TCSS 422: OPERATING SYSTEMS

Proportional Share Scheduling, Linux Completely Fair Scheduler, Introduction to Concurrency

Wes J. Lloyd
School of Engineering and Technology
University of Washington - Tacoma

OBJECTIVES

- Assignment 0 / Linux Tutorial
- C Tutorial
- Assignment 1
- Feedback 1/28
- CPU Scheduling:
- Chapter 9 – Proportional Share Scheduler
  Linux Completely Fair Scheduler (CFS)
- Parallel programming with P-threads:
- Chapter 26 – Intro to concurrency
- Chapter 27 – Linux Thread API
- Chapter 28 – Intro to locks

FEEDBACK FROM 1/28

- **What are tickets in proportional share schedulers? (e.g. lottery)**
- Goal: model CPU job scheduling as a ticket system
- Jobs with more tickets have higher priority to run
- They can obtain a greater share of the CPU time
- Can think of a ROUND-ROBIN scheduler as a lottery scheduler where everyone has same number of tickets
  - Time proportions are all equal
- Lottery scheduler rotates among jobs in a run queue similar to RR, but jobs have different runtime proportions based on their share of tickets

FEEDBACK - 2

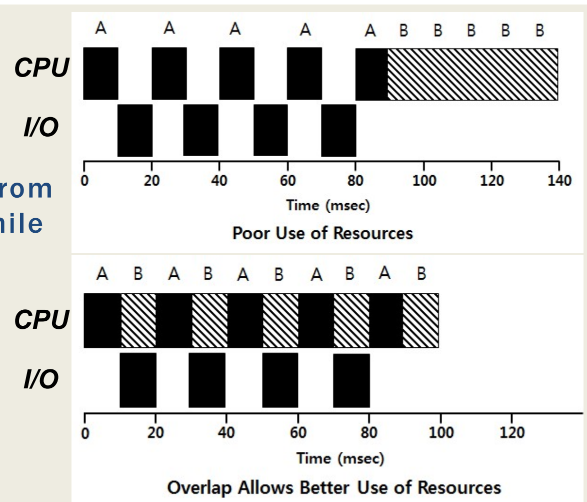
- **Does the lottery scheduler cause overhead?**
- Rate overhead of lottery scheduler tasks: ***HIGH, MEDIUM, LOW***
- Consider a lottery scheduler with 1 user, having 100 jobs
- User has 1000 tickets, system has 10,000:
- **Task 1.** A context switch occurs and the scheduler chooses a job to run
- **Task 2.** Perform currency conversion between user tickets and system tickets
- **Task 3.** A new job arrives in the scheduler. User redistributed tickets to give new job 100 tickets.

FEEDBACK - 3

- **Could the lottery scheduler evenly distribute tickets based on a priority metric to reduce starvation?**
- *Yes, assuming job priority information is available*
- A new job arrives in the system, how do we assign priority?
- **Do we ask the user?**
- MLFQ approach- place in the high priority queue, observe behavior, and slowly adjust priority
- **How should the OS assign tickets upon job arrival?**
- **What do we know about incoming jobs a priori ?**
- ***Runtime? Behavior - I/O bound? Batch? Priority?***
- *Ticket assignment is an open problem…*
  (no optimal one size fits all approach)

FEEDBACK - 4

- **Incorporating I/O, how does overlap work?**
- Within a single CPU core during I/O Job A moves from RUNNING → BLOCKED while I/O is performed
- During these IDLE CPU times,Job B moves from READY→ RUNNING



CHAPTER 9 - PROPORTIONAL SHARE SCHEDULER

PROPORTIONAL SHARE SCHEDULER

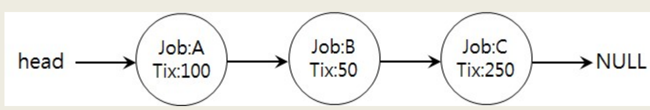
- Also called fair-share scheduler
  or lottery scheduler
  - Guarantees each job receives some percentage of CPU time based on share of "tickets"
  - Each job receives an allotment of tickets
  - % of tickets corresponds to potential share of a resource
  - Can conceptually schedule any resource this way
    - CPU, disk I/O, memory

LOTTERY SCHEDULER

- Simple implementation
  - Just need a random number generator
    - Picks the winning ticket
  - Maintain a data structure of jobs and tickets (list)
  - Traverse list to find the owner of the ticket
  - Consider sorting the list for speed

January 30, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L7.9

LOTTERY SCHEDULER IMPLEMENTATION




```c
// counter: used to track if we’ve found the winner yet
int counter = 0;

// winner: use some call to a random number generator to
// get a value, between 0 and the total # of tickets
int winner = getrandom(0, totaltickets);

// current: use this to walk through the list of jobs
node_t *current = head;

// loop until the sum of ticket values is > the winner
while (current) {
        counter = counter + current->tickets;
        if (counter > winner)
                break; // found the winner
        current = current->next;
}
// ’current’ is the winner: schedule it...
```

January 30, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L7.10

TICKET MECHANISMS

- Ticket currency / exchange
  - User allocates tickets in any desired way
  - OS converts user currency into global currency
- Example:
  - There are 200 global tickets assigned by the OS

User A → *500* (A's currency) to A1 → *50* (global currency)
→ *500* (A's currency) to A2 → *50* (global currency)

User B → *10* (B's currency) to B1 → *100* (global currency)

TICKET MECHANISMS - 2

- Ticket transfer
  - Temporarily hand off tickets to another process
- Ticket inflation
  - Process can temporarily raise or lower the number of tickets it owns
  - If a process needs more CPU time, it can boost tickets.

LOTTERY SCHEDULING

- Scheduler picks a winning ticket
  - Load the job with the winning ticket and run it

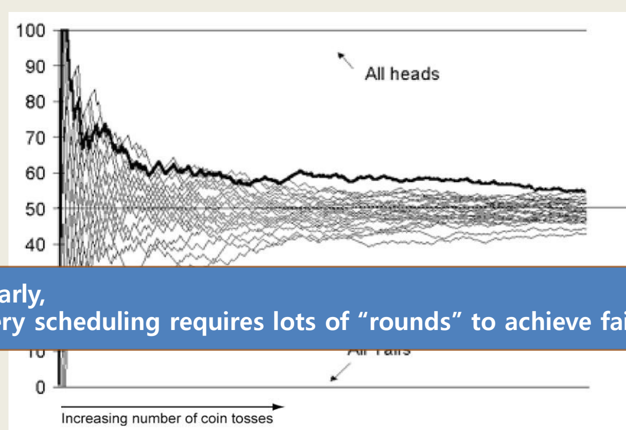
- Example:
  - Given 100 tickets in the pool
    - Job A has 75 tickets: 0 - 74
    - Job B has 25 tickets: 75 – 99

Scheduler's winning tickets:	63	85	70	39	76	17	29	41	36	39	10	99	68	83	63
Scheduled job:	A	B	A	A	B	A	A	A	A	A	A	B	A	B	A

- But what do we know about probability of a coin flip?

COIN FLIPPING

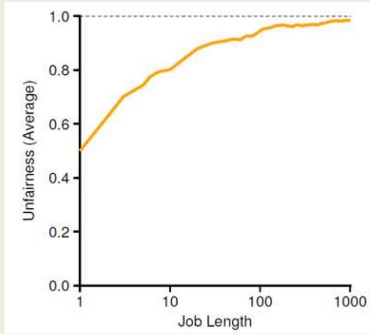
- Equality of distribution (fairness) requires a lot of flips!



Similarly,
Lottery scheduling requires lots of "rounds" to achieve fairness.

LOTTERY FAIRNESS

- With two jobs
  - Each with the same number of tickets (t=100)



When the job length is not very long, average unfairness can be **quite severe**.

LOTTERY SCHEDULING CHALLENGES

- What is the best approach to assign tickets to jobs?
  - Typical approach is to assume users know best
  - Users are provided with tickets, which they allocate as desired
    - System performs currency conversion
- How should the OS automatically distribute tickets upon job arrival?
  - What do we know about incoming jobs a priori ?
    - Runtime? Behavior - I/O bound? Batch? Priority?
- *Ticket assignment is really an open problem…*

STRIDE SCHEDULER

- Addresses statistical probability issues with lottery scheduling
- Instead of guessing a random number to select a job, simply count…

STRIDE SCHEDULER - 2

- Jobs have a "stride" value
  - A stride value describes the counter pace when the job should give up the CPU
  - Stride value is <u>inverse in proportion</u> to the job's number of tickets (more tickets = smaller stride)
- Total system tickets = 10,000
  - Job A has 100 tickets → A_{stride} = 10000/100 = 100 stride
  - Job B has 50 tickets → B_{stride} = 10000/50 = 200 stride
  - Job C has 250 tickets → C_{stride} = 10000/250 = 40 stride
- Stride scheduler tracks "pass" values for each job (A, B, C)

STRIDE SCHEDULER - 3

- Basic algorithm:
  1. Stride scheduler picks job with the lowest pass value
  2. Scheduler increments job's pass value by its stride and starts running
  3. Stride scheduler increments a counter
  4. When counter exceeds pass value of current job, pick a new job (go to 1)

- KEY: When the counter reaches a job's "PASS" value, the scheduler passes on to the next job...

STRIDE SCHEDULER - EXAMPLE

- Stride values
  - Tickets = priority to select job
  - Stride is inverse to tickets
  - Lower stride = more chances to run (higher priority)

Priority

C stride = 40

A stride = 100

B stride = 200

STRIDE SCHEDULER EXAMPLE - 2

- Three-way tie: randomly pick job A (all pass values=0)
- Set A's pass value to A's stride = 100
- Increment counter until > 100
- Pick a new job: two-way tie

Tickets
C = 250
A = 100
B = 50

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	…

Initial job selection is random. All @ 0

C has the most tickets and receives a lot of opportunities to run…

STRIDE SCHEDULER EXAMPLE - 3

- We set A's counter (pass value) to A's stride = 100
- Next scheduling decision between B (pass=0) and C (pass=0)
  - Randomly choose B
- C has the lowest counter for next 3 rounds

Tickets
C = 250
A = 100
B = 50

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	…

C has the most tickets and is selected to run more often …

STRIDE SCHEDULER EXAMPLE - 4

- Job counters support determining which job to run next
- Over time jobs are scheduled to run based on their priority represented as their share of tickets…
- Tickets are analogous to job priority

Tickets
C = 250
A = 100
B = 50

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	…

LINUX: COMPLETELY FAIR SCHEDULER (CFS)

- Linux ≥ 2.6.23: Completely Fair Scheduler (CFS)
- Linux < 2.6.23: O(1), O(n) schedulers
- Every thread/process has a scheduling class (policy):
- Normal classes: SCHED_OTHER (TS), SCHED_IDLE, SCHED_BATCH
  - TS = Time Sharing
- Real-time classes: SCHED_FIFO (FF), SCHED_RR (RR)
- Show scheduling class and priority:
- `ps -elfc`
- `ps ax -o pid,ni,cls,pri,cmd`

COMPLETELY FAIR SCHEDULER - 2

- Loosely based on the stride scheduler
- CFS models system as a Perfect Multi-Tasking System
  - In perfect system every process of the same priority (class) receive exactly 1/n^{th} of the CPU time
- Can compare with <u>ideal fair scheduling</u>
  - Divide processor equally among processes

Each process receives (100/n)% CPU time

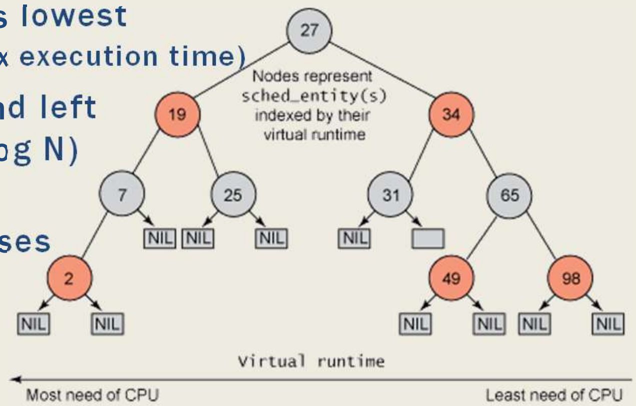
Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

COMPLETELY FAIR SCHEDULER - 3

- Scheduling classes each have a runqueue
  - Groups process of same priority
  - Process priority groups use different sets of runqueues for priorities
  - ***<u>Scheduler chooses job with lowest accumulative runtime to run</u>***
  - Time quantum varies based on how many jobs in shared runqueue
    - Time quantum is proportional to system CPU load in the runqueue
    - No fixed time quantum (e.g. 10 ms)

COMPLETELY FAIR SCHEDULER – 4

- Runqueues are stored using a linux red-black tree
  - Self balancing binary tree - nodes indexed by **`vruntime`**
- Leftmost node has lowest **`vruntime`** (approx execution time)
- Walking tree to find left most node is ~O(log N) for N nodes
- Completed processes removed



January 30, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L7.27

COMPLETELY FAIR SCHEDULER - 5

- CFS tracks virtual run time in vruntime variable
- The task on a given runqueue with the lowest **`vruntime`** is scheduled next
- **`struct sched_entity`** contains **`vruntime`** parameter
  - Describes process execution time in nanoseconds
  - Value is not pure runtime, but weighted based on priority
  - Perfect scheduler → achieve equal **`vruntime`** for all processes of same priority
- Key takeaway:
  identifying the next job to schedule is really fast!

January 30, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L7.28

CFS: JOB PRIORITY

- Time slice: Linux ***"Nice value"***
  - Nice value predates the CFS scheduler
  - Top shows nice values
  - Process command (nice & priority):
    `ps ax -o pid,ni,cmd,%cpu, pri`
- Nice Values: from -20 to 19
  - Lower is ***higher*** priority, default is 0
  - Vruntime is a weighted time measurement
  - Priority weights the calculation of vruntime within a runqueue to give high priority jobs a boost.
    - Influences job's position in rb-tree

CFS: TIME QUANTUM

- Scheduling quantum is calculated at runtime based on targeted latency and total number of running processes
- Will vary between:
- `cat /proc/sys/kernel/sched_min_granularity_ns`
  (3 ms – minimum quantum)
- `cat /proc/sys/kernel/sched_latency_ns`
  (24 ms – target quantum)
- Target quantum (latency):
  - Interval during which task should run at least once
  - Automatically increases as number of jobs increase

CFS: TIME QUANTUM - 2

- How do we map a nice value to an actual CPU time quantum (timeslice) (ms)? What is the best mapping?
- O(1) scheduler (< 2.6.23)
  - tried to map nice value to timeslice (fixed allotment)
- Linux completely fair scheduler
  - Nice value suggests priority to assign runqueue for job
  - Time proportion varies based on # of jobs in runqueue
  - With fewer jobs in runqueue, time proportion is larger

COMPLETELY FAIR SCHEDULER REFERENCES

- More information:
- Man page: "man sched" : Describes Linux scheduling API
- http://manpages.ubuntu.com/manpages/bionic/man7/sched.7.html
- https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt
- https://en.wikipedia.org/wiki/Completely_Fair_Scheduler
- See paper: The Linux Scheduler – a Decade of Wasted Cores
- http://www.ece.ubc.ca/~sasha/papers/eurosys16-final29.pdf

CHAPTER 26 - CONCURRENCY: AN INTRODUCTION

January 30, 2019 TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma L7.33

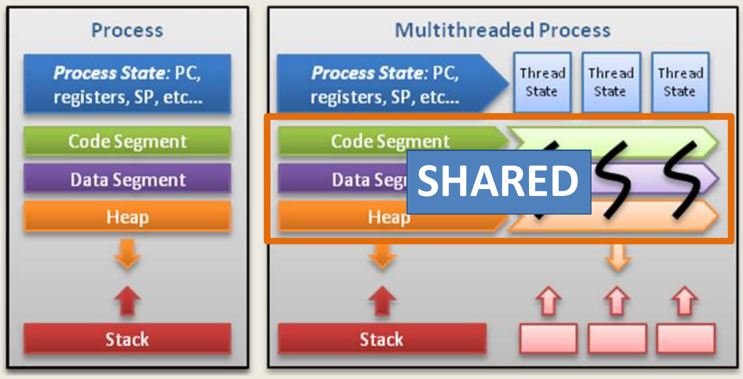
OBJECTIVES

- Introduction to threads
- Race condition
- Critical section
- Thread API

January 30, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L7.34

THREADS

Single Threaded Process



Multiple Threaded Process

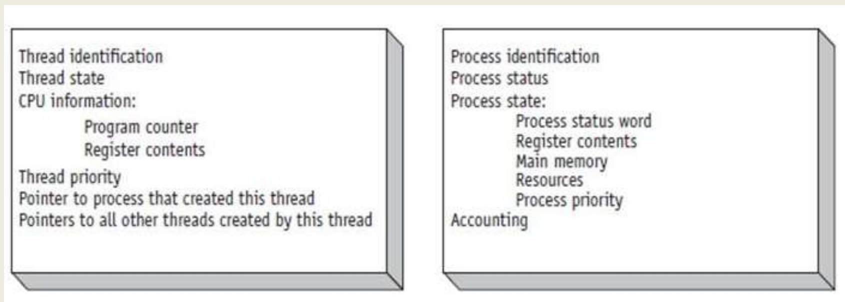
©Alfred Park, http://randu.org/tutorials/threads

THREADS - 2

- Enables a single process (program) to have multiple "workers"
  - This is parallel programming...
- Supports independent path(s) of execution within a program *with shared memory ...*
- Each thread has its own Thread Control Block (TCB)
  - PC, registers, SP, and stack
- Threads share code segment, memory, and heap are shared
- ***What is an embarrassingly parallel program?***

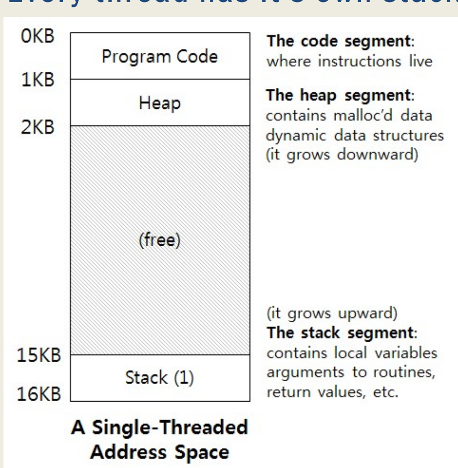
PROCESS AND THREAD METADATA

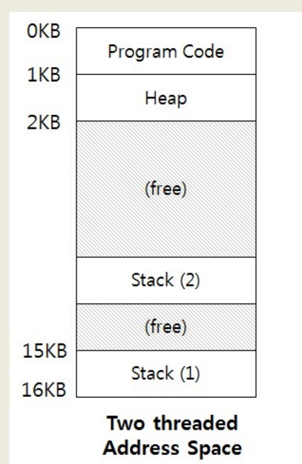
- Thread Control Block vs. Process Control Block



SHARED ADDRESS SPACE

- Every thread has it's own stack / PC





THREAD CREATION EXAMPLE

```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```


POSSIBLE ORDERINGS OF EVENTS

int main()	Thread 1	Thread 2
Starts running		
Prints 'main: begin'		
Creates Thread 1		
Creates Thread 2		
Waits for T1		
	Runs	
	Prints 'A'	
	Returns	
Waits for T2		
		Runs
		Prints 'B'
		Returns
Prints 'main: end'		

POSSIBLE ORDERINGS OF EVENTS - 2

int main()	Thread 1	Thread 2
Starts running		
Prints 'main: begin'		
Creates Thread 1		
	Runs	
	Prints 'A'	
	Returns	
Creates Thread 2		
		Runs
		Prints 'B'
		Returns
Waits for T1	*Returns immediately*	
Waits for T2		*Returns immediately*
Prints 'main: end'		

January 30, 2019 | TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L7.41

POSSIBLE ORDERINGS OF EVENTS - 3

int main()	Thread 1	Thread 2
Starts running		
Prints 'main: begin'		
Creates Thread 1		
Creates Thread 2		
Waits for T		
	Runs	
	Prints 'A'	
	Returns	
Waits for T2		*Immediately returns*
Prints 'main: end'		

What if execution order of events in the program matters?

January 30, 2019 | TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L7.42

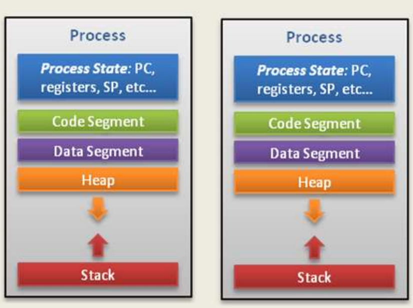
COUNTER EXAMPLE

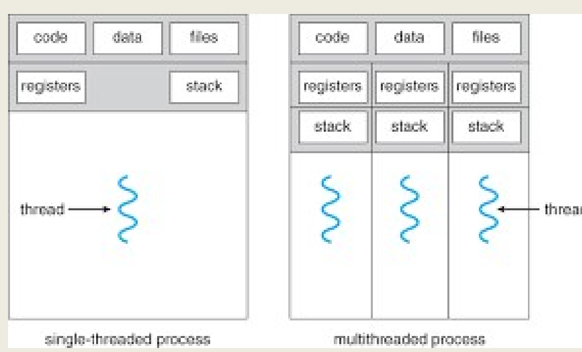
- Counter example
- A + B : ordering
- Counter: incrementing global variable by two threads
- ***Is the counter example embarrassingly parallel?***
- ***What does the parallel counter program require?***

January 30, 2019 | TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L7.43

PROCESSES VS. THREADS

- What's the difference between forks and threads?
  - Forks: duplicate a process
  - Think of ***CLONING*** **-** There will be two identical processes at the end
  - Threads: no duplicate of code/heap, lightweight execution threads





January 30, 2019 | TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L7.44

RACE CONDITION

- What is happening with our counter?
  - When counter=50, consider code: counter = counter + 1
  - If synchronized, counter will = 52

OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	50
	mov %eax, 0x8049a1c		113	51	**51**

CRITICAL SECTION

- Code that accesses a shared variable must not be ***concurrently*** executed by more than one thread
- Multiple *active* threads inside a ***critical section*** produce a ***race condition***.
- ***Atomic execution*** (*all code executed as a unit*) must be ensured in ***critical*** sections
  - These sections must be ***mutually exclusive***

LOCKS

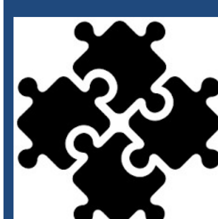
- To demonstrate how critical section(s) can be executed "atomically-*as a unit*" Chapter 27 & beyond introduce locks

```
lock_t mutex;
. . .
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Critical section

- Counter example revisited

CHAPTER 27 - LINUX THREAD API



THREAD CREATION

- pthread_create

```
#include <pthread.h>

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

- thread: thread struct
- attr: stack size, scheduling priority… (*optional*)
- start_routine: function pointer to thread routine
- arg: argument to pass to thread routine (*optional*)

PTHREAD_CREATE – PASS ANY DATA

```
#include <pthread.h>

typedef struct __myarg_t {
        int a;
        int b;
} myarg_t;

void *mythread(void *arg) {
        myarg_t *m = (myarg_t *) arg;
        printf("%d %d\n", m->a, m->b);
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t p;
        int rc;

        myarg_t args;
        args.a = 10;
        args.b = 20;
        rc = pthread_create(&p, NULL, mythread, &args);
        …
}
```

PASSING A SINGLE VALUE

**Using this approach on your Ubuntu VM,
How large (in bytes) can the primitive data type be?**

How large (in bytes) can the primitive data type be on a 32-bit operating system?

```
        printf("%d\n", m);
...
        int rc, m;
        pthread_create(&p, NULL, mythread, (void *) 100);
        pthread_join(p, (void **) &m);
        printf("returned %d\n", m);
        return 0;
    }
```


WAITING FOR THREADS TO FINISH

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

- thread: which thread?
- value_ptr: pointer to return value
  type is dynamic / agnostic

- Returned values *must* be on the heap
- Thread stacks destroyed upon thread termination (join)
- Pointers to thread stack memory addresses are invalid
  - May appear as gibberish or lead to crash (seg fault)
- Not all threads join – ***What would be Examples ??***

What will this code do?

```
struct myarg {
  int a;
  int b;
};

void *worker(void *arg)
{
  struct myarg *input = (struct myarg *) arg;
  printf("a=%d b=%d\n",input->a, input->b);
  struct myarg output;
  output.a = 1;
  output.b = 2;
  return (void *) &output;
}

int main (int argc, char * argv[])
{
  pthread_t p1;
  struct myarg args;
  struct myarg *ret_args;
  args.a = 10;
  args.b = 20;
  pthread_c
  pthread_j
  printf("
  return 0;
}
```

Data on thread stack (pointing to `struct myarg output;`)

```
$ ./pthread_struct
a=10 b=20
Segmentation fault (core dumped)
```

How can this code be fixed?

How about this code?

```
struct myarg {
  int a;
  int b;
};

void *worker(void *arg)
{
  struct myarg *input = (struct myarg *) arg;
  printf("a=%d b=%d\n",input->a, input->b);
  input->a = 1;
  input->b = 2;
  return (void *) &input;
}

int main (int argc, char * argv[])
{
  pthread_t p1;
  struct myarg args;
  struct myarg *ret_args;
  args.a = 10;
  args.b = 20;
  pthread_create(&p1, NULL, worker, &args);
  pthread_join(p1, (void *)&ret_args);
  printf("returned %d %d\n", ret_args->a, ret_args->b);
  return 0;
}
```

```
$ ./pthread_struct
a=10 b=20
returned 1 2
```

ADDING CASTS

- Casting
- Suppresses compiler warnings when passing "typed" data where (void) or (void *) is called for
- Example: uncasted capture in pthread_join

```
pthread_int.c: In function 'main':
pthread_int.c:34:20: warning: passing argument 2 of 'pthread_join'
from incompatible pointer type [-Wincompatible-pointer-types]
   pthread_join(p1, &p1val);
```

- Example: uncasted return

```
In file included from pthread_int.c:3:0:
/usr/include/pthread.h:250:12: note: expected 'void **' but argument
is of type 'int **'
 extern int pthread_join (pthread_t __th, void **__thread_return);
```


ADDING CASTS - 2

- pthread_join

```
int * p1val;
int * p2val;
pthread_join(p1, (void *)&p1val);
pthread_join(p2, (void *)&p2val);
```

- return from thread function

```
int * counterval = malloc(sizeof(int));
*counterval = counter;
return (void *) counterval;
```

QUESTIONS