

Scheduling & Threads

CIS 4480/5480 Fall 2025

Overview

Threads

Background

pthread

Concurrency

Scheduling

Background

FCFS

Algorithms

Processes vs. Threads

- Threads are sequential execution streams within a process
- Processes have:
 - Address space
 - OS resources (file descriptors)
- Threads have unique:
 - Stack
 - Stack Pointer
 - Program Counter
 - Registers
- Threads share:
 - Address spaces
 - Heap
 - Globals



Why Threads?

- Summary: lower overhead
- Smaller memory footprint than subprocesses
- Faster to create and destroy
- Can share memory with other threads
- Faster context switching
- Simpler synchronization primitives available



Why Processes?

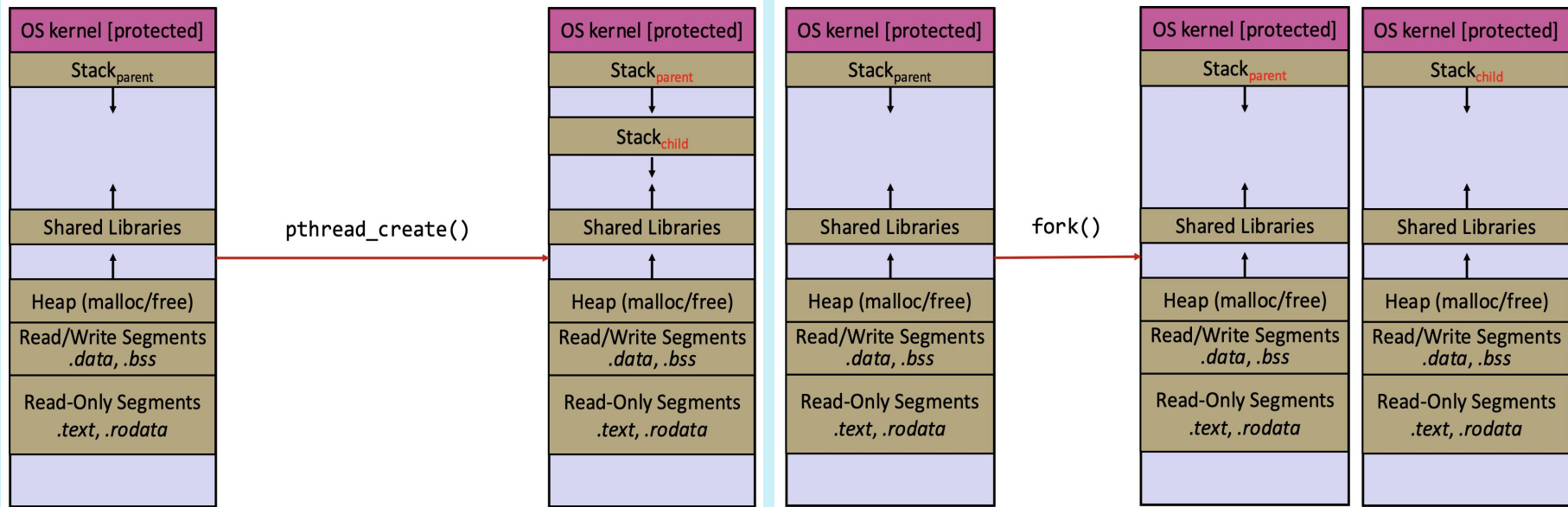
- Portability (Language Support): Just call `fork()`
- Lack of shared memory means it's harder to run into concurrent modification issues
- One process crashing doesn't bring the others down so easily

```
odulo@ubuntu32: ~  
connection to org.freedesktop.compiz failed --> skipping compiz related tasks  
11.3%] Tasks: 132, 193 thr: 1 running  
12.6%] Load average: 0.08 0.08 0.12  
or.ServiceUnknown: The name org.freedesktop.compiz was not provided by any .serv  
0/1021MB]
```

	PRI	NI	VRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
	20	0	3648	2000	1284	S	0.0	0.1	0:01.05	/sbin/init
	20	0	10292	5980	576	S	0.0	0.3	0:07.58	/bin/bash /usr/local/sbin
	20	0	4184	280	228	S	0.0	0.0	0:00.00	└─ sleep 30
	30	10	3172	1212	336	S	0.0	0.1	0:00.33	/usr/local/maldetect/inot
lo	20	0	23988	14144	4972	S	0.0	0.7	0:00.26	/usr/bin/python /usr/lib
lo	20	0	93472	16660	11108	S	0.0	0.8	0:07.58	gnome-terminal
lo	20	0	93472	16660	11108	S	0.0	0.8	0:00.00	└─ gnome-terminal
lo	20	0	7440	3868	1620	S	0.0	0.2	0:00.16	└─ bash
lo	20	0	2384	744	616	S	0.0	0.0	0:00.00	└─ gnome-pty-helper
lo	20	0	93472	16660	11108	S	0.0	0.8	0:00.01	└─ gnome-terminal
lo	20	0	93472	16660	11108	S	0.0	0.8	0:00.00	└─ gnome-terminal
lo	20	0	7440	3860	1616	S	0.0	0.2	0:00.16	└─ bash
lo	20	0	5524	1992	1316	R	2.0	0.1	0:24.57	└─ htop
lo	20	0	172M	52492	20248	S	3.0	2.5	0:03.14	└─ /usr/bin/perl /usr
lo	20	0	172M	52492	20248	S	0.0	2.5	0:00.00	└─ /usr/bin/perl /usr
lo	20	0	172M	52492	20248	S	0.0	2.5	0:00.05	└─ /usr/bin/perl /usr
lo	20	0	172M	52492	20248	S	0.0	2.5	0:00.00	└─ /usr/bin/perl /usr
lo	20	0	91340	9192	7124	S	0.0	0.4	0:00.01	└─ /usr/lib/gnome-online-ac
lo	20	0	91340	9192	7124	S	0.0	0.4	0:00.00	└─ /usr/lib/gnome-online
lo	20	0	43200	6440	5408	S	0.0	0.3	0:00.04	└─ /usr/lib/telepathy/missi

```
etup F3Search F4Filter F5Tree F6SortBy F7Nice F8Nice F9kill F10Quit
```

pthread_create() vs fork()



Pthreads

Overview

- POSIX API for dealing with threads
- `#include <pthread.h>`

pthread_create()

- Creates a new thread and runs a given start function
- Returns 0 on success, error num on error

pthread_join()

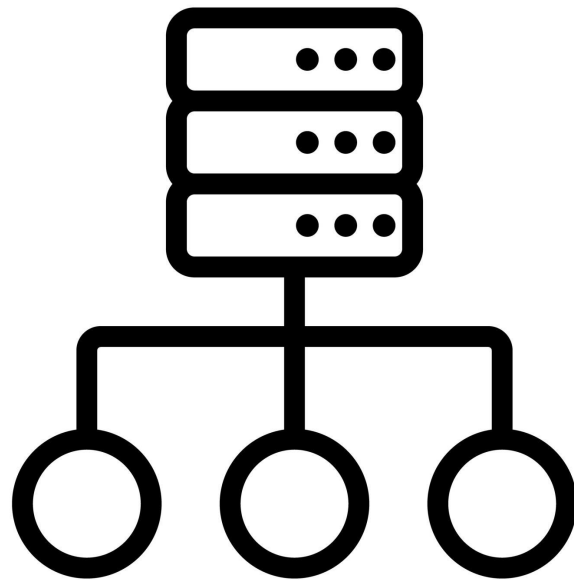
- Calling thread waits for a specified thread to terminate
- Joins execution streams together

```
int pthread_create(  
    pthread_t* thread,  
    const pthread_attr_t* attr,  
    void* (*start)(void*),  
    void* arg);
```

```
int pthread_join(pthread_t thread, void** retval);
```

Concurrency Intro

- Doing multiple tasks in the same time window
- Different from parallelism (see lecture)
 - Parallelism can help us achieve concurrency well
- Could require working together
 - May need synchronization (upcoming)
 - Mutexes, semaphores, etc.



Scheduling

- In operating systems, scheduling assigns tasks to CPUs
- OS-specific policies/algorithms decide which tasks to schedule next
 - Each have different strengths/weaknesses
- Linux has used a few schedulers
 - Completely Fair Scheduler
 - Earliest Eligible Virtual Deadline First (latest)



Scheduling (cont.)

- We consider tasks to be threads
 - To Linux, threads are just a special process
- We care about
 - Fairness
 - Wait Time
 - Latency
 - Overhead
 - Throughput
 - In real OSes, priorities



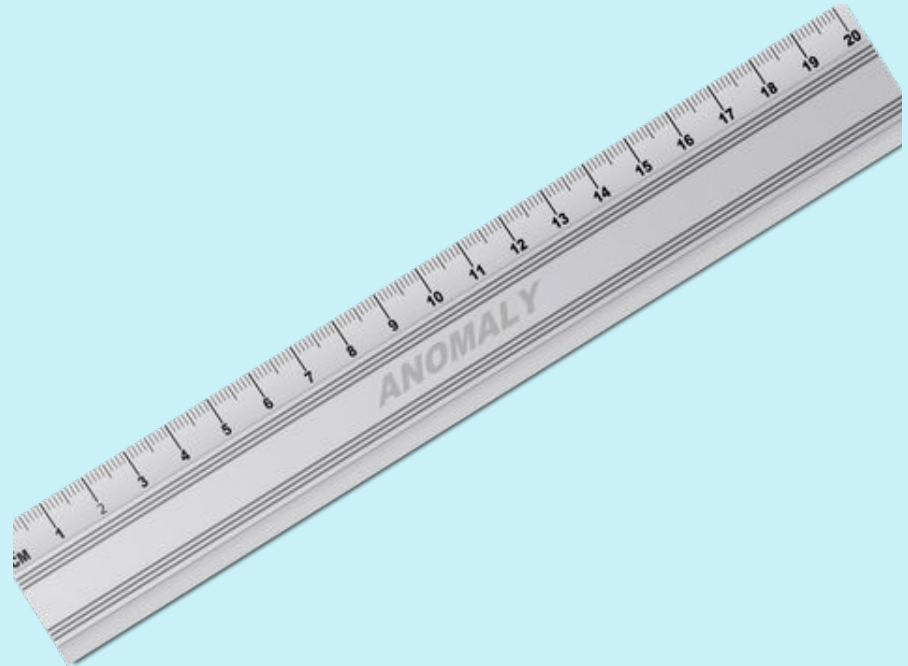
First Come First Serve (FCFS)

- Schedule whichever job is ready first, and run it to completion/blocking
- Queue for ready tasks
 - Add task when created or unblocked
- Drawbacks:
 - Not so interactive
 - May be very unfair
 - Long CPU-bound task
 - No priority



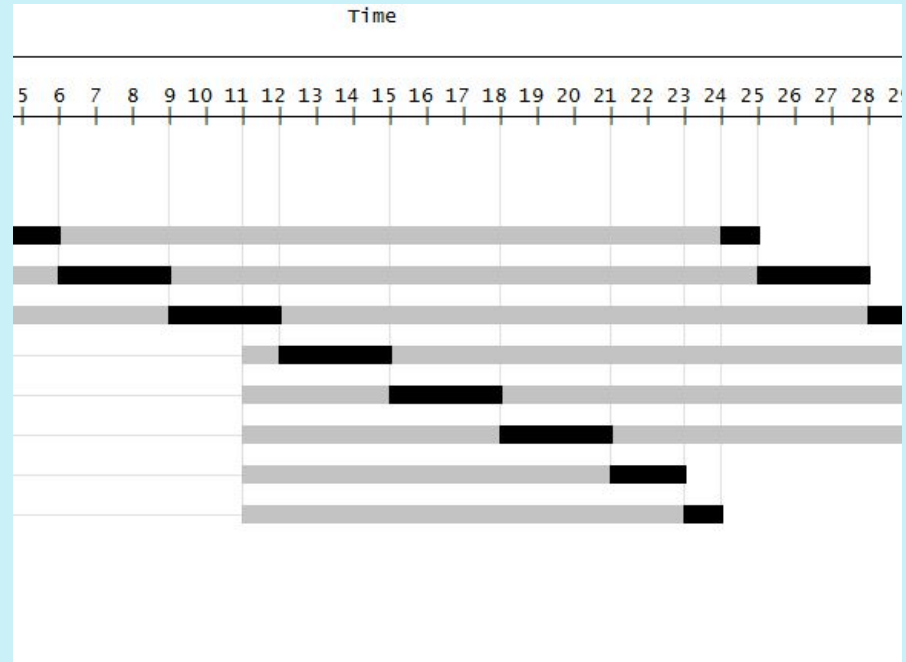
Shortest Job First (SJF)

- The next job is the shortest one in our current pool
- Queue for ready tasks
- Disadvantages:
 - Hard to estimate true job length
 - We may starve longer tasks



Round Robin (RR)

- Each job gets to run for at most a time quantum, then the next job in the ready queue is scheduled
- Preemptive: will stop a running process
- Disadvantages:
 - May be unfair to tasks that yield the CPU
 - No priorities
 - Context switching



Priority Round Robin (PRR)

- Round robin but tasks have priority now
- Runs lower priority tasks if the higher priority queues are empty
- In PennOS, you're asked to implement a PRR variant with 3 priority levels
 - We can schedule lower priority jobs even if there are higher priority jobs
 - We wait until the time quantum is complete before checking the thread status again



Multi-Level Feedback (MLF)

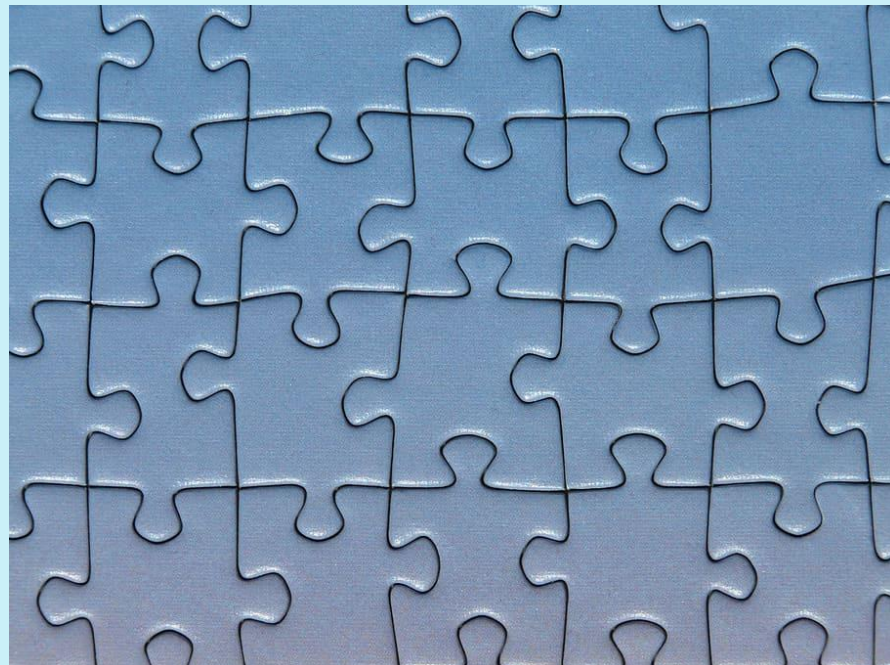
- Priority levels each have queues and associated time quanta
- Threads enter at highest priority level
- If a thread voluntarily gives up CPU, it gets moved to the end of its queue
- If it's pre-empted by the scheduler (forced to give up CPU), it's moved to lower-level queue
- Higher queues must be empty for any given queue to get scheduled



CFS

Completely Fair Scheduler

- Another approach to fairness
- Track cumulative CPU usage (vruntime) for each task and try to even it out
- Priorities can be handled by weighting runtime adjustments differently
- Minimum time quantum is set to avoid excessive context switching
- Manage state with Red-black tree
- New tasks start with the minimum vruntime in the system



Nice

- Feature of Linux systems to adjust process importance
- Used by the scheduler in priority calculations
- The “nicer” a process is, the higher it’s nice value, and the lower its priority
- Can be adjusted by users with the “nice” command

QEMU									
ranges: 0.00, 0.00, 0.00; up 0+00:13:17 2									
sses: 16 sleeping, 1 on CPU									
es: 0.0% user, 0.0% nice, 0.0% system, 0.0% interrupt, 100%									
29M Act, 6028K Wired, 8336K Exec, 14M File, 184M Free									
ERNAME	PRI	NICE	SIZE	RES	STATE	TIME	WCPU	CPU	COMMA
ot	43	0	17M	1800K	CPU	0:00	0.00%	0.00%	top
ot	96	0	0K	2912K	atath	0:00	0.00%	0.00%	lsyst
ot	85	0	22M	6128K	pause	0:00	0.00%	0.00%	ntpd
ot	85	0	48M	3912K	kqueue	0:00	0.00%	0.00%	maste
stfix	85	0	48M	3856K	kqueue	0:00	0.00%	0.00%	qmgr
stfix	85	0	48M	3820K	kqueue	0:00	0.00%	0.00%	picku
ot	85	0	59M	3736K	wait	0:00	0.00%	0.00%	login
ot	85	0	23M	1876K	kqueue	0:00	0.00%	0.00%	syslo
ot	85	0	13M	1740K	wait	0:00	0.00%	0.00%	sh
ot	85	0	13M	1352K	tttyraw	0:00	0.00%	0.00%	getty
ot	85	0	13M	1352K	tttyraw	0:00	0.00%	0.00%	getty
ot	85	0	13M	1352K	tttyraw	0:00	0.00%	0.00%	getty
ot	85	0	11M	1320K	nanoslp	0:00	0.00%	0.00%	cron
ot	85	0	13M	1316K	wait	0:00	0.00%	0.00%	init
ot	85	0	15M	1104K	kqueue	0:00	0.00%	0.00%	inetd
ot	85	0	11M	1072K	select	0:00	0.00%	0.00%	dhcli
ot	85	0	13M	1020K	kqueue	0:00	0.00%	0.00%	power

EEVDF (Not Tested)

Earliest Eligible Virtual Deadline First

- Current scheduler for Linux
- Similar goal to CFS, but aims to also consider latency
- Tasks with shorter time slices are prioritized in a virtual deadline calculation
 - Often latency-sensitive tasks
- Task with earliest virtual deadline is selected



Recitation Check-In

Must submit by **7:00pm** on Gradescope for credit.

Thread Execution

```
#define NUM_PROCESSES N
#define LOOP_NUM M

int sum_total = 0;

void loop_incr() {
    for (int i = 0; i < LOOP_NUM; i++) {
        sum_total++;
    }
    printf("Process ID: %d with sum total of %d.\n", getpid(), sum_total);
}

int main(int argc, char** argv) {

    pid_t pids[NUM_PROCESSES]; // array of process ids

    // create processes to run loop_incr()
    for (int i = 0; i < NUM_PROCESSES; i++) {
        pids[i] = fork();
        if (pids[i] == 0) { // child
            loop_incr();
            exit(EXIT_SUCCESS);
        }
    }

    // wait for all child processes to finish
    for (int i = 0; i < NUM_PROCESSES; i++) {
        waitpid(pids[i], NULL, 0);
    }

    printf("The ultimate sum total is %d\n", sum_total);
    return EXIT_SUCCESS;
}
```

What is the maximum value of `sum_total` after `main` executes?

Algorithm Selection

You're developing a Linux distribution with a graphical user interface packaged in it and recurring long-running background jobs on a single-core CPU. Rank the following scheduling algorithms from most to least supportive of the interactivity requirements of the system.

1. FCFS
2. RR
3. CFS



Scheduling

Assume Job A (length 3) arrives at time 0, Job B (length 2) arrives after A but also at time 0 (before scheduling), and Job C (length 2) arrives some time strictly between time 1 and 2.

Task lengths are multiples of 1 time quantum, and 1 unit of time is a time quantum. Assume switching has no overhead. For round robin, assume a process is immediately added to the end of the queue once it completes its time quantum.

Which task is scheduled at time 5 under each of the following policies:

1. First Come First Serve
2. Shortest Job First
3. Classic Round Robin



Let's Have A Great Midterm!

