UNIVERSITY of WEST FLORIDA

COP4634: Systems & Networks I

Threads

Threads Concept

- Definition: A thread (= job) is a lightweight process, representing a single unit of program execution within a process.
  - shares code, data & file descriptor table
  - has individual stack, registers & thread control block (TCB)

- A process may run multiple threads concurrently.

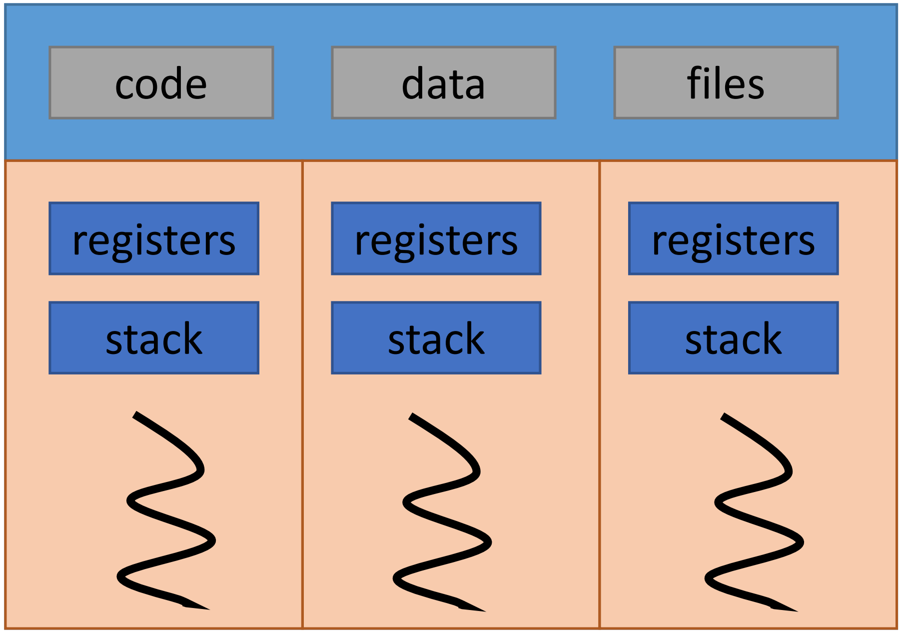
Threads Concept (cont.)

- Two types of threads
  - User-level threads
  - Kernel threads

- Thread creation similar to process creation except:
  - Processes begins execution at main().
  - Thread begins execution at specified function.
  - Creation call identifies initial function.

Multithreaded Programming

- Multiple threads may be executed simultaneously in a single process.
- Threads share code, global data, & open files of the process.



Process – Threads Differences

- Threads share process resources, processes share computer resources.
  - OS maintains per thread
    - program counter,
    - stack
    - state
    - registers
  - OS maintains per processes
    - an address space,
    - list of open files,
    - child process,
    - signals and signal handlers
    - accounting

Benefits of Multithreaded Programs

- Threads are more responsive than processes.
- Threads share resources.
- Threads are more economical than processes.

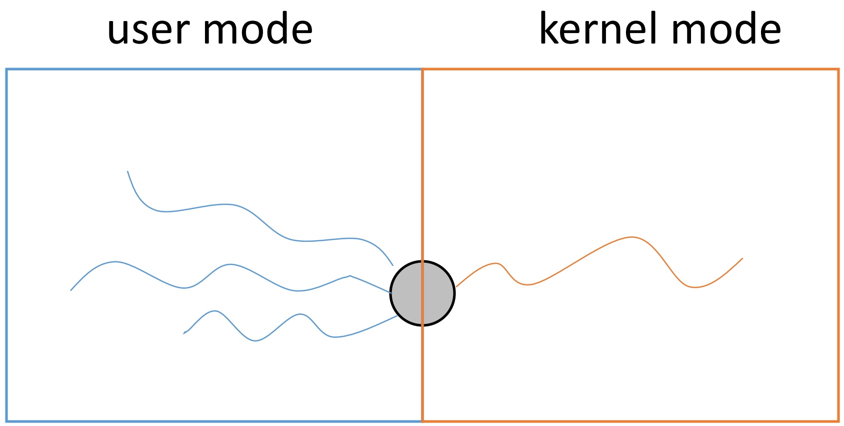
Sample Multithreaded Programs

- Client – Server program
  - server handles each client request in a separate thread
  - server constructs a pool of threads to distribute work

- Web Browser
  - multiple browser windows downloading Web pages simultaneously
  - may not be multi-threaded but multi-processed

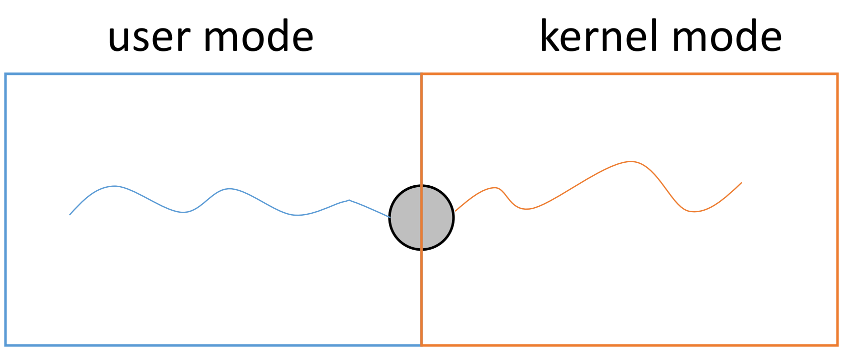
Multithreading Models

- Many-to-one model: many user-level threads maps to one kernel thread.
  - if a single thread makes a blocking call, all threads will be blocked



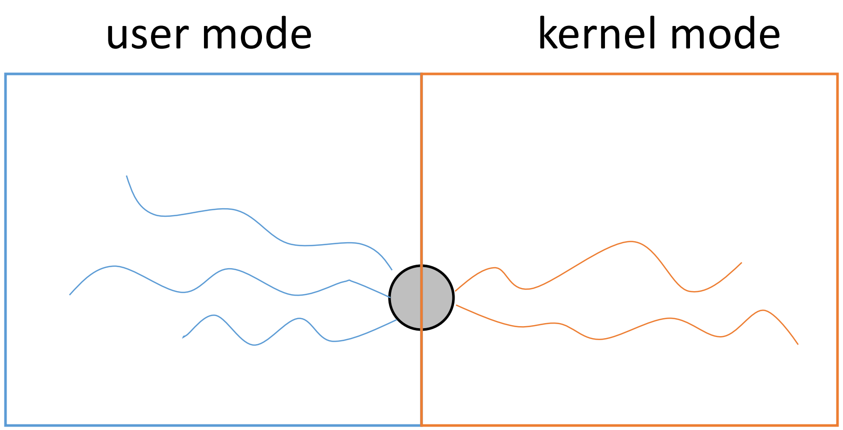
Multithreading Models (cont.)

- One-to-one model: a user-level thread maps to a single kernel thread.
  - provides more concurrency
  - more overhead to create a new user level thread
  - can only create as many user level threads as there are kernel threads available



Multithreading Models (cont.)

- Many-to-many model: many user-level threads map to many kernel threads.
  - combines advantages of one-to-one and many-to-many model



Example – Web Server

- Parent waits for http request from network
- Parent creates child thread to handle request
- Child thread:
  - reads disk to find http page
  - sends http page to client
  - exits
- Parent thread:
  - waits for next request

Review: Process P_{296} Before fork()

Stack	i	**?**
	j	**100**
Data	g	**5**
Code	LOAD R2, 100	
	STOR R2, j	
	LOAD R3, 5	
	STOR R3, g	
	CALL fork	
	STOR RV, i	
PCB	PID	**296**
	R1	**?**
	R2	**100**
	R3	**5**
	PC	**5**

```
int g;
int main() {
   int i, j;
   j = 100;
   g = 5;
   i = fork();
   printf(“%d:%d:%d\n”,g,i,j);
   return 0;
}
```

Review: Processes P_{296} After fork()

Process P_{296}

Stack	i	**?**
	j	**100**
Data	g	**5**
Code	`LOAD R2, 100`	
	`STOR R2, j`	
	`LOAD R3, 5`	
	`STOR R3, g`	
	`CALL fork`	
	`STOR RV, i`	
PCB	PID	**296**
	R1	**?**
	R2	**100**
	R3	**5**
	PC	**5**

← parent

child →

created with fork()

nothing shared

Process P_{321}

Stack	i	**?**
	j	**100**
Data	g	**5**
Code	`LOAD R2, 100`	
	`STOR R2, j`	
	`LOAD R3, 5`	
	`STOR R3, g`	
	`CALL fork`	
	`STOR RV, i`	
PCB	PID	**321**
	R1	**?**
	R2	**100**
	R3	**5**
	PC	**5**

Thread of Execution

An indication of

- where is current execution state
    - Register values in PCB per thread (aka TCB)
- how this point was reached
    - Runtime stack per thread
- what should be done next
    - PC per thread

Process – Another View of Process P_{296}

Thread

(of control)

"single-threaded process"

OR

"single-threaded task"

OR

process

Data g **5**

File Descriptor Table

stdin
stdout
stderr

Code

```
LOAD R2, 100
STOR R2, j
LOAD R3, 5
STOR R3, g
CALL fork
STOR RV, i
```

Stack

i	**?**
j	**100**

TCB

TID	**296**
R1	**?**
R2	**100**
R3	**5**
PC	**5**

A New Thread in Process P_{296}

New Thread

"multi-threaded process"

OR

"multi-threaded task"

Data g **5**

File Descriptor Table

stdin
stdout
stderr

Code

```
LOAD R2, 100
STOR R2, j
LOAD R3, 5
STOR R3, g
CALL fork
STOR RV, i
```

Stack

i	**?**
j	**100**

Stack

m	**?**
n	**?**
p	**?**

TCB

TID	**296**
R1	**?**
R2	**100**
R3	**5**
PC	**5**

TCB

TID	**312**
R1	**?**
R2	**?**
R3	**?**
PC	**?**

Creating a Thread in C

- Define a function for the thread
  - AR for function pushed onto thread stack
  - Thread terminates when function returns
  - must be return type `void *`
  - must accept one param of type void *

```
void * threadMain(void *threadParam);
void * doCompute(void *records);
void * sortThis(void *array);
```

What is void *?

type *

pointer to type

`void`

anything - or nothing

`void *`

pointer to anything

can be assigned any pointer

```
void * vp;
char * cp;
int * ip;
double * dp;
. . .
vp = cp;
vp = ip;
vp = dp;
```

What can be param?

What about other types? Send pointer.

Must be typecasted to `void *`

Compiler will complain otherwise

```
double d;
int i;
…
threadFunctionA( (void *)&d );
threadFunctionB( (void *)&i );
```

Typecast?

Typecasting doesn't change data;
it only shuts up compiler

```
long val1, val2;
char *cp;
val1 = 123456789;
cp = (char *)val1;
val2 = (long)cp;
printf(“%ld\t%ld\n”, val1, val2);
```

How to Use void *

1. Typecast what you are passing

```
int i =123456;
threadFunctionB( (void *)&i );
```

do it

2. Typecast what you received

```
void* threadFunctionB(void *param)
{
  int x = *(int *)param;
  printf("%d\n", x);
```

undo it

Example Problem in C

```
void * threadFunctionB( void * p) {
  int n = *(int *)p;
  printf(“%d\n”, n);
}
```

```
   int i;  // a global variable
  i = 1;  // initialize variable
  create_thread(threadFunctionB((void *)&i ));
  i = 2;
  create_thread(threadFunctionB((void *)&i));
  i = 3;
```

One Thread

Data i **1**

File Descriptor Table

stdin
stdout
stderr

Code

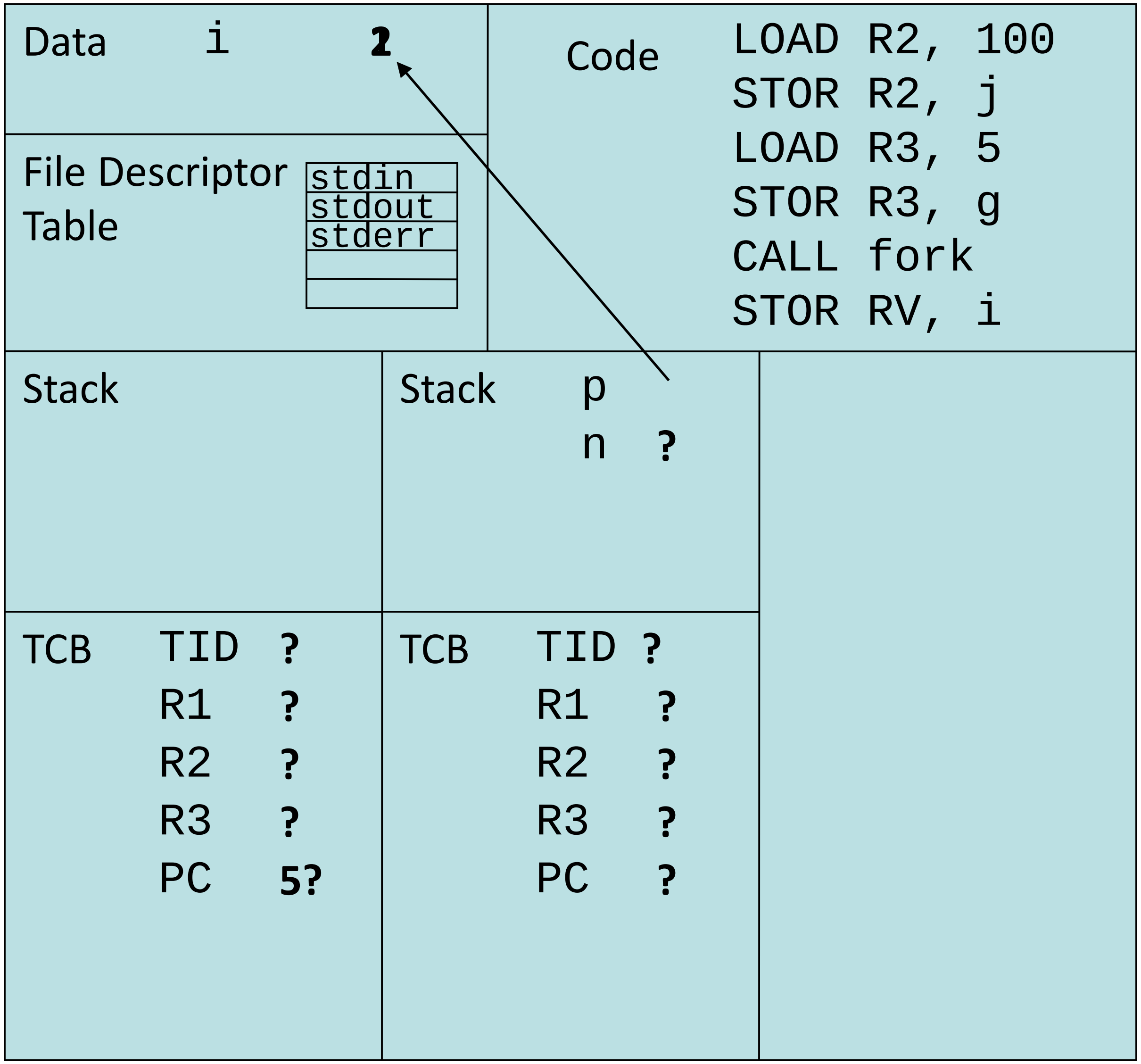
```
LOAD R2, 100
STOR R2, j
LOAD R3, 5
STOR R3, g
CALL fork
STOR RV, i
```

Stack

TCB

TID	**?**
R1	**?**
R2	**?**
R3	**?**
PC	**5?**

Two Threads



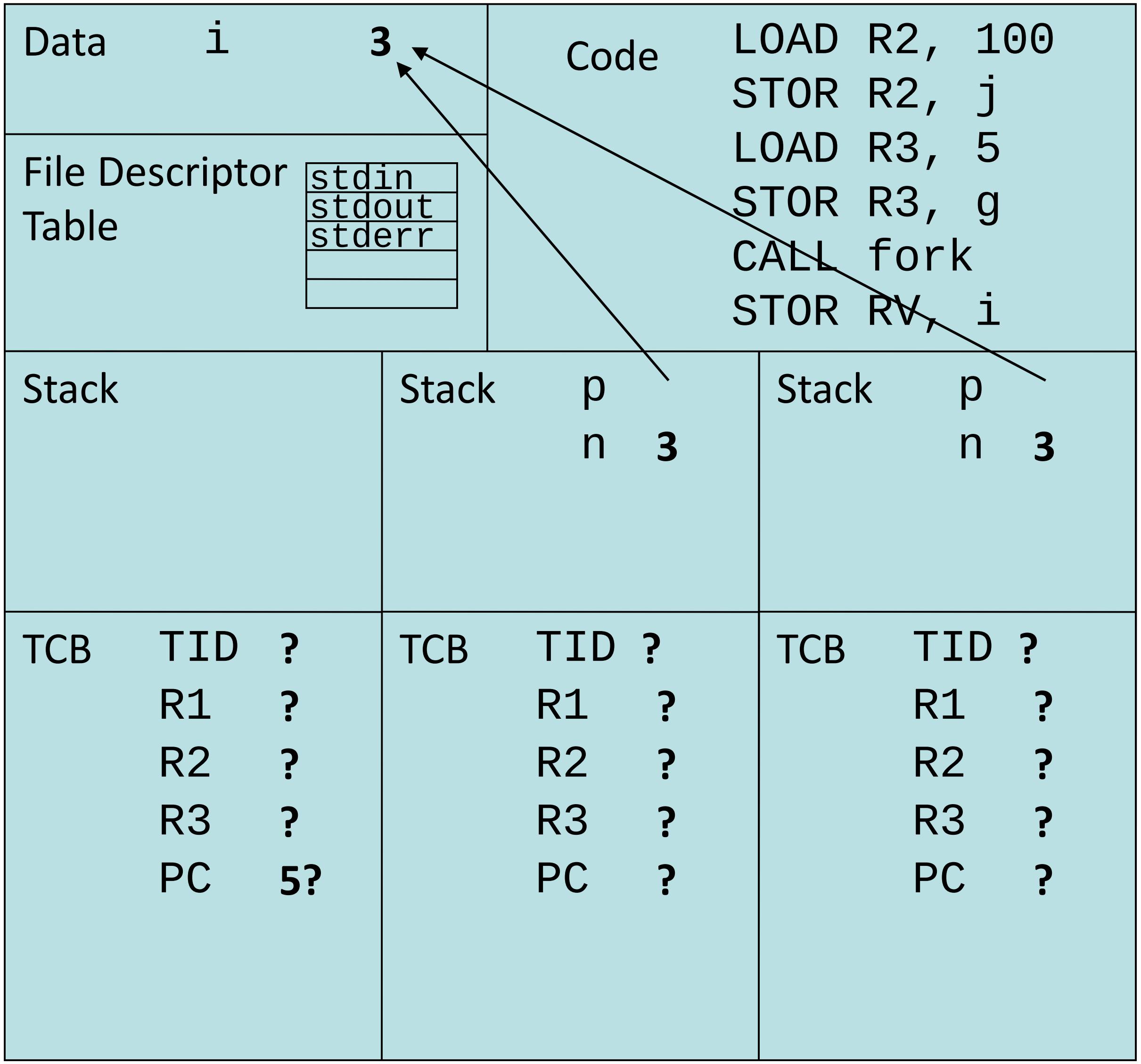
Data i 2
File Descriptor Table
stdin
stdout
stderr
Code
LOAD R2, 100
STOR R2, j
LOAD R3, 5
STOR R3, g
CALL fork
STOR RV, i
Stack
Stack p
n ?
TCB TID ?
R1 ?
R2 ?
R3 ?
PC 5?
TCB TID ?
R1 ?
R2 ?
R3 ?
PC ?

Three Threads

Data i 3	Code	LOAD R2, 100 STOR R2, j LOAD R3, 5 STOR R3, g CALL fork STOR RV, i
File Descriptor Table	stdin stdout stderr	

Stack	Stack p n ?	Stack p n ?
TCB TID ? R1 ? R2 ? R3 ? PC 5?	TCB TID ? R1 ? R2 ? R3 ? PC ?	TCB TID ? R1 ? R2 ? R3 ? PC ?

Shared Memory Location



POSIX Threads in C

- IEEE 1003.1c POSIX API standard.
- Requirement for interface.
- Not a requirement for implementation.

```
pthread_create( … );
pthread_exit( RV );
pthread_join( threadID );
```

How To Use Functions in C

- Create a `void *function(void *)`
- Create a thread / call the function
  - Typecast the argument (`void *`)
- Function (child)
  - undo the argument typecasting
  - perform child thread requirements
  - return from the child thread
- Main (parent)
  - perform main thread requirements
  - wait for child thread to return

Another Example (in C)

```
int g;  /* global variable */

void * tFunc(void *param){
  int i;
  char *str = (char *)param;
  for( i=0; i<3; i++ ){
    printf(“%s [%d:%d]\n”,str, i, g);
    g++;
  }
  pthread_exit(0);
}
```

Calling threadfunc()

```
int main(int argc, char ** argv){
  pthread_t tidA, tidB;

  g = 1;
  printf("prethreads [%d]\n", g);

  pthread_create(&tidA, NULL, tFunc, (void *)"threadA");
  pthread_create(&tidB, NULL, tFunc, (void *)"threadB");

  printf("postthreads [%d]\n", g);

  pthread_join(tidA, NULL);
  pthread_join(tidB, NULL);

  printf("main done [%d]\n", g);

  return 0;
}
```

Creating a Thread in C++

- Passing thread function similar to C but simpler.

```
int main(int argc, char ** argv){
  g = 1;
  printf("prethreads [%d]\n", g);

  // create and start threads
  std::thread first (tFunc, "threadA");
  std::thread second (tFunc, "threadB");

  printf("postthreads [%d]\n", g);

  // join threads
  first.join();
  second.join();

  printf("main done [%d]\n", g);

  return 0;
}
```

One Possible Execution

```
prethreads [1]

postthreads [1]

threadA [0:1]

threadA [1:2]

threadB [0:3]

threadB [1:4]

threadB [2:5]

threadA [2:6]

main done [7]
```

Assumptions

- Main thread runs before any child
- Main thread waits for new threads to terminate
- Thread A prints 2 lines before quantum expires
- Thread B runs to completion
- Thread A completes

Implementation

- Library or Kernel
- Library – original method
  - thread is executed in user mode
  - kernel unaware of threads
  - processes are scheduled, not threads
  - fast to create/switch/destroy
  - `yield()` gives CPU to next process
  - quantum shared among threads executed within process

Time – Library Threads

- Assumptions
  - 10 tick context switch in kernel
  - 2 tick context switch in library
  - 100 tick quantum
- We already have 1 process in system (D)
- New process (P) assumptions
  - arrives at time 33
  - composed of 1 thread
  - needs 500 ticks to complete computation
  - quantum shared equally among threads

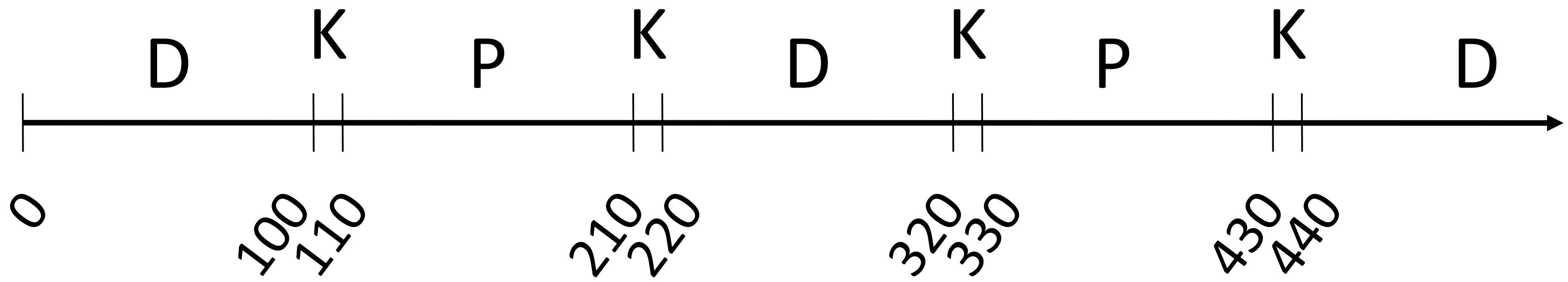
Time – 1 Library Thread

- 0 D
- 100 K
- **110** **P** 100
- 210 K
- 220 D
- 320 K
- **330** **P** 200
- 430 K

440 D
540 K
550 **P** 300
650 K
660 D
760 K
770 **P** 400
870 K

880 D
980 K
990 **P** 500
1090 K
1100 D
1200 K
1210 D
1310 K

Time – 1 Library Thread



Timeline of what process is on the CPU

Assumes P arrives at time 33 (*new*)

P goes to end of ready list (*ready*)

D already on CPU

Execution Statistics

- t_a arrival time
- t_e execution start time
- t_d departure time
- Response = $t_e – t_a$
  - fastest possible user-observable reaction
- Turnaround = $t_d – t_a$
  - press enter until prompt returns
  - enter system until exit system

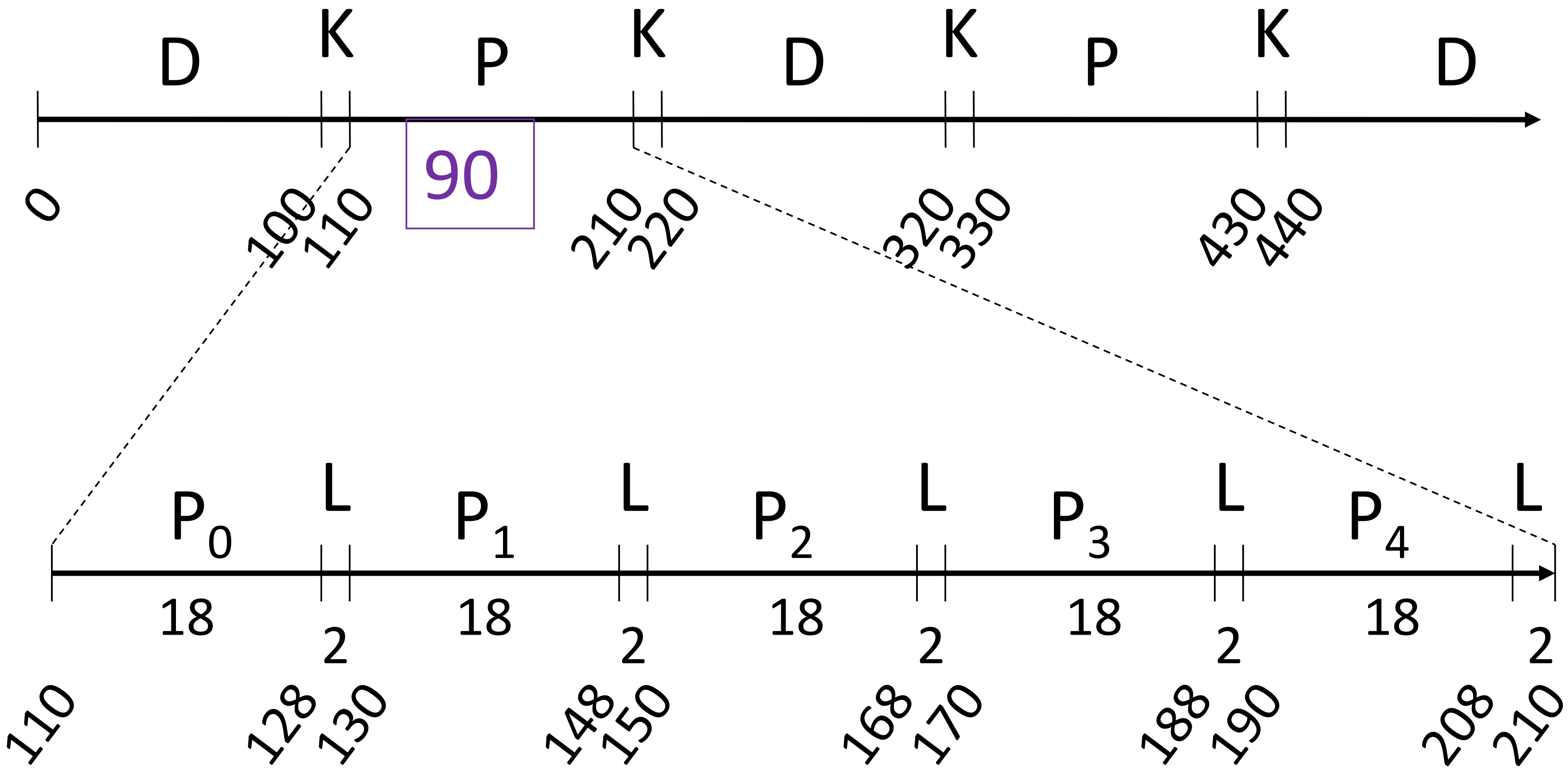
Stats from Examples

	T_a	T_e	T_d	Resp	T/A
1 thread	33	110	1090	77	1057
5 library threads					
5 kernel threads					

Time – 5 Library Threads

0	D	440	D	880	D
100	K	540	K	980	K
110	**P**	**550**	**P**	**990**	**P**
210	K	650	K	1090	K
220	D	660	D	1100	D
320	K	760	K	1200	K
330	**P**	**770**	**P**	**1210**	**P**
430	K	870	K	1264	K

Time – 5 Library Threads

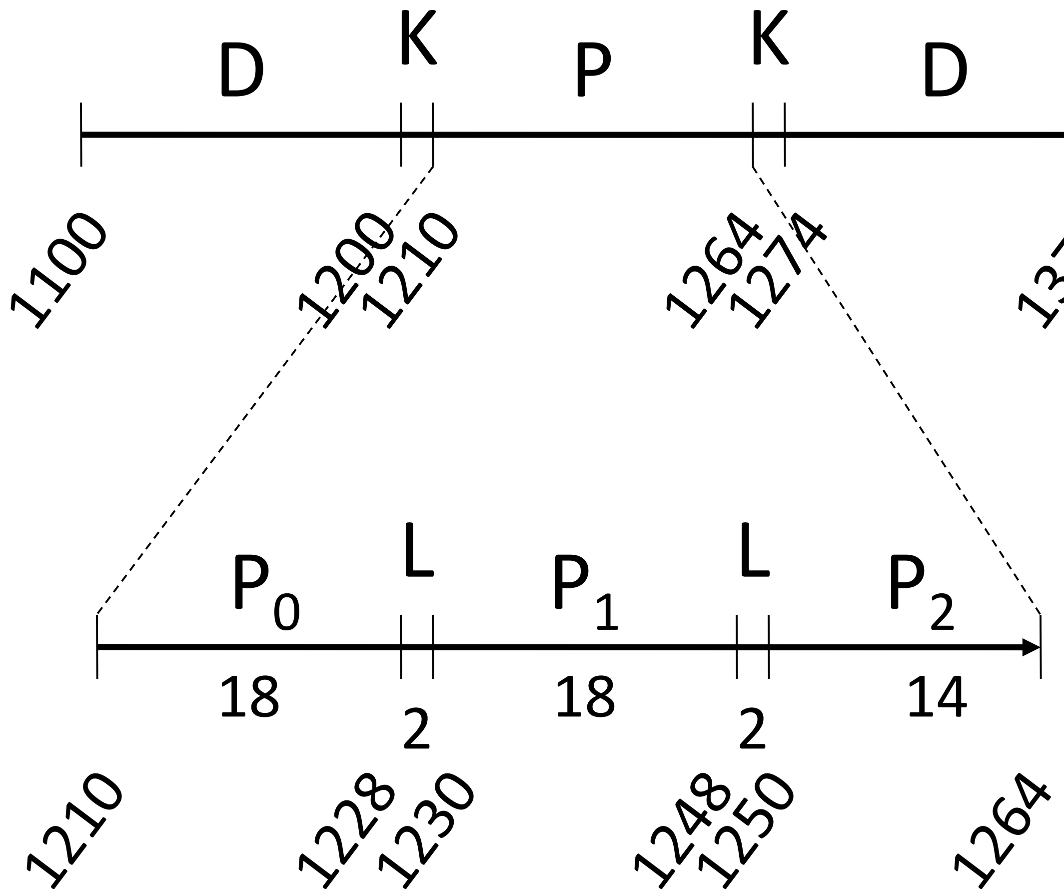


Time – 5 Library Threads

0	D		440	D		880	D	
100	K		540	K		980	K	
110	**P**		**550**	**P**	270	**990**	**P**	450
210	K	90	650	K		1090	K	
220	D		660	D		1100	D	
320	K		760	K		1200	K	
330	**P**		**770**	**P**	360	**1210**	**P**	500
430	K	180	870	K		1264	K	

Ready List: D, P

Time – 5 Library Threads



Stops at 500

Continues 18/2

Doesn’t use all of quantum

Stats from Examples

	T_a	T_e	T_d	Resp	T/A
1 thread	33	110	1090	77	1057
5 library threads	33	110	1264	77	1231
5 kernel threads					

Implementation

- Library or Kernel
- Kernel – newer method
  - implemented in kernel code
  - kernel aware of threads (TID & PID)
  - threads are scheduled
  - slow to create/switch/destroy
  - `yield()` gives CPU to next thread
  - quantum explicitly for thread

Time – Kernel Threads

- Assumptions
  - 10 tick context switch in kernel
  - 100 tick quantum
- We already have 1 process in system (D)
- New process (P) assumptions
  - composed of 1 thread
  - needs 500 ticks to complete computation

Time – 1 Kernel Thread

0	D		440	D		880	D	
100	K		540	K		980	K	
110	**P**	100	**550**	**P**	300	**990**	**P**	500
210	K		650	K		1090	K	
220	D		660	D		1100	D	
320	K		760	K		1200	K	
330	**P**	200	**770**	**P**	400	1210	D	
430	K		870	K		1310	K	

Stats from Examples

	T_a	T_e	T_d	Resp	T/A
1 thread **Same**	33	110	1090	77	1057
5 library threads	33	110	1264	77	1234
5 kernel threads					

Time – 5 Kernel Threads

0	D	**440**	**P_3**	880	D
100	K	540	K	980	K
110	**P_0**	**550**	**P_4**	990	D
210	K	650	K	1090	K
220	**P_1**	660	D	1100	D
320	K	760	K	1200	K
330	**P_2**	770	D	1210	D
430	K	870	K	1264	K

100

200

300

400

500

Ready List: D, P_0, P_1, P_2, P_3, P_4

Stats from Examples

	T_a	T_e	T_d	Resp	T/A
1 thread	33	110	1090	77	1057
5 library threads	33	110	1264	77	1231
5 kernel threads	33	110	650	77	617

Thread I/O

- Library threads
  - kernel schedules process
  - thread initiates I/O, process blocks
  - no other thread can run
- Kernel threads
  - kernel schedules threads
  - thread initiates I/O, thread blocks
  - doesn’t effect other threads

Question

- Which is better for web server implementation?
  - library threads?
  - kernel threads?
  - multiple processes (1 thread each)?
- Why?

Summary Threads

- Threads are units of executions within a process.
- Multiple threads may be executed simultaneously within a process.
- Threads have the states ready, running, and blocked.
- Threads can be kernel or user-level threads.
- Kernel threads: OS executes threads, not processes.
- Threads may need to be synchronized to avoid race conditions.