Introduction to Computer Organization

DIS 1A – Week 8

Slides modified from Eric Kim

Agenda

- Synchronization
- Deadlocks
- Thread Safety and Reentrancy
- Lab – Parallel Performance

Q: Synchronization

```
int main() {
  pthread_t tid[N]; int i,
  *ptr;  for (i=0; i<N; i++) {
    ptr = Malloc(sizeof(int)); *ptr =
    i;
    Pthread_create(&tid[i],NULL,fn,ptr)
    ;
  }
  for (i=0; i<N; i++)
    Pthread_join(tid[i], NULL);
  exit(0);
}
```

```
void *fn(void *vargp) {
  int myid = *((int *)vargp);
  Free(vargp);
  printf("%d\n",myid);
  return NULL;
}
```

Question: Are there any race conditions in this code?

Answer: Nope. Careful use of Malloc/Free prevents possible bugs.

Q: Synchronization

```
int main() {
  pthread_t tid[N]; int i,
  *ptr;  for (i=0; i<N; i++) {
    ptr = Malloc(sizeof(int)); *ptr =
    i;
    Pthread_create(&tid[i],NULL,fn,ptr)
    ;
  }
  for (i=0; i<N; i++)
}   Pthread_join(tid[i], NULL);
  exit(0);
```

```
void *fn(void *vargp) {
  int myid = *((int *)vargp);
  Free(vargp);
  process(myid);
  return NULL;
}
```

Question: Outline an approach to avoid race conditions that doesn't use Malloc/Free. What are the advantages/disadvantages of your approach?

Q: Synchronization

```
int main() {
  pthread_t tid[N]; int i, *ptr;
  for (i=0; i<N; i++) {
    Pthread_create(&tid[i],NULL,fn,(void*)i)
    ;
  }
  for (i=0; i<N; i++)
    Pthread_join(tid[i], NULL);
} exit(0);
```

```
void *fn(void *vargp) {
  int myid = (int) vargp;
  process(myid);
  return NULL;
}
```

Answer: Simply pass in the int directly!
Pro: No added overhead due to malloc/free.
Con: Assumes that pointer datatype is at least bigger than size of int. May not be true on all systems.

Intro to Deadlocks

- In order to coordinate multiple threads/processes, there are times in which we must wait in order to ensure a particular ordering.
- This is accomplished via P(&s) (sem_wait), V(&s) (sem_post), pthread_join, wait_pid, etc.
- We are implementing behavior where the execution of a program is unconditionally halted.

Intro to Deadlocks

- Consider the following code:

-
  ```
  void* run_wait(void * arg)
  {
    printf("PEER THREAD RUNNING\n");
  }

  int main()
  {
    pthread_t tid;
    pthread_create(&tid, 0, run_wait, NULL);
    printf("MAIN THREAD RUNNING\n");
  }
  ```

- When I run this, “PEER THREAD RUNNING” is never printed. What's going on?

Intro to Deadlocks

- It seems that given that the main function exits so quickly, the main thread exits before the helper thread can do any of it's work.
- Upon completion it looks like the main thread is killing the helper thread?
- Is this a job for pthread_detach?

Intro to Deadlocks

- Consider the following code:

-
  ```
  void* run_wait(void * arg)
  {
    pthread_detach(pthread_self());
    printf("PEER THREAD RUNNING\n");
  }

  int main()
  {
    pthread_t tid;
    pthread_create(&tid, 0, run_wait, NULL);
    printf("MAIN THREAD RUNNING\n");
  }
  ```

Intro to Deadlocks

- The same race condition exists.
- If the peer thread never got around to printing, it may not get around to detaching.

Intro to Deadlocks

- Since threads are shared among a single processes, ending the process that threads are running on ends all threads.
- If you want to guarantee that all launched threads are completed before executing, you must have a pthread_join.
- This can also be resolved by having the main thread calls pthread_exit(0), which will wait for all peer threads to exit before continuing.

Deadlocks

- Consider the case where we have two shared variables and if we want to access them, we must first sem_wait for a semaphore.
- We have a thread function that reads from the shared variables and prints them out.
- In the mean time, some other thread is responsible for writing to the variables.
- Only one thread should be reading or writing to the variables.

Deadlocks

```
sem_t t;
sem_t u;

char shared_t = 0x74;
char shared_u = 0x75;

void* run_wait(void * arg)
{
  printf("PEER THREAD
RUNNING\n");
  sem_wait(&t);
  sem_wait(&u);
  printf("%d\n", shared_t);
  printf("%d\n", shared_u);
  sem_post(&u);
  sem_post(&t);
}
```

```
int main()
{
  sem_init(&t, 0, 1);
  sem_init(&u, 0, 1);
  pthread_t tid;
  pthread_create(&tid, 0, run_wait,
NULL);
  printf("MAIN THREAD
RUNNING\n");

  sem_wait(&u);
  sem_wait(&t);
  shared_t = 0;
  shared_u  =  1;
  sem_post(&t);
  sem_post(&u);

  void * ret;
  pthread_join(tid, &ret);
}
```

Deadlocks

- This certainly seems to accomplish the goal.
- If you run it, it usually works.
- ...except that it might not.
- Remember that even though there is a certain behavior that we expect when threads are running, we can make no assumptions about it when considering correctness.

Deadlocks

- The main thread:
  - sem_wait(&u) then sem_wait(&t)
- The helper thread:
  - sem_wait(&t) then sem_wait(&u)
- What if the order of instruction executions is as follows:
  - 1. main: sem_wait(&u)
  - 2. helper: sem_wait(&t)
  - 3. Anything else.

Deadlocks

- The main thread will attempt to get a hold of t which the helper currently has. Meanwhile, the helper will try to get a hold of u, which the main thread has.
- Both are blocked indefinitely and neither can continue.
- This is the quintessential deadlock case.

Deadlocks

- The case in which deadlock can occur:
  - There is a cyclic dependency of locks where one thread holds lock A and waits for lock B while another thread holds lock B and waits for lock A.
  - T1 holds L1 and waits for L2, T2 holds L2 and waits for L3, T3 …, TN holds LN and waits for L1.

Deadlocks

- According to the book, a way to guarantee a deadlock free program is to follow the simple rule where for any pair of locks, if there are any threads that acquire both, then they must acquire them in the same order.

Deadlocks

- However, in my previous statement:
  - T1 holds L1 and waits for L2, T2 holds L2 and waits for L3, T3 …, TN holds LN and waits for L1.
- ...that's a deadlock case in which no pair of deadlocks are held by multiple threads.
- The revised rule: define a total ordering for the locks, make sure that when acquiring, the total ordering is followed.
- Thus, in this case, if TN is forced to acquire L1 before LN, deadlock won't occur

Q: Deadlock

```c
int main() {
  sem_t s, t;
  pthread_t tid1, tid2;
  int v1 = 1; int v2 = 2;
  Sem_init(&s, 0, 2);
  Sem_init(&t, 0, 2);
  P(&s); P(&t); P(&t);
  Pthread_create(&tid1, NULL, fn, &v1);
  Pthread_create(&tid2, NULL, fn, &v2);
  while (1);
}
```

```c
void* thread(void* vargp) {
  P(&s);
  V(&s);
  P(&t);
  V(&t);
  printf("HERE: %d\n",
      *((int*)vargp));
  return NULL;
}
```

Question: What are the possible outputs of this program? Explain your answer.

Q: Deadlock

```
int main() {
  sem_t s, t;
  pthread_t tid1, tid2;
  int v1 = 1; int v2 = 2;
  Sem_init(&s, 0, 2);
  Sem_init(&t, 0, 2);
  P(&s); P(&t); P(&t);
  Pthread_create(&tid1, NULL, fn, &v1);
  Pthread_create(&tid2, NULL, fn, &v2);
  while (1);
}
```

```
void* thread(void* vargp) {
  P(&s);
  V(&s);
  P(&t);
  V(&t);
  printf("HERE: %d\n",
      *((int*)vargp));
  return NULL;
}
```

Answer: Nothing - this program will always deadlock!

Q: Deadlock

```
int main() {
  sem_t s, t;
  pthread_t tid1, tid2;
  int v1 = 1; int v2 = 2;
  Sem_init(&s, 0, 2);
  Sem_init(&t, 0, 2);
  P(&s); P(&t);
  Pthread_create(&tid1, NULL, fn, &v1);
  Pthread_create(&tid2, NULL, fn, &v2);
  while (1);
}
```

```
void* thread(void* vargp) {
  P(&s);
  V(&s);
  P(&t);
  V(&t);
  printf("HERE: %d\n",
      *((int*)vargp));
  return NULL;
}
```

Question: Now, what are the possible outputs of the program? Can deadlock still happen?

Q: Deadlock

```
int main() {
  sem_t s, t;
  pthread_t tid1, tid2;
  int v1 = 1; int v2 = 2;
  Sem_init(&s, 0, 2);
  Sem_init(&t, 0, 2);
  P(&s); P(&t);
  Pthread_create(&tid1, NULL, fn, &v1);
  Pthread_create(&tid2, NULL, fn, &v2);
  while (1);
}
```

```
void* thread(void* vargp) {
  P(&s);
  V(&s);
  P(&t);
  V(&t);
  printf("HERE: %d\n",
      *((int*)vargp));
  return NULL;
}
```

Answer: Either "Here: 1" -> "Here: 2", or vice-versa. Dead lock can't happen anymore.

Q: Deadlock

```
Thread 1:
   P(&s)
   P(&t)
   do_work();
   V(&t)
   V(&s)
```

```
Thread 2:
   P(&t)
   P(&s)
   do_work();
   V(&s)
   V(&t);
```

```
sem_t t;    // N = 1
sem_t s;    // N = 1
```

Will this always deadlock? Sometimes deadlock? Never deadlock? Show execution order for possible cases.

Q: Deadlock

```
Thread 1:
   P(&s)
   P(&t)
   do_work();
   V(&t)
   V(&s)
```

```
Thread 2:
   P(&t)
   P(&s)
   do_work();
   V(&s)
   V(&t);
```

```
sem_t t;    // N = 1
sem_t s;    // N = 1
```

Deadlock:

```
T1          T2
P(&s)
            P(&t)
            P(&s)
P(&t)
   T1,T2 stuck!
```

OK:

```
T1          T2
P(&s)
P(&t)
do_work()
V(&t)
V(&s)
            P(&t)
            P(&s)
            ...
```

Q: Deadlock

```
Thread 1:
    P(&t)
    P(&s)
    do_work();
    V(&s)
```

```
Thread 2:
    P(&t)
    P(&s)
    do_work();
    V(&s)
    V(&t);
```

```
sem_t t;     // N = 1
sem_t s;     // N = 1
```

Will this always deadlock? Sometimes deadlock? Never deadlock? Show execution order for possible cases.

Q: Deadlock

Thread 1:
```
    P(&t)
    P(&s)
    do_work();
    V(&s)
```

Thread 2:
```
    P(&t)
    P(&s)
    do_work();
    V(&s)
    V(&t);
```

```
sem_t t;    // N = 1
sem_t s;    // N = 1
```

OK:
```
T1        T2
        P(&t)
        P(&s)
        do_work()
        V(&s)
        V(&t)
P(&t)
...
```

Deadlock:
```
T1              T2
P(&t)
P(&s)
do_work()
V(&s)
                P(&t)
        T2 is stuck!
```

Thread Safety

- The book defines thread safety quite thoroughly (if a little confusingly).
- A function is “thread-safe” if it will be correct even if called repeatedly by multiple threads.
- Functions that are thread-unsafe fall in one of four categories:
- Class 1: Functions that don't protect shared variables.

Thread Safety

- Class 2: Functions that keep state across multiple invocations (functions whose current result depends on previous invocations)
- Class 3: Functions that return pointers to static variables.
- Class 4: Functions that call class 2 thread unsafe functions and functions that call class 1 and 3 thread unsafe functions and don't protect the function calls (with synchronization).

Thread-safe functions

Typically achieve thread-safety by mechanisms:

- Synchronization (ie semaphores/mutexes)
- Careful handling of shared data

Thread Safety

- Aside: What does the static keyword mean?
- If applied to a global variable this means that this variable is visible only to the module in which it's located.
- Ex. if I have a project that includes main.c and blah.c and blah.c defines a static global variable called “foo”, that instance of foo is not visible to main.c

Thread Safety

- If applied to a local variable, static means across all threads and invocations of that function, there is only one instance of that variable.
- Consider:

```
int foo()
{
  static int i = 0;
  i += 1;
}
```

Thread Safety

```
int foo()
{
  static int i = 0;
  i += 1;
}
```

- The first thread to call this will initialize int i. From then on, all calls to foo will refer to this singular int.
- If thread 1 calls foo, i will be incremented. If thread 2 calls foo, it will find that i = 1 and it will increment it.

Thread Safety

- Which one of these “static” variables is the one that is referred to by this “thread unsafe” case 3?

Thread Safety

- Which one of these “static” variables is the one that is referred to by this “thread unsafe” case 3?
- Probably both right? Static globals will be shared among threads in a module (this has nothing to do with the static keyword) and static locals will be shared among threads calling the function.

Thread Safety

- Consider the ctime function which converts a time to a string:
- char * ctime(const time_t * timer)
- time_t is a data type that is essentially (but not quite) a number that corresponds to the time since the epoch.
- The return value is a C-string that is in a readable format.

Thread Safety

- The problem is that the pointer that ctime returns is a static pointer to a special location.
- Therefore, this is class 3 thread-unsafe.
- Ex. if two threads call ctime in quick succession, both will modify the data that the pointer points to. If the second call to ctime modifies the pointer before the first call can read from it, then the first call to ctime will return the same string as the second call.

Reentrancy

- Reentrancy is defined as a function that doesn't use any shared variable at all.

- This is an incredibly simple definition that tells us exactly nothing about what “reentrancy” really means.

Reentrancy

- Reentrancy is a concept that predates multi-threading.
- Essentially, a re-entrant function is one that can be interrupted by a signal and then re-entered safely... *all from within the same thread*.
- By safely, we mean that the result will be correct in terms of value and in terms of execution behavior.
- How can this “re-entering” behavior happen?

Interrupts

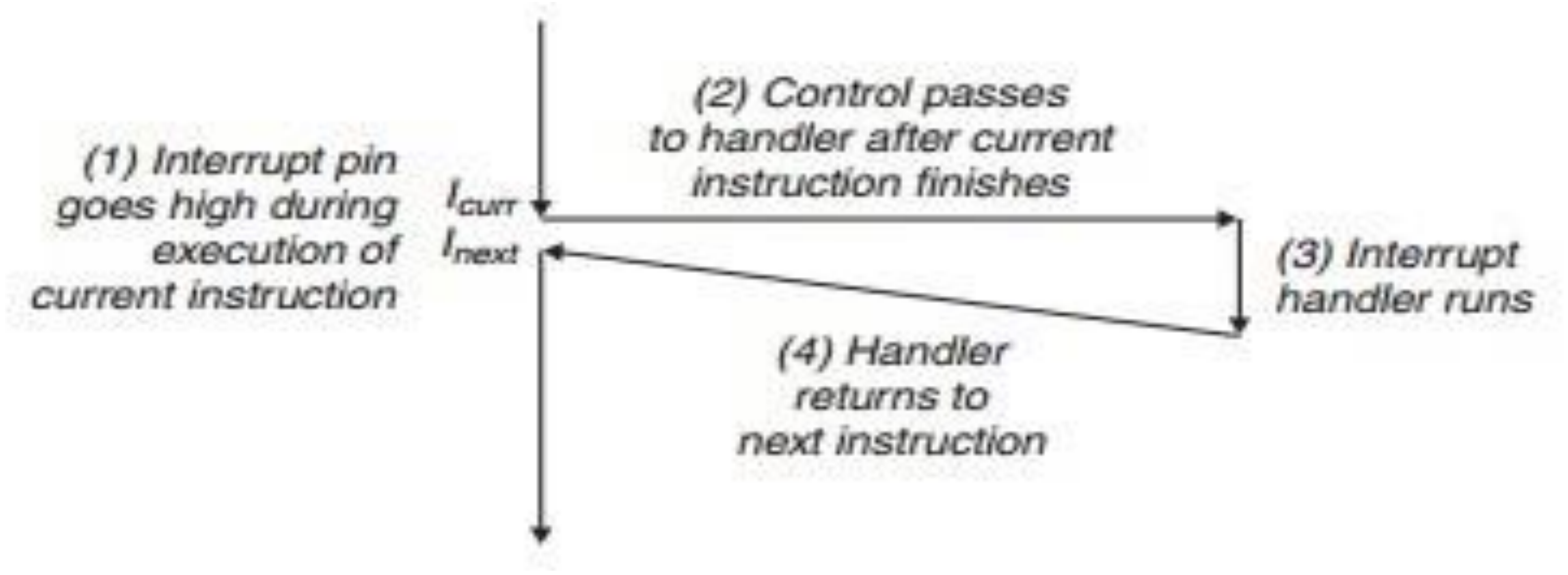
- Sometimes when your program is running, it has to be able to respond to outside stimuli.
- Most commonly signals from I/O devices.
  - Keyboard key presses.
  - Mouse movement
  - Network adapter activity
- These signals will be sent to running programs.
- Asynchronous
  - Occurs independently of currently executing program

Interrupt Handling

- I/O device triggers the “interrupt pin”
- After current instruction, stop executing current thread of instructions and control switches to interrupt handler.

Interrupt Handling

- Interrupt handler handles interrupt.
- Control is given back to previously executing thread of instructions.
- Previous program executes the next instruction.


(1) Interrupt pin goes high during execution of current instruction
I_{curr}
I_{next}
(2) Control passes to handler after current instruction finishes
(3) Interrupt handler runs
(4) Handler returns to next instruction

Reentrancy

- Thus, if a program is running a particular thread and that thread is in a function, it is possible that an interrupt causes the thread's execution to be switched to different code to handle the interrupt.
- If the interrupt handler calls the same function that you were in when you were interrupted, you better hope that function was reentrant.
- Consider ctime again.

Reentrancy

- Pretend that ctime has some code that looks like this (char* ptr is shared):

```
…
  strcpy(ptr, local_ptr); // Copies the string from
  local_val = 10;         //  local_ptr to ptr
  return ptr;
}
```

Reentrancy

- Say in our thread, we have just finished executing the strcpy

```
…
  strcpy(ptr, local_ptr);
  local_var = 10;        ← Current, ptr = “current”
  return ptr;
}
```

- Now, a signal is received and the interrupt handler causes the thread to run the interrupt handler (with the intent of returning to the “Current” line once the interrupt is handled)

Reentrancy

- This thread should be returning ptr which contains the string we just copied “current”.
- What if the interrupt handler calls ctime?

Reentrancy

- …
  ```
    strcpy(ptr, local_ptr); ← Interrupt Handler
    local_var = 10;         ← Current, ptr = “current”
    return ptr;
  }
  ```

- Say the interrupt handler calls ctime again except now local_ptr = “interrupt”.
- The interrupt handler copies local_ptr to ptr, returns and completes.

Reentrancy

- …
  strcpy(ptr, local_ptr);
  local_var = 10;   ← Current, ptr = “interrupt”
  return ptr; ← Interrupt Handler
  }

- When the interrupt handler completes, the thread goes back to the instruction that it would have executed before the interrupt.

Reentrancy

- …
  ```
    strcpy(ptr, local_ptr);
    local_var = 10;        ← Current, ptr = “interrupt”
    return ptr;
  }
  ```
- Now the result is wrong.

Reentrancy

- The solution for class 3 thread-unsafe functions is as follows:

```
char *ctime_ts(const time_t *timep, char *privatep)
{
    char *sharedp;

    sem_wait(&mutex);
    sharedp = ctime(timep);
    strcpy(privatep, sharedp); /* Copy string from shared to private */
    sem_post(&mutex);
    return privatep;
}
```

Reentrancy

- This wraps the call to ctime in a lock that means only one thread can access a call to ctime.
- The locked critical section will copy the string pointed to by the static pointer of ctime and copy it into a local non-shared pointer.
- Thus, one thread's call to ctime_ts cannot be affected by another thread's call to ctime_ts.

Reentrancy

- However, as Practice Problem suggests, this is non-reentrant. Why?
- The book's answer:
  - “The ctime_ts function is not reentrant because each invocation shares the same static variable returned by the ctime function. “
  - ...and the award for least helpful answer goes to...

Reentrancy

- Exactly why is it such a problem if a single thread is interrupted and ctime_ts is called again?
- Consider the following flow of execution:

Reentrancy

```
char *ctime_ts(const time_t *timep, char *privatep)
{
    char *sharedp;

    P(&mutex);  ← Current
    sharedp = ctime(timep);
    strcpy(privatep, sharedp);
    V(&mutex);
    return privatep;
}
```

Reentrancy

```
char *ctime_ts(const time_t *timep, char *privatep)
{
    char *sharedp;

    P(&mutex);
    sharedp = ctime(timep); ← Current, mutex = 0
    strcpy(privatep, sharedp);
    V(&mutex);
    return privatep;
}
```

- INTERRUPT!
- Interrupt calls ctime_ts.

Reentrancy

```
char *ctime_ts(const time_t *timep, char *privatep)
{
    char *sharedp; ← Interrupt

    P(&mutex);
    sharedp = ctime(timep); ← Current, mutex = 0
    strcpy(privatep, sharedp);
    V(&mutex);
    return privatep;
}
```

Reentrancy

```
char *ctime_ts(const time_t *timep, char *privatep)
{
    char *sharedp;

    P(&mutex);  ← Interrupt
    sharedp = ctime(timep); ← Current, mutex = 0
    strcpy(privatep, sharedp);
    V(&mutex);
    return privatep;
}
```

- When the interrupt reaches P, it must wait until the mutex is released.

Reentrancy

- But it will never be released. This isn't a multithreaded context in which context can switch to the original execution.
- This thread IS the original thread that acquired the lock.
- This is a case of deadlock caused by one thread waiting on itself.

Example: sum

```
int result = 0;
void sum_n(int n) {
  if (n == 0) {
    result = n;
  } else {
    sum_n(n-1);
    result = result + n;
  }
}
```

Suppose Kim tries to make this code thread safe...

Example: fib

Question: Is there anything wrong with this code?

```
int result = 0;
sem_t s; // sem_init(&s,1);
void sum_n(int n) {
  if (n == 0) {
    P(&s); result = n; V(&s);
  } else {
    P(&s);
    sum_n(n-1);
    result = result + n;
    V(&s);
  }
}
```

Answer: Yes, deadlock! sum_n(5) calls sum_n(4), but sum_n(4) can't acquire mutex. sum_n(5) can't make progress without sum_n(4) - thread is stuck.

Ex: strtoupper

```
/* non-reentrant function */
char *strtoupper(char *string) {
  static char buffer[MAX_STRING_SIZE];
  int index;
  for (index = 0; string[index]; index++)
    buffer[index] = toupper(string[index]);
  buffer[index] = 0
  return buffer;
}
```

Question: Is this threadsafe?

Answer: Nope! Two threads running strtoupper() will write to shared buffer.

Ex: strtoupper

```c
/* reentrant function (a poor solution) */
char *strtoupper(char *string) {
  char *buffer;
  int index;
  /* error-checking should be performed! */
  buffer = malloc(MAX_STRING_SIZE);
  for (index = 0; string[index]; index++)
    buffer[index] = toupper(string[index]);
  buffer[index] = 0
  return buffer;
}
```

Ex: strtoupper

```
/* reentrant function (a better solution) */
char *strtoupper_r(char *in_str, char *out_str) {
  int index;
  for (index = 0; in_str[index]; index++)
    out_str[index] = toupper(in_str[index]);
  out_str[index] = 0
  return out_str;
}
```

Reentrancy vs Thread Safety

Question: Are threadsafe functions always reentrant?

```
void f() {
  mutex_acquire();
  // suppose signal handler gets invoked here!
  do_important_stuff();
  mutex_release();
}
```

Answer: Nope! Suppose function f() is used as a signal handler. Suppose we are executing f(), and acquire the mutex. Then, suppose signal handler gets invoked again, and we invoke f() again. The signal handler will get stuck trying to acquire the mutex!

Reentrancy vs Thread Safety

Question: Are reentrant functions always threadsafe?

Answer: According to your textbook, yes. This is using the definition that reentrant functions never access shared data.

Supplemental Readings

Helpful reading on threads:

https://randu.org/tutorials/threads/

Helpful reading on thread safety and reentrancy:

https://www-01.ibm.com/support/knowledgecenter/ssw_aix_61/com.ibm.aix.genprogc/writing_reentrant_thread_safe_code.htm?cp=ssw_aix_61%2F13-3-12-18

.

Lab

- Loop unrolling
- Function inlining/Macros
- Cache locality (loop-reordering)
- Creating multithreaded program