CS310 Operating Systems

Lecture 22: Condition Variables

Ravi Mittal
IIT Goa

Acknowledgements !

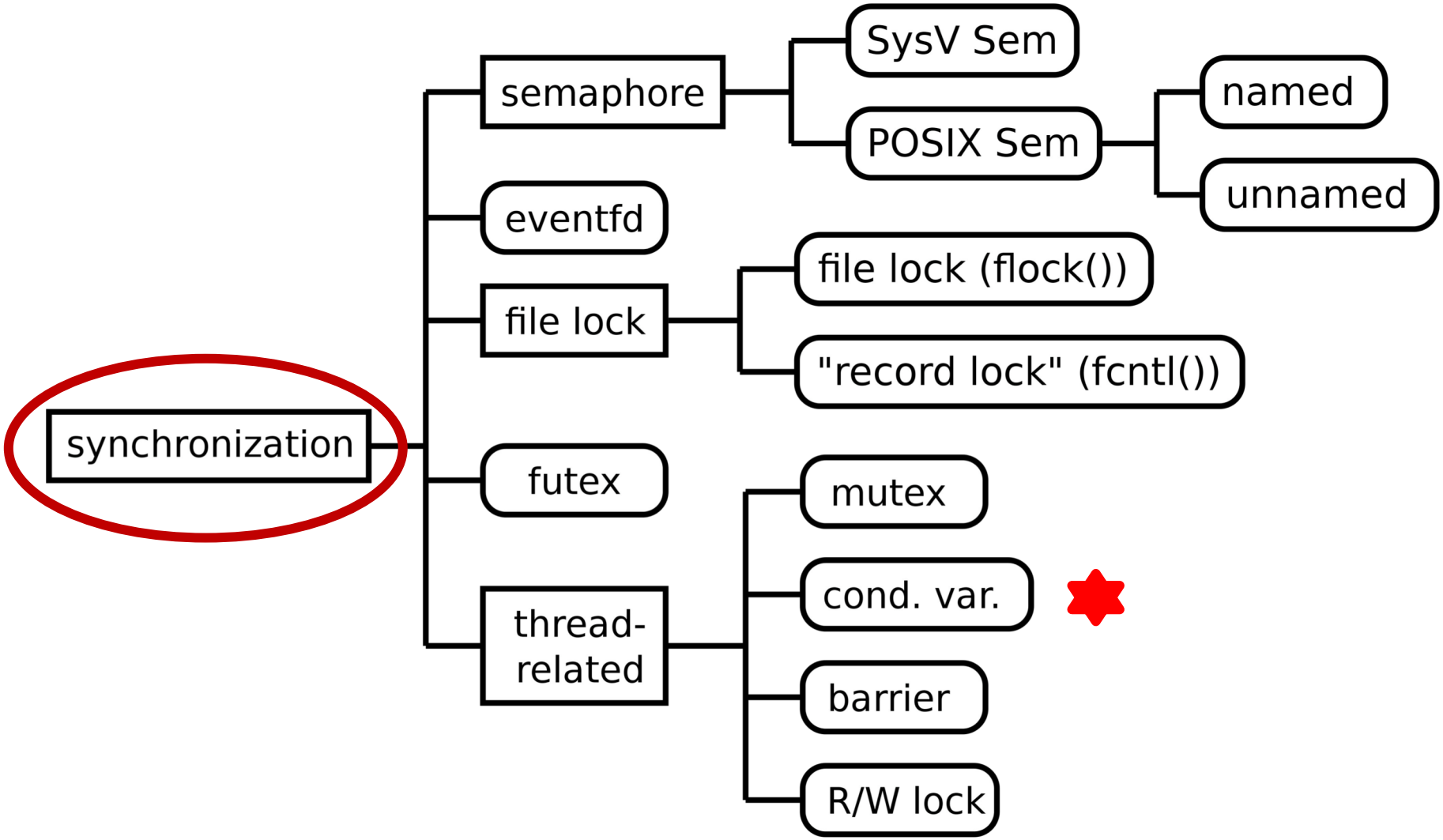
- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - Book: Operating System: Three Easy Pieces, by Remzi H Arpaci-Dusseau, Andrea C Arpaci-Dusseau, Chapter 30 Condition Variables
  - https://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf

Reading

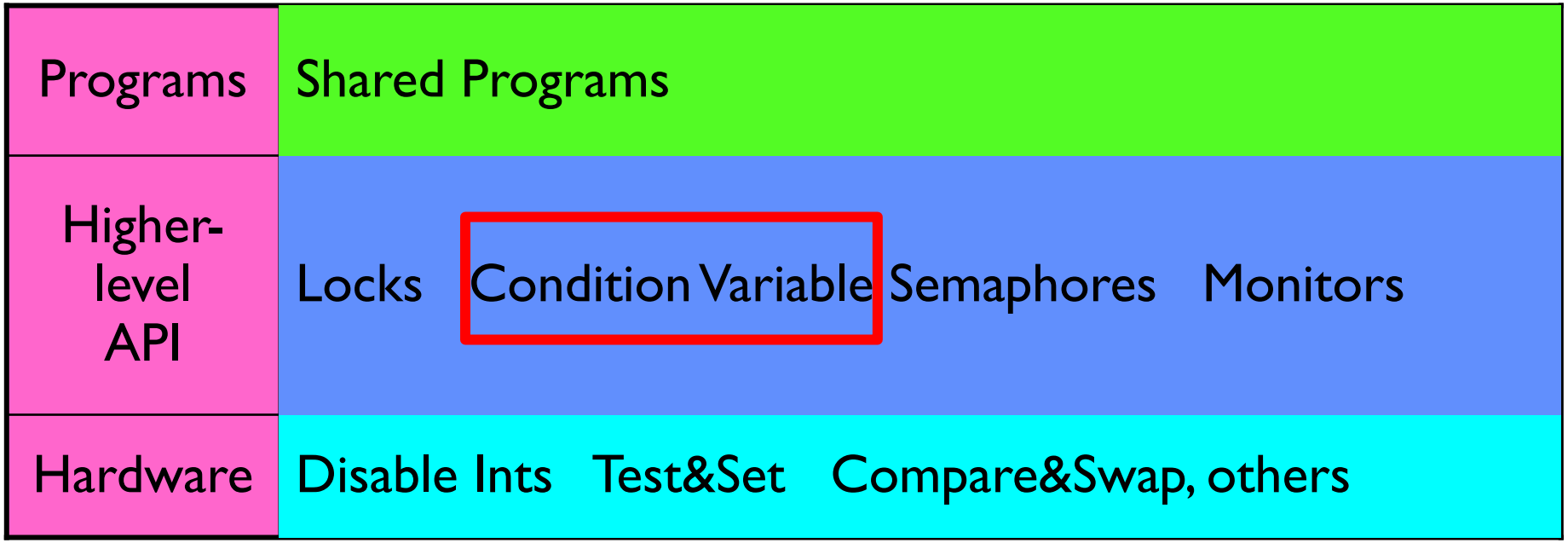
- Book: Operating Systems: Three Easy Pieces, Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau
  - Chapter 30 Condition Variable
    - https://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf

Previous Classes on IPC and Synchronization

Needs for Synchronization



We will start with High level primitives



Atomic Read-Modify-Write Instructions

- Hardware instructions that allows us to test and set or compare and swap, operations atomically

- Test_and_Set
- Compare_and_Swap
- Load-Linked and Store-Conditional
- Fetch-And-Add

Last class

- We can build locks with these instruction

From book: Operating Systems: Three Easy Pieces

Today, we will study

- wait - implementation
- Condition Variable: Introduction
- Example: Parent waiting for the child
- Example: Producer Consumer Problem

Wait implementation

Wait implementation !

- Locks are meant for mutual exclusion
- However, threads may just want to synchronize!

- Locks are not the only primitives to build concurrent programs
- There are many cases where a thread wishes to check whether a condition is true before continuing its execution
  - Example: A parent thread might wish to check whether a child thread has completed before continuing
    - Example: join() system call
- How should this wait be implemented?

From book: Operating Systems: Three Easy Pieces

Parent waiting for it's child to finish

```
1  void *child(void *arg) {
2      printf("child\n");
3      // XXX how to indicate we are done?
4      return NULL;
5  }
6
7  int main(int argc, char *argv[]) {
8      printf("parent: begin\n");
9      pthread_t c;
10     Pthread_create(&c, NULL, child, NULL); // create child
11     // XXX how to wait for child?
12     printf("parent: end\n");
13     return 0;
14 }
```

- We would like to see the following output?

```
parent: begin
child
parent: end
```

From book: Operating Systems: Three Easy Pieces

Parent Waiting For Child: Spin-based Approach

```
1  volatile int done = 0;
2
3  void *child(void *arg) {
4      printf("child\n");
5      done = 1;
6      return NULL;
7  }
8
9  int main(int argc, char *argv[]) {
10     printf("parent: begin\n");
11     pthread_t c;
12     Pthread_create(&c, NULL, child, NULL); // create child
13     while (done == 0)
14         ; // spin
15     printf("parent: end\n");
16     return 0;
17 }
```

- Solution works but inefficient
- Parent spins and wastes CPU time
- Is there a way for parent to sleep until a condition comes true ?

From book: Operating Systems: Three Easy Pieces

Condition Variable (CV): Introduction

Definition

- Thread can make use of condition variable
  - To wait for a condition to become true
- A condition variable is an explicit queue
  - A thread is put into this queue when some state of execution (some condition) is not as desired
    - By waiting on a condition
  - Another thread when it changes its state can then wake up one (or more) waiting threads
    - By signalling on the condition
    - Thus allow it (them) to continue
- Any example from real life?

From book: Operating Systems: Three Easy Pieces

Condition Variable - primitives

- To declare a conditional variable

```
pthread_cond_t  c;
```

- This declares **c** as a condition variable
- A CV has two associated operations: wait() and signal()
  - `wait()` call is executed when a thread wishes itself to sleep
  - `signal()` is executed when an executing thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition

- POSIX calls

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

From book: Operating Systems: Three Easy Pieces

Condition Variable

- `wait()` call takes a mutex as a parameter
  - It assumes that this mutex is locked when `wait()` is called
  - `wait()` releases the lock and puts the itself (calling thread) to sleep (atomically)
  - The sleeping thread wakes up only after some other thread signals it
  - It must reacquire lock before returning to the caller

  - Note that (lock and unlock) is required to prevent certain race condition when a thread is trying to put itself to sleep

From book: Operating Systems: Three Easy Pieces

Example: Parent waiting for the child

Example: parent waits for child (1/5)

```
1   int done  = 0;
2   pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3   pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5   void thr_exit() {
6       Pthread_mutex_lock(&m);
7       done = 1;
8       Pthread_cond_signal(&c);
9       Pthread_mutex_unlock(&m);
10  }
11
12  void *child(void *arg) {
13      printf("child\n");
14      thr_exit();
15      return NULL;
16  }
17
18  void thr_join() {
19      Pthread_mutex_lock(&m);
20      while (done == 0)
21          Pthread_cond_wait(&c, &m);
22      Pthread_mutex_unlock(&m);
23  }
24
25  int main(int argc, char *argv[]) {
26      printf("parent: begin\n");
27      pthread_t p;
28      Pthread_create(&p, NULL, child, NULL);
29      thr_join();
30      printf("parent: end\n");
31      return 0;
32  }
```

From book: Operating Systems: Three Easy Pieces

Example (2/5)

- Parent creates a child thread
- Parent thread waits for child to complete
- Parent thread puts itself to sleep (on condition variable c)
- On completion the child thread signals sleeping thread (using c)
- There are two cases:
- Case 1:
  - Parent creates a child thread and continues running
  - Calls `thr_join()` to wait for the child to complete
  - In `thr_join()` routine
    - It acquires lock m
    - Checks if child is done (when done = 1)
    - Child has not done: so, it puts itself to sleep (CV queue) using `pthread_cond_wait()` on c and releases lock
  - Eventually child runs and prints “child” and calls `thr_exit()` to wakeup the parent

From book: Operating Systems: Three Easy Pieces

Example (3/5)

- Case 1 (continued)
  - `thr_exit()` call does the following
    - It acquires the lock
    - Sets state variable `done = 1`
    - Signals the parent to wake up by signaling on c
  - In `thr_join()` routine, Parent returns from `wait()` and unlocks the lock;
  - Parent prints “parent: end”
- Case 2: Child runs immediately after creation
  - Sets `done = 1`
  - Calls signal to wake up the parent thread
    - There is none, so it the routine just returns
  - The parent then runs and calls `thr_join()`
    - `Now done = 1`. It immediately returns;
  - Parent prints “parent: end”

From book: Operating Systems: Three Easy Pieces

Example (4/5): Why check condition on while loop?

- Why check condition with “while” loop and not “if”?
  - To avoid corner cases of thread being woken up even when condition not true (may be an issue with some library implementations)
  - Just a good practise

From book: Operating Systems: Three Easy Pieces

lock() when calling wait() ? Why ?

- Consider when the lock is not held in order to signal and wait
- What problem could occur ?

```
1  void thr_exit() {
2      done = 1;
3      Pthread_cond_signal(&c);
4  }
5
6  void thr_join() {
7      if (done == 0)
8          Pthread_cond_wait(&c);
9  }
```

- Race Condition: Missed wake up
  - The parent checks the value of done = 0
  - It is pre-empted
  - Child runs and sets done = 1
  - Since no thread is sleeping, none gets woken up
  - Now parent thread runs and sleeps forever

From book: Operating Systems: Three Easy Pieces

Producer Consumer Problem

Producer/Consumer Problem

- This is a common scenario in multithreaded programs
- Consider multiple (>= 1) producer threads and multiple (>= 1) consumer threads
- Producers general data items and place them in a buffer
- Consumers grab items from the buffer and consume them in some way
- Example:
  - Multi-threaded web server
  - A producer puts HTTP requests into a work queue (bounded buffer)
  - Consumer threads take request out of this queue and process them

From book: Operating Systems: Three Easy Pieces

Producer/Consumer Problem

- Bounded buffer is shared resource
  - Must synchronize access to it
- Producer can put data into buffer only if it has available slot
  - Not full
- Consumer can take data from the buffer only if it is not empty

From book: Operating Systems: Three Easy Pieces

Producer/Consumer with 2 Condition Variables

```
1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);
8          while (count == MAX)
9              Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

From book: Operating Systems: Three Easy Pieces

Put and Get routine

```
1   int buffer[MAX];
2   int fill_ptr = 0;
3   int use_ptr  = 0;
4   int count    = 0;
5
6   void put(int value) {
7       buffer[fill_ptr] = value;
8       fill_ptr = (fill_ptr + 1) % MAX;
9       count++;
10  }
11
12  int get() {
13      int tmp = buffer[use_ptr];
14      use_ptr = (use_ptr + 1) % MAX;
15      count--;
16      return tmp;
17  }
```

From book: Operating Systems: Three Easy Pieces

Producer/Consumer with 2 Condition Variables

- Two condition variables: `empty, fill`
  - To signal which type of thread should be woken up
- Producer thread
  - checks if the buffer is full (count = MAX)
    - If so, it puts itself in sleeping state (`empty` CV)
    - If not, it puts item into the buffer, and
      - Signals thread waiting on `fill` CV
- Consumer thread
  - Checks if the buffer is empty (count = 0)
  - If so, it puts itself into sleeping state (`fill` CV)
  - If not, it gets an item from buffer, and
    - Signals (wakes up) a thread sleeping on CV `empty`

From book: Operating Systems: Three Easy Pieces

Lecture Summary

- We have studied an important synchronization primitive
  - Condition Variable (CV)
- By allowing threads to sleep when some program state is not desired and signaling it to wake up
  - A large number of important synchronization problems can be solved
- We have looked into a solution of a very important producer/consumer problem
  - With the help of two condition variables