Processes and Threads

Week 04 – Tutorial

Outline

- Review of Week 3
- Quiz
- Review of Processes
- Review of Threads

A word of caution on #define

- What is the pitfall in the following code:

```c
#include <stdio.h>


#define YYY 7
#define predecessor(X) X-1
#define nullify(X) (X)-(X)


int main(void) {
     int a, b;
     a = YYY;
     b = predecessor(a)*2;
     a = nullify(b--);
     printf(" %d %d\n",a, b);
}
```

A word of caution on #define

- What is the pitfall in the following code:

```c
#include <stdio.h>


#define YYY 7
#define predecessor(X) X-1
#define nullify(X) (X)-(X)


int main(void) {
      int a, b;
      a = YYY;
      b = predecessor(a)*2;
      a = nullify(b--);
      printf(" %d %d\n",a, b);
}
```

The code after being substituted by the pre-compiler:

```c
a = 7;
// so far, so good

b = a-1*2;
// b = 5

a = (b--)-(b--);
// b = 3, a = 1
```

Example of Array and Pointer in C

```c
#include <stdio.h>
int main() {
    /*Array declaration*/
    int val[7] = { 11, 22, 33, 44, 55, 66, 77 };
    /*Pointer variable*/
    int *p = &val[0];

    for (int i = 0; i < 7; i++) {
        printf("val[%d]: value is %d and address is %p\n", i, *p, p);
        p++;
    }
    return 0;
}
```

Example of Array and Pointer in C

```c
#include <stdio.h>
int main() {
    /*Array declaration*/
    int val[7] = { 11, 22, 33, 44,
    /*Pointer variable*/
    int *p = &val[0];

    for (int i = 0; i < 7; i++) {
        printf("val[%d]: value is %d a
        p++;
    }
    return 0;
}
```

Output:
val[0]: value is 11 and address is 0xffffcb80
val[1]: value is 22 and address is 0xffffcb84
val[2]: value is 33 and address is 0xffffcb88
val[3]: value is 44 and address is 0xffffcb8c
val[4]: value is 55 and address is 0xffffcb90
val[5]: value is 66 and address is 0xffffcb94
val[6]: value is 77 and address is 0xffffcb98

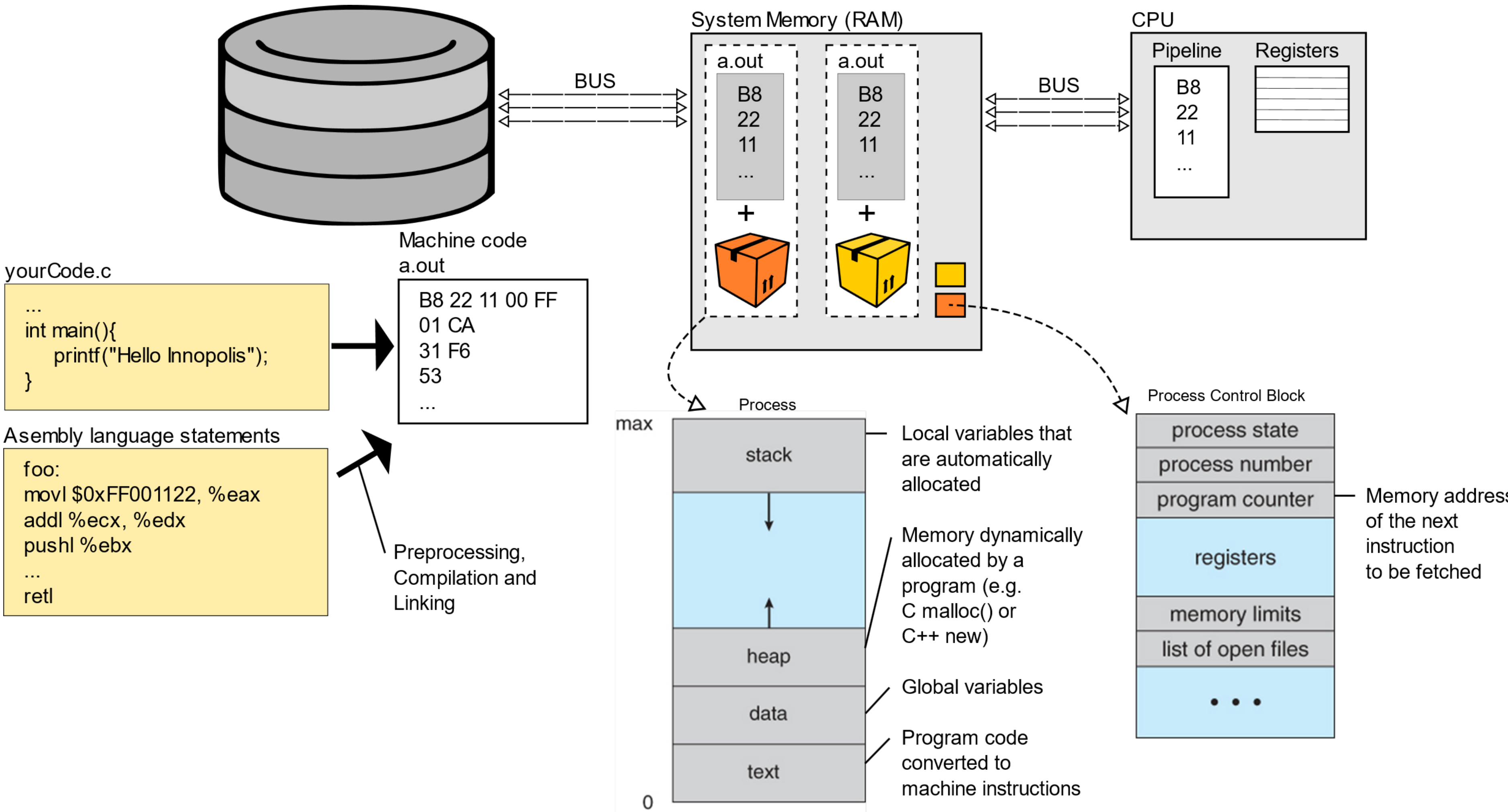
Example of a Linked List using Struct

- However, the main point of the task was to check if you understand the difference between a structure and a pointer to a structure. Again, differ->hours is a syntactic sugar for (*differ).hours so both syntaxes are good for use
- differ.hours, &differ.hours, *differ.hours are incorrect

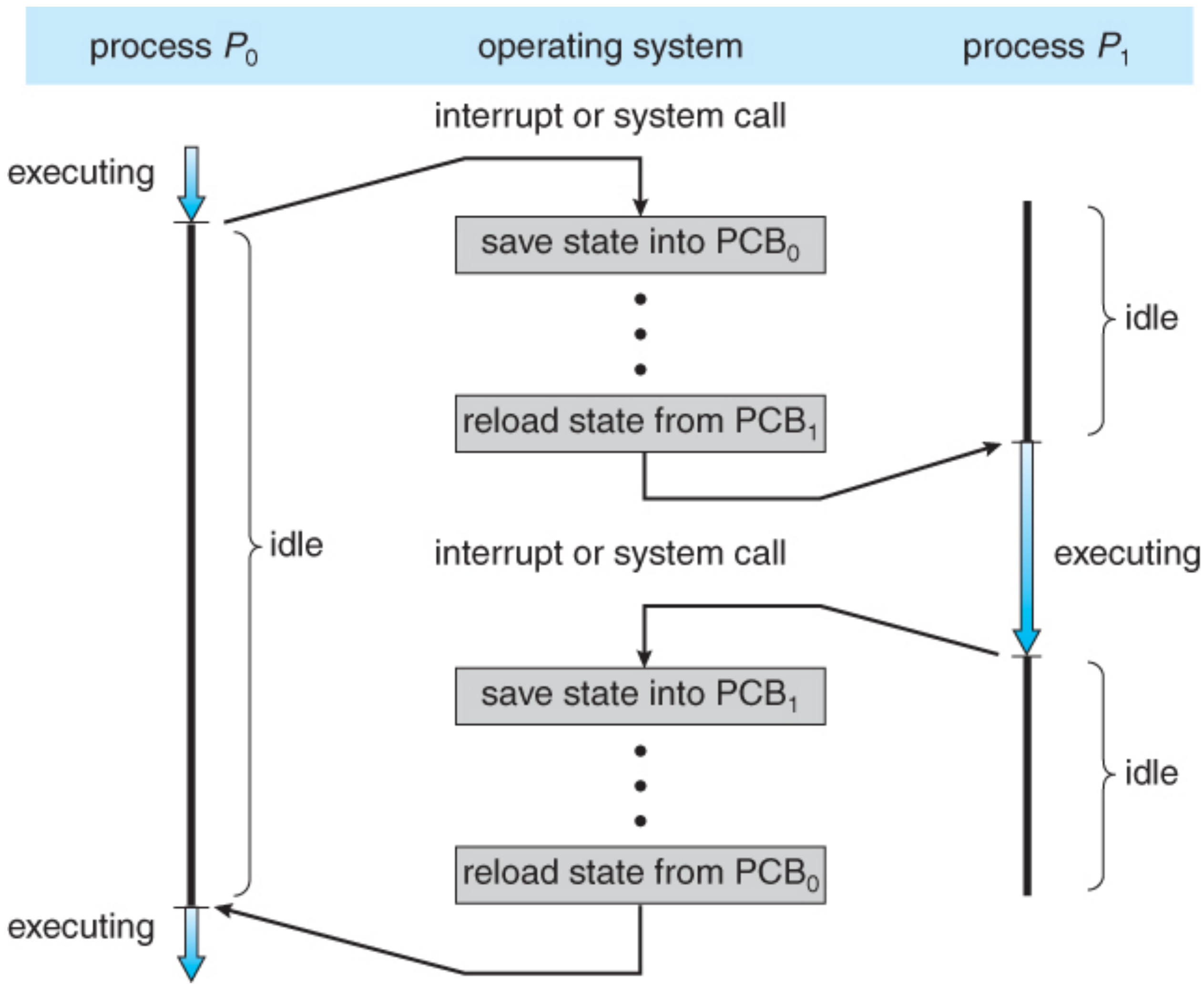
Quiz – 15 min.

Break – 5 min.

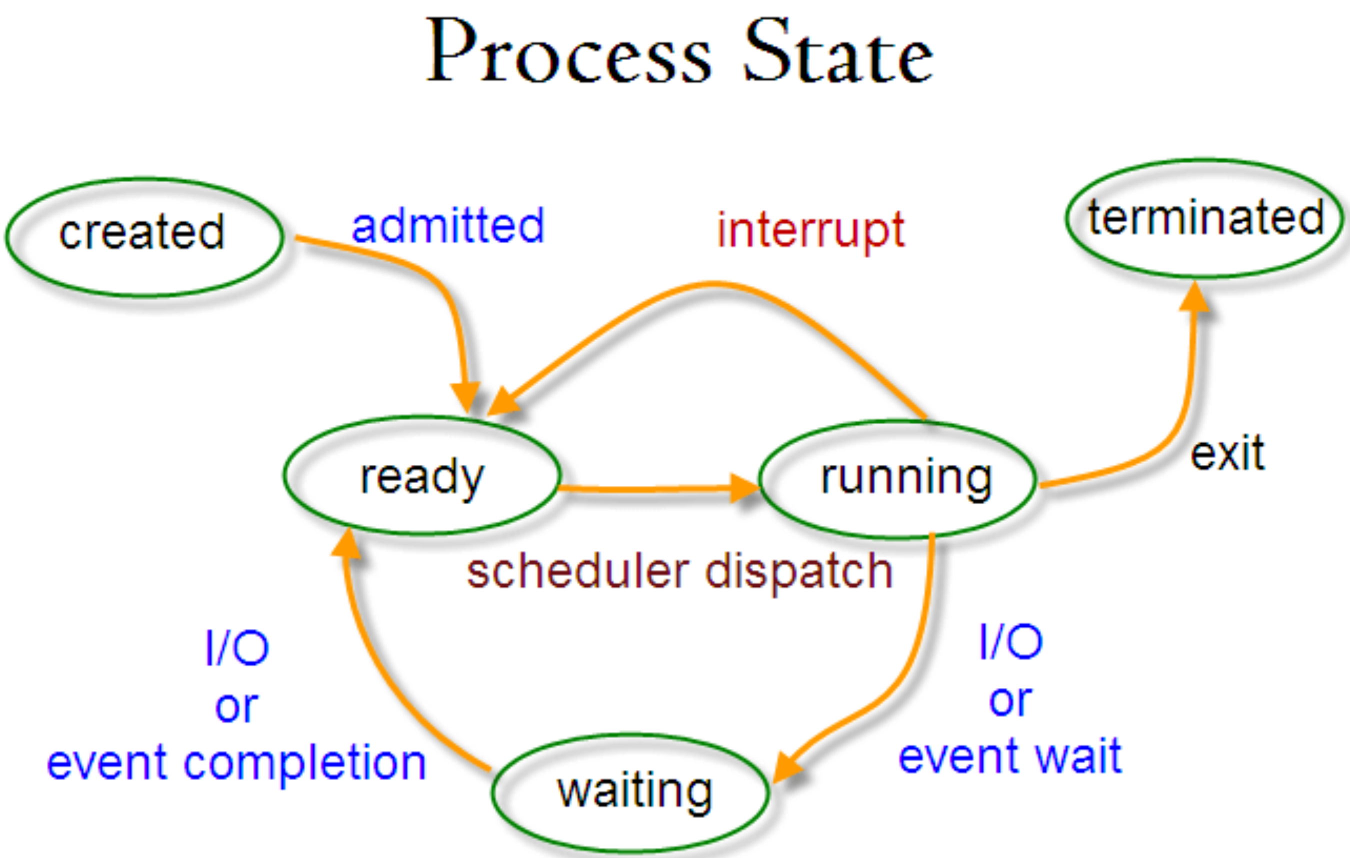
Process and Process Control Block



Context Switch



Process State

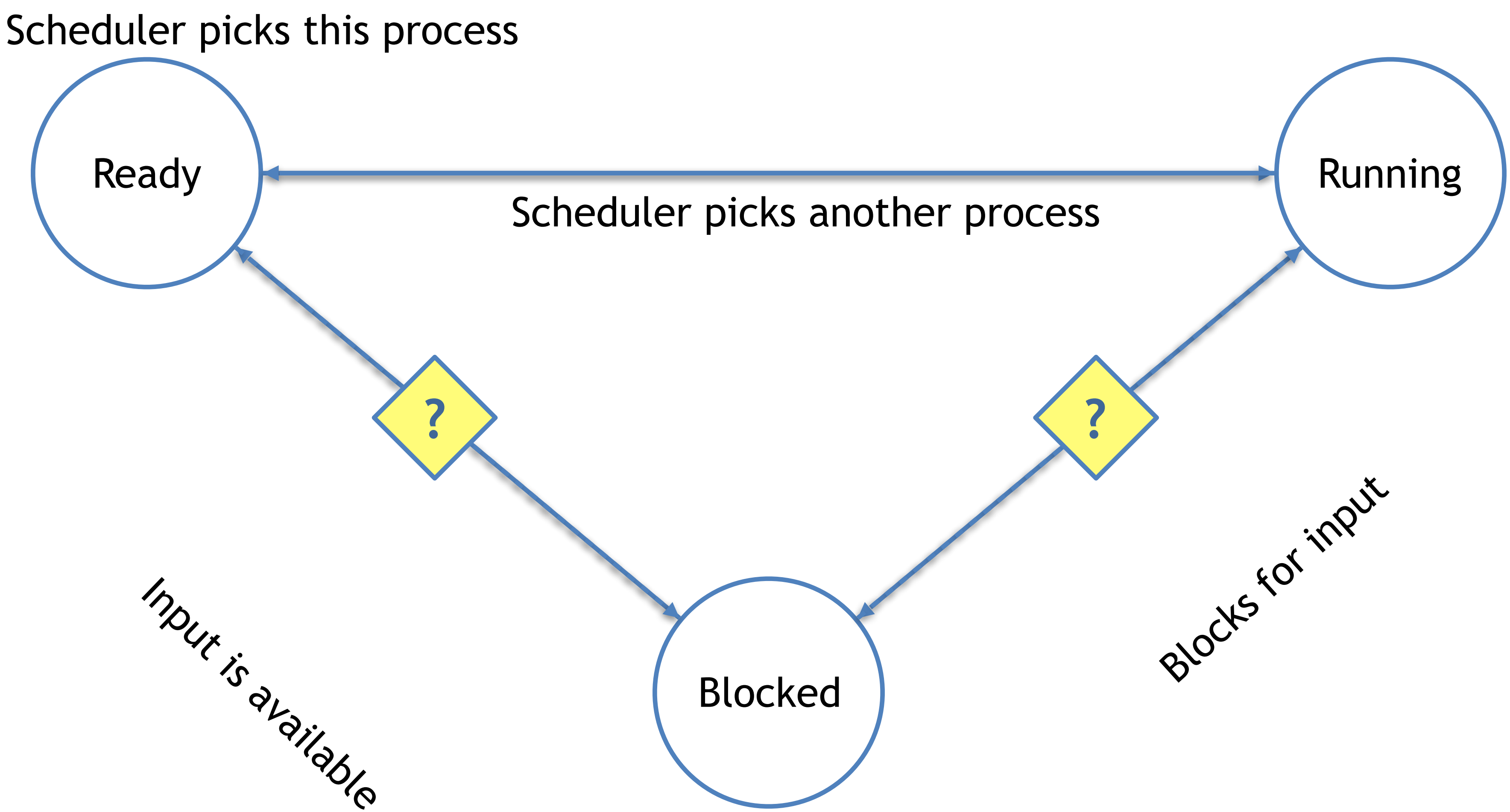


[States of process in Linux: shed.h]

Problem 2.1 (1/2)

- In the following figure, three process states are shown.
- In theory, with three states, there could be six transitions, two out of each state. However, only four transitions are shown.
- Are there any circumstances in which **either or both** of the missing transitions might occur?

Problem 2.1 (2/2)



Process states

Problem 2.1 - Solution (1/2)

- The transition that from the **Ready** state to the **Blocked** state is impossible since in order to be switched to **Blocked** state a process has to perform an I/O operation which is possible only if the process is active, i.e. is in the **Running** state

Problem 2.1 - Solution (2/2)

- A transition from **Blocked** to **Running** state is possible if and only if an I/O operation that was the reason of blocking is finished and the CPU is idle at this time

Problem 2.4

- When an interrupt or a system call transfers control to the operating system, a kernel stack area separate from the stack of the interrupted process is generally used. Why?

Problem 2.4 – Solution

- The separate stack area is used for the interrupted processes. The causes for using the distinct stack for kernel are as given below:
  – By using distinct stack, the data is not overwritten on the kernel
  – By doing so, the operating system will not crash
  – To protect the information of processes from malicious users
  – The separate memory space can be used to make system calls

Problem 2.8

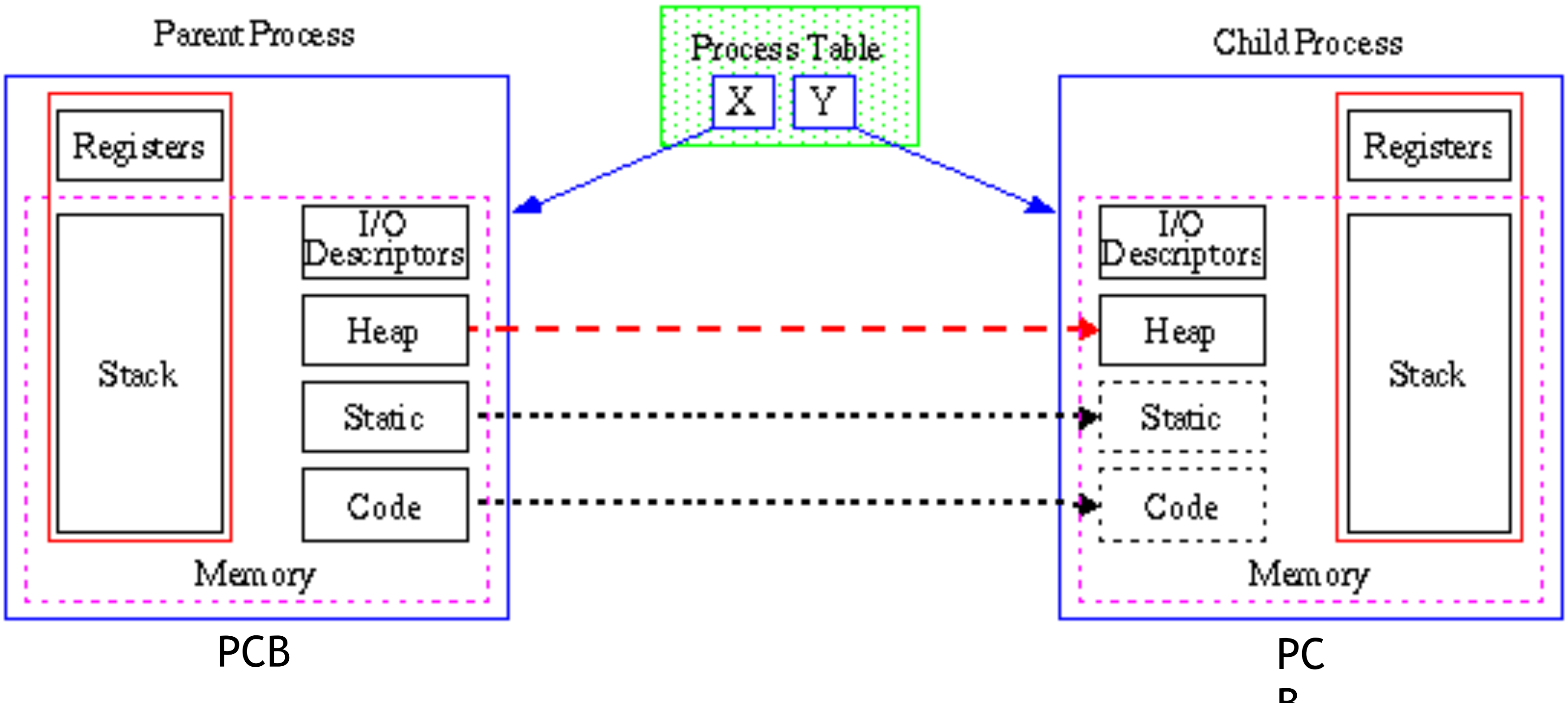
- Consider a multiprogrammed system with degree of 6 (i.e., six programs in memory at the same time).
- Assume that each process spends 40% of its time waiting for I/O.

- What will be the CPU utilization?

Problem 2.8 – Solution

- Given that there are 6 programs in memory, n=6
- Each process spends 40% of time waiting for I/O, therefore the fraction of time each process spends waiting for I/O denoted by P = 0.4
- CPU utilization is given as = $1-P^n = 1-(0.4)^6 = 1 - 0.004096 = 0.99590$
- Therefore, the CPU utilization is 99%

Processes - Fork



"Fork is the only way to create a new process in POSIX (standard for UNIX that most versions of UNIX support). *It creates an exact duplicate of the original process, including all the file descriptors, registers - everything. After the fork, the original process and the copy (the parent and child) go their separate ways."*
Modern Operating Systems – Tanenbaum and Bos

Processes - Fork

```c
#include <stdio.h>
#include <sys/types.h>
int main() {
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

Output:
hello
hello
hello
hello
hello
hello
hello
hello

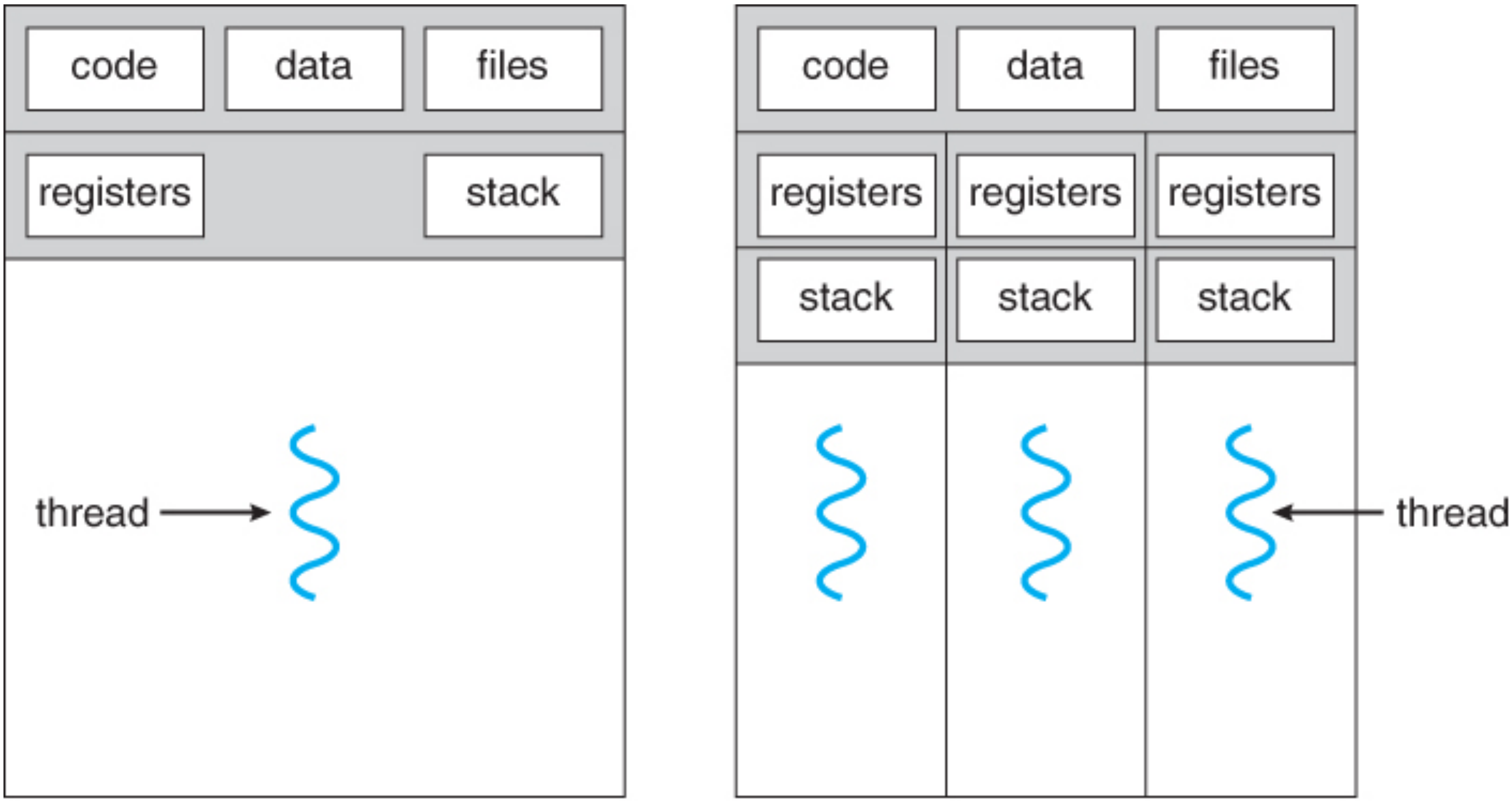
```
        L1           // There will be 1 child process
      /    \         //  created by the 1st fork()
   L2        L2      // There will be 2 child processes
  /  \      /  \     //  created by the 2nd fork()
L3   L3   L3   L3    // There will be 4 child processes
                     //  created by the 3rd fork()
```

Processes - Fork

```c
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

void main(void) {
   int pid;
   int i = 100;
   char *a = "data";
   char buf[BUF_SIZE];

   fork();
   pid = getpid();
   for (i = 1; i <= MAX_COUNT; i++) {
      // printf() will group the output o
      // printf("This line is from pid %d

      // While buffering the output for t
also use
      // printf to print out some information, which will also be buffered.
      sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
      write(1, buf, strlen(buf));
   }
}
```

```
Output:
This line is from pid 2244, value =
1
This line is from pid 2244, value =
2
…
This line is from pid 2244, value =
8
This line is from pid 3028, value =
1
This line is from pid 3028, value =
2
This line is from pid 3028, value =
3
This line is from pid 2244, value =
9
This line is from pid 3028, value =
4
…
```

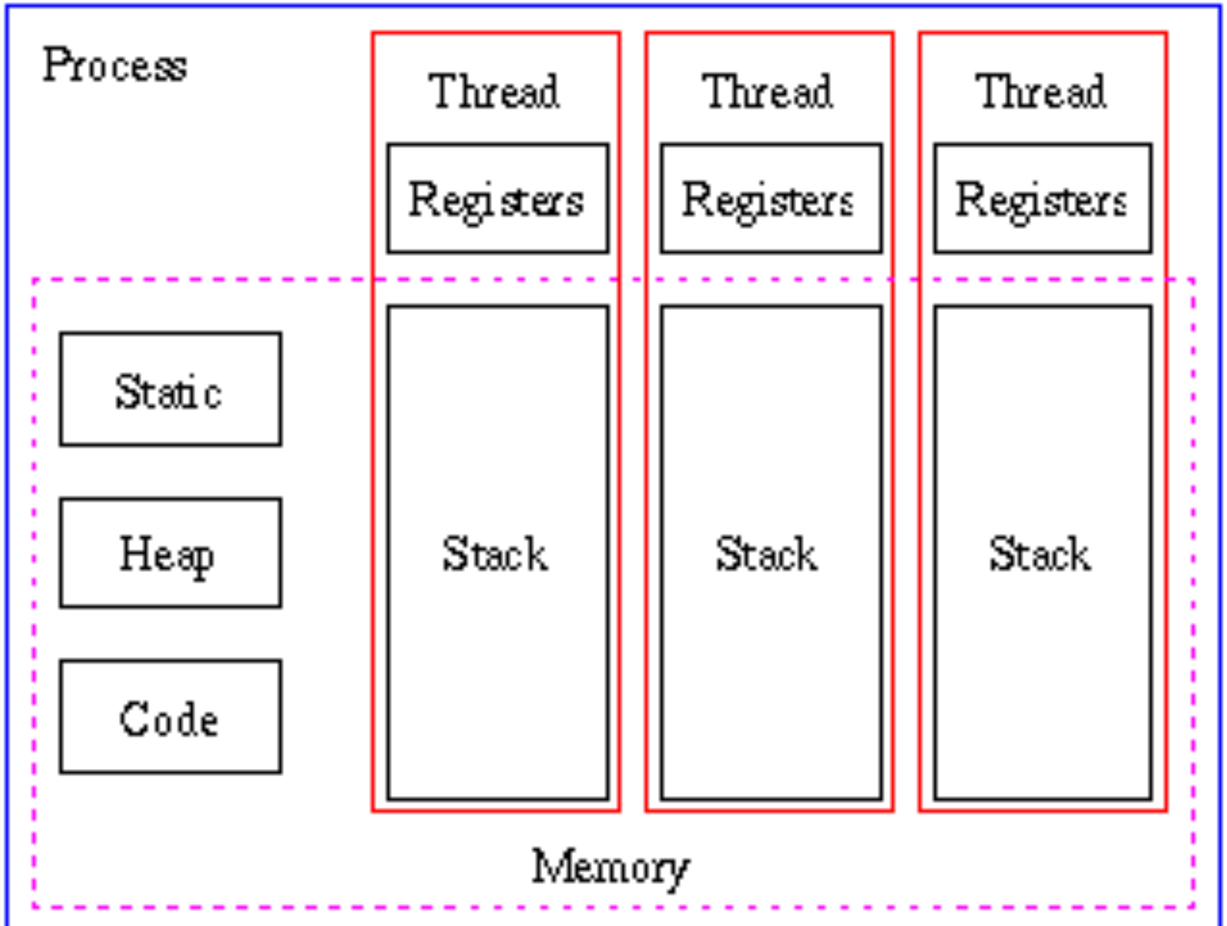
Processes and Threads



single-threaded process

multithreaded process

Threads in a process



Types of Threads

- **User Level Threads** – User managed threads.

- **Kernel Level Threads** – Operating System managed threads acting on kernel, an operating system core.

Example of User Threads

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

#define NUM_THREADS 3

int thread_id[NUM_THREADS];

void * PrintHello(int i) {
   printf("Hello from thread %d - I was created in iteration %d !\n",
      (int) pthread_self(), i);
   // The function's return value serves as the thread's exit status
   // pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_exit.html
   pthread_exit(NULL);
}

int main(int argc, char * argv[]) {
   int rc, i;
   for (i = 0; i < NUM_THREADS; i++) {
      // Create a new thread that will execute 'PrintHello'
      // See: http://pubs.opengroup.org/onlinepubs/007908775/xsh/pthread_create.html
      rc = pthread_create(&thread_id[i], NULL, PrintHello, i);
      if (rc) {
         printf("\n ERROR: return code from pthread_create is %d \n", rc);
         exit(1);
      }
      printf("\n I am thread %d. Created new thread (%d) in iteration %d ...\n",
         (int) pthread_self(), (int) thread_id[i], i);
      if (i % 5 == 0)
         sleep(1);
    }
    pthread_exit(NULL);
}
```

Example of User Threads

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define NUM_THREADS 3

int thread_id[NUM_THREADS];

void * PrintHello(int i) {
   printf("Hello from thread %d
      (int) pthread_self(), i);
   // The function's return valu
   // pubs.opengroup.org/onlinep
   pthread_exit(NULL);
}

int main(int argc, char * argv[]
   int rc, i;
   for (i = 0; i < NUM_THREADS; i++) {
      // Create a new thread that will execute 'PrintHello'
      // See: http://pubs.opengroup.org/onlinepubs/007908775/xsh/pthread_create.html
      rc = pthread_create(&thread_id[i], NULL, PrintHello, i);
      if (rc) {
         printf("\n ERROR: return code from pthread_create is %d \n", rc);
         exit(1);
      }
      printf("\n I am thread %d. Created new thread (%d) in iteration %d ...\n",
         (int) pthread_self(), (int) thread_id[i], i);
      if (i % 5 == 0)
         sleep(1);
    }
    pthread_exit(NULL);
}
```

```
Hello from thread 468864 - I was created in iteration 0 !

 I am thread 98368. Created new thread (468864) in iteration
0 ...
Hello from thread 534672 - I was created in iteration 1 !

 I am thread 98368. Created new thread (534672) in iteration
1 ...

 I am thread 98368. Created new thread (534928) in iteration
2 ...
Hello from thread 534928 - I was created in iteration 2 !
```

User Level vs. Kernel Level Threads

- User-level threads
  - Faster to create, manipulate and synchronize
  - User managed
  - Not integrated with OS (uninformed scheduling)
- Kernel-level threads
  - Slow to create, manipulate and synchronize
  - Managed by the OS and acting on kernel
  - Integrated with OS (informed)

Problem 2.11

- If a multithreaded process forks, a problem occurs if the child gets copies of all the parent's threads.
- Suppose that one of the original threads was waiting for keyboard input. Now two threads are waiting for keyboard input, one in each process.
- Does this problem ever occur in single-threaded processes?

Problem 2.11 – Solution (1/2)

- A single-threaded process cannot fork if it is waiting for a keyboard input as it would remain in waiting state until it receives the input from the keyboard.
- Once it receives the input, it would resume its execution.

Problem 2.11 – Solution (2/2)

- In a multithreaded process, as two processes are waiting for a keyboard input, only one of them can resume execution once the keyboard input is received and the other would always stay suspended
- Thus, the problem of two threads waiting for an input will occur in a multithreaded process

Problem 2.14

In the table the register set is listed as a per-thread rather than a per-process item. Why? After all, the machine has only one set of registers.

Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Problem 2.14 – Solution

- The register is called a per-thread item because the context saved in a register is thread-specific information
- The register stores the state of every thread so that it can be used during context switching – these data are saved and reloaded on the next execution

References

- http://www.sanfoundry.com/operating-system-questions-answers-basics/