# PS3 Programming

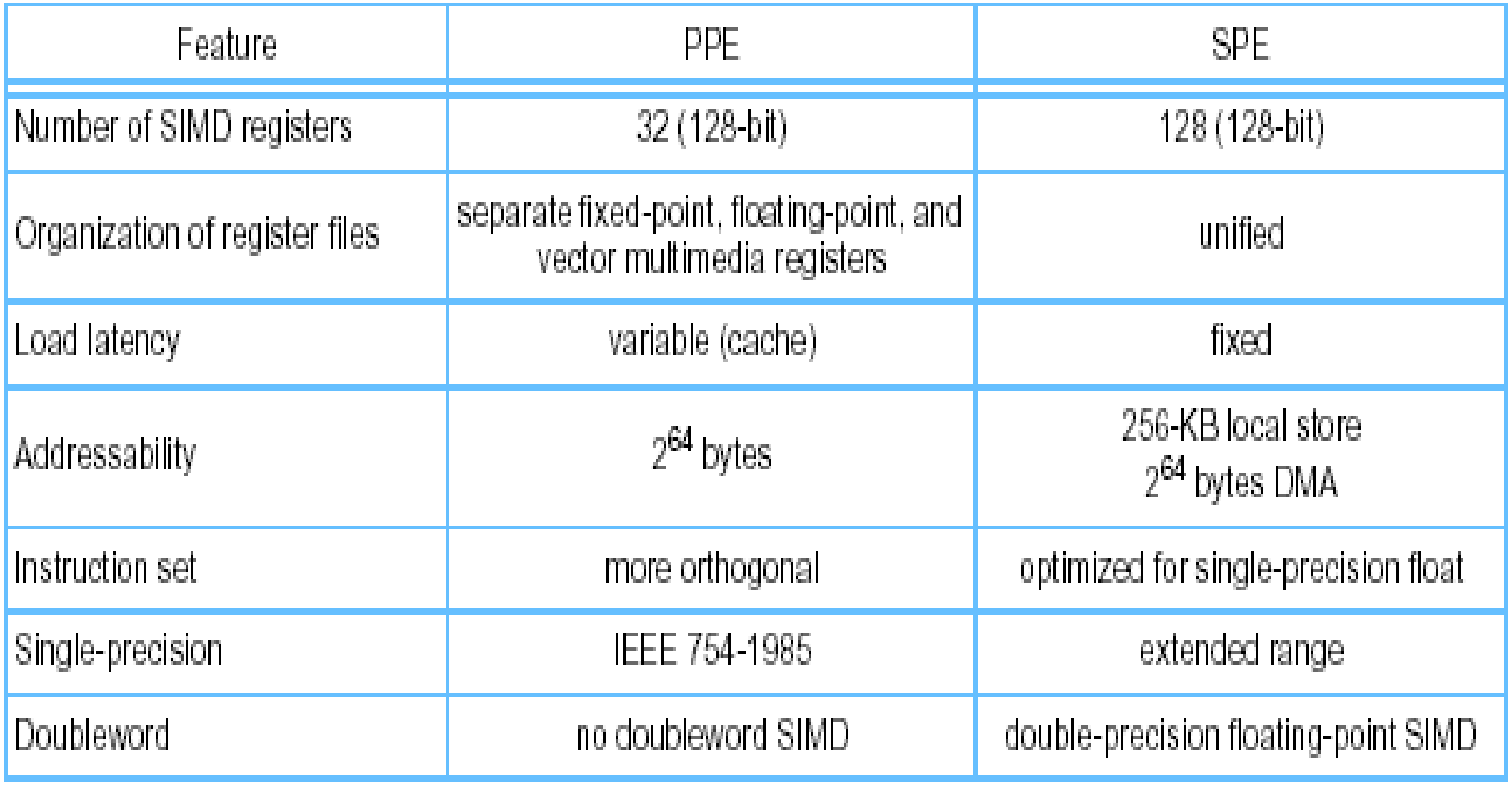
Week 2. PPE and SPE

The SPE runtime management library (libspe)

# Outline

- Overview
- Hello World
- Pthread version
- Data transfer and DMA
- Homework

# PPE/SPE Architectural Differences

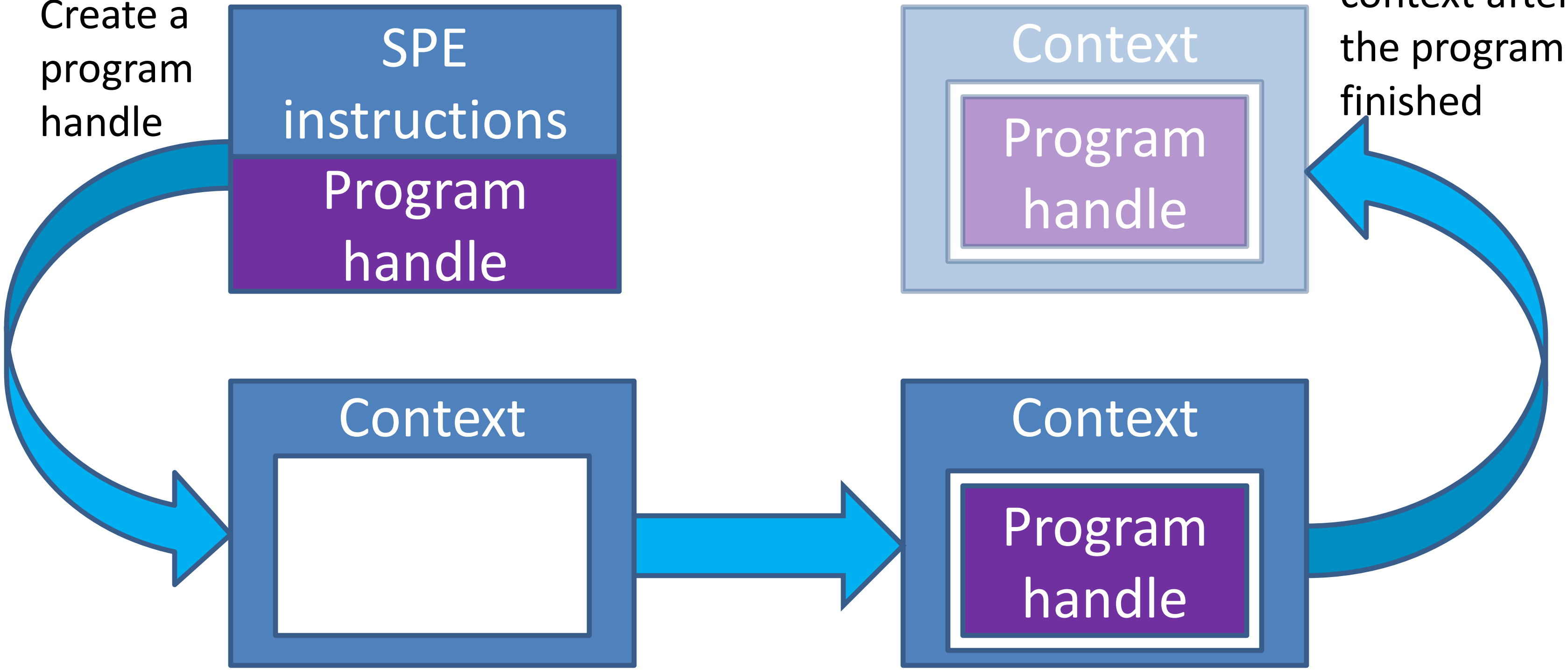


| Feature | PPE | SPE |
|---|---|---|
| Number of SIMD registers | 32 (128-bit) | 128 (128-bit) |
| Organization of register files | separate fixed-point, floating-point, and vector multimedia registers | unified |
| Load latency | variable (cache) | fixed |
| Addressability | $2^{64}$ bytes | 256-KB local store<br>$2^{64}$ bytes DMA |
| Instruction set | more orthogonal | optimized for single-precision float |
| Single-precision | IEEE 754-1985 | extended range |
| Doubleword | no doubleword SIMD | double-precision floating-point SIMD |

# The big picture

- SPE does not have OS to manage its resources. PPE creates a temp one, called a *context*, for it.
- Context: a data structure contains fields that access
  - the SPE’s processing unit
  - SPE’s memory
  - SPE’s communication resource
- Every context must be loaded with SPE instructions, represented by a *program handle*.

# Flow of SPE context



Create a program handle

SPE instructions

Program handle

Create context and load the program into the context

Context

Execute the program inside the context

Context

Program handle

Deallocate the context after the program is finished

Context

Program handle

# HELLO WORLD

# spu_basic.c

```
#include <stdio.h>
int main (unsigned long long spe_id,
          unsigned long long argp,
          unsigned long long envp) {
   printf("Hello World! My thread id is %lld\n“,
          spe_id);
   return 0;
}
```

- spe_id: Identifies of the SPE execution thread
- argp: Data send by the PPU to the SPE
- envp: Environmental data passed to the SPE

# ppu_basic.c

```
#include <stdio.h>
#include <stdlib.h>
#include <libspe2.h>

/* SPE program handle */
extern spe_program_handle_t spu_basic;

int main(int argc, char **argv) {
  spe_context_ptr_t spe;          /* SPE context */
  unsigned int entry_point;       /* SPE start address */
  int  retval;                    /* Return value */
  spe_stop_info_t stop_info;      /* Stop information */
```

```
/* Create the SPE Context */
  spe = spe_context_create(0, NULL);
  if (!spe) {
    perror("spe_context_create");
    exit(1);
  }

  /* Load the program handle into the context */
  retval = spe_program_load(spe, &spu_basic);
  if (retval) {
    perror("spe_program_load");
    exit(1);
  }
```

```
/* Run the program inside the context */
  entry_point = SPE_DEFAULT_ENTRY;
  retval = spe_context_run(spe, &entry_point, 0,
                  NULL, NULL, &stop_info);
  if (retval < 0) {
    perror("spe_context_run");
    exit(1);
  }

  /* Deallocate the context */
  retval = spe_context_destroy(spe);
  if (retval) {
    perror("spe_context_destroy");
    exit(1);
  }
  return 0;
}
```

# Create a context

```
spe_context_ptr_t spe_context_create
                  (unsigned int flags,
                  spe_gang_context_ptr_t gang)
```

  - flags: control context's behaviors and communication
  - gang: a collection of contexts for multiple SPEs
- It creates a file system for SPU (SPUFS)
  - Expensive: about 400ms to create
  - Allow programmer access SPU's resource using file comannds

# Load SPE's program

```
int spe_program_load
   (spe_context_ptr_t spe,
  spe_program_handle_t *program)
```

- – Program handle: pointer to the function defined in

```
  extern spe_program_handle_t spu_basic;
```

- This is compile-time program embedding.
- You can embed program at runtime by using `spe_program_handle_t spe_image_open(const char *filename)` and `spe_image_close`.

# Run program

```
int spe_context_run(spe_context_ptr_t spe,
                    unsigned int *entry,
                    unsigned int *runflags,
                    void *argp, void *envp,
                    spe_stop_info_t *stopinfo)
```

- entry: contains an address in SPE’s local store, telling where to start the program.
  - SPE_DEFAULT_ENTRY will start a default address
  - On exit, entry holds the address of the last executed instruction

# spe_context_run

- runflags: control SPE’s execution
  - SPE_RUN_USER_REGS and SPE_NO_CALL_BACK:
  - Set it 0 if you don’t know what the flags means
- argp, envp: You can use them to pass the addresses of the data (more on DMA later)
- stopinfo: use `int spe_stop_info_read (spe_context_ptr_t ctx, spe_stop_into_t stopinfo)` to read it

# Compilation

- Three steps

```
spu_gcc spu_basic.c –o spu_basic
```

  - Build SPU executable from .c file

```
ppu-embedspu –m64 spu_basic spu_basic.o
```

  - Convert the SPU executable into a PPU .o file

```
ppu_gcc –o ppu_basic ppu_basic.c
  spu_basic.o –lspe2
```

  - Build a PPU executable

- Run `./ppu_basic`

# PTHREAD VERSION

# Problems of previous code

- PPU is waiting the finish of `spe_context_run`
- Only one SPE can be invoke at a time

➔No parallelism

- Solution: create multithread code.
  - The only changes are on the PPU side.

# ppu_threads.c

```
#include <stdio.h>
#include <stdlib.h>
#include <libspe2.h>
#include <pthread.h>

/* The data sent to the pthread */
typedef struct ppu_pthread_data {
    spe_context_ptr_t speid;
    pthread_t pthread;
    void* argp;
} ppu_pthread_data_t;
```

It is a convention to define this structure, not necessary to do so to write a pthread code

# Define the pthread function

```
/* The function executed in the pthread */
void *ppu_pthread_function(void *arg) {
  ppu_pthread_data_t *data =
                    (ppu_pthread_data_t *)arg;
  int retval;
  unsigned int entry = SPE_DEFAULT_ENTRY;
  if ((retval = spe_context_run(data->speid,
    &entry, 0, data->argp, NULL, NULL)) < 0) {
    perror("spe_context_run");
    exit (1);
  }
  pthread_exit(NULL);
}
```

Need to define this for pthread void *arg

Call spe_context_run inside this function.

# Get the number of useable SPU

```
/* SPU program handle */
extern spe_program_handle_t spu_basic;
ppu_pthread_data_t data[16];

int main(int argc, char **argv) {
   int i, retval, spus;
   /* Determine number of available SPUs */
   spus = spe_cpu_info_get(SPE_COUNT_USABLE_SPES, 0);
```

# The context/pthread creation (next slide)

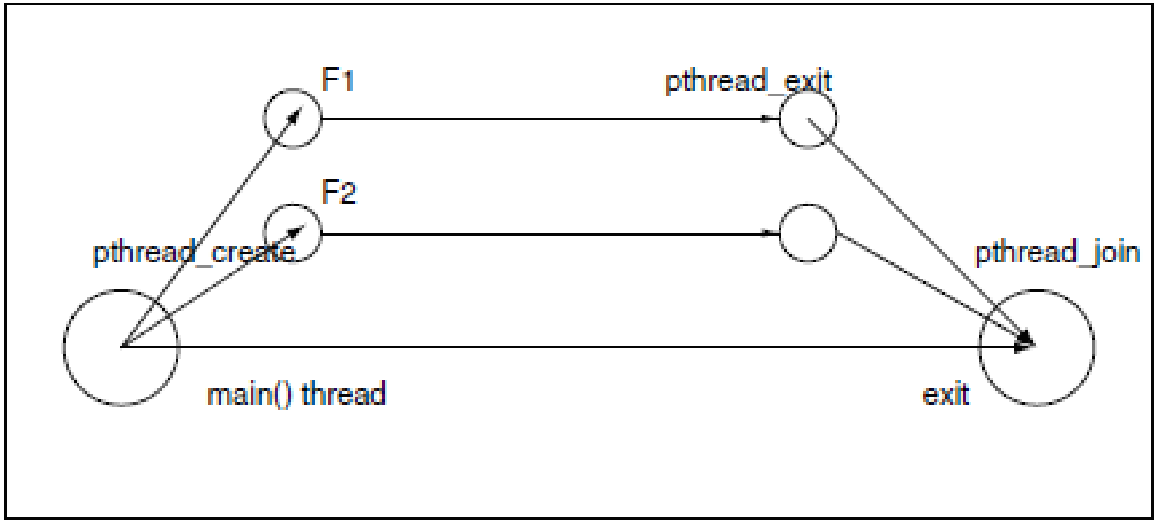
```
 /* Create a context and thread for each SPU */
for (i=0; i<spus; i++) {       /* Create context */
  if ((data[i].speid = spe_context_create (0, NULL)) == NULL) {
    perror("spe_context_create");       exit(1);
  }

   /* Load program into the context */
  if ((retval=spe_program_load(data[i].speid, &spu_threads)) != 0) {
    perror("spe_program_load");       exit (1);
  }

   /* Create thread */
  if ((retval = pthread_create(&data[i].pthread,
    NULL, &ppu_pthread_function, &data[i])) != 0) {
    perror("pthread_create");        exit (1);
  }
}
```

# Finalization

```
/* Wait for the threads to finish processing */
  for (i = 0; i < spus; i++) {
    if ((retval = pthread_join(data[i].pthread, NULL)) != 0) {
       perror("pthread_join");          exit (1);
    }
    if ((retval = spe_context_destroy (data[i].speid)) != 0) {
       perror("spe_context_destroy");          exit (1);
    }
  }
  return 0;
}
```

# pthread_create and pthread_join

```
int pthread_create(pthread_t *tid, const
  pthread_attr_t *attr, void *program, void
  *arg);
int pthread_join(pthread_t thread, void
  **value_ptr);
```



# DATA TRANSFER

# Use argp for data transfer

- The code in the PPU’s side

```
unsigned long long address=0;
retval = spe_context_run(spe, &entry_point,
0,(void*)address, NULL, &stop_info);
```

- The code in the SPE’s side

```
#include <stdio.h>
int main (unsigned long long spe_id,
          unsigned long long argp,
          unsigned long long envp) {
   printf("argp is %lld\n", argp);
   return 0;
}
```

# MFC

- SPUs use MFC to get/set data
  - mfc_get: get data into the local store
  - mfc_put: send data out of the local store
- Both of them have the same argument list
  - volatile void *ls: LS address
  - unsigned long long ea: external address
  - unsigned int size: number of bytes to transffer
  - unsigned int tag: value to identify transfer
  - unsigned int tid: transfer class
  - unsigned int rid: L2 replacement policy

# Example

- PPU’s code

```
/* The array to be DMAed into the SPU's LS */
unsigned int ch_array[SIZE] __attribute__ ((aligned (128)));
…
spe_context_run(spe, &entry, 0, ch_array, NULL, &stop_info);
```

- SPU’s code

```
int main(unsigned long long speid,
        unsigned long long argp,  unsigned long long envp) {
  int i, j;
  vector unsigned int buff[SIZE]  __attribute__ ((aligned(128)));
```

# SPU’s code continue

```
/* Read unprocessed data from main memory */
 mfc_get(buff, argp,  sizeof(buff), TAG, 0, 0);
 mfc_write_tag_mask(1<<TAG);
 mfc_read_tag_status_all();

 /* Process the data */
 ….

 /* Write the processed data to main memory */
 mfc_put(buff, argp,  sizeof(buff), TAG, 0, 0);
 mfc_write_tag_mask(1<<TAG);
 mfc_read_tag_status_all();
```

# Check DMA completion

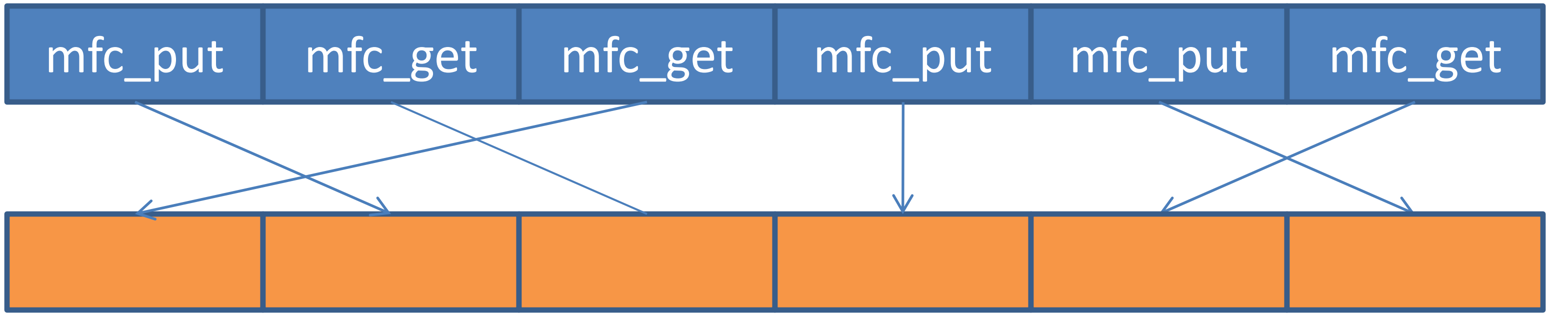
- mfc_get/mcf_put are asynchronized calls. Their completion can be checked by calling
  - `mfc_read_tag_status_immediate`
  - `mfc_read_tag_status_any`
  - `mfc_read_tag_status_all`
- Before calling them, you need to set tag_mask
- Tag mask is set by mfc_write_tag_mask
- DMA has 32 tag group: 0-31 (just like network's package id). The mask is 32 bit long.

# Limitation of MFC/DMA

- The data size need be 1, 2, 4, 8, 16, 16x
- Data aligned at 128-byte makes transfer most efficient
  - Smaller alignment is ok, but need to agree with data size
- A max DMA transfer is 16K byte
- MFC has a queue to hold mfc commands, but is limited to 16 commands
  - Use mfc_stat_cmd_queue to check the number of open slots

# MFC “queue” scheduling

- Requests in the MFC queue do not guarantee to be executed in order (not FIFO)

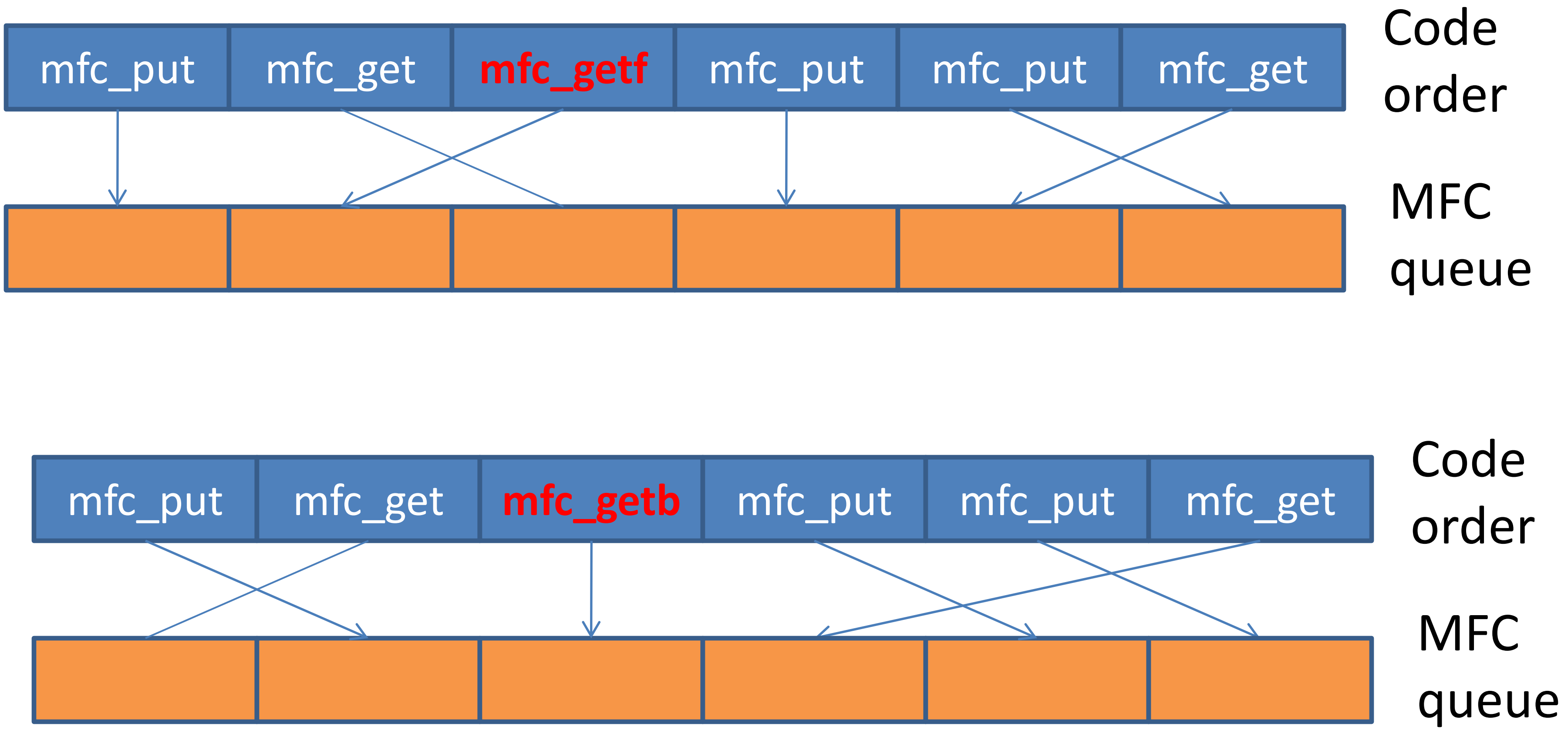


Code order

MFC queue

- Use fences command: mfc_getf, mfc_putf
  - All commands before it will be executed first.
- And barrier commands: mfc_getb, mfc_putb
  - All commands before/after it will be executed first/later

# Example of fences/barrier command



mfc_put | mfc_get | **mfc_getf** | mfc_put | mfc_put | mfc_get

Code order

MFC queue

mfc_put | mfc_get | **mfc_getb** | mfc_put | mfc_put | mfc_get

Code order

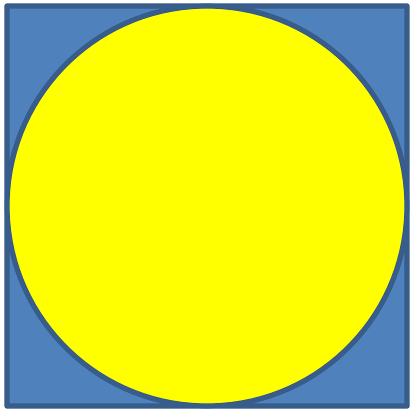
MFC queue

# HOMEWORK

# Monte Carlo for computing PI

- The area of a unit circle is π. The area of the square bounding the unit circle is 4.
- If we randomly throw darts to that board, the probability of the darts inside the circle is π/4.
- Sequential code

```
count=0;
for ( i=0; i<niter; i++) {
        x = (double)rand()/RAND_MAX;
        y = (double)rand()/RAND_MAX;
        if (x*x+y*y<=1) count++; }
pi=(double) count/niter*4;
```



# Reading assignment

- SPU invocation and P-thread approach is in chap 7
- DMA is in chap 12.1-12.3
- Random number library is in chap 18.2 (libmc_rand)
  - Just use mc_rand_mt_init and mc_rand_mt_minus1_to_1_d2 should work