TCSS 422: OPERATING SYSTEMS

Concurrency: An Introduction

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

FEEDBACK FROM 4/11

- Why "ticketing"? It sounds like a waste to keep track of it?
  - Ticket-based schedulers feature a simple implementation
    - E.g. pick a random number to determine next job to run
  - Ticket-based approaches enable proportional time sharing
    - MLFQ, RR use time quantums (e.g. 10ms, 20ms)
    - Tickets provide a mechanism for a proportional quantum based on number of jobs
- Ticket assignment
  - User selects number of tickets for their jobs
  - OS converts (currency exchange) user allotment to system allotment
  - Totals user tickets to find proportions

FEEDBACK - 2

- What are users?
  - Any user on a system, with potentially multiple jobs
- Are they (users) the same as jobs?
  - No, a user owns and runs one or more jobs on the system
  - Not every user is a person
  - The "root" user runs most OS jobs (e.g. kernels, daemons, servers)
- Do all users get the same number of tickets?
  - OS "converts" user tickets to system tickets through currency exchange mechanism

FEEDBACK - 3

- What is the benefit of a stride scheduler? (pros and cons)
  - Stride solves the problem with poor fairness for short running jobs under the lottery scheduler
  - Achieves fairness (even time distribution) more quickly
  - In general, stride scheduler suffers from similar issues as ticket schedulers, except for improving on fairness
  - Ticket assignment is still an open problem …
  - Stride value based on number of tickets

FEEDBACK - 4

- What is the difference between tickets and strides?
- Ticket represents the proportion of CPU a job should receive relative to other jobs
- Stride is value counter must reach for scheduler to pass to the next job.
- Scheduler always chooses jobs with lowest pass value.
- Stride value is inverse in proportion to number of tickets held.
- Jobs with low stride **always** favored for execution.

STRIDE SCHEDULER EXAMPLE

- 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

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	…

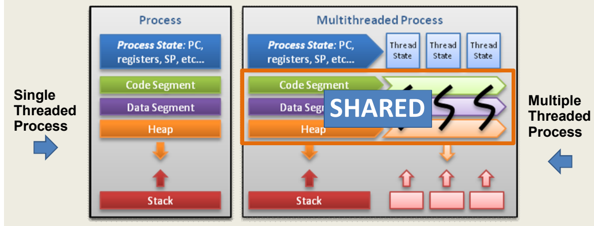
Initial job selection is random. All @ 0

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

OBJECTIVES

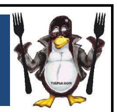
- Chapter 26
  - Introduction to threads
  - Race condition
  - Critical section

THREADS

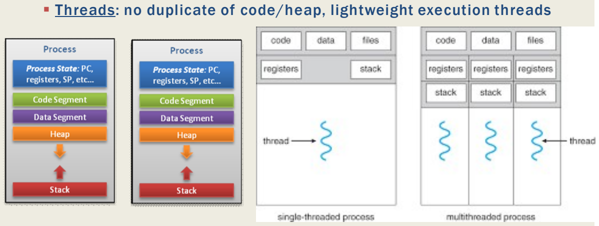


©Alfred Park, http://randu.org/tutorials/threads

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

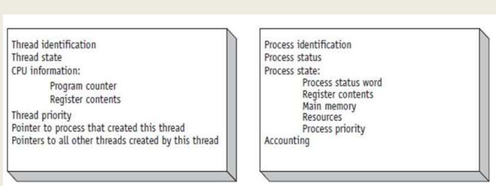


THREADS - 2

- Enables a single process (program) to have multiple "workers"
- Supports independent path(s) of execution within a program
- Each thread has its own Thread Control Block (TCB)
  - PC, registers, SP, and stack
- Code segment, memory, and heap are shared

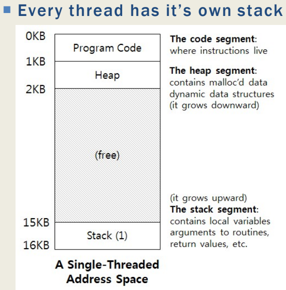
PROCESS AND THREAD METADATA

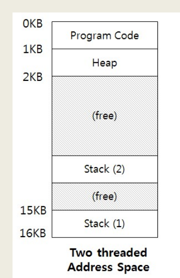
- Thread Control Block vs. Process Control Block



SHARED ADDRESS SPACE

- Every thread has it's own stack / PC





THREAD CREATION EXAMPLE

```c
#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'		

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?

COUNTER EXAMPLE

- Show example
- A + B : ordering
- Counter: incrementing global variable by two threads

RACE CONDITION

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

OS	Thread1	Thread2	PC (after instruction)	%eax (after instruction)	counter (after instruction)
	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 produces a ***race condition***.
- ***Atomic execution*** (*all code executed as a unit*) must be ensured in ***critical*** sections
  - These sections must be ***mutually exclusive***

April 13, 2017 | TCSS422: Operating Systems [Spring 2017] Institute of Technology, University of Washington - Tacoma | L5.19

LOCKS

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

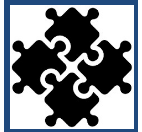
```
lock_t mutex;
. . .
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Critical section

- Counter example revisited

April 13, 2017 | TCSS422: Operating Systems [Spring 2017] Institute of Technology, University of Washington - Tacoma | L5.20

LINUX THREAD API



April 13, 2017 | TCSS422: Operating Systems [Spring 2017] Institute of Technology, University of Washington - Tacoma | L5.21

QUESTIONS

April 13, 2017 | TCSS422: Operating Systems [Spring 2017] Institute of Technology, University of Washington - Tacoma | L5.22