TA3 – Unix Signals & Threads

OPERATING SYSTEMS COURSE

THE HEBREW UNIVERSITY

SPRING 2022

Today’s Plan

- Reminder: interrupts
- Signals
- Processes and Threads
  - User Level Threads
  - Kernel Level Threads
- Ex2

Reminder from TA2

Reminder: Kernel Mode

- When the CPU is in *kernel mode*, it is assumed to be executing *trusted* software, and thus it can execute any instructions and reference any memory addresses.
- The *kernel* is the core of the operating system and it has complete control over everything that occurs in the system.
- The kernel is *trusted* software, all other programs are considered *untrusted* software.

Some Definitions

- A process is an executing instance of a program. An active process is a process that is currently advancing in the CPU (while other processes are waiting in memory for their turns to use the CPU).
- The execution of a process can be interrupted by an interrupt.
- An interrupt is a notification to the operating system that an event has occurred, which results in changes in the sequence of instructions that is executed by the CPU.

Types of Interrupts

- Hardware \ External interrupts are ones in which the notification originates from a hardware device such as a keyboard, mouse or system clock.
- Software \ Internal interrupts include exceptions and traps
  - Exceptions: similar to HW interrupt, but not caused by an external source, but during the program execution when errors occur (division by zero, access to paged memory, etc.)
  - Traps: occurs in the usual run of the program, but unlike exceptions, it is not product of some error. The execution of an instruction that is intended for user programs and transfers control to the operating system. Such a request from the kernel is called a *system call*.

Signals

DEFAULT HANDLERS
SETTING PERSONALIZED HANDLERS
BLOCKING SIGNALS

Signals

- Signals are notifications sent to a process in order to notify it of various "important" events.
- Signals cause the process to stop whatever it is doing (after it finishes executing the current CPU cycle), and force the process to handle them immediately
- The process may configure how it handles a signal
  - (Except for some signals that it cannot configure)

Signals, cont.

- Signals are different from interrupts:
  - Interrupts are generated by the HW (or software), and received and handled by the OS
  - Signals are generated by the OS, and received and handled by a process
- Signals in Unix have names and numbers
  - Use 'man kill' to see the types of signals.

Triggers for Signals

- Some examples for signal triggers:
  - Asynchronous input from the user such as ^C (SIGINT), or typing ‘kill pid’ at the shell
  - The system or another process, for instance if an alarm set by the process has timed out (SIGALRM)
  - A software interrupt caused by an illegal instruction
    - The illegal instruction causes a software interrupt.
    - The software interrupt is received by the OS.
    - The OS generates a signal and sends it to the process.

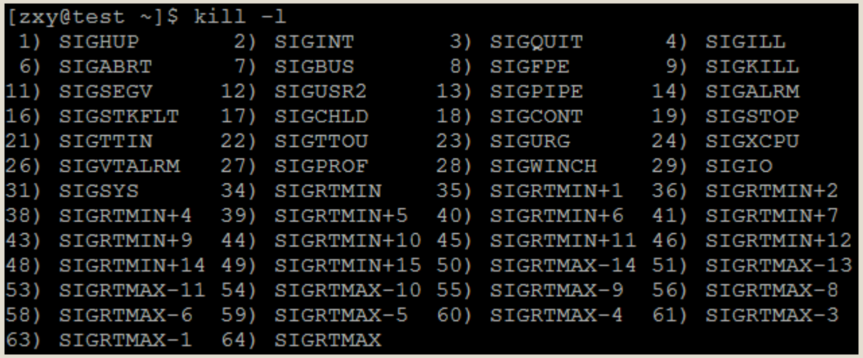
Sending Signals Using the Keyboard

The most common way of sending signals to processes is using the keyboard:

- Ctrl-C: Causes the system to send an INT signal (SIGINT) to the running process.
- Ctrl-\:causes the system to send a QUIT signal (SIGQUIT) to the running process.
- Ctrl-Z: causes the system to send a TSTP signal (SIGTSTP) to the running process.

Sending Signals from the Command Line

- Kill command sends the specified signal to the specified process.
  kill [option] pid.
  - use 'kill –l' to see list of all the signal you can send.



```
[zxy@test ~]$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3
63) SIGRTMAX-1  64) SIGRTMAX
```

- The 'fg pid' command resumes execution of a process (that was suspended with Ctrl-Z), by sending it a CONT signal.

Sending a Signal From One Process To Another

- Signals can be used to send messages from one process to another.
- This is done by using

```
int kill(pid_t pid, int sig)
```

- The messages that are sent in this manner are predefined
  - We cannot send any data

- SIGUSR1, SIGUSR2

Reminder: strace

- **strace** is a debugging utility to monitor the system calls **and signals**
  - Easy to use.
  - Fast debugginng
- **strace** command
  - Shows system calls, arguments, and return values
  - **-t** to display when each call is executed
  - **-T** to display the time spent in the call
  - **-e** to limit the types of calls
  - **-o** to redirect the output to a file
  - **-s** limit the length of print strings.

Handling Signals

- A process must run to handle signals.

- There are several types of handling for signals:
  - The process can exit, ignore, stop or continue execution (all options are default for some signals).

  - The process can specify which action to take when a signal is received, i.e. execute a signal handler.

Handling Signals (cont.)

- There are some signals that the process cannot catch.
  - For example: KILL and STOP
- If you install no signal handlers of your own, the runtime environment sets up a set of default signal handlers
  - For example:
    - The default signal handler for TERM calls exit().
    - The default handler for ABRT is to dump the process's memory image into a file, and then exit.

sigaction

```
int sigaction(int sig,
     struct sigaction *new_act,
          struct sigaction *old_act);
```

- Allows the calling process to examine and/or specify the action to be associated with a specific signal.
  - action = signal handler + signal mask + flags

`sigaction` cont.

- The signal mask is calculated and installed only for the duration of the signal handler.
- By default, the signal itself is also blocked when the signal occurs.
- Once an action is installed for a specific signal using sigaction, it remains installed until another action is explicitly installed.

Sigaction Example

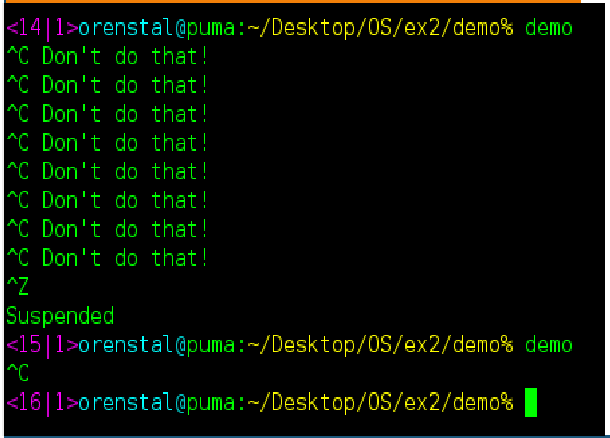
Return values check omitted due to space constraints

```
#include<stdio.h>
#include <unistd.h>
#include <signal.h>

void catch_int(int sig_num) {
  printf("Don't do that\n");
  fflush(stdout);
}

int main(int argc, char* argv[]) {
  // Install catch_int as the
  // signal handler for SIGINT.
  struct sigaction sa;
  sa.sa_handler = &catch_int;
  sigaction(SIGINT, &sa, NULL);

  for ( ;; )
     //wait until receives a signal
     pause();
}
```



```
<14|1>orenstal@puma:~/Desktop/OS/ex2/demo% demo
^C Don't do that!
^C Don't do that!
^C Don't do that!
^C Don't do that!
^C Don't do that!
^C Don't do that!
^C Don't do that!
^C Don't do that!
^Z
Suspended
<15|1>orenstal@puma:~/Desktop/OS/ex2/demo% demo
^C
<16|1>orenstal@puma:~/Desktop/OS/ex2/demo%
```

Pre-defined Signal Handlers

- There are two pre-defined signal handler functions that we can use instead of writing our own:
  - SIG_IGN: Causes the process to ignore the specified signal.
  - SIG_DFL: Causes the system to set the default signal handler for the given signal.

Intermediate Summary

- Each signal may have a signal action, which is a action than will be taken when the process receives that signal. An action includes a signal handler.
- If a signal is sent to the process, the next time the process runs, the operating system causes the process to run the signal handler, no matter what it was doing before.
- When that signal handler function returns, the process continues execution from wherever it happened to be before the signal was received.

Masking Signals - Motivation

- Assume that a process performs a cleanup
  - deleting old data, etc.
- If during the cleanup the program exits abruptly, some old files will remain
  - Data will be inconsistent/corrupted
- In order to avoid this situation, signals that can cause us to exit (such as SIGINT) should be blocked during cleanup
  - During the cleanup only! Masking/blocking is intended for specific parts of the code

Masking Signals - Avoiding Signal Races

- Because signals are handled asynchronously, race conditions can occur:
  - A signal may be received and handled in the middle of an operation that should not be interrupted
  - A second signal may occur before the current signal handler finished
    - The second signal may be of a different type or of the same type as the first one
- Therefore we need to block signals from being processed when they are harmful
  - The blocked signal will be processed after the block is removed
  - Some signals cannot be blocked

sigprocmask

Allows to specify a set of signals to block, and/or get the list of signals that were previously blocked

```
int sigprocmask(int how, const sigset_t *set,
                         sigset_t *oldset)
```

1. `int how`:
   - Add (`SIG_BLOCK`)
   - Delete (`SIG_UNBLOCK`)
   - Set (`SIG_SETMASK`).
2. `const sigset_t *set`:
   - The set of signals
3. `sigset_t *oldset`:
   - If not NULL, the previous mask will be returned

```
sigset_t set;

sigemptyset(&set);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGTERM);
sigprocmask(SIG_SETMASK, &set, NULL);
//blocked signals: SIGINT and SIGTERM

sigemptyset(&set);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGALRM);
sigprocmask(SIG_BLOCK, &set, NULL);
//blocked signals: SIGINT, SIGTERM, SIGALRM

sigemptyset(&set);
sigaddset(&set, SIGTERM);
sigaddset(&set, SIGUSR1);
sigprocmask(SIG_UNBLOCK, &set, NULL);
//blocked signals: SIGINT and SIGALRM
```

Handling Signals

So far we saw two system calls:

- sigaction
  - Specifies the action to take when a signal is received.
  - Has a lot of features, we focus on the sa_handler.
  - Can block signals while the handler is running.
- sigprocmask
  - Defines which signals to block

Reading online, you might see signal as an alternative to sigaction. **Don’t use it**.

Signals: Summary

- Signals are notifications sent to a process
- The OS causes the process to handle a signal immediately the next time it runs
- There are default signal handlers for processes
- These handlers can be changed using `sigaction`
- To avoid race conditions, one usually needs to block signals some of the time using `sigprocmask` and/or `sigaction`

Threads

KERNEL AND USER LEVEL THREADS

The many CPUs illusion

- Given a machine with one CPU, how can we efficiently execute number of tasks?
    - Each task lives in it's own world.
    - Virtualizing the CPU.
    - Multitasking systems (Time sharing).

What is a Process?

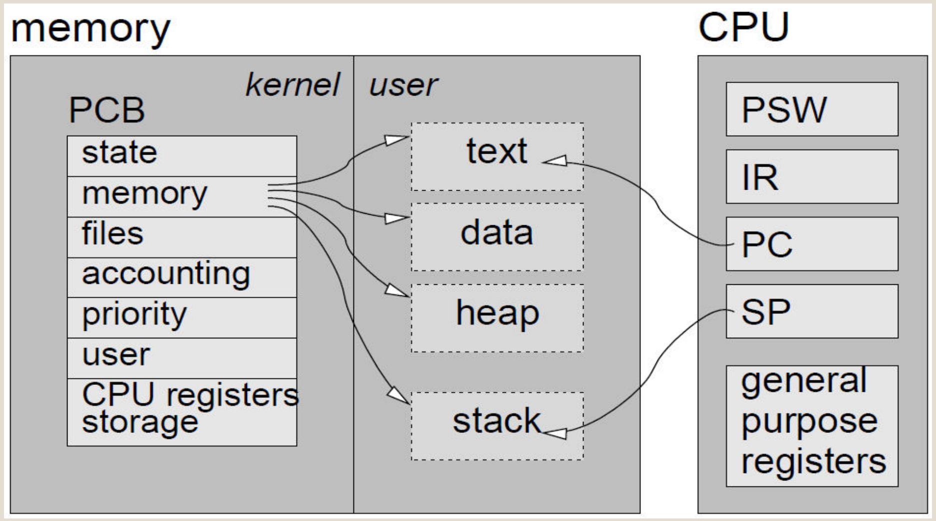
- Definition: an instance of an application execution.

- What defines a process?
  - registers (PC, SP etc.)
  - memory (data, heap, stack and text)
  - environment (files etc.)

- But how does the OS know all this?

Process Control Block (PCB)

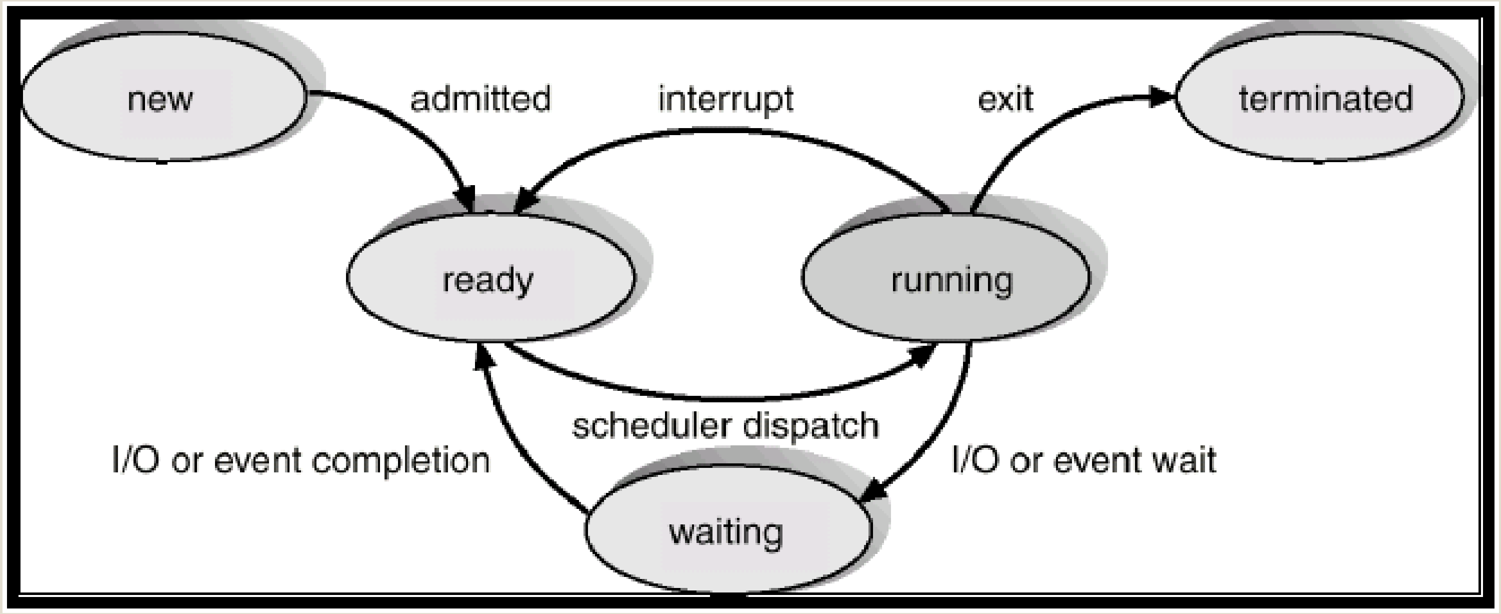
- What is saved in the PCB?
  - Process data:
    - registers (PC, SP etc.)
    - memory (data, heap, stack and text)
    - environment (files etc.)



  - OS data:
    - priority-relevant data (time, priority, resources etc.)
    - the user - access rights
    - **State**

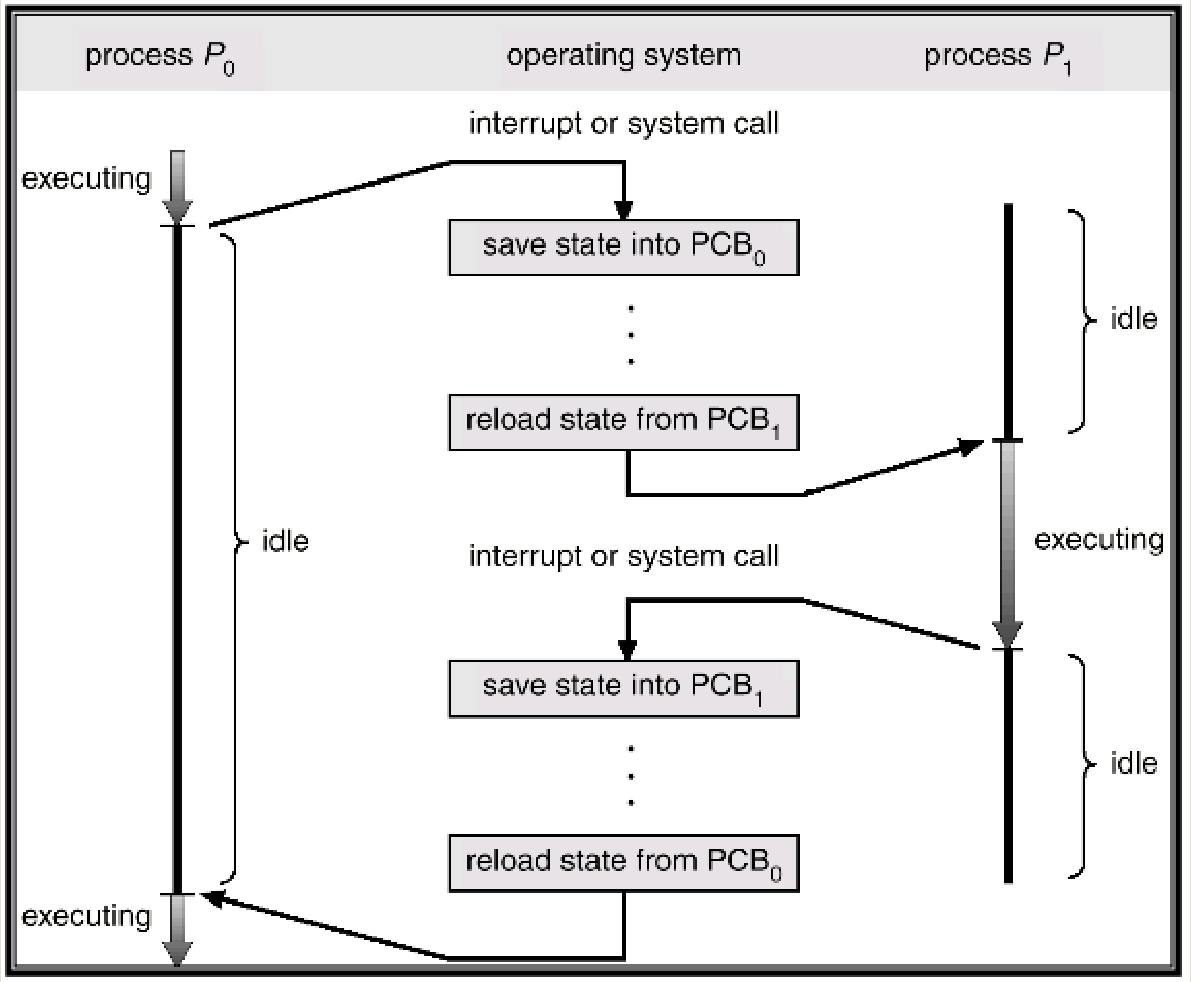
States

- A process's life cycle:

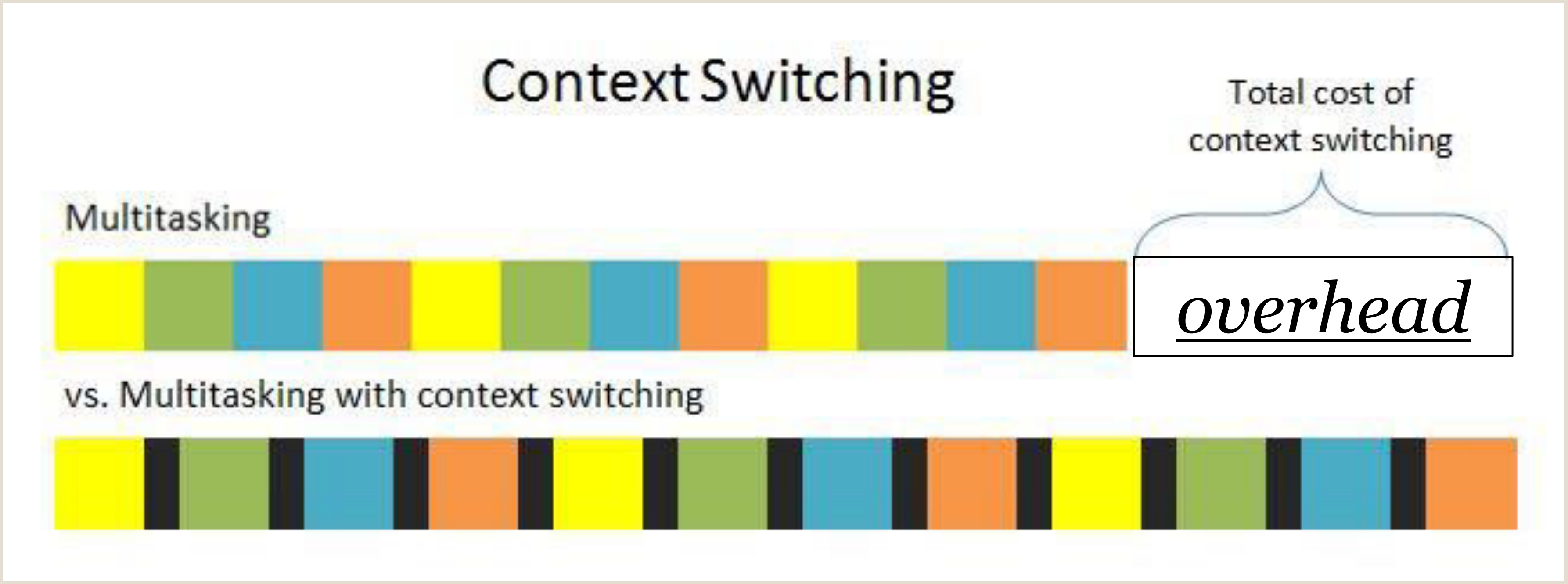


- Scheduler? stay tuned!

Context Switch



Context Switch


Context Switching
Total cost of context switching
Multitasking
vs. Multitasking with context switching

__overhead__

What is a Thread?

- A thread lives within a process
- A process can have several threads
- A thread possesses an independent flow of control, and can be scheduled to run separately from other threads, because it maintains its own:
  - stack
  - registers (CPU state)
- The other resources of the process are shared by all its threads:
  - code
  - memory
  - open files
  - and more…

Thread Implementations

- User level threads
  - Kernel unaware of threads
- Kernel level threads (lightweight processes)
  - Thread management done by the kernel

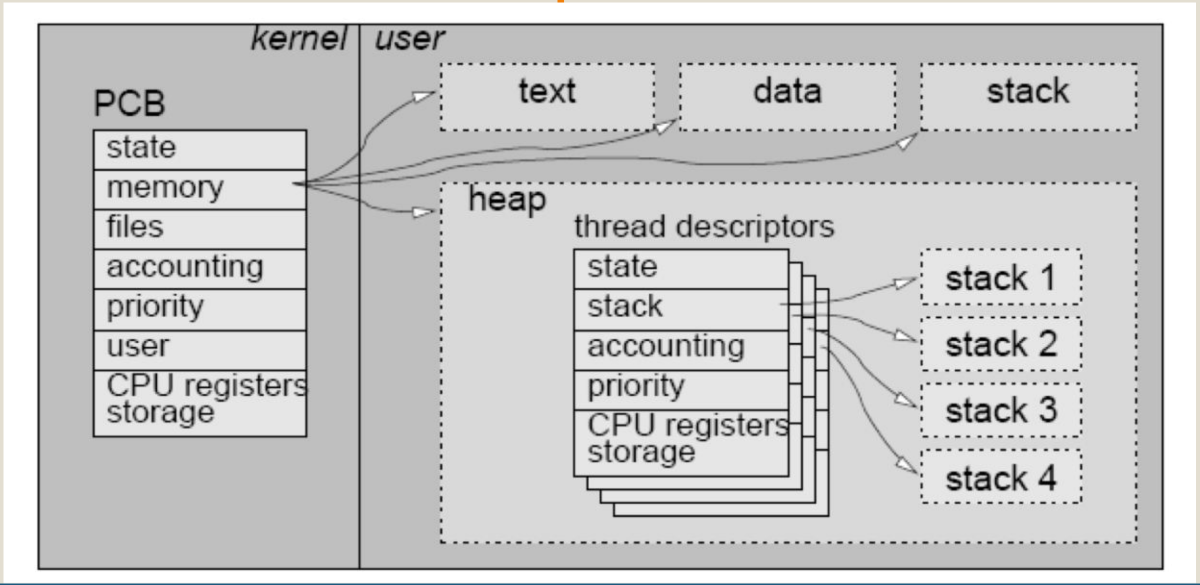
User Level Threads

User Level Threads

- Implemented as a thread library, which contains the code for thread creation, termination, scheduling and switching.
- Kernel sees one process and it is unaware of its thread activity.
- Switching from thread to thread is done in user space
  - So the penalty for a context switch is lower than an operating system context switch
- Scheduling depends on implementation
- If one thread is blocked by the kernel (from read() for example), all the process is blocked.

Implementing a Thread Library

- Only one thread can modify a shared resource at a time (if implemented correctly), so some of the locks required for threads may not be needed
  - Still, be careful!
- Maintain a thread descriptor for each thread



Implementing a Thread Library

- Switch between threads:
  1. Stop running current thread
  2. Save current state of the thread
  3. Jump to another thread
     - continue from where it stopped before, by using its saved state
- This requires special functions: `sigsetjmp` and `siglongjmp`
  - `sigsetjmp` saves the current location, CPU state and signal mask
  - `siglongjmp` goes to the saved location, restoring the state and the signal mask

`sigsetjmp` – save a "bookmark"

`sigsetjmp(sigjmp_buf env, int savesigs)`

- Saves the stack context and CPU state in `env` for later use
- If `savesigs` is non-zero, saves the current signal mask as well
- We can later jump to this code location and state using `siglongjmp`
- Return value:
  - 0 if returning directly
  - A user-defined value if we have just arrived here using `siglongjmp`

What is Saved in env?

Saved	Not Saved
• Program counter (PC) – Location in the code • Stack pointer (SP) – Locations of local variables – Return address of called functions • Signal mask – if specified • Rest of environment (CPU state) – Calculations can continue from where they stopped	• Global variables • Variables allocated dynamically • Values of local variables • Any other global resources

`siglongjmp` – use a "bookmark"

`siglongjmp(sigjmp_buf env, int val)`

- Jumps to the code location and restore CPU state specified by `env`
- The jump will take us into the location in the code where the `sigsetjmp` has been called
- If the signal mask was saved in `sigsetjmp`, it will be restored as well
- The return value of `sigsetjmp` after arriving from `siglongjmp`, will be the user-defined `val`

The Switch

Thread 0:

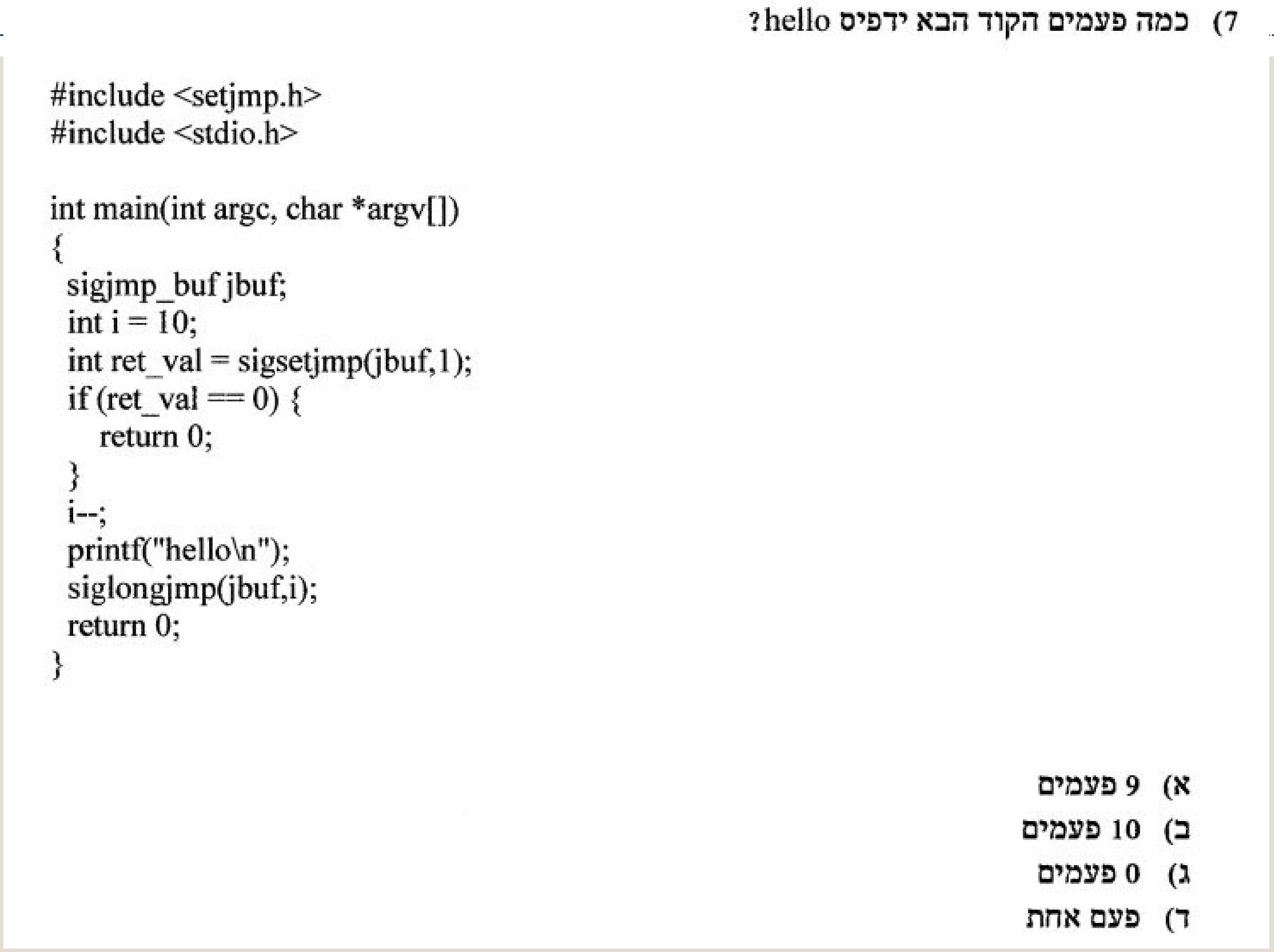
```
void switchThreads()
{
    static int curThread = 0;
    int ret_val =
     sigsetjmp(env[curThread],1);
    if (ret_val == 5) {
        return;
    }
    curThread =
      1 - curThread;
    siglongjmp(env[curThread],5);
}
```

Thread 1:

```
void switchThreads()
{
    static int curThread = 0;
    int ret_val =
     sigsetjmp(env[curThread],1);
    if (ret_val == 5) {
        return;
    }
    curThread =
      1 - curThread;
    siglongjmp(env[curThread],5);
}
```

- Full demo code is available as part of exercise 2.

Exam Question



7) כמה פעמים הקוד הבא ידפיס hello?

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

int main(int argc, char *argv[])
{
  sigjmp_buf jbuf;
  int i = 10;
  int ret_val = sigsetjmp(jbuf,1);
  if (ret_val == 0) {
      return 0;
  }
  i--;
  printf("hello\n");
  siglongjmp(jbuf,i);
  return 0;
}
```

א) 9 פעמים

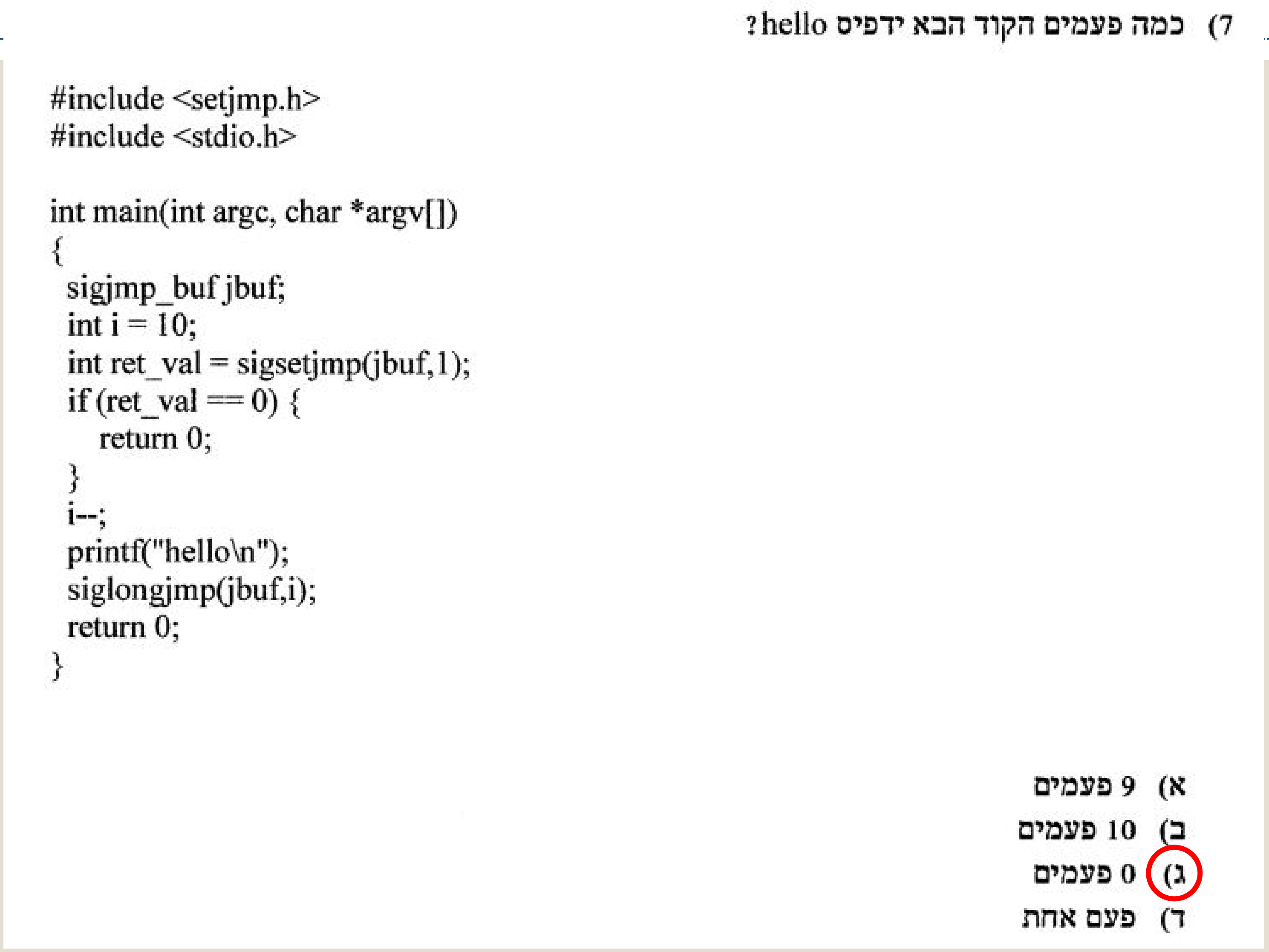
ב) 10 פעמים

ג) 0 פעמים

ד) פעם אחת

Taken from: http://www4.huji.ac.il/exams/67808_2012_2_2_1.pdf

Exam Question



7) כמה פעמים הקוד הבא ידפיס hello?

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

int main(int argc, char *argv[])
{
  sigjmp_buf jbuf;
  int i = 10;
  int ret_val = sigsetjmp(jbuf,1);
  if (ret_val == 0) {
      return 0;
  }
  i--;
  printf("hello\n");
  siglongjmp(jbuf,i);
  return 0;
}
```

א) 9 פעמים
ב) 10 פעמים
[✓ ג) 0 פעמים]
ד) פעם אחת

Taken from: http://www4.huji.ac.il/exams/67808_2012_2_2_1.pdf

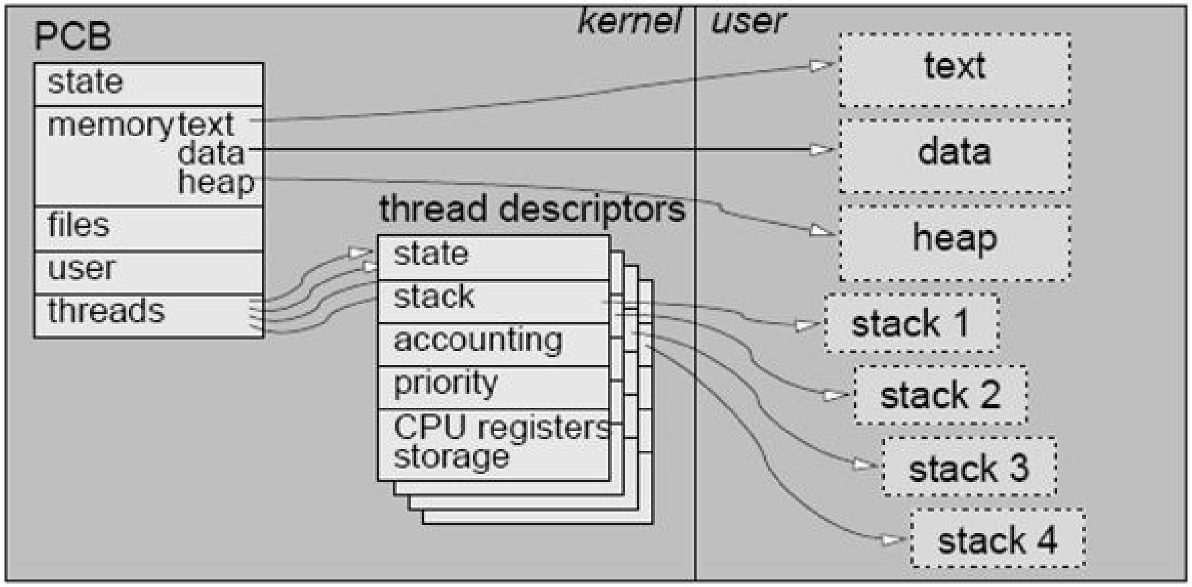
Kernel Level Threads

Kernel Level Threads

- The kernel has a threads table that keeps track of all the threads
- Scheduling is done according to the OS's scheduling algorithm
- If one thread is blocked, the rest can keep on running.
- Switching between kernel-level threads is more expensive than user-level threads
  - Involve more steps than just saving some registers.

Kernel level threads (lightweight processes)

- Thread management done by the kernel



Kernel Level Threads

Not to be confused with Kernel Threads

- Kernel threads are threads of the kernel running kernel code in kernel mode
- Often called kernel daemons
- They are used to perform some tasks of the kerel in parallel to other processes (and in parallel to the kernel itself)

User Level vs. Kernel Level Threads

Advantages

User Level Threads	Kernel Level Threads
• Switching between threads is cheaper • Often don’t need to worry about concurrent access to data structures • Scheduling can be application-specific	• Blocking is done on a thread level • Multiple threads can possibly be executed on different processors • Scheduler can make intelligent decisions amongst threads and processes

Disadvantages

User Level Threads	Kernel Level Threads
• Can't enjoy the benefits of a multi-core machine • One blocked thread blocks the entire process • Can suffer from poor OS scheduling	• Greater cost for switch between threads • Need to pay more attention to shared resources

Which is Better?

- When choosing the implementation, one must consider the specifications and needs of the application.
- For an application that switches between threads often, user level threads may be better.
- For an application that has many threads, or many that are I/O bound, kernel threads may be better.

Exam Question

- הבדל אחד בין תהליך לבין kernel level thread הוא

1. עם ריבוי תהליכים ניתן לנצל ריבוי מעבדים, אבל kernel level threads לא
2. עם kernel level thread מונעים את הבעיה של חסימה כאשר אחד מבצע פעולת I/O, אבל הבעיה קיימת כשמשתמשים בריבוי תהליכים
3. תהליכים זקוקים לתיווך של מערכת ההפעלה כדי לתקשר, ו-kernel level threads לא
4. ואילו תהליכים מריצים רק קוד של מערכת ההפעלה, Kernel threads יכולים להריץ קוד משתמש

Exam Question

- הבדל אחד בין תהליך לבין kernel level thread הוא

1. עם ריבוי תהליכים ניתן לנצל ריבוי מעבדים, אבל kernel level threads לא
2. [✓ עם kernel level thread מונעים את הבעיה של חסימה כאשר אחד מבצע פעולת I/O, אבל הבעיה קיימת כשמשתמשים בריבוי תהליכים]
3. תהליכים זקוקים לתיווך של מערכת ההפעלה כדי לתקשר, ו-kernel level threads לא
4. Kernel threads מריצים רק קוד של מערכת ההפעלה, ואילו תהליכים יכולים להריץ קוד משתמש

Ex 2

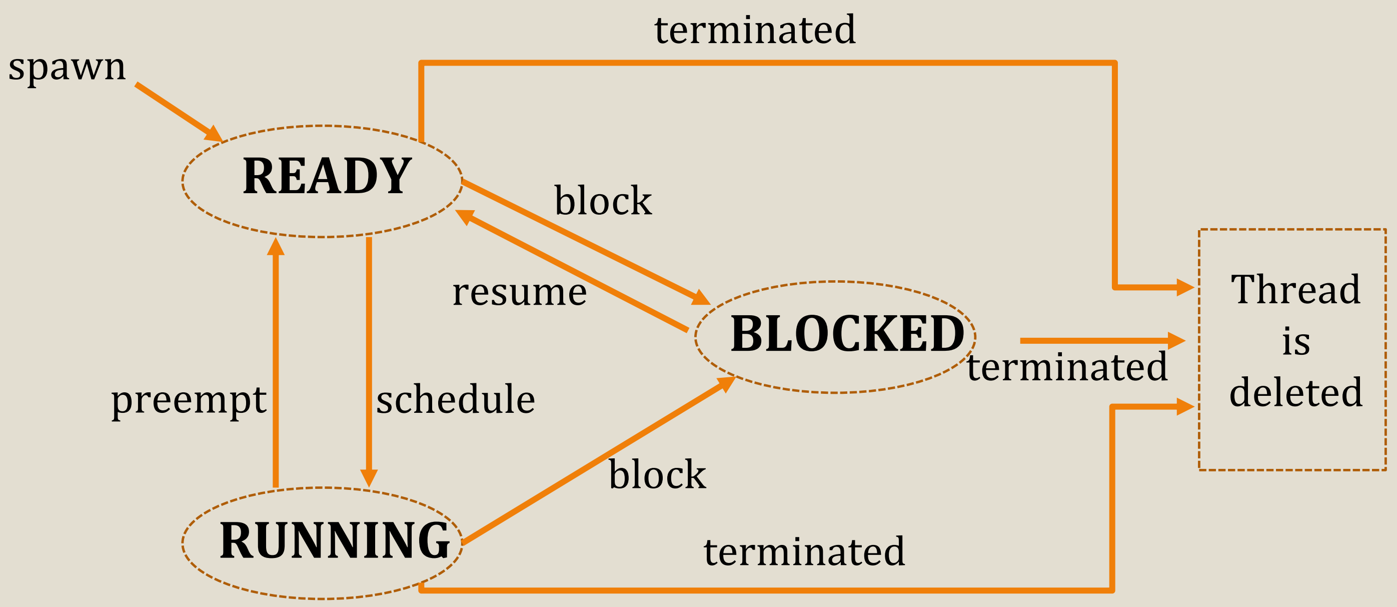
Implement a User-Threads Library

- The library should provide thread manipulation functions:
  - Init
  - Spawn
  - Terminate
  - Block
  - Resume
  - Sleep
- Library users can create their own threads and use the library functions to manipulate them.
- The library is in charge of thread management and scheduling.

Thread States

- **RUNNING** – The thread is the active thread.
- **READY** – The thread is allowed to run, but another thread is currently active.
- **BLOCKED** – The thread is not allowed to run.
  - Becomes READY if user calls “resume” or if synced.

Thread States Diagram



The Scheduler – Round Robin

- Each thread is spawned into the READY state.

- A list of READY threads is maintained:
    - Threads are added to this list once their state becomes READY.

- Newly added threads are scheduled to run only after all prior READY threads.

The Scheduler – Time Allocation and Preemption

- Every time a thread is transitioned to the RUNNING state it is allocated a predefined (virtual) time quantum to run.

- The RUNNING thread is preempted when either:
  - Its quantum has expired (turns to READY state).
  - It is blocked.
  - It terminates.

The Scheduler (Cont.)

- The scheduler decides which thread to run when:
  - The library is initialized (run the main thread).
  - The RUNNING thread is preempted/blocked.
- Take extra care to avoid signal races by blocking and unblocking signals where necessary.
- Use demo code for examples of:
  - Using interval timers and timer signals (SIGVTALRM).
  - Thread switching using sigsetjmp/siglongjmp.

This exercise is difficult,
so start early!

Good Luck!