MODULE 10 :
Processes, Threads and Scheduling

What is a Process and what is Process Creation?

- First, clarifications: In a lot of embedded systems, the term ‘process’ means the same as task/thread, and the terms are used interchangeably. We will use the term ‘user thread’ only to refer to a user entity that directly corresponds to a kernel schedulable entity and not to the much more rarely used user-level thread with a user space scheduler – an increasingly out of favour idea.

- In more general purpose systems (systems based on Unix), a process is a container of memory:
  - ✓ Process creation sets up a distinct logical-to-physical address set of mappings (fork-exec in Unix/Linux terms)
  - ✓ A main task/thread is created, a task control block(TCB) is allocated, a start function for the task is assigned, stack and heap markers are set up, and the thread is put on a ready queue, i.e. a pointer to the TCB is enqueued on a ready queue

How does a Thread Start Execution?

- Once the current instruction pointer or equivalent of the TCB of the task is set to point to the entry function (generally the main() function of a module), everything is in readiness for execution of application code

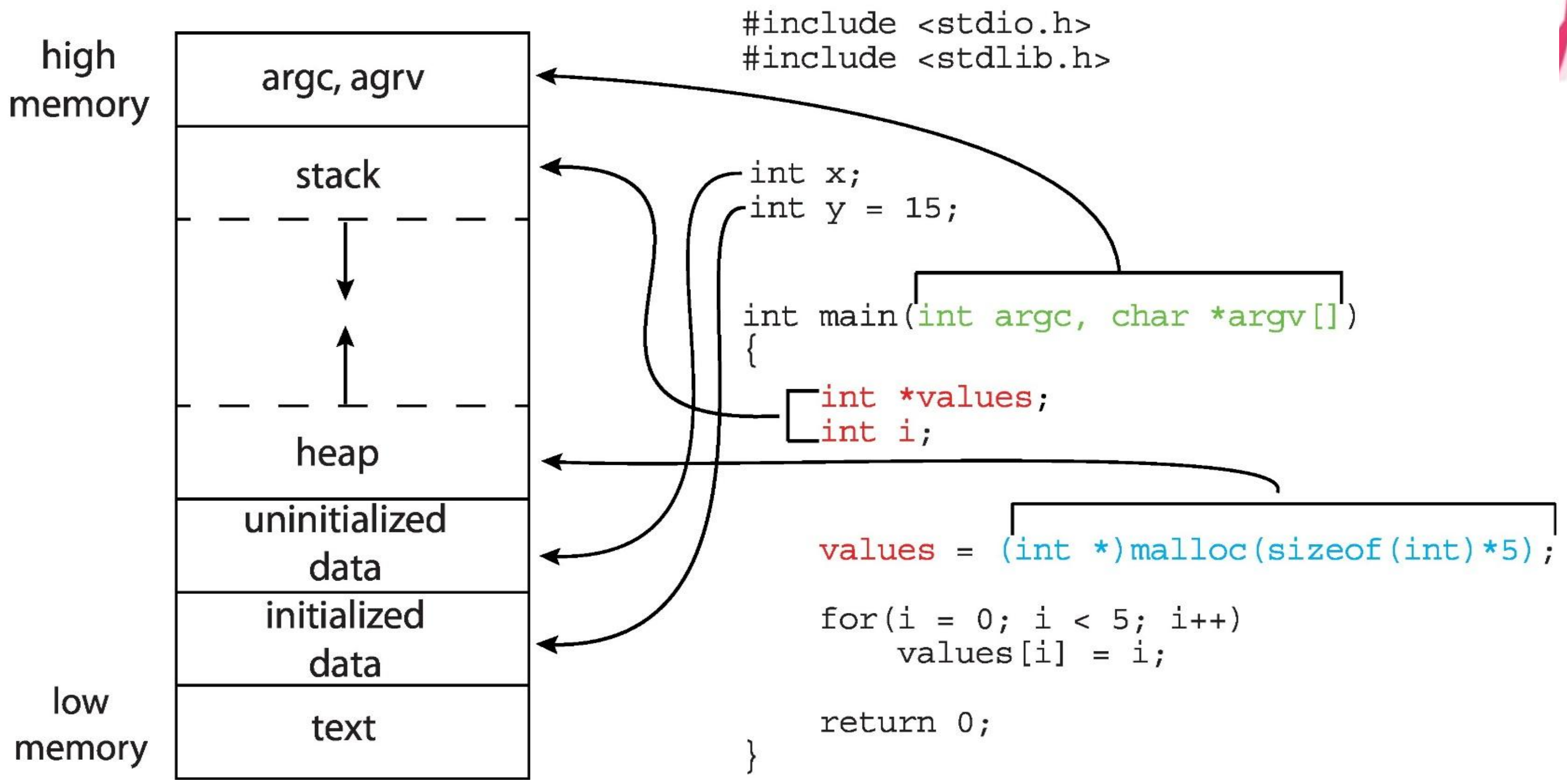
- When the scheduler (based on its algorithm) picks up/dequeues the pointer to the TCB from the ready queue, it also uses a subtle exception-level mechanism of the processor to set things rolling, i.e. make the program counter/instruction register of the processor to point to the entry function of the application on returning from exception – remember, the scheduler code always executes in kernel mode

How does a Thread Start Execution? (Contd.)

- We will look at different scheduler algorithms shortly, but first, let’s take a look at a neat little trick that the processor architecture supports to get the program counter to point to the start function of the application thread

- There are two processor registers that are used as program counter and processor status register backups during exception/interrupt processing. If these are set up as part of the scheduler thread selection function, the return from exception/interrupt instruction will push these backup register values atomically into the program counter and processor status registers.

Memory Layout of a Process – Sample Program



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

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

What is Process Context?

- There are several kernel data structures that manage the process address space for all threads that run from within the same process. We won’t look at this is in detail for now.

- Let’s look at a list of some items shared across all threads of a process (these make up the process context):
  - ✓ Text segment (instructions)
  - ✓ Data segment (static and global data)
  - ✓ BSS segment (uninitialized data)
  - ✓ Open file descriptors
  - ✓ Signals
  - ✓ Current working directory
  - ✓ User and group IDs

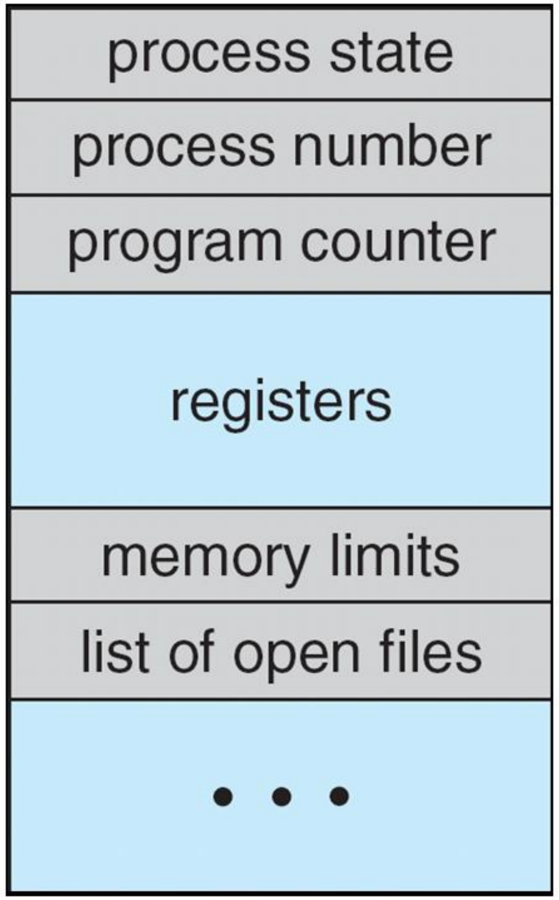
What data is private to a thread? (aka what is thread context?)

- **Threads do not share:**
  - ✓ Stack (local variables on the stack, return addresses, thread local storage)
  - ✓ Processor registers – these need to be saved and restored on thread context switch
  - ✓ Signal masks
  - ✓ Priority (and everything else that gets pushed into a TCB)

Process/Task Control Block - contents

Information associated with each process (also called task control block)

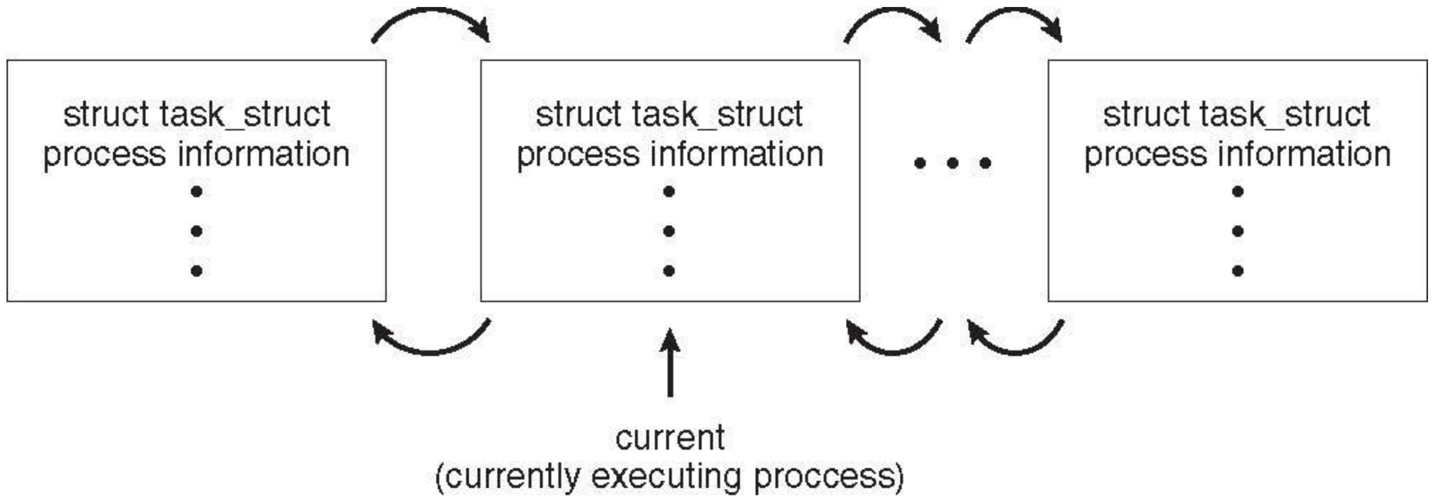
- Process state – Running, waiting, etc.
- Program counter – Location of instruction to execute next
- CPU registers – Contents of all general-purpose and some special-purpose registers
- CPU scheduling information- Priorities, scheduling queue pointers
- Memory-management information – Pointers to memory management data structures of memory allocated to the process
- Accounting information – Time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



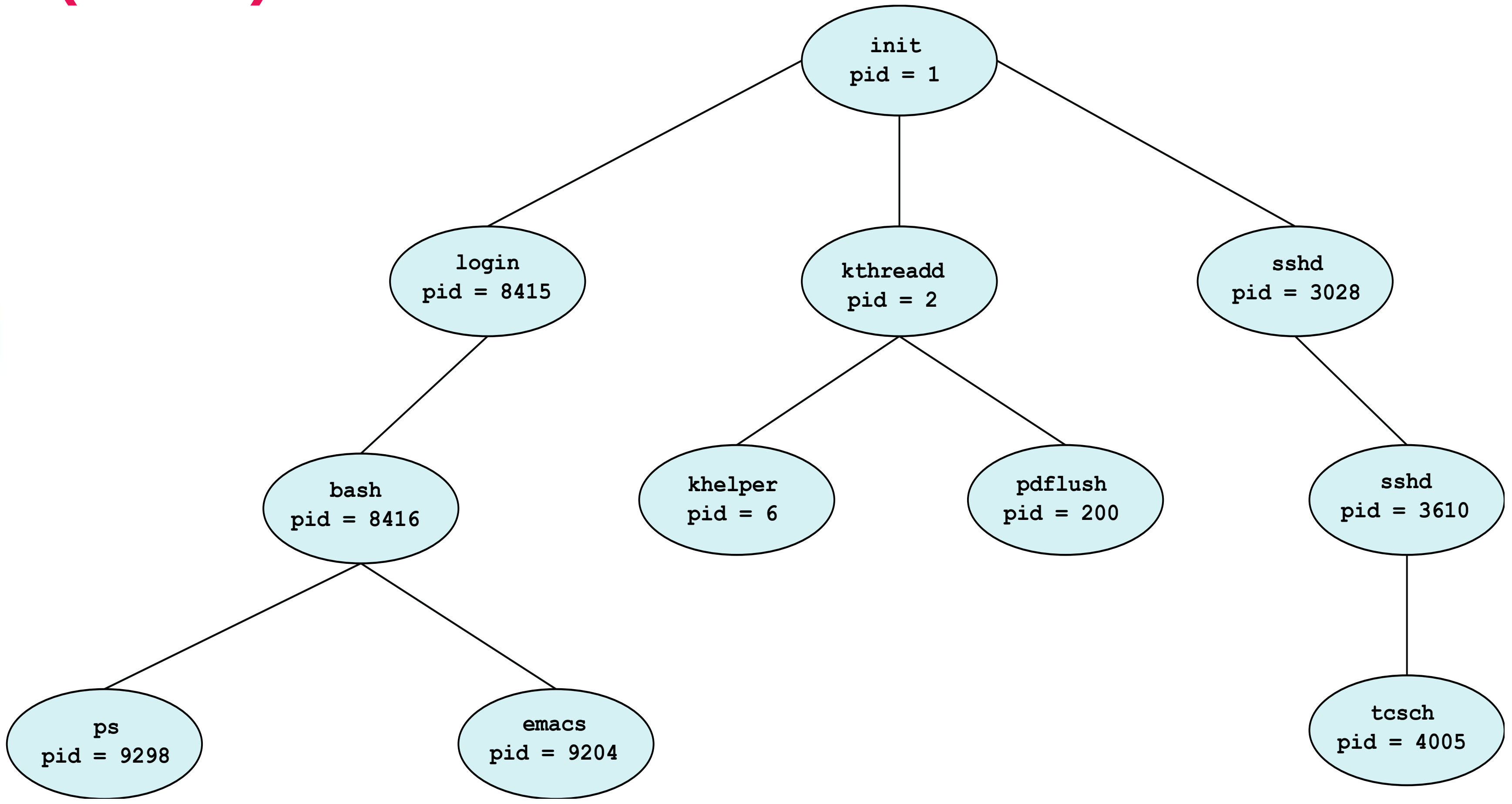
Process Representation (Linux)

Represented by the C structure task_struct

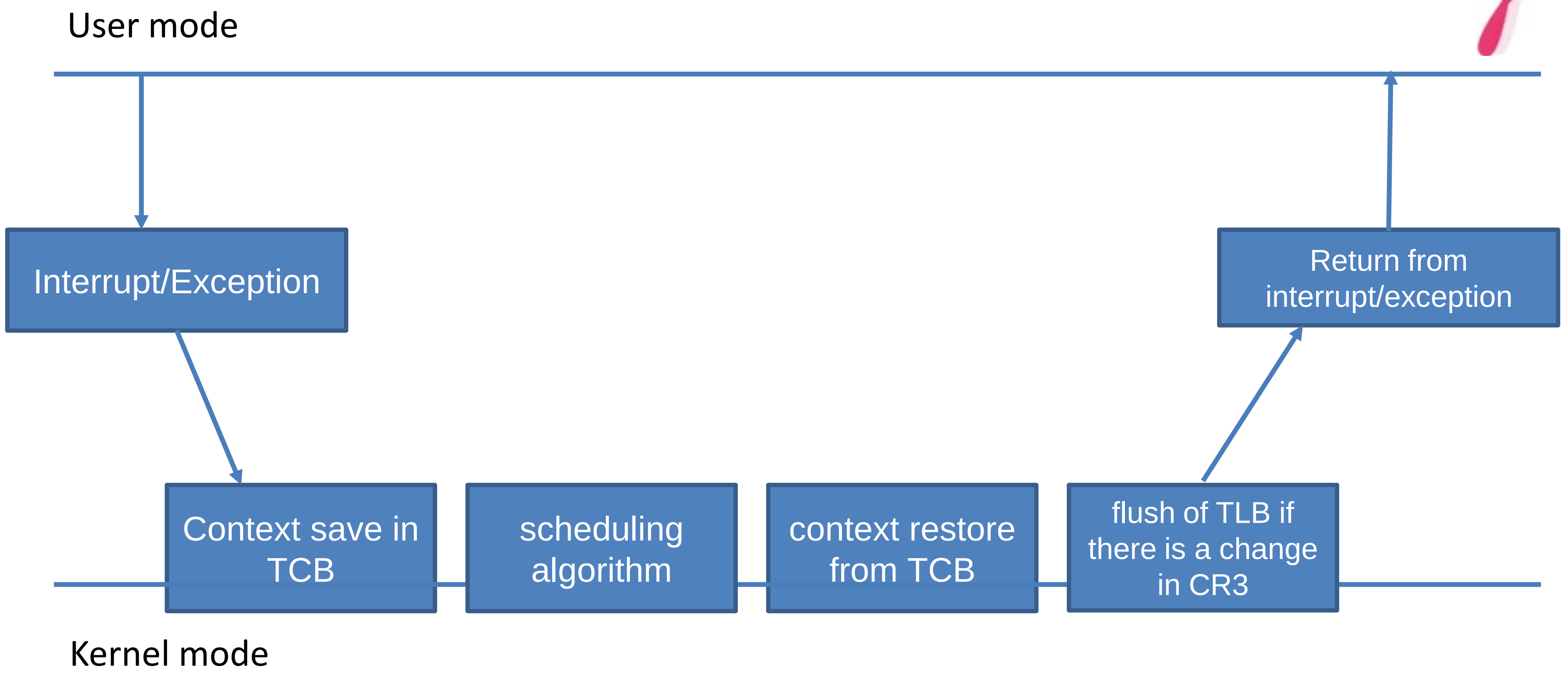
```
pid t_pid;                    /* process identifier */
long state;                   /* state of the process */
unsigned int time_slice       /* scheduling information */
struct task_struct *parent;   /* this process's parent */
struct list_head children;     /* this process's children */
struct files_struct *files;    /* list of open files */
struct mm_struct *mm;          /* address space of this
process */
```



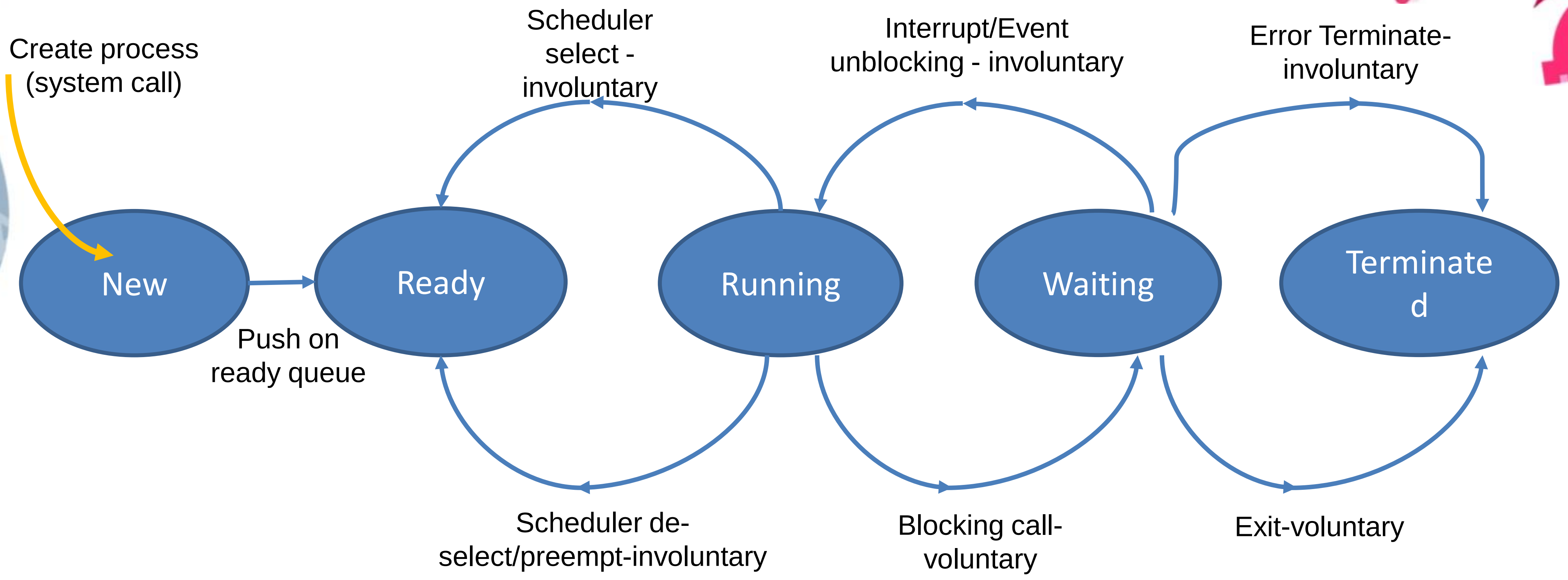
A Parent-Child Hierarchy of Processes (Linux)



What happens inside the Scheduler?

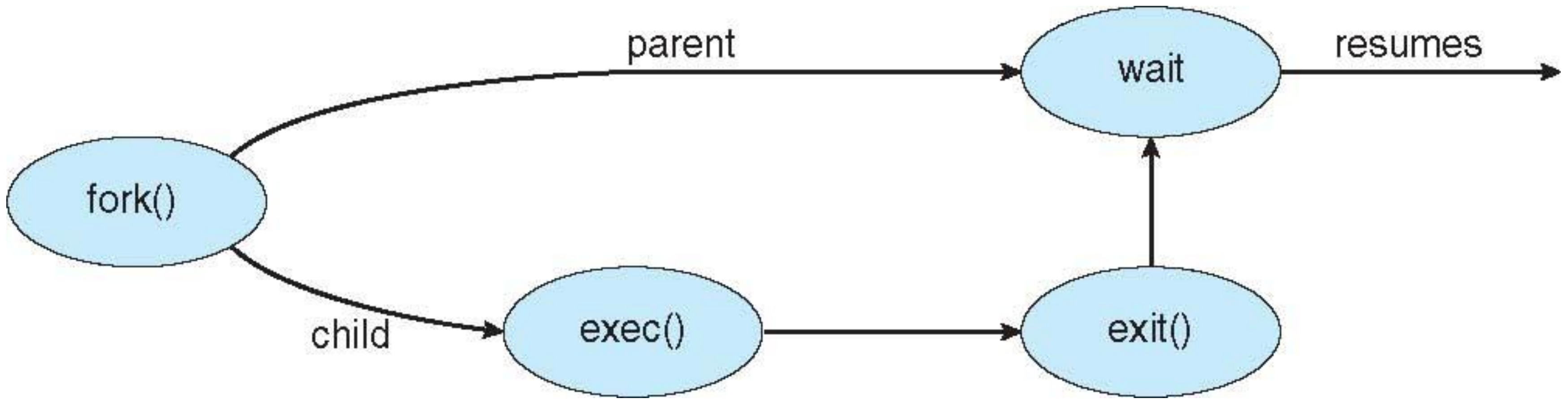


What are the various thread states and how do state transitions happen? – Operation on threads


Create process (system call)
Scheduler select - involuntary
Interrupt/Event unblocking - involuntary
Error Terminate- involuntary
New
Ready
Running
Waiting
Terminated
Push on ready queue
Scheduler de-select/preempt-involuntary
Blocking call- voluntary
Exit-voluntary

Operations on processes/threads (contd.)

- Process/Thread creation: We have dealt with aspects of process/thread (fork-exec) creation as a summary view (slides 6-8). Here is a state diagram that describes the execution sequence of a fork-exec set of routines:



Operations on processes/threads (contd.)

- Process/Thread creation: Let us look at fork-exec in more detail.
- Just to re-iterate: From a scheduler point of view, there is no distinction between a process and a thread entity. When a process is scheduled, it is really its main thread that is being scheduled.
- The difference between a process and a thread is when it comes to the memory and other memory resources - file, device and memory management structures are not shared across processes, while they are, across threads of the same process.

Operations on processes/threads (fork-exec)

- This distinction is particularly sharp when it comes to page tables.
- When a process does a fork, a copy of the page tables is done so that the same logical-to-physical mappings (as the parent process) hold for the newly forked process. In addition, the page table entries (PTEs) of the pages are marked copy-on-write.
- If the fork is followed by an exec of a new binary, then a different set of mappings are created for both code and data.
- If the fork is not followed by an exec, copies of pages are made when they are written to.

A sample fork-exec-wait sequence

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
   }

   return 0;
}
```

Expected output:
ls (dump of current directory files)
Child complete

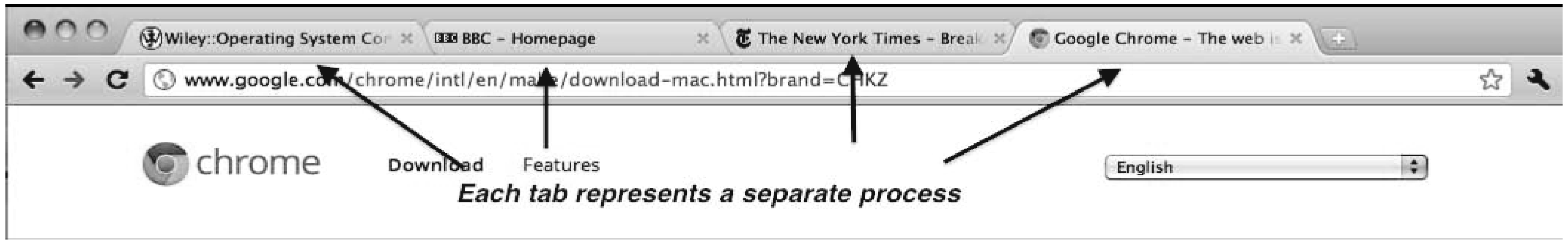
Multi-Process Architecture – Chrome Browser

Many old web browsers ran as a single process (some still do)

If one web site causes trouble, entire browser can hang or crash

The Google Chrome Browser is multi-process with 3 different processes:

- A Browser process that manages the user interface, disk and network I/O
- A Renderer process that renders web pages, deals with HTML, Javascript. A new renderer is created for each website opened
- Runs in sandbox restricting disk and network I/O, minimizing effect of security exploits
- A Plug-in process for each type of plug-in


Wiley::Operating System Cor
BBC – Homepage
The New York Times – Breal
Google Chrome – The web i
www.google.com/chrome/intl/en/make/download-mac.html?brand=CHKZ
chrome
Download
Features
English
Each tab represents a separate process

Thank You!