Overview

Work-stealing scheduler

- $O(pS_1)$ worst case space
- small overhead

Narlikar scheduler[1]

- $O(S_1+pKT_\infty)$ worst case space
- large overhead

Hybrid scheduler

- Idea: combine space saving ideas from Narlikar with the work-stealing scheduler

1. Girija J. Narlikar and Guy E. Blelloch. Space-Efficient Scheduling of Nested Parallelism. *ACM Transactions on Programming Languages and Systems (TOPLAS),* 21(1), January, 1999.

What We Did

- Implemented Narlikar Scheduler for Cilk
  - Replaced WS scheduling code
  - Modified cilk2c
- Designed WS-Narlikar Hybrid Scheduler
- Implemented Hybrid Scheduler
  - Modified WS scheduling code
  - Modified cilk2c
- Performed empirical tests for space and time comparisons

Results

Data from running the modified fib program on 16 processors

	Space (Kb)	Ratio (scheduler/Cilk WS)	Time (sec)	Ratio (scheduler/Cilk WS)
Cilk WS	491520	1.00	1.8	1.0
Narlikar	204800	0.41	837.0	465.0
Hybrid	368640	0.75	2.3	1.3

- Hybrid retains some of the space saving benefits of Narlikar with a much smaller overhead.

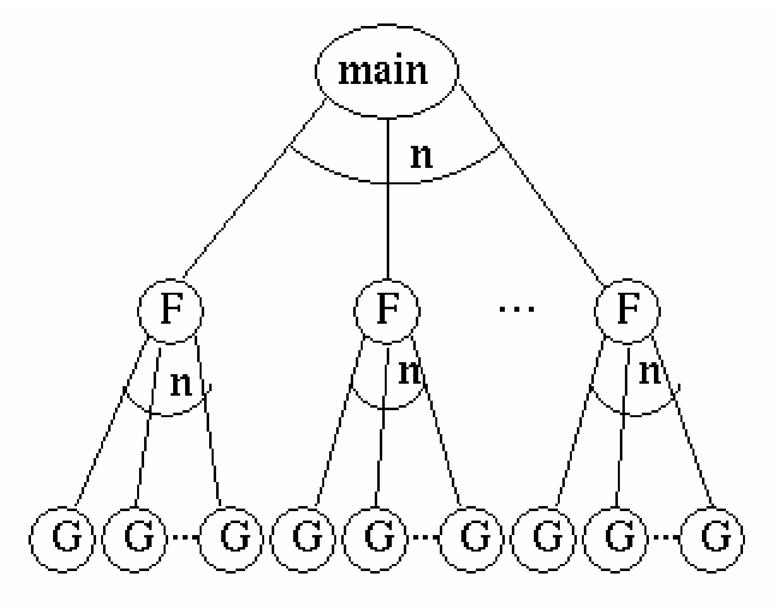
Outline

I. Example
II. Narlikar Algorithm
    a. Description
    b. Overheads/Bottlenecks
III. Hybrid Algorithm
    a. Motivation
    b. Description
IV. Empirical Results
V. Future Work
VI. Conclusions

Example

```
main() {
  for(i = 1 to n)
      spawn F(i, n);
}

F(int i, int n) {
  Temp B[n];
  for(j = 1 to n)
      spawn G(i, j, n);
}
```

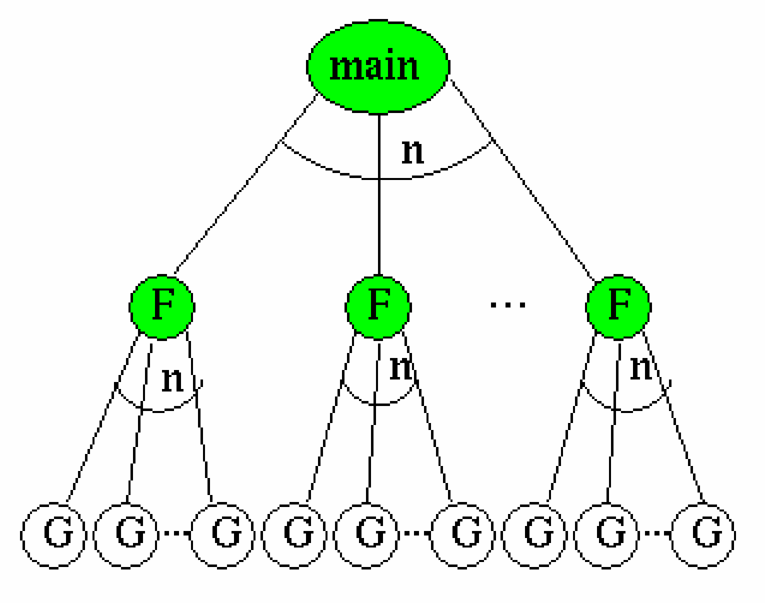


Schedule 1

Schedule outer parallelism first

Memory used (heap): $\theta(n^2)$

Similar to work-stealing scheduler ($\theta(pn)$ space)



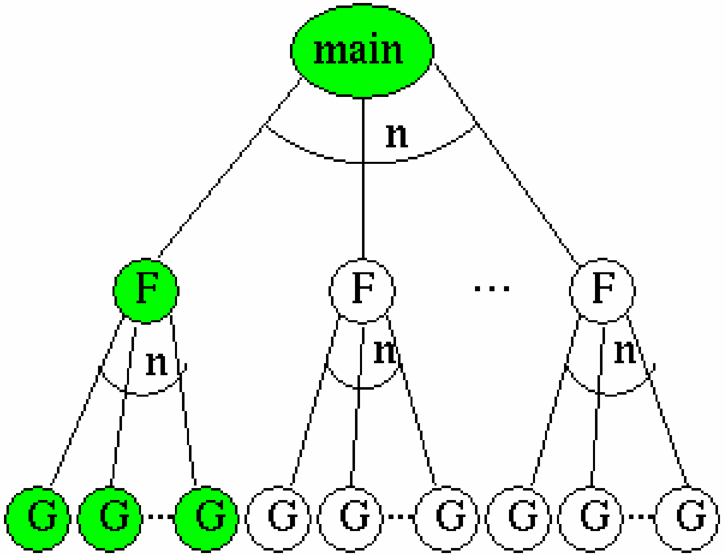
Green nodes are executed before white nodes

Schedule 2

Schedule inner parallelism first

Memory used (heap): $\theta(n)$

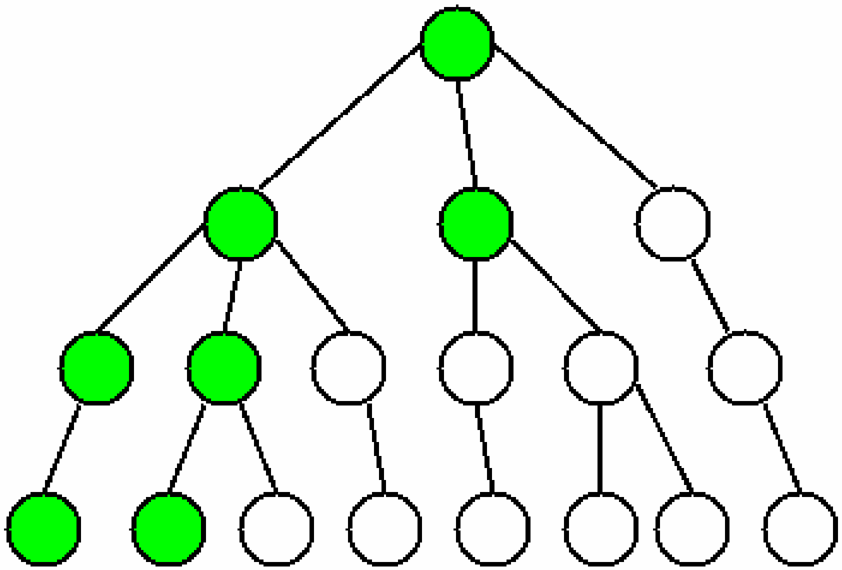
Similar to Narlikar scheduler ($\theta(n + pKT_\infty) = \theta(n)$ space)



Green nodes are executed before white nodes

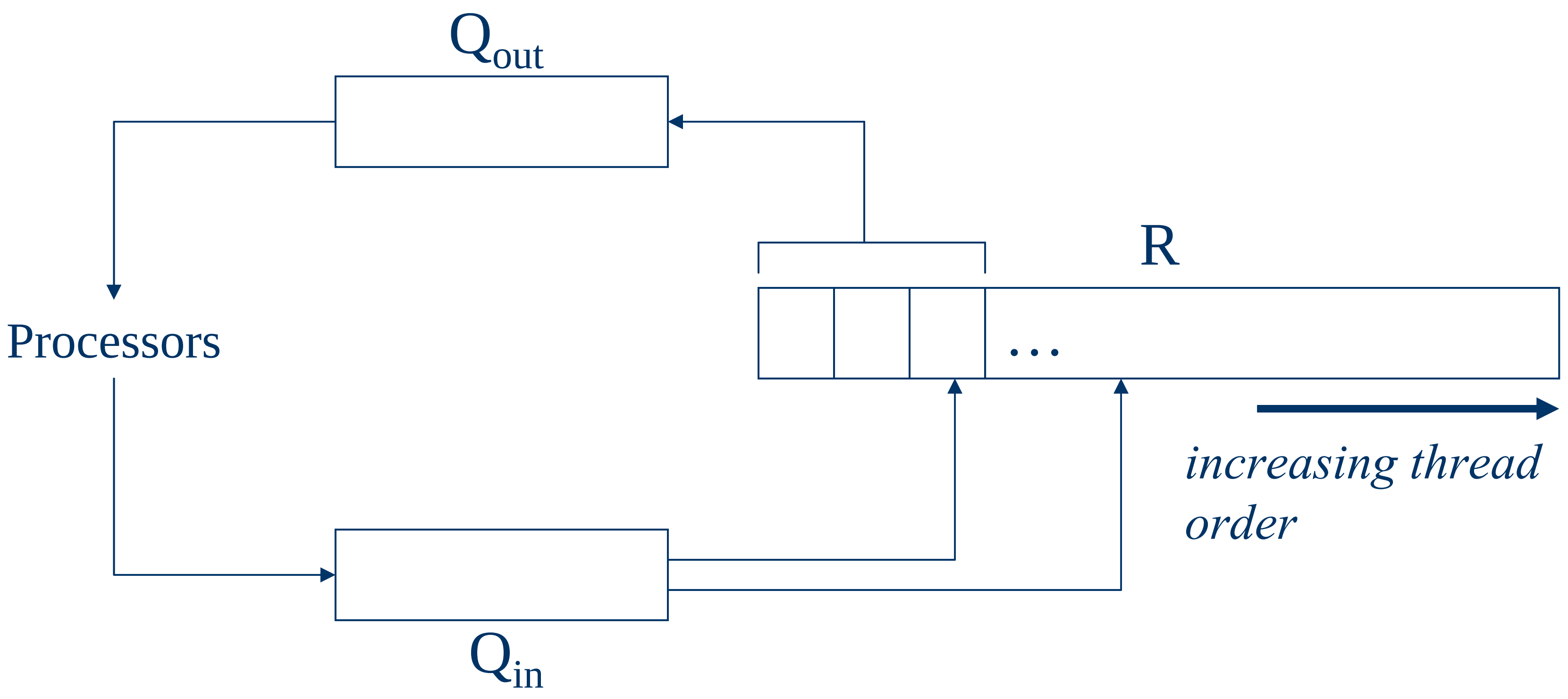
Narlikar Algorithm - Idea

- Perform a p-leftmost execution of the DAG



p-depth first execution for p = 2

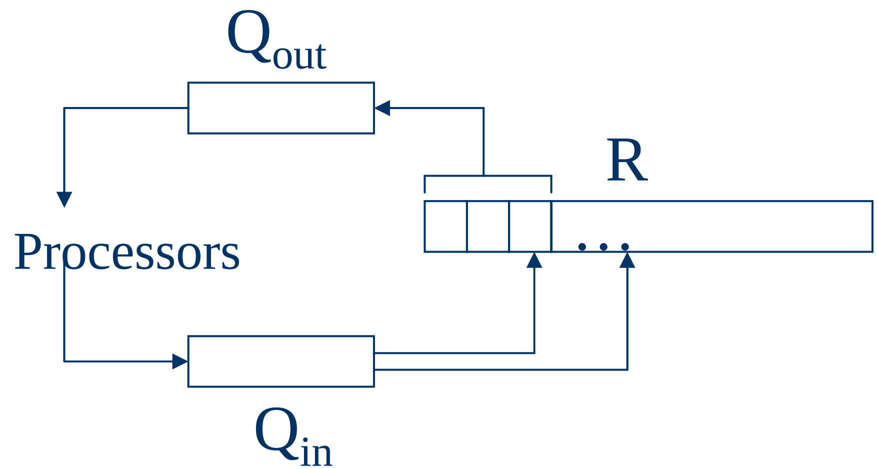
Narlikar Data Structures



- Q_{in}, Q_{out} are FIFO queues that support parallel accesses
- R is a priority queue that maintains the depth first order of all threads in the system

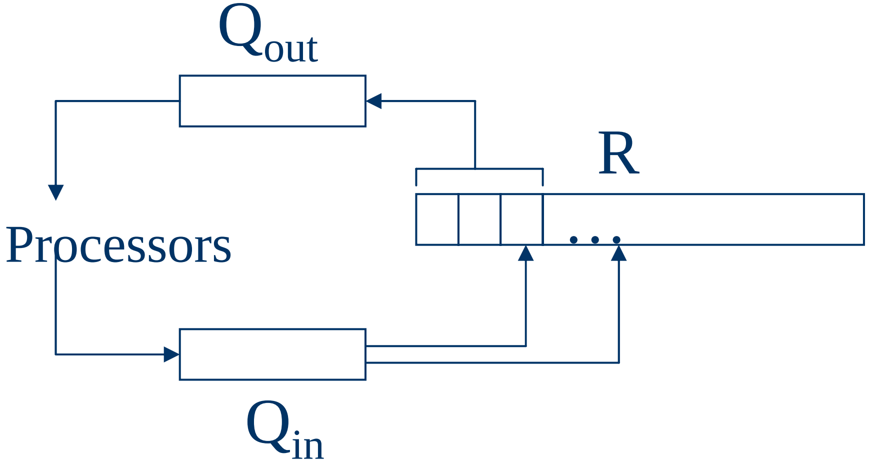
Narlikar – Thread Life Cycle

- A processor executes a thread until:
  - spawn
  - memory allocation
  - return
- Processor puts thread in Q_{in}, gets new thread from Q_{out}
- Scheduler thread moves threads from Q_{in} to R, performs spawns, moves the leftmost p to Q_{out}



Narlikar – Memory Allocation

- “Voodoo” parameter K
- If a thread wants to allocate more than K bytes, preempt it
- To allocate M, where M > K, put thread to sleep for M/K scheduling rounds.



Problems with Narlikar

- Large scheduling overhead (can be more than 400 times slower than the WS scheduler)
  - Bad locality: must preempt on every spawn
  - Contention on global data structures
  - Bookkeeping performed by scheduling thread
  - Wasted processor time (bad scalability)
- As of yet, haven't performed empirical tests to determine a breakdown of overhead

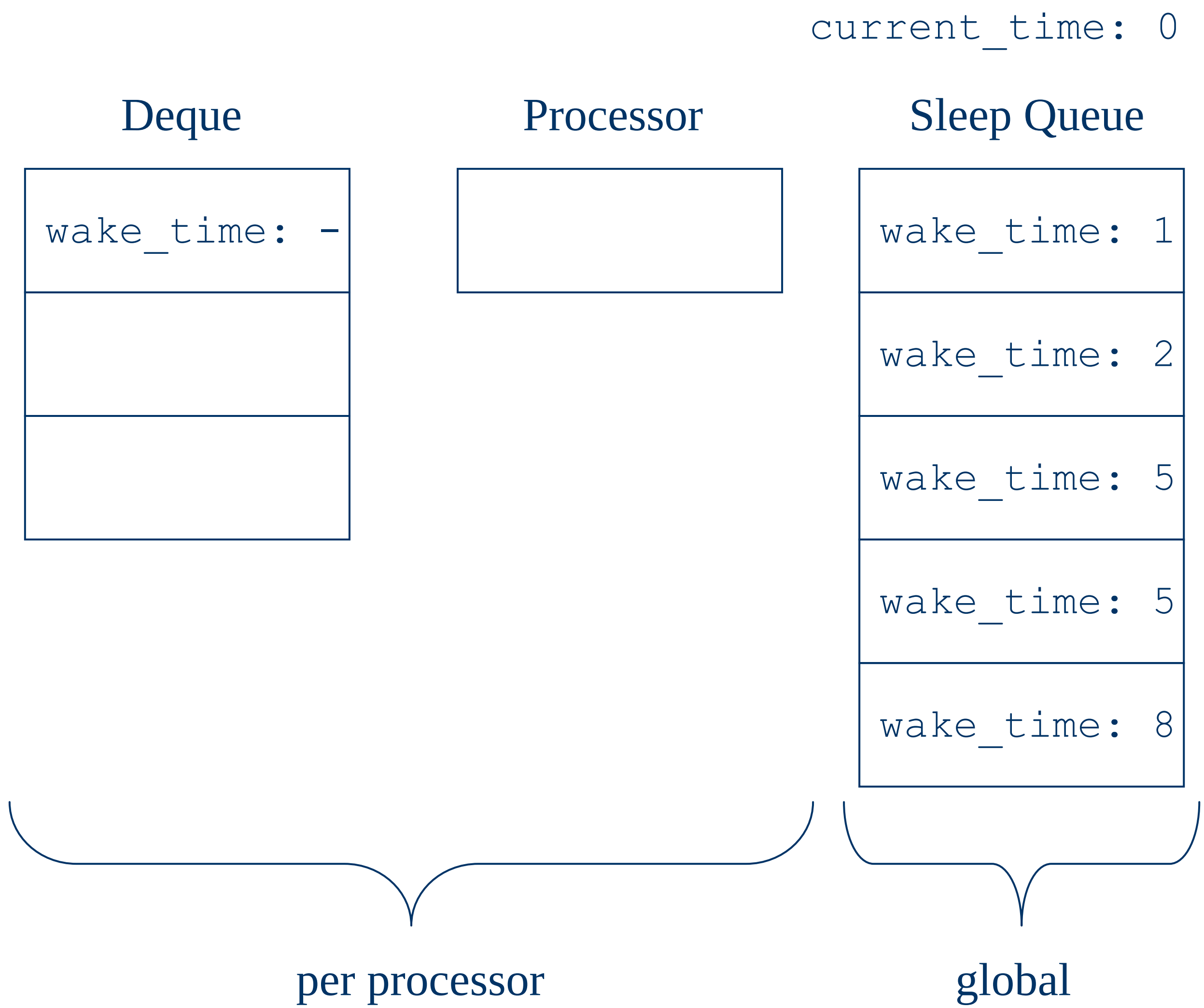
Hybrid Scheduler Idea

- Keeping track of left-to-right ordering is expensive
- What about just delaying the threads that wish to perform large memory allocations?
- Can we achieve some space efficiency with a greedy scheduler biased toward non-memory intensive threads?

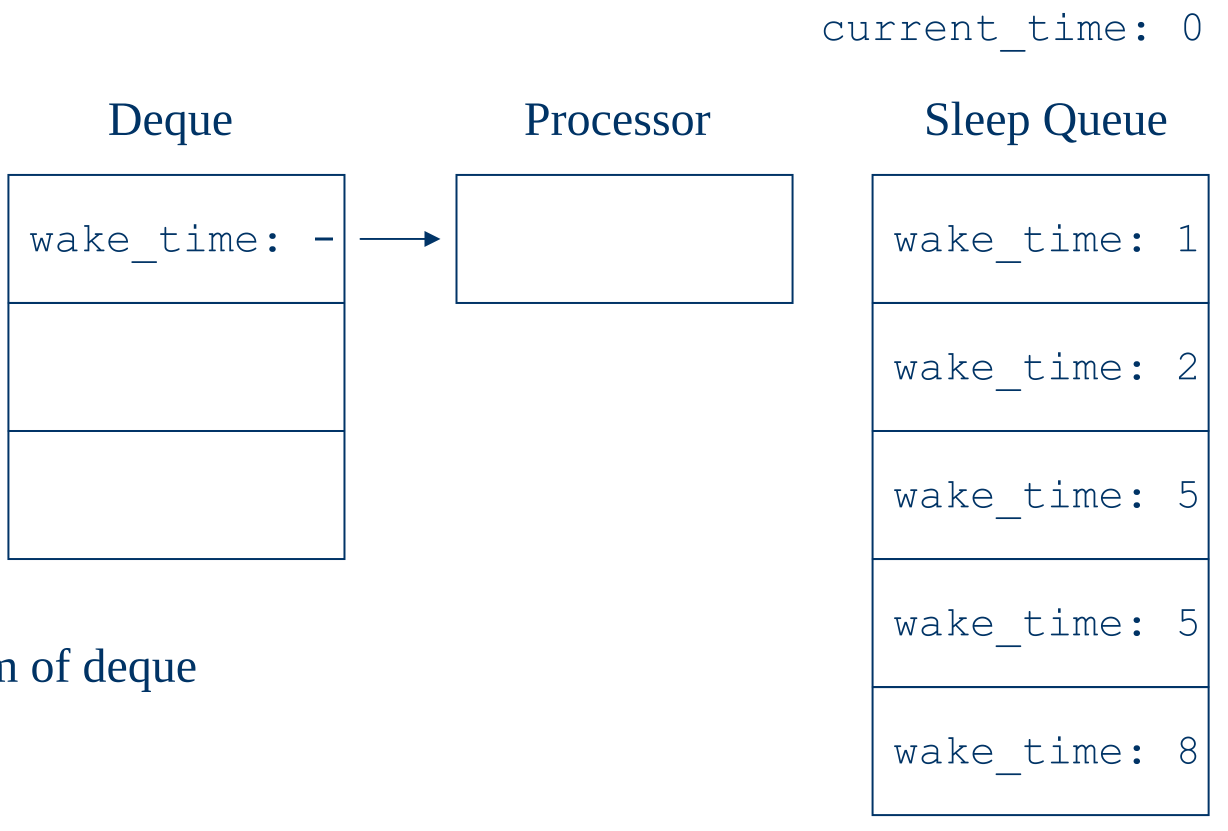
Hybrid Algorithm

- Start with randomized Work-stealing scheduler
- Preempt threads that perform large memory allocations and put them to sleep
- Reactivate sleeping threads when work-stealing

Hybrid Algorithm

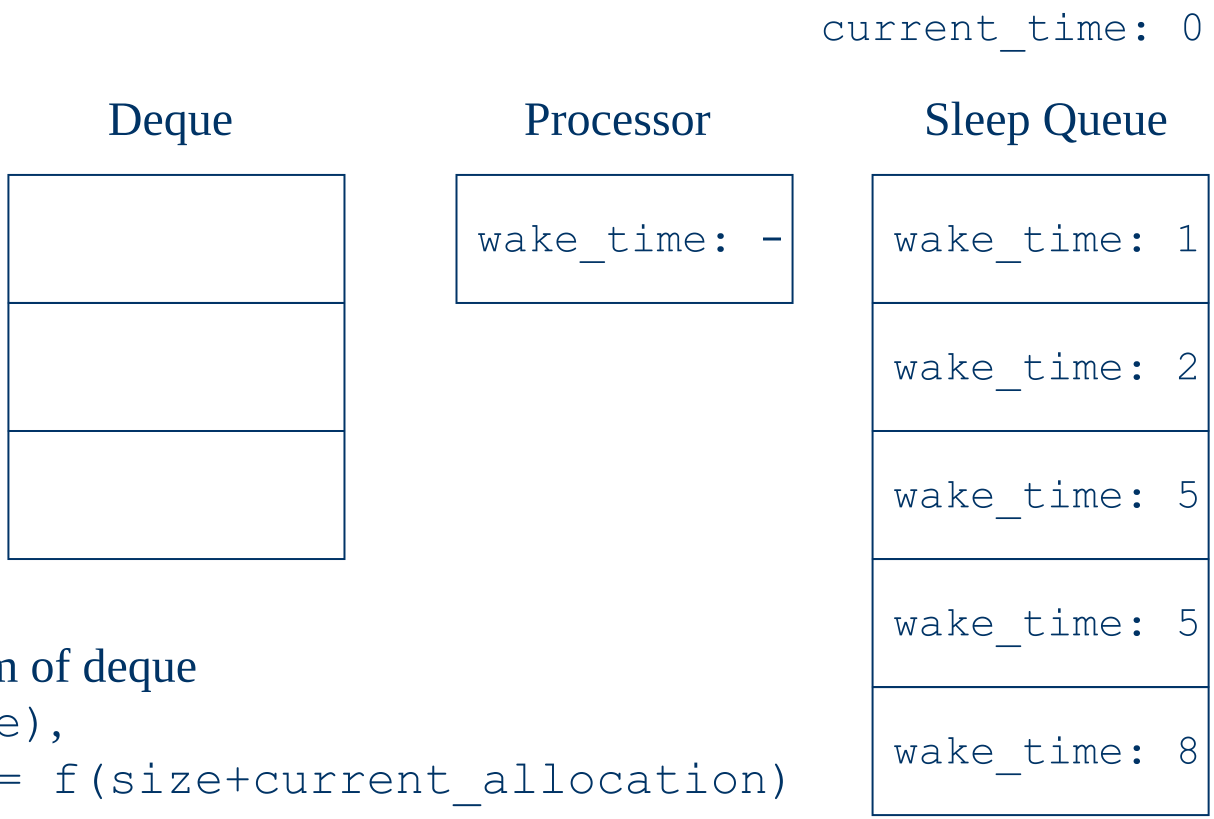


Hybrid Algorithm



Get thread from bottom of deque

Hybrid Algorithm



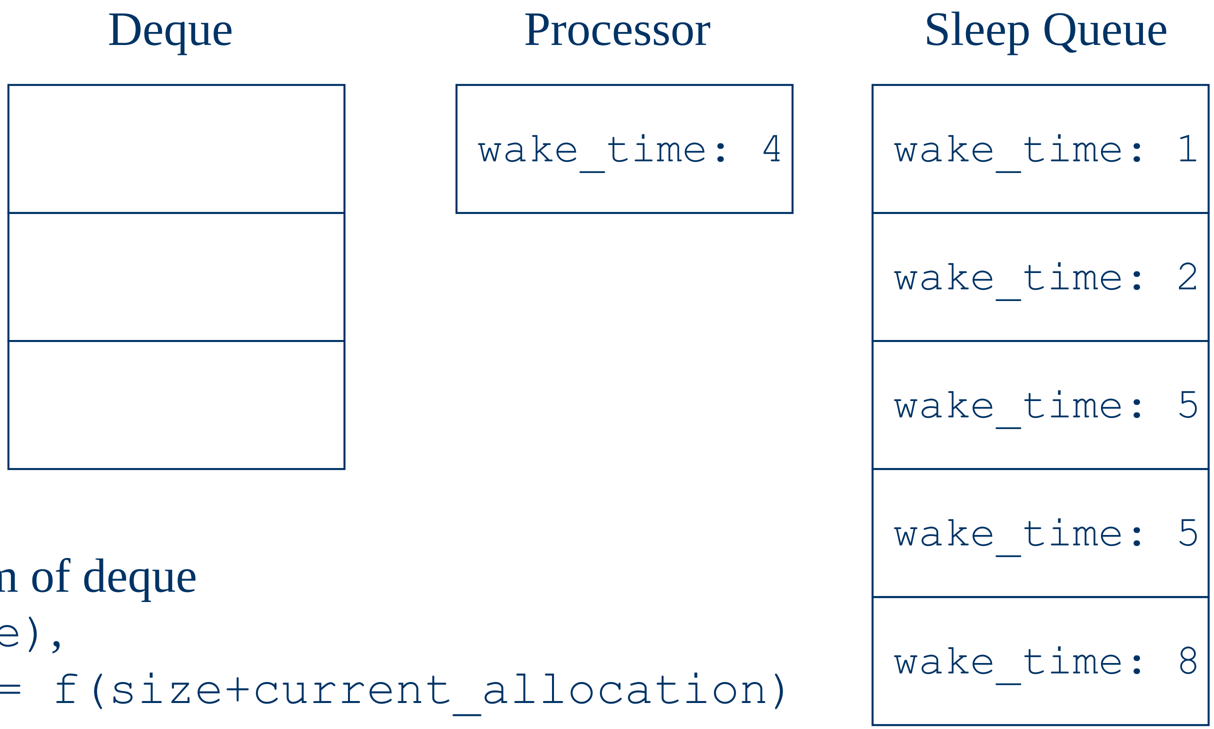
Get thread from bottom of deque

Before `malloc(size)`,

```
sleep_rounds = f(size+current_allocation)
```

Hybrid Algorithm

current_time: 0



Get thread from bottom of deque

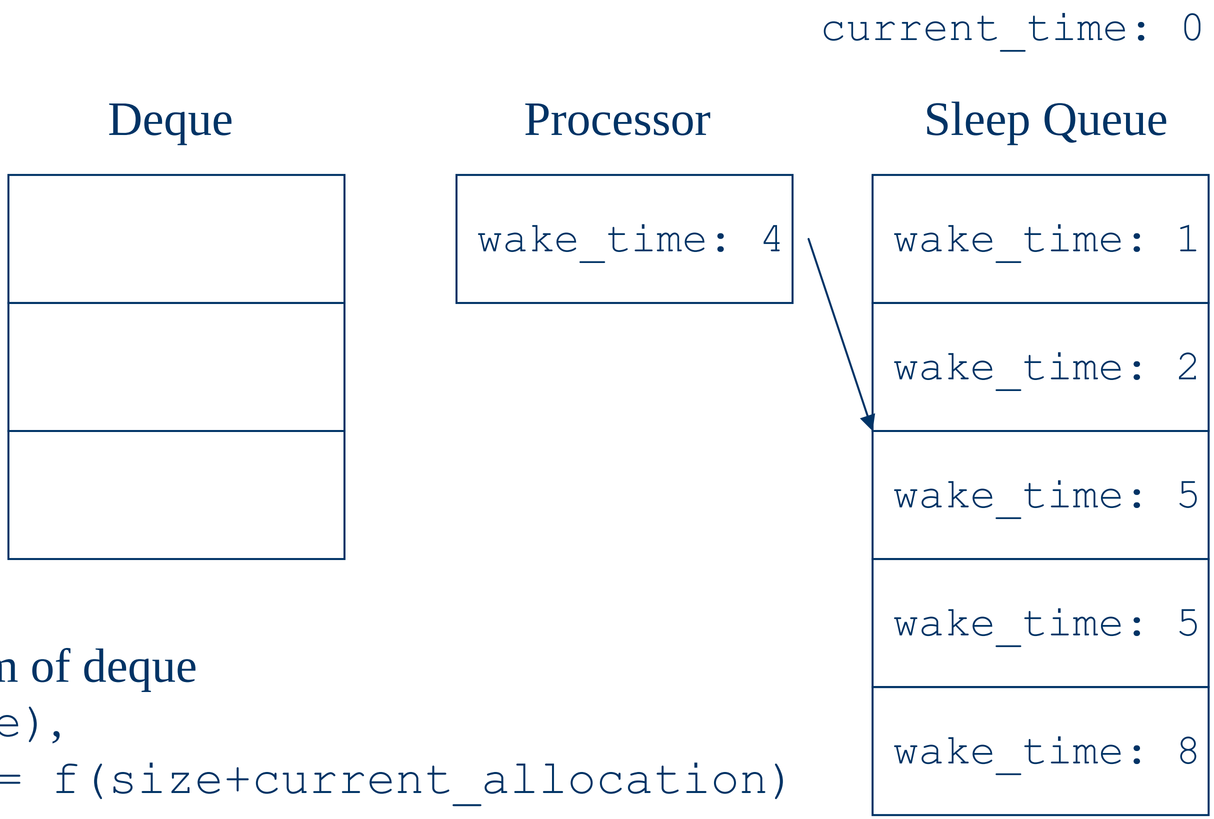
Before `malloc(size)`,

`sleep_rounds = f(size+current_allocation)`

If `sleep_rounds > 0`,

`wake_time = sleep_rounds + current_time`

Hybrid Algorithm



Get thread from bottom of deque

Before `malloc(size)`,

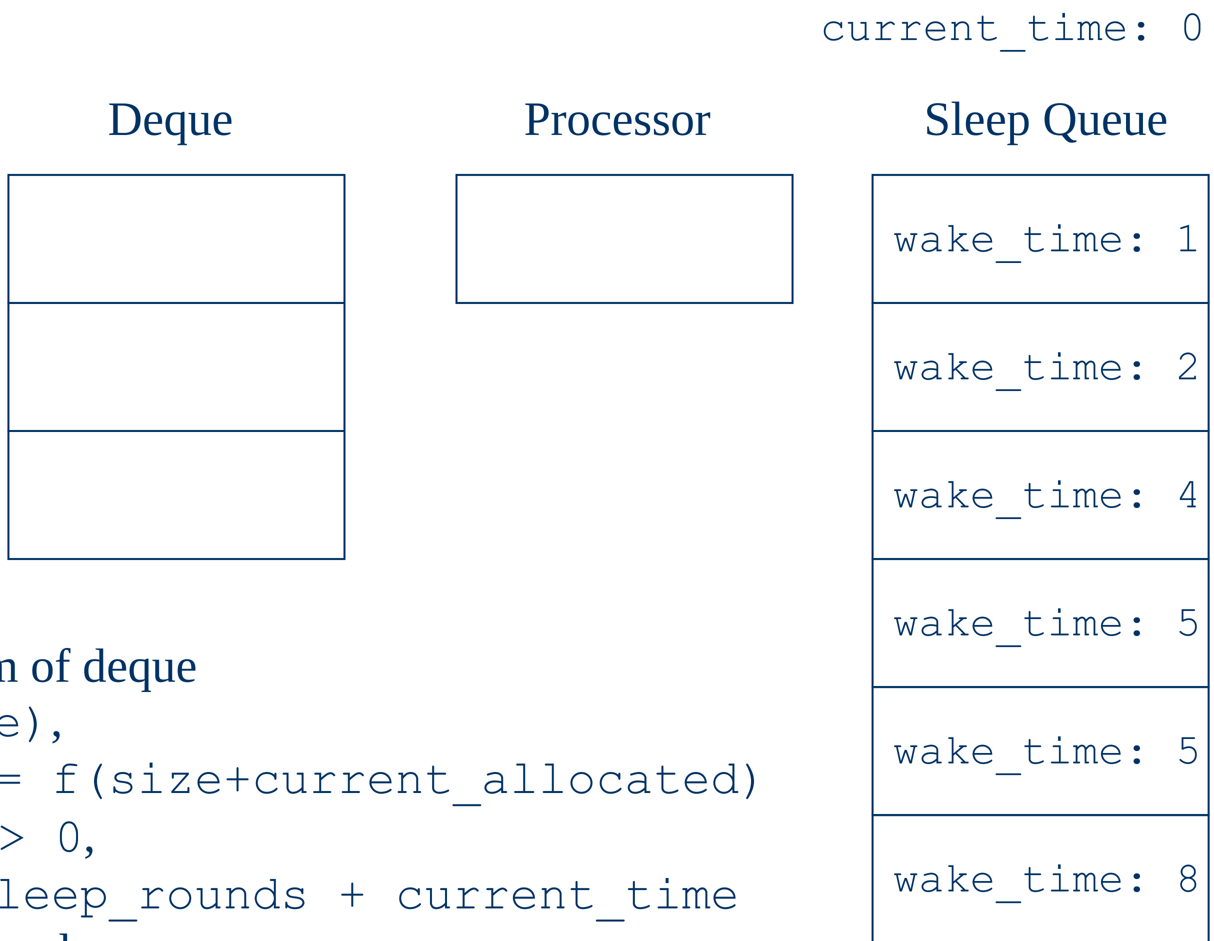
`sleep_rounds = f(size+current_allocation)`

If `sleep_rounds > 0`,

`wake_time = sleep_rounds + current_time`

and insert thread into sleep queue

Hybrid Algorithm



Get thread from bottom of deque

Before `malloc(size)`,

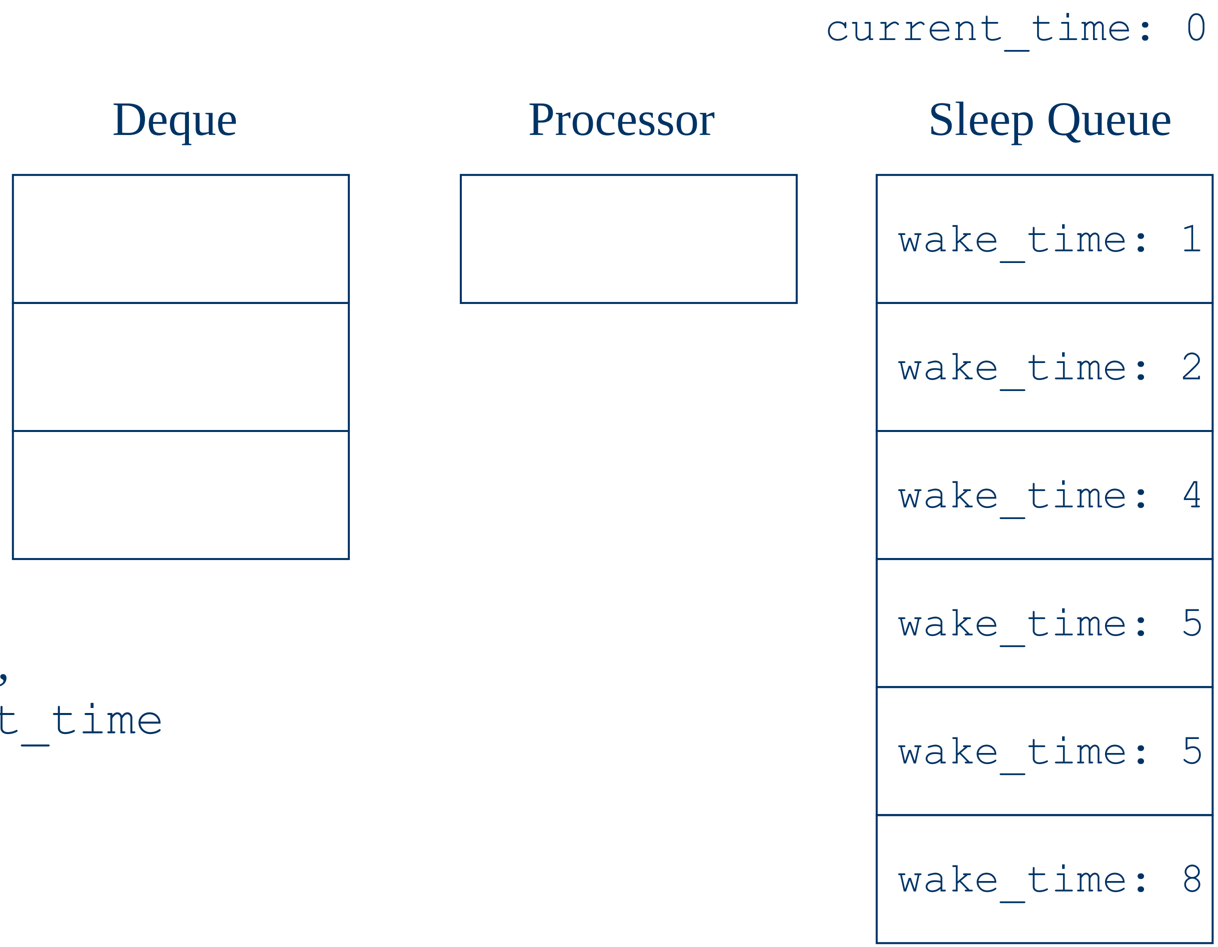
`sleep_rounds = f(size+current_allocated)`

If `sleep_rounds > 0`,

`wake_time = sleep_rounds + current_time`

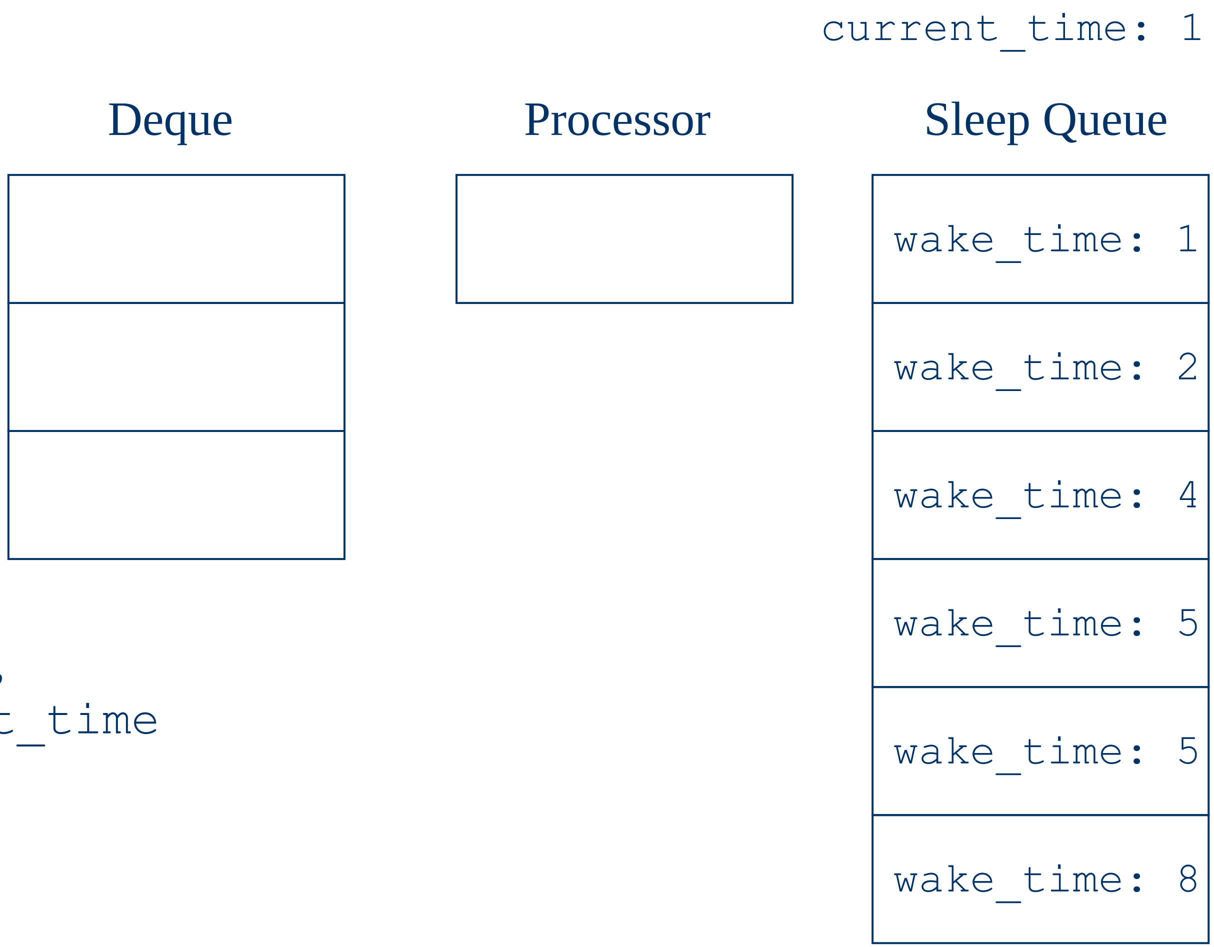
and insert thread into sleep queue

Hybrid Algorithm



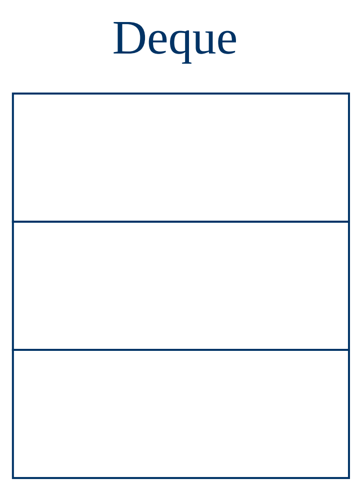
If no threads on deque, increment `current_time`

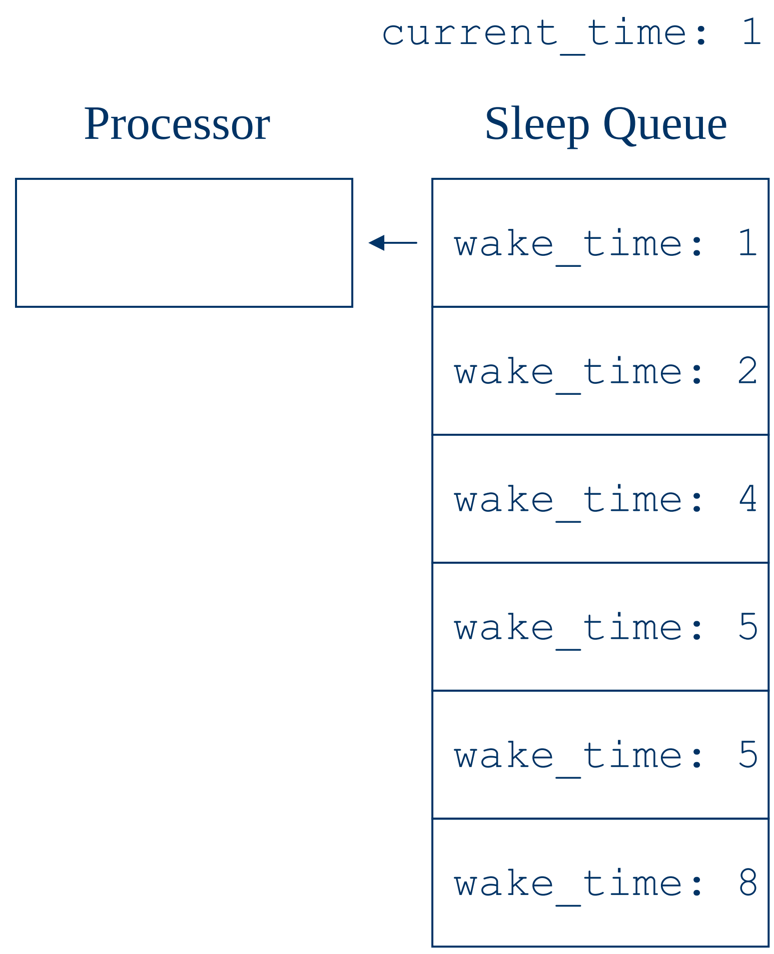
Hybrid Algorithm



If no threads on deque, increment `current_time`

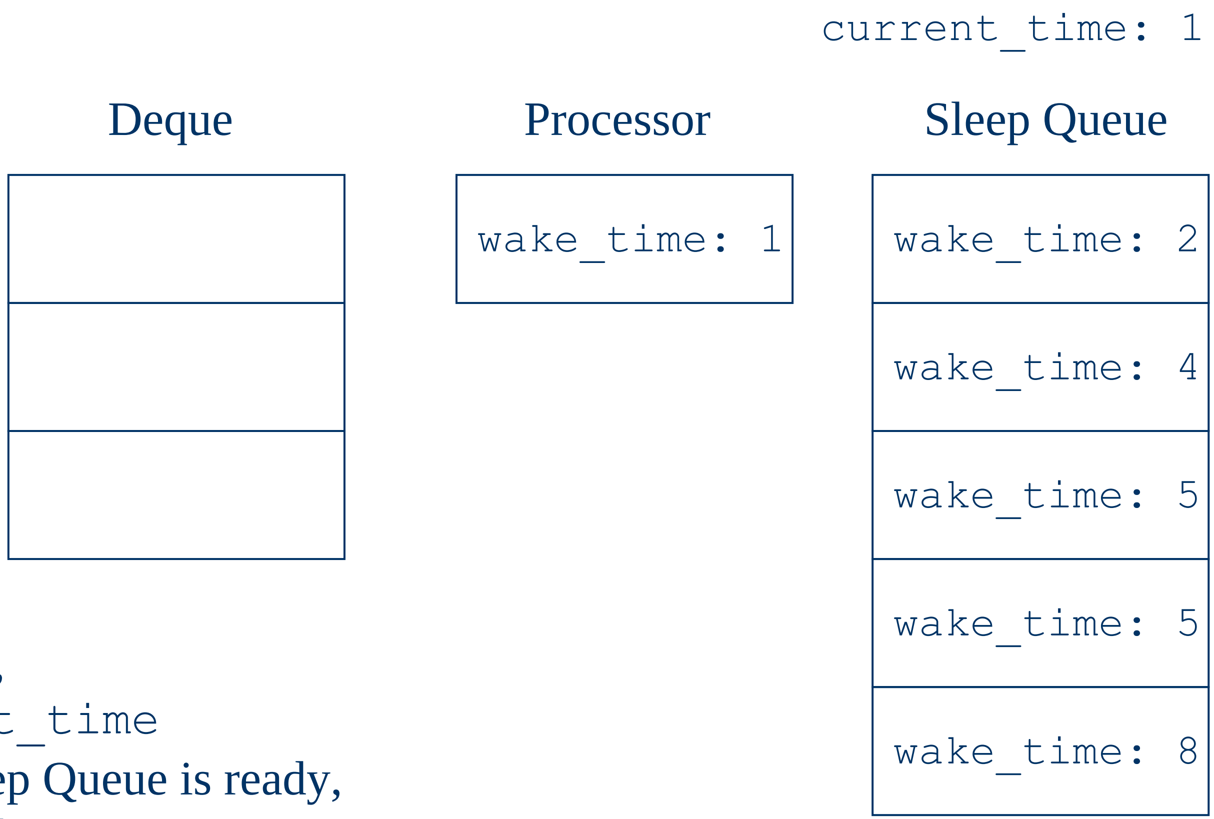
Hybrid Algorithm





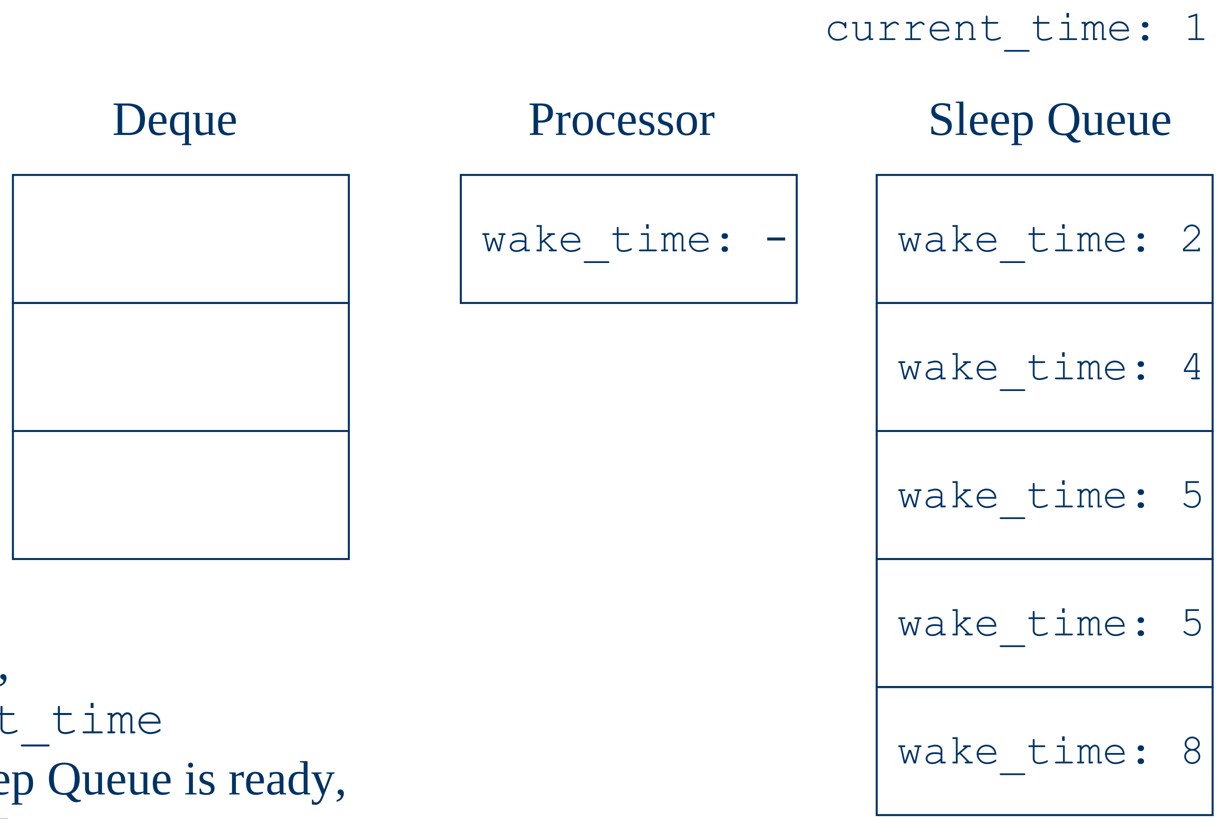
- If no threads on deque,
  - increment `current_time`
  - if first thread in Sleep Queue is ready,
    - get thread from Sleep Queue

Hybrid Algorithm



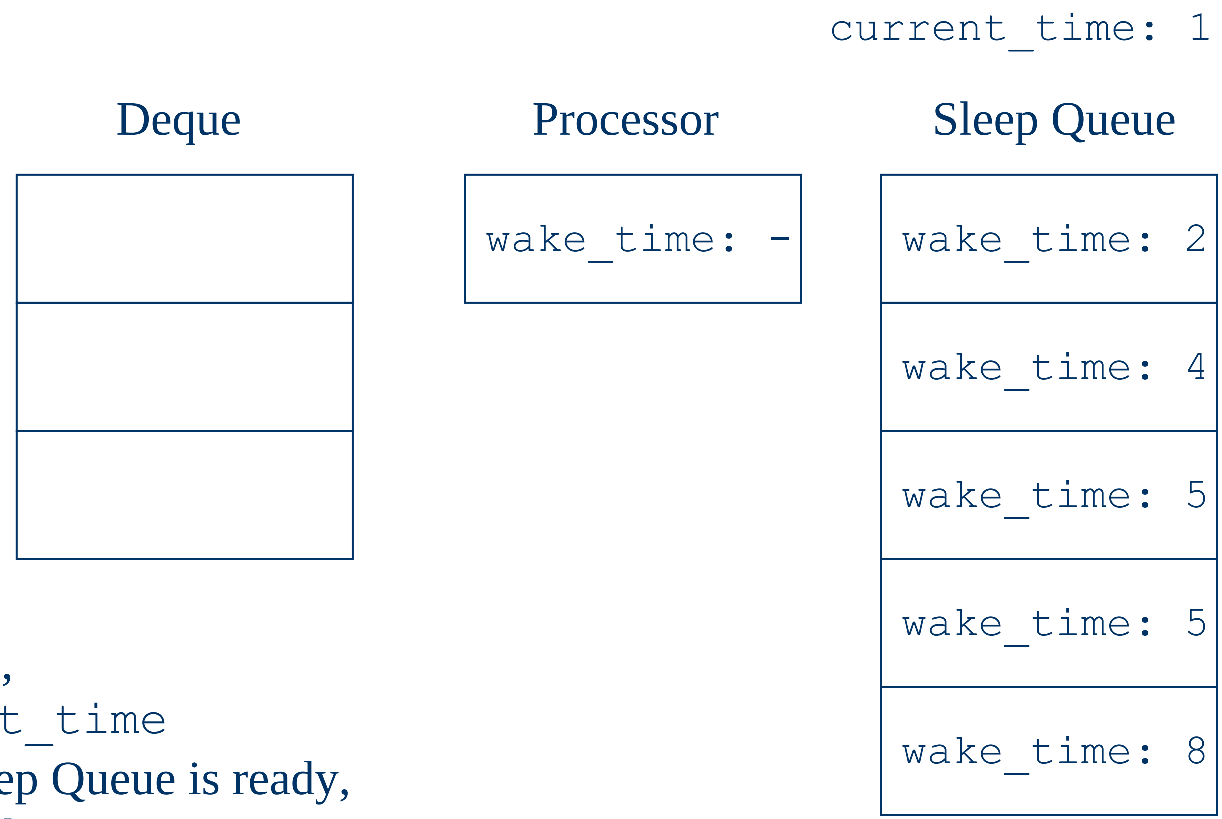
If no threads on deque,
- increment `current_time`
- if first thread in Sleep Queue is ready,
  - get thread from Sleep Queue

Hybrid Algorithm



If no threads on deque,
- increment `current_time`
- if first thread in Sleep Queue is ready,
  - get thread from Sleep Queue
  - reset `wake_time` and `current_allocated`
  - execute it

Hybrid Algorithm



If no threads on deque,
- increment `current_time`
- if first thread in Sleep Queue is ready,
  - get thread from Sleep Queue
  - reset `wake_time` and `current_allocated`
  - execute it
- otherwise, work-steal

How long to Sleep?

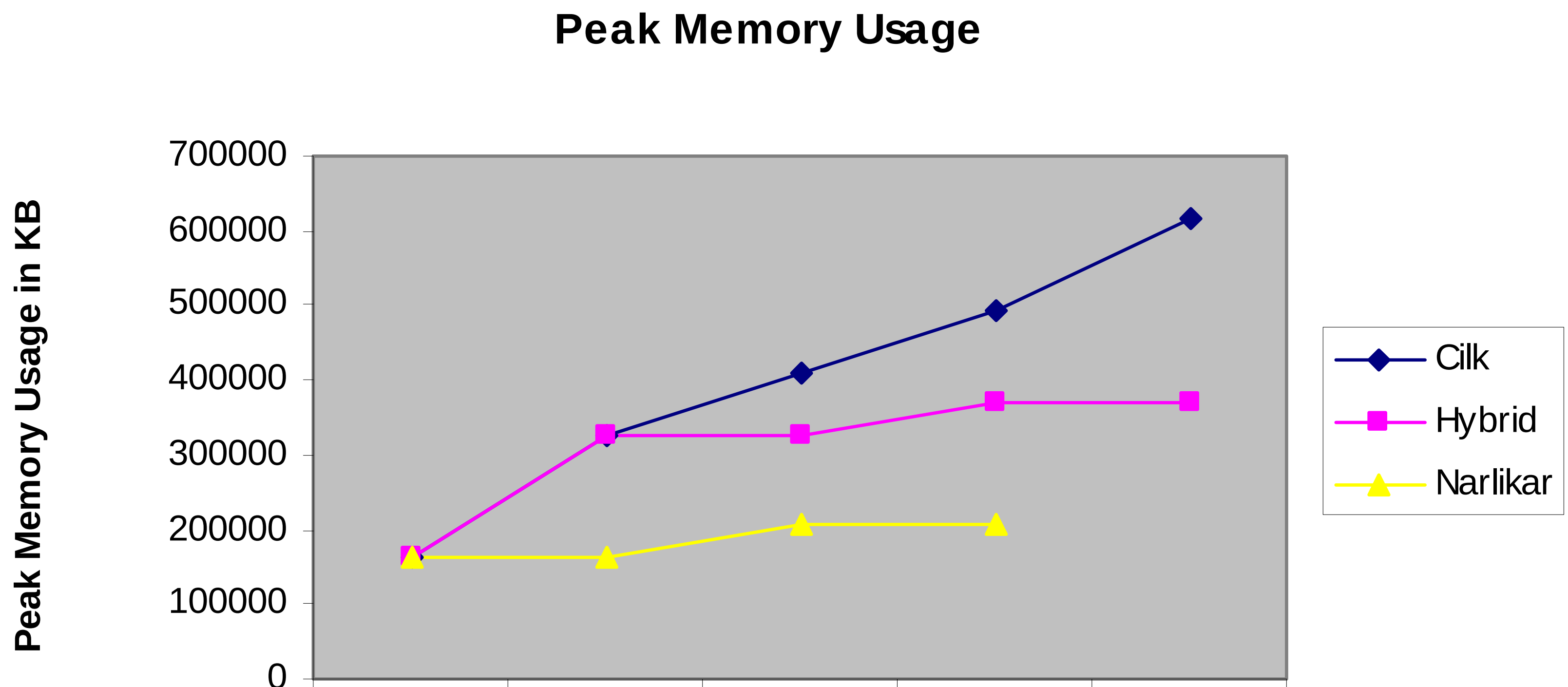
- Want sleep time to be proportional to the size of the memory allocation
- Increment time on every work-steal attempt
- Scale with number of processors
- Place for future improvement?

<u>Current function</u>

```
sleep_rounds = floor(size/(α+β*p))
```

α and β are “voodoo” parameters

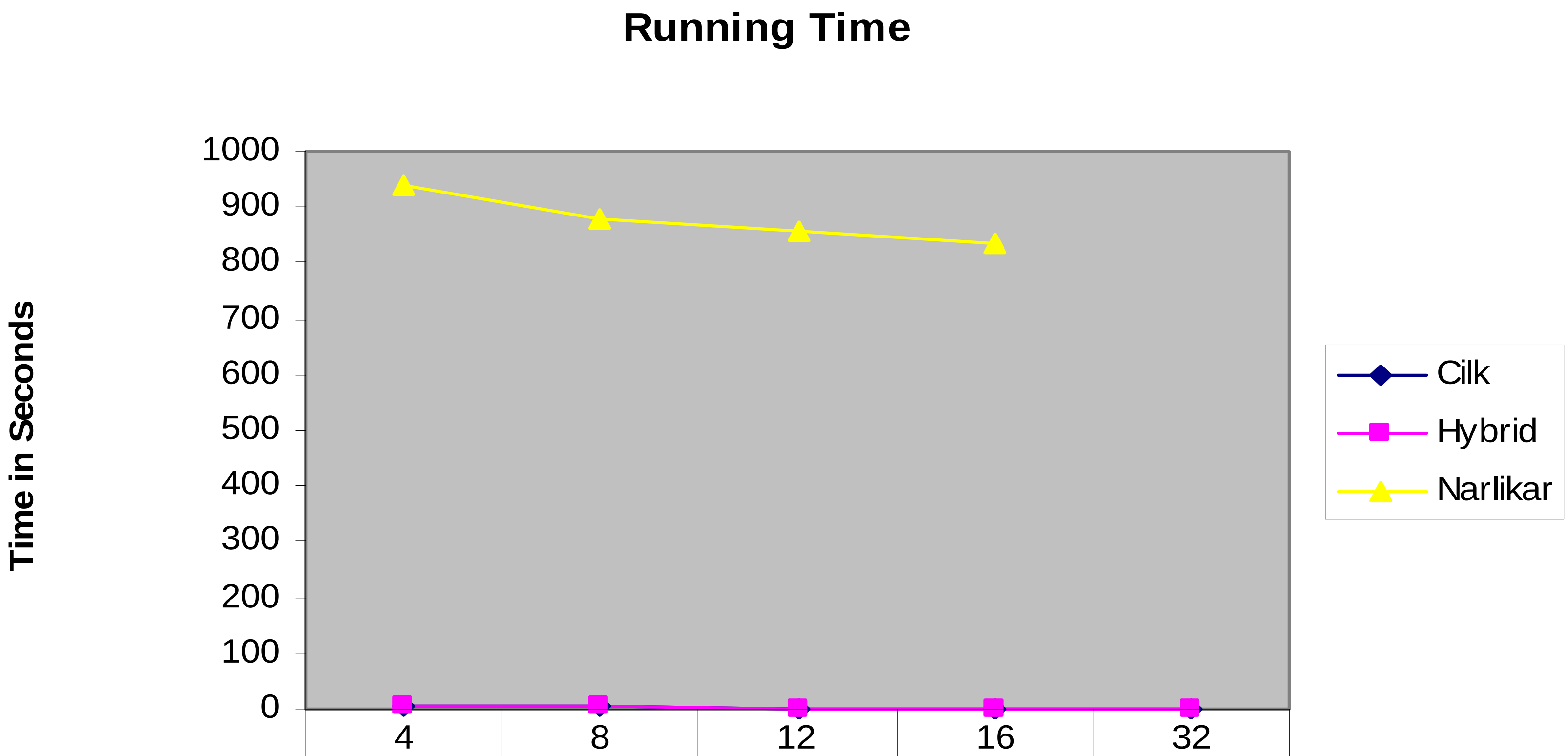
Empirical Results



	4	8	12	16	32
Cilk	163840	327680	409600	491520	614400
Hybrid	163840	327680	327680	368640	368640
Narlikar	163840	163840	204800	204800	

Number of Processors

Empirical Results



	4	8	12	16	32
Cilk	6.3	3.2	2.3	1.8	1.8
Hybrid	7.8	4.5	2.6	2.3	2.2
Narlikar	937	878	855	837	

Number of Processors

Future Work on Hybrid Scheduler

- Find the best sleep function and values for “voodoo” parameters
- Optimize the implementation to reduce scheduling overhead
- Determine theoretical space bound
- More detailed empirical analysis

Conclusions

- Narlikar scheduler provides a provably good space bound but incurs a large scheduling overhead
- It appears that it is possible to achieve space usage that scales well with the number of processors while retaining much of the efficiency of work-stealing