MODULE 13 :
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

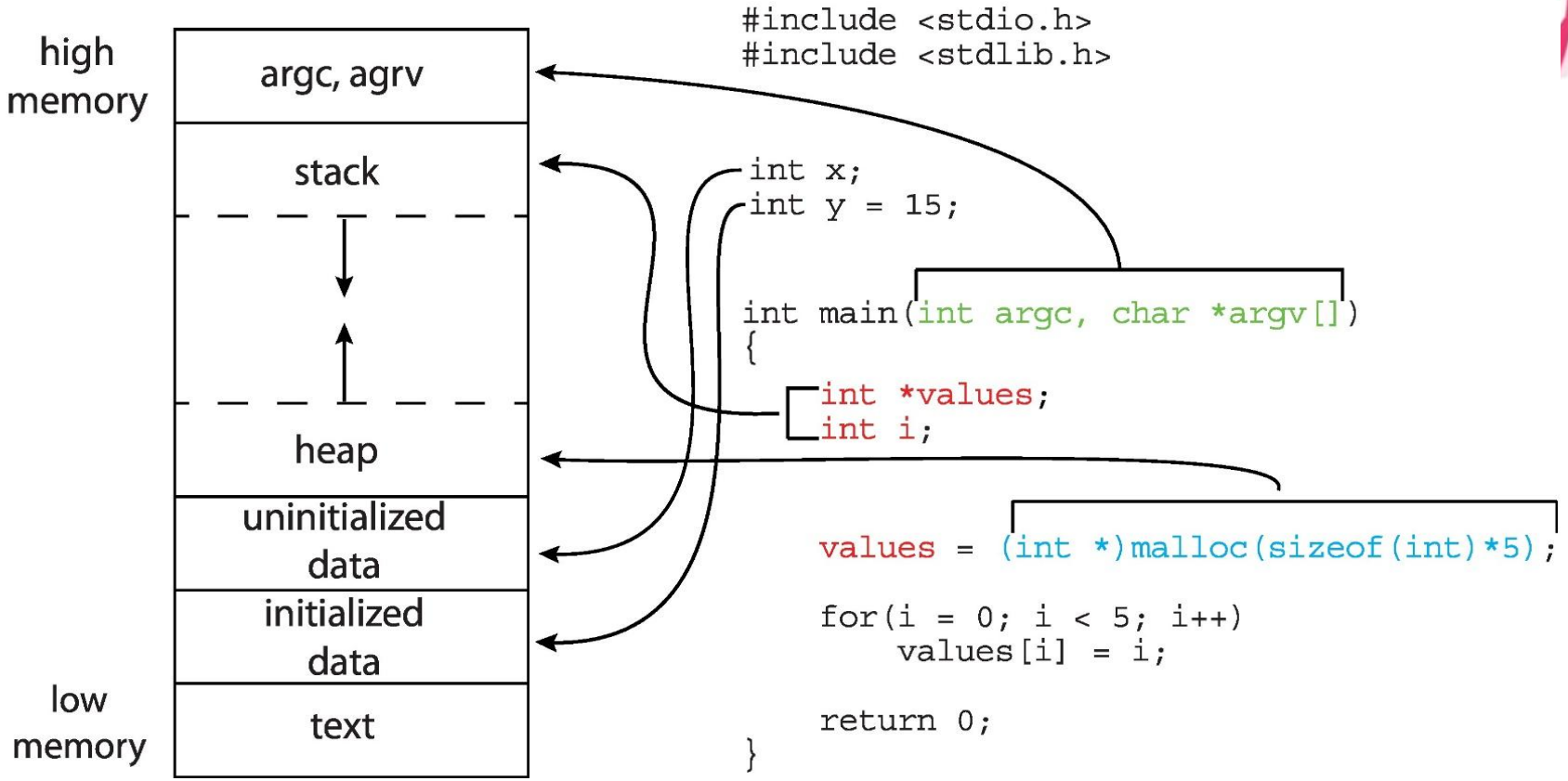
- When the scheduler (based on its algorithm) picks up/dequeues the pointer to the TCB from the ready queue, it also uses a subtle exception-level mechanism of the processor to set things rolling, i.e. make the program counter/instruction register of the processor to point to the entry function of the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



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

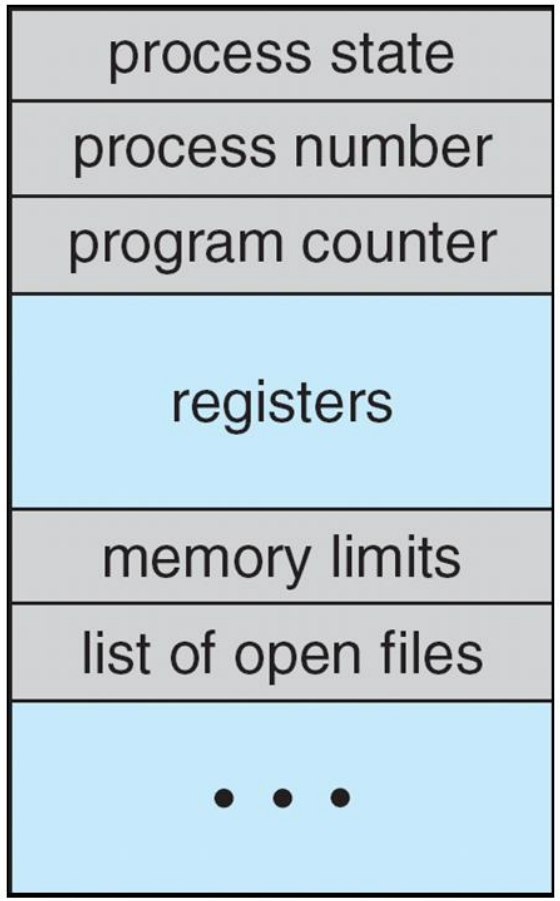
What data is private to a thread? (aka what is thread context?)

- **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)

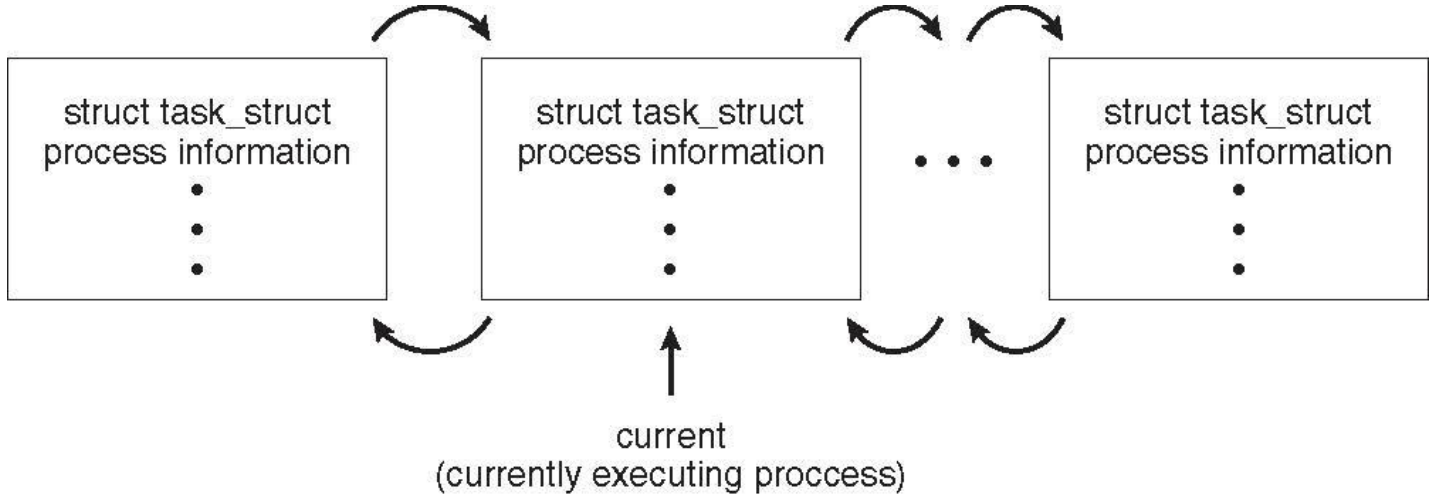
- Process state – Running, waiting, etc.
- Program counter – Location of instruction to execute next
- CPU registers – Contents of all general-purpose and some special-purpose registers
- CPU scheduling information- Priorities, scheduling queue pointers
- Memory-management information – Pointers to memory 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



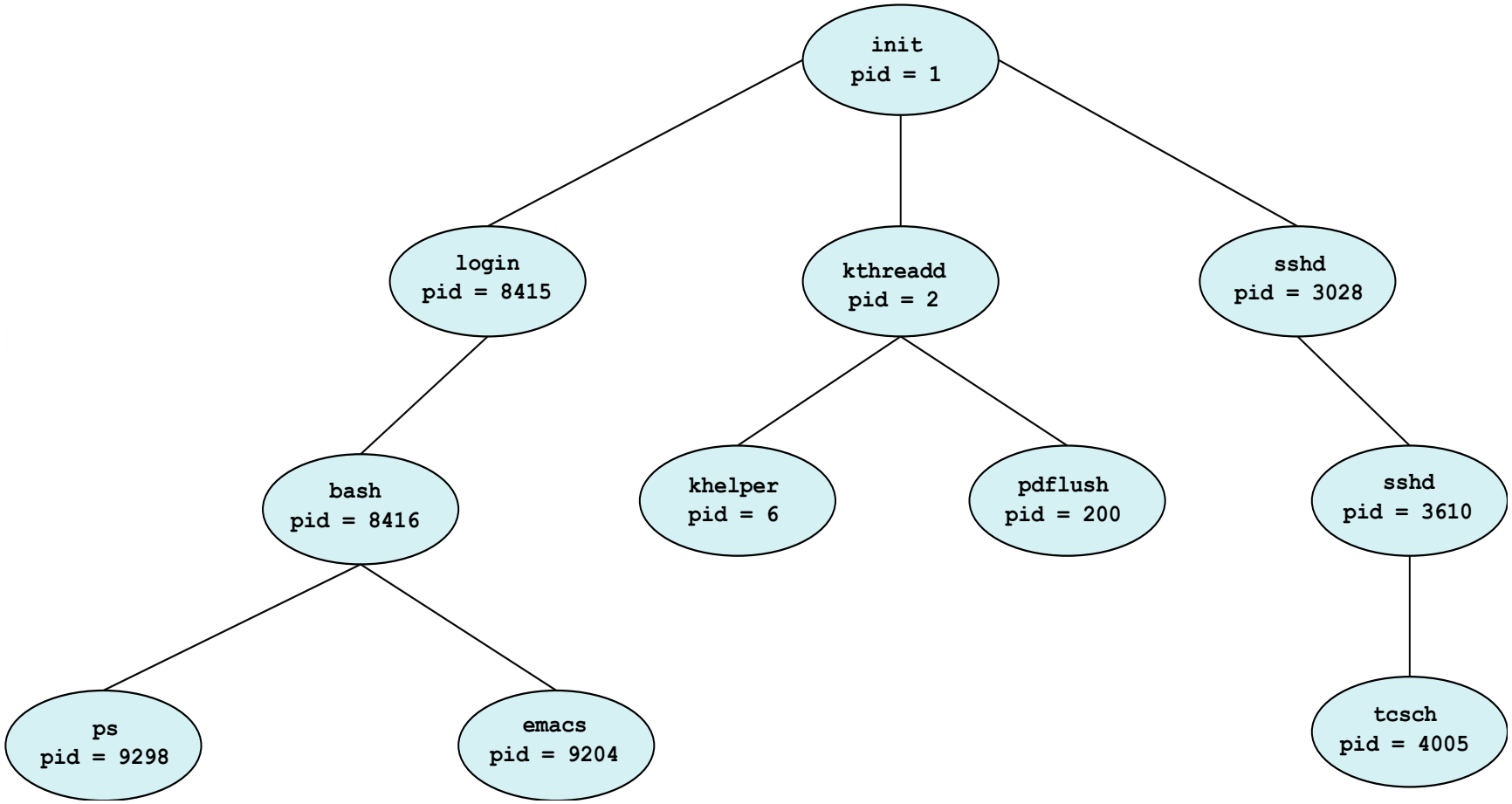
Process Representation (Linux)

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)

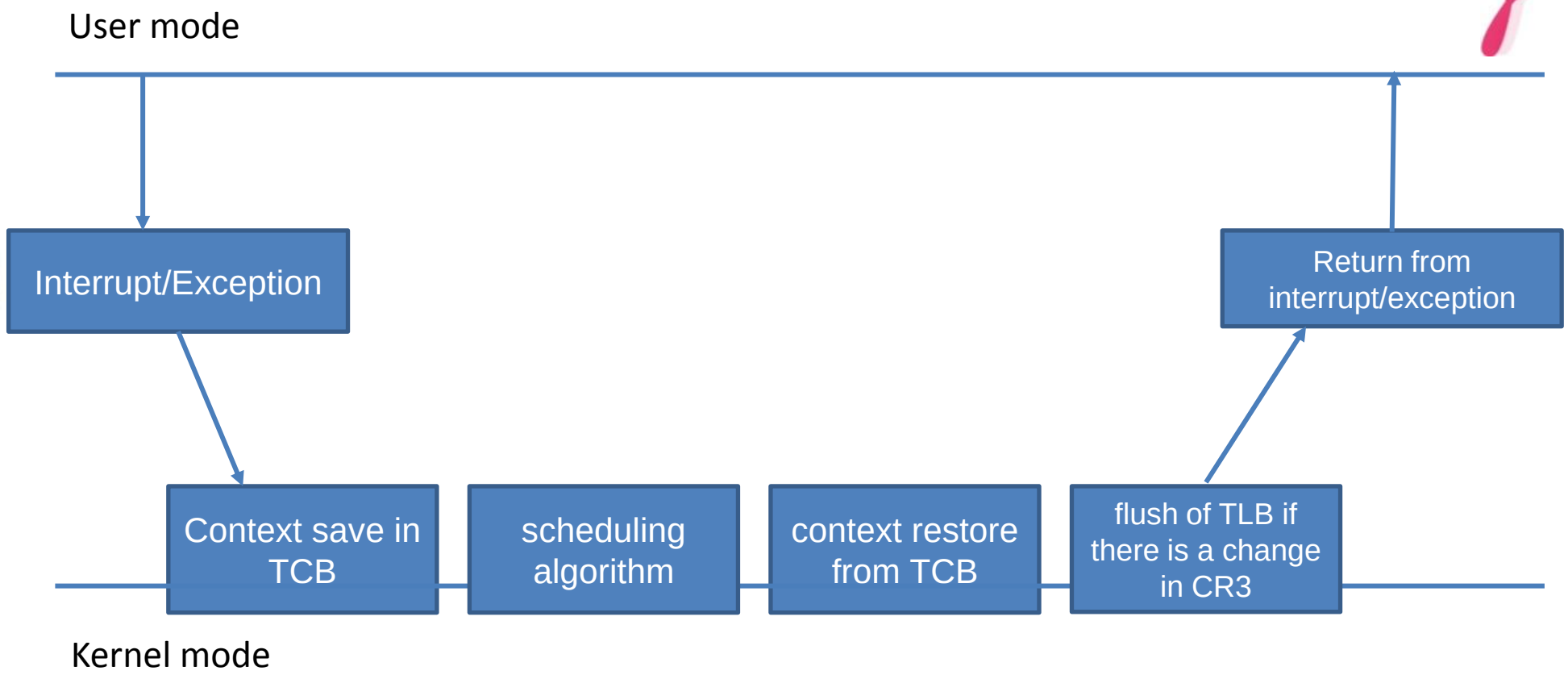


When/how does a thread get scheduled out?

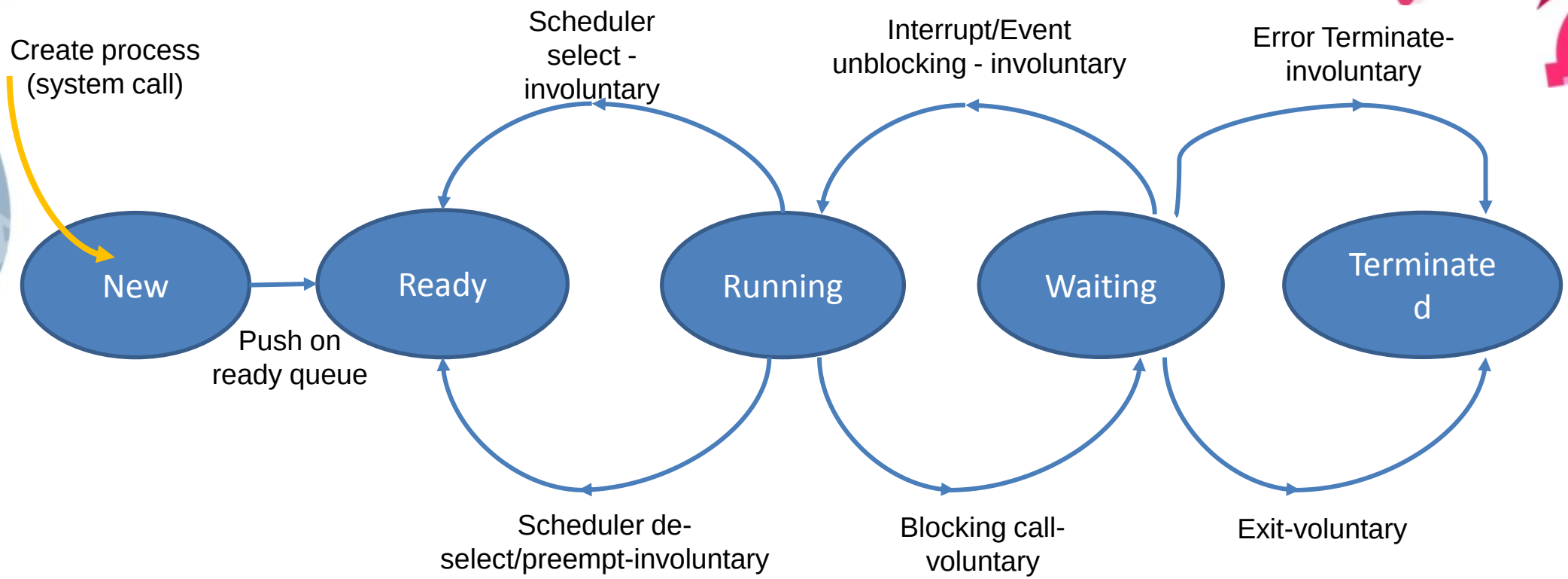
- Voluntarily: By virtue of a blocking system call

- Involuntarily:
  - ✓ An asynchronous interrupt waking up another thread that has execution precedence over the currently running thread according to the scheduling algorithm
  - ✓ A periodic timer interrupt causing the timer interrupt handler to run, that results in threads with more precedence over the currently running thread, being woken up
  - ✓ Or the timer interrupt resulting in the currently running thread to timeslice out on certain OSes

What happens inside the Scheduler?

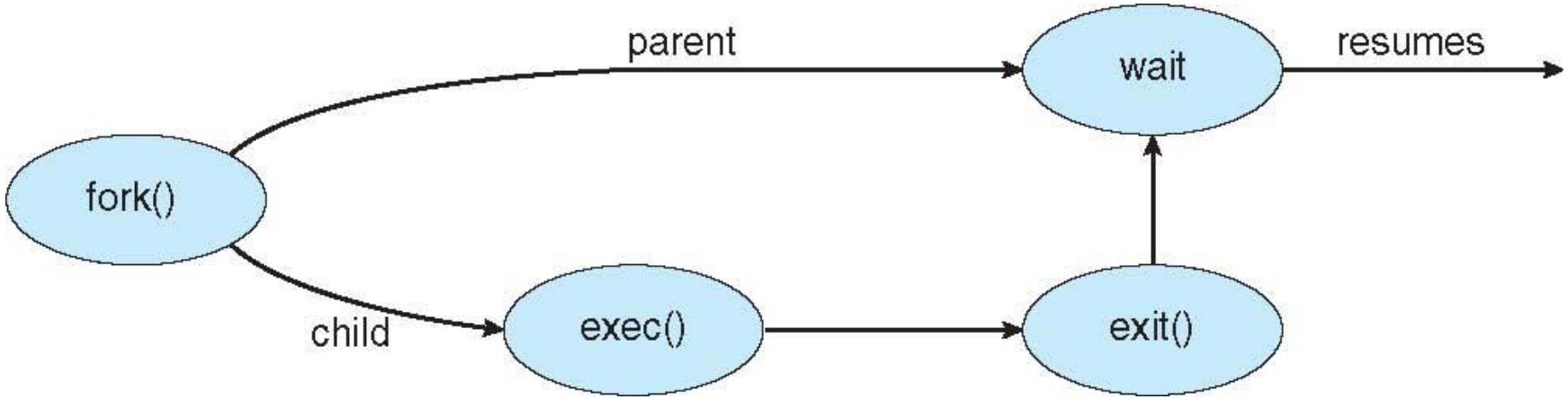


What are the various thread states and how do state transitions happen? – Operation on threads



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.

When do you Multithread?

Multithreading fetches you most benefits:

- When you need to do parallel independent computation
- When you need to do IO in parallel with computation
- When you need faster responses to several independent events. Then, you break down one big task into several independent units, so that these events don’t need to wait for each other for their processing
- When you have multiple cores

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

What are the different Scheduling Algorithms ?

- There are a whole host of scheduling algorithms in use as part of a variety of OSes. In interests of clarity and brevity, we will look at only three:

  - ✓ A fair-share scheduler (Fair-share, time-sharing)
  - ✓ A priority-based scheduler (Soft real time)
  - ✓ An earliest-deadline-first scheduler (Hard real time)

A Fair-Share Scheduler

Basic ideas:

- A fair-share scheduler gives a certain precedence to higher priority tasks, but doesn't let this totally override considerations of fairness
- Every task starts off with a static priority.
- A fair-share scheduler promotes "good" behaviour, i.e. if a task runs for a relatively short time, and schedules itself out, its "niceness" increases. If it hogs more of the CPU, then its "niceness" decreases.
- This niceness is linked to a dynamic priority, which falls within a short range (say, +/- 5) around the static priority

A Fair-Share Scheduler (contd.)

Basic ideas:

- The dynamic priority is the selection criterion that the scheduler uses in choosing the next task to run
- Timeslices are assigned to tasks based on their static priorities
- Timeslices place an upper bound on the continuous time that a task can run (have the CPU for itself)
- So, overall, there is a reasonable amount of fairness leading to lesser scenarios of lower priority tasks getting starved out
- Priorities get importance as well

The Linux O(1) Scheduler

The Linux O(1) scheduler is a good case study of a fair-share scheduler, it ran on several production systems for 5-10 years before being replaced by the current CFS (Completely Fair-Share) scheduler.

Broad scheduling principle:

- Idea: Multi-level (priority) feedback queues and the selection of the highest priority from the priority queue bitmap
- From a source code point of view:
- The schedule() function will swap out the current running thread and swap in the newly chosen thread.

The Linux O(1) Scheduler

- This is done by a one-instruction find of the highest priority bit set in the scheduler bit map array. If time slices have expired meanwhile, the scheduler_tick will set the stage for the current thread to be swapped out. In any case, the selection of the next thread is done by looking up the bitmap in O(1) time.

- The broad objective of schedule() – there are corner cases which we will ignore for now – is to swap the current process out (either voluntarily or because scheduler_tick has explicitly asked for it to be swapped out of the run queue).

The Linux O(1) Scheduler (contd.)

- Static priorities (0-139) (0-99 RT)
- Nice values (dynamic priorities: -20 - +19) (Direct mapping from 0-139 to -20 - +19)
- Time-slices based on static priorities
- Dynamic priorities change -5 - + 5 as a result of “good” behaviour:
  dynamic priority = max (100, min(static priority – bonus +5, 139),
  bonus = 0 to 10 depending on average sleep time of process
- Note: Higher priority threads can become interactive faster (with lesser average sleep time than lower priority threads)
- Favour interactivity - interactivity is favoured over time within certain limits imposed by priority
- Selection is based on dynamic priority – queues are indexed by dynamic priority

The Linux O(1) Scheduler (contd.)

- Two arrays: Active, expired arrays
- Upon expiry of timeslice of individual task, switch it to the expired array, thus working toward lesser starvation of lower-priority tasks
- Favouring interactivity again – if interactive task, then push it back on active array.
- Unless: An older expired process has waited for a long time on the expired array OR If there are higher priority expired processes

The Linux O(1) Scheduler (contd.)

- Real-time threads are always favoured, and they don’t have timeslice limits imposed on them if they are SCHED_FIFO. SCHED_RR does RR among RT threads of same priority.

- Kernel threads are picked for scheduling just like other user threads based on priority – only difference is their mm field points to NULL.

A Priority-Based Scheduler

- A purely priority-based scheduler (VxWorks) is designed to schedule in tasks of higher priority whenever these are ready to run.
- When you have more than one task of the same priority ready to run, then a round-robin policy is used.
- Its design can leave the door open for starvation of lower priority tasks.
- It needs to implement priority-inversion-safe semaphores

Priority Inversion

- Priority inversion happens when you have 3 tasks T1, T2, T3 in a ready to run state, where the priority of T1 > priority of T2 > priority of T3.

- Suppose T3 is running and has taken a resource (a semaphore) that T1 requires and is blocked on, and suppose T2 has pre-empted T3 and doesn't block for a fair while, you have a scenario where a lower priority task T3 has effectively blocked out a higher priority task T1 from running.

- Operating systems that use a priority-based scheduler need to guard against this priority inversion, and boost the priority of lower priority tasks that hold a sema to the priority of the highest priority task that has blocked on the sema

An Earliest-Deadline-First Scheduler

- In this hard-real time scheduler (RTEMS), the highest priority is assigned to the task with the earliest deadline, always.

- This is most suited to systems with hard real-time requirements and where tasks are strictly periodic:
  - ✓ The task whose period is the shortest is assumed to have the earliest deadline

- For instance, if you have 3 tasks P1, P2 and P3 with the following periods and execution times, you will get the following execution pattern with an EDF scheduler

Thank You!