MEC302: Embedded Computer Systems

Theme II: Design of Embedded Computer Systems

Lecture 8 – Multitasking

Dr. Timur Saifutdinov

Assistant Professor at EEE, SAT

Email: Timur.Saifutdinov@xjtlu.edu.cn

Outline

- Multitasking
  - What is multitasking;
  - Multitasking mechanisms;
- Threads
  - Creating and handling threads;
  - Memory consistency models of threads;
  - Mutual exclusions
- Processes
  - Creating processes;
  - Communication between processes;
  - Example of files handling

What is multitasking

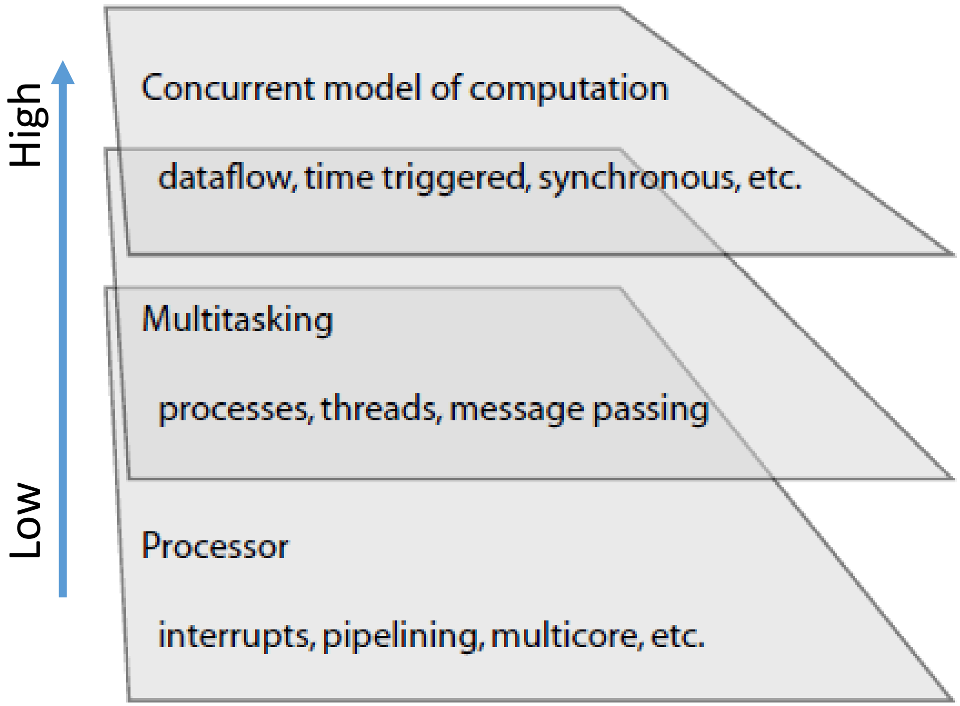
Multitasking – software mechanisms that provides concurrent execution of sequential code (i.e., simultaneous execution of multiple tasks).

Reasons for concurrency (i.e., **advantages**):

- Increase responsiveness (reduce latency) to external signal (stimuli);
- Improve performance when running on multiple processors or cores;
- Control timing of external interactions (e.g., update display while executing other tasks).

Multitasking bridges the high-level of abstract computational and dataflow models and low-level hardware mechanisms.

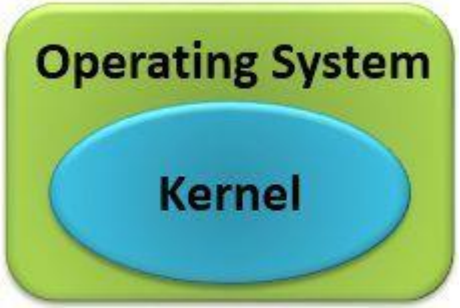
Layers of abstraction for concurrency in programs:



Multitasking mechanisms

Multitasking is provided by an **Operating System (OS)** through:

- Library of procedures;
- (Micro)**kernel**.



OS is a software that manages computer hardware and software resources to provide interface between user and computer.

OS can differ by application:

- **General-purpose**: MS Windows, Mac OS X, Linux;
- **Real-time OS (RTOS)** for embedded applications: WinCE, QNX, UNIX-RT;
- **Mobile OS** for handheld devices: Symbian OS, iOS, Android.

The core of an OS is **kernel**, which controls all programs running on the computer:

- Converts user commands to machine instructions;
- Allocate memory for processes;
- Communicate with peripherals and networks;
- Decide on the order of processes execution (i.e., **scheduler** for multiple tasks).

Multitasking mechanisms

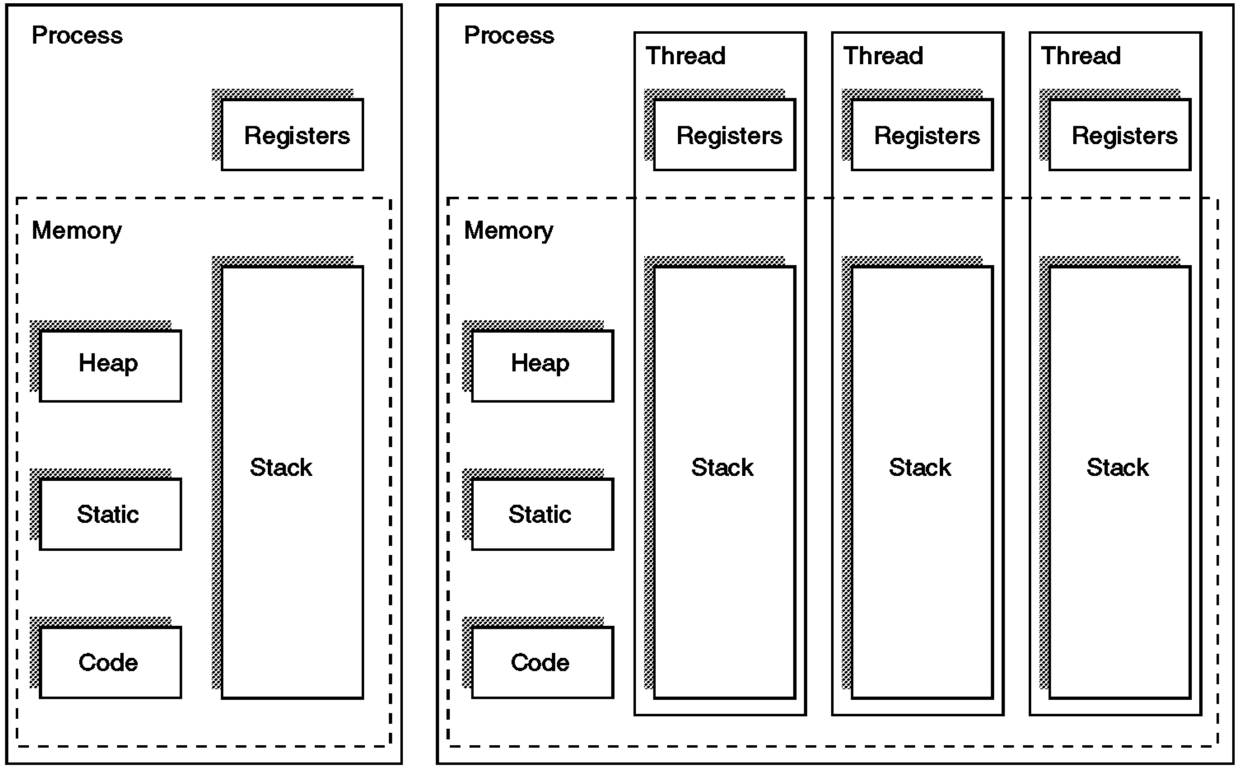
Multitasking mechanisms provided by an **OS** include:

- **Threads** are imperative programs that run concurrently and share a memory space. They can:
  - Directly access each others' variables.
- **Processes** are imperative programs with their own memory spaces. Communication is done via:
  - File system – exchange data through files;
  - Message passing – controlled blocks of memory

#Memory allocation for processes and threads [2]:

Process 1
Thread 1

Process 2
Thread2 Thread3 Thread4

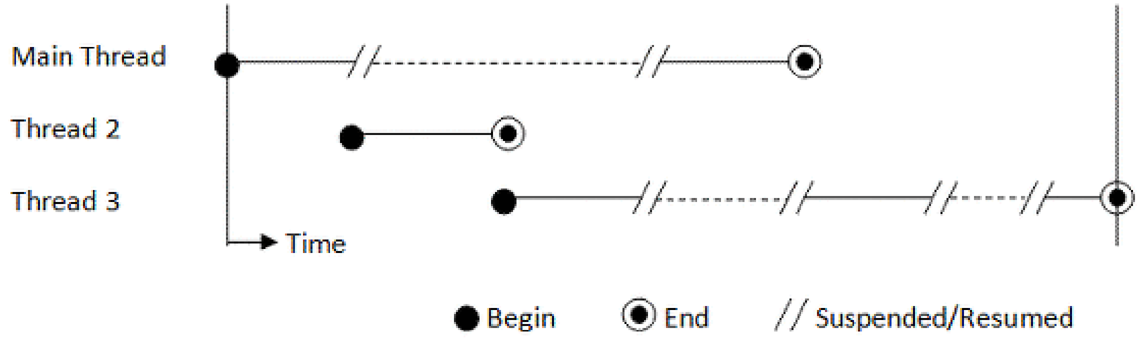


[2] Process Management: Processes & Threads, https://slideplayer.com/slide/15058369/

Handling threads and processes (i.e., tasks)

Scheduler is a procedure (provided by a **kernel**) that decides which task(s) to execute next, e.g., based on:

- **Fairness** – equal opportunity for a task routines to be called;
- **Timing** – amount of time given for a task;
- **Priority** – a lower priority task is executed only after all higher priority tasks terminated;
- Other methods (to be studied **next lecture**)



Techniques to invoke the **scheduler** procedure can follow:

- **Cooperative multitasking** – does not interrupt a task unless the thread/process itself calls a certain procedure or terminates;
- **Interrupt Service Routine (ISR)** – tasks are interrupted by the timer every fixed duration of time (i.e., **jiffy**):
  - Small **jiffy** (eg, 1µs) can degrade overall performance;
  - Large **jiffy**(eg,10ms) compromise real-time response.

Creating threads

Thread library supports using **threads**, e.g.:

- **POSIX threads (Pthreads)** is standardized Application Program Interface (API) supported by many OS (including WinCE, Unix).

To create a **thread** you need:

- Define processor directive:
  - *#include <pthread.h>*
- Declare ID, create and exit status variables:
  - *pthread_t ID;*
  - *int createStatus;*
  - *void* exitStatus;*
- Declare and specify thread procedure (i.e., function):
  - void fun_name(void){…}
- Create thread:
  - *createStatus = pthread_create(&ID,&attr,fun_name,&arg);*

#Consider a multi-thread C program:

```c
#include <pthread.h>
#include <stdio.h>
void* printN(void* arg) {  // Procedure assigned to threads
    int i;
    for (i = 0; i < 10; i++) {
        printf("My ID: %d\n", *(int*)arg);
    }
    return NULL;
}
int main(void) {
    pthread_t threadID1, threadID2;
    void* exitStatus;
    int x1 = 1, x2 = 2;
    pthread_create(&threadID1, NULL, printN, &x1);
    pthread_create(&threadID2, NULL, printN, &x2);
    printf("Started threads.\n");
    pthread_join(threadID1, &exitStatus);
    pthread_join(threadID2, &exitStatus);
    return 0;
}
```

Memory consistency models of threads

All possible implementation of threads* defines a **Memory Consistency (MC) model**.

MC model defines how variables that are read and written by different threads can be accessed by those threads (e.g., allowed ordering of access).

#Consider example:

1. w, x, y, z = 0;

Thread A:	Thread B:
❑ x = 1;	❑ y = 1;
❑ w = y;	❑ z = x;

Depending on the execution order, we:

- Know that x, y and {w or z} are ones – **sequential consistency**;
- If the order of threads execution affects program results, then it is called **Race conditions.**

!Race conditions can be disastrous for applications, but not necessarily.

* - A thread may be suspended between any two **atomic operations** to execute another thread and/or an **ISR**.

Mutual exclusion (mutex)

A **mutual exclusion lock (mutex)** prevents any two threads from simultaneously accessing or modifying a shared resource (i.e., memory).

Mutexes are initialized by creating an instance of a structure:

- *pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;*

The lock is acquired by a thread though:

- pthread_mutex_lock(&lock);

The lock is released by a thread through:

- *pthread_mutex_unlock(&lock);*

Deadlocks (side-effect of **mutex**) may occur when some threads become permanently blocked trying to acquire locks held by other threads:

- May occur when using multiple locks.

#Consider the same example with mutex:

1. w, x, y, z = 0;
2. pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Thread A:

- ❑ *pthread_mutex_lock(&lock);*
- ❑ x = 1;
- ❑ w = y;
- ❑ *pthread_mutex_unlock(&lock);*

Thread B:

- ❑ *pthread_mutex_lock(&lock);*
- ❑ y = 1;
- ❑ z = x;
- ❑ *pthread_mutex_unlock(&loc*

#Consider the example with multiple locks:

1. w, x, y, z = 0;
2. *pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;*
3. *pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;*

Thread A:

- ❑ *pthread_mutex_lock(&lock1);*
- ❑ *x = 1;*
- ❑ *pthread_mutex_lock(&lock2);*
- ❑ *w = y;*
- ❑ *pthread_mutex_unlock(&lock1);*
- ❑ *pthread_mutex_unlock(&lock2);*

Thread B:

- ❑ *pthread_mutex_lock(&lock2);*
- ❑ *y = 1;*
- ❑ *pthread_mutex_lock(&lock1);*
- ❑ *z = x;*
- ❑ *pthread_mutex_unlock(&lock2);*
- ❑ *pthread_mutex_unlock(&lock1);*

Mutual exclusion (mutex)

Techniques to avoid **deadlocks**:

- **Use only one lock** – may not be practical for concurrency (i.e. real-time constraints and program modularity);
- **Avoid nested locks** – equivalent to the above;
- **Use global mutex** to disable blocks at some frequency to pass deadlocks;
- **Use a hierarchy of locks** – acquire locks in the same order in all threads that use them.
- Other methods…

#Consider example with the hierarchy of locks:

1. w, x, y, z = 0;
2. *pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;*
3. *pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;*

Thread A:

- ❑ *pthread_mutex_lock(&lock1);*
- ❑ *x = 1;*
- ❑ *pthread_mutex_lock(&lock2);*
- ❑ *w = y;*
- ❑ *pthread_mutex_unlock(&lock2);*
- ❑ *pthread_mutex_unlock(&lock1);*

Thread B:

- ❑ *pthread_mutex_lock(&lock1);*
- ❑ *y = 1;*
- ❑ *pthread_mutex_lock(&lock2);*
- ❑ *z = x;*
- ❑ *pthread_mutex_unlock(&lock2);*
- ❑ *pthread_mutex_unlock(&lock1);*

Creating processes

Processes are imperative programs with their own memory spaces that can run concurrently.

#In UNIX, two or more programs can be run from the command line (i.e., bash) as follows:

prog1 &

prog2 &

where “&” sends a program to the background.

Communication between processes

Processes* cannot refer to each others' variables (ensured by the hardware – MMU**), and consequently they do not exhibit the same difficulties as threads.

To achieve concurrency, communication between the programs must occur via mechanisms provided by:

- Operating system via files: *fopen(), fread(), fwrite(), fclose()*;
- Libraries via messages, e.g., *send(), get(), etc.;*
- Other mechanisms, e.g., kernel.

*** – Processes** are imperative programs with their own memory spaces.

** – Memory management unit (protects the memory of one process from accidental reads and writes by another).

Example of files handling by processes

#Write to a binary file using fwrite()

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

struct threeNum{
  int n1, n2, n3};

int main(){
  int n;
  struct threeNum num;
  FILE *fptr;

  if ((fptr = fopen("C:\\program.bin","wb")) == NULL){
     printf("Error! opening file");
     exit(1); // Program exits if the file pointer returns NULL.
  }

  for(n = 1; n < 5; ++n){
    num.n1 = n;
    num.n2 = 5*n;
    num.n3 = 5*n + 1;
    fwrite(&num, sizeof(struct threeNum), 1, fptr);
  }
  fclose(fptr);
  return 0;
}
```

#Read from a binary file using fread()

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

struct threeNum{
  int n1, n2, n3};

int main(){
  int n;
  struct threeNum num;
  FILE *fptr;

  if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
     printf("Error! opening file");
     exit(1); // Program exits if the file pointer returns NULL.
  }

  for(n = 1; n < 5; ++n){
    fread(&num, sizeof(struct threeNum), 1, fptr);
    printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);
  }
  fclose(fptr);

  return 0;
}
```

To sum up

- Multitasking is a mid-layer level of concurrency (between hardware and conceptual abstraction) supported by the OS kernel through:
  - Memory allocation;
  - Scheduling of tasks.
- Two main multitasking mechanisms provided by an OS are:
  - Processes – imperative programs with their own memory spaces.
  - Threads – imperative programs that run concurrently and share a memory space.
- Shared memory of threads impose several challenges:
  - Memory consistency – defined from all possible implementation of threads;
  - Race conditions - the order of threads execution affects program results (can be resolved with mutex);
  - Deadlocks – side-effect of mutexes when threads become permanently blocked trying to acquire locks held by other threads;
- Processes are not prone to the above issues until they start communicating with other processes.

The end!

See you next time – **April 24**.

Well done with the timely submission of Assignment 1!

Late submission for Assignment 1 will be cut-off on **April 20, 23:59**.