Real-Time Scheduling

Contents:

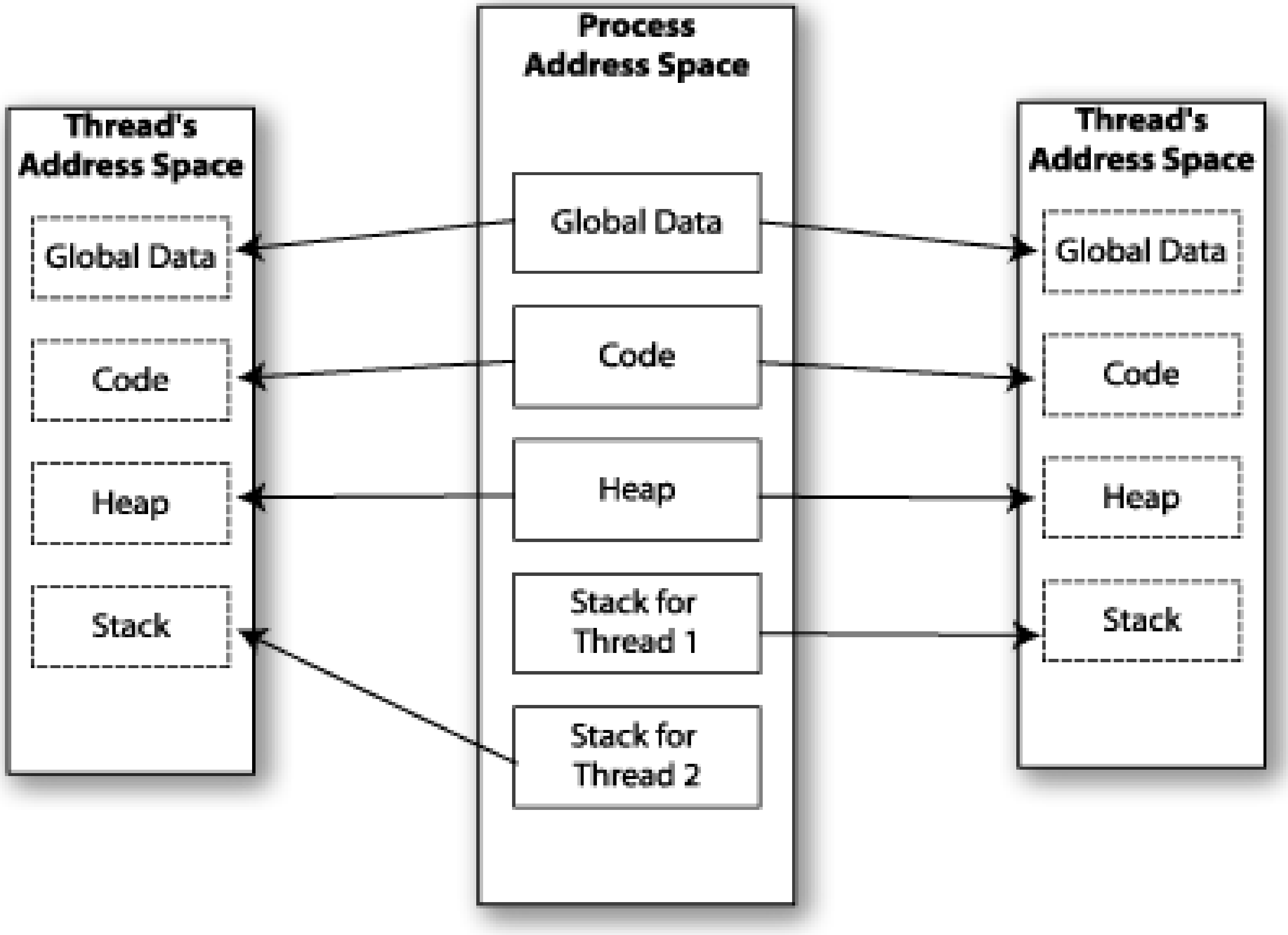
Threads

Event-driven program

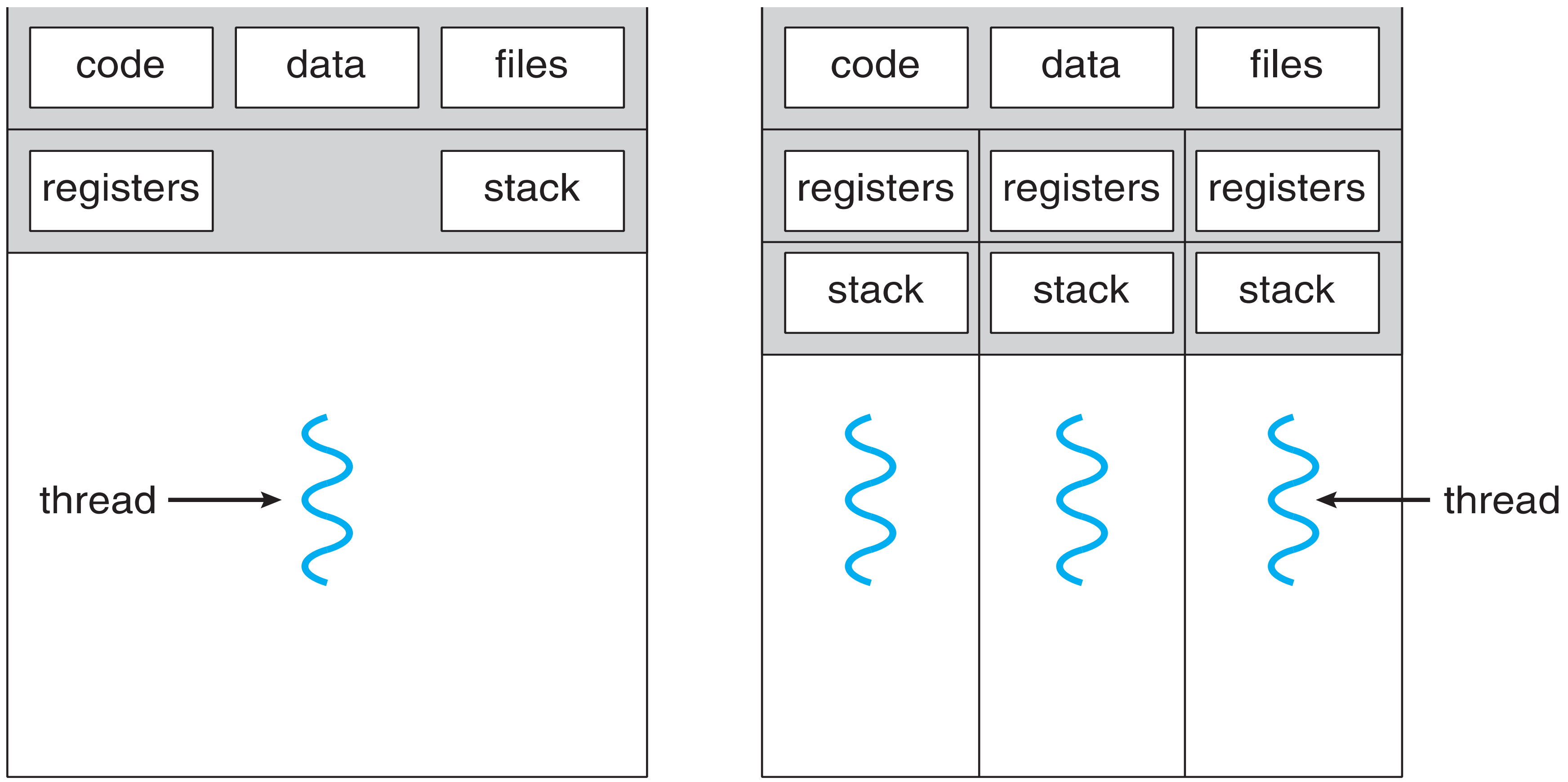
Threads

- Threads give us a more efficient way to implement a task.
- With threads, multiple subtasks can be implemented as separate streams in a single process.
- In the threads model, we break the memory space of a process into two parts:
  - One part contains the program-wide resources such as global data and program instructions.
  - The other contains information pertaining to the execution state of control stream, such as the PC and the stack.

Memory Layout for Multithreaded Program



Single and Mutlitheaded Process



single-threaded process

multithreaded process

Advantages of Threads

- Shared Address space implies that the communication among threads is more efficient.
- Context switching between threads in the same process is typically faster than context switching between processes.
- It is much quicker to create a thread than a process.
- Thread programming is supported by POSIX

What is POSIX

- Portable Operating System Interface
- An interface standard developed by IEEE and approved by ANSI
- Ensures portability of applications across variations of Unix OSes
- Provides system calls for creation of a process or a thread
- Has real time extensions

Disadvantages of Threads

- Need of Synchronization – Global variables are shared between threads: Inadvertent modification of shared variables can be disastrous.

- Security: Many library functions are not thread safe.

- Lack of robustness: If one thread crashes, the whole application crashes.

Question To Think

- Do we benefit from using a multi-threaded process when it runs on a uniprocessor system?
  - Speed of a program is either I/O bound or CPU bound
  - If I/O bound (in most cases), multiple threads will make the process more efficient.

int pthread_create(pthread_t ****thread*, **const pthread_attr_t ****attr*, **void *(**start_routine*) **(void *), void ****arg***)**

pthread

- #include <pthread.h>
- Define a worker function

```
void *foo(void *args) { }
```

- Initialize pthread attr t

```
pthread_attr_t attr;
pthread_attr_init(attr);
```

- Create a thread

pointer to func

```
pthread_t thread;
pthread_create(&thread, &attr, worker function, arg);
```

- Exit current thread

```
pthread_exit(status)
```

Thread Programming Example

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 5

void *print_hello(void *threadid)
{
    long* tid = threadid;
    printf("Hello World! It’s me, thread #%ld!\n", *tid);
    pthread_exit(NULL);
}
```

```
int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for (t = 0; t < NUM_THREADS; t++)
    {
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(threads + t, NULL, print_hello, (void *) &t);
        sleep(1);
        if (rc)
        {
            printf("ERROR; return code from pthreadh_create() is %d\n", 
rc);
            return -1;
        }
    }
    return 0;
}
```

Must use “-pthread” option to compile.

Race Condition

- An error condition to parallel programs in which the outcome of a program changes as the relative scheduling of different control flows varies.

- Generally, **race conditions** can happen where the ordering of events can affect the outcome of some computation.

- What is wrong with the race condition?

In engineering, we would like the outcome be predictable and repeatable.

Another Example

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

int g = 0;

void *aThread()
{
    g++;
    sleep(1);
    pthread_exit(NULL);
}
```

```
int main (int argc, char *argv[])
{
    int i;
    pthread_t thread[20];
    for (i=0; i<20; i++)
    {
        if( pthread_create(thread+i, NULL, aThread, NULL) )
        {
         printf("ERROR; return code from pthread_create()\n");
         return -1;
        }
     printf("The value of g is %d after creating thread %d\n", g, i);
    }
    return 0;
}
```

Possible Results

hew11@mills:~/test/thread

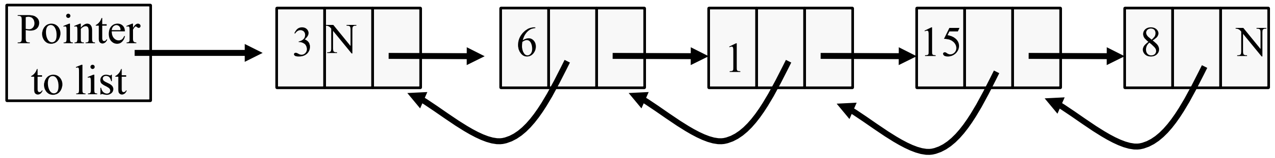
```
[hew11@mills ~/test/thread] ./test2
The value of g is 0 after creating thread 0
The value of g is 1 after creating thread 1
The value of g is 2 after creating thread 2
The value of g is 3 after creating thread 3
The value of g is 4 after creating thread 4
The value of g is 5 after creating thread 5
The value of g is 6 after creating thread 6
The value of g is 7 after creating thread 7
The value of g is 8 after creating thread 8
The value of g is 9 after creating thread 9
The value of g is 10 after creating thread 10
The value of g is 11 after creating thread 11
The value of g is 12 after creating thread 12
The value of g is 13 after creating thread 13
The value of g is 14 after creating thread 14
The value of g is 15 after creating thread 15
The value of g is 16 after creating thread 16
The value of g is 17 after creating thread 17
The value of g is 18 after creating thread 18
The value of g is 19 after creating thread 19
```

hew11@mills:~/test/thread

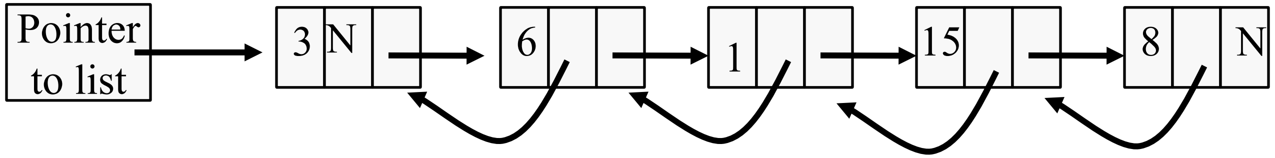
```
[hew11@mills ~/test/thread] ./test2
The value of g is 0 after creating thread 0
The value of g is 1 after creating thread 1
The value of g is 2 after creating thread 2
The value of g is 3 after creating thread 3
The value of g is 4 after creating thread 4
The value of g is 5 after creating thread 5
The value of g is 6 after creating thread 6
The value of g is 7 after creating thread 7
The value of g is 8 after creating thread 8
The value of g is 10 after creating thread 9
The value of g is 10 after creating thread 10
The value of g is 11 after creating thread 11
The value of g is 12 after creating thread 12
The value of g is 13 after creating thread 13
The value of g is 14 after creating thread 14
The value of g is 15 after creating thread 15
The value of g is 16 after creating thread 16
The value of g is 17 after creating thread 17
The value of g is 18 after creating thread 18
The value of g is 19 after creating thread 19
```

Race Condition

- If function A is inserting a number to the list and function B is printing the list, race condition occurs.



Do we have a race condition now?

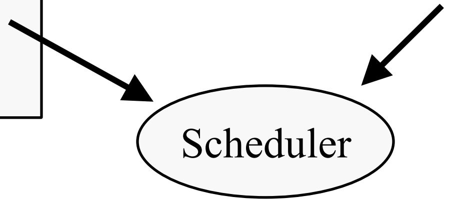


Function A

```
while (1){
    if (X>0) {
        X=0;
        insert an elem in list;
        X=1;
        }
     sleep(1);
}
```

Function B

```
while (1){
    if (X>0) {
        X=0;
        print list;
        X=1;
        }
     sleep(1);
}
```



int X=1 is a shared variable used to enforce mutually exclusive access to a linked list.

Possible Solutions

- What are the possible solutions to race conditions in a uniprocessor system?
  - disable preemption/parallization when scheduling processes (only the process itself can voluntarily relinquish the CPU)
  - Use semaphores as **atomic** operation

Event Driven Program

- Let’s see two examples implementing the same functionality.

Example 1

```
#include<stdio.h>

#include<pthread.h>

#include<stdlib.h>

#include<signal.h>


int r1=0;

int r2=0;

int sum=0;

int stopped=0;


void *myThread()

{

   int i;

   for(i=0; i<5; i++)

   {

      sleep(1);

      time_t t;

      //initialize random number generator

      srand((unsigned) time(&t));
      r1=rand();
      r2=rand();
      sum=r1+r2;
      printf("i=%d, Sum=%d\n", i, sum);
   }

   stopped=1;
   pthread_exit(NULL);
}


int main ()
{
   pthread_t thread;

   if( pthread_create(&thread, NULL, myThread, NULL) )
   {
      printf("ERROR; return code from pthread_create()\n");
      return -1;
   }

   while(!stopped);

   pthread_join(thread, NULL);
   return 0;
}
```

Example 2

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

int r1=0;
int r2=0;
int sum=0;

void handle_addRNGs(int sig)
{
   time_t t;
   //initialize random number genreator
   srand((unsigned) time(&t));

   r1=rand();
   r2=rand();
   sum=r1+r2;

   kill(getpid(), SIGUSR2);
}
```

```c
void handle_printSum(int sig)
{
   printf("Sum=%d\n", sum);
}

int main ()
{
   pid_t pid;
   if ((pid = fork()) == 0)
   {
      int i;
      for(i=0; i<5; i++)
      {
         sleep(1);
         kill(getppid(), SIGUSR1);
      }
      exit(0);
   }
   else{
      signal(SIGUSR2, handle_printSum);
      signal(SIGUSR1, handle_addRNGs);

      waitpid(pid, NULL, 0);
      //printf("The End!\n");
      exit(0);
   }
}
```

How a Process Knows that an Event Occurs?

Polling and Interrupt

- Polling: Constantly reading a memory location, in order to receive updates of an event or input value.

- Interrupt: Upon receiving an interrupt signal, the processor interrupts whatever it is doing and serves the request.

Polling

- We repetitively test a flag to capture the occurrence of an event.

- Consider a system that handles packets of data that arrive at the rate of 1 per second. On arrival of a packet a flag packet-here is set to 1.

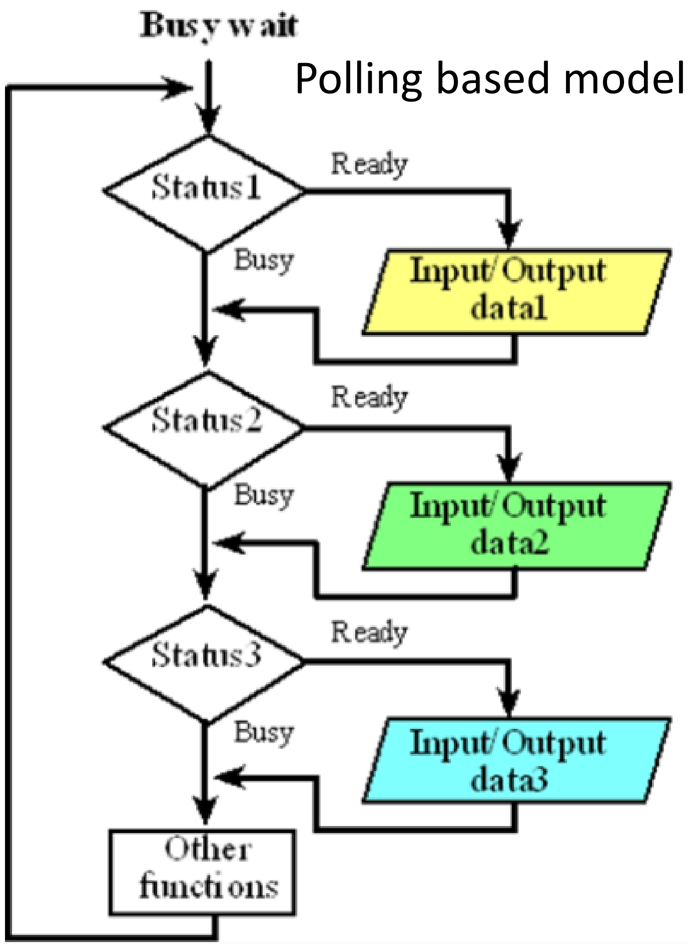
```
for(; ;) {
    if (packet-here)
    {
      process-data();
      packet-here = 0;
    }
}
```

Interrupt

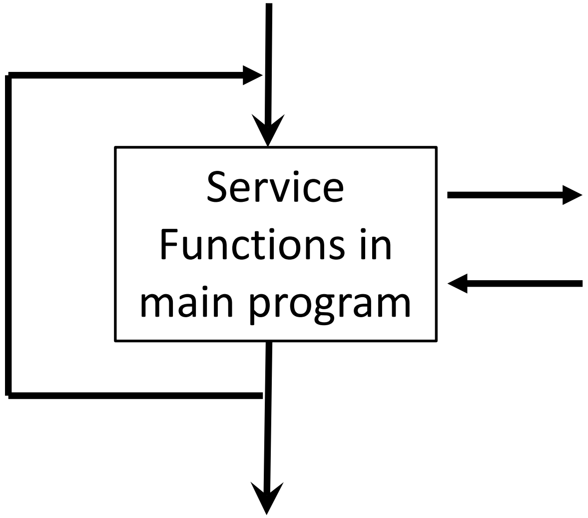
```
signal(singal, handler);

void  handler(int sig)
{
   process-data();
}

int main()
{
   ...
   while (1)
   {
      //Do some work
   }
}
```

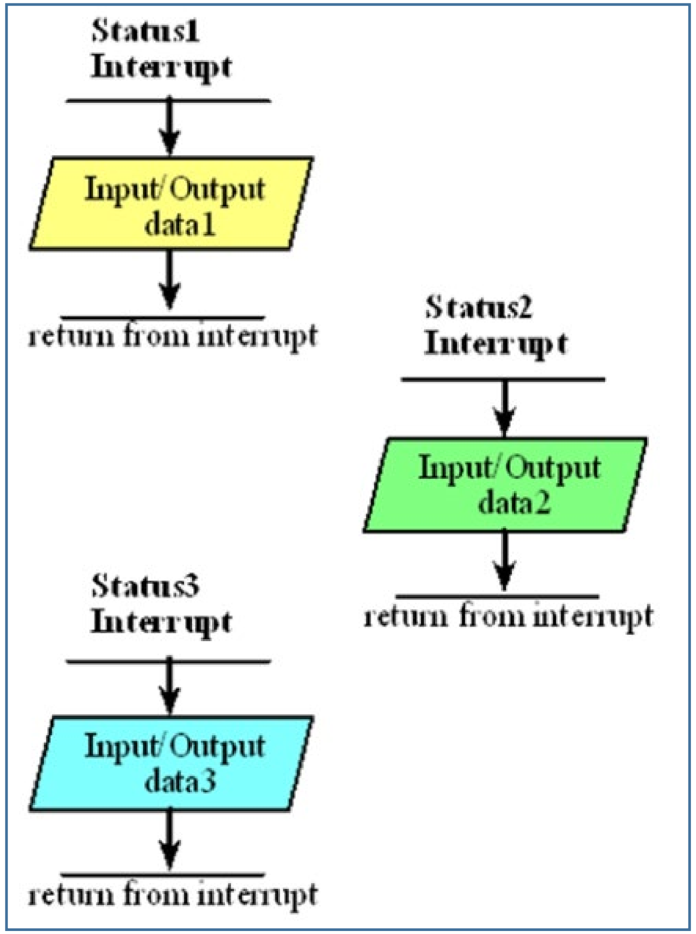
Handling of Multiple Tasks

Polling based model



Interrupt based model





Pros and Cons of Polling

- Pros:

  Simple to write and debug

  Response time easy to determine

Polled loops are used for fast response to single devices.

- Cons:

  Generally not sufficient to handle complex systems or burst events

  Waste of CPU time particularly when events polled occur infrequently

Brief Comparison

	Interrupt	Polling
Speed	fast	slow
Efficiency	good	poor
CPU waste	low	high
Multitasking	yes	yes
Complexity	high	low
Debugging	difficult	easy