Lecture 23: Deadlock Avoidance

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE], result=0;
void array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  return sum; result += sum;
}

typedef struct {
  int low;
  int high;
  int sum;
} thread_args;

void *thread_func(void *ptr) {
  thread_args * t = ((thread_args *) ptr);
  t->sum = array_sum(t->low, t->high);
  return NULL;
}
```

Race condition !!!

```c
int main(int argc, char *argv[]) {
  int result;
  if (SIZE < 1024) {
    result = array_sum(0, SIZE);
  } else {
    pthread_t tid[NTHREADS];
    thread_args args[NTHREADS];
    int chunk = SIZE/NTHREADS;
    for (int i=0; i<NTHREADS; i++) {
      args[i].low=i*chunk; args[i].high=(i+1)*chunk;
      pthread_create(&tid[i],
                     NULL,
                     thread_func,
                     (void*) &args[i]);
    }
    for (int i=0; i<NTHREADS; i++) {
      pthread_join(tid[i]);
      result += args[i].sum;
    }
  }
  printf("Total Sum is %d\n", result);
  return 0;
}
```

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int A[SIZE], result=0;
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;
void array_sum(int low, int high) {
  int sum = 0;
  for (int i=low; i<high; i++) {
    sum += A[i];
  }
  pthread_mutex_lock(&m);
  result += sum;
  pthread_mutex_unlock(&m);
}
```

Race condition is fixed using mutual exclusion

Consumer(s):

1. pthread_mutex_lock(&mutex);
2. while(task_queue_size() == 0)
3. pthread_cond_wait(&cond, &mutex);
4. }
5. task = pop_task_queue();
6. pthread_mutex_unlock(&mutex);
7. execute_task (task);

Producer:

1. pthread_mutex_lock(&mutex);
2. int queue_size = task_queue_size();
3. push_task_queue(&task);
4. if(queue_size == 0) {
5. pthread_cond_broadcast(&cond);
6. }
7. pthread_mutex_unlock(&mutex);

- Race condition
- Producer consumer problem

Today's Class

- Properties of good locking algorithm
- The Dining philosophers
- Deadlock creation
- Deadlock avoidance
- Out of syllabus discussion

Properties of a Good Locking Algorithm

- Safety guarantee
  - Mutual exclusion
- Progress guarantee
  - Deadlock freedom
  - Starvation freedom

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Properties of a Good Locking Algorithm

- Mutual exclusion
- ***Deadlock freedom:*** *system as a whole makes progress.* If some thread calls **lock()** and never returns, then other threads must complete **lock()** and **unlock()** calls infinitely often.
- Starvation freedom

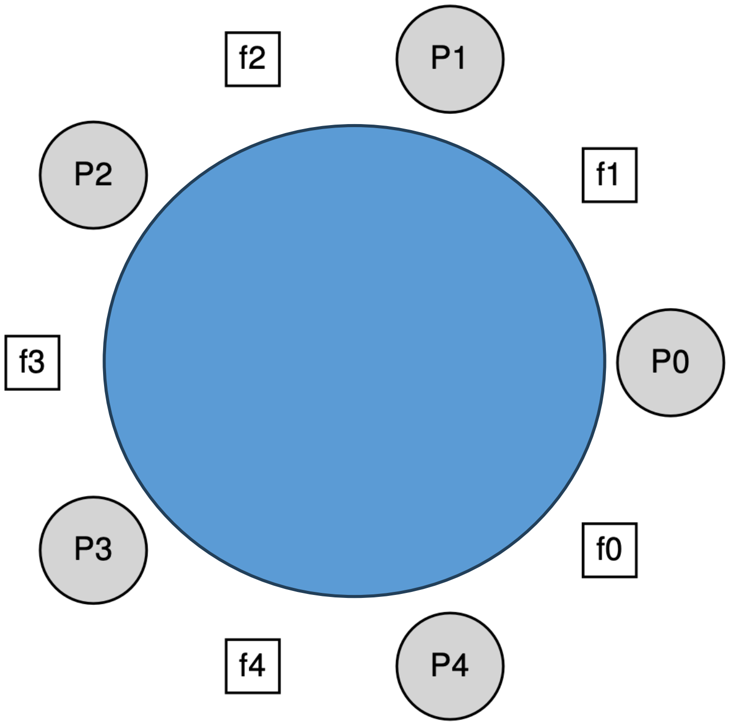
Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Properties of a Good Locking Algorithm

- Mutual exclusion
- Deadlock freedom: *system as a whole makes progress.* If some thread calls **lock()** and never returns, then other threads must complete **lock()** and **unlock()** calls infinitely often.
- ***Starvation freedom*** : A thread should not indefinitely hold the lock for doing some big computation while other threads keep waiting to get this lock

Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

The Dining Philosophers



- "N" number of philosophers sit on a round table
- One chopstick placed on the table between each philosophers
- Philosophers alternate between two states
  - Thinking → they don't use chopsticks
  - Eating → they have to pick the chopsticks on their left and right
- Goal: the philosophers should not go into indefinite waiting stage for picking up the chostick
  - Why there would be deadlock?

Money Transaction Between Accounts

```
class Account {
  int id;
  double balance;
  void debit(double amount);
  void credit(double amount);
};
```

```
class Bank {
  void fund_transfer() {
    Accounts numAccounts[N];
    Transfer pending[TOTAL];
    for(int i=0; i<TOTAL; i++) {
      pending[i].run();
    }
  }
};
```

```
class Transfer {
  Account source, destination;
  double amount;
  void run() {
    source.debit(amount);
    destination.credit(amount);
  }
};
```

- How to parallelize?
  - The for-loop is similar to the for-loop in parallel array sum we discussed in last lecture
  - Parallelize using multithreading

Money Transaction Between Accounts

```
class Account {
  int id;
  double balance;
  void debit(double amount);
  void credit(double amount);
};
```

```
class Bank {
  void fund_transfer() {
    Accounts numAccounts[N];
    Transfer pending[TOTAL];
    parallel_for(int i=0; i<TOTAL; i++) {
      pending[i].run();
    }
  }
};
```

```
class Transfer {
  Account source, destination;
  double amount;
  void run() {
    source.debit(amount);
    destination.credit(amount);
  }
};
```

- **parallel_for**
  - Shorthand to denote parallelization approach similar to parallel array sum (as in your assignment-5)
- Do you see any issues?
  - Race condition !!

Money Transaction Between Accounts

```
class Account {
  int id;
  double balance;
  void debit(double amount);
  void credit(double amount);
};
```

```
class Bank {
  void fund_transfer() {
    Accounts numAccounts[N];
    Transfer pending[TOTAL];
    parallel_for(int i=0; i<TOTAL; i++) {
      pending[i].run();
    }
  }
};
```

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
class Transfer {
  Account source, destination;
  double amount;
  void run() {
    pthread_mutex_lock(&mutex);
    source.debit(amount);
    destination.credit(amount);
    pthread_mutex_unlock(&mutex);
  }
};
```

- We can use mutex lock to fix race condition
- Do we still have parallelism?

Money Transaction Between Accounts

```
class Account {
  int id;
  double balance;
  pthread_mutex_t m =
         PTHREAD_MUTEX_INITIALIZER;
  void debit(double amount);
  void credit(double amount);
};
```

```
class Bank {
  void fund_transfer() {
    Accounts numAccounts[N];
    Transfer pending[TOTAL];
    parallel_for(int i=0; i<TOTAL; i++) {
      pending[i].run();
    }
  }
};
```

```
class Transfer {
  Account source, destination;
  double amount;
  void run() {
    source.lock(); destination.lock();
    source.debit(amount);
    destination.credit(amount);
    destination.unlock(); source.unlock();
  }
};
```

- Is this correct?

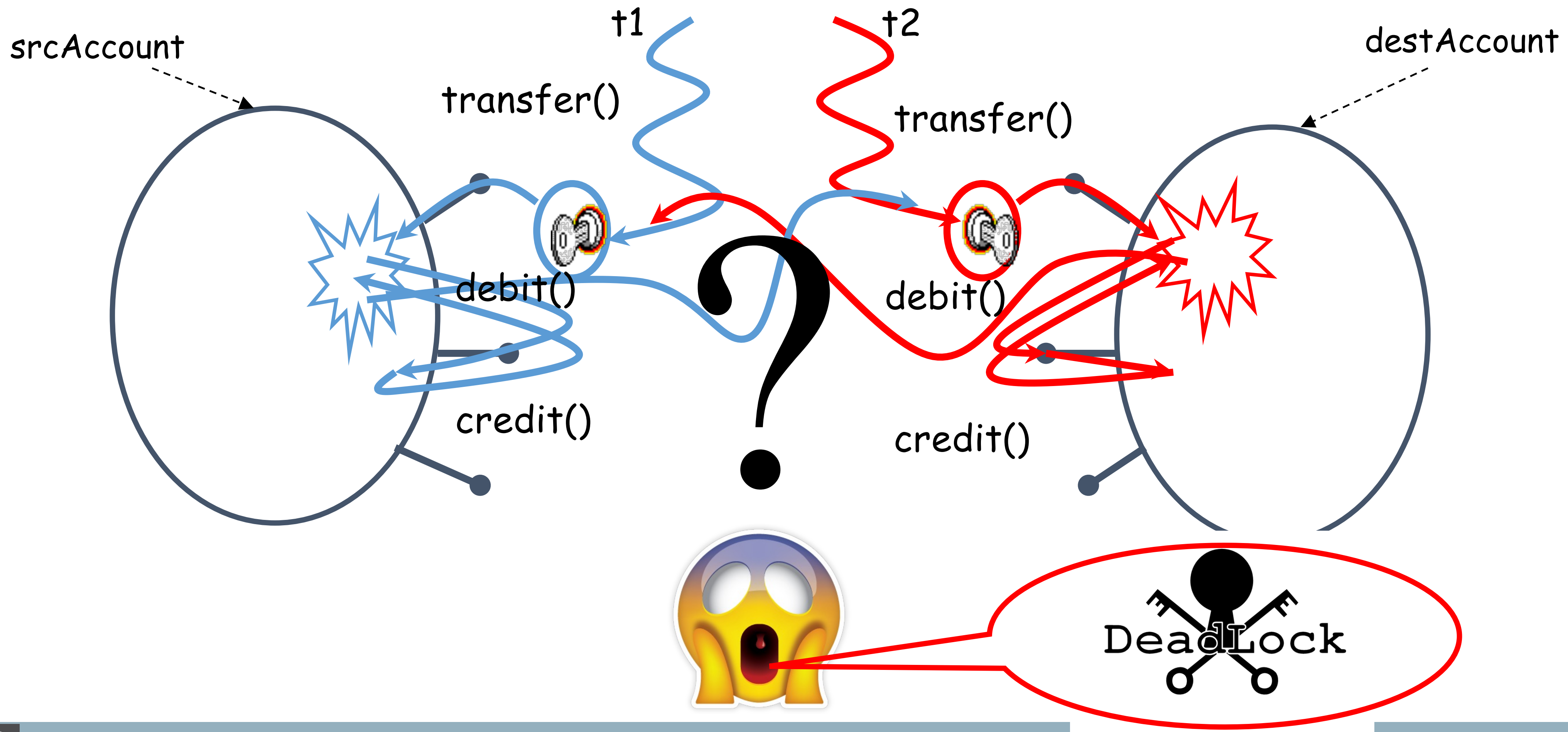
Money Transaction Between Accounts

```
class Account {
  int id;
  double balance;
  pthread_mutex_t
        PTHREAD_
  void debit(doub
  void credit(dou
};
```

```
class Transfer {
                                   n.lock();
                                   );
                                   rce.unlock();
```

```
class Bank {
  void fund_trans
    Accounts numAc
    Transfer pend
    parallel_for(
      pending[i].
    }
  }
};
```

Let's Analyze Our Money Transaction



Deadlock Avoidance

- **Deadlock occurs when multiple threads need the same locks but obtain them in different order**
- Not so easy to avoid deadlocks
- It's an active research area

Deadlock Avoidance

- Lock timeout
  - Put a timeout on lock attempts
    - pthread_mutex_timedlock
- Lock ordering
  - Ensure that all locks are taken in same order by any thread
    - **Let's try using it to fix our Bank Transaction program**

Money Transaction Between Accounts

```
class Account {
  int id;
  double balance;
  pthread_mutex_t m =
         PTHREAD_MUTEX_INITIALIZER;
  void debit(double amount);
  void credit(double amount);
};
```

```
class Bank {
  void fund_transfer() {
    Accounts numAccounts[N];
    Transfer pending[TOTAL];
    parallel_for(int i=0; i<TOTAL; i++) {
      pending[i].run();
    }
  }
};
```

```
class Transfer {
  Account source, destination;
  double amount;
  void run() {
    Account a1, a2;
    if(source.id < destination.id) {
      a1 = source; a2 = destination;
    } else {
      a1 = destination; a2 = source;
    }
    a1.lock(); a2.lock();
    source.debit(amount);
    destination.credit(amount);
    a2.unlock(); a1.unlock();
  }
};
```

Deadlock resolved using lock ordering

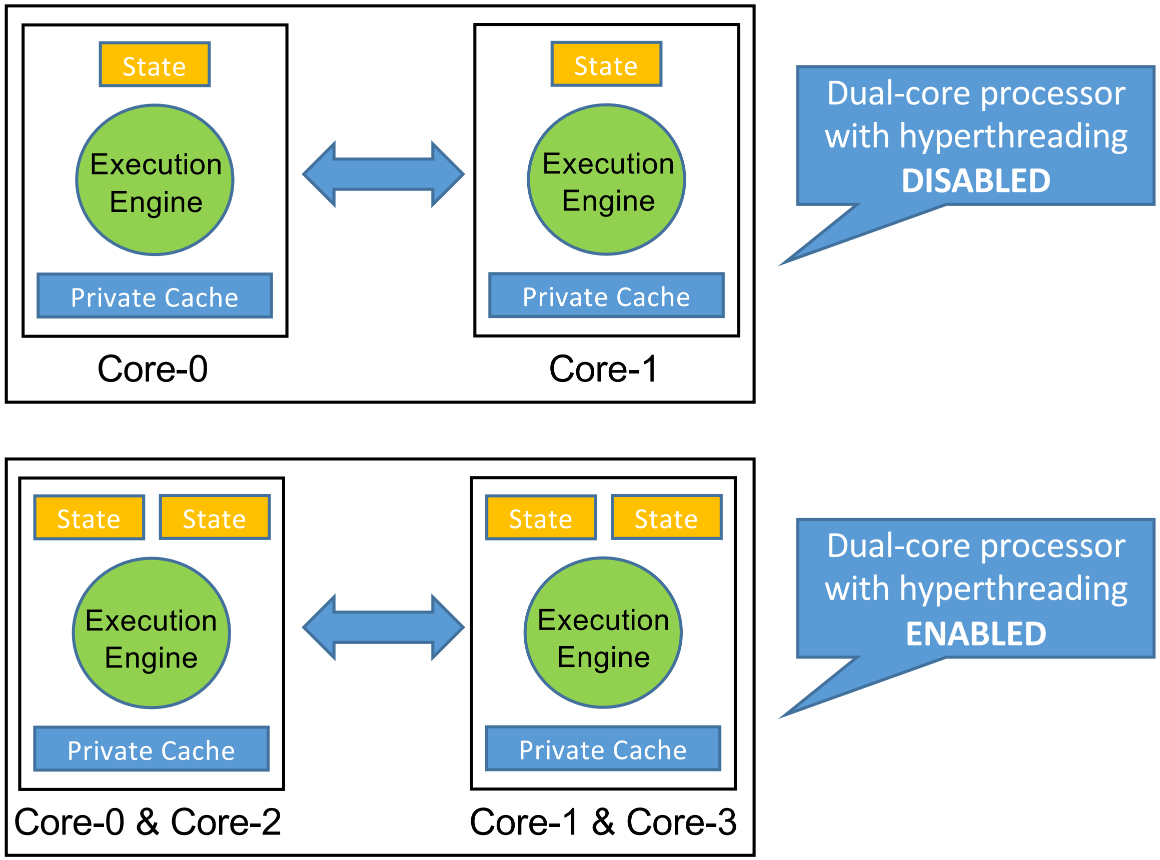
Miscellaneous Topics (Out of Syllabus)

- Physical core v/s logical cores
- NUMA architecture
- Power consumption
- Challenges with multithreading on modern processors

None of it will come in your exams (Enjoy!)

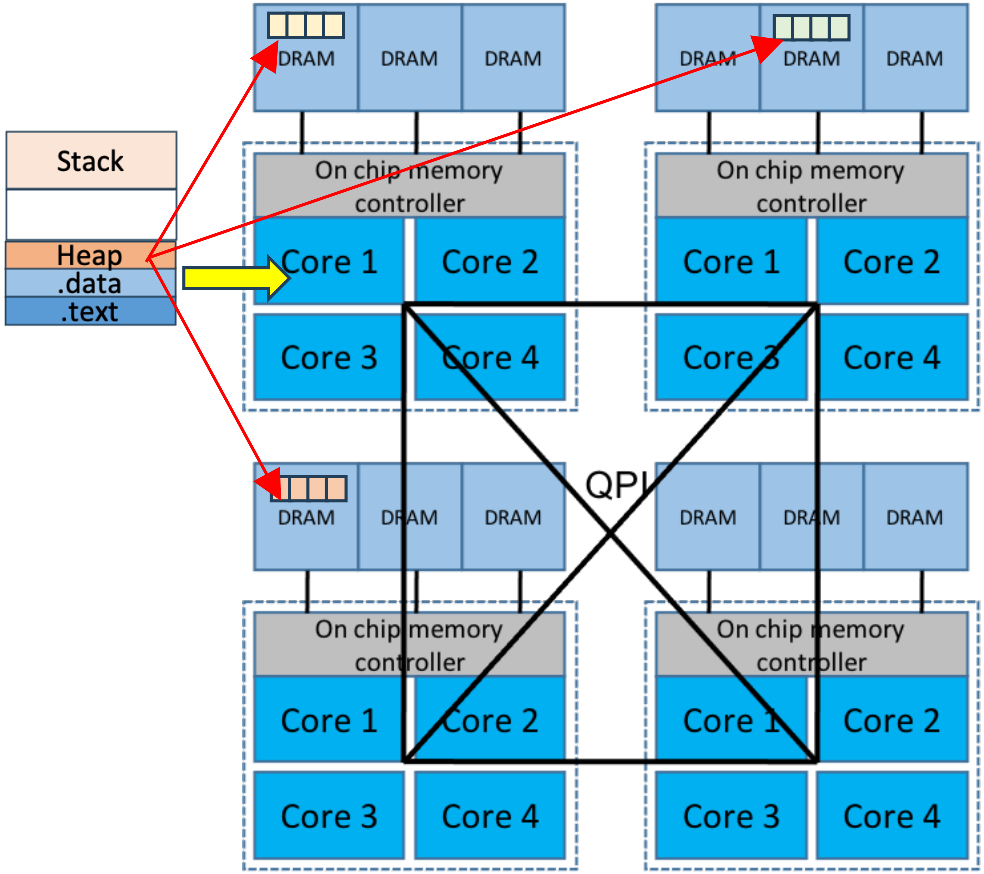
Food For Thought

Physical VS. Logical Cores



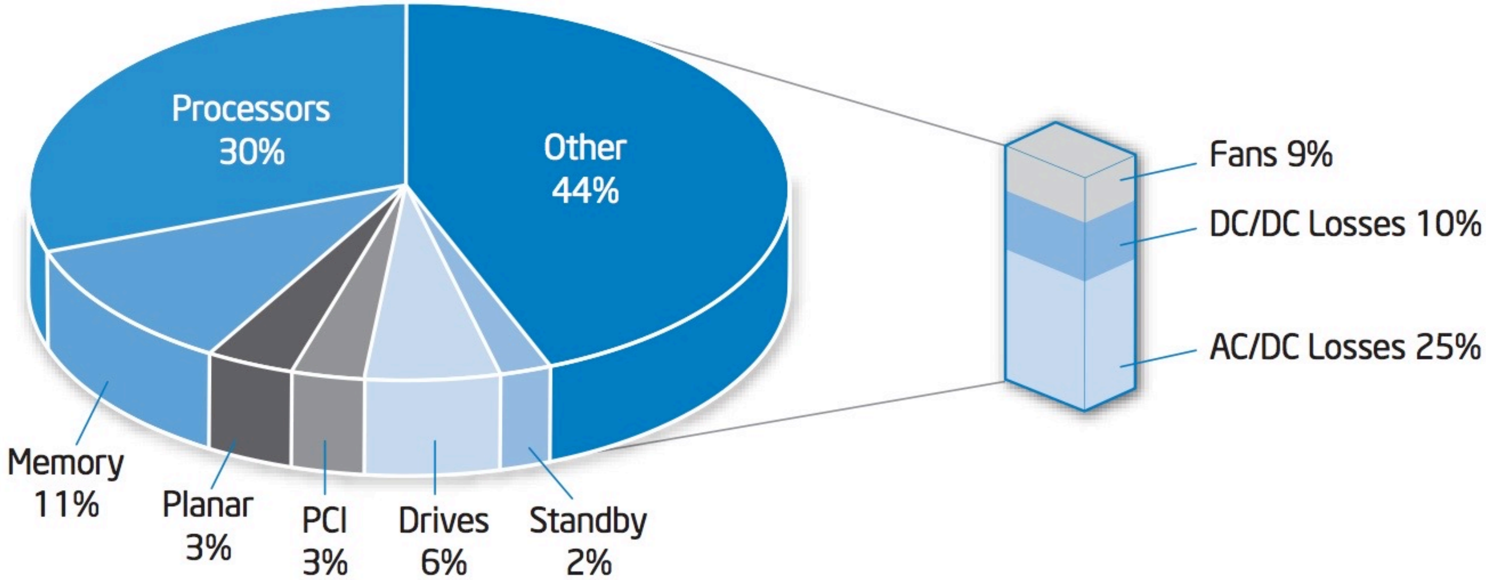
- Architectural state of a core are the registers (EBP, ESP, EIP, etc.)
- Logical cores of a processor share
  - Private cache
  - Execution engine
  - System bus interface
- If the execution of one of the logical core blocks (e.g., Core-0 waiting for a memory fetch from the DRAM) then the other logical core (Core-2) can resume its execution with its own state

Non Uniform Memory Access (NUMA)



- Multiple processors (sockets) on a single motherboard, each with local DRAM(s)
  - Connected together using fast interconnect that also offers cache coherency (e.g., Quick Path Interconnect in Intel)
- One socket can directly access memory of another socket
  - Non-uniform memory access time to local v/s remote memory
- Virtual page (VP) to physical page (PP) mapping matters
  - PP on local DRAM has faster access v/s PP on remote DRAM

Component wise Power Consumption



Modern processors provide several features in userspace for achieving energy efficiency

- As per studies of power consumption in a data center
  - 50% of incoming power is consumed by air-conditioning and power-delivery subsystems, even before reaching the servers in a rack
  - Rest 50% consumed by the servers, which can be further broken down into the various elements as shown above

Source: https://www.intel.com/content/dam/support/us/en/documents/motherboards/server/sb/power_management_of_intel_architecture_servers.pdf

Challenges With Multithreading

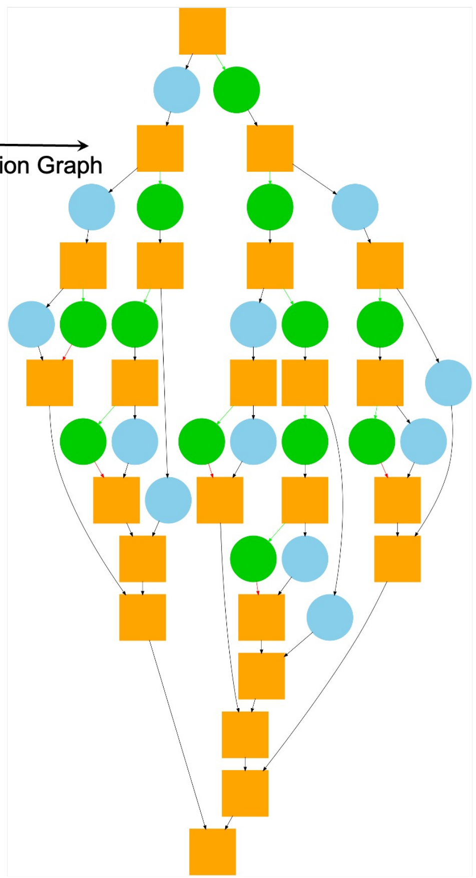
- Challenging in achieving high productivity and high performance in parallel programming over large number of cores
  - Same program should run over a variety of multicore processors without requiring any modifications (high productivity!)
    - One-to-one mapping between thread and core
      - Using logical cores might not benefit in each program
  - Threads should attempt to equally divide the total work (high performance!)
    - If there is not enough parallelism then having one-to-one mapping between thread and core may not benefit (avoiding oversubscribing)
    - Thread running on a core should have most of its data (physical pages) allocated on the local DRAM (avoiding NUMA overheads)
    - Parallel programs should achieve optimal of performance and energy utilization (achieving energy efficiency)

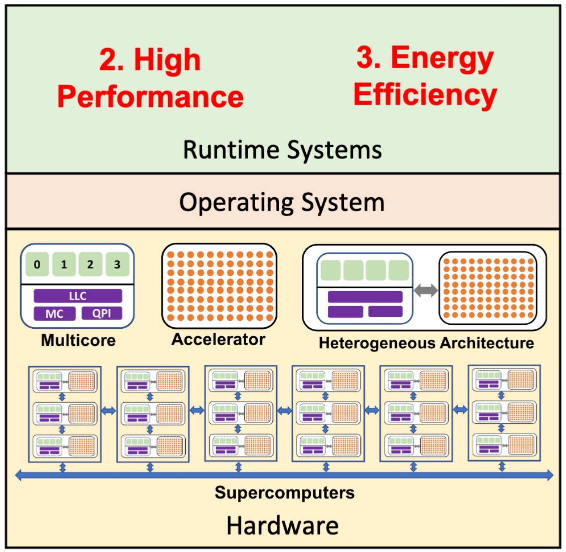
How to Deal With Those Challenges?

```
void recursive(int low, int high) {
  if ((high-low) > threshold) {
    int mid = (high-low) / 2;
    finish {
      async recursive(low, mid);
      recursive(mid, high);
    }
  } else {
    serial_computation(low, high);
  }
}
```

1. High Productivity

Execution Graph





- Parallel programming model
  - Use tasks instead of threads
  - Tasks (async) are composed of a function pointer and the argument to the function
    - Much lightweight than threads
- Parallel runtime systems
  - Performs dynamic load balancing of tasks by mapping tasks to threads based on their current workload
  - Provides several opportunities for achieving high performance and energy efficiency

Next Lecture

- Introduction to Filesystem
- Quiz-4
    - Syllabus: Lectures 18-23