scheduling (finish) / threads

last time

lottery scheduler
- randomness maybe not so great in practice

shortest *remaining* time first
- special case: thread interrupted when almost done

multi-level feedback queue: SRTF approximation with priority
- heuristic: thread CPU bursts consistent
- key idea 1: priority estimates CPU burst length
- key idea 2: time quantum at priority X $\sim$ CPU burst length at priority X
- oscillating behvaior

MLFQ in practice: need to deal with starvation

Linux Completely Fair Scheduler (CFS)
- prioritize by virtual runtime
- like round robin, with adjustments for programs doing I/O
- supports proportional share (today)

CFS: tracking runtime

each thread has a *virtual runtime* ($\sim$ how long it's run)

incremented when run based how long it runs

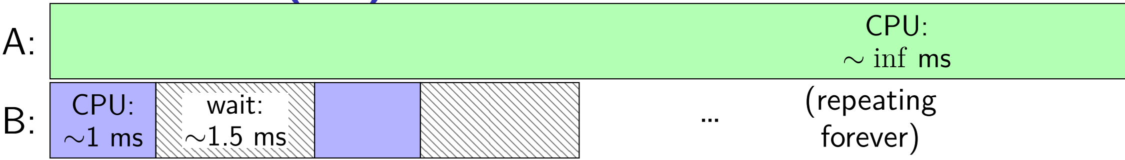
adjustments for threads that are new or were sleeping

- too big an advantage to start at runtime `0`

scheduling decision: run thread with lowest virtual runtime

- data structure: balanced tree

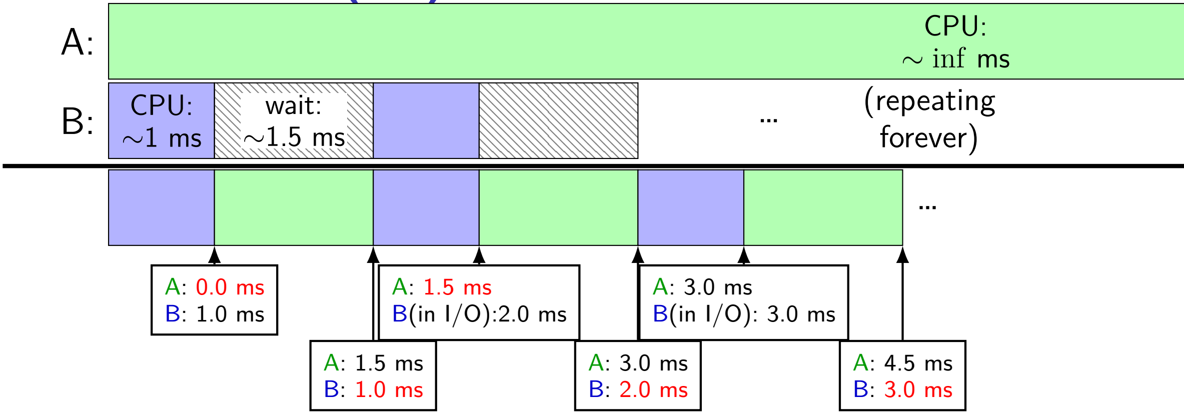
CFS exercise (0a)



suppose programs A, B; max 1 ms time quanta

with CFS (and equal weights) and **no adjustments to virtual time for programs waking up from sleep**, **about** what portion of CPU does program A get?

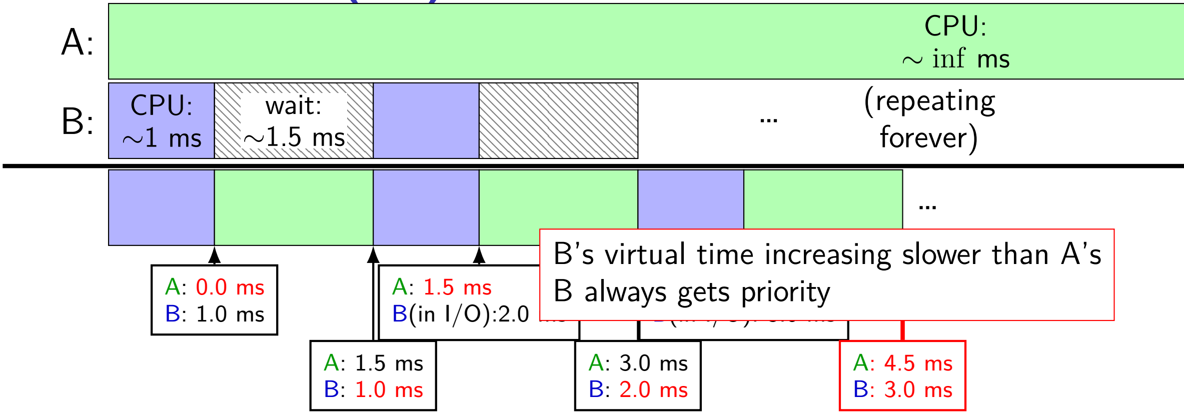
CFS exercise (0a)



suppose programs A, B; max 1 ms time quanta

with CFS (and equal weights) and **no adjustments to virtual time for programs waking up from sleep**, **about** what portion of CPU does program A get?

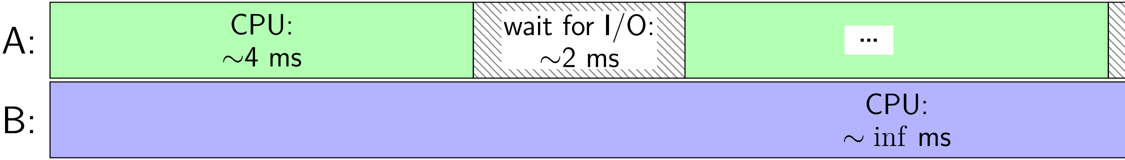
CFS exercise (0a)



suppose programs A, B; max 1 ms time quanta

with CFS (and equal weights) and **no adjustments to virtual time for programs waking up from sleep**, **about** what portion of CPU does program A get?

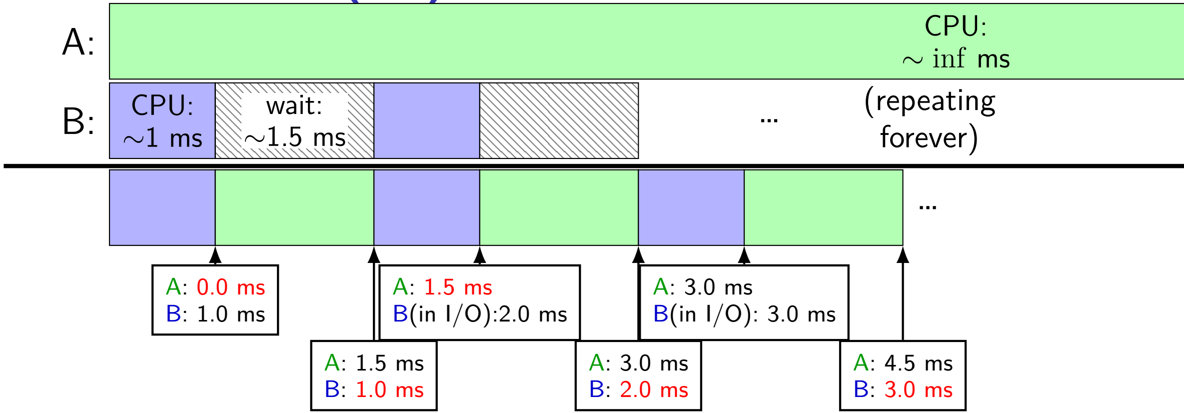
CFS exercise (0b)



suppose programs A, B; 1 ms time quata

with CFS (and equal weights) and **no adjustments to virtual time for programs waking up from sleep**, **about** what portion of CPU does program A get?

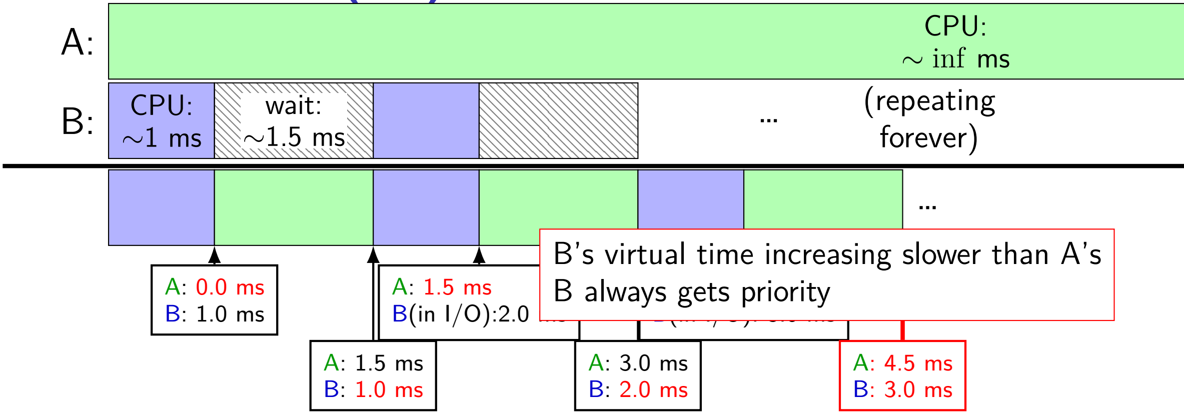
CFS exercise (0a)



suppose programs A, B; max 1 ms time quanta

with CFS (and equal weights) and **no adjustments to virtual time for programs waking up from sleep**, **about** what portion of CPU does program A get?

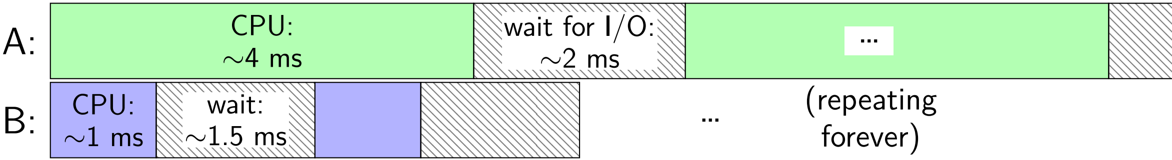
CFS exercise (0a)



suppose programs A, B; max 1 ms time quanta

with CFS (and equal weights) and **no adjustments to virtual time for programs waking up from sleep**, **about** what portion of CPU does program A get?

CFS exercise (0c)



suppose programs A, B with alternating CPU + I/O as above

with CFS (and equal weights) and **no adjustments to virtual time for programs waking up from sleep**, **about** what portion of CPU does program A get?

exercise solution

if A, B, were running alone, could get at most 1/2 the CPU

B can't use that much time

so B will run 2/5ths of the time (the most it can)

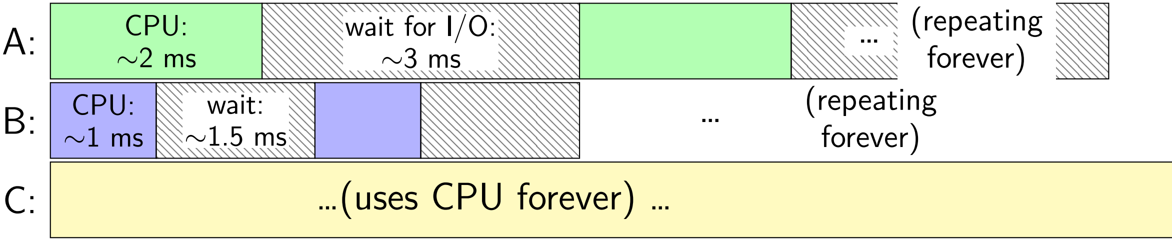
so B will almost always have lower virtual time than A

A will get the remaining about 3/5ths

exception: time both A and B are both doing I/O

exception: extra time A gets to run if no preemption during its time quantum?

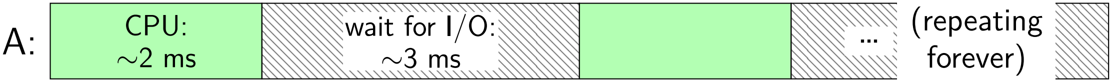
CFS exercise (1)



suppose programs A, B, C with alternating CPU + I/O as above

with CFS (and equal weights) and **no adjustments to virtual time for programs waking up from sleep**, about what portion of CPU does program A get?

CFS exercise: maximum time for A



A running alone: A runs 2/5ths of the time

A, B, C sharing fairly: each runs 1/3rd of the time

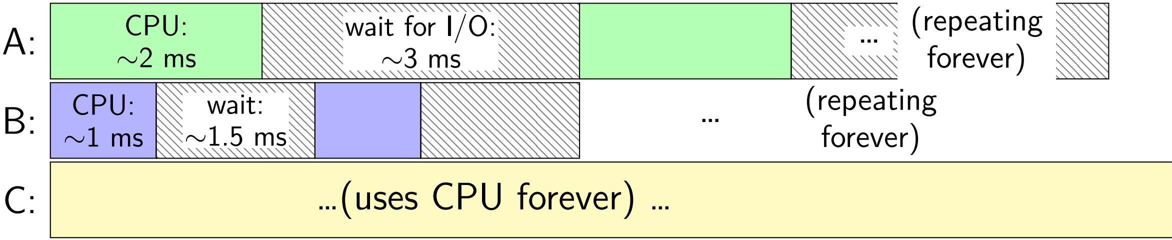
- if A used more than 1/3rd of the time...
- then it would have a higher virtual time...
- and B and C would catch up
- (and same for B or C)

result: A runs at most 1/3rd of the time...

unless B can't use its full share because of I/O

- (because of being interrupted by A too much?)

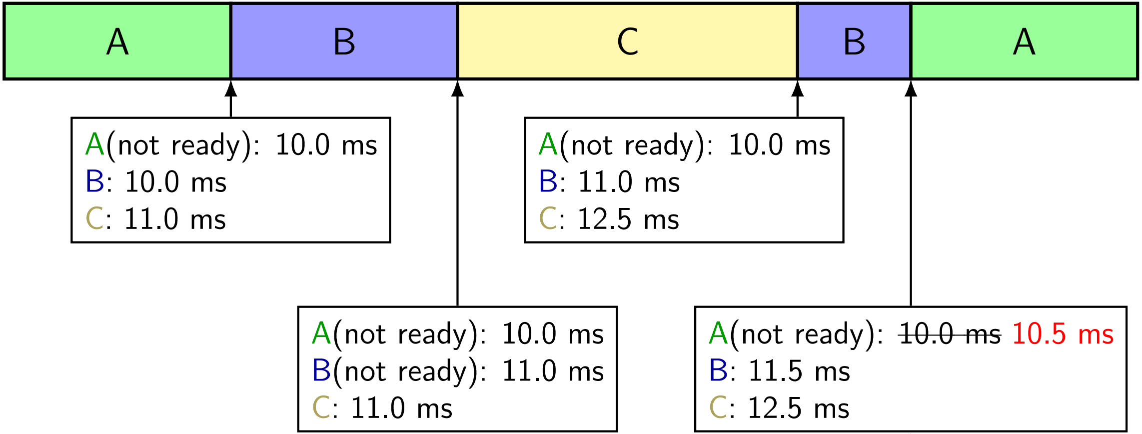
CFS exercise (2)



suppose we add adjustments to virtual time for waking up from sleep

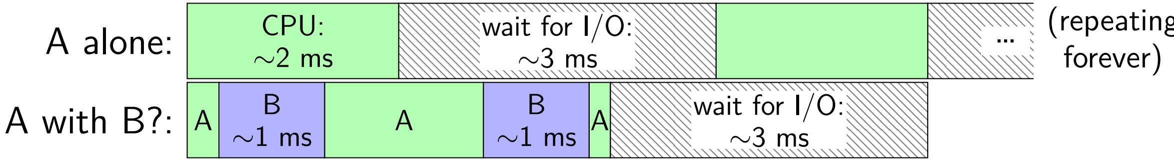
expected direction of change in how much compute time A gets?

CFS exercise: A disadvantage from sleep



if scheduler configured to limit advantage
of newly ready threads enough:
A might 'lose' some virtual time
because it waits for I/O "too long"
and since A waits for I/O longer

CFS exercise: A interrupted by B?



A interrupted by B a bunch sometimes...?

might not start I/O as often

might not be able to run 1/3rd of the time

e.g. sometimes $2/(2+2+3) \approx 28\%$ of CPU

handling *proportional* sharing

solution: multiply used time by weight

e.g. 1 ms of CPU time costs process 2 ms of virtual time

higher weight $\implies$ process less favored to run

CFS: tracking runtime

each thread has a *virtual runtime* ($\sim$ how long it's run)

incremented when run based how long it runs

more/less important thread? multiply adjustments by factor

adjustments for threads that are new or were sleeping

- too big an advantage to start at runtime `0`

scheduling decision: run thread with lowest virtual runtime

- data structure: balanced tree

CFS quantum lengths goals

first priority: constrain minimum quantum length (default: 0.75ms)

avoid too-frequent context switching

second priority: run every process “soon” (default: 6ms)

avoid starvation

CFS quantum lengths goals

first priority: constrain minimum quantum length (default: 0.75ms)

avoid too-frequent context switching

second priority: run every process “soon” (default: 6ms)

avoid starvation

quantum $\approx$ max(fixed window / num processes, minimum quantum)

CFS: avoiding excessive context switching

conflicting goals:

schedule newly ready tasks immediately

(assuming less virtual time than current task)

avoid excessive context switches

CFS rule:
if virtual time of new task $<$ current virtual time by threshold

default threshold: 1 ms

(otherwise, wait until quantum is done)

other CFS parts

dealing with multiple CPUs

handling groups of related tasks

special 'idle' or 'batch' task settings

…

CFS versus others

very similar to *stride scheduling*

- presented as a deterministic version of lottery scheduling
- Waldspurger and Weihl, "Stride Scheduling: Deterministic Proportional-Share Resource Management" (1995, same authors as lottery scheduling)

very similar to *weighted fair queuing*

- used to schedule network traffic
- Demers, Keshav, and Shenker, "Analysis and Simulation of a Fair Queuing Algorithm" (1989)

which scheduler should I choose?

I care about...

CPU throughput: first-come first-serve

average response time: SRTF approximation

I/O throughput: SRTF approximation

fairness — medium-term CPU usage: something like Linux CFS

fairness — wait time: something like RR

(not covered this semester) real-world deadlines: earliest deadline first or similar

favoring certain users: strict priority

a note on multiprocessors

what about multicore?

want two cores to schedule without waiting for each other

want to keep process on same core (better for cache)

what core to preempt when three+ choices?

common approach:

- separate ready list per core
- regularly 'rebalance' threads between cores

which scheduler should I choose?

I care about...

CPU throughput: first-come first-serve

average response time: SRTF approximation

I/O throughput: SRTF approximation

fairness — medium-term CPU usage: something like Linux CFS

fairness — wait time: something like RR

(not covered this semester) real-world deadlines: earliest deadline first or similar

favoring certain users: strict priority

using threads

why threads?

concurrency: different things happening at once

- one thread per user of web server?
- one thread per page in web browser?
- one thread to play audio, one to read keyboard, ...?
- ...

parallelism: do same thing with more resources

- multiple processors to speed-up simulation (life assignment)

aside: alternate threading models

we'll talk about kernel threads

OS scheduler deals **directly** with threads

alternate idea: library code handles threads

kernel doesn't know about threads w/in process

hierarchy of schedulers: one for processes, one within each process

not currently common model — awkward with multicore

thread versus process state

thread state — kept in **thread control block**

- registers (including stack pointer, program counter)
- scheduling state (runnable, waiting, …)
- other information?
- …

process state — kept in **process control block**

- address space (memory layout, heap location, …)
- open files
- process id
- list of thread control blocks
- …

Linux idea: task_struct

Linux model: single "task" structure = thread

pointers to address space, open file list, etc.

pointers can be shared

- e.g. shared open files: open fd 4 in one task → all sharing can use fd 4

`fork()`-like system call "clone": choose what to share

- `clone(0, ...)` — similar to `fork()`
- `clone(CLONE_FILES, ...)` — like fork(), but **sharing** open files
- `clone(CLONE_VM, new_stack_pointer, ...)` — like fork(), but **sharing** address space

Linux idea: task_struct

Linux model: single "task" structure = thread

pointers to address space, open file list, etc.

pointers can be shared

- e.g. shared open files: open fd 4 in one task → all sharing can use fd 4

`fork()`-like system call "clone": choose what to share

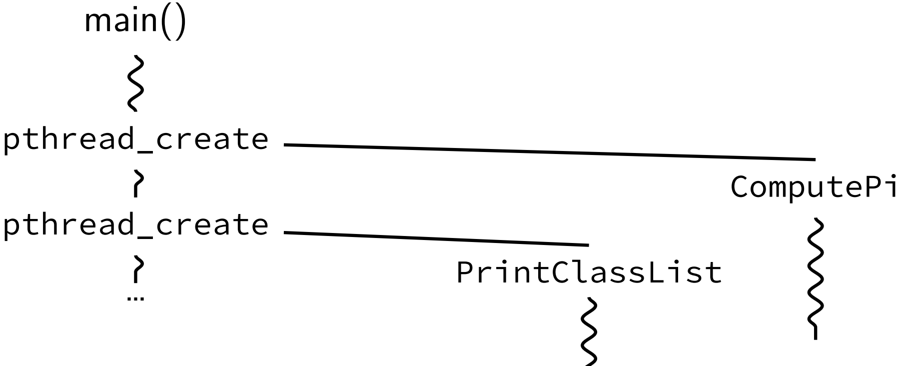
- `clone(0, ...)` — similar to `fork()`
- `clone(CLONE_FILES, ...)` — like fork(), but **sharing** open files
- `clone(CLONE_VM, new_stack_pointer, ...)` — like fork(), but **sharing** address space

advantage: no special logic for threads (mostly)

- two threads in same process = tasks sharing everything possible

pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```



pthread_create

```
void *ComputePi(void *argument) { ... }
void *PrintClassList(void *argument) { ... }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

pthread_create arguments:

thread identifier

function to run

- thread starts here, terminates if this function returns

thread attributes (extra settings) and function argument

pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

pthread_create arguments:

thread identifier

function to run

thread starts here, terminates if this function returns

thread attributes (extra settings) and function argument

pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

pthread_create arguments:

thread identifier

function to run

- thread starts here, terminates if this function returns

thread attributes (extra settings) and function argument

pthread_create

```
void *ComputePi(void *argument) { ... }
void *PrintClassList(void *argument) { ... }
int main() {
    pthread_t pi_thread, list_thread;
    pthread_create(&pi_thread, NULL, ComputePi, NULL);
    pthread_create(&list_thread, NULL, PrintClassList, NULL);
    ... /* more code */
}
```

pthread_create arguments:

thread identifier

function to run

thread starts here, terminates if this function returns

thread attributes (extra settings) and function argument

a threading race

```
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In the thread\n");
    return NULL;
}
int main() {
    printf("About to start thread\n");
    pthread_t the_thread;
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done starting thread\n");
    return 0;
}
```

My machine: outputs `In the thread` about 4% of the time.

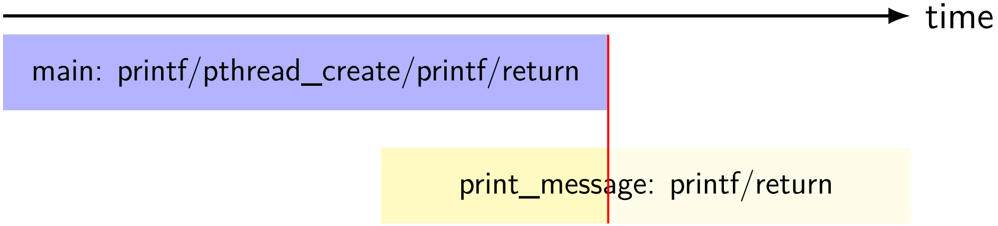
What happened?

a race

returning from main exits the entire process (all its threads)

same as calling exit; not like other threads

race: main's return 0 or print_message's printf first?



return from main
ends all threads
in the process

fixing the race (version 1)

```
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In the thread\n");
    return NULL;
}
int main() {
    printf("About to start thread\n");
    pthread_t the_thread;
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done starting thread\n");
    pthread_join(the_thread, NULL);  /* WAIT FOR THREAD */
    return 0;
}
```

fixing the race (version 2; not recommended)

```
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In the thread\n");
    return NULL;
}
int main() {
    printf("About to start thread\n");
    pthread_t the_thread;
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done starting thread\n");
    pthread_exit(NULL);
}
```

pthread_join, pthread_exit

`pthread_join`: wait for thread, retrieves its return value

- like `waitpid`, but for a thread
- return value is pointer to anything

`pthread_exit`: exit current thread, returning a value

- like `exit` or returning from main, but for a single thread
- same effect as returning from function passed to pthread_create

sum example (only globals)

```
int values[1024];
int results[2];
void *sum_front(void *ignored_argument) {
    int sum = 0;
    for (int i = 0; i < 512; ++i)
        sum += values[i];
    results[0] = sum;
    return NULL;
}
void *sum_back(void *ignored_argument) {
    int sum = 0;
    for (int i = 512; i < 1024; ++i)
        sum += values[i];
    results[1] = sum;
    return NULL;
}
int sum_all() {
    pthread_t sum_front_thread, sum_back_thread;
    pthread_create(&sum_front_thread, NULL, sum_front, NULL);
    pthread_create(&sum_back_thread, NULL, sum_back, NULL);
    pthread_join(sum_front_thread, NULL);
    pthread_join(sum_back_thread, NULL);
    return results[0] + results[1];
}
```

sum example (only globals)

```
int values[1024];
int results[2];
void *sum_front(void *ignored_argument) {
    int sum = 0;
    for (int i = 0; i < 512; ++i)
        sum += values[i];
    results[0] = sum;
    return NULL;
}
void *sum_back(void *ignored_argument) {
    int sum = 0;
    for (int i = 512; i < 1024; ++i)
        sum += values[i];
    results[1] = sum;
    return NULL;
}
int sum_all() {
    pthread_t sum_front_thread, sum_back_thread;
    pthread_create(&sum_front_thread, NULL, sum_front, NULL);
    pthread_create(&sum_back_thread, NULL, sum_back, NULL);
    pthread_join(sum_front_thread, NULL);
    pthread_join(sum_back_thread, NULL);
    return results[0] + results[1];
}
```

values, results: global variables — shared

sum example (only globals)

two different functions

happen to be the same except for some numbers

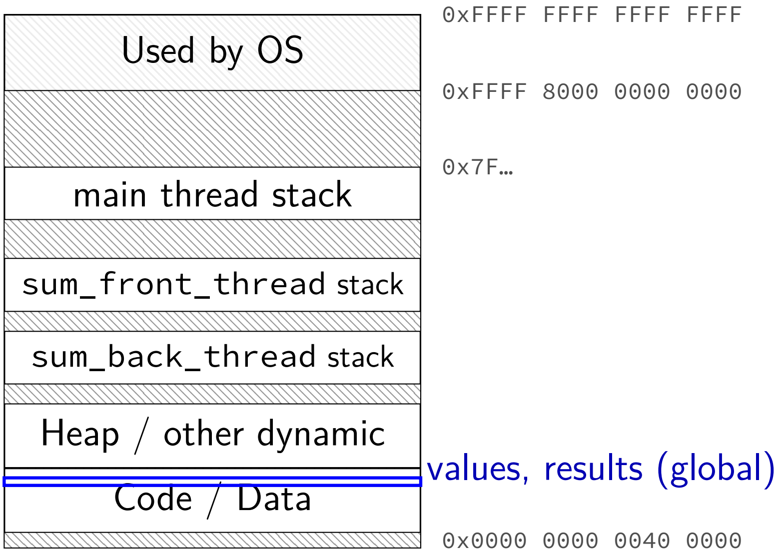
```
int values[1024];
int results[2];
void *sum_front(void *
    int sum = 0;
    for (int i = 0; i < 512; ++i)
        sum += values[i];
    results[0] = sum;
    return NULL;
}
void *sum_back(void *ignored_argument) {
    int sum = 0;
    for (int i = 512; i < 1024; ++i)
        sum += values[i];
    results[1] = sum;
    return NULL;
}
int sum_all() {
    pthread_t sum_front_thread, sum_back_thread;
    pthread_create(&sum_front_thread, NULL, sum_front, NULL);
    pthread_create(&sum_back_thread, NULL, sum_back, NULL);
    pthread_join(sum_front_thread, NULL);
    pthread_join(sum_back_thread, NULL);
    return results[0] + results[1];
}
```

sum example (only globals)

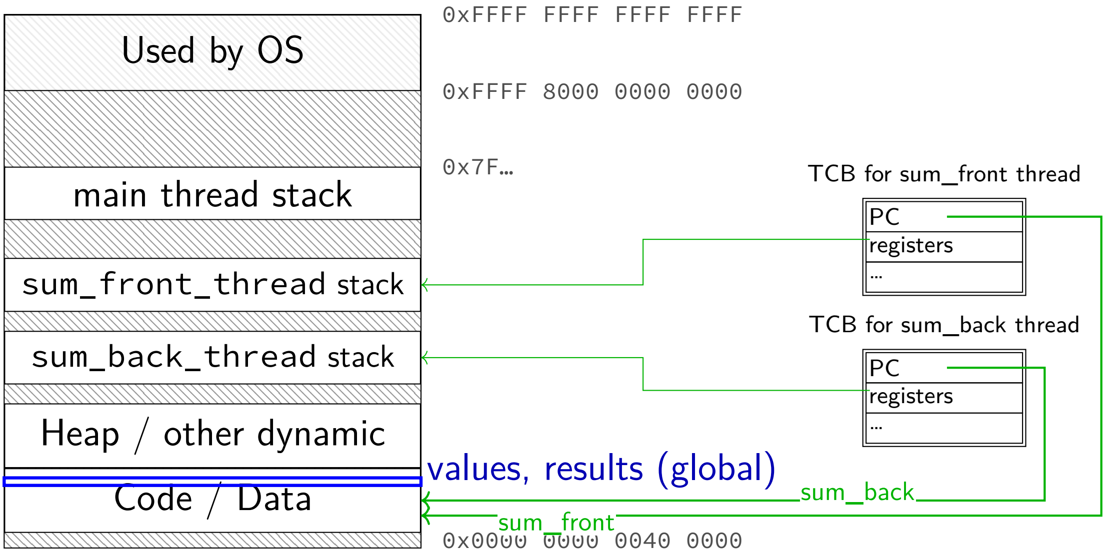
values returned from threads
via global array instead of return value
(partly to illustrate that memory is shared,
partly because this pattern works when we don't join (later))

```
int value
int resul
void *sum
    int s
    for (
        sum += values[i];
    results[0] = sum;
    return NULL;
}
void *sum_back(void *ignored_argument) {
    int sum = 0;
    for (int i = 512; i < 1024; ++i)
        sum += values[i];
    results[1] = sum;
    return NULL;
}
int sum_all() {
    pthread_t sum_front_thread, sum_back_thread;
    pthread_create(&sum_front_thread, NULL, sum_front, NULL);
    pthread_create(&sum_back_thread, NULL, sum_back, NULL);
    pthread_join(sum_front_thread, NULL);
    pthread_join(sum_back_thread, NULL);
    return results[0] + results[1];
}
```

thread_sum memory layout



thread_sum memory layout



sum example (to global, with thread IDs)

```
int values[1024];
int results[2];
void *sum_thread(void *argument) {
    int id = (int) argument;
    int sum = 0;
    for (int i = id * 512; i < (id + 1) * 512; ++i) {
        sum += values[i];
    }
    results[id] = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return results[0] + results[1];
}
```

sum example (to global, with thread IDs)

values, results: global variables — shared

```
int values[1024];
int results[2];
void *sum_thread(void *argument) {
    int id = (int) argument;
    int sum = 0;
    for (int i = id * 512; i < (id + 1) * 512; ++i) {
        sum += values[i];
    }
    results[id] = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return results[0] + results[1];
}
```

sum example (info struct)

```
int values[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    ThreadInfo *my_info =  (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += values[i];
    }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

sum example (info struct)

```
int values[1024];
struct ThreadInfo
    int start, end, result;
};
void *sum_thread(void *argument) {
    ThreadInfo *my_info =  (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += values[i];
    }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

values: global variable — shared

sum example (info struct)

```
int values[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->
        sum += values[i];
    }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

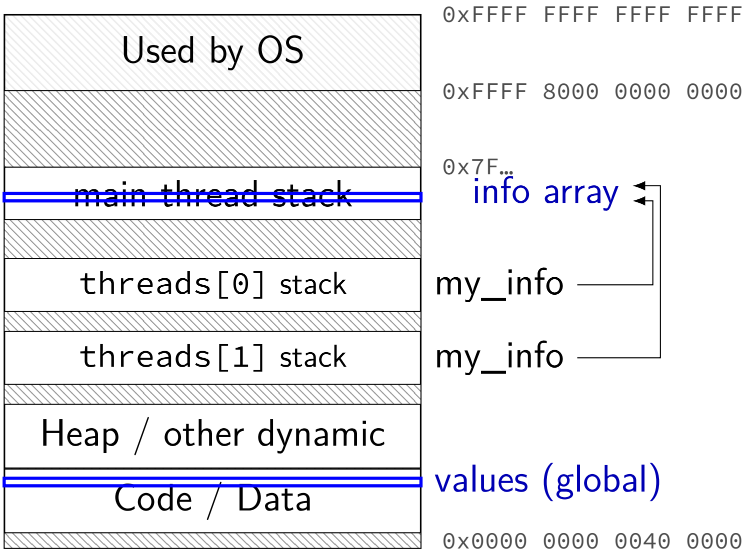
my_info: pointer to sum_all's stack

only okay because sum_all waits!

sum example (info struct)

```
int values[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    ThreadInfo *my_info =  (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += values[i];
    }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

thread_sum memory layout (info struct)



sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

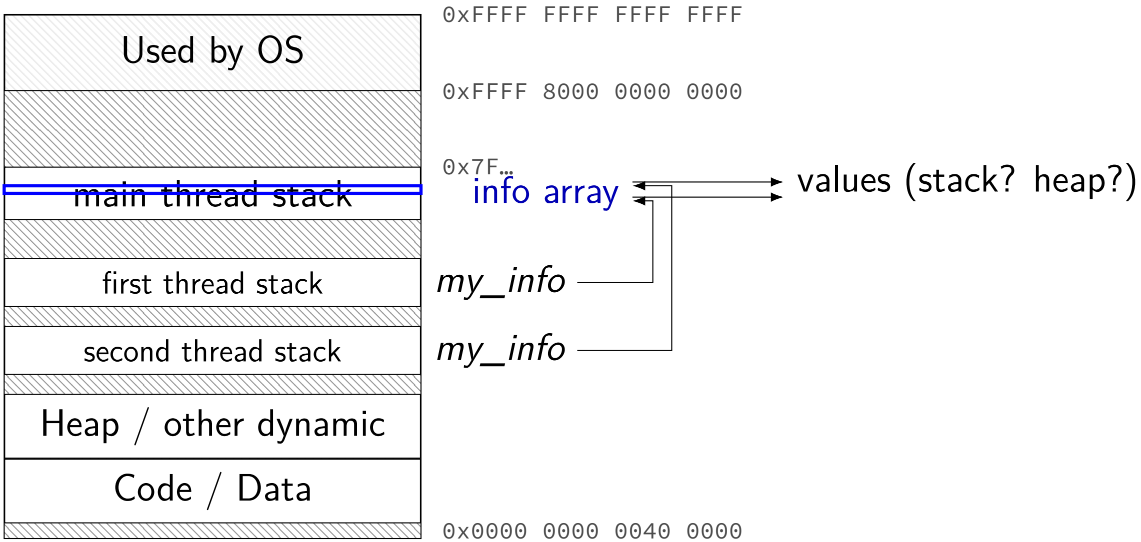
sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

program memory (to main stack)



sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result }
void *sum_thread(void *argument) {
    ...
}

ThreadInfo *start_sum_all(int *values) {
    ThreadInfo *info = new ThreadInfo[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}

int finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    delete[] info;
    return result;
}
```

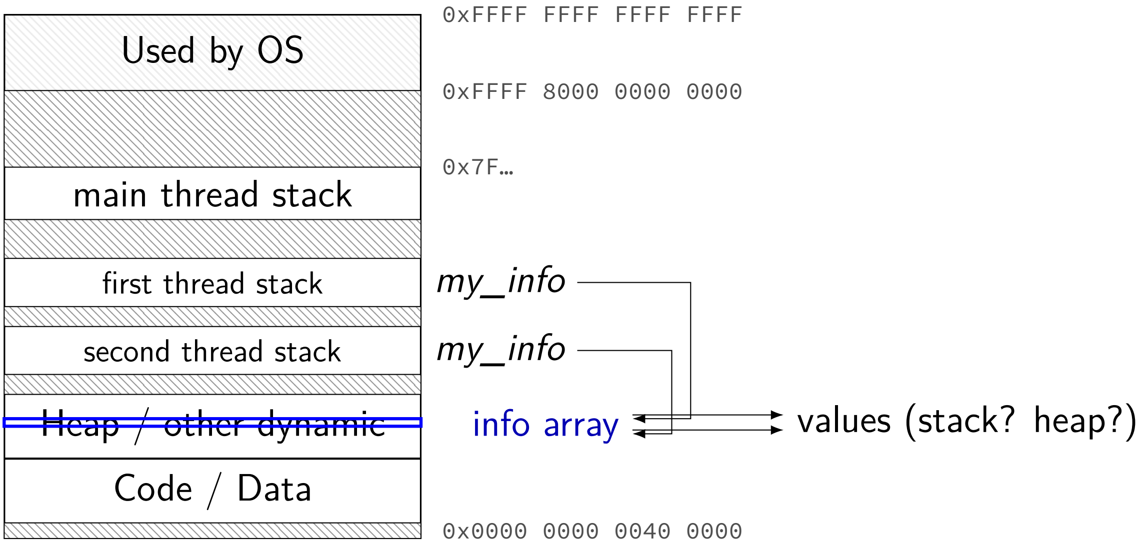
sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result }
void *sum_thread(void *argument) {
    ...
}

ThreadInfo *start_sum_all(int *values) {
    ThreadInfo *info = new ThreadInfo[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}

int finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    delete[] info;
    return result;
}
```

sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ...
}

ThreadInfo *start_sum_all(int *values) {
    ThreadInfo *info = new ThreadInfo[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}

int finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    delete[] info;
    return result;
}
```

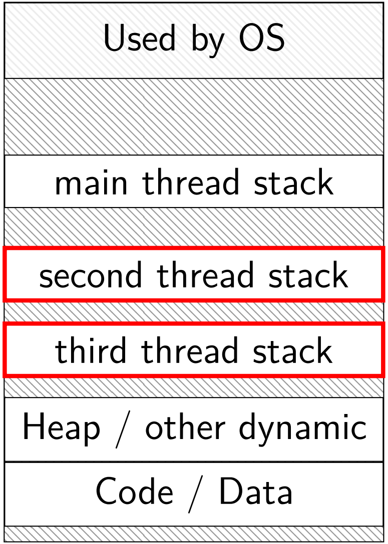
thread_sum memory (heap version)



what's wrong with this?

```
/* omitted: headers */
#include <string>
using std::string;
void *create_string(void *ignored_argument) {
  string result;
  result = ComputeString();
  return &result;
}
int main() {
  pthread_t the_thread;
  pthread_create(&the_thread, NULL, create_string, NULL);
  string *string_ptr;
  pthread_join(the_thread, (void*) &string_ptr);
  cout << "string is " << *string_ptr;
}
```

program memory



0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

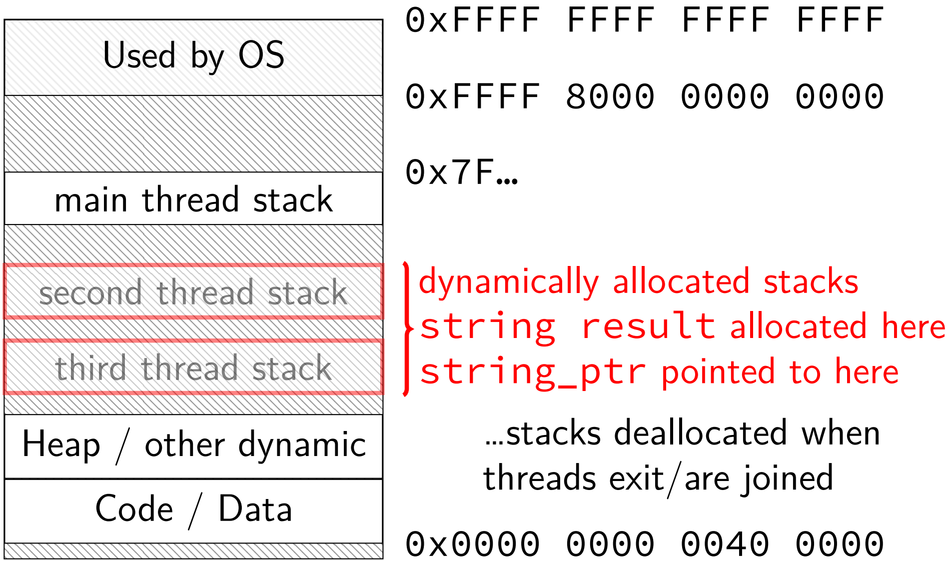
0x7F…

dynamically allocated stacks
`string result` allocated here
`string_ptr` pointed to here

…stacks deallocated when threads exit/are joined

0x0000 0000 0040 0000

program memory



thread resources

to create a thread, allocate:

new stack (how big???)

thread control block

deallocated when …

thread resources

to create a thread, allocate:

new stack (how big???)

thread control block

deallocated when ...

can deallocate stack when thread exits

but need to allow collecting return value

- same problem as for processes and waitpid

pthread_detach

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_create(&show_progress_thread, NULL, show_progress, NULL)

    /* instead of keeping pthread_t around to join thread later: */
    pthread_detach(show_progress_thread);
}

int main() {
    spawn_show_progress_thread();
    do_other_stuff();
    ...
}
```

detach = don't care about return value, etc.
system will deallocate when thread terminates

starting threads detached

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_attr_t attrs;
    pthread_attr_init(&attrs);
    pthread_attr_setdetachstate(&attrs, PTHREAD_CREATE_DETACHED);
    pthread_create(&show_progress_thread, attrs,
                   show_progress, NULL);
    pthread_attr_destroy(&attrs);
}
```

setting stack sizes

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_attr_t attrs;
    pthread_attr_init(&attrs);
    pthread_attr_setstacksize(&attrs, 32 * 1024 /* bytes */);
    pthread_create(&show_progress_thread, attrs,
                   show_progress, NULL);
}
```

a note on error checking

from pthread_create manpage:

```
ERRORS
       EAGAIN Insufficient  resources  to  create  another thread, or a system-imposed limit on the number of
              threads was encountered.  The latter case may occur in two ways: the RLIMIT_NPROC soft resource
              limit  (set  via  setrlimit(2)),  which  limits  the  number of process for a real user ID, was
              reached; or the kernel's system-wide limit on the number of threads,  /proc/sys/kernel/threads-
              max, was reached.

       EINVAL Invalid settings in attr.

       EPERM  No permission to set the scheduling policy and parameters specified in attr.
```

special constants for *return value*

same pattern for many other pthreads functions

will often omit error checking in slides for brevity

error checking pthread_create

```
int error = pthread_create(...);
if (error != 0) {
    /* print some error message */
}
```

backup slides

backup sides

4.4BSD scheduler

4.4BSD / FreeBSD pre-2003 scheduler was a variation on MLFQ

64 priority levels, 100 ms quantum

same quantum at every priority

priorities adjusted periodically

- in retrospect not good for performance — iterate through all threads
- part of why FreeBSD stopped using this scheduler

priority of threads that spent a lot of time waiting for I/O increased

priority of threads that used a lot of CPU time decreased

Linux's Completely Fair Scheduler (CFS)

Linux's default scheduler is a proportional share scheduler…

…without randomization (consistent)

…with $O(\log N)$ scheduling decision

(handles many threads/processes)

…which favors interactive programs

…which adjusts timeslices dynamically

shorter timeslices if many things to run

Linux's Completely Fair Scheduler (CFS)

Linux's default scheduler is a proportional share scheduler…

…without randomization (consistent)

…with $O(\log N)$ scheduling decision

(handles many threads/processes)

…which favors interactive programs

…which adjusts timeslices dynamically

shorter timeslices if many things to run

CFS: tracking runtime

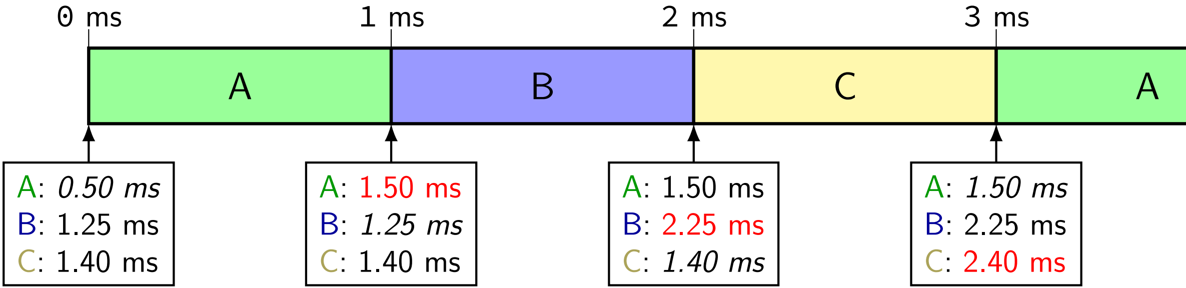
each thread has a *virtual runtime* ($\sim$ how long it's run)

incremented when run based how long it runs

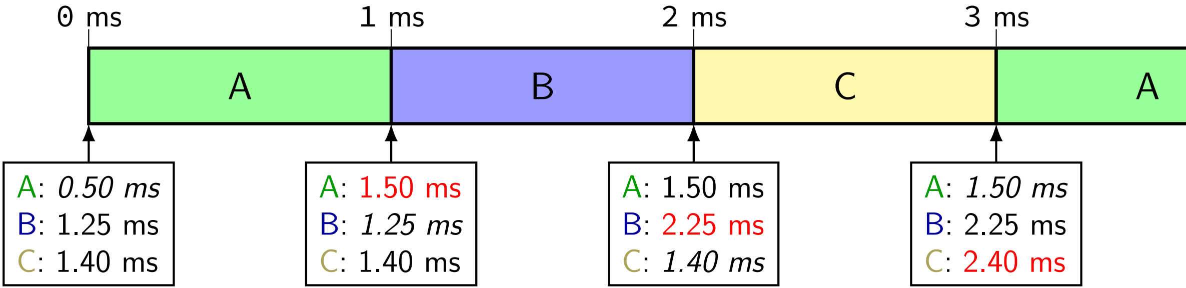
scheduling decision: run thread with lowest virtual runtime

- data structure: balanced tree

virtual time, always ready, 1 ms quantum



virtual time, always ready, 1 ms quantum

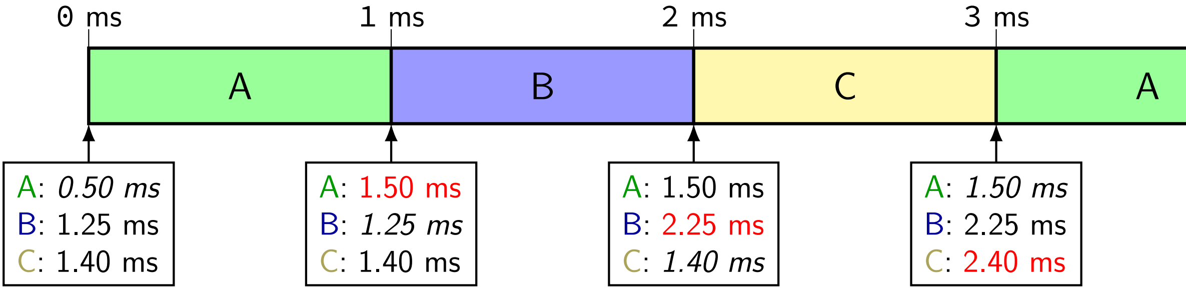


at each time:

update current thread's time

run thread with lowest total time

virtual time, always ready, 1 ms quantum



at each time:

update current thread's time

run thread with lowest total time

same effect as round robin
if everyone uses whole quantum

MLFQ variations

version of MLFQ I described is in Anderson-Dahlin

problems:

starvation

- worse than with real SRTF — based on *guess*, not real remaining time

oscillation not great for predictability

variation to prevent starvation

Apraci-Dusseau presents variant of MLFQ w/o starvation

two changes:

don't increase priority when whole quantum not used

instead keep the same — more stable

periodically increase priority of *all threads*

allow compute-heavy threads to run a little

still deals with thread's behavior changing over time

replaces finer-grained upward adjustments

FreeBSD scheduler

current default FreeBSD scheduler based on MLFQ idea

...but: time quantums don't depend on priority

computes *interactivity score* $\sim \dfrac{\text{I/O wait}}{\text{I/O wait} + \text{runtime}}$

note: deliberately not estimating remaining time

(using "recent" history of thread)

thread priorities set based on interactivity score

real-time

so far: “best effort” scheduling

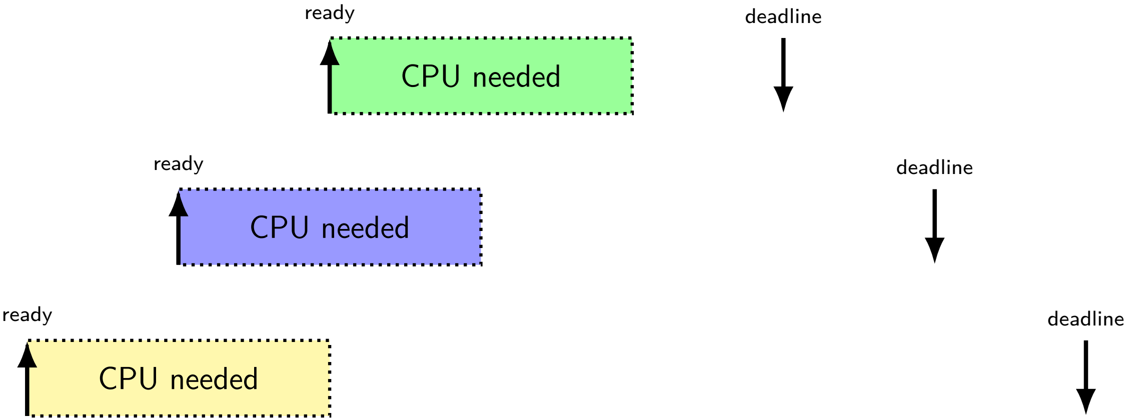
best possible (by some metrics) given some work

alternate model: need gaurnetees

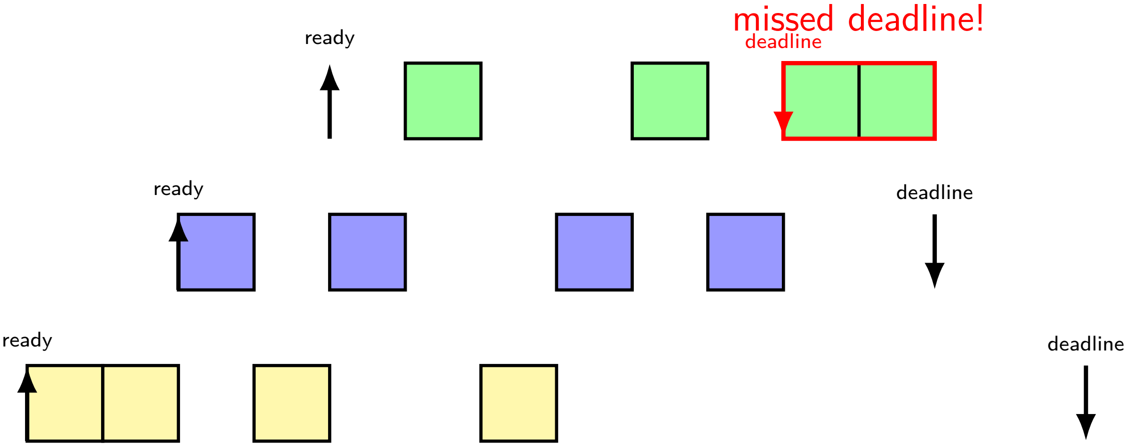
deadlines imposed by real-world

process audio with 1ms delay
computer-controlled cutting machines (stop motor at right time)
car brake+engine control computer
...

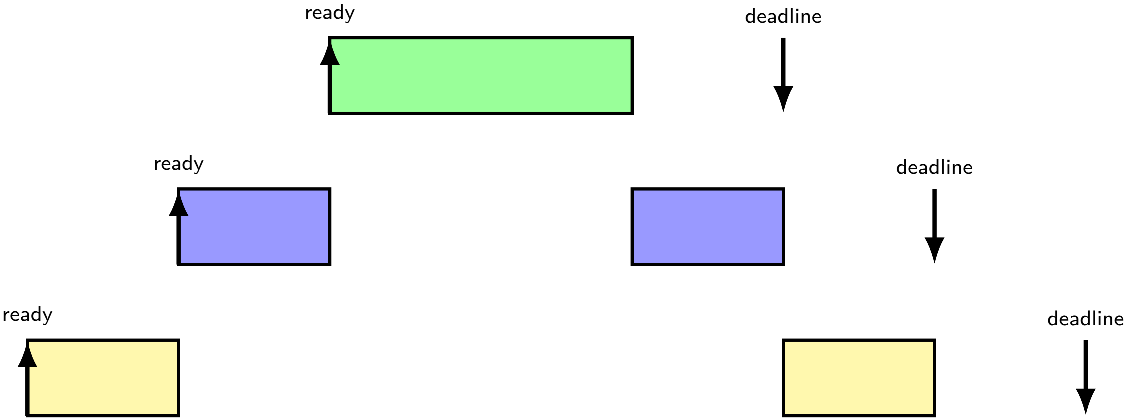
real time example: CPU + deadlines



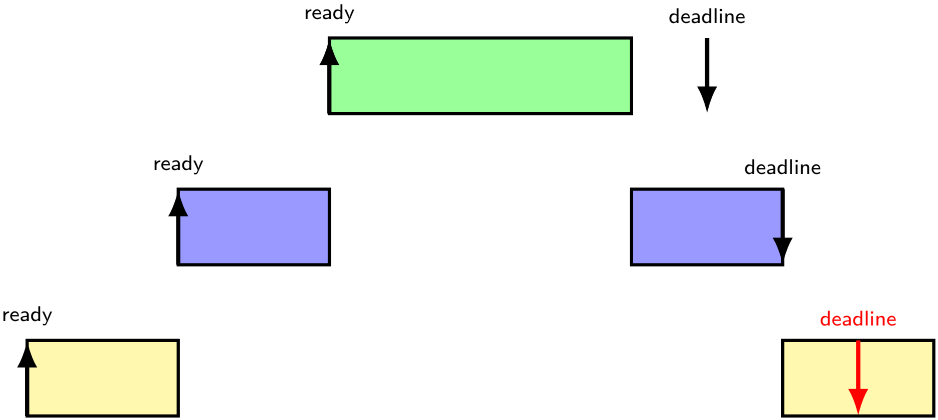
example with RR



earliest deadline first



impossible deadlines



no way to meet all deadlines!

admission control

given *worst-case* runtimes, start times, deadlines, scheduling algorithm,...

figure out whether it's possible to gaurentee meeting deadlines

- details on how — not this course (probably)

if not, then

- change something so they can?
- don't ship that device?
- tell someone at least?

earliest deadline first and...

earliest deadline first does *not* (even when deadlines met)

- minimize response time
- maximize throughput
- maximize fairness

exercise: give an example

other real-time schedulers

typical real time systems: *periodic tasks with deadlines*

“*rate monotonic*”

commonly approximate EDF with lower period = higher priority

easier to implement than true EDF

well-known method to determine if schedule is admissible

= won’t exceed deadline (under some assumptions)

aside: measuring fairness (1)

first question: what needs to be divided fairly?

problem: what about programs waiting for I/O?

answer 1:
don't consider what happens when program waiting for I/O

answer 2:
give program credit for time not running while waiting for I/O

aside: measuring fairness (1)

first question: what needs to be divided fairly?

problem: what about programs waiting for I/O?

answer 1:
don't consider what happens when program waiting for I/O

answer 2:
give program credit for time not running while waiting for I/O

aside: measuring fairness (1)

first question: what needs to be divided fairly?

problem: what about programs waiting for I/O?

answer 1:
don't consider what happens when program waiting for I/O

answer 2:
give program credit for time not running while waiting for I/O

aside: measuring fairness (2)

one way: max-min fairness

choose schedule that maximizes the minimum resource given to anyone

aside: measuring fairness (2)

one way: max-min fairness

choose schedule that maximizes the minimum resource given to anyone