioms, proofs and Lean code that avoids races and never goes beyond the memory available.
- 2026-11-27: Reviewed before implementation. A survey of what a project build writes outside its own folders found the compiled catalogs, the Typst package cache and no lock; it also found that nothing kept two builds out of one records folder, so every project build now locks it. Two catalog writers may differ in `gettext_allow_fuzzy_translations`, so the shared-file model lets each writer write its own value. The review also stated the parent's memory, the fixed clock, the consent when the memory available cannot be read, and failures that are not a property of a job, and took out a wall time the draft claimed before it was measured.
- 2026-11-27: Implemented and reviewed. The end-to-end test of a failing site found that a child's log could not be created in a work folder that did not exist yet; the runner now makes the folder. A fault put into the schedule showed that without the failure guard Theorems 4 and 5 still hold, since every job then runs; the guard only spares the work after a failure, so the enumeration now checks it at every start. The measurements above replaced the plan of this section.
- 2026-11-27: An independent review of the code found that the records lock was released when the build's writing ended, before its commit; a second build could then have recovered the journal of a commit still running. The lock now lives in the value the commit consumes, and `tests/records_lock.rs` builds while another holder has the lock. The review also found that an interrupted job recorded a low peak, which would have made the next build start every job at once; that a failure to write `memory.toml` hid the build's own outcome; and that condition 5 refused parallel builds for include roots that merely contain the work folder, which are read file by file. All three are corrected above. Ctrl+C now reports `host.cancelled`.
- 2026-11-27: The maintainer asked for the same behaviour on all three systems. macOS and Windows now report the memory available and a process's peak, and build in parallel as Linux does; the release workflow's native runs on those systems are their first test, before 1.0.0-beta.16 is tagged.

10. References

[proc5] `proc_meminfo(5)` and `proc_pid_status(5)`, Linux man-pages: `MemAvailable` and `VmHWM`. https://man7.org/linux/man-pages/man5/proc_meminfo.5.html

[vmstat] `vm_stat(1)`, macOS: the page counts of the virtual memory system and its page size.

[getrusage] `getrusage(2)`, macOS: `ru_maxrss`, the maximum resident set size, in bytes.

[memstatus] Microsoft, `GlobalMemoryStatusEx` function and `MEMORYSTATUSEX` structure. https://learn.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-globalmemorystatusex

[memcounters] Microsoft, `GetProcessMemoryInfo` function and `PROCESS_MEMORY_COUNTERS` structure. https://learn.microsoft.com/en-us/windows/win32/api/psapi/nf-psapi-getprocessmemoryinfo

[cgroup2] The Linux kernel, "Control Group v2", section Memory: `memory.max` and `memory.current`. https://docs.kernel.org/admin-guide/cgroup-v2.html

[open2] `open(2)`, Linux man-pages: `O_EXCL`. https://man7.org/linux/man-pages/man2/open.2.html

[gnumake] GNU make manual, section 5.4, "Parallel Execution": `-j` and `--load-average`. https://www.gnu.org/software/make/manual/html_node/Parallel.html

[graham69] R. L. Graham, "Bounds on multiprocessing timing anomalies", SIAM Journal on Applied Mathematics 17(2):416-429, 1969. doi:10.1137/0117039

GDS 0004, memory budgets: the fitting budget and `GUIDEDOG_AVAILABLE_MIB`. GDS 0013, static publishing: the mounts, editions and assembly this record parallelizes. GDS 0018, confinement: the folders a build's documents read. GDS 0019, file locks: the site lock and the records lock.

Lifecycle event

```
2026-11-27: prediscussion -> discussion. Assigned a permanent number and
opened for discussion.
```

Lifecycle event

```
2026-11-27: discussion -> accepted. The maintainer asked for parallel site
builds within memory on Linux, macOS and Windows, and for their release in
Guidedog 1.0.0-beta.16; the design is implemented.
```

Lifecycle event

```
2026-11-27: accepted -> committed. Ship the implemented parallel site builds
in Guidedog 1.0.0-beta.16, as the maintainer asked. gd_host::Schedule and its
exhaustive model test; the records lock and its build test; the end-to-end
parallel and failure tests in tests/integration/tests/projects/ssg.rs; 17
theorems in docs/gds/proofs checked by cargo xtask proofs with Lean 4.19; the
Guidedog website built in parallel and sequentially with the same 3173 files
(section Acceptance criteria). The macOS and Windows memory calls are first
run by the release workflow.
```